Monitoring Streams-A New Class of DBMS Applications

Don Carney, Ugur Cetintemel, Mitch Cherniack,
Sangdon Lee,Greg Seidman, Michael Stonebraker,
Nesime Tatbul, Stan Zdonik

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-02-01
February 2002

Monitoring Streams – A New Class of DBMS Applications

Don Carney
Brown University
dpc@cs.brown.edu

Ugur Cetintemel
Brown University
ugur@cs.brown.edu

Mitch Cherniack
Brandeis University
mfc@cs.brandeis.edu

Sangdon Lee
Brown University
sdlee@cs.brown.edu

Greg Seidman
Brown University
gss@cs.brown.edu

Michael Stonebraker
M.I.T.
stonebraker@lcs.mit.edu

Nesime Tatbul
Brown University
tatbul@cs.brown.edu

Stan Zdonik
Brown University
sbz@cs.brown.edu

Abstract

This paper introduces monitoring applications, which we will show differ substantially from conventional business data processing. The fact that a software system must process and react to continual inputs from many sources (*e.g., sensors*) *rather than from human operators requires one to rethink the fundamental architecture of a DBMS for this application area. In this paper, we present the architecture of a new DBMS that is currently under construction at Brown, Brandeis, and M.I.T. We describe the basic system architecture, a stream-oriented set of operators, optimization tactics, support for real-time operation, and the design-time environment.*

1 Introduction

Traditional DBMSs have been oriented toward business data processing, and consequently are designed to address the needs of these kinds of applications. First, they have assumed that the DBMS is a passive repository storing a large collection of data elements and that humans initiate queries and transactions to this repository. We call this a *Human-Active*, *DBMS-Passive* (*HADP*) model. Second, they have assumed that the current state of the data is the only thing that is important. Hence, current values of data elements are easy to obtain, while previous values can only be found torturously by decoding the DBMS log. The third assumption is that triggers and alerters are second-class citizens. These constructs have been added as an after thought to current systems, and none have an implementation that scales to a large number of triggers. In addition, there is typically little or no support for querying a "trigger base." Fourth, DBMSs assume that data elements have precise values (e.g., employee salaries), and have little or no support for data elements that are imprecise or out-of-date. Lastly, DBMSs assume that applications require no real-time services.

There is a substantial class of applications where all five assumptions are problematic. Monitoring applications are applications that monitor continuous streams of data. This class of applications includes military applications that monitor readings from sensors worn by soldiers (e.g., blood pressure, heart rate, position), financial analysis applications that monitor streams of stock data reported from various stock exchanges, and tracking applications that monitor the locations of large numbers of objects for which they are responsible (e.g., audio-visual departments that must monitor the location of borrowed equipment). Because of the high volume of data monitored, and querying requirements for these applications, monitoring applications would benefit from DBMS support. Existing DBMS systems, however, are ill suited for such applications since they target business applications.

First, monitoring applications get their data from external sources (e.g., sensors) rather than from humans issuing transactions. The role of the DBMS in this context is to alert humans when abnormal activity is detected. This is a *DBMS-Active*, *Human-Passive* (*DAHP*) model.

Second, monitoring applications require data management extending over the entire history of values reported in a stream, and not just over the most recently reported values. Consider a monitoring application that tracks the location of items of interest, such as overhead transparency projectors and laptop computers, using electronic property stickers attached to the objects. Data is generated by ceiling-mounted sensors inside a building and the GPS system in the open air. If a reserved overhead projector is not in its proper location, then one might want to know the geographic position of the missing projector. In this case, the last value of the monitored object is required. However, an administrator might also want to know the duty cycle of the projector, thereby requiring access to the entire historical time series.

Third, most monitoring applications are trigger-oriented. If one is monitoring a chemical plant, then one wants to alert an operator if a sensor value gets too high or if another sensor value has recorded a value out of range more than twice in the last 24 hours. Similarly, one might want to notify the campus police if an overhead projector is noticed to be leaving a building. Every application could potentially monitor multiple streams of data, requesting alerts if complicated conditions are met. Thus, the scale of trigger processing required in this environment far exceeds that found in traditional DBMS applications.

Fourth, stream data is often lost, stale, or imprecise. An object being monitored may move out of range of a sensor system, thereby resulting in lost data. The most recent report on the location of the object becomes more and more inaccurate over time. Moreover, sensors that record bodily functions (such as heartbeat) are quite imprecise, and have margins of error that are significant in size.

Lastly, many monitoring applications have real-time requirements. Applications that monitor mobile sensors (e.g., military applications monitoring soldier locations) often have a low tolerance for stale data, making these applications effectively real time. The added stress on a DBMS that must serve real-time applications makes it imperative that the DBMS employ intelligent and graceful degradation strategies (i.e., *load shedding*) during periods of high load.

Monitoring applications are very difficult to implement in traditional DBMSs. First, the basic computation model is wrong: DBMSs have a HADP model while monitoring applications often require a DAHP model. In addition, to store time series information one has only two choices. First, he can encode the time series as current data in normal tables. In this case, assembling the historical time series is very expensive because the required data is spread over many records, thereby dramatically slowing performance. Alternately, he can encode time series information in BLOBs to achieve physical locality, but at the expense of making queries to individual values in the time series very difficult. The only system that we are aware of that tries to do something more intelligent with time series data is the Informix Universal Server (now owned by IBM), which implemented a time-series data type and associated methods that speed retrieval of values in a time series [1].

If a monitoring application had very large number of triggers or alerters, then current DBMSs would fail because they do not scale past a few triggers per table. The only alternative is to encode triggers in some middleware

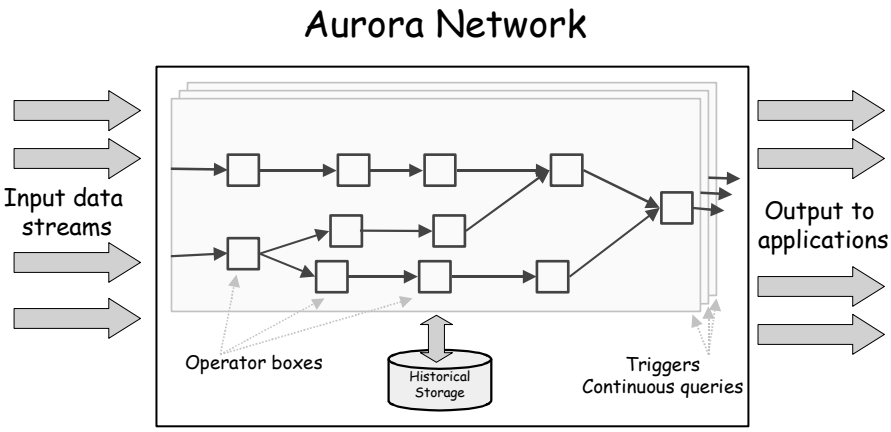

Figure 1: Aurora system model

application. Using this implementation, triggers cannot be queried because they are outside the DBMS. Moreover, performance is typically poor because middleware must poll for data values that triggers and alerters depend on.

Lastly, no DBMS that we are aware of has built-in facilities for imprecise or stale data. The same comment applies to real-time capabilities. Again, the user must build custom code into his application.

For these reasons, monitoring applications are difficult to implement using traditional DBMS technology. To do better, all the basic mechanisms in current DBMSs must be rethought. In this paper, we describe a prototype system, *Aurora*, which is designed to better support monitoring applications.

Monitoring applications are applications for which streams of information, triggers, imprecise data, and real-time requirements are prevalent. We expect that there will be a large class of such applications. For example, we expect the class of monitoring applications for physical facilities (e.g., monitoring unusual events at nuclear power plants) to grow in response to growing needs for security). In addition, as GPS-style devices are attached to a broader and broader class of objects, monitoring applications will expand in scope. Currently such monitoring is expensive and is restricted to costly items like automobiles (e.g., Lojack technology). In the future, it will be available for most objects whose position is of interest.

In Section 2, we begin by describing the basic Aurora architecture and fundamental building blocks. Section 3 continues with the design of the run-time system for Aurora. In Section 4, we show why traditional query optimization fails in our environment, and present our alternate strategies for optimizing Aurora applications. Section 5 shows how Aurora deals with the needs of real-time applications. Section 6 follows with a discussion of the notion of Aurora transactions. In Section 7, we discuss the myriad of related work that has preceded our effort. Finally, we conclude in Section 8.

2 Aurora system structure

Aurora data is assumed to come from a variety of data sources. Sometimes such data sources are computer programs that generate values at regular or irregular intervals. In other applications, values are provided directly by hardware *sensors*. We will use the term *data source* for either case. In addition, a data *stream* is the term we will use for the

collection of data values that are presented by a data source. Each data source is assumed to have a unique source identifier and Aurora timestamps every incoming message to monitor the quality of service being provided.

