BOREALIS APPLICATION PROGRAMMER'S GUIDE

Borealis Team

May 31, 2006

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Borealis is a second-generation distributed stream processing engine developed as part of a collaboration among Brandeis University, Brown University, and MIT. In this document, we provide a guideline to those who wish to write applications that use Borealis. We refer those interested in the internals of Borealis to our "Borealis Developer's Guide" [7].

Before you start, you need to install Borealis and compile the sample applications. The detailed installation procedure is presented in the "Borealis Installation Guide" [5].

Borealis builds on two earlier systems: Aurora and Medusa. Borealis inherits core stream processing functionality from Aurora and distribution capabilities from Medusa. Borealis however does modify and extend both systems with various features and mechanisms [1].

This document is organized as follows. In Chapter 2, we show how to run some simple demo applications. In Chapters 3 and 4 we present the details of the Borealis operators and query diagrams. Chapter 5 discusses how to build your own Borealis application. Chapter 6 briefly overviews how to make your applications fault-tolerant.

Chapter 2

Running a Simple Application

In this chapter, we show how to run a Borealis application. In particular, we show how to run the `mytest` and `mytestdist` applications distributed with Borealis and located under `borealis/test/simple/`.

2.1 Single-Node Deployment

`mytest` demonstrates a simple query deployed at a single Borealis node. Inside the file: `mytest.xml`, you can find the description of the query diagram corresponding to this application. The description includes the input streams, output streams, and the operators. We discuss how to build applications such as `mytest` in Chapter 5. In this Chapter, we only show you how to run them.

To run the `mytest` application, invoke the following script:

```
> ./runtest mytest
```

The script first starts a Borealis node, and then launches the `mytest` application. You should see two windows:

1. A Borealis nodes starts in a window called: Borealis
2. `mytest` starts in a window called: mytest

What happens under the covers is the following:

1. First, the Borealis node starts empty, and waits for requests.
2. Second, the client application starts. To deploy a query diagram, applications normally talk to the global catalog, called `BigGiantHead` (or *Head* for short). No such catalog is running at this point. The application launches the catalog and submits the query to the catalog.
3. The catalog deploys the query on the Borealis node and exits. As part of the deployment, the global catalog also sets-up the subscription of `mytest` to the output stream.
4. Finally, `mytest` starts pushing data into the Borealis node and receiving data from the Borealis node.

To stop the application, invoke:

```
> ./runtest stop
```

2.2 Distributed Deployment

`mytestdist` is a simple application, similar to `mytest` except that it deploys a query diagram over two Borealis nodes. With this example, we also show how to launch and use the Global Catalog separately from the application.

Inside the file `mytestdist.xml`, you can find the description of the query diagram corresponding to the application. The query diagram is the same as for mytest, except for an extra operator that filters the output produced by count, dropping every other result.

Because we want to run each operator on a separate processing node, we create a second file that specifies our desired deployment. Examine the file `mytestdist-deploy.xml` to see how the deployment can be specified. The deployment is specified in terms of "queries". A query is simply a group of one or more operators.

Given these files, we could simply deploy the query diagram as above. However, to demonstrate dynamic modifications of the query diagram, we will start the global catalog separately, as a third process. To see how the application sends requests to the global catalog, examine the method: `MytestdistMarshal::launchDiagram` and compare it with `MytestMarshal::launchDiagram`.

To run the application, simply invoke the following script:

```
> ./runtest mytestdist
```

You should see four windows:

1. First, the global catalog starts in a window called: HEAD
2. Second, a Borealis nodes starts in a window called: Borealis
3. Third, a *second* Borealis nodes starts in a window called: Borealis Note that the window will appear on top of the previous one.
4. Finally, mytestdist starts in a window called: mytestdist

In this example, `mytestdist` submits requests to the global catalog process. These requests include the description of the query diagram and the desired deployment. The global catalog performs the required operations by communicating with the Borealis nodes. Because the global catalog runs continuously, it can later accept and perform requests to modify the query diagram at runtime.

To stop the application, invoke:

```
> ./runtest stop
```

In both examples, the demo applications send their outputs to the terminal as well as to a text file with extension ".log". For more information about running these applications and the output they produce, please consult the README file provided in `borealis/test/simple/README`.

Chapter 3

Background

3.1 Stream Data Model

The Borealis stream data model is based on the model introduced in Aurora, which defines a stream as an append-only sequence of data items. Data items are composed of attribute values and are called *tuples*. All tuples on the same stream have the same set of attributes. This set of attributes defines the type or *schema* of the stream. As an example, in an application that monitors the environmental conditions inside a building, suppose that sensors produce a stream of temperature measurements. A possible schema for the stream is (t.time, t.location, t.temp), where t.time is a timestamp field indicating the time when the measurement was taken, t.location is a string indicating the location of the sensor, and t.temp is an integer indicating the temperature value.

3.2 Operators

The goal of a stream processing engine is to filter, correlate, aggregate, and otherwise transform input streams to produce outputs of interest to applications, making the applications themselves easier to write. For instance, a stream processing engine could produce an alert when the combination of temperature and humidity inside a room goes outside a range of comfort. The application might then simply transform the alert into an email and send it to the appropriate party.

Inside an SPE, input streams are transformed into output streams by traversing a series of operators (*a.k.a.*, boxes). We now describe the core Borealis operators. A detailed description of these operators appears in Abadi et. al. [2].

3.2.1 Stateless Operators

Stateless operators perform their computation on one tuple at a time without holding any state between tuples. There are three stateless operators in Borealis: *Filter*, *Map*, and *Union*.

Filter is the equivalent of a relational selection operator. Filter applies a predicate to every input tuple, and forwards tuples that satisfy the predicate on its output stream. For example, a Filter applied to a stream of temperature measurements may forward only tuples with a temperature value greater than some threshold (e.g., "temperature $> 101°\mathrm{F}$"). Tuples that do not satisfy the predicate are either dropped or forwarded on a second output stream. A Filter can have multiple predicates. In that case, the Filter acts as a case statement and propagates each tuple on the output stream corresponding to the first matched predicate.

A Map operator extends the Projection operator. Map transforms input tuples into output tuples by applying a set of functions on the tuple attributes. For example, Map could transform a stream of temperature readings expressed in Fahrenheit into a stream of Celsius temperature readings. As a more complex

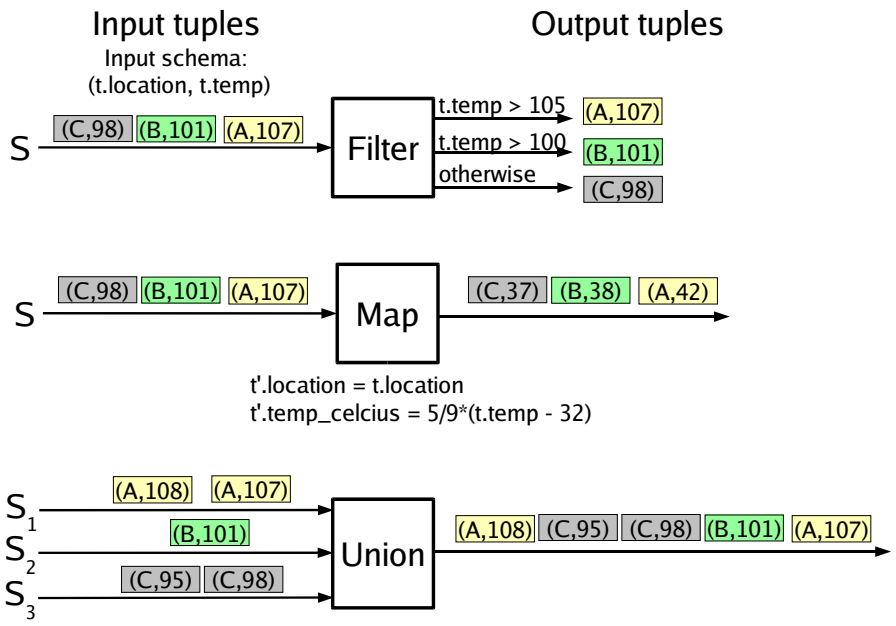

Figure 3.1: **Sample outputs from stateless operators.** Tuples shown on the right of each operator are the output tuples produced after processing the tuples shown on the left.

example, given an input tuple with two attributes, d and t, indicating a distance and a time period, Map could produce an output tuple with a single attribute indicating a speed, $v = \frac{d}{t}$.

