Exploiting Punctuation Semantics in Continuous Data Streams

Peter A. Tucker, David Maier, *Member*, *IEEE*, Tim Sheard, and Leonidas Fegaras, *Member*, *IEEE*

Abstract—As most current query processing architectures are already pipelined, it seems logical to apply them to data streams. However, two classes of query operators are impractical for processing long or infinite data streams. Unbounded stateful operators maintain state with no upper bound in size and, so, run out of memory. Blocking operators read an entire input before emitting a single output and, so, might never produce a result. We believe that a priori knowledge of a data stream can permit the use of such operators in some cases. We discuss a kind of stream semantics called punctuated streams. Punctuations in a stream mark the end of substreams allowing us to view an infinite stream as a mixture of finite streams. We introduce three kinds of invariants to specify the proper behavior of operators in the presence of punctuation. *Pass invariants* define when results can be passed on. *Keep invariants* define what must be kept in local state to continue successful operation. *Propagation invariants* define when punctuation can be passed on. We report on our initial implementation and show a strategy for proving implementations of these invariants are faithful to their relational counterparts.

Index Terms—Continuous queries, stream semantics, continuous data streams, query operators, stream iterators.

1 Introduction

There are many examples of stream-processing applications: Financial applications process stock-ticker streams, telephone monitoring applications process call-data streams [7], [10], and traffic monitoring applications process streams from sensors on the highway [16]. These applications can often be viewed as queries over streams. Hence, it may be desirable to use a DBMS to implement all or part of them. However, two properties of streams make them difficult to process: First, data is often generated from a source that can potentially produce an unbounded stream. Second, a stream's contents can only be accessed sequentially. Traditional queries are comprised of relational operators that assume a finite data source that can be accessed randomly. We need operators designed for stream inputs.

1.1 Streaming Data Examples

Example 1 ***Tracking Temperature in a Warehouse.*** A warehouse containing temperature-sensitive merchandise may deploy temperature sensors to regularly report the current temperature to a monitoring system, as shown in Fig. 1a. The monitoring system is implemented as a query that unions all temperature reports (removing duplicates), groups all reports for each hour, and outputs the maximum temperature reported for each group.

Unfortunately, the approach above will fail for two reasons: First, duplicate elimination requires an unbounded amount of state. Thus, the system will ultimately run out of memory. Second, group-by must wait until all data from its input has been read before it can output results. Since the input is a continuous stream, it will never emit a result.

Example 2 ***Tracking Bids at an Online Auction.*** The XMark benchmark [21] tests queries over stored auction data. We extend this to an online auction, shown in Fig. 1b. Sellers post items for sale to the `Sellers` portal, which merges submissions into a single output stream. Bidders post increase bids to the `Buyers` portal, which merges bids into a single output stream. A useful query tracks final prices for items, joining items for sale with item bids, and then summing up bid-increase values for each item.

We use the symmetric hash join [27] to join the two inputs. This implementation does not block on its inputs, but maintains an unbounded amount of state. Throughout the auction, hash tables used for the join grow without bound, so the system eventually runs out of memory. Further, the group-by operator that calculates the sum does not output any results until all items have arrived.

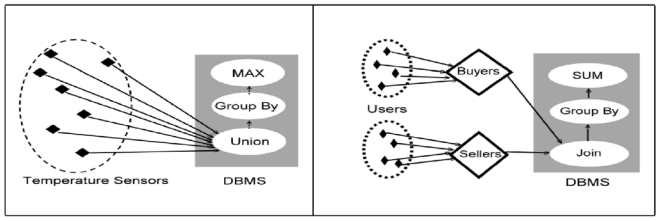
1.2 Motivation for Punctuated Streams

The examples illustrate two kinds of operators that are impractical for continuous data streams. *Blocking operators*, such as group-by, wait until all data from at least one input has been read before producing a result. *Unbounded stateful operators*, such as join, maintain state that grows with no upper bound. Our goal is to find stream-based analogues for table-based (finite) operators, exploiting stream semantics to overcome these problems.

Our work builds on a simple observation: When an input has been read entirely, blocking operators emit results and unbounded stateful operators discard state. Could query operators do similar work if the end of a specific subset of data had been read entirely? A blocking operator might be able to emit a subset of its result early. An unbounded stateful operator might reduce state. We use special annotations embedded in data streams, called *punctuations*,

- *P.A. Tucker, D. Maier, and T. Sheard are with the OGI School of Science and Engineering at OHSU, 20000 NW Walker Road, Beaverton, OR 97006. E-mail: {ptucker, maier, sheard}@cse.ogi.edu.*
- *L. Fegaras is with the Department of Computer Science and Engineering, The University of Texas at Arlington, 416 Yates Street, 301 Nedderman Hall, PO Box 19015, Arlington, TX 76019. E-mail: fegaras@cse.uta.edu.*

Manuscript received 15 May 2002; revised 15 Nov. 2002; accepted 4 Dec. 2002.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number 117896.



(a) (b)

Fig. 1. (a) System to monitor temperature reports within a warehouse. (b) System to track bids for an auction. Diamond outlines represent `Buyers` and `Sellers` portals.

to specify the end of a subset of data. Informally, a punctuation indicates no more tuples will follow that match the punctuation. We show that punctuations allow blocking and unbounded stateful operators to be used over continuous data streams for a larger range of queries.

1.3 Enhancing the Examples with Punctuations

We can fix the temperature monitoring system (Example 1) by embedding punctuations in sensor reports. Suppose reports are of the form $< \texttt{sid}, \texttt{hour}, \texttt{minute}, \texttt{currtmp} >$. We enhance the temperature sensors to embed punctuations at the end of each hour, stating that all reports have been emitted for that hour. When punctuations from all sensors for a particular hour have been received by union, we know that there will be no reports from any sensor for that hour. Tuples that match those punctuations no longer need be retained in operator state. Likewise, when group-by receives punctuation for a particular hour, results can be emitted for that hour, and state relating to those groups is no longer needed.

Punctuations also fix the online auction (Example 2). Items submitted to the `Sellers` portal are assigned a unique `itemid`. Once emitted, a punctuation follows stating no more items with that `itemid` will appear. As mentioned earlier, join maintains two hash tables: one for items for sale and the other for bids. Punctuation from the `Sellers` portal tells the join that items in its bid hash table with that `itemid` can be removed since all items with that `itemid` have arrived from the `Sellers` input. Further, when the auction for an item has closed, a punctuation can be embedded into the bid stream stating no more bids for that item will arrive. We can use this information to discard state in the items-for-sale hash table related to that item. Finally, once punctuations for a particular item are present for both inputs, join can emit its own punctuation, stating that it will not emit any more tuples with that `itemid`. The group-by operator can use that punctuation to determine that no more bid increases will arrive for that item and can emit its result for that item.

1.4 Organization

The rest of this paper is organized as follows: We formalize the notion of stream iterators in Section 2. Section 3 introduces punctuation semantics and Section 4 discusses our stream iterators model. Section 5 presents the theory behind punctuated data streams. Section 6 discusses concerns regarding punctuated streams. Section 7 gives our experiences and results of modifying relational operators in a query engine to use punctuated streams. We discuss related work in Section 8 and conclude in Section 9.

2 Streams and Stream Iterators

We first deal with streams and stream iterators without punctuations. For the moment, we will represent streams as infinite sequences, like the usual `cons`-based formulation of lists, but with no `nil` list. Thus, a stream over elements of type `T` can be defined as $\texttt{Stream(T)} = \texttt{T} \oplus \texttt{Stream(T)}$, where $\oplus$ is an infix constructor.

We use $\{|\ldots|\}$ to denote stream values, to distinguish them from finite lists. Thus, for `Stream(Int)`, we write $1 \oplus 3 \oplus 5 \oplus 7 \oplus \ldots$ as $\{|1, 3, 5, 7, \ldots|\}$. We write `S[i]` for the function that extracts the first `i` elements from a stream `S`, with type $\texttt{Stream(T)} \times \texttt{Int} \rightarrow \texttt{List(T)}$. Thus, `S[3]` is `[1, 3, 5]`. Further, for $\texttt{n} > \texttt{i}$, we use $\texttt{S}[\texttt{i} \rightarrow \texttt{n}]$ for the list of elements from $\texttt{i} + 1$ to `n`. Note that $\texttt{S}[\texttt{i}] = \texttt{S}[0 \rightarrow \texttt{i}]$. We use `S@i` to mean the `i`th element of the stream. We use $\otimes$ to construct streams from a finite list and another stream. Thus, `[2, 4, 6]` $\otimes$ `S` means $2 \oplus 4 \oplus 6 \oplus \texttt{S}$.

2.1 Stream Iterators

We do not want to use arbitrary stream-to-stream functions for operating on data streams. In particular, we want to avoid formulations that must access the entire stream at once. Thus, we introduce stream iterators that access the input incrementally. Function $\texttt{f} : \texttt{Stream(T)} \rightarrow \texttt{Stream(U)}$ is a *stream iterator* if there exists $\texttt{q} : \texttt{List(T)} \rightarrow \texttt{List(U)}$ such that for any $\texttt{S} \in \texttt{Stream(T)}$,

$$\texttt{f(S)} = (\texttt{q(S[1])} \otimes \texttt{q(S[2])} \otimes \ldots \otimes \texttt{q(S[i])} \otimes \ldots).$$

That is, `f` is a stream iterator if it can be defined as repeated application of `q` over all finite prefixes of the input stream. For example, select can be expressed as a stream iterator. Given predicate `p`, we define `q` by:

$$\texttt{q(L} + + \texttt{[a])} = \texttt{[a]}\ \texttt{if}\ \texttt{p(a)},\ \texttt{[\,]}\ \texttt{otherwise}.$$

That is, `q` just looks at the last element of each prefix and emits it if it satisfies the predicate ("++" is list concatenation). There is also a stream iterator for duplicate elimination (which we will henceforth refer to as dupelim), where `q` is:

$$\texttt{q(L} + + \texttt{[a])} = \texttt{[a]}\ \texttt{if}\ \neg\texttt{elem(a, L)},\ \texttt{[\,]}\ \texttt{otherwise}.$$

That is, `q` checks that the final element in the prefix does not appear earlier in the prefix. However, sort and group-by cannot be expressed as stream iterators in our formulation, as their results cannot be determined with only a prefix of the input.

