Semantics of Data Streams and Operators

David Maier[1], Jin Li[1], Peter Tucker[2], Kristin Tufte[1], and Vassilis Papadimos[1]

[1] Portland State University, Computer Science Department, Portland, OR, 97207
{maier, jinli, tufte, vpapad}@cs.pdx.edu
[2] Whitworth College, Spokane, WA 99251
ptucker@whitworth.edu

Abstract. What does a data stream mean? Much of the extensive work on query operators and query processing for data streams has proceeded without the benefit of an answer to this question. While such imprecision may be tolerable when dealing with simple cases, such as flat data, guaranteed physical order and element-wise operations, it can lead to ambiguities when dealing with nested data, disordered streams and windowed operators. We propose *reconstitution functions* to make the denotation and representation of data streams more precise, and use these functions to investigate the connection between monotonicity and non-blocking behavior of stream operators. We also touch on a reconstitution function for XML data. Other aspects of data stream semantics we consider are the use of punctuation to delineate finite subsets of a stream, adequacy of descriptions of stream disorder, and the formal specification of windowed operators.

1 Introduction

Data streams arise in many application domains, such as sensor processing, network monitoring and financial analysis. Streams from different domains could mean quite diverse things: a discrete signal, an event log, a combination of time series. Most work on algorithms and architectures for data stream management, however, never defines what a stream means. Thus it is hard to judge whether the definition of a particular stream operator is sensible. The default seems to be that a stream operator should behave like the pipelined version of a relational operator, but that may be inappropriate if the stream denotes something other than an unbounded relation, or if the representation the stream uses is different from the usual serialization of a finite table. When new operators are introduced, such as windowed versions of *group-by* and *join*, the situation becomes even fuzzier, especially if the semantics of the operator depends on the physical presentation order of items in the data stream.

In this paper, we propose *reconstitution functions* as a means to make precise the denotation and representation of a data stream. A reconstitution function is applied incrementally to prefixes of a stream to give successive approximations of its denotation. While generally we do not expect to actually apply a reconstitution function to a stream, it is useful in specifying the correct behavior of a stream analogue of an existing operator over the denotation domain. Reconstitution functions also prove useful in examining the subtle interplay between monotonicity and non-blocking behavior in stream operators.

We also consider additional semantics that may be available about the content or physical presentation of data streams, and discuss stream *punctuation* as one means to make such additional information available to stream operators and queries. One aspect we cover at more length is disorder in data streams. We explore some of the existing proposals for describing the expected disorder in a data stream. From our own investigations of disorder, we point out two areas in which disorder descriptions could be enhanced, namely non-uniform disorder, and statistical distributions of item displacement.

Our final topic is the semantics of windowed operators: operators that manipulate a data stream by decomposing it into a sequence of finite subsets and processing each subset in turn. Many such operators are defined in operational terms, which can make them sensitive to the physical presentation order of a stream. We propose a formal semantics for window definition that appears able to capture the underlying semantics of almost all window operators proposed to date. That semantics is independent of the physical order of a stream, and hence can describe the expected behavior of a window operator in the presence of disorder, and also leads to operator implementations with no internal buffering.

2 Stream Denotation and Representation

As we pointed out in the introduction, the proper interpretation of a stream might vary from application to application, but papers on data streams do not always make clear what interpretation they use. Even where the interpretation is provided, it can be somewhat confusing. For example, Law et al. [LWZ04] view data streams as "bags of append-only ordered tuples," or, alternatively, as "unbounded append-only bags of elements" when there is an explicit timestamp associated with each tuple. Such definitions conflate the kind of structure a stream denotes (an unbounded sequence?) with a particular representation (tuples with timestamps) with the function for recovering one from another (append). There are several questions left unanswered here. Are the timestamps considered part of the content of stream items — and thus, for example, available in selection conditions — or do they simply serve to define an order on the stream? Is the order total, or can two tuples have the same timestamp? Might the bag of tuples be viewed as a set by ignoring duplicates? The answers to such questions are important in evaluating whether or not a proposed stream operator is reasonable.

We think it important to distinguish the *denotation* of a stream from the particular *representation* of the denotation that the stream uses. The denotation is an abstract interpretation of what the stream means as a mathematical structure in some domain, whereas the representation is a particular encoding being used for elements of that domain. For example, a stream might be viewed as denoting a sequence of (finite) relation states over a common schema R : $[r_1(R),$

$r_2(R)$, $r_3(R)$, $\ldots$]. Let us assume for concreteness that the individual relations are unordered sets. There are many ways a stream could represent such a relation sequence:

(a) as the concatenation of serializations of the r_i (similar to the Rstream function of CQL [AW04R]);
(b) as a list of tuple-index pairs, where $\langle t, j\rangle$ indicates $t \in r_j$;
(c) as a serialization of r_1, followed by a series of "delta" tuples that indicate updates to make to obtain r_2, r_3, etc.;
(d) as a "replacement sequence," where some attribute A is treated as a key, and arriving tuple t replaces any existing tuple with the same $t(A)$-value to form a new relation state;
(e) as a "broadcast disk" format [AA95], where the state of a single r_i might be repeated multiple times;
(f) as an "overlapped window" encoding, in which each subsequence of 50 tuples represents a relation state in the sequence.

