Replay-Based Approaches to Revision Processing in Stream Query Engines

Anurag S. Maskey
Brandeis University
Waltham, MA 02454
anurag@cs.brandeis.edu

Mitch Cherniack
Brandeis University
Waltham, MA 02454
mfc@brandeis.edu

ABSTRACT

Data stream processing systems have become ubiquitous in academic and commercial sectors, with application areas that include financial services, network traffic analysis, battlefield monitoring and traffic control. 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 by implication, erroneous query results. Many data stream sources (e.g., Reuters ticker feeds) issue "revision tuples" (*revisions*) that amend previously issued tuples (e.g. erroneous share prices). A stream processing engine might reasonably respond to revision inputs by generating revision outputs that correct previously emitted query results. We know of no stream processing system that presently has this capability.

In this paper, we describe how a stream processing engine can be extended to support revision processing via *replay*. Replay-based revision processing techniques assume that a stream engine maintains an archive of recent data seen on each of its input streams. These archives are then queried in response to a revision, with the resulting tuples replayed through the system so as to generate corrected query outputs. We first present the design and implementation of the revision processing engine for the Borealis stream processing engine [1]. We then compare techniques for archiving streams to support replay, and then compare the performance and overhead of two revision processing techniques that replay input tuples to recompute and thereby revise previously output query results. These experiments reveal scalability issues due to the overhead required to maintain stream archives, and has motivated our current research on using sampling and data summarization (e.g., histograms) to reduce the data that must be stored in a stream archive.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems–*Query Processing*

General Terms

Design, Experimentation.

Keywords

Revisions, Corrections, Revision Processing, Streams.

1. INTRODUCTION

Stream processing systems have become ubiquitous in academic [1, 2, 4, 7] and commercial [18] 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 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. So common are such errors in practice, that many ticker feeds (e.g. Reuters) issue *revision tuples* (or, *revisions*) to amend previously issued tuples. For example, a revision might announce that the 2:00 price of $75/share for IBM that was reported earlier should in fact have been reported as $72/share. This is an example of a *replacement revision* (or, just *replacement*): a revision that changes the value associated with a non-key field of a tuple in the stream (in this case, `Price`). A revision can also be a *deletion*, which reports that a tuple that was previously issued should not have been, or a *late insertion*, which reports a tuple that should have been issued previously but wasn't.[1]

Depending on the needs of the stream processing application, a stream processing engine might respond to the IBM share price revision in any of the following ways:

1. *Ignore it.*
2. *Revise stored history*: Modify any *stream archive* that recorded the incorrect 2:00 price for IBM.
3. *Revise previously emitted query results*: Correct (by issuing revisions) any streaming query result previously emitted by the stream processing engine and invalidated by the revision.

The three options shown reflect an increasingly sophisticated and complex response to a revision. The first option is what stream processing engines we know of do by default. The second option assumes that all streamed tuples are stored in a *stream archive* [6] to enable ad hoc querying or to support high availability [11], and that revisions are processed by modifying invalidated data in the archive. For example, a system responding in this way to the revision would ensure that any *future* query computation sees that IBM's price at 2:00 was $72. The third option not only modifies the stream archive, but also revises previously emitted stream query outputs generated from the original, incorrect quote. For example, a system responding in this way to the revision would itself issue a revision that corrects the hourly average for IBM from 2:00-2:59 that it previously emitted. From conversations with customers who use stream processing engines (especially in financial services), we have learned that the latter of the three options is often the sole acceptable response to a revision, but that it presently falls on stream

[1] A late insertion is sometimes referred to as an *out-of-order* tuple.

query application designers to ensure that revisions are handled in this way. But leaving an application to process revisions adds unnecessary complexity to the application, is error-prone and can miss potential optimizations. Our work focuses on how a stream processing engine can internally support revision processing.

In this paper, we consider *replay-based* techniques to revision processing. Replay-based techniques process revisions by replaying (i.e., reprocessing) all tuples that were originally processed with the revised tuple so as to regenerate and revise the results originally generated from these inputs. Our contributions are enumerated below:

1. *A performance comparison of replay-based revision processing techniques*: In Section 3, we introduce two replay-based revision processing techniques and compare their performance in Section 4.2.

2. *Identification of an effective storage strategy for stream archives*: All replay-based techniques require maintaining an *input stream archive*: a very large window of the most recently received tuples on every input stream. In Section 4.1, we show that horizontally partitioning is an effective strategy for maintaining and querying an input stream archive.

3. *Identification of stream characteristics which make replay-based revision processing practical*: We show in Section 4.2 that replay-based revision processing techniques do not scale well to streams with high arrival rates because of the overhead of inserting stream data into stream archives. This observation has motivated follow-up work on *approximate revision processing* (described in Section 4.3) which incurs lower overhead at the expense of sacrificing some accuracy in responding to revisions.

The paper proceeds as follows. In Section 2, we present a set of requirements that must be satisfied by any stream processing engine for which these replay-based techniques would apply. We also describe the Borealis stream processing engine [1] and show how it was adapted to meet these requirements. In Section 3, we present two different replay-based revision processing techniques. Then in Section 4, we compare possible representations of stream archives that are required to perform replay, and then compare the performance and overhead of the two replay-based revision processing techniques as implemented on Borealis. Finally, we discuss related work in Section 5 and conclude in Section 6.

2. BACKGROUND

It is important to distinguish *revisions* (with which this paper is concerned) from *updates* [8, 9]. Unlike updates, revisions are **corrections**. That is, they *invalidate* previously processed inputs and by implication, all query results that were produced from them. On the other hand, updates do not invalidate previously processed inputs but simply end the interval during which they were valid. Consider Figure 1 as an example. Figure 1a shows the price of IBM shares between 2:00 and 2:59 assuming a 2:00 quote that reported a price of $75, and a 2:30 *update* that changed this price to $72. Figure 1b shows the price of IBM shares between 2:00 and 2:59 assuming a 2:00 quote that reported a price of $75, and a 2:30 *revision* that corrected the 2:00 price to $72. Note that the average price of IBM shares from 2:00 to 2:59 is $73.50 as the result of the 2:30 update, but $72.00 as the result of the 2:30 revision.

We begin in Section 2.1 by describing two classes of techniques (*replay-based* and *direct*) for revision processing and show that replay-based techniques are more generally applicable. Then in

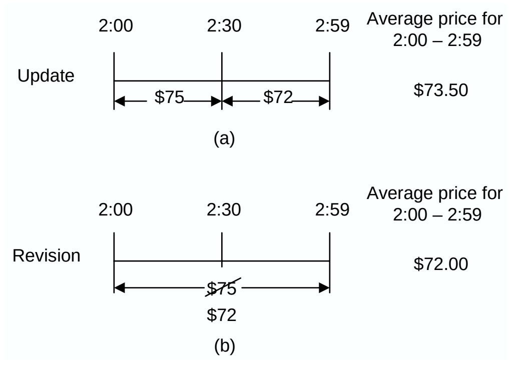


Figure 1: Updates vs. Revisions

Section 2.2, we describe the generic functionality required of a stream processing engine to process revision using replay, and then in Section 2.3, describe how the Borealis stream processing engine [1] was extended to meet this functionality.

