The Design of the Borealis Stream Processing Engine

Daniel J. Abadi[1], Yanif Ahmad[2], Magdalena Balazinska[1], Uğur Çetintemel[2], Mitch Cherniack[3], Jeong-Hyon Hwang[2], Wolfgang Lindner[1], Anurag S. Maskey[3], Alexander Rasin[2], Esther Ryvkina[3], Nesime Tatbul[2], Ying Xing[2], and Stan Zdonik[2]

[1]MIT Cambridge, MA
[2]Brown University Providence, RI
[3]Brandeis University Waltham, MA

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 [14] and distribution functionality from Medusa [51]. 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 and a new fault-tolerance model with flexible consistency-availability trade-offs.

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, 16] and many papers have been published on detailed aspects of the technology such as data models [2, 5, 46], scheduling [8, 15], and load shedding [9, 20, 44]. 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 [10], 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 considerations 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. We envision the following three fundamental requirements for second-generation SPEs:

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 [44]. 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 runtime. For example, in the financial services domain, traders typically wish to be alerted of *interesting* events, where the definition of "interesting" (i.e., the corresponding filter predicate) varies based on current context and results. In network monitoring, the system may want to obtain more precise results on a specific subnetwork, if there are signs of a potential Denial-of-Service attack. Finally, in a military stream application from Mitre, 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. Although current SPEs allow applications to substitute query networks with others at runtime, such manual substitutions impose high overhead and are slow to take effect as the new query network starts with an empty state. Our goal is to support low overhead, fast, and automatic modifications.

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

Proceedings of the 2005 CIDR Conference

in most current SPE prototypes 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). 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 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 [14, 17, 32] or a sensor-heavy environment [31]. Off into the future, there will be a need for a more flexible optimization structure that can deal with a large number of devices and perform cross-network sensor-heavy server-heavy resource management and optimization. The two main challenges of such an optimization framework are the ability to simultaneously optimize different QoS metrics such as processing latency, throughput, or sensor lifetime and the ability to perform optimizations at different levels of granularity: a node, a sensor network, a cluster of sensors and servers, etc.

Such new integrated environments also require the system to tolerate various possibly frequent failures in input sources, network connections, and processing nodes. If a system favors consistency then partial failures, where some inputs are missing, may appear as a complete failures to some applications. We therefore envision fault-tolerance through more flexible consistency-availability trade-offs.

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, optimization capabilities and fault tolerance. As we will demonstrate, adding these advanced capabilities requires significant changes to the architecture of an SPE. 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 presents 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 presents our new fault-tolerance approach that leverages CP, time travel, and revision tuples to efficiently handle node failures, network failure, and network partitions. Section 7 summarizes the related work in the area, and Section 8 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 the *Storage Manager*, which is responsible for storage and retrieval of data that flows

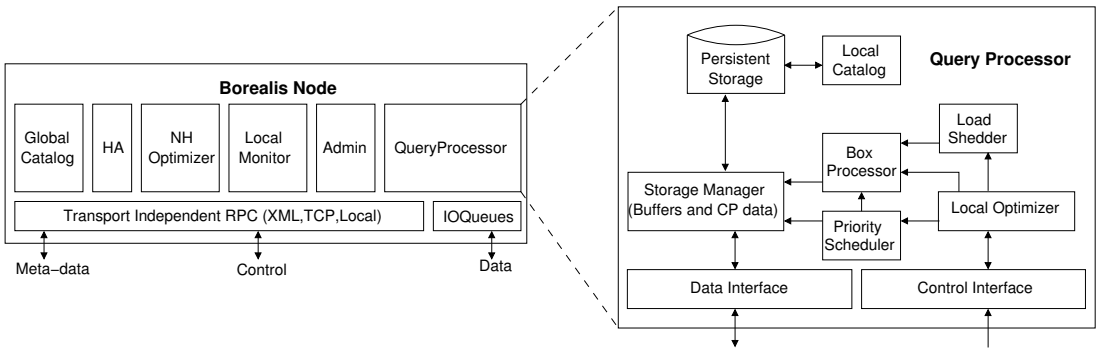

Figure 1: Borealis Architecture

through the arcs of the local query diagram. Lastly, the *Local Catalog* stores query diagram description and metadata, 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 *Global Catalog*, which may be either centralized or distributed across a subset of processing nodes, holds information about the complete query network and the location of all query fragments. 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).
- *Deletion* messages, $(-, \mathtt{t})$ such that $\mathtt{t}$ consists of the key attributes for some previously processed message.
- *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.

