The VLDB Journal manuscript No.
(will be inserted by the editor)

Retrospective on Aurora

Hari Balakrishnan[3], Magdalena Balazinska[3], Don Carney[2], Uğur Çetintemel[2], Mitch Cherniack[1], Christian Convey[2], Eddie Galvez[1], Jon Salz[3], Michael Stonebraker[3], Nesime Tatbul[2], Richard Tibbetts[3], Stan Zdonik[2]

[1] Department of Computer Science, Brandeis University, Waltham, MA 02454, USA
email: {mfc, eddie}@cs.brandeis.edu
[2] Department of Computer Science, Brown University, Providence, RI 02912, USA
email: {dpc, ugur, cjc, tatbul, sbz}@cs.brown.edu
[3] Department of EECS and Laboratory of Computer Science, M.I.T., Cambridge, MA 02139, USA
email: {hari, mbalazin, jsalz, stonebraker, tibbetts}@lcs.mit.edu

Received: date / Revised version: date

Abstract This experience paper summarizes the key lessons we learned throughout the design and implementation of the Aurora stream processing engine. For the past two years, we have built five stream-based applications using Aurora. We first describe in detail these applications and their implementation in Aurora. We then reflect on the design of Aurora based on this experience. Finally, we discuss our initial ideas on a follow-on project, called Borealis, whose goal is to eliminate the limitations of Aurora, as well as to address new key challenges and applications in the stream processing domain.

Key words Data stream management – Stream processing engines – Monitoring applications – Distributed stream processing – Quality-of-Service

1 Introduction and History

Over the last several years, a great deal of progress has been made in the area of stream processing engines (SPEs) [7,9, 15]. Three basic tenets distinguish SPEs from current data processing engines. First, they must support primitives for streaming applications. Unlike OLTP, which processes messages in isolation, streaming applications entail time series operations on streams of messages. Although a time series "blade" was added to the Illustra Object-Relational DBMS, generally speaking, time series operations are not well supported by current DBMSs. Second, streaming applications entail a real-time component. If one is content to see an answer later, then one can store incoming messages in a data warehouse and run a historical query on the warehouse to find information of interest. This tactic does not work if the answer must be constructed in real time. Real time also dictates a fundamentally different storage architecture. DBMSs universally store and index data records before making them available for query activity. Such *outbound processing*, where data is stored before being processed cannot deliver real-time latency, as required from SPEs. To meet more stringent latency requirements, SPEs must adopt an alternate model, *inbound processing*, where query processing is performed directly on incoming messages before (or instead of) storing them. Lastly, an SPE must have capabilities to gracefully deal with spikes in message load. Fundamentally incoming traffic is bursty, and it is desirable to selectively degrade the performance of the applications running on an SPE.

The Aurora stream processing engine, motivated by these three tenets, is currently operational. It consists of some 100K lines of C++ and Java and runs on both Unix- and Linux-based platforms. It was constructed with the cooperation of students and faculty at Brown, Brandeis, and M.I.T. The fundamental design of the engine has been well documented elsewhere: the architecture of the engine is described in [7], while the scheduling algorithms are presented in [8]. Load shedding algorithms are presented in [18], and our approach to high availability in a multi-site Aurora installation is covered in [10,13]. Lastly, we have been involved in a collective effort to define a benchmark that described the sort of monitoring applications that we have in mind. The result of this effort is called Linear Road, and is described in [4].

Recently, we have used Aurora to build five different application systems. Throughout the process, we have learned a great deal about the key requirements of streaming applications. In this paper, we reflect on the design of Aurora based on this experience.

The first application is an Aurora implementation of Linear Road, mentioned above. In addition to Linear Road, we

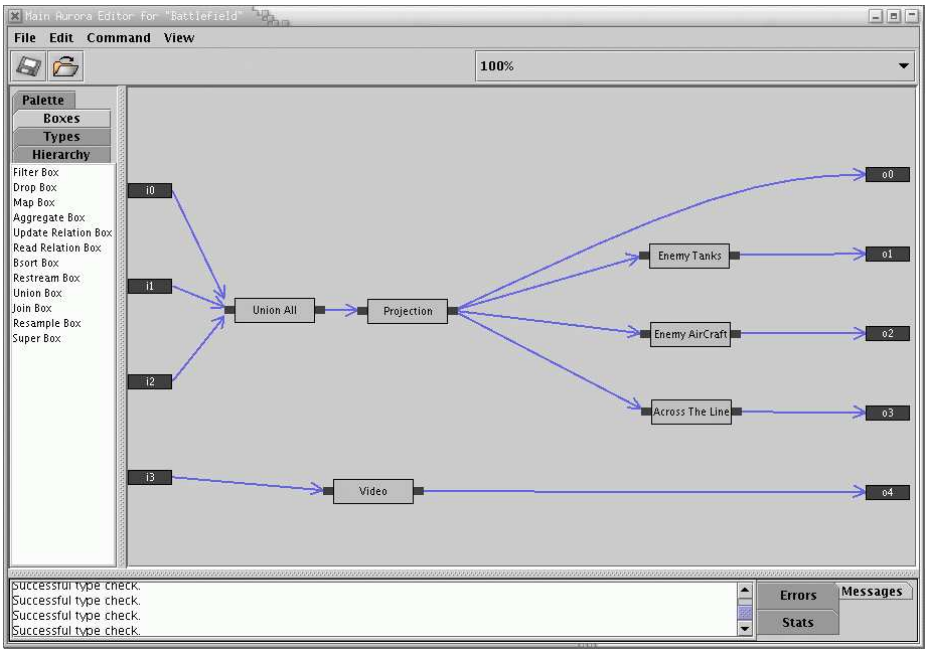


Fig. 1 Aurora Graphical User Interface

have implemented a pilot application that detects late arrival of messages in a financial-services feed-processing environment. Furthermore, one of our collaborators, a military medical research laboratory [20], asked us to build a system to monitor the levels of hazardous materials in fish. We have also worked with a major defense contractor on a pilot application that deals with battlefield monitoring in a hostile environment. Lastly, we have used Aurora to build Medusa, a distributed version of Aurora that is intended to be used by multiple enterprises that operate in different administrate domains. Medusa uses an innovative agoric model to deal with cross system resource allocation, and is described in more detail in [5].

We start with a short review of the Aurora design in Section 2. Following this, we discuss the five case studies mentioned above in detail in Section 3, so the reader can understand the context for the retrospection that follows. In Section 4, we present the lessons we have learned on the design of SPEs. These include the necessity of supporting stored tables, the requirement of synchronization primitives to main consistency of stored data in a streaming environment, the need for supporting primitives for late arriving or missing messages, the requirement for a myriad of adaptors for other feed formats, and the need for globally-accessible catalogs and a programming notation to specify Aurora networks (in addition to the "boxes and arrows" GUI). Since stream processing applications are usually time critical, we also discuss the importance of light-weight scheduling and quantify the performance of the current Aurora prototype using a micro-benchmark on basic stream operators. Aurora performance on the Linear Road benchmark is documented elsewhere [4].

The current Aurora prototype is being transferred to the commercial domain, with venture capital backing. As such, the academic project is hard at work on a complete redesign of Aurora, which we call Borealis. The intent of Borealis is to overcome some of the shortcomings of Aurora, as well as make a major leap forward in several areas. Hence, in Section 5, we discuss the ideas we have for Borealis in several new areas including mechanisms for dynamic modification of query specification and query results, and a distributed optimization framework that operates across server and sensor networks.

2 Aurora Architecture

Aurora is based on a dataflow-style "boxes and arrows" paradigm. Unlike other stream processing systems that use SQL-style declarative query interfaces (e.g., STREAM [15]), this approach was chosen because it allows query activity to be interspersed with message processing (e.g., cleaning, correlation, etc.). Systems that only perform the query piece must ping-pong back and forth to an application for the rest of the work, thereby adding to system overhead and latency. An Aurora network can be spread across any number of machines to achieve high scalability and availability characteristics.