2.1 Approaches to Revision Processing

There are two possible approaches to processing revisions:

1. ***replay-based*** approaches respond to a revision by reprocessing (*replaying*) the sequence of input tuples used to generate any of the query results invalidated by the revision, and

2. ***direct*** approaches respond to a revision by finding all query results invalidated by the revision, and modifying these results to reflect the effects of the revision.

To illustrate, consider a continuous query over a stream of quotes that reports the hourly average price of every stock on the hour. In response to a revision to the 2:00 price of IBM stock, a replay-based technique might recompute the 2:00-2:59 average price by replaying all IBM quotes that were issued between 2:00 and 2:59 (while revising the 2:00 quote according to the revision). On the other hand, a direct technique would look up the average IBM price emitted for the 2:00-2:59 timeframe (as well as other data required to revise this result, such as the number of quotes used to generate that average), and adjust this result according to the revision. Both approaches would then emit a revision reporting the correction of the 2:00-2:59 average price of IBM.

The example above illustrates the potential advantage of the direct approach; rather than reprocessing an hour's worth of quotes, it need only look up one average price computation and adjust it. But while this approach is often advantageous for speed, it is not generally applicable to all stream queries. For example, a query that reports an hourly maximum price for every stock cannot be corrected using the direct approach if the revision applies to the price that was previously reported as the maximum. In general, the direct approach to revision processing is only applicable when a revision can be used to correct previously output query results without requiring the additional tuples that were used for their computation. On the other hand, replay-based techniques enable revision processing for any stream queries. Ideally, a stream processing engine would use some combination of replay-based and direct approaches to respond to revisions, depending on the queries involved. In this paper, we focus our study exclusively on pure replay-based revision processing techniques.

2.2 Requirements for Replay

For a stream processing engine to process revisions using a replay-based approach, it must satisfy the following requirements:

1. ***Ordered and Keyed Streams***: The tuples on any input stream are processed in ascending order of some field of the stream (the stream's *order field*). Typically, this attribute is a timestamp that marks the time when the tuple was *issued*, but in the absence of such a field, a stream processing engine can add a timestamp field that marks the time when the tuple *arrived* to the system. Note that there is **no** requirement that tuples be processed in the order that they arrive to the system. Systems such as STREAM [2] and Borealis [1] employ a buffering strategy at the edges of the system to reorder tuples whose disorder, according to the order field, does not exceed some declared threshold.

   Every input stream must also have a declared set of attributes which serve as a key for the tuples that arrive on the stream. For example, a stream of stock quotes might be keyed by `Time` and `Symbol` given that a given symbol will have a single price at any given time. This key serves as an identifier for tuples being revised.

2. ***Input Stream Archives***: Input tuples from every input stream must be stored so as to enable a stream processing engine to replay a set of tuples through the system in response to a revision. A stream archive for a given input stream acts much like a very large window over the stream, maintaining a set of the last τ tuples to arrive on that stream, for some declared threshold τ. Note that input stream archives are often maintained by stream processing engines (e.g., StreamBase [18]) to support high availability [11] and historical ad-hoc queries. Thus in such systems, revision processing can leverage these archives and be supported without substantial changes to the architecture.

3. ***Deterministic Execution***: For the result of replay to be meaningful, a stream engine must have deterministic execution. That is, given the same sequence of input tuples and set of queries, a system should always produce the same query answers. This precludes, for example, queries that *timeout* as the output of such a query can vary depending on the difference in arrival times of two consecutive inputs on a stream.

Ordered streams are required to ensure that replayed tuples are replayed in the order that they were originally processed. Keyed streams are required for a revision to uniquely identify the input tuple that it revises. Input stream archives are required as the source of tuples that get replayed. Deterministic execution is required to ensure that replay-based revision processing corrects only those query outputs that are invalidated by a revision.

For simplicity, we assume in this paper that a replacement revision contains the incorrect value of the revised field as well as its corrected value. Note that if the incorrect previously reported value cannot be provided by the stream's source, we would simply add this value upon the revision's entry into the system via a lookup on the input stream archive. With an appropriate archive representation (see Section 4.1), such a lookup takes roughly 30 msec, which is negligible compared to the time required to process a revision.

2.3 Borealis

All of the revision processing techniques described in this paper were implemented on top of Borealis [1]: a second-generation distributed stream processing engine out of Brandeis, Brown, and MIT. Borealis uses the *boxes-and-arrows* paradigm that is found in most workflow systems to express continuous queries. Figure 2 illustrates the Borealis system model. In this Figure, a box (B_i) denotes a query operator and an arrow (or *arc*) between two boxes represents the stream of data that is output from one box and sent

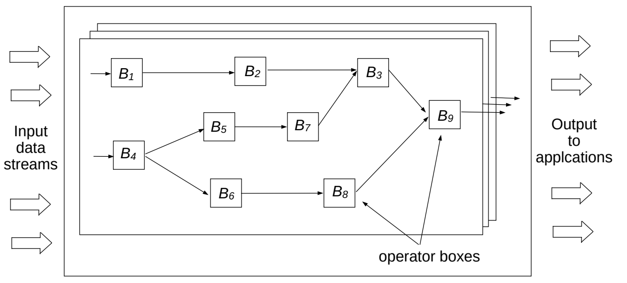


Figure 2: The Borealis System Model

as input to the next. We refer to a workflow diagram that shows one or more continuous queries over some set of input stream as a *query network*. A *path* in a query network is a sequence of boxes that are visited when traversing from an input stream to an output stream (e.g., $< B_1, B_2, B_3, B_9 >$ in Figure 2).

Like other stream processing engines such as STREAM [2] and TelegraphCQ [5], Borealis includes query operators that perform filtering, transformation, aggregation, correlation and storage: **Filter** routes or filters incoming tuples according to predicates that they satisfy; **Map** transforms individual tuples by adding, removing or adjusting the values of fields of incoming tuples; **Aggregate** is a windowed aggregation operation that supports both SQL-like *row-based* or *range-based* windows; and **Join** is a binary windowed stream correlation operation.

