Evaluating Window Joins over Punctuated Streams

Luping Ding and Elke A. Rundensteiner
Worcester Polytechnic Institute
100 Institute Road, Worcester, MA 01609
{lisading, rundenst}@cs.wpi.edu

ABSTRACT

We explore join optimizations in the presence of both time-based constraints (sliding windows) and value-based constraints (punctuations). We present the first join solution named PWJoin that exploits such combined constraints to shrink the runtime join state and to propagate punctuations to benefit downstream operators. We design a state structure for PWJoin that facilitates the exploitation of both constraint types. We also explore optimizations enabled by the interactions between window and punctuation, e.g., early punctuation propagation. The costs of the PWJoin are analyzed using a cost model. We also conduct an experimental study using CAPE continuous query system. The experimental results show that in most cases, by exploiting punctuations, PWJoin outperforms the pure window join with regard to both memory overhead and throughput. Our technique complements the joins in the literature, such as symmetric hash join or window join, to now require less runtime resources without compromising the accuracy of the result.

Categories and Subject Descriptors

H.2 [**Database Management**]: Miscellaneous

General Terms

Algorithms

Keywords

Join Algorithm, Streaming Data Processing, Punctuation, Sliding Window

1. INTRODUCTION

1.1 Stream Join Processing

As more and more applications need to query continuous data streams, such as sensor networks [13], online transaction management [17], and online spreadsheet [10], to name a few, continuous query processing is emerging as an important research area. The join processing techniques in this new context have received increasing attention because evaluating a join over continuous data streams may require potentially unbounded memory in order to maintain the history data to join with the future data. The well-known pipelined join solutions, including symmetric hash join [21], ripple joins [10], XJoin [19, 20], and hash-merge join[14], will indefinitely accumulate input data in the join state as data continuously streams in. Thus they may easily consume a huge amount of memory in a short time. As the join state becomes bulky, the join probe may perform inefficiently, thereby affecting the operator's throughput.

It is clearly not practical to compare every tuple in one potentially infinite stream with all tuples in another also possibly infinite stream [2]. This problem has been addressed in recent work on window joins [1, 9, 12]. They extend traditional join semantics to only join tuples that occur within a certain bounded time period, i.e., the *window*. This way the memory usage of the join state can be bounded by removing tuples that drop out of the window in a timely fashion.

Recently, it has been recognized that real data streams may conform to some semantic constraints that can be utilized to detect and thus purge no-longer-needed data in the join state as the join proceeds [4, 17]. One case is that the data may be known to arrive in clusters or in rough clusters grouped by the join attribute. For example, the course grades of a particular student may be clustered together in the online spreadsheet [10]. This way the termination point of each join value is known. [17] has proposed to insert metadata, namely *punctuations*, into data streams to explicitly announce these termination points, thereby punctuating a continuous stream into sub-streams. Data streams that carry punctuations are referred to as *punctuated streams*.

In a binary join over streams A and B, whenever a punctuation about a certain join value inside stream A has arrived, the B tuples, either already-arrived ones or future-coming ones, that contain this punctuated value as their join value will not be joining with any future tuples from stream A. Hence these B tuples no longer need to be maintained in the join state. The same purge rule applies to tuples from stream A. As our example in Section 1.2 illustrates, punctuations can also be derived by the join operator and be propagated to benefit downstream operators.

1.2 Motivating Example for Exploiting Combined Constraints

In this paper, we present the first stream join solution

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CIKM'04, November 8–13, 2004, Washington, DC, USA.
Copyright 2004 ACM 1-58113-874-1/04/0011 ...$5.00.

that achieves optimizations by exploiting both punctuations (value-based constraints) and sliding windows (time-based constraints). While either constraint type can help bound the memory usage, we now illustrate via an example in the online auction application [18] that in some situations more advantages can be achieved by considering both of them.

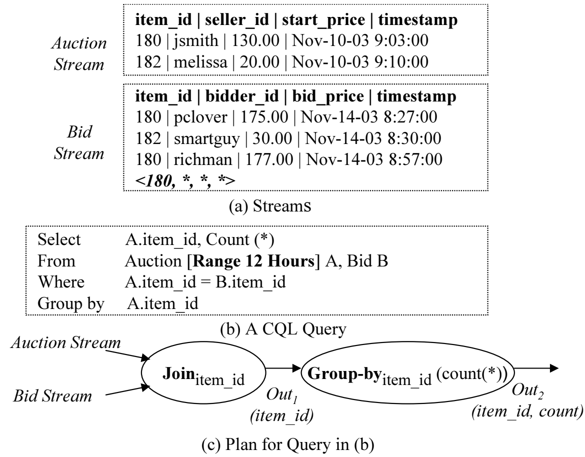

Figure 1: Example Streams and Query in Online Auction System.

As shown in Figure 1, in the auction system, each auction is represented by a tuple in the *Auction* stream. Each bid placed by the bidder is represented by a tuple in the *Bid* stream. Tuples in both streams arrive in the order of their timestamp, which represents their open auction or bidding time. When the open duration for an auction expires, the auction system can insert a punctuation into the Bid stream to signal the end of bids for that particular auction. The punctuation <180, *, *, *> in Bid stream in Figure 1 (a) indicates that no more bids will be placed for auction 180.

Let us consider the CQL [3] query in Figure 1 (b). This query asks for the number of bids for each auction within 12 hours of its opening (if the auction has at least one bid). The corresponding query plan (Figure 1 (c)) contains an equi-join operator which joins streams Auction and Bid on *item_id*. A group-by operator groups tuples in the output stream (Out_1) of the join by *item_id* and then evaluates the aggregate function count() for each group. The join operator applies a 12-hour window to the Auction stream.

However, if many auctions are opened within 12 hours and for each auction there is a large number of bids, the join state of both streams may become huge. We notice that each auction has a unique *item_id*. Hence each tuple from the Bid stream does not need to be maintained in the state after being joined because it is not going to match any future Auction tuples.

Second, each Auction tuple, whose open duration is less than 12 hours, can be removed from the state before the 12-hour window ends once the corresponding punctuation is received from the Bid stream. Meanwhile a punctuation can be sent to the Out_1 stream to announce no more join result will be generated for this auction. The group-by operator can thus produce a result for this auction prior to the end of the 12-hour window. This way punctuations enable the query to emit partial results earlier.

Third, for each Auction tuple whose open period is longer than 12 hours, when it moves out of its window, no more join results will be produced for this auction, though the Bid tuples for this auction may still come. Then the join operator can propagate a punctuation for this auction earlier than it would have been possible in the non-windowed case. In addition, any future Bid tuples for this auction can be directly dropped without even being processed because they will not contribute to any future results.

