The Aurora and Medusa Projects

Stan Zdonik
Brown University
sbz@cs.brown.edu

Michael Stonebraker
MIT
stonebraker@lcs.mit.edu

Mitch Cherniack
Brandeis University
mfc@cs.brandeis.edu

Uğur Çetintemel
Brown University
ugur@cs.brown.edu

Magdalena Balazinska
MIT
mbalazin@lcs.mit.edu

Hari Balakrishnan
MIT
hari@lcs.mit.edu

Abstract

This document summarizes the research conducted in two interrelated projects. The Aurora project being implemented at Brown and Brandeis under the direction of Uğur Çetintemel, Mitch Cherniack, Michael Stonebraker and Stan Zdonik strives to build a single-site high performance stream processing engine. It has an innovative collection of operators, workflow orientation, and strives to maximize quality of service for connecting applications. A further goal of Aurora is to extend this engine to a distributed environment in which multiple machines closely co-operate in achieving high quality of service.

In contrast, the Medusa project being investigated at M.I.T. under the direction of Hari Balakrishnan and Michael Stonebraker is providing networking infrastructure for Aurora operations. In addition, Medusa is stressing distributed environments where the various machines belong to different organizations. In this case, there can be no common goal, and much looser coupling is needed. Medusa is working on an innovative agoric infrastructure in which various participants can co-operate in distributed streaming operations.

Finally, some have questioned whether specialized stream processing software is necessary. They speculate that conventional data base systems can adequately deal with the needs of stream-oriented applications. In order to answer this question one way or the other, both groups are working on a stream-oriented benchmark, called the Linear Road benchmark, which we intend to run on both specialized and conventional system infrastructure.

1 Introduction

Many people have pointed out the rationale for stream-oriented storage systems. Rather than repeat their observations, we want to note just two points here. Business intelligence is typically performed by extracting relevant data from operational systems, transforming it into a common representation, and then loading it into a data warehouse. Business analysts can then run data mining queries against the warehouse to find information of interest, on which to base changes to business practices. Business analysis through warehousing and Extract-Transform-and Load (ETL) techniques is widely used in large enterprises at the current time. However,

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

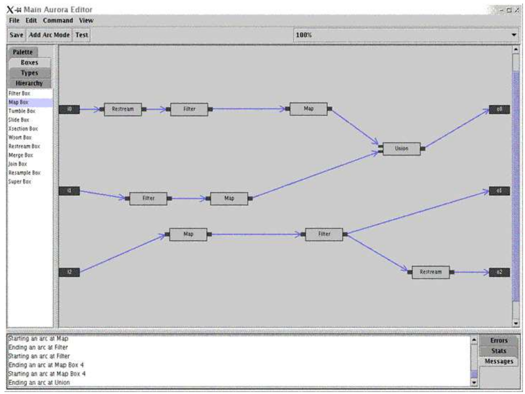


Figure 1: The Aurora GUI

warehouses are out-of-date by 1/2 of the refresh interval on average, typically 24 hours. Hence, business analysis is similarly out-of-date.

A widespread goal is ”the real-time enterprise”, through which business analysis could be done in real time. Such tactical analysis requires streams of data to be captured from operational systems, combined and then acted on in real time. Supporting the real time enterprise is one goal of specialized stream processing engines such as Aurora and Medusa.

A second observation is that micro-sensor technology is declining in cost precipitously. Micro-sensors that reflect their state to a nearby active device are about the size of a United States dime, and are rapidly converging on a price less than $0.10. In fact, Gillette just placed an order for 500,000,000 ”dimes”, presumably to put on each package of razor blades. This and other sensor technology will allow most any object of value to report its state (including its geographic position) in real time. Such technology will enable a new class of application to collect and act on real-time streams of state information from objects. We will call these monitoring applications, and a major focus of stream processing engines is to support such applications.

In the remainder of this paper we discuss two stream processing prototypes, Aurora and Medusa. We begin in Section 2 with the Aurora engine, then turn in Section 3 to the Medusa network infrastructure. Section 4 continues with the two different approaches to distributed processing. The first, Aurora*, assumes a tight coupling between the various machines, while the second, Medusa, is appropriate when the various systems belong to different organizations. Also discussed in Section 4 is a co-operative study on high availability in stream processing systems. Section 5 then turns to the Linear Road benchmark and offers a brief rationale for this work. The paper concludes in Section 6 with the current state of the two prototypes.

