Enhancing Relational Operators for Querying over Punctuated Data Streams

Pete Tucker[1] David Maier[1] Tim Sheard[1] Leonidas Fegaras[2]

[1]OBI School of Science and Engineering at OHS
20000 NW Walker Road
Beaverton, OR, USA
{ptucker,maier,sheard}@cse.ogi.edu

[2]University of Texas at Arlington
701 S. Needlewoman Drive
Arlington, TX, USA
fegaras@cse.uta.edu

Abstract

As most current query processing architectures are already pipelined, it seems logical to try 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 its size, and so run out of memory. Blocking operators read the 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 rules to specify the proper behavior of operators in the presence of punctuation. *Pass rules* unblock blocking operators by defining when such an operator can pass results on. *Purge rules* decrease the size of state by defining when an unbounded stateful operator can discard state. *Propagation rules* define when an operator can pass punctuation on. We report on our initial implementation, and show a strategy for proving stream operators that implement these rules are faithful to their table counterparts.

Proceedings of the 28th VLDB Conference, Hong Kong, China, 2002

1 Introduction

There are numerous industry and research examples of applications that process data streams: Financial applications process streams of stock ticker data; telephone monitoring applications process streams of call data [4, 6]; traffic monitoring applications process streams from sensors on the highway [9]. As a running example, consider a warehouse containing temperature-sensitive merchandise. Sensors that report temperatures are scattered throughout the warehouse. In order to monitor the temperature throughout the warehouse, we want an application that outputs the maximum temperature reported from any sensor each hour.

Stream-processing applications can often be viewed as queries over data streams. As such, it is desirable to use a DBMS to implement these applications. However, two properties of data streams make them difficult for a DBMS to process: First, data is often generated by a device that can potentially produce a continuous data stream. Second, a stream's contents can only be accessed sequentially, and cannot be reset to the beginning.

We could try to implement the warehouse monitoring system in a DBMS with a query that unions tuples from all of the temperature sensors, then groups the tuples by hour, and outputs the maximum temperature for that hour, as shown in Figure 1. Each report from the sensors is streamed into the union operator. The data is merged, and duplicates are removed. The output from the union operator is sent to the group-by operator, and it determines the maximum temperature for each hour.

Unfortunately, this system will fail for two reasons: First, the union operator removes all duplicates in its output. This ability requires an unbounded amount of state, which means that the system will ultimately run out of memory trying to process its inputs. Second, the group-by operator must wait until all data from its input has been read be-

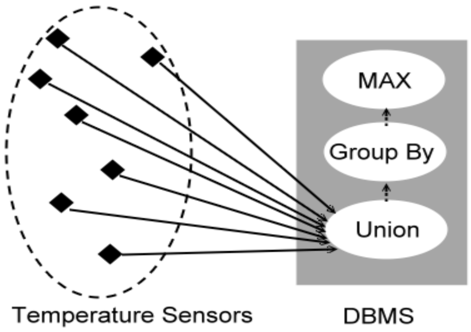

Figure 1: System to monitor the temperature readings from various places within a warehouse.

Operator	Blocking?	Unbounded State?
select	No	No
project	No	No
dupelim	No	Yes
join	No	Yes
merge	No	No
intersect	No	Yes
difference	Yes	Yes
group-by	Yes	Yes
sort	Yes	Yes

Table 1: Traditional relational database operators, and whether they have properties that make them inappropriate for data stream processing.

fore it can output results. Since the data is coming from a continuous data stream, it will not emit a result.

1.1 Motivation for Punctuated Streams

This 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, exploiting stream semantics to overcome these problems.

Two things happen at the end of an input's data with table operators: blocking operators produce results, and unbounded stateful operators purge state. Our work extends this behavior for querying on continuous streams. We embed information into the stream itself that describes the end of a *subset* of data. We call this information textitpunctuations. Informally, a punctuation says that no more tuples will follow that match the punctuation. Blocking operators can use this information to output results before encountering the end of the stream. Unbounded stateful operators can also use this information, to purge unnecessary state before encountering the end of the stream. Punctuations allow a larger range of queries on continuous data streams. A list of traditional relational database operators and whether they are blocking or unbounded stateful operator is shown in Table 1. Note that we treat the union operator as a combination of two operators: merge and duplicate elimination.

Embedding punctuations in data streams can fix the problems in the warehouse example. Suppose tuples from the sensors contain `sensorid`, `hour`, `minute`, and `currenttemp`. Also, suppose the sensors have been modified to embed punctuations each hour, stating that all reports have been read for a particular hour. When the union operator has seen punctuations from all sensors for a particular hour, it can deduce that there will be no more tuples for that hour. It can purge all tuples corresponding to that hour in its state. Likewise, when the group-by operator gets punctuation for a particular hour, it can output the results for groups that match the punctuation. Additionally, state relating to those groups can be purged. These specific enhancements are discussed further in Section 3.

