The VLDB Journal (2003) / Digital Object Identifier (DOI) 10.1007/s00778-003-0095-z

Aurora: a new model and architecture for data stream management

Daniel J. Abadi[1], **Don Carney**[2], **Uğur Çetintemel**[2], **Mitch Cherniack**[1], **Christian Convey**[2], **Sangdon Lee**[2], **Michael Stonebraker**[3], **Nesime Tatbul**[2], **Stan Zdonik**[2]

[1] Department of Computer Science, Brandeis University, Waltham, MA 02254
e-mail: {dna,mfc}@cs.brandeis.edu
[2] Department of Computer Science, Brown University, Providence, RI 02912
e-mail: {dpc,ugur,cjc,sdlee,tatbul,sbz}@cs.brown.edu
[3] Department of EECS and Laboratory of Computer Science, M.I.T., Cambridge, MA 02139
e-mail: stonebreaker@lcs.mit.edu

Edited by ♣. Received: ♣/ Accepted: ♣
Published online: ♣ 2003 –

Abstract. This paper describes the basic processing model and architecture of Aurora, a new system to manage data streams for monitoring applications. Monitoring applications 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 Aurora, a new DBMS currently under construction at Brandeis University, Brown University, and M.I.T. We first provide an overview of the basic Aurora model and architecture and then describe in detail a stream-oriented set of operators.

Keywords: Ddata stream management – Continuous queries – Database triggers – Real-time systems – Quality-of-service

1 Introduction

Traditional DBMSs have been oriented toward business data processing, and consequently are designed to address the needs of these 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 on 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 afterthought to current systems, and none has an implementation that scales to a large number of triggers. Fourth, DBMSs assume that data elements are synchronized and that queries have exact answers. In many stream-oriented applications, data arrive asynchronously and answers must be computed with incomplete information. 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 monitored data and the query 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 that extends over some 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. Ceiling-mounted sensors inside a building and the GPS system in the open air generate large volumes of location data. 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 h. 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 are often lost, stale, or intentionally omitted for processing reasons. 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, in managing data streams with high input rates, it might be necessary to shed load by dropping less important input data. All of this, by necessity, leads to approximate answers.

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 resource management (e.g., scheduling) and graceful degradation strategies (e.g., load shedding) during periods of high load. We expect that applications will supply Quality of Service (QoS) specifications that will be used by the running system to make these dynamic resource allocation decisions.

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 tuples, thereby dramatically slowing performance. Alternately, he can encode time series information in binary large objects to achieve physical locality, at the expense of making queries to individual values in the time series very difficult. One system that tries to do something more intelligent with time series data is the Informix Universal Server, which implemented a time-series data type and associated methods that speed retrieval of values in a time series [1]; however, this system does not address the concerns raised above.

If a monitoring application had a 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 application. Using this implementation, the system cannot reason about the triggers (e.g., optimization), 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 approximate query answering. 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. We use Aurora to illustrate design issues that would arise in any system of this kind.

figure=e:/vldb/095/slide1.eps,width=

Fig. 1. Aurora system model

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 an ever broader class of objects, monitoring applications will expand in scope. Currently such monitoring is expensive and restricted to costly items like automobiles (e.g., Lojack technology [2]). In the future, it will be available for most objects whose position is of interest.

In Sect. 2, we begin by describing the basic Aurora architecture and fundamental building blocks. In Sect. 3, we show why traditional query optimization fails in our environment and present our alternate strategies for optimizing Aurora applications. Section 4 describes the run-time architecture and behavior of Aurora, concentrating on storage organization, scheduling, introspection, and load shedding. In Sect. 5, we describe Aurora's data stream operators in detail. In Sect. 6, we discuss the myriad of related work that has preceded our effort. We describe the status of our prototype implementation in Sect. 7 and conclude in Sect. 8.

2 Aurora system model

Aurora data are assumed to come from a variety of data sources such as computer programs that generate values at regular or irregular intervals or 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 presented by a data source. Each data source is assumed to have a unique source identifier, and Aurora timestamps every incoming tuple 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, tuples flow through a loop-free, directed graph of processing operations (i.e., boxes). Ultimately, output streams are presented to *applications*, which must be programmed to deal with the asynchronous tuples in an output stream. Aurora can also maintain historical storage, primarily in order to support ad hoc queries. Figure 1 illustrates the high-level system model.

Aurora's query algebra (SQuAl[1]) contains built-in support for seven primitive operations for expressing its stream processing requirements. Many of these have analogs in the relational query operators. For example, we support a *filter* operator that, like the relational operator *select*, applies any number of predicates to each incoming tuple, routing the tu-

[1] SQuAl is short for [S]tream [Qu]ery [Al]gebra.

figure=e:/vldb/095/slide2.eps

Fig. 2. Aurora query model

ples according to which predicates they satisfy. Another operator, (*Aggregate*), computes stream aggregates in a way that addresses the fundamental push-based nature of streams, applying a function across a window of values in a stream (e.g., a moving average). In environments where data can be stale or time imprecise, windowed operations are a necessity.

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 *Union* operation, whereby two streams can be put together. If, additionally, one tuple 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.

Arcs in an Aurora diagram actually represent a collection of streams with common schema. The actual number of streams on an arc is unspecified, making it easy to have streams appear and disappear without modification to the Aurora network.

2.1 Query model

Aurora supports continuous queries (real-time processing), views, and ad hoc queries all using substantially the same mechanisms. All three modes of operation use the same conceptual building blocks. Each mode processes flows based on *QoS specifications* – each output in Aurora is associated with two-dimensional QoS graphs that specify the utility of the output in terms of several performance-related and quality-related attributes (see Sect. 4.1). The diagram in Fig. 2 illustrates the processing modes supported by Aurora.

The topmost path represents a *continuous query*. In isolation, data elements flow into boxes, are processed, and flow further downstream. In this scenario, there is no need to store any data elements once they are processed. Once an input has worked its way through all reachable paths, that data item is drained from the network. The QoS specification at the end of the path controls how resources are allocated to the processing elements along the path. One can also view an Aurora network (along with some of its applications) 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 tuple is delivered to an application, which can take the appropriate action.

The dark circles on the input arcs to boxes b_1 and b_2 represent *connection points*. A connection point is an arc that supports dynamic modification to the network. New boxes can be added to or deleted from a connection point. When a new application connects to the network, it will often require access to the recent past. As such, a connection point has the potential for persistent storage (see Sect. 4.2). Persistent storage retains data items beyond their processing by a particular box. In other words, as items flow past a connection point, they are cached in a persistent store for some period of time. They are not drained from the network by applications. Instead, a persistence specification indicates exactly how long the items are kept, so that a future ad hoc query can get historical results. In the figure, the leftmost connection point is specified to be available for *2 h*. This indicates that the beginning of time for newly connected applications will be 2 h in the past.

Connection points can be generalized to allow an elegant way of including *static data sets* in Aurora. Hence we allow a connection point to have no upstream node, i.e., a dangling connection point. Without an upstream node the connection point cannot correspond to an Aurora stream. Instead, the connection point is decorated with the identity of a stored data set in a traditional DBMS or other storage system. In this case, the connection point can be *materialized* and the stored tuples passed as a stream to the downstream node. In this case, such tuples will be *pushed* through an Aurora network. Alternately, query execution on the downstream node can *pull* tuples by running a query to the store. If the downstream node is a filter or a join, pull processing has obvious advantages. Moreover, if the node is a join between a stream and a stored data set, then an obvious query execution strategy is to perform iterative substitution whenever a tuple from the stream arrives and perform a lookup to the stored data. In this case, a window does not need to be specified as the entire join can be calculated.

The middle path in Fig. 2 represents a view. In this case, a path is defined with no connected application. It is allowed to have a QoS specification as an indication of the importance of the view. Applications can connect to the end of this path whenever there is a need. Before this happens, the system can propagate some, all, or none of the values stored at the connection point in order to reduce latency for applications that connect later. Moreover, it can store these partial results at any point along a view path. This is analogous to a materialized or partially materialized view. View materialization is under the control of the scheduler.

The bottom path represents an ad hoc query. An ad hoc query can be attached to a connection point at any time. The semantics of an ad hoc query is that the system will process data items and deliver answers from the earliest time T (persistence specification) stored in the connection point until the query branch is explicitly disconnected. Thus, the semantics for an Aurora ad hoc query is the same as a continuous query that starts executing at $t_{now} - T$ and continues until explicit termination.

2.2 Graphical user interface

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.

Boxes and groups have a tag, an argument list, a description of the Functionality, and, ultimately, a manual page. Users can teleport to specific places in an Aurora network by querying these attributes. 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. 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 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 large number of boxes. Furthermore, high data rates add another dimension to the problem. Lastly, we expect many changes to be made to an Aurora network over time, and it seems unreasonable to take the network offline to perform a compile time optimization. We now present our strategies to optimize an Aurora network.

