Punctuating Continuous Data Streams

Pete Tucker[1*] David Maier[1] Tim Sheard[1] Leonidas Fegaras[2]
[1]OGI School of Science & Engineering at OHSU [2]University of Texas at Arlington
{ptucker,maier,sheard}@cse.ogi.edu fegaras@cse.uta.edu

Abstract

As most query processing architectures are pipelined, it seems logical to extend them to process data streams. However, operators that maintain state with no upper bound, and operators that read their entire input before emitting an output, are problematic. A priori knowledge of the data stream can permit the use of such operators. Punctuations in a stream mark the end of substreams, logically breaking a stream into a combination of finite streams. We present a framework for stream operators in the presence of punctuation, and provide definitions for when a stream operator should be judged "faithful" to its finite counterpart.

1 Introduction

There are numerous examples of applications that process data streams: Financial applications process streams of stock ticker data; telephone network monitoring applications process streams of call data [GKMS01]. As a running example, consider an outage tracking system for an electrical power company. Devices installed at substations and on major distribution lines stream problem reports to the main system. Additionally, customers can report problems to an automated phone bank. These reports are streamed from the phone bank to the main system, and compared against the monitoring information to filter out known outages.

These applications often are simply queries over data streams. We would like to implement them with generic DBMS-like functionality, rather than custom code. However, two properties of data streams diverge from what standard DBMS technology handles: Data is often generated by a device that can produce perpetual input, and a stream's contents can only be accessed sequentially.

Traditional queries are comprised of relational operators. To execute queries on data streams, we need to define operators that are functions from input streams to output streams. Parker calls such operators *stream transducers* [Ste90]. We can implement the outage tracking system in a DBMS with a query that finds all problems in the customer report stream that were not reported in the equipment monitoring streams, as shown in Figure 1. Each report from an equipment monitor is streamed into the union operator. The output from the union operator is sent to the difference operator as the negative input, with the customer reports stream being the positive input.

Unfortunately, this system will fail for two reasons: First, the difference operator must wait until all of its data from the negative input has been read before it can output results. Since the data is coming from a continuous data stream, it has no determinate end. Second, the union operator removes duplicates in its output, requiring an unbounded amount of state over time. Thus, the query's storage requirements will grow without limit.

Our example illustrates two kinds of operators that are impractical for processing continuous data streams. A *blocking operator* must wait until all data has been read from at least one of its inputs before producing a result. An *unbounded stateful operator* maintains state that grows with no upper bound. Our goal is to find stream-based analogues for table-based (finite) operators,

[*]Paper 122. Contact information: Pete Tucker, Oregon Graduate Institute at OHSU, 20000 NW Walker Road, Beaverton, OR 97006 USA. Ph: (503)748-1371. Fax: (503)748-1285. E-mail: ptucker@cse.ogi.edu.

exploiting stream semantics to overcome these problems.

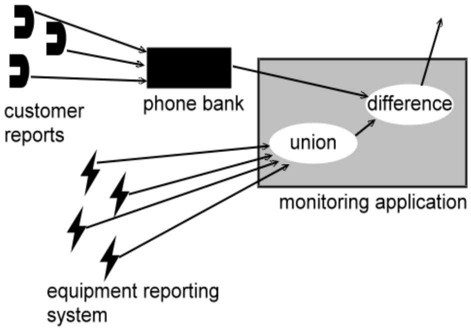

Figure 1: System to track electrical outages. A lightning bolt represents an equipment monitoring device, and the telephones represent customer reports.

Not all stream-to-stream functions are reasonable candidates for table analogues. Consider sorting on positive integers. For an infinite stream S of positive integers, we can define a function sort(S) that returns the integers of S in increasing order. However, we can't implement it, as it requires seeing the entire (infinite) contents of the stream before emitting any output. We want to say that sort has no stream analogue, even though a stream-to-stream function exists for it. A practical stream function should compute its output "as you go". We capture this idea more precisely with the notion of a *stream iterator*. A stream iterator is a stream-to-stream function whose output is a sequence of partial computations over finite prefixes of its input. Thus, a stream iterator never sees the entire contents of an infinite stream at one time, but increasing (finite) portions of it.

2 Stream Iterators

To define this notion more precisely, we need a data type for streams. For the moment, we represent them as infinite sequences; a cons-based formulation of lists but with no nil list. A stream over type T elements is defined as Stream(T) = T $\oplus$ Stream(T), where $\oplus$ is an infix constructor.

We will use {|...|} to denote a stream value, to distinguish them from finite lists. Thus, considering Stream(Int), we can write 1 $\oplus$ 3 $\oplus$ 5 $\oplus$ 7 $\oplus$ ... as {|1,3,5,7,...|}. If S is a stream, we write S[i] for the list of the first i elements. Thus, if S is the stream above, S[3] is [1,3,5]. We use S@i for the ith 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 $\oplus$ S.

We can now define a stream iterator. Function f: Stream(T) $\rightarrow$ Stream(U) is a stream iterator if there exists a function q: List(T) $\rightarrow$ List(U) such that for any S in Stream(T), f(S) = (q(S[1]) $\otimes$ q(S[2]) $\otimes$ q(S[3]) $\otimes$ ... $\otimes$ q(S[i]) $\otimes$ ...).

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. If the selection predicate is p, then we define q by

```
q(L ++ [a]) = [a] if p(a), q(L ++ [a]) = [ ] otherwise.
```

That is, q just looks at the last element of each prefix, and emits it if it satisfies the predicate. (Here '++' is list concatenation.) There is also a stream iterator for dup_elim. It uses q such that:

```
q(L ++ [a]) = [a] if ~elem(a, L), q(L ++ [a]) = [ ] otherwise.
```

That is, q checks that the final element in the prefix does not appear earlier in the prefix. However, sort cannot be expressed as a stream iterator, nor can group-by. We can never emit the aggregate for a group, because members of the group may still arrive. (We are skirting the important issue of what is an appropriate extension of a table operator to infinite streams. In Section 3.3 we formally define what it means for a stream iterator to be *faithful* to a table operator.)