2 The Aurora Stream Processing Engine

The Aurora stream processing engine has five features that distinguish it from other proposals in the same general space such as [5, 6, 10]. They are a workflow orientation, a novel collection of operators, a focus on efficient scheduling, a focus on maximizing quality of service, and a novel optimization structure. We discuss each of these aspects in turn.

2.1 Workflow Orientation

There are two main considerations that led us to build Aurora as a workflow system. First, most monitoring applications contain a component that performs either sensor fusion or data cleansing. Many sensors (especially mobile ones) are low power, and therefore noisy. Hence, signal processing must be performed on the resulting signal. In addition, it is often necessary to triangulate observations from two different sensors in order to define the position of an object. Such front-end signal processing can either be part of the stream processing system or contained in a separate subsystem. Aurora supports signal-processing through powerful programmable operators that allow users to specialize their behavior on either a tuple-by-tuple or an aggregate (window) basis.

The feeling of the Aurora designers is that both data storage, and real-time processing should be in the same system, in order to provide optimized system performance. In this way, the number of boundary crossings between the application layer and the storage/processing layer can be minimized. To facilitate this combination of function, one must adopt a more general processing model than just a query language.

The second reason has to do with query optimization. When an application designer wishes to add new capabilities to an Aurora workflow, he merely locates the correct place in the workflow diagram to add the needed functionality. In contrast, if he submitted one or more SQL commands to an Aurora system, then Aurora would have to do multiple query optimization in order to construct an optimized composite query plan. Common subexpression elimination is at the heart of multiple query optimization and is known to be a very hard problem [12]. The Aurora designers wished to avoid this particular ”snake pit”.

For these reasons, the Aurora application designer is presented with a ”boxes and arrows” GUI through which he can build his workflow. The boxes in the diagram represent primitive Aurora operators, and the arrows indicate data flow. A screen shot of the Aurora GUI is shown in Figure 1.

2.2 The Aurora Operators

The primitive Aurora operators (boxes) have gone through several iterations, and will doubtless go through a few more before they ultimately stabilize. The current collection is defined precisely in [1], and includes the standard filtering, mapping, windowed aggregate, and join boxes. These operators are found in many other stream-oriented systems. Aurora extends this collection with four aspects of novel functionality. First, windowed operations have a timeout capability. If a windowed operation is expecting to operate on three messages, then it must block until all three are received. It is possible for Aurora to block indefinitely in this situation, for example if the generator of the missing message has become disconnected from the Aurora system. In order to avoid this undesirable behavior, Aurora windowed operators can timeout, and produce an output message based on less than the required number of input messages.

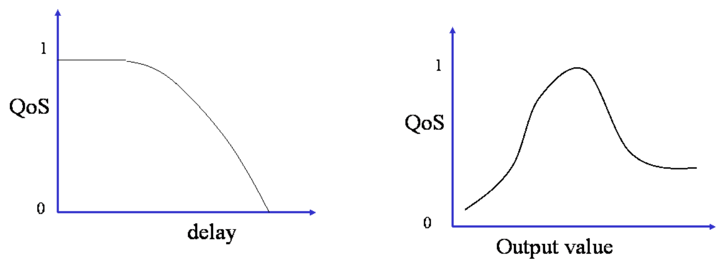
The second feature deals with out-of-order input messages. When there is significant network delay, messages may arrive in an order not intended by the application designer. Further, since Aurora allows windows on any attribute, monotonic attribute values cannot be guaranteed. If a windowed operator is expecting to process all the 9AM messages, then it must wait a little while after receiving a 9:01 message before ”closing” the 9AM window. A slack parameter on each windowed operator allows this sort of controlled waiting. A simpler (and less functional) mechanism is presented in [10] to deal with the same problem.

