Joining Punctuated Streams

Luping Ding, Nishant Mehta, Elke A. Rundensteiner, and George T. Heineman

Department of Computer Science, Worcester Polytechnic Institute
100 Institute Road, Worcester, MA 01609
{lisading, nishantm, rundenst, heineman}@cs.wpi.edu

Abstract. We focus on stream join optimization by exploiting the constraints that are dynamically embedded into data streams to signal the end of transmitting certain attribute values. These constraints are called *punctuations*. Our stream join operator, PJoin, is able to remove no-longer-useful data from the state in a timely manner based on punctuations, thus reducing memory overhead and improving the efficiency of probing. We equip PJoin with several alternate strategies for purging the state and for propagating punctuations to benefit down-stream operators. We also present an extensive experimental study to explore the performance gains achieved by purging state as well as the trade-off between different purge strategies. Our experimental results of comparing the performance of PJoin with XJoin, a stream join operator without a constraint-exploiting mechanism, show that PJoin significantly outperforms XJoin with regard to both memory overhead and throughput.

1 Introduction

1.1 Stream Join Operators and Constraints

As stream-processing applications, including sensor network monitoring [14], online transaction management [18], and online spreadsheets [9], to name a few, have gained in popularity, continuous query processing is emerging as an important research area [1] [5] [6] [15] [16]. The join operator, being one of the most expensive and commonly used operators in continuous queries, has received increasing attention [9] [13] [19]. Join processing in the stream context faces numerous new challenges beyond those encountered in the traditional context. One important new problem is the *potentially unbounded runtime join state*. Since the join needs to maintain in its join state the data that has already arrived in order to compare it against the data to be arriving in the future. As data continuously streams in, the basic stream join solutions, such as symmetric hash join [22], will indefinitely accumulate input data in the join state, thus easily causing memory overflow.

XJoin [19] [20] extends the symmetric hash join to avoid memory overflow. It moves part of the join state to the secondary storage (disk) upon running out of memory. However, as more data streams in, a large portion of the join state will be paged to disk. This will result in a huge amount of I/O operations. Then the performance of XJoin may degrade in such circumstances.

In many cases, it is not practical to compare every tuple in a potentially infinite stream with all tuples in another also possibly infinite stream [2]. In response, the recent work on window joins [4] [8] [13] extends the traditional join semantics to only join tuples within the current time windows. This way the memory usage of the join state can be bounded by timely removing tuples that drop out of the window. However, choosing an appropriate window size is non-trivial. The join state may be rather bulky for large windows.

[3] proposes a k-constraint-exploiting join algorithm that utilizes *statically specified constraints*, including clustered and ordered arrival of join values, to purge the data that have finished joining with the matching cluster from the opposite stream, thereby shrinking the state.

However, the static constraints only characterize restrictive cases of real-world data. In view of this limitation, a new class of constraints called *punctuations* [18] has been proposed to dynamically provide meta knowledge about data streams. Punctuations are embedded into data streams (hence called *punctuated streams*) to signal the end of transmitting certain attribute values. This should enable stateful operators like *join* to discard partial join state during the execution and blocking operators like *group-by* to emit partial results.

In some cases punctuations can be provided actively by the applications that generate the data streams. For example, in an *online auction management system* [18], the *sellers portal* merges items for sale submitted by sellers into a stream called *Open*. The *buyers portal* merges the bids posted by bidders into another stream called *Bid*. Since each item is open for bid only within a specific time period, when the open auction period for an item expires, the auction system can insert a punctuation into the *Bid* stream to signal the end of the bids for that specific item.

The query system itself can also derive punctuations based on the semantics of the application or certain static constraints, including the join between key and foreign key, clustered or ordered arrival of certain attribute values, etc. For example, since each tuple in the *Open* stream has a unique *item_id* value, the query system can then insert a punctuation after each tuple in this stream signaling no more tuple containing this specific *item_id* value will occur in the future. Therefore punctuations cover a wider realm of constraints that may help continuous query optimization. [18] also defines the rules for algebra operators, including join, to purge runtime state and to propagate punctuations down-stream. However, no concrete punctuation-exploiting join algorithms have been proposed to date. This is the topic we thus focus on in this paper.

1.2 Our Approach: PJoin

In this paper, we present the first punctuation-exploiting stream join solution, called PJoin. PJoin is a binary hash-based equi-join operator. It is able to exploit punctuations to achieve the optimization goals of reducing memory overhead and of increasing the data output rate. Unlike prior stream join operators stated above, PJoin can also propagate appropriate punctuations to benefit down-stream operators. Our contributions of PJoin include:

1. We propose alternate strategies for purging the join state, including eager and lazy purge, and we explore the trade-off between different purge strategies regarding the memory overhead and the data output rate experimentally.
2. We propose various strategies for propagating punctuations, including eager and lazy index building as well as propagation in push and pull mode. We also explore the trade-off between different strategies with regard to the punctuation output rate.
3. We design an event-driven framework for accommodating all PJoin components, including memory and disk join, state purge, punctuation propagation, etc., to enable the flexible configuration of different join solutions.
4. We conduct an experimental study to validate our preformance analysis by comparing the performance of PJoin with XJoin [19], a stream join operator without a constraint-exploiting mechanism, as well as the performance of using different state purge strategies in terms of various data and punctuation arrival rates. The experimental results show that PJoin outperforms XJoin with regard to both memory overhead and data output rate.

In Section 2, we give background knowledge and a running example of punctuated streams. In Section 3 we describe the execution logic design of PJoin, including alternate strategies for state purge and punctuation propagation. An extensive experimental study is shown in Section 4. In Section 5 we explain related work. We discuss future extensions of PJoin in Section 6 and conclude our work in Section 7.

