Using Punctuation Schemes to Characterize Strategies for Querying over Data Streams

Peter A. Tucker, *Member*, *IEEE*, David Maier, *Senior Member*, *IEEE*,
Tim Sheard, *Member*, *IEEE*, and Paul Stephens

Abstract—Many systems and strategies have been proposed for processing nonterminating data streams. Each approach has advantages and disadvantages, including the kinds of queries that can be executed. We present a framework for characterizing the kinds of queries that can be executed over streams based on a notion of compact sets from topology. We first apply our framework to queries over punctuated data streams. Previous work on punctuations focused primarily on the behavior of individual query operators. We use our framework to determine if an entire query can benefit from punctuations available from stream sources. We then consider other common strategies proposed in the literature for executing queries over streams, and we discuss how our framework can characterize the kinds of queries each strategy can answer.

Index Terms—Data streams, query execution, punctuation.

1 Introduction

Two kinds of traditional query operators pose problems for nonterminating data streams: *blocking operators*, which produce output only after the input has ended, and *stateful operators*, which maintain an unbounded amount of state. A number of strategies have been proposed to allow the use of blocking and stateful operators over nonterminating streams, including querying over ordered data, applying windows over the input, and embedding punctuations. Each strategy has advantages, but little research has been done to help formally decide which strategy can be used to execute a particular query. In this work, we introduce the concept of *punctuation schemes* and use punctuation schemes to determine if a given query can be executed over nonterminating inputs. Further, we use punctuation schemes in discussing the kinds of queries other strategies can address.

1.1 Motivating Example

A number of commercial and research systems process and monitor online auctions, including eBay [1], Yahoo! Auctions [2], the Fishmarket [3], and the Michigan Internet AuctionBot [4]. Software agents may represent humans in the auction to bid on or sell items. A user registers through an agent, then participates in auctions as a buyer or seller. We model an online auction with three kinds of stream sources that supply data to an auction monitoring system, as shown in Fig. 1. Bids for an item currently for sale arrive on one of several `Bid` streams, new items for sale arrive on the `Auction` stream, and newly registered users arrive on the `Person` stream.

For simplicity, assume two bid streams (`Bid1` and `Bid2`) with schema: `BidN(auctionid, bidderid, value, hour, minute)`. An auction administrator will want a query that counts the bids placed each hour where the bid price is considered high (say, greater than $250). The SQL for such a query is

Query 1. Count high price bids

```
SELECT hour, COUNT(*)
FROM (SELECT * FROM Bid1
      UNION
      SELECT * FROM Bid2)
WHERE value > 250
GROUP BY hour;
```

However, as group-by is a blocking operator, this query cannot be executed over nonterminating inputs without help.

Auction items arrive on a separate stream from bids. For concreteness, we will use the following schema: `Auction(id, sellerid, itemname, expires)`. A second important query reports the maximum bid submitted for each item. Such a query reports the sale price for each item when the auction for that item closes. The SQL for such a query is

Query 2. Closing price for items

```
SELECT A.id, A.itemname, B.close
FROM Auction A,
     (SELECT auctionid,
             MAX(value) AS close
      FROM (SELECT * FROM Bid1
            UNION
            SELECT * FROM Bid2)
      GROUP BY auctionid) B
WHERE A.id = B.auctionid;
```


- P.A. Tucker and P. Stephens are with the Department of Math and Computer Science, Whitworth University, 300 W. Hawthorne Road, Spokane, WA 99218. E-mail: {ptucker, pstephens07}@whitworth.edu.
- D. Maier and T. Sheard are with the Department of Computer Science, Portland State University, PO Box 751, Portland, OR 97207. E-mail: {maier, sheard}@cs.pdx.edu.

Manuscript received 4 Dec. 2006; revised 29 Mar. 2007; accepted 6 Apr. 2007; published online 24 Apr. 2007.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number TKDE-0545-1206.
Digital Object Identifier no. 10.1109/TKDE.2007.1052.