A Union operator simply merges a set of input streams (all with the same schema) into a single output stream. Union merges tuples in arrival order without enforcing any order on output tuples. Output tuples can later be approximately sorted with a *BSort* operator. The latter operator maintains a buffer of a parameterizable size $n + 1$. Every time a new tuple arrives, BSort outputs the lowest-valued tuple from the buffer.

Figure 3.1 shows examples of executing Filter, Map, and Union on a set of input tuples. In the example, all input tuples have two attributes, a room number indicated with a letter and an integer temperature reading. The Filter has two predicates and a third output stream for tuples that match neither predicate. The Map converts Fahrenheit temperatures into Celsius. Union merges tuples in arrival order.

3.2.2 Stateful Operators

Rather than processing tuples in isolation, stateful operators perform computations over groups of input tuples. Borealis has a few stateful operators but we present only Join and Aggregate, the most fundamental and most frequently used stateful operators.

An aggregate operator computes an aggregate function such as average, maximum, or count. The function is computed over the values of one attribute of the input tuples (e.g., produce the average temperature from a stream of temperature readings). Before applying the function, the aggregate operator can optionally partition the input stream using the values of one or more other attributes (e.g., produce the average temperature for each room). The relational version of aggregate is typically *blocking*: the operator may have to wait to read all its input data before producing a result. This approach is not suitable for unbounded input streams. Instead, stream processing aggregates perform their computations over *windows* of data that move with time (e.g., produce the average temperature every minute). These windows are defined over the values of one attribute of the input tuples, such as the time when the temperature measurement was taken. Both the window size and the amount by which the window slides are parameterizable. The operator does

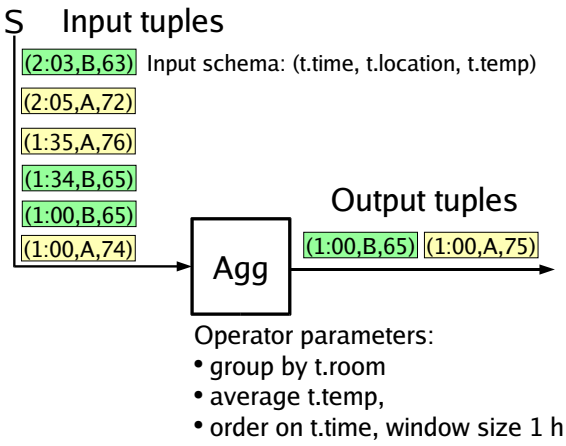

Figure 3.2: **Sample output from an aggregate operator.**

not keep any history from one window to the next, but windows can overlap. As an example, suppose an aggregate operator computes the average temperature in a room, and receives the following input. Each tuple has two attributes, the time of the measurement and the measured temperature.

Input: (1:16, 68), (1:21, 69), (1:25, 70), (1:29, 68), (1:35, 67), (1:41, 67), ...

The aggregate could perform this computation using many different window specifications. As a first example, the aggregate could use a landmark window [**?**], keeping a running average of the temperature starting from the landmark value, and producing an updated average for every input tuple. The following is a possible sequence of outputs, assuming 1:00 is the landmark.

Output 1: (1:16, 68), (1:21, 68.5), (1:25, 69), (1:29, 68.75), (1:35, 68.4), (1:41, 68.17), ...

Alternatively, the aggregate could use a sliding window [2]. Assuming a 10-minute-long window advancing by 10 minutes, the aggregate could compute averages for windows [1:16,1:26), [1:26,1:36), etc. producing the following output:

Output 2: (1:16, 69), (1:26, 67.5), ...

In this example, the aggregate used the value of the first input tuple (1:16) to set the window boundaries. If the operator started from tuple (1:29, 68), the windows would have been [1:29,1:39), [1:39,1:49), etc. The operator could also round down the initial window boundary to the closest multiple of 10 minutes (the value of the advance), to make these boundaries independent of the tuple values. With this approach, the aggregate would have computed averages for windows [1:10,1:20), [1:20,1:30), [1:30,1:40), etc., producing the following output:

Output 3: (1:10, 68), (1:20, 69), (1:30, 67), ...

Borealis supports only sliding windows. In Borealis, windows can also be defined directly on a static number of input tuples (e.g., produce an average temperature for every 60 measurements).

In general, window specifications render a stateful operator sensitive to the order of its input tuples. Each operator assumes that tuples arrive ordered on the attribute used in its window specification. The order then affects the state and output of the operator. For example, when an operator with a 10-minute

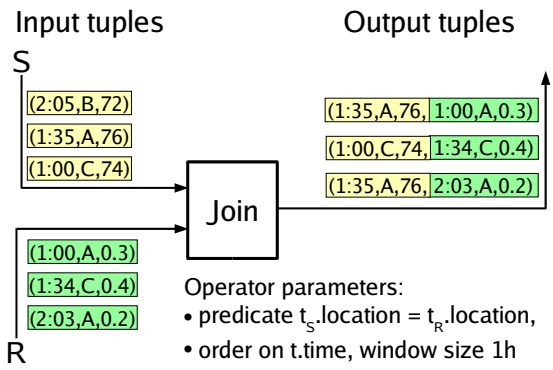

Figure 3.3: **Sample output from a join operator.**

sliding window computes the average temperature for 1:10 pm and receives a tuple with a measurement time of 1:21 pm, the operator *closes the window*, computes the average temperature for 1:10 pm, produces an output tuple, and *advances* the window to the next ten-minute interval. If a tuple with a measurement time of 1:09 pm arrives after the window closed, the tuple is dropped. Hence, applying an aggregate operator to the output of a Union produces approximate results, since the Union does not sort tuples while merging its input streams. Figure 3.2 illustrates a simple aggregate computation. In the example, tuples have three attributes: a measurement time, a location (room identified with a letter), and a temperature value. The aggregate produces separately for each room, the average temperature every hour.

Join is another stateful operator. Join has two input streams and, for every pair of input tuples (each tuple from a different stream), Join applies a predicate over the tuple attributes. When the predicate is satisfied, Join concatenates the two tuples and forwards the resulting tuple on its output stream. For example, a Join operator could concatenate a tuple carrying a temperature reading with a tuple carrying a humidity reading, every time the location of the two readings is the same. The relational Join operator accumulates *state that grows linearly with the size of its inputs*, matching every input tuple from one relation with every input tuple from the other relation. With unbounded streams, it is not possible to accumulate state continuously and match all tuples. Instead, the stream-based Join operator matches only tuples that fall *within the same window*. For two input streams, R and S, both with a `time` attribute, and a window size, w, Join matches tuples that satisfy $|r.\texttt{time} - s.\texttt{time}| \leq w$, although other window specifications are possible. Figure 3.3 illustrates a simple Join operation. In the example, tuples on stream S have three attributes: a measurement time, a measurement location (room identified with a letter), and a temperature value. Tuples on stream R have a humidity attribute instead of a temperature attribute. The Join matches temperature measurements with humidity measurements taken within one hour of each other in the same room.

In summary, stateful operators, in Borealis, perform their computations over windows of data. Because operators do not keep any history between windows, at any point in time, the *state of an operator consists of the current window boundaries and the set of tuples in the current window*. Operators can keep their state in aggregate form. For instance, the state of an average operator can be summarized with a sum and a number of tuples.

Stateful operators can have a *slack* parameter forcing them to wait for a few extra tuples before closing a window. Slack enables operators to support bounded disorder on their input streams. Operators can also have a timeout parameter. When set, a timeout forces an operator to produce a value and advance its window even when no new tuples arrived. Timeouts use the local time at the processing node. When a window of computation first opens, the operator starts a local timer. If the local timer expires before the window closes, the operator produces a value.

3.3 Query Diagrams

In Borealis, the application logic takes the form of a dataflow. To express queries over streams, users or applications compose operators together into a "boxes and arrows" diagram, called a *query diagram*. Figure 3.4 shows an example of a query diagram.

