Optimizing Timestamp Management in Data Stream Management Systems

Yijian Bai
UCLA
bai@cs.ucla.edu

Hetal Thakkar
UCLA
hthakkar@cs.ucla.edu

Haixun Wang
IBM T. J. Watson
haixun@us.ibm.com

Carlo Zaniolo
UCLA
zaniolo@cs.ucla.edu

Abstract

It has long been recognized that multi-stream operators, such as union and join, often have to wait idly in a temporarily blocked state, as a result of skews between the timestamps of their input streams. Recently, it has been shown that the injection of heartbeat information through punctuation tuples can alleviate this problem. In this paper, we propose and investigate more effective solutions that use timestamps generated on-demand to reactivate idle-waiting operators. We thus introduce a simple execution model that efficiently supports on-demand punctuation. Experiments show that response time and memory usage are reduced substantially by this approach.

1 Introduction

Data stream management systems (DSMSs) normally rely on the property that data streams are ordered according to their timestamps. Therefore, DSMS query operators are designed to continuously consume input streams ordered by their timestamps and produce output streams also ordered by their timestamps. For instance, union is in fact a sort-merge operation that combines its input data streams into a single output stream where tuples are ordered by their timestamp values. Thus, when tuples are present at each input the union operator moves a tuple with the least timestamp to the output. However, when any of its inputs are empty the union operator cannot proceed, since future tuples on the empty inputs could have timestamps smaller than those of the current input tuples because of skews between different streams. Therefore, the traditional approach is to have the union operator enter an *idle-waiting* mode until tuples become available in all its input buffers. (In this paper, we use the term 'idle-waiting' instead of 'blocking' to prevent confusion with non-monotonic blocking operators[1] that pose quite different challenges for data streams [2, 11].)

The idle-waiting problem was recently studied in [9], where a punctuation-based approach was proposed to propagate heartbeats that can reactivate idle-waiting operators. Thus, in [9], heartbeats are generated at regular time intervals and injected into the data streams as punctuation tuples, which are delivered to union and join operators down the path, independent of whether these are idle-waiting or not. The rate at which punctuation tuples must be injected into the data streams represents a difficult optimization decision that largely depends on the load conditions of the various streams. For instance, say, that we want to compute the union of two data streams A and B, where B is experiencing heavier traffic than A. Then, the B tuples might have to wait a while for the A tuples—unless frequent punctuation tuples are injected into A. Too few punctuation tuples in A will result in many tuples of B experiencing significant idle-waiting; however, too many punctuation tuples in A will result in extra overhead to service punctuation tuples, which are unlikely to unblock any tuple in B. The best results can be expected when the frequency of tuples in A matches those in B—a goal that is very hard to achieve when the traffic is not stationary and if A or B are bursty.

Thus in this paper we propose an approach whereby timestamps are only generated and sent to the idle-waiting operators on-demand, rather than periodically. On-demand generation of punctuation tuples was considered but discarded in [9] because of the complexity it created for code generation and execution. Therefore, our first research challenge is that of devising a simple and efficient execution model that supports the generation and propagation of on-demand timestamps for idle-waiting operators. The robustness and practicality of our proposed solution is tested through its incorporation in the Stream Mill DSMS [1, 3]. Several experiments discussed in this paper confirm that the on-demand approach substantially improves the response time and memory usage. Throughout the paper, we will use the term Enabling Time-Stamps (ETSs) to describe timestamps that are generated on-demand and sent to idle-waiting operators to enable them to resume their activity; the term ETS avoids the connotation of periodicity that is associated with "Heartbeats".

The rest of this paper is organized as follows: the next section discusses general execution model for union and join operators, which face the idle-waiting problem. Section 3 introduces an execution model that supports the generation of critical timestamps on demand. Section 4 presents mechanisms for the propagation of ETS through the operator network, which effectively reduces idle-waiting. In Section 5 we discuss the different kinds of timestamps supported in Stream Mill and on-demand generation of ETS. In Section 6 we present the results of our experiments that study the effectiveness of different approaches under different load conditions.

