# The Design of the Borealis Stream Processing Engine

Daniel Abadi, Yanif Ahmad, Hari Balakrishnan,
Magdalena Balazinska, Ugur Cetintemel,
Mitch Cherniack, Jeong-Hyon Hwang,
John Jannotti, Wolfgang Lindner, Samuel Madden,
Alexander Rasin, Michael Stonebraker,
Nesime Tatbul, Ying Xing, Stan Zdonik

Department of Computer Science
Brown University
Providence, Rhode Island 02912

**CS-04-08**
June 2004

**Paper Title** The Design of the Borealis Stream Processing Engine


**Authors** Daniel Abadi, MIT (dna@mit.edu)
Yanif Ahmad, Brown University (yna@cs.brown.edu)
Hari Balakrishnan, MIT (hari@csail.mit.edu)
Magdalena Balazinska, MIT (mbalazin@lcs.mit.edu)
Uğur Çetintemel, Brown University (ugur@cs.brown.edu)
Mitch Cherniack, Brandeis University (mfc@cs.bradeis.edu)
Jeong-Hyon Hwang, Brown University (jhhwang@cs.brown.edu)
John Jannotti, Brown University (jj@cs.brown.edu)
Wolfgang Lindner, MIT (wolfgang@csail.mit.edu)
Samuel Madden, MIT (madden@csail.mit.edu)
Alexander Rasin, Brown University (alexr@cs.brown.edu)
Michael Stonebraker, MIT (stonebraker@lcs.mit.edu)
Nesime Tatbul, Brown University (tatbul@cs.brown.edu)
Ying Xing, Brown University (yx@nms.lcs.mit.edu)
Stan Zdonik, Brown University (sbz@cs.brown.edu)

# The Design of the Borealis Stream Processing Engine

The Borealis Team


## Abstract

Borealis is a second-generation distributed stream processing engine that is being developed at Brandeis University, Brown University, and MIT. Borealis inherits core stream processing functionality from Aurora [11] and distribution functionality from Medusa [45]. Borealis modifies and extends both systems in non-trivial and critical ways to provide advanced capabilities that are commonly required by newly-emerging stream processing applications.

In this paper, we outline the basic design and functionality of Borealis. Through sample real-world applications, we motivate the need for dynamically revising query results and modifying query specifications. We then describe how Borealis addresses these challenges through an innovative set of features, including revision records, time travel, and control lines. Finally, we present a highly flexible and scalable QoS-based optimization model that operates across server and sensor networks.


## 1 Introduction

Over the last several years, a great deal of progress has been made in the area of stream processing engines (SPE). Several groups have developed working prototypes [1, 4, 13] and many papers have been published on detailed aspects of the technology such as data models [2, 5, 39], scheduling [12, 8], and load shedding [38, 9, 17]. While this work is an important first step, fundamental mismatches remain between the requirements of many streaming applications and the capabilities of first-generation systems.

This paper is intended to illustrate our vision of what second-generation SPE's should look like. It is driven by our experience in using Aurora [1], our own prototype, in several streaming applications including the Linear Road Benchmark [6] and several commercial opportunities. We present this vision in terms of our own design for Borealis, the successor to Aurora, but it should be emphasized that the issues raised here represent general challenges for the field as a whole. We present specifics of our design as concrete evidence for why these problems are hard and as a first cut at how they might be approached.

The fundamental requirements of second-generation SPE's are described below. They are described with an eye to how they arise in practice. The rest of the paper elaborates on how these requirements might be addressed in the next crop of systems.

**1. Dynamic revision of query results**: In many real-world streams, corrections or updates to previously processed data are available only after the fact. For instance, many popular data streams, such as the Reuters stock market feed, often include so-called *revision records*, which allow the feed originator to correct errors in previously reported data. Furthermore, stream sources (such as sensors), as well as their connectivity, can be highly volatile and unpredictable. As a result, data may arrive late and miss its processing window, or may be ignored temporarily due to an overload situation [38]. In all these cases, applications are forced to live with imperfect results, unless the system has means to revise its processing and results to take into account newly available data or updates.

**2. Dynamic query modification**: In many stream processing applications, it is desirable to change certain attributes of the query at run time. For example, in the financial services domain, traders typically wish to be alerted of *interesting* events, where the definition of "interesting" (i.e., the corresponding filter predicate) varies based on current context and results. In network monitoring, the system may want to obtain more precise results on a specific subnetwork, if there are signs of a potential Denial-of-Service attack. Finally, in a military stream application that Mitre explained to us, they wish to switch to a "cheaper" query when the system is overloaded. For the first two applications, it is sufficient to simply alter the operator parameters (e.g., window size, filter predicate), whereas the last one calls for altering the operators that compose the running query.

Another motivating application comes again from the financial services community. Universally, people working on trading engines wish to test out new trading strategies as well as debug their applications on historical data before they go live. As such, they wish to perform "time travel" on input streams. Although this last example can be supported in most current SPE prototypes (i.e., by attaching the engine to previously stored data), a more user-friendly and efficient solution would obviously be desirable.

**3. Flexible and highly-scalable optimization**: Currently, commercial stream processing applications are popular in industrial process control (e.g., monitoring oil refineries and cereal plants), financial services (e.g., feed processing, trading engine support and compliance), and network monitoring (e.g., intrusion detection, fraud detection). Here we see a *server heavy* optimization problem — the key challenge is to process high-volume data streams on a collection of resource-rich "beefy" servers. Over the horizon, we see a very large number of applications of wireless sensor technology (e.g., RFID in retail applications, cell phone services). Here, we see a *sensor heavy* optimization problem — the key challenges revolve around extracting and processing sensor data from a network of resource-constrained "tiny" devices. Further over the horizon, we expect sensor networks to become faster and increase in processing power. In this case the optimization problem becomes more balanced, becoming *sensor heavy, server heavy*. To date systems have exclusively focused on either a server-heavy environment [11, 29, 14] or a sensor-heavy environment [28]. Off into the future, there will be a need for a more flexible optimization structure that can deal with a very large number of devices and perform cross-network sensor-heavy server-heavy resource management and optimization.

In summary, a strong need for many target stream-based applications is the ability to modify various data and query attributes at run time, in an undisruptive manner. Furthermore, the fact that many applications are inherently distributed and potentially span large numbers of heterogeneous devices and networks necessitates scalable, highly-distributed resource allocation and optimization capabilities. As we will demonstrate, adding these advanced capabilities requires fundamental changes to the architecture of an SPE and cannot be effectively realized as an afterthought. As a result, we have designed a second-generation SPE, appropriately called *Borealis*. Borealis inherits core stream processing functionality from Aurora and distribution capabilities from Medusa. Borealis does, however, radically modify and extend both systems with an innovative set of features and mechanisms. This paper reports on the functionality and preliminary design of Borealis.

Section 2 provides an overview of the basic Borealis architecture. Section 3 describes support for *revision records*, the Borealis solution for dynamic revision of query results. Section 4 discusses two important features that facilitate on-line modification of continuous queries: *control lines* and *time travel*. Control lines extend Aurora's basic query model with the ability to change operator parameters as well as operators themselves on the fly. Time travel allows multiple queries (different queries or versions of the same query) to be easily defined and executed concurrently, starting from different points in the past or "future" (hence the name time travel). Section 5 discusses the basic Borealis optimization model that is intended to optimize various QoS metrics across a combined server and sensor network. This is a challenging problem due to not only the sheer number of machines that are involved, but also the various resources (i.e., processing, power, bandwidth, etc.) that may become bottlenecks. Our solution uses a hierarchy of complementary optimizers that react to "problems" at different timescales. Section 6 summarizes the related work in the area, and Section 7 concludes the paper with directions for future work.