2 Punctuated Streams

2.1 Motivating Example

We now explain how punctuations can help with continuous query optimization using the *online auction* example [18] described in Section 1.1. Fragments of *Open* and *Bid* streams with punctuations are shown in Figure 1 (a). The query in Figure 1 (b) joins all items for sale with their bids on *item_id* and then sum up bid-increase values for each item that has at least one bid. In the corresponding query plan shown in Figure 1 (c), an *equi-join* operator joins the *Open* stream with the *Bid* stream on *item_id*. Our PJoin operator can be used to perform this equi-join. Thereafter, the *group-by* operator groups the output stream of the join (denoted as Out_1) by *item_id*. Whenever a punctuation from *Bid* is obtained which signals the auction for a particular item is closed, the tuple in the state for the *Open* stream that contains the same *item_id* value can then be purged. Furthermore, a punctuation regarding this *item_id* value can be propagated to the Out_1 stream for the *group-by* to produce the result for this specific item.

2.2 Punctuations

Punctuation semantics. A punctuation can be viewed as a predicate on stream elements that must evaluate to $false$ for every element $following$ the

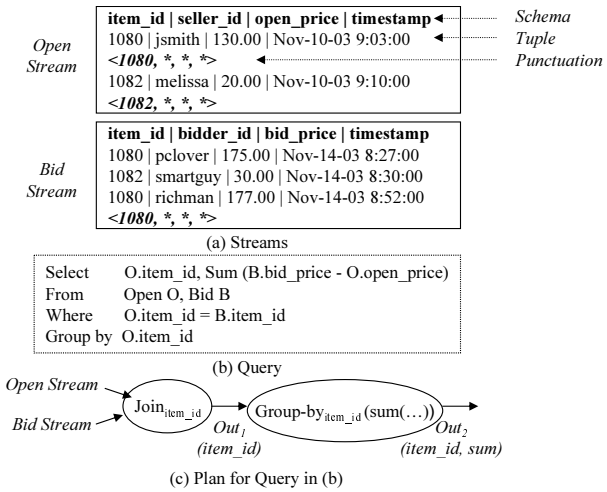

Fig. 1. Data Streams and Example Query.

punctuation, while the stream elements that appear $before$ the punctuation can evaluate either to $true$ or to $false$. Hence a punctuation can be used to detect and purge the data in the join state that *won't* join with any future data.

In PJoin, we use the same punctuation semantics as defined in [18], i.e., a punctuation is an ordered set of *patterns*, with each pattern corresponding to an attribute of a tuple. There are five kinds of patterns: wildcard, constant, range, enumeration list and empty pattern. The "and" of any two punctuations is also a punctuation. In this paper, we only focus on exploiting punctuations over the join attribute. We assume that for any two punctuations p_i and p_j such that p_i arrives before p_j, if the patterns for the join attribute specified by p_i and p_j are Ptn_i and Ptn_j respectively, then either $Ptn_i \wedge Ptn_j = \emptyset$ or $Ptn_i \wedge Ptn_j = Ptn_i$. We denote all tuples that arrived before time T from stream A and B as tuple sets $TS_A(T)$ and $TS_B(T)$ respectively. All punctuations that arrived before time T from stream A and B are denoted as punctuation sets $PS_A(T)$ and $PS_B(T)$ respectively. According to [18], if a tuple t has a join value that matches the pattern declared by the punctuation p, then t is said to match p, denoted as $match(t, p)$. If there exists a punctuation p_A in $PS_A(T)$ such that the tuple t matches p_A, then t is defined to also match the set $PS_A(T)$, denoted as $setMatch(t, PS_A(T))$.

Purge rules for join. Given punctuation sets $PS_A(T)$ and $PS_B(T)$, the purge rules for tuple sets $TS_A(T)$ and $TS_B(T)$ are defined as follows:

$$\begin{aligned} &\forall t_A \in TS_A(T),\ purge(t_A)\ \ if\ \ setMatch(t_A, PS_B(T)) \\ &\forall t_B \in TS_B(T),\ purge(t_B)\ \ if\ \ setMatch(t_B, PS_A(T)) \end{aligned} \tag{1}$$

Propagation rules for join. To propagate a punctuation, we must guarantee that no more tuples that match this punctuation will be generated later. The propagation rules are derived based on the following theorem.

Theorem 1. *Given $TS_A(T)$ and $PS_A(T)$, for any punctuation p_A in $PS_A(T)$, if at time T, no tuple t_A exists in $TS_A(T)$ such that match(t_A,p_A), then no tuple t_R such that match(t_R, p_A) will be generated as a join result at or after time T.*

Proof by contradiction. Assume that at least one tuple t_R such that match(t_R, p_A) will be generated as a join result at or after time T. Then there must exist at least one tuple t in $TS_A(T_k)$ ($T_k \geq T$) such that match(t, p_A). Based on the definition of punctuation, there will not be any tuple t_A to be arriving from stream A after time T such that match(t_A, p_A). Then t must have been existing in $TS_A(T)$. This contradicts the premise that no tuple t_A exists in $TS_A(T)$ such that match(t_A, p_A). Therefore, the assumption is wrong and no tuple t_R such that match(t_R, p_A) will be generated as a join result at or after time T. Thus p_A can be propagated safely at or after time T. □

The propagation rules for $PS_A(T)$ and $PS_B(T)$ are then defined as follows:

$$\begin{aligned} &\forall p_A \in PS_A(T),\ propagate(p_A) \quad if \quad \forall t_A \in TS_A(T), \neg match(t_A, p_A) \\ &\forall p_B \in PS_B(T),\ propagate(p_B) \quad if \quad \forall t_B \in TS_B(T), \neg match(t_B, p_B) \end{aligned} \tag{2}$$

3 PJoin Execution Logic

3.1 Components and Join State

Components. Join algorithms typically involve multiple subtasks, including: (1) probe in-memory join state using a new tuple and produce result for any match being found (*memory join*), (2) move part of the in-memory join state to disk when running out of memory (*state relocation*), (3) retrieve data from disk into memory for join processing (*disk join*), (4) purge no-longer-useful data from the join state (*state purge*) and (5) propagate punctuations to the output stream (*punctuation propagation*).