Given stream iterators as a definition of what constitutes a "practical" stream-to-stream function, can we remedy the lack of stream iterators for operators such as sort and group-by? Obviously, if we had a way to represent finite as well as infinite streams, we could emit output when the end of the stream is seen. One way to represent a finite stream is with special value called EOS to indicate that no more regular elements will be seen. Thus a finite stream would look like {|8, 2, 6, 11, 3, EOS, EOS, EOS, ...|}. If we were defining sort, then the function q could emit [2,3,6,8,11] on the prefix [8, 2, 6, 11, 3, EOS], and emit [EOS] for all longer prefixes.

This addition represents no real progress; it shows we can provide a result in the finite case, which we already knew. However, suppose we knew at some point that we would see no more elements in the range 1 to 10. At that point (assuming a stream of positive integers), we could emit part of the output. Let EOS(1-10) to indicate this condition. For {|8, 2, 6, 11, 3, EOS(1-10), 12, 24, 15, EOS(11-20), 28, 21, ...|}, q can emit [2,3,6,8] on prefix [8, 2, 6, 11, 3, EOS(1-10)] and emit [11,12,15] on the prefix ending with EOS(11-20). We call an annotation such as EOS(1-10) a *punctuation*, as it signals the end of a certain part of the input. Punctuation does not require the stream be partitioned into ranges of values: The value 11 appears before EOS(1-10). Punctuation only speaks about what comes subsequently in a stream.

The sequel discusses our framework for defining stream operators that exploit punctuation. Punctuation in a stream will let us "pass along" outputs early that we wouldn't be able to emit otherwise. We can also "propagate" punctuation into the output of a stream iterator for use by other stream iterators.

2.1 Representation Issues

We have presented one formulation for data streams and "realizable" functions over those streams. In our investigations we adopt a slightly different representation of streams and iterators than the one used here. Rather than representing a stream by an infinite sequence of elements, we use an infinite sequence of lists of elements — a "sliced list". Thus {| 1, 2, 3, 4, 5, ... |} might appear in sliced form as {| [1, 2], [3], [ ], [4, 5], ... |}, or as {| [1, 2, 3], [4, 5], ... |}, or in myriad other forms. There are several reasons for using a sliced-list form:

- It allows direct modeling of finite streams, using a trailing sequence of empty slices: {| [1, 2], [3], [4], [ ], [ ], [ ], ... |}. However, it does not represent quite the same range of streams as the original representation. Any infinite sequence has an infinite number of equivalent sliced list representations, but some sliced lists have no equivalent infinite sequence.
- It intuitively captures that stream arrivals are not synchronized with operator iterations. With an infinite sequence, there is always a single element when reading from a stream. In practice, we might see multiple arrivals when we read an input, or no arrivals for an arbitrarily long time.
- It does not assume that elements of different input streams arrive "in step". We had considered presenting two input streams as a "merge" of their infinite sequences, tagging elements as left or right. However, there can be unfair merges, where all or part of one of the inputs is delayed indefinitely behind the other input. Slicing handles different arrival rates between the input streams, without having to define special merge conditions.

The second change is not to provide the entire stream prefix to the operator at each iteration. Some operations, such as select, need only the most recent input element to figure out their next output. Other operators only need a summary of the prefix. For example, dup_elim only needs the *distinct* values in the prefix. Thus, in our actual formulation, operators keep state from iteration to iteration. The explicit representation of state points out that there are stream analogues of table operators that must remember arbitrary amounts of the prefix, such as dup_elim. We will see that punctuation can also help in *purging* state.

2.2 Punctuation Representation and Manipulation

Punctuation can be considered a predicate on stream elements that will not be seen past the punctuation. Thus there are many choices for representing punctuation. We represent punctuation as data, allowing easy storage, searching and manipulation. Our scheme is fairly simple, but has the property that the "and" of any two punctuation is representable as another punctuation.

Elements in our streams are tuples of scalars, and punctuations for such tuples. A punctuation is an ordered set of patterns, each pattern corresponds to an attribute of a tuple. A tuple matches a punctuation if each attribute in the tuple matches each corresponding pattern in the punctuation. We define five kinds of patterns, and a match condition for each, as shown in Table 1.

name	**pattern**	**match condition**
wildcard	$*$	$true$
constant	c	$i == c$
range	$[c_1, c_2]$	$i \geq c_1 \wedge i \leq c_2$
list	$\{c_1, c_2, \ldots\}$	$i \in \{c_1, c_2, \ldots\}$
empty	$\emptyset$	$false$

Table 1: Match conditions for each pattern, given attribute value i

For example, if the tuple structure for phone reports is: $< zoneid, hour >$, a punctuation that describes reports for hour 3 is $< *, 3 >$. The punctuation that describes reports for zones 1 and 6 for hours between 5 and 8 is $< \{1, 6\}, [5, 8] >$.

Embedding punctuations into the input streams can fix the electric company example. Suppose the tuples for both inputs contain `zoneid` and `hour`, and the equipment monitors and the phone bank embed punctuations describing reports for a particular hour. When the union operator receives punctuations for a particular hour from its inputs, it can purge all tuples in its state for that hour. Likewise, when difference gets punctuation on its negative input for a particular hour, it can output tuples from the positive input that match that punctuation and have not appeared on the negative input. Further, output tuples can be purged.