The third novel feature of Aurora operators is extendability. Aggregates, filters, and mapping operators can all be user-defined. Hence, Aurora is not restricted to a built-in collection of aggregates, as in most SQL systems. Instead, if an Aurora application designer wishes to have ”third largest” as an aggregate, then he is free to define it. Aurora closely follows the extendability features in POSTGRES [14] this regard.

Lastly, Aurora implements a novel resample operator. This operator allows interpolation between two values of a first stream, thereby constructing a message that matches the timestamp of a message in a second stream.

2.3 The Aurora Scheduler

A common misperception about scheduling algorithms is that the cost of running them does not need to be considered. On the contrary, our initial experimental results from the Aurora engine implementation is that the CPU cost of running the scheduler is comparable to or exceeds the cost of executing an Aurora operation on an incoming message [4]. Put differently, if the inner loop of control flow in a stream-based system is:



(a) Drop-based (b) Value-based

Figure 2: Two QoS Graphs

```
Run the scheduler to decide which message to process and then
Cause a single box to the execute that message
```

then the scheduling overheard may exceed the execution cost and an unreasonable (greater than 50%) degradation in performance will be observed. In contrast, the inner loop in Aurora can assemble large trains of messages and then move them collectively through a substantial number of boxes in the workflow diagram. Such train scheduling is the only way to keep the overhead of the scheduler from being onerous.

The objective of our train scheduling algorithms is to maximize quality of service, which is the topic of the next section. We have investigated a collection of heuristic algorithms, having given up long ago on finding algorithms that are provably optimal according to some metric. Our initial scheduling algorithms are presented in [3] and their experimental performance in [4]. We expect to continue our heuristic algorithm development in combination with ongoing experimental studies on our prototype. In particular, we must strive for lower overhead scheduling strategies that increase the percentage of useful work that an Aurora system can deliver.

2.4 Quality of Service

A fundamental tenet of Aurora is that Quality of Service (QoS) should be maximized in a stream processing system. Since Aurora is fundamentally a real-time engine, it is obvious that some applications should get better service than others. In military applications, for example, the task that tracks an incoming missile should get all possible resources and other less important tasks should face temporary resource starvation. Differential resource allocation has long been a tenet in real-time systems [11], but has not been utilized in current DBMSs.

The Aurora approach is for each application that receives output messages from the Aurora engine to specify a QoS graph. Two example graphs are shown in Figure 2. Figure 2(a) shows the utility that the application receives from varying response times. QoS graphs can also be based on metrics other than response time. For example, in a heart monitoring application, "normal" output is much less important than messages that represent abnormal behavior. In this case, a value-based QoS graph is appropriate as noted on Figure 2(b).

The basic Aurora idea is to have human-specified QoS graphs at the outputs of an Aurora workflow, and then to infer approximate QoS curves for all interior workflow nodes by "pushing" the QoS curves "upstream" through the operator boxes. The details of this operation for value-based QoS are indicated in [15].

With a derived QoS curve on every arc of a workflow, two activities are possible. First, the scheduler can make intelligent decisions based on these curves, as noted in the previous section. In this way, unimportant portions of the workflow can be "starved" to allow more important ones to get the majority of the resources.

Secondly, in certain Aurora applications, it is acceptable to discard messages in overload conditions. The rationale is that this behavior is desired in real-time systems. Also there may be lost messages in the sensor network when an object goes out of range. Hence, ACID properties may be impossible to achieve anyway. As such, Aurora can optionally be instructed to shed messages when overloaded. This pruning is accomplished by adding temporary ”drop” boxes. The purpose of a drop box is to filter messages, either based on value or randomly, in order to reduce load and provide better overall quality of service at the expense of accuracy.

One assumption that Aurora must make is that applications connected to different outputs in an Aurora workflow have a common notion of quality of service. As such, it is essential that they come from a single organization, in which context it is possible to have a common notion of QoS.

2.5 Aurora Optimization

An Aurora workflow is a dynamic object. When new applications get connected to the workflow, the number of boxes and arcs changes. Also ad-hoc queries that are run just once and then discarded have a similar effect. In addition, an Aurora workflow may become quite large; after all it is the stream processing activity of all applications that deal with a collection of sensor inputs.