1.2 Other Examples

There are other situations where users would like to execute queries over data streams, but the operators required are not appropriate for streams. An electric company, for example, might implement an outage tracking system in the following manner: Monitoring devices installed at substations stream problem reports to the main system. Customers report problems to an automated phone bank. These reports are streamed from the phone bank to the main system. A difference operator is used to filter out customer reports that coincide with problems reported by the monitor devices. Unfortunately, the difference operator must wait until all of its data from the negative input (monitor reports) has been read before it can output results. Punctuations can be used to fix this problem. Suppose the tuples for both inputs contain `zoneid` and `time`, and the subsystem devices and the phone bank have been modified to embed punctuations stating that all reports have been received for a particular zone and hour. When the difference operator receives punctuation on its negative input, it can output tuples from the positive input that match that punctuation and have not been seen in the negative input. Additionally, those tuples that were output can also be purged.

The rest of this paper is organized as follows: We lay a groundwork for punctuations in Section 2. Section 3 discusses how we extended an existing XML query engine to use punctuation. Section 4 describes how we define and model operators that process data

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 2: Match conditions for each pattern, given attribute value i

streams. Section 5 gives a more formal description of punctuations, and Section 6 shows how we enhance our stream model in the presence of punctuated streams. We discuss related work in Section 7, and conclude in Section 8.

2 A Foundation for Punctuation

A punctuation might be seen simply as a predicate on stream elements that must evaluate to `false` for all elements that follow the punctuation. Thus we might represent punctuations as "black box" Boolean functions. However, we represent punctuation as data to allow easy storage, searching and manipulation. Our current punctuation scheme is fairly simple. However, it has the important property that the "and" of any two punctuation is itself a punctuation.

Currently, elements in our streams are tuples of scalars and punctuations for such tuples. We intend to handle tuples with nested values, e.g. XML, in the future. A punctuation is patterns, each pattern corresponds to an attribute of a tuple. A tuple matches a punctuation if each attribute in the tuple matches the corresponding pattern in the punctuation. We define five kinds of patterns, and a match condition for each, as shown in Table 2.

For example, if the tuple structure for the temperature sensor reports is: <sensorid, hour, minute, temp>, a punctuation that describes all reports for hour 3 would be <*, 3, *, *>. Also, a punctuation that describes all reports from sensors `S1`, `S5`, and `S6` for hours between 5 and 8 would be <{1,5,6}, [5,8], *, *>.

We define two basic functions for manipulating punctuated streams. First, the `match` function takes a punctuation `p` and a tuple `t`. It returns `True` if `t` satisfies the predicate described by `p`. We define two special, additional punctuations: The punctuation Π matches all tuples, that is, $\forall$ `t, match t` Π `= True`. The punctuation Ψ matches no tuples. That is, $\forall$ `t, match t` Ψ `= False`. One can think of Π as punctuation that has `wildcard` for all its attributes, and Ψ as punctuation that has `empty` for all its attributes.

The second function we define is the `combine` function, which takes two punctuations. It returns a new punctuation that is either the intersection of the two inputs or Ψ, indicating that no valid combination exists. Any tuple that matches the output from `combine` must match both of the input punctuations. That is, `match t (combine` p_1 p_2`)` $\Leftrightarrow$ `match t` p_1 $\wedge$ `match t` p_2. For shorthand, we will use p_1&p_2 to mean `combine` p_1 p_2. We also define auxiliary functions that operate on lists of inputs. A subset of these are shown in Table 3.

Function	Return Value
`tups(xs)`	tuples in `xs`
`puncts(xs)`	punctuations in `xs`
`nomatch(t,p)`	$\neg$ `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` $\in$ `ts` matching any `ps`
`setNomatchTs(ts,ps)`	all `t` $\in$ `ts` matching no `ps`
`setNomatchPs(ts,ps)`	all `p` $\in$ `ps` matching no `ts`
`setCombine(ps1,ps2)`	all combinations of punctuations in `ps1`, `ps2`

Table 3: Auxiliary 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 Implementation

To get a feeling for the complexity of implementing punctuations and their effect on performance, we enhanced the Niagara Query Engine [10]. Niagara executes queries over XML data. We defined a punctuation format for XML, and enhanced some of Niagara's query operators to exploit the punctuation. We then used our implementation with a query for the warehouse example described earlier.

3.1 XML Punctuation Format

Our punctuation format for XML has three goals: First, the size of a punctuation should be similar to the size of the tuples it describes. Second, punctuations should not affect the results of query operators that do not understand punctuations. Third, query operators that do understand punctuations should be able to easily determine the subset of data described by a punctuation.

We achieve these goals using the *Namespaces for XML* recommendation [3]. We define a namespace called `punct`, and define an element in it that mirrors a tuple structure. Query operators 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`. A sample XML tuple and punctuation are shown in Table 4. We also define a format for each of the patterns shown in Table 2.

tuple	punctuation
<tempdata>	<punct:tempdata>
<id>S01</id>	<id>*</id>
<hour>17</hour>	<hour>17</hour>
<min>30</min>	<min>*</min>
<temp>77</temp>	<temp>*</temp>
</tempdata>	</punct:tempdata>

Table 4: An XML tuple and matching punctuation

3.2 Enhancements to Niagara Operators

We will describe modifications made to Niagara that are used in the warehouse query, namely union and group-by. The query itself is as follows:

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

3.2.1 Union

Since our union operator performs duplicate elimination, it is an unbounded stateful operator. We use punctuations to decrease the amount of state required by the operator. For duplicate elimination, our implementation stores tuples that have arrived in a hash table. If a new tuple arrives that already exists in the hash table, it is not output. If it does not exist in the hash table, it is output and added to the hash table. A punctuation tells the operator that no more tuples that match it will follow it in the stream. When the union operator receives punctuations from all inputs that match the same set of tuples, it can purge tuples from the hash table that match that punctuation.

