Time-Stamp Management and Query Execution in Data Stream Management Systems

Relational query languages can effectively express continuous queries on data streams after modest extensions. However, implementing such queries efficiently in data stream management systems (DSMSs) requires major changes in execution models and optimization techniques. In particular, finer-granularity execution models that are conducive to effective time-stamp management and response-time optimization must replace databases' relational algebra schemes. This article introduces such a model and uses it to solve the idle-waiting problems of data stream operators, such as union, joins, and aggregates over windows with slides.

Yijian Bai, Hetal Thakkar, and Carlo Zaniolo
University of California, Los Angeles

Haixun Wang
IBM T.J. Watson Research Center

Data stream management systems (DSMSs) must efficiently support continuous queries on massive and bursty data streams with real-time or almost-real-time responses. They therefore need flexible execution models that deal effectively with the temporal aspects of data stream operators, which consume and produce tuples ordered by their time stamps.

For instance, the *union* operator sorts and merges inputs according to tuple time stamps — that is, it compares the time stamps in its input buffers and moves the tuples with the least time stamp to the output. When any of its inputs is empty, the union operator can't proceed, because subsequent tuples arriving in the empty buffers could have smaller time stamps because of skew between streams. A naive implementation would then let the union operator enter an idle-waiting mode until tuples become available in all its input buffers. (We use "idle-waiting" instead of "blocking" to avoid confusion with nonmonotonic blocking operators that pose different challenges for data streams.[1,2]) Theodore Johnson and his colleagues address the idle-waiting-prone (IWP) operator problem using a punctuation-based approach[3] (see the "Related Work on Idle-Waiting-Prone Operators" sidebar). In their approach, the operator scheduler generates heartbeats at regular time intervals and injects them into the data streams as punctuation tuples, which are deliv-

 1089-7801/08/$25.00

ered to IWP operators down the path regardless of whether these operators are currently idle.[3]

The rate at which the scheduler must inject punctuation tuples is a difficult optimization decision that largely depends on the various streams' load conditions. Too few punctuation tuples might leave significant idle waiting unresolved in the system. However, too many punctuation tuples might result in extra overhead even when idle waiting doesn't occur. We therefore take an alternate approach. Our integrated techniques for time-stamp management and query execution reduce both memory usage and latency in queries with union and join operators, as well as single-stream IWP operators such as aggregates on windows with slides.[4] Our approach also significantly reduces high-output burstiness, a frequent result of idle waiting. Our experiments show that the improvements we obtained significantly surpass those obtainable with the periodic time-stamp approach.[3]

Basic Execution Model

Although the abstract semantics of the various operators (selection, projection, union, window join, and aggregates) on data streams is based on relational database semantics, their concrete semantics is quite different. We designed the execution techniques of the stream operators for main memory and data-stream-specific requirements related to ordering and time stamps. Such time-stamp-related requirements in particular directly affect the union, join, and aggregates on logical windows with slides operators, which are exposed to the idle-waiting problem.

Figure 1 shows an efficient implementation for executing IWP operators. Here, the union performs a simple sort-merge operation on multiple input streams based on time stamps. The symmetric window join maintains internal window buffers to convert unbounded data streams to bounded sequences, for which we use Jaewoo Kang and his colleagues' widely accepted semantics.[5] Finally, the aggregates on logical window with slides illustrate that unary operators can also experience idle waiting. Consider an aggregate sum being computed on a 10-minute (600-second) slide. If the current slide started at time t, and our clock shows a time greater than t + 600, we still can't output a result until a tuple with time stamp t + 600 arrives to guarantee that the slide doesn't miss any qualified tuples.

Whereas Johnson and colleagues use a regular-interval punctuation-based approach to solve the idle-waiting problem,[3] a better ap-

Union. When tuples are present at all inputs, select the one with the minimal time stamp and

- (production) add this tuple to the output, and
- (consumption) remove it from the input.

Aggregates on logical windows with slides. When a tuple is present at the input,

- (production) if this input tuple's time stamp is greater than the slide's expiration time, compute the aggregate on its internal window buffer and add the result to the output; and
- (consumption) remove the input tuple and add it to the internal window buffer on which the aggregate is computed (from which you also remove the expired tuples).

Window join of streams A and B. When tuples are present at both inputs, and A's time stamp is less than or equal to B's, perform the following operations (if B's time stamp is less than or equal to A's, perform symmetric operations):

- (production) compute the join of the tuple in A with the tuples in B's window buffer — W(B) — and add the resulting tuples to the output buffer (these tuples take their time stamps from the tuple in A); and
- (consumption) remove the current tuple in A from the input and add it to W(A) (from which you also remove the expired tuples).