## 2 Borealis System Overview

### 2.1 Architecture

Borealis is a distributed stream processing engine. The collection of continuous queries submitted to Borealis can be seen as one giant network of operators (aka query diagram) whose processing is distributed to multiple sites. Sensor networks can also participate in query processing behind a sensor proxy interface which acts as another Borealis site.

Each site runs a Borealis server whose major components are shown in Figure 1. *Query Processor (QP)* forms the core piece where actual query execution takes place. The QP is a single-site processor. Input streams are fed into the QP and results are pulled through *I/O Queues*, which route tuples to and from remote Borealis nodes and clients.

The QP is controlled by the *Admin* module that sets up locally running queries and takes care of moving query diagram fragments to and from remote Borealis nodes, when instructed to do so by another module. System control messages issued by the *Admin* are fed into the *Local Optimizer*. Local Optimizer further communicates with major run-time components of the QP to give performance improving directions. These components are:

- *Priority Scheduler*, which determines the order of box execution based on tuple priorities;
- *Box Processors*, one for each different type of box, that can change behavior on the fly based on control messages from the Local Optimizer;
- *Load Shedder*, which discards low-priority tuples when the node is overloaded.

The QP also contains *Storage Manager* which is responsible for storage and retrieval of data that flows through the arcs of the local query diagram. Lastly, the *Local Catalog* stores query diagram description and meta-data, and is accessible by all the components.

Other than the QP, a Borealis node has modules which communicate with their peers on other Borealis nodes to take collaborative actions. The *Neighborhood Optimizer* uses local load information as well as information from other Neighborhood Optimizers to improve load balance between nodes. As discussed in Section 5, a single node can run several optimization algorithms that make load management decisions at different levels of granularity. The *High Availability (HA)* modules on different nodes monitor each other and take over processing for one another in case of failure. *Local Monitor* collects performance-related statistics as the local system runs to report to local and neighborhood optimizer modules. The *Distributed Catalog* is used to access the global catalog through a Distributed Hash Table (DHT). All communication between the components within a Borealis node as well as between multiple Borealis nodes is realized through transport independent RPC, with the exception of data streams that go directly into the QP.

### 2.2 Data Model

Borealis uses an extended Aurora data model [2]. Aurora models streams as append-only sequences of tuples of the form $(k_1, \ldots, k_n, a_1, \ldots, a_m)$, where $k_1, \ldots, k_n$ comprise a key for the stream and $a_1, \ldots, a_m$ provide attribute values. To support the revision of information on a stream, Borealis generalizes this model to support three kinds of stream messages (i.e. tuples):

- *Insertion* messages, $(+, \mathtt{t})$, where $\mathtt{t}$ is a new tuple to be inserted with a new key value (note that all Aurora messages implicitly are insertion messages),

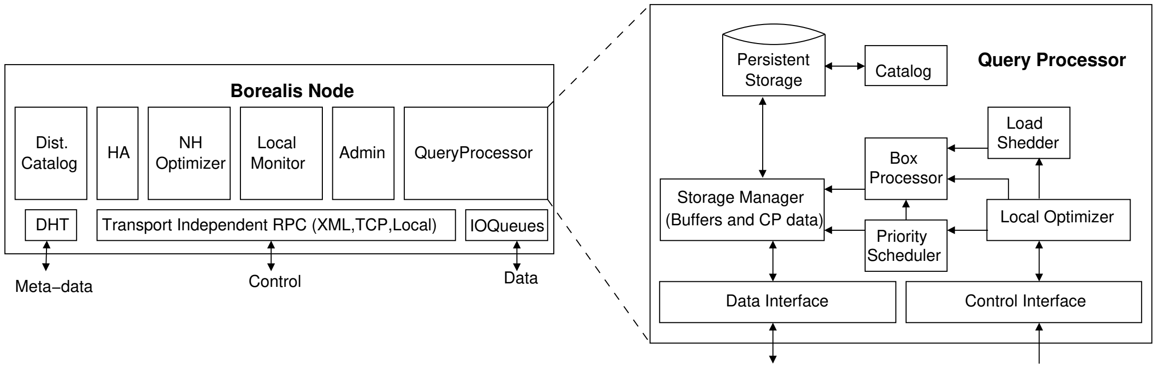


Figure 1: Borealis Architecture

- *Deletion* messages, $(-, \mathtt{t})$ such that $\mathtt{t}$ consists of the key attributes for some previously processed message, and
- *Replacement* messages, $(\leftarrow, \mathtt{t})$, such that $\mathtt{t}$ consists of key attributes for some previously processed message, and non-key attributes with revised values for that message.

Additionally, each Borealis message may carry QoS-related fields as described in Section 2.4.

### 2.3 Query Model

Borealis inherits the boxes-and-arrows model of Aurora for specifying continuous queries. Boxes represent query operators and arrows represent the data flow between boxes. Queries are composed of extended versions of Aurora operators that support revision messages. Each operator processes revision messages based on its available message history and emits other revision messages as output. Aurora's *connection points* (CPs) buffer stream messages that compose the message history required by operators. In addition to revision processing, CPs also support other Borealis features like time travel and query diagram versions.

An important addition to the Aurora query model is the ability to change box semantics on the fly. Borealis boxes are provided with special *control lines* in addition to their standard data input lines. These lines carry control messages that include revised box parameters and functions to change box behavior. Details of control lines and dynamic query modification are presented in Section 4.

### 2.4 QoS Model

As in Aurora, a Quality of Service model forms the basis of resource management decisions in Borealis. Unlike Aurora, where each query output is provided with QoS functions, Borealis allows QoS to be accessible at any point in a data flow. For this purpose, messages are supplied with a *Vector of Metrics (VM)*. These metrics include content-related properties (e.g., message importance) or performance-related properties (e.g., message arrival time, total resources consumed for processing the message up to the current point in the query diagram, number of dropped messages preceding this message). The attributes of the VM are predefined and identical on all streams. As a message flows through a box, some fields of the VM can be updated by the box code. A diagram administrator (DA) can also place special Map boxes into the query diagram to change VM.

Furthermore, there is a universal, parameterizable *Score Function* for an instantiation of the Borealis System that takes in VM and returns a value in $[0, 1]$, that shows the current QoS value of a message. This function is known to all run-time components (such as the scheduler) and shapes their processing strategies. The overall goal is to deliver maximum average QoS value at system outputs. Section 5 presents our optimization techniques to achieve this goal.

## 3 Dynamic Revision of Query Results

As with most stream data management systems, Aurora assumed an append-only model in which a message (i.e. tuple) cannot be updated once it is placed on a stream. If the message gets dropped or contains incorrect data, applications are forced to live with approximate or imperfect results.

In many real-world streams, corrections or updates to previously processed data are available after the fact. The Borealis data model extends Aurora by supporting such corrections by way of revision messages. The goal is to process revisions intelligently, correcting query results that have already been emitted in a manner that is consistent with the corrected data. Revision messages can arise in several ways:

1. The input can contain them. For example, a stock ticker might emit messages that fix errors in previously emitted quotes.
2. They can arise in cases in which the system has shed load, as in Aurora in response to periods of high load [38]. Rather than dropping messages on the floor, a Borealis system might instead designate certain messages for delayed processing. This could result in messages being processed out-of-order, thus necessitating the revision of emitted results that were generated earlier.
3. They can arise from time-travel into the past or future. This topic is covered in detail in Section 4.