New applications can take advantage of this extended model by distinguishing the types of tuples they receive. Legacy applications may simply drop all replacement and deletion tuples.

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 CP views.

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 predicted 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 predicted impact of a message on QoS. 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 at system outputs. Section 5 presents our optimization techniques to achieve this goal.

3 Dynamic Revision of Query Results

As most stream data management systems, Borealis' predecessor, Aurora assumes 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 [44]. 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 [28]. 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 assumed to be an exceptional occurrence, and therefore, the replay mechanism for recovery can tolerate some run-time overhead. On the other hand, *revisions* are a part of normal processing, 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. In most cases, however, revisions on the input affect only a limited subset of output tuples, and to regenerate unaffected output is wasteful and unnecessary. To minimize 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, input revision messages comprise a small percentage (e.g, less than 1%) of all messages input to the system. 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 revision 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 messages that specify the differences between the outputs that result.

The diagram history for a box is maintained in the connection point (CP) of the input queue to that box. Clearly, it is infeasible for a query diagram to maintain an entire diagram history of all input messages it has ever seen. Therefore, a CP must have an associated *history bound* (measured in time or number of tuples) that specifies how much history to keep around. This in turn limits how far back historically 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). The replayed message will arrive at the box in its input queue, identifying itself as a replayed message, and the box 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,

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

- 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 has its own diagram history (in the CP in its input queue). 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.

3.3 Processing Cost vs. Storage

It is clear that the cost of maintaining a diagram history for every box can become prohibitive. It should be observed, however, that discrepancies in *history bounds* between boxes contained in the same query make 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.

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

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). This means, however, 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 interested in dealing with revisions. This can be specified directly as a boolean value or it can be inferred from a QoS specifica-

tion 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 [44]. 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. Control lines blur the distinction between procedures and data, allowing queries to automatically self-adjust depending on data semantics. This can be used in, for example, dynamic query optimization, semantic load-shedding, data modeling (and corresponding parameter adjustments), and upstream feedback.

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 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,

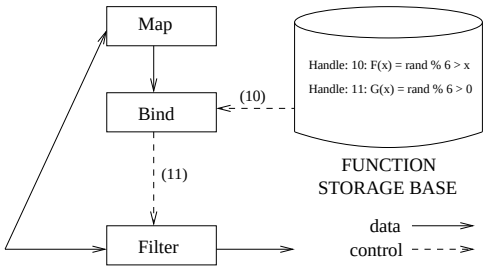

Figure 2: Control-Line Example Use

with the function definition and optionally its parameters stored in the associated record.

We expect that common practice will require parameters to a function to change at run-time. Hence a new operator is required that will bind new parameters (that were potentially produced by other Borealis boxes) to free variables within a function definition, thereby creating a new function. Borealis introduces a new operator, called *Bind*:

$$\mathrm{Bind}(B_1 = F_1, ..., B_m, = F_m)(S)$$

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. This type of automatic selectivity adjusting is useful in applications with expensive operators or systems near overload, where processing unimportant data can be costly.

Timing. Since control lines and data lines generally come from separate sources, in some cases it is desirable to specify precisely what data is to be processed according to what control parameters. In such cases, two problems can potentially occur: the data is ready for processing too late or too early.

The former scenario occurs if tuples are processed out of order. If a new control message arrives, out-of-order tuples that have not yet been processed should use the older parameters. The old parameters must thus be buffered and later applied to earlier tuples on the stream. 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 latter scenario 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 and extend *connection points* to allow for CP views and generation of revision records. These extensions are described below.

Connection Point (CP) Views. To enable time travel, we leverage Aurora's connection points [2] which store message histories from specified arcs in the query diagram. CPs were originally designed to support ad-hoc queries, that can query historical as well as real-time data. We extend this idea with *CP Views*: independent views of a connection point through which different branches of a query diagram can access the data maintained at a CP. Every CP has at least one and possibly more CP views through which its data can be accessed. The CP view abstraction makes every application appear to have exclusive control of the data contained in the associated CP. But in fact, a CP maintains all data defined by any of its associated views.