The frequencies of executing each of these subtasks may be rather different. For example, *memory join* runs on a per-tuple basis, while *state relocation* executes only when memory overflows and *state purge* is activated upon receiving one or multiple punctuations. To achieve a fine-tuned, adaptive join execution, we design separate components to accomplish each of the above subtasks. Furthermore, for each component we explore a variety of alternate strategies that can be plugged in to achieve optimization in different circumstances, as further elaborated upon in Section 3.2 through Section 3.5. To increase the throughput, several components may run concurrently in a multi-threaded mode. Section 3.6 introduces our event-based framework design for PJoin.

Join state. Extending from the symmetric hash join [22], PJoin maintains a separate state for each input stream. All the above components operate on this shared data storage. For each state, a *hash table* holds all tuples that have arrived but have not yet been purged. Similar to XJoin [19], each hash bucket has an in-memory portion and an on-disk portion. When memory usage of the join state reaches a *memory threshold*, some data in the memory-resident portion will be moved to the on-disk portion. A *purge buffer* contains the tuples which should be purged based on the present punctuations, but cannot yet be purged safely because they may possibly join with tuples stored on disk. The purge buffer will be cleaned up by the disk join component. The punctuations that have arrived but have not yet been propagated are stored in a *punctuation set*.

3.2 Memory Join and Disk Join

Due to the memory overflow resolution explained in Section 3.3 below, for each new input tuple, the matching tuples in the opposite state could possibly reside in two different places: memory and disk. Therefore, the join operation can happen in two components. The *memory join* component will use the new tuple to probe the memory-resident portion of the matching hash bucket of the opposite state and produce the result, while the *disk join* component will fetch the disk-resident portion of some or all the hash buckets and finish the left-over joins due to the state relocation (Section 3.3). Since the disk join involves I/O operations which are much more expensive than in-memory operations, the policies for scheduling these two components are different. The memory join is executed on a per-tuple basis. Only when the memory join cannot proceed due to the slow delivery of the data or when punctuation propagation needs to finish up all the left-over joins, will the disk join be scheduled to run. Similar to XJoin [19], we associate an *activation threshold* with the disk join to model how aggressively it is to be scheduled for execution.

3.3 State Relocation

PJoin employs the same memory overflow resolution as XJoin, i.e., moving part of the state from memory to secondary storage (disk) when the memory becomes full (reaches the *memory threshold*). The corresponding component in PJoin is called *state relocation*. Readers are referred to [19] for further details about the state relocation.

3.4 State Purge

The state purge component removes data that will no longer contribute to any future join result from the join state by applying the purge rules described in Section 2. We propose two state purge strategies, *eager (immediate) purge* and *lazy (batch) purge*. *Eager purge* starts to purge the state whenever a punctuation is obtained. This can guarantee the minimum memory overhead caused by the

join state. Also by shrinking the state in an aggressive manner, the state probing can be done more efficiently. However, since the state purge causes the extra overhead for scanning the join state, when punctuations arrive very frequently so that the cost of state scan exceeds the saving of probing, eager purge may instead slow down the data output rate. In response, we propose a *lazy purge* which will start purging when the number of new punctuations since the last purge reaches a *purge threshold*, which is the number of punctuations to be arriving between two state purges. We can view eager purge as a special case of lazy purge, whose purge threshold is 1. Accordingly, finding an appropriate purge threshold becomes an important task. In Section 4 we experimentally assess the effect on PJoin performance posed by different purge thresholds.

3.5 Punctuation Propagation

Besides utilizing punctuations to shrink the runtime state, in some cases the operator can also propagate punctuations to benefit other operators down-stream in the query plan, for example, the *group-by* operator in Figure 1 (c). According to the propagation rules described in Section 2, a join operator will propagate punctuations in a *lagged* fashion, that is, before a punctuation can be released to the output stream, the join must wait until all result tuples that match this punctuation have been safely output. Hence we consider to initiate propagation periodically. However, each time we invoke the propagation, each punctuation in the *punctuation sets* needs to be evaluated against all tuples currently in the same state. Therefore, the punctuations which were not able to be propagated in the previous propagation run may be evaluated against those tuples that have already been compared with last time, thus incurring duplicate expression evaluations. To avoid this problem and to propagate punctuations correctly, we design an incrementally maintained *punctuation index* which arranges the data in the join state by punctuations.

Punctuation index. To construct a punctuation index (Figure 2 (c)), each punctuation in the punctuation set is associated with a unique ID (*pid*) and a *count* recording the number of matching tuples that reside in the same state (Figure 2 (a)). We also augment the structure of each tuple to add the *pid* which denotes the punctuation that matches the tuple (Figure 2 (b)). If a tuple matches multiple punctuations, the *pid* of the tuple is always set as the *pid* of the first arrived punctuation found to be matched. If the tuple is not valid for any existing punctuations, the *pid* of this tuple is null. Upon arrival of a new punctuation p, only tuples with *pid* field being *null* need to be evaluated against p. Therefore the punctuation index is constructed incrementally so to avoid the duplicate expression evaluations. Whenever a tuple is purged from the state, the punctuation whose *pid* corresponds the *pid* contained by the purged tuple will deduct its *count* field. When the *count* of a punctuation reaches 0 which means no tuple matching this punctuation exists in the state, according to Theorem 1 in Section 2, this punctuation becomes propagable. The punctuations being propagated are immediately removed from the punctuation set.