2.2 Representation Issues

The model of a stream as an infinite sequence has an attractive simplicity to it and is directly supported in languages with lazy evaluation, such as Haskell. However, we found it had certain limitations for stream iterators. Those limitations concerned both modeling capabilities and pragmatic programming issues.

Representation of finite streams: A stream that actually has only a finite number of elements in it is not directly representable as an infinite sequence. We considered various alternatives, such as "stalling" after the last input element, padding the sequence with an infinite number of null values, or using both finite and infinite sequences. All of these options tended to complicate the programming model for streams.

Independence of iterator and stream rates: We do not want to build any assumption into our model that operator iterators are synchronized with stream arrival. There

may be multiple arrivals between iterator steps. More problematically, there may be no stream elements arriving at a given step. With an infinite sequence representation, no available input means blocking on its arrival (creating problems for operators with multiple inputs) or requiring a change in the sequence interface.

Interleaving of multiple units: Just as we do not want to assume an input stream is synchronized with operator iterations, different inputs need not necessarily proceed "in step." "Desynchronizing" multiple inputs requires some kind of extension to infinite sequences as input to ensure fairness and avoid blocking on one stream when input is available on the other.

To deal with these difficulties, we model a data stream as an infinite sequence of finite lists of elements—sort of a "sliced list." For example, the sequence $\{|1,2,3,4,5,\ldots|\}$ might appear in sliced form as $\{|[1,2],[3],[\,],[4,5],\ldots|\}$ or as $\{|[1,2,3],[4,5],\ldots|\}$ or in many other forms.

We still use `S[i]` to represent a finite prefix of a stream, but now it refers to the first `i` slices. So, if

$$S = \{|[1,2],[3],[],[4,5],[6],\ldots|\},$$

then $S[4] = [1,2,3,4,5]$. Stream iterators with multiple inputs take slices from each input in turn. Thus, an iterator with `n` inputs will consume $n * i$ slices after completing stage `i`.

This sliced representation overcomes the problems mentioned above. It can model a finite stream using a trailing sequence of empty slices: $\{|[1,2],[3],[\,],[\,],[\,],\ldots|\}$. The slices model the variability in stream arrivals relative to iterator steps. Finally, it can capture different interleavings of multiple inputs with alternative slicings of the streams. We use this representation mainly for modeling stream iterators and reasoning about them. In our prototype (see Section 7), operators use nonblocking reads from buffers to retrieve data.

We did consider a variety of other representations, but they all seemed to add complexity without discernable improvement in modeling stream behavior or iterator implementation. Interleaving inputs into a single sequence requires tagging elements for which input they belong to, and development of a notion of "fair merge" of streams. We considered adding an extra input sequence to each iterator that could indicate which stream sequence to access next or where input was available, but that approach led to considerable complication in iterator implementation. Labeling stream elements with an arrival time proved to be essentially equivalent to sliced streams. We also considered capturing input interleaving inside iterators rather than in the data stream representation. Strict element-at-a-time alternation was unrealistic. Deciding which input to read from next by a random choose function leads to problems in repeatability.

The change to stream iterators is to not provide the entire stream prefix at each iteration. Some iterators, such as select, only require the most recent input element to determine the next output. Other iterators only need a summary of the prefix. For example, dupelim only needs the distinct values in the prefix. Thus, we let iterators keep state from iteration to iteration. That state might be empty, or all input seen to that point, or some summary. The explicit representation of state also points up that there exist table operators with stream analogues, but where the stream iterator must keep track of arbitrary amounts of the prefix, such as dupelim. Keeping state does not change the set of expressible stream functions. Any function in the prefix version can be converted to the stateful version and vice versa.

3 Punctuation Semantics

A punctuation can be seen as a predicate on stream elements that must evaluate to `false` for every element following the punctuation. Thus, we might represent punctuations as "black box" Boolean functions. We instead represent punctuations as data to allow their easy storage, searching, and manipulation. There are many choices for what punctuation to represent. The scheme in our current implementation is fairly simple, but has the important property that the "and" of any two punctuations is also a punctuation.

Our stream elements are tuples of scalars and punctuations for such tuples. A *punctuation* is an ordered set of *patterns*, each pattern corresponding to an attribute of a tuple. We define five kinds of patterns and the values they match as:

- A wildcard, denoted as $*$, matches all values.
- A constant is a single value and matches only that value.
- A range is denoted with $[a,b]$ for inclusive ranges or (a,b) for exclusive ranges and matches values that fall in the range.
- A list is denoted as $\{a,b,c\}$ and matches values in the list.
- The empty pattern, denoted as $\emptyset$, does not match any values.

A punctuation `p` thus describes a set of tuples where, for each tuple `t`, value `t.A` matches pattern `p.A` for every attribute `A`. In Example 2, if the bid structure is:

$$< \texttt{itemid}, \texttt{increase}, \texttt{buyerid} >,$$

the punctuation $< \{1001, 2004\}, *, * >$ describes all bids on items 1001 and 2004.

3.1 Manipulating Punctuations

We define two basic functions for manipulating punctuations in streams. The `match` function takes a tuple `t` and a punctuation `p`, and returns `True` if `t` belongs to the subset described by `p`. The `combine` function takes two punctuations and returns a new punctuation that is either the intersection of the two inputs (if the sets they describe intersect) or the special punctuation Ψ, indicating that no nonempty combination exists. Thus,

$$\texttt{match}(t, (\texttt{combine}(p_1, p_2))) \Leftrightarrow \texttt{match}(t, p_1) \wedge \texttt{match}(t, p_2).$$

We define auxiliary functions for single and lists of inputs based on `match` and `combine`, as defined in Table 1. Additionally, since our input is now a mixture of tuples and punctuations, we need two functions to split them out. The function `tups` takes a list of stream elements and returns only the tuples, and `puncts` takes a list of stream elements and returns only the punctuations in its input.

A *punctuated stream* is a data stream that contains additional information describing a (possibly empty) subset of data over the domain of the stream. A punctuated stream `S` is *grammatical* if for all `i`, for all $j > i$, if $p \in S[i]$ and $t \in S[i \rightarrow j]$, then $\texttt{nomatch}(t,p)$. Stream sources and stream iterators must output grammatical streams.

TABLE 1
Functions Built from the Basic Functions, where t is a Tuple, ts is a Finite List of Tuples, p is a Punctuation, and ps is a Finite List of Punctuations

Function	Return Value
nomatch(t,p)	$\neg$ match(t,p)
setMatch(t,ps)	True if t matches any p $\in$ ps
setNomatch(t,ps)	True if t matches no p $\in$ ps
setMatchTs(ts,ps)	all t $\in$ ts that match any p $\in$ ps
setNomatchTs(ts,ps)	all t $\in$ ts matching no p $\in$ ps
setNomatchPs(ts,ps)	all p $\in$ ps with no match $\in$ ts
setCombine(ps1,ps2)	all non-Ψ combinations of punctuations $\in$ ps1 and ps2

3.2 Puncutation Behaviors: Pass, Propagate, and Keep

We have found it helpful to separate three aspects of stream iterator behavior, in the presence of punctuated streams:

Pass behavior. Tuples that can be output based on punctuation received.

Propagate behavior. Punctuations that can be properly output based on punctuation (and possibly tuples) received.

Keep behavior. The portion of state that must be retained based on punctuation.

These three behaviors manifest themselves in our work in two ways. First, in Section 4, we present a generic framework for punctuated-stream iterators. In this framework, we show that the specific behavior of each iterator can be defined using *behavior functions*, called pass, prop, and keep. Behavior functions are called repeatedly as the iterator works through its input(s). Second, in Section 5, we define invariants that specify the cumulative behavior of an iterator enhanced for punctuated streams. We call these formal invariants cpass, cprop, and ckeep. We use these invariants to prove that our implementations of stream iterators are reasonable counterparts to relational operators.

4 Our Stream Processing Framework

We initially made ad hoc extensions to the Niagara query engine [18] to test the feasibility of our approach (see Section 7). After encouraging results, we abstracted the behavior of stream iterators into a high-level model. The result is a common control structure for stream iterators, customized by plugging in functions specific to each iterator. We chose the Haskell programming language [14], a *lazy evaluation* functional language, to express our semantics, though any other languages or formalisms could have been used. We chose Haskell for three reasons: First, lazy evaluation languages do not evaluate expressions until required. Functions that take streams as arguments evaluate items from the stream only as needed. Thus, they are very suitable for modeling stream processing systems and have been used to model streams elsewhere [12], [15], [20]. This behavior is similar to Graefe's iterator function next [11], which retrieves only the next item from its input. Second, since functions are first-class objects, they can be arguments to other functions. We can formulate the general behavior of all stream iterators in a single function and pass iterator specifics as function arguments to the general function. Finally, since it is a language with formal semantics (at least the part we use), it helps us prove our implementations conform to the appropriate punctuation behaviors.

It is appropriate to give a very brief explanation of the Haskell features used in our framework, defined in the standard Haskell library (prelude.hs). The fst function takes a tuple and returns the first value from the tuple. For example, fst$((1,2))$ returns the value 1. The infix function "++" concatenates two lists. The infix function "\\" removes items from its first argument that exist in its second argument. We also use the infix ":" operator, which takes an item and a list and adds the item to the head of the list. In this discussion, we modified the Haskell syntax slightly to better match the logic notation often used in database semantics. We explain other Haskell features as they are encountered.

4.1 A Formulation for Stream Iterators

Without punctuations, stream iterators closely resemble Parker's *stream transducers* [19]. A unary stream iterator is a triple (initial_state, step, final), where:

- initial_state is the iterator state before tuples arrive from the input.
- step is called when new data arrives. It takes the new tuples and the current state, and returns any new output tuples and a modified state.
- final is called when the stream ends. It takes the current state and returns any new tuples and a modified state.