We envision that time travel will be performed on a *copy* of some portion of the running query diagram, so as not to interfere with processing of current data by the running diagram. CP views help in this respect, by enabling time travel applications, ad hoc queries, and the query diagram to access the CP independently and in parallel. A new CP view can be associated with an automatically generated copy of the operators downstream of the connection point. Alternatively, the view can be associated with a new query diagram.

Every CP view is declared with a *view range* that specifies the data from the CP to which it has access. A view range resembles a *window* over the data contained in a CP, and can either *move* as new data arrives to the CP or remain fixed. A CP view range is defined by two parameters: *start_time* and *max_time*. Start_time determines the oldest message in the view range, and can be specified as an absolute value or a value relative to the most recent message seen by the CP. Max_time determines the last message in the view range, and can also be an absolute value (when the CP view will stop keeping track of new input data) or relative to the most recent input message. A CP view that has both start_time and max_time set to absolute values is *fixed*. Any other CP view is *moving*.

A CP view includes two operations that enable time travel:

1. *replay*: replays a specified set of messages within the view's range, and
2. *undo*: produces deletion messages (revisions) for a specified set of messages within the view's range.

The replay operation enables time travel either into the past or into the future. For time travel into the past, the CP view retransmits historical messages. For time travel into the future, the CP view uses a prediction function supplied as an argument to the replay operation in conjunction with historical data to generate a stream of predicted future data. The undo operation "rewinds" the stream engine to some time in the past. To accomplish this, the CP view emits deletion messages for all messages transmitted since the specified time.

Every CP view has a unique identifier that is either assigned by the application that creates it or generated automatically. When multiple versions of the same query network fragment co-exist, a stream is uniquely identified by its originally unique name and the identifiers of the CP views that are directly upstream. An application that wants to receive the output of a stream must specify the complete identifier of the stream. For human users, a GUI tool hides these details. The system may also create CP views for purposes of high availability and replication. These CP views are invisible to users and applications.

Time Travel and Revision Records. A request to time travel can be issued on a CP view, and this can result in the generation of revision records as described below. When a CP view time travels into the *past* to some time, t, it generates a set of revision (or more specifically, deletion) messages that "undo" the messages sent along the arc associated with a CP since t.[3] The effect of an operator processing these revisions is to roll back its state to time t. The operator in turn issues revision messages to undo/revise the output since time t. Therefore, the effect of deleting all messages since time t from some CP view is to rollback the state of all operators downstream from this view to time t.

Once the state is rolled back, the CP view retransmits messages from time t on. If the query diagram is non-deterministic (e.g., it contains timeouts) and/or history has been modified, reprocessing these messages may produce different results than before. Otherwise, the operators will produce the exact same output messages for a second time.

When time traveling into the future, a prediction function is used to predict future values based on values currently stored at a CP. Predicted messages are emitted as if they were the logical continuation of the input data, and downstream operators 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 support an optional reset command that clears their state.

As new data becomes available, more accurate predictions can (but do not have to) be produced and inserted into the stream as revisions. Additionally, when a predictor receives revision messages, it can also revise its previous predictions.

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

[3] To reduce the overhead of these deletions, these messages are encapsulated into a single macro-like message.

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 data sources. As a result, Borealis faces a unique, multi-resource, multi-metric optimization challenge that is significantly different than those explored in the past.

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 refer to as *sites*. Borealis optimization consists of multiple collaborating monitoring and optimization components, as shown in Figure 3. These components continuously optimize the allocation of query network fragments to processing sites.

Monitors. 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 sensor proxies). Second, an *end-point monitor (EM)* runs at every site that produces Borealis outputs. EM evaluates QoS for every output message and keeps statistics on QoS for all outputs for the site.

Optimizers. 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. Monitoring components run continuously and trigger optimizer(s) when they detect problems (e.g., resource overload) or optimization opportunities (e.g., neighbor with significantly lower load). The local monitor triggers the local optimizer or neighborhood optimizer while the end-point monitors trigger the global optimizer. Each optimizer tries to resolve the situation itself. If it can not achieve this within a pre-defined time period, monitors trigger the optimizer at the higher level. This approach strives to handle problems locally when possible because 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.