Clearly there are many other possible representations and variants for the relation-sequence denotation. Properly reflecting the behavior of an operation from the denotation domain in a stream operator requires consideration of the representation being used. Consider, for example, component-wise selection. That is, the desired outcome is $[\sigma_C(r_1), \sigma_C(r_2), \sigma_C(r_3), \ldots]$ for some selection condition C. For representations (a) – (e), one can apply the condition C individually to the items in an input stream S to get an output stream that represents the result. (There may be issues with representation (a) if σ_C selects away all tuples in some r_i, depending on how successive serializations are delimited.) However, applying C itemwise to a stream using representation (f) will not give the correct result, unless the output stream adopts a different representation.

As another example, consider a stream of sensor readings from, say, a temperature probe. We might view such a stream as denoting a discrete signal with a regular sampling rate (which in turn approximates a continuous physical measurement in the environment). A temperature stream might represent such a signal in several different ways:

(a) as a sequence of readings, one for each sampling point;
(b) as a sequence of changes in temperature from the previous sampling point;
(c) as a sequence of reading-timestamp pairs, with a pair included only if the reading differs from the previous reading included.

Representation (c) might be desirable for logistical considerations, such as power conservation, but it may require care in implementing certain operations. For example, if the average of two signals is the desired output, one needs to deal with the situation where there are long pauses in one stream because the temperature has not changed.

We also note that a single data stream might be viewed more than one way. That is, it can be construed with different denotations by interpreting it using different representations. Consider a stream of stock-trade items of the form `<ticker, time, shares, price>`. We can consider this stream as denoting a relation sequence, using the replacement-sequence representation (a) above with `ticker` as the key. Each item in the stream produces a new relation state in the sequence. An alternative view is that the trade stream denotes a relation sequence of recent trades, using representation (f) above. A third interpretation is that the stream denotes a collection of time series of prices, one for each different stock. Each item in the stream extends one of these time series. Observe that conversions between these different denotation values might be a no-op on the stream itself — just a change in the representation used to interpret it.

3 Reconstitution Functions

The notions of denotation and representation are useful for thinking about the semantics of a given stream, but are not necessarily precise enough yet for use in proofs of operator correctness or query equivalence. As a practical matter, the denotations we have used as examples are potentially infinite structures giving meaning to a whole stream, whereas there is a general desire to treat streams incrementally. We thus propose *reconstitution functions* as a mechanism for expressing and reasoning about stream semantics and representations. One can view a reconstitution function as constructing successive approximations to the denotation of a stream from successive finite prefixes of that stream. Consider streams with items of type **T**, and let **D** be the desired domain of interpretation. A reconstitution function *reconst* for type **stream(T)** will map each prefix P (of type **sequence(T)**) of a stream into **D**: $\mathit{reconst}(P) = d \in \mathbf{D}$. We give some example reconstitution functions below.

The Insert Reconstitution Function: If the domain of interpretation is **bag(T)**, then a reconstitution function *ins* that starts with an empty bag and inserts each successive stream item is appropriate:

$$\begin{aligned}&\mathit{ins}([]) = \varnothing\\&\mathit{ins}(P : i) = \mathit{insert}(i, \mathit{ins}(P)).\end{aligned}$$

Here we use [] for the empty sequence, and $P : i$ to denote sequence P extended by item i.

The Insert-Unique Reconstitution Function: If the intended domain of interpretation for a stream is **set(T)**, then we can define a reconstitution function ins_u that checks for duplicates:

$$\begin{aligned}&\mathit{ins}_u([]) = \varnothing\\&\mathit{ins}_u(P : i) = \text{if } i \notin \mathit{ins}_u(P) \text{ then } \mathit{insert}(i, \mathit{ins}_u(P)) \text{ else } \mathit{ins}_u(P).\end{aligned}$$

The Insert-Replace Reconstitution Function: Here we assume that each item in the stream has a component A that is treated as a key, and define a reconstitution function ins_r that guarantees only the most recent item with a given key is included:

$$\begin{aligned}&\mathit{ins}_r([]) = \varnothing\\&\mathit{ins}_r(P : i) = \mathit{insert}(i, \{j \mid j \in \mathit{ins}_r(P) \wedge j.A \neq i.A\}).\end{aligned}$$

The Insert-Replace-Collect Reconstitution Function: The ins_r reconstitution function yields the final state resulting from applying all the items in the stream. If the desired domain of interpretation is sequences of states (that is, type **sequence(set(T))**), similar to the relation-sequence example of Section 2, we can use the ins_{rc} reconstitution function to collect successive states:

$$\begin{aligned}&ins_{rc}([]) = []\\&ins_{rc}(P : i) = ins_{rc}(P) : ins_r(P : i).\end{aligned}$$

Remarks:

