Aurora Box Research

Christina M. Erwin
Department of Computer Science
Brown University

Submitted in partial fulfillment of the requirements
for the Degree of Master of Science
in the Department of Computer Science
at Brown University

[illegible]
Signature (Professor Stanley Zdonik, Advisor)

1/30/03
Date

I. Aurora Overview

Aurora is a database management system for streaming data that breaks from the current day pull-based architecture. This system assumes that there is data coming from a variety of different input sources, ranging from sensors to other computer programs. The data streams that come from these sources are made up of tuples, which contain a timestamp, an id, and the selected relevant data. Once the data enters the system, it will flow through a loop-free, directed graph of processing operators (i.e., *boxes*) which was created by an Aurora application administrator (a.k.a. user). Boxes are similar to individual, simple queries like a SELECT or UPDATE in SQL. These boxes can be set up to perform complex queries upon the incoming data. Once the tuples have flowed through the relevant operators, the data will be output to other applications. More information about the Aurora project can be found at http://www.cs.brown.edu/research/aurora/vldb02.pdf.

The current set of boxes consists of simple unary operators (Filter, Map and Drop), an n-ary merge operator (Union), a lossy windowed sort (WSort), and an aggregation operator (Tumble). Aurora also includes two additional aggregate operators (XSection and Slide), a join operator (Join), and an extrapolation operator (Resample).

II. Introduction

This summer we completed several efforts including creating a parser for functions and predicates, changing the old boxes (MapBox and SelectBox) to use the parser, and adding a Union box. First, I will lay down the work that was done in the previous semester (Spring 2002) and then I will show how it evolved into what it is today.

This work was done under the guidance of Professor Mitch Cherniack from Brandeis University. The box group consisted of Daniel Abadi, Anurag Maskey, Eddie Galvez (all Brandeis students) and me. The GUI group, especially Jeong-Hyon Hwang and Robin Yan, also contributed to our success.

III. Boxes

Last semester, Andrew Flinders, a fellow Brown student, and I created several boxes. The ones that were the focus of my work were the SelectBox, MapBox and DropBox. We also spent time determining all of the parameters required for the boxes (stored in QBox.H) and the best way to store the data in queues, then read and copy this data into other queues as needed. Much of the code for the specific boxes was re-written after the addition of a parser for the predicates and simple functions. However, we were able to re-use a lot of the base code that gets information from tuples.

Design Issues

One of the first design issues that we debated was whether the box's data should be public or private and accessed through accessor methods. Having private data increases clarity, at times, but it has a slight speed cost. Furthermore, accessor methods are usually used to keep data from being modified by the wrong people (or units, more specifically). Speed was a major issue and the probability of people using the data incorrectly was relatively low. Therefore, we chose to keep the box's data public.

SelectBox

The SelectBox was created to separate tuples that meet user-provided criteria from non-compliant tuples. Using the GUI, the user specifies the desired condition in the form of a predicate. Every time that a tuple is passed through the SelectBox, the specified data within the tuple is compared to the predicate. If the condition is found to be true, the tuple is passed through the SelectBox and is sent to the next box. If the condition is found to be false, this tuple will be dropped from this flow or possibly sent to a different box.

Previously, the SelectBox was only capable of parsing simple predicates of the form LHS OP RHS. LHS could be set to one of the fields in the tuple. OP could be any of the common operators (<, >, <=, >=, =). RHS could be either one of the fields in the tuple or a value (i.e. hard coded number). This short-term solution enabled us to do some preliminary testing. However, as the GUI and other interfaces evolved, the SelectBox did not have sufficient power to meet our needs. We needed the capability for a user to be able to enter a predicate of "infinite" complexity. Then, we needed to find a way to store this expression in a way to best minimize storage and parse this expression such that the run-time system could use this information efficiently.

The SelectBox evolved to meet these needs. The GUI created an interface where the user was able to type in a predicate of "infinite" complexity, which was sent to the catalog as a string. The catalog parsed the predicate immediately into an Expression tree and stored a pointer to the root of the tree into the Box's data (QBox.H). By parsing immediately when it is stored, the storage time increases slightly but the run-time improves immensely. This also removes the possibility of having the string parsed upon every execution of the box, which would have wasted a significant amount of time. Since I was the most familiar with the Box code, I re-designed and wrote the code required to integrate the parser into the SelectBox code. I also added code that allowed the train size of the queue to be greater than one and the code that properly managed the input and output data. I performed testing of this code and verified that it worked as desired. The SelectBox's improved functionality has been successfully demonstrated on several occasions.