[1]This distinction is desirable, since operators, such as union and join, behave quite differently from blocking aggregates, and other non-monotonic operators such as difference.

Union. When tuples are present in all inputs, select one with minimal timestamp and

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

Window Join of Stream A and Stream B. When tuples are present at both inputs, and the timestamp of A is $\leq$ than that of B then perform the following operations (symmetric operations are performed if timestamp of B is $\leq$ than that of A):

- (production) compute the join of the tuple in A with the tuples in W(B) and add the resulting tuples to output buffer (these tuples take their timestamps from the tuple in A)
- (consumption) the current tuple in A is removed from the input and added to the window buffer W(A) (from which the expired tuples are also removed)

Figure 1. Basic Execution of Query Operators

2 Basic Operators

As discussed in [9] the union and join operators are Idle-Waiting Prone (IWP) operators. The basic execution steps for these IWP operators are summarized in Figure 1[2].

We will use the widely accepted semantics proposed in [10] for symmetric window-join as shown Figure 1. Observe that when input tuples in A and B share the same timestamp (simultaneous tuples) we can nondeterministically process one before the other and similar consideration holds for the union operator (we will return to simultaneous tuples in Section 4.1). For simplicity of discussion we omit here the discussion of multi-way joins and asymmetric joins, whose treatment is however similar to that of binary joins.

The execution of non-IWP operators is straightforward: compute the result(s) and add that to the output (production)—with timestamp equal to that of the input tuple—and remove the tuple from the input (consumption).

3 Query Graphs and the Execution Model

A simple technique for avoiding the idle-waiting problems of IWP operators is to propagate heartbeat information by punctuation tuples generated at *regular intervals* [9]. A better approach consists in generating critical timestamp information as needed, on-demand. Although such a policy was considered too complex in [9], in this paper, we introduce an execution model that makes it simple and efficient. Our model is based on query operator graphs, that have been widely used in DSMSs to describe the scheduling of continuous queries. Figure 2 shows an example where the graph is a simple path. In general, the nodes of the graph denote query operators and the arcs denote the buffers connecting them. Therefore, our directed arc from Q_i to Q_j *represents a buffer*, whereby Q_i adds tuples to the tail of the buffer (production) and Q_j takes tuples from the front of the buffer (consumption). In addition to the actual query operators, the graph also contains *source* nodes and *sink* nodes as shown in Figure 2. The arcs leaving the source nodes represent input buffers. In Stream Mill (and many other DSMSs) these are being filled by external wrappers. If an input buffer is found empty during execution, we might (i) wait until some tuple arrives in the buffer, (ii) return control to the DSMS scheduler (that will then attend to other tasks), or (iii) generate punctuation tuples or other form of ETSs and send them to idle-waiting operators in the query graph. Likewise, 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[3].

[2]There are also other IWP operators, which we omit in this paper due to space limitations.

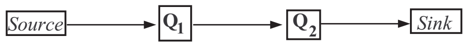


Figure 2. Simple Path Query

In general, a DSMS query graph can have several connected components, where each component is a DAG. Each such DAG represents a scheduling unit that is assigned a share of the system resources by the DSMS scheduler/optimizer (not discussed in this paper).

The execution of each component takes place using the two-step cycle shown in Figure 3. We first execute the current operator and then select the next operator according to the execution strategy being implemented. The choice made by the execution strategy is based on the boolean values of two state variables: $\mathbf{yield}$, and $\mathbf{more}$. The variable $\mathbf{yield}$ is set to true if the output buffer of the current operator contains some tuples (typically, just produced by the operator); $\mathbf{more}$ is true if there are still tuples in the input buffer of the current operator.

1. **[Execution Step]** Execute the current operator, and
2. **[Continuation Step]** Select the $\mathbf{next}$ operator for execution according to the conditions $\mathbf{yield}$ and $\mathbf{more}$ that, respectively, denote the presence of output and input tuples for the current operator.