1.3 Issues in Exploiting Combined Constraints

From the above example, we obtain the following observations that motivate our stream join optimization exploiting both punctuations and time windows.

1. For relatively long-lasting windows or rapid data arrivals, the join state would typically contain a huge number of tuples. By additionally exploiting punctuations, we may be able to further shrink the join state. This may also improve the probe efficiency.

2. A join operator may be able to help downstream operators by propagating punctuations, for instance, to unblock blocking operators such as the group-by. [17] defines formal punctuation propagation rules for algebra operators. However, no formal semantics for passing meta information between operators have thus far been proposed in the literature for windowed queries.

3. Further optimizations may even be achievable due to the coexistence of these two constraint types. For example, with tuples being invalidated by windows, some punctuations can be propagated much earlier. In addition, more unnecessary state probes and tuple insertions can be avoided by the early propagation.

While these huge potential benefits exist, to our best knowledge the join operator that exploits such combined constraints has not yet been considered in the literature.

Sliding windows and punctuations are constraints about different aspects of the data, i.e., the timestamp and the value respectively. In many cases they have an overlapping effect on shrinking join state. An ill-designed join algorithm may achieve minor gains at possibly double overhead, thereby resulting in worse performance than the join solutions customized to only exploit one of these constraint types. Correspondingly, the following questions arise regarding potential join algorithms that could exploit both punctuations and windows.

1. Can the join storage structure and execution logic be designed to facilitate some optimization strategy to exploit both constraint types?

2. If no punctuations are provided, would performance of punctuation-exploiting window join be considerably worse than that of the pure window join due to the overhead of being aware of punctuations?

3. If both constraint types are applicable simultaneously, would punctuation-exploiting window join always perform better than the join that only exploits one constraint type? In which cases would it perform worse?

4. What optimization strategies can be devised that exploit the interaction of these two constraint types and

that would not be applicable in the presence of only punctuation or only sliding window?

In this paper, we seek to address these questions by presenting our approach, namely PWJoin, the first punctuation-exploiting window join solution.

1.4 Summary of Contributions

Our contributions of this research include: (1) we propose the first **P**unctuation-exploiting binary **W**indow **Join** algorithm (PWJoin), (2) we design a novel storage structure for the PWJoin state that effectively accomplishes the awareness of both constraint types, (3) we derive a formula for estimating the cost of the PWJoin, and (4) we conduct an extensive experimental study in our CAPE continuous query system [16] that compares PWJoin with a pure punctuation-exploiting join [7] and a pure window join [12]. The experimental results show that by utilizing punctuations in addition to sliding windows, PWJoin consumes less memory than the pure window join. For relatively large windows, PWJoin achieves a higher tuple output rate than the pure window join. On the flip side, even in some cases when the join that exploits only punctuations is not able to propagate punctuations, PWJoin can still achieve a steady punctuation output rate to benefit downstream operators. This way the performance of the overall query plan can be improved.

1.5 Roadmap

Section 2 discusses related work. Section 3 provides background knowledge about punctuation and sliding window. The PWJoin state structure and the PWJoin algorithm are described in Section 4. We derive the cost formula for PWJoin in Section 5 and show the experimental study in Section 6. Finally we conclude our work in Section 7.

2. RELATED WORK

As query evaluation over continuous data streams receives increasing attention, several data stream management systems have been built, including Aurora [1], STREAM [15] TelegraphCQ [5], NiagaraCQ [6], to name a few.

Specific to join processing, the first well-known pipelined join solution is the *symmetric hash join* [21]. *XJoin* [19, 20] and *ripple joins* [10] are extended pipelined joins that are designed for special optimization purposes. However, all these join solutions face the problem of potentially unbounded runtime join state as data continuously streams in.

A lot of work has been done in bounding memory consumption in join evaluation by exploiting constraints. *Window join* [1, 9, 12] exploits *time-based constraints* called windows to constrain the relative time range for tuples to be joined. Tuples that drop out of the window can be removed from the state. [1] defines formal semantics for a binary window join operator. Kang et al. [12] provide a unit-time-basis cost model for analyzing the performance of binary window join algorithms, which we apply and then extend here to estimate the memory cost of PWJoin. They also propose strategies for maximizing the join efficiency in various scenarios. [9] studies algorithms for handling sliding window multi-join processing. [11] researches the shared execution of multiple window join operators. They provide alternative strategies that favor different window sizes. [22] investigates the migration between query plans that contain window join operators in order to achieve dynamic query optimization. [8] proposes storage structures and indexing methods for sliding windows to improve the join efficiency. The design of the PWJoin state extends from their work.

Value-based constraints have also begun to be considered in the literature, though to a much lesser degree than windows. The *k-constraint-exploiting algorithm* [4] exploits clustered data arrival patterns to detect and purge expired data to shrink the join state. These clustered patterns are statically specified, and hence only characterize restrictive cases of real-world data. If the actual data fails to obey these static constraints, the precision of the join result may suffer due to the incorrect purge of tuples. Moreover, this work focuses on value-based constraint exploitation instead of exploring the interaction between window and value-based constrains, as now done in our work.

Punctuations [17] are dynamic constraints embedded inside data streams. Static constraints such as unique key and clustered arrival of attribute values can also be modeled by punctuations. Therefore, punctuation covers a wide class of constraints that may help continuous query optimization. [17] provides pass, purge and propagation rules enabled by punctuations for algebra operators. In response, a punctuation-exploiting stream join solution, *PJoin* [7], is proposed. PJoin extends XJoin to maintain a more compact join state by timely purging useless data according to punctuations. However, it does not handle window semantics.

We apply the ideas of constraint-exploiting join optimization in our PWJoin solution. Different from the k-constraint-exploiting algorithm, PWJoin exploits dynamic constraints, i.e., punctuations, which cover both static k-constraints and dynamic data value arrival patterns. PWJoin enhances PJoin by considering an additional dimension of constraints, namely, windows. In addition, we now investigate the performance impact as well as synergy of both constraint types.

3. PRELIMINARIES

3.1 Punctuation

A punctuation is an ordered set of patterns, each corresponding to an attribute of tuple. All tuples that arrive *after* the punctuation are guaranteed to never contain any attribute values specified in the corresponding patterns. The pattern can be a wildcard $*$, a constant (also, single-value), a data range, an enumeration list or an empty pattern.