Also, we want the union operator to pass punctuations on to the next operator (group-by). It cannot simply pass on punctuations as they arrive. When a punctuation arrives from one input, it is still possible for a tuple that matches that punctuation to arrive on the other input. We must make sure that any punctuations output from the operator agree with punctuations received from both of the inputs.

3.2.2 Group-by

The group-by operator is both a blocking operator and an unbounded stateful operator. We use punctuations to output results early and to purge unnecessary state. The traditional implementation of the group-by operator 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. Further, we can use that same punctuation to purge state for such a group, thus reducing the state held by the group by operator.

3.3 Results

We simulated sensors with Java applications to stream data into the Niagara query engine. These sensors were programmed to 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. To determine the additional cost of embedding punctuations, our sensors outputted data for each minute instantaneously, to determine how long it took to run each query. The data was over the first 60 hours.

We see from Figure 2 that the amount of memory use by the hash table in the union operator is greatly reduced. For streams that are 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 in fact bounded, since each tuple that arrives will match a punctuation within one hour.

We see from Figure 3 that embedding punctuations into the stream unblocks the group-by operator, allowing it to output results at the conclusion of each hour. Without punctuations, the operator must wait until the input has completed before outputting results. We can 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 can be attributed to the fact that the hash table used in the union operator has a bounded state when punctuations exist in the data stream. If punctuations do not exist, then the hash table grows and grows, forcing it to allocate more and more memory.

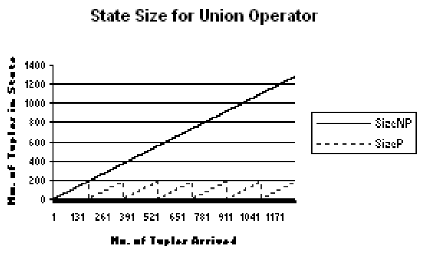

Figure 2: The state held by the union operator as tuples arrive. The dotted line shows the size of state when punctuations are present, and the solid line shows the size of state when punctuations are not.

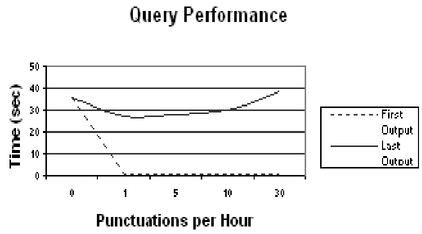

Figure 3: Query Performance. The dashed line indicates when the first tuple arrived, and the solid line is the total time to execute the query.

3.4 Discussion

Our experience with our extensions with the Niagara system indicated that the cost overhead of handling punctuations was not prohibitive, and hence worth pursuing further. At the same time, the ad-hoc approach we used for modifying the operators was not completely satisfactory for two reasons: First, while we are convinced our extended operators "do the right thing", we had no formal notion of stream operators or correctness. Second, we lacked a principled approach to constructing stream operators. Thus, we began a study of semantics and correctness of stream operators (Sections 4 and 5) and developed a common framework for defining stream operator implementations (Section 6). These developments have provided a common control structure for stream operators (thus reducing implementation effort, while at the same time facilitating proofs of correctness).

4 Semantics of Stream Iterators

Traditional queries are comprised of operators on finite tables. To execute queries on data streams, we need to define operators that are functions from input streams to output streams. Parker [11] calls such operators *stream transducers*. Not all stream-to-stream functions are reasonable candidates for table analogues. Consider sorting for collections of positive integers. For an infinite stream `S` of positive integers, we can define a function `sort(S)` that returns the same collection of integers as a stream in increasing order. However, we cannot implement such a function in a computer system, 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 (at least for streams over certain domains). Our intuition is that 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 can be defined in terms of a sequence of partial computations over finite prefixes of an infinite stream. Thus, a stream iterator is never presented with the entire contents of an infinite stream at one time, but rather sees increasing (but finite) portions of it.