Figure 3. The Basic Execution cycle forever iterates over these two steps.

3.1 Execution Strategy

In this paper, we consider the depth-first strategy (DFS), which is basically equivalent to a first-in-first-out strategy: to expedite tuple progress toward output, i.e. the tuples are sent to the next operator down the path as soon as they are produced. Thus, DFS strategy is implemented with following three Next Operator Selection Rules (NOS): *Forward*, *Encore*, and *Backtrack*.

Next Operator Selection (NOS): Depth-First Rules.

Forward: if $\mathbf{yield}$ then $\mathbf{next} := \mathbf{succ}$

Encore: else if $\mathbf{more}$ then $\mathbf{next} := \mathbf{self}$

[3]*sink* nodes should also eliminate punctuation tuples since they are only needed internally.

Backtrack: else $\mathbf{next} := \mathbf{pred}$ and repeat this NOS step on $\mathbf{pred}$.

Thus, after executing the query operator Q_1 in Figure 2, the algorithm checks if Q_1's output buffer is empty. If that is not the case[4], $\mathbf{yield}$ is true, then we execute the Q_2 operator. The Q_2 operator is the last operator in the path before the sink node: thus the tuples produced by Q_2 will actually be consumed by an output wrapper—a separate process in Stream Mill. Therefore, the algorithm continues with the consumption of all the input tuples of Q_2—i.e., the *Forward* condition is ignored when the current operator is the last before *Sink*, and instead we directly execute the *Encore* condition.

Once all the input tuples of Q_2 are processed, the $\mathbf{more}$ variable becomes false and the algorithm backtracks to its predecessor, i.e., the Q_1 operator, and executes the NOS rules on this operator. When operator Q_1's $\mathbf{more}$ condition becomes false, the algorithm backtracks to its predecessor: however in this case, the predecessor is the *Source* node, denoting that an external wrapper is responsible for filling the input buffer of Q_1 with new tuples. Until these new tuples arrive, there is nothing left to do on this path. In this situation, control could be returned to the query-scheduler to allow the DSMS to attend to other tasks.

So far, we have studied simple path queries, we now extend our approach to query graphs containing operators with multiple inputs, such as the union in Figure 4.

3.2 Unions and Joins

For multiple input operators, such as unions and joins, the $\mathbf{more}$ condition evaluates to true when tuples are present in all their inputs. Thus, when some of their input buffers are empty, **more** evaluates to false, and DFS backtracks to a predecessor operator. Naturally, we backtrack to a predecessor feeding into a buffer that is currently empty. Therefore, if **more** is false because, say, the j^{th} input for the current operator is empty and $\mathbf{pred}_j$ is the operator feeding into that buffer, then the *backtrack* rule will be modified as follows:

Backtrack: $\mathbf{next} := \mathbf{pred}_j$ and repeat this NOS rule on $\mathbf{pred}_j$.

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

However, the basic execution strategy just discussed can lead to idle-waiting. Idle-waiting is highly undesirable because it increases both the delay with which tuples are delivered to the output and the memory usage. Thus, in the rest of the paper, we study techniques for solving these problems.

4 Activating IWP Operators

Using the query-graph based execution model introduced in the previous section, DFS can be extended to support on-demand generation of ETS information. Indeed, once the backtracking process takes us all the way back to the source node, we can generate a new ETS value and send it down along the path on which backtracking just occurred, to reactivate idle-waiting IWP operators. The generation of these ETSs depends on the type of timestamp involved and will be discussed in the Section 5. The ETSs are propagated through punctuation tuples. But before that, we must address the issue of simultaneous tuples, which have not been discussed in previous papers, in spite of their obvious importance in the idle-waiting problem.

[4]Tuples could have been produced by the last execution of $\overline{Q_1}$, or by previous executions. In the first case, we can omit checking $\mathbf{yield}$ and go to $\mathbf{succ}$ directly.