In Aurora, a developer uses the GUI to wire together a network of boxes and arcs that will process streams in a manner that produces the outputs necessary to his or her application. A screen shot of the GUI used to create Aurora networks is shown in Figure 1: the black boxes indicate input and

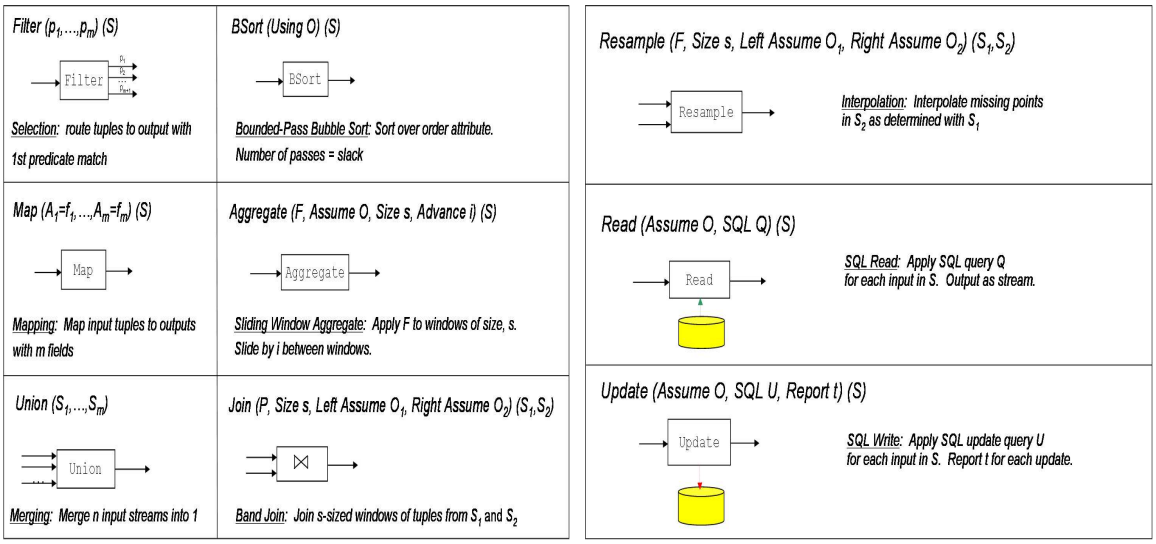


Fig. 2 Aurora Operators

output streams that connect Aurora with the stream sources and applications, respectively. The other boxes are Aurora operators and the arcs represent data flow among the operators. Users can drag-and-drop operators from the palette on the left and connect them by simply drawing arrows between them. It should be noted that a developer can name a collection of boxes and replace it with a "superbox". This "macro-definition" mechanism drastically eases the development of big networks.

The Aurora operators are presented in detail in [3] and are summarized in Figure 2. Aurora's operator choices were influenced by numerous systems. The basic operators Filter, Map and Union are modeled after the Select, Project and Union operations of the relational algebra. Join's use of a distance metric to relate joinable elements on opposing streams is reminiscent of the relational band join [12]. Aggregate's sliding window semantics is a generalized version of the sliding window constructs of SEQ [17] and SQL-99 (with generalizations including allowance for disorder (SLACK), timeouts, value-based windows etc.). The ASSUME ORDER clause (used in Aggregate and Join), which defines a result in terms of an order which may or may not be manifested, is borrowed from AQuery [14].

Each input must obey a particular schema (a fixed number of fixed or variable length fields of the standard data types). Every output is similarly constrained. An Aurora network accepts inputs, performs message filtering, computation, aggregation, and correlation, and then delivers output messages to applications. Moreover, every output is optionally tagged with a Quality of Service (QoS) specification. This specification indicates how much latency the connected application can tolerate, as well as what to do if adequate responsiveness cannot be assured under overload situations. Note that the Aurora notion of QoS is different from the traditional QoS notion that typically implies hard performance guarantees, resource reservations and strict admission control.

On various arcs in an Aurora network, the developer can note that Aurora should remember historical messages. The amount of history to be kept by such "connection points" can be specified by a time range or a message count. The historical storage is achieved by extending the basic message-queue management mechanism. New boxes can be added to an Aurora network at connection points at any time. History is replayed through the added boxes, and then conventional Aurora processing continues. This processing continues until the extra boxes are deleted.

The Aurora optimizer can rearrange a network by performing box swapping when it thinks the result will be favorable. Such box swapping cannot occur across a connection point; hence connection points are arcs that restrict the behavior of the optimizer as well as remember history.

When a developer is satisfied with an Aurora network, he or she can compile it into an intermediate form, which is stored in an embedded database. At run time this data structure is read into virtual memory and drives a real-time scheduler. The scheduler makes decisions based on the form of the network, the QoS specifications present, and the length of the various queues. When queues overflow the buffer pool in virtual memory, they are spooled to the embedded database. More detailed information on these various topics can be obtained from the referenced papers [3,7,8,18].

3 Aurora Case Studies

In this section, we present five case studies of applications built using the Aurora engine and tools.

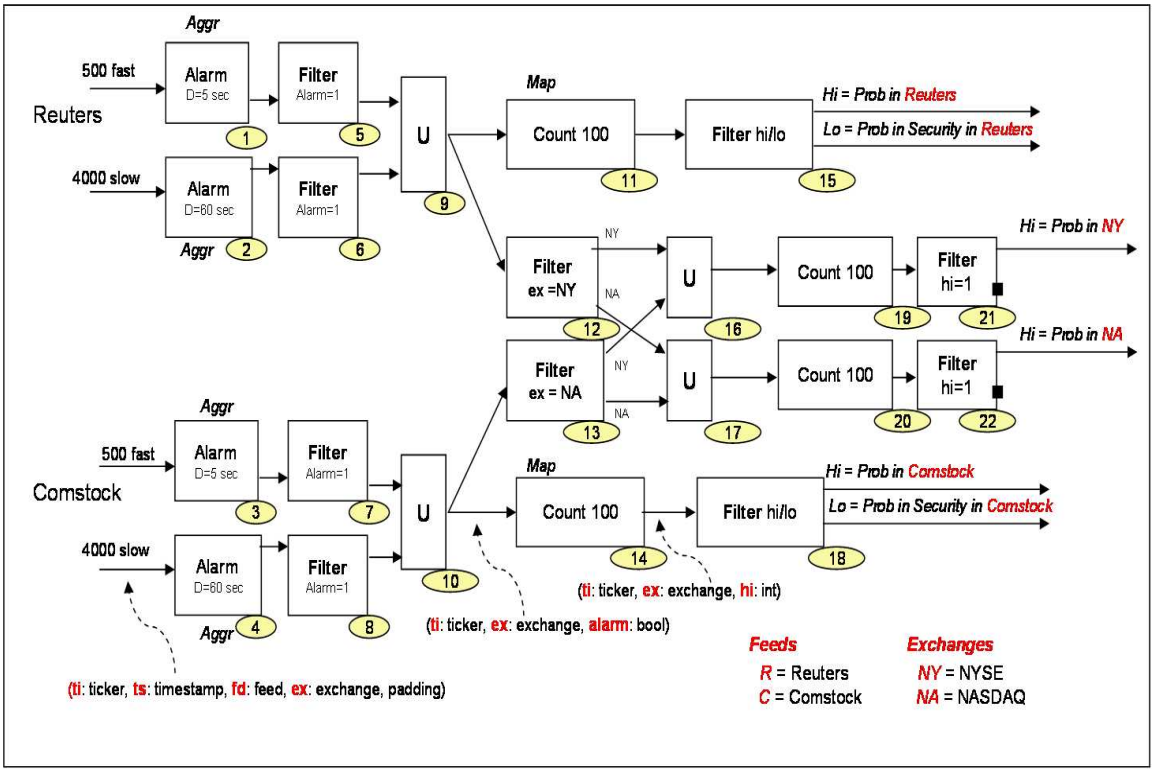


Fig. 3 Aurora Query Network for the Alarm Correlation Application

3.1 Financial Services Application

Financial service organizations purchase stock ticker feeds from multiple providers and need to switch in real time between these feeds if they experience too many problems. We worked with a major financial services company on developing an Aurora application that detects feed problems and triggers the switch in real time. In this section, we summarize the application (as specified by the financial services company) and its implementation in Aurora.

An unexpected delay in the reporting of new prices is an example of a feed problem. Each security has an expected reporting interval and the application needs to report an alarm if a reporting interval exceeds its expected value. Furthermore, if more than some number of alarms are recorded, a more serious alarm is raised that could indicate that it is time to switch feeds. The delay can be caused by the underlying exchange (e.g., NYSE, NASDAQ) or by the feed provider (e.g., Comstock, Reuters). If it is the former, switching to another provider will not help so the application must be able to rapidly distinguish between these two cases.

Ticker information is provided as a real-time data feed from one or more providers and a feed typically reports more than one exchange. As an example, let us assume that there are 500 securities within a feed that update at least once every 5 seconds and they are called “fast updates”. Let us also assume that there are 4000 securities that update at least once every 60 seconds and they are called “slow updates”.

If a ticker update is not seen within its update interval, the monitoring system should raise a *low alarm*. For example, if MSFT is expected to update within 5 seconds, and 5 seconds or more elapse since the last update, a low alarm is raised.

Since the source of the problem could be in the feed or the exchange, the monitoring application must count the number of low alarms that are found in each exchange and the number of low alarms that are found in each feed. If the number for each of these categories exceeds a threshold (100 in the following example), a *high alarm* is raised. The particular high alarm will indicate what action should be taken. When a high alarm is raised, the low alarm count is reset and the counting of low alarms begins again. In this way, the system produces a high alarm for every 100 low alarms of a particular type.