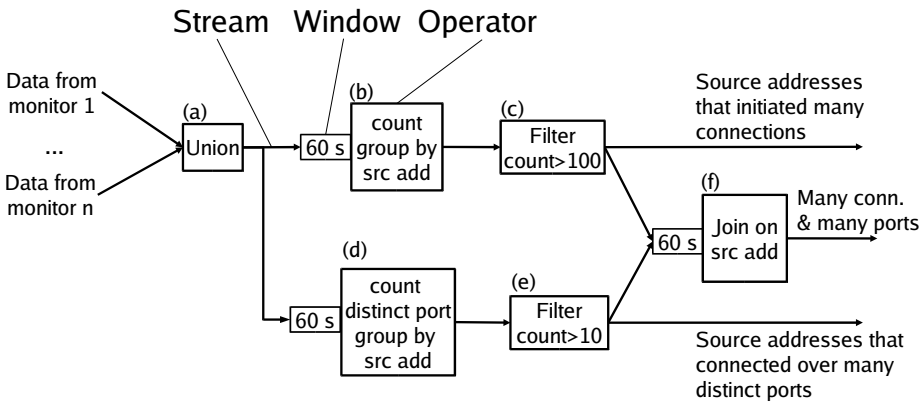

Figure 3.4: **Example of a query diagram from the network monitoring application domain.**

Chapter 4

Query Algebra

4.1 Network Definition

Networks are defined by Borealis Application XML files as summarized in Figure 4.1. The boralis.dtd contains a formal definition with comments. A copy that corresponds to the current release is posted at:

`http://www.cs.brown.edu/research/borealis/borealis.dtd`

The reference copy that corresponds to the source code is in:

`borealis/src/src/boralis.dtd`

```
<?xml version="1.0"?> <!DOCTYPE borealis SYSTEM
"http://www.cs.brown.edu/research/borealis/borealis.dtd">

<borealis>
  <input   stream={stream name}  schema={schema name}    />
  <output  stream={stream name} [schema={schema name}]   />

  <box  name={box name}    type={transform}  >
    <in   stream={input stream name}   />
    <out  stream={output stream name}  />
    <parameter  name={parameter name}   value={parameter value} />
    <access    table={table name} />
  </box>

  <query  name={query name} />
    <box  name={box name}    type={transform}  >
       <in   stream={input stream name}   />
       <out  stream={output stream name}  />
       <parameter  name={parameter name}   value={parameter value} />
    </box>
  </query>

  <connection_point_view name={view name} stream={stream name} >
    <order field={field name} />?
  ( <size  value={number of tuples} />
  | <range start={start tuple}  end={end tuple} />
  )
  </connection_point_view>
</borealis>
```

Figure 4.1: Summary of a Borealis Network Definition XML

4.2 Query Deployment

A deployment file specifies how the Borealis network is distributed over Borealis components. Deployment XML is summarized below and described in detail in Section 5.2.4.

When no deployment XML files are passed to the Head then a default deployment is created. The default deployment uses default values for endpoints and deploys the network to a single node. A publish element is a assigned to each input element and a subscribe is a assigned to each output. No regional or global components are deployed. Only simple applications can get by without a deployment file.

When testing a system with multiple components it is often convenient to deploy them all on a single computer. Running the Head with the -l option will override the ports specified in the deployment XML with ports on the local host computer. After the application is debugged on a single computer then you can distribute it over multiple computers.

Applications may also pass XML to the Head using dynamic deployment. When no XML files are given to the Head via the command line it runs in persistent mode. The "-p" option forces persistent operation when XML files are given. Note that Global components require that the Head be run in persistent mode.

Applications pass XML to a persistent Head using the RPC calls deployXmlString and deployXmlFile in the HeadServer class. Applications can include borealis/tool/head/HeadClient.h to access these methods.

```
<?xml version="1.0"?>
<!DOCTYPE borealis SYSTEM "http://www.cs.brown.edu/research/borealis/borealis.dtd">

<deploy>
  <publish    stream={input stream}   [endpoint={node}]  />
  <subscribe  stream={output stream}  [endpoint={monitor}]  [gap={gap size}] />

  <node  endpoint={node}  [query={query name ...} ] />

  <region  node={node}  [endpoint={regional component}] />
  <global  endpoint={global component} />

  <replica_set   name={set name} query={query name ...} >
     <node ... />  ...
  </replica_set>

  <client  [prefix={class prefix}]  [endpoint={monitor}] >
    <publish   ... />  ...
    <subscribe ... />  ...
  </client>
</deploy>
```

Figure 4.2: Summary of a Borealis Deployment XML

4.3 XML Attributes

4.3.1 Endpoint and Node Attributes

Data and control ports are specified by endpoint and node attributes. They designate a communications port for a component that receives data or control information from another component. A port is designated by the IP address of the computer running the receiving component and a port number. The IP address may be designated by the name of the computer or a dotted IP address. The port number is a 16 bit unsigned integer used to select a unique communications channel. They are encoded as:

```
[{host address}] [:{port number}]
```

The host address defaults to the computer running the Head. Similarly if "localhost" or the IP address "127.0.0.1" refers to the computer running the Head. Different default values are used for port numbers depending on the type of component. Constants for the defaults are defined in:
`borealis/src/modules/common/common.h`

Host Type	Default Port	XML Elements
Borealis Node	15000	node endpoint, region node, publish endpoint
Output Monitor	25000	subscribe endpoint, client endpoint
Head	35000	Persistent Head; used for dynamic deployment.
Regional Component	45000	region endpoint
Global Component	55000	global endpoint

Table 4.1: Default ports for Programs and Elements.

4.3.2 XML Tags

XML is a case sensitive language. By convention the tags in Application XML are lowercase. They consist of English keywords using an underscore between words (e.g. replica_set).

The names of Borealis objects are case insensitive. They are converted to lowercase names by the Head when they are deployed to Borealis nodes. Any name tag or a reference to one is also case insensitive. Note that some parameter values can have references to names. Only for these parameters, the value elements are lowercased and are consequently case insensitive:

```
expression.*                        key
output-expression.*                 sql
out-field.*                         group-by
out-field-name.*                    order-on-field
aggregate-function.*
aggregate-function-output-name.*
```

Figure 4.3: Parameter names whose values are case insensitive.

When using the marshal tool you might want to capitalize object names depending on how you want the generated C++ code to correspond to your coding conventions. The case rules for the marshal tool are described in section 3.

4.4 Stream Definition

A Borealis network is a dataflow diagram. The network can be distributed over several processing nodes.

There are three types of dataflows: streams, inputs and outputs. Streams are dataflows that are inside a node. They are implicitly declared from their name in stream attributes on boxes. Inputs and outputs are dataflows connecting an application program to Borealis nodes.

```
<input   stream={stream name}   schema={schema name}  />
<output  stream={stream name}  [schema={schema name}] />
```

An input passes data from an application to a node and an output sends data from a node to an application. For now output streams must be connected to streams that are in turn connected to box outputs. In other words an output can not be directly connected to an input.

The publish and subscribe elements in deployment XML connect application components to streams. Multiple components may publish data to an input stream and several components may subscribe to an output stream. Consequently there may be one or more publish or subscribe elements per stream.

Streams have one incoming (source) connections and may fan out to several outgoing (sink) connections.

The data passed over inputs and outputs are flat structures defined by a schema. Schemas for outputs will eventually be declared implicitly. For now they must be defined.

```
<schema  name={schema name} />
    <field  name={field name}  type={field type}  [size={string size}] /> ...
</schema>
```

Field types are in

Field	**Contents**
int	A 32 bit signed integer.
long	A 64 bit signed integer.
single	A 32 bit IEEE floating point value.
double	A 64 bit IEEE floating point value.
string	A fixed length, zero filled sequence of bytes.
timestamp	A 32 bit time value.

Table 4.2: Field Types

4.5 Query, Box and Table Definition

Processing in a Borealis node is performed by boxes. A query is a group of boxes. They are useful for referencing the group as a whole. For clarity other related elements may be included within a query block, but this does not effect the behavior of the network.

```
<query  name={query name} />
    <box  name={box name}    type={transform}  >
        <in   stream={input stream name}   />
        <out  stream={output stream name}  />
        <parameter  name={parameter name}  value={parameter value} />
    </box>  ...

    <table  name={table name}  schema={schema name} >
        <key       field={field name} />  ...
       [<index     field={field name} />  ...]
       [<parameter  name={parameter name}   value={parameter value} />  ...]
    </table>  ...
</query>
```

If a box or table is not wrapped in a query, a query will be defined containing just the one box or table. The name of the query will be the same as the box or table name.

4.6 Connection Point View Definition

```
<connection_point_view name={view name} stream={stream name} >
        <order field={field name} />?
      ( <size  value={number of tuples} />
      | <range start={start tuple}  end={end tuple} />
      )
</connection_point_view>
```

CPViews can be placed on any arc of the network at the time of the construction or modification of the network. CPView accumulates tuples that flow on that arc according to specifications defined by the user. CPView can be fixed or moving, and that is also defined by its specification. Fixed view stores the same set of tuples that does not change over time. Moving view stores a window of tuples defined with respect to the latest tuple seen on this arc, so as new tuples are flowing along the arc, the set of tuples stored by a moving view changes. Several CPViews with different specifications can co-exist on the same arc.