In this paper, we consider the punctuations that specify a *single-value* pattern for the *join* attribute, i.e., each punctuation signals the end of a single *join value*. We choose this pattern because it occurs very commonly in the real-world data streams, as shown in our example in Section 1.2. Other punctuation patterns including range and enumeration list can also be modeled by single-value patterns in most cases. In addition, the granularity of single-value patterns are consistent so that we can obtain a concise cost model without compromising the generality. We term the value that is announced to be no longer arriving by a punctuation as a *punctuated value* of that stream.

We assume that all data streams consist of relational tuples. Each tuple contains an attribute att representing the join attribute and a timestamp ts that records the time when the tuple enters into the stream. Given a punctuation p from a stream that specifies the punctuated value val_p, any tuple t from any stream, whose join value equals val_p, is defined to *match* p, denoted as $match(t,\ p)$. Two punctuations p_a

and p_b from streams A and B respectively are defined to match each other if they specify the same punctuated value.

Below we restate the *purge* and *propagation* rules defined in [17] for the equi-join operator regarding single-value punctuations. Both rules depend on the join execution logic that the join process for each tuple must be finished before the next tuple is retrieved and processed from the stream. Tuples from both input streams are retrieved alternatively in the ascending order of their timestamp.

Purge rule. For any tuple t_a from stream A, if there exists a punctuation p_b that has already been received from stream B such that match(t_a, p_b), t_a will not be joining with any future arriving tuples from stream B. Hence t_a does not need to be maintained in the A state after being processed. This purge rule similarly applies to any tuples from stream B.

Propagation rule. The join operator can also propagate punctuations to the output stream in order to help downstream operators. Based on punctuation semantics, we derive the following theorem as the foundation of our punctuation propagation algorithm.

THEOREM 3.1. *Let p_a and p_b be punctuations retrieved from streams A and B at time TS_a and TS_b respectively specifying the same punctuated value val of join attribute att. Then no output tuples with val being the value of attribute att will be generated after time max(TS_a, TS_b).*

Proof. Let time TS be $\max(TS_a, TS_b)$. According to punctuation semantics, no more tuples with the join value *val* will be arriving from either stream after time TS. Based on the join execution logic, all tuples from both streams that precede p_a and p_b have finished their join processes before time TS. Therefore no more tuples with attribute *att* having value *val* will be produced after time TS. □

We now are ready to derive our punctuation propagation algorithm based on Theorem 3.1. Whenever a punctuation is received from one stream, we check whether the matching punctuation from the other stream has already been seen. If yes, since no more output tuples containing this punctuated value for attribute *att* will be generated thereafter, this value becomes a punctuated value for the attribute *att* in the output stream. Therefore, the corresponding punctuation can be emitted to the output stream.

3.2 Sliding Window

The sliding window constrains the relative time range within which tuples from both streams can be joined. Our PWJoin supports the basic window join semantics illustrated in Figure 2. We are only interested in the join of each new A tuple whose timestamp is TS_a with all B tuples that arrived within the last T_b time units prior to TS_a and the join of each new B tuple whose timestamp is TS_b with all A tuples that arrived within the last T_a time units prior to TS_b. T_a and T_b are called *time windows* for streams A and B respectively. Below we define the rule for invalidating [1] tuples from the join state based on the sliding window.

Tuple invalidation rule. Let tuple t_a be the latest tuple with timestamp TS_a from stream A that has been processed. The tuple in the B state with timestamp TS_b such that

[1]In order to distinguish the purge of tuples by sliding window from that by punctuation, thereafter we will use *invalidation* and *purge* to refer to the purge of tuples by window and by punctuation respectively.

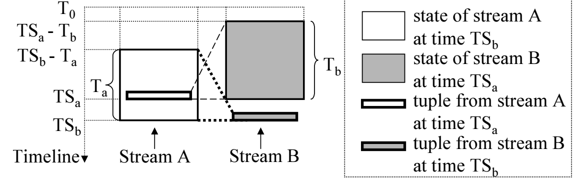

Figure 2: Basic Window Join.

$TS_b + T_b < TS_a$ is called a *time-expired tuple* and can be invalidated. The same invalidation rule applies to tuples in the A state. We call this *cross invalidation*.

4. PWJOIN SOLUTION

Being aware of both window and punctuation, the PWJoin execution logic is composed of three operations: (1) *probing* state to find matching tuples for producing join results, (2) *purging* no-longer-joining tuples by punctuations and (3) *invalidating* expired tuples by windows. Among these operations, *probe* and *purge* conduct value-based searches, while *invalidation* needs time-based searches. Therefore the join state storage structure must be designed to be efficient for both value-based search and time-based search.

4.1 Storage Structure for PWJoin State

State. Similar to most pipelined join operators, PWJoin performs a symmetric execution logic for processing tuples and punctuations from both input streams. Thus it maintains two states, each containing tuples from one input stream that have been processed so far and may still be joining with the future incoming tuples from the other stream.

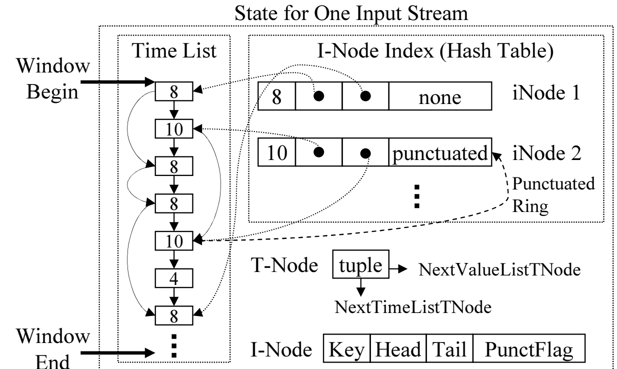

Figure 3: Storage Structure for PWJoin State.

Figure 3 shows the storage structure for one state of PWJoin. We employ the LIST structure [8] to link all tuples in the state in chronological order (newest tuple at the tail) into a *time list*. The head and the tail of the time list are indicated by *Window Begin* and *Window End* pointers respectively. As the window moves, tuples are in turn removed from the head of the time list. Moreover, tuples containing the same join value are linked into a *value list* (also in chronological order). In short, all tuples in the state form a single time list and multiple value lists. Each tuple is held by a linked list node, which we call *T-Node*. Each T-Node contains two additional pointers: *NextTimeListTNode* points to the next

T-Node in the time list and *NextValueListTNode* points to the next T-Node in the same value list.