We define two basic functions for manipulating punctuations. The `match` function takes a punctuation `p` and a tuple `t`, and returns `True` if `t` satisfies `p`. The `combine` function takes two punctuations, and returns a new punctuation that is the intersection of the two inputs (we discard punctuations that contain `empty`). Logically, `match (t, (combine (`p_1`, `p_2`)))` $\Rightarrow$ `match (t,`p_1`)` $\wedge$ `match (t, `p_2`)`. We use helper functions for lists of inputs. A subset is shown in Table 2.

Function	**Return Value**	**Function**	**Return Value**
`nomatch(t,p)`	¬ `match(t,p)`	`setMatch(t,ps)`	`True` if `t` matches any `ps`
`setNomatch(t,ps)`	`True` if `t` matches no `ps`	`setMatchTs(ts,ps)`	all t ∈ `ts` matching any `ps`
`setNomatchTs(ts,ps)`	all t ∈ `ts` matching no `ps`	`setNomatchPs(ts,ps)`	all p ∈ `ps` matching no `ts`
`tups(xs)`	tuples in `xs`	`puncts(xs)`	punctuations in `xs`
`setCombine(ps1,ps2)`	all combinations of punctuations in `ps1`, `ps2`		

Table 2: Helper functions based on `match` and `combine`, where `t` is a tuple, `ts` is a finite list of tuples, `p` is a punctuation, `ps` is a finite list of punctuations, and `xs` is a finite list of either.

3 Our Framework

Our framework for stream iterators is described in Haskell, a *lazy evaluation* functional language. Lazy evaluation languages do not evaluate expressions until required. Thus, they are suitable for modeling systems that process continuous data streams, as tuples in a stream are only produced when required. Other research projects, such as ASPEN and Tangram [LM89, PMC89], and Fudgets [HC98], have used lazy evaluation languages to model data stream processors as well. (We use Haskell syntax in this section. A function application such as `match (t,p)` looks like `match t p`).

3.1 Basic Stream Iterators

Without punctuations, stream iterators closely resemble Parker's stream transducers [Ste90]. A unary stream iterator is a triple, `(initial_state, step, final)`, where:

- `initial_state` is the state of the iterator before tuples arrive form the input stream.
- `step` is called when a new data slice arrives from the input stream. 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 output tuples and a modified state.

The behavior of unary stream iterators is modelled by the `unary` function:

```
unary state step final [] = fst(final state)
unary state step final (xs:rest) = let (out,new_state) = step xs state
                                   in out ++ (unary new_state step final rest)
```

For example, the definition of duplicate elimination is:

```
sdupelim xs = unary [] destep definal xs
  where destep ts tsSeen = ((nub ts \\ tsSeen), tsSeen ++ ts)
        definal tsSeen   = ([], [])
```

State is all unique values from the input. The `destep` function returns new tuples not in state and a modified state with new tuples appended. Two functions are from the Haskell `prelude` library: `nub` removes duplicates from a list, and `\\` returns values in a first list not in a second.

A binary stream iterator is 5-tuple: `(initial_state, stepL, stepR, finalL, finalR)`. It has step and final functions for both left and right inputs. In the general operation of a binary stream iterator, when new data arrives from an input, the appropriate `step` function is called. When a stream ends, the appropriate `final` is called, and then the other stream is read to completion, and then its `final` is called. The formal definition of `binary` is similar to `unary`, and is omitted.

For example, consider the definition for the difference stream iterator:

```
sdiff xs = dupelim (binary (False,[],[]) dstepL dstepR dfinalL dfinalR xs)
  where dstepL ts (False, lts, rts) = ([], (False, ts ++ lts, rts))
        dstepL ts (True, lts, rts) = (ts \\ rts, (True, [], rts))
        dstepR ts (f, lts, rts) = ([], (f, lts, ts ++ rts))
        dfinalL st = ([], st)
        dfinalR (f, lts, rts) = (lts \\ rts, (True, [], rts))
```

State contains a Boolean indicating if the negative input has ended, and one list of tuples for each input. The step functions add the new data to the appropriate list. Additionally, if the negative input has ended, `dstepL` outputs tuples not in the negative input. The `dfinalR` function outputs any tuples from the positive input that are not in the negative input, and modifies state by setting the flag to `True` and empties the list for the positive input (since those tuples were output).

This example illustrates two points: First, it is clear that each input must have its own implementation of `step` and `final`, as the tasks for these functions can be different for each input. Second, `final` must return a state value, since it may be needed for processing the other input.

3.2 Extension to Punctuated Streams

To support punctuated streams we define three new functions: `pass`, `prop`, and `purge`, and we redefine a unary stream iterator to be `(initial_state, step, pass, prop, purge)`, where:

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

As before, binary iterators implement each function for each of their inputs. Note that `final` was removed. The `final` function was called if the input stream ended. The task of `final` will be performed by `pass`, `prop`, and `purge` on an end-of-stream punctuation.

The changes to `unary` and `binary` are similar, so we focus on `unary`. New input may contain both tuples and punctuation, so they are separated. Then `step` is called, then calls to `pass`, `prop`, and purge with new punctuations and state. The execution order of punctuation functions is important: The prop function could output punctuation matching tuples from pass, so it is called first. The purge function may purge state related to the outputs of pass or prop, so it comes last.