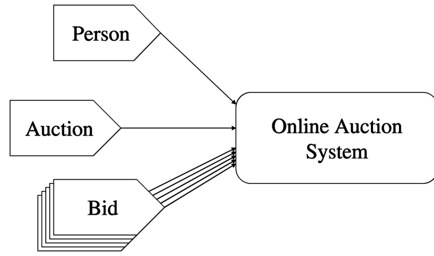

Fig. 1. Simple architecture for a system to monitor an online auction.

As in Query 1, this query has a group-by operator, which is a blocking operator. Additionally, this query has a join operator, which requires an unbounded amount of state. Both kinds of operators are problematic when dealing with nonterminating input, so this query also cannot be executed over nonterminating inputs without help.

1.2 Comparing Approaches to Querying over Data Streams

Several strategies have been proposed for executing queries over nonterminating data streams. These strategies include relying on data arriving in some order based on specific attribute values (typically in the form of a timestamp), defining windows over the input streams, using heartbeats, and embedding punctuations in the stream. We discuss these approaches further in Section 8.

The performance and expressiveness of data stream systems can be compared using benchmarks such as Linear Road [5] and NEXMark [6]. However, there has not been much investigation in formally comparing different data stream strategies. In this work, we will develop a framework inspired by compact sets, then use our framework as a method for comparing strategies based on the kinds of queries that each can handle.

1.3 Organization

The rest of this paper is organized as follows: In Section 2, we give a brief overview of punctuation semantics. We present definitions of groupings and punctuation schemes in Section 3. In Section 4, we discuss how to determine if punctuation schemes unblock query operators and, in Section 5, we discuss how to determine if punctuation schemes reduce operator state. In Section 6, we show how to determine if all of the queries benefit from punctuation schemes. We introduce a new punctuation-specific query operator in Section 7. In Section 8, we use punctuation schemes to compare various data stream approaches. We review related work in Section 9 and conclude in Section 10.

2 Brief Overview of Punctuated Data Streams

A punctuation is an item embedded into a stream that describes a subset of the domain of that stream. We say that a data item d *matches* a punctuation p (denoted $match(d,p)$) if d belongs to the subset described by p. A punctuation p embedded in a data stream states that no data items will arrive subsequently that match p. A stream that adheres to this property is called *grammatical*. In this work, we consider only streams that are grammatical.

TABLE 1
Example Punctuation Invariants for Various Operators

Helper Functions	
$setMatch(d, ps) =$	$true$ if $\exists p \in ps$ such that $match(d,p)$ $false$ otherwise
$setNomatch(d, ps) =$	$true$ if $\forall p \in ps, match(d,p) = false$ $false$ otherwise
$setMatchDs(ds, ps) =$	$\{d \mid d \in ds \wedge setMatch(d, ps)\}$
$setNomatchDs(ds, ps) =$	$\{d \mid d \in ds \wedge setNomatch(d, ps)\}$

Op	**Pass Invariant**
trivial	$cpass(ds, ps) = \emptyset$
group-by ($\mathcal{G}_A^F$)	$cpass(ds, ps) = \{d :: <f_i(U_d)> \mid d \in setMatchDs(\pi_A(ds), grpP(A, ps)) \wedge f_i \in F \wedge U_d = \{u \mid u \in ds \wedge d[A] = u[A]\}\}$

Op	**Keep Invariant**
trivial	$ckeep(ds, ps) = ds$
dupelim (δ)	$ckeep(ds, ps) = setNomatchDs(ds, ps)$
group-by ($\mathcal{G}_A^F$)	$ckeep(ds, ps) = \{d \mid d \in ds \wedge setNomatch(\pi_A(d), grpP(A, ps))\}$

Op	**Propagation Invariant**
trivial	$cprop(ds, ps) = \emptyset$
select (σ_q)	$cprop(ds, ps) = ps$
group-by ($\mathcal{G}_A^F$)	$cprop(ds, ps) = \{p :: <w_i> \mid p \in grpP(A, ps) \wedge \forall f_i \in F, w_i = \text{'*'}\}$

Here, ds represents data items, and ps represents punctuations that have arrived. The function $grpP$ outputs punctuations in the output schema that match an entire group when enough input punctuations have arrived to guarantee that all data items for that group have arrived. The wildcard pattern ("$*$") matches all values for that attribute.