In addition, an index node, which we call *I-Node*, is created for each value list. Each I-Node contains the join value (Key), two pointers pointing respectively to the head and the tail T-Nodes of the value list it is indexing, and a $PunctFlag$ indicating whether a punctuation for this value has been seen from this stream. For a particular punctuation type, we have a customized index for organizing I-Nodes. For single-valued or list-valued punctuations, we use a hash-based index while for range-valued punctuations we instead use a tree-structured index. Since we only focus on single-value punctuations in this paper, the I-Nodes in Figure 3 are maintained in a hash table, which we call the *I-Node index*.

Using this storage structure, the *probe* and the *purge* first search the I-Node index to find the matching value list while the *invalidation* first checks the head of the time list for detecting the expired tuples. Hence all three operations perform efficiently because they directly obtain the tuples that they are interested in while the access of irrelevant tuples is avoided. In addition, only two pointers (*Head* and *Tail*) are maintained in the I-Node for each distinct value. This incurs little cost for maintaining the index structure as tuples dynamically enter and leave the state.

4.2 PWJoin Algorithm

Now we describe the PWJoin algorithm regarding processing tuples and punctuations from input stream A. The processing of tuples and punctuations from input stream B is similar due to the symmetric execution logic. The pseudo code is shown in Figure 4.

1. *Invalidation.* Once a new tuple t is retrieved from stream A, its timestamp is used to invalidate expired tuples from the head of the time list of stream B. This process stops when the first unexpired tuple is encountered, which becomes the new head of this time list.

2. *Probe.* After invalidation is done, the join value of t is used to probe the I-Node index of the B state. If the matching I-Node $iNode$ is found, the corresponding value list is located by following the Head pointer of $iNode$. Tuple t then joins with all tuples in this value list by following the NextValueListTNode pointer of each T-Node. Finally, the PunctFlag of $iNode$ is checked. If it is "punctuated", t is discarded, called *on-the-fly discard*. If it is "none", t is inserted into the A state. The value of this flag can also be "propagated" for directly dropping tuples without even processing them, as will be explained in Section 4.4.

3. *Purge.* When a new punctuation p is retrieved from stream A, p is used to probe the I-Node index of the B state. If the matching I-Node $iNode$ is found, all tuples in the corresponding value list are deleted. $iNode$ is removed from the I-Node index as well. If the PunctFlag of $iNode$ is "punctuated", p is discarded. If $iNode$ is not found or $iNode$'s PunctFlag is "none", p is used to probe the I-Node index of the A state and set the PunctFlag of the matching I-Node $iNode_a$ as "punctuated". If $iNode_a$ does not exist, a new I-Node is created with its PunctFlag marked as true and inserted into the I-Node index of the A state.

```
01. PROCEDURE PWJoin()
02.   IF (a new object obj is received from stream A)
03.         IF (obj is a tuple)
04.             Invalidate(obj, S_b);
05.             Probe(obj, S_a, S_b);
06.         ELSE IF (obj is a punctuation)
07.             Purge(obj, S_a, S_b);
08.   IF (a new object obj is received from stream B)
09.         /* Similar to above. */
10.
11. PROCEDURE Invalidate(Tuple t, State probeState)
12.   T-Node tNode = probeState.getTimeList().getHead();
13.   WHILE (tNode != NULL)
14.         Tuple tHead = tNode.getTuple();
15.         Long T_w = probeState.getWindow();
16.         IF (tHead.ts + T_w < t.ts)
17.             tNode.setTuple(NULL);
18.             tNode = tNode.getNextTimeListTNode();
19.         ELSE /* Encounter the first time-valid tuple. */
20.             RETURN;
21.
22. PROCEDURE Probe(Tuple t, State ownState,
              State probeState)
23.   I-Node iNode = probeState.GetIndexNode(t.att);
24.   IF (iNode == NULL)
25.         ownState.insertTuple(t);
26.   ELSE
27.         T-Node tNode = iNode.getHead();
28.         WHILE (tNode != NULL)
29.             Tuple t_i = tNode.getTuple();
30.             outStream.append(join(t, t_i));
31.             tNode = tNode.getNextValueListTNode();
32.         IF (iNode.PunctFlag == NONE)
33.             ownState.insertTuple(t);
34.
35. PROCEDURE Purge(Punctuation p, State ownState,
              State probeState)
36.   I-Node iNode = probeState.GetIndexNode(p.att);
37.   IF (iNode == NULL)
38.         ownState.insertPunct(p);
39.   ELSE
40.         IF (iNode.PunctFlag == PUNCTUATED)
41.             iNode.removeValueListTuples();
42.             probeState.removeIndexNode(iNode);
43.         ELSE
44.             iNode.removeValueListTuples();
45.             /* Mark the matching I-Node as punctuated. */
46.             ownState.markPunctuated(p);
```

Figure 4: PWJoin Algorithm.

By examining the PWJoin algorithm, we can see that the purge operation is triggered by arrival of punctuations. For data streams not containing punctuations, the *purge* operation will never be performed, thus causing zero overhead. In addition, the cost for on-the-fly discard is minimal because it is accomplished as a side effect of the probe operation, i.e., by checking the PunctFlag of the matching I-Node. Therefore, we predict that our design enables PWJoin to achieve almost the same performance as the pure window join for non-punctuated streams. In dealing with punctuated streams, on-the-fly discard may provide huge gains by avoiding a good number of unnecessary tuple insertions and deletions. The purge can also effectively shrink the state and hence improve the probe efficiency. So PWJoin should perform better than the pure window join in most cases.

4.3 Tuple Insertion and Deletion

The PWJoin algorithm involves frequent operations for inserting and deleting tuples into and from the state. These operations must guarantee that both the time list and the value lists are updated correctly.

Tuple insertion. Inserting a new tuple t_a into the A state

follows the steps listed below:

1. A new T-Node $tNode_a$ is created to contain t_a and is appended to the end of the time list (pointed by *Window End* pointer). This can thus be done within constant time.

2. The join value of t_a is used to probe the I-Node index of the A state. If the matching I-Node $iNode_a$ exists, a NextValueListTNode pointer is pointed from the tail T-Node of the value list to $tNode_a$. The $Tail$ pointer of $iNode_a$ also points to $tNode_a$. Else if $iNode_a$ does not exist, a new I-Node $iNode_a$ is created and both $Head$ and $Tail$ pointers point to $tNode_a$.

Tuple deletion. To delete a tuple t_a from the A state, two cases must be considered: (1) the tuple is deleted by *invalidation* and (2) the tuple is deleted by *purge*. Assume that t_a is contained in a T-Node $tNode_a$. In case (1), we first remove $tNode_a$ from the head of the time list by pointing the *Window Begin* pointer to the next T-Node in the time list. Then we need to adjust the $Head$ pointer (sometimes also the $Tail$ pointer) of the corresponding I-Node that is currently pointing to $tNode_a$. However, this will incur an extra probe on the I-Node index to find the I-Node. And this may become a significant overhead because it happens for each tuple being invalidated from the time list.