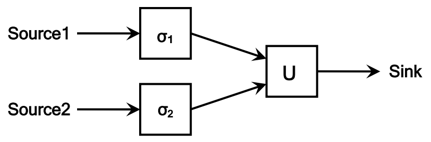


Figure 4. Simple Union

4.1 Simultaneous Tuples

Simultaneous tuples are tuples that have the same timestamp. These are common in applications where coarse timestamp values are used. To illustrate the issues arising from simultaneous tuples, let us consider a union operator with inputs A and B. If both A and B contain simultaneous tuples, these can all be processed and added to the output. But the current rules in Figure 1 fail to do that since they only move one tuple at a time: thus, either A or B will be emptied first and the other will be left holding one or more simultaneous tuples. One possible fix is to change the rules in Figure 1, whereby ALL the tuples having minimal timestamp are now added to the union operator at once. However, this does not completely solve the problem, since the simultaneous tuples that arrive in buffers A or B, after all their previously-arrived simultaneous tuples have been processed, will incur idle-waiting. To deal with simultaneous tuples, we instead make the following improvements to IWP operators:

- A Time-Stamp Memory ($\mathcal{TSM}$) register is introduced for each input of the IWP operator. The value of the $\mathcal{TSM}$ register is automatically updated with the timestamp value of the current input tuple and it remains in the register until the next tuple updates it.

Then the execution rules for the IWP operators are replaced with those in Figure 6. The **more** condition must also be modified as follows:

$\mathbf{more}$ holds true for the query operator $\overline{Q}$ if there is at least one input tuple with timestamp value τ, where τ is the minimal value in the input $\mathcal{TSM}$ registers of $\overline{Q}$.

Figure 5. A relaxed more condition

The $\mathcal{TSM}$ registers and the relaxed $\mathbf{more}$ condition alleviate the simultaneous tuples problem and reduce idle-waiting in IWP operators as discussed next.

4.2 Propagation of Punctuation Tuples

We use punctuation tuples to deliver ETS information to activate IWP operators, which can either be generated periodically, as discussed in [9], or on-demand as discussed in Section 5. Independent of the way ETS are generated, both the IWP operators and the non-IWP operators must be revised to assure their propagation.

Figure 6 shows the execution rules for IWP operators extended for both simultaneous tuples and punctuation tuples. These rules take advantage of $\mathcal{TSM}$ registers, and the relaxed **more** condition of Figure 5. These rules consider the minimal value τ in the $\mathcal{TSM}$ registers of the operator.

Union. If $\mathbf{more}$ is true select an input tuple with timestamp τ and deliver it to the output (production); then remove it from the input (consumption).

Window Join of Stream A with Stream B. If $\mathbf{more}$ is true, then:

- If input A contains a data tuple with timestamp value τ then perform the following operations (and the symmetric operations are performed if B contained a data tuple with timestamp value τ):
  - (production) join of the tuple in A with the tuples in W(B) and send the result of the join to the output
  - (consumption) current input tuple in A is removed from the input and added to W(A).
- (production) If neither A nor B contain an input data tuple with timestamp τ, add to the output a punctuation tuple with timestamp τ.

Figure 6. Execution using $\mathcal{TSM}$ Registers.

Comparing Figure 6 with Figure 1, we see that the only change for the union operator is that, rather than checking that all inputs are present, we now use the relaxed $\mathbf{more}$ condition of Figure 5. We also start the computation of window joins by checking this condition and if the tuple is in fact a data tuple no other modification is needed. However if this is instead a punctuation tuple, then we must remove it from the input and add it to the output. Furthermore, when we cannot generate a data tuple, we simply produce a punctuation tuple for the benefit of the IWP operators down the path.

Non-IWP operators must also be modified to deal with punctuation tuples, so that they let the punctuation tuples go through unchanged except for possible reformatting required by the specific operator.