3.1 Dynamic continuous query optimization

We begin execution of an unoptimized Aurora network, i.e., the one that the user constructed. During execution we gather run-time statistics such as the average cost of box execution and box selectivity. Our goal is to perform run-time optimization of a network, without having to quiesce it. Hence combining all the boxes into a massive query and then applying conventional query optimization is not a workable approach. Besides being NP-complete [26], it would require quiescing the whole network. Instead, the optimizer will select a portion of the network for optimization. Then it will find all connection points that surround the subnetwork to be optimized. It will hold all input messages at upstream connection points and drain the subnetwork of messages through all downstream connection points. The optimizer will then apply the following local tactics to the identified subnetwork.

- *Inserting projections.* It is unlikely that the application administrator will have inserted map operators (see Sect. 5) to project out all unneeded attributes. Examination of an Aurora network allows us to insert or move such map operations to the earliest possible points in the network, thereby shrinking the size of the tuples that must be subsequently processed. Note that this kind of optimization requires that the system be provided with operator signatures that describe the attributes that are used and produced by the operators.
- *Combining boxes.* As a next step, Aurora diagrams will be processed to combine boxes where possible. A pairwise 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. In general, combining boxes at least saves the box-execution overhead and reduces the total number of boxes, leading to a simpler diagram.
- *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 commute.

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 tuple. Additionally, each box has a *selectivity*, $s(b)$, which is the expected number of output tuples per input tuple. Consider two boxes, b_i and b_j, with b_j following b_i. In this case, for each input tuple 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:

$$(1 - s(b_j))/c(b_j) < (1 - s(b_i))/c(b_i))$$

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. We use this result in a heuristic algorithm that iteratively reorders boxes (to the extent allowed by their commutativity properties) until no more reorderings are possible.

When the optimizer has found all productive transformations using the above tactics, it constructs a new subnetwork, binds it into the composite Aurora network that is running, and then instructs the scheduler to stop holding messages at the input connection points. Of course, outputs affected by the subnetwork will see a *blip* in response time; however, the remainder of the network can proceed unimpeded.

An Aurora network is broken naturally into a collection of k subnetworks by the connection points that are inserted by the application administrator. Each of these subnetworks can be optimized individually, because it is a violation of Aurora semantics to optimize across a connection point. The Aurora optimizer is expected to cycle periodically through all k subnetworks and run as a background task.

3.2 Ad hoc query optimization

One last issue that must be dealt with is ad hoc query optimization. Recall that the semantics of an ad hoc query is

figure=e:/vldb/095/slide3.eps,width=

Fig. 3. Aurora run-time architecture

that it must run on all the historical information saved at the connection points to which it is connected. Subsequently it becomes a normal portion of an Aurora network until it is discarded. Aurora processes ad hoc queries in two steps by constructing two separate subnetworks. Each is attached to a connection point, so the optimizer can be run before the scheduler lets messages flow through the newly added subnetworks.

Aurora semantics require the historical subnetwork to be run first. Since historical information is organized as a B-tree, the Aurora optimizer begins at each connection point and examines the successor box(es). If the box is a filter, then Aurora examines the condition to see if it is compatible with the storage key associated with the connection point. If so, it switches the implementation of the filter box to perform an indexed lookup in the B-tree. Similarly, if the successor box is a join, then the Aurora optimizer costs performing a merge-sort or indexed lookup, chooses the cheapest one, and changes the join implementation appropriately. Other boxes cannot effectively use the indexed structure, so only these two need be considered. Moreover, once the initial box performs its work on the historical tuples, the index structure is lost, and all subsequent boxes will work in the normal way. Hence, the optimizer converts the historical subnetwork into an optimized one, which is then executed.

When it is finished, the subnetwork used for continuing operation can be run to produce subsequent output. Since this is merely one of the subnetworks, it can be optimized in the normal way suggested above.

In summary, the initial boxes in an ad hoc query can pull information from the B-tree associated with the corresponding connection points. When the historical operation is finished, Aurora switches the implementation to the standard *push-based* data structures and continues processing in the conventional fashion.

4 Run-time operation

The basic purpose of an Aurora run-time network is to process data flows through a potentially large workflow diagram. Figure 3 illustrates the basic Aurora architecture. Here, inputs from data sources and outputs from boxes are fed to the router, which forwards them either to external applications or to the storage manager to be placed on the proper queue. The storage manager is responsible for maintaining the box queues and managing the buffer. Conceptually, the scheduler picks a box for execution, ascertains what processing is required, and passes a pointer to the box description (together with a pointer to the box state) to the multithreaded box processor. The box processor executes the appropriate operation and then forwards the output tuples to the router. The scheduler then ascertains the next processing step and the cycle repeats. The QoS monitor continually monitors system performance and activates the load shedder when it detects an overload situation and poor system performance. The load shedder then sheds load till the performance of the system reaches an acceptable level. The catalog in Fig. 3 contains information regarding the network topology, inputs, outputs, QoS information, and relevant statistics (e.g., selectivity, average box processing costs) and is essentially used by all components.

We now describe Aurora's primary run-time architecture in more detail, focusing primarily on the storage manager, scheduler, QoS monitor, and load shedder.

4.1 QoS data structures

Aurora attempts to maximize the perceived QoS for the outputs it produces. QoS, in general, is a multidimensional function of several attributes of an Aurora system. These include:

- *Response times* – output tuples should be produced in a timely fashion, as otherwise QoS will degrade as delays get longer.
- *Tuple drops* – if tuples are dropped to shed load, then the QoS of the affected outputs will deteriorate.
- *Values produced* – QoS clearly depends on whether or not important values are being produced.

Asking the application administrator to specify a multidimensional QoS function seems impractical. Instead, Aurora relies on a simpler tactic that 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 processing delay of output tuples produced (as illustrated in Fig. 4a). Here, the QoS of the output is maximized if delay is less than the threshold, δ, in the graph. Beyond δ, QoS degrades with additional delay.

Optionally, the application administrator can give Aurora two additional QoS graphs for all outputs in an Aurora system. The first, illustrated in Fig. 4b, shows the percentage of tuples delivered. In this case, the application administrator indicates that high QoS is achieved when tuple delivery is near 100% and that QoS degrades as tuples are dropped. The second optional QoS graph for outputs is shown in Fig. 4c. 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.

Aurora makes several assumptions about the QoS graphs. First, it assumes that all QoS graphs are normalized so that QoS for different outputs can be quantitatively compared. Second, Aurora 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 in steady state. 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 may not produce good results. Third, unless otherwise stated, Aurora assumes that

figure=e:/vldb/095/slide4.eps

Fig. 4. QoS graph types

figure=e:/vldb/095/slide5.eps,width=

Fig. 5. Queue organization

figure=e:/vldb/095/slide6.eps,width=

Fig. 6. Scheduler-storage manager interaction

all its QoS graphs are convex (the value-based graph illustrated in Fig. 4c is an exception). This assumption is not only reasonable but also necessary for the applicability of *gradient walking* techniques used by Aurora for scheduling and load shedding.

Note that Aurora's notion of QoS is general and is not restricted to the types of graphs presented here. Aurora can work with other individual attributes (e.g., throughput) or composite attributes (e.g., a weighted, linear combination of throughput and latency) provided that they satisfy the basic assumptions discussed above. In the rest of this paper, however, we restrict our attention to the graph types presented here.

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

4.2 Storage management

The job of the Aurora Storage Manager (ASM) is to store all tuples required by an Aurora network. There are two kinds of requirements. First, ASM must manage storage for the tuples being passed through an Aurora network, and second, it must maintain extra tuple storage that may be required at connection points.

Queue management. Each windowed operation requires a historical collection of tuples to be stored, equal to the size of the window. Moreover, if the network is currently saturated, then additional tuples may accumulate at various places in the network. As such, ASM must manage a collection of variable-length queues of tuples. There is one queue at the output of each box, which is shared by all successor boxes. Each such successor box maintains two pointers into this queue. The head indicates the oldest tuple that this box has not processed. The tail, in contrast, indicates the oldest tuple that the box needs. The head and tail indicate the box's current window, which slides as new tuples are processed. ASM will keep track of these collections of pointers and can normally discard tuples in a queue that are older than the oldest tail pointing into the queue. In summary, when a box produces a new tuple, it is added to the front of the queue. Eventually, all successor boxes process this tuple and it falls out of all of their windows and can be discarded. Figure 5 illustrates this model by depicting a two-way branch scenario where two boxes, b_1 and b_2, share the same queue (w's refer to window sizes).

Normally queues of this sort are stored as main memory data structures. However, ASM must be able to scale arbitrarily and has chosen a different approach. Disk storage is divided into fixed-length blocks, of a tunable size, *block_size*. We expect the typical environment will use 128-KB or larger blocks. Each queue is allocated one block, and queue management proceeds as above. As long as the queue does not overflow, the single block is used as a circular buffer. If an overflow occurs, ASM looks for a collection of two blocks (contiguous if possible) and expands the queue dynamically to $2 \times$ *block_size*. Circular management continues in this larger space. Of course, queue underflow can be treated in an analogous manner.