Furthermore, the posting of a high alarm is a serious condition and low alarms are suppressed when the threshold is reached to avoid distracting the operator with a large number of low alarms.

Figure 3 presents our solution realized with an Aurora query network. We assume for simplicity that the securities within each feed are already separated into the 500 fast updating tickers and the 4000 slowly updating tickers. If this is not the case, then the separation can be easily achieved with a lookup. The query network in Figure 3 actually represents

six different queries (one for each output). Notice that much of the processing is shared.

The core of this application is in the detection of late tickers. Boxes 1, 2, 3, and 4 are all Aggregate boxes that perform the bulk of this computation. An Aggregate box groups input tuples by common value of one or more of their attributes, thus effectively creating a sub-stream for each possible combination of these attribute values. In this case, the aggregates are grouping the input on common value of ticker symbol. For each grouping or sub-stream, a window is defined that demarcates interesting runs of consecutive tuples called *windows*. For each of the tuples in one of these windows, some memory is allocated and an aggregating function (e.g., Average) is applied. In this example, the window is defined to be every consecutive pair (e.g., tuples 1 and 2, tuples 2 and 3, etc.) and the aggregating function generates one output tuple per window with a boolean flag called *Alarm*, which is a one when the second tuple in the pair is delayed (call this an *Alarm tuple*), and a zero when it is on time.

Aurora's operators have been designed to react to imperfections such as delayed tuples. Thus, the triggering of an Alarm tuple is accomplished directly using this built-in mechanism. The window defined on each pair of tuples will *timeout* if the second tuple does not arrive within the given threshold (5 seconds in this case). In other words, the operator will produce one alarm each time a new tuple fails to arrive within five seconds, as the corresponding window will automatically timeout and close. The high-level specification of Aggregate boxes 1 through 4 is:

```
Aggregate(Group by ticker,
          Order on arrival,
          Window (Size = 2 tuples,
                  Step = 1 tuple,
                  Timeout = 5 sec))
```

Boxes 5 through 8 are Filters that eliminate the normal outputs, thereby letting only the Alarm tuples through. Box 9 is a Union operator that merges all Reuters Alarms onto a single stream. Box 10 performs the same operation for Comstock.

The rest of the network determines when a large number of Alarms is occurring and what the cause of the problem might be.

Boxes 11 and 15 count Reuters Alarms and raise a high alarm when a threshold (100) is reached. Until that time, they simply pass through the normal (low) alarms. Boxes 14 and 18 do the same for Comstock. Note that the boxes labeled *Count 100* are actually Map boxes. Map takes a user-defined function as a parameter and applies it to each input tuple. That is, for each tuple t in the input stream, a Map box parameterized by a function f, produces the tuple $f(x)$. In this example, *Count 100* simply applies the following user-supplied function (written in pseudo code) to each tuple that passes through:

```
F (x:tuple) = cnt++
if (cnt % 100 != 0)
    if !suppress
        emit lo-alarm
    else
        emit drop-alarm
else
    emit hi-alarm
    set suppress = true
```

Boxes 12, 13, 16, and 17 separate the alarms from both Reuters and Comstock into alarms from NYSE and alarms from NASDAQ. This is achieved by using Filters to take NYSE alarms from both feed sources (Boxes 12 and 13) and merging them using a Union (Box 16). A similar path exists for NASDAQ Alarms. The results of each of these streams are counted and filtered as explained above.

In summary, this example illustrates the ability to share computation among queries, the ability to extend functionality through user-defined Aggregate and Map functions, and the need to detect and exploit stream imperfections.

3.2 The Linear Road Benchmark

Linear Road is a benchmark for stream processing engines [2,4]. This benchmark simulates an urban highway system that uses "variable tolling" (also known as "congestion pricing") [11,1,16], where tolls are determined according to such dynamic factors as congestion, accident proximity, and travel frequency. As a benchmark, Linear Road specifies input data schemas and workloads, a suite of continuous and historical queries that must be supported, and performance (query and transaction response time) requirements.

Variable tolling is becoming increasingly prevalent in urban settings because it is effective at reducing traffic congestion and because recent advances in micro-sensor technology make it feasible. Traffic congestion in major metropolitan areas is an increasing problem as expressways cannot be built fast enough to keep traffic flowing freely at peak periods. The idea behind variable tolling is to issue tolls that vary according to time-dependent factors such as congestion levels and accident proximity with the motivation of charging higher tolls during peak traffic periods to discourage vehicles from using the roads and contributing to the congestion. Illinois, California, and Finland are among the highway systems that have pilot programs utilizing this concept.

The benchmark itself assumes a fictional metropolitan area (called "Linear City") that consists of 10 expressways of 100 mile-long segments each, and 1,000,000 vehicles that report their positions via GPS-based sensors every 30 seconds. Tolls must be issued on a per-segment basis automatically, based on statistics gathered over the previous 5 minutes concerning average speed and number of reporting cars. A segment's

tolls are overridden when accidents are detected in the vicinity (an accident is detected when multiple cars report close positions at the same time), and vehicles that use a particular expressway often are issued "frequent traveler" discounts.

The Linear Road benchmark demands support for 5 queries: two continuous and three historical. The first continuous query calculates and reports a segment toll every time a vehicle enters a segment. This toll must then be charged to the vehicle's account when the vehicle exits that segment without exiting the expressway. Again, tolls are based on current congestion conditions on the segment, recent accidents in the vicinity, and the frequency of use of the expressway for the given vehicle. The second continuous query involves detecting and reporting accidents and adjusting tolls accordingly. The historical queries involve requesting an account balance or a day's total expenditure for a given vehicle on a given expressway, and a prediction of travel time between two segments on the basis of average speeds on the segments recorded previously. Each of the queries must be answered with a specified accuracy and within a specified response time. The degree of success for this benchmark is measured in terms of the number of expressways the system can support, assuming 1000 position reports issued per second per expressway, while answering each of the 5 queries within the specified latency bounds.

An early Aurora implementation of this benchmark supporting one expressway was demonstrated at SIGMOD 2003 [2].

3.3 Battalion Monitoring

We have worked closely with a major defense contractor on a battlefield monitoring application. In this application, an advanced aircraft gathers reconnaissance data and sends it to monitoring stations on the ground. This data includes positions and images of friendly and enemy units. At some point, the enemy units cross a given demarcation line and move toward the friendly units thereby signaling an attack.

Commanders in the ground stations monitor this data for analysis and tactical decision making. Each ground station is interested in particular subsets of the data, each with differing priorities. In the real application, the limiting resource is the bandwidth between the aircraft and the ground. When an attack is initiated, the priorities for the data classes change. More data becomes critical, and the bandwidth likely saturates. In this case, selective dropping of data is allowed in order to service the more important classes.

For our purposes, we built a simplified version of this application to test our load shedding techniques. Instead of modeling bandwidth, we assume that the limited resource is the CPU. We introduce load shedding as a way to save cycles.

Aurora supports two kinds of load shedding. The first technique inserts random drop boxes into the network. These

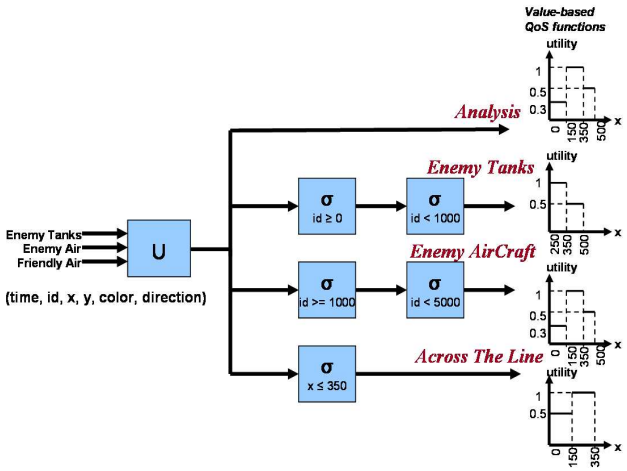


Fig. 4 Aurora Query Network for the Battlefi eld Monitoring Application

boxes discard a fraction of their input tuples chosen randomly. The second technique inserts semantic, predicate-based drop filters into the network. Based on QoS functions, system statistics (like operator cost and selectivity), and input rates, our algorithms choose the best drop locations, and the drop amount as indicated by a drop rate (random drop) or a predicate (semantic drop). Drop insertion plans are constructed and stored in a table in advance. As load levels change, drops are automatically inserted and removed from the query networks based on these plans [18].