The basic job of Aurora is to process incoming streams in the way defined by an *application administrator*. Aurora is fundamentally a data-flow system and uses the popular *boxes and arrows* paradigm found in most process flow and workflow systems. Hence, messages flow through a directed graph of processing operations (i.e., *boxes*). Ultimately, output streams are presented to *applications*, which must be programmed to deal with the asynchronous messages in an output stream. Aurora can also maintain historical storage, primarily in order to support ad-hoc queries (which we discuss later in this section). Figure 1 illustrates this basic system structure.

Applications express their stream processing requirements by way of triggers[1]. Aurora contains built-in support for seven primitive operations for expressing its stream processing requirements. The *filter* box applies a predicate to the tuple contained in each incoming stream element, referred to as a *message*, passing only the ones that satisfy the predicate. *Merge* combines two streams of data into a single stream. *Resample* is a powerful operation that predicts additional values that are not originally contained in a stream. This operation can be used to create new values in the stream by interpolating between existing values. In addition, Aurora provides a *drop* box that removes messages from the stream, and is used primarily to shed load when Aurora is not providing reasonable service (see Section 5.3). *Join* is a windowed version of the standard relational join. Join pairs elements in separate streams whose *distance* (e.g., difference in time, relative position, etc.) is within a specified bound. This bound is considered the size of the window for the join. For example, the window might be 30 minutes if one wanted to pair all securities that had the same price within a half hour of each other. In environments where data can be stale or time imprecise, windowed operations are a necessity. The last box is called *map* and it supports the application of a function to every element of a data stream. Some functions just transform individual items in the stream to other items, while others, such as moving average, apply a function across a window of values in a stream. Hence, map has both a windowed and non-windowed version.

A precise definition for each of these operators is contained in Appendix 1. The reader can notice that both versions of map provide for user-defined functions. In this way, significant extensibility can be provided. At the current time, we do not plan for user-defined boxes, because our optimization strategies, to be presented in Section 4, rely on fairly complex knowledge about the semantics of our box types. Extensibility in this area would be tricky to support. Notice that three of the seven Aurora primitive operations (i.e., resample, join, and windowed map) process windows of data elements. To support such operations, Aurora must buffer previous values of a data stream. These windows define the historical storage requirements for stream data. Aurora is aware of the processing that must be done on this buffer database, and can automatically choose indexes, where appropriate. In general, this database is relatively small, and we expect to keep it in main memory to provide good response time. However, the reader should notice that the number of messages contained in a window might be variable. This complicates buffer allocation issues, which will be studied elsewhere.

[1] In this paper, we will ignore the trigger actions taken by applications once they receive data. Instead, we focus on the way that applications express the *query portion* of the trigger that specifies the ways that stream data must be processed before its delivery to the application.

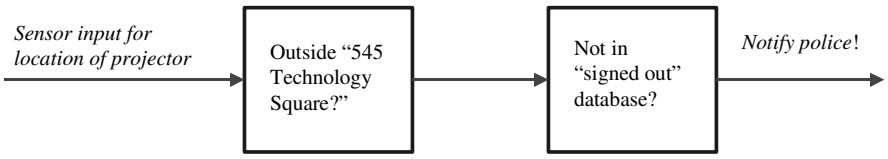

Figure 2: A simple Aurora application

The job of the application administrator is to construct a directed graph of these operations to accomplish a specific monitoring task. Figure 2 shows a simple Aurora application that tracks the location of an overhead transparency projector and notifies the police if it is in an improper location. It processes an input stream detailing the location of the projector. First, it filters the stream to only those data values that are outside the building where the projector is located (in this case "545 Technology Square"). The stream is further filtered to only contain a value if the projector is not checked out. If both conditions are true, Aurora sends a message to an external application that will notify the campus police. The processing required of Aurora is to *push* each input from a data source through the network and generate output messages in a timely fashion.

Every Aurora network is required to be free from loops. A graph with cycles has unpredictable behavior, and we wish to avoid this situation. Of course, it is possible for an application that receives a message to update a data source control that in turn modifies the future behavior of an input stream. Hence, there may be dependencies between output streams and input streams; however, these dependencies involve an intervening application program that is external to Aurora. As a result, Aurora is unaware of this dependency.

Furthermore, there are no condition boxes in an Aurora network. Instead, an application would have to include two filter boxes providing both the condition and the converse of the condition. At some inconvenience to an application, this results in a simpler system structure. Additionally, there is no explicit *split* box; instead, the application administrator can connect the output of one box to the input of several others. This implements an implicit split operation. On the other hand, there is an explicit Aurora merge operation, whereby two streams can be put together. If, additionally, one message must be delayed for the arrival of a second one, then a resample box can be inserted in the Aurora network to accomplish this effect.

One can view an Aurora network as a large collection of triggers. Each path from a sensor input to an output can be viewed as computing the *condition* part of a complex trigger. An output message is delivered to an application, which can take any appropriate action. Hence, an Aurora network is a large collection of (possibly) complex triggers. It is certainly possible for more than one of these streams to have a value at the same time. For example, if both high temperature and high humidity are true, then two of the above example streams will have values. In this case, the objective of Aurora is to present both streams to the application in a timely fashion. The decision on which stream to act on first (i.e., prioritization) rests with the application and is outside the scope of Aurora.

Furthermore, Aurora can also easily implement so-called continuous queries. Any path through the network can correspond to the processing normally involved in such a query. If output messages are directed to an application that renders them onto the screen, then a user watching the screen will see the result of a continuous query. Of course, Aurora processing of continuous queries is radically different from what is required in conventional systems (e.g., [32]). Streams from one or more data sources are pushed by Aurora to a rendering application. In contrast, a

conventional system records many user updates to a DBMS, and complex logic is required to figure out when new records qualify for the predicate of a continuous query.

Another issue is Aurora support for ad-hoc queries; i.e., ones not designed by the application administrator and hence not present in the initial Aurora network. These are accomplished by an end-user (or his agent) adding additional boxes to a running Aurora network. Such boxes are different from normal Aurora boxes because they are removed from the network when the application connected to them terminates. As such, they are useful in providing a snapshot of Aurora behavior.

If the user requests historical information in his ad-hoc query, (e.g., the duty cycle of a slide projector over the last month) then Aurora will only be able to provide an answer if the appropriate information has been retained. To this end, the application administrator can request Aurora to keep more information than would normally be required to provide semantically correct network processing. One aspect of the design-time environment is a facility to specify such larger windows. Ultimately, a window can be made infinite, and Aurora asked to "save everything."

The Aurora user interface cannot be covered in detail because of space limitations. Here, we mention only a few salient features. To facilitate designing large networks, Aurora will support a hierarchical collection of groups of boxes. A designer can begin near the top of the hierarchy where only a few *superboxes* are visible on the screen. A *zoom* capability is provided to allow him to move into specific portions of the network, by replacing a group with its constituent boxes and groups. In this way, a browsing capability is provided for the Aurora diagram. Groups also provide a convenient abstraction for alternate implementations of functionality. Often an application has a "fast and sloppy" implementation as well as a "slow and careful" one for a portion of diagram processing. One may want to switch between implementations, depending on system load. Our concept of groups will support this construct.

Boxes and groups have a tag, an argument list, a description of the functionality and ultimately a manual page. By querying these attributes a user can "teleport" to specific places in an Aurora network. Additionally, a user can place *bookmarks* in a network to allow him to return to places of interest.

These capabilities give an Aurora user a mechanism to query the Aurora diagram, thereby satisfying one of the design goals in the introduction, namely query facilities for the trigger base.

The user interface also allows monitors for arcs in the network to facilitate debugging, as well as facilities for "single stepping" through a sequence of Aurora boxes. We plan a graphical performance monitor, as well as more sophisticated query capabilities.

3 The Aurora run-time environment

The basic purpose of the execution engine is to process data flows through a potentially large workflow diagram. Although the size of the network depends on the application, our discussion with potential Aurora users indicates requirements between 10^5 and 10^7 data sources and a like number of boxes. Hence, allocating a thread for each processing box in a network will not scale to large Aurora networks. Instead, we propose an alternate approach. We plan to compile the network into a main memory data structure that can be thought of as the following tables:

- *Boxes* (box-id, box-definition, pointer-to-box-state)
- *Connections* (box-id, successor-box-id)

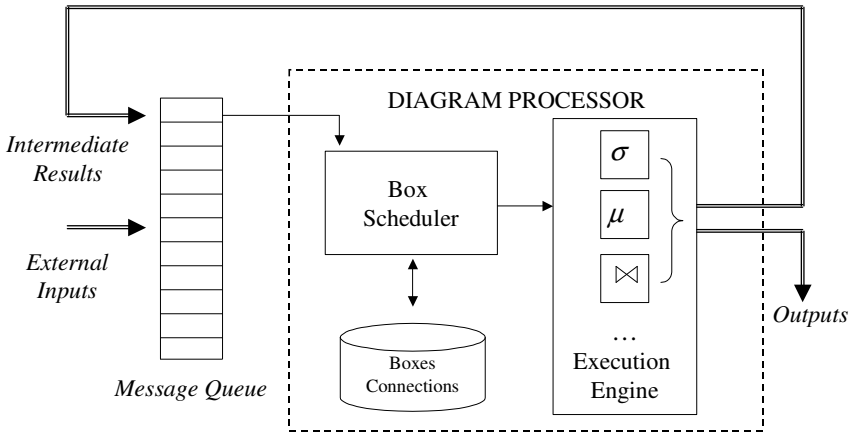

Figure 3: Aurora run-time architecture

The Boxes table contains the definition and state for each box. The state includes parameters for the box and any historical messages required for correct window processing. Whenever two boxes share overlapping state (e.g., one requires a window for elements that arrive in 10-minute intervals, while the second requires a window for elements that arrive in 8-minute intervals), the compiler will notice the shared state and allocate the larger state just once. Multiple boxes can then share the required data structure. The Connections table contains the arcs in the network.