In response, we propose a *lazy* T-Node deletion strategy. After $tNode_a$ is removed from the time list, we don't immediately adjust the pointers of the corresponding I-Node. Instead we only set the tuple in the T-Node to null to release the memory taken by this tuple. Next time when the *probe* operation accesses the value list associated with this I-Node, all T-Nodes containing a null tuple, which can be located at the head of the value list, will be detected and removed from the value list. Similarly, in case (2), when the *purge* operation deletes all T-Nodes from a value list, we only set the tuple in the T-Node to null to release the memory taken by the tuple. Next time when the time list is probed by the *invalidation*, all T-Nodes containing a null tuple will be removed from the time list.

4.4 Punctuation Propagation

As we explained in Section 1.2, in some cases the PWJoin is able to propagate punctuations to help downstream operators. The propagation is not beneficial to the operator itself and rather it costs extra overhead. However, from the perspective of a whole query plan, the propagated punctuations may bring huge advantages to the downstream operators. For example, they may unblock the blocking operators such as the *group-by*. If we are able to achieve propagation at a negligible cost, a significant performance gain may be achieved for evaluating the whole query plan.

According to the propagation rule described in Section 3.1, once the punctuation about a particular join value has been received from both streams, a punctuation about this value can be announced to the output stream. Therefore, we can modify the *purge* operation above (Section 4.2) to enable the propagation. That is, upon retrieving a new punctuation p from stream A, p is used to probe the I-Node index of the B state. If the matching I-Node $iNode$ is found and its $PunctFlag$ is "punctuated", a punctuation about this join value will be sent to the output stream. The pseudo code for the modified *purge* operation is shown in Figure 5. Note that line 09 is the only code being added.

```
01. PROCEDURE Purge(Punctuation p, State ownState,
                 State probeState)
02.   I-Node iNode = probeState.GetIndexNode(p.att);
03.   IF (iNode == NULL)
04.        ownState.insertPunct(p);
05.   ELSE
06.        IF (iNode.PunctFlag == PUNCTUATED)
07.             iNode.removeValueListTuples();
08.             probeState.removeIndexNode(iNode);
09.             outStream.append(new Punctuation(p.att));
10.        ELSE
11.             iNode.removeValueListTuples();
12.             ownState.markPunctuated(p);
```

Figure 5: Purge with Propagation.

Early propagation. By considering punctuations in combination with sliding windows, we discover that a new propagation rule can be derived that enables further optimizations. As illustrated in Figure 6, given that the punctuation p_a is received before the punctuation p_b, without window semantics, according to the propagation rule in Section 3.1, the punctuation <180> can only be propagated after p_b is received. However, by considering sliding windows, once the last A tuple containing join value 180 moves out of the window, no more such tuple will appear in the A state. Hence no more result tuples containing 180 will be generated in the future. Therefore the punctuation <180> can be propagated without needing to wait for the arrival of p_b. We can see that this new propagation point is always earlier or at least same as the one that occurs for the pure punctuation-exploiting join operator. We have derived the following theorem that the early propagation is based on. The proof is omitted due to space reasons.

THEOREM 4.1. *Let $t_{a,i}$ be the last tuple from stream A that contains value val_i for join attribute att and $t_{a,i}$ is invalidated by the time window T_a at time T. Then no result tuple that contains value val_i for the attribute att will be generated after time T. Hence a punctuation announcing "att = val_i" can be appended to the output stream.*

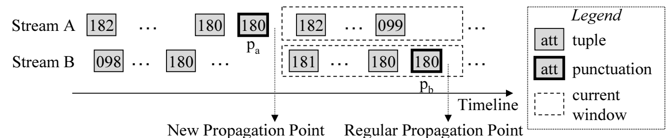

Figure 6: Early Punctuation Propagation.

Based on this theorem, we are able to achieve early propagation by simply keeping track of the last tuple for each punctuated join value. To do this, we introduce a *PunctuatedINode* pointer into the T-Node. The markPunctuated() method is also revised so that the PunctuatedINode pointer of the tail T-Node in corresponding value list is set to point back to the matching I-Node after this I-Node is marked as punctuated. This way a *punctuated ring* is created for each punctuated join value, as shown in Figure 3. In the $Invalidate$ procedure, when a T-Node with a not null PunctuatedINode pointer is removed from the time list, the corresponding punctuation will be propagated.

When an early propagation happens (assuming it is due to the last A tuple that contains the punctuated join value being invalidated from the window), we will not remove the corresponding I-Node from the A state immediately. Since tuples containing this join value may still be arriving from stream B but they are not joining with any future A tuples, we keep this I-Node and set its PunctFlag to "propagated" in order to drop these B tuples. This I-Node will be removed only when the matching punctuation arrives from stream B, i.e., at the regular propagation point.

5. ESTIMATING COST OF PWJOIN

In this section, we apply the *unit-time-basis* cost model proposed in [12] to derive the formula for estimating the memory cost of the PWJoin algorithm. We use this formula to compare the performance of PWJoin with the pure window join algorithm. Table 1 lists the notations related to stream B that will be used in our cost analysis. The notations related to stream A can be derived by switching the subscript "b" of these notations to "a".

Notation	*Meaning*
λ_b	input rate of tuples from stream B
λ_{pb}	input rate of punctuations from stream B
T_b	time window for stream B
$NK_{b,T}$	# of distinct join values having occurred in stream B up to the T'th time unit
S_b	join state of stream B
$\|S_b\|_T$	number of tuples in S_b at the T'th time unit
$\|S_b\|_T^{insert}$	# of tuples inserted into S_b at the T'th time unit
$\|S_b\|_T^{purge}$	# of tuples purged from S_b at the T'th time unit

Table 1: Notations Used in Cost Model.

5.1 Data Arrival Patterns and Assumptions

In our cost analysis in this section and the experimental study in Section 6, we consider three different *data arrival patterns* for the join values: (1) *Unique data arrival* in which each tuple is followed by a punctuation. In other words, the join attribute acts as a unique key. (2) *Clustered data arrival* in which tuples having the same join value arrive successively in the stream. A corresponding punctuation is appended to the end of each cluster. The number of tuples within a cluster is called *cluster size*. The unique data arrival is a special case of the clustered arrival in which the cluster size is equal to 1. (3) A *general punctuated stream* in which a punctuation is declared immediately after the tuple that contains the last occurrence of a particular join value. It does not constrain that tuples with the same join value have to occur adjacent to each other. The tuples between two consecutive punctuations form a *punctuated segment*.