At start-up time ASM is allocated a buffer pool for queue storage. It pages queue blocks into and out of main memory using a novel replacement policy. The scheduler and ASM share a tabular data structure that contains a row for each box in the network containing the current scheduling priority of the box and the percentage of its queue that is currently in main memory. The scheduler periodically adjusts the priority of each box, while the ASM does likewise for the main memory residency of the queue. This latter piece of information is used by the scheduler for guiding scheduling decisions (see Sect. 4.3). The data structure also contains a flag to indicate that a box is currently running. Figure 6 illustrates this interaction.

When space is needed for a disk block, ASM evicts the lowest-priority main memory resident block. In addition, whenever ASM discovers a block for a queue that does not correspond to a running block, it will attempt to "upgrade" the block by evicting it in favor of a block for the queue corresponding to a higher-priority box. In this way, ASM is continually trying to keep all the required blocks in main memory that correspond to the top-priority queues. ASM is also aware of the size of each queue and whether it is contiguous on disk. Using this information it can schedule multiblock reads and writes and garner added efficiency. Of course, as blocks move through the system and conditions change, the scheduler will adjust the priority of boxes and ASM will react by adjusting the buffer pool. Naturally, we must be careful to avoid the well-known *hysteresis* effect, whereby ASM and the scheduler start working at cross purposes, and performance degrades sharply.

Connection point management. As noted earlier, the Aurora application designer indicates a collection of connection points to which collections of boxes can be subsequently connected. This satisfies the Aurora requirement to support ad hoc queries. Associated with each connection point is a history requirement and an optional storage key. The history requirement indicates the amount of historical information that must be retained. Sometimes the amount of retained history is less than the maximum window size of the successor boxes. In this case, no extra storage need be allocated. Additional history is usually requested.

In this case, ASM will organize the historical tuples in a B-tree organized on the storage key. If one is not specified, then a B-tree will be built on the timestamp field in the tuple. When tuples fall off the end of a queue associated with a connection point, then ASM will gather up batches of such tuples and insert them into the corresponding B-tree. Periodically, it will make a pass through the B-tree and delete all tuples that are older than the history requirement. Obviously, it is more efficient to process insertions and deletions in batches than one by one.

Since we expect B-tree blocks to be smaller than *block_size*, we anticipate splitting one or more of the buffer pool blocks into smaller pieces and paging historical blocks into this space. The scheduler will simply add the boxes corresponding to ad hoc queries to the data structure mentioned above and give these new boxes a priority. ASM will react by prefetching index blocks, but not data blocks, for worthy indexed structures. In turn, it will retain index blocks as long as there are not higher-priority buffer requirements. No attempt will be made to retain data blocks in main memory.

4.3 Real-time scheduling

Scheduling in Aurora is a complex problem due to the need to simultaneously address several issues including large system scale, real-time performance requirements, and dependencies between box executions. Furthermore, tuple processing in Aurora spans many scheduling and execution steps (i.e., an input tuple typically needs to go through many boxes before potentially contributing to an output stream) and may involve multiple accesses to secondary storage. Basing scheduling decisions solely on QoS requirements, thereby failing to address end-to-end tuple processing costs, might lead to drastic performance degradation, especially under resource constraints. To this end, Aurora not only aims to maximize overall QoS but also makes an explicit attempt to reduce overall tuple execution costs. We now describe how Aurora addresses these two issues.

Train scheduling. In order to reduce overall processing costs, Aurora observes and exploits two basic *nonlinearities* when processing tuples:

- *Interbox nonlinearity.* End-to-end tuple processing costs may drastically increase if buffer space is not sufficient and tuples need to be shuttled back and forth between memory and disk several times throughout their lifetime. Thus one important goal of Aurora scheduling is to minimize tuple trashing. Another form of interbox nonlinearity occurs when passing tuples between box queues. If the scheduler can decide in advance that, say, box b_2 is going to be scheduled right after box b_1 (whose outputs feed b_2), then the storage manager can be bypassed (assuming there is sufficient buffer space) and its overhead avoided while transferring b_1's outputs to b_2's queue.
- *Intrabox nonlinearity.* The cost of tuple processing may decrease as the number of tuples that are available for processing at a given box increases. This reduction in unit tuple processing costs may arise for two reasons. First, the total number of box calls that need to be made to process a given number of tuples decreases, cutting down low-level overheads such as calls to the box code and context switch. Second, a box may, depending on its semantics, optimize its execution better with a larger number of tuples available in its queue. For instance, a box can materialize intermediate results and reuse them in the case of windowed operations or use merge-join instead of nested loops in the case of joins.

Aurora exploits the benefits of nonlinearity in both interbox and intrabox tuple processing primarily through *train scheduling*, a set of scheduling heuristics that attempt to (1) have boxes queue as many tuples as possible without processing, thereby generating long tuple trains; (2) process complete trains at once, thereby exploiting intrabox nonlinearity; and (3) pass them to subsequent boxes without having to go to disk, thereby exploiting interbox nonlinearity. To summarize, train scheduling has two goals: its primary goal is to minimize the number of I/O operations performed per tuple. A secondary goal is to minimize the number of box calls made per tuple. Henceforth we will use the term *train scheduling* to describe the batching of multiple tuples as input to a single box and the term *superbox scheduling* to describe scheduling actions that push a tuple train through multiple boxes.

One important implication of train and superbox scheduling is that, unlike traditional blocking operators that wake up and process new input tuples as they arrive, the Aurora scheduler tells each box when to execute and how many queued tuples to process. This somewhat complicates the implementation and increases the load of the scheduler but is necessary for creating and processing tuple trains, which will significantly decrease overall execution costs.

Priority assignment. The latency of each output tuple is the sum of the tuple's processing delay and its waiting delay. Unlike the processing delay, which is a function of input tuple rates and box costs, the waiting delay is primarily a function of scheduling. Aurora's goal is to assign priorities to outputs so as to achieve the per-output waiting delays that maximize the overall QoS.

The priority of an output is an indication of its urgency. Aurora currently considers two approaches for priority assignment. The first one, a state-based approach, assigns priorities to outputs based on their expected utility under the current system state and then picks for execution, at each scheduling instance, the output with the highest utility. In this approach, the utility of an output can be determined by computing how much QoS will be sacrificed if the execution of the output is deferred. A second, feedback-based approach continuously observes the performance of the system and dynamically reassigns priorities to outputs, properly increasing the priorities of those that are not doing well and decreasing priorities of the applications that are already in their good zones.

Putting it all together. Because of the large scale, highly dynamic nature of the system, and granularity of scheduling, searching for optimal scheduling solutions is clearly infea-

figure=e:/vldb/095/slide7.eps,width=

Fig. 7. Execution overhead

sible. Aurora therefore uses heuristics to simultaneously address real-time requirements and cost reduction by first assigning priorities to select individual outputs and then exploring opportunities for constructing and processing tuple trains.

We now describe one such heuristic used by Aurora. Once an output is selected for execution, Aurora will find the first downstream box whose queue is in memory (note that for a box to be schedulable, its queue must be nonempty). Going upstream, Aurora will then consider other boxes until it either considers a box whose queue is not in memory or runs out of boxes. At this point there is a sequence of boxes (i.e., a *superbox*) that can be scheduled one after another.

In order to execute a box, Aurora contacts the storage manager and asks that the queue of the box be pinned to the buffer throughout the box's execution. It then passes the location of the input queue to the appropriate box processor code, specifies how many tuples the box should process, and assigns it to an available worker thread.

Scheduler performance. We now consider some simple measurements of our Aurora prototype. We built an Aurora application (a query network) in response to an advanced data dissemination problem that the Mitre Corporation [3] was working on. The resulting network contained approximately 40 boxes, and we ran it against a simulated input of 50,000 tuples. The tuples represented position reports of enemy units, and the Aurora outputs corresponded to different consumers of this data. Each consumer had different requirements (QoS). For this network, which had a depth of about four, running a scheduler that uses both superbox and tuple train scheduling we were able to process about 3200 boxes per second, which produced about 830 tuples per second at the outputs.

To illustrate the effectiveness of our scheduling approaches, we ran a set of experiments that measured the scheduling overhead that was incurred in running the above 40-box network with the same inputs as before. The results are shown in Fig. 7. In this figure, it can be seen that the time spent in the scheduler can be reduced by a factor of 0.48 when we shift from a box-at-a-time scheduling discipline to using tuple trains. Further, adding a simple version of superbox scheduling decreases the time spent in the scheduler by an additional factor of 0.43.