By embedding punctuations into a stream, query operators know about the ends of particular subsets of data. A blocking operator might use this information to output results for a completed subset. A stateful operator might purge state for such a subset. A query operator that is enhanced to exploit punctuations implements three *punctuation behaviors*: First, *pass behavior* defines what data items can be output when punctuations arrive. Second, *keep behavior* defines the state an operator must retain to continue outputting correct results. Finally, *propagation behavior* defines what punctuations may be emitted from the operator to operators further up the query tree. For each behavior, we have defined corresponding *punctuation invariants* for many common query operators to formally define how punctuations should be handled and have shown that operators which adhere to the invariants behave correctly [7].

Punctuation invariants are cumulative. They consider a prefix of the input stream (or streams). *Pass invariants* return the data items that can be output beyond those that would have been output during normal execution. Operators that do not block on their input(s) can use the trivial pass invariant, which outputs the empty set. *Propagation invariants* return the punctuations that can be emitted from the operator such that the operator's output remains grammatical. *Keep invariants* return the data items that must be retained in operator state beyond what is kept during normal operator execution. Examples of the three kinds of invariants for specific operators are given in Table 1.

Fig. 2 shows a possible query plan for Query 1. Suppose the `Bid` sources emit a punctuation after the last value for each `hour`. Consider punctuation p that signals that more data items will arrive with `hour` value 6 from `Bid1`. When p arrives at the union operator, it must wait for a

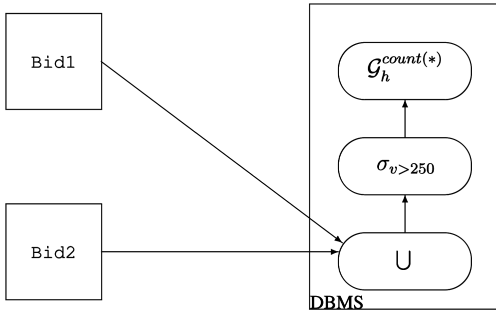

Fig. 2. Query plan for Query 1.

punctuation with the same hour from Bid2. After p is seen on both streams, the punctuation can be output (propagation behavior). If union is eliminating duplicates, then all data items that match p can be removed from the state (keep behavior).

When the group-by operator receives p, it knows that no more data items will arrive that contribute to the group for hour value 6. It can output results for that group (pass behavior), then reduce the state required for that group (keep behavior). Finally, it can also output punctuation for hour value 6 (propagation behavior).

To this point, our work on punctuations has focused on defining correct behavior for individual query operators under punctuations. Indeed, punctuations can fix the problems with Queries 1 and 2. For Query 1, punctuations embedded into the bid streams that mark the end of each hour could be used to unblock the group-by operator since it is grouping on the hour attribute. For Query 2, punctuations that mark the end of bidding for each auction can be used both to reduce the amount of state required by the join operator (since the join attributes are id and auctionid) and unblock group-by (since the group-by attribute is auctionid).

We used punctuations in previous work to produce results and reduce state [8] but have avoided an important question, namely, "how do we determine what kinds of punctuation will help a given query?" Punctuations are not always beneficial. Indeed, there are queries that cannot be improved with any kind of punctuation. Query 1 groups data items on the hour attribute and, so, has a natural *grouping of interest* based on hour. (Grouping of interest is somewhat akin to the idea of an "interesting order" [9].) A grouping is a collection of groups in the data domain based on values of the grouping attribute(s). Many operators have natural groupings of interest and these help decide if a particular set of punctuations benefits a query. Section 3 provides details on groupings of interest.

