Revision Processing in a Stream Processing Engine: A High-Level Design

Esther Ryvkina[1] Anurag S. Maskey[1] Mitch Cherniack[1] Stan Zdonik[2]
[1]Brandeis University, Waltham, MA [2]Brown University, Providence, RI
{essie,anurag,mfc}@cs.brandeis.edu sbz@cs.brown.edu

1 Introduction

Data stream processing systems have become ubiquitous in academic [1, 2, 5, 6] and commercial [11] sectors, with application areas that include financial services, network traffic analysis, battlefield monitoring and traffic control [3]. The append-only model of streams implies that input data is immutable and therefore always correct. But in practice, streaming data sources often contend with noise (e.g., embedded sensors) or data entry errors (e.g., financial data feeds) resulting in erroneous inputs and therefore, erroneous query results. Many data stream sources (e.g., commercial ticker feeds) issue "revision tuples" (*revisions*) that amend previously issued tuples (e.g. erroneous share prices). Ideally, any stream processing engine should process revision inputs by generating revision outputs that correct previous query results. We know of no stream processing system that presently has this capability.

In this paper we describe the high-level design for the revision-processing facility of Borealis [1, 4]. Note that revisions (on which our work focuses) are not the same as *updates*, which have been discussed extensively in the context of streams (e.g. [7, 8]). Revisions are *corrections* as they *invalidate* previously processed inputs, and by implication, all query results that were produced from them. On the other hand, updates close the time interval during which previously processed inputs were valid, and therefore do not invalidate any previously output query results.

The paper proceeds as follows. We begin by presenting the goals for the design of the revision processing component in Borealis in Section 2. We then present two approaches to revision processing and an example that illustrates them in Section 3. We conclude by discussing future work in Section 4. Further details and implementation considerations can be found in our technical report [10].

2 Design Goals

In designing a revision processing component for Borealis, our goals are twofold:

- **Goal #1:** Minimize disruption to the existing Borealis data model and runtime system, and
- **Goal #2:** Impose as few constraints as possible on a Borealis application performing revision processing.

The Borealis stream processing engine (SPE) is an extension of the Aurora SPE [4]. Both use the *boxes-and-arrows* paradigm that is found in most workflow systems to express continuous queries. Each box denotes a query operator and each arrow (*arc*) between two boxes represents the stream of data (consisting of flat tuples) that is output from one box and sent as input to the next. An arc can be annotated with a *connection point*: a repository containing all "recent tuples" that have been seen on that arc. A *query network* consists of a set of continuous queries over a fixed set of input streams. A *scheduler* schedules boxes, which when scheduled, process tuples in their input queues and deliver outputs to the input queues of subsequent boxes.

To minimize disruption to the existing Aurora/Borealis system, we *extend* rather than *modify*, the data model and runtime engine. Revisions extend the data model by adding a distinction between *insertions* (which add to the contents of a stream), *deletions* (which signal that a previously seen tuple should be removed), and *replacements* (which change the value of a previously seen tuple)[1]. This generalizes the Aurora model which consists solely of insertions. Further, the operators (boxes) of Aurora are extended to process these new forms of tuples. Thus far, we have designed extensions for the following operators: **Map** (which applies a function to every tuple on the stream), **Filter** (which applies one or more predicates to a tuple and routes the tuple to the output that corresponds to the predicate that evaluated to true), and **Aggregate** (which applies an aggregation function, e.g. MAX, AVG, to windows of tuples at a time). To minimize disruption to the runtime engine, all new components introduced to process revision tuples are designed as special-purpose "boxes", thereby making them schedulable when tuples appear on their input queues. For example, connection points are special-purpose boxes which process input tuples by updating the contents of their repositories. *Sweepers* (described in Section 3.3) are similarly designed.

Any Borealis query network can process revisions provided that it satisfies the minimal constraint of having a connection point on each input stream upon which revisions can arrive. This constraint ensures that any prior computation can be recomputed from its original inputs, and also

[1]Taken collectively, revisions refer to insertions that arrive out-of-order, replacements, and deletions.