The figure further shows that the overall execution costs are also reduced. This is because execution costs include such things as the cost of making a box call, the cost of managing the tuple queues and their associated currency pointers, and the cost of mapping scheduled actions to the pool of worker threads.

4.4 Introspection

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

Static analysis. The goal of static analysis 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 above, each box b in an Aurora network has an expected tuple processing cost, $c(b)$, and a selectivity, $s(b)$. If we also know the expected rate of tuple production $r(d)$ from each data source δ, 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.,

$$1/c(b_i) \geq r(d_i) \ .$$

We can then calculate the output data rate from b_i as:

$$\min(1/c(b_i), r(d_i)) \times s(b_i)$$

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

$$C \times H < \textit{min\_cap}$$

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.

Dynamic analysis. Even if the system has sufficient resources to execute a given Aurora network under expected conditions, unpredictable, long-duration spikes in input rates may deteriorate performance to a level that renders the system useless.

Our technique for detecting an overload relies on the use of delay-based QoS information. Aurora timestamps all tuples from data sources as they arrive. Furthermore, all Aurora operators preserve the tuple timestamps as they produce output tuples (if an operator has multiple input tuples, then the earlier timestamp is preserved). When Aurora delivers an output tuple to an application, it checks the corresponding delay-based QoS graph (Fig. 4a) for that output to ascertain that the delay is at an acceptable level (i.e., the output is in the *good* zone). If enough outputs are observed to be outside of the good zone, this is a good indication of overload.

4.5 Load shedding

When an overload is detected as a result of static or dynamic analysis, Aurora attempts to reduce the volume of Aurora tuple processing via *load shedding*. The naïve approach to load

shedding involves dropping tuples at random points in the network in an entirely uncontrolled manner. This is similar to dropping overflow packets in packet-switching networks [32] and has two potential problems: (1) overall system utility might be degraded more than necessary and (2) application semantics might be arbitrarily affected. In order to alleviate these problems, Aurora relies on QoS information to guide the load-shedding process. We now describe two load-shedding techniques that differ in the way they exploit QoS.

Load shedding by dropping tuples. The first 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 tuples 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 (without assuming the availability of a priori knowledge of the tuple delays or frequency distribution of values). On the other hand, if load shedding is triggered as a result of dynamic analysis, we can also use delay-based QoS graphs.

We use a greedy algorithm to perform load shedding. Let us initially describe the static load-shedding algorithm driven by drop-based QoS graphs. 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 QoS curve has a smaller negative slope at that point. This horizontal difference gives us an indication of the output tuples to drop (i.e., the selectivity of the drop box to be inserted) that would result in the minimum decrease in the overall QoS.[2] 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. Meanwhile, we can calculate the amount of recovered resources. If the system resources are still not sufficient, then we repeat the process.

For the run-time case, the algorithm is similar except that we can use delay-based QoS graphs to identify the problematic outputs, i.e., the ones that are beyond their delay thresholds, and we repeat the load-shedding process until the latency goals are met.

In general, there are two subtleties in dynamic load shedding. First, drop boxes inserted by the load shedder should be among the ones that are given higher priority by the scheduler. Otherwise, load shedding will be ineffective in reducing the load of the system. Therefore, the load shedder simply does not consider the *inactive* (i.e., low-priority) outputs, which are indicated by the scheduler. Second, the algorithm tries to move the drop boxes as close to the sources as possible to discard tuples before they redundantly consume any resources. On the other hand, if there is a box with a large existing queue, it makes sense to *temporarily* insert the drop box at that point rather than trying to move it upstream closer toward the data sources.

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

Semantic load shedding by filtering tuples. The load-shedding scheme described above effectively reduces the amount of Aurora processing by dropping *randomly selected* tuples 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 tuples on application semantics. Semantic load shedding addresses this limitation by using value-based QoS information, if available. Specifically, semantic load shedding drops tuples in a more controlled way, i.e., it drops less important tuples, rather than random ones, using filters.

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 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). In an alternative strategy, *forward interval propagation*, Aurora starts from an output and goes upstream until it encounters a split point (or reaches the source). It then *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 it reaches 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 is shed.

The details of the algorithms that determine where to insert drop boxes and how selective they should be are not discussed here. This will be the topic of a future paper.

5 SQuAl: the Aurora query algebra

The Aurora [S]tream [Qu]ery [Al]gebra (SQuAl) supports seven operators that are used to construct Aurora networks (queries). These are analogous to operators in the relational algebra; however, they differ in fundamental ways to address the special requirements of stream processing. The current design has been driven by real example queries from several stream processing applications. This section will describe

[2] *Drop* is a system-level operator that drops tuples randomly from a stream at some specified rate. Users cannot use it explicitly. Thus, we do not include it in our discussion of the Aurora query algebra, SQuAl, which follows.

these operators in some detail as well as the rationale for their design.

5.1 The model

A stream is an append-only sequence of tuples with uniform type (schema). In addition to application-specific data fields $A_1, \ldots, A_n$, each tuple in a stream has a timestamp (ts) that specifies its time of origin within the Aurora network and is used for Quality-Of-Service (QoS) calculations (while hidden from queries).[3] In summary, a stream type has the form

$(\mathbf{TS}, A_1, \ldots, A_n)$

and stream tuples have the form

$(ts, v_1, \ldots, v_n)$

For notational convenience, we will describe a tuple in terms of its type and values as seen by queries:

$(A_1 = v_1, \ldots, A_n = v_n)$

adding the implicit QoS timestamp attribute ($\mathbf{TS} = ts$) only when it is relevant to the discussion.

It is common practice to assume that tuples arrive in an order that corresponds to a monotonically increasing data value such as a counter or time. This assumption simplifies the definition of a window, which can be defined as a range over the ordered value. However, no arrival order is assumed in the Aurora data model. This decision is motivated by the following observations:

1. The Aurora network might not be able to guarantee ordered delivery – for example, when multiple streams are merged.
2. Aurora permits windowing on *any* attribute (not just timestamps and tuple counts). Naturally, this means that windows might be defined on attributes for which there is some disorder.
3. By tolerating disorder, we gain latitude for producing outputs out of order, allowing us to service high-priority tuples first.

While some operators are *order-agnostic* (*Filter*, *Map*, *Union*) others are *order-sensitive* (*BSort*, *Aggregate*, *Join*, *Resample* in that they can only be guaranteed to execute with finite buffer space and in finite time if they can assume some ordering (with bounded disorder) over their input streams. Therefore, SQuAl's order-sensitive operators require *order specification* arguments that indicate the expected tuple arrival order. We discuss order specifications further in Sect. 5.2.2.

5.2 The operators

We divide the discussion of SQuAl operators into two sections. In Sect. 5.2.1, we describe SQuAl's three order-agnostic operators (*Filter*, *Map*, and *Union*). Then in Sect. 5.2.2, we describe SQuAl's four order-sensitive operators (*BSort*, *Aggregate*, *Join*, and *Resample*). We close the section with two example queries that illustrate the use of these operators.

It should be noted that much like the relational algebra, we make no attempt to provide a minimal operator set. Operator redundancy enables opportunities for optimization. Further, given the extensible nature of our *Map* and our *Aggregate* operators, this operator set is Turing complete.

5.2.1 Order-agnostic operators

In this section, we present the three SQuAl operators that can always process tuples in the order in which they arrive. *Filter* is similar to relational selection but can filter on multiple predicates and can route tuples according to which predicates they satisfy. *Map* is similar to relational projection but can apply arbitrary functions to tuples (including user-defined functions), and *Union* merges two or more streams of common schema. Each operator is described assuming input tuples of the form

$t = (\mathbf{TS} = ts, A_1, = v_1, \ldots, A_k = v_k)$

where **TS** is t's QoS timestamp attribute and $A_1, \ldots, A_k$ are t's value attributes.

5.2.1.1 Filter

Filter acts much like a case statement and can be used to route input tuples to alternative streams. It takes the form

Filter$(P_1, \ldots, P_m)(S)$

such that $P_1, \ldots, P_m$ are predicates over tuples on the input stream, S. Its output consists of $m+1$ streams, $S_1, \ldots, S_{m+1}$, such that for every input tuple, t, t is output to stream S_i if $(i = m + 1$ or $P_i(t))$ *and* $\forall j < i(\neg P_j(t))$. In other words, the tuples on the input stream that satisfy predicate P_1 are output on the first output stream, tuples on the input stream that satisfy predicate P_2 (but not P_1) are output on the second output stream, and so on. (The $(m+1)^{\text{st}}$ stream contains all tuples satisfying none of the predicates.) Note that because *Filter* outputs the tuples from its input, output tuples have the same schema and values as input tuples, including their QoS timestamps.