Punctuations that collectively match all possible data items in a grouping of interest may benefit a query, but only if the number of punctuations needed to cover each specific group is finite and will eventually arrive on the stream.[1] (The total number of punctuations in a stream need not be finite.) To illustrate, suppose that each bid stream contains punctuations marking the end of bids from a specific user for a particular hour. Fig. 3 depicts the data items that match each punctuation. Using this set of punctuations, we

1. Note that, in practice, we do not need such a strong statement. It is sufficient to assume that every punctuation will eventually appear or that a punctuation will arrive that subsumes it.

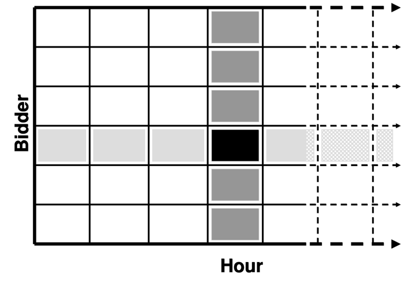

Fig. 3. Data items that match punctuations for a specific bidder and hour. Each square in the grid represents all bids from a bidder that have arrived during a specific hour. The darker area, containing all bids for a particular hour, can be covered by a finite number of punctuations. The lighter area, containing all bids from a particular bidder, cannot.

can match the group of all bids for a given hour using a finite number of punctuations. Therefore, such punctuations benefit queries that group solely on hour. However, we cannot realistically match all bids from a given bidder with a finite number of punctuations. Therefore, queries that group on bidderid will not benefit from this set of punctuations. The need to cover a group with a finite number of punctuations suggests a notion similar to *compact sets* [10], [11] from topology. We will use analogs to compactness to formally determine if a specific set of punctuations will benefit a particular query.

3 Groupings and Punctuation Schemes

We want to determine the utility of a given set of punctuations for a particular query. To this end, we introduce several concepts. A *dataspace* represents the domain of all possible data items that may appear on a stream. For example, the dataspace for the Bid streams is

$$D_B = \{< a, b, v, h, m > | a \in \mathbb{Z}, b \in \mathbb{Z}, v \in \mathbb{R}, \\ h \in \mathbb{Z}, m \in [0, 59]\}.$$

The term *subspace* means any subset of a dataspace.

3.1 Groupings for Dataspaces and Groupings of Interest

A *grouping* for a dataspace or subspace is a collection of groups where each group contains items that have equal values for specified attributes and the union of the groups equals the dataspace. For example, the grouping of D_B on h is

$$\big\{\{< a, b, v, h, m > | a \in \mathbb{Z}, b \in \mathbb{Z}, v \in \mathbb{R}, m \in [0, 59]\} | h \in \mathbb{Z}\big\}.$$

For the subspace

$$S = \{< 1, 1, 50, 1, 15 >, < 1, 2, 70, 1, 45 >, \\ < 1, 3, 75, 2, 10 >\},$$

the grouping on h is (with h value in bold)

$$\big\{\{< 1, 1, 50, \mathbf{1}, 15 >, < 1, 2, 70, \mathbf{1}, 45 >\}, \\ \{< 1, 3, 75, \mathbf{2}, 10 >\}\big\}.$$

A *grouping of interest* for a query operator is a grouping that arises naturally from the definition or implementation of that operator. Many operators have groupings of interest.

For example, a group-by's grouping of interest is the one defined by the group by attributes. Join has two groupings of interest, one for each input, based on the join attributes. We discuss groupings of interest for specific operators in Section 5.

3.2 Punctuation Format and Punctuation Schemes

For a given dataspace, a punctuation is a tuple of patterns, one pattern per attribute of the dataspace. There are many options for patterns to construct punctuations. The patterns we present have the property of closure under intersection (and, therefore, punctuations are closed under conjunction). This property is important for some punctuation behaviors such as the propagation behavior for union.