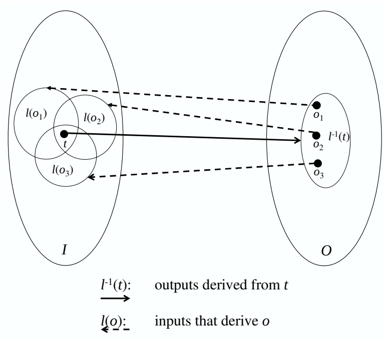
We have modified the Borealis implementation by adding revision processing support to every operator, as well as modifying the run-time system to meet the requirements for replay-based revision processing described above. Ordered streams were guaranteed by the input streams generated for our experiments (for the order field, `Time`). Support for keyed streams was added by allowing keys to be declared as part of the schema of each input stream. Input stream archives were added to Borealis by adding Postgres tables to maintain a recent history of tuples seen on all input streams.[2] Deterministic execution was achieved by disallowing timeouts in all queries supporting revision processing, and by globally timestamping input tuples across all streams in the order in which they were processed so as to ensure that they can be processed in the same order during replay.

3. REPLAY-BASED TECHNIQUES

All replay-based techniques respond to a revision by reprocessing (i.e., replaying through the query network) all input tuples that would have contributed to any query output to which the revision applies. For replacements and deletions of some input tuple t, this consists of all tuples that were involved in the same computations as t. For late insertions of some tuple t, this consists of all tuples that would have been involved in the same computations as t had it arrived "on time" (i.e., in order).

As an example, consider a windowed aggregation stream query, q that computes a 5 minute average price of a stock every minute. Suppose that the first 100 inputs streamed to this query are: $T \;=< t_1, \ldots, t_{100} >$ such that each tuple, t_j, reports a price as of minute j. Then, a revision to t_{20} (correcting the price reported) could be answered in a replay-based system by *replaying* (and therefore reprocessing through q) the *replay sequence*, $T_{RS}(t_{20}) =< t_{16}, \ldots, t_{24} >$, as this sequence contains all tuples from the original input that contributed to the same 5 minute average computations as t_{20}. In fact, this sequence is the *minimal*

[2]We compare representations of stream archives in Section 4.1.



$l(o_1)$
$l(o_2)$
$l(o_3)$
t
o_1
o_2
o_3
$l^{-1}(t)$
I
O
$l^{-1}(t)$: outputs derived from t
$l(o)$: inputs that derive o

Figure 3: Minimal Replay Sequence of t

replay sequence ($T_{MRS}(t_{20})$) as it contains no extraneous tuples that did not contribute to the same query outputs as t_{20}.

More formally, we define a *minimal replay sequence* as follows:

Given the set of inputs, I, and outputs, O, to a stream query, q, let l be the *lineage mapping function* [19] that maps every output, $o \in O$, to the subset of I from which o was derived ($l(o)$). Then, the minimal replay sequence of t wrt q is:

$$T_{MRS}(t) = \bigcup_{o \in l^{-1}(t)} l(o).$$

The lineage mapping function determines the *lineage set* of an output tuple, which is the set of of all input tuples used to produce it. Figure 3 illustrates. Observe that in this example, input t, is contained in the lineage sets of outputs: o_1, o_2 and o_3. Thus, the minimal replay sequence of t is:

$$T_{MRS}(t) = l(o_1) \cup l(o_2) \cup l(o_3).$$

Although the above example considered the revision to t_{20} to be a replacement that corrected the price previously reported, the minimal replay sequence of a revision that deletes t_{20} is also the same and determined in exactly the same way. Now, consider a situation where tuple t_{20} did not get reported until after all the tuples in T (i.e., until after t_{100}). In this situation, t_{20} is considered a late insertion. We know the query results that t_{20} would contribute to had it arrived on time. Thus, when it is late, these query results must be corrected. The minimal replay sequence is this case is also the same as when the revision is a replacement or deletion. The process of determining the minimal replay sequence is the same for all types of revisions: first, determine which query results will change as a result of processing the revision, then determine and replay the input tuples that are required to produce those affected query results.

In general, a revision processing technique can return as a replay sequence, any supersequence of the minimal replay sequence, though extraneous tuples included in the replay sequence may result in corrections of results that have not changed (e.g., such corrections may indicate that an average price has changed from \$20 to \$20). These are easily filtered from the result of revision processing, though represent unnecessary computation performed in response to a revision. The replay-based revision processing techniques that we present in this paper differ by the replay sequences that they generate in response to a revision. Those that return a replay sequence that is a close approximation of the minimal replay sequence, do so by incurring some overhead while processing standard tuples. Conversely, those that return large supersequences of the minimal replay sequence incur less overhead but must replay and reprocess more data.

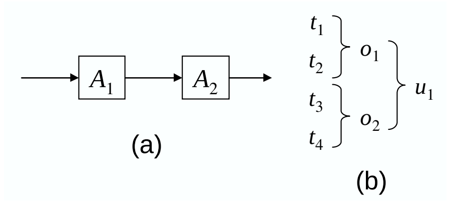


Figure 4: Query Network with Two Aggregates

3.1 Replaying a Stream in Borealis

To support replay, all input tuples to a Borealis query network are automatically stored in a dedicated *input stream archive* whose schema includes the attributes shown below in Table 1.

Attribute	Value
`Str`	an identifier for the stream on which the tuple arrived
`Time`	a timestamp indicating when the tuple arrived into the system
`Ord`	the value of the order field for this tuple
`Body`	a byte-sequence that constitutes the contents of the tuple

Table 1: Schema of the Input Stream Archive

The input stream archive is associated with a *size threshold*, τ, which constrains how many of the most recently arrived tuples on the streams the archive contains. Thus, τ serves doubly as a size constraint on the stream archive, as well as a "rough" bound on the age of the tuples to which a revision can be applied. In fact, a revision may not be able to refer to the oldest tuples in the stream archive if regenerating all outputs that involved those tuples also requires replaying older tuples not contained in the archive. In this case, our system would throw a run-time exception in response to the revision. In Section 4.1, we compare possible representation schemes for stream archives.

In response to a revision, every replay-based revision processing technique does the following:

1. determines a replay sequence for that revision,
2. issues a *replay query* to the stream archive to request the tuples in the replay sequence, and
3. reprocesses the tuples returned by the replay query, generating revisions for any query results that have changed as a result of the revision.

Note that newly arriving tuples are buffered at the input streams by Borealis while a revision is being processed. If the queries have stringent real-time deadlines to generate query results, then revision processing can be deferred to periods of low workloads or can be assigned to a different machine dedicated for revision processing.

To illustrate how replay-based revision processing is implemented in Borealis, Figure 4a shows a Borealis query network consisting of two **Aggregate** operators: A_1 and A_2 such that both A_1 and A_2 have row-based windows of size 2, and both windows *tumble*, i.e., if A_1's first window is $\{t_1, t_2\}$, its next window will be $\{t_3, t_4\}$, and so on. For the purposes of discussion, we will refer to the tuple produced by A_1's calculation over window $\{t_1, t_2\}$ as o_1, and its calculation over window $\{t_3, t_4\}$ as o_2.