1. All the examples of reconstitution functions above are *incremental*: Each can be cast in the form $reconst(P : i) = g(reconst(P),\ i)$ for some function g. We do not require this property for a reconstitution function, but it may prove to have useful consequences. However, there are situations where a non-incremental reconstitution function is called for. For example, for some of the representations for relation sequences in Section 2, it might be that $reconst(P)$ only returns that part of the relation sequence for which it can construct complete relation states.
2. Note that if *reconst* returns a sequence, $reconst(P)$ need not be a prefix of $reconst(P : i)$, even though P is a prefix of $P : i$. For example *reconst* might sort according to some component A: $reconst(P) = sort_A(P)$.
3. The element type **T** of the stream need not be the element type in the domain of interpretation (though in the examples above they are the same). For example, Hammad et al. [HG04] have some stream items with "negative" flags that cancel previous normal items in the stream. Presumably, the reconstitution of such a stream would not contain any negative items.
4. If $reconst(P) = d$, we will sometimes write $const(d) = P$. We caution that this notation is informal, however. There may in fact be more than one P where $reconst(P) = d$, or no such P at all. That is, there can be values in the domain of interpretation **D** that are not the reconstitution of any stream.
5. The condition for a stream operator *sop* being the on-line analogue of an operation *dop* over the domain **D** is given by the commutative diagram in Figure 1. We note that the reconstitution function need not be constant throughout a query.
6. We see from the examples above that presentation order of items in a stream is sometimes significant for a reconstitution function (ins_r and ins_{rc}) and sometimes not (ins and ins_u). The question arises whether applications where stream order is not important show up in practice much. We think the more common case is that order does convey part of the semantics of a data stream, but that the other case does arise. Consider, for example, a stream of URLs arising from a web crawl. While some aspects of the crawl process, or posting times of pages, might influence the order of URLs in the stream, most applications will treat it as an unordered collection. There are also cases where there are some global aspects of order in a stream, but locally order is not significant. For example, consider the stream of network packets

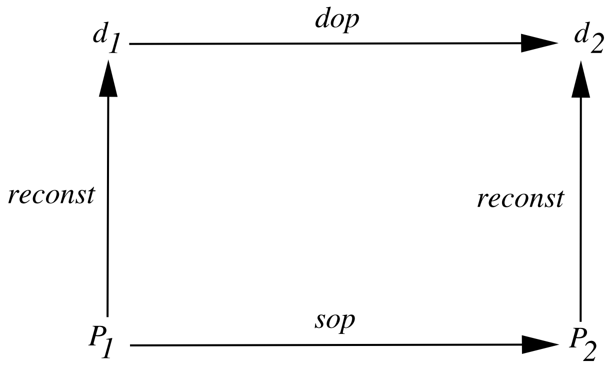
$$\begin{tikzcd} d_1 \arrow[r, "dop"] & d_2 \\ P_1 \arrow[u, "reconst"] \arrow[r, "sop"] & P_2 \arrow[u, "reconst"'] \end{tikzcd}$$

Fig. 1. A stream operator as an on-line analogue of a domain operator

passing through a router. The denotation might be a collection of sessions under various protocols, where each session is a sequence of messages. However, the packets for a given message might not be in order, because of taking different routes or being retransmitted.

4 Monotonicity, Reconstitution and Non-blocking Operators

We are using reconstitution functions to study the connection between monotonicity of a domain operation and the existence of a non-blocking stream analogue, particularly for hierarchically structured data such as XML. Several papers [ST97, LWZ04] have singled out monotone relational operators (such as *select*, *join*, *dupelim*), as they are easy to carry over to stream counterparts. This connection relies on the common reconstitution functions for relational data, typically *ins* or ins_u. For a monotone relational operator *rop*, $r_1 \subseteq r_2 \implies rop(r_1) \subseteq rop(r_2)$. If $const(rop(r_1)) = U_1$ (that is, $ins(U_1) = rop(r_1)$), then we can find a sequence U_2 such that $const(rop(r_2)) = U_1 : U_2$. That is, the representation for $rop(r_1)$ is a prefix of the representation for $rop(r_2)$. Thus a stream analogue *sop* for *rop* can emit $const(rop(ins(P)))$ in response to prefix P of a stream, and know that $const(rop(ins(P : i)))$ will extend that response.

We note, however, that the definition of monotone depends on the definition of containment. The appropriate definition is fairly clear for relations (tuple subset), but there are alternatives when considering hierarchical data such as nested relations and XML. Consider the relational operation $nest_B$, which nests B-values of a relation based on equality of values on the remaining attributes. Consider relation r containing the first three tuples of Figure 2(a) (in bold type). Then $nest_B(r) = v$, where nested relation v is given in Figure 2(b). Let r^+ be the relation in Figure 2(a) with the fourth tuple included. Then $nest_B(r^+) = w$, for w in Figure 2(c). The question is now whether $nest_B$ is monotone. Specifically, is $v \subseteq w$?

r(A	B)
1	**c**
2	**e**
1	**d**
2	f

(a)

v(A	{B})
1	{c, d}
2	{e}

(b)

w(A	{B})
1	{c, d}
2	{e, f}

(c)

Fig. 2. Monotonicity of nesting

The answer depends on the definition of containment for nested relations. There are at least two possibilities:

1. Containment is simply tuple subset, in which case $v \not\subset w$, since $\langle 2, \{e\}\rangle$ is not in w. Hence $nest_B$ is not monotone.
2. Containment is subsumption. Thus $v \subseteq w$, because every tuple in v is subsumed by some tuple in w. In particular, $\langle 2, \{e\}\rangle$ is subsumed by $\langle 2, \{e, f\}\rangle$, and $nest_B$ is monotone.