Figure 1. Basic execution of query operators. In this implementation, the idle-waiting-prone (IWP) operators (union, window join, and aggregates on logical windows with slides) are still subject to the idle-waiting problem.

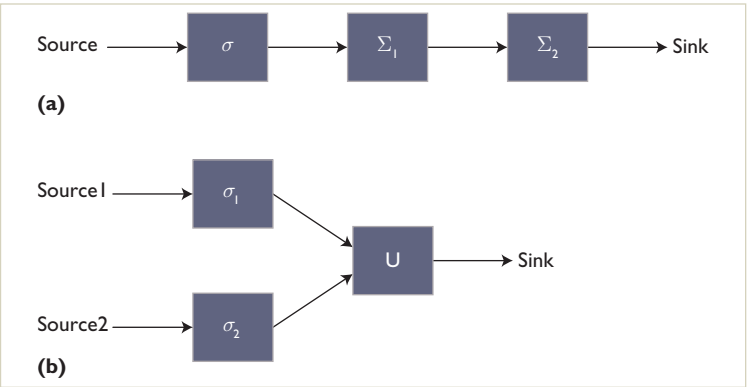

Figure 2. Query graphs. (a) A simple path query consists of source and sink nodes and query operators (σ, Σ_1, and Σ_2) connected by buffers. (b) In a simple union graph, a selection operator filters each input data stream before feeding tuples to the union operator.

proach is to generate critical time-stamp information on-demand using our simple execution model. We based our execution model on query graphs used widely in the DSMS literature, with the nodes denoting query operators and arcs denoting the buffers connecting them, such as those in Figure 2.

In the example in Figure 2a, the graph is a simple path. A directed arc from Q_i to Q_j represents a buffer, in which Q_i adds tuples to the buffer's tail (production) and Q_j takes tuples from the buffer's head (consumption). In addition to the actual query operators (σ, Σ_1, and Σ_2), the graph also contains source and sink nodes. The arcs leaving the source nodes represent input buffers. In our Stream Mill DSMS (as well as many others), external wrappers fill these buffers. When the input buffers are empty, the source nodes have three options. They can

- wait until some tuple arrives in the buffer,
- return control to the DSMS scheduler (which will then attend to other tasks), or
- generate time-stamp information and propagate it through the operator graph.

The arcs leading to the sink nodes denote the output buffers from which output wrappers take the tuples to be sent to users or to other processes.

A query graph can have several strongly connected components, in which each component is a directed acyclic graph. Each DAG represents a scheduling unit that's assigned a share of the system resources by the DSMS scheduler/optimizer (a problem outside this article's scope).

We execute each component using a two-step cycle to iterate through the operators in the component. In the execution step, we execute the current operator. Next, in the continuation step, we select the next operator for execution by the next-operator-selection (NOS) rules, which are defined according to the Boolean values of two state variables:

- *yield* is true if the current operator's output buffer contains some tuples, and
- *more* is true if the current operator's input buffer still contains tuples.

With minor modifications, this general two-step execution model is useful for specifying a wide range of execution strategies.

Depth-First Strategy

DFS is equivalent to a first-in-first-out strategy, but with backtracking upon detection of empty buffers (a necessary step in generating time stamps on-demand). DFS expedites tuple progress toward output by sending tuples to the next operator on the path as soon as they're produced. To specify depth-first strategy, the next-operator-selection (NOS) rules are defined as follows:

```
Forward: if yield then next := succ
Encore: else if more then next := self
Backtrack: else next := pred
  Repeat this NOS step on next
```

Thus, after executing the first query operator σ in Figure 2a, the algorithm checks if σ's output buffer is empty. If not (that is, if *yield* is true), the algorithm executes the Σ_1 operator. It repeats the same steps to reach the Σ_2 operator, which is the last operator in the path before the sink node. Thus, an output wrapper will consume Σ_2's output tuples — a separate process in Stream Mill. The algorithm continues the consumption of all Σ_2's input tuples — that is, it ignores the forward condition when the current operator is the last before the sink node and directly executes the encore condition.

Once all of Σ_2's tuples are processed, the *more* variable becomes false and the algorithm backtracks to its predecessor — the Σ_1 operator. If it backtracks further to reach operator σ and its *more* condition becomes false, the algorithm should again backtrack to σ's predecessor. In this case, however, the predecessor is the source node, denoting that an external wrapper must fill σ's input buffer with new tuples. Until these new tuples arrive, nothing is left to do on this path. In this situation, we could return control to the query scheduler to let the DSMS attend to other tasks. Alternatively, we could propagate the time stamp from the source, as we discuss later.