### 3.1 Revisions and "Replayability"

Revision messages give us a way to recover from mistakes or problems in the input. Processing of a revision message must replay a portion of the past with a new or modified value. Thus, to process revision messages correctly, we must make a query diagram "replayable".

Replayability is useful in other contexts such as recovery and high availability [25]. Thus, our revision scheme generalizes a replay-based high-availability (HA) mechanism. In HA, queued messages are pushed through the query diagram to recover the operational state of the system at the time of the crash. In our revision mechanism, messages are also replayed through the query diagram. But *failure* is typically infrequent, and therefore, the replay mechanism for recovery can tolerate some run-time overhead. *Revisions* are more frequent, and therefore, the replay mechanism for processing revisions must be more sensitive to run-time overhead to prevent disastrous effects on system throughput.

In theory, we could process each revision message by replaying processing from the point of the revision to the present. However, in most cases revisions on the input affect only a limited subset of output tuples, and to regenerate unaffected output is wasteful and unnecessary. To minimize run-time overhead and message proliferation, we assume a *closed* model for replay that generates revision messages when processing revision messages. In other words, our model processes and generates "deltas" showing only the effects of revisions rather than regenerating the entire result.

While the scheme that we describe below may appear to complicate the traditional stream model and add significant latency to processing, it should be noted that in most systems, revision messages are exceptional rather than the norm. Further, because a revision message refers to historical data (and therefore the output it produces is stale regardless of how quickly it is generated), it may often be the case that revision message processing can be deferred until times of low load without significantly compromising its utility to applications.

### 3.2 A Revision Processing Scheme

We begin by discussing how revision messages are processed in a simple single-box query diagram before considering the general case. The basic idea of this scheme is to process a replacement message by *replaying* the diagram with previously processed inputs (the *diagram history*), but using the revised values of the message in place of the original values during the replay.[1] To minimize the number of output tuples generated, the box would replay the original diagram history as well as the revised diagram history, and emit revision tuples that specify the differences between the outputs that result.

It is obviously infeasible for a query diagram to maintain an entire diagram history of all input messages it has ever seen. Thus, each input to the box must be associated with a *history bound* (measured in time or number of tuples) that specifies how much history to keep around. This in turn limits how far back in time a revision message can be applied, and any revisions for messages that exceed the history bound must be ignored.

Given a diagram history, replay of box processing is straightforward. Upon seeing a replacement message, $t'$, a *stateless* box will retrieve the original message, $t$, from its diagram history (by looking up its key value) and will emit a revision message as appropriate. For example, filter with predicate $p$ will respond in one of four ways:

- if $p$, is true of $t$ and also of $t'$, the replacement message is propagated,
- if $p$ is true of $t$ but not of $t'$, a deletion message is emitted for $t$,
- if $p$ is not true of $t$ but is true of $t'$, an insertion message is emitted for $t'$, and
- if $p$ is not true of either $t$ or $t'$, no message is emitted.[2]

The processing of revision messages for *stateful* operators (e.g., aggregate) is a bit more complex because stateful operators process multiple input messages in generating a single output message (e.g., window computations). Thus, to process a replacement message, $t'$, for original message, $t$, an aggregate box must look up all messages in its diagram history that belonged to some window that also contained $t$, and reproduce the window computations both with and without the revision to determine what revision messages to emit. For example, suppose that aggregate uses a window of size 15 minutes and advances in 5 minute increments. Then, every message belongs to exactly 3 windows, and every replacement message will result in replaying the processing of 30 minutes worth of messages to emit up to 3 revision messages.

Revision processing for general query diagrams is a straightforward extension of the single-box diagram. In the general case, each box maintains its own diagram history. Because the processing model is closed, each downstream box is capable of processing the revision messages generated by its upstream neighbors.

One complication concerns *message-based* windows (i.e., windows whose sizes are specified in terms of numbers of messages). While replacement messages are straightforward to process with such windows, insertion and deletion messages can trigger misalignment with respect to the original windows, meaning that revision messages must be generated from the point of the revision all the way to the present. Unless the history bound for such boxes are low, this can result in the output of many revision messages. This issue is acute in the general query diagram case, where messages can potentially increase exponentially in the number of stateful boxes that process them. We consider this *revision proliferation* issue in Section 3.4, but first we consider how one can reduce the size of diagram histories in a general query diagram at the expense of increasing revision processing cost.

[1] Analogously, insertion messages would be added to the diagram history and the deletion messages would remove the deleted message from the diagram history.

[2] The processing of insertion and deletion messages is similar and therefore omitted here.

### 3.3 Processing Cost vs. Storage

It is clear that the cost of maintaining a diagram history for every box can become prohibitive. However, it should be observed that discrepancies in *history bounds* between boxes contained in the same query makes some diagram history unnecessary. For example, consider a chain of two aggregate boxes such that:

- the first aggregate in the chain specifies a window of 2 hours and has a history bound of 5 hours, and
- the second aggregate in the chain specifies a window of 1 hour and has a history bound of 10 hours.

Note that the first aggregate box in the chain can correctly process revisions for messages up to 3 hours old, as any messages older than this belonged to windows with messages more than 5 hours old. As a result, the second aggregate box will have an effective history bound of 4 hours as it will never see revisions for messages more than 3 hours old, and therefore need messages more than 1 hour older than this. Thus, the diagram can be *normalized* as a result of this static analysis so that no history is stored that can never be used.

While query diagrams can be normalized in this manner, it may still be necessary to reduce the storage demands of diagram histories. This can be done by moving diagram histories upstream so that they are shared by multiple downstream boxes. For example, given the two box diagram described above, a single diagram history of 5 hours could be maintained at the first aggregate box, and processing of a revision message by this box would result in the emission of new revision messages, *piggybacked with* all of the messages in the diagram history required by the second box to do its processing. This savings in storage comes at the cost of having to dynamically *regenerate* the diagram history for the second box by reprocessing messages in the first box. In the extreme case, minimal diagram history can be maintained by maintaining this history only at the edges of the query diagram (i.e., on the input streams). However, this means that the arrival of a revision message to the query diagram must result in emitting all input messages involved in its computation, and regenerating all intermediate results at every box. In other words, as we push diagram histories towards the input, revision processing results in the generation of fewer "delta's" and more repeated outputs.

At the other extreme, with more storage we can reduce the processing cost of replaying a diagram. For example, an aggregate box could potentially maintain a history of all of its previous state computations so that a revision message can increment this state rather than waiting for this state to be regenerated by reprocessing earlier messages in the diagram history. This illustrates both extremes of the tradeoff between processing cost and storage requirements in processing revision messages.

### 3.4 Revision Proliferation vs. Completeness

Our previous discussion has illustrated how messages can proliferate as they pass through aggregates, thereby introducing additional overhead. We now turn to the question of how to limit the proliferation of revision messages that are generated in the service of a revision message. This is possible provided that we can tolerate incompleteness in the result. In other words, we limit revision proliferation by ignoring revision messages or computations that are deemed to be less important.

