The Aurora and Borealis Stream Processing Engines

Uğur Çetintemel[1], Daniel Abadi[2], Yanif Ahmad[1], Hari Balakrishnan[2], Magdalena Balazinska[2], Mitch Cherniack[3], Jeong-Hyon Hwang[1], Wolfgang Lindner[2], Samuel Madden[2], Anurag Maskey[3], Alexander Rasin[1], Esther Ryvkina[3], Mike Stonebraker[2], Nesime Tatbul[1], Ying Xing[1], and Stan Zdonik[1]

[1] Department of Computer Science, Brown University
[2] Department of EECS and Laboratory of Computer Science, M.I.T.
[3] Department of Computer Science, Brandeis University

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 Online Transaction Processing (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. The need for real-time answers 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 are stored before being processed, cannot deliver real-time latency, as required by SPEs. To meet more stringent latency requirements, SPEs must adopt an alternate model, which we refer to as "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. Incoming traffic is usually 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 MIT. 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 [5].

We have used Aurora to build various application systems. The first application we describe here is an Aurora implementation of Linear Road, mentioned above. Second, we have implemented a pilot application that detects late arrival of messages in a financial-services feed-processing environment. Third, one of our collaborators, a military medical research laboratory [19], asked us to build a system to monitor the levels of hazardous materials in fish. Lastly, we have used Aurora to build Medusa [6], a distributed version of Aurora that is intended to be used by multiple enterprises that operate in different administrative domains.

The current Aurora prototype has been 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. Borealis is a distributed stream-processing system that inherits core stream-processing functionality from Aurora and distribution functionality from Medusa. Borealis modifies and extends both systems in nontrivial 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.

We start with a review of the Aurora design and implementation in Section 2. We then present the case studies mentioned above in detail in Section 3 and provide a brief retrospective on what we have learned throughout the process in Section 4. We conclude in Section 5 by briefly discussing 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 The Aurora Centralized Stream Processing Engine

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

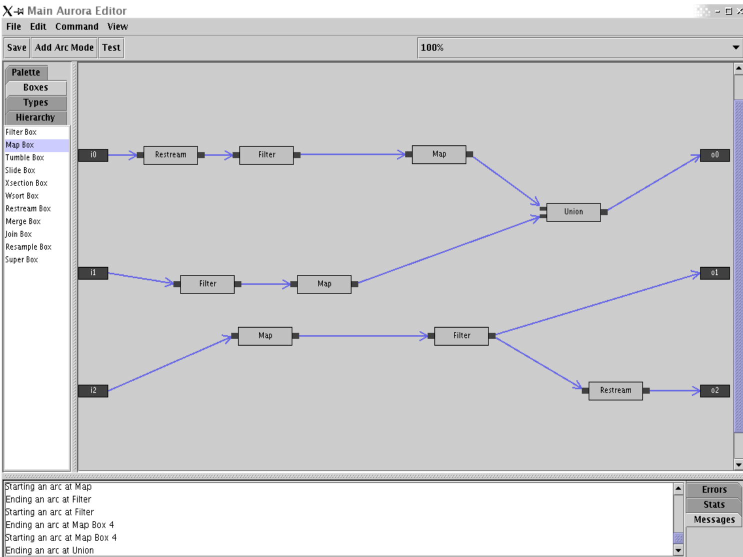

Fig. 1. Aurora Graphical User Interface

and forth to an application for the rest of the work, thereby adding to system overhead and latency.

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 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.

As illustrated in Figure 2, the Aurora system is in fact a directed sub-graph of a workflow diagram that expresses all simultaneous query computations. We refer to this workflow diagram as the Aurora network. Queries are built from a standard set of well-defined operators (a.k.a. boxes). The arcs denote tuple queues that represent streams. Each box, when scheduled, processes one or more tuples from its input queue and puts the results on its output queue. When tuples are generated at the queues attached to the applications, they are assessed according to the application's QoS specification (more on this below).