The set of valid patterns over a domain A includes two special patterns, $*$ and ϵ, where $\{*, \epsilon\} \cap A = \emptyset$. The set of patterns then is defined as

$$\Pi_A = \{*\} \cup \{\epsilon\} \cup \{A' | A' \subseteq A \ \wedge A' \ is \ finite\}.$$

The $matchPat$ function takes an element a from A and a pattern p from Π_A and returns $true$ if a matches p:

$$\begin{aligned}
&matchPat :: A \times \Pi_A \rightarrow Boolean,\\
&matchPat(a, *) = true,\\
&matchPat(a, \epsilon) = false,\\
&matchPat(a, a_1) = a == a_1,\\
&matchPat(a, A') = true \text{ if } \exists a' \in A' | matchPat(a, a')\\
&\qquad\qquad false \text{ otherwise.}
\end{aligned}$$

If A is totally ordered by $\leq$, we can supplement these patterns to include range matching. We add $\perp$ and $\top$, where $\{\perp, \top\} \cap A = \emptyset \wedge \forall a \in A, \perp < a < \top$. We extend Π_A with

$$\begin{aligned}
&\{[a_1, a_2] | a_1, a_2 \in A \wedge a_1 \leq a_2\} \cup\\
&\{[\perp, a_2] | a_2 \in A\} \cup\\
&\{[a_1, \top] | a_1 \in A\}
\end{aligned}$$

and extend $matchPat$

$$\begin{aligned}
&matchPat(a, [a_1, a_2]) = a_1 \leq a \leq a_2,\\
&matchPat(a, [\perp, a_2]) = a \leq a_2,\\
&matchPat(a, [a_1, \top]) = a_1 \leq a.
\end{aligned}$$

Two points are worth mentioning. First, note that we do not need the case $[\perp, \top]$ as that would be the same as the * pattern. Second, we only specify closed intervals above. It is a simple extension to specify open and mixed intervals as well.

For dataspace D (over $A_1 \times A_2 \times \ldots \times A_n$), the *punctuation space* $\mathbb{P}_D = \Pi_{A_1} \times \Pi_{A_2} \times \ldots \times \Pi_{A_n}$ is the set of possible punctuations over D. The $match$ function determines when an item from D matches punctuation from $\mathbb{P}_D$ by comparing values and patterns from corresponding attributes. For D,

$$\begin{aligned}
&match :: D \times \mathbb{P}_D \rightarrow boolean,\\
&match(d, p) = \forall i \in [1, n], matchPat(d(i), p(i)),
\end{aligned}$$

where $d(i) \in A_i$ is the value of the ith attribute of d and $p(i) \in \Pi_{A_i}$ is the pattern value of the ith attribute of p. Consider a punctuation $p = < *, *, [5.00, 8.00], 10, * >$ for D_B. Given data items $d_1 = < 1, 2, 6.00, 10, 20 >$ and $d_2 = < 1, 2, 7.00, 11, 20 >$, it is clear that $match(d_1, p) = true$ and $match(d_2, p) = false$.

A *punctuation scheme* $\mathcal{P}_D$ for D is the set of punctuations that will be emitted from a stream source.[2] Clearly, $\mathcal{P}_D \subseteq \mathbb{P}_D$. We specify individual punctuation schemes using set notation. Suppose each bid stream source outputs punctuations at the end of each hour. Then, the punctuation scheme is $\mathcal{P}_B^h = \{< *, *, *, h, * > | h \in \mathbb{Z}\}$. Alternatively, if a bid source outputs a punctuation marking the end of each 10-minute interval for each hour, the punctuation scheme is

$$\begin{aligned}
\mathcal{P}_B^{h,m} = &\{< *, *, *, h, [m, m+9] > | h \in \mathbb{Z}\\
&\wedge m \in \{0, 10, 20, 30, 40, 50\}\}.
\end{aligned}$$