There are cases where operators do not need to implement particular punctuation functions. For example, the duplicate elimination operator is not a blocking operator. It does not need to implement the pass function. We define *trivial punctuation functions* for this case:

```
passT  ps st = []; propT  ps st = []; purgeT ps st = st
```

The difference iterator is redefined as follows:

```
sdiff lxs rxs = dupelim (binary ([],[],[],[]) lstep passT lprop lpurge
                         rstep rpass propT rpurge lxs rxs)
  where lstep ts (lts,rts,lps,rps) = ([],(ts ++ lts,rts,lps,rps))
        rstep ts (lts,rts,lps,rps) = ([], (lts,ts ++ rts,lps,rps))
        rpass ps (lts,rts,lps,rps) = (setMatchTs (lts \\ rts) (ps ++ rps))
        lprop ps (lts,rts,lps,rps) = setNomatchPs lts (ps ++ lps)
        lpurge ps (lts,rts,lps,rps) = let ps0 = lprop ps (lts,rts,lps,rps)
                                      in (lts,rts,(ps ++ lps) \\ ps0,rps)
        rpurge ps (lts,rts,lps,rps) = let rpsNew = ps ++ rps
                                          ltsNew = setNomatchTs lts rpsNew
                                      in (ltsNew,rts,lps,rpsNew)
```

State has lists for tuples and punctuations from the left and right inputs. The step functions add new tuples to the appropriate list. The rpass function returns tuples from the left input not in the right input that match right-input punctuation. The lprop function returns punctuations that do not match any tuples in state for the left input. The lpurge function discards punctuations that were output. and rpurge discards tuples from the left input returned from rpass.

3.3 Faithfulness and Propriety

We have presented our framework for stream iterators, and the use of punctuation to pass output early and purge state. However, we have not said when a stream iterator is a "reasonable" counterpart of the corresponding table operator. Clearly one can define stream versions of operators with inappropriate behavior. For example, a "stream sort" operator that appends the sorted elements of each stream slice as it arrives does not seem an appropriate analog of finite sort. We base our notion of correctness of a stream iterator upon its series of partial outputs after processing each slice of the input(s). We want the iterator's output at any point input to be consonant with any possible further input it might see. We adapt the S[i] notation to the sliced representation of streams by letting S[i] be the concatenation of the first i lists in S. So if S = {| [1, 2], [3], [ ], [4, 5], [6], ... |}, then S[4] = [1, 2, 3, 4, 5]. For a stream iterator f, we define the i^{th} *partial* of f as

$\hat{\texttt{f}}$(S, i) = f(S)[i], if f is unary

$\hat{\texttt{f}}$(R, S, i) = f(R, S)[2i] if f is binary.

That is, the $\texttt{i}^{th}$ partial is the list of elements emitted after processing the first i slices of (each) input. (Note the 2i index is because f(R,S) emits slices of output for the $\texttt{i}^{th}$ slices of R and S.)

We first define faithfulness for streams without punctuation. Let f be a unary stream iterator Stream(T) $\rightarrow$ Stream(U) and g a table operator List(T) $\rightarrow$ List(U). f is *faithful* to g if:

1. (Safety) For all S in Stream(T), for all i, and for every L in List(T), $\hat{\texttt{f}}$(S, i) $\subseteq$ g(S[i]:L). (We interpret $\subseteq$ as prefix or subset depending on whether order is significant for g.)
2. (Completeness) For all S in Stream(T), for all i, if t $\in$ g(S[i]:L) for every L in List(T), then t $\in$ $\hat{\texttt{f}}$(S, i).

The safety condition says we never emit output unless we can be sure that it won't conflict with any later input we see. The completeness condition says we always emit an output if it will necessarily be generated by the table operator under any additional input.

The corresponding conditions for binary operators are similar: $\hat{\texttt{f}}$`(R, S, i)` $\subseteq$ `g(R[i]:L1, S[i]:L2)` for all `L1` and `L2`, and `t` $\in$ $\hat{\texttt{f}}$`(R, S, i)` if `t` is always in `g(R[i]:L1, S[i]:L2)`.

Note that every monotone table operator `g` has a faithful stream counterpart. In the case of unary operators, it is the iterator `f` where `f(S)@i = g(S[i]) - g(S[i-1])`. (We are using `S@i` now to mean the $\texttt{i}^{th}$ slice of `S`.) Thus `f(S)[i] = g(S[i])`, and the two conditions are met.

With punctuation, we modify the conditions such that any possible additional input (represented by `L`) is constrained to obey any punctuation already seen.

1. (Safety) For all `S` in `Stream(T)`, for all `i`, and for every `L` in `List(T)` such that `setMatch(L, puncts(S[i]))` is empty, `tups(`$\hat{\texttt{f}}$`(S, i))` $\subseteq$ `g(tups(S[i]):L)`.
2. (Completeness) For all `S` in `Stream(T)`, for all `i`, if `t` $\in$ `g(tups(S[i]):L)` for every `L` in `List(T)` such that `setMatch(L, puncts(S[i]))` is empty, then `t` $\in$ `tups(`$\hat{\texttt{f}}$`(S, i))`.

The definition for binary operators is extended in a similar way. We will give an example proof shortly of the faithfulness of one of our stream operators.

The other condition to enforce on stream operators is that they respect punctuation. 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 investigating stronger notions, such as requiring an iterator to emit the *maximal* amount of punctuation that can be inferred from the input punctuation. A stronger condition still is that an iterator be *deterministic*: For any possible element `t` in the output, we are guaranteed of either eventually seeing `t` or seeing a punctuation that covers `t`.

We have been developing stream analogs of a range of table operators, and have been proving properties about them. We include here a proof that our stream version of difference is faithful and proper. Our general strategy for such proofs is to define global rules (that is, rules defined in terms of all input seen so far) for passing output, propagating punctuation and purging state. We show that a stream operator that obeys these global rules is faithful and proper, then show that the `pass`, `prop` and `purge` functions used in our incremental framework conform to the global rules.