Proceedings of the 22nd International Conference on Data Engineering (ICDE'06)
8-7695-2570-9/06 $20.00 © 2006 IEEE

IEEE COMPUTER SOCIETY

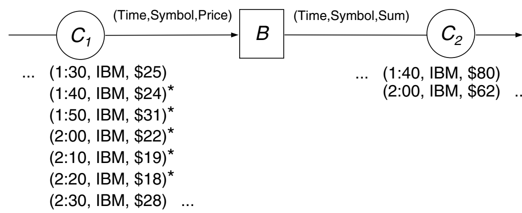

Figure 1. Aggregate Revision Processing

adds no additional historical storage requirements to an application beyond those required to ensure high availability, as described in [9]. Additional connection points can be placed on any arc to potentially improve the performance of revision processing, though these are not strictly required for functionality.

3 Revision Processing

We introduce two approaches to revision processing.

1. *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.

2. *Downstream processing* "retrieves" all previously produced *output* tuples to which the tuple being revised originally contributed, 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.

Both of these approaches require storing a certain amount of *history* (i.e. tuples that previously flowed in the system) which is maintained by connection points. A box in Borealis can always process revisions in upstream processing mode and can sometimes process revisions in downstream processing mode depending on the type of the box, its location in the network and the locations of connection points.

3.1 Example

To illustrate the difference between upstream and downstream processing, consider the example in Figure 1 which shows a query network consisting of two connection points (C_1 and C_2) and an **Aggregate** box (B), which computes the sum of prices of IBM shares during 30 minute intervals every 20 minutes. Figure 1 shows the partial contents of C_1 and C_2 at the point when a revision, t, which revises the IBM 2:00 share price from $25 to $22, arrives at box B, and after having already corrected the 2:00 share price in C_1. Note that the 1:40 and 2:00 sums of share prices stored at C_2 still reflect the erroneous 2:00 price of IBM of $25.

If B operates in upstream processing mode, it will request from C_1 all tuples from the two windows that contained the original IBM 2:00 quote: 1:40-2:09 and 2:00-2:29. The tuples contained in either of these two windows are shown with an asterisk. For each of these two windows, the old sum is recalculated using the original value of $25 and the new sum is calculated using the revised value of $22. The results are compared and since they are different for both windows, two revisions are emitted - one revising the 1:40 sum from $80 to $77, and the other revising the 2:00 sum from $62 to $59.

If B operates in downstream processing mode, it will request from C_2 the results ($80 and $62) that it previously emitted for the 1:40-2:09 and 2:00-2:29 windows. For each of these results, the revised result is derived by subtracting $25 from and adding $22 to these previous results. This results in identical revisions that would be output in upstream processing mode.

3.2 Upstream Processing

The key idea behind upstream processing is to replay the *input* tuples required to produce previously generated *output* results, and to regenerate these results using both the original and revised values reported by the revision[2]. Upstream processing has minimal connection point requirements (only requiring connection points on the input streams to the query network) and any box can be made to process revisions in this mode. On the other hand, upstream processing must recreate outputs from scratch rather than revising them directly (as in downstream processing).

In upstream processing mode, a stateless box (e.g. **Map** or **Filter**) processes a revision without requesting any previously processed (*historical*) tuples. In our design, we assume that a replacement contains the revised value and the original value of the tuple being revised, a deletion contains the original value of the tuple being deleted, and an insertion contains just the new value. The box uses these values to produce the original and revised results and issue a revision if the results are different. In order to process a revision by a stateful box (e.g. **Aggregate**) in upstream processing mode, all tuples that were in the same windows as the tuple being revised need to be replayed. The nearest upstream connection point is designated as the source for those tuples (the *anchor*). The anchor places these tuples into its associated arcs (queues) and these get processed and propagated downstream. Note that a connection point must replay all historical tuples needed by all boxes that have this connection point as their anchor, so that boxes can process that revision and all resulting revisions produced in flight. The number of tuples to replay is specified by the *radius* of the connection point, which is calculated statically on the basis of window sizes of **Aggregates** in the query network when the network is created. When **Aggregate** processes a revision, it recalculates the original and revised results using the original and revised values respectively and emits a revision

[2]Upstream processing is so named because the connection point containing these tuples must be upstream of the box processing revisions.

Proceedings of the 22nd International Conference on Data Engineering (ICDE'06)
8-7695-2570-9/06 $20.00 © 2006 IEEE
IEEE COMPUTER SOCIETY

for each window if the results are different.

