

Out-of-Order Processing: A New Architecture for High-Performance Stream Systems

Jin Li[¶], Kristin Tufte[¶], Vladislav Shkapenyuk[§], Vassilis Papadimos[¶],
Theodore Johnson[§], David Maier[¶]

[¶]Computer Science Department
Portland State University
Portland, OR, 97201

{jinli, tufte, vpapad, maier}@cs.pdx.edu

[§]AT&T Labs – Research
Florham Park, NJ 07932
{vshkap, johnson}@research.att.com

ABSTRACT

Many stream-processing systems enforce an order on data streams during query evaluation to help unblock blocking operators and purge state from stateful operators. Such in-order processing (IOP) systems not only must enforce order on input streams, but also require that query operators preserve order. This order-preserving requirement constrains the implementation of stream systems and incurs significant performance penalties, particularly for memory consumption. Especially for high-performance, potentially distributed stream systems, the cost of enforcing order can be prohibitive. We introduce a new architecture for stream systems, out-of-order processing (OOP), that avoids ordering constraints. The OOP architecture frees stream systems from the burden of order maintenance by using explicit stream progress indicators, such as punctuation or heartbeats, to unblock and purge operators. We describe the implementation of OOP stream systems and discuss the benefits of this architecture in depth. For example, the OOP approach has proven useful for smoothing workload bursts caused by expensive end-of-window operations, which can overwhelm internal communication paths in IOP approaches. We have implemented OOP in two stream systems, Gigascope and NiagaraST. Our experimental study shows that the OOP approach can significantly outperform IOP in a number of aspects, including memory, throughput and latency.

1. INTRODUCTION

Current stream-processing systems commonly require that input streams arrive in order, or enforce order on their input streams, and further require that stream query operators maintain order [1][2][7][9][12][13]. We refer to such order-dependent stream systems as having in-order-processing (IOP) architectures. In this paper, we introduce a new, more flexible, out-of-order-processing (OOP) architecture. We have implemented OOP in two stream systems, Gigascope [7] and NiagaraST, and our experimental study has shown significant performance improvements in both systems—for example, reducing memory use by 50% under reasonable circumstances.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Database Endowment. To copy otherwise, or to republish, to post on servers or to redistribute to lists, requires a fee and/or special permissions from the publisher, ACM.

VLDB '08, August 24-30, 2008, Auckland, New Zealand.

Copyright 2008 VLDB Endowment, ACM 000-0-00000-000-0/00/00.

The fundamental challenge in processing stream queries is that the input streams are potentially unbounded. Blocking operators need to produce output and the resource usage of stateful operators should not grow without bound under continuous input. Coping with these requirements requires information about stream progress, including progress of both input and inter-operator streams. The essential drawback of IOP is that it confuses a logical property, stream progress, with a physical stream property, stream order. IOP systems rely on the physical order of streams to implicitly provide information on stream progress. This confusion leads to extra burdens and added constraints on stream-system implementations, in addition to significant performance penalties. “Abnormalities” that arise naturally in stream systems, such as out-of-order tuples, highly selective predicates, and lulls, require special mechanisms in IOP systems because relying on stream order to communicate stream progress requires ordered streams that produce continual output. In addition, enforcing order on incoming and intermediate (inter-operator) streams significantly limits the implementation and optimization options in IOP systems; techniques requiring out-of-order processing [3][31] cannot be applied in IOP systems. Finally, IOP architectures do not scale well in distributed or parallel computing environments in which the input data of a query operator may come from nodes in different locations with heterogeneous computing power and transmission delays.

OOP is a new architecture that explicitly provides stream progress to operators and thus separates stream progress from physical stream-arrival properties. This separation allows more flexible implementations of query operators and leads to more efficient evaluation of stream queries. Figure 1 shows a comparison of the memory usage in Gigascope for OOP and IOP evaluations of a tumbling-window count query over the union of two input

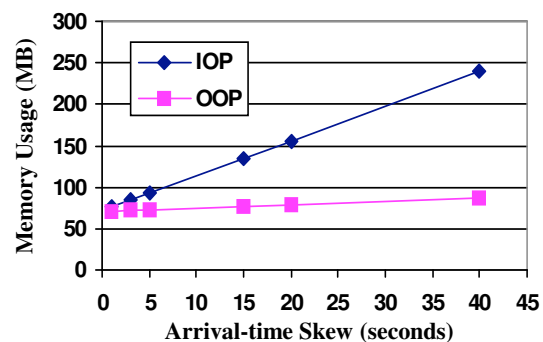


Figure 1. Memory usage for OOP and IOP on a window count query over the union of two streams, for varying skew.

streams, one of which arrives late. The rate of each input stream is 110,000 IP packets per second. The range of the tumbling window is 10 seconds. As Figure 1 shows, as the delay of the late stream increases from 1 second to 40 seconds, the memory usage for IOP evaluation increases significantly. The memory usage for OOP evaluation starts below that of IOP and grows slowly as skew increases. At a 40-second delay, OOP requires only 30% of the memory that IOP requires. OOP uses less memory because the IOP union must buffer the earlier stream to enforce order on the union’s output, while the OOP union can pass tuples through immediately, because its output goes to an order-agnostic window-aggregate operator.

Techniques exist for handling disorder on an operator level, including WID [16], which we proposed previously, and the window aggregation and join implementations proposed by Hwang et al. [11]. Both techniques require punctuation. A punctuation is a special tuple embedded in a data stream that indicates the end of a subset of the stream [29]. Also, Aurora uses slack to handle disorder [1]. Slack allows an operator to deduce stream progress based on stream arrival. We believe that disorder is best handled at a system level. Disorder may occur even when input streams are ordered due to time skew between input streams, operator-induced disorder [13] and multiple processing paths. In such cases, query operators may not be able to effectively infer stream progress themselves: The progress information may be most effectively provided by other operators—particularly operators at the edge of the query plan or operators that themselves induce disorder. In contrast to existing techniques, OOP handles disorder in the context of an entire query. OOP detects stream progress and requires query operators to propagate progress information throughout the query plan. Our OOP architecture can support high-volume, (near) real-time stream processing with high throughput and low memory usage, as demonstrated by our experimental results.

OOP is not difficult to implement and lends itself to incremental implementation. Based on our experience with Gigascope and NiagaraST, we argue that the implementation overhead of OOP is very low. Although OOP requires a mechanism to explicitly communicate stream progress, such mechanisms are already present in real-world stream systems, often for handling stream lulls. For example, Gigascope has punctuation-carrying heartbeats to unblock merge and to purge the state of join during stream lulls [17]; StreamBase supports heartbeats [27] to handle lulls; and NiagaraST [29] and CAPE [24] support punctuation to exploit data-stream semantics for unblocking operators and purging state. With such a mechanism in place, converting a stream system to OOP is not difficult. We extended Gigascope to support OOP with one person-month of effort.

Our contributions: We introduce the OOP architecture for stream systems. We first present a new data model for streams, and then discuss OOP implementation, including operator implementations, propagation of stream progress, and workload smoothing for processing high-volume data streams. We also demonstrate the benefits of OOP by performance experiments in two systems. We have implemented OOP in NiagaraST, an extension of the Niagara [21] net data management system to support stream processing, and in Gigascope [7], developed at AT&T to monitor traffic in their backbone networks. We find that with OOP, the memory overhead of aggregation queries under reasonable conditions is significantly reduced; that the throughput of such queries is sig-

nificantly increased, especially for high-volume data streams; and that the memory usage of join queries is reduced in certain conditions.

The rest of the paper is organized as follows. Section 2 presents an example comparing OOP to the existing stream query evaluation approaches. Section 3 presents our data model for streams and discusses detection and communication of stream progress. Section 4 discusses relevant prior work. Section 5 describes the OOP architecture, including punctuation generation and query operators in our OOP systems. Section 6 covers workload smoothing in high-performance OOP systems. Section 7 discusses fine-granularity punctuation. Experimental results are presented in Section 8 and we conclude in Section 9.

2. An Example

Before presenting the OOP architecture in detail, we briefly examine the evaluation of a query, Q1, based on a network-monitoring scenario described by the Gigascope team [17].

```
Q1: SELECT count (*)
      FROM Control union Main1 union Main2
      [RANGE 1 minute, SLIDE 1 minute, WA ts]
```

Q1 monitors streams of network packets arriving on three separate links and computes the number of packets received over a tumbling window of one minute defined on window attribute (WA) ts. Tuples from each stream have the same (simplified) schema of <srcIP, srcPort, destIP, destPort, len, ts>. Packets from each link arrive in order of the timestamp attribute ts. Control is a low-volume link; Main1 and Main2 are high-volume links, and might not be synchronized with respect to their timestamp attributes. We note that streams with widely varying volumes and delays are common in applications such as network-traffic monitoring, financial data processing, and intelligent transportation systems. Figure 2 shows a logical query plan for Q1. The essential requirement for Q1 is that the Window Count operator knows when it has received all tuples for each window.

We first describe two possible query evaluation plans for Q1 using previously-proposed techniques before presenting the OOP evaluation of Q1. Option 1: Union preserves order, and Window Count relies on an ordered input stream to determine when to close windows. Option 2: Union does not preserve order, and Window Count handles disorder using slack.