Theorem 1 *The stream iterator `sdiff` is proper and faithful to the table operator difference.*

Proof: We give the following rules for the behavior of a stream iterator for difference. Let `L` and `R` be any two input streams, and assume we are at slice `i` in both streams. Let $\texttt{lts}_i$ `= tups(L[i])`, $\texttt{lps}_i$ `= puncts(L[i])`, $\texttt{rts}_i$ `= tups(R[i])` and $\texttt{rps}_i$ `= puncts(R[i])`. The rules are

$\texttt{cpass lts}_i\ \texttt{lps}_i\ \texttt{rts}_i\ \texttt{rps}_i$ = $\{\texttt{t} \mid \texttt{t} \in \texttt{lts}_i,\ \texttt{t} \notin \texttt{rts}_i,\ \texttt{setMatch t rps}_i\}$

$\texttt{cprop lps}_i\ \texttt{rps}_i$ = $\texttt{setCombine lps}_i\ \texttt{rps}_i$

$\texttt{cpurgeL lts}_i\ \texttt{lps}_i\ \texttt{rts}_i\ \texttt{rps}_i$ = $\{\texttt{t} \mid \texttt{t} \in \texttt{lts}_i \texttt{ and } (\texttt{t} \in \texttt{rts}_i \texttt{ or setMatch t rps}_i)\}$

$\texttt{cpurgeR lts}_i\ \texttt{lps}_i\ \texttt{rts}_i\ \texttt{rps}_i$ = $\{\texttt{t} \mid \texttt{t} \in \texttt{rts}_i \texttt{ and setMatch t rps}_i\}$

These rules are labelled with '`c`' to reflect their cumulative behavior. The `cpass` rule says what tuples should be emitted to the output by this point, namely tuples that are in the left input, not currently in the right input and that will not appear in the right input in the future. The `cprop` rule says what punctuation can be emitted, in this case `combine lp rp` for all `lp` in $\texttt{lps}_i$ and `rp` in $\texttt{rps}_i$. The two purge functions say which tuples can be discarded from the inputs at this point with out impairing our ability to process later inputs correctly. From the left input we can toss any tuple that has appeared in the right input (it will never be emitted), or for which no matching tuple will every appear in the right input (so it has already been emitted). From the right input we toss any tuple where there will be no further matching tuples on the left input.

We first show that any stream iterator that conforms to `cpass` is faithful to table difference. Second, we show that we can drop the tuples indicated by the `cpurge` rules without impairing future computation, as long as the tuples in `cpass` have been emitted already (so dropping these tuples will not compromise faithfulness). Third, we show that emitting any of the punctuation in `cprop` after the tuples in `cpass` will be grammatical. Finally, we show that `sdiff` conforms to those rules, and hence is faithful.

Faithfulness (safety): We need to show that

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

for any lists `ls` and `rs` where $\texttt{match ls lps}_i = \emptyset$ and $\texttt{match rs rps}_i = \emptyset$. 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 $\texttt{setMatch t rps}_i$. 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 $\texttt{t} \notin \texttt{rps}_i\texttt{++rs}$. Thus `t` is in $\texttt{(lts}_i\texttt{++ls) - (rts}_i\texttt{++rs)}$ and the containment is proved.

Faithfulness (completeness): Suppose $\texttt{t} \in \texttt{(lts}_i\texttt{++ls) - (rts}_i\texttt{++rs)}$ for every `ls` and `rs` where $\texttt{setMatch ls lps}_i = \texttt{setMatch rs rps}_i = \emptyset$. Choosing `ls` to be `[ ]`, we must have $\texttt{t} \in \texttt{(lts}_i\texttt{++[ ]) - (rts}_i\texttt{++rs)}$, hence $\texttt{t} \in \texttt{lts}_i\texttt{++[ ]}$ and thus $\texttt{t} \in \texttt{lts}_i$. We know $\texttt{t} \notin \texttt{rts}_i$, otherwise it would be in $\texttt{rts}_i\texttt{++rs}$. Consider any tuple `s` where $\texttt{setMatch s rps}_i = \texttt{False}$. 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 $\texttt{setMatch t rps}_i = \texttt{True}$. Since $\texttt{t} \in \texttt{lts}_i$, $\texttt{t} \notin \texttt{rts}_i$ and $\texttt{setMatch t rps}_i$, we have $\texttt{t} \in \texttt{cpass lts}_i\ \texttt{lps}_i\ \texttt{rts}_i\ \texttt{rps}_i$, as required for completeness.

Purging: Consider some later point `j`, and define $\texttt{lts}_j$, $\texttt{lps}_j$, $\texttt{rts}_j$ and $\texttt{rps}_j$ analogously to $\texttt{lts}_i$, $\texttt{lps}_i$, $\texttt{rts}_i$ and $\texttt{rps}_i$. Let $\overline{\texttt{lts}_i} = \texttt{cpurgeL lts}_i\ \texttt{lps}_i\ \texttt{rts}_i\ \texttt{rps}_i$ and $\overline{\texttt{rts}_i} = \texttt{cpurgeR lts}_i\ \texttt{lps}_i\ \texttt{rts}_i\ \texttt{rps}_i$. We want to show that

$$\begin{aligned}\texttt{cpass lts}_j\ \texttt{lps}_j\ \texttt{rts}_j\ \texttt{rps}_j = {} & \texttt{cpass lts}_i\ \texttt{lps}_i\ \texttt{rts}_i\ \texttt{rps}_i \cup \\ & \texttt{cpass (lts}_j\texttt{ - }\overline{\texttt{lts}_i}\texttt{) lps}_j\texttt{ (rts}_j\texttt{ - }\overline{\texttt{rts}_i}\texttt{) rps}_j.\end{aligned}$$

That is, if we have emitted all of $\texttt{cpass lts}_i\ \texttt{lps}_i\ \texttt{rts}_i\ \texttt{rps}_i$ at time `i`, then we can produce the correct additional output at time `j` without the tuples in $\overline{\texttt{lts}_i}$ and $\overline{\texttt{rts}_i}$.