The general behavior of unary stream iterators is modeled by the unary function, which takes a state variable (st), the step and final functions, and the input stream:

```
unary(st, step, final, []) = fst(final(st))
unary(st, step, final, (xs:rest)) =
      out ++ unary(new_st, step, final, rest)
      where (out,new_st) = step(xs, st)
```

Note there are two equations that define unary. Haskell uses pattern-matching to determine which of the equations to evaluate. If the fourth parameter to unary matches the pattern [], then the first equation is evaluated. In this case, we have reached the end of the input and output the tuples returned from final. If the fourth parameter is a nonempty list, then the second equation is evaluated. The first item in the list is matched to the variable xs and the rest of the list is matched to the variable rest. In this case, we call the step function with xs and the state. We output any results from the step function, then recursively call unary with the new state and the rest of the input. Note, we also take advantage of

Haskell's `where` syntax, which allows us to assign values to variables.

We can define dupelim in terms of `unary` as follows:

```
sdupelim(xs) =unary([], step, final, xs)
              where step(ts, st) =
                        ((δ (ts - st)), st ∪ ts)
                    final (tsSeen) = ([], [])
```

We maintain a list of all unique values that have arrived from the input so far in state (`st`), with the empty list as the initial state. The `step` function returns any tuples not currently in the state and a new state containing the union of new tuples and tuples in the original state. The `final` function clears the state. We use δ to mean duplicate elimination over a finite list, as per Albert [1]. That is, given some finite list A, $\delta(A) = \{t | t \in A\}$.

Let us see how `sdupelim` works with an example. Suppose we have the input stream

$$\mathtt{S} = \{|[1,5],[3],[],[5,6,7],\ldots|\}.$$

To evaluate `sdupelim(S)`, each slice of the stream and the current state will be passed to the `step` function as follows:

function	input	state	output	newstate
step	[1,5]	[ ]	[1,5]	[1,5]
step	[3]	[1,5]	[3]	[1,5,3]
step	[ ]	[1,5,3]	[ ]	[1,5,3]
step	[5,6,7]	[1,5,3]	[6,7]	[1,5,3,6,7]
...				

A binary stream iterator is defined as a five-tuple: (`initial_state`, `stepL`, `stepR`, `finalL`, `finalR`). The meanings of `stepL` and `stepR` are the same as `step` for a unary operator, defined for each input. Likewise, `finalL` and `finalR` are the same as `final`, defined for each input. The formal definition of `binary` is similar to `unary` and is omitted. We can implement stream difference (`sdiff`) using `binary`:

```
sdiff(lxs,rxs) =
  sdupelim(binary((False,[],[]), stepL,
  finalL, stepR, finalR))
  where stepL(ts,(False, xs, ys)) =
             ([], (False, ts ∪ xs, ys))
      stepL(ts,(True, xs, ys)) = ((ts - ys),
           (True, [], ys))
      stepR(ts,(fYs, xs, ys)) = ([], (fYs, xs,
            ts ∪ ys))
      finalL(f, xs, ys) = ([], (f, xs, ys))
      finalR(f, xs, ys) =(xs - ys, (True, [],
            ys))
```

We maintain a structure with a Boolean and two lists of tuple in state. The Boolean is set to `True` when the negative input reaches its end. The two lists store tuples from each input. We see from `stepL` and `stepR` that `sdiff` blocks on its negative input—results are emitted only after the negative input has been read completely. Additionally, `sdiff` maintains an unbounded amount of state. The two lists grow as more input arrives.

4.2 Extension to Punctuated Streams

To enhance stream iterators for punctuated streams, we define three new behavior functions: `pass`, `prop`, and `keep`, and we redefine unary stream iterators to (`initial_state`, `step`, `pass`, `prop`, `keep`), where:

- `initial_state` and `step` have the same meaning as before.
- `pass` takes new punctuations and state, and returns any additional tuples that can be output based on the punctuations.
- `prop` takes new punctuations and state, and returns punctuations to be output.
- `keep` takes new punctuations and state, and returns a modified state.

Changes to `unary` and `binary` for punctuated streams are similar, so we focus on `unary`. Input slices may now contain both tuples and punctuation, so they are first separated. Then, `step` is called with new tuples and the state, followed by calls to `pass`, `prop`, and `keep` with new punctuations and the state. The execution order of the punctuation functions is important: The `prop` function could output punctuation that matches tuples output by `pass`, so `pass` must be executed before `prop`. Further, `keep` must follow `pass` and `prop` since both depend on state that `keep` might modify. Note `final`, called when the input stream ended, was removed. Clearly `final` performs the equivalent tasks to `pass`, `prop`, and `keep` combined at the end of stream.

Some stream iterators do not need to implement particular behavior functions. For example, dupelim is not a blocking operator. It does not need to implement a special `pass` function. We define trivial behavior functions for cases such as this:

$$\mathtt{passT(ps, st)} = [\,]\,;\ \mathtt{propT(ps, st)} = [\,]\,;\ \mathtt{keepT(ps, st)} = \mathtt{st}$$

We redefine `sdupelim` to exploit punctuations as follows:

```
sdupelim(xs) = unary([],step,passT,
               prop,keep,xs)
               where step(ts,st) = ((δ
                         (ts - st)), st ∪ ts)
                     prop(ps,st) = ps
                     keep(ps,st) = st -
                           setMatchTs(st,ps)
```

In the original version, we kept all unique tuples that had arrived. However, punctuations tell us what tuples will never again appear in the stream. Any tuples in state that we know can have no more duplicate values in the stream can be removed, and that is what the `keep` function does. The `prop` function propagates all punctuation as it arrives.

Let us revisit `sdupelim` with the same example enhanced with punctuations. We add a punctuation that no elements follow between 0 and 4, so

$$\mathtt{S} = \{|[1,5],[3],[\mathtt{P}(0,4)],[5,6,7],\ldots|\}.$$

In this illustration, we omit calls to `passT`.

function	input	state	output	newstate
step	[1,5]	[ ]	[1,5]	[1,5]
prop	[ ]	[1,5]	[ ]	
keep	[ ]	[1,5]		[1,5]
step	[3]	[1,5]	[3]	[1,5,3]
prop	[ ]	[1,5,3]	[ ]	
keep	[ ]	[1,5,3]		[1,5,3]
step	[ ]	[1,5,3]	[ ]	[1,5,3]
prop	[P(0,4)]	[1,5,3]	[P(0,4)]	
keep	[P(0,4)]	[1,5,3]		[5]
step	[5,6,7]	[5]	[6,7]	[5,6,7]
prop	[ ]	[5,6,7]	[ ]	
keep	[ ]	[5,6,7]		[5,6,7]
	...			

For the first two slices of input, execution follows the nonpunctuated case. Since no punctuations arrive during the first two slices, the prop and keep functions have no effect. When a punctuation arrives, however, we see new behavior. The prop function receives the new punctuation and returns it. The keep function receives the punctuation and keeps only tuples that do not match the punctuation in the new state.

Now, consider the enhanced version of sdiff with new behavior functions:

$$\begin{aligned}
&\texttt{sdiff}(\texttt{lxs},\texttt{rxs}) = \texttt{sdupelim}(\texttt{binary}(([\,],[\,],[\,],[\,]),\\
&\qquad\qquad \texttt{stepL},\texttt{passT},\texttt{propT},\texttt{keepL},\\
&\qquad\qquad \texttt{stepR},\texttt{passR},\texttt{propR},\texttt{keepR},\texttt{lxs},\texttt{rxs}))\\
&\quad \texttt{where } \texttt{stepL}(\texttt{ts},(\texttt{lts},\texttt{rts},\texttt{lps},\texttt{rps})) =\\
&\qquad\qquad ([\,],(\texttt{ts}+\!+\texttt{lts},\texttt{rts},\texttt{lps},\texttt{rps}))\\
&\qquad \texttt{stepR}(\texttt{ts},(\texttt{lts},\texttt{rts},\texttt{lps},\texttt{rps})) =\\
&\qquad\qquad ([\,],(\texttt{lts},\texttt{ts}+\!+\texttt{rts},\texttt{lps},\texttt{rps}))\\
&\qquad \texttt{passR}(\texttt{ps},(\texttt{lts},\texttt{rts},\texttt{lps},\texttt{rps})) =\\
&\qquad\qquad \texttt{setMatchTs}((\texttt{lts}\setminus\!\setminus\texttt{rts}),(\texttt{ps}+\!+\texttt{rps}))\\
&\qquad \texttt{propR}(\texttt{ps},(\texttt{lts},\texttt{rts},\texttt{lps},\texttt{rps})) =\\
&\qquad\qquad \texttt{setCombine}(\texttt{lps},(\texttt{ps}+\!+\texttt{rps}))\\
&\qquad \texttt{keepL}(\texttt{ps},(\texttt{lts},\texttt{rts},\texttt{lps},\texttt{rps})) =\\
&\qquad\qquad (\texttt{lts},\texttt{rts},\texttt{ps}+\!+\texttt{lps},\texttt{rps})\\
&\qquad \texttt{keepR}(\texttt{ps},(\texttt{lts},\texttt{rts},\texttt{lps},\texttt{rps})) =\\
&\qquad\qquad (\texttt{ltsNew},\texttt{rtsNew},\texttt{lps},\texttt{ps}+\!+\texttt{rps})\\
&\qquad\quad \texttt{where } \texttt{ltsNew} = \texttt{setNomatchTs}((\texttt{lts}\setminus\!\setminus\texttt{rts}),\\
&\qquad\qquad\qquad\qquad (\texttt{ps}+\!+\texttt{rps}))\\
&\qquad\qquad\quad \texttt{rtsNew} = \texttt{setNomatchTs}(\texttt{rts},\texttt{lps})
\end{aligned}$$

The data structure for state has changed. We no longer need a Boolean value to tell us when we have reached the end of the negative input. Punctuations tell us this. However, we do need to maintain two additional lists for punctuations from each input. We can see that the stepL and stepR functions still do not produce output. However, passR outputs tuples before the end of the stream. Tuples can be emitted from the positive input that match punctuation from the negative input and have not appeared so far from the negative input. Thus, sdiff is no longer blocking. The propR function only outputs certain punctuations. We cannot simply output punctuations as they arrive, as we did for sdupelim. Instead, we output punctuations that are nonempty combinations of punctuations received from each input. Finally, keepR decreases the amount of state required when punctuations arrive. We only keep tuples that do not match punctuations from the opposite input.