In Option 1, the logical Union operators are implemented with order-preserving *Merge* operators, which combine ordered inputs and guarantee an ordered output, but may require extra space and processing time. For example, during lulls on the Control link, the Merge operators have to buffer all tuples that arrive on the high-volume Main links. Also, if there is skew on timestamp between the links, due to, say, variable transmission delays, the Merge operators will have to buffer tuples to

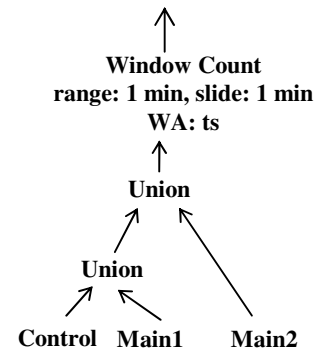


Figure 2. Query plan for query Q1

synchronize the links. The exact amount of buffer space that the Merge operators require depends on the arrival pattern of the input streams, the duration of lulls, the packet rates on the three links, and their timestamp skew, but there is no a priori upper bound. In addition to the memory overhead, buffering increases tuple latency.

Option 2 uses slack to cope with disorder. With slack, query operators accommodate tuples that arrive late during a grace period s , specified as a time interval or a number of tuples, and discard tuples that are delayed more than s . Although slack allows the aggregate operator to cope with disordered input, the aggregate operator still relies on the arrival order of the input stream to deduce stream progress. In this option, the Union operators do not need buffering, but the system must determine the slack parameter for Window Count. Unless the time skew of the input streams is known and fixed, it is difficult, if not impossible, to set slack for the Window Count operator so that it precisely captures the disorder of the unioned input streams. Tuples will be dropped if the slack parameter s is too small, while a latency penalty will be incurred if s is too large.

In contrast, with the OOP approach, query operators do not need to enforce order nor deduce stream progress on an operator-by-operator basis. Rather, we need to conjure up and pass on appropriate progress indicators on ts such as punctuations. Consider an OOP evaluation of Q1 using WID [16]: Union passes tuples through without preserving order, and produces punctuation for use by the Window Count operator based on input-stream punctuations. Inserting punctuation into the input streams is easy, as they are ordered. WID maintains a partial aggregate for each window and relies on punctuation to close windows. The only state that the OOP approach needs to maintain for Q1 is the partial aggregates, thus the required space is much less than for Option 1. Compared to Option 2, no tuples will be dropped and latency will be minimal, without any a priori bounds on stream skew.

Consider now a slightly more complex query in which the Union below the Window Count in Q1 is replaced with a Band Join. Band Join naturally produces a disordered output stream [13], so the slack parameter needed by Window Count is related to both input time skew and join processing. To our knowledge, no one has presented a comprehensive method for calculating the appropriate slack parameter on an operator’s output stream from the slack of its input streams. The Gigascope team previously tried using slack to propagate stream progress, but switched to punctuation, as they found propagating slack through complex queries to be complicated and error prone. In OOP, the input operators can deduce progress information from the ordered input streams and insert that progress information into the streams (as punctuation in our implementation); Join receives that progress information and propagates it at the appropriate time to the next operator—Window Count in this example. Join may use knowledge of its state and implementation to determine when to propagate punctuation. We argue that a global mechanism for detecting and propagating stream progress is required. The OOP architecture proposed in this paper is built upon such a global mechanism, thus freeing query operators from maintaining stream order and avoiding the associated cost.

3. STREAM PROGRESS

In this section, we formally define a stream model, *progressing streams*, for OOP stream systems, and discuss mechanisms for detecting and propagating stream progress.

3.1 The Progressing-Stream Model

Previous work on data streams commonly models a stream as an unbounded sequence of data items arriving in order of some time-stamp-like attribute. However, disorder naturally occurs in real-world stream systems. A few examples:

1. Items arriving over a network from a remote origin may have taken different paths with different delays.
2. In a parallel or distributed system, a data stream may be a combination of several sub-streams from different nodes. The merged stream can be disordered if there are different processing or transmission delays associated with those nodes.
3. Some data streams have multiple timestamp attributes with different orders. For example, NetFlow [22] records from a router might arrive in order of “flow end” time, but are disordered on “flow start” time. Some queries may window on “flow end” and others on “flow start.”
4. Even when data streams arrive in order, some query operators, such as band join, can introduce disorder in intermediate results.

Obviously, one can try to reorder a stream on the relevant attribute with some kind of buffered sort operator (such as BSort [1]). However, it is often hard to obtain a priori time or space bounds on disorder. The key observation behind the OOP approach is that total order on an attribute is not required to unblock blocking operators and purge state from stateful operators. Rather, we only need the weaker notion that there is “progress” on some attribute A : The value of A in a stream always eventually exceeds any fixed value v .

To make this notion more precise, we first define the *low-water mark* (lwm) for an attribute A of stream S on every prefix S_n of length n :

$$lwm(n, S, A) = \min\{t.A \mid t \in S - S_n\} \quad (\text{Eq. 1})$$

That is, $lwm(n, S, A)$ is the smallest value for A that occurs after prefix S_n of stream S .

Definition: Stream S is *progressing on attribute* A if for every value v in the domain of A , there exist an n such that $lwm(n, S, A) > v$. When this condition holds, we say A is a *progressing attribute* for S , and that S is a *progressing stream*.

The crux of our approach is observing that any operator that can be unblocked and purged using an ordered attribute can also be handled with a progressing attribute, as long as we can detect and communicate stream progress. Note that progressing attributes exist in Examples 1-4 above. (In Example 3, both timestamp attributes are progressing)

3.2 Detecting and Propagating Stream Progress

Both IOP and OOP systems need to detect the progress of input streams. IOP needs to detect progress to enforce stream order, and

OOP needs to detect progress to bound the low-water mark. Any information that IOP systems use to enforce order on input streams can be used in OOP systems to bound the progress of input streams. Examples of such information include knowledge that an input stream is ordered, or limitations on the amount of delay expected or allowed (e.g., slack [1]), or time skew and transmission delay of data sources, as used by Widom et al. [25] in heartbeat generation.

A key difference between IOP and OOP architectures is how they communicate stream progress through query plan. In IOP systems, every query operator needs to rely on the order of its input stream(s) to deduce stream progress. In OOP systems, each query operator must produce progress information for its result stream so that all query operators receive explicit progress information and do not have to deduce stream progress from observations of their input stream(s). Punctuation is used by both NiagaraST and Gigascope to conveniently express and propagate stream progress. Punctuation is a general mechanism that has been proposed to help unblock blocking operators and purge state from stateful operators over data streams [8][9][14]. In this paper, we assume *linear punctuation*. Linear punctuation is defined on the progressing attribute, and the punctuating values are monotonic. For example, the punctuation $p(*, *, *, *, *, 12:00:00AM)$ indicates the current stream low-water mark is at least 12:00:00AM, which means all packets with ts attribute value smaller than 12:00:00AM have arrived.

Discussion: Although we choose a data-driven mechanism, punctuation, to propagate stream progress, OOP can also work with other non-data-driven mechanisms, such as operators periodically polling their input operators for progress bounds, or having a global scheduler track operator progress.

4. PRIOR WORK

We review implementations of stream aggregate and join operators previously proposed in the literature.

4.1 Window Aggregation Implementations

Many implementations of windowed aggregation rely on ordered input to determine the completion of both tumbling (disjoint) and sliding (overlapping) windows. When processing a window, tumbling-window aggregation simply maintains a partial aggregate. A common implementation of sliding-window aggregation, which we term the *buffered implementation*, will buffer each tuple until it does not belong to any future window. At the completion of a window, the aggregate is computed over the buffered tuples and then the expired tuples are purged from the buffer. WID [16] is an order-agnostic implementation of windowed aggregation. It assumes that punctuation signals the completion of windows. WID uses a *Bucket* operator to tag each input tuple with the set of window-ids for windows to which the tuple belongs. The aggregate operator then uses window-id as an additional grouping attribute and incrementally maintains partial aggregates for each group in a hash table. When a punctuation arrives, the aggregate operator outputs aggregates matching the punctuation and purges them from the hash table. Hwang et al. [11] describe punctuation-assuming implementations of window aggregation, which are used to achieve replication transparency for high-availability stream systems. Our OOP architecture aims to support high-volume, (near) real-time stream processing such as required

by Gigascope, which supports multi-gigabit-per-second stream rates. The order-agnostic operators of Hwang et al. [11] are relatively heavy weight, and are designed for latency reduction in a low-throughput system. This difference is reflected in the experiments, as the implementation of Hwang et al. processes tens of packets per second, while one of our implementations processes in excess of 800,000 packets per second.

Sub-aggregation techniques can improve the evaluation of sliding-window aggregation, such as our paned-window aggregation technique [15] and sub-aggregation techniques for shared execution of multiple queries proposed by Arasu and Widom [4] and Krishnamurthy et al. [20].