The user can define size or range and optionally prediction function.

Size can be in terms of number of tuples or in terms of values of the order_by field of the stream, on which the CPView is defined. If specification defines the size, it means CPView is moving, and size determines how many most recent tuples or values CPView stores. Range is defined by specifying start and end parameters (also, either tuples or order_by values). Range can be defined for either fixed or moving CPView. If start and end are absolute, then the CPView is fixed as the tuples stored do not change. However, if the start and end are relative (eg. now and "now - 1 hour"), then the view is moving. Prediction function is an optional parameter and should be defined only for CPViews that are used for time travel into future. For now this parameter is not used.

CPViews can be used for attaching ad-hoc queries to them and for time travel. Time travel can be either into past or into future and it can happen either on the main query network or on a copy of a branch of the network downstream from the CPView. In order for a CPView to time travel into future, prediction function F parameter has to be defined for this view. Time travel happens via issuing replay() command on the CPView. The parameters of replay() determine in what part of past/future time travel happens.

4.7 Update Queues

The default queue behavior in Borealis is based on an "append semantics". In other words, every new tuple that arrives on a stream gets appended to the end of a FIFO queue for that stream. Alternatively, Borealis allows an application to selectively define some or all of its queues as "update queues".

An update queue is essentially a lossy tuple storage model which allows a new tuple for a certain key field value to overwrite an old tuple of the same key field value. For example, assume that we have a stock stream of the schema `(time, symbol, price)` where the key field is the `symbol` field. If we receive `(9:00, IBM, 20)` followed by `(9:01, IBM, 21)`, then the latter tuple replaces the former tuple in the update queue.

Borealis provides several types of update queues, each having a different preemption policy. An application can specify update queues in the network description at different levels of the hierarchy. If no specification for a queue, then it is append queue by default.

For example, the following indicates that all queues in the complete borealis query network are update queues of type 0, where the update key is the "symbol" field.

```
<borealis update_queue="1" queue_type="0" key="symbol" >
    ...
</borealis>
```

Similarly, the following indicates that all queues *only* in the given query block, are update queues of type 0, where the update key is the "symbol" field.

```
<borealis>
    ...
    <query name="query1" update_queue="1" queue_type="0" key="symbol" >
        ...
    </query>
    ...
</borealis>
```

Similar definitions can be made for a specific box element, or for a specific input stream of a given box element. Please see `borealis.dtd` for the detailed syntax.

Please note that the queue specifications for child elements override those for the parent elements. For example, one could define that all queues in the complete borealis network are update queues, and then could further define a certain query element in that network to have append queues. As a result, all queues except the ones contained in that specific query block would have update queues.

4.8 Expressions

An expression consists of one or more operands and a set of operations to be applied to them.

An operand is a field name or a constant literal. Constant literals include integer literal (e.g. 10, -20, +35), float point literal (e.g. 4.39) or string literal. A string literal is a sequence of characters enclosed within single quotes (e.g. 'abc'). Currently, Borealis does not support special characters such as "
n", or quotes.

Borealis supports the following basic operators:

- **Arithmetic operator:** +, -, *, /, % (mod)
- **Comparison operator:** >, <, >=, <= , = or ==, != (not equal to)
- **Logical operator:** and, or, not

Note that in the query xml, operator "<" should be written as "<"; operator "<=" should be written as "<=".

Here are some example expressions:

```
A
A + B
A % 10 == 3
A = 'abc'
(A > 2.5) or (A &lt; B)
not((A - B > 10) and (A != 20))
```

where A and B are field names.

4.9 Operators

For details on the semantics of the operators and example, please consult [6].

```
<schema name="StockType_USD">
    <field  name="symbol"     type="string" size="4" />
    <field  name="time"       type="int"             />
    <field  name="price_usd"  type="double"          />
</schema>
<schema name="StockType_CAD">
    <field  name="symbol"     type="string" size="4" />
    <field  name="time"       type="int"             />
    <field  name="price_cad"  type="double"          />
</schema>
<schema name="StockType_ALL">
    <field  name="symbol"     type="string" size="4" />
    <field  name="time"       type="int"             />
    <field  name="price"      type="double"          />
</schema>
```

Figure 4.4: Schemas that will be used in this section.

NB:
* Parameters in *italics* are optional parameters.
* < # > is a zero-based decimal index.

4.9.1 Map

Description: Map composes each output tuple by transforming each input tuple. The transform is specified by pairs of parameters, one pair for each field in the Out stream. At least one pair of parameters is required.

Input: 1 input stream.

Parameters: Table 4.3 shows the list of parameters that Map takes.

Output: 1 output stream whose schema is defined by the expressions in Map.

Parameter	Value
expression.< # >	an expression over the input stream
output-field-name.< # >	output field name receiving the expression result

Table 4.3: Map Parameters

Example: Suppose that one wanted to translate every quote price in the `Stock_USD` stream from US dollars to Canadian dollars, assuming 1 USD to be equal to 0.75 CAD.

```
<input  stream="Stock_USD" schema="StockType_USD" />
<output stream="Stock_CAD" schema="StockType_CAD" />

<box name="USQuoteToCanadian" type="map">
    <in  name="Stock_USD" />
    <out name="Stock_CAD" />
    <parameter name="expression.0"        value="symbol"         />
    <parameter name="output-field-name.0" value="symbol"         />
    <parameter name="expression.1"        value="time"           />
    <parameter name="output-field-name.1" value="time"           />
    <parameter name="expression.2"        value="price_usd/0.75" />
    <parameter name="output-field-name.2" value="price_cad"      />
</box>
```

Functions: The following functions can be used for expressions in Map:

- pad : pad a string to a certain length
- min : find lowest value
- max : find highest value
- sequence : sequence number starting at 0
- strlen : calculate the length of a string
- int : get integer value
- long : get long value
- abs : compute the absolute value of an integer
- now : current Unix timestamp
- cos : cosine function
- acos : arc cosine function
- cosh : hyperbolic cosine function
- tan : tangent function
- atan : arc tangent function
- tanh : hyperbolic tangent function
- sin : sine function
- asin : arc sine function
- sinh : hyperbolic sine function
- ln : natural logarithmic function
- log : base-10 logarithmic function
- sqrt : square root function
- exp : base-e exponential function

- floor : floor function: largest integral value not greater than argument
- ceil : ceiling function: smallest integral value not less than argument
- pow : ppower function
- atan2 : arc tangent function of two variables

4.9.2 Filter

Description: Filter applies predicates to each input tuple and route tuples to the output stream of the first predicate that evaluates to *true*. If none of the predicates evaluate to *true*, tuples can be routed to a separate stream.

Input: 1 input stream.

Parameters: Table 4.4 shows the list of parameters that Filter takes.

Parameter	Value
expression.< # >	a predicate over the input stream
pass-on-false-port	`1` to output false tuples, `0` otherwise

Table 4.4: Filter Parameters

Output: There is an output stream for each predicate plus an additional stream when `pass-on-false-port` is `1`. The schemas of the output streams are all the same as the schema of the input stream.

Example: Suppose that one wanted to separate quotes from `Stock_USD` stream into two streams - `Stock_USD_large` containing quote prices greater than 10 and `Stock_USD_small` containing quote prices less than or equal to 10.

```
<input  stream="Stock_USD" schema="StockType_USD"       />
<output stream="Stock_USD_large" schema="StockType_USD" />
<output stream="Stock_USD_small" schema="StockType_USD" />

<box name="FilterQuotes" type="filter">
    <in  name="Stock_USD"       />
    <out name="Stock_USD_large" />
    <out name="Stock_USD_small" />
    <parameter name="expression.0"        value="price>10"  />
    <parameter name="pass-on-false-port"  value="1"         />
</box>
```

4.9.3 Aggregate

Description: Aggregate applies one or more aggregate functions to sliding windows over its input stream. At least one aggregate function is required.

Input: 1 input stream.

Parameters: Table 4.5 shows the list of parameters that Aggregate takes.

Output: 1 output stream whose schema depends on the `order-by` parameter as follows:
If `order-by` is `FIELD`, output schema: (`group-by`*, `order-on-field`, `aggregate-function.`< # >).
If `order-by` is `FIELD`, output schema: (<`group-by`>*,[`order-on-field`],`aggregate-function.`< # >).