The first and simplest idea limits the paths along which revisions will travel. This can be achieved by allowing applications to declare whether or not they are able to and interested in dealing with revisions. This can be specified directly as a boolean value or it can be inferred from a QoS specification that indicates an application's tolerance for imprecision. For example, high tolerance for imprecision might imply a preference for ignoring revision messages. Revision processing might also be restricted to paths that contain updates to tables since the implication of a relational store is that the application likely cares about keeping an accurate history. Further revision processing beyond the point of the update may be unnecessary.

Another way to limit revision proliferation is to limit which revisions are processed. If a tuple is considered to be "unimportant", then it would make sense to drop it. This is similar to semantic load shedding [38]. In Borealis, the semantic value of a message (i.e., its *importance*) is carried in the message itself. The score function that computes QoS value of a message can be applied to a revision message as well, and revisions whose importance falls below a threshold can be discarded.

## 4 Dynamic Modification of Queries

### 4.1 Control Lines

**Basic Model.** Borealis boxes are provided with special control lines in addition to their standard data input lines. Control lines carry messages with revised box parameters and new box functions. For example, a control message to a Filter box can contain a reference to a boolean-valued function to replace its predicate. Similarly, a control message to an Aggregate box may contain a revised window size parameter. Additionally, each control message must indicate when the change in box semantics should take effect. Change is triggered when a monotonically increasing attribute received on the data line attains a certain value. Hence, control messages specify an <attribute, value> pair for this purpose. For windowed operators like Aggregate, control messages must also contain a flag to indicate if open windows at the time of change must be prematurely closed for a clean start.

Borealis stores a selection of parameterizable functions applicable to its operators. Two types of functions are stored in the *function storage base*: functions with specified parameters and functions with open parameters. Functions with specified parameters indicate what their arguments are in the function specification. For example, $h(\$3, \$4) = \$3 * \$4$ will multiply the third and fourth attributes of the input messages. In contrast, functions with open parameters do not specify where to find their arguments. Instead they use the same binding of arguments in the function that

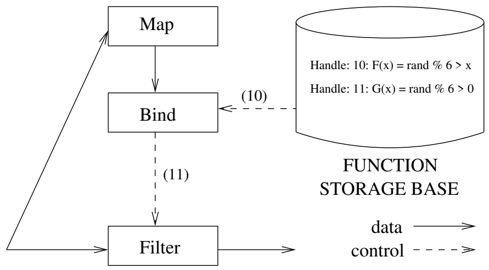


Figure 2: Control-Line Example Use

they replace. For example, if a box was applying the function: $g(x, y) = x - y$ to input messages with data attributes x and y, then sending $f(x, y) = x + y$ along the control line will replace the subtraction with an addition function on the same two attributes of input messages.

The design of the function store is fairly straight forward; it is a persistent table hashed on the function handle, with the other attributes contained in the stored record.

We expect the common practice in Borealis will be to observe a running Borealis diagram, decide to change the parameters to a box, and output these parameters. Hence, a new box is required that will Bind a function to these parameters. Borealis introduces a new operator, called *Bind*:

$Bind(B_1 = F_1, ..., B_m, = F_m)(S)$

Bind accepts one or more function handles, $F_i(t)$, and binds parameters to them, thereby creating a new function. For example, Bind can create a specialized multiplier function, $B_i$, by binding the fourth attribute of an input message $S$ to the second parameter of a general multiplier function.

**Example.** To illustrate the use of control lines and the Bind operator, consider the example in Figure 2, which will automatically decrease the selectivity of a Filter box if it begins to process important data. Assume that the Map operator is used to convert input messages into an importance value ranging from 1 to 5. The Bind box subtracts the importance value from 5 and binds this value to $x$ in function 10. This creates a new function (with handle 11), which is then sent to the Filter box.

**Semantics.** A design decision in Borealis is to optionally allow a diagram administrator (DA) to specify that messages can be processed out of order in order to allow more important messages to be moved before less important ones. To deal with Borealis rearranging message order, operators must buffer old values of their parameters so that they can be used for messages that arrive out of order. In order to bound the number of control messages which must be buffered, the DA can specify a time bound after which old control messages can be discarded.

A second issue occurs if control line data arrives late and the box has already processed some messages using the old box functionality which were intended for the new box parameters. In this case, Borealis can resort to revision messages and time travel, which is discussed next.

### 4.2 Time Travel

Borealis time travel is motivated by the desire of applications to "rewind" history and then repeat it. In addition, one would like a symmetric version of time travel, i.e., it should be possible to move forward into the future, typically by running a simulation of some sort.

To support these capabilities, we leverage three advanced mechanisms in Borealis: enhanced connection points, connection point versions, and revision messages. Each will be discussed next, followed by a discussion of semantics of time travel.

**Enhanced Connection Points.** To enable time travel, we leverage Aurora's connection points (CP) [2]. In Aurora, arcs designated as CPs store message histories. Later, ad-hoc queries can be connected to such points to receive historical as well as real-time answers.

Two parameters determine the set of messages buffered at a CP: start_time and max_time. Start_time determines the oldest buffered message and can be specified as an absolute value or a value relative to the latest input message. Max_time is always an absolute value and determines when the CP will stop accepting new input data. Max_time is optional and serves to isolate a CP from the upstream diagram.

To facilitate time travel, we define two new operations on CPs:

1. *replay operation*, to replay messages that are stored at a CP from an arbitrary message in the past, and
2. *offset operation*, to set the CP offset in time.

When offset into the past, a CP delays current messages before pushing them downstream. When offset into the future, the CP predicts future data. When producing future data, various prediction algorithms can be used based on the application.

To replay messages and thus travel back in time, a replay command is issued. This command takes two parameters: the message at which the replay should start and the replay speed. To support variable speed replays, CPs also store message inter-arrival time.

In addition to time travel, we allow CPs to switch to suspended mode. In suspended mode the CP accumulates new input messages without propagating them downstream.

**Connection Point Versions.** In general, we expect that users will want to continue normal processing on a running diagram, while time traveling a parallel version of a piece of the diagram. For this purpose, we introduce the notion of CP versions, which is a distinctly named logical copy of a CP. Each named version can be manipulated independently. It is possible to shift a CP version backward and forward in time without affecting other versions.

When the DA or the application creates a new named version of a CP, it can choose to (1) make a copy of all operators downstream from the connection point or (2) attach a new query diagram to the version.

CPs can be hierarchical. To uniquely identify a connection point, we require a hierarchical naming system and use XML descriptions for this purpose. A given CP is the root and the sequence of CP versions that lead to the specified one form the descendant nodes.

If an application subscribes to a stream without specifying a version name, it receives all versions of the stream. Each message is tagged with the identifier of the stream it belongs to, and applications may also subscribe to specific versions of output streams by using an XPath specification [40] using stream names.

**Time Travel Using Revisions.** To replay history from a previous point in time $P$, we use revision messages. When a CP receives a replay command, it first generates a set of revision messages that delete all the messages and revisions that occurred since $P$. To avoid the overhead of transmitting one revision per deleted message, we use a macro-message that summarizes all deletions. Once all messages are deleted, the CP produces a series of revisions that insert the messages and possibly their following revisions back into the stream.

During replay, all messages and revisions received by the connection point are buffered and processed only after the replay terminates thus ensuring that simultaneous replays on any path in the query diagram are processed in sequence and do not conflict.

When offset into the future, time-offset operators predict future values. As new data becomes available, these predictors can (but do not have to) produce more accurate revisions to their past predictions. Additionally, when a predictor receives revision messages, possibly due to time travel into the past, it can also revise its previous predictions.