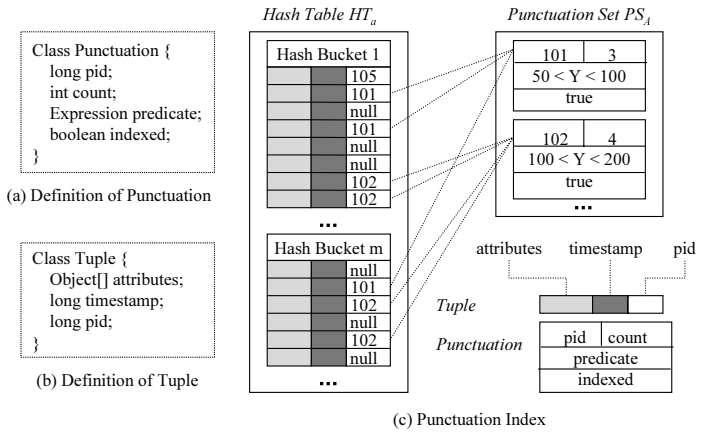

Fig. 2. Data Structures for Punctuation Propagation.

Algorithms for index building and propagation. We can see that punctuation propagation involves two important steps: *punctuation index building* which associates each tuple in the join state with a punctuation and *propagation* which outputs the punctuations with the *count* field being zero. Clearly, propagation relies on the index building process. Figure 3 shows the algorithm for constructing a punctuation index for tuples from stream B (Lines 1-14) and the algorithm for propagating punctuations from stream B to the output stream (Lines 16-21).

```
1.  Procedure Index-Build-B () {
2.    ArrayList pIndexSet = new ArrayList();
3.    /* Select all punctuations from B punctuation set PS_B not being used for indexing tuples. */
4.    foreach p_i in PS_B
5.        if (! (p_i.indexed))
6.            pIndexSet.add(p_i);
7.    /* Index all tuples in the B hash table HT_b that have not yet been indexed. */
8.    foreach bucket_k in HT_b
9.        foreach t_j in bucket_k
10.           if (t_j.pid == null)
11.               foreach p_i in pIndexSet
12.                   if (match(t_j, p_i)) {
13.                       t_j.pid = p_i.pid; /* Assign pid to the matching tuple. */
14.                       continue; } }
15.
16. Procedure Propagate-B () {
17.   /* Output and remove all punctuations whose count field is 0 in B punctuation set PS_B. */
18.   foreach p_i in PS_B
19.       if (p_i.count == 0) {
20.           output(p_i); /* Release p_i to output stream. */
21.           remove(PS_B, p_i); /* Remove p_i from B punctuation set PS_B */ } }
```

Fig. 3. Algorithms of Punctuation Index Building and Propagation.

Eager and lazy index building. Although our incrementally constructed punctuation index avoids duplicate expression evaluations, it still needs to scan the entire join state to search for the tuples whose *pid*s are null each time it is executed. We thus propose to batch the index building for multiple punctuations in order to share the cost of scanning the state. Accordingly, instead of triggering the index building upon the arrival of each punctuation, which we call *eager index building*, we run it only when the punctuation propagation is invoked, called *lazy index building*. However, eager index building is still preferred in some cases. For example, it can help guarantee the steady instead of bursty output of punctuations whenever possible. In the eager approach, since the index is incrementally built right upon receiving each punctuation and the index is indirectly maintained by the state purge, some punctuations may be detected to be propagable much earlier than the next invocation of propagation.
Propagation mode. PJoin is able to trigger punctuation propagation in either *push* or *pull* mode. In the *push* mode, PJoin actively propagates punctuations when either a fixed time interval since the last propagation has gone by, or a fixed number of punctuations have been received since the last propagation. We call them *time propagation threshold* and *count propagation threshold* respectively. On the other hand, PJoin is also able to propagate punctuations upon the request of the down-stream operators, which would be the beneficiaries of the propagation. This is called the *pull* mode.

3.6 Event-driven Framework of PJoin

To implement the PJoin execution logic described above, with components being tunable, a join framework which incorporates the following features is desired.

1. The framework should keep track of a variety of runtime parameters that serve as the triggering conditions for executing each component, such as the size of the join state, the number of punctuations that arrived since the last state purge, etc. When a certain parameter reaches the corresponding threshold, such as the purge threshold, the appropriate components should be scheduled to run.
2. The framework should be able to model the different coupling alternatives among components and easily switch from one option to another. For example, the lazy index building is coupled with the punctuation propagation, while the eager index building is independent of the punctuation propagation strategy selected by a given join execution configuration.

To accomplish the above features, we have designed an *event-driven* framework for PJoin as shown in Figure 4. The memory join runs as the main thread. It continuously retrieves data from the input streams and generates results. A *monitor* is responsible for keeping track of the status of various runtime parameters about the input streams and the join state being changed during the execution of the memory join. Once a certain threshold is reached, for example the size of the join state reaches the memory threshold or both input streams are

temporarily stuck due to network delay and the disk join activation threshold is reached, the monitor will invoke the corresponding event. Then the listeners of the event, which may be either disk join, state purge, state relocation, index build or punctuation propagation component, will start running as a second thread. If an event has multiple listeners, these listeners will be executed in an order specified in the event-listener registry described below.

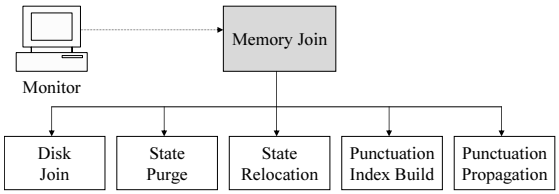

Fig. 4. Event-Driven Framework of PJoin.

The following events have been defined to model the status changes of monitored *runtime parameters* that may cause a component to be activated.

1. *StreamEmptyEvent* signals both input streams run out of tuples.
2. *PurgeThresholdReachEvent* signals the *purge threshold* is reached.
3. *StateFullEvent* signals the size of the in-memory join state reaches the *memory threshold*.
4. *NewPunctReadyEvent* signals a new punctuation arrives.
5. *PropagateRequestEvent* signals a propagation request is received from downstream operators.
6. *PropagateTimeExpireEvent* signals the *time propagation threshold* is reached.
7. *PropagateCountReachEvent* signals the *count propagation threshold* is reached.

