Borealis Developer's Guide

Borealis Team

May 31, 2006

Contents

List of Figures

Chapter 1

Introduction

Borealis is a distributed stream processing engine (SPE). It takes a set of streams as input, and continuously correlates, aggregates, and filters them to produce outputs of interest to applications. In Borealis, the operators that process streams can be spread across multiple physical machines, called *processing nodes* (or simply *nodes*). A separate process, the distributed catalog, holds information about the overall configuration of the system. An overview of the Borealis system and its usage can be found in the "Borealis Application Programmer's Guide" [3]. For an introduction to the main Borealis functions and features we also refer you to the overview papers [1, 2, 9]. In this chapter, we present the overall system architecture and present the outline for the rest of the document.

1.1 System Components

A Borealis system comprises the following components:

- The *Distributed Catalog* holds information about the overall system. This information includes the description of the query diagram (i.e., the set of all stream processing operators and all streams) and the deployment information, which specifies the assignment of operators to processing nodes. The architecture of the Distributed Catalog is detailed in "A Distributed Catalog for the Borealis Stream Processing Engine" [8], which also explains how client applications specify and modify the query diagram and its deployment.
- *Nodes* perform the actual stream processing. Each node runs a fragment of the query diagram and stores information about that fragment in a local catalog. The local catalog also holds information about other nodes that either send input streams into the node or receive locally produced outputs streams. Nodes collect statistics about their load conditions and processing performance (e.g., processing latency). Each node also performs the tasks necessary to manage its load and ensure fault-tolerant stream processing. We present the software architecture of a Borealis node in Chapter 2.
- *Client applications*: Application developers can write and deploy applications that provide various services within a Borealis system. A client application can create and modify pieces of the query diagram, assign operators to processing nodes, and later request that operators move between nodes. A client application can also act as a data source or a data sink, producing or consuming data streams. A Borealis system comes by default with two client applications. One client provides a graphical user interface (GUI) for an administrator to modify the running query diagram graphically. A second client, *Monitor*, is a monitoring tool that displays the current deployment and load conditions.
- *Data sources*: These are client applications that produce streams of data and send them to processing nodes. Figure 1.1 shows the overall data flow in a Borealis system.

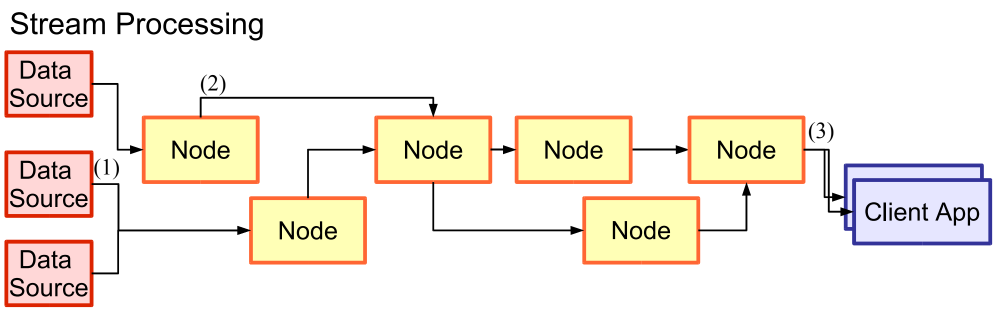

Figure 1.1: Data flow (stream flow) in a Borealis system.

1.2 Outline

In the rest of this document, we present the details of the software architecture of a Borealis processing node, including the main software components in Chapter 2, the details of the query processor in Chapter 3, the statistics manager in Chapter 4, and the fault-tolerance module in Chapter 7. We also present two global components that control groups of Borealis nodes. These components are a global load manager (Chapter 5) and a global load shedder (Chapter 6).

Chapter 2

Architecture of Borealis Node

Each processing node runs a Borealis server whose major software components are shown in Figure 2.1. We now briefly present each one of these components.

The *Query Processor (QP)* forms the core piece where actual stream processing takes place. The QP is a single-site processor, composed of:

- An *Admin* interface for handling all incoming requests. These requests can modify the locally running query diagram fragment or ask the QP to move some operators to remote QPs. The Admin handles the detailed steps of these movements. Requests can also set-up or tear down subscriptions to locally produced streams.
- A *Local Catalog* that holds the current information about the local query diagram fragment. The local catalog includes information about local operators, streams, and subscriptions.
- The input streams are fed into the QP and results are pulled through the *DataPath* component that routes tuples to and from remote Borealis nodes and clients.
- *AuroraNode* is the actual local stream processing engine. AuroraNode receives input streams through the DataPath, processes these streams, and produces output streams that go back to the DataPath. To perform the processing, AuroraNode instantiates operators and schedules their execution. Each operator receives input tuples through its input queues (one per input stream) and produces output results on its output queues. AuroraNode also collects statistics about runtime performance, data rates on streams, CPU utilization of various operators, etc. These statistics are made available to other modules through the Admin interface.
- The QP has a few additional components (such as a Consistency Manager) that enable fault-tolerant stream processing.

Other than the QP, a Borealis node has modules that communicate with their peers on other Borealis nodes to take collaborative actions:

- The *Availability Monitor* monitors the state of other Borealis nodes and notifies the query processor when any of these states change. The Availability Monitor is a generic monitoring component that we use as part of the fault-tolerance protocols. We describe the Availability Monitor in Chapter 7.
- The *Load Manager* uses local load information as well as information from other Load Managers to improve load balance between nodes. We present the Load Manager in detail in [4] and [6].

Control messages between components and between nodes go through a transport independent Remote Procedure Call (RPC) layer that enables components to communicate in the same manner whether they

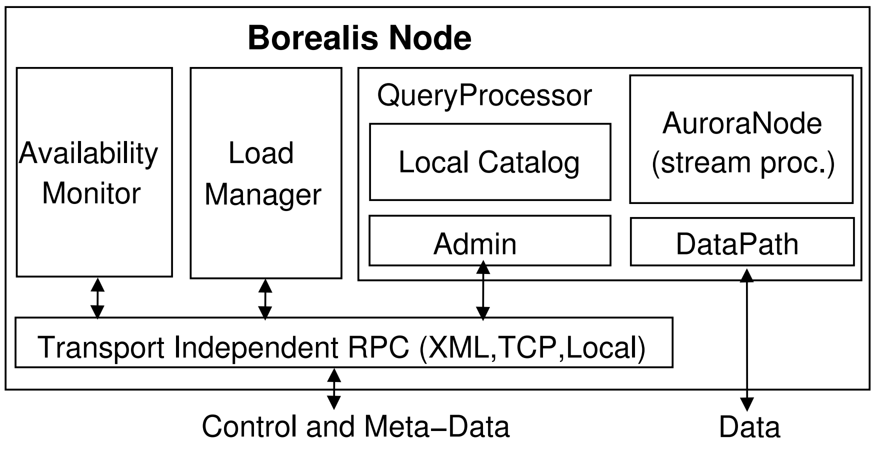

Figure 2.1: Software architecture of a Borealis node.

are local or remote. Calls are automatically translated into local or remote messages. Communication is asynchronous. The RPC layer also seamlessly supports remote clients that use different RPC protocols to communicate with Borealis.

Chapter 3

Query Processor

We now present details on the implementation of the various subcomponents of the Query Processor module. Note that this is by no means a complete description of the methods available within the Query Processor, but is simply intended to provide the reader with a basic understanding of the layout and locations of the functionality provided by the module. With this information, the reader should proceed to inspecting the codebase itself.

We start with the local catalog, describing key data structures which keep track of the locally-deployed query diagram fragment and its connections to the rest of the query diagram. We then present the mechanisms available in the Admin component to dynamically modify the locally running query diagram. Finally, we present the DataPath component which manages streams entering and exiting the node, before describing the AuroraNode component, our core stream processing engine based on Aurora [2]. As we present the AuroraNode, we discuss the implementation of the operators, streams and the scheduler, which together enable data stream processing.

3.1 Local Catalog

The local catalog maintains the acyclic graph structure of the query diagram fragment running at the node, and provides an interface to accessing this query metadata. The metadata includes schemas, operators, streams and tables, in addition to network connection information, such as the network address of upstream neighbors (i.e., the nodes that produce the input streams for the local query diagram fragment), and client subscriptions to whom query results must be sent. The local catalog's data structures are also used by other Borealis components, such as the global and regional catalogs, and encapsulates the functionality for validating a query's correctness. We describe the classes comprising the query metadata, and the internal representation of the overall query in the *Catalog* class.

Directory:	*borealis/src/modules/catalog*
Files:	*borealis/src/modules/catalog/Catalog.*{*h*|*cc*}
	borealis/src/modules/catalog/CatalogSchema.{*h*|*cc*}
	borealis/src/modules/catalog/CatalogStream.{*h*|*cc*}
	borealis/src/modules/catalog/CatalogBox.{*h*|*cc*}
	borealis/src/modules/catalog/CatalogTable.{*h*|*cc*}
	borealis/src/modules/catalog/CatalogSubscription.{*h*|*cc*}

3.1.1 Query Diagram Elements

The four primary classes that represent the elementary components of a query diagram are: *CatalogSchema*, *CatalogStream*, *CatalogBox* and *CatalogTable*. All of these objects are named, and may be accessed by name through the local catalog's interface.

- *CatalogSchema* – represents a stream's schema, and consists of a set of typed fields, internally maintained as a vector. These fields are instances of the *SchemaField* class, and track the name, type, size and offset of the field within the schema. The size and offset fields are used to access the value of a field within a tuple during expression evaluation when an operator runs.

- *CatalogStream* – represents a stream in the query diagram, and consists of an input box_port, and a set of output box_ports. A box_ports represent the connection between a stream and an operator, and is defined by the *BoxPort* class. This *BoxPort* class tracks a box_port number and a *CatalogBox*, which indicates that the stream is the nth input or output stream of the box (based on whether the box_port is an input or output box_port). Additionally, a stream is associated with a schema, and optionally an input subscription, and a set of output subscriptions. The input subscription is used whenever there is an upstream Borealis node feeding data into the stream, while output subscriptions maintain a list of clients or other Borealis nodes interested in receiving data flow through the stream. The set of output box_ports and output subscriptions are maintained internally as vectors.