Suppose we choose the second definition, where nest is monotone. Can we derive a non-blocking stream version of that operation? Doing so requires an appropriate choice of reconstitution function. Let us call the proposed stream operator *snest*, and consider how we want it to behave. We want $snest(P)$ to be a prefix of $snest(P : i)$, and, of course, $reconst(snest(P : i))$ should subsume $reconst(snest(P))$. If upon receiving $\langle 2, f\rangle$, *snest* emits $\langle 2, \{f\}\rangle$, then the simple *ins* reconstitution function will not give the desired relationships. However, if the reconstitution function performs a deep union [BDT99] with the cumulative result and combines $\langle 2, \{f\}\rangle$ with $\langle 2, \{e\}\rangle$ to form $\langle 2, \{e, f\}\rangle$, we will satisfy the conditions. Alternatively, *snest* could maintain state and emit $\langle 2, \{e, f\}\rangle$ upon receiving $\langle 2, f\rangle$. In that case, we need a "subsume-replace" reconstitution function that overwrites $\langle 2, \{e\}\rangle$ with $\langle 2, \{e, f\}\rangle$.

A particular case of interest to us is streams of XML. If a stream of XML elements simply denotes a sequence of independent documents, then not much new mechanism is needed beyond what is used for flat data items in a stream. On the other hand, we may want to view an XML stream as a series of fragments that constitute a single XML document. We have been working on a deep-union-like operator for XML we call *merge* [TM01]. The *merge* operator is logically performing a lattice-join of two XML documents in a subsumption lattice. One use we have for *merge* is a reconstitution-like structural aggregation operator called *accumulate*. The *accumulate* operator successively merges in XML fragments with a base document, called an *accumulator*, and makes the accumulator available to further query-processing steps. For example, in Figure 3(a) we have an accumulator for auction data that is grouping bids under their appropriate items. (This example is based on the XMark benchmark [XM03].) Figure 3(b) shows a new bid coming in as an XML element, and Figure 3(c) is the result of merging that element into the accumulator. The behavior of the merge operator is modulated by a *merge template*, which in essence indicates which lattice we are using to define the lattice-join. In the example of Figure 3, the merge template would indicate,

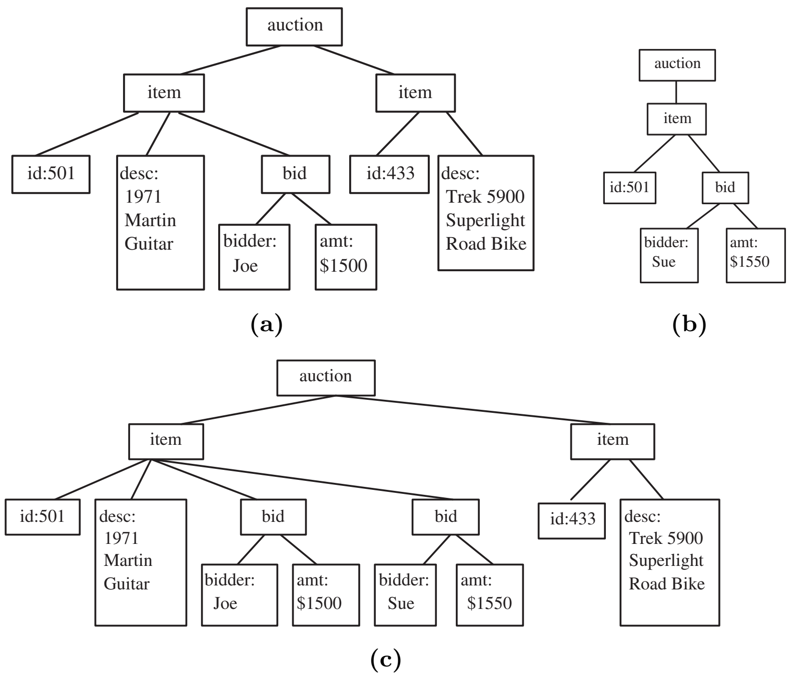

Fig. 3. Illustration of the *merge* operator showing (a) initial accumulator, (b) a fragment to be merged, and (c) the resulting accumulator

for instance, that the merge process should combine corresponding `<item>` elements, but create new `<bid>`, `<bidder>`, and `<amt>` elements, as opposed, say, to creating a new element in the accumulator for each `<item>` element added.

5 Additional Stream Semantics

There may be information known about a stream in addition to its denotation and representation, related to its content or presentation order, that is useful for query processing. Some examples on content are whether or not the stream contains duplicates, and if some subset of attributes forms a key (that is, there are no duplicate values over these attributes). Another example, for a stream with a relation-sequence denotation, is whether there is a constant bound on the size of the relation states. For instance, if the stream contains position reports on a fleet of vehicles, and each relation state consists of the most recent report on each vehicle, then the size of any state is at most the number of vehicles n. Such information can be useful in determining whether a query has bounded state requirements or not [BB02]. Information on the physical presentation of streams is useful as well, such as if the stream is ordered on a particular attribute, or whether there is limited skew among the arrival times of items on different streams [BU04].