Further, we are planning to implement the run-time architecture of Figure 3. Here, inputs from data sources are marked with an arc-id and a timestamp, and added to a (potentially large) queue. A multi-threaded diagram processor consists of a scheduler and an execution engine. The scheduler picks an element from the queue, ascertains what processing is required and passes the queue message, together with a pointer to the box-description and state to the correct box processor (which is also multi-threaded). There is one box processor for each kind of Aurora operation (e.g., join, map, etc.). The box processor performs the appropriate operation and then puts an output message on the queue. The diagram processor then ascertains the next processing step and the cycle repeats. If the message must be sent to an external application as an output, then the diagram processor performs the appropriate action. We expect to achieve high performance on large computer systems by allocating multiple threads for the diagram interpreter and each box processor. Another motivation for implementing our own scheduler (instead of using the built-in thread scheduler) is that our scheduler will be able to react to the semantic character of the network, allocating resources in Aurora-specific ways.

4 Aurora optimization

In traditional relational query optimization, one of the primary objectives is to minimize the number of iterations over large data sets. Stream-oriented operators that constitute the Aurora network on the other hand, are designed to operate in a data flow mode in which data elements are processed as they appear on the input. Although the amount of computation required by an operator to process a new element is usually quite small, we expect to have a very large number of boxes in the Aurora workflow diagram that represents all of the triggers and ad-hoc queries. Furthermore, high data rates add another dimension to the problem. In this section, we present our strategies to

construct an optimized Aurora network based on static information such as estimated cost of the boxes and selectivity statistics. Real-time optimizations, which mostly concern the data rates, are presented in Section 5.

Combining all the boxes into a massive query and then applying conventional query optimization is not a workable approach for the Aurora system; however, some conventional principles still apply. As in multiple-query optimization [28], one way to go about optimizing multiple triggers is to try to globally optimize them, which is shown to be NP-complete [29]. Instead, we focus on the following collection of alternative tactics to come up with locally optimal triggers which can then be compiled into a single network of boxes (also using common subexpression elimination techniques where possible):

1. Inserting projections
2. Combining boxes
3. Reordering boxes
4. R-tree processing
5. Trading computation for storage
6. Special techniques for ad-hoc queries

The first four tactics deal with rearrangement and efficient processing of boxes, while the fifth lowers memory consumption when main memory buffering is an issue. The final optimization addresses the addition of ad-hoc queries to an Aurora diagram. In the rest of this section, we explore the above tactics in detail.

4.1 Inserting projections

It is unlikely that the application administrator will have inserted map operators to project out all unneeded attributes. Examination of an Aurora network allows us to insert/move such map operations to the earliest possible points in the network, thereby shrinking the size of the messages that must be subsequently processed. Note that this kind of an optimization requires that the system be provided with function/operator signatures that describe the attributes that are used and produced by the operators.

4.2 Combining boxes

As a next step, Aurora diagrams will be processed to combine boxes where possible. A pair-wise examination of the operators suggests that, in general, map and filter can be combined with almost all of the operators whereas windowed or binary operators cannot.

It is desirable to combine two boxes into a single box when this leads to some cost reduction. As an example, a map operator that only projects out attributes can be combined easily with any adjacent operator, thereby saving the box execution overhead for a very cheap operator. In addition, two filtering operations can be combined into a single, more complex filter that can be more efficiently executed than the two boxes it replaces. Not only is the overhead of a second box activation avoided, but also standard relational optimization on one-table predicates can be applied in the larger box. Likewise, two map operations can be combined. Although additional operations can be combined without affecting the semantics of the outcome (such as merge and filter), there may not be a noticeable gain. In general, combining boxes at least saves the box execution overhead, and reduces the total number of boxes, leading to a simpler diagram. The next step is to change the order of box processing where advantageous and possible.

4.3 Reordering boxes

Reordering the operations in a conventional relational DBMS to an equivalent but more efficient form is a common technique in query optimization. For example, filter operations can sometimes be pushed down the query tree through joins. In Aurora, we can apply the same technique when two operations are commutative.

	filter	map	w. map	merge	resample	join	drop
filter	*yes*						
map	*maybe*	*maybe*					
w. map	*no*	*maybe*	*maybe*				
merge	*yes*	*yes*	*no*	*yes*			
resample	*maybe*	*maybe*	*no*	*no*	*maybe*		
join	*maybe*	*maybe*	*no*	*no*	*no*	*maybe*	
drop	*yes*	*yes*	*no*	*yes*	*no*	*no*	*yes*

Table 1: Operator commutativity table

We present the commutativity of each pair of primitive boxes in Aurora in Table 1. We observe that a few operators always commute, while many commute under certain circumstances and a few never commute. The operations that conditionally commute usually depend on the size of the window. As one would expect, the operators with window size one are more well behaved than the ones with larger windows are. In some cases, the format of the input can also play a role. In Appendix 2, we examine the commutativity of our operations in detail.

To decide when to interchange two commutative operators, we make use of the following performance model. Each Aurora box, b, has a *cost*, $c(b)$, defined as the expected execution time for b to process one input message and produce one output message. Additionally, each box has a *selectivity*, $s(b)$, which is the expected number of output messages per input message. For instance, filter will always have a selectivity less than or equal to one, whereas resample and join may have a selectivity greater than one.

Consider two boxes, b_i and b_j, with b_j following b_i. In this case, for each input message for b_i, we can compute the amount of processing as $c(b_i) + c(b_j) \times s(b_i)$. Reversing the operators gives a like calculation. Hence, we can compute the condition used to decide whether the boxes should be switched as:

$$\frac{1 - s(b_j)}{c(b_j)} > \frac{1 - s(b_i)}{c(b_i)}$$

The above condition says that if the ratio of the rate reduction caused by b_j to its cost is greater than the corresponding ratio for b_i, then b_j should be executed before b_i. It is straightforward to generalize the above calculation to deal with cases that involve fan-in or fan-out situations. Moreover, it is easy to see that we can obtain an optimal ordering by sorting all the boxes according to their corresponding ratios in decreasing order (i.e., the box with the highest rate reduction-to-cost ratio should be the one closest to data sources). We can use this result to come up with a heuristic algorithm that iteratively reorders boxes (to the extent allowed by their commutativity properties) until no more reorderings are possible.

4.4 R-tree processing

Consider the situation that a box output is split a large number of times and sent to a large collection of subsequent filter boxes. In a chemical plant, for example, there might be many different actions to take, depending on the value of an input sensor. In this case, Aurora can assemble all the filters into a single superbox, supported in the following way.

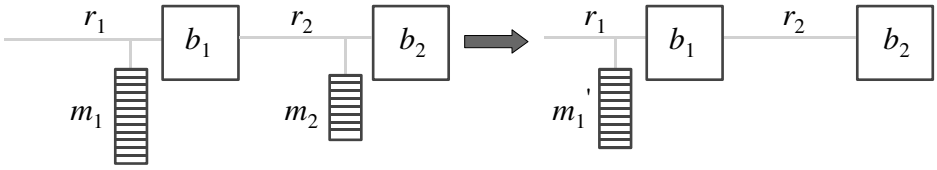

Figure 4: Trading computation for storage

All the predicates in all the boxes are examined and an R-tree [15] is implemented on all the attributes that appear in some predicate. If there is too much predicate diversity, then the *outliers* are tossed out of the superbox to be processed in the normal fashion. Hence, each predicate in the superbox is a rectangle in this R-tree. If there are k filters in the superbox, then the R-tree has k rectangles.

Whenever the superbox receives an input, it goes through the motion of inserting the corresponding point in this R-tree in order to find which rectangles cover this point. It then activates only those boxes corresponding to the correct rectangles.

This concept can be generalized somewhat. For example, if a box requires two values in a certain range within a certain time period, then the rectangle can include time as a dimension. R-tree processing must then actually insert the points into the R-tree and then activate the box only when it counts the correct number of points inside the appropriate rectangle.