One of the query networks that we used in this study is shown in Figure 4. There are four queries in this network. The *Analysis* query merges all tuples about positions of all units for analysis and archiving. The next two queries labeled *Enemy Tanks* and *Enemy Aircraft* select enemy tank and enemy aircraft tuples using predicates on their ids. The last query, *Across The Line*, selects all the objects that have crossed the demarcation line towards the friendly side.

Each query has a value-based QoS function attached to its output. A value-based QoS function maps the tuple values observed at an output to utility values that express the importance of a given result tuple. In this example, the functions are defined on the *x-coordinate* attribute of the output tuple which indicates where an object is positioned horizontally. The functions take values in the range [0, 500], of which 350 corresponds to the position of the vertical demarcation line. Initially all friendly units are on [0, 350] side of this line whereas enemy units are on the [350, 500] side. The QoS functions are specified by an application administrator and reflect the basic fact that tuples for enemy objects that have crossed the demarcation line are more important than others.

We ran this query network with tuples generated by the Aurora workload generator based on a battle scenario that we got from the defense contractor. We fed the input tuples

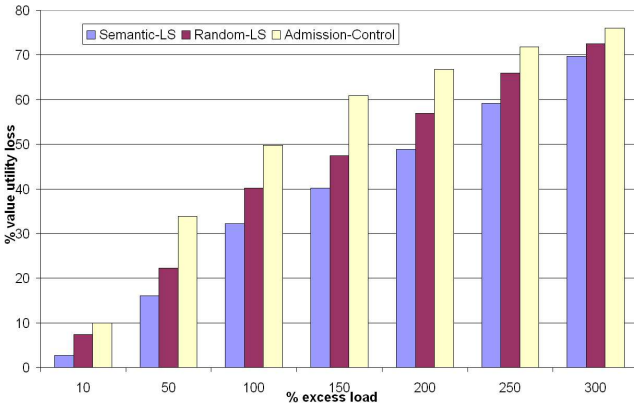


Fig. 5 Comparison of Various Load Shedding Approaches (%excess load vs. % value utility loss)

at different rates to create specific levels of overload in the network; then we let the load shedding algorithm remove the excess load by inserting drops to the network. Figure 5 shows the result. We compare the performance of three different load shedding algorithms in terms of their value utility loss (i.e., the average degradation in the QoS provided by the system) across all outputs at increasing levels of load.

We make the following important observations: First, our semantic load shedding algorithm, which drops tuples based on attribute values, achieves the least value utility loss at all load levels. Second, our random load shedding algorithm inserts drops of the same amounts at the same network locations as the semantic load shedder. Since tuples are dropped randomly, however, loss in value utility is higher compared to the semantic load shedder. As excess load increases the performance of the two algorithms becomes similar. The reason is that at high load levels, our semantic load shedder also drops tuples from the high utility value ranges. Lastly, we compare both of our algorithms against a simple admission control algorithm which sheds random tuples at the network inputs. Both our algorithms achieve lower utility loss compared to this algorithm. Our load shedding algorithms may sometimes decide to insert drops on inner arcs of the query network. On networks with box sharing among queries (e.g., the union box is shared among all four queries in Figure 4), inner arcs may be preferable to avoid utility loss at multiple query outputs. On the other hand, at very high load levels, since drops at inner arcs become insufficient to save the needed CPU cycles, our algorithms also insert drops close to the network inputs. Hence, all algorithms tend to converge to the same utility loss levels at very high loads.

3.4 Environmental Monitoring

We have also worked with a military medical research laboratory on an application that involves monitoring toxins in the water. This application is fed streams of data indicating fish behavior (e.g., breathing rate) and water quality (e.g., temperature, pH, oxygenation, and conductivity). When the fish behave abnormally, an alarm is sounded.

Input data streams were supplied by the army laboratory as a text file. The single data file interleaved fish observations with water quality observations. The alarm message emitted by Aurora contains fields describing the fish behavior, and two different water quality reports: the water quality at the time the alarm occurred and the water quality from the last time the fish behaved normally. The water quality reports contain not only the simple measurements, but also the 1-/2-/4-hour sliding window deltas for those values.

The application's Aurora processing network is shown in Figure 6 (snapshot taken from the Aurora GUI): The input port (1) shows where tuples enter Aurora from the outside data source. In this case, it is the application's C++ program that reads in the sensor log file. A Union box (2) serves merely to split the stream into two identical streams. A Map box (3) eliminates all tuple fields except those related to water quality. Each superbox (4) calculates the sliding window statistics for one of the water quality attributes. The parallel paths (5) form a binary join network that brings the results of (4)'s sub-networks back into a single stream. The top branch in (6) has all the tuples where the fish act oddly, and the bottom branch has the tuples where the fish act normally. For each of the tuples sent into (1) describing abnormal fish behavior, (6) emits an alarm message tuple. This output tuple has the sliding window water quality statistics for both the moment the fish acted oddly, and for the most recent previous moment that the fish acted normally. Finally the output port (7) shows where result tuples are made available to the C++-based monitoring application. Overall, the entire application ended up consisting of 3400 lines of C++ code (primarily for file-parsing and a simple monitoring GUI) and a 53-operator Aurora query network.

During the development of the application, we observed that Aurora's stream model proved very convenient for describing the required sliding-window calculations. For example, a single instance of the aggregate operator computed the 4-hour sliding-window deltas of water temperature.

Aurora's GUI for designing query networks also proved invaluable. As the query network grew large in the number of operators used, there was great potential for overwhelming complexity. The ability to manually place the operators and arcs on a workspace, however, permitted a visual representation of "subroutine" boundaries that let us comprehend the entire query network as we refined it.

We found that small changes in the operator language design would have greatly reduced our processing network complexity. For example, Aggregate boxes apply some window function (such as `DELTA(water-pH)`), to the tuples in a sliding window. Had an Aggregate box been capable of

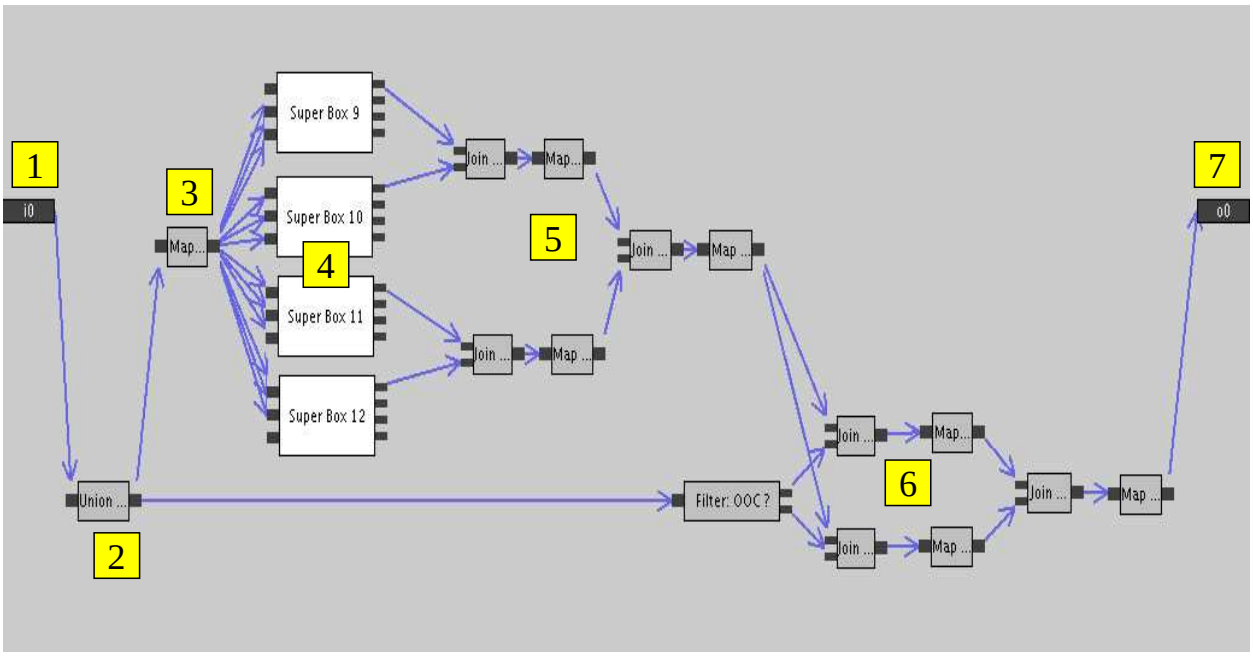


Fig. 6 Aurora Query Network for the Environmental Contamination Detection Applications (GUI snapshot)

evaluating multiple functions at the same time on a single window (such as `DELTA(water-pH)` and `DELTA(water-temp)`), we could have used significantly fewer boxes. Many of these changes have since been made to Aurora's operator language.