To define this notion more precisely, we need a data type for streams. For the moment, we shall represent them as infinite sequences, much 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 `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, ...|`$\}$. We then need a function that extracts the first `i` elements from a stream (and returns it as a list of the appropriate type), and so has the signature `Stream(T)` $\rightarrow$ `Int` $\rightarrow$ `List(T)`. If `S` is a stream, we will write `S[i]` for the list of the first `i` elements. Thus, if `S` is the stream above, `S[3]` is `[1, 3, 5]`. Further, for `n > i`, we use `S[i`$\rightarrow$`n]` for the list of elements from `i+1 to n`. Note that `S[i] = S[0`$\rightarrow$`i]`. We use `S@i` to mean the $\mathtt{i}^{th}$ element of the stream. We also 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, it is easily seen that select can be expressed as a stream iterator. If the selection predicate is p, then we define q by:

```
q(S[i]) = [S@i] if p(S@i)
q(S[i]) = [ ] otherwise.
```

That is, `q` just looks at the last element of each prefix, and emits it if it satisfies the predicate. There is also a stream iterator for duplicate elimination. It uses `q` such that:

```
q(S[i]) = [S@i] if S@i not in S[i-1]
q(S[i]) = [ ] otherwise.
```

That is, `q` just checks that the final element in the prefix doesn't appear earlier in the prefix. However, it should be obvious that sort cannot be expressed as a stream iterator. Another table operator without a stream-iterator analogue is group-by. We can never emit the aggregate for a group, because there may still be members of the group coming later in the stream. (We are skirting an important issue in these examples, namely what is an appropriate extension of a table operator to infinite streams. We are assuming here that stream versions of select and duplicate elimination should behave as described. In Section 5.5 we will formally define what it means for a stream iterator to be *faithful* to table operator.)

4.1 Motivation for Punctuations

Given that we now have stream iterators as definitions of what constitutes a "practical" stream-to-stream function, can we do anything about 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 if the end of the stream is seen. One way to represent a finite stream in our infinite sequence formulation of streams is to add a special value called `EOS` to the domain of the stream elements 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 just says we can provide a result in the finite case, which we already knew. However, we might have information about a stream that puts us between the finite and infinite cases. Suppose we knew at some point in the stream that we would not see any more elements in the range 1 to 10. At that point (assuming that we are still working with positive integers), we could emit part of the output. Suppose we use `EOS(1-10)` to indicate this condition. Then on a stream such as {| `8, 2, 6, 11, 3, EOS(1-10), 12, 24, 15, EOS(11-20), 28, 21, ...` |} we could have `q` emit `[2, 3, 6, 8]` on prefix `[8, 2, 6, 11, 3, EOS(1-10)]` and emit `[11, 12, 15]` on the prefix that ends with `EOS(11-20)`. We call an annotation such as `EOS(1-10)` in a stream a *punctuation*, as it signals the end of a certain part of the input. Note that punctuation doesn't require that the stream be partitioned into ranges of values: the value `11` in the stream above appears before `EOS(1-10)`. Punctuation only speaks about what comes subsequently in a stream.

The remainder of this paper discusses our framework for defining stream operators that exploit the knowledge provided by punctuation. Punctuation in a stream will let us "pass along" outputs early that we wouldn't be able to emit if we had no knowledge of the remainder of the stream. We can also "propagate" punctuation into the output of a stream iterator for use by other stream iterators. For instance, in the example above, `q` could emit `[2, 3, 6, 8, EOS(1-10)]` on prefix `[8, 2, 6, 11, 3, EOS(1-10)]`, possibly allowing subsequent stream operators to pass along outputs early.

4.2 Representation Issues

We have presented one formulation for data streams and "realizable" functions over those streams. In our investigations we have come to adopt a slightly different representation of streams and stream iterators than the one used here for introductory purposes. Rather than representing a data stream by an infinite sequence of elements, we model it as an infinite sequence of lists of elements – sort of a "sliced list". Thus for example the stream {| `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 this change to sliced-list form:

- It allows direct modeling of finite streams, using a trailing sequence of empty slices: {| `[1, 2], [3], [4], [ ], [ ], [ ], [ ], ...` |}. Note that the sliced representation doesn't represent quite the same range of streams as the infinite sequence representation. Any infinite sequence has an equivalent sliced list representation (in fact, an infinite number of them), but some sliced lists have no equivalent infinite sequence, such as the one above.

- It intuitively captures that stream arrivals are not synchronized with operator iterations. In the infinite sequence representation, there is always a single next element available when reading from a stream. However, in practice, we might see multiple arrivals when we look at an input, or no arrivals for an arbitrarily long time. Processing a slice at a time means an operator has freedom on what order to process inputs that arrive "at the same time". It also means an operator must be prepared for arbitrarily long stretches with no input (and it cannot tell a priori if any more input is coming, in the absence of punctuation).

- It generalizes to operators with multiple inputs better than the infinite sequence representation. We do not want to assume that elements of different input streams arrive "in step". We had originally thought to present two input streams as a "merge" of their infinite sequences where elements are tagged as from the left or right input. However, there can be unfair merges, where all or part of one of the inputs is delayed indefinitely. Instead of the merged presentation, an operator sees separate sliced lists for each input, and processes a slice from each list in tandem. The slicing models different arrival rates between the input streams, without having to define special merge conditions.

We still use the S[i] to talk about a finite prefix of a stream, but now it refers to the first i slices. So, for example, if S = {| [1, 2], [3], [ ], [4, 5], [6], ... |}, then S[4] = [1, 2, 3, 4, 5].