Parameter	Value
order-by	`FIELD` or `TUPLENUM` specifying whether the order is to be maintained by values in the input tuples or the arrival order of input tuples
order-on-field	integer field to order input tuples on (mandatory if `order-by` is `FIELD` OR `window-size-by` is `VALUES`)
group-by	fields separated by "," if grouping is required
window-size	integer specifying the size of the window
window-size-by	`VALUES` or `TUPLES` specifying whether the number of tuples in a window is given by the values of the `order-on-field` of the input tuples or the arrival order of the input tuples
advance	integer specifying how to advance/slide the window
independent-window-alignment	boolean value (set to either "1" or "0"). When true (i.e., value="1"), the window alignment will always be rounded-down to the closest multiple of the advance. When false (i.e., value="0" or default setting), the window alignment is defined by the value of the first input tuple processed by the operator.
slack	integer specifying how much out-of-order tuples are tolerated in the input tuples
timeout	integer specifying how long to wait before windows close automatically
aggregate-function.< # >	aggregate function
aggregate-function-output-name.< # >	output field name receiving the aggregate result
drop-empty-outputs	boolean value (set to either "1" or "0"). False by default. When true, the aggregate does not output any tuples for windows that were empty.

Table 4.5: Aggregate Parameters

Example 1: Suppose that one wanted to calculate the 30 minutes moving average of every symbol every 15 minutes for quote prices from the `Stock_USD` stream. Let us assume that the `time` field in `Stock_USD` denotes seconds since the start time. Also, let's assume that we allow slack of 5 for out-of-order tuples and that windows timeout after 10 minutes.

```
<input  stream="Stock_USD" schema="StockType_USD"     />
<output stream="Stock_average" schema="StockType_USD" />

<box name="MovingAverage" type="aggregate">
    <in  name="Stock_USD"     />
    <out name="Stock_average" />
    <parameter name="aggregate-function.0"             value="average(price)"  />
    <parameter name="aggregate-function-output-name"   value="movingaverage"   />
    <parameter name="order-by"                         value="time"            />
    <parameter name="order-on-field"                   value="FIELD"           />
    <parameter name="group-by"                         value="symbol"          />
    <parameter name="window-size"                      value="1800"            />
    <parameter name="window-size-by"                   value="VALUES"          />
    <parameter name="advance"                          value="900"             />
    <parameter name="slack"                            value="5"               />
```

```
    <parameter name="timeout"                              value="600"             />
</box>
```

Example 2: Suppose that one wanted to calculate the moving average of the last 7 quote prices advancing each window by 5 tuples from the `Stock_USD` stream.

```
<input  stream="Stock_USD" schema="StockType_USD"         />
<output stream="Stock_average" schema="StockType_average" />

<schema name="StockType_tuple_average">
    <field  name="symbol"         type="string" size="4" />
    <field  name="movingaverage"  type="double"          />
</schema>

<box name="MovingAverage" type="aggregate">
    <in  name="Stock_USD"           />
    <out name="Stock_tuple_average" />
    <parameter name="aggregate-function.0"             value="average(price)"  />
    <parameter name="aggregate-function-output-name"   value="movingaverage"   />
    <parameter name="order-on-field"                   value="TUPLENUM"        />
    <parameter name="group-by"                         value="symbol"          />
    <parameter name="window-size"                      value="7"               />
    <parameter name="window-size-by"                   value="TUPLES"          />
    <parameter name="advance"                          value="5"               />
</box>
```

4.9.4 AuroraJoin, Join

Description: Both join operators pair tuples from two input streams that are within a specified distance on their order fields and that satisfy some input predicate.

The difference between the two joins is how the trimming of the window buffers is done. *Join* trims tuples from the buffer of the same input stream as the tuple currently being processed whereas *AuroraJoin* trims tuples from the buffer of the other stream.

Input: 2 input streams (schemas do not need to match).

Parameters: Table 4.6 shows the list of parameters that Filter takes.

Output: If `out-field.`$< \# >$ are specified, fields mentioned are included in the output schema. Otherwise, fields from both the input streams are concatenated.

Example: Suppose that one wanted to join tuples from the `Stock_USD` and `Stock_CAD` streams with the same `symbol` over a 1 minute window and we care only about the values from the `Stock_USD` stream in the output stream.

```
<input  stream="Stock_USD" schema="StockType_USD"     />
<input  stream="Stock_CAD" schema="StockType_CAD"     />
<output stream="Stock_joined" schema="StockType_USD"  />

<box name="MovingJoin" type="aurorajoin">
    <in  name="Stock_USD"           />
    <in  name="Stock_CAD"           />
```

Parameter	Value
predicate	predicate over the input streams
left-buffer-size	size of the window for the first stream
right-buffer-size	size of the window for the second stream
left-order-by	`VALUES` or `TUPLES` (same specification as Aggregate)
right-order-by	`VALUES` or `TUPLES` (same specification as Aggregate)
left-order-on-field	integer field to order input tuples on the first stream
right-order-on-field	integer field to order input tuples on the second stream
timeout	integer specifying how long to wait before eliminating tuples from the join window
out-field.< # >	field to be included in the output
out-field-name.< # >	field name for output field

Table 4.6: Join Parameters

```
    <out name="Stock_joined" />
    <parameter name="predicate"              value="left.symbol==right.symbol" />
    <parameter name="left-buffer-size"       value="60"                        />
    <parameter name="right-buffer-size"      value="60"                        />
    <parameter name="left-order-by"          value="VALUES"                    />
    <parameter name="right-order-by"         value="VALUES"                    />
    <parameter name="left-order-on-field"    value="time"                      />
    <parameter name="right-order-on-field"   value="time"                      />
    <parameter name="out-field.0"            value="left.time"                 />
    <parameter name="out-field-name.0"       value="time"                      />
    <parameter name="out-field.1"            value="left.symbol"               />
    <parameter name="out-field-name.1"       value="symbol"                    />
    <parameter name="out-field.2"            value="left.price"                />
    <parameter name="out-field-name.2"       value="price"                     />
</box>
```

4.9.5 BSort

Description: BSort is an approximate sorting operator that accepts an integer order field, a pass count n and an optional grouping fields to perform an approximate sort equivalent to n passes of bubble sort over each group.

Input: 1 input stream

Parameters: Table 4.7 shows the list of parameters that BSort takes.

Parameter	Value
order-on	integer field to sort on
slack	integer specifying the number of passes of bubble sort
group-by	fields separated by "," if the sort is to performed in groups

Table 4.7: BSort Parameters

Output: 1 output stream with the same schema as the input stream.

Example: Suppose that one wanted to perform 5 passes of bubble sort on the `time` of each quote price on the `Stock_USD` stream for each `symbol`.

```
<input  stream="Stock_USD" schema="StockType_USD"         />
<output stream="Stock_USD_bsorted" schema="StockType_USD" />

<box name="BSortQuotes" type="bsort">
    <in  name="Stock_USD"         />
    <out name="Stock_USD_bsorted" />
    <parameter name="order-on"  value="time"    />
    <parameter name="slack"     value="5"       />
    <parameter name="group-by"  value="symbol"  />
</box>
```

4.9.6 Union

Description: Union merges one or more streams of tuples with the same schema into a single stream that has arbitrarily determined order.

Input: 1 or more input streams with the same schema.

Parameters: Union does not take any parameters. Streams to merge are defined in the `in` tag.

Output: 1 output stream with the same schema as the input streams.

Example: Suppose that one wanted to merge the quote prices `Stock_USD` and `Stock_CAD` streams to one stream containing all the stock prices called `Stock_ALL`.

```
<input  stream="Stock_USD" schema="StockType_USD" />
<input  stream="Stock_CAD" schema="StockType_CAD" />
<output stream="Stock_ALL" schema="StockType_ALL" />

<box name="UnionQuotes" type="union">
    <in  name="Stock_USD" />
    <in  name="Stock_CAD" />
    <out name="Stock_ALL" />
</box>
```

4.9.7 Lock, Unlock

Description: Lock and Unlock are used to synchronize concurrent access to an integer key field of a given stream.

When Lock receives input tuple with key field of a certain value, if that value has not already been locked, then Lock puts a lock on the key value and passes the tuple to its output. Otherwise, the tuple is stored in a list of pending tuples. Each time the box is scheduled, Lock attempts to acquire the lock for any pending and newly arriving tuples, and releases the tuples for which it is able to acquire the lock.

When Unlock receives input tuple with key field of a certain value, it unlocks the corresponding key and passes the tuple to its output.

Lock and Unlock have the same input, output, and parameter properties, as described below.