  A stream may be classified as an input stream, or an output stream based on the presence of either input or output subscriptions respectively. Note that a stream may be both an input stream and an output stream. A special subclass of input streams are injected streams. An injected stream is one that is directly fed by a data source. A stream may also be classified as an intermediate stream when there are no input or output subscriptions, indicating that the stream connects two boxes local to the Borealis node.

- *CatalogBox* – represents an operator in the query diagram, and consists of an operator type (e.g. filter, map, join, etc.), a set of input and output streams and a set of parameters. The input and output streams are instances of the *CatalogStream* class described above, and the parameters are represented by the *BoxParameter* class. This class maintains a map associating named parameters to their corresponding values. For example filters have a parameter named *"predicate"* which corresponds to the filtering expression. A complete list of box types and their parameters may be found in the Borealis Application Programmer's Guide [3]. In addition to the above metadata, depending on its type, a box may access a *CatalogTable*, a HA unit, Connection Points, and may specify a processing mode (such as for revisions).

- *CatalogTable* – represents a persistent table in the query diagram, and consists of a schema, a set of keys on the schema, a set of indexed fields, and set of initialization parameters. This metadata is later used to instantiate a BerkeleyDB data store from within the AuroraNode component, providing Borealis has been configured with support for tables. Please refer to the "Borealis Installation Guide" [7] for details on configuring Borealis with support for tables.

The query diagram elements described above are internally maintained by STL maps, and are indexed by name. The following accessors allow retrieval of these objects. For brevity we omit the corresponding functions to add or to update query diagram elements to these maps.

```
CatalogSchema* Catalog::get_schema(Name schema_name);
CatalogStream* Catalog::get_stream(Name stream_name);
CatalogBox*    Catalog::get_box(Name box_name);
CatalogTable*  Catalog::get_table(Name table_name);
```

3.1.2 Catalog Validation (Typechecking)

The schemas for the output streams of a box (the *Out streams*) can be inferred from the schemas of the input streams to the box (the *In streams*), the type of the box, and the box parameters. Schemas for streams that flow beteween boxes do not need to be explicitly declared. Ensuring that all schemas are consistent is called type checking. Schemas for Out streams are inferred whenever a box is added to any component containing a local, global or regional catalog.

3.1.3 Connections Topology

The local catalog also bears the responsibility for maintaining information about streams that cross node boundaries. To do so, the catalog keepts track of all the upstream and downstream neighbors of a node. An upstream neighbor is a Borealis node that produces a stream used as input by the local query diagram. A downstream neighbor is a Borealis node or a client application that consumes a stream produced by the local query diagram.

To receive data on an input stream, a Borealis node or client application must *subscribe* itself to that stream. Hence, to keep track of all its upstream neighbors, a Borealis node keeps track of all its *subscriptions*. Similarly, to keep track of its downstream neighbors, a node keeps track of all its *subscribers*.

Information about each upstream neighbor (i.e., each subscription) and each downstream neighbor (i.e., each subscriber) is stored in the following data structure:

- *CatalogSubscription* – denotes either a subscription to an input stream produced by an upstream neighbor or a downstream neighbor subscribing to a local stream. This class consists of a stream name and an IPv4 network address (the address of the upstream or downstream neighbor).

  For fault tolerance purposes, a *CatalogSubscription* can optionally contain information about the last input tuple received by the subscriber[1], whether the subscriber received any tentative data (c.f., Chapter 7), and the gap size indicating the maximum number of missing input tuples the subscriber is willing to tolerate.

In a similar manner to the query diagram elements, the connection oriented portions of a running query are internally maintained by STL maps and are indexed by a combination of stream name, and the remote endpoint's network address. The following functions may be used to retrieve incoming and outgoing connection metadata respectively:

```
CatalogSubscription* get_subscription(InetAddress subscription_address, Name stream_name);
CatalogSubscription* get_subscriber(InetAddress subscriber_address, Name stream_name);
```

3.2 Query Processor API

While the local catalog provides a fine-grained, low-level interface to the data structures maintaining the query diagram, the *QueryProcessor* class itself provides a high-level abstraction of the operations applied to the queries, and more importantly, synchronizes the local catalog with the queries implemented by the AuroraNode stream processing core. In this section, we describe the dynamic modification interface provided by the QueryProcessor that may be used to instantiate queries or update any running queries. The programmatic interface described here is made available externally to the node through a transport-independent implementation of RPC.

Directory: *borealis/src/modules/queryProcessor*
Files: *borealis/src/modules/queryProcessor/QueryProcessor.*$\{h|cc\}$

3.2.1 Deploying a Query

Application XML for a static diagram can be passed to the Head as command line arguments or via RPC calls. The Head in turn validates the Application XML and divides it into individual Update XML elements. It then determines which element goes to which node and deploys them using the deploy method.

- `Status Diagram::deploy()`

[1]By indicating the last input tuple it received, a client switching between replicas of the same processing node continues to receive data from the correct point in the stream without loss or duplication

Directory: *borealis/src/modules/catalog/*
Files: *borealis/src/modules/catalog/Diagram.{h|cc}*

Topology changes are passed between Borealis system components as Update XML strings. Each Update XML string contains a single sub-element from the borealis or deploy elements as defined in the borealis/src/src/borealis.dtd file.

Update XML is similar to Borealis Application XML elements, but there may be differences in particular attributes. There are also additional elements not found in Application XML. These differences are described in the comments in the borealis.dtd file.

As new capabilities are added Update XML is often extended to support them. These changes are internal so changes should not affect applications and should be backward compatible.

When extending Update XML be sure to update the borealis.dtd file. The naming conventions for attributes is to use English words (no abbreviations) in lowercase letters and underscores between words. The audience for comments is both application and system developers. Since it is a reference document keep the comments clear and concise.

Both the local catalog and the AuroraNode component are populated via the Global Component, which deploys the query to the individual nodes with the following function call.

- `AsyncRPC<void> add_xml_string(string update_xml)`

Directory: *borealis/src/modules/queryProcessor/*
Files: *borealis/src/modules/queryProcessor/QueryProcessor.h,*
borealis/src/modules/queryProcessor/AdminCatalog.cc

The argument represents an XML document defining a query diagram element, according to the Borealis query DTD [REF]. The QP parses and validates the specified query element, and subsequently adds the element to the local catalog, and the AuroraNode component if necessary. The QP requires query elements to be added in an order that respects the element dependencies. For example a schema must be added before any stream that uses it. The QP does not lock the local catalog and thus does not support concurrent additions. Instead the QP relies upon the global catalog to perform any necessary serialization of the query modifications.

The reference documentation for all Update XML that can be received by a node can be found in the comments for QueryProcessor::add_xml_string and in the methods that precess each element. An additional helper method, QueryProcessor::add_xml_batch sends a vector of Update XML strings to a node.

3.2.2 Regional and Global Components

Optimizers can also send Update XML to nodes with requests to move boxes or insert load reducing “Drop” boxes. Usually these requests are made by Regional and Global Components (e.g., Load Manager described in Chapter 5, and Load Shedder described in Chapter 6), but they can also be initiated internally from local optimizers within a node. Nodes can also send Update XML between each other to process changes that apply to multiple nodes.

After succesfully applying Update XML, Nodes reflect the XML to Regional Components and the Head as required. Not all updates need to be reflected and nodes may annotate the XML with additional attributes. Updates sent from the Head to nodes do not get reflected back to the Head. Regional Components do not need all updates either; only basic elements.

The components use the reflected updates to maintain their catalogs. Both Regional and Global Components us the RegionServer::add_xml_string method in RegionServer.cc to recieve updates and apply them to their catalogs.

- `AsyncRPC<void> add_xml_string(string update_xml)`

Directory: *borealis/src/utility/client/region/*
Files: *borealis/src/utility/client/region/RegionServer.{h|cc}*

3.2.3 Dynamic Query Modifications

The top-level query modification function call presented above utilizes the following QP methods to realise the desired changes.

- `void local_connect_box( AsyncRPC<void> completion, vector<ConnectBoxData> connect_boxes)`

  This mechanism inserts an operator between two connected operators. Currently the mechanism only supports the insertion of a single-input, single-output operator (such as a single predicate filter, or a map). In order to connect the new operator, we first suspend the execution of the existing upstream operator. This ensures any future outputs of the upstream operator may be processed by the newly inserted operator. We then add the specified operator to the local catalog and the Aurora engine. During insertion to the engine, we must also take care to disconnect the existing stream connecting the two operators, and redirect it to feed the new operator. Thus the newly connected operator is set up to feed the downstream box. In a more general scenario of this mechanism's application, we must additionally update any subscriptions associated with the connecting stream. An example of this is a scenario where the stream connecting the two operators crosses a node boundary (i.e. traverses two Borealis nodes). In the current implementation, the mechanism supports operator connections to the upstream node alone.

  Examples of this mechanism's use may be found in the LoadShedder.

- `void local_disconnect_box( AsyncRPC<void> &completion, vector<Name: > box_names)`

  This mechanism disconnects a single-input, single-output operator from the query diagram, and connects the operator's input stream to its downstream operator to ensure there is no resulting orphaned operator.

- `void local_add_boxes( AsyncRPC<void> &completion, vector<CatalogBox> boxes, BoxPackage packed_boxes, set<Name: > potential_subscriptions, InetAddress prev_node)`

  This mechanism adds a set of boxes to both the local catalog and the Aurora engine. The boxes may optionally include serialized state via the *BoxPackage* argument, and a set of stream names and an endpoint to which the local node may potentially have to subscribe for the new operators' inputs.

- `void local_remove_boxes( AsyncRPC<void> &completion, vector<Name: > box_names)`

  This mechanism removes a set of boxes from the local catalog and the Aurora engine. The boxes are first removed from the Aurora engine to halt data flow to the specified boxes, prior to their removal from any data structures.