The ease with which the processing flow could be experimentally reconfigured during development, while remaining comprehensible, was surprising. It appears that this was only possible by having both a well-suited operator set, and a GUI tool that let us visualize the processing. It seems likely that this application was developed at least as quickly in Aurora as it would have been with standard procedural programming.

We note that, for this particular application, real-time response was not required. The main value Aurora added in this case was the ease of developing stream-oriented applications.

3.5 Medusa: Distributed Stream Processing

Medusa is a distributed stream-processing system built using Aurora as the single-site query processing engine. Medusa takes Aurora queries and distributes them across multiple nodes. These nodes can all be under the control of one entity or can be organized as a loosely coupled federation under the control of different autonomous participants.

A distributed stream-processing system such as Medusa offers several benefits:

1. It allows stream processing to be incrementally scaled over multiple nodes.
2. It enables high-availability because the processing nodes can monitor and take over for each other when failures occur.
3. It allows the composition of stream feeds from different participants to produce end-to-end services, and to take advantage of the distribution inherent in many stream processing applications (e.g., climate monitoring, financial analysis, etc.).
4. It allows participants to cope with load spikes without individually having to maintain and administer the computing, network, and storage resources required for peak operation. When organized as a loosely coupled federated system, load movements between participants based on pre-defined contracts can significantly improve performance.

Figure 7 shows the software structure of a Medusa node. There are two components in addition to the Aurora query processor. The *Lookup* component is a client of an inter-node distributed catalog that holds information on streams, schemas, and queries running in the system. The *Brain* handles query setup operations and monitors local load using information about the queues (*IOQueues*) feeding Aurora and statistics on box load. The *Brain* uses this information as input to a *bounded-price* distributed load management mechanism that converges efficiently to good load allocations [5].

The development of Medusa prompted two important changes to the Aurora processing engine. First, it became apparent that it would be useful to offer Aurora not only as a standalone system, but also as a library that could easily be integrated within a larger system. Second, we felt the need for an

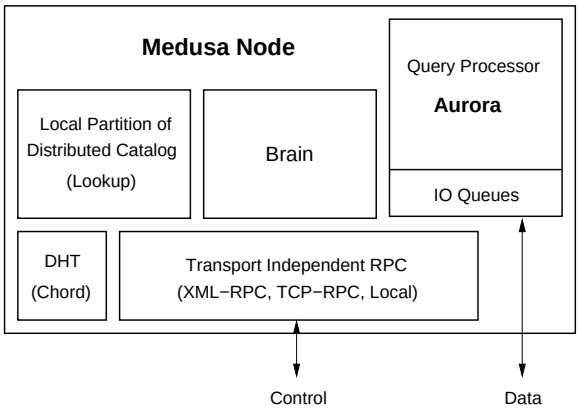


Fig. 7 Medusa Software Architecture

Table 1 Overview of a Subset of the Aurora API

`start` and `shutdown`: Respectively starts processing and shuts down a complete query-network.
`modifyNetwork`: At runtime, adds or removes schemas, streams and operator boxes from a query network processed by a single Aurora engine.
`typecheck`: Validates (part of) a query network. Computes properties of intermediate and output streams.
`enqueue` and `dequeue`: Push and pull tuples on named streams.
`listEntities` and `describe(Entity)`: Provide information on entities in the current query network.
`getPerfStats`: Provides performance and load information.

Aurora API, summarized in Table 1. This API is composed of three types of methods: (1) methods to set up queries and push or pull tuples from Aurora, (2) methods to modify query networks at runtime (operator additions and removals) and (3) methods giving access to performance information.

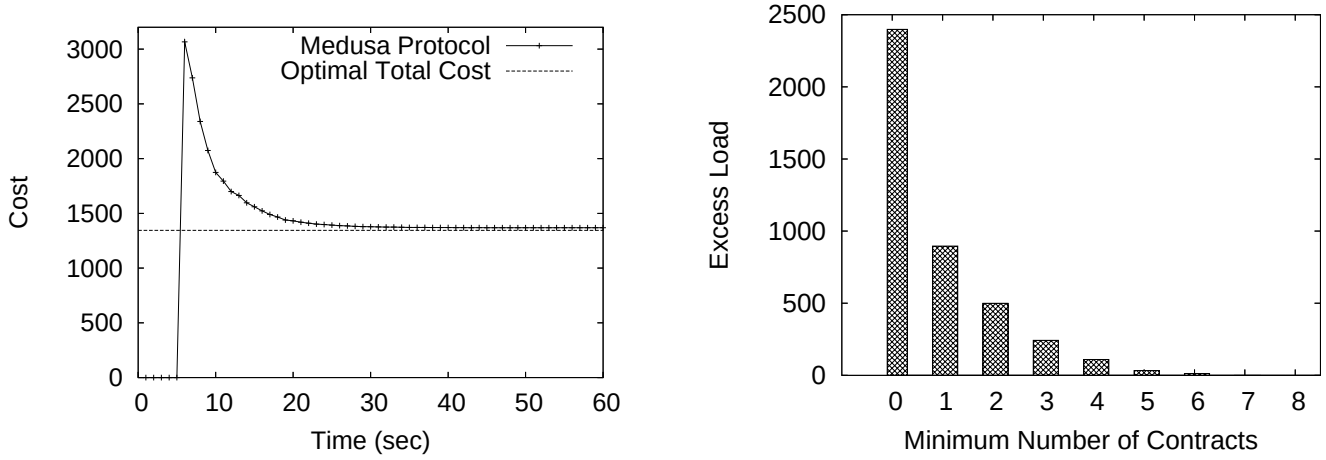
Load movement. To move operators with a relatively low effort and overhead compared to full- blown process migration, Medusa participants use *remote definitions*. A remote definition maps an operator defined at a node onto an operator defined at another. At runtime, when a path of operators in the boxes-and-arrows diagram needs to be moved to another node, all that is required is for the corresponding operators to be instantiated remotely and for the incoming streams to be diverted to the appropriately named inputs on the new node.

For some operators, internal operator state may need to be moved when a task moves between machines, unless some "amnesia" is acceptable to the application. Our current prototype restarts operator processing after a move from a fresh state and the most recent position of the input streams. To support the movement of operator state, we are adding two new functions to the Aurora API and are modifying the Aurora engine. The first method freezes a query network and removes an operator with its state by performing the following sequence of actions atomically: stop all processing, remove a box from a query network, extract the operator's internal state, subscribe an outside client to what used to be the operator's input streams, and re- start processing. The second method performs the converse actions atomically. It stops processing, adds a box to a query network, initializes the box's state, and re-starts processing. To minimize the amount of state moved, we are exploring freezing operators around the windows of tuples on which they operate, rather than at random instants. When Medusa moves an operator or a group of operators, it handles the forwarding of tuples to their new locations.

Medusa employs an agoric system model to create incentives for autonomous participants to handle each other's load. Clients outside the system pay Medusa participants for processing their queries and Medusa participants pay each other to handle load. Payments and load movements are based on *pairwise contracts* negotiated offline between participants. These contracts set tightly bounded prices for migrating each unit of load and specify the set of tasks that each participant is willing to execute on behalf of its partner. Contracts can also be customized with availability, performance, and other clauses. Our mechanism, called the *bounded-price mechanism*, thus allows participants to manage their excess load through private and customized service agreements. The mechanism also achieves a low runtime overhead by bounding prices through offline negotiations.

Figure 8 shows the simulation results of a 995-node Medusa system running the bounded-price load management mechanism. Figure 8(a) shows that convergence from an unbalanced load assignment to an almost optimal distribution is fast with our approach. Figure 8(b) shows the excess load remaining at various nodes for increasing numbers of contracts. A minimum of just seven contracts per node in a network of 995 nodes ensures that all nodes operate within capacity when capacity exists in the system. The key advantages of our approach over previous distributed load management schemes are (1) lower runtime overhead, (2) possibility of service customization and price discrimination, and (3) relatively invariant prices that a participant pays another for processing a unit of load.

High availability. We are also currently exploring the runtime overhead and recovery time tradeoffs between different approaches to achieve high-availability (HA) in distributed stream processing, in the context of Medusa and Aurora* [4]. These approaches range from classical Tandem-style process-pairs [6] to using upstream nodes in the processing flow as backup for their downstream neighbors. Different approaches also provide different recovery semantics where either: (1) some tuples are lost, (2) some tuples are re-processed, or (3) operations take-over precisely where the failure happened. We discuss these algorithms in more de-



(a) convergence speed with a minimum of 7 contracts/node

(b) final allocation for increasing number of contracts

Fig. 8 Performance of Medusa Load Management Protocol

tail in [13]. An important HA goal for the future is handling network partitions in addition to individual node failures.