5.2.1.2 Map

Map is a generalized projection operator that takes the form

Map $(B_1 = F_1, \ldots, B_m = F_m)(S)$

such that $B_1, \ldots, B_m$ are names of attributes and $F_1, \ldots, F_m$ are functions over tuples on the input stream, S. Map outputs a stream consisting of tuples of the form

$(\mathbf{TS} = t.\mathbf{TS}, B_1 = F_1(t), \ldots, B_m = F_m(t))$

for each input tuple, t. Note that the resulting stream can (and in most cases will) have a different schema than the input stream, but the timestamps of input tuples are preserved in corresponding output tuples.

[3] Every input tuple to Aurora is tagged with this timestamp upon entry to the Aurora network. Every tuple generated by an Aurora operator is tagged with the timestamp of the *oldest* tuple that was used in generating it.

5.2.1.3 Union

Union is used to merge two or more streams into a single output stream. This operator takes the form

$Union\ (S_1, \ldots, S_n)$

such that $S_1, \ldots, S_n$ are streams with common schema. *Union* can output tuples in any order, though one obvious processing strategy is to emit tuples in the order in which they arrive regardless of which input stream they arrive on. Note that because *Union* outputs all input tuples it processes, output tuples have the same schema and values as input tuples including their QoS timestamps.

5.2.2 Order-sensitive operators

SQuAl's other four operators can only be guaranteed to execute with finite buffer space and in finite time if they can assume some ordering (potentially with bounded disorder) over their input streams. These operators require *order specification* arguments that describe the tuple arrival order they expect. Order specifications have the form

$Order\ (On\ A,\ Slack\ n,\ GroupBy\ B_1, \ldots, B_m)$

such that $A, B_1, \ldots, B_m$ are attributes and n is a nonnegative integer.[4] When *Slack* = 0 and the *GroupBy* set is empty,[5] an order specification indicates that tuples are expected to arrive in ascending order on A and all tuples that arrive *out-of-order* (i.e., whose value for A is less than that for some tuple that appears before it in the stream) are ignored. A nonzero slack specification relaxes this constraint somewhat, making for fewer tuples being discarded at the expense of some latency in computation. More specifically, the effect of nonzero *Slack* and nonempty *GroupBy* arguments is to relax the determination of what constitutes an out-of-order tuple. A tuple, t, is "*out of order by n wrt A in S*" if there are more than n tuples, u, preceding t in S such that $u.A > t.A$. Alternatively, we say that a tuple t is "*out of order wrt O in S*" (for O = *Order (On A, Slack n, GroupBy $B_1, \ldots, B_m$)* if t is out of order by n wrt A in S. Thus, an operator that assumes an order specification, O, over its input stream discards all tuples in its input stream, S, that are out of order wrt O in S.

GroupBy arguments $(B_1, \ldots, B_m)$ implicitly partition the input stream according to each tuple's values for $B_1, \ldots, B_m$ and considers the order of each partition independently. An operator assuming the order specification "*Order (On A, Slack n, GroupBy $B_1, \ldots, B_m$)*" discards all tuples, t, in S that are out of order by n wrt A in the partition of S consisting of all tuples whose values for $B_1, \ldots, B_m$ are $t.B_1, \ldots, t.B_m$, respectively.

In the next section, we describe SQuAl's remaining four operators: *BSort* is an approximate sort operator with semantics equivalent to a bounded pass bubble sort; *Aggregate* applies a window function to sliding windows over its input stream; *Join* is a binary operator that resembles a *band join* [28], but applied to infinite streams; and *Resample* is an interpolation operator used to align streams. Each of these operators requires an order specification for each of its input streams and ignores tuples on these input streams that are out of order according to these specifications. Thus when an operator is described as performing some action with "every tuple t" on some input, it should be assumed that this does not refer to the out-of-order tuples that are ignored.

5.2.2.1 BSort

BSort is an approximate sort operator that takes the form

BSort (Assuming O) (S)

such that *O = Order (On A, Slack n, GroupBy $B_1, \ldots, B_m$)* is a specification of the assumed ordering over the output stream. While a complete sort is not possible over an infinite stream with finite time or space, *BSort* performs a buffer-based approximate sort equivalent to n passes of a bubble sort where *slack* = n. This is achieved by maintaining a buffer of $n + 1$ tuples while processing the input stream. Every time the buffer fills, a tuple in the buffer with minimal value for A is evicted from the buffer and emitted as output. For example, suppose tuples in a stream have A values:

$1, 3, 1, 2, 4, 4, 8, 3, 4, 4$

Evaluating *BSort* with *Slack* = 2 results in the behavior illustrated in the diagram in Fig. 8. The top row of this diagram shows a timeline and below it the input stream with tuples arriving to the *BSort* operator in order from left to right. Also shown at each time are the contents of the three-tuple buffer maintained by *BSort*. Each arriving tuple is added to the buffer and evicted (as shown in the output stream). Observe that the output stream is equivalent to the sequence resulting from executing two passes of a bubble sort over the input stream (more accurately the first eight values of this sequence, as the remaining two values are still in the buffer). Note also that because *BSort* outputs exactly the tuples it processes (though potentially in different order), output tuples have the same schema and values as input tuples, including their QoS timestamps.

5.2.2.2 Aggregate

Aggregate applies "window functions" to sliding windows over its input stream. This operator has the form

Aggregate (F, Assuming O, Size s, Advance i) (S)

such that F is a "window function" (either a SQL-style aggregate operation or a Postgres-style user-defined function), *O = Order (On A, Slack n, GroupBy $B_1, \ldots, B_m$)* is an order specification over input stream S, s is the size of the window (measured in terms of values of A), and i is an integer or predicate that specifies how to advance the window when it slides.[6] This operator outputs a stream of tuples of the form

$(\mathbf{TS} = ts, A = a, B_1 = u_1, \ldots, B_m = u_m) + +(F(W))$

[4] A can be the "virtual" attribute, *Tup#*, which maps consecutive tuples in a stream to consecutive integers. This allows for an order to be specified on arrival time, which is useful when one wants to define a window based on a tuple count.

[5] These are the default values for *Slack* and *GroupBy* taken when these clauses are omitted.

[6] UDFs have the form

```
Agg (init, incr, final)
```

figure=e:/vldb/095/slide8.eps

Fig. 8. An example application of BSort

figure=e:/vldb/095/slide9.eps

Fig. 9. An example trace of aggregate

such that W is a "window" (i.e., subsequence) of tuples from the input stream with values of A between a and $a + s - 1$ (inclusive) and values for $B_1, \ldots, B_m$ of $u_1, \ldots, u_m$, respectively, and ts is the smallest of the timestamps associated with tuples in W. We use the notation "++" to denote the concatenation of two tuples. Thus it is assumed that the function F returns a tuple of aggregate computations (perhaps with only one field) and that this tuple is concatenated to a tuple consisting of fields that identify the window over which the computation took place ($B_1, \ldots, B_m$, and A).

As a simple example, suppose one wishes to compute an hourly average price (*Price*) per stock (*Sid*) over a stream of stock quotes that is known to be ordered by the time the quote was issued (*Time*). This can be expressed as:

Aggregate [Avg (Price),
Assuming Order (On Time, GroupBy Sid),
Size 1 hour,
Advance 1 hour]

which will compute an average stock price for each stock ID for each hour. This is illustrated in Fig. 9 where 11 input stock price tuples with schema

(Sid, Time, Price)

are shown on the left. Each window computation is summarized by a box in the middle of the figure, with each window uniquely identified by a stock ID and an hour-long time interval over which the average price is computed. At the right of the figure are the emitted tuples with schema

(Sid, Time, AvgPrice)

The *Time* field in result tuples specifies the initial time of the interval whose calculation is shown in the tuple, and the *AvgPrice* field shows the average price for the indicated stock over that interval.

The order specification argument to *Aggregate* makes it possible to cope with disorder in the input stream. For example, suppose the input stream of Fig. 9 had the arrival order shown below where the 1:45 IBM quote arrives late:

1. (MSF,1:00,$20)
2. (INT,1:00,$16)
3. (IBM,1:00,$24)
4. (IBM,1:15,$20)
5. (IBM,1:30,$23)
6. (MSF,1:30,$24)
7. (INT,1:30,$12)
9. (IBM,2:00,$17)
10. (INT,2:00,$16)
11. (MSF,2:00,$22)
8. (IBM,1:45,$13)

...

If the order specification associated with *Aggregate* has a slack argument of 1 or more, then the late-arriving tuple will still be accounted for in the 1:00–1:59 IBM computation and the output of *Aggregate* will be as in Fig. 9. This is because there is only one tuple in the partition of the stream defined by *Sid* = `IBM`, whose value for *Time* exceeds the out-of-order tuple. Hence a slack setting of 1 suffices to ensure that this tuple is included in the computation.