- `void local_move_boxes( AsyncRPC<void> &completion, vector<Name: > box_names, MedusaID dest)`

  This mechanism enables the migration of a set of operators from the local node, to the Borealis node specified by the *MedusaID* argument. The migration operation itself is a four-stage process, starting with the above function. The first step involves inspecting the operators that are to be moved off the local node to determine which of the operators' input and output streams may require connections between nodes. For example any local upstream operator from an operator to be moved requires a new outgoing connection from the local node. Once the new connections have been identified, we proceed to suspend the execution of the given operators and serialize any internal state. Following serialization, we add the operators to the destination node of the migration through the use of an RPC mechanism. This mechanism deserializes any internal state and injects it into the newly added operator at the destination site. At this point the first stage completes with the operators successfully set up at the destination node. We now describe three subtasks responsible for finalizing the migration process:

- `void local_move_boxes_orphan_subs(AsyncRPC<void> completion, MoveBoxData data, RPC<void> result);`

  The second stage is to set up any incoming connections from the destination node to the original source of the operators. This occurs in scenarios where an upstream box is moved to the destination, requiring a subscription to obtain the input stream for any downstream box.

- `void local_move_boxes_redirect_streams(AsyncRPC<void> completion, MoveBoxData data, RPC<void> result);`

  The third stage is to redirect any connections no longer necessary at the source site. These include two types of connections, first, input streams that were previous subscriptions at the local node, and are no longer needed since the operator they fed has been moved. Second, output streams that were previously subscribed to, and are no longer needed for the same reason. In addition to removing these connections from the local node, this function also instructs the destination node to correctly set up its connections for the same set of streams.

- `void local_move_boxes_engine_and_catalog(AsyncRPC<void> completion, MoveBoxData data, RPC<void> result);`

  Once all the connections have been set up correctly, the fourth and final stage proceeds to remove all the remaining elements at the source node. The includes any unnecessary input streams that do not require redirection, and the operators themselves. Note that our migration mechanism does not remove the operators from the source node until the operator has been successfully set up at the destination and has had all of its connections appropriately modified. Thus the mechanism ensures a valid query diagram at all stages of its functioning.

As a final note on moving operators, the system does not currently support concurrent migrations, but assumes that migration requests are serialized through the global catalog.

3.2.4 Dynamic Connection Modifications

In addition to the above modifications of the query diagram, the QP provides mechanisms to create connections between Borealis nodes, data sources and clients. These connections are manipulated via update XML processed by the top-level modification handler, which in turn invokes the following methods.

- `void local_add_subscriptions( vector<CatalogSubscription>& subscriptions, Callback<void, RPC<void> > cb)`

  This mechanism adds a subscription for a set of incoming input streams at the local node. The sources of the streams should be specified as the endpoint of each *CatalogSubscription* object. We explicitly notify the upstream source of our intention to become a subscriber, prior to adding the information in the local catalog. Upon successful setup, we invoke the given callback function.

- `void local_remove_subscriptions( AsyncRPC<void> completion, vector<CatalogSubscription*>& subs)`

  This mechanism removes a set of subscriptions for input streams from the local node. We explicitly notify the upstream source of the removal prior to deleting from the local catalog.

- `void local_subscribe_streams( AsyncRPC<void> &completion, vector<CatalogSubscription> &subs)`

  This mechanism adds a set of subscribers to the specified output streams from the local node. The local node must contain the operator that feeds the given stream, except in the case that the stream is an injected stream.

- `Status local_subscribe_stream( CatalogSubscription sub, Boolean *forward)`
  This mechanism adds a single subscriber to the the specifed output stream at the local node by adding the stream as an output path in the DataPath module, and to the local catalog.
- `void local_unsubscribe_stream( AsyncRPC<void> completion, CatalogSubscription& subs)`
  This mechanism removes a subscriber at the local node by removing the output path within the DataPath module and from the local catalog. The deletion of the output path occurs in a delayed manner to help ensure that any pending output tuples complete.

3.3 DataPath

The DataPath component is responsible for providing functionality to support a direct flow of tuples from the network interface to our stream processing engine. As such, the DataPath implements the "glue" between TCP sockets and the AuroraNode component described below. The DataPath is designed in an event-driven paradigm, relying on the NMSTL library [REF] to handle low-level network connectivity and data transfer. The DataPath uses two additional threads to accomplish its job, a thread to enqueue tuples arriving on the network to the AuroraNode stream processor and a thread to dequeue tuples once they have been successfully processed and route them to their network destination. Both the enqueue and dequeue threads make use of IOQueues, a data structure that maintains information on the state of each input and output stream to Aurora, and tracks the content of tuples for an application-level retransmit protocol to cope with networked data loss, amongst other fault tolerance mechanisms.

Directory:	*borealis/src/modules/queryProcessor/*
Files:	*borealis/src/modules/queryProcessor/DataPath.{h|cc}*
	borealis/src/modules/queryProcessor/IOQueue.{h|cc}

The *DataPath* class itself maintains a set of *IOQueue* objects associated with each input stream on which data sources may append tuples, and output streams on which clients expect processed results. We refer to these IOQueues as input and output paths. The DataPath provides mechanisms to dynamically add and remove input and output paths, based on the queries implemented by the underlying stream processor. For example, operator migration may change the set of input and output streams at a Borealis node, thus requiring redirection of input and output paths.

- `void add_input_path(Name stream_name, InetAddress source);`
  `void remove_input_path(Name stream_name);`
- `void add_output_path(Name stream_name);`
  `void remove_output_path(Name stream_name);`

We now describe the key methods in the *IOQueues, EnqueueThread, DequeueThread and FastDequeue-Forwarder* classes.

- *IOQueue*
  - `void buffer(char* buf, int length, size_t tuple_size);`
  - `bool inspect_data(char* buf, int length, size_t tuple_size);`
- *EnqueueThread*
  - `Status do_handle_input_event( ptr<StreamEvent> event);`
- *DequeueThread, FastDequeueForwarder*
  - `void DataPath::DequeueThread::run()`
  - `Status FastDequeueForwarder::send_data(Name stream_name, ptr<StreamEvent> event);`
  - `Status FastDequeueForwarder::update_client(ClientInfo& client_info, Name stream_name);`

3.4 AuroraNode

The AuroraNode component encapsulates the Aurora stream processing engine at the heart of Borealis. The engine is responsible for maintaining the operator implementations, given the query specifications in the local catalog. The engine routes data tuples as they arrive from the network on the DataPath, using streams to directly place the tuples onto operators' queues. The Aurora scheduler maintains a list of operators to execute, inserting operators into this list whenever their queues contain inputs. During its execution by the scheduler thread, the operator retrieves tuples from its queues, and evaluates expressions to transform data tuples, according to its type. Subsequently the operator enqueues any results on its output stream, resulting in data flow, and a repetition of the basic push-based operator execution cycle. In this section, we describe the implementations of the streams, operators and expressions, and the scheduler components briefly outlined above.

Directory: *borealis/src/modules/queryProcessor/runtime/*
Files: *borealis/src/modules/queryProcessor/runtime/AuroraNode.*{*h*|*cc*}
borealis/src/modules/queryProcessor/runtime/QBox.{*h*|*cc*}
borealis/src/modules/queryProcessor/runtime/Stream.{*h*|*cc*}

3.4.1 Operators

Operators in the Aurora engine inherit the *QBox* class, our abstract operator representation that defines a set of common methods for manipulating any connected streams and queues, amongst other implementation oriented data structures. Furthermore, the *QBox* class defines three virtual functions that a concrete operator must implement, functions intended to set up the operator, initialise any internal state, and provide an execution entry point for the scheduler. In addition to describing these fields and methods of the *QBox* class, we present the implementation details behind the simple expression language used in operator parameters and outline how an operator developer may evaluate such expressions.

Operator Construction and Declaration

The Aurora engine uses a dictionary structure, known as the *Registry* class to instantiate concrete operators. Thus within all operators must register themselves with the registry in their class definition. We provide two macros to simplify this process:

- `AURORA_DECLARE_QBOX(`*Class name, box type name*`)`, this is used inside the operator's class definition to instantiate private class variables used during for registration.
- `AURORA_DEFINE_QBOX(`*Class name, box type name*`)`, this is used in a global scope (for example in a .cc file) to initialise static variables used during registration.

Operator members

The *QBox* class provides access to the basic query diagram metadata through its associated representation in the local catalog (i.e. the corresponding *CatalogBox*). These accessor functions provide similar functionality to the local catalog's accessors to components within the engine. Additionally, the *QBox* class contains the input streams and queues for these streams to push tuples into for the operator to process, as well as the output streams along which to send results.

TODO: explain box_port numbers.

- `TupleQueue* get_input_queue(unsigned int input);`

  This function returns the input queue present at the given box_port number. The box_port number is an 1-based array index of input streams to the operator.

- `Stream& get_input(unsigned int n);`

  This function returns the Stream implementation at the given box_port number.

- `Stream& get_output(unsigned int n);`

  This function returns the *nth output stream.*

User-implemented Methods

Each operator inherting from the *QBox* class must implement the methods listed below to perform the desired processing. As their names suggest, these functions are called by Aurora at different stages of an operator's lifetime.

- `void setupImpl() throw( AuroraException );`

  This function should determine the output schemas of a particular operator, given its input streams and parameters specified in the operator instantiation. Once it has inferred all output schemas, the function should set the stream schemas of all of its output streams.

- `void initImpl() throw( AuroraException );`

  This function is intended to provide the operator developer an opportunity to initialize all of the operator's internal data structures. For example a group-by aggregate may set up a hash table of windows within this function.

- `void runImpl(QBoxInvocation& inv) throw( AuroraException );`

  This function implements the actual processing performed by an operator, and is continually called by the scheduler whenever inputs are available on the operator's queues. The *QBoxInvocation* class provides a method to dequeue tuples from a given input stream, allowing the user to subsequently define any data stransformation on the tuples. The implementation in the *QBox* class simply invokes the implementation in any subclass.