4 Lessons Learned

4.1 Support for Historical Data

From our work on a variety of streaming applications, it became apparent that each required maintaining and accessing a collection of historical data. For example, the Linear Road benchmark, which represents a realistic application, required maintaining ten weeks of toll history for each driver, as well as the current positions of every vehicle and the locations of accidents tying up traffic. Historical data might be used to support *historical queries* (e.g., tell me how much driver X has spent on tolls on expressway Y over the past 10 weeks), or serve as inputs to *hybrid* queries involving both streaming and historical data (e.g., report the current toll for vehicle X based on its current position (streamed data) and the presence of any accidents in its vicinity (historical data)).

In the applications we have looked at, historical data takes three different forms. These forms differ by their *update patterns*: the means by which incoming stream data is used to update the contents of a historical collection. These forms are summarized below.

1. **Open Windows (Connection Points)**: Linear Road requires maintaining the *last 10 weeks worth of toll data for each driver* to support both historical queries and integrated queries. This form of historical data resembles a window in its FIFO-based update pattern, but must be shared by multiple queries and therefore openly accessible.
2. **Aggregate Summaries (Latches)**: Linear Road requires maintaining such aggregated historical data as: the current toll balance for every vehicle (`SUM(Toll)`), the last reported position of every vehicle (`MAX(Time)`), and the average speed on a given segment over the past 5 minutes (`AVG(Speed)`). In all cases, the update patterns involve maintaining data by key value (e.g., vehicle or segment id) and using incoming tuples to update the aggregate value that has the appropriate key. As with open windows, aggregate summaries must be shared by multiple queries and therefore must be openly accessible.
3. **Tables**: Linear Road requires maintaining tables of historical data whose update patterns are arbitrary and determined by the values of streaming data. For example, a table must be maintained that holds every accident that has yet to be cleared (such that an accident is detected when multiple vehicles report the same position at the same time). This table is used to determine tolls for segments in the vicinity of the accident and to alert drivers approaching the scene of the accident. The update pattern for this table resembles neither an open window nor an aggregate summary. Rather, accidents must be deleted from the table when an incoming tuple reports that the accident has been cleared. This requires the declaration of an arbitrary update pattern.

Whereas open windows and aggregate summaries have fixed update patterns, tables require update patterns to be explicitly specified. Therefore, the Aurora query algebra (SQuAl) includes an "Update box" that permits an update pattern to be specified in SQL. This box has the form,

```
UPDATE (Assume O, SQL U, Report t)
```

such that U is an SQL update issued with every incoming tuple, and including variables that get instantiated with the values contained in that tuple. O specifies the assumed ordering of input tuples, and t specifies a tuple to output whenever an update takes place. Further, because all three forms of historical collections require random access, SQuAl also includes a "Read box" that initiates a query over stored data (also specified in SQL) and returns the result as a stream. This box has the form,

```
READ (Assume O, SQL Q)
```

such that Q is an SQL query issued with every incoming tuple, and including variables that get instantiated with the values contained in that tuple.

In short, the streaming applications we have looked at share the need for maintaining and randomly accessing collections of historical data. These collections, used for both historical and hybrid queries, are of three forms differing by their update patterns. To support historical data in Aurora, we include an update operation (to update tables with user-specified update patterns) and a read operation (to read any of the forms of historical data).

4.2 Synchronization

As continuous queries, stream applications inherently rely on shared data and computation. Shared data might be contained in a table that one query updates and another query reads. For example, the Linear Road application requires that vehicle position data be used to update statistics on highway usage which in turn are read to determine tolls for each segment on the highway. Alternatively, box output can be shared by multiple queries to exploit common sub-expressions, or even by a single query as a way of merging intermediate computations after parallelization.

Transactions are required in traditional databases because data sharing can lead to data inconsistencies. An equivalent synchronization mechanism is required in streaming settings, as data sharing in this setting can also lead to inconsistencies. For example, if a toll charge can expire, then a toll assessment to a given vehicle should be delayed until a new toll charge is determined. The need for synchronization with data sharing is achieved in SQuAl via the "WaitFor box" whose syntax is shown below:

```
WaitFor (P: Predicate, T: Timeout)
```

This binary operator buffers each tuple t on one input stream until a tuple arrives on the second input stream that with t satisfies P (or until the timeout expires, in which case t is discarded). If a Read operation must follow a given Update operation, then a WaitFor can buffer the Read request (tuple) until a tuple output by the Update box (and input to the second input of WaitFor) indicates that the Read operation can proceed.

In short, the inherent sharing possible in streaming environments makes it sometimes necessary to synchronize operations to ensure data consistency. We currently implement synchronization in SQuAl with a dedicated operator.

4.3 Resilience to Unpredictable Stream Behavior

Streams are by their nature unpredictable. Monitoring applications require the system to continue operation even when the unpredictable happens. Sometimes, the only way to do this is to produce approximate answers. Obviously, in these cases, the system should try to minimize errors.

We have seen examples of streams that do not behave as expected. The financial services application that we described earlier requires the ability to detect a problem in the arrival rate of a stream. The military application must fundamentally adjust its processing to fit the available resources during times of stress. In both of these cases, Aurora primitives for unpredictable stream behavior were brought to bear on the problem.

Aurora makes no assumptions that a data stream arrive in any particular order or with any temporal regularity. Tuples can be late or out of order due to the nature of the data sources, the network that carries the streams, or due to the behavior of the operators themselves. Accordingly, our operator set includes user-specified parameters that allow handling such "damaged" streams gracefully.

For many of the operators, an input stream can be specified to obey an expected order. If out-of-order data is known to the network designer not to be of relevance, the operator will simply drop such data tuples immediately. Nonetheless, Aurora understands that this may at times be too drastic a constraint, and provides an optional slack parameter to allow for some tolerance in the number of data tuples that may arrive out of order. A tuple that arrives out-of-order within the slack bounds will be processed as if it had arrived in order.

With respect to possible irregularity in the arrival rate of data streams, the Aurora operator set offers all windowed-operators an optional timeout parameter. The timeout parameter tells the operator how long to wait for the next data tuple to arrive. This has two benefits: it prevents blocking (i.e. no output) when one stream is stalled, and it offers another way for the network designer to characterize the value of data that arrives later than it should, as in the financial services application in which the timeout parameter was used to determine when a particular data packet arrived late.

4.4 XML and Other Feed Formats Adaptor Required

Aurora provides a network protocol that may be used to enqueue and dequeue tuples via Unix or TCP sockets. The protocol is intentionally very low-level: to eliminate copies and improve throughput, the tuple format is closely tied to the format of Aurora's internal queue format. For instance, the protocol requires that each packet contain a fixed amount of padding reserved for bookkeeping, and that integer and floating-point fields in the packet match the architecture's native format.

While we anticipate that performance-critical applications will use our low-level protocol, we also recognize that the formats of Aurora's input streams may be outside the immediate control of the Aurora user or administrator; e.g., stock quote data arriving in XML format from a third-party information source. Also, even if the streams are being generated or consumed by an application within an organization's control, in some cases protocol stability and portability (e.g., not requiring the client to be aware of the endian-ness of the server architecture) are important enough to justify a minor performance loss.

One approach to address these concerns is to simply require the user to build a proxy application that accepts tuples in the appropriate format, converts them to Aurora's internal format, and pipes them into the Aurora process. This approach, while simple, conflicts with one of Aurora's key design goals-to minimize the number of boundary crossings in the system-since the proxy application would be external to Aurora and hence live in its own address space.

We resolve this problem by allowing the user to provide plug-ins called *converter boxes*. Converter boxes are shared libraries that are dynamically linked into the Aurora process space; hence their use incurs no boundary crossings. A user-defined *input converter box* provides a hook that is invoked when data arrive over the network. The implementation may examine the data and inject tuples into the appropriate streams in the Aurora network. This may be as simple as consuming fixed-length packets and enforcing the correct byte-order on fields, or as complex as transforming fully-formed XML documents into tuples. An *output converter box* performs the inverse function: it accepts tuples from streams in Aurora's internal format and converts them into a byte stream to be consumed by an external application.

Input and output converter boxes are powerful connectivity mechanisms: they provide a high level of flexibility in dealing with external feeds and sinks without incurring a performance hit. This combination of flexibility and high performance is essential in a streaming database that must assimilate data from a wide variety of sources.

4.5 Programmatic Interfaces and Globally-Accessible Catalogs are a Good Idea

Initially, Aurora networks were created using the GUI and all Aurora metadata (i.e., catalogs) were stored in an internal representation. Our experience with the Medusa system quickly made us realize that, in order for Aurora to be easily integrated within a larger system, a higher-level, *programmatic interface* is needed to script Aurora networks and metadata need to be globally accessible and updatable.