Our own work in this area has dealt with the case where a stream can be viewed as a mixture of finite sub-streams, and where the ends of sub-streams can be determined. The sub-streams may occur naturally, for example, all the bids for a single auction, or be externally imposed, such as all sensor readings in a ten-minute interval. The knowledge about when a sub-stream ends might be supplied by the stream source, or arise from measurements of network delay, or be deduced from application semantics, such as knowing that each vehicle reports its position at least every 20 seconds. Where we have such knowledge, we can explicitly augment the stream with it, via *punctuations* [TM03]. A punctuation is a pattern p inserted into the data stream with the meaning that no data item i matching p will occur further on in the stream. For example, a punctuation $\langle \{\mathrm{site3}, \mathrm{site5}\}, 1663, *, [\text{6:30p}, \text{6:45p}], * \rangle$ in the bid stream for an auction server signals that all bids from Sites 3 and 5 for auction item 1663 made during the 15-minute period starting at 6:30p have been seen. Punctuations can be used to improve stream operators in at least two ways. First, punctuations can unblock blocking operators. For example, a *group-by* operator computing the maximum bid for each auction item over each 1-hour period could emit answers after seeing a collection of punctuations similar to the one above. Second, punctuations may allow a stateful operator to safely discard parts of its state. For example, a *dupelim* operator receiving the punctuation above could purge all data items from its state that match that punctuation.

While we have been working with punctuation-aware operators for several years now, there are still many questions and extensions to investigate. We have a good understanding of how single operators can exploit punctuation. However, we are less far along in understanding when particular punctuation helps a given query, or, a more challenging problem, starting from a query, determining what punctuation, if any, would benefit the query [TMS03].

Currently our punctuation marks the end of a sub-stream. We believe there may also be advantages to "forward-looking" punctuation that describes data that will appear further on in a stream. We are also starting to investigate the notion of a *deterministic stream*: a stream in which for any possible data item i, one is guaranteed to eventually see either i or a punctuation matching i. Another variation we are considering is where a bound is known on the amount of unpunctuated data (items at a given instant with no corresponding punctuation received). Reasoning with such information could lead to bounds on the amount of state a query needs. Finally, our current implementation of punctuation is for flat data, though the underlying query engine handles general XML [NDM]. Punctuation for XML data is still an open area.

6 Disorder in Streams

Disorder in data streams can arise from many sources, such as stream items being routed by different paths in a network, or combining streams that are out of synch. A stream may have multiple natural orders, such as start time and end time of a network flow, and cannot be sorted on both simultaneously. There are also algorithms for stream operators that produce disordered output, such as windowed multi-join [HF03]. In order to deal with disorder in stream query processing, it is useful to have some description of the expected or maximum disorder in a stream. There have been several proposals in this regard. Some describe disorder operationally, that is, in terms of what kind of operation will restore order. An example is the order specifications of Aurora [AC03], which say how much buffer space is needed to sort the stream (or partitioned sub-streams of it) on a particular attribute. Other disorder descriptions express the maximal displacement of any item from its correct position. The displacement is usually measured from a "high-water mark," and expressed either as a number of items or as a difference in the ordering attribute. For example, consider the stream of bid items in Figure 4, where we are considering order on bid time (the fourth column). We see that item i_4 is out of order. It is displaced by 2 items from its correct position (between items i_1 and i_2) and by 3 seconds based on the value of the *time* attribute. Examples of this maximum-displacement approach to describing disorder include *slack* in the early versions of the Aurora system [C02] and the *banded-increasing* property of the Gigascope project [CJ02]. The *k-ordering* [BU04] and *out-of-order generation* [SW04] constraints of the STREAM project are similar.

$$\begin{array}{ll}
i_1 & \langle \text{site3}, 1663, \text{b420}, 3{:}15{:}32, \$11.50 \rangle \\
i_2 & \langle \text{site2}, 7287, \text{b812}, 3{:}15{:}35, \$8.00 \rangle \\
i_3 & \langle \text{site5}, 1663, \text{b173}, 3{:}15{:}36, \$12.50 \rangle \\
i_4 & \langle \text{site1}, 1601, \text{b662}, 3{:}15{:}33, \$65.00 \rangle \\
i_5 & \langle \text{site3}, 1663, \text{b420}, 3{:}15{:}38, \$13.00 \rangle \\
 & \ldots
\end{array}$$

Fig. 4. A disordered auction stream

While such disorder descriptions are useful, our own investigations have shown that they are somewhat limited in their expressive power. First, they assume that the disorder bound is constant across the stream. Figure 5 shows netflow records from a router in the Abilene Network Observatory [Abi], ordered by the sequence in which they were emitted, and showing the start time of each netflow. (A netflow record summarizes packet traffic between two $\langle \text{IP}, \text{port} \rangle$-pairs.) We have termed such a stream *block sorted*, and it is clear no items are displaced across block boundaries.

A second issue is that existing descriptions focus on the maximum disorder, rather than the average displacement or a distribution of displacements. Consider Figure 6, which shows the observation time of the 8^{th} packet in a network flow, ordered by the start time of each flow, for a network trace gathered by the PMA project [PMA]. While one packet is significantly displaced (perhaps a retransmission), the rest occur in a close band of their desired position. We refer to such a sequence as *band disordered.* It would be useful to have some statistical characterization of such disorder, so, for example, one could estimate how the accuracy of a query is affected by a given cutoff on late items.