Breadth-First Strategy

We can specify BFS by switching the order of the DFS strategy's forward and encore rules, selecting the next operator as follows:

```
Encore: if more then next := self
Forward: else if yield then next := succ
Backtrack: else next := pred
      Repeat this NOS step on next
```

Round Robin

As in BFS, for RR we finish every tuple on one operator's input before visiting the next operator. However, when input becomes empty and the operator produces no result tuples, we simply exit the component and return control to the scheduler:

```
Encore: if more then next := self
Forward: else if yield then next := succ
Exit: else exit
       Repeat this NOS step on next
```

Unions and Joins

Some query graphs, such as the graph in Figure 3, contain operators with multiple inputs.

For operators such as unions and joins, the *more* condition evaluates to true when all of their inputs contain tuples. When some of their input buffers contain no tuples, *more* evaluates to false, and both DFS and BFS backtrack to a predecessor operator. Naturally, the algorithm backtracks to a predecessor feeding into a buffer that's currently empty (if multiple inputs are empty, pick randomly). Therefore, if *more* is false because, say, the *j*th input for the current operator is empty, and pred_j is the operator feeding into that buffer, we modify the backtrack rule as follows:

```
Backtrack: next := pred_j
```

Except for these changes, the execution of operator graphs containing joins and unions is the same as that for graphs consisting of single-input operators.

The execution strategies discussed so far can suffer from idle waiting. We next discuss techniques for solving this problem.

Managing IWP Operators

We first discuss simultaneous tuples, which are tuples with the same time stamp. Such tuples are common in applications using coarse time-stamp values. Simultaneous tuples give rise to several issues. Consider, for example, a union operator with inputs A and B. If both A and B contain simultaneous tuples, an operator can process them and add them to the output. But the rules in Figure 1 only move one tuple at a time. Thus, the operator will empty either A or B first, leaving the other holding one or more simultaneous tuples. One possible fix is to change the rules in Figure 1, so we now move all tuples having the minimal time stamp through the union operator at once. However, tuples arriving in buffers A or B (with the same time stamp due to arrival skew) after the operator has processed the simultaneous tuples in those buffers will still incur idle waiting.

Our solution to the simultaneous tuples problem is to introduce a time-stamp memory (TSM) register for each of the IWP operator's inputs. We automatically update the TSM's value with the current input tuple's time-stamp value. This value remains in the register until the next tuple updates it.

We then replace the execution rules for the IWP operators with the rules in Figure 4 (we omit the aggregate operators in this figure because of space constraints). We also relax the *more* condition so it holds true for the query operator *Q* if there is at least one input tuple with time-stamp value τ, where τ is the minimal val-

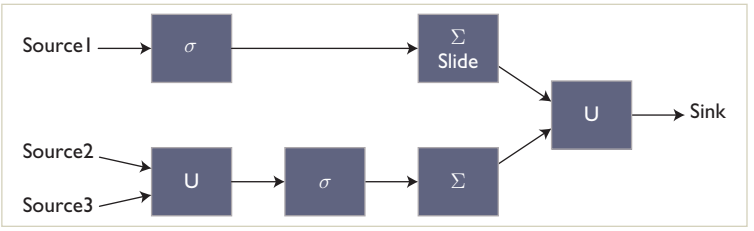

Figure 3. Query with nested union nodes. In this example, one multi-input operator leads to another multi-input operator (both unions here).

Union. If *more* is true, select an input tuple with time stamp τ and deliver it to the output (production); then remove it from the input (consumption).

Window join of stream A with stream B. If *more* is true, then

- if input A contains a data tuple with time-stamp value τ, perform the following operations (perform symmetric operations if B contains a data tuple with time-stamp value τ):
  - (production) join the tuple in A with the tuples in W(B) and send the result to the output, and
  - (consumption) remove the current input tuple in A from the input and add it to W(A) (from which you also remove the expired tuples).
- (production) if neither A nor B contain an input data tuple with time stamp τ, add a punctuation tuple with time stamp τ to the output.

Figure 4. Execution using time-stamp memory registers. Adding TSM registers and relaxing the more *condition solves the simultaneous tuples problem and reduces idle waiting.*