4.2 Sliding-Window Join Implementations

Sliding-window join is discussed extensively in the literature. Q2 below is an example of a sliding-window join, defined on an attribute, ts , with a 3-minute window on the first input and a 2-minute window on the second input. This join specifies that a tuple, t , from the first input, joins with tuples with ts value greater than $(t.ts - 2 \text{ min})$ and smaller than $(t.ts + 3 \text{ min})$ from the second input, when the IP addresses match.

```
Q2: SELECT *
     FROM Main1 [WA  $ts$ , RANGE 3 min],
          Main2 [WA  $ts$ , RANGE 2 min]
     WHERE Main1.srcIP = Main2.destIP;
```

Many previous implementations of sliding-window join assume that windows are defined on arrival time [9][10][13][18][26], or that the join’s input streams arrive in order and are synchronized on a shared timestamp attribute. Such implementations may deliver incorrect results when these assumptions are not met. Observe that these assumptions imply that tuples from both streams have a “global order” – the timestamp of a new tuple is guaranteed to be no smaller than the timestamps of tuples that have already arrived on both input streams. Based on this assumption, a window join implementation needs to store only tuples in the latest window of each input stream. When a new tuple arrives, join can purge state based on the timestamp of the new tuple. For example, when the Join operator of Q2 receives a new tuple, t , from Main1, it purges Main2 tuples with ts value smaller than $t.ts - 2 \text{ min}$. Then, t is compared with stored tuples of Main2, and composite tuples of t and the matching tuples are produced, and then t is stored. Hammad et al. [13] propose sliding-window implementations that support ordered input streams, but with potential arrival-time skew and analyze the implementation’s average response. Hwang et al. [11] describe a punctuation-assuming window join implementation. Ding et al. [8][9] propose join algorithms leveraging punctuation on data attributes (instead of the progressing attribute) to purge state more efficiently.

5. OOP ARCHITECTURE

We have implemented the OOP architecture twice, once starting from Gigascope and once from Niagara. We added OOP to an existing version of Gigascope, which is an operational IOP system. With Niagara, we extended the publicly-available version of Niagara to an OOP stream processing system, called NiagaraST. In this section, we present the OOP architecture.

5.1 Punctuation Generation

Gigascoppe generates punctuation in a timer-driven fashion [17]. In a low-level sub-query, a timer callback function fires every second (in wall-clock time), and a punctuation that carries a progressing attribute value indicating the stream low-water mark is inserted into the input stream. This mechanism assumes that Gigascoppe’s input streams are ordered. Query operators propagate punctuation. Every time an operator receives punctuation from its input stream, it produces a punctuation for its output stream. Timer-driven punctuation also serves to detect operator failure and monitor query latency.

NiagaraST generates punctuation in a data-driven fashion. In the absence of external punctuation provided by a data source, NiagaraST inserts punctuation into the data stream based on observation of the progressing attribute and data stream semantics as discussed in Section 3.2. As a simple example, if a data stream is known to be ordered, NiagaraST inserts punctuation when it observes that the value of the progressing attribute has changed by a predefined amount.

5.2 Aggregation

We briefly summarize the aggregation semantics allowed by stream systems, and then discuss aggregation implementation in OOP systems and the benefits of OOP for aggregate queries. See detailed algorithm in the appendix.

Aggregation Semantics: Stream systems often place restrictions on the types of aggregation queries allowed to ensure that queries can progress. Stream systems may allow only aggregations that can potentially be unblocked. Informally, this condition translates to the requirement that each group in an aggregation must eventually be complete, even though the input stream is unbounded. We formalize this restriction as follows: Aggregations in stream queries must have a grouping condition that includes a progressing attribute. Window aggregation is an aggregation with a special grouping condition on the progressing attribute that maps each tuple to one or more groups, which ensures the condition above is satisfied.

Order-agnostic Aggregation Implementations: WID is an order-agnostic implementation for both time-based (e.g., a window of “5 seconds”) and row-based (e.g., a window of “100 tuples”) window aggregation. For a row-based window, unlike a time-based window, it makes a difference if the 100 tuples are counted on the original input stream or on the stream presented to the aggregate operator. The latter stream could have fewer items and they could be in a different order. In NiagaraST, the system tags each input tuple explicitly with its sequence number (*seq-num*) as presented to the system, and inserts punctuation on the *seq-num* attribute of input tuples. WID uses *seq-num* as the progressing attribute for row-based window aggregation. (We believe that defining row-based windows by the number of tuples presented at the operator, which some stream systems seem to implement, is problematic, because it can give different answers for different query executions.) The aggregate operator produces a punctuation for each window after it outputs all groups in the window. Hwang et al. [11] also describe an order-agnostic, punctuation-assuming implementation of window aggregation for time-based window aggregation, but their implementation requires ordered input for row-based window aggregation.

Benefits of OOP for Aggregate Queries: Even for queries with a single, ordered input stream, disorder may occur in intermediate streams. For example, if an input stream is split and processed through different sub-queries (such as might be needed for network-protocol simulation), the union of the sub-query results may be disordered due to different sub-query processing delays. Figure 3 shows such an example: The input is split according to an inexpensive predicate A; tuples not satisfying A are put through an expensive predicate B before being merged with the stream of tuples satisfying A; the result is fed to a window count aggregate.

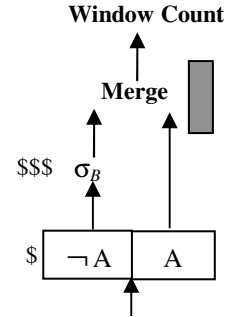


Figure 3. Enforcing order on intermediate results

The non-OOP alternatives for this query are similar to those in Section 2. Either the Merge operator can enforce order and pay the associated memory and latency costs, or the Window Count operator can use slack. In contrast, in OOP, stream progress information (punctuation) is inserted into the streams as they arrive and is propagated through the selections and Merge, thus providing Window Count precise progress information. In this example, using slack amounts to trying to infer information that the system already knows.

In OOP, Merge is implemented as a non-blocking, non-buffering union. In addition, tuples are incrementally reduced into partial aggregates by the Window Count operator. In general, maintaining partial aggregates is much less expensive than buffering tuples and keeps tuple-processing delay minimal. Figure 1 in the Introduction and Figure 13 in Section 8 show the memory benefit of OOP for aggregation queries under similar scenarios.

5.3 Join

We first discuss the semantics of join in our OOP systems, and then present our stream-join algorithm, and discuss the benefits of OOP for join queries.

Join Semantics: Stream systems allow only joins whose state cannot grow indefinitely. The join condition must ensure that a tuple of one input only joins with a bounded range of tuples from the other input. Specifically, a join in a stream query must have an equality or band predicate between progressing attributes of its two inputs. Placing previously proposed join implementations in this context, we note that tumbling-window join is a join with an equality predicate on (a function of) progressing attributes and sliding-window join can be seen as an alternative way of expressing a band join.

Stream-join implementations select the timestamp attribute(s) in the output in various ways. Each input stream has (at least) one timestamp attribute, which we call $S_0.ts$, and $S_1.ts$ for input streams S_0 and S_1 , respectively. Some implementations specify that one or the other of $S_0.ts$ and $S_1.ts$ be the timestamp of join result; other implementations use $\max(S_0.ts, S_1.ts)$. A traditional relational join would include both timestamps in the result, which is our semantics. Both $S_0.ts$, and $S_1.ts$ are attributes of the join result and subsequent operators can use $S_0.ts$, $S_1.ts$, both $S_0.ts$ and $S_1.ts$ (e.g., the pair $(S_0.ts, S_1.ts)$), or a function of $S_0.ts$ and $S_1.ts$

(e.g., $\max(S_0.ts, S_1.ts)$ or $\min(S_0.ts, S_1.ts)$) as a progressing attribute. Since we need not enforce any particular order, we can allow subsequent operators the flexibility in selecting a progressing attribute; as we discuss later, we can tailor punctuation to the particular choice of progressing attribute.

Order-Agnostic Join Implementation: We propose an order-agnostic join algorithm, *OOP-Join*, with a band predicate on the progressing attribute. OOP-Join places no restrictions on the order or synchronization of its inputs, but assumes punctuation on the progressing attribute(s). Figure 4 shows the algorithm. The input streams are S_0 and S_1 , the progressing attribute is ts for both, and the band predicate is $(S_0.ts - \text{RANGE}_0) \leq S_1.ts \leq (S_0.ts + \text{RANGE}_1)$. For ease of presentation, we ignore join predicates on other data attributes.

```

State Maintained:
 $b_0, b_1$ : bounds on the low-water mark of left and right input, respectively; initialized to  $-\infty$ ;
 $M_0, M_1$ : tuple sets maintained on left and right input, respectively; initialized to  $\emptyset$ ;

Join( $x$ )
let  $S_i$  be the input stream to which  $x$  belongs;
if  $x$  is a tuple ProcessTuple( $x, S_i$ );
else if  $x$  is a punctuation ProcessPunctuation( $x, S_i$ );

ProcessTuple( $t, S_i$ )
join  $t$  with matching tuples in  $M_{1-i}$ ;
if  $t.ts \geq b_{1-i} - \text{RANGE}_i$ 
  add  $t$  to  $M_i$ ;

ProcessPunctuation( $p, S_i$ )
 $b_i = p.ts$ ;
 $\forall k$  in  $M_{1-i}$ 
  if  $k.ts < p.ts - \text{RANGE}_{1-i}$ 
    purge  $k$ ;
ProducePunctuation( $p, S_i$ );

ProducePunctuation( $p, S_i$ )
output a punctuation for  $S_{1-i}.ts$  with value  $\min(b_i - \text{RANGE}_{1-i}, b_{1-i})$ ;
output a punctuation for  $S_i.ts$  with value  $\min(b_{1-i} - \text{RANGE}_i, b_i)$ ;

```

Figure 4: OOP-Join.

Note that in this algorithm, new tuples do not always need to be stored: As the ProcessTuple() function shows, if the ts value of a new tuple is smaller than the low-water mark bound of the other

input minus RANGE, it can be processed on the fly and discarded, because all the tuples with which it needs to join have already arrived. The amount of state that join needs to maintain depends on input stream progress. In general, the progress of the left input indicates which tuples from the right input can be purged, and vice versa. The algorithm for join with an equality predicate on progressing attributes is similar, but simpler.