PJoin maintains an *event-listener registry*. Each entry in the registry lists the event to be generated, the additional conditions to be checked and the listeners (components) which will be executed to handle the event. The registry while initiated at the static query optimization phase can be updated at runtime. All parameters for invoking the events, including the *purge*, *memory* and *propagation threshold*, are specified inside the monitor and can also be changed at runtime.

Table 1 gives an example of this registry. This configuration of PJoin is used by several experiments shown in Section 4. In this configuration, we apply the *lazy purge* strategy, that is, to purge state whenever the purge threshold is reached. Also the *lazy index building* and the *push* mode propagation are applied, that is, when the count propagation threshold is reached, we first construct the punctuation index for all newly-arrived punctuations since the last index building and then start propagation.

Events	*Conditions*	*Listeners (Activated In Order)*
StreamEmptyEvent	Activation threshold is reached.	Disk Join
PurgeThresholdReachEvent	none	State Purge
StateFullEvent	C1*	State Purge
StateFullEvent	C2*	State Relocation
PropagateCountReachEvent	none	Index Build, Propagation
C1*: There exists punctuations which haven't been used to purge the state. C2*: No punctuations exist that haven't been used to purge the state.		

Table 1. Example Event-Listener Registry.

4 Experimental Study

We have implemented the PJoin operator in Java as a query operator in the Raindrop XQuery subscription system [17] based on the event-based framework presented in Section 3.6. Below we describe the experimental study we have conducted to explore the effectiveness of our punctuation-exploiting stream join optimization. The test machine has a 2.4GHz Intel(R) Pentium-IV processor and a 512MB RAM, running Windows XP and Java 1.4.1.01 SDK. We have created a benchmark system to generate synthetic data streams by controlling the arrival patterns and rates of the data and punctuations. In all experiments shown in this section, the tuples from both input streams have a Poisson inter-arrival time with a mean of 2 milliseconds. All experiments run a many-to-many join over two input streams, which, we believe, exhibits the most general cases of our solution. In the charts, we denote the PJoin with purge threshold n as PJoin-n. Accordingly, PJoin using eager purge is denoted as PJoin-1.

4.1 PJoin vs. XJoin

First we compare the performance of PJoin with XJoin [19], a stream join operator without a constraint-exploiting mechanism. We are interested in exploring two questions: (1) how much memory overhead can be saved and (2) to what degree can the tuple output rate be improved. In order to be able to compare these two join solutions, we have also implemented XJoin in our system and applied the same optimizations as we did for PJoin.

To answer the first question, we compare PJoin using the eager purge with XJoin regarding the total number of tuples in the join state during the length of the execution. The input punctuations have a Poisson inter-arrival with a mean of 40 tuples/punctuation. From Figure 5 we can see that the memory requirement for the PJoin state is almost insignificant compared to that of XJoin.

As the punctuation inter-arrival increases, the size of the PJoin state will increase accordingly. When the punctuation inter-arrival reaches infinity so that no punctuations exist in the input stream, the memory requirement of PJoin becomes the same as that of XJoin.

In Figure 6, we vary the punctuation inter-arrival to be 10, 20 and 30 tuples/punctuation respectively for three different runs of PJoin accordingly. We

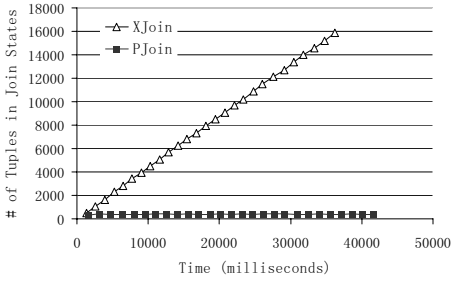

Fig. 5. PJoin vs. XJoin, Memory Overhead, Punctuation Inter-arrival: 40 tuples/punctuation.

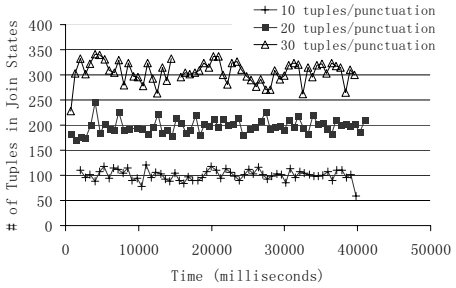

Fig. 6. PJoin Memory Overhead, Punctuation Inter-arrival: 10, 20, 30 tuples/punctuation.

can see that as the punctuation inter-arrival increases, the average size of the PJoin state becomes larger correspondingly.

To answer the second question, Figure 7 compares the tuple output rate of PJoin to that of XJoin. We can see that as time advances, PJoin maintains an almost steady output rate whereas the output rate of XJoin drops. This decrease in XJoin output rate occurs because the XJoin state increases over time thereby leading to an increasing cost for probing state. From this experiment we conclude that PJoin performs better or at least equivalent to XJoin regarding both the output rate and the memory resources consumption.

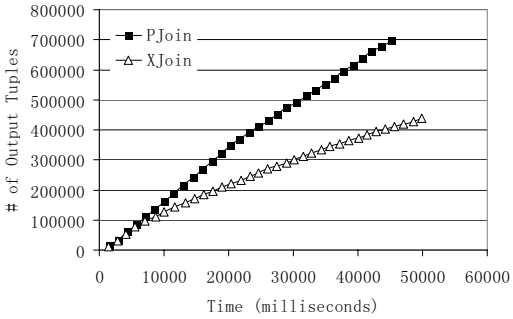

Fig. 7. PJoin vs. XJoin, Tuple Output Rates, Punctuation Inter-arrival: 30 tuples/punctuation.