Contrary to current DBMSs which perform optimization at compile-time by examining most of the possible ways to execute a given query, Aurora takes a radically different approach. First, it is implausible to exhaustively examine all (or most of) the possible permutations of the boxes in an Aurora workflow; the computational complexity is just too high in a large workflow. Second, every time the workflow is changed, this process would have to be repeated, leading to prohibitive overhead. As a result, Aurora does not examine all possible box rearrangements, and performs only run-time optimization.

An Aurora workflow has the notion of connection points. These are marked arcs in a workflow to which additional pieces of workflow can be connected. In addition, connection points also keep a user-specified amount of history, so that newly added Aurora boxes can examine historical messages if they so choose. Since new boxes can be added to connection points, they mark places of invariance in an Aurora workflow. The optimizer cannot reorder boxes by pushing a downstream box through a connection point. This reduces the complexity of the optimization process. In addition, not all Aurora operators commute as noted in [1]; again limiting the possible changes to an Aurora network.

While an Aurora workflow is executing, it is the job of the Aurora optimizer to examine small subnetworks for the possibility of a box rearrangement with better performance. If such a rearrangement is found, then messages are held on the upstream arcs while the subnetwork is ”drained”. Then the rearrangment can be put in place and processing resumed.

Notice that this results in an optimizer architecture that implements a ”greedy” algorithm. It has been clearly shown that such an approach may not produce a globally optimal plan, but we don’t see a way to do better on large workflows.

3 Medusa Network Infrastructure

Many stream processing applications run on multiple computers in a network. In contrast to several other projects which have focused on single machine environments, both Aurora and Medusa are actively pursuing distributed architectures. To facilitate both projects, a common networking infrastructure has been written by the Medusa group at M.I.T.

This infrastructure performs several functions, as discussed in detail in [7]. First, it supports a distributed naming scheme so that the output of an Aurora workflow at one site can be named and connected to a named input node on a second Aurora workflow at a second site. This allows local Aurora workflows to be assembled into larger distributed workflows. In addition, the networking layer multiplexes messages from different streams onto a much smaller number of TCP/IP connections. This multiplexing cuts down on the number of

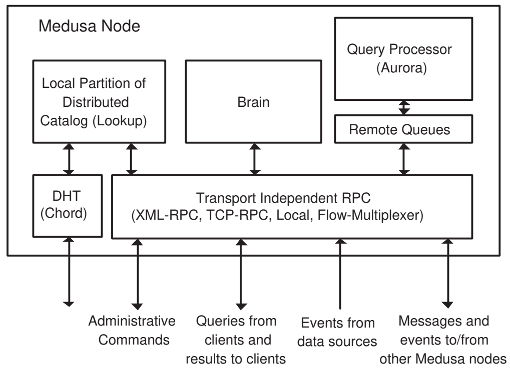


Figure 3: High-level architecture of a Medusa node. Components asynchronously process messages that arrive through their respective input queue(s). *Brain* processes all administrative commands such as creation of new schemas and streams. It receives ad-hoc queries which it partitions and distributes across other nodes. It also monitors and manages load at runtime by moving operators to and from other nodes. The local query processor runs an instance of Aurora. The remote queues component is a wrapper around the local query processor. It forwards events to and from other Medusa nodes and routes events coming from data sources to appropriate locations. *Lookup* holds one fraction of the catalog running over a distributed hash-table (DHT) such as Chord [13]. The transport independent RPC allows clients and Medusa nodes to seamlessly communicate with each other using various transport mechanisms. Within a node, components also communicate through that layer, but the communication is then a light-weight message exchange between threads.

physical connections between sites and improves efficiency. Similar connection pooling is widely performed by application servers and current commercial DBMSs. A report on the flow multiplexer is in preparation [9].

A second piece of infrastructure functionality is a collection of distributed catalogs that store the pieces of a distributed workflow and their interconnections. We are investigating using Chord [13] as underlying distributed hash-table to assign distinct catalog partitions to Medusa nodes.

A last piece of infrastructure functionality is transport. Basically once a distributed workflow is executing, this component efficiently delivers messages from an output arc on one machine to an input arc on a second machine. Figure 3 illustrates the architecture of a Medusa node and the components described above.