5 Timestamp Generation

Flexible and robust mechanisms for timestamps and heartbeats are needed to achieve power and versatility in DSMSs [2, 12, 9]. Thus, the Stream Mill system supports three kinds of timestamps: *external*, *internal*, and *latent*:

1. Tuples are timestamped by the application that generated them (*External Timestamps*)
2. Tuples are timestamped when entering the DSMS using system time (*Internal Timestamps*)
3. Tuples without internal or external timestamps are timestamped on-the-fly by individual query operators that require timestamps (*latent timestamps*).

There is no idle-waiting when tuples with latent timestamps go through IWP operators: for instance for union, tuples can be added to the output as soon as they arrive, without any check on their timestamps. Therefore, in our experiments, we will measure the effectiveness of ETS-based strategies by comparing them against data streams having latent timestamps. Thus, ETS will not be used on data streams with latent timestamps, but they are critical for data streams with internal or external timestamps. The generation of on-demand ETS is discussed next.

On-Demand Generation of ETS at Source Nodes: When execution backtracks to a source node, whose input buffer is empty, then the node generates an ETS as follows:

(i) for internally timestamped tuples the ETS value generated is that of the current system clock.

(ii) For externally timestamped tuples the ETS value produced is application-dependent. Several interesting techniques have been discussed in the literature [12, 9]. For instance, if the maximum skew between two arrivals is δ and time τ has passed since the last tuple arrived, which had timestamp t, then we might want to produce an ETS of $t+\ \tau$ - δ.

These tuples are then processed by the successor operators.

6 Experiments and Results

To test the effectiveness of different techniques for solving the idle-waiting problem, we compare following four scenarios:

A Internally timestamped data streams (no ETS)

B Internally timestamped data streams (periodic ETS)

C Internally timestamped streams (on-demand ETS)

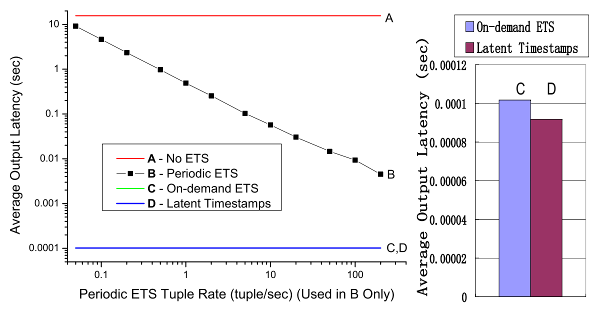
D Data streams with latent timestamps.

Experiment Setup For our experiments, the Stream Mill DSMS server was hosted on a Linux machine with P4 2.8GHz processor and 1 GB of main memory. The input data tuples were randomly generated under a Poisson arrival process with the desired average arrival rates.

We measure latency and memory consumption using the simple query graph of Figure 4, where each of the input data streams is filtered by a selection operator with low selectivity (95% tuples pass through), before the streams are unioned together. The data rates average at 50 tuples per second on the first stream and 0.05 tuples per second for the second stream; this rate diversity can cause significant idle-waiting for tuples on the faster stream.

Latency Reduction. Minimization of response time is a key query-optimization objective for most DSMSs. The results of our experiments are shown in Figure 7 (log-scale). The average output latency drops regularly (line B) as the frequency is increased for the ETS punctuation tuples periodically injected in the sparser of the two data stream.

Independent of their frequency, periodic ETS cannot match the performance of ETS on-demand (line C), which reduces the latency by several orders of magnitude with respect to A (no ETS used). More remarkably, C comes very



(a) (b)

Figure 7. Average Output Latency

close the optimal performance of streams with latent timestamps (line D). Line C is so close to D that the two are indistinguishable in the scale of 7 (a), and we have to use Figure 7 (b) to show their actual difference, which is about 0.1 milliseconds—four orders of magnitude smaller than A.