left $\subseteq$ right: If $\texttt{t} \in \texttt{cpass lts}_j\ \texttt{lps}_j\ \texttt{rts}_j\ \texttt{rps}_j$, then $\texttt{t} \in \texttt{lts}_j$, $\texttt{t} \notin \texttt{rts}_j$ (hence $\texttt{t} \notin \texttt{rts}_i$) and $\texttt{setMatch t rps}_j$. (Case 1) Assume $\texttt{setMatch t rps}_i$. If $\texttt{t} \in \texttt{lts}_i$, then $\texttt{t} \in \texttt{cpass lts}_i\ \texttt{lps}_i\ \texttt{rts}_i\ \texttt{rps}_i$. If $\texttt{t} \in \texttt{lts}_j\texttt{ - lts}_i$, then $\texttt{t} \in \texttt{lts}_j\texttt{ - }\overline{\texttt{lts}_i}$, since $\overline{\texttt{lts}_i}$ is smaller than $\texttt{lts}_i$. Since $\texttt{t} \notin \texttt{rts}_j$, then $\texttt{t} \notin \texttt{rts}_j\texttt{ - }\overline{\texttt{rts}_i}$. We already have $\texttt{setMatch t rps}_j$. So $\texttt{t} \in \texttt{cpass (lts}_j\texttt{ - }\overline{\texttt{lts}_i}\texttt{) lps}_j\texttt{ (rts}_j\texttt{ - }\overline{\texttt{rts}_i}\texttt{) rps}_j$. (Case 2) Assume $\texttt{setMatch t rps}_i = \texttt{False}$. Since $\texttt{t} \notin \texttt{rts}_i$, then $\texttt{t} \notin \overline{\texttt{lts}_i}$, since neither alternative for membership in $\overline{\texttt{lts}_i}$ is fulfilled. So $\texttt{t} \in \texttt{lts}_j\texttt{ - }\overline{\texttt{lts}_i}$. Since $\texttt{t} \notin \texttt{rts}_i$, then $\texttt{t} \notin \texttt{rts}_j\texttt{ - }\overline{\texttt{rts}_i}$. We have $\texttt{setMatch t rps}_j$, so $\texttt{t} \in \texttt{cpass (lts}_j\texttt{ - }\overline{\texttt{lts}_i}\texttt{) lps}_j\texttt{ (rts}_j\texttt{ - }\overline{\texttt{rts}_i}\texttt{) rps}_j$.

right $\subseteq$ left: omitted.

Propriety: Suppose we emit the punctuation in $\texttt{cprop lps}_i\ \texttt{rps}_i$ at point `i`, and that all the tuples in $\texttt{cpass lts}_i\ \texttt{lps}_i\ \texttt{rts}_i\ \texttt{rps}_i$ have appeared. We want to makes sure that any tuple `t` emitted later does not match this punctuation. Let $\texttt{t} \in \texttt{cpass lts}_j\ \texttt{lps}_j\ \texttt{rts}_j\ \texttt{rps}_j$ where $\texttt{setMatch t (cprop lps}_i\ \texttt{rps}_i\texttt{)}$. It must then be that $\texttt{setMatch t lps}_i$ and $\texttt{t} \notin \texttt{rts}_i$ since `t` had to match some `combine lp rp` where $\texttt{lp} \in \texttt{lps}_i$ and $\texttt{rp} \in \texttt{rps}_i$. Since $\texttt{t} \in \texttt{lts}_j$ but matches $\texttt{lps}_i$, `t` must already have appeared in $\texttt{lts}_i$. Since $\texttt{t} \notin \texttt{rts}_j$, also $\texttt{t} \notin \texttt{rts}_i$. Since $\texttt{t} \in \texttt{lts}_i$, $\texttt{t} \notin \texttt{rts}_i$ and $\texttt{t} \notin \texttt{rts}_j$, we see that `t` was already emitted as part of $\texttt{cpass lts}_i\ \texttt{lps}_i\ \texttt{rts}_i\ \texttt{rps}_i$.

Conformance: We show that `sdiff` conforms to the rules, hence is faithful.

$$\begin{array}{lll}\text{let} & \texttt{cpass}_i & = \texttt{cpass lts}_i\ \texttt{lps}_i\ \texttt{rts}_i\ \texttt{rps}_i \\ & \texttt{(ols}_{i+1}\texttt{, stls}_{i+1}\texttt{)} & = \texttt{lstep ltsnew}_{i+1}\ \texttt{(lts}_i\texttt{ - cpass}_i\texttt{, rt}s_i\texttt{, lps}_i\texttt{, rps}_i\texttt{)} \\ & \texttt{(ors}_{i+1}\texttt{, strs}_{i+1}\texttt{)} & = \texttt{rstep rtsnew}_{i+1}\ \texttt{stls}_{i+1} \\ & \texttt{olp}_{i+1} & = \texttt{lpass lpsnew}_{i+1}\ \texttt{strs}_{i+1}\end{array}$$