Although we initially assumed that only Aurora itself (i.e., the runtime and the GUI) would need direct access to the catalog representation, we encountered several situations where this assumption did not hold. For instance, in order to manage distribution operation across multiple Aurora nodes, Medusa required knowledge of the contents of nodes' catalogs and the ability to selectively move parts of catalogs from node to node. Medusa needed to be able to create catalog objects (schema, streams, and boxes) without direct access to the Aurora catalog database, which would have violated abstraction. In other words, relying on the Aurora runtime and GUI as the sole software components able to examine and modify catalog structures turned out to be an unworkable solution when we tried to build sophisticated applications on the Aurora platform. We concluded that we needed a simple, transparent, catalog representation that is easily readable and writable by external applications. This would make it much easier to write higher-level systems that use Aurora (such as Medusa) and alternative authoring tools for catalogs.

To this end, Aurora currently incorporates appropriate interfaces and mechanisms (see Section 3.5) to make it easy to develop external applications to inspect and modify Aurora query networks. A universally readable and writable catalog representation is crucial in an environment where multiple applications may operate on Aurora catalogs.

4.6 Performance Critical

During the development of Aurora, our primary tool for keeping performance in mind was a series of "micro-benchmarks". Each of these benchmarks measured the performance of a small part of our system, such as a single operator, or the raw performance of the message bus. These benchmarks allowed us to measure the merits of changes to our implementation quickly and easily.

Fundamental to an SPE is a high performance "message bus". This is the system that moves tuples from one operator to the next, storing them temporarily, as well as in to and out of the query network. Since every tuple is passed on the bus a number of times, this is definitely a performance bottleneck. Even such trivial optimizations as choosing the right `memcpy()` implementation gave substantial improvements to the whole system.

Table 2 Micro-benchmark results

	Query(q)	# Dequers(d)	Batch size(b)	Average Latency
A	NULL	0	1	1211 ns
B	NULL	0	10	176 ns
C	NULL	0	100	70 ns
D	NULL	0	1000	60 ns
E	NULL	1	10	321 ns
F	NULL	1	100	204 ns
G	NULL	1	1000	191 ns
H	NULL	5	1000	764 ns
I	NULL	10	1000	1748 ns
J	FILTER	1	1000	484 ns
K	UNION	1	1000	322 ns
L	UNION-CHAIN	1	1000	858 ns

Second to the message bus, the scheduler is the core element of an SPE. The scheduler is responsible for allocating processor time to operators. It is tempting to decorate the scheduler with all sorts of high level optimization, such as intelligent allocation of processor time or real-time profiling of query plans. But it is important to remember that scheduler overhead can be substantial in networks where there are many operators, and that the scheduler makes no contribution to the actual processing. All addition of scheduler functionality must be greeted with skepticism, and should be aggressively profiled.

Once the core of the engine has been aggressively optimized, the remaining hot spots for performance are to be found in the implementation of the operators. In our implementation, each operator has a "tight loop", which processes batches of input tuples. This loop is a prime target for optimization. We make sure nothing other than necessary processing occurs in the loop. In particular, housekeeping of data structures such as memory allocations and deallocation needs to be done outside of this loop, so that its cost can be amortized across many tuples.

Data structures are another opportunity for operator optimization. Many of our operators are stateful; they retain information or even copies of previous input. Because these operators are asked to process and store large numbers of tuples, efficiency of these data structures is important. Ideally, processing of each input tuple is accomplished in constant time. In our experience, processing that is linear in the amount of state stored is unacceptable.

In addition to the operators themselves, any parts of the system that are used by those operators in the tight loops must be carefully examined. For example, we have a small language used to specify expressions for Map operators. Because these expressions are evaluated in such tight loops, optimizing them was important. The addition of an expensive compilation step may even be appropriate.

To assess the relative performance of various parts of the Aurora system, we developed a simple series of micro-benchmarks. Each micro-benchmark follows the following pattern:

1. Initialize Aurora using a query network q.
2. Create d dequeuers receiving data from the output of the query network. (If d is 0, then there are no dequeuers, i.e., tuples are discarded as soon as they are output.)
3. Begin a timer.
4. Enqueue n tuples into the network, in batches of b tuples at a time. Each tuple is 64 bytes long.
5. Wait until the network is drained, i.e., every box is done processing every input tuple, and every dequeuer has received every output tuple. Stop the timer. Let t be the amount of time required to process each input tuple, i.e., the total amount of time passed divided by n.

For the purposes of this benchmark, we fixed n at 2,000,000 tuples. We used several different catalogs. Note that these networks are functionally identical: every input tuple is output to the dequeuers, and the only difference is the type and amount of processing done to each the tuple. This is necessary to isolate the impact of each stage of tuple processing; if some networks returned a different number of tuples, any performance differential might be attributed simply to there being less or more work to do because of the different number of tuples to enqueue or dequeue.

- NULL: A catalog with no boxes, i.e., input values are passed directly to dequeuers.
- FILTER: A catalog with a filter box whose condition is true for each tuple.
- UNION: A union box that combines the input stream with an empty stream.
- UNION-CHAIN: A chain of five union boxes, each of which combines the input stream with an empty stream.

Table 2 shows the performance of the benchmark with various settings of q, d, and b.

We observe that the overhead to enqueue a tuple in Aurora is highly dependent on the batch size, but for large batch sizes settles to 60 ns. Dequeuers add a somewhat higher overhead (between 130 ns (G-D) and 200 ns (I-H)/5] each) because currently one copy of each tuple is made per dequeuer. Comparing cases G and K, or cases G and L, we see that adding a box on a tuple path incurs a delay of approximately 130 ns per tuple; evaluating a simple comparison predicate on a tuple adds about 160 ns (J-K).

These micro-benchmarks measure the overhead involved in passing tuples into and out of Aurora boxes and networks; they do not measure the time spent in boxes performing non-trivial operations such as joining and aggregation. Messaging passing overhead, however, can be significant time sink in streaming databases (as it was in earlier versions of Aurora). Micro benchmarking was very useful in eliminating performance bottlenecks in Aurora's message-passing infrastructure. This infrastructure is now fast enough in Aurora that non-trivial box operations are the only noticeable bottleneck; i.e., CPU time is overwhelmingly devoted to useful work and not simply shuffling around tuples.

5 Future Plans: Borealis

The Aurora team has secured venture capital backing to commercialize the current code line. Some of the group is morphing into pursuing this venture. Because of this event, there is no reason for the Aurora team to improve the current system. This section presents the initial ideas that we plan to explore in a follow-on system, called Borealis, which is a distributed stream processing system. Borealis inherits core stream processing functionality from Aurora and distribution functionality from Medusa. Borealis modifies and extends both systems in non-trivial and critical ways to provide advanced capabilities that are commonly required by newly-emerging stream processing applications.

The Borealis design is driven by our experience in using Aurora and Medusa, in developing several streaming applications including the Linear Road Benchmark, and several commercial opportunities. Borealis will address the following requirements of newly-emerging streaming applications.

5.1 Dynamic Revision of Query Results

In many real-world streams, corrections or updates to previously processed data are available only after the fact. For instance, many popular data streams, such as the Reuters stock market feed, often include messages that allow the feed originator to correct errors in previously reported data. Furthermore, stream sources (such as sensors), as well as their connectivity, can be highly volatile and unpredictable. As a result, data may arrive late and miss its processing window, or may be ignored temporarily due to an overload situation. In all these cases, applications are forced to live with imperfect results, unless the system has means to correct its processing and results to take into account newly available data or updates.

The Borealis data model will extend that of Aurora by supporting such corrections by way of revision records. The goal is to process revisions intelligently, correcting query results that have already been emitted in a manner that is consistent with the corrected data. Processing of a revision message must replay a portion of the past with a new or modified value. Thus, to process revision messages correctly, we must make a query diagram "replayable". In theory, we could process each revision message by replaying processing from the point of the revision to the present. In most cases, however, revisions on the input affect only a limited subset of output tuples, and to regenerate unaffected output is wasteful and unnecessary. To minimize run-time overhead and message proliferation, we assume a closed model for replay that generates revision messages when processing revision messages. In other words, our model processes and generates "deltas" showing only the effects of revisions rather than regenerating the entire result. The primary challenge here is to develop efficient revision processing techniques that can work with bounded history.

5.2 Dynamic Query Modification

In many stream processing applications, it is desirable to change certain attributes of the query at run time. For example, in the financial services domain, traders typically wish to be alerted of *interesting* events, where the definition of "interesting" (i.e., the corresponding filter predicate) varies based on current context and results. In network monitoring, the system may want to obtain more precise results on a specific subnetwork, if there are signs of a potential Denial-of-Service attack. Finally, in a military stream application that MITRE [19] explained to us, they wish to switch to a "cheaper" query when the system is overloaded. For the first two applications, it is sufficient to simply alter the operator parameters (e.g., window size, filter predicate), whereas the last one calls for altering the operators that compose the running query. Another motivating application comes again from the financial services community. Universally, people working on trading engines wish to test out new trading strategies as well as debug their applications on historical data before they go live. As such, they wish to perform "time travel" on input streams. Although this last example can be supported in most current SPE prototypes (i.e., by attaching the engine to previously stored data), a more user- friendly and efficient solution would obviously be desirable.