Problem Identification. A monitor detects specific resource bottlenecks by tracking the utilization for each resource type. When bottlenecks occur, optimizers either request that a site sheds load, or, preferably, identify slack resources to offload the overloaded resource. Similarly, a monitor detects load balance opportunities by comparing resource utilization at neighboring sites. Optimizers use this information to improve overall processing performance

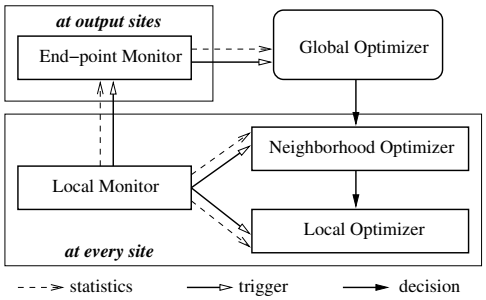

Figure 3: Optimizer Components

as we discuss in Sections 5.3.1 and 5.3.2.

Dealing with QoS is more challenging. In our model, each tuple carries a VM. These metrics include information such as the processing latency or semantic importance of the tuple. For each tuple, the score function maps the values in VM to a score that indicates the current predicted impact on QoS. For instance, the score function may give a normalized weighted average of all VM values. The local optimizer uses differences in raw score values to optimize box scheduling and tuple processing as we discuss in Section 5.3.1.

To allow the global optimizer to determine the problem that affects QoS the most and take corrective actions, Borealis allows the DA to specify a vector of weights: `[Lifetime, Coverage, Throughput, Latency]` for multiple discrete segments along these four dimensions, which indicates the relative importance of each of these components to the end-point QoS. The most interesting of these dimensions, lifetime, is the mechanism by which Borealis balances sensor network optimization goals (primarily power) with server network optimization goals. The lifetime attribute indicates how long the sensor network can last under its current load before it stops producing data. The second dimension, coverage, indicates the amount of important, high quality data that reaches the end-point. Coverage is impacted negatively by lost tuples, but the relative impact is lower if less important or low quality messages are lost. We address these issues further in Section 5.3.3. Because each of these metrics is optionally a component of the VM, the end-point monitor can keep statistics on the components that are in VM. 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.

Sensor Proxies. We assume a model for sensor networks like [31] where each node in a sensor network performs the same operation. Thus, the box movement optimization question is not where to put a box in a sensor network, but whether to move a box into the sensor network at all. This allows one centralized node to make a decision for the entire sensor network. We call this centralized node a proxy, which is located at the wired root of the sensor network at the interface with the Borealis server network. There is one proxy for each sensor network that produces stream data for Borealis. This proxy is charged

with reflecting optimization decisions from the server network into appropriate tactics in its sensor network. Furthermore, the proxy must collect relevant statistics (such as power utilization numbers and message loss rates) from the sensor network that have an impact on Borealis QoS.

In the following sections, we first describe how Borealis performs the initial allocation of query network fragments to sites. We then present each optimizer in turn. We also discuss how to scale the Borealis optimizer hierarchy to large numbers of sites and administrative domains.

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, in order to avoid potentially costly remote table operations, it is desirable to *colocate* Borealis tables with the boxes which read and write them as well as those boxes that operate on the resulting streams.

Our notion of cost here includes a combination of per-site (I/O) access costs and networked access costs, capturing latency and throughput characteristics of reads and writes to tables. Our objective is to minimize the total access cost for each table while ensuring each table is placed at a site with sufficient storage and I/O capacity. Initial diagram distribution faces several challenges in its attempt to place tables. Clearly, we must deal with arbitrary interleavings of read and write boxes operating on arbitrary tables. Interleaved access to tables limits our ability to colocate tables with all boxes that operate on their content because the boxes that use the content of one table read or write the content of another. Co-locating multiple tables at one site may not be feasible. Furthermore the consideration of diagram branches, and the associated synchronization and consistency issues, constrains the set of valid placement schemes.