**Semantics of Time Travel.** We now examine the effects of changing time at a CP on the query diagram downstream from that CP. When a CP time travels into the past to time $T$, it issues a single revision that deletes all messages produced since time, $T$. When an operator receives and applies such revisions, it rolls back its state to time $T+\epsilon$. The operator then issues revisions that undo all its operations since time $T+\epsilon$. Deleting all messages after a time $T$ therefore results in all operators downstream from a CP rolling back their state to time $T$.

Once the state is rolled back, the CP inserts new messages. If the query diagram is non-deterministic (e.g., it contains timeouts), re-processing the messages may result in a different sequence of output messages. If the diagram is deterministic, however, unless history is modified, the operators will produce the exact same output messages for a second time.

When time moves into the future, the predicted messages are forwarded as if they were the logical continuation of the input data. Downstream operators will process them normally. If there is a gap between the latest current and the first predicted message, a window that spans this gap may produce strange results. To avoid such behavior, all operators and CPs support an optional reset command that clears their state.

# 5 Borealis Optimization

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

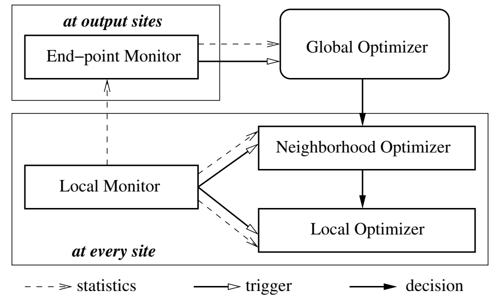


Figure 3: Optimizer Components

## 5.1 Overview

A Borealis application, which is a single connected diagram of processing boxes, is deployed on a network of $N$ servers and sensor proxies, which we will refer to as *sites*. There is a proxy for each type of sensor network that exists in the application. This proxy is charged with reflecting optimization decisions from the server network into appropriate tactics in its sensor network. The application diagram will be partitioned across the sites for distributed processing.

**Components.** The basic Borealis optimization consists of multiple collaborating monitoring and optimization components, as shown in Figure 3.

There are two types of monitors. First, a *local monitor (LM)* runs at each site and produces a collection of local statistics, which it forwards periodically to the end-point monitor (EM). LM maintains various box- and site-level statistics regarding utilization and queuing delays for various resources including CPU, disk, bandwidth, and power (only relevant to proxies). Second, an *end-point monitor (EM)* runs at every site that produces Borealis output and is primarily responsible for watching every Borealis output for fluctuations in QoS. EM evaluates QoS for every output message and keeps statistics on QoS for all outputs for the site.

There are three levels of collaborating optimizers. At the lowest level, a *local optimizer* runs at every site and is responsible for scheduling messages to be processed as well as deciding where in the locally running diagram to shed load, if required. A *neighborhood optimizer* also runs at every site and is primarily responsible for load balancing the resources at a site with those of its immediate neighbors. At the highest level, a *global optimizer* is responsible for accepting information from the end-point monitors and making global optimization decisions.

**Control Flow.** The monitoring components run continuously and trigger the appropriate optimizer(s) to take action. Each optimizer tries to resolve the problem (e.g., resource overload) itself first. If it cannot do this within a prespecified time period, the optimizer at the next higher level is triggered. This approach strives to handle problems locally as much as possible. The intuition here is that, in general, local decisions are cheaper to make and realize, and are less disruptive. Another implication is that transient problems are dealt with locally, whereas more persistent problems potentially require global intervention.

The local monitor triggers the corresponding local optimizer when it detects (1) a problem (i.e., a resource overload) or (2) an opportunity for further optimization (i.e., a "notable" difference between the load levels of the neighboring sites). In the first case, if the problem is not resolved within a specific time window, the monitor triggers the neighborhood optimizer. If the problem persists even longer, the global optimizer is triggered. Similarly, the end-point monitor triggers the global optimizer when it detects a problem (i.e., low average QoS) or an optimization opportunity (i.e., a "notable" difference between the load levels of *any* two sites).

**Problem Identification.** Detecting a specific resource bottleneck is a relatively trivial task that can be accomplished by tracking the utilization for each resource type. Conceptually, the solution is also straightforward: either shed load, or, preferably, identify slack resources to offload the overloaded resource.

Dealing with QoS is more challenging. If no information about the structure of the QoS function is available, then the optimizer can only take generic actions that are likely to improve performance, such as load balancing. If additional information is available, however, then Borealis can take different corrective actions depending on the problem type. To facilitate this, Borealis allows two levels of specification of additional information. The first is a vector: `[Lifetime, Coverage, Throughput, Latency]`, which indicates the relative importance of each of these components to overall QoS. Because each of these metrics is optionally a component of the VM, the end-point monitor can keep statistics on the components that are present. Together with the vector of weights, these statistics allow the end-point monitor to make a good prediction about the cause of the QoS problem. The second more elaborate mechanism allows the DA to specify one or more score functions corresponding to the above metrics. The end-point monitor can evaluate these auxiliary functions and maintain statistics on these components of overall QoS.

**Tactics.** The Borealis objective is to deliver maximum average QoS to the output applications. We have a variety of optimization tactics at our disposal: (1) Exploiting box commutativity and alternate implementations — these are the tools of conventional query optimization; (2) allocation of read and update boxes — since these boxes access and update stored tables, they are potentially vastly more expensive than the other box types and their location should be carefully considered; (3) allocation of (conventional) processing boxes; (4) table replica allocation; (5) priority scheduling; and (6) load shedding.

### 5.2 Initial Diagram Distribution

The goal of the initial diagram distribution, performed by the global optimizer, is to produce a "feasible" allocation of boxes and tables to sites using preliminary statistics obtained through trial runs of the diagram. The primary focus is on the placement of read and write boxes with the Borealis tables that they access. Because these boxes access stored state, they are significantly more expensive than regular processing boxes. Furthermore, it is desirable to *co-locate* Borealis tables with the boxes which read and write them in order to avoid potentially costly remote table operations.

The initial placement operates in two phases: the first phase identifies a set of "candidate" groups of boxes and tables that should be co-located. The second phase completes the process by appropriately assigning the remaining boxes.

We define the *bounding box* of a table $T$ as the minimal connected sub-graph of the Borealis diagram that spans all the read/write boxes that operate on $T$. We first compute the set of bounding boxes for all tables, which constitute the initial candidate set for co-location. We then extend this set iteratively by inserting combined bounding boxes: if there is an overlap between the two individual bounding boxes, we combine them into one big bounding box and insert it into the candidate set. For each candidate, we compute its CPU and I/O consumption per unit time using our initial statistics. We then order the candidates by their I/O requirements and the sites by their available I/O resources. Starting from the top of the candidate list, we try to allocate each candidate to a site. Whenever we make an allocation, we eliminate from consideration all the candidates that include the already allocated tables and boxes. We continue in this fashion, until the candidate set becomes empty or there remains bounding boxes that exceed the resources available at any site. In the latter case, we first break such bounding boxes into one for the table boxes and perhaps several with everything else, and repeat the process. If a table box still exceeds the resources available at any site, then we replicate the table and break the corresponding bounding box apart into two and repeat the process. At this point, all tables and corresponding read/write boxes are allocated and the first phase of the allocation is completed.