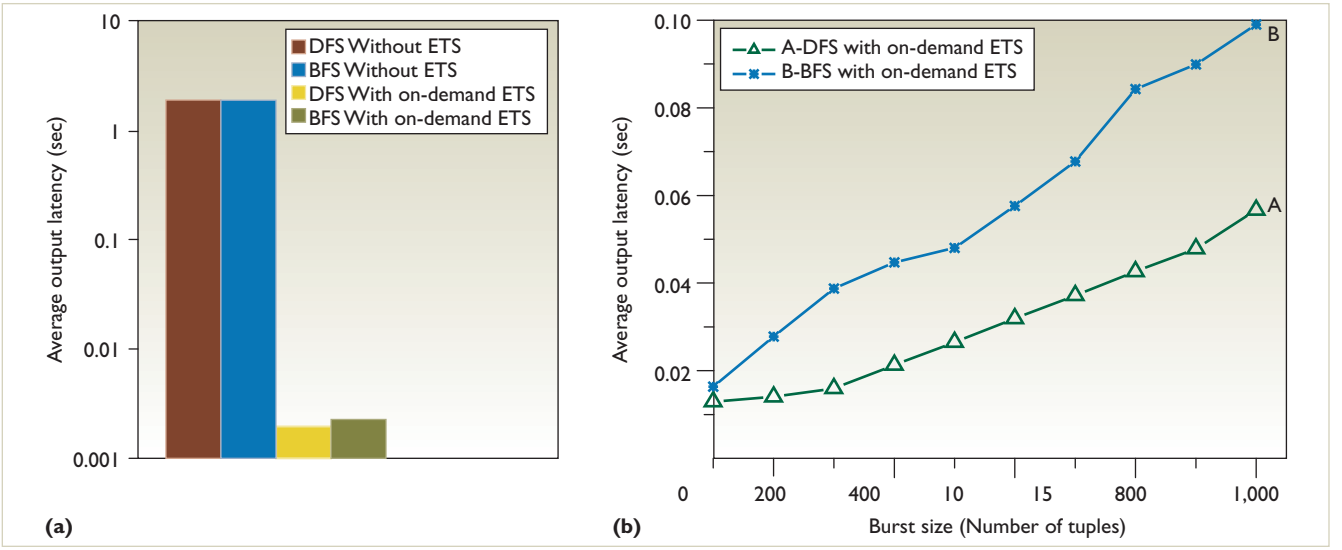

Figure 5. Latency for the breadth-first and depth-first strategies. The diagrams show (a) BFS and DFS latencies with and without enabling time-stamp (ETS) propagation, and (b) BFS and DFS performance for various burstiness factors.

ue in the input TSM registers of *Q*.

The TSM registers and relaxed *more* condition alleviate the simultaneous tuples problem and reduce idle waiting in IWP operators.

Propagating Time-Stamp Information

We seek an approach that generates enabling time stamps (ETSs) on-demand and sends them to idle-waiting operators so they can resume their activity. We use the term "ETS" to avoid the connotation of periodicity associated with heartbeats. Using our query-graph-based execution model, we can extend DFS (and BFS) to support on-demand generation of ETS information. Indeed, once the backtracking process takes us back to the source node, we can generate a new ETS value and send it as a punctuation tuple down the path on which backtracking just occurred to update TSM registers along the way and reactivate idle-waiting operators.

The extended rules for IWP operators shown in Figure 4 also apply to punctuation tuples. Using these rules, we no longer require tuples to be present in all of the union operator's input buffers. As long as a data tuple's time-stamp value is less than or equal to the other input buffers' TSM values, the operator will remove the tuple from the input buffer and move it to the output buffer (independent of whether it's a data tuple or a punctuation tuple).

If we compare Figure 4 to Figure 1, we see that the only change for the union operator is that it uses the relaxed *more* condition. We also start the window join computation by checking this condition. If the tuple is in fact a data tuple, we need no other modification. However if the tuple is a punctuation tuple, we move it from the input to the output. Furthermore, when we can't generate a data tuple, we simply produce a punctuation tuple for the benefit of the IWP operators down the path. Therefore, the latest time-stamp information is always carried by either the data tuples or the punctuation tuples to resolve idle waiting in the system.

We also modify non-IWP operators to let punctuation tuples go through unchanged (modulo reformatting required by the operators).

Generating Time Stamps

For data tuples with internal time stamps, we inject the system clock's current time into the data tuples as we add these tuples to our query graph's input buffer (the "source" in our diagrams). In the absence of data tuples, on-demand ETS generation for a data stream simply inserts a punctuation tuple with the time-stamp value from the system clock. For external time stamps, the application generating the data stream injects time stamps into the data tuples, whereas our DSMS generates the punctuation tuples' ETSs using the heartbeat mechanisms previously discussed elsewhere.[1,3] Our Stream Mill DSMS also allows tuples that are time stamped neither at the source nor when they enter the system. Operators that require time stamps, such as aggregates on logical windows, will generate *latent time stamps* at the time when the tuples arrive at the operator. Because tuples with latent time stamps

Related Work on Idle-Waiting-Prone Operators

Much previous work has sought to optimize latency and memory on massive data streams, but Theodore Johnson and his colleagues first discussed the idle-waiting problem caused by mismatched arrival rates of input streams in binary operators.[1] Their solution uses punctuation tuples, which have proven useful in dealing problems such as blocking operators,[2] data stream joins,[3] and out-of order tuples.[4]