User-Defined Operator Skeleton

In this section we provide an (incomplete) outline of the implementation of a user-defined operator. The interested reader should refer to the existing operators for a concrete example, the purpose of this section is simply to provide a flavor of the process of definiing a custom operator.

```
--- File: MyBox.h ---
class MyBox : public QBox
{
  protected:
    void setupImpl() throw (AuroraException)
    {
        // Retrieve parameters...
        string my_param = param(‘‘my-param’’, PARAM_REQUIRED);

        // Set up expression context...
        ExprContext input_ctxt;
        input_ctxt.set_context_schema(get_in_schema(0));

        // Parse expression parameters...
        ptr<Expression> output_expr = Expression::parse(my_param, input_ctxt);

        // Construct output stream description...
        TupleDescription out_td;
        ...

        // Set up output stream schemas
        set_out_description(0, out_td);
    }

    void initImpl() throw (AuroraException) {
        // Initialize local structures
        tuple_size = TupleDescription(get_in_stream(0)).get_size();
    }

    void runImpl(QBoxInvocation& inv) throw (AuroraException)
    {
        DeqIterator di = deq(0);
        EnqIterator ei = enq(0);

        // Dequeue any inputs available...
        while ( inv.continue_dequeue_on(di, 0) ) {
            Tuple input = Tuple::factory(di.tuple());

            // Process tuple
            ...

            // Send outputs downstream...
            memcpy(ei.tuple(), output, tuple_size);
            ++ei;
        }

        // Downstream operator scheduling notification.
        get_output(0).notify_enq();
    }

    AURORA_DECLARE_QBOX(MyBox, ‘‘mybox’’);
};

--- File: MyBox.cc ---
AURORA_DEFINE_QBOX(MyBox, ‘‘mybox’’);
```

Operator Instantiation

The Aurora engine maintains data structures allowing access to the implementation objects corresponding to the streams and operators metadata in the local catalog. For reference, we briefly list the methods used in the second stage of dynamically modifiying the running query diagram, which instantiate new implementation objects. The first stage, which modifies the catalog data structures was described above in section [REF].

- `void connect_box( PortSwaps boxes_and_ports_swaps, BoxPackage packed_boxes ) throw( AuroraException );`
  `void disconnect_box( vector<CatalogBox*> boxes ) throw( AuroraException );`

  These functions create and delete *QBox* and *Stream* objects as described in the catalog metadata given as input arguments. Additionally, these functions manipulate box_ports defining the streams through which operator queues are fed, swapping them as specified in the function's arguments. These box_port swaps are applied by a scheduler task (described in the scheduler section below) so that no data flow occurs until the box_port swap is complete, and the operator implementations have been added. Since scheduler tasks run asynchronously from the view of the calling thread, the caller should not rely upon the existence (or removal) of the operator in Aurora upon this function's return.

- `void add_box( vector<CatalogBox*> boxes, BoxPackage packed_boxes )`
  `throw( AuroraException );`
  `void remove_box( vector<CatalogBox*> boxes, set<Name:  > streams_to_preserve)`
  `throw( AuroraException );`

  These functions create and delete *QBox* and *Stream* objects based on their description in the catalog. These mechanisms use a scheduler task, to ensure that the modification happens asynchronously in a manner that is atomic with respect to running operators. Thus the function will return prior to the actual creation or deletion of the operator, and the caller should rely upon its existence in (or removal from) Aurora at the point of return.

In addition to modifiying the query diagram, other components may control the execution state of operators, for example suspending an operator from running, or preventing a data stream from enqueueing onto an operator's queues (referred to as *choking*). A description of these mechanisms may be found in the scheduler section below.

3.4.2 Streams

The *Stream* class in the Aurora engine implements the actual dataflow between operators, physically copying over the tuples produced by an operator, to any downstream operator's queues. The dataflow is implemented in a multicast fashion, and performs common expression elimination.

- `void add_queue(TupleQueue* q);`
  `void remove_queue(TupleQueue* q);`
- `void enqueue_tuple();`
  `void push_tuple( const void *_buf );`
- `void* EnqIterator::tuple() const;`
  `EnqIterator& EnqIterator::operator++();`
- `void notify_enq();`
- `void begin_buffering();`
  `void stop_buffering();`

3.4.3 Expressions, Functions and Aggregates

Expressions in Borealis form a simple interpreted language, implemented with the aid of the ANTLR tool [REF]. Borealis expressions include simple arithmetic, boolean and comparison operators, conditionals, numerical functions and a small subset of SQL (the SELECT, INSERT, UPDATE and DELETE commands). Additionally users may define their own functions to be interpreted by the expression language. Details of the predefined expression language may be found in the Borealis application developer guide [REF]. In this section we focus on the data structures used for parsing and interpreting expressions, and outline the process of defining a new expression function.

Directory: *borealis/src/modules/queryProcessor/expr/*
Files: *borealis/src/modules/queryProcessor/expr/expr.g*
borealis/src/modules/queryProcessor/expr/Expression.{h|cc}
borealis/src/modules/queryProcessor/expr/ExprUtil.{h|cc}
borealis/src/modules/queryProcessor/expr/NArgs.{h|cc}

Grammar and Parsing

The expression grammar may be found in the *expr.g* file listed above. This grammar is implemented in the using the ANTLR syntax, whose details may be found in the ANTLR documentation. The ANTLR tool generates a lexer and recursive-descent parser to construct the expression's syntax tree. Our grammar contains definitions for several types of expressions, each of which has an associated parser available. These include:

- `static ptr<Expression> Expression::parse(string expr, const ExprContext &ctxt) throw(ExprException);`

  Used to parse arithmetic, boolean, comparison, conditional and function call expressions.

- `static ptr<Aggregate> Aggregate::parse(string expr, const ExprContext& ctxt) throw (ExprException);`

  Used to parse aggregate function expressions, which includes the aggregate function itself, and the expressions representing the arguments to the aggregate.

-
  ```
  static ptr<SQLSelect> parse(string expr, const ExprContext& ctxt, string table_name)
  throw (ExprException);
  static ptr<SQLInsert> parse(string expr, const ExprContext& ctxt, string table_name)
  throw (ExprException);
  static ptr<SQLUpdate> parse(string expr, const ExprContext& ctxt, string table_name)
  throw (ExprException);
  static ptr<SQLDelete> parse(string expr, const ExprContext& ctxt, string table_name)
  throw(ExprException);
  ```

  Used to parse SQL statements for accessing tables. This includes any where clauses and field expressions present as subexpressions of the SQL statements.

In addition to the textual expression, all of the above functions require a context as an argument, implemented by the *ExprContext* class. Our context class defines the schemas and their fields bound in the scope of the expression. The context may consist of an unnamed schema, allowing direct use of field names, or a set of named schemas, allowing use of field names providing they are prepended with the schema name. The following *ExprContext* accessors aid in manipulating contexts:

-
  ```
  void ExprContext::set_context_schema(CatalogSchema desc);
  const CatalogSchema* ExprContext::get_context_schema(CatalogSchema desc);
  ```

  These accessors set and retrieve an unnamed schema in the context. For example, if the unnamed schema has a field named *a*, an expression parsed in the context may be of the form "a < 10".

-
  ```
  void ExprContext::set_named_schema_info( string name, CatalogSchema *desc,
                                          unsigned int index);
  void ExprContext::get_named_schema_info( string name, const CatalogSchema *&desc,
                                          unsigned int &index) const;
  ```

  These accessors set and retrieve a named schema in the context, and associate the schema with an index number. The index number is a zero based number and defines an order on tuples present in the context for expression evaluation purposes. Accessing a field of a named schema is accomplished through the "." operator. For example, if a schema named "left" is added to the context, and the schema has a field named *a*, an expression parsed in the context may include "left.a < 10".

AST Implementation

The AST representation of expressions in Borealis is abstractly represented by the *Expression* class. This class is inherited by various classes specialized to support different types of expressions, for example *FieldExpression*, *ConstantExpression*, *PlusExpression*, *EQExpression* amongst many others. The complete list of classes may be found in the *ExprUtil.h* file listed above. Expressions consist of functions to return their type and size, and support coercion to other types. These functions are:

- `DataType Expression::getType();`
- `int32 Expression::getLength();`

For more details on expression coercion, please refer to the *Expression.h* file.
TODO: Expression type hierarchy.

Evaluating Expressions

Evaluating expressions requires an evaluation context, implemented by the *EvalContext* class. The context must be initialized to contain any tuples used in the *ExprContext* within which the expression was parsed. Furthermore the position of the tuples in the *EvalContext* must correspond with any indices specified in the *ExprContext*. With this context, expressions may be evaluated by using the following methods.

- `template<typename T> Expression::eval( EvalContext& ctxt );`
- `static void Expression::eval_vector_into( const vector<ptr<Expression> &exprs, char *buf, const EvalContext &ctxt ) throw( EvalException );`

User-defined Functions and Aggregates

Files:	*borealis/src/modules/queryProcessor/expr/NArgs.{h|cc}*
	borealis/src/modules/queryProcessor/expr/StandardFunctions.cc
	borealis/src/modules/queryProcessor/expr/Aggregate.{h|cc}
	borealis/src/modules/queryProcessor/expr/StandardAggregates.{h|cc}
Directory:	*borealis/src/external/udb*

A developer may extend the expression language with her own user-defined functions, and adding the implementation to the function registry. Aurora's function registry is similar to the one used for operators, and makes use of two macros to set up the function for use within expressions. These are:

- `AURORA_DECLARE_FUNCTION(`*Class name*`)`
- `AURORA_DEFINE_FUNCTION(`*Class name, function name*`)`

User-defined functions inherit the *Function* class, and must implement the virtual function *makeExpression()*. The signature of this function may be found in the *NArgs.h* file. Note this is also the location containing the declaration of the function registry. Example functions may be found in the *StandardFunctions.cc* file listed above.

Aggregate functions are designed in a different manner than regular functions, due to their association with windows. The relevant base classes for an aggregate function are the *Aggregate* and *AggregateWindow* classes. We return to the design of these classes momentarily. Aggregates functions, like operators and regular functions, must also be registered with a registry. The following macros aid in this task:

- `AURORA_DECLARE_AGGREGATE(`*Class name*`)`
- `AURORA_DEFINE_AGGREGATE(`*Class name, function name*`)`

The *Aggregate* class maintains the argument expressions and the output schema of the aggregate function. The *AggregateWindow* class captures the internal state needed by the aggregate function within any value-based or count-based window and presents the standard `init(), incr(), final()` interface for a user to customize and specify how an aggregate value is computed. The details of these functions are as follows:

- `virtual void init();`

  This method should implement any data structure initialization needed by the internal state of the aggregate function. For example the min aggregate initializes its state to the largest representable number.

- `virtual void incr(const EvalContext& ctxt) throw (EvalException);`

  This method should compute the update to the aggregates' internal state by evaluating the aggregate function's argument expressions within the given evaluation context. This context contains the tuple with which to increment state. The expressions are made available through the *Aggregate* object associated with this *AggregateWindow*. For example the min aggregate evaluates its one and only input argument, and assigns its internal state as the minimum value of this argument and the current state.