As shown in the ProducePunctuation() function, OOP-Join produces distinct punctuation for $S_0.ts$ and $S_1.ts$, which we term *individual punctuation*. Individual punctuation indicates the progress of the join result on either $S_0.ts$ or $S_1.ts$, and allows a subsequent operator to deduce stream progress even if its progressing attribute involves both $S_0.ts$ and $S_1.ts$, or a function of them. For example, if the operator's progressing attribute is $\max(S_0.ts, S_1.ts)$, the subsequent operator can progress to s when it receives punctuation for s from both S_0 and S_1 . However, as we will explain in Section 7, providing the progress of the join result on $(S_0.ts, S_1.ts)$ pairs may allow subsequent operators to produce results sooner.

Benefits of OOP for Join queries: In OOP systems, join operators may often have a smaller footprint and are able to produce results with less delay. In the following we elaborate the benefits of OOP-Join in more detail.

In OOP systems, late tuples do not delay the processing of “on time” tuples. In particular, join may process and then purge on-time tuples at the earliest possible moment, thus reducing latency and memory usage. Consider a join query with a band predicate, $S_0.ts - 2 \leq S_1.ts \leq S_0.ts + 2$. Assume S_0 and S_1 are approximately synchronized, which means that—ignoring delayed tuples—tuples from S_0 and S_1 with the same ts value arrive at about the same time. Assume that input stream S_0 may potentially contain a small fraction of tuples that are delayed by at most 5 minutes, and input stream S_1 arrives ordered. Figures 5(a) and 5(b) show the IOP and OOP evaluations of the band-join query. With IOP, due to potentially delayed tuples in S_0 , the Sort operator needs to buffer up to 5 minutes of S_0 tuples, and the join maintains 7 minutes of S_1 tuples (2 minutes due to the band predicate and 5 minutes due to the delayed S_0 tuples). Note that the Join operator can process S_0 tuples on the fly and does not need to maintain any state for S_0 . (S_0 tuples arrive 5 minutes behind S_1 tuples at the join and hence all matching S_1 tuples are available when each S_0 tuple arrives.) With OOP, there is no sort operator, and tuples from S_0 are maintained by the Join operator. In Figure 5(b), the join maintains 5 minutes of S_1 tuples, similar to IOP, but needs only maintain 2 minutes of S_0 tuples, because most S_1 tuples arrive on time and thus punctuation from S_1 can purge S_0 tuples regularly without

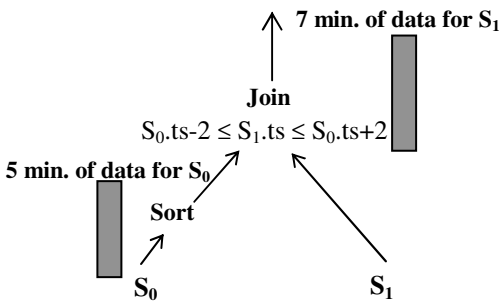


Figure 5(a). IOP evaluation of a band join (maximum allowed delay in S_0 is 5 min-

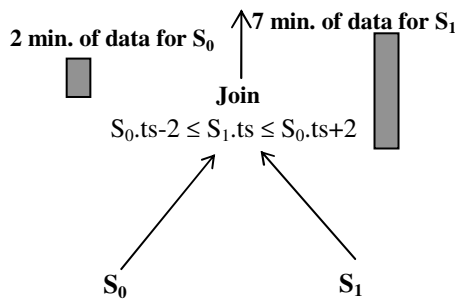


Figure 5(b). OOP evaluation of a band join (maximum allowed delay in S_0 is 5 min-

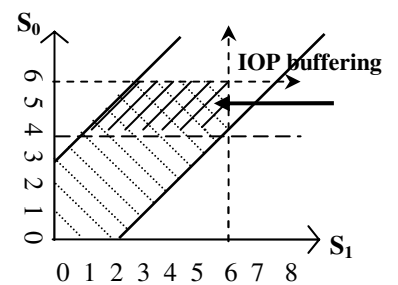


Figure 6. Output buffering in IOP band join (output ordered on $S_0.ts$)

delay. Overall, the OOP evaluation of the join query maintains 3 minutes fewer of S_0 tuples, and produces most join results earlier than the IOP evaluation. Figure 12 in Section 8 shows the memory benefits of OOP in this case.

In OOP systems, band join does not need to enforce order on its result, and thus has a reduced memory footprint and lower latency than an IOP join that must enforce output order. The double cross-hatched area in Figure 6 illustrates the amount of buffering required to order the output of an IOP band join with a band predicate $S_0.ts - 3 \leq S_1.ts \leq S_0.ts + 2$ on $S_0.ts$, assuming that input streams S_0 and S_1 are approximately synchronized, and assuming the join result needs to be ordered on $S_0.ts$. As the figure shows, when both S_0 and S_1 progress to time 6, the join needs to buffer results produced by S_0 tuples with ts values between 4 and 6. In general, the required buffering for ordering join output in IOP systems increases with the band size of the join predicate. (The exact amount of buffering is determined by the desired output order, the band predicate, the data rate of the input streams, and the arrival-time skew of the input streams.) In OOP systems, join results can be released on the fly, without any delay or buffering, and processed immediately by a subsequent operator. Figure 11 in Section 8 shows the benefit of OOP in terms of memory, latency and CPU for this case.

5.4 Other Operators

We briefly summarize the implementation of other query operators in our OOP systems. Except for the *Input* operator, query operators in OOP assume their input streams are punctuated; they do not need to enforce or maintain order, but they do need to propagate punctuation.

Input: Input operators in OOP systems need to periodically insert punctuation on the progressing attribute into the input stream. In contrast, input operators in IOP systems might buffer and sort the input stream. Ordering a potentially disordered input stream and inserting punctuation into it require the same knowledge about input stream arrival properties.

Select, Apply, Project: The basic pipelined implementations of Select, Project and Apply work for both the IOP and OOP approaches. The processing of punctuations in the OOP implementations is similar to tuple processing.

Dupelim: Duplicate elimination (Dupelim) naturally preserves order; the issue is when state can be purged. The IOP implementation of Dupelim can remove state whenever the ordering attribute advances. The OOP implementation of Dupelim must rely on punctuation to purge its state; punctuation is passed through as received.

Union: In an IOP system, Union must buffer one input during a lull or delay on the other input to assure ordered output. In OOP systems, the (bag) Union operator can pass through input tuples immediately and needs to maintain no tuple state. To produce output punctuation, the Union operator in OOP systems needs to remember input punctuation from each input stream. Assuming linear punctuation, it only needs to remember the last punctuation on each input. When Union receives a punctuation, p , from one input, it produces a punctuation with value $\min(p.ts, p'.ts)$, where p' is the last punctuation on the other input stream. See the detailed algorithm in appendix.

5.5 Discussion

OOP is a more scalable architecture, especially in distributed or parallel environments, where the input data for a query operator may come from different processors, or even different machines far removed from one another. An issue with IOP in such an environment is that operators can block due to network congestion and routing problems at a single node. For example, a TCP connection might break and need to be re-instantiated. These network problems can cause a significant delay and even hang an IOP system. In addition, even when the network is reliable, enforcing order on data coming from multiple processors may incur prohibitive memory and latency costs.

OOP is also a more permissive architecture. It can accommodate operator implementations that require out-of-order processing. For example, to improve throughput, stream systems may want to process tuples out of order. Avnur and Hellerstein propose an adaptive query processing mechanism, called Eddies, that dynamically routes tuples to query operators based on operator load [3]. To improve interactive query performance, Franklin et al. [23][31] propose algorithms that re-order tuples based on their importance. Also, OOP can utilize order-sensitive operator implementations, by first sorting the input. (Punctuation can unblock sort.) Further, OOP enables interesting optimizations. In traditional database systems, one logical operator may have multiple physical implementations and the system may choose among them based on the properties of input relations, OOP systems can similarly choose among different physical implementations of logical query operators based on properties of input streams. For example, for window aggregation, out-of-order tuples delay the completion of windows in which they participate, and thus increase the number of “open windows” in the operator state. Thus, in situations where the number of tuples per window is small and the size of partial aggregates is large, the memory cost of WID may exceed that of the buffered implementation. In such situations, a buffered implementation that processes windows sequentially may be preferable, although it incurs more delay for enforcing tuple order and computing aggregates.

6. WORKLOAD SMOOTHING

Workload smoothing is critical for systems dealing with high-volume streams in (near) real time. For such systems, workload bursts may overload the system, delay data processing, and lead to loss of input data or obsolete query results. An important reason for us to consider using OOP in Gigascope is that OOP is useful for workload smoothing. Workload bursts can occur either on input or intermediate streams, caused either by input data bursts or blocking operators that are periodically unblocked, respectively. For example, when a window ends, window aggregation needs to scan the hash table of partial aggregates to produce results and purge completed items, and outer join needs to locate and output tuples that were not matched. Here we focus on smoothing intermediate workload bursts created by the unblocking of blocking operators.

In the following, we first present a workload smoothing mechanism, *slow-flush*, originally implemented in the IOP version of Gigascope [17]. Similar workload-smoothing mechanisms are also used in other network traffic monitoring systems [19]. We then discuss the issues with IOP in applying slow-flush and workload smoothing in our implementation of OOP in Gigascope. We start

with a short review of window aggregation implementation in Gigascope.