A related approach is Aurora's time-out mechanism,[5] in which tuples are discarded if they wait for future tuple arrivals for longer than a specific time threshold. The time-out approach resolves the idle-waiting problem but sacrifices the correctness of query results. It might therefore be acceptable for the Aurora aggregate operators,[5] but much less so for operators such as union and joins. Yijian Bai and his colleagues first discussed the use of on-demand time stamps for idle-waiting-prone (IWP) operators,[6] but didn't cover unary operators, burstiness, and the results of other experiments we've discussed here. Others have discussed related techniques to support composite event semantics[7] and RFID applications.[8,9]

References

1. T. Johnson et al., "A Heartbeat Mechanism and its Application in Gigascope," *Proc. 31st Conf. Very Large Databases* (VLDB 05), VLDB Endowment, 2005, pp. 1079–1088.
2. P.A. Tucker et al., "Exploiting Punctuation Semantics in Continuous Data Streams," *IEEE Trans. Knowledge and Data Eng.* (TKDE), vol. 15, no. 3, 2003, pp. 555–568.
3. L. Ding and E.A. Rundensteiner, "Evaluating Window Joins Over Punctuated Streams," *Proc. 13th ACM Int'l Conf. Information and Knowledge Management*, (CIKM 04), ACM Press, 2004, pp. 98–107.
4. U. Srivastava and J. Widom. "Flexible Time Management in Data Stream Systems," *Proc. 23rd ACM SIGMOD-SIGACT-SIGART Symp. Principles of Database Systems*, (PODS 04), ACM Press, 2004, pp. 263–274.
5. H. Balakrishnan et al., "Retrospective on Aurora," *VLDB J.*, 2004. vol. 13, no. 4, pp. 370–383.
6. Y. Bai et al., "Optimizing Time Stamp Management in Data Stream Management Systems," *Proc. 23rd IEEE Int'l Conf. Data Eng.*, (ICDE 07), IEEE CS Press, 2007, pp. 1334–1338.
7. S. Chakravarthy and D. Mishra, "Snoop: An Expressive Event Specification Language for Active Databases," *Data Knowledge Eng.*, vol. 14, no. 1, 1994, pp. 1–26.
8. Y. Bai et al., "A Flexible Query Graph-Based Model for the Efficient Execution of Continuous Queries," *Proc. 23rd IEEE Int'l Conf. Data Eng.*, (ICDE 07), IEEE CS Press, 2007, pp. 634–643.
9. Y. Bai et al., "RFID Data Processing with a Data Stream Query Language," *Proc. 23rd IEEE Int'l Conf. Data Eng.*, (ICDE 07), IEEE CS Press, 2007, pp. 1184–1193.

never experience idle waiting, we use them as a baseline case in our performance study.

Experiments and Results

Our experiments used the Stream Mill DSMS server hosted on a Linux machine with a P4 2.8-GHz processor and 1 Gbyte of main memory. We randomly generated the input data tuples under a Poisson arrival process with the desired average arrival rates.

BFS Versus DFS

We compared the effectiveness of BFS and DFS with on-demand ETS propagation on bursty data streams using the simple query graph in Figure 2b. In this graph, a selection operator with low selectivity (95 percent of tuples pass through) filters each input data stream before the streams are unioned together. The data rates average 1,000 tuples per second on the first stream and 0.5 tuples per second on the second stream. This rate diversity can cause significant idle waiting for tuples on the faster stream.

Figure 5 quantifies this effect. Figure 5a shows that BFS and DFS have similar latencies without ETS propagation and that the latencies reduce dramatically with on-demand ETS (the y-axis is log scale). Figure 5b compares BFS and DFS performance for various burstiness factors. We keep the average rate of 1,000 tuples per second while introducing bursts of nearly simultaneous tuples. Both strategies show increasing average latency with increasing input burstiness, but BFS is affected more severely.

This steeper increase of latency with burst size is intrinsic in the BFS execution and has little do with ETS propagation policies – similar delay differences exist with latent time stamps. In fact, with latent time stamps, if we assume that the cost of processing one tuple is c for both the σ and union operators in Figure 2b, the average output latency for DFS is $(n + 1)c$, whereas for BFS it's $(1.5 + 0.5n)c$, where n is the size of bursts in the Source1 input.

Latency Reduction

Minimizing response time is a key query-optimization objective for most DSMSs. Figures 6a and 6b shows the (log-scale) results of our experiments using the query in Figure 2b. The average output latency drops regularly (line B) as we increase the frequency of the ETS punctuation tuples periodically injected into the sparser of the two data streams.