5 Correctness of Punctuated Iterators

There is a significant issue we have not yet addressed, namely: When is a stream iterator a "reasonable" counterpart of the corresponding relational operator? We hope the examples presented thus far seem sensible, but clearly, one can define stream versions of operators with obviously inappropriate behavior. We use three kinds of invariants to specify how a stream iterator should process punctuation: *pass invariants*, *propagation invariants*, and *keep invariants*. These invariants are defined below in terms of the allowable action on any given prefix of the input stream(s). We use the invariants in our proofs that specific stream iterators satisfy correctness conditions.

The correspondence between cpass and ckeep invariants and pass and keep functions is not quite direct. Our framework is based upon the definitions for the nonpunctuated case, captured in the step function. Thus, the cpass and ckeep invariants actually specify the desired behavior of the pass and keep functions in combination with step.

5.1 Faithfulness and Propriety

We base our notions of correctness of a stream iterator (termed "faithfulness" and "propriety") upon its series of partial outputs after processing each slice of its input(s). The output of iterators at any point in the input should be consistent with any possible further input.

We first define faithfulness for streams without punctuation. Let f be a unary stream iterator from Stream(T) to Stream(U), and let g be a relational operator from List(T) to List(U). We say f is *faithful* to g if the following two conditions are met (note, we interpret $\subseteq$ as prefix or subset depending on whether order is significant for g or not):

Safety. For all streams S in Stream(T), for all subsets (prefixes) of S of size (length) i, and for every finite addition A in List(T), $\texttt{f(S)[i]} \subseteq \texttt{g(S[i]}+\!+\texttt{A)}$. That is, we never emit output unless we can be sure it will not conflict with any later input.

Completeness. For all streams S in Stream(T), for all subsets (prefixes) of S of size (length) i, and for all M, if $\texttt{M} \subseteq \texttt{g(S[i]}+\!+\texttt{A)}$ for all finite additions A in List(T), then $\texttt{M} \subseteq \texttt{f(S)[i]}$. That is, we always emit an output if it will necessarily be generated by the relational operator under any additional input, including no input.

The corresponding conditions for binary operators are similar:

$$\texttt{f(S1,S2)[2i]} \subseteq \texttt{g(S1[i]}+\!+\texttt{A1,S2[i]}+\!+\texttt{A2)}$$

for all A1 and A2, and $\texttt{M} \subseteq \texttt{f(S1,S2)[2i]}$ if $\texttt{M} \subseteq \texttt{g(S1[i]}+\!+\texttt{A1,S2[i]}+\!+\texttt{A2)}$. (Note the 2i index is because f will emit a slice of output for both the ith slices of S1 and S2.)

Every monotone relational operator g has a faithful stream counterpart. For unary operators, it is the iterator f where $\texttt{f(S)@i} = \texttt{g(S[i])} - \texttt{g(S[i}-1\texttt{])}$. Thus, $\texttt{f(S)[i]} = \texttt{g(S[i])}$, satisfying the two conditions.

With punctuation, we modify the conditions of faithfulness such that any possible additional input is constrained to obey any punctuation already seen.

TABLE 2
Pass Invariants for Traditional Query Operators

Op	Pass Invariant
difference	[t | t ∈ ts_1 ∧ t ∉ ts_2 ∧ setMatch(t, ps_2)]
group-by	[t | t ∈ ts_1 ∧ setMatch(t, (groupPs(ps_1)))]
sort	[t | t ∈ ts_1 ∧ setMatch(t, (init($Sort_A$, ps_1)))]

Safety. For all S in Stream(T), for all subsets (prefixes) of S of size (length) i, and for every possible finite addition A in List(T) such that setMatchTs(A, puncts(S[i])) = ∅, tups(f(S)[i]) ⊆ g(tups(S[i]) + + A).

Completeness. For all S in Stream(T), for all subsets (prefixes) of S of size (length) i, and for all M, if for all possible finite additions A in List(T) such that setMatchTs(A, puncts(S[i])) = ∅ and M ⊆ g(tups(S[i]) + + A), then M ⊆ tups(f(S)[i]).

The definition for binary operators is extended in a similar way. We will provide correctness proofs for some stream iterators later in this section.

The other condition we want to enforce on our stream operators is that they are well-behaved with respect to punctuation. We say that a stream iterator f is *proper* if f(S) is guaranteed to be grammatical whenever S is grammatical. Note that propriety is not a very strong condition. An iterator that emits no punctuation will always be proper. We are currently investigating stronger notions regarding punctuation, such as requiring an iterator to emit the *maximal* amount of punctuation that can be inferred from the input punctuation.

5.2 Pass Invariants

Pass invariants define the allowed output for a stream iterator after seeing a prefix of the input. Pass invariants have the form:

$$\texttt{tsOut} = \texttt{cpass}(\texttt{ts}_1, \texttt{ps}_1, \ldots, \texttt{ts}_n, \texttt{ps}_n),$$

where tsOut represents tuples that can be output, given tuples $\texttt{ts}_j$ and punctuations $\texttt{ps}_j$ that have arrived from the jth input so far. Table 2 lists some nontrivial pass invariants.

Two pass invariants require further explanation. The group-by iterator can output a result when all tuples have been received for that group. Some kinds of punctuations determine which groups are complete. The groupPs function in the group-by pass invariant returns those kinds of punctuations. In the auction example, we group on items. If a punctuation arrives that says all tuples have arrived for items #1001 and #1004, then results for those groups can be output. However, if a punctuation arrives that says all tuples have arrived for seller #9932, no results can be output.

In order for sort to output correct results early, we need to know when the values that appear first in sort order have arrived. Punctuations that match a prefix of the final sorted output give us this information. The init function returns punctuations based on the sort order and the input punctuations such that no tuple can still arrive that would be sorted before tuples that match those punctuations. In the warehouse example, suppose we have a query that sorts its input on the hour attribute in ascending order. If we receive a punctuation that reads $< *, [2, 4], *, * >$ (recall the schema is $<$ sid, hour, minute, currtmp $>$), then we cannot output anything. If we later receive a punctuation that reads $< *, [0, 3], *, * >$, then we can output results through the fourth hour. We know by the attribute's type that 0 is the first value for the sorted output. The init function, given the two punctuations shown above, will return $< *, [0, 4], *, * >$.

TABLE 3
Propagation Invariants for Traditional Query Operators

Op	Propagation Invariant
select	$\texttt{ps}_1$
project	projPs($\texttt{ps}_1$)
dupelim	$\texttt{ps}_1$
join	($\texttt{modPs}_1$(setNomatchPs($\texttt{ts}_1$, $\texttt{ps}_1$))) ++ ($\texttt{modPs}_2$(setNomatchPs($\texttt{ts}_2$, $\texttt{ps}_2$)))
merge	setCombine($\texttt{ps}_1$, $\texttt{ps}_2$)
intersect	setCombine($\texttt{ps}_1$, $\texttt{ps}_2$)
difference	setCombine($\texttt{ps}_1$, $\texttt{ps}_2$)
group-by	groupPs($\texttt{ps}_1$)
sort	init($Sort_A$, $\texttt{ps}_1$)

5.3 Propagation Invariants

Propagation invariants specify what punctuations can be output by a stream iterator. Propagation invariants have the form:

$$\texttt{psOut} = \texttt{cprop}(\texttt{ts}_1, \texttt{ps}_1, \ldots, \texttt{ts}_n, \texttt{ps}_n),$$

where psOut represents punctuations that can be output given $\texttt{ts}_j$ and $\texttt{ps}_j$. Propagation invariants assume the corresponding pass invariant has already been satisfied. Table 3 lists propagation invariants. We consider union to be a combination of merge and dupelim.

The propagation invariant for project outputs punctuations returned by projPs. Output punctuations must have the same structure as output tuples. If the stream iterator is projecting out attributes, then the same attributes must be projected out of punctuations. However, we cannot simply modify all punctuations. If an attribute being projected out contains a pattern that is not the wildcard, then the punctuation cannot be output. (We are developing a more informative propagation invariant that attempts to combine patterns in projected-away attributes to get a wildcard.) The invariant for join uses $\texttt{modPs}_1$ and $\texttt{modPs}_2$. They are similar, so we focus on $\texttt{modPs}_1$. Like project, the structure of output punctuations must be modified to match output tuples. The $\texttt{modPs}_1$ function modifies tuples from one join input, appending attributes from the other input with wildcards.

5.4 Keep Invariants

Keep invariants specify the state a stream iterator must preserve, which will be a subset of the state preserved in the nonpunctuated case. Unlike pass invariants, we specify one invariant for each input of an iterator. They have the form:

$$\texttt{tsOut} = \texttt{ckeep}_j(\texttt{ts}_1, \texttt{ps}_1, \ldots, \texttt{ts}_n, \texttt{ps}_n),$$

where $\texttt{ckeep}_j$ specifies the state kept for the jth input given $\texttt{ts}_j$ and $\texttt{ps}_j$ are as defined before. Table 4 lists nontrivial keep invariants.

The keep invariant for join uses a function called joinPs. It returns punctuations with wildcards for all attributes not participating in the join. Thus, join only keeps tuples that do no match punctuations from the other input in states that describe only the join attribute(s).

5.5 The Minimality Condition

In addition to the specific behavior invariants for each kind of operator, we assume an additional condition on iterator implementations. The *minimality condition* states that an iterator never produces more output than needed to satisfy its behavior invariants. This condition is necessary to

TABLE 4
Keep Invariants for Traditional Query Operators

Op	Keep Invariant
dupelim	[t|t∈ts ∧ setNomatch(t,ps)]
$join_1$	[t|t∈ts_1 ∧ setNomatch(t, joinPs(ps_2))]
$join_2$	[t|t∈ts_2 ∧ setNomatch(t, joinPs(ps_1))]
$intersect_1$	[t|t∈ts_1 ∧ setNomatch(t,ps_2)]
$intersect_2$	[t|t∈ts_2 ∧ setNomatch(t,ps_1)]
$difference_1$	[t|t∈ts_1 ∧ t ∉ ts_2 ∧ setNomatch(t,ps_2)]
$difference_2$	[t|t∈ts_2 ∧ setNomatch(t,ps_1)]
group-by	[t|t∈ts ∧ setNomatch(t, groupPs(ps))]
sort	[t|t∈ts ∧ setNomatch(t, init($Sort_A$,ps))]

prevent an operator from gratuitously repeating an output tuple after punctuation matching that tuple has been emitted. We note that it is generally straightforward to prove that iterators defined in our framework observe the minimality condition. We henceforth assume that all stream iterators satisfy the minimality condition.