4.2 State Purge Strategies for PJoin

Now we explore how the performance of PJoin is affected by different state purge strategies. In this experiment, the input punctuations have a Poisson inter-arrival with a mean of 10 tuples/punctuation. We vary the *purge threshold* to start

purging state after receiving every 10, 100, 400, 800 punctuations respectively and measure its effect on the output rate and memory overhead of the join.

Figure 8 shows the state requirements for the eager purge (PJoin-1) and the lazy purge with purge threshold 10 (PJoin-10). The chart confirms that the eager purge is the best strategy for minimizing the join state, whereas the lazy purge requires more memory to operate.

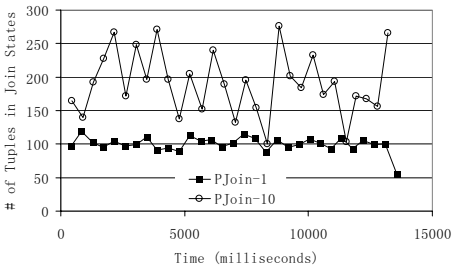

Fig. 8. Eager vs. Lazy Purge, Memory Overhead, Punctuation Inter-arrival: 10 tuples/punctuation.

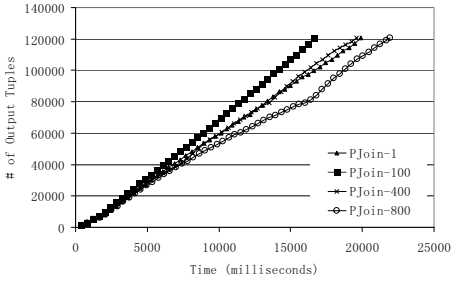

Fig. 9. Eager vs. Lazy Purge, Tuple Output rates, Punctuation Inter-arrival: 10 tuples/punctuation.

Figure 9 compares the PJoin output rate using different purge strategies. We plot the number of output tuples against time summarized over four experiment runs, each run with a different purge threshold (1,100,400 and 800 respectively). We can see that up to some limit, the higher the *purge threshold*, the higher the output rate. This is because there is a cost associated with purge, and thus purging very frequently such as the eager strategy leads to a loss in performance. But this gain in output rate is at the cost of the increase in memory overhead. When the increased cost of probing the state exceeds the cost of purge, we start to lose on performance, such as the case of PJoin-400 and PJoin-800. This is the same problem as encountered by XJoin, that is, every new tuple enlarges the state, which in turn increases the cost of probing the state.

4.3 Asymmetric Punctuation Inter-arrival Rate

Now we explore the performance of PJoin in terms of input streams with asymmetric punctuation inter-arrivals. We keep the punctuation inter-arrival of stream A constant at 10 tuples/punctuation and vary that of stream B. Figure 10 shows the state requirement of PJoin using eager purge. We can see that the larger the difference in the punctuation inter-arrival of the two input streams, the larger will be the memory requirement. Less frequent punctuations from stream B cause the A state to be purged less frequently. Hence the A state becomes larger.

Another interesting phenomenon not shown here is that the B state is very small or insignificant compared to the A state. This happens because punctuations from stream A arrive at a faster rate. Thus most of the time when a B

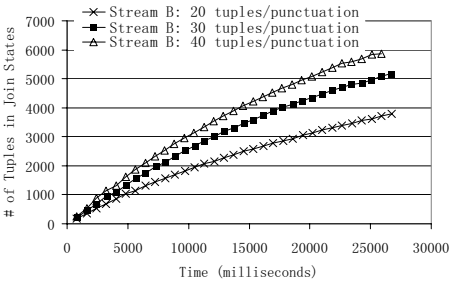

Fig. 10. Memory Overhead, Asymmetric Punctuation Inter-arrival Rates, A Punctuation Inter-arrival: 10 tuples/punctuation, B Punctuation Inter-arrival: 20, 30, 40 tuples/punctuation.

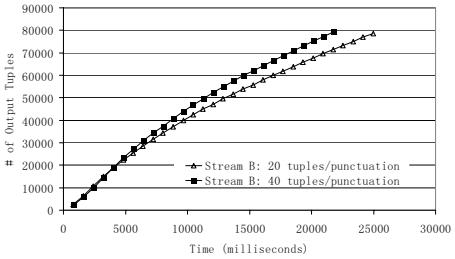

Fig. 11. Tuple Output Rates, Asymmetric Punctuation Inter-arrival Rates, A Punctuation Inter-arrival: 10 tuples/punctuation, B Punctuation Inter-arrival: 20, 40 tuples/punctuation.

tuple is received, there already exists an A punctuation that can drop this B tuple on the fly [7]. Therefore most B tuples never become a part of the state.

Figure 11 gives an idea about the tuple output rate of PJoin for the above cases. The slower the punctuation arrival rate, the greater is the tuple output rate. This is because the slow punctuation arrival rate means a smaller number of purges and hence the less overhead caused by purge.

Figure 12 shows the comparison of PJoin against XJoin in terms of asymmetric punctuation inter-arrivals. The punctuation inter-arrival of stream A is 10 tuples/punctuation and that of stream B is 20 tuples/punctuation. We can see that the output rate of PJoin with the eager purge (PJoin-1) lags behind that of XJoin. This is mainly because of the cost of purge associated with PJoin. One way to overcome this problem is to use the lazy purge together with an appropriate setting of the *purge threshold*. This will make the output rate of PJoin better or at least equivalent to that of XJoin. Figure 13 shows the state requirements for this case. We conclude that if the goal is to minimize the memory overhead of the join state, we can use the eager purge strategy. Otherwise the lazy purge with an appropriate purge threshold value can give us a significant advantage in tuple output rate, at the expense of insignificant increase in memory overhead.