Input: 1 input stream.

Parameters: Table 4.8 shows the list of parameters that Lock and Unlock take.

Output: There is 1 output stream of the same schema as the input stream.

Parameter	Value
lockset	a string representing the name for the mutex lock managing access to a set of integer key values
key	a string representing the name of the integer field for the lock key

Table 4.8: Lock/Unlock Parameters

4.9.8 WaitFor

Description: WaitFor is another operator that is used to achieve synchronization in Borealis. It accepts two input streams. The first stream is buffered until a tuple on the second stream is received that satisfies a certain synchronization criteria. WaitFor essentially buffers each tuple t on one input stream either until a tuple arrives on the second input stream that, with t, satisfies a given predicate, or until the tuple times out, in which case t is discarded.

Input: 2 input streams (not necessarily of the same schema).

Parameters: Table 4.9 shows the list of parameters that WaitFor takes.

Parameter	Value
timeout	integer value specifying how long a tuple should be buffered before it is discarded
predicate	a predicate expression on both of the input streams that defines the condition under which a tuple should be released

Table 4.9: WaitFor Parameters

Output: There is 1 output stream of the same schema as the first input stream.

4.9.9 Random Drop

Description: Random drop is a system-level operator used for load shedding. For each input tuple, it probabilistically decides if the tuple should be dropped from the stream.

Input: 1 input stream.

Parameters: Table 4.10 shows the list of parameters that Random Drop takes.

Parameter	Value
drop_rate	float specifying the tuple drop probability
max_batch_size	integer specifying the maximum gap (i.e., an upper limit for the number of tuples that can be dropped in a row). If not important, use a large value for this parameter.

Table 4.10: Random Drop Parameters

Output: There is 1 output stream of the same schema as the input stream.

Example:

```
<input  stream="Stock_USD1" schema="StockType_USD" />
<output stream="Stock_USD2" schema="StockType_USD" />

<box name="RandomDropBox1" type="random_drop">
    <in  name="Stock_USD1" />
    <out name="Stock_USD2" />
    <parameter name="drop_rate"      value="0.5" />
    <parameter name="max_batch_size" value="10"  />
</box>
```

4.9.10 Window Drop

Description: Window drop is a system-level operator used for load shedding. For each input window, it probabilistically decides if the window should be kept or dropped. Accordingly, it marks window starting tuples with a tuple specification flag as part of the tuple's header (0: drop, T: keep, -1:don't care). The positive T value indicates until which windowing field value the tuples should be retained in the stream. Window drop also drops those tuples from the stream that can be early-dropped when all the windows that those tuples belong to are decided to be dropped.

Input: 1 input stream.

Parameters: Window Drop takes parameters that are similar to those of an Aggregate. Table 4.11 shows the list of parameters that Window Drop takes.

Parameter	Value
order-by	`FIELD` or `TUPLENUM` specifying whether the order is to be maintained by values in the input tuples or the arrival order of input tuples
order-on-field	integer field to order input tuples on (mandatory if `order-by` is `FIELD` OR `window-size-by` is `VALUES`)
group-by	fields separated by "," if grouping is required
window-size	integer specifying the size of the window
window-size-by	`VALUES` or `TUPLES` specifying whether the number of tuples in a window is given by the values of the `order-on-field` of the input tuples or the arrival order of the input tuples. (Note that the current code supports only the VALUES option).
window-slide	integer specifying how to advance/slide the window
drop_rate	float specifying the window drop probability
max_batch_size	integer specifying the maximum gap (i.e., an upper limit for the number of windows that can be dropped in a row).

Table 4.11: Window Drop Parameters

Output: There is 1 output stream of the same schema as the input stream.

Example:

```
<input  stream="Stock_USD1" schema="StockType_USD" />
<output stream="Stock_USD2" schema="StockType_USD" />

<box name="WindowDropBox1" type="window_drop">
    <in  name="Stock_USD1" />
```

```
    <out name="Stock_USD2" />
    <parameter name="window-size-by" value="VALUES" />
    <parameter name="window-size"    value="10"     />
    <parameter name="window-slide"   value="1"      />
    <parameter name="order-by"       value="FIELD"  />
    <parameter name="order-on-field" value="time"   />
    <parameter name="group-by"       value="symbol" />
    <parameter name="drop_rate"      value="0.5"    />
    <parameter name="max_batch_size" value="10"     />
</box>
```

4.9.11 Table Operators

Borealis has a number of table operators that execute SQL queries on Berkeley DB tables for each tuple that they receive on their input stream. The XML definition for these operators has an `<access table="..." />` element which specifies the associated table that needs to be queried. Below we briefly describe these operators in terms of their parameters and functionality, what they expect as input stream, and what they produce as output stream(s).

1. **Select**

   Description: For each input tuple t, a SQL expression is executed on the table associated with this operator. The resulting table tuples are delivered as an output stream. Additionally, the incoming stream tuple as well as the number of resulting table tuples can be optionally delivered on two output streams separately.

   Input: 1 input stream.

   Parameters: Table 4.12 shows the list of parameters that Select takes.

Parameter	Value
sql	SQL expression
pass-on-no-results	1 (to pass input on no results), 0 (otherwise)
pass-result-sizes	1 (to pass the result count), 0 (otherwise)

Table 4.12: Select Parameters

   Output: Select has at least 1 and at most 3 output streams depending on the values of its parameters. The schemas for these output streams are determined as follows. The first output stream is always used to deliver the result tuples originating from the table and hence, is of the same schema as the table. If the `pass-on-no-results` parameter is set, then the second output stream is used to deliver tuples originating from the input stream, and hence, is of the same schema as the input stream. If additionally the `pass-result-sizes` parameter is set, then the third output stream is used to deliver tuples whose schema consists of the fields in the where clause + an additional integer field indicating the count of the resulting table tuples. If only the `pass-result-sizes` parameter is set, then the second output stream is used to deliver the count tuples, and the third output stream is omitted. If neither `pass-on-no-results` nor `pass-result-sizes` is set, then Select delivers only 1 output stream.

2. **Insert**

   Description: This operator inserts each tuple it receives on its input stream into an associated table. It also emits the same tuple as output if its `pass-input` parameter is set.

Parameter	Value
sql	SQL expression
pass-input	1 (to pass input after insertion), 0 (otherwise)

Table 4.13: Insert Parameters

Input: 1 input stream.

Parameters: Table 4.13 shows the list of parameters that Insert takes.

Output: There is 1 output stream of the same schema as the input stream.

3. **Delete**

Description: For each input tuple, this operator deletes rows from an associated table. It can output the input tuple and/or the deleted tuple based on its parameters.

Input: 1 input stream.

Parameters: Table 4.14 shows the list of parameters that Delete takes.

Parameter	Value
sql	SQL expression
pass-deletions	1 (to pass deleted tuples after deletion), 0 (otherwise)
pass-input	1 (to pass input after deletion), 0 (otherwise)

Table 4.14: Delete Parameters

Output: Delete has at most 2 output streams depending on the values of its parameters. The schemas for these output streams are determined as follows. If the `pass-deletions` parameter is set, then the first output stream is used to deliver the tuples deleted from the associated table, and hence, is of the same schema as the table. If additionally the `pass-input` parameter is set, then the second output stream is used to deliver the tuples arriving on the input stream, and hence, is of the same schema as the input stream. If only the `pass-input` parameter is set, then the first output stream is used to deliver the input tuples instead, in which case the second output is omitted.

4. **Update**

Description: This operator performs an update operation on the associated table for each input tuple received. It then optionally outputs the input tuple, or the values for the output fields specified as its parameters.

Input: 1 input stream.

Parameters: Table 4.15 shows the list of parameters that Update takes.

Parameter	Value
sql	SQL expression
pass-input	1 (to pass input after insertion), 0 (otherwise)
output-expression.i	expression
output-field-name.i	output field name

Table 4.15: Update Parameters

Output: There is 1 output stream. If `pass-input` parameter is set, then the output schema is the same as the input schema. Otherwise, the output schema consists of the fields specified as `output-field-name.i`.

4.9.12 User-Defined Operators

In addition to its pre-defined set of operators, Borealis also allows applications to define their custom operators in the form of user-defined boxes. Such operators are placed under a `udb` directory, and can use or extend existing operator and expression constructs available in Borealis. In this section, we give examples for two of such boxes: Delay and CountDistinct.

1. **Delay**

Description: For each input tuple, Delay withholds the tuple for a specified amount of time and then releases it as an output tuple.

Input: 1 input stream.