By default, queries are continuous in that they can potentially run forever over push-based inputs. Ad hoc queries can also be defined at run time and are

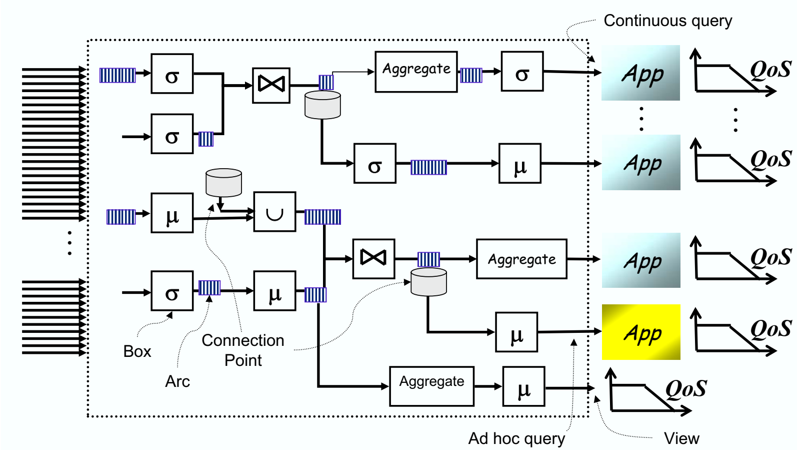

Fig. 2. The Aurora Processing Network

attached to connection points, which are predetermined arcs in the network where historical data is stored. Connection points can be associated with persistence specifications that indicate how long a history to keep. Aurora also allows dangling connection points that can store static data sets. As a result, connection points enable Aurora queries to combine traditional pull-based data with live push-based data. Aurora also allows the definition of views, which are queries to which no application is connected. A view is allowed to have a QoS specification as an indication of its importance. Applications can connect to the view whenever there is a need.

The Aurora operators are presented in detail in [4] and are summarized in Figure 3. 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 con-

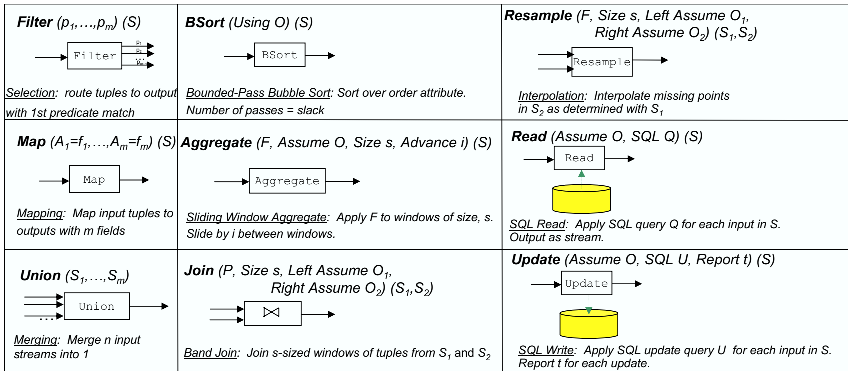

Fig. 3. Aurora Operators

nected 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. Specifically, QoS is a multidimensional function of several attributes of an Aurora system. These include (1) response times — output tuples should be produced in a timely fashion; as otherwise QoS will degrade as delays get longer); (2) tuple drops — if tuples are dropped to shed load, then the QoS of the affected outputs will deteriorate; and (3) values produced — QoS clearly depends on whether important values are being produced or not. Figure 4 illustrates these three QoS graph types.

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 as part of the system catalog. At run-time this data structure is read into virtual memory. The Aurora run-time architecture is shown in Figure 5. The heart of the system is the scheduler that determines which box (i.e., operator) to run. The scheduler also determines how many input tuples of a box to process