A second change is not to provide the entire stream prefix to the operator at each iteration. Some operations, such as select, don't need anything but the most recent input element to figure out their next output. Other operators only need a summary of the prefix. For example, duplicate elimination only needs the distinct values in the prefix. Thus, in our actual formulation, operators 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 are certain table operators with stream analogues, but where the stream iterator has to keep track of arbitrary amounts of the prefix. For example, the state in duplicate elimination grows without bound.

Keeping state doesn't change the set of stream functions that can be expressed. Any function in the prefix version can be converted to the stateful version by keeping prior input in the state. A function in the stateful version can be adapted to the prefix form by computing its state from the prefix on each iteration. (Obviously, a function might run with different efficiency in the different versions.)

5 Theory of Punctuations

We now explain how we augment our streams with punctuations, and discuss the actions a stream iterator can take upon receipt of a punctuation. First, a normally blocking stream iterator can pass on results early. Second, a stream iterator can purge state early. Third, a stream iterator should propagate punctuation to its output. From these actions, we propose three classes of rules that formally define how a stream iterator should process punctuation: *pass rules*, *purge rules*, and *propagation rules*. We will define these rules below in terms of the allowable action on any given prefix of the input streams. That is, they consider S_j[i] for each input stream S_j. In this discussion, when we refer to "join" we mean non-blocking join implementations such as the symmetric hash join mentioned earlier.

5.1 Grammatical Data Streams

We assume that punctuations found in data streams do not lie. That is, there are no tuples following a punctuation that match that punctuation, for all punctuations in the stream. We call such streams *grammatical*. Formally, a stream S is grammatical if for all i, for all j > i, for all punctuations p, and for all tuples t, p $\in$ S[i] $\wedge$ t $\in$ S[i$\rightarrow$j] $\Rightarrow$ nomatch(t,p). Enforcing grammatical streams is a requirement, both for the stream sources and the stream iterators that implement these rules.

5.2 Pass Rules

Pass rules define when a stream iterator can correctly output data. One pass rule is defined for each stream iterator, and has the form: tsOut = pass(ts_1, ps_1, ..., ts_n, ps_n), where tsOut represents the tuples that can be output early, ts_j are tuples that have arrived from the j^{th} input, and ps_j are punctuations that have arrived from the j^{th} input. The trivial pass rule, passT, returns the empty set. Table 5 lists some pass rules.

Two of the pass rules use functions that require further explanation. The pass rule for group-by uses a function called groupPs. This function returns punctuations based on the input punctuations that have wildcard values for all non-grouping attributes. With these punctuations, we can determine if all the tuples for particular groups have been received, and output such groups early. In the warehouse example, if the group-by iterator receives a punctuation that says that all tuples for the first, second, and third hour have arrived, then we can pass on results for those groups. However, if group-by iterator receives a punctuation that says that all tuples have arrived for sensor S1, nothing can be done.

The pass rule for sort requires a function called init. In order to output correct results early for sort, we need to know when the values that sort first 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 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 punctuation that reads <*, [2,3], *, *> (the schema from the sensors is <sensorid, hour, minute, temp¿), then we cannot output anything. If, however, we receive punctuation that reads <*, [0,3], *, *>, then we can output results through the third hour. We know by the attribute's type that there are no negative values, so 0 is the first value for the sorted output. The init function, given the two punctuations shown above, will return <*, [0,3], *, *>.

5.3 Purge Rules

Purge rules define when a stream iterator can correctly purge state. Purge rules are applied in addition to any purging that a stream iterator might normally do. Unlike pass rules, a purge rule is defined on each input of an iterator. This is because the

Op	Pass Rule
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)}

Table 5: Pass rules for traditional database operators. Operators with trivial rules are omitted.

Op		Purge Rule
dupelim	$purge_1$=	{t|t∈ts_1 ∧ setMatch t ps_1}
join	$purge_1$=	{t|t∈ts_1 ∧ setMatch t (joinPs ps_2)}
	$purge_2$=	{t|t∈ts_2 ∧ setMatch t (joinPs ps_1)}
intersect	$purge_1$=	{t|t∈ts_1 ∧ setMatch t ps_2}
	$purge_2$=	{t|t∈ts_2 ∧ setMatch t ps_1}
difference	$purge_1$=	{t|t∈ts_1 ∧ t ∉ ts_2 ∧ setMatch t ps_2}
	$purge_2$=	{t|t∈ts_2 ∧ setMatch t ps_1}
group-by	$purge_1$=	{t|t∈ts_1 ∧ setMatch t (groupPs ps_1)}
sort	$purge_1$=	{t|t∈ts_1 ∧ setMatch t (init $Sort_A$ ps_1)}

Table 6: Purge rules for traditional database operators. Operators with trivial rules are omitted.

implementations for iterators like symmetric hash join and difference maintain state for each of their inputs. They have the form: `tsOut = purge`$_i$`(ts`$_1$`, ps`$_1$`, ..., ts`$_n$`, ps`$_n$`)`, where `purge`$_j$ returns the tuples from the j^{th} input to purge, and `ts`$_j$ and `ps`$_j$ are as defined before. The trivial purge rule, `purgeT`, returns the empty set. Table 6 lists some purge rules. Notice that the purge rules for group-by and sort are the same as their pass rules. Purge rules assume all tuples specified by the corresponding pass rule have already been output.