5.6 Proving Correctness of Stream Iterators

We prove sdupelim is faithful and proper and we provide a portion of our proof for sdiff. Our general strategy for such proofs is to specify cumulative invariants and show that any stream iterators that obey these cumulative invariants are faithful and proper, then show that our particular iterator implementation obeys the cumulative invariants. This two-stage approach allows us some flexibility in the implementation of our iterators. We only have to prove once that any iterator satisfying the invariants is faithful and proper relative to its relational counterparts. Then, for each implementation of a particular iterator, we only have to prove it conforms to the invariants for that iterator to have a complete proof.

Theorem 1. *The stream iterator sdupelim is proper and faithful to dupelim.*

Proof. Consider the cumulative invariants for dupelim:

$$\begin{aligned}\texttt{cpass}(\texttt{ts},\texttt{ps}) &= \delta(\texttt{ts})\\ \texttt{cprop}(\texttt{ts},\texttt{ps}) &= \texttt{ps}\\ \texttt{ckeep}(\texttt{ts},\texttt{ps}) &= \texttt{ts} - \texttt{setMatch}(\texttt{ts},\texttt{ps})\end{aligned}$$

We will first show that any stream iterator that conforms to these invariants and the minimality condition is faithful and proper to dupelim. We then show that the particular implementation of sdupelim given in Section 4.2 conforms to the invariants. For this proof, it is useful to denote the tuples and punctuations present in the first i slices of input. We use the notation $\texttt{ts}_i = \texttt{tups}(\texttt{S}[\texttt{i}])$ and $\texttt{ps}_i = \texttt{puncts}(\texttt{S}[\texttt{i}])$, where S is the input stream. Also, for $\texttt{j} > \texttt{i}$, let $\texttt{ts}_{ij} = \texttt{tups}(\texttt{S}[\texttt{i} \rightarrow \texttt{j}])$.

As noted earlier, any monotone relational operator such as dupelim has a faithful stream counterpart. If $\texttt{ckeep}(\texttt{ts},\texttt{ps}) = \texttt{ts}$, we would have the "standard" faithful version. We need to show for faithfulness that retaining less state does not change the output. Consider what is output between two points i and j (assuming minimality):

$$\begin{aligned}\texttt{cpass}(\texttt{ts}_j) - \texttt{cpass}(\texttt{ts}_i) &= \delta(\texttt{ts}_j) - \delta(\texttt{ts}_i)\\ &= \delta(\texttt{ts}_j - \texttt{ts}_i)\\ &= \delta(\texttt{ts}_{ij} - \texttt{ts}_i),\\ &\quad \text{since } \texttt{ts}_j = \texttt{ts}_{ij} + + \texttt{ts}_i.\end{aligned}$$

This equivalent expression indicates how state is used in an iterator implementation: Past input is kept and used to filter subsequent input. So, the question is, what is output if our state is per ckeep:

$$\begin{aligned}&\delta(\texttt{ts}_{ij} - \texttt{ckeep}(\texttt{ts}_i,\texttt{ps}_i))\\ &\quad= \delta(\texttt{ts}_{ij} - (\texttt{ts}_i - \texttt{setMatchTs}(\texttt{ts}_i,\texttt{ps}_i)))\\ &\quad= \delta((\texttt{ts}_{ij} - \texttt{ts}_i) \cup (\texttt{ts}_{ij} \cap \texttt{ts}_i \cap \texttt{setMatchTs}(\texttt{ts}_i,\texttt{ps}_i)))\\ &\qquad[\text{since } \texttt{A} - (\texttt{B} - \texttt{C}) = (\texttt{A} - \texttt{B}) \cup (\texttt{A} \cap \texttt{B} \cap \texttt{C})]\\ &\quad= \delta((\texttt{ts}_{ij} - \texttt{ts}_i) \cup \emptyset)\\ &\qquad[\text{since } \texttt{ts}_{ij} \text{ and } \texttt{setMatchTs}(\texttt{ts}_i,\texttt{ps}_i)\\ &\qquad\text{can have no tuples in common}]\\ &\quad= \delta(\texttt{ts}_{ij} - \texttt{ts}_i).\end{aligned}$$

This value is the same as the standard version, so reducing state per ckeep does not affect faithfulness.

For propriety, we see that punctuations emitted by stage i are all the punctuations received, namely $\texttt{ps}_i$. Can a tuple t such that $\texttt{setMatch}(\texttt{t},\texttt{ps}_i)$ be emitted after stage i? If so, it is emitted by some stage $\texttt{j} > \texttt{i}$. So, it must be that $\texttt{t} \in \texttt{cpass}(\texttt{ts}_j,\texttt{ps}_j)$. So, $\texttt{t} \in \texttt{ts}_j$. However, $\texttt{t} \notin \texttt{ts}_{ij}$, by grammaticality of the input. Hence, $\texttt{t} \in \texttt{ts}_i$, and $\texttt{t} \in \texttt{cpass}(\texttt{ts}_i,\texttt{ps}_i)$. So, t must already be emitted at stage i, and by the minimality condition, will not be output again later. Thus, the output of any iterator satisfying cpass and cprop is grammatical given grammatical input, and is therefore proper.

Conformance: We show that sdupelim conforms to the invariants, hence is faithful. First, we show that the output at each iteration i+1 (call it $\texttt{incrts}_{i+1}$) is equivalent to the pass rule at i+1 minus the pass rule for the previous iteration i. That is,

$$\texttt{incrts}_{i+1} = \texttt{cpass}(\texttt{ts}_{i+1},\texttt{ps}_{i+1}) - \texttt{cpass}(\texttt{ts}_i,\texttt{ps}_i).$$

Note the output tuples at any iteration are tuples returned by step and pass. The state value in our proof is denoted as $\texttt{st}_i$. New tuples and punctuations arriving at slice i are denoted as $\texttt{tsNew}_i$ and $\texttt{psNew}_i$, therefore

$$\texttt{ts}_i \cup \texttt{tsNew}_{i+1} = \texttt{ts}_{i+1}$$

and

$$\texttt{ps}_i \cup \texttt{psNew}_{i+1} = \texttt{ps}_{i+1}.$$

Pass:

$$\begin{aligned}&\texttt{incrts}_{i+1} = \texttt{step}(\texttt{tsNew}_{i+1},\texttt{st}_i) + + \texttt{pass}(\texttt{psNew}_{i+1},\texttt{st}_i)\\ &\quad= \delta(\texttt{tsNew}_{i+1} - \texttt{st}_i) + + [\,]\\ &\quad= \delta(\texttt{tsNew}_{i+1} - (\texttt{ts}_i - \texttt{setMatchTs}(\texttt{ts}_i,\texttt{ps}_i)))\\ &\quad= \delta((\texttt{tsNew}_{i+1} - \texttt{ts}_i) \cup (\texttt{tsNew}_{i+1} \cap \texttt{ts}_i \cap\\ &\qquad\texttt{setMatchTs}(\texttt{ts}_i,\texttt{ps}_i)))\\ &\qquad[\text{since } \texttt{A} - (\texttt{B} - \texttt{C}) = (\texttt{A} - \texttt{B}) \cup (\texttt{A} \cap \texttt{B} \cap \texttt{C})]\\ &\quad= \delta((\texttt{tsNew}_{i+1} - \texttt{ts}_i) \cup \emptyset)\\ &\qquad[\text{since } \texttt{tsNew}_{i+1} \text{ and } \texttt{setMatchTs}(\texttt{ts}_i,\texttt{ps}_i)\\ &\qquad\text{have no common tuples}]\\ &\quad= \delta(\texttt{tsNew}_{i+1} - \texttt{ts}_i)\\ &\quad= \texttt{cpass}(\texttt{ts}_{i+1},\texttt{ps}_{i+1}) - \texttt{cpass}(\texttt{ts}_i,\texttt{ps}_i)\\ &\qquad[\text{from the equality shown earlier}].\end{aligned}$$

Showing conformance to cprop is trivial since sdupelim outputs punctuations as they arrive, and so is omitted. To prove conformance to ckeep, we show by induction that if the current state at any stage i equals $\texttt{ckeep}(\texttt{ts}_i, \texttt{ps}_i)$, then the result of $\texttt{keep}(\texttt{psNew}_{i+1}, \texttt{st}_i)$ is $\texttt{ckeep}(\texttt{ts}_{i+1}, \texttt{ps}_{i+1})$.

Base case: $\texttt{i} = 0$, so $\texttt{ts}_i = \texttt{ps}_i = [\,]$, and the initial state $\texttt{st}_0 = [\,]$. Then:

$$\begin{aligned}
&\texttt{keep}((\texttt{st}_0 \cup \texttt{tsNew}_1), \texttt{psNew}_1)\\
&\quad = \texttt{keep}(([\,] \cup \texttt{tsNew}_1), ([\,] \cup \texttt{psNew}_1))\\
&\quad = \texttt{keep}((\texttt{ts}_0 \cup \texttt{tsNew}_1), (\texttt{ps}_0 \cup \texttt{psNew}_1))\\
&\quad = \texttt{keep}(\texttt{ts}_1, \texttt{ps}_1)\\
&\quad = \texttt{ts}_1 - \texttt{setMatchTs}(\texttt{ts}_1, \texttt{ps}_1)\\
&\quad = \texttt{ckeep}(\texttt{ts}_1, \texttt{ps}_1).
\end{aligned}$$

Induction step: Assume $\texttt{st}_i = \texttt{ckeep}(\texttt{ts}_i, \texttt{ps}_i)$. Show that $\texttt{keep}(\texttt{psNew}_{i+1}, \texttt{st}_i) = \texttt{ckeep}(\texttt{ts}_{i+1}, \texttt{ps}_{i+1})$.