4.5 Trading computation for storage

As mentioned earlier, windowed operations require Aurora to maintain a database of historical values of the recent past. This database may be small enough that it fits or nearly fits in main memory. When the size of the database is larger than the amount of available main memory, the algorithms in this section should be used. These algorithms compress the required amount of storage, at the possible expense of extra processing. The basic tactic is to move one or more data sets in this database from their current location between two operators, to an earlier position in the diagram. This will force Aurora to reprocess intervening boxes, thereby adding to the processing load. On the other hand, the total amount of storage required is potentially reduced.

Due to space considerations, we restrict our discussion to two scenarios with the greatest potential storage gain. Our approach, however, is general and can be applied to other cases as well. Let r_i represent the input message rate for box b_i. Further, let $w(m_i)$ represent the window size and $t(m_i)$ represent the width of a message in the database. The total window size for b_i is then equal to $t(b_i) \times w(b_i)$. Furthermore, let m_i' be the resulting database after storage has been moved forward in the diagram.

Our first scenario involves two boxes that are serially connected. We first demonstrate our approach using the simple example illustrated in Figure 4. Here, we observe two boxes, b_1 and b_2, that implement windowed map operations performing an aggregate function (e.g., *average*). Assume that r_1 equals r_2 (i.e., selectivity of b_1 is 1). If b_1 has a window, m_1, with the size of four, and b_2 has a window, m_2, with the size of three, then the total size of the composite windows is seven. However, m_2 can be moved forward and combined with m_1 into a single window, m_1'. This move reduces the total amount of storage required from seven messages to six. Whenever b_2 requires a message

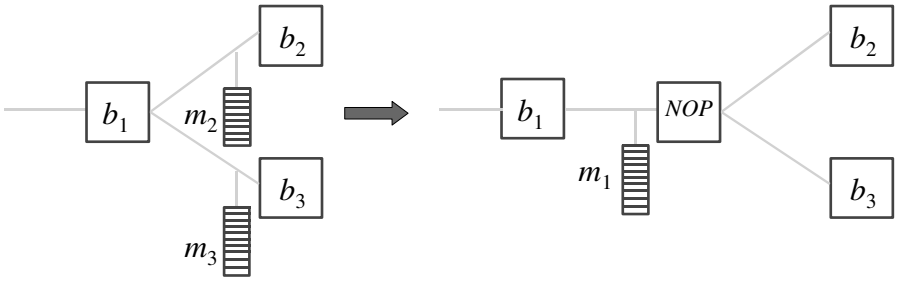

Figure 5: Fan-out case

from b_1, b_1 must *re*create the message using the messages stored in m_1'. This requires b_1 to be called two additional times. The result of combining b_2's window with b_1's is a savings of one message slot but an increase in computation of $2 \times c(b_1)$, where $c(b_1)$ represents the computation cost of b_1 as described earlier.

We can generalize the above discussion in a straightforward manner, and compute the pre- and post-move computational costs as:

$$c(b_1) + c(b_2) \text{ and } w(m_2) \times c(b_1) + c(b_2)$$

Similarly, we can also compute the pre- and post-move storage requirements as:

$$w(m_1) \times t(m_1) + w(m_2) \times t(m_2), \text{ and}$$
$$w(m_1') \times t(m_1') = (r_1 / r_2) \times (w(m_2) + w(m_1) - 1) \times t(m_1)$$

Using these equations, we can easily compute the storage gained and extra computation required. If the ratio of these two values is above a certain threshold *T*, then we decide to move the storage.

The second case we discuss involves a fan-out situation, where the output of one operation feeds data to multiple subsequent windowed operations. In this case, which is illustrated in Figure 5, Aurora will always add a *no-op* operation between the operations and move storage in front of no-op. This will result in considerable savings from having a single shared storage rather than multiple ones at no additional computational cost. The new window size will be equal to that of the operation with the largest window size. Unlike the previous case, where there was a trade off between storage and computation, this case is *always* advantageous.

We plan to apply the above techniques to an Aurora diagram and move storage iteratively until no further moves are possible. Although this will not always produce an optimal assignment of memory to an Aurora diagram, it is an effective heuristic that is computationally feasible.

4.6 Ad-hoc query optimization

Ad-hoc queries, which usually refer to historical values, are temporarily inserted into an Aurora network after the network has been constructed, optimized, and already started running. When an ad-hoc query is inserted into a running Aurora network, there are two issues to consider. First, if the application administrator has specified historical storage of earlier messages in the network, then Aurora must generate the correct answer to the historical query posed. Logically, all relevant previously-stored upstream messages must be processed by the newly added boxes. Second, the added boxes must then continue processing live Aurora messages until they get deleted. The second issue can be dealt with by the optimizations discussed in Sections 4.1– 4.5. Hence, we now discuss how we can process the historical part of an ad-hoc query efficiently.

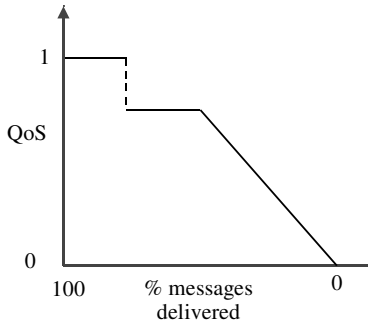

Figure 6: A message-based QoS graph

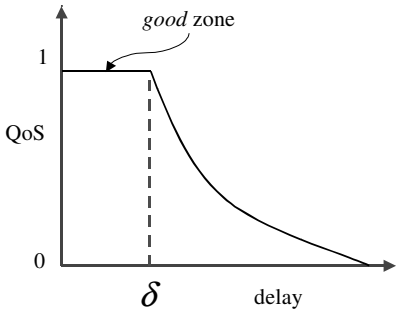

Figure 7: A delay-based QoS graph

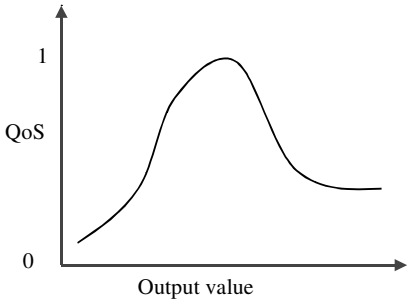

Figure 8: A value-based QoS graph

There are two possible solution tactics. First, the stored messages can be pushed through the extra network, after the optimizations of Sections 4.1– 4.5 have been applied. Since Aurora knows the cost and selectivity of each box and the number of historical messages stored, the overall cost of this tactic can be easily computed. Alternately, Aurora can combine all the boxes into a single query, optimize the query using standard relational techniques, and then run the query against the stored historical dataset(s). This will implement a *pull* strategy, whereby only valid messages are pulled from the dataset(s) and sent to the output application. Using conventional techniques, the expected cost of this pull tactic can also be computed.

The plan for Aurora is to use the cheaper of these two techniques to solve the historical portion of the query. Of course, there is also a synchronization issue because the result of the historical query must be integrated with the continuing answer to be subsequently generated.

5 Real-time operation

This section is concerned with maintaining quality of service in a real-time environment. In Section 5.1, we begin with the auxiliary data structures that are required by the algorithms presented in this section. In Section 5.2, we first present a static analysis to determine if Aurora system resources are adequate to handle the expected system load. We then turn to investigating the dynamic behavior of an Aurora system and discuss the issue of determining whether an Aurora system is overloaded due to long-duration load spikes that result from increased data source activity. In Section 5.3, we present our load shedding algorithms that intelligently drop messages from an Aurora network to ensure that the expected data arrival rates will not overload system resources in steady state and that the best possible quality of service is achieved for Aurora outputs even under unexpected loads. In Section 5.4, we present a summary of Aurora's overall approach to introspection and load shedding.

5.1 Data structures

When the input rates exceed system resource limits, load must be shed to maximize the perceived quality of service (QoS) for the outputs produced by Aurora. QoS, in general, is a multidimensional function of several attributes of an Aurora system. These include:

- *response times*─ output messages should be produced in a timely fashion; as otherwise QoS will degrade as delays get longer;
- *message drops*─if messages are dropped to shed load, then the QoS of the affected outputs will deteriorate;

- *values produced*—QoS clearly depends on whether important values are being produced or not.

Asking the application administrator to specify a multidimensional QoS function seems unworkable. Instead, Aurora relies on a simpler tactic, which is much easier for humans to deal with: for each output stream, we expect the application administrator to give Aurora a two-dimensional QoS graph based on the percentage of messages delivered (as illustrated in Figure 6). In this case, the application administrator indicates that high QoS is achieved when message delivery is near 100% and that QoS degrades as messages are dropped.

Optionally, the application administrator can give Aurora either one of two additional QoS graphs for all outputs in an Aurora system. The first, illustrated in Figure 7, is a delay-based QoS graph. Here, the QoS of the output is maximum if delay is less than the threshold, δ, in the graph. Beyond δ, QoS degrades with additional message delay. The application administrator should present Aurora with the information in Figure 7 for each output, if he is capable of doing so.

Aurora also assumes that all quality of service graphs are normalized, so that we can compare QoS for different outputs quantitatively. Aurora further assumes that the value chosen for δ is *feasible*, i.e., that a properly sized Aurora network will operate with all outputs in the *good* zone to the left of δ. This will require the delay introduced by the total computational cost along the longest path from a data source to this output not to exceed δ. If the application administrator does not present Aurora with feasible QoS graphs, then the algorithms in the subsequent sections will not produce good results.