Aggregation in Gigascope: Gigascope has a two-level architecture typical of high-performance, potentially distributed, data-monitoring systems [7], where the low level is used for data reduction and must be lightweight, and the high level handles more complex processing. A low-level sub-query processes network packets from a fixed-size ring buffer. Low-level and high-level queries may run in different processes or even on different machines. Gigascope supports only tumbling-window aggregation natively. An aggregation query is split into a low-level sub-aggregation and a high-level aggregation that rolls up sub-aggregates. To ensure the low-level sub-aggregation is fast, it uses a fixed-size hash table to maintain aggregates for different groups. On hash-table collision, the existing aggregate in the hash table is output to accommodate the new aggregate. At the end of a window, the low-level query flushes the hash table and outputs all aggregates in it. However, if the number of groups is large, flushing the hash table causes a workload burst, during which time the ring buffer can overflow and lose packets.

Slow-flush mechanism: Gigascope uses slow-flush to smooth workload bursts at window boundaries in low-level aggregation. With slow-flush, when a window completes, the low-level sub-

```

State:
ht: a fixed-size hash table of (partial) aggregates;
status: a table indicating the content for each hash table
entry: new, old, or empty;

Init:
flush_finished = true;
flush_pos = 0;
Init entries of status as empty;

SlowFlush():
if (status[flush_pos] == old)
    output the existing aggregate in ht[flush_pos];
    status[flush_pos] = empty;
    flush_pos++;
    if (flush_pos > ht.size)
        flush_finish = true;

ProcessTuple(t):
if t indicates the start of a new window
    if (!flush_finish)
        flush entries in ht marked old; mark them empty;
        flush_finish = false;
        flush_pos = 0;
    if (!flush_finish)
        SlowFlush();
    key = hashkey (t);
    if status[key] == empty or old
        if status[key] == old
            output the existing aggregate in ht[key];
            init an aggregate with t in ht[key]; mark it new;
    if status[key] == new
        if t belongs to the existing group
            update the existing aggregate with t
        else
            flush entries in ht marked as old;
            output the existing aggregate in ht[key];
            init an aggregate with t in ht[key]; mark it new;

```

Figure 7: Low-level aggregation with slow-flush.

query gradually outputs aggregates of the previous window while processing new packets, instead of flushing all aggregates from the hash table at once. Figure 7 outlines Gigascope’s IOP implementation of slow flush.

The status table indicates the content of each hash entry—whether a hash entry is empty, contains a partial aggregate for the new window, or a completed aggregate for an old window. As the *ProcessTuple* function shows, on hash-table collision, if the existing aggregate belongs to an old window, it is output and the slot is used for the new aggregate. However, a problem occurs if the existing aggregate belongs to the new window. Because low-level aggregation must preserve output order, it has to first flush all the aggregates of old windows before it can output the existing colliding aggregate¹. Therefore, because slow-flush must satisfy the order requirement, it may not effectively smooth out the output of the low-level aggregates, especially when the number of groups is large. Hash-table flushing creates a burst during which incoming tuples cannot be processed, limiting the maximum stream rate supported by IOP, as discussed in Section 8.1 and illustrated in Figure 9.

Workload Smoothing in OOP: OOP Gigascope has two ways to smooth workload, slow-flush and *lazy-flush*. In contrast to IOP with slow-flush, OOP (with either *lazy-flush* or *slow-flush*) may permit much higher throughput. The most important benefit of OOP in terms of workload smoothing is that, as it has no order requirement, the low-level aggregation does not need to flush all partial aggregates from the previous window when collision of two aggregates from the new window occurs. In general, slow-flush intentionally increases result latency to smooth out the workload. However, the latency that IOP can use to smooth out the workload is very limited, due to its order-maintenance requirement. In contrast, OOP systems can improve workload smoothing and thus achieve better throughput by allowing more delay (as long as the desired upper bound on latency is guaranteed). In detail, suppose the desired maximum latency is m windows. OOP can address workload smoothing in two ways. First, it may use *lazy-flush*, which simply relies on hash-table collisions to naturally flush old aggregates, but with a check that aggregates are flushed with a maximum delay of m windows. Alternatively, OOP can explicitly use *slow-flush*. OOP with *slow-flush* outputs one old aggregate every i new packets, and guarantees a maximum result delay of m windows. Both i and m are tunable parameters of the low-level sub-aggregation. As we show in Section 8.1 (Figure 9), both OOP with *lazy-flush* and OOP with *slow-flush* achieve better throughput than IOP with *slow-flush* when there is a large number of groups.

7. FINER-GRANULARITY PUNCTUATION

We introduce a new type of punctuation—*joint punctuation*—which may reduce delay in operators consuming join output. Consider query, Q3, which counts established TCP connections per time period from link S_0 to link S_1 . SYN packets from S_0 are matched with SYN_ACK packets from S_1 and the result is grouped on the timestamps from S_0 and S_1 . The result is connec-

¹ The working Gigascope actually uses a better replacement policy—if the existing aggregate belongs to the new window, *ProcessTuple* also checks the next hash entry to see whether it can accommodate the new aggregate without flushing all old aggregates.

tion counts for timestamp pairs ($S_0.ts$, $S_1.ts$). For the purpose of discussion, we assume timestamps are rounded to the nearest second. To limit the space required by join and reduce spurious matches, a band predicate that limits the difference between $S_0.ts$ and $S_1.ts$ is added to the join.

```

Q3: SELECT S0.ts, S1.ts, count(*)
      FROM S0 [WA ts, RANGE 2 min],
           S1 [WA ts, RANGE 2 min]
      WHERE S0.destIP = S1.srcIP AND S0.destPort = S1.srcPort
            AND S0.flag = SYN AND S1.flag = SYN_ACK
            AND S0.ts < S1.ts
      GROUP BY S0.ts, S1.ts;

```

Q3 can be evaluated with a band join that feeds a count operator that groups on $S_0.ts$ and $S_1.ts$. We discuss the effects of different types of punctuation on the output of count. Consider the aggregate group in the Count operator with $S_1.ts=1$ and $S_0.ts=0$. Count can output the result for this group when it knows one of two things: all tuples with $S_1.ts=1$ have been received; or all tuples with $S_1.ts=1$ and $S_0.ts=0$ have been received. The time at which count outputs the result for this group is directly dependent on what punctuation join produces and when. More specifically, count can output this group when it receives either of the following two punctuations: $(*,1)$ or $(2,1)$. Recall that we call punctuation such as $(*,1)$ *individual punctuation* as it punctuates only one of the two ts attributes; we call punctuation of the second form $(2,1)$ *joint punctuation* as it punctuates $S_0.ts$ and $S_1.ts$ together.

Join derives its output punctuation based on its predicate and input punctuation state. In this example, the individual punctuation is dependent on the band predicate; $(*,1)$ can be produced when input punctuation on $S_1.ts$ with value (3) is received (the band is 2 minutes long). However, the joint punctuation $(2,1)$ can be produced when input punctuation (2) on $S_1.ts$ and (1) on $S_2.ts$ are both received. Joint punctuation is independent of band size, and therefore can be produced earlier than individual punctuation. This difference may be significant for joins where band size is large relative to timestamp step, as may be the case in Q3. Figures 8(a) and (b) illustrate the progress information communicated by joint and individual punctuation. In Figure 8, the x- and y-axes are labeled with punctuations on ts values of S_0 and S_1 , respectively; the solid lines indicate the region of timestamps that satisfy the band predicate. Number pairs represent output punctuation and dotted lines outline the coverage of each output punctuation. Observe that joint punctuations have smaller coverage, but can be output sooner.

8. EXPERIMENTAL EVALUATION

In this section, we present an experimental study of our OOP implementations in both Gigascope and NiagaraST. Gigascope focuses on processing high-speed network-traffic streams, and NiagaraST focuses on flexibility and expressiveness of stream queries. As discussed before, we converted a version of Gigascope to OOP and extended the publicly-available version of Niagara into a full-fledged OOP stream engine; we call the converted systems OOP-Gigascope and NiagaraST, respectively.

8.1 OOP with Gigascope

The experiments with Gigascope were conducted using network feeds generated by the RouterTester® traffic generator. Our focus is to show the memory and throughput benefits of OOP over high-

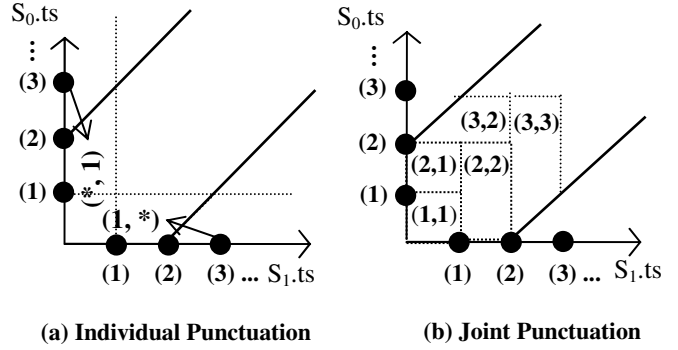


Figure 8. Individual vs. joint punctuation – for band join

speed streams. All experiments were conducted on a dual-processor dual-core Intel(R) Xeon(TM) CPU 2.80GHz processor with 4 GB of RAM running Linux 2.4.21.