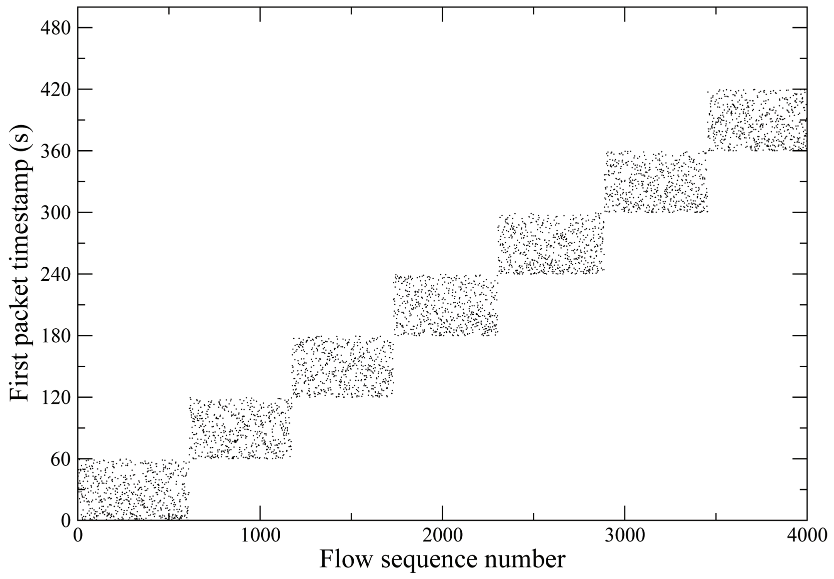

Fig. 5. Block-sorted disorder

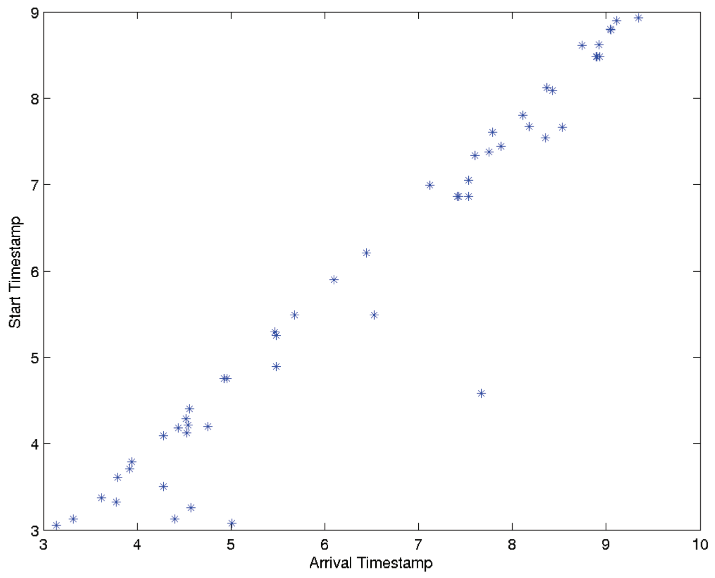

Fig. 6. Band disorder

We note that any disorder descriptions of these kinds can be used to generate punctuations in a data stream, marking the end of particular subsets of data. For example, if a stream is known to be bound by a slack of 20 seconds, a *punctuate* operator can insert a punctuation of the form $\langle *, *, *, t - 20s, * \rangle$

when it sees an item with bid time t. (However, it likely would not insert a punctuation based on every item, but less frequently, based on the needs of the query.)

7 Windowed Operators

Another area where semantics of streams is still a bit fuzzy is windowed operators. One way to modify a blocking or stateful operator to work with data streams is to change it from considering the totality of a stream to instead operating over a series of finite subsets of the stream. (There are other ways to modify such an operator. For example, a blocking aggregate such as sum can be converted to report a "running sum" after each input item.) The most studied windowed operators are *group-by* (aggregation) [AW04, C04, SH98, AC03] and *join* [HF03, KNF03, HAK03, GO03].

Windowed operators predate their current use in data streams. A WINDOW construct over stored data appears in SQL 1999 [SQL99]. In fact, the CQL formulation for windows draws from the SQL counterpart [ABW03]. There has been a considerable range of proposals on how to define windowed operators, based, for example, on whether one end or both of the window moves (and in which direction), the size of the window (its *range*), how much and how often it moves (its *slide*), and where it is located relative to the current point in the stream (its *offset*). The range, slide and offset can be denominated in terms of a number of items, or a quantity or duration of some attribute. In the case that the window range is expressed by a number of items, and the operator partitions the stream (such as a group-by aggregation), there are variants where the range is applied to the whole stream or separately to each partition. The windowing attribute can be a sequence number, an internally assigned arrival timestamp, or a value supplied by the stream source.