The second optional QoS graph for outputs is shown in Figure 8. The possible values produced as outputs appear on the horizontal axis, and the QoS graph indicates the importance of each one. This value-based QoS graph captures the fact that some outputs are more important than others. For example, in a plant monitoring application, outputs near a critical region are much more important than ones well away from it. Again, if the application administrator has value-based QoS information, then Aurora will use it to shed load more intelligently than would occur otherwise.

The last item of information required from the application administrator is H, the *headroom* for the application, defined as the percentage of the computing resources that can be used in steady state. The remainder are reserved for the expected ad-hoc queries, which are added dynamically.

In summary, Aurora requires the application administrator to specify the headroom as well as message-based QoS graphs for each output. In addition, the administrator can optionally give Aurora delay-based or value-based QoS graphs for all outputs. In the rest of this section, we discuss how to use these QoS graphs to ensure adequate steady-state performance and handle spikey input data rates by intelligently shedding load.

5.2 Introspection

Aurora employs static and run-time introspection techniques to predict or detect potential overload situations.

5.2.1 Static analysis

The first task for Aurora is to determine if the hardware running the Aurora network is sized correctly. If insufficient computational resources are present to handle the steady state requirements of an Aurora network, then queue lengths will increase without bound and response times will become arbitrarily large.

As described before, each box, b, in an Aurora network has an expected message processing cost, $c(b)$, and a selectivity, $s(b)$. If we also know the expected rate of message production $r(d)$ from each data source d, then we can use the following static analysis to ascertain if Aurora is sized correctly.

From each data source, we begin by examining the immediate downstream boxes: if box b_i is directly downstream from data source d_i, then, for the system to be stable, the throughput of b_i should be at least as large as the input data rate; i.e.,

$$\frac{1}{c(b_i)} \geq r(d_i)$$

Moreover, we can calculate the output data rate from b_i as:

$$min(\frac{1}{c(b_i)}, r(d_i)) \times s(b_i)$$

Proceeding iteratively, we can compute the output data rate and computational requirements for each box in an Aurora network. We can then calculate the minimum aggregate computational resources required per unit time, *min_cap*, for stable steady-state operation. Clearly, the Aurora system with a capacity C cannot handle the expected steady state load if C is smaller than *min_cap*. Furthermore, the response times will assuredly suffer under the expected load of ad-hoc queries if

$$C \times H < min_cap$$

Clearly, this is an undesirable situation and can be corrected by redesigning applications to change their resource requirements, by supplying more resources to increase system capacity, or by load shedding (see Section 5.3).

5.2.2 Dynamic analysis

Even if the system has sufficient resources to execute a given Aurora network under expected conditions, unpredictable and long-duration spikes in input data rates may deteriorate overall system performance to a level that renders the system useless. We now describe two run-time techniques to detect such cases.

Our first technique for detecting an overloaded system relies on the use of delay-based QoS information, if available. Aurora timestamps all messages from data sources as they arrive. Furthermore, all Aurora operators preserve the message timestamps as they produce output messages (if an operator has multiple input messages, then the earlier timestamp is preserved). When Aurora delivers an output message to an application, it checks the corresponding delay-based QoS graph (see Figure 7) for that output to ascertain that the delay is at an acceptable level (i.e., the output is in the *good* zone). Of course, a single errant output is not cause for alarm, so Aurora watches to see if the number of errant outputs exceeds a threshold to decide whether the system is behaving abnormally and corrective action needs to be taken.

If delay-based QoS information is not available to guide Aurora in detecting abnormal operation, then Aurora employs a somewhat cruder technique. Specifically, Aurora watches its internal dispatching queue for evidence of sustained growth of one or more of the logical queues (in front of some box(es)). If a buildup is observed for a period of time exceeding a threshold, then Aurora takes corrective action. Of course, longer queue lengths do not necessarily mean that the delivered QoS is bad or unacceptable. Unless delay-based QoS information is available, however, this is the most reliable piece of information on which to base load shedding decisions.

5.3 Load shedding

When an overload is detected as a result of static or dynamic analysis, Aurora attempts to reduce the volume of Aurora message processing via load shedding. The naïve approach to load shedding involves dropping messages at randomly picked points in the network and in an entirely uncontrolled manner. Such an approach has two potential problems: (1) overall system utility, which is defined by the QoS graphs, might be degraded much more than necessary (or, depending on the type of QoS graph used, might be improved much less than possible); and (2) application semantics might be arbitrarily affected. In order to alleviate these problems, Aurora relies on the QoS graphs to guide its load shedding process. We now describe two load shedding schemes that differ in the way they exploit QoS information.

5.3.1 Load shedding by dropping messages

Our first load shedding approach addresses the former problem mentioned above: it attempts to minimize the degradation (or maximize the improvement) in the overall system QoS (i.e., the QoS values aggregated over all the outputs). This is accomplished by dropping messages on network branches that terminate in *more tolerant* outputs.

If load shedding is triggered as a result of static analysis, then we cannot expect to use delay-based or value-based QoS information to guide load shedding (without assuming the availability of a priori knowledge of the message delays or frequency distribution of values). Therefore, for the static analysis case, we can only use message-based QoS graphs. If load shedding is triggered as a result of dynamic analysis, however, we can use delay-based QoS graphs if they are available.

We employ a greedy algorithm to perform load shedding as follows (without loss of generality, we initially assume that message-based QoS graphs are used to drive the load shedding process). We first identify the output with the *smallest* negative slope for the corresponding QoS graph. We move horizontally along this curve until there is another output whose message-based QoS curve has a smaller negative slope at that point. This horizontal difference gives us an indication of the *output* messages to drop (i.e., the selectivity of the drop box to be inserted) that would result in the minimum decrease in the overall QoS. We then move the corresponding drop box as far upstream as possible until we find a box that affects other outputs (i.e., a *split point*), and place the drop box at this point in the network. Meanwhile, we can calculate the amount of recovered resources or the new delay values that result from this exercise. If the system resources are still not sufficient or response time goals are not met, then we identify another output and repeat the process.

If we used a delay-based QoS graph instead, we would then be interested in identifying the output with the *largest* negative slope (i.e., the output that would yield the maximum increase in overall QoS). The rest of the algorithm would proceed similarly.

Presumably, the application is coded so that it can tolerate missing messages from a data source caused by communication failures or other problems. Hence, this mechanism for shedding load just artificially introduces additional missing messages. Although the semantics of the application are now different, the harm should not be too damaging.

5.3.2 Semantic load shedding by filtering messages

The load shedding scheme described above effectively reduces the amount of Aurora processing by dropping *randomly selected* messages at strategic points in the network. While this approach attempts to minimize the loss in overall system utility, it fails to control the impact of the dropped messages on application semantics. Semantic load shedding addresses this limitation by using value-based QoS information, if available. Recall that this QoS graph provides information about the relative importance of various values for a given output. Specifically, semantic load shedding drops messages in a more controlled way; i.e., it drops less important messages, rather than random ones, using filter boxes (instead of drop boxes).

If value-based QoS information is available, then Aurora can watch each output and build up a histogram containing the frequency with which value ranges have been observed. In addition, Aurora can calculate the expected utility (QoS) of a range of outputs by multiplying the QoS values with the corresponding frequency values for every interval and then summing these values. To shed load, Aurora identifies the output with the *lowest utility interval*; converts this interval to a filter predicate; and then, as before, attempts to propagate the corresponding filter box as far upstream as possible to a split point. This strategy, which we refer to as *backward interval propagation*, admittedly has limited scope because it requires the application of the inverse function for each operator passed upstream (Aurora boxes do not necessarily have inverses). An alternative strategy, *forward interval propagation*, *estimates* a proper filter predicate and propagates it in *downstream* direction to see what results at the output. By trial-and-error, Aurora can converge on a desired filter predicate. Note that a combination of these two strategies can also be utilized. First, Aurora can apply backward propagation until a box, say *B*, whose operator's inverse is difficult to compute. Aurora can then apply forward propagation between the insertion location of the filter box and *B*. This algorithm can be applied iteratively until sufficient load has been shed.

5.4 Summary

Aurora's approach to introspection and load shedding can be outlined as follows. We use static analysis to correctly size an Aurora network. If the system lacks sufficient resources, we use message-based QoS graphs to place drop boxes into an Aurora network in advance of operation. If all outputs have value-based QoS graphs, we can also transform the drop boxes to appropriate filter boxes once the network is operational. This will ensure adequate steady-state operation.

Dynamically, there is also a need to shed load to deal with unpredictably heavy loads. We can use delay-based QoS for each output to identify when load must be shed. If this information is not available, then we must use cruder queue-length metrics. If all outputs have value-based QoS graphs, then we can insert filter nodes into an Aurora network; otherwise, we insert drop nodes using delay-based QoS information.