Experiment G1: This experiment shows how OOP can improve throughput during workload bursts, and uses the following query, Q4, which computes the number of packets from a network interface for each (srcIP, destIP) pair for every minute.

```

Q4: SELECT srcIP, destIP, count(*)
      FROM M [WA ts, RANGE 1 min, SLIDE 1 min]
      GROUP BY srcIP, destIP

```

We executed Q4 with Gigascope and OOP-Gigascope, varying the number of groups in the stream and the size of the hash table used by the low-level sub-aggregation. In addition, we experimented with two OOP implementations of the low-level sub-aggregation—slow-flush (sf) and lazy-flush. We measured the maximum stream rate that Gigascope and OOP-Gigascope could support, by increasing the stream rate until tuples were dropped.

The number of groups was varied from 66k to 520k; the low-level hash-table size was dependent on the number of groups. For each case, we used three hash table sizes: half, equal to, and twice the number of groups. As discussed in Section 6, OOP may use either lazy-flush or slow-flush to improve workload smoothing and thereby throughput. In this experiment, both OOP-Gigascope with lazy-flush and slow-flush allow an extra delay of two windows to spread the workload across window boundaries. Further, OOP-Gigascope with slow-flush explicitly flushes an aggregate for an old window every 160 incoming packets. In contrast, Gigascope uses an aggressive slow-flush, explicitly flushing an aggregate once per incoming packet.

Figure 9 shows the results of this experiment. Therein, OOP and OOP(sf) represent the OOP implementations of low-level sub-aggregation without slow-flush (lazy-flush) and with slow-flush, respectively. Values 1/2, 1, and 2 indicate the relative size of the hash table in the sub-aggregation. In addition to measuring the maximum supported data rate, we also measured CPU utilization. In general, when the number of groups is small, for example, at 66k, the stream rates that IOP and OOP can support are about the same; and the CPU utilizations are close to saturation. However, when the number of groups is large, with a reasonable hash table size, OOP can support a much higher stream rate than IOP. For example, at 260k groups, with 540k hash table entries, OOP and OOP(sf) can support 760k pkts/sec and 800k pkts/sec, respectively, while IOP can only support 400k pkts/sec. However, an overly large hash table may adversely affect the throughput of the

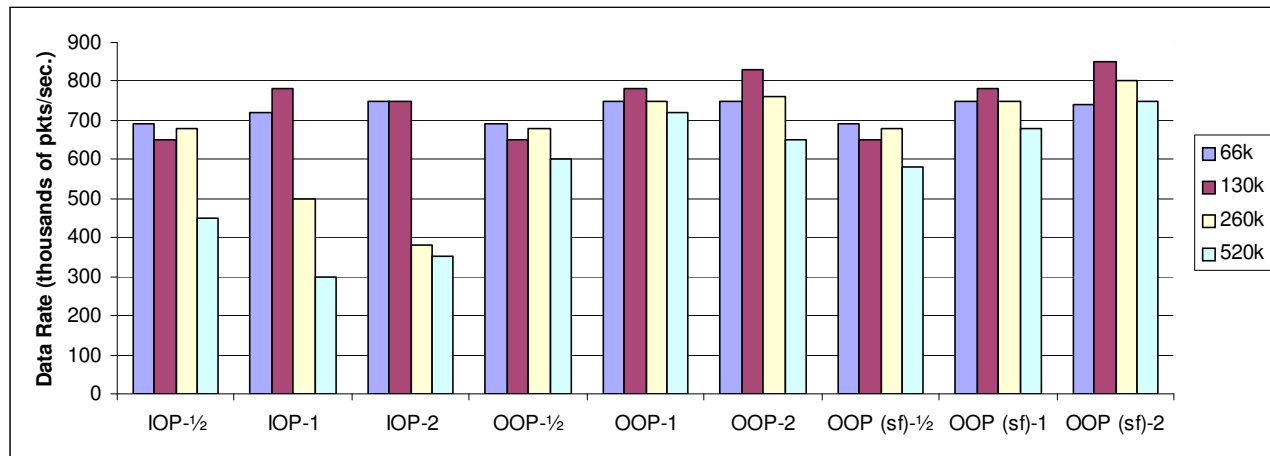


Figure 9. Throughput comparison of IOP and OOP for a count query, Q4, in GigaScope

query, especially for the IOP cases. With 520k groups, IOP-2 only supports a maximum data rate of 350,000 packets per second, with CPU utilization of 50% and IOP-1 supports only 300,000 packets per second with CPU utilization of 51%. As discussed in Section 6, when there is a hash-table collision between an incoming packet and an existing aggregate from the current window, IOP needs to flush all aggregates from the previous window before any new tuples can be processed. During the hash table flush, new packets in the ring buffer are not processed and packets will be dropped if the ring buffer fills. When the number of groups is large and the data rate is high, an overly large hash table (over a million entries in our example) causes an increase in the number of aggregates to be flushed (a larger workload burst), and reduces the data rate that IOP can support without dropping tuples. The low CPU utilization for IOP with an overly large hash table is associated with the low data rates that IOP can support in these cases. OOP is generally better than IOP, especially for streams with large numbers of groups, and is less sensitive to the hash table size.

Experiment G2: This experiment demonstrates the potential memory-usage benefits of OOP for aggregation queries monitoring multiple data sources, using the following query, Q5.

```
Q5: SELECT srcIP, destIP, count(*)
      FROM M1 UNION M2
      [RANGE 1 min, SLIDE 1 min, WA ts]
      GROUP BY srcIP, destIP
```

The rates of M1 and M2 are both 110k pkts/sec, and the total number of groups in them is 65,536. We varied the arrival skew of M1 and M2, and executed Q5 with both GigaScope and OOP-GigaScope, recording the maximum memory usage. Figure 1 (in the introduction) shows the results of this experiment. OOP generally uses less memory than IOP; as arrival skew increases, the memory usage of OOP remains relatively flat, while that of IOP increases dramatically.

Experiment G3: This experiment provides a comparison of memory usage of a tumbling-window join query, Q6, with a window size of 10 seconds and input from multiple sources. Each input to the Join operator is a union of two streams: A and B, and C and D, respectively. The rate of each stream is 10k packets/second. (In practical stream join queries, the input rates for join are often rather low, because of prior data reduction by sam-

pling or aggregation.) We varied the arrival skew of A and B, and C and D, and recorded the maximum memory usage of each query run. Figure 10 shows the results of this experiment. The number of tuples that the IOP and OOP approaches need to maintain is the same. The difference is that in the IOP version, the tuples reside in input buffers of merge operators; in the OOP version, they are held in join hash tables. This experiment shows that the structural overhead of an OOP join in an IOP-oriented system is acceptable. When the arrival skew is below 20 seconds, the memory overhead is inconsequential. In the extreme case, the OOP join uses about 20% more memory than the IOP case. Our OOP join implementation used the hash-table structure of the original IOP join, which were not optimized for memory overhead.

```
Q6: SELECT M1.srcIP, M1.destIP, M1.ts
      FROM A UNION B as M1, C UNION D as M2
      WHERE M1.ts/10 = M2.ts/10 and M1.srcIP = M2.destIP and
            M1.destIP = M2.srcIP and M1.srcPort = M2.destPort and
            M1.destPort = M2.srcPort
```

8.2 OOP with NiagaraST

Experiments in NiagaraST were conducted on a Dual-Core AMD Opteron(TM) Processor 2214 with 4GB main memory, running Ubuntu Linux 2.6.17-10-server, and Sun® Java VM 1.5. We focus on comparing OOP and IOP in terms of memory usage, execution time, and latency. We added IOP support to NiagaraST,

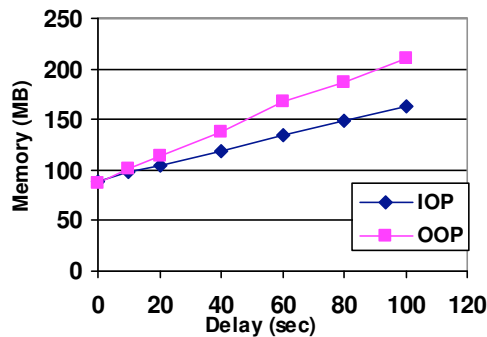


Figure 10. Memory usage of a tumbling-window join, Q6, in GigaScope

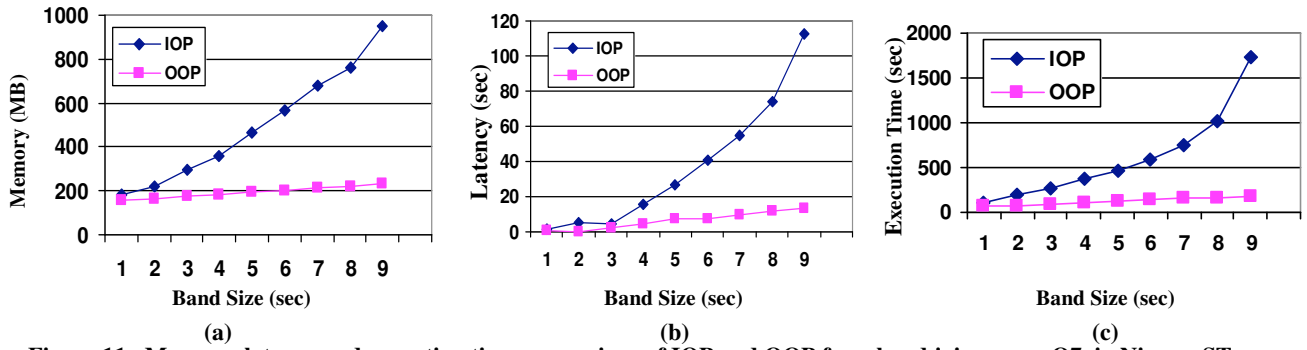


Figure 11. Memory, latency and execution time comparison of IOP and OOP for a band join query, Q7, in NiagaraST

just for this performance study.