The cost formula derived here assumes an *input-limited mode* [21], i.e., the processor can always keep up with the tuple input rate. Hence at any time unit T, the number of tuples having been inserted into the join state so far equals the number of tuples that have been processed and also equals the number of tuples that have arrived thus far.

5.2 Cost Analysis

We now analyze the memory overhead of the PWJoin algorithm in terms of the number of tuples in the join state. According to window semantics, at each time unit, λ_b arriving tuples will enter the B state and stay there for T_b time units. Therefore, without purging by punctuations, at any time unit the B state will contain $\lambda_b T_b$ tuples. Similarly, $\lambda_a T_a$ tuples will be maintained in the A state.

Considering punctuations, we sketch the best case in Figure 7 in which both streams have a clustered arrival pattern and punctuations from both streams arrive in the same order regarding the punctuated value. Moreover, each B cluster arrives right *after* the matching punctuation from stream A. Then at any time unit, the maximum memory overhead equals the maximum cluster size of stream A or $\lambda_a T_a$, whichever is smaller, because no B tuple needs to be maintained in the B state. If the average cluster size is 1, which is the unique data arrival pattern, at most one tuple needs to be maintained in the A state at any time unit.

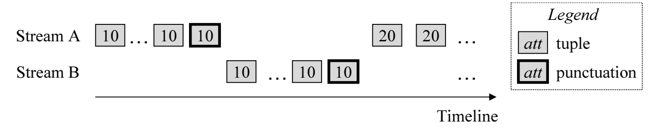

Figure 7: Synchronized Clustered Arrival.

In general, the number of tuples maintained in the B state at any time unit equals the number of tuples entering the state minus the number of tuples being purged by punctuations within that time unit. Since the join attributes of stream A and that of stream B have the same domain, so we assume here that both streams have the same number of distinct join values. Then at the T'th time unit, the probability of each B tuple to be purged by the A punctuations equals the ratio of the number of A punctuations that have been processed to the number of distinct join values that have occurred in stream B thus far, i.e., $(\lambda_{pa}\mathrm{T})/NK_{b,T}$. Then the number of B tuples being purged is $\lambda_b T_b(\lambda_{pa}\mathrm{T}/NK_{b,T})$. This is also the memory savings by PWJoin compared with the pure window join. Equation (1) computes the number of tuples in the B state at the T'th time unit.

$$\begin{aligned} |S_b|_T &= |S_b|_T^{insert} - |S_b|_T^{purge} = |S_b|_T^{arrive} - |S_b|_T^{purge} \\ &= \lambda_b T_b - \lambda_b T_b \frac{\lambda_{pa} T}{NK_{b,T}} = \lambda_b T_b \left(1 - \frac{\lambda_{pa} T}{NK_{b,T}}\right) \end{aligned} \quad (1)$$

By examining this equation, we can see that given a fixed tuple arrival rate, λ_b, and a window size, T_b, the memory overhead of PWJoin at the T'th time unit is affected by two factors: λ_{pa} and $NK_{b,T}$. On one hand, at a higher punctuation arrival rate (λ_{pa}), more punctuations will arrive from stream A up to the T'th time unit. Then more tuples in S_b may get purged so that S_b will use less memory. On the other hand, given a fixed λ_{pa}, the smaller $NK_{b,T}$ is, the higher the probability of each tuple in S_b to match and hence be purged by the already arrived punctuations from stream A could be.

In the best case illustrated in Figure 7, since the tuples containing each distinct join value will only occur within their own value-based cluster instead of widely distributed in the stream, thus $NK_{b,T}$ equals λ_{pb}T at time unit T. We know that λ_{pb} equals λ_{pa} because every B cluster immediately follows the matching A cluster. Thus $NK_{b,T}$ equals λ_{pa}T at time unit T. So we compute that $|S_b|$ remains 0 on average at all times.

In addition, the number of tuples in the PWJoin state at any time unit is also affected by the elapsed time unit T. The time window bounds the number of tuples in each join state to be $\lambda_a T_a$ or $\lambda_b T_b$, which would be a constant if the tuple arrival rate stays the same. However, as more punctuations stream in over time, the probability of tuples in the state to be purged is increased. Hence the memory overhead is reduced correspondingly.

6. EXPERIMENTAL STUDY

6.1 Experimental Setup

We have implemented the PWJoin algorithm for the equi-join operator in our Java-based continuous query system named CAPE [16]. We now report on an experimental study we have conducted to explore the effectiveness of this punctuation-exploiting window join solution. Below we show the most important results obtained from this study. Our testing machine has a 733 MHz Intel(R) Celeron(TM) processor and a 512MB RAM, running Windows2000 and Java 1.4.1.01 SDK. In order to compare PWJoin with other stream join solutions in the literature including *PJoin* [7], a pure punctuation-exploiting join, and a pure window join [12] (here we call it *WJoin*) in a comparable manner, we have implemented both PJoin and WJoin in our system and applied the same optimizations as for PWJoin.

In all experiments shown in this section, we run a many-to-many join over two input streams, which, we believe, exhibits the most general case of our solution. Moreover, the join operators are running in a *CPU-limited* mode [21], i.e., tuples stream into the join operator at a high rate so that the input streams are never drained of tuples. This way we can observe the performance of different join operators, including PJoin, WJoin and PWJoin, at their full processing ability. One may wonder whether the extra overhead of PWJoin caused by purging tuples and propagating punctuations would become a predominant cost such that the savings gained in memory then become meaningless. We note that if PWJoin's throughput is almost the same or even better than other join solutions in the *CPU-limited* mode, then in terms of slow-delivering streams, the savings in memory would be an extra gain of PWJoin because the additional time taken by purge and propagation can be hidden by the intermittent delay between tuples.

We have also built a benchmark system to generate and send synthetic data streams with control on the arrival patterns of data and punctuations. The data streams used in our experiments can have any one of the arrival patterns specified in Section 5.1. We use the following notations to specify the particular arrival patterns of each stream.