In general, the *Aggregate* expression

Aggregate (F, Assuming Order (On A, Slack n, GroupBy $B_1, \ldots, B_m$), Size s, Advance i)(S)

is equivalent to the composition of an *Aggregate* with 0 slack and a buffered sort where *Slack* is n:

Aggregate (F, Assuming Order (On A, Slack 0, GroupBy $B_1, \ldots, B_m$), Size s, Advance i)
(BSort (Assuming Order (On A, Slack n, GroupBy $B_1, \ldots, B_m$))(S))

This is potentially a useful identity for query optimization, as the result of a prior sort might be used by multiple queries that apply window functions to the same stream, assuming an ordering over the same attribute. However, it is important to point out that the evaluation of *Aggregate* **does not** require first sorting its input stream.[7]

Aside from disorder, another complication with regard to stream aggregation is *blocking*: waiting for lost or late tuples to arrive in order to finish window calculations. Given the real-time requirements of many stream applications, it is essential that it be possible to "time out" aggregate computations, even if this happens at the expense of accuracy. For example, suppose we are monitoring the geographic center of mass for a military platoon. Soldiers may turn off their communication units, go out of range, or merely be located in a region with very high network latency. In such circumstances, the center-of-mass computation might block while

such that `init` is a function called to initialize a computation state whenever a window is opened, `incr` is called to update that state whenever a tuple arrives, and `final` is called to convert the state to a final result when the window closes.

[7] An algorithm for *Aggregate* that does not require sorting its input simply delays emitting result tuples according to the slack specification, allowing for late-arriving tuples to contribute to the window computation.

waiting for missing data, and it is likely beneficial to output an approximate center-of-mass calculation before all relevant sensor readings have been received.

To counter the effects of blocking, *Aggregate* can accept an optional *Timeout* argument, as in:

Aggregate (F*, Assuming* O*, Size* s*, Advance* i*, Timeout* t*)*

where t is some measure of time. When *Aggregate* is declared with a timeout, each window's computation is timestamped with the local time when the computation begins. A window's computation then times out if a result tuple for that window has not been emitted by the time that the local time exceeds the window's initial time $+t$.[8] As a result of a timeout, a result tuple for that window gets emitted early and all tuples that arrive afterwards and that would have contributed to the computation had it not timed out are ignored.

5.2.2.3 Join

Join is a binary join operator that takes the form

Join (P*, Size* s*, Left Assuming* O_1*, Right Assuming* O_2*)*
(S_1, S_2)

such that P is a predicate over pairs of tuples from input streams S_1 and S_2, s is an integer, and O_1 (on some numeric or time-based attribute of S_1, A) and O_2 (on some numeric or time-based attribute of S_2, B) are specifications of orderings assumed of S_1 and S_2, respectively. For every in-order tuple t in S_1 and u in S_2, the concatenation of t and u ($t++u$) is output if $|t.A - u.B| \leq s$ and P holds of t and u. The QoS timestamp for the output tuple is the minimum timestamp of t and u.

As a simple example, suppose that we are processing two streams, X and Y, consisting of soldier position reports for two respective platoons. Suppose each stream has schema

(Sid, Time, Pos)

such that *Sid* is the soldier ID, *Pos* is the position report,[9] and *Time* is the time the report was issued, and that we wish to detect all pairs of soldiers from respective platoons who report identical positions within 10 min of each other. This could be expressed as:

Join (P*, Size 10 min, Assuming Left* O*, Assuming Right* O*)*
(X, Y)

such that:

- O = *Order (On Time)* and
- $P(x, y) \Leftrightarrow x.pos = y.pos$

The execution of this query on example readings from X and Y is shown in Fig. 10. A simple variation of the symmetric hash join [29] (that prunes internal tables of tuples that, because of the assumed orderings, are guaranteed not to join with tuples that have yet to arrive from the opposite stream) produces the correct join result, regardless of the interleaving of input streams (i.e., independent of the arrival times of tuples).

figure=e:/vldb/095/slide10.eps,width=

Fig. 10. An example trace of join

figure=e:/vldb/095/slide11.eps,width=

Fig. 11. An example trace of resample

As with *Aggregate*, *Join* can be expressed so as to tolerate disorder in input streams. Thus the following identity holds:

Join (P*, Size* s*, Assuming Left* O_1*, Assuming Right* O_2*)*
$(S_1, S_2) =$
Join (P*, Size* s*, Assuming Left* O_1'*, Assuming Right* O_2'*)*
(BSort (Assuming O_1*)* (S_1) *BSort (Assuming* O_2*)* (S_2)*)*

such that:

- O_1 = *Order (On* A*, Slack* n_1*, GroupBy* $A_1, \ldots, A_{m1}$*)*,
- O_2 = *Order (On* B*, Slack* n_2*, GroupBy* $B_1, \ldots, B_{m2}$*)*,
- O_1' = *Order (On* A*, GroupBy* $A_1, \ldots, A_{m1}$*)*, and
- O_2' = *Order (On* B*, GroupBy* $B_1, \ldots, B_{m2}$*)*

As with *Aggregate*, *Join* need not sort its inputs in order to process disordered streams but can instead delay pruning tuples to account for slack.

Join also permits one or both of its inputs to be static tables. A static table is a special case of a window on a stream that is infinite in size.

5.2.2.4 Resample

Resample is an asymmetric, semijoin-like synchronization operator that can be used to align pairs of streams. This operator takes the form

Resample (F*, Size* s*, Left Assuming* O_1*, Right Assuming* O_2*)*
(S_1, S_2)

such that F is a "window function" over S_1 (defined in the same fashion as the window function arguments to *Aggregate*), s is an integer, A is an attribute over S_1 (upon which S_1 is assumed to be ordered), and O_1 (on some numeric or time-based attribute of S_1, A) and O_2 (on some numeric or time-based attribute of S_2, B) are specifications of orderings assumed of S_1 and S_2, respectively. For every tuple, t, from S_1, tuple

$(B_1 : u.B_1, \ldots, B_m : u.B_m, A : t.A) ++ F(W(t))$

is output such that:

$W(t) =$
$\{u \in S_2 | u$ *in order* $wrt O_2$ *in* $S_2 \wedge |t.A - u.B| \leq s\}$

Thus, for every tuple in S_1, an interpolated value is generated from S_2 using the interpolation function, F, over a window of tuples of size $2s$.

[8] The actual time a window computation times out may exceed the declared timeout if the *Aggregate* box is not scheduled immediately following the declared timeout.

[9] To keep the example simple, we are assuming position values are integers. Obviously in a more realistic scenario, the representation of position and the calculation of distance will be more complicated.

figure=e:/vldb/095/slide12.eps,width=

Fig. 12. Query 1

figure=e:/vldb/095/slide13.eps,width=

Fig. 13. An example trace of query 1

figure=e:/vldb/095/slide14.eps,width=

Fig. 14. A streamlined version of query 1

figure=e:/vldb/095/slide15.eps,width=

Fig. 15. An example trace of query

As a simple example, suppose that X is a stream of position reports for soldiers from platoon X emitted at irregular time intervals and H is a stream of "heartbeats" that are emitted every 15 min. Suppose a simple interpolation of X is desired whereby the position of every soldier is estimated at some time t, by applying some function F over the window of all of the soldiers' position reports that are within 10 min of t.[10] This can be expressed as:

Resample (F*, Size 10, Left Assuming* O_1*, Right Assuming*O_2*)*
(H, X)

such that O_1 = *Order (On Time)* and O_2 = *Order (On Time, GroupBy Sid)*. A sample trace of this operator above is shown in Fig. 11 in terms of a heartbeat stream (with schema *Time* and values in 15-min intervals) and a soldier position stream (as before, with schema *Sid, Time, Pos*. Note that tuples are emitted in the order in which their computations conclude. Therefore, even though a position report for the soldier with *Sid* = 1 (tuple #1) arrives before one from the soldier with *Sid* = 2 (tuple #2), the first interpolated result comes from the latter soldier as this calculation concludes with the arrival of the first tuple from this soldier with a timestamp that is later than 2:10 (tuple #3).

5.3 Query examples

We now describe two queries that demonstrate SQuAl, motivated by conversations with Mitre Corporation concerning military logistics applications. For these queries we assume that inputs are streams of soldier position reports with schema *Sid, Time, Pos*, as in previous examples. Once again, we simplify the examples by assuming integer-valued positions and also assume integer-valued timestamps.

Query 1: *Produce an output whenever m soldiers are across some border k at the same time (with border crossing detection determined by the predicate Pos $\geq k$).*

The SQuAl expression of this query is shown in Fig. 12. The first box of the query filters position reports for those that reveal that a soldier is across the border. Then, an aggregation on the resulting stream (assuming an ordering on *Time*) produces a count of soldiers reporting a position beyond border k at each point in time. Finally, a filter identifies those times where the soldier count exceeds m.

An *example* trace of this query on a stream of soldier reports (assuming $k = 30$, $m = 3$, and n(*Slack*) = 1) is shown in Fig. 13. It can be seen from this example that outputs are not produced immediately when m soldiers are recognized as being across the border. For example, upon the processing of the fifth tuple to *Aggregate* it is known that a tuple should be output indicating that there are three soldiers across the border at $Time = 2$. However, this tuple does not get output until the seventh tuple to *Aggregate* gets processed, as it is only then that the window for $Time = 2$ is closed.