- `virtual void final(char *out_data);`

  This method should perform any finalization of the internal state, and copy the aggregated result into the memory location specified by the argument. For example the min aggregate simply copies its state, which represents the minimum value seen thus far, as the output result.

- `virtual void remove(const EvalContext& ctxt) throw (EvalException);`

  This method should define a reversal of the *incr* function defined above, and is an optional method.

- `virtual void update(OSerial& o);`
  `virtual void update(ISerial& i);`

  These methods allow a direct manipulation of the aggregate's internal state, allowing the state to either be serialized into an output object, or be populated directly from a serialized object.

As examples, the basic aggregates (min, max, sum, avg, count) may be found in the *StandardAggregates.*{*h*|*cc*} files listed above. Other examples of user-defined aggregates may be found in the user defined boxes directory.

3.4.4 Scheduler

Files: *borealis/src/modules/queryProcessor/runtime/Scheduler.*{*h*|*cc*}
borealis/src/modules/queryProcessor/runtime/PseudoScheduler.{*h*|*cc*}

The Aurora scheduler provided in the current release of Borealis is a simple single-threaded round-robin scheduler, implemented by the *PseudoScheduler* class. The scheduler's primary responsibility is to execute the query diagram operators which it does in its run loop. In addition, the scheduler provides mechanisms that other components may use to control the execution of operators (for example suspend them), and thus the data flow within the query diagram. Finally the scheduler is also responsible for collecting query statistics, and making them available to the rest of the system via the statistics manager, described in the next chapter. We now describe the implementation of these functionalities within the *PseudoScheduler*.

Run Loop

The scheduler iterates over its internal list of runnable elements following instantiation of the *AuroraNode* class, until instructed to terminate. The *PseudoScheduler* support two types of runnable objects, operators and tasks. The scheduler maintains an internal list of runnable operators as an STL set. The scheduler adds operators to this set upon notification of new tuples in the operator's queues. The notification mechanism is implemented through the use of listener objects attached to queues, and is described in more detail below. With every iteration of its run loop, the scheduler invokes the *run* (and consequently the concrete *runImpl*) method of every operator in its set. After running any operator, the scheduler collects a core set of statistics regarding execution behaviour. This is also described below.

Scheduler tasks are used primarily to alter the set of running operators in the system or to rewire their interconnecting streams. It is the scheduler's responsibility to ensure that no operators are running during these modifications to the query diagram, and thus to guarantee the correctness of data flow. In the current single-threaded scheduler, this requirement is met implicitly – only the single scheduler thread can run operators, so no task will conflict with any operator it modifies. Upon completing an iteration, the scheduler checks to see if there are any pending tasks and runs them.

Execution Control Mechanisms

Other Borealis components may request the scheduler alters the set of running operators through the use of several control mechanisms. These include:

- `void choke_sub_network( vector<string> boxNames );`

  This method prevents any of the given operators' input streams from enqueueing tuples onto the operators' queues, preventing any further execution of the operator. The streams are set into a buffering mode so that no tuples are lost, and any listeners on the operators' queues are suspended from emitting notifications. This function checks if the opeartor is already in the scheduler's list, and removes it if found.

- `void resume_sub_network ( vector<string> boxNames );`

  This method resumes any boxes previously choked. That is, it instructs all of the given boxes' input streams to stop buffering, and resumes any queue listeners. The boxes are then scheduled as normal, upon notification of the arrival of a new input tuple by their queue listeners.

- `void drain_sub_network( vector<string> boxNames );`

  While the *choke_sub_network* method described above prevents any new tuples being inserted into an operator's queues, it does not explicitly clear any existing contents of these queues. This is the task performed by this drain function. Draining simply involves checking for any operators with non-empty input queues, and adding these operators onto the scheduler's list of runnable operators. The thread invoking this function is blocked until the scheduler has processed all inputs available at the operators.

Note that components outside the engine encapsulation layer (i.e. those that are not members or declared friends of the *AuroraNode* class) do not have direct access to the scheduler, and thus must use the equivalient functions provided by *AuroraNode*. These functions simply wrap the above calls and provide an identical signature.

Listeners and Input Notification

Statistics Collection

Scheduler Configuration

Files: *borealis/src/modules/common/common.h*

Our scheduler supports the concept of limiting the number of tuples an operator may process before it releases its use of the CPU. By default, the tuple limit used is defined by the `TUPLE_PROCESSING_LIMIT` directive in *common.h* file listed above. Tuple limits are tracked in the *QBoxInvocation* class passed as an argument to the operator's *runImpl()* method.

Chapter 4

Statistics Manager

The Statistics Manager resides in each Borealis node's query processor. It provides an interface for different components to read, write and copy statistics from that node. Currently the statistic types we maintained are :

- Stream rates
- Tuple latency and also the square tuple latency and number of output tuples, which allows the estimation of the variance and standard deviation for the tuple latency.
- Operator cost
- Operator selectivities
- Queue length for an operator

Directory:	*borealis/src/modules/queryProcessor/statsMgr*
Files:	*borealis/src/modules/queryProcessor/statsMgr/StatsMgr.*{*h*|*cc*}
	borealis/src/modules/queryProcessor/statsMgr/Statistics.{*h*|*cc*}
	borealis/src/modules/queryProcessor/statsMgr/TSStats.{*h*|*cc*}
	borealis/src/modules/queryProcessor/statsMgr/FixLenTimeSeries.{*h*|*cc*}
	borealis/src/modules/queryProcessor/statsMgr/RWlock.{*h*|*cc*}
	borealis/src/modules/queryProcessor/statsMgr/VersionRWlock.{*h*|*cc*}

4.1 Statistics Representation

The statistics data in Borealis is represented by time series. For example, the data rate of a particular stream can be represented by (4, 3, 5, 6). Each data item in the time series is the average stream rate during a time interval. The duration of this time interval is called the *precision* of the time series, the number of samples in the time series is called the *window size*. For the previous example the number of samples is 4. The time series is implemented by the `FixLenTimeSeries` class. It can return any samples as well as the average value of the samples in a range.

Each type of statistics is contained in a `TSStats` data structure. For example, all the stream rate statistics data is contained in one `TSStats` object and all the box processing time statistics data is contained in another `TSStats` object. A `TSStats` is basically a map between object identifiers to the time series for the statistics of these objects.

All the time series that refer to the same type of statistics have the same precision and window size. Different statistics types may have different precision and window sizes. Thus the precision and window size are also specified per `TSStats`. Moreover, all the statistics of a particular type are updated at the same time. Thus we attach a timestamp to each `TSStats` data structure, which specifies when the last data sample was collected.

All types of statistics for a node are contained in a `Statistics` class. It maintains a map between statistics types to their `TSStats` data structure. This class also provides useful interface to retrieve statistics. For example, it can return all time series for a particular statistic type or return the time series for a particular object and a particular statistic type.

4.2 Access Control

The statistical information of a Borealis node, is managed by the statistics manager, implemented by the `StatsMgr` class. The `Statistics` object is a private member of the `StatsMgr` class. Thus, all other components must access the statistics through the `StatsMgr` class. The statistics manager provides a coarse grain locking mechanism for concurrent access control. Currently, the statistic of a specific type is either read locked or write locked.

If an object, such as box, stream, is added/removed to a Borealis node, then the data structures for the different statistics types of the box (*e.g.*, queue length, cost) must be added/removed to/from the statistics manager. In this case, the component that makes the change must obtain a write lock before the change and release it after the change.

If a component just wants to read the statistics or update them, then it must obtain a read lock before reading or updating and release this lock after. In addition, we implemented a special lock inside the `TSStates` data structure so that after a writer obtains a read lock from the statistics manager, it can update the data structure without waiting for the readers to finish reading the data structure. On the other hand, the readers of the statistics can only read the statistics if there is no active writers of the statistics.

In summary, there are two kinds of writers and two kinds of locks for access control. The first kind of writer adds/deletes object statistics from statistics manager and it must obtain a write lock from the statistics manager. The second kind of writer updates the statistics and it must obtain a read lock from the statistics manager. When the second kind for writers update data in a `TSStats` data structure, it must write lock the `TSStats` first.

4.3 Statistics Re-sampling

When statistics are copied from the statistics manager, they can be re-samples according to the time stamp, precision and window size of the user. The time interval [time stamp, time stamp + precision * window size] is called the *time window of the statistics*. If the statistics manager has enough statistics data that covers the time window specified by the user, then the data will be returned in the format specified by the user. However, if the statistics maintained in the statistics manager do not contain enough data, then only statistics in the time window maintained in the statistics manager is returned. For example, if a user wants to get statistics starting from time stamp 7.5, with precision 1.5 and window size 20, but the statistics maintained by the statistics manager has time stamp 10 and precision 2 and window size 5, then the returned statistics will have time stamp 9, precision 1.5 and window size 8. Note that the time stamp returned is always the specified time stamp plus a constant integer times the specified precision.

4.4 Components

4.4.1 Statistics Manager API

We provide here the main methods of the statistics manager.

- `Statistics copy_stats()`
  Copies the whole statistics structure. Locking/unlocking has been implemented inside the function.

- `TSStats copy_tsstats(StatisticsName name, int32 window_size, TimeSec precision, TimeStamp time_stamp) throw( Statistics::error )`
  Copies a particular kind of statistics (e.g. stream rate of all streams) given the statistic type. If the window size, the precision and the timestamp are zero, then the default value is used. Otherwise, the returned data are resampled according to these parameters. `Stats::STATISTICS_NAME_NOT_FOUND` is thrown if the given statistics type is not valid. Locking/unlocking has been implemented inside the function.

- `TSStats copy_tsstats(StatisticsName name, IDVect ids, int32 window_size, TimeSec precision, TimeStamp time_stamp)throw( Statistics::error )`
  Copies the `TSStats` structure for the given set of objects. Only statistics of the specified type are returned. Other parameters are the same as the previous function. `TSStats::ID_NOT_FOUND` is thrown if a given object is not valid. Locking/unlocking has been implemented inside the function.

- `void add_tsstats (StatisticsName name,int32 window_size, TimeSec precision, const IDVect &ids)`
  Adds an empty `TSStats` structure for a given statistics type for all given objects with the given window size and precision. Locking/unlocking has been implemented inside the function.