Data Generation: For our experiments, we generated stream sources with different data volumes and different time skews using network packet headers from the Passive Measurement and Analysis project [22]. We generated three streams, two with high volume (approximately 4000 tuples/second), called M1 and M2, and one with very low volume, called C. The total data set size is approximately 135 MB. We simulated time skews among M1, M2, and C by manipulating the placement of tuples in the data file used to generate the three streams.

Experiment N1: This experiment compares the memory, execution time and latency performance of IOP and OOP evaluation of a query involving a band join, using query Q7. We varied the band size n of the Join operator, and measured the maximum memory usage, latency and execution time of the query. In this experiment, the input streams of the join, M1 and M2, are ordered and synchronized.

Q7: SELECT count(*)
[RANGE 1 min, SLIDE 1 min, WA M1.ts]
FROM M1, M2
WHERE M1.ts \geq M2.ts - n and M1.ts \leq M2.ts + n
and M1.srcIP = M2.destIP and M1.destIP = M2.srcIP
and M1.srcPort = M2.destPort and M1.destPort = M2.srcPort

Figure 11 shows (a) memory usage, (b) latency, and (c) execution time comparisons between IOP and OOP. Latency is the difference between the output time of the aggregate and the arrival time of punctuation from the input streams that covers it. Execution time reflects the CPU cost, and is the elapsed time of a query running at full speed over the input data set. The latency and execu-

tion-time numbers are the average of 8 runs. OOP significantly outperforms IOP on Q7, especially on memory and latency, as OOP can avoid sorting the output of the band join before aggregating, as discussed in Section 7.1.

Experiment N2: This experiment compares memory usage of IOP and OOP for an equality join on progressing attributes, using query Q8. One input to the join contains late tuples, which are simulated by combining M2 with a version of C that is delayed. Figure 12 shows that with the increase in the delay of the late tuples, the memory use for OOP increases more slowly than for IOP. The memory advantage of OOP comes from the Join operator in OOP purging M2 tuples sooner than in IOP, as discussed in Section 5.3.

Q8: SELECT M1.srcIP, M1.destIP, M1.ts
FROM M1, M2 Union C as M3
WHERE M1.ts = M3.ts and M1.srcIP = M3.destIP and
M1.destIP = M3.srcIP and M1.srcPort = M3.destPort and
M1.destPort = M3.srcPort

Experiment N3: This experiment compares memory usage of IOP and OOP on a sliding-window aggregate over multiple sources, using query Q9.

Q9: SELECT count(*) [RANGE 5 min, SLIDE 1 min, WA ts]
FROM M1 UNION M2 UNION C

We varied the delay of the arrival of C, and measured the maximum memory usage. Figure 13 shows that the memory usage of IOP grows significantly as the delay of C increases, while that of OOP is relative stable. Here, the memory benefit of OOP is due to OOP aggregation directly reducing tuples into aggregates without

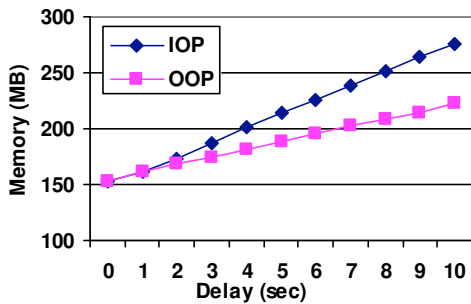


Figure 12. Memory comparison of IOP and OOP evaluation for a tumbling-window join query, Q8, with late tuples on one input, in NiagaraST

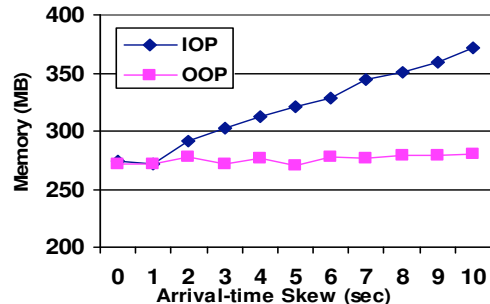


Figure 13. Memory comparison of IOP and OOP for a sliding-window count, Q9, with arrival-time skew of multiple data sources, in NiagaraST

first buffering and sorting the input.

9. CONCLUSION

Our initial experience with OOP architectures is encouraging. We have seen improvement over IOP in memory, latency and throughput under a variety of conditions. The fact that improvements were seen in two substantially different stream systems, NiagaraST and Gigascope, suggests that the benefits of OOP are widely applicable. The implementation overhead for supporting OOP does not seem severe, realizing that any practical stream system will need a stream-progress mechanism beyond just tuple arrival. There are several obvious next steps:

1. We believe we can reduce the memory overhead of several of our operators in NiagaraST and OOP-Gigascope, removing most of the differential with order-assuming implementations.
2. Gigascope supports user-defined aggregates [6]. That framework will need to be extended to support multiple simultaneous windows and correct handling of punctuation.
3. In Section 5.4, we noted that which operator implementations perform best can depend on data properties of a stream, such as the number of tuples per group in an aggregation. However, such properties can change over time. Hence, we are interested in “hybrid” implementations that monitor stream properties and adapt between approaches as appropriate.

10. ACKNOWLEDGMENT

This work was supported by NSF grants IIS-0086002 and IIS-0612311.

11. REFERENCES

- [1] Abadi, D., et al. Aurora: A New Model and Architecture for Data Stream Management. *VLDB Journal* 12(2), August 2003.
- [2] Arasu, A., Babu, S., Widom, J. The CQL Continuous Query Language: Semantic Foundations and Query Execution. *VLDB Journal* 14(1), March 2005.
- [3] Avnur, R., Hellerstein, J. M. Eddies: Continuously Adaptive Query Processing. *SIGMOD* 2000.
- [4] Arasu, A., Widom, J. Resource Sharing in Continuous Sliding-window Aggregates. *VLDB* 2004.
- [5] Balazinska, M, Balakrishnan, H., Madden, S., Stonebraker M. Fault-Tolerance in the Borealis Distributed Stream Processing System. *SIGMOD* 2005.
- [6] Cormode, G., et al. Holistic UDAFs at Streaming Speeds. *SIGMOD* 2004.
- [7] Cranor, C., Johnson, T., Spatashek, O. Gigascope: A Stream Database for Network Applications. *SIGMOD* 2003.
- [8] Ding, L., et al. Joining Punctuated Streams. *EDBT* 2004.
- [9] Ding, L., Rundensteiner, E.A. Evaluating Window Joins over Punctuated Streams. *CIKM* 2004.
- [10] Golab, L., Ozsu, M. T. Processing Sliding Window multi-joins in Continuous queries over Data Streams. *VLDB* 2003.
- [11] Hwang, J-H, Cetintemel, U., Zdonik, S. Fast and Highly-Available Stream Processing over Wide Area Networks. *ICDE* 2008.
- [12] Hammad, M. et al. Optimizing In-Order Execution of Continuous Queries over Streamed Sensor Data. *SSDBM* 2005.
- [13] Hammad, M., et al. Scheduling for Shared Window Joins over Data Streams. *VLDB* 2003.
- [14] Li, Hua-Gang, et al. Safety Guarantee of Continuous Join Queries over Punctuated Data Streams. *VLDB* 2006.
- [15] Jin Li, et al. No Pane, No Gain: Efficient Evaluation of Sliding-Window Aggregates over Data Streams. *SIGMOD Record* 34(1), March 2005.
- [16] Jin Li, et al. Semantics and Evaluation Techniques for Window Aggregates in Data Streams. *SIGMOD* 2005.
- [17] Theodore Johnson, et al. A Heartbeat Mechanism and Its Application in Gigascope. *VLDB* 2005.
- [18] Kang, J., Naughton, J. F., Viglas, S. Evaluating Window Joins over Unbounded Streams. *ICDE* 2003.
- [19] Ken Keys, et al. A Robust System for Accurate Real-time Summaries of Internet Traffic. *ACM SIGMETRICS Performance Evaluation Review* 33(1), June 2005.
- [20] Krishnamurthy, S., Wu, C., Franklin, M.J. On-the-Fly Sharing for Streamed Aggregation. *SIGMOD* 2006.
- [21] Naughton, J. et al. The Niagara Internet Query System. *IEEE Data Eng. Bulletin* 24(2), June 2001.
- [22] Passive Measurement and Analysis Project. San Diego Supercomputer Center. <http://pma.nlanr.net/PMA>.
- [23] Raman, V., Raman, B., Hellerstein, J. M. Online Dynamic Reordering for Interactive Data Processing. *VLDB* 1999.
- [24] Rundensteiner, E. A., et al. CAPE: Continuous Query Engine with Heterogeneous-Grained Adaptivity. *VLDB* 2004.
- [25] Srivastava, U, Widom, J. Flexible Time Management in Data Stream Systems. *PODS* 2004.
- [26] Srivastava, U, Widom, J. Memory-Limited Execution of Windowed Stream Joins. *VLDB* 2004.
- [27] StreamSQL. <http://www.streamsql.org>.
- [28] Tucker, P. *Punctuated Data Streams*. Doctoral Dissertation. Oregon Health & Science University, Portland, OR, 2005.
- [29] Tucker, P., et al. Exploiting Punctuation Semantics in Continuous Data Streams. *IEEE Trans. on Knowledge and Data Engineering*, 15(3), May 2003.
- [30] Urhan, T., Franklin, M. J. XJoin: A Reactively-Scheduled Pipelined Join Operator. *IEEE Data Eng. Bull.* 23(2), 2000.
- [31] Urhan, T. and Franklin, M. J. Dynamic Pipeline Scheduling for Improving Interactive Query Performance. *VLDB* 2001.
- [32] Viglas, S., Naughton, J. F., Burger, J. Maximizing the Output Rate of Multi-Way Join Queries over Streaming Information Sources. *VLDB* 2003.