In the second phase, we compute the CPU slack that resulted from the first phase, and then distribute the remaining (regular) boxes by iteratively allocating boxes to sites with slack, which connect directly to a box already allocated to that site.

### 5.3 Dynamic Optimization

The job of the run-time components is to monitor the diagram and continually improve the initial allocation based on observed run-time statistics.

### 5.3.1 Local Optimization

The local optimizer can apply a variety of "local" tactics when triggered by the local monitor. If there is an overload situation, then it will initiate load shedding to (temporarily) deal with the situation. Of course, it can always explore conventional optimization techniques, including changing the order of commuting operators and using alternate operator implementations.

A more interesting local optimization opportunity exists when scheduling boxes. Unlike Aurora that could evaluate QoS only at outputs and had a difficult job inferring QoS at upstream nodes, Borealis can evaluate the QoS score function on a message at any time. By comparing the average QoS scores between the inputs and the outputs of each box, Borealis can compute the average *QoS Gradient* for each box, and then schedule the highest scoring message for the box with the highest QoS Gradient. Making decisions on a per message basis does not scale well; therefore Borealis borrows Aurora notion of train scheduling [12] of boxes and tuples to cut down on scheduling overhead.

Unlike Aurora, which insisted that messages be processed through boxes in order of arrival, Borealis has further box scheduling flexibility. In Borealis, it is possible to delay messages (i.e., take them out of order) since we can use our revision mechanism to process them later as insertions. An interesting side effect of this out-of-order approach is that the amount of revision history is bounded. Hence, a message that is delayed beyond this bound will by necessity be dropped. Thus, priority scheduling under load has an inherent load shedding behavior. The above tactic of processing the highest QoS message from the queue before the box with highest QoS gradient may generate substantial revision messages as well as shedding load. It is plausible that this kind of load shedding is superior to the Aurora-style drop-based load shedding because a delayed message will be processed if the overload subsides quickly. Hence, it is more flexible than the Aurora scheme. However, there is a cost to using revisions; hence out-of-order processing can be turned on or off by the DA. If it is turned off, then conventional "drop-based" load shedding must be performed. Also, for queries with stateless operators and when all revisions are in the form of insertions, revision processing behaves no different than regular Aurora processing. In such cases, explicit drop insertion to discard tuples with low QoS values should be used for load shedding.

### 5.3.2 Neighborhood Optimization

The actions taken by the neighborhood optimizer in response to local resource bottleneck or an optimization opportunity are similar — both scenarios involve balancing resource usage at the involved sites. The optimizer will communicate with each of its neighboring upstream or downstream sites and ask what their corresponding resource metrics are. The optimizer will then sort its "edge" boxes (which are easily slide-able [15] to a neighboring site) by decreasing resource cost. It will then migrate these boxes to neighboring sites, starting from the site with the highest resource slack. It continues to perform this greedy algorithm until there are no neighboring sites with higher resource slack.

An interesting situation arises when the neighboring nodes do not collectively have sufficient slack resources to deal with the extra load. Since there is no solution possible in the neighborhood, the overload will likely persist unless input rates change or the global optimizer takes an action. Meanwhile, it is at least desirable to move load shedding from the bottleneck site to an upstream site, thereby eliminating extra load as early as possible, before investing further resources to them.

### 5.3.3 Global Optimization

The job of the global optimizer is to react to messages from the end-point monitors indicating a specific problem with a Borealis output or a bottleneck situation at some neighborhood.

The global optimizer knows the allocation of boxes to sites and the statistics from the local monitors. From this information, it can construct a site list of the intermediate sites through which messages are routed from the data sources to the output. The optimizer will then take appropriate action depending on the nature of the problem.

**Lifetime problem.** If the problem is related to sensor lifetime (i.e., power), then messages can be dispatched to the appropriate proxies. These proxies can either initiate operator movements between the sensor and the server networks (by moving data reducing operators to the sensor network and data producing operators out of the sensor network), or by reducing sensor sampling (and transmission) rates.

**Coverage problem.** Coverage decreases if tuples get dropped during wireless transmission inside the sensor network or during load shedding inside the server network. In the former case, sensor proxies can move operators that incur high inter-node communication (e.g., a distributed join) out of the network. If this solution is not sufficient, the optimizer then notifies sites in the site list iteratively (in increasing order of distance from the data source) to decrease their load shedding on the relevant path of boxes.

**Throughput problem.** The optimizer attempts to locate the throughput bottleneck by searching backwards from the output, looking for queues (to operators or network links) that are growing without bound. Once the optimizer finds such a queue (and a site), it examines local statistics for the site, checking for inadequate resource slack. If the problem is the CPU, the optimizer will identify a nearby site with CPU slack and initiate load movement by communicating with the relevant neighborhood optimizers. Load migration will then take place as discussed in Section 5.3.2. In the case that the global optimizer identifies a path of intervening sites that execute boxes that connect the sender to the recipient, then a shuffle of boxes over the path must occur. If the problem involves I/O resources, then the global optimizer runs the table allocation algorithm from Section 5.2 using current statistics to correct the I/O imbalance. If the problem is network bandwidth, a message is sent to the site at each end of the network link whose queue is growing without bound. If either site can identify a lower bandwidth cut point, then a corresponding box movement can be initiated.

In all resource bottleneck scenarios, there may be no mechanism to generate improvement. If so, the global optimizer has no choice but to instruct one or more sites to shed load. If the QoS function is monotonically increasing with the processing applied to a tuple, then load shedding should be applied at a data source (i.e., at the sensor proxy). However, QoS will not be monotonic if there is downstream processing that can provide semantically valuable information about the message. In this case, the global optimizer can look through the statistics to identify the box with minimum average QoS as the load shedding location and contact the corresponding site.

**Latency problem.** If the problem is latency, a similar algorithm is used as for throughput. The difference is that latency is additive along the latency critical path so finding and fixing inadequate CPU, I/O, or network slack on any site on this path will improve latency. For this reason, there is no need to perform improvements starting at the endpoint and working backwards. However, a backwards path traversal is still necessary to isolate the latency critical path (binary operators join and re-sample often constantly wait for inputs from one branch; improving the latency of the other branch will have no observable effect at the output).

In the case that no information is available from the end point monitor concerning the source of the problem, then the global optimizer has no choice but to try the above tactics in an iterative fashion, hoping that one of them will work and cause improvement. Admittedly, it is entirely possible that improving one bottleneck will merely shift the problem to some other place. This "hysteresis effect" may be present in Borealis networks, and it is a challenging future problem to try to deal with such instabilities.

### 5.4 Scalability

The algorithms in the preceding sections are all practical and should scale easily to hundreds of sites. We intentionally avoided any algorithms that entail significant computation to achieve optimal results, as we believe that this is not a realistic goal for large-scale distributed systems. However, there is certainly a system size for which the global optimizer will become a bottleneck. If this is the case, then we must create more levels in our optimization hierarchy.

We use the term *region* to denote a collection of sites whose size is below this scalability threshold. Each region will have a regional optimizer that will run the algorithms of Section 5.3.3. Each region will also have a neighborhood optimizer that will treat each region as a (virtual) node and run the algorithms of Section 5.3.2 across neighboring regions. There will also be a global optimizer that will run the algorithms of Section 5.3.3 across regions, again treating each region as an individual node.

### 5.5 Federated Operation

The algorithms presented above assume that all sites are mutually co-operating and a single DA manages the query diagram. This assumption constrains the scalability of the approaches to a single administrative domain. To scale the global optimizer to cross administrative boundaries, we propose to leverage the mechanisms developed in Medusa [10].