The alternative query shown in Fig. 14 with a C-like user-defined window function eliminates this latency. This query first sorts all position reports showing soldiers across the border on *Time*. The following *Aggregate* box then maintains a window of exactly m tuples, and whenever **every** tuple in this window has the same value for *Time* (indicating that there are m soldier reports at that time for soldiers across the border), a tuple is output reporting the time. The trace of this query on the same input stream as in Fig. 12 is shown in Fig. 15 and shows that this query will output k tuples any time there are $n + k - 1$ soldiers across the border (e.g., three tuples are emitted for $Time = 2$ because five soldiers are across the border at this time.) The additional tuples can be removed by adding an additional *Aggregate* box whose window has a size of two tuples and that emits a tuple only when it is the newest tuple in the window and with a different value for time than the oldest tuple in the window.

figure=e:/vldb/095/slide16.eps,width=

Fig. 16. Query 2

figure=e:/vldb/095/slide17.eps,width=

Fig. 17. An example trace of query 2

Query 2: *Compute the center of mass of every soldier's position for every timestamp. Alert when it differs from the previous reading by 100 m. Assume that it is not worth waiting more than t seconds for soldier position reports when calculating center of mass for a given time.*

The SQuAl expression of this query is shown in Fig. 16. The first box calculates the center of mass (using a user-defined function) over all position reports assuming an ordering on *Time*. Note that the *Timeout* argument to this box ensures that a center-of-mass calculation never requires waiting more than t seconds from the time of the first soldier report to the last. A second *Aggregate* box maintains a window of size two tuples, so as to compare the most recent two center-of-mass calculations to see if they differ by 100 or more. The user-defined window function, F, performs this calculation and outputs a flag (`Diff`) which is set to `TRUE` if the last center-of-mass calculation differs by the previous one by 10 m or more. The final *Filter* box returns the center-of-mass calculations and the associated timestamps for those that have this flag set. An example trace with $n(Slack) = 2$ is shown in Fig. 17.

[10] F could, for example, calculate a weighted average of positions reported, with weights determined by the difference between the time of the reports and t. For this example, we assume a much simpler function that simply reports the average position reported by tuples in the window.

6 Related work

Query indexing [4] is an important technique for enhancing the performance of large-scale filtering applications. In Aurora, this would correspond to a merge of some inputs followed by a fan-out to a large number of filter boxes. Query indexing would be useful here, but it represents only one Aurora processing idiom.

As in Aurora, active databases [24,25] 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 queries over streams or the conversion of these queries into monitored conditions. There has also been extensive work on making active databases highly scalable (e.g., [14]). 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., [5,16,29]) address efficient query execution in unpredictable and dynamic environments by revising the query execution plan as the characteristics of incoming data changes. Of particular relevance is the Eddies work [5]. Unlike traditional query processing where every tuple from a given data source gets processed in the same way, each tuple processed by an Eddy is dynamically routed to operator threads for partial processing, with the responsibility falling upon the tuple to carry with it its processing state. Recent work [19] extended Eddies to support the processing of queries over streams, mainly by permitting Eddies systems to process multiple queries simultaneously and for unbounded lengths of time. The Aurora architecture bears some similarity to that of Eddies in its division of a single query's processing into multiple threads of control (one per query operator). However, queries processed by Eddies are expected to be processed in their entirety; there is neither the notion of load shedding nor QoS.

A special case of Aurora processing is as a continuous query system. A system like NiagaraCQ [9] is concerned with combining multiple data sources in a wide area setting, while we are initially focusing on the construction of a general stream processor that can process very large numbers of streams. Further work on continuous queries by Viglas and Naughton [30] discusses rate-based query optimization for streaming wide-area information sources in the context of NiagaraCQ.

Recent work on stream data query processing architectures shares many of the goals and target application domains with Aurora. The Fjords architecture [18] combines querying of push-based sensor sources with pull-based traditional sources by embedding the pull/push semantics into queues between query operators. It is fundamentally different from Aurora in that operator scheduling is governed by a combination of schedulers specific to query threads and operator-queue interactions. Tribeca [28] is an extensible, stream-oriented data processor designed specifically for supporting network traffic analysis. While Tribeca incorporates some of the stream operators and compile-time optimizations supported by Aurora, it does not address scheduling or load-shedding issues and does not have the concept of ad hoc queries. Like Aurora, the STREAM project [7] attempts to provide comprehensive data stream management and processing functionality. Even though both systems address many common data and resource management issues, the proposed approaches and solutions are different due to different assumptions and performance criteria. In particular, Aurora drives all resource management decisions, such as scheduling, storage management, and load shedding, based on various QoS specifications, whereas Stream does not have a notion of QoS. Another high-level difference involves the specification of stream processing requirements: Stream uses a variant of SQL, whereas Aurora assumes more direct, workflow-style specification of queries. A more detailed comparison of these systems is beyond the scope of this paper and can be found elsewhere [21].

The SEQ model [27], developed for sequence databases, provides a set of positional and record-oriented operators. Even though some SEQ operators demonstrate strong resemblance to Aurora operators (e.g., our *Filter* and *Map* operators can be encoded using SEQ's *Transform* operator), there are also significant differences. First, due to the conceptual differences between stored data sequences and online data streams, SEQ operators do not deal with issues such as one-time occurrence of tuples, late delivery, or tuple ordering issues; they assume that the sequence data is finite and readily available from persistent storage. Second, SEQ does not provide any binary windowed operators such as Aurora's *Join*. Instead, positional binary operators in SEQ act only on tuples at matching positions of two sequences. Therefore, notions such as Timeout and Slack do not exist in SEQ. Finally, SEQ does not provide any operator that works across streams in a composite stream, such as Aurora's *XSection*.

The Chronicle Data Model [24] defined a restricted view definition and manipulation language over append-only sequences (so-called *chronicles*). The operators used in this model are sequence counterparts of relational operators such as *Selection*, *Projection*, *Join*, *Union*, *Difference*, *GroupBy*, and *CrossProduct*. *Selection* corresponds to our *Filter*, whereas *Projection* is a special case of Aurora's general-purpose *Map* operation. Unlike our *Join* operator, Chronicle's join predicate is restricted to equality on the sequencing. As a result, windowed joins are not possible. *GroupBy* is similar to our *Tumble* except that, as in SEQ, Chronicle Data Model is not concerned with blocking issues and hence does not have *Timeout* and *Slack* concepts. The composite stream concept used in Aurora somewhat corresponds to Chronicle's group concept, which, however, is mainly used to restrict binary operators to act on chronicles of the same group.

Our work is also relevant to materialized views [13], which are essentially stored continuous queries that are reexecuted (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, thus making it much easier to incrementally materialize the view. Also, 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.

Our work is likely to benefit from and contribute to the considerable research on temporal databases [23], main-

figure=e:/vldb/095/slide18.eps

Fig. 18. Aurora GUI

memory databases [11], and real-time databases [17,23]. These studies commonly assume an HADP model, whereas Aurora proposes a DAHP model that builds streams as fundamental Aurora objects. 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. The Aurora notion of QoS extends the soft and hard deadlines used in real-time databases to general utility functions. Furthermore, real-time databases associate deadlines with individual transactions, whereas Aurora associates QoS curves with outputs from stream processing and, thus, must support continuous timing requirements. Relevant research in workflow systems (e.g., [20]) primarily focused on organizing long-running interdependent activities but did not consider real-time processing issues.

There has been extensive research on scheduling tasks in real-time and multimedia systems and databases [22,23]. The proposed approaches are commonly deadline driven, i.e., at each scheduling point, the task that has the earliest deadline or the one that is expected to provide the highest QoS (e.g., throughput) is identified and scheduled. In Aurora, such an approach is not only impractical because of the sheer number of potentially schedulable tasks (i.e., tuples) but is also inefficient because of the implicit assumption that all tasks are memory-resident and are scheduled and executed in their entirety. To the best of our knowledge, however, our train scheduling approach is unique in its ability to reduce overall execution costs by exploiting intra- and interbox nonlinearities described here.

The work of [29] takes a scheduling-based approach to query processing; however, they do not address continuous queries, are primarily concerned with data rates that are too slow (we also consider rates that are too high), and only address query plans that are trees with single outputs. Chain [6] is an operator scheduling algorithm for stream processing streams. Chain's goal is to minimize run-time memory consumption; it does not address user-level performance metrics such as latency, whereas our goal is to maximize QoS.