$$\begin{aligned}
&\texttt{keep}(\texttt{psNew}_{i+1}, (\texttt{st}_i \cup \texttt{tsNew}_{i+1}))\\
&\quad = (\texttt{st}_i \cup \texttt{tsNew}_{i+1}) - \texttt{setMatchTs}((\texttt{st}_i \cup \texttt{tsNew}_{i+1}),\\
&\qquad \texttt{psNew}_{i+1})\\
&\quad = (\texttt{ckeep}(\texttt{ts}_i, \texttt{ps}_i) \cup \texttt{tsNew}_{i+1}) -\\
&\qquad \texttt{setMatchTs}((\texttt{ckeep}(\texttt{ts}_i, \texttt{ps}_i) \cup \texttt{tsNew}_{i+1}), \texttt{psNew}_{i+1})\\
&\quad = (\texttt{ts}_{i+1} - \texttt{setMatchTs}(\texttt{ts}_i, \texttt{ps}_i)) -\\
&\qquad \texttt{setMatchTs}((\texttt{ts}_{i+1} - \texttt{setMatchTs}(\texttt{ts}_i, \texttt{ps}_i)), \texttt{psNew}_{i+1})\\
&\qquad [\text{By properties of } \cup \text{ and } \cap]\\
&\quad = (\texttt{ts}_{i+1} - \texttt{setMatchTs}(\texttt{ts}_i, \texttt{ps}_i)) -\\
&\qquad \texttt{setMatchTs}(\texttt{ts}_{i+1}, \texttt{psNew}_{i+1}) - \texttt{setMatchTs}(\texttt{ts}_i, \texttt{ps}_i)\\
&\qquad [\text{Consequence of the definition of } \texttt{setMatchTs}]\\
&\quad = (\texttt{ts}_{i+1} - \texttt{setMatchTs}(\texttt{ts}_i, \texttt{ps}_i)) -\\
&\qquad \texttt{setMatchTs}(\texttt{ts}_{i+1}, \texttt{psNew}_{i+1})\\
&\quad = \texttt{ts}_{i+1} - (\texttt{setMatchTs}(\texttt{ts}_i, \texttt{ps}_i) \cup\\
&\qquad \texttt{setMatchTs}(\texttt{ts}_{i+1}, \texttt{psNew}_{i+1}))\\
&\qquad [\text{Consequence of the definition of } \texttt{setMatchTs}]\\
&\quad = \texttt{ts}_{i+1} - \texttt{setMatchTs}((\texttt{ts}_i \cup \texttt{ts}_{i+1}), (\texttt{ps}_i \cup \texttt{psNew}_{i+1}))\\
&\quad = \texttt{ts}_{i+1} - \texttt{setMatchTs}(\texttt{ts}_{i+1}, \texttt{ps}_{i+1})\\
&\quad = \texttt{ckeep}(\texttt{ts}_{i+1}, \texttt{ps}_{i+1}).
\end{aligned}$$

□

Theorem 2. *The stream iterator* sdiff *is faithful to the relational operator difference.*

We present here only the faithfulness part of our proof, which, unlike Theorem 1, is nontrivial. Our pass invariant for difference is:

$$\texttt{cpass}(\texttt{lts}, \texttt{lps}, \texttt{rts}, \texttt{rps}) = [\texttt{t} | \texttt{t} \in \texttt{lts} \wedge \texttt{t} \notin \texttt{rts} \wedge \texttt{setMatch}(\texttt{t}, \texttt{rps})].$$

Here, we use the "l" and "r" prefix to distinguish left and right inputs. Thus, $\texttt{lts}_i$ and $\texttt{lps}_i$ are analogs to $\texttt{ts}_i$ and $\texttt{ps}_i$ for the left input and, similarly, $\texttt{rts}_i$ and $\texttt{rps}_i$ for the right.

Faithfulness (safety): We need to show that at any stage i

$$\texttt{cpass}(\texttt{lts}_i, \texttt{lps}_i, \texttt{rts}_i, \texttt{rps}_i) \subseteq (\texttt{lts}_i{+}{+}\,\texttt{ls}) - (\texttt{rts}_i{+}{+}\,\texttt{rs})$$

for any Lists ls, rs where $\texttt{setMatchTs}(\texttt{ls}, \texttt{lps}_i) = [\,]$ and $\texttt{setMatchTs}(\texttt{rs}, \texttt{rps}_i) = [\,]$. Suppose t is a tuple in the left-hand side above. It must be that $\texttt{t} \in \texttt{lts}_i$, $\texttt{t} \notin \texttt{rts}_i$, and $\texttt{setMatch}(\texttt{t}, \texttt{rps}_i)$. Therefore, $\texttt{t} \in \texttt{lts}_i{+}{+}\,\texttt{ls}$. We have $\texttt{t} \notin \texttt{rs}$ since it matches some punctuation in $\texttt{rps}_i$. So, $\texttt{t} \notin \texttt{rts}_i{+}{+}\,\texttt{rs}$. Thus, t is in $(\texttt{lts}_i{+}{+}\,\texttt{ls}) - (\texttt{rts}_i{+}{+}\,\texttt{rs})$ and the containment is proven.

Faithfulness (completeness): Suppose $\texttt{t} \in (\texttt{lts}_i{+}{+}\,\texttt{ls}) - (\texttt{rts}_i{+}{+}\,\texttt{rs})$ for every ls and rs where

$$\texttt{setMatchTs}(\texttt{ls}, \texttt{lps}_i) = \texttt{setMatchTs}(\texttt{rs}, \texttt{rps}_i) = [\,].$$

Choosing ls to be [], we must have

$$\texttt{t} \in (\texttt{lts}_i{+}{+}\,[]) - (\texttt{rts}_i{+}{+}\,\texttt{rs}),$$

hence $\texttt{t} \in \texttt{lts}_i{+}{+}\,[\,]$ and, thus, $\texttt{t} \in \texttt{lts}_i$. We know $\texttt{t} \notin \texttt{rts}_i$, otherwise, it would be in $\texttt{rts}_i{+}{+}\,\texttt{rs}$. Now, consider any tuple s where $\texttt{setMatch}(\texttt{s}, \texttt{rps}_i) = \texttt{False}$. We must have $\texttt{t} \in (\texttt{lts}_i{+}{+}\,[\,]) - (\texttt{rts}_i{+}{+}\,[\texttt{s}])$, hence $\texttt{t} \notin \texttt{rts}_i{+}{+}\,[\texttt{s}]$, so $\texttt{t} \neq \texttt{s}$. Thus, $\texttt{setMatch}(\texttt{t}, \texttt{rps}_i) = \texttt{True}$. Since $\texttt{t} \in \texttt{lts}_i$, $\texttt{t} \notin \texttt{rts}_i$, and $\texttt{setMatch}(\texttt{t}, \texttt{rps}_i)$, we have

$$\texttt{t} \in \texttt{cpass}(\texttt{lts}_i, \texttt{lps}_i, \texttt{rts}_i, \texttt{rps}_i),$$

as required for completeness.

The complete proof can be found elsewhere [25].

6 Concerns for Punctuating Data Streams

We have assumed in our examples that punctuations already exist in the data stream. We have not addressed how punctuations are embedded in the first place. We address two main issues: how punctuations get into streams and how frequently should punctuations be embedded. Before we consider these issues, we compare punctuated streams to other techniques proposed to query over data streams.

6.1 Punctuated Streams versus Other Stream Techniques

Many current stream processing systems [9], [16], [24] use *fixed windows* and *sliding windows* for the same reasons we propose punctuated streams: They regulate the size of state and they allow blocking operators to emit results before the end of the stream. With fixed windows, stream inputs are divided into consecutive subsets, which can be marked explicitly with *landmarks* or implicitly with specific data values. In the warehouse example, we could implicitly define the end of a window at the end of each hour. Sliding windows allow query operators to process a bounded interval of tuples. When new tuples arrive, old tuples are discarded and the operator computes over the current window of tuples.

When a complete window (sliding or fixed) of tuples is available, a query operator computes its result for that window and emits the result. The operator then clears out

any state it might be maintaining for that window and starts over with the next window. In starting over, the semantics of some query operators change. For simple operators, such as select and project, the semantics remain the same. Tuples are processed and output immediately. However, more complex operators only process the current window, and not the entire data set. The group-by operator only groups within the current window rather than the entire input, and dupelim only removes duplicates within the window. Operators that exploit punctuations in a data stream may be able to process the entire stream. Group-by groups the entire input, and dupelim removes duplicates from the entire input.

Windows also add complexity to many operators, especially operators that accept more than one input. N-ary operators such as merge and join must maintain the semantics of the windows over all inputs. Suppose in the warehouse example we had only one sensor reporting temperatures. It would be straightforward to use the value of the hour attribute to denote a fixed window. Then, we group the input data on the hour attribute and output the maximum temperature when the hour attribute changes. However, if we have many sensors, we need the merge operator to combine all input streams. Merge must wait until it receives the end of a window from all inputs before outputting the end of the window. Data that pertains to the original window is output, but data that pertains to the new window must be held in state until all data for the current window has arrived from all inputs. This behavior introduces latency in the merge operator.

Sequence database systems [22] could also solve the warehouse query since it is time-based. Sequence database systems require the stream be globally sorted on sensor time. To preserve order throughout the query, each operator's output must be sorted on the sequence attribute. Therefore, n-ary operators must do extra work to maintain the sort order.

The auction example cannot be solved with either technique. In that example, we join bids and items on `itemid`. A join operator in a sequence database joins on the sequence value and cannot be used. Further, the items for sale in an online auction do not align neatly into a window. Some items may sell sooner than others. Punctuated streams is the only technique we are aware of that can give completely accurate results to this query.

6.2 Embedding Punctuations in Data Streams

We have discussed how punctuations embedded into a data stream can assist stream iterators. So far, we have ignored how punctuations get into a data stream. Let us suppose a logical operator existed that embedded punctuations, and could be placed in a variety of places: at the stream source, at the edge of the query processor, or after query operators within the query itself. We will call this operator the *punctuate* operator.

There are many different schemes we could use to implement the punctuate operator. Which scheme to choose depends on where the information resides for generating the punctuation. We list several possibilities below:

Source or sensor intelligence. As data items in a stream are often generated, the source may know enough to emit punctuation. Sources in the warehouse example (Example 1) were emitting data based on time. When an hour ended, the sensor emitted punctuation that all reports for that time period have been output.

Knowledge of access order. Scan or fetch operations may know something about the source and generate punctuations based on that knowledge. For example, if scan is able to use an index to read a source, it may use information from that index to tell when all values for an attribute have been read.