- `void add_empty_statistics (StatisticsName name,idtype id)`
  Add an empty `FixLenTimeSeries` structure for the given statistics type and object. Locking/unlocking has been implemented inside the function.

- `void delete_statistics ( StatisticsName name, const IDVect &ids)`
  Deletes the `FixLenTimeSeries` structure for the given statistics type and set of objects. Locking/unlocking has been implemented inside the function.

- `void delete_statistics (StatisticsName name,idtype id)`
  Deletes a `FixLenTimeSeries` structure for the given statistics type and object. Locking/unlocking has been implemented inside the function.

- `void write_lock_statistics(StatisticsName name)`
  Before the statistics of an object are updated, the statistics for that specific type must be write locked.

- `void write_un_lock_statistics(StatisticsName name)`

- `Statistics &get_stats()`
  Returns the statistics of the node. Must read lock before this function and read unlock after this function.

- `const TSStats &get_stats(StatisticsName name)`
  Returns all statistics for the given statistics type.

- `const FixLenTimeSeries &get_stats( StatisticsName name, idtype id)`
  Returns statistics for the given statistics type and object.

- `const double &get_last_sample(StatisticsName name, idtype id)`
  Returns last statistics sample for the given statistics type and object.

4.4.2 Statistics

This class implements a data structure that contains all statistics for a Borealis node. This data structure maps a specific statistic type to an `TSSStats` object. We briefly describe the main methods of the class, without providing their signatures, as they have similar signatures with the methods of the statistics manager. These methods are mainly used by the statistics manager.

The class provides the following API:

- retrieve statistics for all objects for a given statistics type
- retrieve statistics for a given object and statistics type
- retrieve the last statistic sample for a given object and statistics type
- copy all the statistics of a node
- resample the statistics and return the resampled copy for a given statistics type
- resample the statistics and return the resampled copy for a set of objects and a given statistics type

4.4.3 TSStats class

This class implements the Time Series structure. The most important methods of the class are:

- `TSStats copy(int32 window_size,TimeSec precision,TimeStamp time_stamp)`
  Returns a `TSStats` structure that is resampled from this `TSStats` using the given parameters.
- `TSStats copy(IDVect ids,int32 window_size,TimeSec precision,TimeStamp time_stamp)`
  Returns a `TSStats` that is resampled from this `TSStats` with the given parameters. Only the statistics for the objects in the given list are returned.

4.4.4 Fixed Length Time Series

This class implements fixed length time series for maintaining statistics. Its main API includes the following methods:

- `FixLenTimeSeries( int32 number_of_samples ) throw( error )`
  Creates a time series with a given number of samples. All samples are initialized to zero.
- `double get_sample( int32 i = 0 ) const throw( error )`
  Returns the $i-th$ sample.The samples are indexed from 0 to the number of samples minus one. The $0-th$ sample is the latest sample. By default, the latest sample is returned.
- `double get_average( double p_old = 0, double p_new = 0 ) const throw( error )`
  Returns the average value form position `p_old` to `p_new`. If `p_old` and `p_new` are zero, then the average value of all samples is returned.
- `void add_sample( double sample )`
  Adds a sample.
- `static void resample_fix_len_time_series(const FixLenTimeSeries &old_ts, double sample_period, double start_point, FixLenTimeSeries& new_ts)`
  Creates a copy of an given `FixLenTimeSeries` structure based on the given parameter.

Chapter 5

Global Load Manager

The global load manager monitors the load of each machine and moves operators the odes to balance their load. Currently, the load manager can perform three kinds of load distribution strategies:

- **Static Operator Distribution:** This strategy focuses on the static distribution of operators that are impractical to be moved on the fly (such as table read/write operators and operators with large state size). To obtain static operator distribution plans, we first run the system long enough to gather processing statistics of the operators and streams. After this warm-up period, the static operator distributor produces multiple operator deployment XML files using different operator distribution algorithms. Then the system can be restarted using one of the deployment XML files produced by the load manager. Currently, the static operator distributor can produce static operator distribution plans using the following algorithms:
  - Resilient Operator Distribution
  - Correlation-Based Operator Distribution
  - Largest-Load-First Load Balancing
  - Max-Rate Load Balancing
  - Connected Load Balancing
  - Random Operator Distribution
- **Initial Operator Distribution:** The load manager can also produce an initial operator distribution plan after the warm-up period. Unlike the static operator distribution where the system needs to be restarted, when initial operator distribution is enabled, the load manager moves operators directly after the warm-up period. Currently, the load manager uses the correlation-based algorithm to produce the initial operator distribution plan.
- **Dynamic Operator Distribution:** If dynamic operator distribution is enabled, the load manager periodically checks whether operators need to be moved on the fly based on the most recent data processing statistics. Currently, it uses the correlation-based algorithm to make operator redistribution decisions.

Directory: *borealis/tool/optimizer/loadManager*
Files: *borealis/tool/optimizer/loadManager/LoadManager.*$\{h|cc\}$

5.1 Components

The load manager consists of the following components:

- Controller
- Catalog Reader
- Statistics Reader
- Statistics Manager
- Operator Distributors

Next, we introduce each of these components in detail:

5.1.1 Controller

The controller determines what action should be taken and when. For example, it sets up timers to read catalog information and collect statistics from Borealis nodes periodically. It also determines whether static operator distribution or initial operator distribution should be performed after the warm-up period and whether dynamic operator distribution should be performed periodically.

5.1.2 Catalog Reader

The catalog reader reads query topology and deployment information from the global catalog. It fills these information in the local data structures that are used by the operator distributors to make operator distribution decision.

5.1.3 Statistics Reader

The statistics reader of the load manager collects data processing statistics from each Borealis node through RPC calls. It then fills in these information in the local statistics manager.

5.1.4 Statistics Manager

The local statistics manager stores the data processing statistics such as stream rates, operator cost and selectivities as time series. It also computes the load time series of the operators and nodes. These information are then used by the operator distributors to make operator distribution decisions.

5.1.5 Operator Distributors

Operator distributors make operator distribution decisions based on the data processing statistics. The current load manager includes a static operator distributor and a correlation-based operator distributor.

5.2 Usage

The load manager can be used as a separate process itself or it can be used as a component in the global optimizer. The executable file that runs the load manager as as separate process is:
borealis/tool/optimizer/loadManager/LoadManager

The global optimizer that runs the global load manager together with the global load shedder is:
borealis/tool/optimizer/MetaOptimizer

The configuration parameters of for the load manager can be read from a configuration file or from command line arguments. The detailed description of the configuration parameters can be found in `LMParams.h` under the load manager directory.

Chapter 6

Load Shedder

A Borealis query network may span one or more processing nodes. Each node has a certain amount of CPU capacity. During the course of system execution, one or more of these nodes may become overloaded due to increasing data rates or query workload. In this case, the load shedder kicks in to remove excess load from all of the overloaded nodes in order to avoid queueing and late query answers. Note that presence of even a single overloaded node is sufficient to cause latency increase. Therefore, it is important to make sure that all nodes operate below their processing capacity at all times.

Load is reduced by selectively dropping tuples from certain streams in the system. Load shedding will lead to approximate query results. Therefore, it must be performed in a way that would minimize the total percent data loss (or, maximize the total percent data delivery) at query end-points.

Borealis employs various alternative or complementary techniques to address the load shedding problem. These techniques are designed to operate at three different levels of the system architecture: local, neighborhood, and distributed. The current release provides an implementation where a single coordinator node makes global load shedding decisions for the complete query network. This implementation essentially provides a global optimizer which makes distributed load shedding decisions, and a local optimizer which cooperates with the global optimizer to apply these decisions at the local nodes. We next describe the implementation of these two pieces.

6.1 Global Load Shedder

Directory: *borealis/tool/optimizer/loadShedder*
Files: *borealis/tool/optimizer/loadShedder/LoadShedder.{h|cc}*

Like the global load manager, the global load shedder is a global optimizer component and is external to the core query processing modules. Therefore, it is located under `borealis/tool/optimizer`. A MetaOptimizer under this directory coordinates the operation of both the LoadManager and the LoadShedder. The MetaOptimizer also provides these components with the access to the global catalog information and the node statistics.

The global load shedder, when activated by the meta-optimizer, operates in a loop (see `LoadShedder::run()`). At each iteration, based on the most current statistics, it first generates a globally optimal plan. Then it breaks this plan into local pieces for each node and uploads these local plans to each node, by contacting their local optimization modules (see `LoadShedder::generate_lsplans()`). Finally, it sends a plan-id to each node, instructing them to apply the plan with that id (see `LoadShedder::shed_load()`).

6.1.1 Load Shedding Plans

Directory: *borealis/src/modules/common*

Files: *borealis/src/modules/common/LSPlan.{h|cc}*

The global optimizer can generate a number of global load shedding plans, each with a unique id. Each plan consists of a number of drop boxes, with specifications for their parameters and for where they should be inserted in the Borealis query network. The global optimizer then breaks each such plan into local pieces and sends these local pieces to the nodes that they belong. The local pieces preserve their global plan id's, and hence, local plans at different nodes with the same plan id correspond to the same global plan and therefore will be applied together. Plan id 0 represents the default plan, which is a load shedding plan with no drops.

Each local load shedding plan is a vector of drop boxes. Each drop box has the following structure:

```
typedef struct drop_box
{
    Name            box_name;       // Drop box name.
    string          box_type;       // "window_drop" or "random_drop".
    string          node;           // Location of the drop box.
    BoxParameter    box_params;     // Parameters of the drop box.
    Name            in_stream_name; // Existing stream into the drop box.
    Name            dest_box_name;  // Box receiving the drop box output.
    uint16          dest_box_port;  // Receiving port.
    string          dest_node;      // Location of the receiving box.
} drop_box;
```

6.1.2 Plan Generation

We model the distributed load shedding problem as a linear optimization problem. Based on the most recent system statistics and the query network topology, we formulate this problem in a form that we can then solve using the GNU Linear Programming Kit (GLPK) [10] (see `LoadShedder::construct_LP2()`). The problem variables in this case represent the fraction of the tuples at certain points in a query network (i.e., drop locations) that must be kept in the stream.