Of course, Aurora will contain heuristics to delay shedding load until bad behavior has been observed for a while. Likewise, good behavior must also persist before load shedding is reversed. Note that we can reverse load shedding simply by removing the drop (or filter) boxes in the reverse order they were inserted into network.

In summary, we can perform static and run-time analysis to detect (potential) overload situations and perform load shedding to decrease the number of messages Aurora needs to process in those cases. Depending on the

availability of certain QoS information, Aurora can perform either standard load shedding using drop boxes or more controlled semantic load shedding using filter boxes.

6 Transaction processing

It is clear that Aurora can shed load if required to meet responsiveness goals. A crash is really nothing but a massive shedding of load. Hence, recovering from crashes does not necessarily require standard write ahead logging and redo/undo techniques. Instead, Aurora will simply resume processing with as much preserved data as possible. Hence, Aurora develops *partial amnesia* upon crash occurences. If the disk is not intact, then Aurora's amnesia may become total.

Similarly, Aurora needs a semaphore for the top of the message queue. In addition, when Aurora rearranges the elements of the queue, it must implement some sort of locking scheme. The same comment applies when a box updates any state that it shares with one or more other boxes. However, there is no concept of transaction abort or deadlock. As such, Aurora transaction management can be very simple, and will not entail a lot of complexity.

7 Related work

In this section, we summarize previous research related to Aurora. In particular, we discuss efforts on continuous queries, triggers, adaptive query processing, stream processing, materialized views, and approximate query answering.

A special case of Aurora processing is as a continuous query system. Many continuous query systems have focused their interests in performance on the topic of indexing the queries (e.g., [3]). In contrast, Aurora is a much more general effort. However, we plan to examine previous query indexing systems for applicability in the Aurora environment. A system like Niagara [7] is concerned with combining multiple data sources in a wide area setting while we are initially focusing on the construction of a stream server that can process very large numbers of streams. Niagara also focuses on Internet-style data such as XML and other semi-structured sources while our emphasis is on data sources that produce time-series.

As in Aurora, active databases [26, 27] are concerned with monitoring conditions. These conditions can be a result of any arbitrary update on the stored database state. In our setting, updates are append-only, thus requiring different processing strategies for detecting monitored conditions. Triggers evaluate conditions that are either true or false. Our framework is general enough to support generalized queries over stream or the conversion of these queries into monitored conditions. There has also been extensive work on making active databases highly scalable (e.g., [16]). Similar to continuous query research, these efforts have focused on query indexing, while Aurora is constructing a more general system.

Adaptive query processing techniques (e.g., [4, 17, 21, 33]) address efficient query execution in unpredictable and dynamic environments (e.g., Internet data sources) by revising the query execution plan as the characteristics of incoming data changes. The adaptation approach we described in this paper, load shedding, is radically different from previous adaptive techniques in that it involves dropping heuristically selected data from an incoming data stream, albeit in a controlled manner. It is possible that adaptive query processing is also useful for efficient processing of stream data and we plan to investigate the utility of this complementary technique in the future.

Directly related work on stream data query processing architectures shares many of the goals and target application domains with Aurora. The Streams project [5] attempts to provide complete DBMS functionality along with support for continuous queries over streaming data. Aurora's main emphasis is on efficiently supporting sophisticated continuous queries over a large number of potentially very fast data streams, thus sacrificing some traditional database functionality (e.g., transaction management, storage management, etc.) for extreme scalability. The Fjords architecture [24] addresses continuous query execution strategies that combine push-based sensor data with data from traditional sources that produce data using a pull-based, blocking interface.

Tools for data mining of stream-based data have received considerable attention lately. Some tools only allow one pass over stored stream data, for example [19]. Their main motivation is to be able to handle standard graph problems in one pass. Others [9, 20, 35] are interested in addressing data mining problems and, thus, require multiple passes over stream data. In contrast to efforts that are performing historical analysis of streams, Aurora is focused on monitoring streams in real time.

Work in sequence databases [30, 31] defined sequence definition and manipulation languages over discrete data sequences (typically by assigning each incoming tuple with a specific point in time). The Chronicle data model [22] defined a restricted view definition and manipulation language over append-only sequences. Aurora's algebra extends relevant aspects of previous proposals by proposing a binary windowed operator (i.e., join), which are indispensable for continuous query execution over data streams.

Our work is also relevant to materialized views [14], which are essentially stored continuous queries that are re-executed (or incrementally updated) as their base data are modified. However, Aurora's notion of continuous queries differs from materialized views primarily in that Aurora updates are append-only; query results are streamed (rather than stored); and high stream data rates may require load shedding or other approximate query processing techniques that trade off efficiency for result accuracy.

Systems have been proposed that address the processing of large scientific data sets that represent streams. These systems focus on processing stored data as opposed to real-time data and provide a stream-based programming model that is at a much lower level of abstraction than our set of operations. For example, DataCutter [6] provides a stream abstraction that supports reading a section of the stream and not higher-level operators like Select or Join. However, DataCutter has investigated the use of multidimensional indexing techniques to support range queries over very large stored data sets. Hence, it has similarities to our proposed R-tree optimizations.

Previous work investigated techniques that trade off the efficiency of query execution with the accuracy of query results. The basic approach is to compute approximate answers (with corresponding confidence bounds) to ad-hoc queries by scanning subsets of base data at query time (e.g., [18]) or by avoiding accesses to base data through the use of pre-computed data summaries (e.g., [2]). Clearly, these work do not directly apply to continuous queries over data streams and, thus, to Aurora. Recent work addressed approximate answers to queries over data streams. [12] uses histogram-based techniques to compute approximate answers *correlated aggregation* queries. [13] describes wavelet-based techniques to compute data summaries, which are then used to provide approximate answers to ad-hoc queries. We believe that these and similar techniques can profitably be used in conjunction with Aurora's

dynamic load shedding approach (which also produces approximate query results), and plan to further explore them in future.

Our work is likely to benefit from and contribute to the considerable research on temporal databases [25], since Aurora has a fundamental temporal aspect to it. Likewise, we can benefit from the literature on real-time databases [23, 25], and main-memory databases [11]. In a real-time database system, transactions are assigned timing constraints and the system attempts to ensure a degree of confidence in meeting these timing requirements. Hence, this work deals with a form of load shedding. However, the Aurora notion of QoS specification extends the soft and hard deadlines employed in real-time databases to general utility functions. Furthermore, real-time databases associate deadlines with individual ad hoc transactions, whereas Aurora associates its QoS curves with the output from stream processing and, thus, has to support continuous timing requirements.

Main-memory databases are often used in applications with high performance requirements. They assume that most, if not all, operational data is kept in main memory at all times. However, they assume a HADP model, whereas Aurora proposes a DAHP model that builds streams as fundamental Aurora objects.

Finally, many recent papers [5, 8, 10, 34] have addressed the data management needs of specific monitoring or tracking applications. We take inspiration from these works as we try to design a general-purpose system that could be used to efficiently support all these tasks.

8 Conclusions

Monitoring applications are those where streams of information, triggers, real-time requirements, and imprecise data are prevalent. Traditional DBMSs are based on the HADP model, and thus cannot provide adequate support for such applications. In this paper, we have described the architecture of Aurora, a DAHP system, oriented towards monitoring applications. We have presented the basic system architecture, along with the primitive building blocks for workflow processing. We followed with the structure for optimizing a large Aurora network that entails combining and reordering boxes, combining collections of filters into *superboxes* for R-tree processing, and trading computation for storage where necessary. Our approach to shedding load entails maintaining satisfactory response time for high priority outputs at the expense of more tolerant ones. This work was then generalized to account for the value of the output that is being produced—processing on low-value outputs are deferred or discarded. The paper also presents a sketch of transaction processing issues.

As we emphasized in the introduction, a large and important subset of monitoring applications, in particular those that involve sensor-generated data streams, commonly needs to process missing and imprecise data values. We are currently extending the Aurora model to support monitoring applications in the presence of such values.

The load shedding approach that we described in this paper is based on intelligently shedding data. As another promising research direction, we plan to investigate a complementary *cycle shedding* approach, where the idea is to replace *expensive* operator boxes with cheaper counterparts.

The implementation of Aurora is already underway, and we hope to have a running prototype available by the middle of next year.

References

[1] Informix White Paper. Time Series: The Next Step for Telecommunications Data Management. http://www-4.ibm.com/software/data/informix/pubs/whitepapers/nextstep_wp.pdf.

[2] S. Acharya, P. B. Gibbons, V. Poosala, and S. Ramaswamy. The Aqua Approximate Query Answering System. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, 1999.

[3] M. Altinel and M. J. Franklin. Efficient Filtering of XML Documents for Selective Dissemination of Information. In *Proceedings of the 26th International Conference on Very Large Data Bases (VLDB)*, Cairo, Egypt, 2000.

[4] R. Avnur and J. Hellerstein. Eddies: Continuously Adaptive Query Processing. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, Dallas, TX, 2000.