Some work is still required to allow the SelectBox to have two output ports, one true and one false. The GUI has already added this feature where the uppermost output would contain all the tuples for which the condition evaluated to true. The lowermost output would contain the tuples for which the condition evaluated to false. The user has the option to connect to one or two of the output ports. There were several choices in how to create this variant on the SelectBox.

First of all, there was the question of whether the GUI should flatten out the SelectBox, if it had two output queues, into two separate SelectBoxes. The first SelectBox would contain the predicate entered by the operator and the second would contain the inverse of the predicate. This option would require no changes to the current implementation of the SelectBox and would keep the number of available boxes to a minimum. However, it would do so at the expense of adding extra boxes to be scheduled and the overhead of running virtually the same predicate on the same data twice. Furthermore, these two boxes would have to be handled as separate entities by the scheduling algorithm. Therefore, we chose to not flatten out the SelectBox.

Since speed is a very important concern for the boxes, we did not want the overhead of checking if this was a one or two output box every time this box was scheduled or a tuple was processed. Therefore, we will most likely create a new box, FilterSplitBox. When the GUI stores the schema to the catalog, if only one output port is used, it will be stored as a FilterBox. In the case where the user only connects to the false output port, the lowermost connection, the predicate will be negated. If there are connections to both output ports then the box will be stored as a FilterSplitBox. Although this increases the number of boxes slightly, the cost of running the boxes will be minimized which is far more important.

DropBox

The DropBox has a rather interesting functionality. This is the only box that is not available from the GUI to be placed into the query structure by the operator. Instead, it is automatically added during run-time by the load shedder process. During run-time, the system is constantly being monitored in order to ensure that it is able to process all the data it is given. In this streaming data system, the arrival times of the data is completely unknown and the system needs to be able to react to problems resulting from high or low load. In cases where the load is so heavy that the system has difficulty processing the data in a timely fashion, it might be necessary to drop some of the load. The system can place these DropBoxes in certain portions of the system in order to drop a desired percentage of the load from the incoming queue.

Currently, the DropBox intakes data and drops the given percentage of tuples. Since we did not want the boxes to have memory, we were not able to do a count wherein every x^{th} tuple would be dropped. Instead, I used a random number generator to output numbers between 0 and 1. The value output from the random number generator is compared to the percentage to drop (converted into decimal form) and if it is greater than the percentage value the tuple is placed into the output queue. Otherwise, the tuple is dropped. This code has not yet been formally tested because it is not available from the GUI. For testing purposes, it will be necessary to temporarily add a DropBox to the GUI, which will allow the user to add them to their schemas.

MapBox

The MapBox can change the structure of the tuple, the data within the tuple, or both. Via the GUI, the operator can specify the form of the tuple to be output from this box. The user can then specify the data that will fill each field. The data output into the fields can consist of hard-coded values, the original values, or even manipulation of the different fields from the input tuple. This is the only box in which the input tuple can differ in form from the output tuple.

As of the spring semester, the MapBox only had the capability to run several hard-coded, pre-defined functions. There was no GUI support for selecting which of the functions to run on incoming tuples or for creating user-defined functions. Furthermore, since there was no GUI support the input tuple had to have the same form as the output tuple. Therefore, our functions did operations like changing one field in the tuple from Celsius to Fahrenheit, for example, and other such simple operations. Although having some pre-defined functions would be beneficial, the inability of the user to specify the information in the new tuples did not meet our needs. This was never demonstrated since it was too preliminary and didn't support the capabilities for which MapBox exists.

The MapBox can now create new tuples as specified by the operator. The GUI group created an interface for creating new tuples where the user can specify the number of fields in the tuple and the values for fields. These values can be as simple as just being a number or a string or it can be computed by combining or manipulating values from the incoming tuple. The routines for computing the new fields were stored as an array of functions.

Just as with the predicates, we needed an efficient way to store these functions. Again we chose to store them in string form and then parse these strings every time that the function is run. The Brandeis group created a function parser, which processes and outputs the desired tuple for each function. I changed the MapBox to loop through all the functions with the current tuples in the input stream and properly manage the output data.

UnionBox

The UnionBox was made to merge input streams with the same tuple structure and to output one stream containing all the tuples from the input streams. If the tuple types do not match, the MapBox must be used on the non-matching streams to create matching tuples before the UnionBox can be used. The UnionBox performs the converse operation to the RestreamBox, which enables output from one box to be duplicated and sent to multiple boxes.