In Medusa, we designed and implemented a distributed load management mechanism for federated systems, where autonomous participants handle each other's excess load in exchange for payment. Participants establish prices, availability, and performance guarantees in bilateral contracts (also called service-level agreements or SLAs). Every time a participant is overloaded, it moves some fraction of its processing to one of its partners and pays the pre-negotiated price. The goal of the approach is not to achieve optimal load balance but to ensure that all participants operate within capacity.

In Borealis, we plan to explore how participants can take advantage of their pair-wise load management contracts not only to move excess load but to actually optimize QoS. To facilitate inter-participant query optimizations, we propose a two-tier approach, where intra- and inter-participant optimizers work together. Contracted prices define the minimum local load levels for which using a partner's resources becomes cost effective. The inter-participant optimizer monitors local load and detects when these conditions arise. The inter-participant optimizer then uses performance guarantees defined in SLAs to model each partner as a local server with a given load. Similarly, prices and local load conditions define when it is profitable to accept a new task from a partner. When the inter-participant optimizer accepts a new task, it creates a local cost function for the task using the contracted performance guarantees. With this information, the intra-participant optimizer (i.e., the global optimizer) can readily incorporate the extra resources and tasks in its optimization.

## 6 Related Work

Our work relates to various past efforts in data management and distributed systems.

**Query Processing.** Borealis query processing relates to adaptivity techniques of CONTROL [23] and TelegraphCQ [14] projects. Online aggregation [24] progressively improves the query answer as more tuples contribute to the result, in a similar way to our insertion messages. Borealis can additionally delete and replace previously delivered results. Juggle [31] reorders tuples based on content, processing more interesting items first. Borealis also reorders tuples based on their QoS values. However, QoS value can be a function of both content and resource consumption metrics. In general, Borealis has wider scope than individual aggregate queries, and the focus is on effective resource management. Borealis scheduling relates to Eddies [7]. An eddy decides in which order to route tuples through operators of a query. In Borealis, the operator order for a query is fixed. A scheduler decides which tuples and which operators to prioritize in execution based on a QoS model. Borealis introduces several novel features which makes it fundamentally different from these projects, including revisions, time travel, QoS-driven multi-level optimization model, and combination of sensor and server networks under a single framework.

**Dynamic Revision of Query Results.** Revision record processing is similar to updating a view in response to an update of underlying base relation. Our approach of propagating only revision records that reflect the changes resulting from a revision is similar in spirit to incremental view maintenance [22], which confines the effect of an update to that part of the view that changes. The key difference between the two approaches is that the latter has no notion of "historical correction": an update to a base relation invalidates the previous value of the data being updated *as of the time of the update*, whereas revision records invalidate previous values of data *as of the time that data was first processed*. Borealis may therefore need to correct previously processed output and thus must be able to reason about all previously generated output, and not just that generated most recently. Additionally, unlike view maintenance, Borealis treats revisions as first-class citizens that can be processed and generated by queries. This approach is similar to that taken in Heraclitis [21] and in several rule-based active database settings [42, 16, 35], where updates are elevated to first-class (i.e., queryable) "deltas". Similar work on "querying the log" (the log can be thought of as a specialized stream of revision records) was discussed in [32], though unlike Heraclitis, this work permits the querying but not the generation of deltas.

**Distributed Optimization.** Table and replica placement problems have been studied extensively with the goal of minimizing storage, bandwidth and delay-centric access costs, particularly in the context of the data allocation problem [3, 44] and more recently content-delivery networks [26, 30]. In addition to these concerns, the initial allocation algorithm in Borealis considers placement of all operators, since stream operators must be placed in conjunction with table operators. Our model relates to elements common in several optimization problems, especially the allocation of boxes to nodes in a knapsack-style manner. The problem of load distribution and operator placement has been studied in depth in traditional distributed and parallel computing systems [34, 43, 27, 20]. In these systems, tasks are finite and are independent from one another. Optimization typically involves offloading tasks from overloaded nodes to more lightly loaded ones. In contrast, Borealis optimizes the placement of networks of operators that run continuously and interact with each other. Our optimization goals are thus more similar to routing packets through a network and optimizing the rate and end-to-end latencies. Unlike proposals for optimal routing [19], however, Borealis must consider both network and processing bottlenecks as well as variable data rates on different segments in the query diagram.

**Time Travel.** The idea of traveling in time has long been discussed. The Postgres [36] storage manager maintains a complete history of database objects by archiving the transaction log. Furthermore, it adds temporal operators to SQL allowing users to query the state of the database at any given point in the past. The Elephant file system [33] automatically retains different versions of user files. It allows users to add a time stamp tag to any pathname. If this tag is present, Elephant accesses the version of the file that existed at the specified time, allowing users to travel into the past. These approaches, however only support an asymmetric version of time travel (i.e., back in time) and do not consider data streams.

**Federated Operation.** Load balancing naturally extends to systems composed of loosely coupled autonomous participants. Existing approaches assume that participants are willing to share their resources with others and only require a mechanism to price these resources [18, 37, 41]. In contrast, in Borealis, we build on Medusa's [10] approach that favors privacy and enables participants to customize the service they offer one another. We extend Medusa to allow intra-participant optimizers to leverage customized agreements to use their partner's resources in their optimization.

## 7 Discussion and Future Plans

This paper has presented some of the challenges that must be met by the next generation of stream processing engines. We have cast these research problems in the context of the current Borealis design in order to draw out the issues and to show how they might interact. Our discussion focused on advanced capabilities that facilitate dynamic result and query modification and scalable, QoS-based resource allocation and optimization.

Thus, our vision includes a far more flexible stream processing model (revisions, time travel, and control lines) and a distribution model that dynamically reconfigures as network conditions change. This is fundamentally different from distributed query optimization in a pull-based system since for us queries do not end. In addition, our distribution model tries to unify the server-based techniques of most current SPE's with the bandwidth and power-aware processing of sensor networks.

Another part of the vision that we did not discuss here due to space limitations is recovery and high availability. High availability demands that node failure is masked by seamless handoff of processing to an alternate node. This is complicated by the fact that the optimizer will dynamically redistribute processing, making it more difficult to keep backup nodes synchronized. Furthermore, wide-area Borealis applications are not only vulnerable to node failures but also to network failures and more importantly to network partitions. We have preliminary research in this area that leverages Borealis mechanisms including connection point versions, revision tuples, and time travel.

We have already started building Borealis. As Borealis inherits much of its core stream processing functionality from Aurora, we can effectively borrow many of the Aurora modules, including the GUI, the XML representation for query diagrams, portions of the run-time system, and much of the logic for boxes. Similarly, we are borrowing some networking and distribution logic from Medusa.

With this starting point, we hope to have a working prototype within a year. This will allow us to experiment with many of the capabilities that were outlined in this paper. Our feeling is that many of the issues raised here represent deeper challenges than those that have been addressed to date.

## References


[1] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, C. Erwin, E. F. Galvez, M. Hatoun, A. Maskey, A. Rasin, A. Singer, M. Stonebraker, N. Tatbul, Y. Xing, R. Yan, and S. B. Zdonik. Aurora: A Data Stream Management System. In *ACM SIGMOD Conference*, June 2003.

[2] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A New Model and Architecture for Data Stream Management. *VLDB Journal*, 12(2), August 2003.