Parameters: Table 4.16 shows the list of parameters that Delay takes.

Parameter	Value
delay-ms	an integer representing the amount of delay in milliseconds
churn	1 (busywait the CPU), 0 (sleep)

Table 4.16: Delay Parameters

Output: There is 1 output stream of the same schema as the input stream.

Example:

```
<input  stream="Stock_USD"         schema="StockType_USD" />
<output stream="Stock_USD_delayed" schema="StockType_USD" />

<box name="DelayBox1" type="delay">
    <in  name="Stock_USD"         />
    <out name="Stock_USD_delayed" />
    <parameter name="delay-ms" value="100"/>
    <parameter name="churn"    value="1"/>
</box>
```

Implementation: The implementation of Delay box extends the basic `QBox` class provided in Borealis. It provides custom definitions for 2 basic methods of this class: `setup_impl()` and `run_impl()`. The `setup_impl()` method is essentially used to set the values of the Delay parameters and the description for the output stream. The `run_impl()` method codes what Delay should do when it receives a new input tuple. We refer the interested reader to examine the code for Delay to get more insight on how to write his/her own custom box in Borealis. The "Borealis Developer's Guide" also explains the details of writing custom box code.

2. **CountDistinct**

CountDistinct is an example for a user-defined windowed aggregate in Borealis. It counts the number of distinct values on a given field of a stream (as opposed to a standard aggregate provided in Borealis which counts all the values in a window). It is actually not a user-defined operator as Delay, but rather a user-defined aggregate function. We are including it here to illustrate how one can write such custom functions to create user-defined windowed aggregate boxes.

The implementation of CountDistinct extends the basic `Aggregate` class provided in Borealis. The `setup` method is defined to make sure that CountDistinct produces an output field of type integer. Additionally, a new `CountDistinctWindow` class is created to extend `AggregateWindow` such that `init()`, `incr()`, and `final()` functions reflect the semantics of the CountDistinct functionality. We refer the interested reader to examine the code for CountDistinct to get more insight on how to write his/her own custom windowed aggregate function in Borealis. Again the "Borealis Developer's Guide" would also be a good reference for details.

4.9.13 RevisionMap, RevisionFilter, RevisionAggregate

Description: These boxes process revision tuples. There are two modes of revision processing [8]: *upstream processing* and *downstream processing*.

Upstream processing "replays" previously processed *input* tuples that were involved in the same computations as the tuple being revised. Results are generated using both the old and the new values of the tuple, and revisions are output if those results differ.

Downstream processing "retrieves" all previously produced *output* tuples to which the tuple being revised originally contributed to, and modifies these tuples according to the revision to produce the revised results. Again, revisions are output if the revised results differ from the original results.

Input: The input stream(s) are the same as the input stream(s) of the respective box that does not process revisions. A field called `revised_value` (of type `string`) is required for the boxes to be able to process revisions. This field is used in conjunction with the `rev_field` field in the header of the tuple. `rev_field` gives the index of the field (starting at 0) that being revised and `revised_value` gives the revised value of that field. The actual field that is being revised has the original value.

For example: Let's say a tuple has the schema of `StockType_USD`. If `price_usd` is being revised from 25.00 to 22.00, then `rev_field=2`, `price_usd=25.00`, and `revised_value="22.00"`.

Parameters: The parameters for the boxes are the same as the parameters for the respective boxes with the addition of the two parameters shown in Table 4.17, both of which are mandatory.

Parameter	Value
processing-mode	`UPSTREAM` or `DOWNSTREAM` for the mode of revision processing
anchor	(stream name+"`CP`") where the CP that provides historical tuples for this box resides

Table 4.17: Mandatory Parameters for Revision Boxes in Addition to the Parameters of the Boxes

Output: The output schema is the same as the output schema of the respective box that does not process revisions, with the addition of a final field `revised_value`.

4.9.14 SControl, SUnion, SOutput, and SJoin

These operators are used with the DPC fault-tolerance protocol discussed in Chapter 6.

Chapter 5

Building Borealis Applications

5.1 Program Build Process

Borealis runs on GNU/Linux i86 computers and is developed with open source packages. Before starting you will need to install the packages as described in the Borealis Installation Guide. Even if the packages are set up for you by an administrator, you will need to set up your programming environment as described in the Installation Guide. See `borealis/report/install_guide/` or visit the web page posted at:

`http://www.cs.brown.edu/research/borealis/public/install/install.borealis.html`

A typical Borealis application consists of application C++ code (<program>.cc, ...), Application XML files (<program>.xml, <program>_deploy.xml, ...) and marshaling code (<Program>Marshal.h, <Program>Marshal.cc, ...). The marshal program parses your XML files using the borealis.dtd and generates the marshaling code. For examples showing how to make and run a simple Borealis applications see the examples in the `borealis/test/simple/` directory.

```
                                  <program>.cc
<program>.xml -> [ marshal ] -> <Program>Marshal.h -> [ c++ ] -> <program>
borealis.dtd                   <Program>Marshal.cc
```

Note that you should not put application code within the Borealis source tree. Otherwise it will be difficult to upgrade the Borealis source tree when new versions or updates are released.

Run a copy of the `borealis/src/src/borealis` program for each Borealis node used in your application. The nodes may be run on the same or different computers. To build the `borealis/src/src/borealis` program read the comments in the following script, setup your environment, and then run the script:

```
> borealis/utility/unix/build.borealis.sh
```

To build a Borealis application you will need to build the `borealis/tool/marshal/marshal` tool and place it in a directory listed on your PATH variable. To run the application you will also need the `borealis/tool/head/BigGiantHead` tool. It too needs to be accessible via the PATH variable.

The build script can also be used to build the tools as well as tests and demo programs. At a minimum you will want to build the client interface and marshal and head tools.

```
> build.borealis.sh  -client  -tool.head  -tool.marshal
```

Run a copy of the `borealis/src/src/borealis` program for each Borealis node used in your application. The nodes may be run on the same or different computers. Launch any borealis nodes before your application program.

There are several ways to construct and deploy a Borealis application; depending on the interplay of components and level of control needed. The simplest way is to use the marshal tool to build your application. In this case, the BigGiantHead program can be launched by the application. The Head will then set up and start the network and then goes away. Reference documentation for the Head is in: `borealis/tool/head/BigGiantHead.cc`

```
                     [ program ]   <-----> [ borealis ]
<program>.xml --> [ BigGiantHead ] <-----> [ borealis ]
```

The Head can also continue to run instead of just quitting. A persistent Head can read additional XML to modify the network on the fly. It will also validate the network (type checking) as it is changed. A persistent Head is also required for updating the catalog in any global components (see section 4.2).

5.2 Marshaling Tool

The program, borealis/tool/marshal/marshal, generates C++ code to simplify programming stream connections for Borealis applications. It is not required as application programmers might want to write applications in other programming languages, directly manage low-level stream RPC calls or need to make dynamic stream connections. It may not generate the exact code variation required by a given application as it generates code used by typical applications.

The marshal tool reads the same Application XML files as the Head. However, only the elements related to input and output streams are processed. The marshal tool can then generate useful output from partially constructed XML files early in the development process.

```
<?xml version="1.0"?>
<!DOCTYPE borealis SYSTEM "http://www.cs.brown.edu/research/borealis/borealis.dtd">

<borealis>
    <input   stream={stream name}  schema={schema name}  />
    <output  stream={stream name}  schema={schema name}  />

    <schema  name={schema name} >
        <field  name={field name}  type={field type}  [size={string size}]/>  ...
    </schema>
</borealis>


<?xml version="1.0"?>
<!DOCTYPE borealis SYSTEM "http://www.cs.brown.edu/research/borealis/borealis.dtd">

<deploy>
    <publish    stream={stream name}  [endpoint={node}]  />
    <subscribe  stream={stream name}  [endpoint={monitor}]  />

    <client  [prefix="<class prefix>"]  [endpoint={monitor}] >
        <publish   ... />  ...
        <subscribe ... />  ...
    </client>
</deploy>
```

Figure 5.1: Summary of marshaling XML.

5.2.1 Generated Network Diagram Code

An application may be written as a single program or as a collection of cooperating programs. C++ applications are written using the NMSTL event loop architecture. The code generated by the marshal program hides the details of interfacing with NMSTL. Still, you need to keep in mind that application programs are event driven.

You want to connect to applications that are not event driven or that use a different stream format or processing model (e.g. push versus pull). In these cases the generated code can be used to quickly create a front-end program for your application. Using a front-end program also allows your application to be written in any programming language.