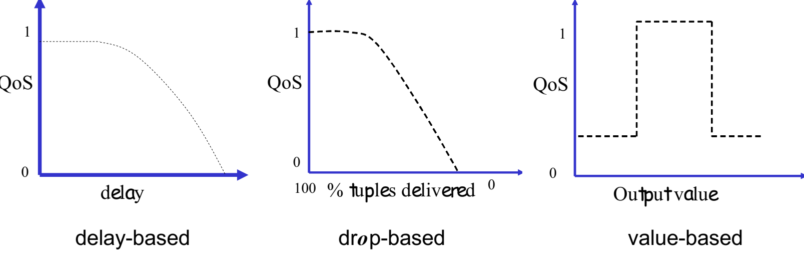

Fig. 4. QoS Graph Types

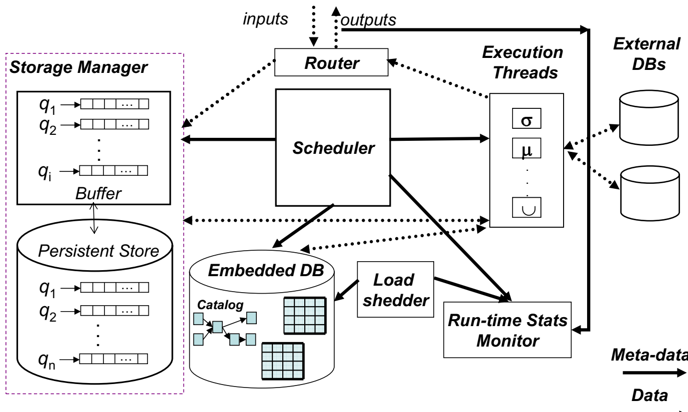

Fig. 5. Aurora Run-Time Architecture

and how far to "push" the tuples toward the output. Aurora operators can store and access data from embedded in-memory databases as well as from external databases. Aurora also has a Storage Manager that is used to buffer queues when main memory runs out. This is particularly important for queues at connection points since they can grow quite long.

The Run-Time Stats Monitor continuously monitors the QoS of output tuples. This information is important since it drives the Scheduler in its decision-making, and it also informs the Load Shedder when and where it is appropriate to discard tuples in order to shed load. Load shedding is only one of the techniques employed by Aurora to improve the QoS delivered to applications. When load shedding is not working, Aurora will try to re-optimize the network using standard query optimization techniques (such as those that rely on operator commutativities). This tactic requires a more global view of the network and thus is used more sparingly. The final tactic is to retune the scheduler by gathering new statistics or switching scheduler disciplines. 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. More detailed information on these various topics can be obtained from the referenced papers [4, 7, 8, 18].

3 Aurora Case Studies

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

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 raise 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 s and they are called "fast updates". Let us also assume that there are 4000 securities that update at least once every 60 s 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 s, and 5 s 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 found in each exchange and the number of low alarms 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 6 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 Fig. 6 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

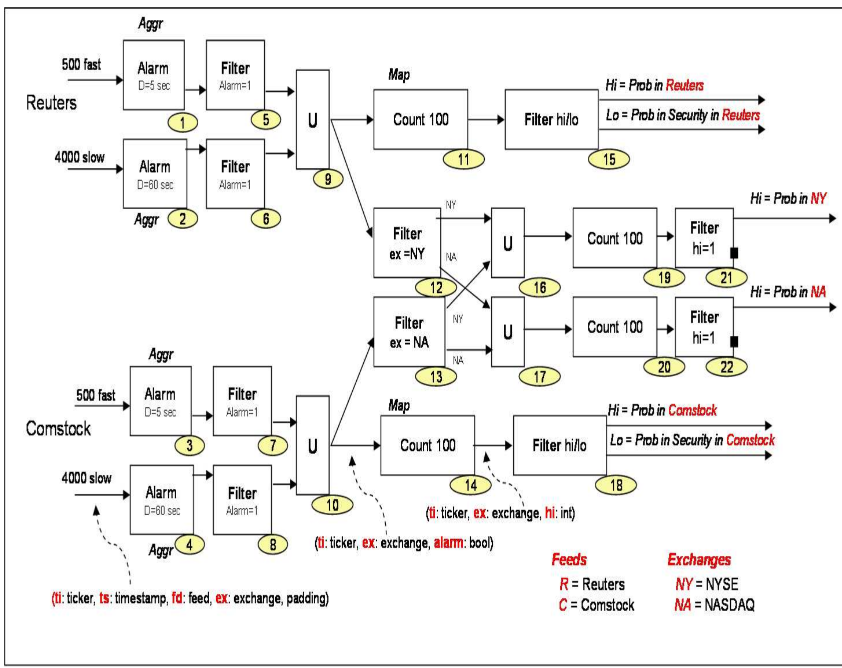

Fig. 6. Aurora query network for the alarm correlation application

Aggregate box groups input tuples by common value of one or more of their attributes, thus effectively creating a substream 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 substream, 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 1 when the second tuple in the pair is delayed (call this an *Alarm tuple*) and a 0 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 s in this case). In other words, the operator will produce one alarm each time a new tuple fails to arrive within 5 s, 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 pseudocode) 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 [3, 5]. 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 microsensor 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 s. Tolls must be issued on a per-segment basis automatically, based on statistics gathered over the previous 5 min 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 five 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 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 five queries within the specified latency bounds.