Suppose that a revision that modifies t_3, is input to this query network. This revision would affect (only) output o_2 of A_1. To generate a revision for o_2, input tuples $\{t_3, t_4\}$ would need to be replayed. As the result of replay, A_1 may generate a revision (r_2) to o_2 that would then be input to A_2. Now consider the minimal replay sequence for A_2. Its input revision (r_2) affects the output, u_1 resulting from aggregating over the $\{o_1, o_2\}$ window. To generate

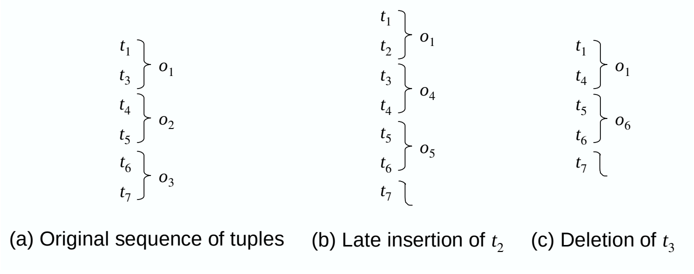


Figure 5: Window Realignment in Row-based Windows

a revision to u_1, A_2 requires A_1 to regenerate its outputs o_1 and o_2, and this in turn requires replay of tuples $< t_1, t_2, t_3, t_4 >$ (Figure 4b). But note that in processing this replay sequence, A_1 must generate not only a revision (for o_2), but a replay tuple (for o_1). In general, once the replay sequence is determined for the query network, every box processes its sequence of replay and revision tuples, and produces as output, revision tuples for every output that changes, and replay tuples for every output that remains the same.

A special consideration, however, has to be made to the replay sequence when the stream query contains a row-based window. The arrival order of tuples into an operator with a row-based window determines the sequence of tuples that are processed together. Consider the sequence of tuples in a row-based window of size 2 that tumbles in Figure 5a. Observe that t_2 is missing from this sequence. The output from these three windows are o_1, o_2, and o_3 respectively. Now, if a late insertion of t_2 is received, the windows need to be realigned so that we can process it as if it arrived on time. Figure 5b shows the what the new windows will be. To achieve this state, we must delete the old results o_2 and o_3 and insert the new results o_4 and o_5. Figure 5c shows what the windows will be if t_2 is deleted from the original sequence of tuples in Figure 5a. Both Figures 5b and 5c show that when tuples are deleted or inserted into row-based windows, all windows after the ones directly affected by the revision must be realigned. Thus, to deal with such situations, the replay query for a query network with row-based windows must be modified so that the replay sequence ends with the most recently processed input tuple. The start of the replay sequence is determined by the replay-based technique being used to process revisions.

3.2 Revision Processing Techniques

In this section, we present two possible approaches to replay-based revision processing: *Replay-All* (RA), and *Replay-Minimal* (RM). These two techniques differ by how each balances the trade-off of:

1. overhead of processing standard tuples, and
2. the cost of processing revisions.

RA minimizes the overhead of processing standard tuples but incurs high processing cost for revision, while on the other hand, RM reduces the processing cost per revision but at the expense of added overhead for processing standard tuples.

3.2.1 Replay-All (RA)

The simplistic RA revision processing approach responds to a revision by replaying the entire contents of the input stream archive. For example, given a revision r, the replay query generated in response to r is:

```
SELECT * FROM InputStreamArchive
ORDER BY Time ASC.
```

In other words, the replay sequence generated by RA for any revision is the set of all tuples in the stream archive in ascending order of the time when they were processed. This replay query can be improved somewhat if the computation over S involved grouping over some attribute, $\mathtt{g_{att}}$. In this case, the schema of the input stream archive can be extended to include this attribute, and the replay query becomes:

```
SELECT * FROM InputStreamArchive
WHERE g_att == 'X' ORDER BY Time ASC,
```

such that X is the value of $\mathtt{g_{att}}$ named in the revision. The replay query is the same as above even when the stream query contains a row-based window. Since all tuples in the input stream archive are replayed, the replay sequence includes tuples necessary to realign all windows following the ones affected by the revision.

Although this replay-based technique incurs no overhead to determine a replay sequence in response to a revision, it may return a very large replay sequence. Consider the example in Figure 4 and assume that tuples $t_1, \ldots, t_4$ are currently stored in the input stream archive of this query network. When a revision to t_3 is received, RA replay all tuples in the input stream archive in response to the revision, i.e., $T_{RS}(t_3) = < t_1, \ldots, t_4 >$.

It might appear that RA is a highly undesirable revision processing technique because it can lead to very expensive revision processing costs. On the other hand, because it adds the least possible overhead to the processing of standard tuples, RA may be an effective approach to revision processing if revisions are rare and if the size threshold for the input stream archives is small.

3.2.2 Replay-Minimal (RM)

Whereas RA incurs **no** overhead to produce a **very large** replay sequence in response to a revision, RM incurs more overhead to produce an **almost minimal** replay sequence in response to a revision. Beyond the input stream archive, RM also requires an *output stream archive* that maps every output tuple (o) to its lineage set of input tuples ($l(o)$) (as was illustrated in Figure 3). Given the global timestamps associated with every input tuple, the lineage set of o can be represented in the output stream archive by associating a range of timestamps for each input stream containing tuples from which o was derived. Therefore, the schema of the output stream archive for a query network with n input streams includes the attributes shown below in Table 2.

Attribute	Value
ID	identifier for the output tuple
$\mathtt{S_1}$	min Time of tuples from stream 1 in lineage set of o
$\mathtt{E_1}$	max Time of tuples from stream 1 in lineage set of o
...	...
$\mathtt{S_n}$	min Time of tuples from stream n in lineage set of o
$\mathtt{E_n}$	max Time of tuples from stream n in lineage set of o

Table 2: Output Stream Archive Schema for n Input Streams

Leaving aside for the moment how this archive gets populated, the replay query generated in response to a revision, r, that revises an input tuple i that arrived at `Time = t` on the k^{th} input stream involves a query to the output stream archive followed by a query to the input stream archive, as shown below:

```
SELECT MIN(S_1) AS S_1, MAX(E_1) AS E_1, ...,
       MIN(S_n) AS S_n, MAX(E_n) AS E_n
INTO Temp
FROM OutputStreamArchive
WHERE t BETWEEN S_k AND E_k

SELECT * FROM InputStreamArchive I, Temp T
WHERE
   (I.Str = 1 AND I.Time BETWEEN T.S_1 AND T.E_1)
   OR ...  OR
   (I.Str = n AND I.Time BETWEEN T.S_n AND T.E_n)
ORDER BY I.Time.[3]
```

The initial query on the output stream archive identifies, with the WHERE clause, all output tuples that include i in their lineage sets, and with the subsequent FROM clause, the input tuples on all streams required to regenerate all of these output tuples. The subsequent query uses the one-line result of the initial query (Temp) to return the sequence of input tuples identified.