Schemas In C++

```
<schema  name={schema name} >
   <field  name={field name}  type={field type}  [size={string size}]/>  ...
</schema>
```

A structure is generated for each schema within the header files generated by the marshal program. A second structure is generated to permit access to the fields in the tuple header. The generated C++ types that correspond to the fields in a schema are listed in Table 5.1. They are declared in the header: `borealis/src/modules/common/common.h`

Schema Field	C++ Type	Description
int	int32	A 32 bit signed integer.
long	int64	A 64 bit signed integer.
single	single	A 32 bit IEEE floating point value.
double	double	A 64 bit IEEE floating point value.
string	char[size]	A fixed length, zero filled array.
timestamp	int32	A 32 bit time value.

Table 5.1: Default ports for Programs and Elements.

The field names are lowercased from the schema field name element. The structure name is the mixed case name of the schema. A helper method, setStringField, is generated to copy text in a C++ STL String variable into a field in a generated tuple.

```
<schema name="PacketTuple">
    <field  name="time"        type="int" />
    <field  name="protocol"    type="string"  size="4" />
</schema>

struct PacketTuple
{
    int32    time;
    char     protocol[ 4 ];
} __attribute__((__packed__));
```

Figure 5.2: A Structure generated from a schema.

Inputs In C++

```
<input   stream={stream name}  schema={schema name} />
```

A structure is generated for each input stream containing the tuple data and header information used by Borealis to process the tuple. The structure name is the proper cased name from the schema name.

```
<input   stream="Packet"  schema="PacketTuple />


struct  Packet : public TupleHeader
{
    PacketTuple   _data;
} __attribute__((__packed__));
```

Figure 5.3: A structure generated from an input.

Several tuples can be transmitted together to improve network efficiency. Two methods are generated to facilitate sending data to Borealis nodes. The batch method enqueues a tuple and the header to be transmitted. The send method transmits any enqued tuples.

A header is also generated for a method to be written by the application programmer that equeues and transmits tuples. It is called after a previous call to the send method and a delay. The delay is given in milliseconds as the argument to the send method.

The generated names for these three methods is the function followed by the proper case name of the input stream.

```
/// Enque a Packet for input.
///
void  batchPacket(Packet  *tuple);


/// Send enqued Packet inputs.
///
void  sendPacket(uint32  sleep);


/// Return here after sending a packet and a delay.
///
sentPacket();
```

Figure 5.4: Method signatures for the input stream Packet.

Outputs In C++

```
<output  stream={stream name}  schema={schema name} />
```

Note that output schemas can be inferred from a complete diagram, but not in the partial diagrams allowed by the marshal tool. Consequently they are required by the marshal tool (even in a full diagram), but are not required by the Head. The Head uses the output schema declaration as a type check so it is still a good idea to supply it.

```
/// The receivedAggregate method must be defined by the application.
/// It is called after a(n) aggregateTUPLE is received.
///
static void  receivedAggregate(aggregateTUPLE  *tuple);
```

5.2.2 Generated Deployment Diagram Code

The deployment diagram determines how objects are distributed over multiple nodes and connect to multiple application programs. Once objects are deployed optimizers, tools, and applications can relocate them or remove them from the network. As objects are moved any input and output stream connections are managed by the Borealis system.

Methods are generated to initiate deployment. Using these methods an application program can initiate deployment in total or in separate phases. The open method is the simplest way for a monolithic application to activate a Borealis system. It will launch the Head, deploy the Application XML files, and connect all streams to the application. After the Head is finished deploying the network it terminates.

Not every application program will want to launch the Head. If the Head needs to be run persistently instead of terminating it needs to be launched by exterior means (see the section about the Head). Applications consisting of multiple programs only need to launch the Head from one of the programs. It can use the launchDiagram method to only launch the Head and deploy the Application XML files.

An application program that does not want to launch the Head can use the openInputOutput method to connect it's streams. If an application consists of multiple programs the client XML element can be used to identify which streams are connected to each program. Note that the client element is only processed by the marshal program and is ignored by the Head.

```
<client  [prefix={class prefix}]  [endpoint={monitor}] >
    <publish   ... />  ...
    <subscribe ... />  ...
</client>
```

A separate header and program file will be generated for each client program. The common name of the generated class is the proper cased root name of the first Borealis XML file plus the suffix "Marshal". For example, the file name mytest.xml yields the class "MytestMarshal" defined by the files "MytestMarshal.h" and "MytestMarshal.cc". The class produced for each client is the proper cased prefix attribute plus the suffix "Marshal".

After the network is deployed and the streams are connected an application will typically call it's sent methods to activate publication of it's input streams. Once the inputs are initiated the NMSTL event loop can be started using the runClient method. If an application program should ever want to terminate itself it can do so using the generated method, terminateClient.

5.2.3 Publishing and Subscribing to Streams

The publish element directs input to a particular node. For each publish element a connect method is generated. They allow individual streams to be published by a node, should an application want fine-grained control over publications. The name of each method is "connect" plus the proper cased stream name.

```
<publish  stream={input stream}  [endpoint={node}]  />
```

A subscribe element lets the node producing the output stream know to direct it to an application monitor program. For each subscribe element a subscribe method is generated. They allows an individual stream subscription, should an application want fine-grained control over subscriptions. The name of each method is "subscribe" plus the proper cased stream name.

```
<subscribe  stream={output stream}  [endpoint={monitor}]  />
```

Subscriptions to outputs must occur after the outputs are deployed. The node producing the output needs to know about the stream in order to permit and register the subscription. Input connections are unrestricted and can be made before or after deployment.

5.2.4 Application Deployment

The Head uses deployment XML to set up the network on each node. The publish and subscribe elements are described in the previous section 3.2.1.

Box and Table Deployment

A node element corresponds to each Borealis node in the network. The endpoint is the receiving channel for a node and provides unique identification. The query attribute identifies a set of boxes and tables to be deployed to the node. It is a list of query, box or table names (see section 2.3) separated by blanks. If no query element is given then the node is free for use by optimizers and high availablility components.

```
<node  endpoint={node}  [query={query name ...} ] />
```

Regional and Global Component Deployment

Some components such as load optimizers and monitoring tools may need catalog information spanning several Borealis nodes. A region contains an arbitrary set of distributed Borealis nodes. The region element determines which nodes are included in a region. Regions may overlap and not all nodes need to be included in any region. In general nodes are assigned to a region based on some criteria such as connectedness, network latency or physical locality.

The node attribute designates a node to be included in a region and the endpoint attribute designates the regional component. If the endpoint attribute is omitted the default endpoint for regional components is used.

```
<region  node={node}  [endpoint={regional component}] />
```

Similar to Regional Components, a Global Component spans all Borealis nodes. However, a global component is updated through a persistent Head. Consequently it is should run on the same machine as the Head or one with a reliable low latency connection to the Head.

```
<global  endpoint={global component} />
```

Replica Set Deployment

```
<replica_set   name={set name} query={query name ...} >
     <node ... />  ...
</replica_set>
```

See Chapter 6 for details about replica sets.

Chapter 6

Fault-Tolerance

Borealis includes different fault-tolerance mechanisms. One of those mechanisms, called *Delay, Process, and Correct* (DPC), is based on replication and enables a distributed SPE to handle both node failures and network failures. DPC is described in detail in [3] and [4]. We only provide a brief overview here.

6.1 Replication

In DPC, every fragment of the query diagram runs on multiple processing nodes. With this approach, if a node crashes or becomes disconnected, the system can continue processing by using the output of another replica of that node.

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:

```
...
<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>
...
```

6.2 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.
- 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.
- 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 [3] 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.

6.3 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 [3] and [4].

[1] Note that SUnion performs additional tasks when failures occur and heal. See [3] for details.

6.4 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. However, the sunion-new application also demonstrates the use of a replica-set. The fault/faulttest application shows a node running a query diagram composed of many different operators.

Chapter 7

Conclusion

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

[4] 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.

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

[6] M. Cherniack. SQuAl: The Aurora [S]tream [Qu]ery [Al]gebra, Nov. 2003.

[7] Borealis developer's guide. `http://www.cs.brown.edu/research/borealis/public/publications/borealis_%developer_guide.pdf`.

[8] E. Ryvkina, A. S. Maskey, M. Cherniack, and S. Zdonik. Revision Processing in a Stream Processing Engine: A High-Level Design. In *Proc. of the 22nd International Conference on Data Engineering (ICDE)*, Atlanta, GA, April 2006. (to appear).