Two important features that will facilitate on-line modification of continuous queries in Borealis are *control lines* and

time travel. Control lines extend Aurora's basic query model with the ability to change operator parameters as well as operators themselves on the fly. Control lines carry messages with revised box parameters and new box functions. For example, a control message to a Filter box can contain a reference to a boolean-valued function to replace its predicate. Similarly, a control message to an Aggregate box may contain a revised window size parameter. Additionally, each control message must indicate when the change in box semantics should take effect. Change is triggered when a monotonically increasing attribute received on the data line attains a certain value. Hence, control messages specify an <attribute, value> pair for this purpose. For windowed operators like Aggregate, control messages must also contain a flag to indicate if open windows at the time of change must be prematurely closed for a clean start.

Time travel allows multiple queries (different queries or versions of the same query) to be easily defined and executed concurrently, starting from different points in the past or "future" (typically by running a simulation of some sort). In order to support these capabilities, we leverage three advanced mechanisms in Borealis: enhanced connection points, connection point versions, and revision messages. To facilitate time travel, we define two new operations on connection points. The *replay operation* replays messages that are stored at a connection point from an arbitrary message in the past. The *offset operation* is used to set the connection point offset in time. When offset into the past, a connection point delays current messages before pushing them downstream. When offset into the future, the connection point predicts future data. When producing future data, various prediction algorithms can be used based on the application. A connection point version is a distinctly named logical copy of a connection point. Each named version can be manipulated independently. It is possible to shift a connection point version backward and forward in time without affecting other versions.

To replay history from a previous point in time t, we use revision messages. When a connection point receives a replay command, it first generates a set of revision messages that delete all the messages and revisions that occurred since t. To avoid the overhead of transmitting one revision per deleted message, we use a macro message that summarizes all deletions. Once all messages are deleted, the connection point produces a series of revisions that insert the messages and possibly their following revisions back into the stream. During replay, all messages and revisions received by the connection point are buffered and processed only after the replay terminates thus ensuring that simultaneous replays on any path in the query diagram are processed in sequence and do not conflict. When offset into the future, time-offset operators predict future values. As new data becomes available, these predictors can (but do not have to) produce more accurate revisions to their past predictions. Additionally, when a predictor receives revision messages, possibly due to time travel into the past, it can also revise its previous predictions.

5.3 Distributed Optimization

Currently, commercial stream processing applications are popular in industrial process control (e.g., monitoring oil refineries and cereal plants), financial services (e.g., feed processing, trading engine support and compliance), and network monitoring (e.g., intrusion detection, fraud detection). Here we see a *server-heavy* optimization problem - the key challenge is to process high-volume data streams on a collection of resource-rich "beefy" servers. Over the horizon, we see a very large number of applications of wireless sensor technology (e.g., RFID in retail applications, cell phone services). Here, we see a *sensor-heavy* optimization problem - the key challenges revolve around extracting and processing sensor data from a network of resource-constrained "tiny" devices. Further over the horizon, we expect sensor networks to become faster and increase in processing power. In this case the optimization problem becomes more balanced, becoming *sensor-heavy/server-heavy*. To date systems have exclusively focused on either a server-heavy environment, or a sensor-heavy environment. Off into the future, there will be a need for a more flexible optimization structure that can deal with a very large number of devices and perform cross-network sensor-heavy/server-heavy resource management and optimization.

The purpose of the Borealis optimizer is threefold. First, it is intended to optimize processing across a combined sensor and server network. To the best of our knowledge, no previous work has studied such a cross-network optimization problem. Second, QoS is a metric that is important in stream-based applications, and optimization must deal with this issue. Third, scalability, size-wise and geographical, is becoming a significant design consideration with the proliferation of stream-based applications that deal with large volumes of data generated by multiple distributed sensor networks. As a result, Borealis faces a unique, multi-resource, multi-metric optimization challenge that is significantly different than those explored in the past. Our current thinking is that Borealis will rely on a hierarchical, distributed optimizer that runs at different time-granularities.

Another part of the Borealis vision involves addressing recovery and high availability issues. High availability demands that node failure is masked by seamless handoff of processing to an alternate node. This is complicated by the fact that the optimizer will dynamically redistribute processing, making it more difficult to keep backup nodes synchronized. Furthermore, wide-area Borealis applications are not only vulnerable to node failures but also to network failures and more importantly to network partitions. We have preliminary research in this area that leverages Borealis mechanisms

including connection point versions, revision tuples, and time travel.

5.4 Implementation Plans

We have started building Borealis. As Borealis inherits much of its core stream processing functionality from Aurora, we can effectively borrow many of the Aurora modules, including the GUI, the XML representation for query diagrams, portions of the run-time system, and much of the logic for boxes. Similarly, we are borrowing some networking and distribution logic from Medusa. With this starting point, we hope to have a working prototype within a year.

Acknowledgements This work was supported in part by the National Science Foundation under the grants IIS-0086057, IIS-0325525, IIS-0325703, and IIS-0325838; and by the Army contract DAMD17-02-2-0048. We would like to thank all members of the Aurora and the Medusa projects at Brandeis University, Brown University, and M.I.T. We are also grateful to the anonymous reviewers for their invaluable comments.

References

1. A guide for hot lane development: A U.S. department of transportation federal highway administration. `http://www.itsdocs.fhwa.dot.gov/JPODOCS/REPTSTE/13668.html`.
2. D. Abadi, D. Carney, U. Çetintemel, 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, and S. Zdonik. Aurora: A Data Stream Management System (demo description). In *ACM SIGMOD Conference*, June 2003.
3. D. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A New Model and Architecture for Data Stream Management. *The VLDB Journal*, 12(2), August 2003.
4. A. Arasu, M. Cherniack, E. Galvez, D. Maier, A. Maskey, E. Ryvkina, M. Stonebraker, and R. Tibbetts. Linear Road: A Benchmark for Stream Data Management Systems. In *VLDB Conference*, Toronto, Canada, August 2004. (to appear).
5. M. Balazinska, H. Balakrishnan, and M. Stonebraker. Contract-Based Load Management in Federated Distributed Systems. In *NSDI Symposium*, March 2004.
6. J. Barlett, J. Gray, and B. Horst. Fault Tolerance in Tandem Computer Systems. Technical Report TR-86.2, Tandem Computers, March 1986.
7. D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring Streams - A New Class of Data Management Applications. In *VLDB Conference*, Hong Kong, China, August 2002.
8. D. Carney, U. Çetintemel, A. Rasin, S. Zdonik, M. Cherniack, and M. Stonebraker. Operator Scheduling in a Data Stream Manager. In *VLDB Conference*, Berlin, Germany, September 2003.
9. S. Chandrasekaran, A. Deshpande, M. Franklin, J. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. Shah. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *CIDR Conference*, January 2003.
10. M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Çetintemel, Y. Xing, and S. Zdonik. Scalable Distributed Stream Processing. In *CIDR Conference*, Asilomar, CA, January 2003.
11. Congestion pricing: A report from intelligent transportation systems (ITS). `http://www.path.berkeley.edu/leap/TTM/DemandManage/pricing.html`.
12. D. DeWitt, J. Naughton, and D. Schneider. An Evaluation of Non-Equijoin Algorithms. In *VLDB Conference*, Barcelona, Catalonia, Spain, September 1991.
13. J. Hwang, M. Balazinska, A. Rasin, U. Çetintemel, M. Stonebraker, and S. Zdonik. A Comparison of Stream-Oriented High-Availability Algorithms. Technical Report CS-03-17, Department of Computer Science, Brown University, October 2003.
14. A. Lerner and D. Shasha. AQuery: Query Language for Ordered Data, Optimization Techniques, and Experiments. In *VLDB Conference*, Berlin, Germany, September 2003.
15. R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma. Query Processing, Approximation, and Resource Management in a Data Stream Management System. In *CIDR Conference*, January 2003.
16. R. W. Poole. Hot lanes prompted by federal program. `http://www.rppi.org/federalhotlanes.html`.
17. P. Seshadri, M. Livny, and R. Ramakrishnan. SEQ: A Model for Sequence Databases. In *IEEE ICDE Conference*, Taipei, Taiwan, March 1995.
18. N. Tatbul, U. Çetintemel, S. Zdonik, M. Cherniack, and M. Stonebraker. Load Shedding in a Data Stream Manager. In *VLDB Conference*, Berlin, Germany, September 2003.
19. The MITRE Corporation. `http://www.mitre.org/`.
20. The US Army Medical Research and Materiel Command. `http://mrmc-www.army.mil/`.