We propose a two-phase strategy in approaching our initial placement problem. The first phase identifies a set of "candidate" groups of boxes and tables that should be colocated. This is based on a bounding box computation of operations on each table. Our bounding boxes are initially combined based on overlaps, and subsequently refined during our search for sites to accommodate all operations and tables within each bounding box. This search uses a heuristic to assign the most demanding (in terms of I/O requirements) bounding box, to the site with greatest capacity. We utilize a table replication mechanism to deal with scenarios where no sites have sufficient capacity. This additionally involves fragmenting any boxes operating on the table. The second phase completes the process by appropriately assigning the remaining boxes. We do so by computing the CPU slack resulting from the first phase, and then distribute the remaining boxes. We propose iteratively allocating boxes to sites with slack, which connect directly to a box already allocated to that site.

5.3 Dynamic Optimization

Starting from the initial allocation, the local, neighborhood, and global optimizers continually improve the allocation of boxes to sites based on observed run-time statistics.

5.3.1 Local Optimization

The local optimizer applies a variety of "local" tactics when triggered by the local monitor. In case of overload, the local optimizer (temporarily) initiates load shedding. The load shedder inserts drop boxes in the local query plan to decrease resource utilization. The local optimizer also explores 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 predicted-QoS score function on each message by using the values in VM. By comparing the average QoS-impact scores between the inputs and the outputs of each box, Borealis can compute the average *QoS Gradient* for each box, and then schedule 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 [15] of boxes and tuples to cut down on scheduling overhead.

Unlike Aurora, which always processed messages in order of arrival, Borealis has further box scheduling flexibility. In Borealis, it is possible to delay messages (i.e., process them out of order) since we can use our revision mechanism to process them later as insertions. Interestingly, because the amount of revision history is bounded, a message that is delayed beyond this bound will be dropped. Thus, priority scheduling under load has an inherent load shedding behavior. The above tactic of processing the highest QoS-impact message from the input queue of the box with highest QoS gradient may generate substantial revision messages and may lead to load shedding. It is possible 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. There is, however, a cost to using revisions; hence we propose that out-of-order processing be turned on or off by the DA. If it is turned off, conventional "drop-based" load shedding must be performed [44]. Also, for queries with stateless operators and when all revisions are in the form of insertions, revision processing behaves like regular Aurora processing. In such cases, the system should use explicit drop boxes to discard tuples with low QoS-impact values.

5.3.2 Neighborhood Optimization

The actions taken by the neighborhood optimizer in response to a local resource bottleneck or an optimization opportunity are similar — both scenarios involve balancing resource usage and optimize resource utilization between

the local and neighboring sites.

Other than balancing load with the neighboring sites, the neighborhood optimizer also tries to select the best boxes to move. These are the boxes that improve resource utilization most while imposing the minimum load migration overhead. If network bandwidth is a limited resource in the system, then "edge" boxes (which are easily slide-able [18]) are moved between upstream and downstream nodes. This solution is similar to the diffusion-based graph repartitioning algorithm [38]. If network bandwidth is abundant and network transfer delays are negligible, then a correlation-based box distribution algorithm [50] is used to minimize average load variation and maximize average load correlation, which will accordingly result in small average end-to-end latency. More specifically, we store the load statistics of each box/node as fixed-length time series. When determining which box to move, a node computes a score for each candidate box, which is defined as the correlation coefficient between the load time series of that box and that of the sender node minus the correlation coefficient between the load time series of that box and that of the receiver node. A greedy box selection policy chooses the box with the largest score to move first.

When neighboring nodes do not collectively have sufficient resources to deal with their load, the overload will likely persist unless input rates change or the global optimizer changes the box allocation. 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. To achieve this, the neighborhood optimizer of the bottleneck node triggers distributed load shedding by asking the upstream neighborhood optimizers to shed load, which in turn contact their parent nodes and so on.

5.3.3 Global Optimization

The global optimizer reacts to messages from the end-point monitors indicating a specific problem with a Borealis output or a bottleneck 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 list of the intermediate sites through which messages are routed from the data sources to the output. The optimizer then takes appropriate actions depending on the nature of the problem:

Lifetime problem. If the problem is related to sensor lifetime (i.e., power), the global optimizer informs the corresponding sensor proxies. These proxies 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 reduce sensor sampling (and transmission) rates. This latter solution comes with a fundamental trade-off with coverage. Slower sample rates are essentially equivalent to load shedding at the inputs and have a similar impact on QoS. Depending on the upstream operators, decreasing the sample rate can also affect throughput.