APPENDIX

A.1. Technical Considerations

In this section, we discuss implementation issues unique to OOP stream systems.

A.1.1. Punctuation

In OOP systems, punctuation plays a central role—blocking operators rely on punctuation to output results and stateful operators rely on it to purge state. In this paper, we assume linear punctuation on the progressing attribute and assume that punctuation is “grammatical”—that is, no tuples violate previously received punctuation.

In order to adapt an IOP system to OOP, we must either add punctuation to the system, or, if the system already supports punctuation, we must extend it to fully support out-of-order processing. (Some existing IOP systems such as Gigascope support punctuation for handling lulls.) Punctuation can be initiated by timer callbacks. Assuming an input stream is ordered, the callback function can insert in the stream a punctuation carrying the largest progressing attribute value observed so far every time the timer fires. We call the current value of the progressing attribute *data time*. During lulls, the observed data time drifts away from the system time. When the difference between the data time and system time is above a predefined threshold, s , the callback function inserts punctuation to advance the data time to (*current system time* - s). Note that group-wise linear punctuation, a simple extension of linear punctuation, may be useful if groups are significantly unsynchronized. For example, if a query groups packets by network protocol, and packets for a certain protocol tend to lag behind others, having linear punctuation per protocol will allow groups that finish early to be output early.

One must be careful when adding punctuation to IOP systems, as punctuation may change the scheduling of tuple processing. For stream systems that support batch processing (i.e., query operators are invoked for a “batch” of tuples instead of for each individual tuple), punctuation should be treated as a high-priority tuple: Once a punctuation arrives, the in-progress batch should be considered complete and should be shipped to downstream operators. Note that this completion of a batch affects only the timing of tuple transmission and does not affect result values. Punctuation delayed by batch processing may delay result production and thus increase latency, particularly for sparse streams.

Even IOP systems that already support punctuation require non-trivial effort to extend punctuation to fully support OOP. First, OOP systems rely on punctuation to make progress, thus the system should produce punctuation at a granularity finer than both the smallest window size and the smallest window slide allowed in the stream system. The granularity of punctuation used for handling lulls in IOP systems can be much coarser, as it only needs to guarantee that stream queries make progress during lulls. Second, timer callbacks for generating punctuation may initiate duplicate punctuation, if the timer is set at a granularity fine enough to satisfy the smallest window slide. For efficiency, it is desirable to avoid such duplicates; further, it is also desirable to produce only punctuation that matches the boundaries of the smallest window slide currently used in the system. For example, if the smallest window slide used by queries currently running in the system is 5 seconds, it is desirable to produce punctuation with a 5-second

granularity and no finer. Third, as discussed in Section 7, to provide stream progress information efficiently, a query operator should choose what punctuation to produce based on the requirements of the operator that consumes its result. Tucker [28] has proposed a *describe* operator that provides punctuation appropriate for downstream operators. The describe operator filters out punctuation that will not help downstream operators and rolls incoming punctuation up to the appropriate level.

A.1.2. Overhead

In this section, we discuss the memory and computation overhead involved in supporting OOP.

A.1.2.1. Memory Overhead

Memory-structure overhead needs to be carefully considered in OOP system implementation. Typically, when memory usage of OOP and IOP systems is analytically compared, the comparison is based only on the number of tuples that each approach needs to retain, and ignores differences in the space overhead of various data structures. For example, if the input stream of a join operator potentially contains late tuples, an IOP system may buffer the on-time tuples in a tuple buffer, while an OOP system may store them in a hash table, which may have more space overhead. In general, OOP systems tend to have higher space overhead than IOP system, as they often require more complex data structures. Thus, in implementing the OOP architecture, it is important to implement the primary data structures in a space-efficient manner.

A.1.2.2. Punctuation-Processing Overhead

Punctuation processing incurs certain computational overhead costs. If the punctuation is too fine-grained, it may degrade the efficiency of the stream system, as it may increase processing time and consume transmission bandwidth in distributed stream systems. However, even with punctuation-to-tuple ratios as high as 15%, Tucker [28] observed very limited punctuation-processing overhead. These results assume that punctuation is grammatical. Otherwise, query operators (or, at least the input operators) also need to block any tuples violating punctuation, which indicates increased computational cost per tuple.

A.2. OOP Operator Implementation

An OOP query-operator implementation typically includes two primary functions, *processTuple()* and *processPunctuation()*. Query operators, especially those that need to enforce output stream order in IOP systems, can often be implemented more simply in OOP systems, as they need not maintain output stream order and thus need not devote resources to order enforcement. Both Join and especially Union benefit from reduced complexity in output generation in OOP implementations. The OOP implementation of Join is presented in Section 5.3. The OOP implementation of Union is discussed below.

A.2.1. Union

The OOP implementation of Union is shown below in Figure A-1. As compared to the order-enforcing IOP implementation, the OOP implementation is light weight in terms of both memory and latency. The only state that OOP Union maintains is the most recent punctuation value from each input stream and for the output stream. Group-wise punctuation may require maintaining such

```

State Maintained:
b0, b1: bounds on the low-water mark of left and right in-
put, respectively; initialized to -infinity;
o: low-water mark of the output stream; initialized to -
infinity;

Union(x)
let Si be the input stream to which x belongs;
if x is a tuple
  ProcessTuple(x, Si);
else if x is a punctuation
  ProcessPunctuation(x, Si);

ProcessTuple(t, Si)
output t;

ProcessPunctuation(p, Si)
bi = p.ts;
if o < min(bi, b1-i)
  output a punctuation with value min(bi, b1-i);
  o = min(bi, b1-i);

```

Figure A-1. OOP implementation of Union.

state for each group. OOP Union passes tuples through immedi-ately, and it emits punctuation with the min progressing attribute value observed from both streams (minus duplicates). Since Union is necessary for stream queries monitoring data from multiple sources, such as multiple network traffic links, the light weight implementation can be a great advantage. When an order-preserving union is used, both memory and delay incurred by Union can be prohibitive during lulls or in the presence of time skew.

A.2.2. Window Aggregation

The OOP implementation of window aggregation is more complex than its IOP implementation, as allowing out-of-order tuples requires managing multiple window extents simultaneously. Our implementation of window aggregation consists of two query operators, Bucket and Aggregate. The Bucket operator tags each input tuple with window-id(s) representing the window extent(s) to which the tuple belongs. We have formally defined how to map a tuple to window-ids in our previous paper [16]. There, we defined a function, *wids*, that maps a tuple to a set of window-ids based on the window specification of the window aggregation and the value of the tuple's windowing attribute. In our implementation, we use a pair of window-ids to represent the range of window-ids for each tuple. The basic structure of Bucket implementation is straight forward as shown in Figure A-2. The *processTuple()* function implements *wids* and appends a pair of attributes, *wid_start* and *wid_end*, to each input tuple; the *processPunctuation()* function applies the same *wids* to punctuation, appends a *wid_start* value and a wild star for *wid_end*, and output the punctuation. Note that all the complexity of tagging tuples with window-ids is encapsulated in the *wids* function. As discussed in our previous paper [16], the bucket operator does not need to maintain any state for the most commonly used, time-based sliding-window, but does need to maintain certain state for other types of window such as partitioned window.

```

State Maintained:
range: window size of the aggregation;
slide: window slide of the aggregation;
wa: windowing attribute used;

Bucket(x)
if x is a tuple
  ProcessTuple(x);
else if x is a punctuation
  ProcessPunctuation(x);

ProcessTuple(t)
(wid_start, wid_end) = wids(t);
append wid_start and wid_end to t as two data attributes;
output t;

ProcessPunctuation(p)
(wid_start, wid_end) = wids(p);
append wid_start and * to p;
output p;

wids(t)
compute wid_start and wid_end based on range, slide, t.wa
return (wid_start, wid_end);

```

Figure A-2. OOP implementation of window aggregation: the Bucket operator

```

State Maintained:
ht: hashtable maintaining partial window aggregates ;

Aggregate(x)
if x is a tuple
  ProcessTuple(x);
else if x is a punctuation
  ProcessPunctuation(x);

ProcessTuple(t)
for each wid in [t.wid_start, t.wid_end]
  compute hash value, hval, for t with its grouping attributes and wid;
  update ht[hval] using t;

ProcessPunctuation(p)
scan ht and output any group with wid value equaling p.wid_start;
output a punctuation with value p.wid_start;

```

Figure A-3. OOP implementation of window aggregation: the Aggregate operator

An aggregate operator, such as Max, will use the window-id(s) produced by the bucket operator as an additional grouping attribute—a tuple belongs to multiple groups if it is tagged with multiple window-ids. An algorithm for such an operator is shown in Figure A-3. In this algorithm, the hash-table contains partial aggregates. When punctuation arrives, the hash-table must be scanned in order to output the appropriate aggregate values. An alternative that avoids a hash-table scan is to output aggregates on

hash-table collisions, similarly to the slow flush mechanism discussed in Section 6. Although such aggregate implementations are more complex than their IOP counterparts, these implementations naturally support out-of-order tuples and do not require earlier operators to enforce order.