We see some semantic problems, however. Most approaches to windows are described in terms of the physical presentation of the stream, rather than its denotation, often on an operator-by-operator basis [AC03]. Such operational definitions can lead to problems when the stream appears out of order with respect to the windowing attribute. We have been developing a formal approach to window specifications that is independent of physical stream order [LMP04]. In our approach, the various window extents that arise as a window slides over a stream are each given an explicit *window identifier* (*window id*), and an *extent* function defines the stream items that are associated with each window id. Our approach assumes that windows are always defined against an explicit attribute W, though in practice W might be a sequence number or timestamp supplied by the stream management system.

For illustration, consider window specifications having the form [RANGE r, SLIDE u], where r and u are quantities compatible with the domain of W. For example, if the domain of W is time, then a possible specification is [RANGE $30s$, SLIDE $10s$], which defines window extents of length 30 seconds, spaced every 10 seconds.

The functions that define window extents are expressed in terms of the collection I of items in input stream S. The window function gives the set of window ids for a particular specification. In our illustration,

$$windows(I, r, u) = \{0, 1, 2, \dots\}.$$

Here the set of window ids does not depend in the stream contents, nor the range and slide parameters, but it may for other window types. The *extent* function determines the items associated with each window extent. In our illustration, for $w \in windows$,

$$extent(I, w, r, u) = \{i \in I \mid w \cdot u \leq i.W \leq w \cdot u + r\}.$$

(This definition is slightly simplified; in general, it must account for the boundary conditions at stream startup.)

We have found this approach quite expressive, being able to capture many flavors of windows mentioned in the literature: landmark, tumbling, slide-by-tuple, partitioned, etc. Moreover, it has led to a class of algorithms for windowed aggregates that often outperform approaches based on intra-operator buffering. Our approach requires an inverse, *wids*, for the *extent* function, giving the set of window ids of window extents that a given item appears in. In our running illustration,

$$wids(I, i, r, u) = \{w \mid \lceil i.W/u \rceil - 1 < w \leq \lceil (i.W + r)/u \rceil - 1\}.$$

(Again, this definition is simplified.) Our approach uses a *bucket* operator to extend each stream item with its associated window ids. The resulting output can then be fed to a *group-by* operator that treats the window id as just another grouping attribute. We rely on punctuation (also supplied by *bucket*) to keep the aggregation unblocked.

There are still several issues we are investigating with our window semantics. One is to classify window specifications in terms of properties of their *wids* functions. In the example above, *wids* is "context-free" in the sense that it can be applied by *bucket* to each stream item in isolation. Other window specifications, such as slide-by-tuple with a time-interval range require *bucket* to be stateful. A second area of investigation is the interaction of windowing with reconstitution. Suppose *wind* is a reconstitution function that interprets a stream as a sequence of window extents using an interval-based range and slide, and that we are interested in applying operations window-by-window to such a sequence (the usual situation). In some cases, the appropriate stream analogue is easy to come by. For example, with selection, we have

$$\sigma(wind(P)) = wind(\sigma(P)).$$

If we consider a window based on tuple count, in contrast, the equality no longer holds. Other operators are more challenging. With duplicate elimination

$$dupelim(wind(P)) \neq wind(dupelim(P))$$

even with an interval-based window. In fact, as far as we can determine, there is no function g such that

$$dupelim(wind(P)) = wind(g(P)).$$

The result of $dupelim(wind(P))$ must be a stream with a different reconstitution function, perhaps one with positive and negative tuples, or using explicit window ids.

8 Conclusions

We hope we have taken at least a small step towards answering the question *What do data streams mean?* There are still many gaps and rough edges here. While we think reconstitution functions are a useful device for capturing data-stream semantics, it is not yet tested whether they can deal with the range of data streams seen in practice, or are helpful in proving properties of stream operators. Reconstitution functions help clarify for us the requirements for a non-blocking stream version of a monotone domain operator, and suggest approaches for reconstitution of streams of XML fragments. However, we would like to find ways to encode more general updates in an XML stream, such as deletions. Playing with punctuations has been fun, but the problems are getting harder, such as proving space bounds on stream queries. Having more expressive descriptions for stream disorder is just the starting point. The real challenge is to use them to manage tradeoffs between query latency, accuracy and space usage. Finally, the alert reader will have noted we have not yet integrated reconstitution functions with our semantics for windows. So do something about it.

Acknowledgements

This work was supported by NSF grant IIS 0086002. We thank Ted Johnson for his insights on sources of disorder, and the Abilene Observatory for access to network monitoring data.

References

[AC03] D. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, S. Zdonik. Aurora: a new model and architecture for data stream management. *VLDB Journal (12)2: 120-139, August 2003.*

[Abi] The Abilene Observatory. `http://abilene.internet2.edu/observatory`.

[AA95] S. Acharya, R. Alonso, M. Franklin, and S. Zdonik. Broadcast Disks: Data Management for Asymmetric Communication Environments. In *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD 95).* San Jose, CA, June 1995.

[ABW03] A. Arasu, S. Babu and J. Widom. *The CQL Continuous Query Language: Semantic Foundations and Query Execution* Stanford University Technical Report, Oct. 2003

[AW04] A. Arasu, J. Widom. Resource Sharing in Continuous Sliding-Window Aggregates. In *Proceedings of the 30th International Conference on Very Large Databases (VLDB 2004)*, Toronto, Canada, September 2004.

[AW04R] A. Arasu, J. Widom. A Denotational Semantics for Continuous Queries over Streams and Relations. *SIGMOD Record* 33(3), September 2004.