Coverage problem. Coverage problems are caused by tuples getting dropped during wireless transmission inside the sensor network, low sensor sample rates, or load shedding in 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 notifies sites in the site list iteratively (in increasing order of distance from the data source) to decrease the amount of 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 site statistics, checking for inadequate resource slack. If the problem is the CPU, the optimizer identifies a nearby site with CPU slack and initiates load movement by communicating with the relevant neighborhood optimizers. Load migration then takes place as discussed in Section 5.3.2. 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). QoS, however, is not 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 end-point and working backwards. A backwards path traversal, however, 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 and Federated Operation

Each one of the algorithms in the preceding sections is designed to operate at a different level of granularity in the system, with the global optimizer running at the highest level. There is certainly a system size, however, for which the global optimizer will become a bottleneck. To scale past that threshold, we apply the above algorithms recursively on groups of nodes, or *regions*.

We use the term region to denote a collection of sites whose size is below the scalability threshold. Each region will have a regional optimizer that will run the algorithms of Section 5.3.1. 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. Regions can be further grouped into larger regions.

The above algorithms also assume that sites are mutually co-operating. To scale optimizers past administrative boundaries, we propose to leverage the mechanisms developed in Medusa [12]. In Medusa, autonomous participants establish pair-wise contracts and handle each other's excess load in exchange for contracted payments.

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. We propose a two-tier approach, where intra- and inter-participant optimizers work together. The inter-participant optimizer monitors local load and detects when it becomes cost-effective to either use a partner's resources or accept a partner's load. Using performance guarantees defined in SLAs, the inter-participant optimizer models a partner's resources as a local server with a given load or models the partner's tasks as local tasks with given QoS functions. With this information, the intra-participant optimizer (*i.e.*, the global optimizer) can incorporate the extra resources and tasks in its optimization.

6 Fault Tolerance

In Borealis, we explore how to leverage the new CP, time travel, and revision tuple functionalities to efficiently provide fault-tolerance in a distributed SPE. As in most previous work on masking software failures, we use replication [24], running multiple copies of the same query network on distinct processing nodes.

To maximize availability, when a node detects a failure on one of its input streams, we propose that it first tries to find an alternate upstream replica. For the node to continue its processing from the new replica, however, all upstream replicas must be consistent with each other. To ensure replica consistency, we define a simple data-serializing operator, called *SUnion*, that takes multiple streams as input and produces one output stream with deterministically ordered tuples, ensuring that all operator replicas process the same input in the same order.

If a downstream SPE is unable to find a suitable upstream data source for a previously available input stream, it could either block or continue processing with the remaining (partial) inputs. The former option greatly reduces availability, while the latter option leads to a number of "wrong" results. We propose to give the user explicit control of trade-offs between consistency and availability in the face of network failures [13, 24]. To provide high availability, each SPE guarantees that input data is processed and results forwarded within a user-specified time threshold of its arrival, even if some of its inputs are currently unavailable. At the same time, to prevent downstream nodes from unnecessarily having to react to incorrect data, an SPE tries to avoid or limit the number of tuples it produces during a failure. When the failure heals, we propose that replica reprocess the previously missing information and correct the previously wrong output tuples.

To support the above model, we further enhance the streaming data model introduce in Section 2.2. Results based on partial inputs are marked as *tentative*, with the understanding that they may subsequently be modified; all other results are considered *stable*. When a failure heals, each SPE that saw tentative data reconciles its state and *stabilizes* its output by *replacing* the previously tentative output with stable data tuples forwarded to downstream clients. We believe that traditional approaches to record reconciliation [24] are ill-suited for streaming systems, and explore techniques to reconcile the state of an SPE based on checkpoint/redo, undo/redo, and the new concept of revision tuples.

Our approach is well-suited for applications such as environment monitoring, where high availability and "real-time" response is preferable to perfect answers. Some initial results are available in [11].