4 Distributed Stream Processing

There are two aspects of distributed workflow being investigated. Both groups are refining different processing models, and they are co-operating on a study of high-availability in a stream-based world. This section discussed both classes of efforts.

4.1 Distributed Workflow

Both groups are investigating different notions of distributed workflow. The Aurora project is pursuing Aurora*, which will connect multiple Aurora workflows together in a distributed environment. The basic idea is to push QoS graphs across site boundaries in a similar manner to the way they are pushed ”upstream” on a single host. This will allow a collection of local schedulers to collectively maximize QoS. In addition, the Aurora optimizer will be extended so that the process of quiescing, changing and restarting a subnetwork can cross a site boundary. This will allow cross-site optimization, as well as load balancing across sites. The specific approach is discussed in [7]. The Aurora* architecture is appropriate where all the hosts on which a distributed workflow is executing belong to the same organization. In such a single domain it is reasonable to assume that a common notion of QoS exists across machine boundaries.

In contrast the Medusa group is focused on environments where the hosts belong to different organizations and no common QoS notion is feasible. This multiple domain situation would be found when multiple organizations are providing services to an entity, as would be common in a Web services world. It is also typical in current cell phone environments where multiple providers co-operate to provide roaming.

The Medusa approach to distributed workflow is based on an agoric model. Each participant (host) is assumed to be an economic entity, that is interested in maximizing local profit. As such, it can enter into contracts to provide stream processing services for other participants. For example, one participant might have a geographic database of hotels and a second a similar database of restaurants. A third participant might collect sensor data from cars moving around a metropolitan area. The third participant could contract with each of the first two to provide a service to its mobile consumers to find on demand the hotels within a mile of the consumer with a first-class restaurant on the premises.

The Medusa approach requires a few assumptions in order to avoid economic chaos. For example, messages must have positive value, and applying an Aurora workflow to a message must increase its value. The Medusa group is then focused on refining its contracting model and on finding properties of the model that are enabled by certain additional assumptions. One specific focus is on finding what assumptions are necessary to guarantee that all participants avoid overload situations. A report on the agoric model and its properties is in preparation [2].

4.2 High Availability

Although some stream processing systems are non-transactional as noted above, there are others, especially in the realm of the real-time enterprise that require ACID properties. For example, a large financial institution is interested in exploring the use of a stream processing system to route trades for financial securities to the best "desk" for execution. In this world, losing messages is unthinkable. For this reason, Medusa and Aurora are co-operating in a study of high availability (HA) in stream systems.

The gold standard of HA is the so-called process-pairs model originated by Tandem. Here, an application (process) is allotted a backup on a second site. Whenever the process reaches a checkpoint, it sends a message to its partner with state information. If the primary fails, then the partner resumes computation from the most recent checkpoint, and uninterrupted operation is assured in the presence of a single failure.

If one assumes a distributed workflow of Aurora operators, then one need not adopt a process-pair model. Instead each site can buffer messages for its downstream neighbor. Only when the message has assuredly exited from the downstream neighbor, can the buffer be truncated. Hence, a small collection of "queue trimming" messages must flow upstream to sites buffering messages. If a site fails then the upstream neighbor can notice and restart the piece of the Aurora network on a backup site and send a copy of the queued messages to it. When the dead site resumes operation, the reverse switch can take place. This will achieve resilience from a single site failure with a minimum of messages. Also, HA operation is "lazy", i.e. messages are not actively sent to a backup site until a failure is observed.

As one can readily imagine, a stream-based HA scheme will have slower recovery time than a process-pair approach because of the lazy nature of the backup. However, run time overhead is noticeably lower. As software and hardware becomes more reliable, we believe that this tradeoff of longer recovery time for lower run-time overhead is worth exploring. As such, we have written a simulation model that can explore both kinds of schemes. We are now exploring the operation of the simulation, and a paper on our results is in preparation [8].