[5] S. Babu and J. Widom. Continuous Queries over Data Streams. *SIGMOD Record*, 30(3):109-120, 2001.

[6] M. D. Beynon, R. Ferreira, T. Kurc, A. Sussman, and J. Saltz. DataCutter: Middleware for Filtering Very Large Scientific Datasets on Archival Storage Systems. In *Proceedings of the 8th Goddard Conference on Mass Storage Systems and Technologies*, College Park, MD, 2000.

[7] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, Dallas, TX, 2000.

[8] B. Coifman. Identifying the Onset of Congestion Rapidly with Existing Traffic Detectors. *IEEE Transactions on Intelligent Transportation Systems (ITSC). submitted for publication.*, 2001.

[9] C. Cortes, K. Fisher, D. Pregibon, A. Rogers, and F. Smith. Hancock: A Language for Extracting Signatures from Data Streams. In *Proceedings of the 7th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, San Francisco, CA, 2000.

[10] N. G. Duffield and M. Grossglauser. Trajectory Sampling for Direct Traffic Observation. *IEEE/ACM Transactions on Networking (TON)*, 9(3):280-292, 2001.

[11] H. Garcia-Molina and K. Salem. Main Memory Database Systems: An Overview. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 4(6):509-516, 1992.

[12] J. Gehrke, F. Korn, and D. Srivastava. On Computing Correlated Aggregates over Continual Data Streams. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, Santa Barbara, CA, 2001.

[13] A. C. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. Strauss. Surfing Wavelets on Streams: One-pass Summaries for Approximate Aggregate Queries. In *Proceedings of the 26th International Conference on Very Large Data Bases (VLDB)*, Roma, Italy, 2001.

[14] A. Gupta and I. S. Mumick. Maintenance of Materialized Views: Problems, Techniques, and Applications. *IEEE Data Engineering Bulletin*, 18(2):3-18, 1995.

[15] A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, Boston, MA, 1984.

[16] E. N. Hanson, C. Carnes, L. Huang, M. Konyala, L. Noronha, S. Parthasarathy, J. B. Park, and A. Vernon. Scalable Trigger Processing. In *Proceedings of the 15th International Conference on Data Engineering (ICDE)*, Sydney, Austrialia, 1999.

[17] J. M. Hellerstein, M. J. Franklin, S. Chandrasekaran, A. Deshpande, K. Hildrum, S. Madden, V. Raman, and M. Shah. Adaptive Query Processing: Technology in Evolution. *IEEE Data Engineering Bulletin*, 23(2):7-18, 2000.

[18] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online Aggregation. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, Tucson, AZ, 1997.

[19] M. R. Henzinger, P. Raghavan, and S. Rajagopalan. Computing on Data Streams. Compaq Systems Research Center, Palo Alto, California Technical Report TR-1998-011, May 1998.

[20] C. Hidber. Online Association Rule Mining. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, Philadelphia, PA, 1999.

[21] Z. G. Ives, D. Florescu, M. Friedman, A. Levy, and D. S. Weld. An Adaptive Query Execution System for Data Integration. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, Philadelphia, PA, 1999.

[22] H. V. Jagadish, I. S. Mumick, and A. Silberschatz. View Maintenance Issues for the Chronicle Data Model. In *Proceedings of the 14th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, San Jose, CA, 1995.

[23] B. Kao and H. Garcia-Molina, "An Overview of Realtime Database Systems," in *Real Time Computing*, W. A. Halang and A. D. Stoyenko, Eds.: Springer-Verlag, 1994.

[24] S. Madden and M. J. Franklin. Fjording the Stream: An Architecture for Queries over Streaming Sensor Data. In *Proceedings of the 18th International Conference on Data Engineering (ICDE)*, San Jose, CA, 2002.

[25] G. Ozsoyoglu and R. T. Snodgrass. Temporal and Real-Time Databases: A Survey. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 7(4):513-532, 1995.

[26] N. Paton and O. Diaz. Active Database Systems. *ACM Computing Surveys*, 31(1):63-103, 1999.

[27] U. Schreier, H. Pirahesh, R. Agrawal, and C. Mohan. Alert: An Architecture for Transforming a Passive DBMS into an Active DBMS. In *Proceedings of the 17th International Conference on Very Large Data Bases (VLDB)*, Barcelona, Spain, 1991.

[28] T. K. Sellis. Multiple-Query Optimization. *ACM Transactions on Database Systems (TODS)*, 13(1):23-52, 1988.

[29] T. K. Sellis and S. Ghosh. On the Multiple-Query Optimization Problem. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 2(2):262-266, 1990.

[30] P. Seshadri, M. Livny, and R. Ramakrishnan. The Design and Implementation of a Sequence Database System. In *Proceedings of the 22th International Conference on Very Large Data Bases (VLDB)*, Bombay, India, 1996.

[31] P. Seshadri, M. Livny, and R. Ramakrishnan. SEQ: A Model for Sequence Databases. In *Proceedings of the 11th International Conference on Data Engineering (ICDE)*, Taipei, Taiwan, 1995.

[32] D. B. Terry, D. Goldberg, D. Nichols, and B. M. Oki. Continuous Queries over Append-Only Databases. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, San Diego, CA, 1992.

[33] T. Urhan and M. J. Franklin. Dynamic Pipeline Scheduling for Improving Interactive Query Performance. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB)*, Rome, Italy, 2001.

[34] O. Wolfson, S. Chamberlain, P. Sistla, B. Xu, and J. Zhou. DOMINO: Databases fOr MovINg Objects tracking. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, Philadelphia, PA, 1999.

[35] B.-K. Yi, N. Sidiropoulos, T. Johnson, H. V. Jagadish, C. Faloutsos, and A. Biliris. Online Data Mining for Co-Evolving Time Sequences. In *Proceedings of the 16th International Conference on Data Engineering (ICDE)*, San Diego, CA, 2000.

Appendix 1

In this appendix, we give a formal definition of streams and the Aurora operations on streams.

A stream is a potentially infinite ordered set of tuples (elements). Elements in a stream can be identified by their index values (such as timestamps or integer positions) that specify where the element lies in the ordering. Generally, the index is some sort of encoding of time, but other indexes are certainly possible.

More formally, a *stream*, *S*, is a set of (index, tuple) pairs:

$$S = \{(i_1, t_1), (i_2, t_2), \ldots, (i_n, t_n), \ldots\}$$

such that all indices in *S*, belong to some *index type* (see below), and all tuples in *S* adhere to the same schema.

The indexes used in a given stream must belong to a totally ordered type (i.e., a type for which one of $\prec$, $\succ$, or $=$ holds between any two elements). Such types are *Index Types*. Every index type must be associated with a *unit of measure*, which defines minimal increments in index values. For example, if the index type consists of timestamps (as in stock data), then the associated unit measure might be *15 min* or *1 hour* depending on the maximum arrival frequency of stock data. If an index type consists of positive integers (reflecting the relative positions of elements in the stream), then the associated unit measure would be 1. Units of measure are required in order to express distances between elements, and to define a window size within a query. Note that units of measure do not imply that streams must be periodic ─ they simply define the maximum frequency with which stream elements appear.

Our assumption is that a given stream source (e.g., sensors, broadcasting software, etc.) determines the index type and associated unit of measure for the elements emanating from it. This would require the stream source to have access to a clock (if the index is timestamp-based) or to maintain a counter (if the index is position-based) of the elements it generates. In addition, for streams of the same index type to be properly aligned, indices of a stream must start from a multiple of its unit measure.

An index type must provide the following operations:

- *Ordering relations* ($\prec$, $\succ$, $\leq$, $\geq$): Because the index type is a total order, it must be the case for any elements of the index type, i_1 and i_2, that either $i_1 \prec i_2$, $i_2 \prec i_1$, or $i_1 = i_2$.

- *Position Arithmetic* (+, -, %): For any index value, i, $i + k$ identifies the index element that is k units away from i. For example, if i is a timestamp-based index type with unit 1 hour, then 15:00 + 3 = 18:00. '-' is defined similarly. '%' performs modular arithmetic on index values. That is, for any index value, i and positive integer, k, $i \% k = r$ such that $0 \leq r \leq k\text{-}1$ and for some m, $i = mk + r$. Note that position multiplication is defined in the standard way in terms of position addition.
- *Distance* (Δ): For any two index values, i_1 and i_2, $\Delta(i_1, i_2)$ is the distance (in units) between them. Note that because units represent the smallest observed increment in index values, the result of an expression involving Δ will always be an integer.