1. cluster-[*order*]-[*clustersize*]
2. punctuation-[*order*]-[*segmentsize*]-[*matchpercentage*]

Notation 1 denotes the streams with a clustered data arrival pattern. *order* specifies whether the punctuations arrive in ascending (*asc*), descending (*desc*) or random (*random*) order regarding the punctuated values. The number of tuples in each single cluster conforms to a Poisson distribution with a mean of *clustersize*. Similarly, notation 2 is used to denote the general punctuated streams. *order* and *segmentsize* bear the same meaning as *order* and *clustersize* respectively in the first notation. *matchpercentage* specifies the average percentage of tuples in a punctuated segment matching the punctuation that concludes this segment. In our experiments, this percentage for each punctuated segment conforms to a Poisson distribution where *matchpercentage* acts as a mean. Note that the tuples that match each punctuation may also occur in any other punctuated segments prior to this punctuation. The matching tuples in each punctuated segment (also a cluster) of the *clustered arrival pattern* always has a percentage of 100.

Whenever the join operator begins execution, a statistics gatherer starts to collect the performance data of the join regularly at a specified time interval, which we call a *sampling step*. The data it collects include the number of tuples currently in the states for both input streams and the number of tuples and punctuations being output thus far. All figures in this section will show performance data at each sampling step within a finite time range [2]. The sampling intervals of the experiments shown in Sections 6.2 and 6.3 are 2 seconds and 1 second respectively.

6.2 PWJoin vs. WJoin

First, we compare the performance of PWJoin with WJoin, a pure window join without being aware of punctuations. In this experiment, we explore (1) how much memory overhead can be saved by PWJoin in comparison with WJoin, (2) how the throughput of PWJoin is affected and (3) how the performance of PWJoin is affected in terms of irrelevant punctuations. We evaluate join operators over a pair of *punct-asc-100-40* streams. The inter-arrival time of tuples from each stream conforms to a Poisson distribution with a mean of 10 milliseconds. We vary the window size for input streams in different runs and record the total number of tuples in both states of the join operator and the total number of tuples output up to each sampling step. Within the same experimental run, we apply the same window size to both input streams.

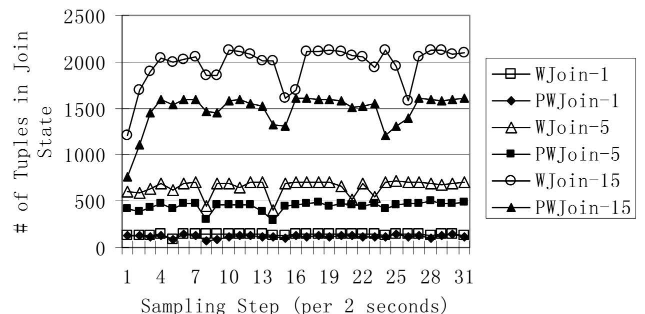

Figure 8: Memory Overhead, PWJoin vs. WJoin, Stream A, B: punct-asc-100-40.

Memory overhead. In Figure 8 we show the result of 3 runs regarding the total number of tuples in the join state of PWJoin and of WJoin, both with the time window being 1, 5, and 15 seconds respectively. Accordingly, we denote PWJoin in these 3 runs as PWJoin-1, PWJoin-5 and PWJoin-15 in the figure. The same notation also applies to WJoin. We can see that as window becomes larger, the memory savings by PWJoin become more and more significant.

[2]The join operator is continuously running. But it is only practical for us to show statistics at finite number of sampling steps in all figures in this section.

Tuple output rate. In this experiment, we also plot the number of output tuples of PWJoin and WJoin for each run. Figure 9 shows the number of output tuples of these two join solutions up to each sampling step in 2 runs, with a 5-second window on both input streams and a 15-second window on both input streams respectively. We observe that when the window is small, since the number of tuples purged by each punctuation is small, the cost of purge by punctuations exceeds the saving in probing so that WJoin performs slightly better. As the window becomes larger, the gains in probe by shrinking the state gradually takes over, the PWJoin would perform apparently better than WJoin.

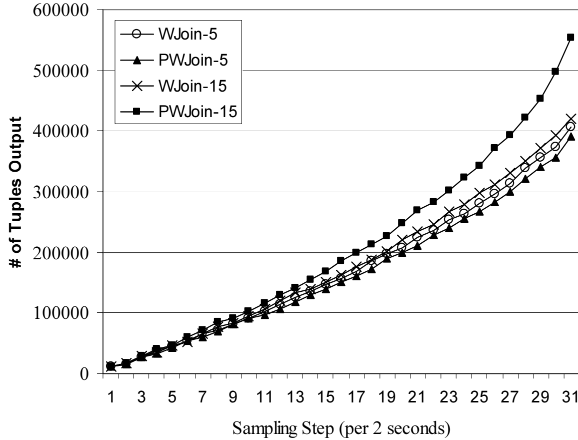

Figure 9: Tuple Output Rate, PWJoin vs. WJoin, Stream A, B: punct-asc-100-40, Window: 5 seconds, 15 seconds.

Irrelevant punctuations. Now we turn to the third question regarding the overhead required for the PWJoin operator to handle irrelevant punctuations. In terms of data streams without punctuations, the cost of PWJoin is almost the same as the WJoin because the operations caused by purge and propagation are only triggered by the arrival of punctuations. If punctuations never happen, no extra cost would be incurred in PWJoin.

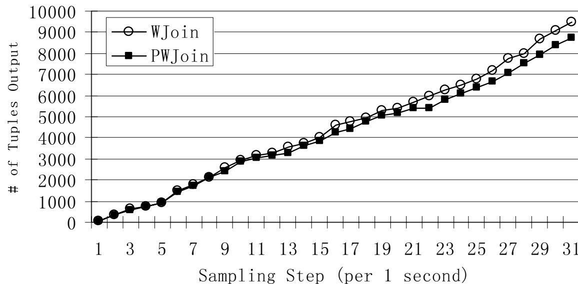

Figure 10: Tuple Output Rate, PWJoin vs. WJoin, Irrelevant Punctuations, Stream A: punct-asc-30-40, Stream B: punct-random-30-40.

Let us now consider an extreme case in which all punctuations are irrelevant, i.e., the punctuation does not match any tuples so that no tuples would ever be purged. However, PWJoin still tries to search for tuples that can be purged for each arrived punctuation. This will cost extra time. Figure 10 shows the number of output tuples by PWJoin and WJoin over a *punctuation-asc-30-40* stream and a *punctuation-random-30-40* stream, however, with all punctuations being irrelevant instead. We can see that even in this case, the extra overhead of PWJoin is insignificant. This is because the cost for processing an irrelevant punctuation equals the cost of probing a hash table for a matching I-Node. This is even less than the cost of processing a tuple because it does not incur the overhead of forming result tuples when any matches are found. Moreover, punctuations normally arrive much more infrequently than the actual tuples, in this case, 30 times less frequent. Hence we conclude that in most cases, the cost of handling punctuations is trivial compared to the potential advantages it may offer.