Knowledge of stream or application semantics. A punctuate operator may know something about the semantics of its source. In the warehouse example, the temperature sensors likely have temperature limits. Suppose the limits are 20F and 95F. A punctuate operator can output punctuations that say there will not be any temperature reports above 95F and there will not be any reports below 20F.

Auxiliary information. Punctuation may be genereated from sources other than the input stream, such as relational tables or other files. In the warehouse example, we might have a list of all the sensor units. A punctuate operator could use that list to determine when all sensor reports for a particular hour have arrived and embed the punctuation. This approach can remove punctuation logic from the sensors.

Stream iterator semantics. Some stream iterators impose semantics on output tuples. The select iterator, for example, can embed a punctuation that no tuples will appear in the stream that would fail its selection predicate. Additionally, the sort iterator can embed punctuations based on its sort order. When it emits a tuple, it can follow it with a punctuation stating that no more tuples will appear that precede that tuple according to the sort order.

6.3 Frequency of Punctuations

There is an obvious trade off in how often to embed punctuations in a data stream. If punctuations arrive frequently, then blocking operators can output data more often, and unbounded stateful operators are able to keep less state. However, punctuations also take up bandwidth in the data stream and, so, slow the arrival rate of tuples. This flexibility is an advantage of punctuations over other more static approaches. If the system executing a query over the data stream has a large amount of resident memory, then punctuations can be emitted less frequently. However, if the system executing a query over the data stream has a small amount of memory, such as a wireless telephone, then the number of punctuations will have to increase to minimize the amount of state required by the query. We are working on formal descriptions of the effects of different punctuation schemes over entire queries.

7 Prototype Implementation

To understand the complexity of implementing punctuations and their effect on performance, we used the Niagara Query Engine [18], which queries XML data. We defined a punctuation format for XML and enhanced some of Niagara's operators to exploit punctuation. We then simulated the warehouse example.

7.1 XML Punctuation Format

We have three goals in designing our XML punctuation format: First, punctuation size should be similar to the size of the tuples it describes. Second, punctuations should not affect the results of iterators that do not process punctuations. Third, iterators that do understand punctuations should be able to easily determine which tuples match a punctuation.

We achieve these goals using the *Namespaces for XML* recommendation [5]. We define a namespace called `punct` and define an element in it that mirrors a tuple structure. Iterators will not confuse punctuations with tuples since they are in a different namespace. For example, given elements of `tempdata`, we define punctuation elements `punct:tempdata`. Samples of an XML tuple and punctuation follow:

tuple	**punctuation**
<tempdata>	<punct:tempdata>
<sid>S01</sid>	<sid>*</sid>
<hour>17</hour>	<hour>[17,20)</hour>
<min>30</min>	<min>*</min>
<currtmp>77</currtmp>	<currtmp>*</currtmp>
</tempdata>	</punct:tempdata>

7.2 Enhancements to Niagara Operators

The query (in SQL) for this experiment is as follows:

```
SELECT MAX(currtmp)AS maxtemp, hour FROM
  SELECT currtmp, hour FROM sensor1 UNION
  SELECT currtmp, hour FROM sensor2
GROUP BY hour;
```

7.2.1 Union

The union operator in Niagara performs duplicate elimination, so it is an unbounded stateful operator. Tuples that have arrived are stored in a hash table. If a new tuple arrives that is already in the hash table, it is not output. If it is not in the hash table, it is output and added to the hash table. When union receives punctuations from all inputs that match the same set of tuples, it removes tuples in the hash table that match the punctuations. Then, union passes the punctuation to the next stream iterator. As we have discussed, punctuations output from union must agree with punctuations received from all inputs.

7.2.2 Group-By

Group-by is both a blocking and an unbounded stateful operator. We use punctuations to output results early and to reduce state size. Group-by must wait until it has read the entire input to ensure that no more values appear for a group. However, if a punctuation arrives that guarantees that all tuples for a given group have been seen, we can output the result for that group early. In the warehouse simulation, we are grouping on the `hour` attribute. Therefore, when a punctuation arrives guaranteeing that no more tuples will arrive for a particular hour we can output the result for that hour along with corresponding punctuation. Further, we can use that same punctuation to reduce the state for that group.

7.2.3 Join

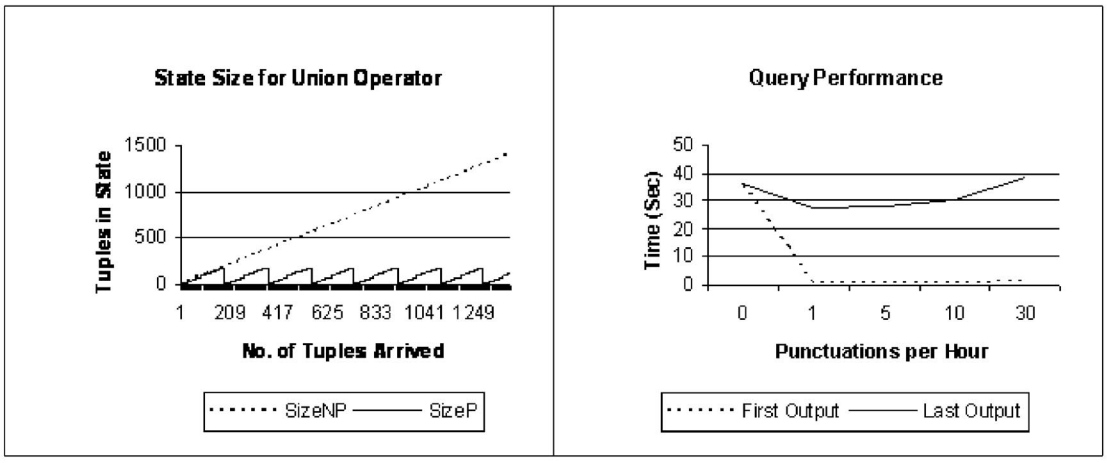
The Niagara Query Engine uses two implementations for the join operator: nested loops and symmetric hash join [27]. The symmetric hash join implementation is more appropriate for queries over data streams since it does not block on either input. We implement symmetric hash join using one hash table for each input. Tuples are stored in the appropriate hash table as they arrive, then used to probe the other hash table to determine if there are new tuples to output. The hash key is based on the value of the join attributes for each tuple. Thus, the hash tables can grow without bound. We use punctuations to decide when tuples can be removed from hash tables, according to the keep invariant for join shown earlier in Table 4. If a punctuation arrives from one input specifying that all tuples have arrived for a particular set of join attribute values, then we remove tuples from the hash table for the other input using the join values described by the punctuation as the hash key.

7.3 Results

We simulated sensors with Java applications that stream data into the Niagara Query Engine. These sensors output tuples as described earlier, reporting the current temperature at that sensor every minute. For these tests, we varied the frequency of embedded punctuations from 0 to 30 per hour. We wanted to evaluate the performance cost of embedding punctuations into data streams. Thus, our sensor applications output data immediately, rather than outputting a single data item every minute. The data was over 60 hours.

We see from Fig. 2a that the amount of memory used by the hash table in union is greatly reduced. If the stream is not punctuated, the size of the hash table grows until the stream ends. If the stream is punctuated, the size of the hash table is much smaller since tuples are removed from state that match punctuations.

We see from Fig. 2b that embedding punctuations in the stream unblocks group-by, allowing it to output results early. Without punctuations, the operator must wait until the input has completed before outputting results. We also see that embedding punctuations does not affect the overall performance of the query. In fact, when using a minimal number of punctuations, the performance improves. This improvement is because the hash table used by union is smaller when punctuations exist in the data stream. Without punctuations, the hash table grows, forcing more frequent memory allocations.



(a) (b)

Fig. 2. (a) The state held by union as tuples arrive. The dotted line shows the size of state when punctuations are present and the solid line shows the size of state without punctuations. (b) Query Performance. The dashed line indicates when the first results are emitted and the solid line is the total time to execute the query.

8 Related Work

Data stream processing systems have been researched for a number of years, though interest has increased recently. The Tangram stream query processing system by Parker et al. [20] is an early investigation attempting to use database technology on a data stream. They discuss a number of stream transducer implementations. They include blocking relational operators such as sort and difference, but do not discuss how these behave in the presence of continuous streams. Parker also discusses in more detail a model for stream transducers [19] that is very similar to the one we present, though he limits the discussion to unary stream transducers and does not address how they interact with continuous data streams.

In the architecture describe by Babu et al. [4], a query over a data stream has: one or more input streams, an output stream, a *store* for holding tuples that might be part of the result, and a *scratch* for holding state. Punctuations could enhance this architecture in two ways: First, implementing pass invariants could reduce the size of store by outputting tuples sooner. Second, implementing keep invariants could reduce the size of scratch.

Like the Tangram system, most stream database systems use a window or summarization scheme to handle blocking and unbounded stateful operators. The Tribeca system [24] is another early example of such a system. The authors introduce a high-level query language for data streams, but add operators specifically for defining stream windows. The system discussed by Madden and Franklin [16] implements operators that efficiently combine data from static (file-based) and streaming sources. Additionally, they have enhanced their adaptive query processing system [3] to support queries over data streams [17]. Finally, work on the Aurora system [6] investigates optimization tactics for queries over data streams and ways to intelligently shed load when the system is about to run out of memory.

Work on partial results by Shanmugasundaram et al. [23] specifically addresses blocking operators. In their system, query operators waiting for output from other operators can request a partial result. Online computation work [13] also addresses blocking operators by reporting partial results at regular intervals. The user interface reports progress with a confidence interval and allows users to cancel queries when the result is "good enough." A priori knowledge of input data is required to determine the confidence level and interval. Punctuations in a data stream could be used to present that knowledge to the query.

Arasu et al. [2] specifically address the problem of unbounded stateful operators. They propose an algorithm to determine if a given select-project-join query can be processed in bounded memory for all possible inputs. In cases where the query can be processed in bounded memory, their algorithm also produces an evaluation plan for the query, characterizing the memory requirements. Their discussion is limited to the select, project, and join operators and does not address other operators such as union, set difference, and grouping.

9 Conclusions and Future Work

We have presented a novel way to execute queries on continuous data streams. Our method improves many traditional relational operators, specifically targeting blocking and unbounded stateful operators. We define stream iterators as a foundation for our work and describe a specific framework for modeling stream iterators on continuous data streams. We define desirable properties for stream iterators and give proofs that our invariants meet those properties for two operators and that the corresponding iterators satisfy the invariants.