Aurora defines seven primitive stream operations: filter (σ), map (μ), windowed map (M), merge (+), join ($\triangleright\triangleleft$), resample ($\rho$) and drop ($\delta$). In describing the semantics of these operators, we adopt the following shorthand notation:

Given a stream, $S = \{(i_1, t_1), (i_2, t_2), \ldots, (i_n, i_n), \ldots\}$:

1. $S[i_k] = \{(i_k, t_k) \mid (i_k, t_k) \in S\}$
2. $S[i_m \ldots i_n] = \{(i_k, t_k) \mid (i_k, t_k) \in S, i_m \leq i_k \leq i_n\}$
3. $index(i_k, t_k) = i_k$
4. $value(i_k, t_k) = t_k$

Using this notation, we define the operations as follows:

1. *Selection* (σ): Given a predicate, p, on (index, tuple) pairs, $\sigma_p(S)$ returns the set of pairs in S that satisfy p. More formally:

   $$\sigma_p(S) = \{x \mid x \in S, p(x)\}$$

2. *Map* (μ): Given a function f, that takes an (index, tuple) pair and returns another (index, tuple) pair, $\mu_f(S)$ returns the stream resulting from applying f to every element in S. More formally:

   $$\mu_f(S) = \{f(x) \mid x \in S\}$$

3. *Windowed-Map* (M): Given a function f, that takes a set of (index, tuple) pairs (the *window*) and returns another (index, tuple) pair, $M_{f,w}(S)$ returns the stream resulting from applying f to every set of elements in S whose index values differ by at most w units. More formally:

   $$M_{f,w}(S) = \{f(S') \mid \exists d(S' = S[d..d + w - 1])\}$$

4. *Merge* (+): Merge performs the union of tuples from separate but compatible streams (i.e., those that have the same index types, units of measure, and tuple schemas). More formally, for streams S_1 and S_2:

   $$S_1 + S_2 = S_1 \cup S_2$$

5. *Resample* (ρ): The resample operator predicts the values in one stream for all index values represented in another. The prediction is made based on a prediction function, f, and a window size, w, that indicates how much history from S_1 must be retained for f to be used for prediction. More formally, for any pair of streams, S_1 and S_2 with compatible index types, window size, w, and function, f, that takes any substream of S_1 and returns an (index, tuple) pair:

$$\rho_{w,f}(S_1, S_2) = \{(i, f(S')) \mid S_2[i] \neq \emptyset, S' = \{(i', x) \in S_1 \mid \Delta(i, i') \leq w\}\}$$

6. *Join* ($\triangleright\triangleleft$): The join operator joins pairs of streams with compatible indexes (i.e., the same types and unit measures). Elements from different streams are *joined* provided that the distance between them (as determined by their indexes) is within a provided window size. More formally, for any streams S_1 and S_2; predicate, p, on pairs of stream elements; function, f, that takes two (index, tuple) pairs and returns another (index, tuple) pair, and window size, w:

$$S1 \triangleright\triangleleft_{p,f,w} S2 = \{f(x, y) \mid \exists i, j(S_1[i] \subseteq S_1, S_2[j] \subseteq S_2, \Delta(i, j) \leq w, x \in S_1[i], y \in S_2[j]), p(x, y)\}$$

7. *Drop* (δ): The drop operator filters elements of streams according to their index values. Specifically, given a *period* of k units, δ_k would block all elements for every k^{th} index value. More formally, given a positive integer, k:

$$\delta_k(S) = \{(i,t) \mid (i,t) \in S, i\%k \neq 0\}$$

Appendix 2

	filter	map	w. map	merge	resample	join	drop
filter	*yes*						
map	1	4					
w. map	*no*	5	8				
merge	*yes*	*yes*	*no*	*yes*			
resample	2	6	*no*	*no*	9		
join	3	7	*no*	*no*	*no*	10	
drop	*yes*	*yes*	*no*	*yes*	*no*	*no*	*yes*

Table 2: Detailed operator commutativity table

This appendix treats conditional commutativity in more detail. In Table 2, we present the commutativity information presented in Table 1 with all the conditional cases numbered.What follows are the precise conditions for commutativity for each of the ten cases.

1. $\sigma_p(\mu_f(S)) \equiv \mu_f(\sigma_p(S))$, if attributes referred to in p are unchanged by f.
2. $\sigma_p(\rho(f, S_1, S_2)) \equiv \rho(f, \sigma_p(S_1), S_2)$, if
   i. p is always *true* (i.e., *selectivity*$(p) = 1$) on both S_1 and $\rho(f, S_1, S_2)$, or
   ii. $\mu_{index}(S_2) \subseteq \mu_{index}(\sigma_p(S_1))$.
3. $\sigma_p(\triangleright\triangleleft_{q,concat,w}(S_1, S_2)) \equiv \triangleright\triangleleft_{p \wedge q,concat,w}(S_1, S_2)$, where

$$\triangleright\!\triangleleft_{p\wedge q,concat,w}(S_1,S_2)\equiv\begin{cases}\triangleright\!\triangleleft_{q,concat,w}(\sigma_p(S_1),S_2), & \text{if } p \text{ is defined on } S_1 \text{ and } p \text{ is not defined on } S_2\\ \triangleright\!\triangleleft_{q,concat,w}(S_1,\sigma_p(S_2)), & \text{if } p \text{ is not defined on } S_1 \text{ and } p \text{ is defined on } S_2\\ \triangleright\!\triangleleft_{q,concat,w}(\sigma_p(S_1),\sigma_p(S_2)), & \text{if } p \text{ is defined on both } S_1 \text{ and } S_2\\ \triangleright\!\triangleleft_{q,concat,w}(\sigma_{p_1}(S_1),\sigma_{p_2}(S_2)), & \text{if } p \text{ is not defined on either } S_1 \text{ or } S_2, \text{ but } p=p_1\wedge p_2,\\ & \text{where } p_1 \text{ is defined on } S_1 \text{ and } p_2 \text{ is defined on } S_2\\ \triangleright\!\triangleleft_{q,concat,w}(S_1,S_2), & \text{otherwise.}\end{cases}$$

In the above formulation, *concat*, which stands for concatenation of two tuples, is given as the default combining function of the join operator. In fact, any function that keeps the attributes of the joined tuples that are also referred by predicate p could be used instead.

4. $\mu_f(\mu_g(S))\equiv\mu_g(\mu_f(S))$, if $fog=gof$.

5. $\mu_f(\mu_{g,w}(S))\equiv\mu_{g,w}(\mu_f(S))$, if for any index i,

   $f(g(S[i..i+w-1]))=g(f(S[i]),f(S[i+1]),...,f(S[i+w-1]))$.

   In other words, f should distribute over g. Note that this formulation assumes that indices act as keys.

6. $\mu_g(\rho(f,S_1,S_2))\equiv\rho(f,\mu_g(S_1),S_2)$, if

   i. f introduces no new values in S_1 (e.g., f is a step-wise constant function), or

   ii. g is the identity function, or

   iii. $\mu_{index}(S_2)\subseteq\mu_{index}(S_1)$.

7. $\mu_f(\triangleright\!\triangleleft_{p,concat,w}(S_1,S_2))\equiv\triangleright\!\triangleleft_{p,concat\ o\ f,w}(S_1,S_2)$, where

$$\triangleright\!\triangleleft_{p,concatof,w}(S_1,S_2)\equiv\begin{cases}\triangleright\!\triangleleft_{p,concat,w}(\mu_f(S_1),S_2), & \text{if } f \text{ is defined on } S_1 \text{ and } f \text{ is not defined on } S_2\\ \triangleright\!\triangleleft_{p,concat,w}(S_1,\mu_f(S_2)), & \text{if } f \text{ is not defined on } S_1 \text{ and } f \text{ is defined on } S_2\\ \triangleright\!\triangleleft_{p,concat,w}(\mu_f(S_1),\mu_f(S_2)), & \text{if } f \text{ is defined on both } S_1 \text{ and } S_2\\ \triangleright\!\triangleleft_{p,concat,w}(\mu_{f_1}(S_1),\mu_{f_2}(S_2)), & \text{if } f \text{ is not defined on either } S_1 \text{ or } S_2, \text{ but } f=f_1\ of_2,\\ & \text{where } f_1 \text{ is defined on } S_1 \text{ and } f_2 \text{ is defined on } S_2\\ \triangleright\!\triangleleft_{p,concat,w}(S_1,S_2), & \text{otherwise}\end{cases}$$

   and attributes referred to in p are unchanged by f.

8. $\mu_{f,w}(\mu_{g,w}(S))\equiv\mu_{g,w}(\mu_{f,w}(S))$, if for any i

   $f(g(S[i..i+u-1]),g(S[i+1..i+u]),..,g(S[i+w-1..i+u+w-2]))=$

   $g(f(S[i..i+u-1]),f(S[i+1..i+u]),..,f(S[i+w-1..i+u+w-2]))$

9. $\rho(f,\rho(g,S_1,S_3),S_2)\equiv\rho(g,\rho(f,S_1,S_3),S_2)$, if $f=g$.

10. $\triangleright\!\triangleleft_{p,concat,w}(\triangleright\!\triangleleft_{q,concat,u}(S_1,S_2),S_3)\equiv\triangleright\!\triangleleft_{q,concat,u}(S_1,\triangleright\!\triangleleft_{p,concat,w}(S_2,S_3))$,

    if p is defined on attributes of S_2 and S_3, and q is defined on attributes of S_1 and S_2.