$$
\begin{aligned}
&\mathtt{orp}_{i+1} && = \mathtt{rpass\ rpsnew}_{i+1}\ \mathtt{strs}_{i+1} \\
&\mathtt{stls}_{i+1} && = (\mathtt{ltsnew}_{i+1}\ \mathtt{++}\ \mathtt{lts}_i\ \mathtt{-}\ \mathtt{cpass}_i,\ \mathtt{rts}_i,\ \mathtt{lps}_i,\ \mathtt{rps}_i) \\
& && = (\mathtt{lts}_{i+1}\ \mathtt{-}\ \mathtt{cpass}_i,\ \mathtt{rts}_i,\ \mathtt{lps}_i,\ \mathtt{rps}_i) \\
&\mathtt{strs}_{i+1} && = (\mathtt{lts}_{i+1}\ \mathtt{-}\ \mathtt{cpass}_i,\ \mathtt{rtsnew}_{i+1}\ \mathtt{++}\ \mathtt{rts}_i,\ \mathtt{lps}_i,\ \mathtt{rps}_i) \\
& && = (\mathtt{lts}_{i+1}\ \mathtt{-}\ \mathtt{cpass}_i,\ \mathtt{rts}_{i+1},\ \mathtt{lps}_i,\ \mathtt{rps}_i) \\
\text{then }&\mathtt{incrts}_{i+1} && = \mathtt{ols}_{i+1}\ \mathtt{++}\ \mathtt{ors}_{i+1}\ \mathtt{olp}_{i+1}\ \mathtt{++}\ \mathtt{orp}_{i+1} \\
& && = \mathtt{[]\ ++\ []\ ++\ []\ ++}\ \mathtt{orp}_{i+1} \\
& && = \mathtt{rpass\ rpsnew}_{i+1}\ \mathtt{strs}_{i+1} \\
& && = \mathtt{rpass\ rpsnew}_{i+1}\ (\mathtt{lts}_{i+1}\ \mathtt{-}\ \mathtt{cpass}_i,\ \mathtt{rts}_{i+1},\ \mathtt{lps}_i,\ \mathtt{rps}_i) \\
& && = \mathtt{setMatchTs}\ (\mathtt{lts}_{i+1}\ \mathtt{-}\ \mathtt{cpass}_i\ \mathtt{-}\ \mathtt{rts}_{i+1})\ (\mathtt{rpsnew}_{i+1}\ \mathtt{++}\ \mathtt{rps}_i) \\
& && = \mathtt{setMatchTs}\ (\mathtt{lts}_{i+1}\ \mathtt{-}\ \mathtt{cpass}_i\ \mathtt{-}\ \mathtt{rts}_{i+1})\ \mathtt{rps}_{i+1} \\
& && = [\mathtt{t}\ |\ \mathtt{t} \in (\mathtt{lts}_{i+1}\ \mathtt{-}\ \mathtt{cpass}_i\ \mathtt{-}\ \mathtt{rts}_{i+1}),\ \mathtt{setMatch\ t\ rps}_{i+1}] \\
& && = [\mathtt{t}\ |\ \mathtt{t} \in \mathtt{lts}_{i+1},\ \mathtt{t} \notin \mathtt{cpass}_i,\ \mathtt{t} \notin \mathtt{rts}_{i+1},\ \mathtt{setMatch\ t\ rps}_{i+1}] \\
& && = [\mathtt{t}\ |\ \mathtt{t} \in \mathtt{lts}_{i+1},\ \mathtt{t} \notin \mathtt{rts}_{i+1},\ \mathtt{setMatch\ t\ rps}_{i+1}]\ \mathtt{-}\ \mathtt{cpass}_i \\
& && = (\mathtt{cpass\ lts}_{i+1}\ \mathtt{lps}_{i+1}\ \mathtt{rts}_{i+1}\ \mathtt{rps}_{i+1})\mathtt{-}(\mathtt{cpass\ lts}_i\ \mathtt{lps}_i\ \mathtt{rts}_i\ \mathtt{rps}_i)
\end{aligned}
$$

End of proof.

4 Related Work

Data stream processing has been researched for a number of years. The Tangram stream query processing system [PMC89] is an early work using database technology on stream input. They discuss a number of stream transducer, including blocking operators such as sort and difference, but do not discuss how these behave in the presence of continuous streams. Parker [Ste90] also discusses in more detail a model for stream transducers that is very similar to ours, though he limits the discussion to unary stream transducers, and does not address how they interact with continuous data streams.

Recently, data stream processing has attracted new interest in the database community. Babu et al. [BW01] describe an architecture for queries that process streams. A query has one or more input streams, an output stream, a *store* for tuples that might be part of the result, and a *scratch* for state. Punctuations can be used in this architecture in two ways: Implementing pass rules minimizes the size of store by moving tuples to the output sooner, and implementing purge rules minimizes the size of scratch.

The Tribeca system [SH98] implements a limited set of relational operators for data streams. Implementations for select and project operate on the entire input stream. Other operators, such as aggregation and join, are implemented only over subsections of the stream described by *fixed windows* and *moving windows*. A fixed window divides the stream into fixed sequential chunks. When a chunk completes, the aggregation or join is computed on it. Operators compute results over a moving window by considering tuple from the current tuple backwards for the length of the window. Fixed window can be considered a limited form of punctuated streams. Blocking operators output results at the end of each window. However, fixed windows only delineate points in the data stream, and do not describe subsets of data based on the semantics of the data. Gehrke et al. [GKS01] use similar structures to compute correlated aggregates over data streams.

Work on partial results by Shanmugasundaram et al. [STD+00] also addresses blocking operators. In this system, query operators that are waiting for output from other, perhaps blocking, operators, can request a partial result. This situation is very similar to a moving window, except that the window's parameters are determined while the query is executing, rather than before. This work does not address unbounded stateful operators.

Querying ordered sequence data is also related [SLR94]. Each tuple in a sequence can be considered a punctuation, since no more tuples will appear in the sequence that are ordered before it. Therefore, techniques for optimizing queries for sequences should be applicable to queries on punctuated streams, which is an area for future work.

5 Conclusions and Future Work

We have presented a framework that allows query operators to exploit punctuation semantics in continuous data streams. Our method improves many table operators, and specifically targets blocking and unbounded stateful operators. We define stream iterators as a foundation for our work, and describe a specific framework for modelling stream iterators on continuous data streams. We define desirable properties for stream iterators, and give prove that our rules meet those properties for the difference operator.

We have implemented much of this framework into the Niagara Query Engine [NDM$^+$00], written in Java. Niagara executes queries over XML data. We defined a punctuation format in XML. We added implementations of punctuation rules to many of Niagara's query operators, and enhanced it to read from data streams using our own "XML Firehose" technology.