The congestion control problem in data networks [32] is relevant to Aurora and its load-shedding mechanism. Load shedding in networks typically involves dropping individual packets randomly, based on timestamps, or using (application-specified) priority bits. Despite conceptual similarities, there are also some fundamental differences between network load shedding and Aurora load shedding. First, unlike network load shedding, which is inherently distributed, Aurora is aware of the entire system state and can potentially make more intelligent shedding decisions. Second, Aurora uses QoS information provided by the external applications to trigger and guide load shedding. Third, Aurora's semantic load-shedding approach not only attempts to minimize the degradation in overall system utility but also quantifies the imprecision due to dropped tuples.

Aurora load shedding is also related to approximate query answering (e.g., [15]), data reduction, and summary techniques [8,12], where result accuracy is traded for efficiency. By throwing away data, Aurora bases its computations on sampled data, effectively producing approximate answers using data sampling. The unique aspect of our approach is that our sampling is driven by QoS specifications.

7 Implementation status

As of March 2003, we have a prototype Aurora implementation. The prototype has a Java-based GUI that allows construction and execution of Aurora networks. The current interface supports construction of arbitrary Aurora networks, specification of QoS graphs, stream-type inferencing, and zooming. Figure 18 is a screenshot of our current Aurora GUI. Users construct an Aurora network by simply dragging and dropping operators from the operator palette (shown on the left of the GUI) and connecting them. The small black boxes on the left of the drawing area represent the input ports, which are connected to the external stream sources. The small boxes on the right are output ports that connect to the applications. Figure 19 shows another screenshot that demonstrates an integrated queue-monitoring tool used to examine the queued tuples in the network. The smaller dark windows display the state of each arc in the network. The interface also supports pause functionality, which is used for debugging and performance monitoring.

The run-time system contains a scheduler, a rudimentary storage manager, and code to execute most of the boxes. We are currently experimenting with competing scheduling algorithms and extending the functionality of the storage manager to define and manage connection points. Aurora metadata is stored in a schema, which is stored in a Berkeley DB database. Aurora is functionally complete, and multibox networks can be constructed and run. However, there is currently no optimization and load shedding.

8 Conclusions and future work

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 presented the basic data stream model and architecture of Aurora, a DAHP system oriented toward monitoring applications. We argued that providing efficient support for these demanding applications requires not only critically revisiting many existing aspects of database design and implementation but also developing novel proactive data storage and processing concepts and techniques.

In this paper, we first presented the basic Aurora model and architecture along with the primitive building blocks for workflow processing. We followed with several heuristics for optimizing a large Aurora network. We then focused on run-time data storage and processing issues, discussing storage organization, real-time scheduling, introspection, and load shedding and proposed novel solutions in all these areas. Finally, we provided a detailed discussion of a new set of data stream processing operators.

figure=e:/vldb/095/slide19.eps

Fig. 19. Queue monitoring

We are currently investigating two important research directions. While the bulk of the discussion in this paper describes how Aurora works on a single computer, many stream-based applications demand support for distributed processing. To this end, we are working on a distributed architecture, Aurora*, that will enable operators to be pushed closer to the data sources, potentially yielding significantly improved scalability, energy use, and bandwidth efficiency. Aurora* will provide support for distribution by running a full Aurora system on each of a collection of communicating nodes. In particular, Aurora* will manage load by replicating boxes along a path and migrating a copy of this subnetwork to another more lightly loaded node. A subset of the stream inputs to the replicated network would move along with the copy. We are also extending our basic data and processing model to cope with missing and imprecise data values, which are common in applications involving sensor-generated data streams.

Acknowledgements. We would like to thank all of the participants of the "Stream Team" meeting held in October 2002 in Berkeley for their helpful suggestions for SQuAl. We would especially like to thank Jennifer Widom who suggested making our sorting operator lossless.

References

1. Informix White Paper. Time Series: The Next Step for Telecommunications Data Management.
2. Lojack.com (2003) http://www.lojack.com/
3. Mitre Corporation (2003) http://www.mitre.org/
4. Altinel M, Franklin MJ (2000) 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
5. Avnur R, Hellerstein J (2000) Eddies: continuously adaptive query processing. In: Proceedings of the 2000 ACM SIGMOD international conference on management of data, Dallas
6. Babcock B, Babu S, Datar M, Motwani R (2003) Chain: operator scheduling for memory minimization in stream systems. In: Proceedings of the international SIGMOD conference, San Diego
7. Babu S, Widom J (2001) Continuous queries over data streams. SIGMOD Record 30(3):109–120
8. Barbara D, DuMouchel W, Faloutsos C, Haas PJ, Hellerstein JM, Ioannidis YE, Jagadish HV, Johnson T, Ng RT, Poosala V, Ross KA, Sevcik KC (1997) The New Jersey Data Reduction Report. IEEE Data Eng Bull 20(4):3–45
9. Chen J, DeWitt DJ, Tian F, Wang Y (2000) NiagaraCQ: a scalable continuous query system for internet databases. In: Proceedings of the 2000 ACM SIGMOD international conference on management of data, Dallas
10. DeWitt DJ, Naughton JF, Schneider DA (1991) An evaluation of non-equijoin algorithms. In: Proceedings of the 17th international conference on very large data bases (VLDB), Barcelona
11. Garcia-Molina H, Salem K (1992) Main memory database systems: an overview. IEEE Trans Knowledge Data Eng 4(6):509–516
12. Gehrke J, Korn F, Srivastava D (2001) On computing correlated *Aggregate*s over continual data streams. In: Proceedings of the 2001 ACM SIGMOD international conference on management of data, Santa Barbara, CA
13. Gupta A, Mumick IS (1995) Maintenance of materialized views: problems, techniques, and applications. IEEE Data Eng Bull 18(2):3–18
14. Hanson EN, Carnes C, Huang L, Konyala M, Noronha L, Parthasarathy S, Park JB, Vernon A (1999) Scalable trigger processing. In: Proceedings of the 15th international conference on data engineering, Sydney
15. Hellerstein JM, Haas PJ, Wang HJ (1997) Online aggregation. In: Proceedings of the 1997 ACM SIGMOD international conference on management of data, Tucson
16. Ives ZG, Florescu D, Friedman M, Levy A, Weld DS (1999) An adaptive query execution system for data integration. In: Proceedings of the 1999 ACM SIGMOD international conference on management of data, Philadelphia
17. Kao B, Garcia-Molina H (1994) An overview of realtime database systems. In: Stoyenko AD (ed) Real time computing. Springer, Berlin Heidelberg New York
18. Madden S, Franklin MJ (2002) Fjording the stream: an architecture for queries over streaming sensor data. In: Proceedings of the 18th international conference on data engineering, San Jose
19. Madden SR, Shaw MA, Hellerstein JM, Raman V (2002) Continuously adaptive continuous queries over streams. In: SIGMOD, Wisconsin
20. Mohan C, Agrawal D, Alonso G, Abbadi AE, Gunther R, Kamath M (1995) Exotica: a project on advanced transaction management and workflow systems. SIGOIS Bull 16(1):45–50
21. Motwani R, Widom J, Arasu A, Babcock B, Babu S, Datar M, Manku G, Olston C, Rosenstein J, Varma R (2002) Query processing, approximation, and resource management in a data stream management system. Stanford University, Computer Science Department 2002-41, August 2002
22. Nieh J, Lam MS (1997) The design, implementation and evaluation of SMART: a scheduler for multimedia applications. In: Proceedings of the 16th ACM symposium on OS principles
23. Ozsoyoglu G, Snodgrass RT (1995) Temporal and real-time databases: a survey. IEEE Trans Knowledge Data Eng 7(4):513–532
24. Paton N, Diaz O (1999) Active database systems. ACM Comput Surv 31(1):63–103
25. Schreier U, Pirahesh H, Agrawal R, Mohan C (1991) Alert: an architecture for transforming a passive DBMS into an active DBMS. In: Proceedings of the 17th international conference on very large data bases, Barcelona
26. Sellis TK, Ghosh S (1990) On the multiple-query optimization problem. IEEE Trans Knowledge Data Eng 2(2):262–266
27. Seshadri P, Livny M, Ramakrishnan R (1995) SEQ: a model for sequence databases. In: Proceedings of the 11th international conference on data engineering, Taipei, Taiwan
28. Sullivan M, Heybey A (1998) Tribeca: a system for managing large databases of network traffic. In: Proceedings of the USENIX annual technical conference, New Orleans
29. Urhan T, Franklin MJ (2001) Dynamic pipeline scheduling for improving interactive query performance. In: Proceedings of the 27th international conference on very large data bases, Rome

30. Viglas S, Naughton JF (2002) Rate-based query optimization for streaming information sources. In: Proceedings of the ACM SIGMOD international conference on management of data, Madison, WI
31. Wilschut AN, Apers PMG (1991) Dataflow query execution in a parallel main-memory environment. In: Proceedings of the international conference on parallel and distributed information systems
32. Yang C, Reddy AVS (1995) A taxonomy for congestion control algorithms in packet switching networks. IEEE Netw 9(5):34–44