5 Linear Road Benchmark

Several researchers have doubted the need for specialized stream processing engines, speculating instead that commercial DBMSs will work just fine on stream applications. To resolve this issue, we have designed a benchmark in co-operation with other stream projects called Linear Road. It simulates collecting tolls on a collection of expressways, based on the amount of traffic that is using each segment of the expressway. It is driven by real-time reports of vehicle locations from assumed in-vehicle sensors. During the current semester we plan on using this benchmark to test performance of our prototypes against a popular commercial DBMS. We have invited other stream projects to participate, and Stanford has agreed to also run the benchmark. A report on this activity is planned for midyear [16].

6 The Prototypes

The Aurora engine consists of about 75,000 lines of C++ code and 30,000 lines of Java for the design-time user interface. It is operational at this time. We are adding the remainder of the planned features and working on improving performance in preparation for the Linear Road benchmarking activity. The Medusa infrastructure is also operational and its integration with the Aurora engine is underway. The Medusa economic model is fairly robust, and a primitive implementation exists. When integration with Aurora is complete, we will switch over to using the Aurora engine for local processing. We are also about to begin serious efforts on Aurora*.

References

[1] Daniel J. Abadi, Don Carney, Uğur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: A new model and architecture for data stream management. *The VLDB Journal: The International Journal on Very Large Data Bases*, Submitted.

[2] Magdalena Balazinska, Can Emre Koksal, Hari Balakrishnan, and Michael Stonebraker. The Medusa economic model for load management and sharing. In preparation.

[3] Don Carney, Uğur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Greg Seidman, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Monitoring streams: A new class of data management applications. In *Proc. of the 28th International Conference on Very Large Data Bases (VLDB'02)*, August 2002.

[4] Don Carney, Uğur Çetintemel, Alexander Rasin, Stan Zdonik, Mitch Cherniack, and Michael Stonebraker. Operator scheduling in a data stream manager. Technical Report CS-03-04, Department of Computer Science, Brown University, February 2003.

[5] Sirish Chandrasekaran, Amol Deshpande, Mike Franklin, and Joseph Hellerstein. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *Proc. of the First Biennial Conference on Innovative Data Systems Research (CIDR'03)*, January 2003.

[6] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. NiagaraCQ: A scalable continuous query system for Internet databases. In *Proc. of the 2000 ACM SIGMOD International Conference on Management of Data*, May 2000.

[7] Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska, Don Carney, Uğur Çetintemel, Ying Xing, and Stan Zdonik. Scalable distributed stream processing. In *Proc. of the First Biennial Conference on Innovative Data Systems Research (CIDR'03)*, January 2003.

[8] Jeong-Hyon Hwang, Hiro Iwashima, Alexander Rasin, Magdalena Balazinska, Hari Balakrishnan, Uğur Çetintemel, Mitch Cherniack, Michael Stonebraker, and Stan Zdonik. High-availability in stream-based processing systems. In preparation.

[9] Hiro Iwashima, Jon Salz, and Hari Balakrishnan. Flexible bandwidth sharing through application-level TCP flow multiplexing. In preparation.

[10] Rajeev Motwani, Jennifer Widom, Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Gurmeet Manku, Chris Olston, Justin Rosenstein, and Rohit Varma. Query processing, approximation, and resource management in a data stream management system. In *Proc. of the First Biennial Conference on Innovative Data Systems Research (CIDR'03)*, January 2003.

[11] Gultekin Ozsoyoglu and Richard T. Snodgrass. Temporal and real-time databases: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 7(4):513–532, 1995.

[12] Timos K. Sellis and Subrata Ghosh. On the multiple-query optimization problem. *IEEE Transactions on Knowledge and Data Engineering*, 2(2):262–266, 1990.

[13] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proc. of the ACM SIGCOMM Conference*, August 2001.

[14] Michael Stonebraker and Lawrence A. Rowe. The design of POSTGRES. In *Proc. of the 1986 ACM SIGMOD International Conference on Management of Data*, 1986.

[15] Nesime Tatbul, Uğur Çetintemel, Stan Zdonik, Mitch Cherniack, and Michael Stonebraker. Load shedding in a data stream manager. Technical Report CS-03-03, Department of Computer Science, Brown University, February 2003.

[16] Richard Tibbetts and Michael Stonebraker. Linear Road benchmark: Performance evaluation of stream-processing engines. In preparation.