To verify that the latency is caused by idle-waiting, we measured the percentage of time the union operator spends in an idle-waiting state. Indeed, 99% of the total time in case A was spent in idle-waiting. At punctuation speeds $\geq$ 100 tuples per second, in case B the waiting time was reduced to 15% of the total time. However, it could not match the on-demand ETS (case C), which reduced the waiting period to less than 0.1% of the total time.

Memory Usage. As discussed in [9], ETS can deliver significant benefits in terms of reducing memory usage. In Figure 8 we measure peak total buffer size, in terms of total number of tuples in the buffers, under the 50/0.05 per second tuple rate on the two input streams. Without ETS line A in Figure in 8 has a peak queue size of thousands 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 since idle-waiting is reduced). However, high punctuation rates eventually cause an increase in peak memory requirements. This is because punctuation tuples produced at high rates tend to occupy memory, when bursts of data tuples are being processed.

7 Conclusions

The optimization of continuous queries and their query graphs has provided a major topic of DSMS research, which has primarily focused on operator scheduling [6, 7] and/or restructuring the query graph [5] to minimize memory or latency. These approaches did not explore benefits of integrating query operator execution with timestamp management that were first studied by [9] using periodic punctuation tuples. In this paper, we show that these benefits can be maximized by managing on-demand ETS as part of the backtrack mechanisms of execution models. The concept of punctuation, originally proposed to deal with blocking operators [8], has proven useful in many different roles, including data stream joins [4], out-of order tuples [12], and heartbeat propagation to idle-waiting unions and joins [9].

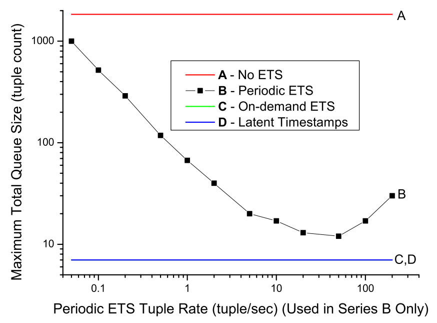


Figure 8. Peak Total Queue Size

In this paper, we have proposed integrated techniques for timestamp management and query execution that can greatly reduce the memory usage and the latency in queries with union and join operators. Our experiments show the improvements so obtained significantly surpass the periodic timestamp approach proposed in [9].

References

[1] Stream Mill home. http://wis.cs.ucla.edu/stream-mill.

[2] B. Babcock, S. Babu, M. Datar, R. Motawani, and J. Widom. Models and issues in data stream systems. In *PODS*, 2002.

[3] Yijian Bai, Hetal Thakkar, Haixun Wang, and Carlo Zaniolo. A data stream language and system designed for power and extensibility. In *CIKM '06*, 2006.

[4] L. Ding and E. A. Rundensteiner. Evaluating window joins over punctuated streams. In *CIKM '04*, pages 98–107, 2004.

[5] B. Babcock et. al. Chain: Operator scheduling for memory minimization in data stream systems. In *SIGMOD*, 2003.

[6] Don Carney et. al. Operator scheduling in a data stream manager. In *VLDB*, 2003.

[7] Mohamed A. Sharaf et. al. Preemptive rate-based operator scheduling in a data stream management system. In *AICCSA*, 2005.

[8] Peter A. Tucker et. al. Exploiting punctuation semantics in continuous data streams. *TKDE*, pages 555–568, 2003.

[9] Theodore Johnson et. al. A heartbeat mechanism and its application in gigascope. In *VLDB*, pages 1079–1088, 2005.

[10] Jaewoo Kang, Jeffrey F. Naughton, and Stratis Viglas. Evaluating window joins over unbounded streams. In *ICDE*, pages 341–352, 2003.

[11] Yan-Nei Law, Haixun Wang, and Carlo Zaniolo. Data models and query language for data streams. In *VLDB*, pages 492–503, 2004.

[12] Utkarsh Srivastava and J. Widom. Flexible time management in data stream systems. In *PODS*, pages 263–274, 2004.