4.4 Punctuation Propagation

Lastly, we test the punctuation propagation ability of PJoin. In this experiment, both input streams have a punctuation inter-arrival with a mean of 40 tuples/punctuation. We show the ideal case in which punctuations from both input streams arrive in the same order and of same granularity, i.e., each punctuation contains a constant pattern. PJoin is configured to start propagation after a pair of equivalent punctuations has been received from both input streams.

Figure 14 shows the number of punctuations being output over time. We can see that PJoin can guarantee a steady punctuation propagation rate in the ideal case. This property can be very useful for the down-stream operators such as *group-by* that themselves rely on the availability of input punctuations.

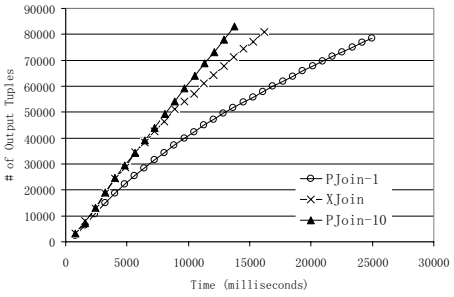

Fig. 12. Eager vs. Lazy Purge, Output Rates, Asymmetric Punctuation Inter-arrival Rates, A Punctuation Inter-arrival: 10 tuples/punctuation, B Punctuation Inter-arrival: 20 tuples/punctuation.

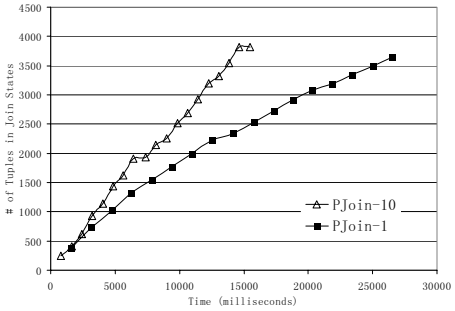

Fig. 13. Eager vs. Lazy Purge, Memory Overhead, Asymmetric Punctuation Inter-arrival Rates, A Punctuation Inter-arrival: 10 tuples/punctuation, B Punctuation Inter-arrival: 20 tuples/punctuation.

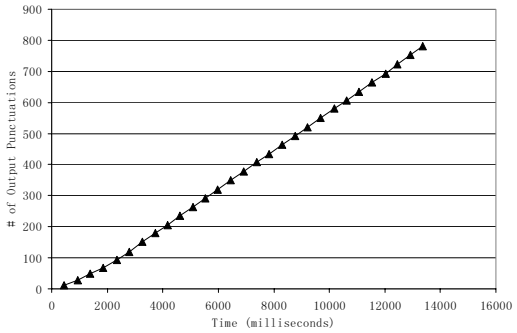

Fig. 14. Punctuation Propagation, Punctuation Inter-arrival: 40 tuples/punctuation

5 Related Work

As the data being queried has expanded from finite and statically available datasets to distributed continuous data streams ([1] [5] [6] [15]), new problems have arisen. Specific to the join processing, two important problems need to be tackled: potentially unbounded growing join state and dynamic runtime features of data streams such as widely-varying data arrival rates. In response, the *constraint-based join optimization* [16] and *intra-operator adaptivity* [11] [12] are proposed in the literature to address these two issues respectively.

The main goal of constraint-based join optimization is to in a timely manner detect and purge the no-longer-useful data from the state. *Window joins* exploit *time-based constraints* called sliding windows to remove the expired data from the state whenever a time window passes. [1] defines formal semantics for a binary join that incorporates a window specification. Kang et al. [13] provide a unit-time-basis cost model for analyzing the performance of a binary window

join. They also propose strategies for maximizing the join efficiency in various scenarios. [8] studies algorithms for handling sliding window multi-join processing. [10] researches the shared execution of multiple window join operators. They provide alternate strategies that favor different window sizes. The *k-constraint-exploiting algorithm* [3] exploits clustered data arrival, a *value-based constraint* to help detect stale data. However, both window and k-constraints are statically specified, which only reflect the restrictive cases of the real-world data.

Punctuations [18] are a new class of constraints embedded into the stream dynamically at runtime. The static constraints such as one-to-many join cardinality and clustered arrival of join values can also be represented by punctuations. Beyond the general concepts of punctuations, [18] also lists all rules for algebra operators, namely, pass, keep (equal to purge) and propagation. In our PJoin design, we apply these functional rules to achieve join optimization, including the exploration of alternate modes for applying these rules.

Adaptive join operators can adjust their behavior in response to the changing conditions of data and computing resources as well as the runtime statistics. *Ripple joins* [9] are a family of physical pipelining join operators which are designed for producing partial results quickly. Ripple joins adjust their behavior during processing in accordance with the statistical properties of the data. They also consider the user preferences about the accuracy of the partial result and the time between updates of the running aggregate to adaptively set the rate of retrieving tuples from each input stream. *XJoin* [19] [20] is able to adapt to insufficient memory by moving part of the in-memory join state to the secondary storage. It also hides the intermittent delays in data arrival from slow remote resources by reactively scheduling background processing.

We apply the ideas of constraint-driven join optimization and intra-operator adaptivity in our work. PJoin is able to exploit constraints presented as punctuations to achieve the optimization goals of reducing memory overhead and increasing data output rates. PJoin also adopts almost all features of XJoin. We differ in that no previous work incorporates both constraint-exploiting mechanism and adaptivity into join execution logic itself. Unlike the k-constraint-exploiting algorithm, PJoin does not always start to purge state upon receiving a punctuation. Instead, it allows tuning options in order to do it in an optimized way, such as the lazy purge strategy. The user can adjust the behavior of PJoin by specifying a set of parameters statically or at runtime. PJoin can also propagate appropriate punctuations to benefit the down-stream operators, which neither window joins nor k-constraint-exploiting algorithms do.