The addition of the UnionBox forced changes in the QBox superclass. The UnionBox can have an "infinite" number of input streams, whereas all of our other boxes had either one or two input streams. Unfortunately, we did not foresee the addition of this box and had coded all the other boxes to use either stream input 1 or stream inputs 1 and 2. We were forced to consider how to best implement this n-ary box so that multiple arcs could flow into a single box. Several designs were considered to accomplish this. First of all, there was the possibility of flattening out the Union box such that it was made up of a tree of many binary input boxes. This would have made it possible to retain the unary and binary inputs; however, it would have done so at the cost of high overhead and more complicated scheduling and execution of this box. We also considered the possibility of using a vector of inputs only for the UnionBox and maintaining the unary and binary inputs for the other boxes. This could have been done but at the loss of commonality for the boxes. Thus, we decided that the best option would be to update all of the boxes to have n-ary inputs (i.e. a vector of inputs). This creates commonality between the boxes at little to no increased storage or run-time cost.

We spent some time trying to decide whether Union should do some preliminary time stamp sorting. We had the option of making Union put the tuples into the output stream in order by comparing all the tuples at the heads of the input queues and putting the tuple with the earliest tuple time into the output stream first. It would continue eating through the queues in this manner until all the input queues were empty. The problem with this type of sorting is that we are unable to sort any tuples that are out of order within a single queue. This forces us to question whether or not this preliminary ordering is worth it. It is important to consider the boxes that might follow a Union so that we can see what the effect of this lack of ordering might be on them. This will have no negative effect on either Map or Select boxes because they are completely independent of order. It should not effect the Join box, either, since it will perform time checks on its own if they are needed. The Tumble box can compensate for the lack of timestamp order by using the SLACK parameter, which enables a group of outputs to be retained for the selected amount of time. Though we must concede that some data may still be lost if the SLACK variable is not large enough. The Slide box would need a WSort box in front of it in order to compensate for the lack of ordering. We must ensure that the WSort succeeding a Union is not too lossy because of the fluctuating timestamp data provided by the Union. It would be interesting to see how preliminary ordering done by Union might aid in the WSort ordering and if the poor ordering provided by Union adversely effects the ordering done by WSort.

IV. Future Work

We hope to add several advancements to the current architecture as time permits, including the ability to add user-defined functions, the creation of actual functions from user created functions, and the addition of pre-defined functions. Although these are very similar problems, they each have their own intricacies.

The ability to create user-defined functions adds to the strength of the system. It enables users of this system to add functions required for the work they are trying to perform without waiting for a new revision to come out with their desired functionality. There was much discussion about user-defined functions and how they should be added to the system. There are several difficult issues to overcome in order to enable us to put user-defined functions to use. There are new advances in Java using Java Spaces that would make storing these functions rather simple; however, since our code is C++ based we could not put this option to use. We did not want the catalog to have to put aside space for these functions since their sizes would be unknown, though we did need the catalog to be aware of all the available functions. There are issues with not only where to store the functions but how to store the functions such that they would be easily accessible and would not be placed on memory which would be changed. There also needs to be a way to not only add functions but to remove functions that do not work as desired. We would also need GUI support to enable the user to name functions.

It would be beneficial to have a way to create an actual compiled function from the user-created functions used in the MapBox. As explained briefly above, the MapBox simply loops through all the functions (which are made up of trees of expressions) with the same input tuple and incrementally outputs the fields in the output tuple. Clearly, it would behoove us to create a single, compiled function to which we would input the tuple and from which the output tuple would be produced. This would improve run-time speed and most likely would have about the same storage space since the whole expression tree would no longer need to be stored.

The addition of pre-defined functions would enable the user to quickly create his desired effect upon tuples. This would most likely be selectable when the user is manipulating data in a MapBox in order to simplify conversions. These functions would be limited to simple operations like converting from radians to degrees, from Fahrenheit to Celsius, or other such conversions.

V. Conclusion

In conclusion, a MapBox, DropBox, SelectBox and UnionBox were created which meet all the requirements that we initially had set out. These boxes all have public methods and variables to make their data more quickly accessible. Furthermore, they all have n-ary inputs vice either unary or binary as a result of the addition of the UnionBox. The boxes have been maximized for speed whenever possible since they will need to be loaded over and over again and an inordinate amount of data will be run through them. These speed concerns make it necessary for us to consider some of the enhancements for storage and use of functions.