In the RM technique, the emission of any standard tuple on any output stream automatically results in the addition of a row representing the lineage set of that tuple in the output stream archive. Every operator in Borealis was modified to compute lineage set for every tuple it produced, expressed (as with the output stream archive) as time intervals over the streams containing data from which the tuple was derived. All input tuples are initialized to have lineage sets containing only themselves. **Filter** and **Map** operators pass on the lineage sets of input tuples to their corresponding output tuples. **Aggregate** and **Join** pass on the union of lineage sets of all input tuples to the output tuples they produce. All lineage set information is carried by each tuple using the time interval technique described for output stream archives.

In the example in Figure 4, the lineage sets for input tuples are initialized to include only themselves, i.e., $l(t_1) = \{1, 1\}$ and so on. When o_1 is produced from the window calculation consisting of t_1 and t_2, its lineage set consists of the union of the lineage sets for t_1 and t_2, i.e., $l(o_1) = \{1, 2\}$. Tuple o_1 carries this lineage set information as it is output by A_1. Similarly, for o_2, $l(o_2) = \{3, 4\}$. Now, the lineage set for u_1 consists of the union of the lineage sets for o_1 and o_2, i.e., $l(u_1) = \{1, 2, 3, 4\}$, since u_1 is produced from the window calculation consisting of o_1 and o_2. The output stream archive extracts this lineage set information from tuple u_1 and stores the smallest and largest values as described in Table 2. When t_3 is revised, the lineage set information stored in the output stream archive identifies u_1 as the query result that included t_3 and causes the input stream archive to replay the following tuples: $T_{RS}(t_3) = T_{MRS}(t_3) = <t_1, \ldots, t_4>$.

It should be noted that because time intervals are used to compactly represent lineage sets, the RM technique is **not** guaranteed to return a minimal replay sequence. This is because a given output tuple can be derived from non-consecutive tuples on a stream, in which case the tuples that fall between will also be included in the replay sequence.

4. EXPERIMENTS

In this section, we compare the performance of the replay-based techniques presented in this paper over workloads with a variety of mixes of revisions and standard tuples and with stream queries of varying complexity. All of our experiments were performed on a 2.40 GHz Pentium 4 machine with 1 GB of memory running Linux and using Postgres (8.2) table for stream archives. The application scenario for our experiments were derived from financial services, with the input stream ((Time, Symbol, Price)) a stock quote feed arriving in ascending order of Time. For the experiments in Section 4.1, we padded the tuples with an extra field so that each tuple was about 100 bytes wide. The input stream assumes 200 different stocks with each stock reporting a price every second.

Every experiment was run assuming a day's worth of data in the stream archive (17.25M rows or 2.16 GB) to start, and 83 minutes (1M rows or 128 MB) of data streaming into the system thereafter.

Reflecting the frequencies with which revisions occur in financial data feeds, we generated 3 different streams varying according to their revision frequencies:

- HIGH denotes an input stream with high frequencies of revisions (1 per 100,000 standard tuples),
- MOD denotes an input stream with moderate frequencies of revisions (1 per 500,000 standard tuples), and
- LOW denotes an input stream with low frequencies of revisions (1 per 1,000,000 standard tuples)

All revisions are generated randomly in random positions in the stream, and change a price for some stock whose original value had appeared in the previous 23 hours[4]. The three streams were generated a priori, so that every experiment was run with the exact same streams. Tuples are inserted into the input stream archive in batches that are constrained in size by some parameter, k. Thus, input stream data is buffered until either k standard tuples arrive, or fewer than k standard tuples arrive followed by 1 revision, the latter case ensuring that that the replay query that follows sees all data prior to the revision. To ensure no caching effects, each run was preceded by flushing the database buffer (by scanning a table with 2M rows (256 MB)), and the I/O buffer (by reading a 1.27 GB file).

For each experiment we report the following: STP (standard tuple processing time), RTP (revision tuple processing time) and COM (completion time). STP is the total time spent processing *standard tuples* from the input stream including index maintenance for stream archive representations that maintain an index. RTP is the total time spent processing *revisions* from the input stream, which includes deleting "aged-out" data (i.e., data more than 24 hours old) from the stream archive, issuing a replay query and reprocessing the query returns.[5] COM is the sum of STP and RTP.

Before evaluating the replay-based techniques, we first compare possible representations for the input and output stream archives. Note that while input stream archives are required for both replay-based techniques, output stream archives are also required for the RM technique. We compare indexed and unindexed representations of stream archives with one or more tables in Section 4.1. The conclusions drawn from this set of experiments will be used to guide the stream archive representation for the experiment that compares the performance of the replay-based techniques in Section 4.2. Finally, we look at scalability issues of our replay-based revision processing system in Section 4.3.

[3]As with RA, this query can be further qualified with a selection on the grouping attribute of the stream query if one exists.

[4]Because a replay set can contain data that arrived prior to the data revised by the revision, we constrained the revision to refer back to at least an hour after the oldest data in the stream archive.)

[5]In our technical report [13], we discuss sensitivity analysis experiments that show that best COM times are achieved when aged-out data is deleted from the archive only when necessary (i.e., just prior to issuing a replay query, and thus deletion from the archive is accounted for as part of revision processing time). In the interest of space, we omit the discussion here.

4.1 Input Stream Archives

An input stream archive differs from a standard OLTP table in a number of respects. Firstly, the update pattern for an input stream archive is consistent: new tuples are appended to the end and old tuples are deleted from the beginning to maintain a constant size (much like a very large window). Stream archives are also naturally clustered on the order field of the input stream (usually `Time`), are subject to high insertion rates (equivalent to stream arrival rates), and only need to support range queries over the clustering attribute, `Time` (i.e., replay queries). Thus, it is worth considering alternative stream archive representations besides a single indexed or unindexed table.

4.1.1 *Stream Archive Representation*

In this section, we compare the performance (STP, RTP, COM) of different input stream archive representations. The RTP times reported in this experiment do **not** include the cost of reprocessing tuples returned by replay queries, as this cost is dependent upon the revision processing technique used. The purpose of this experiment is to choose a storage representation for the input stream archive that will be used for the experiments of Section 4.2 where the replay-based techniques will be compared. These experiments examine whether the stream archive should consist of a single table, or should be horizontally partitioned into multiple tables (*buckets*) each containing no more than k tuples for some value of k. We also examine the costs and benefits of indexing both single table and bucket-based representations. Thus, the representations examined are:

1. a single unindexed table (UT)
2. a single table with a clustered indexed on Time (IT)
3. multiple unindexed "buckets" of size $<= k$ tuples (B_k), and
4. multiple "buckets" of size $<= k$ tuples (B_k) with clustered indexes on `Symbol+Time` on each bucket (IB_k)

For both bucketing techniques (B_k and IB_k), STP involves inserting each batch of tuples into a new (or recycled) table, and then updating a separate *bucket index table* (BIT) which maintains a record for every bucket indicating the range of values of `Time` found in tuples contained in the bucket. To answer a replay query, both bucketing techniques first examine the BIT to limit the subsequent search to the buckets where the data requested by the query can be found.