There is more work to be done. We need to expand our formal model and implement more rules, including modeling more operators in our Haskell code base, and proving properties about those operators. When satisfied with their definition and correctness, we will add them to Niagara.

Not all punctuation schemes will unblock a specific operator, which raises several questions: Given query Q, will a given punctuation scheme on the inputs fully unblock Q? Is there some rewrite of Q that is unblocked? Slightly harder, does a punctuation scheme exist that unblocks Q.

Query operators themselves might be able to create punctuations. For example, the select operator might be able to embed punctuation based on its predicate. If a query includes a selection on phone numbers where hour = 8, then select could output two punctuations before it even reads the first tuple from the stream, describing tuples with hours less than 8, and hours greater than 8.

There is common work here with partial evaluation of queries [STD$^+$00]. Both address applying blocking operators to long or continuous inputs. Punctuation might be used to help partial operators to communicate output that is "possible" versus output that is "certain".

6 Acknowledgements

The authors would like to thank Kristin Tufte and Vassilis Papadimos for the discussions we had, Levent Erkok for his help in developing the Haskell framework, Raghu Ramakrishnan for his pointers to related work, and David DeWitt for his advice on the direction of this work. Funding for this work was provided by DARPA through NAVY/SPAWAR Contract No. N66001-99-1-8908, by NSF ITR award IIS0086002, and by NSF grant IIS-9811525.

References

[BW01] Shivnath Babu and Jennifer Widom. Continuous queries over data streams. *SIGMOD Record*, 30(3), September 2001.

[GKMS01] Anna C. Gilbert, Yannis Kotidis, S. Muthukrishnan, and Martin J. Strauss. Surfing wavelets on streams: One-pass summaries for approximate aggregate queries. In *Proceedings of the 27th Conference on Very Large Data Bases*, pages 79–88, September 2001.

[GKS01] Johannes Gehrke, Flip Korn, and Divesh Srivastava. On computing correlated aggregates over continuous data streams. In *Proceedings of the ACM Special Interest Group on Management of Data*, pages 13–24, May 2001.

[HC98] Thomas Hallgren and Magnus Carlsson. *Fudgets.* PhD thesis, Chalmers University of Technology, March 1998.

[LM89] Brian K. Livezey and Richard R. Muntz. Aspen: A stream processing environment. In *PARLE*, pages 374–388, 1989.

[NDM+00] Jeffrey Naughton, David DeWitt, David Maier, Jianjun Chen, Leonidas Galanis, Kristin Tufte, Jaewoo Kang, Qiong Luo, Naveen Prakash, and Feng Tian. The niagara query system. Technical report, University of Wisconsin, March 2000. URL:http://www.cs.wisc.edu/niagara/papers/NIAGRAVLDB00.v4.pdf.

[PMC89] D. Stott Parker, Richard R. Muntz, and Lewis Chau. The tangram stream query processing system. In *Proceedings of the Fifth International Conference on Data Engineering*, February 1989.

[SH98] Mark Sullivan and Andrew Heybey. Tribeca: A system for managing large databases of network traffic. In *Proceedings of the 1998 USENIX Annual Techincal Conference*, June 1998.

[SLR94] Praveen Seshadri, Miron Livny, and Raghu Ramakrishnan. Sequence query processing. In *Proceedings of the ACM Special Interest Group on Management of Data*, pages 430–441, May 1994.

[STD+00] Jayavel Shanmugasundaram, Kristin Tufte, David J. DeWitt, Jeffrey Naughton, and David Maier. Architecting a network query engine for producing partial results. In *WebDB (Informal Proceedings)*, pages 17–22, May 2000.

[Ste90] L. Sterling, editor. *The Practice of Prolog*, chapter 8. Stream Data Analysis in Prolog. MIT Press, 1990.

A Additional Punctuation Rules for Specific Iterators

We discuss rules for stream analogues of other traditional table operators. We have deduced these rules from our intuition. Constructing formal proofs for these rules remains for future work.

A.1 Pass Rule for Group By

Group by must wait until it is sure that all tuples for a group have been read before outputting results for that group. When a punctuation arrives that describes a group (or a set of groups) completely, then results for that group (or groups) can be emitted. Formally, the pass rule for group by is:

$\texttt{cpass ts}_i\ \texttt{ps}_i\texttt{=}\ \{\texttt{t | t} \in \texttt{ts}_i\texttt{, grpMatch ps}_i\}$

The `grpMatch` function returns punctuations in `ps` that completely describe a group.

A.2 Purge Rule for Group By

The purge rule for the group by operator is identical to its pass rule. When results for a particular group have been emitted, the state for that group can be discarded. Formally, $\texttt{cpurge lts}_i\ \texttt{lps}_i$ $\texttt{= cpass lts}_i\ \texttt{lps}_i$.

A.3 Purge Rule for Duplicate Elimination

When punctuation arrives, the stream iterator for duplicate elimination knows there will be no more duplicates for tuples that match that punctuation, so state for matching tuples can be discarded. Formally, $\texttt{cpurge ts}_i\ \texttt{ps}_i\ \texttt{= setMatchTs ts}_i\ \texttt{ps}_i$.

A.4 Purge Rule for Intersection

When a punctuation is passed in from one input, the intersect stream iterator can purge state for its other input that match the punctuation. Since no more tuples will arrive from that input that match the tuple, there is no reason to hold tuples in state from the other input. Formally,

$\texttt{cpurgeL lts}_i\ \texttt{lps}_i\ \texttt{rts}_i\ \texttt{rps}_i\ \texttt{= setNomatchTs rts}_i\ \texttt{lps}_i$

$\texttt{cpurgeR lts}_i\ \texttt{lps}_i\ \texttt{rts}_i\ \texttt{rps}_i\ \texttt{= setNomatchTs lts}_i\ \texttt{rps}_i$