3.3 Downstream Processing

The key idea behind downstream processing is to request previously generated *output* tuples and correct them using the original and revised values of the revision. The original result is obtained from a downstream connection point and the revised result is derived from it. Unlike upstream processing, which any box can use to process revisions, downstream processing can only be used by certain boxes in the query network. Specifically, a box can use downstream processing to process revisions if there is a connection point downstream from it, and all intermediate boxes on the path from this box to the connection point satisfy certain properties that guarantee that the set of output tuples is generated from a unique and identifiable set of input tuples (see [10] for details).

In downstream processing mode, a stateless box processes a revision in the same way as it would in upstream processing mode because it does not need to request any historical tuples. When a stateful box in downstream processing mode processes a revision, it must retrieve previously output results generated using the tuple being revised. The box first stores the revision and emits request tuples (*requests*) for these outputs. A connection point downstream from the box processes these requests by sending the requested outputs as historical tuples to the box via a *sweeper*: a schedulable "box" that places these tuples in the requesting box's input queue, thus serving as a surrogate "backchannel" between a connection point and boxes upstream from it. When the box receives the requested tuples, it calculates revised results based on the original results (received from connection point) and the revision (previously stored), and emits revisions if the results differ.

4 Status and Future Work

We have implemented a functional, but as yet untuned, revision processing component of Borealis that performs both upstream and downstream revision processing. After we finish tuning the system, we intend to explore the following issues.

1) Since a revision processed by stateful boxes, in general, produces several revisions (as one tuple contributes to more than one window), one revision at the input to the query network can produce many revisions throughout the system and at the outputs. Thus, the number of revisions in the network depends not only on the number of revisions at the inputs but also on the characteristics and number of stateful boxes. If not controlled somehow, this proliferation of revisions can hamper the performance of the system. To address this concern, we plan to explore *approximate revision processing* – processing and emitting only those revisions which will make a significant difference in the end result.

2) We also intend to study the performance differences between upstream and downstream processing, both with respect to throughput of the system and the proliferation of additional tuples to process revisions. It is our conjecture that in many cases, downstream processing is preferable to upstream processing in both respects. (Consider, for example, the case where a revision affects the computation of an aggregate over a 1000 tuple window that advances by 1000 tuples. In this case, upstream processing requires recomputing two aggregate results over 1000 tuples for each arriving revision, whereas downstream processing requires simply retrieving one output tuple and revising it.) We intend to explore the characteristics of Borealis query networks that make one or the other mode of revision processing preferable, to suggest where connection points could be added to best improve revision processing performance. This will also entail adding support for *adaptive revision processing*: the dynamic switching of processing modes in response to addition or removal of a connection point.

References

[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 *Second Biennial Conference on Innovative Data Systems Research (CIDR 2005)*, Asilomar, CA, January 2005.

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

[3] A. Arasu, M. Cherniack, E. Galvez, D. Maier, A. Maskey, E. Ryvkina, M. Stonebraker, and R. Tibbetts. Linear Road: A Stream Data Management Benchmark. In *VLDB Conference*, September 2004.

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

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

[6] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 29(2), 2000.

[7] T. M. Ghanem, M. A. Hammad, M. F. Mokbel, W. G. Aref, and A. K. Elmagarmid. Query Processing using Negative Tuples in Stream Query Engines. Technical Report CSD 04-040, Purdue University, 2005.

[8] L. Golab and M. T. Ozsu. Update-Pattern-Aware Modeling and Processing of Continuous Queries. In *ACM SIGMOD Conference*, Baltimore, MD, June 2005.

[9] J.-H. Hwang, M. Balazinska, A. Rasin, U. Çetintemel, M. Stonebraker, and S. Zdonik. High-Availability Algorithms for Distributed Stream Processing. In *IEEE ICDE Conference*, April 2005.

[10] E. Ryvkina, A. S. Maskey, I. Adams, B. A. Sandler, C. Fuchs, M. Cherniack, and S. Zdonik. Oops, I Streamed it Again: Processing Revision Tuples in a Stream Processing Engine. Technical report, Brandeis University, June 2005. URL: http://nms.lcs.mit.edu/projects/borealis/revisions_techreport_06.pdf.

[11] StreamBase Systems, Inc. URL: http://www.streambase.com/.

Proceedings of the 22nd International Conference on Data Engineering (ICDE'06)

8-7695-2570-9/06 $20.00 © 2006 IEEE

IEEE COMPUTER SOCIETY