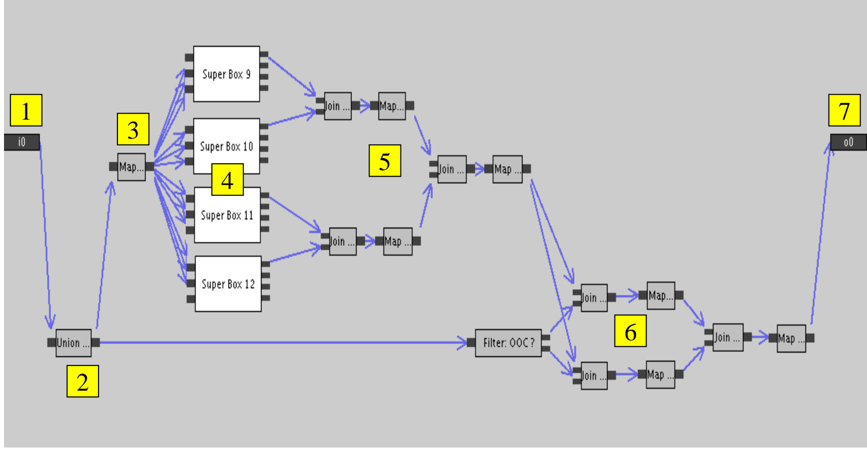

Fig. 7. Aurora query network for the environmental contamination detection applications (GUI snapshot)

3.3 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 Fig. 7 (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 subnetworks 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-h 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 evaluating multiple functions at the same time on a single window [such as `DELTA(water-pH)` and `DELTA(watertemp)`], 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.

3.4 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 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 including incremental scalability over multiple nodes, composition of stream feeds across multiple participants, and high availability and load sharing through resource multiplexing.

The development of Medusa prompted two important changes to the Aurora processing engine. First, it became apparent that it would be useful to

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.

offer Aurora not only as a stand-alone system but also as a library that could easily be integrated within a larger system. Second, we felt the need for an 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.

4 Experience and Lessons Learned

4.1 Support for historical data

From our work on a variety of streaming applications, it became apparent that each application required maintaining and accessing a collection of historical data. For example, the Linear Road benchmark, which represents a realistic application, required maintaining 10 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 take three different forms. These forms differ by their *update patterns* – the means by which incoming stream data are 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 be 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 min (`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 includes 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 includes variables that get instantiated with the values contained in that tuple.

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 up-

dates 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 subexpressions 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.

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 arrives 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 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 are 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 arrive later than they 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, for example, 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 addressing 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* was needed to script Aurora networks and metadata needed 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 node 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 and 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 (Sect. 3.4) 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

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 into 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.

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" that 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 states 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.

These microbenchmarks 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 nontrivial operations such as joining and aggregation. Message-passing overhead, however, can be a significant time sink in streaming databases (as it was in earlier versions of Aurora). Microbenchmarking was very useful in eliminating performance bottlenecks in Aurora's message-passing infrastructure. This infrastructure is now fast enough in Aurora that nontrivial box operations are the only noticeable bottleneck, i.e., CPU time is overwhelmingly devoted to useful work and not simply to shuffling around tuples.

5 Ongoing Work: The Borealis Distributed SPE

This section presents the initial ideas that we have started to explore in the context of the Borealis distributed SPE, which is a follow-on to Aurora. The rest of the section will provide an overview of the new challenges that Borealis will address. More details on these challenges as well as a preliminary design of Borealis can be found in [2].

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 their processing window or 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 extends 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 runtime 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 runtime. 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 online 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). 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 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 have 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 become 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, sizewise 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, multiresource/multimetric optimization challenge that is significantly different than the optimization problems explored in the past.

Our current thinking is that Borealis will rely on a hierarchical, distributed optimizer that runs at different time granularities [2].

5.4 High availability

Another part of the Borealis vision involves addressing recovery and high-availability issues. High availability demands that node failure be 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.

5.5 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 runtime system, and much of the logic for boxes. Similarly, we are borrowing basic networking and distribution logic from Medusa. With this starting point, we hope to have a fully-functional prototype within a year.

Acknowledgement. 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 past members of the Aurora, Medusa, and Borealis projects for their valuable contributions.

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, Y. Ahmad, H. Balakrishnan, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, J. Janotti, W. Lindner, S. Madden, A. Rasin, M. Stonebraker, N. Tatbul, Y. Xing, and S. Zdonik. The Design of the Borealis Stream Processing Engine. Technical Report CS-04-08, Department of Computer Science, Brown University, February 2004.
3. 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.

4. 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.
5. 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).
6. M. Balazinska, H. Balakrishnan, and M. Stonebraker. Contract-Based Load Management in Federated Distributed Systems. In *NSDI Symposium*, March 2004.
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/`.