6.3 Synergy of Punctuation and Window

Next, we consider the synergy of punctuations and windows, i.e., the optimization enabled by the co-existence of both constraint types. As we discussed in Section 4.4, early punctuation propagation can be potentially achieved. We run PWJoin and PJoin over a *punct-asc-30-40* stream and a *punct-random-30-40* stream. Both streams have Poisson tuple inter-arrival time with a mean of 10 milliseconds. For PWJoin, a 1-second window applies to both input streams.

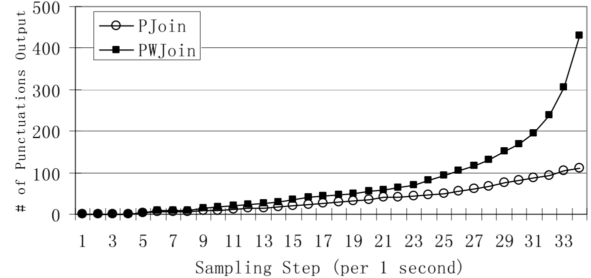

Figure 11: Punctuation Output Rate, PWJoin vs. PJoin, Stream A: punct-asc-30-40, Stream B: punct-random-30-40, Window: 1 second.

We can see from Figure 11 that PWJoin has an overall higher punctuation output rate. In addition, for a join without window specification, if only one input stream has punctuations while the other one does not, according to the punctuation propagation rule in Section 3.1, the join operator cannot derive any punctuations for the output stream. In this case no punctuation will be output. However, by adding a window constraint on the input stream that is served with punctuations, whenever the last tuple that carries a certain punctuated value drops out of the window, the corresponding punctuation can be delivered to the output stream no matter whether the matching punctuation has been declared for the other stream or not. This is also the case shown by our running example in Section 1.2. Since each tuple in the *Auction* stream contains a unique *item_id*, the auction system can derive a punctuation on *item_id* after each *Auction* tuple. Even if the *Bid* stream doesn't have punctuations, when an *Auction* tuple expires after 12 hours of staying in the join state, a punctuation can be sent to the output stream to indicate no more join result related to this particular auction will be generated.

To show these advantages enabled by the interaction between punctuations and windows, we run PWJoin over a *punct-asc-30-40* stream and a stream not containing punctuations. The join values of the tuples in the second stream are uniformly distributed within [0, 15000) and a 5-second window is applied to this stream. Figure 12 shows the total number of output punctuations by PWJoin up to each sampling step. In this case PJoin cannot deliver punctuations. However, PWJoin is still able to regularly provide punctuations to benefit downstream operators. We conclude that punctuations with time windows offer more opportunities for continuous query optimization.

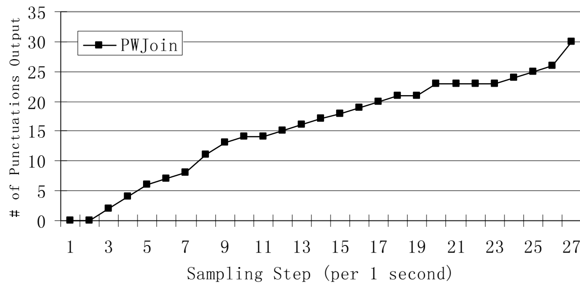

Figure 12: Punctuation Output Rate, PWJoin, Stream A: no punctuation, Stream B: punct-asc-30-40, Window B: 5 seconds.

7. CONCLUSION

In this paper we have explored stream join optimization in the presence of punctuations and sliding windows. We proposed a 3-operation-based algorithm for a punctuation-exploiting binary window join operator, PWJoin. We have derived a cost model for estimating the cost of PWJoin. Lastly, we have conducted an extensive experimental study to confirm the performance gains, synergy as well as potential overhead when exploiting these two constraint types.

As for future work, the current binary join algorithm should be extended to handle n-ary joins. Instead of only probing the B join state upon the arrival of a tuple or a punctuation from stream A, the algorithm then would need to check the states of the remaining (n-1) streams. Additional optimizations such as various purge strategies could also be explored for further tuning.

8. ACKNOWLEDGMENTS

This research was partly supported by WPI Research Development Council Award 2003-04. The authors wish to thank Nishant Mehta for the development of stream generation benchmark, CAPE group members for useful discussions, Leonidas Fegaras for valuable feedback and WPI Database Research Group for refining the talk.

9. 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] A. Arasu, S. Babu, and J. Widom. CQL: A language for continuous queries over streams and relations. In *DBPL*, pages 1–19, Sep 2003.

[4] S. Babu and J. Widom. Exploiting k-constraints to reduce memory overhead in continuous queries over data streams. *ACM Transactions on Database Systems*, 39(3), Sep 2004.

[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 *SIGMOD*, pages 379–390, June 2002.

[7] L. Ding, N. Mehta, E. A. Rundensteiner, and G. T. Heineman. Joining punctuated streams. In *EDBT*, pages 587–604, March 2004.

[8] L. Golab, S. Garg, and M. T. Ozsu. On indexing sliding windows over on-line data streams. In *EDBT*, pages 712–729, Mar 2004.

[9] L. Golab and M. T. Ozsu. Processing sliding window multi-joins in continuous queries over data streams. In *VLDB*, pages 500–511, Sep 2003.

[10] P. Haas and J. Hellerstein. Ripple joins for online aggregation. In *SIGMOD*, pages 287–298, June 1999.

[11] 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.

[12] J. Kang, J. F. Naughton, and S. D. Viglas. Evaluating window joins over unbounded streams. In *ICDE*, pages 341–352, March 2003.

[13] S. Madden and M. Franklin. Fjording the stream: An architecture for queries over streaming sensor data. In *ICDE*, pages 555–566, Feb 2002.

[14] M. F. Mokbel, M. Lu, and W. G. Aref. Hash-merge join: A non-blocking join algorithm for producing fast and early join results. In *ICDE*, Mar 2004.

[15] 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.

[16] E. A. Rundensteiner, L. Ding, T. Sutherland, Y. Zhu, B. Pielech, and N. Mehta. Cape: Continuous query engine with heterogeneous-grained adaptivity. In *VLDB Demo*, pages 1353–1356, Aug/Sep 2004.

[17] 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.

[18] P. A. Tucker, K. Tufte, V. Papadimos, and D. Maier. Nexmark - a benchmark for querying data streams. Technical report, OGI School of Science & Engineering at OHSU, 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] 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.

[22] Y. Zhu, E. A. Rundensteiner, and G. T. Heineman. Dynamic plan migration for continuous queries over data streams. In *SIGMOD*, pages 431–442, June 2004.