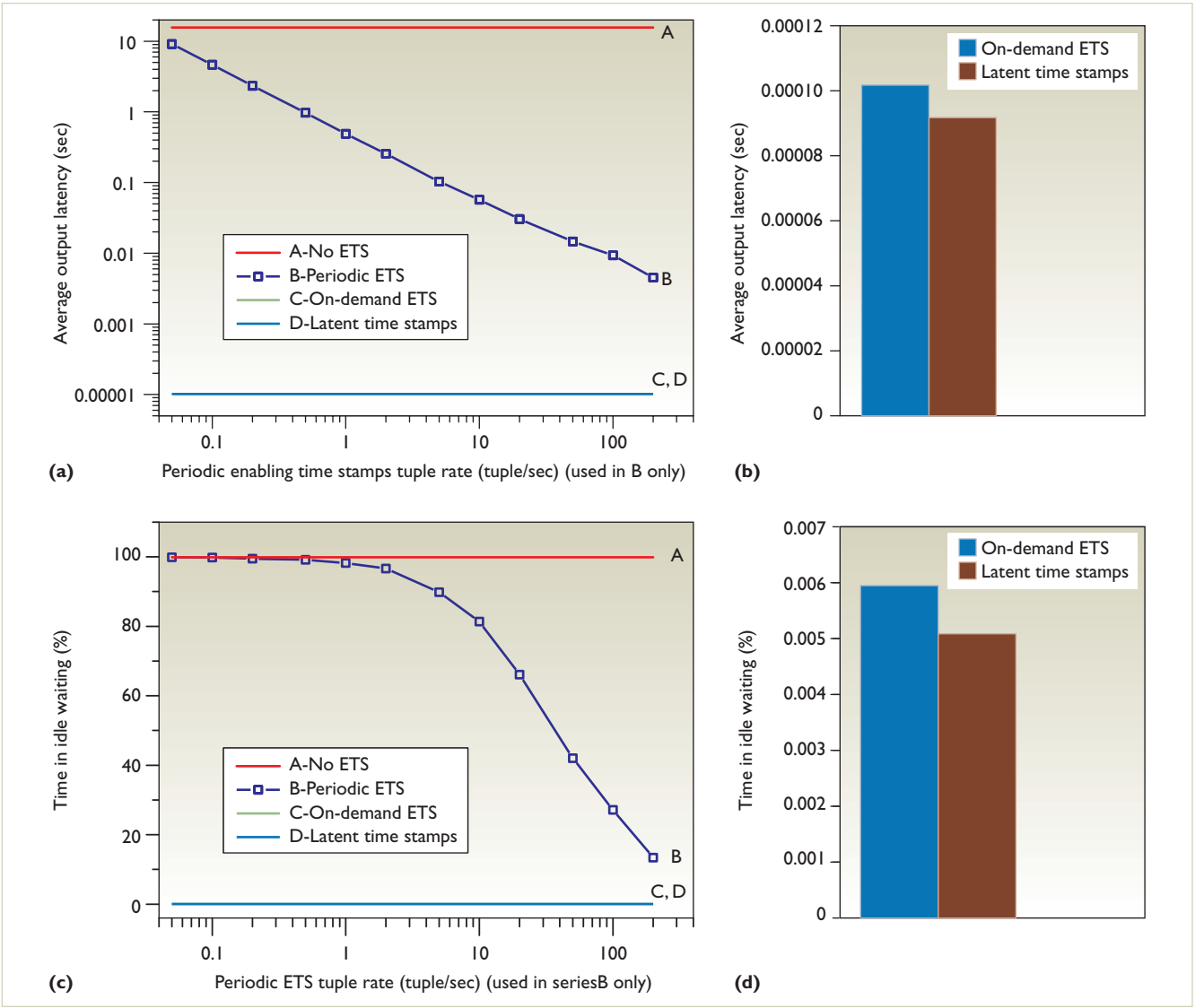

Figure 6. Latency reduction results. The average output latency for (a) no ETS propagation, regular punctuation tuples, ETS on-demand propagation, and latent time stamp; (b) ETS on-demand propagation and latent time stamp (on a finer time scale). The percent of time in waiting state for (c) the same four cases as in (a), and (d) ETS on-demand propagation and latent time stamp (on a finer scale).

Regardless of the frequency, periodic ETS can't match the performance of ETS on-demand (line C), which reduces the latency by several orders of magnitude with respect to A (where no ETS is used). More remarkably, line C comes close to optimal performance of streams with latent time stamps, under which no idle waiting occurs by definition (line D). Line C is so close to line D that the two are indistinguishable in Figure 6a. We therefore use Figure 6b to show their actual difference, which is about 0.1 milliseconds, or four orders of magnitude smaller than A.