6.2 Local Load Shedder

Directory: *borealis/src/modules/optimizer/local*
Files: *borealis/src/modules/optimizer/local/LocalLoadShedder.{h|cc}*

The local load shedder is a local optimizer component. As it is implemented currently, it is not an autonomous component. Its functionality is controlled by the global load shedder. The global load shedder uploads plans to each of the local load shedders and then instructs them to apply certain plans at certain points in time. The local load shedder in turn makes the necessary dynamic modifications on the local query plan to shed load (see `LocalLoadShedder::local_apply_lsplan()`). These modifications include connecting or disconnecting drop boxes to the running query plans, or changing the parameters of already existing drop boxes in these plans. To accomplish this, the local load shedder creates the necessary XML messages to be sent to the query processor admin, which can then initiate the dynamic modifications. These XML messages have the following format:

```
<connect ( box={destination box}
           [port={zero based destination port}]
           [upstream={0 | 1}] )
         | node={node with the destination box} >
```

```
    <box name={box to insert} type={box type} >
        <in  stream={existing source stream} />
        <out stream={new destination stream} />
             :
    </box>
</connect>

<disconnect stream={new stream} />

<modify (box={box name} | table={table name}) >
    <parameter name={parameter name} value={parameter value} />
                    :
</modify>
```

6.3 Practical Issues

To run the load shedder in the system, in addition to starting up the borealis servers at each node, one should also start the MetaOptimizer at one of the nodes. Since MetaOptimizer is a global/regional component, one must also define a regional node as part of the deployment XML. The endpoint port number in this definition must be the same as the port number used when starting up the MetaOptimizer (i.e., specified with the `-d` argument, or DEFAULT_REGION_SERVER_PORT). Please see `borealis/test/developer/ls*` for examples.

Chapter 7

Fault-Tolerance

Borealis includes different fault-tolerance mechanisms. One of those mechanisms, called *Delay, Process, and Correct* (DPC) enables a distributed SPE to handle both node failures and network failures.

DPC is based on replication. Each query diagram fragment is replicated on multiple processing nodes. When a node fails, those nodes that are downstream from the failed one continue processing from a replica of the failed node. Similarly, when the communication between two nodes is interrupted, the downstream node tries to continue processing from another replica of its unreachable upstream neighbor.

When network partitions occur, a node may be unable to communicate with any replica of an upstream neighbor. In that case, DPC ensures that those input streams that remain available and can be processed are processed within an application-defined time-bound. Processing a subset of input streams leads to inaccurate results. DPC tries to minimize the number of such results produced during failures. After the partition heals, DPC corrects earlier results, while continuing to stream the most recent output data.

DPC is described in detail in [4] and [5]. We only provide a brief overview here.

7.1 User Perspective

We first describe DPC in terms of what an application writer must do to use the system.

7.1.1 Replication

In DPC, every fragment of the query diagram runs on multiple processing nodes. To replicate a fragment of a query diagram, the easiest technique is to group boxes into *queries* and assign each query to a *replica set*. A replica set is simply a group of Borealis nodes.

As an example, let's assume that we have four boxes: box$_1$, box$_2$, box$_3$, and box$_4$. We would like to run box$_1$ and box$_2$ on nodes 127.0.0.1:17100, 127.0.0.1:17200, and 127.0.0.1:17300. We would like to run box$_3$ and box$_4$ on nodes 127.0.0.1:18100, 127.0.0.1:18200, and 127.0.0.1:18300.

In the XML file that describes the query diagram, as we define the boxes, we can group them into queries:

```
...
<query name="my_query1_2">
  <box name="box1"   type="..." >
    ... [ here goes the regular definition for the box ]
  </box>
  <box name="box2"   type="..." >
    ... [ here goes the regular definition for the box ]
  </box>
</query>
```

```
<query name="my_query3_4">
  <box name="box3"   type="..." >
    ...
  </box>
  <box name="box4"   type="..." >
    ...
  </box>
</query>
...
```

In the XML file that describes the deployment, we can assign queries to replica sets instead of assigning them to individual nodes. Multiple queries can be assigned to each replica set.

```
 ...
 <replica_set  name="my_set1"  query="my_query1_2" >
    <node   endpoint="127.0.0.1:17100" />
    <node   endpoint="127.0.0.1:17200" />
    <node   endpoint="127.0.0.1:17300" />
 </replica_set>
 <replica_set  name="my_set2"  query="my_query3_4" >
    <node   endpoint="127.0.0.1:18100" />
    <node   endpoint="127.0.0.1:18200" />
    <node   endpoint="127.0.0.1:18300" />
 </replica_set>
...
```

Note that DPC currently does not handle dynamic modifications to the query diagram or its deployment.

7.1.2 Keeping Replicas Consistent

To enable downstream nodes to switch between replicas of an upstream neighbor, and continue processing inputs exactly where they left off, all replicas of the same processing node must be mutually consistent. They must process the same input tuples in the same order and go through the same execution states. To ensure such mutual replica consistency, the query diagram must be made deterministic:

- Timeout parameters are not allowed.
- All Union operators must be replaced with SUnion operators. The latter merge tuples deterministically by interleaving them in increasing tuple_stime values (tuple_stime is an attribute in the tuple headers used only by DPC.[1] Because DPC uses the tuple_stime values, applications must set these values before pushing tuples into Borealis.
- DPC also requires that data sources produce periodic BOUNDARY tuples that indicate the most recent tuple_stime value. tuple_stime values in consecutive BOUNDARY tuples must be nondecreasing.
- All Join operators must be replaced with their deterministic counterpart, SJoin. Note that SJoin is not very well tested.
- Aggregate operators must have their "independent-window-alignment" option set to true. This ensures that all replicas of the same aggregate operator will perform their computations over the same sequence of windows, independently of the exact value of the first tuple they process.

[1]Note that SUnion performs additional tasks when failures occur and heal. See [4] for details.

- The query diagram must be composed only of the following operators: SUnion, SJoin, Aggregate, Filter, and Map. These are the operators that are modified to support all the features of DPC. The modifications are very small, so adding DPC support to other operators is pretty straightforward. See [4] for details.

In Borealis, tuples have unique identifiers, but these unique identifiers were under construction when most of the DPC code was developed. For this reason, in this release, it is best if tuples on streams have one attribute in their schema that serves as a unique identifier for the tuple on the stream.

7.1.3 Handling Node and Network Failures

As long as one replica of each node is running and remains reachable, DPC masks all node and network failures. The client should only see a small delay caused by nodes switching from a failed upstream neighbor to one that is still available and reachable.

7.1.4 Handling Network Partitions

The DPC fault-tolerance mechanism is actually quite fancy. For instance, if all replicas of a node are unavailable, DPC allows Borealis to continue processing data but labels all results as *tentative*. These results are later *corrected* when the failure heals. To appreciate and use these features, please consult the detailed descriptions of the approach in [4] and [5].

7.1.5 Sample Applications

We provide three demo applications for DPC. These applications are located in

```
borealis/test/composite/fault/
borealis/test/composiste/sunion/
```

A detailed README is included in each one of those directories. It describes what the applications do exactly and how to run them. These applications demonstrate primarily what happens when a stream is completely unavailable and the system must produce tentative tuples and later corrections. This is the main feature of interest of DPC. The fault/faulttest application shows a node running a query diagram composed of many different operators. The fault/sunion application demonstrates the interactions between different sunions, including what happens when failures occur during failures or during recovery. The sunion application demonstrates the use of a replica-set and failures in a distributed deployment.

7.2 System Perspective

We now overview the implementation of DPC in Borealis. Each Borealis node runs the DPC protocol by implementing the state machine shown in Figure 7.1.

As long as all upstream neighbors of a node are producing stable tuples, the node is in the STABLE state. In this state, the node processes tuples as they arrive and passes stable results to downstream neighbors.

If one input stream becomes unavailable or starts carrying tentative tuples, a node goes into the UP_FAILURE state, where it tries to find another stable source for the input stream. If no such source is available, the node suspends processing any data for the application-defined time bound. If the failure persists passed that bound, to maintain the required level of availability, the node eventually continues processing the remaining inputs, but it marks all output tuples as "tentative" to indicate the failure condition.

A failure *heals* when a previously unavailable upstream neighbor starts producing stable tuples again or when a node finds another replica of the upstream neighbor that can provide the stable version of the stream. Once a node receives the stable versions of all previously missing or tentative input tuples, it transitions into the STABILIZATION state. In this state, if the node processed any tentative tuples during UP_FAILURE

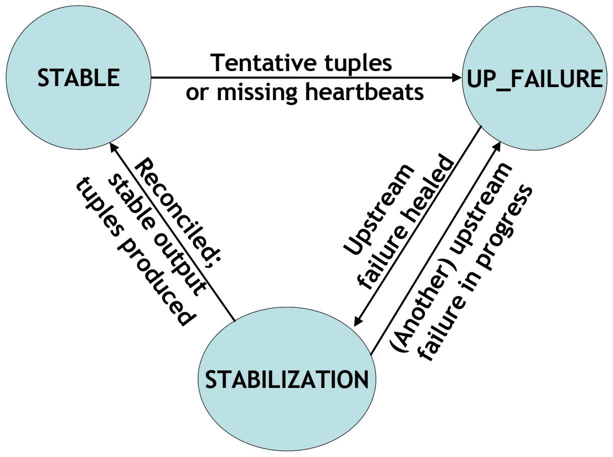
STABLE
UP_FAILURE
STABILIZATION
Tentative tuples or missing heartbeats
Upstream failure healed
(Another) upstream failure in progress
Reconciled; stable output tuples produced

Figure 7.1: DPC's state-machine.

it reconciles its state by restarting from an earlier checkpoint, and stabilizes its outputs. To stabilize its output, a node produces an UNDO tuple, a sequence of correction tuples, and a REC_DONE tuple.

7.2.1 Extended Software Architecture Overview

To run DPC, the software architecture of an SPE must be extended as illustrated in Figure 7.2. A new component, the *Consistency Manager*, is added to control all fault-tolerance related communication between processing nodes. The *Data Path*, which keeps track of and manages the data entering and exiting the node, is extended with extra monitoring and buffering capabilities. Finally, two new operators, *SUnion* and *SOutput*, are added to the query diagram to modify the processing itself. We now present the main role of each component. We discuss these components in detail in [4].