6 Discussion: Extending PJoin Beyond Punctuations

The current implementation of PJoin is a binary equi-join without exploiting window specifications because we want to first focus on exploring the impact of punctuations on the join performance. As we have experimentally shown, simply by making use of appropriate punctuations, the join state may already be kept bounded this way. However, the design of PJoin being based on a flexible

event-driven framework is easily-extendible to support alternate join components, tuning options, sliding windows and to handle n-ary join.

Extension for supporting sliding window. To support sliding window, additional tuple dropping operation needs to be introduced to purge expired tuples as the window moves. This operation can be performed in combination with the state probing in the memory join and the disk join components. In addition, the tuples in each hash bucket can be arranged by their timestamps so that the early-arrived tuples are always accessed first. This way the tuple invalidation by window can perform more effectively. Whenever the first time-valid tuple according to the current window is encountered, the tuple invalidation for this hash bucket can stop. Furthermore, the interaction between punctuations and windows may enable further optimization such as early punctuation propagation.

Extension for handling n-ary join. It is also straightforward to extend the current binary join implementation of PJoin to handle n-ary joins [21]. The modifications to be made for the state purge component are as follows: instead of purging the state of stream B by punctuations from stream A, in an n-ary join, for punctuations from the i^{th} stream, the state purge component needs to purge the states of all other (n-1) streams. The punctuation index building and propagation algorithms for each input stream could remain the same. The memory join component needs to be modified as well. If the join value of a new tuple from one stream is detected to match the punctuations from all other (n-1) streams, this tuple can be on-the-fly dropped after the memory join. Otherwise we need to insert this tuple into its state. There exist prolific optimization tasks in terms of forming partial join results, designing a correlated purge threshold, designing a correlated propagation threshold, to name a few.

7 Conclusion

In this paper, we presented the design of a punctuation-exploiting stream join operator called PJoin. We sketched six components to accomplish the PJoin execution logic. For state purge and propagation, we designed alternate strategies to achieve different optimization goals. We implemented PJoin using an event-driven framework to enable the flexible configuration of join execution for coping with the dynamic runtime environment. Our experimental study compared PJoin with XJoin, explores the impact of different state purge strategies and evaluates the punctuation propagation ability of PJoin. The experimental results illustrated the benefits achieved by our punctuation-exploiting join optimization.

Acknowledgment. The authors wish to thank Leonidas Fegaras for many useful comments on our work, which lead to improvements of this paper.

References

1. D. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A new model and architecture for data stream management. *VLDB Journal*, 12(2):120–139, August 2003.

2. A. Arasu, B. Babcock, S. Babu, J. McAlister, and J. Widom. Characterizing memory requirements for queries over continuous data streams. In *PODS*, pages 221–232, June 2002.
3. S. Babu and J. Widom. Exploiting k-constraints to reduce memory overhead in continuous queries over data streams. Technical report, Stanford Univ., Nov 2002.
4. D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring streams - a new class of data management applications. In *VLDB*, pages 215–226, August 2002.
5. S. Chandrasekaran, O. Cooper, 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*, pages 269–280, Jan 2003.
6. J. Chen, D. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for internet databases. In *ACM SIGMOD*, pages 379–390, June 2002.
7. L. Ding, E. A. Rundensteiner, and G. T. Heineman. MJoin: A metadata-aware stream join operator. In *DEBS*, June 2003.
8. L. Golab and M. T. Ozsu. Processing sliding window multi-joins in continuous queries over data streams. In *VLDB*, pages 500–511, Sep 2003.
9. P. Haas and J. Hellerstein. Ripple joins for online aggregation. In *ACM SIGMOD*, pages 287–298, June 1999.
10. M. A. Hammad, M. J. Franklin, W. G. Aref, and A. K. Elmagarmid. Scheduling for shared window joins over data streams. In *VLDB*, pages 297–308, Sep 2003.
11. J. M. Hellerstein, M. J. Franklin, S. Chandrasekaran, A. Deshpande, K. Hildrum, S. Madden, V. Raman, and M. Shah. Adaptive query processing: Technology in evolution. *IEEE Data Engineering Bulletin*, 23(2):7–18, Jun 2000.
12. Z. G. Ives, D. Florescu, M. Friedman, A. Levy, and D. S. Weld. An adaptive query execution system for data integration. In *ACM SIGMOD*, pages 299–310, 1999.
13. J. Kang, J. F. Naughton, and S. D. Viglas. Evaluating window joins over unbounded streams. In *ICDE*, pages 341–352, March 2003.
14. S. Madden and M. Franklin. Fjording the stream: An architecture for queries over streaming sensor data. In *ICDE*, pages 555–566, Feb 2002.
15. S. Madden, M. Shah, J. M. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *ACM SIGMOD*, pages 49–60, June 2002.
16. R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma. Query processing, resource management, and approximation in a data stream management system. In *CIDR*, pages 245–256, Jan 2003.
17. H. Su, J. Jian, and E. A. Rundensteiner. Raindrop: A uniform and layered algebraic framework for XQueries on XML streams. In *CIKM*, pages 279–286, Sep 2003.
18. P. A. Tucker, D. Maier, T. Sheard, and L. Fegaras. Exploiting punctuation semantics in continuous data streams. *IEEE Transactions on Knowledge and Data Engineering*, 15(3):555–568, May/June 2003.
19. T. Urhan and M. Franklin. XJoin: A reactively scheduled pipelined join operator. *IEEE Data Engineering Bulletin*, 23(2):27–33, 2000.
20. T. Urhan and M. J. Franklin. Dynamic pipeline scheduling for improving interactive query performance. In *VLDB*, pages 501–510, Sep 2001.
21. S. Viglas, J. Naughton, and J. Burger. Maximizing the output rate of multi-way join queries over streaming information. In *VLDB*, pages 285–296, Sep 2003.
22. A. N. Wilschut and P. M. G. Apers. Dataflow query execution in a parallel main-memory environment. *Distributed and Parallel Databases*, 1(1):103–128, 1993.