[3] P. M. G. Apers. Data allocation in distributed database systems. *ACM TODS*, 13(3), 1988.

[4] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, I. Nizhizawa, J. Rosenstein, and J. Widom. STREAM: The Stanford Stream Data Manager. In *ACM SIGMOD Conference*, June 2003.

[5] A. Arasu, S. Babu, and J. Widom. CQL: A Language for Continuous Queries over Streams and Relations. In *DBPL Workshop*, Sep. 2003.

[6] A. Arasu, M. Cherniack, E. Galvez, D. Maier, A. Maskey, E. Ryvkina, M. Stonebraker, and R. Tibbetts. Linear Road: A Stream Data Management Benchmark. Technical report, 2003.

[7] R. Avnur and J. M. Hellerstein. Eddies: Continuously Adaptive Query Processing. In *ACM SIGMOD Conference*, May 2000.

[8] B. Babcock, S. Babu, M. Datar, and R. Motwani. Chain: Operator Scheduling for Memory Minimization in Data Stream Systems. In *ACM SIGMOD Conference*, June 2003.

[9] B. Babcock, M. Datar, and R. Motwani. Load Shedding for Aggregation Queries over Data Streams. In *IEEE ICDE Conference*, 2004 (to appear).

[10] M. Balazinska, H. Balakrishnan, and M. Stonebraker. Contract-Based Load Management in Federated Distributed Systems. In *NSDI Symposium*, March 2004.

[11] D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring Streams - A New Class of Data Management Applications. In *VLDB Conference*, Hong Kong, China, August 2002.

[12] D. Carney, U. Çetintemel, A. Rasin, S. Zdonik, M. Cherniack, and M. Stonebraker. Operator Scheduling in a Data Stream Manager. In *VLDB Conference*, Berlin, Germany, September 2003.

[13] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, F. Reiss, and M. A. Shah. TelegraphCQ: Continuous Dataflow Processing. In *ACM SIGMOD Conference*, June 2003.

[14] S. Chandrasekaran, A. Deshpande, M. Franklin, J. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. Shah. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *CIDR Conference*, January 2003.

[15] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Çetintemel, Y. Xing, and S. Zdonik. Scalable Distributed Stream Processing. In *CIDR Conference*, Asilomar, CA, January 2003.

[16] C. Collett, P. Habraken, T. Coupaye, and M. Adiba. Active rules for the software engineering platform GOODSTEP. In *2nd International Workshop on Database and Software Engineering*, 1994.

[17] A. Das, J. Gehrke, and M. Riedewald. Approximate Join Processing Over Data Streams. In *ACM SIGMOD Conference*, June 2003.

[18] D. Ferguson, C. Nikolaou, J. Sairamesh, and Y. Yemini. Economic models for allocating resources in computer systems. In S. H. Clearwater, editor, *Market based Control of Distributed Systems*. World Scientist, January 1996.

[19] R. Gallager. A minimum delay routing algorithm using distributed computation. *IEEE Transactions on Communication*, 25(1), 1977.

[20] M. N. Garofalakis and Y. E. Ioannidis. Multi-dimensional resource scheduling for parallel queries. In *ACM Sigmod Conf.*, 1996.

[21] S. Ghandeharizadeh, R. Hull, and D. Jacobs. Heraclitus: Elevating deltas to be first-class citizens in a database programming language. *ACM TODS*, 21(3), 1996.

[22] A. Gupta and I. S. Mumick. Maintenance of materialized views: Problems, techniques and applications. *IEEE Quarterly Bulletin on Data Engineering; Special Issue on Materialized Views and Data Warehousing*, 18(2), 1995.

[23] J. M. Hellerstein, R. Avnur, A. Chou, C. Olston, V. Raman, T. Roth, C. Hidber, and P. Haas. Interactive Data Analysis: The Control Project. *IEEE Computer*, August 1999.

[24] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online Aggregation. In *ACM SIGMOD Conference*, Tucson, AZ, May 1997.

[25] J.-H. Hwang, M. Balazinska, A. Rasin, U. Çetintemel, M. Stonebraker, and S. Zdonik. A Comparison of Stream-Oriented High-Availability Algorithms. Technical Report CS-03-17, Department of Computer Science, Brown University, October 2003.

[26] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In *ACM STOC Symposium*, 1997.

[27] O. Koremien, J. Kramer, and J. Magee. Scalable, adaptive load sharing for distributed systems. *IEEE parallel and distributed technology: systems and applications*, 1(3), 1993.

[28] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. The Design of an Acquisitional Query Processor for Sensor Networks. In *ACM SIGMOD Conference*, June 2003.

[29] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma. Query Processing, Approximation, and Resource Management in a Data Stream Management System. In *CIDR Conference*, January 2003.

[30] P. Radoslavov, R. Govindan, and D. Estrin. Topology-informed internet replica placement. In *WCW'01: Web Caching and Content Distribution Workshop, Boston, MA*, June 2001.

[31] V. Raman, B. Raman, and J. M. Hellerstein. Online Dynamic Reordering. *VLDB Journal*, 9(3), 2000.

[32] R. Reiter. On specifying database updates. *Journal of Logic Programming*, 25(1), 1995.

[33] D. S. Santry, M. J. Feeley, N. C. Hutchinson, A. C. Veitch, R. W. Carton, and J. Ofir. Deciding when to forget in the Elephant file system. In *ACM SOSP Symposium*, December 1999.

[34] N. G. Shivaratri, P. Krueger, and M. Singhal. Load distributing for locally distributed systems. *Computer*, 25(12), 1992.

[35] E. Simon and J. Kiernan. *The A-RDL System*. 1996.

[36] M. Stonebraker. The Design of the POSTGRESS Storage System. In *VLDB Conference*, Brighton, England, September 1987.

[37] M. Stonebraker, P. M. Aoki, W. Litwin, A. Pfeffer, A. Sah, J. Sidell, C. Staelin, and A. Yu. Mariposa: a wide-area distributed database system. *VLDB Journal*, 5, January 1996.

[38] N. Tatbul, U. Çetintemel, S. Zdonik, M. Cherniack, and M. Stonebraker. Load Shedding in a Data Stream Manager. In *VLDB Conference*, Berlin, Germany, September 2003.

[39] P. A. Tucker, D. Maier, T. Sheard, and L. Fegaras. Exploiting Punctuation Semantics in Continuous Data Streams. *TKDE*, 15(3), 2003.

[40] W3C. XML path language (XPath). `http://www.w3.org/TR/xpath`, 1999.

[41] C. Waldspurger, T. Hogg, B. Huberman, J. Kephart, and W. Stornetta. Spawn: A distributed computational economy. 18(2), 1992.

[42] J. Widom and S. J. Finkelstein. A syntax and semantics for set-oriented production rules in relational database systems (extended abstract). *SIGMOD Record*, 18(3), 1989.

[43] M. Willebeek and A. Reeves. Strategies for dynamic load balancing on highly parallel computers. *IEEE Trans. on parallel and distributed systems*, 4(9), September 1993.

[44] O. Wolfson, S. Jajodia, and Y. Huang. An adaptive data replication algorithm. *ACM TODS*, 22(2), 1997.

[45] S. Zdonik, M. Stonebraker, M. Cherniack, U. Çetintemel, M. Balazinska, and H. Balakrishnan. The Aurora and Medusa Projects. *IEEE Data Engineering Bulletin*, 26(1), March 2003.