The purge rule for join uses a function that requires explanation called `joinPs`. This is similar to the `groupPs` function. It returns punctuations that have wildcard values for all attributes not participating in the join. The purge rule for join therefore says that tuples can be discarded for one input if they match punctuations from the other input that restrict only the join attribute(s).

5.4 Propagation Rules

Propagation rules define when a stream iterator can correctly output punctuations. One propagation rule is defined for each iterator, and has the form: `psOut = prop(ts`$_1$`, ps`$_1$`, ..., ts`$_n$`, ps`$_n$`)`,

Op	Propagation Rule
select	ps_1
project	projPs ps_1
dupelim	ps_1
join	(modPs$_1$ setNomatchPs ts_1 ps_1) ∪ (modPs$_2$ setNomatchPs ts_2 ps_2)
merge	{setCombine ps_1 ps_2}
intersect	{setCombine ps_1 ps_2}
difference	{setCombine ps_1 ps_2}
group-by	groupPs ps_1
sort	{init $Sort_A$ ps_1}

Table 7: Propagation rules for traditional database operators.

where `psOut` represents the punctuations that can be output, and `ts`$_j$ and `ps`$_j$ are as before. The trivial propagation rule, `propT`, returns the empty set. Propagation rules assume that all tuples specified by the corresponding pass rule have already been output. Table 7 lists some propagation rules.

The propagation rules for project and join require some explanation. The rule for project uses the function `projPs` to determine what punctuations to output. Punctuations that are output must have the same structure as the output tuples. If the stream iterator is projecting out attributes, then the same attributes must be projected out of the punctuations. However, we cannot modify and output the punctuation blindly. If an attribute being projected out contains a pattern that is not the wildcard, then the punctuation cannot be output.

The rule for join uses the functions `modPs`$_1$ and `modPs`$_2$. They have similar functionality, so we will focus only on `modPs`$_1$. Like the project iterator, the structure of output tuples from a join are different from its input tuples. Therefore, we need to modify the structure of the output punctuations to match the output tuples, which `modPs`$_1$ does. It simply includes the attributes of punctuations from the other input, and gives them wildcard values.

5.5 Faithfulness and Propriety

We have presented our approach for stream iterators, and the use of punctuation to help pass output early and purge state. There is, however, a significant issue we have not yet addressed, namely: When is a stream iterator a “reasonable” counterpart of the corresponding table operator? We hope the examples presented thus far seem sensible, but clearly one can define stream versions of operators with obviously inappropriate behavior. For example, a “stream sort” iterator that simply appends the sorted elements of each stream slice as it arrives does not seem an appropriate analog to the standard sort operator on finite inputs. 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 output of our iterator at any point of the input to be consonant with any possible further input it might see. Recall that we adapt the `S[i]` notation from Section 4 to the sliced representation of streams by letting `S[i]` be the concatenation of the first `i` lists in `S`. For a stream iterator `f`, we define the i^{th} *partial* of `f` as

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

$\hat{f}$`(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. (The `2i` index in the second definition is because we assume `f` will emit a slice of output for both the $\texttt{i}^{th}$ slice of `R` and the $\texttt{i}^{th}$ slice `S`.) For a unary function `f` with type `Stream(T)` $\rightarrow$ `Stream(U)`, the function $\hat{f}$ has the signature `Stream(T)` $\rightarrow$ `Int` $\rightarrow$ `List(U)`.

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 table operator from `List(T)` to `List(U)`. We say `f` is *faithful* to `g` if the following two conditions are met:

1. (Safety) For all streams `S` in `Stream(T)`, for all `i`, and for every list `A` in `List(T)`, $\hat{f}$`(S, i)` $\subseteq$ `g(S[i]++A)`. [We interpret $\subseteq$ as prefix or subset depending on whether order is significant for `g`.]

2. (Completeness) For all streams `S` in `Stream(T)` and for all `i`, if `e` $\in$ `g(S[i]++A)` for every `A` in `List(T)`, then `e` $\in$ $\hat{f}$`(S, i)`.

The safety condition says we never emit output unless we are sure that it will not conflict with later input. 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: $\hat{f}$`(S1, S2, i)` $\subseteq$ `g(S1[i]++A1, S2[i]++A2)` for all `A1` and `A2`, and `e` $\in$ $\hat{f}$`(S1, S2, i)` if `e` is always in `g(S1[i]++A1, S2[i]++A2)`.

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 `A`) is constrained to obey any punctuation seen already, and to account for the mixture of tuples and punctuations in the input.

1. (Safety) For all streams `S` in `Stream(T)`, for all `i`, and for every list `A` in `List(T)` such that `setMatchTs(A, puncts(S[i]))` is empty, `tups(`$\hat{f}$`(S, i))` $\subseteq$ `g(tups(S[i])++A)`.