Limitations in the current release of Postgres (8.2) make it impossible to maintain the UT and IT representations in the most natural and cost-effective ways. The proper way to respond to a revision tuple assuming the UT representation is to both delete aged-out tuples and return the results of a replay query in one pass of the input stream archive. While not expressible in SQL, this single-pass *delete-and-replay* revision processing implementation could be expressed as a cursor-based external procedure. However, Postgres identifies cursor-based deletion of tuples as a feature for a future release [14]. Hence, we report two numbers for UT. UT-1 reports an RTP time that is the sum of performing two queries over the stream archive (one that deletes aged-out tuples and one that answers replay queries). We also report a second number (UT-2) where we *predict* the cost of doing *delete-and-replay* by halving the RTP time reported for UT-1. (This is a rough estimate of the time to perform this operation in one pass instead of two.)

The stream archive representation, IT, maintains a clustered index on `Time` over a single table. Postgres supports clustered indexes by declaring all indexes to be secondary, but allowing tables to be reordered according to the index via the explicit command, `CLUSTER`. Examination of query plans before and after the `CLUSTER` command has been issued show that the effect of calling `CLUSTER` after every insertion is equivalent to maintaining a clustered index. However, because data inserted into the archive is already ordered on `Time`, `CLUSTER` sorts an already sorted table. But the Postgres optimizer won't exploit clustering unless this command has been issued since the most recent table update. To account for this unnecessary cost in our experiment results, our implementation of IT calls `CLUSTER` before issuing every replay query, but we discount the time Postgres uses to perform the operation in reporting the RTP and COM times. We should note that even with this favorable accounting procedure for techniques UT and IT, these techniques by far were outperformed by the bucketing-based techniques.

Figures 6a, 6b, and 6c show the STP, RTP and COM times for the input stream archive representations: UT-1, UT-2, IT, $\mathrm{B}_{50\mathrm{K}}$ and $\mathrm{IB}_{50\mathrm{K}}$[6]) assuming a replay query that returns 600 tuples (as would be required to reproduce an aggregate result with a window size of 10 minutes for a given stock). For all mixes of revisions and standard tuples examined, unindexed representations (UT-x and $\mathrm{B}_{50\mathrm{K}}$) require less time for STP than do their indexed counterparts (IT and $\mathrm{IB}_{50\mathrm{K}}$). This was expected as indexed representations must update their indexes as tuples are added to the stream archive. Also, for all mixes the unindexed single table representations (UT-x) require more time for RTP time than do the other representations, even accounting for the predicted one-pass *delete-and-replay* time (UT-2). This is because even streamlined revision processing requires a full pass over the entire stream archive to process a revision, whereas bucketing data allows both deletion and replay queries to target only the affected buckets.

Interestingly, the RTP cost of the indexed single table representation (IT) exceeds the RTP cost for unindexed buckets ($\mathrm{B}_{50\mathrm{K}}$). This is because the replay query costs are largely equivalent for the two representations, as the BIT acts like the index of IT in identifying buckets to search. The difference in RTP costs is due to the differing costs of deleting aged-out tuples from the archive prior to issuing the replay query. For the bucket representations, deletion requires identifying which buckets consist solely of aged-out tuples (a simple query on the BIT) and deleting the contents of these buckets with the inexpensive Postgres `TRUNCATE` command. For IT, deletion requires the additional cost of updating the index with each aged-out tuple that is removed.

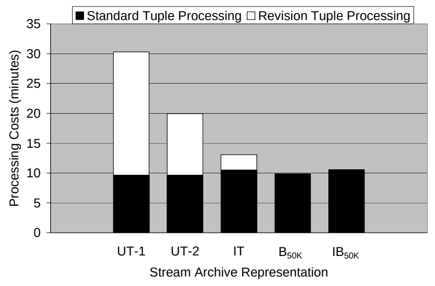
From these experiments, we can conclude that indexing over a stream archive incurs more cost than the benefit it provides. Because revisions tend to be relatively infrequent compared to standard tuples, the overhead of maintaining an index outweighs the benefit it provides for the replay queries that get issued. We also conclude that horizontally partitioning the moving window of tuples in the input stream archive into buckets is a good idea because it serves as a light-weight indexing technique that exploits the natural clustering of data arriving in ascending order of the index search key and reduces maintenance costs.

4.1.2 *Partition Sizing*

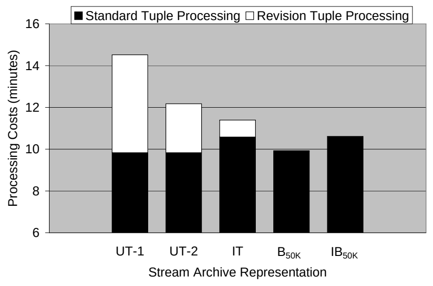
A natural question arises from the previous experiment as to whether bucketing techniques might perform better given buckets of sizes other than 50K. Smaller bucket sizes would lead to finer-grained `Time` ranges and potentially, reads of less unnecessary data. But larger bucket sizes would lead to a smaller BIT to search prior to examining the buckets.

For this experiment, we compared the STP, RTP and COM times for B_k and IB_k archive representations assuming values of k of

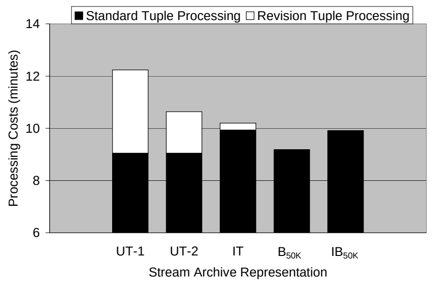
[6]We will experiment with different bucket sizes in Section 4.1.2.



(a) HIGH Revision Mix



(b) MOD Revision Mix



(c) LOW Revision Mix

Figure 6: Stream Archive Representation

50K, 100K, 250K and 500K. The results are shown for the three input streams (HIGH, MOD and LOW) in Figures 7a, 7b and 7c respectively.

As in the previous experiments where k was fixed to 50K, this experiment shows that regardless of the mix of revisions assumed of the input stream, the COM time for B_k is far less than that of IB_k for all values of k tried. While the indexed buckets have somewhat lower RTP times due to the use of indexes within buckets, unindexed buckets have far lower COM times given that these indexes do not need to be built. The smaller the bucket size, the more the COM cost of an index dominates the RTP benefits because of the limited query improvement provided by indexing over small tables.