7 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 [26] and Telegraph(CQ) [17] projects. Online aggregation approach [27] of the CONTROL project, 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. The Telegraph project proposed several operators for adaptive query processing: the Juggle operator reorders input tuples based on their interesting content [35]; the Eddy operator reorders operators that a tuple is processed through based on change in performance [7]; and the JuggleEddy operator combines both functionalities [34]. In Borealis, the operator order for a query is fixed; however, tuples can be reordered. A priority scheduler decides which tuples and which operators to prioritize in execution based on QoS values of the tuples. The QoS value can be a function of both content (e.g., tuple importance) and resource consumption metrics (e.g., tuple latency). Our QoS gradient approach in box/train scheduling resembles the gradient-based payoff estimation approach of JuggleEddy [34].

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 [25], 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 [23] and in several rule-based active database settings [47, 19, 41], 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 [36], 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, 49] and more recently content-delivery networks [29, 33]. 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 [40, 48, 30, 22]. 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 [21], 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 [43] 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 [37] 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.

SPE Fault Tolerance. Until now, work on high availability in stream processing systems has focused on fail-stop failures of processing nodes [28, 39]. Some techniques use punctuations [45], heartbeats [42], or statically defined slack [2] to tolerate bounded disorder and delays on input streams. All current approaches, however, block or drop tuples when disorder or delay exceed expected bounds.

8 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, scalable, QoS-based resource allocation and optimization, as well as fault tolerance.

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. The new stream processing capabilities that we introduce also allow us to explore new fault-tolerance techniques that seamlessly mask node failures and tolerate network failures and partitions by exploiting connection points, time travel, and revision tuples.

We are currently 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 are outlined in this paper.

9 Acknowledgments

We thank Mike Stonebraker, Sam Madden, and Hari Balakrishnan for their continuous help and involvement with the project.

This material is based upon work supported by the National Science Foundation under Grants No. IIS-0205445, IIS-0325838, IIS-0325525, IIS-0325703, and IIS-0086057. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, C. Erwin, E. F. Galvez, M. Hatoun, J.-H. Hwang, 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. In *VLDB Conference*, Sept. 2004.

[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*, April 2004.

[10] H. Balakrishnan, M. Balazinska, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, E. Galvez, J. Salz, M. Stonebraker, N. Tatbul, R. Tibbetts, and S. Zdonik. Retrospective on Aurora. *VLDB Journal, Special Issue on Data Stream Processing*, 2004. to appear.

[11] M. Balazinska, H. Balakrishnan, S. Madden, and M. Stonebraker. Availability-consistency trade-offs in a fault-tolerant stream processing system. Technical Report TR974, MIT, November 2004.

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

[13] E. A. Brewer. Lessons from giant-scale services. *IEEE Internet Computing*, 5(4):46–55, 2001.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

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

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

[22] M. N. Garofalakis and Y. E. Ioannidis. Multi-dimensional resource scheduling for parallel queries. In *ACM SIGMOD Conference*, 1996.

[23] 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.

[24] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *ACM SIGMOD Conference*, June 1996.

[25] A. Gupta and I. S. Mumick. Maintenance of materialized views: Problems, techniques and applications. *IEEE Data Engineering Bulletin*, 18(2), 1995.

[26] 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.

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

[28] J.-H. Hwang, M. Balazinska, A. Rasin, U. Çetintemel, M. Stonebraker, and S. Zdonik. High-availability algorithms for distributed stream processing. In *IEEE ICDE Conference*, April 2005.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] V. Raman and J. M. Hellerstein. Partial results for online query processing. In *ACM SIGMOD Conference*, June 2002.

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

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

[37] 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.

[38] K. Schloegel, G. Karypis, and V. Kumar. Graph partitioning for high performance scientific simulations. In *CRPC Parallel Computing Handbook*. Morgan Kaufmann, 2000.

[39] M. Shah, J. Hellerstein, and E. Brewer. Highly-available, fault-tolerant, parallel dataflows. In *ACM SIGMOD Conference*, June 2004.

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

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

[42] U. Srivastava and J. Widom. Flexible time management in data stream systems. In *ACM PODS Symposium*, June 2004.

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

[44] 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.

[45] P. A. Tucker, D. Maier, and T. Sheard. Applying punctuation schemes to queries over continuous data streams. *IEEE Data Engineering Bulletin*, 26(1), Mar. 2003.

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

[47] 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.

[48] 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.

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

[50] Y. Xing, S. Zdonik, and J.-H. Hwang. Dynamic load distribution in the borealis stream processor. In *IEEE ICDE Conference*, April 2005.

[51] 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.