We verify that idle waiting (rather than implementation problems or system overhead) caused the latency by measuring the percentage of time the union operator spends in an idle-waiting state — that is, the time in which one input data stream is empty while the other isn't. Line A in Figure 6c shows that when we don't use time stamp information, the system spends almost 99 percent of its time idle waiting, and tuples must wait a long time before processing. (Tuple processing takes a comparatively short time because we're well below the DSMS maximum processing capacity.) Line C shows that ETS on-demand reduces this waiting time to less than 0.1 percent of the total time. Line B shows that significantly reducing the waiting percent-

age requires high heartbeat rates, as much as 100 tuples per second. Even with the increased rate, however, we can't match the improvement obtained with on-demand ETS propagation. Again, the results obtained with on-demand ETS are so close to those of latent time stamps that their differences are only visible in Figure 6d with a much finer scale.

Memory Usage

ETS can significantly reduce memory usage. Figure 7 shows the peak buffer size, measured by the total number of tuples in the buffers, for the query in Figure 2b when the two streams have respective average rates of 50 and 0.05 tuples per second. Without ETS, line A in Figure 7 has a peak queue size of thousands of tuples, although the average input rate is only 50/0.05 tuples per second. Line C shows that on-demand ETS propagation reduces the memory usage by more than two orders of magnitude. For periodic ETS (line B) peak memory usage reduces initially with higher punctuation rates (as expected because idle waiting is reduced). However, high punctuation rates eventually increase peak memory requirements. This is because punctuation tuples produced at high rates tend to occupy memory when the scheduler is processing bursts of data tuples.

These results are consistent with Johnson and colleagues' results,[3] which showed that the maximum rate for periodic ETS punctuation was about one per second. At that low rate, the curve is still descending uniformly and the reversal seen in Figure 7 isn't present. Thus, periodic ETS punctuation can provide a reasonable solution in applications that don't require very low latency, and it might be the only viable solution in DSMSs whose architecture isn't conducive to on-demand ETS (such as Gigascope's two-level execution architecture[3]). However, it falls well below the versatility and performance obtainable with on-demand ETS.

Windows with Slides and Output Burstiness

Another undesirable side effect of idle waiting is that the output stream might become much burstier. For example, in Figure 3 an aggregate on a window with slide might hold up tuples in its sliding window. Without time-stamp propagation, tuples on the union operator's other input would accumulate and lead to bursty outputs, even though there might be data tuples in the slide aggregate that can unblock the idle-waiting union operator. In this case, the time-stamp information is accurate and available (carried by data tuples), but is delayed by the slide construct's existence. ETS propagation thus helps to deliver the time-stamp information to the union operator.

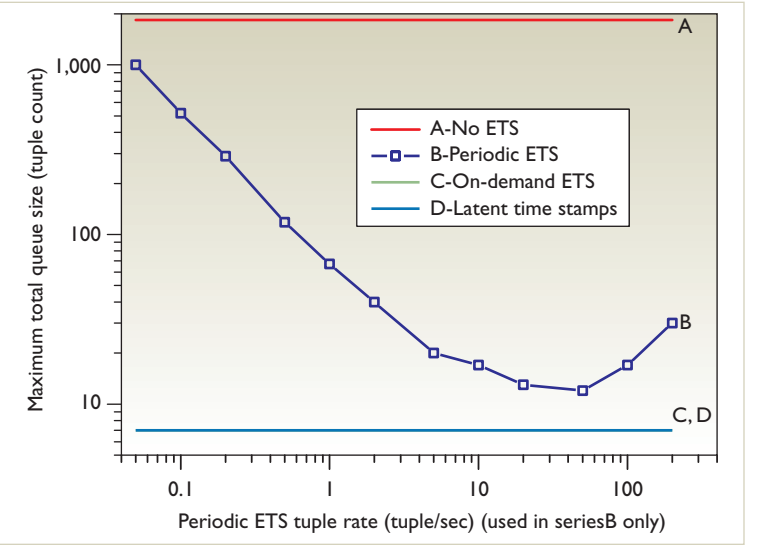

Figure 7. Peak total queue size. As line C shows, on-demand ETS propagation reduces the memory usage by more than two orders of magnitude in this experiment.

Figure 8 shows the burstiness of the output for the query in Figure 3. When we don't use ETS, burstiness increases significantly with the slide's size. (The slide determines the ratio between the number of tuples processed and the number of tuples produced in output.[6]) In our experiment, we counted the number of tuples produced during each 0.02-second period and defined burstiness as the difference between the extremes (maximum and minimum) divided by the average of those counts. Line B in Figure 8 clearly shows that on-demand ETS propagation eliminates the burstiness caused by the slide. The benefits of on-demand ETS therefore go beyond the ability to minimize latency and memory and include the ability to equalize traffic and smooth out traffic peaks caused by certain query operators' bursty arrivals or behavior. Reducing burstiness and its undesirable side effects is highly desirable in DSMSs.