2. (Completeness) For all streams `S` in `Stream(T)`, for all `i`, if `t` $\in$ `g(tups(S[i])++A)` for every `A` in `List(T)` such that `setMatch(A, puncts(S[i]))` is empty, then `t` $\in$ `tups(`$\hat{f}$`(S, i))`.

The definition for binary operators is extended in a similar way.

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.

We have been developing stream analogues of a range of table operators, and have been proving properties about them. Our general strategy for such proofs is to show that a stream operator that obeys the pass, propagation, and purge rules is faithful and proper, then show that the **pass**, **prop** and **keep** functions used in our stream iterator framework (presented in the next section) conform to the rules.

6 Our Stream Processing Framework

As we remarked before, our initial extensions of query operators to streams punctuations were ad hoc. We have taken a step back, and tried to abstract the common portion of stream processing operators. The result is a common control structure for stream iterators, which is customized by plugging in particular auxiliary functions for each operator. We undertook this part of the work in Haskell, a *lazy evaluation* functional language. We chose Haskell for three reasons: First, lazy evaluation languages do not evaluate expressions until required. Thus, they are very suitable for modelling systems that process continuous data streams, since tuples in a stream are only evaluated when required. Second, since it is a functional language, it allowed us to formulate the general behavior of all operators in a single function, and pass the specifics of each operator as function arguments to the general function. Finally, since it is a language with formal semantics, it helps us to prove our implementations satisfy the appropriate pass, propagation and purge rules. Lazy evaluation languages have been used by other research projects [8, 12, 7] to model data stream processing as well.

6.1 A Formulation of Stream Iterators

Without punctuations, stream iterators resemble Parker's stream transducers [11]. A unary stream iterator is defined by its initial state and two functions, (`initial_state, step, final`), where:

- `initial_state` is the state of the iterator before tuples arrive from the input stream.
- `step` is called when new data arrives from the input stream. It takes the new tuples and the 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 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,newstate) = step xs newstate
   in out++(unary newstate step final rest)
```

The definition of the duplicate elimination stream iterator is:

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

The state is a list of all unique values from the input. The `step` function returns any tuples that do not exist in state, and a modified state containing the new tuples concatenated with the tuples from the original state. The `final` function clears the state. We use two functions from the Haskell `prelude` library: `nub` removes duplicates from a list, and `\\` takes two lists and returns all values in the first list that are not in the second.

A binary stream iterator is defined by its initial state and four functions: (`initial_state, stepL, stepR, finalL, finalR`). The meanings of `stepL` and `stepR` are the same as the `step` function for a unary operator, but are defined here for each of the binary iterator's inputs. Likewise, `finalL` and `finalR` are the same as `final` for a unary operator, but defined for each input. The general operation of a binary stream iterator reads from each input. When new data arrives from one of the inputs, the appropriate `step` function is called. When one of the streams ends, the appropriate `final` is called, and then the other input stream is read until completion, and then its `final` is called. The formal definition of `binary` is similar to `unary`, and is therefore omitted.

6.2 Extension to Punctuated Streams

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

- `initial_state` has the same meaning as before.
- `step` has 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.
- `keep` takes new punctuations and the state, and returns a new state based on the punctuations.

The `pass` and `prop` functions are a bit different than the pass and propagation rules. They operator on input incrementally rather than cumulatively. The `keep` function is kind of a compliment to the purge rule. It returns state to hang onto, rather than state to discard.

As before, binary iterators implement each function for each of their inputs. Note that `final` has been removed. The `final` function was called when the input stream ended. It passed on any new tuples, purged out its state, and informed the next operator that the input had ended. We can see that `final` performs the same task as `pass`, `prop`, and `keep`.

The changes to `unary` and `binary` for punctuated streams are similar, so we focus on `unary`. New input may contain both tuples and punctuation, so they are first separated using a function called `splitPunc`. Then `step` is called with the new tuples and the current state, followed by calls to `pass`, `prop`, and `keep` with the new punctuations. 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`. The `keep` function may purge state related to tuples that are output by `pass`, or punctuations that are output by `prop`, so `keep` should follow the other two functions. The new definition of `unary` is:

```
unary state step pass prop keep [] = []
unary state step pass prop keep (xs:rest) =
  let (ts,ps)       = splitPunc xs ([],[])
      (tsOut,stNew) = step ts state
      tsExtra       = pass ps stNew
      psOut         = prop ps stNew
      stNew'        = keep ps stNew
  in [tsOut ++ tsExtra ++ psOut] ++
     (unary stNew' step pass prop keep rest)
```

There are operators that do not need to implement one or more 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 according to the trivial rules:

```
passT ps st = []
propT ps st = []
keepT ps st = st
```

In the presence of punctuated streams, the stream iterator for duplicate elimination is redefined as follows:

```
sdupelim xs =
    unary [] destep passT deprop dekeep xs
  where destep ts tsSeen =
          let tsOut = (nub ts \\ tsSeen)
          in (tsOut, tsSeen ++ tsOut)
        deprop ps _ = ps
        dekeep ps tsSeen =
          setNomatchTs tsSeen ps
```

The step function remains unchanged. Since the duplicate elimination iterator is not blocking, it uses the trivial pass function. The propagation function `deprop` simply outputs punctuations as they arrive. The real enhancement is in the keep function `dekeep`. When a punctuation arrives, we know that there will be no more tuples that match the punctuation. Therefore, tuples in state that match the punctuation can be purged.

7 Related Work

Data stream processing has been researched for a number of years. The Tangram stream query processing system presented by Parker et al. [12] is an early attempt at using database technology on data streams. They discuss a number of stream-transducer implementations but do not discuss how they behave in the presence of continuous streams. Parker also discusses a model for stream transducers [11] that is very similar to ours, though he limits the discussion to unary stream transducers.

Recently, data stream processing has attracted new interest in the database community. Babu et al. [2] present an architecture for queries over data streams. A query has one or more input streams, an output stream, a *store* for holding tuples that might be part of the result, and a *scratch* area for state. Punctuations enhance this architecture in two ways: First, implementing pass rules decreases the size of store by moving tuples to the output stream sooner. Second, implementing purge rules decreases the size of scratch. Madden and Franklin [9] present an architecture for queries over static sources as well as streams. Operators accept data from push-based sources as well as pull-based.

The Tribeca system [15] 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. [5] use similar structures to compute correlated aggregates over data streams.

Work on partial results by Shanmugasundaram et al. [14] specifically addresses blocking operators. In this system, query operators that are waiting for output from other, perhaps blocking, operators, can request a partial result. This approach 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.

Arasu et al. [1] 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 instances of data streams. 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, intersect, set difference, and grouping. Note that punctuations might be useful to expand the class of queries that can be evaluated in bounded memory.

Querying ordered sequence data is also related [13]. 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.

8 Conclusions and Future Work

We have presented a novel way to execute queries on continuous data streams. Our method handles many traditional relational 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 discuss strategies for proving that our rules meet those properties for the difference operator.

Note that we do not guarantee that given punctuation can always unblock an operator or bound the state. We continue to look at more expressive punctuation, other stream semantics, and "stream friendly" variants of query operators. We also need to expand our formal model and implement more of our rules. This includes modelling more operators in our Haskell code base, and proving properties about those operators. It also includes carrying over those techniques to the Niagara code base and handling nested (XML) as well as flat (relational) data.

There are many other directions to pursue in punctuated streams. One direction is to determine what punctuations to embed into a data stream, and how they get embedded. Certainly, different punctuations will be better for different queries. Further, different granularities of the same punctuation will be better for different queries. Stream sources could be enhanced to "advertise" what sorts of punctuations they can support. Further, queries could provide "hints" to stream sources as to what punctuations would help them the most.

There is common work here with partial evaluation of queries [14]. Both address applying blocking operators to long or continuous inputs. Punctuation might be used to help partial operators to communicate output that is "partial" versus output that is "correct".

9 Acknowledgements

The authors would like to thank Kristin Tufte and Vassilis Papadimos for the discussions we had during the development of this work, Levent Erkok for his help in developing the Haskell framework, Raghu Ramakrishnan for 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 IIS-0086002, and by NSF grant IIS-9811525.

References

[1] A. Arasu, B. Babcock, S. Babu, J. McAlister, and J. Widom. Characterizing memory requirements for queries over continuous data streams. Technical report, Stanford University, Nov. 2001. URL:http://dbpubs.stanford.edu/pub/2001-49.

[2] S. Babu and J. Widom. Continuous queries over data streams. *SIGMOD Record*, 30(3), Sept. 2001.

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

[4] C. Cortes, K. Fisher, D. Pregibon, A. Rogers, and F. Smith. Hancock: A language for extracting signatures from data streams. In *Proceedings of the Sizth International Conference on Knowledge Discovery and Data Mining*, pages 9–17, Aug. 2000.

[5] J. Gehrke, F. Korn, and D. 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.

[6] A. C. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. 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, Sept. 2001.

[7] T. Hallgren and M. Carlsson. *Fudgets*. PhD thesis, Chalmers University of Technology, Mar. 1998.

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

[9] S. Madden and M. J. Franklin. Fjording the stream: An architecture for queries over streaming sensor data. In *Proceedings of the 18th International Conference on Data Engineering (to appear)*, Feb. 2002.

[10] 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. Technical report, University of Wisconsin, Mar. 2000.

[11] D. S. Parker. Stream data analysis in prolog. In L. Sterling, editor, *The Practice of Prolog*. MIT Press, 1990.

[12] D. S. Parker, R. R. Muntz, and L. Chau. The tangram stream query processing system. In *Proceedings of the Fifth International Conference on Data Engineering*, Feb. 1989.

[13] P. Seshadri, M. Livny, and R. Ramakrishnan. Sequence query processing. In *Proceedings of the ACM Special Interest Group on Management of Data*, pages 430–441, May 1994.

[14] J. Shanmugasundaram, K. Tufte, D. J. DeWitt, J. Naughton, and D. Maier. Architecting a network query engine for producing partial results. In *WebDB (Informal Proceedings)*, pages 17–22, May 2000.

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