The Consistency Manager keeps a global perspective on the situation of the processing node within the system. It knows about the node's replicas, the upstream neighbors and their replicas, as well as the downstream neighbors and their replicas. The Consistency Manager therefore handles all interactions between processing nodes. It periodically requests state information from upstream neighbors and their replicas, and decides when to switch from one replica to another. Because of its global role, the Consistency Manager also makes decisions that affect the processing node as a whole. For example, it decides when to perform or suspend periodic checkpoints and when to enter the STABILIZATION state, which allows the node to reconcile its state after a failure heals. As we discuss below, the Consistency Manager delegates some of the above functions to other modules in Borealis.

The Data Path establishes and monitors the input and output data streams. The Data Path knows only about the *current* upstream and downstream neighbors. For input streams, the Data Path keeps track of the input received and its origin, ensuring that no unwanted input tuples enter the SPE. For output streams, the Data Path ensures that each downstream client receives the information it needs, possibly replaying buffered data.

Finally, to enable fine grained control of stream processing, we introduce two new operators: SUnion and SOutput. SUnion ensures that replicas remain mutually consistent in the absence of failures, and manages trade-offs between availability and consistency during network partitions. It buffers tuples when necessary, and delays or suspends their processing as needed. SUnion also participates in STABILIZATION. SOutput only monitors output streams, dropping possible duplicates during state reconciliation. Both SUnion and SOutput send signals to the Consistency Manager when interesting events occur (e.g., the first tentative tuple is processed, the last correction tuple is sent downstream). DPC also requires small changes to all operators in the SPE.

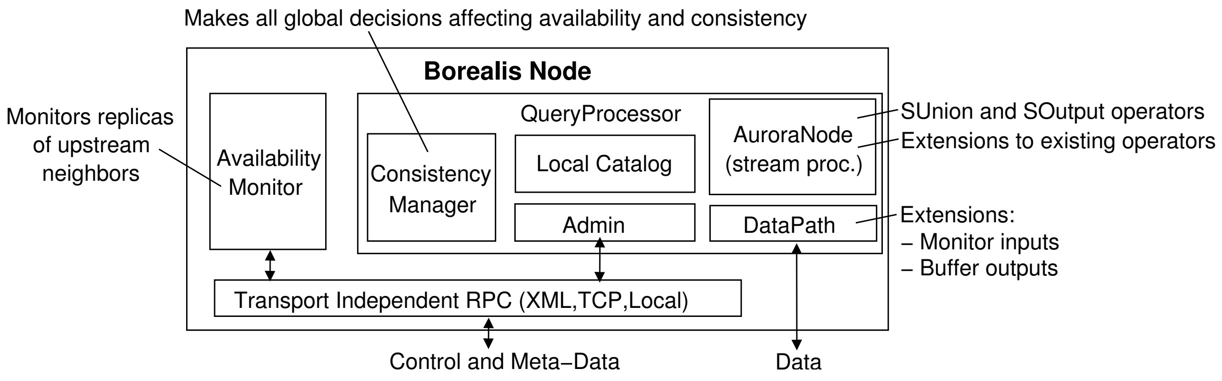
Makes all global decisions affecting availability and consistency
Borealis Node
Monitors replicas of upstream neighbors
Availability Monitor
QueryProcessor
Consistency Manager
Local Catalog
AuroraNode (stream proc.)
SUnion and SOutput operators
Extensions to existing operators
Admin
DataPath
Extensions:
– Monitor inputs
– Buffer outputs
Transport Independent RPC (XML,TCP,Local)
Control and Meta–Data
Data

Figure 7.2: **Extended software node architecture for fault-tolerant stream processing.**

7.2.2 Implementation Details

We now summarize the main functions of each component and add a few implementation details.

Consistency Manager and Availability Monitor

As we mentioned above, the Consistency Manager makes all global decisions related to failure handling:

1. For each input stream, the Consistency Manager selects the replica that should serve as upstream neighbor and optionally the one to serve as background corrections.
2. The Consistency Manager computes the states of output streams that is advertised to all downstream neighbors. In the current implementation, the Consistency Manager assigns the state of the node as the advertised state of all output streams, rather than using the more sophisticated algorithm from [4]. To determine the state of the node, the Consistency Manager uses information from SUnion and SOutput operators. These operators send messages to the Consistency Manager in the form of tuples produced on separate *control output streams*. When an SUnion starts processing tentative data, it produces an UP_FAILURE message. As soon as one SUnion goes into UP_FAILURE, the whole node is considered to be in that state. When an SUnion receives sufficiently many corrections to reconcile its state, it produces a REC_REQUEST message. Once all previously failed SUnions on input streams are ready to reconcile, the Consistency Manager triggers state reconciliation (either checkpoint/redo or undo/redo), taking the node into the STABILIZATION state. Once reconciliation finishes or another failure occurs, each SOutput sends a REC_DONE tuple downstream and to the Consistency Manager. After receiving one such tuple from each SOutput, the Consistency Manager takes the node back to the STABLE or the UP_FAILURE state.
3. After a failure heals, corrections to previously tentative tuples are performed *in the background*: i.e., downstream nodes have access simultaneously to the most recent results and to corrections. To enable both the processing of new data and the correction of old data, the Consistency Manager modules implement an inter-replica communication protocol which ensures that replicas of the same query diagram fragment never reconcile their state at the same time.

The Consistency Manager uses a separate component, the Availability Monitor, to implement the upstream neighbor monitoring protocol. The Availability Monitor observes the state of the streams produced by upstream neighbors and their replicas. The Availability Monitor informs the Consistency Manager every time the state of an observed stream changes. Indeed, such a change may require switching to another replica of an upstream neighbor. To decide when to switch and to select the new replica, the Consistency Manager needs to know both the states of the current upstream neighbors and the states of their replicas. The Availability Monitor is a generic monitoring component that could easily be extended to provide monitoring for purposes other than fault-tolerance.

The Data Path Extensions

The Data Path handles the data coming into the SPE and the data sent to downstream neighbors. DPC requires a few extensions to this component:

1. The Data Path monitors the input tuples entering the SPE. For each input stream, the Data Path remembers the most recent stable tuple and whether any tentative tuples followed the last stable one. The Data Path provides this information to the Consistency Manager when the latter decides to switch to another replica of an upstream neighbor. The information is sent to the new replica in a subscribe message.

2. The greater challenge in managing input streams is for the Data Path to handle both a main input stream and a second input stream with background corrections. The Data Path must verify that only one input stream carries stable tuples at any time. The Data Path must also disconnect the main input stream as soon as a REC_DONE tuple appears on the background corrections stream. The latter then takes the place of the main input stream.

3. The Data Path must also buffer stable output tuples (tentative and UNDO tuples are never buffered), and stabilize, when appropriate, the data sent to downstream neighbors.

Query Diagram Modifications

DPC requires three modifications to the query diagram: the addition of SUnion operators, the addition of SOutput operators, and small changes to all other operators.

SUnion: SUnion operators appear on all input streams of an SPE node but they can also appear in the middle of the local query diagram fragment. SUnions in the middle behave somewhat differently than SUnions on inputs, as we discuss in [5]. Besides differences between an SUnion placed on an input stream and one in the middle of the diagram, the behavior of an SUnion also depends on the operator that follows it in the query diagram. If an SUnion simply monitors an input stream or replaces a Union operator, it produces all tuples on a single output stream. This is not the case for an SUnion that precedes a Join operator. In Borealis, a Join operator has two distinct input streams. To enable a Join to process input tuples deterministically, the SUnion placed in front of the Join orders tuples by increasing tuple_stime values but it outputs these tuples on two separate streams that feed the Join operator. We modify the Join to process tuples by increasing tuple_stime values, breaking ties between streams deterministically. We call the modified Join a *Serializing Join* or *SJoin*.

SOutput: SOutput operators monitor output streams. Independently of the sequence of checkpoints and recovery, SOutputs ensure they produce UNDO and REC_DONE tuples when appropriate. SOutput operators propagate REC_DONE tuples to the Consistency Manager in addition to sending them downstream.

Operator Modifications: DPC also requires a few changes to stream processing operators. In the current implementation, we have modified Filter, Map, Join, and Aggregate to support DPC, and SUnion replaces the Union operator. We already mentioned the modification of the Join operator to make it process tuples in the deterministic order prepared by a preceding SUnion operator. We now outline the other necessary changes. For checkpoint/redo, operators need the ability to take snapshots of their state and recover their state from a snapshot. Operators perform these functions by implementing a packState and an unpackState method. Operators are also modified to set the tuple_stime value on output tuples, produce BOUNDARY tuples, and, upon receiving a tentative tuple, label their output tuples as TENTATIVE.

7.2.3 Implementation Limitations

Detailed in [4].

Bibliography

[1] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. S. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The design of the Borealis stream processing engine. In *Proc. of the Second Biennial Conference on Innovative Data Systems Research (CIDR)*, Jan. 2005.

[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), Sept. 2003.

[3] Borealis application programmer's guide.
`http://www.cs.brown.edu/research/borealis/public/publications/%borealis_application_guide.pdf`.

[4] M. Balazinska. *Fault-Tolerance and Load Management in a Distributed Stream Process ing System.* PhD thesis, Massachusetts Institute of Technology, 2006.

[5] M. Balazinska, H. Balakrishnan, S. Madden, and M. Stonebraker. Fault-tolerance in the Borealis distributed stream processing system. In *Proc. of the 2005 ACM SIGMOD International Conference on Management of Data*, June 2005.

[6] M. Balazinska, H. Balakrishnan, and M. Stonebraker. Contract-based load management in federated distributed systems. In *Proc. of the First Symposium on Networked Systems Design and Implementation (NSDI)*, Mar. 2004.

[7] Borealis installation guide.
`http://www.cs.brown.edu/research/borealis/public/install/`.

[8] A distributed catalog for the borealis stream processing.
`http://www.cs.brown.edu/research/borealis/public/publications/%catalog.pdf`.

[9] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Çetintemel, Y. Xing, and S. Zdonik. Scalable distributed stream processing. In *Proc. of the First Biennial Conference on Innovative Data Systems Research (CIDR)*, Jan. 2003.

[10] The GNU Linear Programming Kit (GLPK).
`http://www.gnu.org/software/glpk/glpk.html`.