[BB02] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and Issues in Data Stream Systems. In *Proceedings of the 21st ACM Symposim on Principles of Database Systems (PODS 2002)*, Madison, Wisconsin, June 2002

[BU04] S. Babu, U. Srivastava, and J. Widom. Exploiting k-Constraints to Reduce Memory Overhead in Continuous Queries over Data Streams. *ACM Transactions on Database Systems*, 29(3):545-580, September 2004.

[BDT99] P. Buneman, A. Deutsch, and W.C. Tan. A Deterministic Model for Semistructured Data. In *Proceedings of the Workshop on Query Processing for Semistructured Data and Non-Standard Data Formats*, Jerusalem, Israel, January 1999.

[C02] Carney, D., *et al.* Monitoring Streams - A New Class of Data Management Applications. In *Proceedings of the 28th International Conference on Very Large Databases (VLDB 2002)*, Hong Kong, China, August 2002.

[C04] Cormode, G., *et al.* Holistic UDAFs at streaming speeds. In *Proceedings of the 2004 ACM SIGMOD International Conference on the Management of Data (SIGMOD 2004)*, Paris, France, June 2004.

[CJ02] C. Cranor, T. Johnson, O. Spatscheck. *How to Query Network Traffic Data Using Data Streams*, unpublished manuscript, 2002.

[GO03] L. Golab, M. Tamer Özsu. Processing Sliding Window Multi-Joins in Continuous Queries over Data Streams. In *Proceedings of the 29th International Conference on Very Large Databases (VLDB 2003)*, Berlin, Germany, September 2003.

[SQL99] P. Gulutzan and T. Pelzer. *SQL-99 Complete, Really.* CMP Books, 1999. ISBN: 0-87930-568-1

[HG04] M. Hammad, T. Ghanem, W. Aref, A. Elmagarmid and M. Mokbel. Efficient Pipelined Execution of Sliding-Window Queries Over Data Streams. Purdue University Department of Computer Sciences Technical Report CSD TR#03-035, June 2004.

[HF03] M. Hammad, M. Franklin, W. Aref, and A. Elmagarmid. Scheduling for shared window joins over data streams. In *Proceedings of the 29th International Conference on Very Large Databases (VLDB 2003)*, Berlin, Germany, September 2003.

[HAK03] M. Hammad, W. Aref, and A. Elmagarmid. Stream Window Join: Tracking Moving Objects in Sensor-Network Databases. In *Proceedings of the 15th International Conference on Scientific and Statistical Database Management (SSDBM 2003)* Cambridge, MA, July 2003.

[KNF03] J. Kang, J. Naughton and J. Viglas. Evaluating Window Joins over Unbounded Streams. In *Proceedings of the 19th International Conference on Data Engineering (ICDE 2003)*, Bangalore, India, March 2003.

[LWZ04] Y. Law, H. Wang, C. Zaniolo. Query Languages and Data Models for Database Sequences and Data Streams. In *Proceedings of the 30th International Conference on Very Large Databases (VLDB 2004)*, Toronto, Canada, September 2004.

[LMP04] J. Li, D. Maier, V. Papadimos, P. A. Tucker and K. Tufte. *Evaluating Window Aggregate Queries over Streams.* OGI Technical Report, available from http://www.cse.ogi.edu/~jinli/papers/WinAggrQ.pdf, May 2004.

[NDM] J. Naughton, D. DeWitt, D. Maier. *et al.* The Niagara Internet Query System. http://www.cs.wisc.edu/niagara.

[PMA] Passive Measurement and Analysis project. San Diego Supercomputer Center. http://pma.nlanr.net/PMA.

[ST97] J. Shanmugasundaram, K. Tufte, D. DeWitt, J. Naughton, D. Maier, Archtecting a Network Query Engine for Producing Partial Results. *Lecture Notes in Computer Science, Vol. 1997/2001 The World Wide Web and Databases; Third International Workshop WebDB 2000, Dallas TX, May 2000, Selected Papers*, Springer-Verlag Publishers 2001.

[SW04] U. Srivastava and J. Widom. Flexible Time Management in Data Stream Systems. In *Proceedings of the 2004 ACM Symposium on Principles of Database Systems (PODS 2004)*, Paris, France, June 2004.

[SH98] M. Sullivan and A. Heybey. Tribeca: A system for managing large databases of network traffic. In *Proceedings of the USENIX Annul Technical Conference*, New Orleans, Louisiana, June 1998.

[TM03] P. A. Tucker, D. Maier, T. Sheard and L. Fegaras. Exploiting Punctuation Semantics in Continuous Data Streams. *Transactions on Knowledge and Data Engineering*, 15(3):555-568, May, 2003

[TMS03] P. A. Tucker, D. Maier and T. Sheard. Applying Punctuation Schemes to Queries over Continuous Data Streams. *IEEE Data Engineering Bulletin*, 26(1):33-40, March, 2003

[TM01] K. Tufte and D. Maier. Aggregation and Accumulation of XML Data. *IEEE Data Engineering Bulletin* 24(2):34-39, June 2001.

[XM03] XMark Benchmark. http://www.xml-benchmark.org/