From these experiments, we conclude that the input stream archive should contain bigger-sized partitions when there are very few revisions. Insertion of tuples into the input stream archive is the most expensive operation. When the size of the buckets are larger, the frequency of insertions is fewer, thus saving a lot of overhead. However, if revisions are more frequent, smaller-sized partitions have an advantage because fewer number of tuples in the appropriate buckets are scanned for replay tuples.

We have used the results of these experiments (and specifically, the COM time results) to determine the appropriate input stream archive representations in comparing the three revision processing techniques introduced in Section 3. For the HIGH input stream which has one revision for every 100K tuples in the stream, we chose the $\text{B}_{50\text{K}}$ representation. For the MOD and LOW input streams which have one revision for every 500K and 1M tuples respectively, we chose the $\text{B}_{500\text{K}}$ representation. Because it sees similar update patterns and queries to the input stream archive, runs that measure the performance of the RM technique assume equivalent unindexed bucket representations on the output stream archive as well.

4.2 Revision Processing Techniques

In this section, we compare the performance of the two replay-based revision processing techniques presented in Section 3.2, assuming queries of varying complexity and different mixes (HIGH, MOD and LOW) of revisions. As with the previous experiments, the measurements are taken for a stream of 1M tuples assuming that the input stream archives already contain a days worth of quote data (and in the case of the RM technique, the output stream archive contains the query results produced from the data in the input stream archive).

For this experiment, we compared the STP, RTP and COM times over HIGH, MOD and LOW concentrations of revisions for three stream queries:

1. Query Q_1 reports the 5-minute average stock price for each company every 5 minutes (using an **Aggregate** with a *Tumbling Window*),
2. Query Q_2 reports the 5-minute average stock price for each company every minute (using an **Aggregate** with a *Sliding Window*), and
3. Query Q_3 issues a "BUY" whenever the 1-minute average price of IBM exceeds the 5-minute average price of IBM, and a "SELL" command otherwise[7] (using a **Filter**, **Aggregate**'s with both Sliding and Tumbling windows and a **Join**.)

Stream	RA			RM		
	STP	RTP	COM	STP	RTP	COM
LOW	4.53	4.45	8.98	5.66	0.26	5.92
MOD	4.92	6.70	11.62	5.82	0.47	6.29
HIGH	5.80	20.77	26.57	6.03	0.44	6.47

(a) Query Q_1 (Minutes)

Stream	RA			RM		
	STP	RTP	COM	STP	RTP	COM
LOW	5.11	4.80	9.91	6.49	0.63	7.12
MOD	5.36	6.73	12.09	7.30	0.92	8.22
HIGH	6.72	20.99	27.70	8.36	0.89	9.26

(b) Query Q_2 (Minutes)

Stream	RA			RM		
	STP	RTP	COM	STP	RTP	COM
LOW	4.42	3.87	8.29	5.15	0.35	5.50
MOD	4.77	6.64	11.41	5.18	0.74	5.92
HIGH	5.04	18.55	23.59	5.37	0.71	6.08

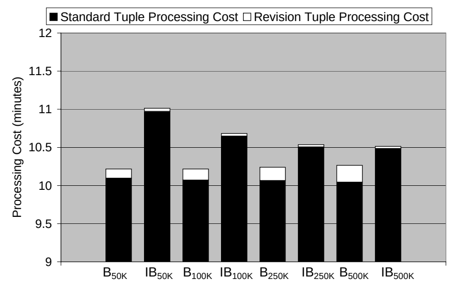
(c) Query Q_3 (Minutes)

Table 3: Revision Techniques Compared

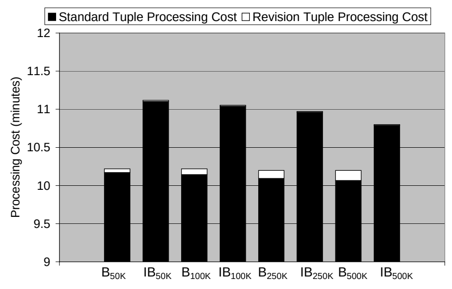
Tables 3a, 3b, and 3c show the performance of the three queries over HIGH, MOD and LOW mixes of revisions respectively. RA has lower STP costs than does RM for all queries and all input streams costs because whereas both techniques insert into the input stream archive as part of standard tuple processing, RM additionally determines the lineage set of each tuple in the query result and inserts this information into the output stream archive. On the other hand, the RTP time for RA is significantly higher because it replays all the tuples in the input stream archive whereas RM only replays the minimal replay sequence.

Observe that the RTP time for RM for the HIGH mix of revisions is lower than that of the MOD input stream. This is because the bucket size in the HIGH input stream is 50K as opposed to

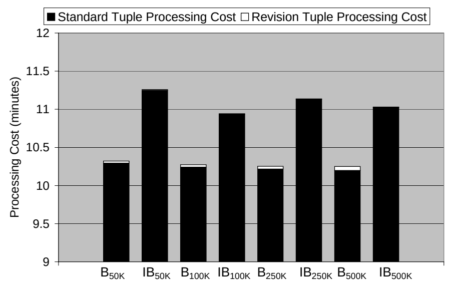
[7]This is a simplistic algorithmic trading technique that compares the long and short-term price trends of a stock to predict its future value.



(a) HIGH Revision Mix



(b) MOD Revision Mix



(c) LOW Revision Mix

Figure 7: Different Bucket Sizes of the Stream Archive

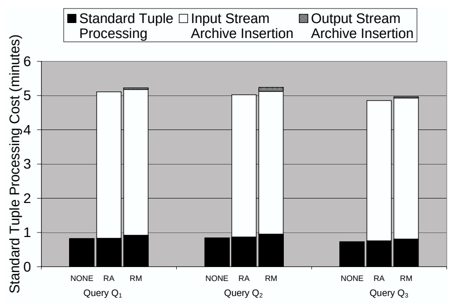


Figure 8: Overhead of Revision Processing

the 500K in the MOD input stream, which means that each replay query in MOD has to scan more data to retrieve the replay tuples. These experiments show that it is worth paying the overhead of maintaining the lineage of query results so that revisions are processed more efficiently.

4.3 Scalability and Current Research Directions

Support for replay-based revision processing incurs overhead in standard tuple processing regardless of the revision processing technique used. For RA, this overhead is the cost of maintaining the input stream archive. For RM, this overhead includes the cost of maintaining both the input and output stream archives. In this section we quantify these costs.

Figure 8 shows the overhead analysis for both revision processing techniques measured over all three stream queries considered in this paper. Assuming an input stream consisting of 1M standard tuples (and no revisions), we compare the STP time for a version of Borealis that has no support for revision processing to the STP time for versions of Borealis extended with RA and RM for revision processing, assuming B_{500K} input and output stream archive representations.