Future work calls for extending our approach to models in which the semantics of data streams ordered by their time stamps is generalized to more advanced temporal models, such

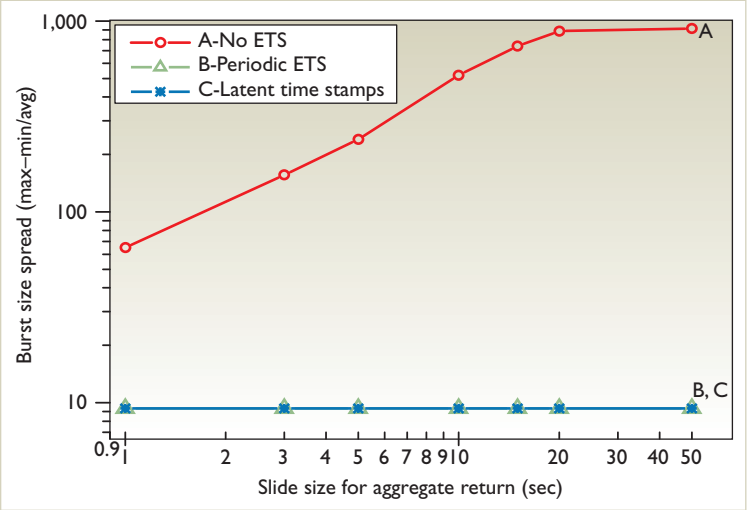

Figure 8. Burstiness versus slide size. As line B shows, on-demand ETS propagation eliminates the burstiness caused by the slide.

as those discussed elsewhere.[7] We must also extend our techniques to handle distributed stream processing, in which streams in the system can live on different processors and interact with each other through distributed protocols.

References

1. B. Babcock et al., "Models and Issues in Data Stream Systems," *Proc. 21st ACM Symp. Principles of Database Systems* (PODS 02), ACM Press, 2002, pp. 1–16.
2. Y.-N. Law, H. Wang, and C. Zaniolo, "Query Languages and Data Models for Database Sequences and Data Streams," *Proc. 30th Int'l Conf. Very Large Databases* (VLDB 04), VLDB Endowment, 2004, pp. 492–503.
3. T. Johnson et al., "A Heartbeat Mechanism and its Application in Gigascope," *Proc. 31st Conf. Very Large Databases* (VLDB 05), VLDB Endowment, 2005, pp. 1079–1088.
4. Y. Bai et al., "A Data Stream Language and System Designed for Power and Extensibility," *Proc. 15th ACM Int'l Conf. Information and Knowledge Management*, (CIKM 06), ACM Press, pp. 337–346.
5. J. Kang, J.F. Naughton, and S. Viglas, "Evaluating Window Joins over Unbounded Streams," *Proc. IEEE Int'l Conf. Data Eng.* (ICDE 03), IEEE CS Press, 2003, pp. 341–352.
6. J. Li et al., "Semantics and Evaluation Techniques for Window Aggregates in Data Streams," *Proc. 2005 ACM SIGMOD Int'l Conf. Management of Data*, (SIGMOD 05), ACM Press, 2005, pp. 311–322.
7. S. Roger et al., "Consistent Streaming Through Time: A Vision for Event Stream Processing," *Proc. 3rd Biennial Conf. Innovative Data Systems Research* (CIDR 07), pp. 363–374; www-db.cs.wisc.edu/cidr/cidr2007/papers/cidr07p42.pdf.

Yijian Bai is a software engineer at Google, primarily working on algorithms to improve ad-targeting systems. His research interests include data stream management systems, data stream mining, and Web data mining. Bai has a PhD in computer science from the University of California, Los Angeles. Contact him at ybai25@gmail.com.

Hetal Thakkar is a PhD candidate in computer science at the University of California, Los Angeles. His research interests include data stream mining systems and data stream languages. Thakkar has an MS in computer science from the University of California, Los Angeles. Contact him at hthakkar@cs.ucla.edu.

Haixun Wang is the a research scientist and technical assistant to the head of computer science at the IBM T.J. Watson Research Center. His research interests include database language and systems, data mining, and information retrieval. Wang has a PhD in computer science from the University of California, Los Angeles. He is a member of the IEEE and the ACM. Contact him at haixun@us.ibm.com.

Carlo Zaniolo is a professor of computer science at the University of California, Los Angeles. His research interests include data stream management systems, data mining and archival information systems. Zaniolo has a PhD in computer science from the University of California, Los Angeles. Contact him at zaniolo@cs.ucla.edu.