There is more work to be done in this area. We need to expand our formal model and prove invariants for more operators, as well as prove the iterators defined in our framework satisfy the invariants. We also need to make more Niagara operators aware of punctuations, to further show the practicality of our approach.

The punctuations we present here are strict—they say that no more tuples will arrive matching a punctuation from the stream. A more relaxed punctuation might tell an operator that there will not be any matching tuples arriving after it for an extended time. Operators could use that information to enhance buffer eviction policies. Tuples that

are not needed for a while are good candidates for swapping to disk. For example, the XJoin implementation [26] might be able to use this kind of punctuation to determine which partition can be written to disk. Another kind of punctuation might convey a constraint about the data that is about to arrive, such as a sort order or a bound on the range of a particular attribute.

Another use for punctuations is to declare information about the arriving data, rather than the data that has been seen. For example, Fegaras et al. [8] use annotations in the data stream to declare the incoming data structure, and whether the data fragment following the annotation is a repeat or an update. One could pursue this direction further to specify other constraints, such as a sort order over particular attributes or even attribute domains.

One issue we ran into during the implementation phase was in evaluating our work. We presented some results that show output arriving early and decreased state throughout the execution of the query, but how do we compare our work to other stream processing systems? How do we know what "good" performance is? We are designing a benchmark for stream processing systems based on the scenario in XMark [21].

Another interesting direction is to be able to determine, given a query over a data stream, whether a given punctuation scheme will be helpful. Stream generators may be able to output different kinds of punctuation to a query. We would like to be able to determine which would be most beneficial.

There is common work with partial evaluation of queries [13], [23]. Both address applying blocking operators to long or continuous inputs. Punctuation might be used to help partial operators communicate output that is "partial" versus output that is "correct," or assist in computing how close we are to the "correct" output.

ACKNOWLEDGMENTS

The authors would like to thank Kristin Tufte and Vassilis Papadimos for the discussions they had, Levent Erkok for his help in developing the Haskell framework, Jennifer Widom and Samuel Madden for their comments, Raghu Ramakrishnan for his pointers to related work, David DeWitt for his advice on the direction of this work, and the anonymous reviewers for their comments on this paper. Funding for this work was provided by DARPA through NAVY/SPAWAR Contract No. N66001-99-1-8908 and by the US National Science Foundation ITR award IIS0086002. Leonidas Fegaras is supported by the US National Science Foundation grant IIS-9811525.

REFERENCES

[1] J. Albert, "Algebraic Properties of Bag Data Types," *Proc. 17th Int'l Conf. Very Large Data Bases,* pp. 211-219, Sept. 1991.

[2] A. Arasu, B. Babcock, S. Babu, J. McAlister, and J. Widom, "Characterizing Memory Requirements for Queries over Continuous Data Streams," *Proc. 21st ACM SIGACT-SIGMOD-SIGART Symp. Principles of Database Systems,* pp. 221-232, June 2002.

[3] R. Avnur and J.M. Hellerstein, "Eddies: Continuously Adaptive Query Processing," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* pp. 261-272, May 2000.

[4] S. Babu and J. Widom, "Continuous Queries over Data Streams," *SIGMOD Record,* vol. 30, no. 3, Sept. 2001.

[5] *Namespaces in XML.* T. Bray, D. Hollander, and A. Layman, eds., World Wide Web Consortium, http://www.w3.org/TR/REC-xml-names/, Jan. 1999.

[6] D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik, "Monitoring Streams—A New Class of Data Management Applications," *Proc. 28th Int'l Conf. Very Large Data Bases,* Aug. 2002.

[7] C. Cortes, K. Fisher, D. Pregibon, A. Rogers, and F. Smith, "Hancock: A Language for Extracting Signatures from Data Streams," *Proc. Sixth Int'l Conf. Knowledge Discovery and Data Mining,* pp. 9-17, Aug. 2000.

[8] L. Fegaras, D. Levine, S. Bose, and V. Chaluvadi, "Query Processing of Streamed XML Data," *Proc. 11th Int'l Conf. Information and Knowledge Management,* Nov. 2002.

[9] J. Gehrke, F. Korn, and D. Srivastava, "On Computing Correlated Aggregates over Continuous Data Streams," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* pp. 13-24, May 2001.

[10] A.C. Gilbert, Y. Kotidis, S. Muthukrishnan, and M.J. Strauss, "Surfing Wavelets on Streams: One-Pass Summaries for Approximate Aggregate Queries," *Proc. 27th Int'l Conf. Very Large Data Bases,* pp. 79-88, Sept. 2001.

[11] G. Graefe, "Query Evaluation Techniques for Large Databases," *ACM Computing Surveys,* vol. 25, no. 2, 1993.

[12] T. Hallgren and M. Carlsson, "Fudgets," PhD thesis, Chalmers Univ. of Technology, Mar. 1998.

[13] J.M. Hellerstein, P.J. Haas, and H.J. Wang, "Online Aggregation," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* pp. 171-182, June 1997.

[14] P. Hudak, *The Haskell School of Expression: Learning Functional Programming through Multimedia.* Cambridge Univ. Press, 2000.

[15] B.K. Livezey and R.R. Muntz, "ASPEN: A Stream Processing Environment," *Proc. Conf. Parallel Architectures and Languages Europe,* pp. 374-388, 1989.

[16] S. Madden and M.J. Franklin, "Fjording the Stream: An Architecture for Queries over Streaming Sensor Data," *Proc. 18th IEEE Int'l Conf. Data Eng.,* pp. 555-566, Feb. 2002.

[17] S. Madden, M. Shah, J.M. Hellerstein, and V. Raman, "Continuously Adaptive Continuous Queries over Streams," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* pp. 49-60, June 2002.

[18] J. Naughton, D. DeWitt, D. Maier, J. Chen, L. Galanis, K. Tufte, J. Kang, Q. Luo, N. Prakash, and F. Tian, "The Niagara Query System," *The IEEE Data Eng. Bull.,* vol. 24, no. 2, pp. 27-33, June 2000.

[19] D.S. Parker, "Stream Data Analysis in Prolog," *The Practice of Prolog,* L. Sterling, ed., chapter 8, MIT Press, 1990.

[20] D.S. Parker, R.R. Muntz, and L. Chau, "The Tangram Stream Query Processing System," *Proc. Fifth IEEE Int'l Conf. Data Eng.,* Feb. 1989.

[21] A. Schmidt, F. Waas, M. Kersten, D. Florescu, I. Manolescu, M.J. Carey, and R. Busse, "The XML Benchmark Project," Technical Report INS-R0103, Centrum voor Wiskunde en Informatica, Apr. 2001.

[22] P. Seshadri, M. Livny, and R. Ramakrishnan, "Sequence Query Processing," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* pp. 430-441, May 1994.

[23] J. Shanmugasundaram, K. Tufte, D. J. DeWitt, J. Naughton, and D. Maier, "Architecting a Network Query Engine for Producing Partial Results," *WebDB (Informal Proceedings),* pp. 17-22, May 2000.

[24] M. Sullivan and A. Heybey, "Tribeca: A System for Managing Large Databases of Network Traffic," *Proc. 1998 USENIX Ann. Techincal Conf.,* June 1998.

[25] P. Tucker, D. Maier, T. Sheard, and L. Fegaras, "Enhancing Relational Operators for Querying over Punctuated Data Streams," http://www.cse.ogi.edu/dot/niagara/pstream/punctuating.pdf., 2002.

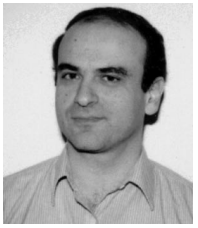
[26] T. Urhan and M.J. Franklin, "Xjoin: A Reactively-Scheduled Pipelined Join Operator," *The IEEE Data Eng. Bull.,* vol. 23, no. 2, pp. 27-33, June 2000.

[27] A.N. Wilschut and P.M.G. Apers, "Dataflow Query Execution in a Parallel Main-Memory Environment," *Proc. First Int'l Conf. of Parallel and Distributed Information Systems,* pp. 68-77, Dec. 1991.

Peter A. Tucker received the BS degree in mathematics and computer science in 1991 from Whitworth College (Spokane, WA). He then worked for eight years at Microsoft Corporation in software development. He is currently in his fourth year of study at the OGI School of Science and Engineering at Oregon Health and Science University, pursuing a PhD degree in database systems. His technical interests include data stream processing, formal semantics, and functional programming.

David Maier received the BA (Honors College) degree from the University of Oregon, with a double major in mathematics and computer science, and the PhD degree from Princeton University in electrical engineering and computer science. He is a professor in the Department of Computer Science and Engineering in the OGI School of Science and Engineering (formerly the Oregon Graduate Institute) at the Oregon Health and Science University, and previously was on the faculty of SUNY Stony Brook. He has held visiting positions at INRIA (Rocquencourt) and University of Wisconsin-Madison. His research interests include query processing, object-oriented systems, databases and data-product management for scientific computing, superimposed information systems, and net data management. He is a fellow of ACM and holds the ACM SIGMOD Innovations Award. He is a member of the IEEE and the IEEE Computer Society.

Tim Sheard graduated with a degree in chemistry from Harvard in 1979. He received the MS degree in 1979 from the University of Vermont, and the PhD degree in 1985 from the University of Massachusetts at Amherst. His research at first investigated database integrity and database query languages. In the early 1990's, he became an assistant professor of computer science at the Oregon Graduate Institute (now the Oregon Health and Science University), where he began investigating approaches to generic programming. Today he is a strong proponent of the use of functional languages and metaprogramming systems. He is the principle designer and implementer of the MetaML system.



Leonidas Fegaras received the PhD degree in computer science from the University of Massachusetts at Amherst in 1993. He is currently an associate professor of computer science and engineering at the University of Texas at Arlington. He was previously a senior research scientist at the OGI School of Science and Engineering at the Oregon Health and Science University. He has contacted research in the areas of database query processing and optimization, object-oriented databases, Web databases, database programming languages, functional programming, and formal semantics. He is a member of the IEEE Computer Society.