As the graphs show, the extension of Borealis to support replay-based revision processing techniques has increased the cost of processing a standard tuple by more than 500%! The graphs also reveal that the bottleneck of replay-based revision processing is the insertion of tuples into the input stream archive. For all queries and revision mixes examined, the insertion of tuples into the input stream archive accounts for approximately 80% of the reported STP times. Recall that the input stream archive is horizontally partitioned into buckets, each containing 500K tuples, without indexes and input tuples are inserted into the input stream archive in batches of 500K tuples. Thus, it is clear that the cost of maintaining an input stream archive in the presence of high volume of insertions dominates the overhead introduced by the implementation of replay-based revision processing.

These scalability concerns have motivated our current research direction, which is to determine how best to sample or summarize input stream data prior to storing it in an input stream archive, and how to modify the replay mechanism accordingly. The goal of this work is to counter the high overhead of insertions into the input stream archive by inserting less data, at the expense of producing *approximate* rather than *exact* revisions in response to an input revision. This work is attempting to find the balance between reducing the overhead of maintaining the input stream archive and making revision responses as accurate as possible.

Given that we want to reduce the amount of data stored by a certain percentage, we have developed some heuristics that determine which input tuples should be stored in an input stream archive. The simplest way to achieve this is to pick a sample of the input tuples to store. Although simple, this heuristic assumes that all tuples are equally important. A more complicated heuristic we are considering stores the tuples that have the biggest "impact" on the query results. For example, if a query is calculating the sum of a window of tuples, then the tuples with the largest values will have the most significant impact on the query result and thus should be stored. The idea behind this heuristic is that when a revision is later processed, the replay sequence will contain the tuples that contribute the most to the query results, thereby minimizing the error on the revised result. Another technique to reduce the amount of data stored in the input stream archives is to use summary-based techniques such as histograms. Input tuples are grouped into distinct sets and a histogram is created for each set. In response to a revision, the range of tuples that need to be replayed are determined and then the appropriate histograms are identified to recreate the tuples for the replay sequence. In all these heuristics, our main goal is to reduce the amount of data needed to be stored in the input stream archives.

5. RELATED WORK

A stream processing system with support for revisions is in a way similar to a temporal database [12, 17] as it allows changing the value for a key as of some time in the past (unlike an update in a traditional database which changes a value for a key as of "now"). However, there is a significant difference since after a value is changed in a temporal database, it means that both the original and revised values are valid but during different time intervals. In our work we consider revisions to be corrections, so as soon as a revision for a certain tuple is issued (by the source or a box), the new value is considered to be valid for all the time during which this tuple existed in the system.

We used a bucketing scheme to maintain our historical data. This is similar to a simple, one-level tree index. Wave-indices [16] suggests using an indexing scheme similar to our bucketing scheme when the number of updates are significantly higher than the number of queries. The data is partitioned and each partition is indexed separately. Rather than deleting entries in the index for data that have expired from the window, the index is dropped when all the data in that partition has expired. This is similar to what we do with our bucketing scheme, however, our buckets are not indexed. Through experiments we found that the cost of indexing each bucket is higher than scanning the contents of the bucket. If buckets are much larger, then an index may be beneficial. Using partitions to speed up B-Tree updates have also been suggested in [10]. Although these work focused on reducing the cost of maintaining a B-Tree index for high update rates, through experiments we found that in our case because the input data was ordered, simply partitioning the data achieved better results than when those partitions were indexed.

Querying historical data in streaming systems have been known to be a bottleneck [6, 15]. Some techniques to alleviate this bottleneck include storing a random sample of historical data and even aggregated values as suggested in [6]. Although this approximation is a great way to optimize the cost of retrieving and reprocessing this data, it can only produce approximate query results. Thus, we are looking to leverage this work as the basis for our ongoing work on producing approximate output revisions in response to revisions in the input.

6. CONCLUSIONS

The append-only model of stream processing engines implies that stream data is immutable and therefore correct. But in practice, streaming data sources often have errors. So important is error correction, that many of these sources issue revisions once the errors are detected. No other streaming data management solution we know of does anything with revisions other than ignore them.

In this paper, we introduced and compared two *replay-based* revision processing techniques. Replay-based techniques assume the existence of an input stream archive that keeps some history of the most recently input tuples to the system. These techniques then process a revision by querying this archive for the context in which the tuple being revised was originally processed, and reprocessing the entire sequence to determine how query results previously emitted have changed. This approach to revision processing is the only one that can be used to process revisions over *any* stream query.

After discussing the necessary requirements for replay-based revision processing, we described how the Borealis stream processing engine was extended to meet them. We then described two revision processing techniques that vary in how they approach the tradeoff of minimizing overhead when processing standard tuples vs avoiding replaying unnecessary data when processing revisions. We experimentally determined the best approach for implementing stream archives using Postgres, and then compared the two approaches on a workload that assumed storage of the last day's worth of stock quotes while streaming of an hour's worth of quotes. During this analysis, we discovered the huge overhead added by the introduction of replay-based revision processing. This scalability issue has motivated our current research on approximate revision processing.

7. 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] 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.

[5] S. Chandrasekaran, A. Deshpande, M. Franklin, J. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. Shah. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *CIDR Conference*, January 2003.

[6] S. Chandrasekaran and M. J. Franklin. Remembrance of Streams Past: Overload-Sensitive Management of Archived Streams. In *VLDB Conference*, pages 348–359, 2004.

[7] 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.

[8] 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.

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

[10] G. Graefe. B-tree Indexes for High Update Rates. *SIGMOD Record*, 35(1):39–44, 2006.

[11] 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.

[12] C. S. Jensen. *Temporal Database Management*. PhD thesis, Aalborg University, 2000.

[13] A. S. Maskey and M. Cherniack. Replay-Based Approaches to Revision Processing in Stream Query Engines. Technical report, Brandeis University, December 2007. URL: http://www.cs.brandeis.edu/%7Eanurag/revision-techreport-07.pdf.

[14] PostgreSQL Weekly News - June 17 2007, URL: http://people.planetpostgresql.org/dfetter/index.php?/archives/123-PostgreSQL-Weekly-News-June-17-2007.html.

[15] F. Reiss, K. Stockinger, K. Wu, A. Shoshani, and J. M. Hellerstein. Enabling Real-Time Querying of Live and Historical Stream Data. In *SSDBM*, 2007.

[16] N. Shivakumar and H. Garcia-Molina. Wave-Indices: Indexing Evolving Databases. In *ACM SIGMOD Conference*, pages 381–392, Tucson, AZ, May 1997.

[17] R. T. Snodgrass, editor. *The TSQL2 Temporal Query Language*. Kluwer Academic Publishers, 1995.

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

[19] J. Widom. Trio: A System for Integrated Management of Data, Accuracy, and Lineage. In *CIDR Conference*, pages 262–276, January 2005.