A punctuation scheme $\mathcal{P}_D$ is *complete* if, for every $d \in D$, there exists $p \in \mathcal{P}_D$ such that $match(d, p)$. Generally, if a punctuation scheme is not complete, then blocking operators cannot be completely unblocked. Query 1 groups on values of the `hour` attribute. The punctuation scheme $\mathcal{P}_B^{2h} = \{< *, *, *, 2h, * > | h \in \mathbb{Z}\}$ emits punctuations that match data items for every even hour and, so, is not complete. Query 1 will not be completely unblocked. The punctuation scheme $\mathcal{P}_B^h = \{< *, *, *, h, * > | h \in \mathbb{Z}\}$ given earlier is complete and does completely unblock Query 1.

We want to know when all items of a group in a grouping have arrived. Punctuations provide this information. The *interpretation* of a punctuation p is the subspace of data items that match p, denoted $\mathcal{I}(p)$. Formally, given dataspace D and punctuation $p \in \mathbb{P}_D$, $\mathcal{I}(p) = \{d | d \in D \wedge match(d, p)\}$. For punctuation scheme $\mathcal{P}_D$, $\mathbf{S}_{\mathcal{P}_D} = \{\mathcal{I}(p) | p \in \mathcal{P}_D\}$ is the collection of interpretations for punctuations in $\mathcal{P}_D$. When D is understood, we will write $\mathcal{P}$ for $\mathcal{P}_D$ and $\mathbf{S}_{\mathcal{P}}$ for $\mathbf{S}_{\mathcal{P}_D}$.

3.3 Benefitting Queries with Punctuation Schemes

A punctuation scheme *benefits* a query if the following conditions hold:

- *Enables*. All result data items for a query will eventually be output.
- *Cleanses*. Every data item that resides in the state for any operator in the query will eventually be removed.

Note that, even if a punctuation scheme cleanses a query, there may still be data in its state at every point during execution.

Our goal is to determine if a punctuation scheme benefits an entire query. Given a query tree Q for a query and the punctuation schemes for each stream source, we can use propagation invariants to determine the output punctuation scheme for each operator in Q. If the root operator in the tree emits a complete punctuation scheme, we know that the query is enabled.

From topology, a collection $\mathbf{S}$ forms a *cover* for a set T if $\bigcup \mathbf{S} \supseteq T$. Further, T is *compact* if every cover for T in $\mathbf{S}$ contains a finite subcollection that is also a cover for T [10], [11]. Applying these concepts to punctuations, given a

2. Note that every punctuation in $\mathcal{P}_D$ will either appear eventually or be subsumed by some other punctuation in $\mathcal{P}_D$. Punctuations that will not appear on the data stream are not in $\mathcal{P}_D$.

TABLE 2
Definitions for $preimage$ for Various Operators for Punctuations $p \in \mathcal{P}_R[O]$

select	$preimage[\sigma](p) = \mathcal{I}(p)$
dupelim	$preimage[\delta](p) = \mathcal{I}(p)$
project	$preimage[\pi_A](p) = \mathcal{I}(p(A) : *)$
group-by	$preimage[\mathcal{G}_A^F](p) = \mathcal{I}(p(A) : *)$
union	$preimage[\cup](p) = (\mathcal{I}(p), \mathcal{I}(p))$
intersect	$preimage[\cap](p) = (\mathcal{I}(p), \mathcal{I}(p))$
difference	$preimage[-](p) = (\mathcal{I}(p), \mathcal{I}(p))$
equi-join	$preimage[\bowtie_B](p) =$ $(\mathcal{I}(p(A) : p(B)), \mathcal{I}(p(B) : p(C)))$ with input schemas $A \cup B$ and $B \cup C$

We use the notation " : $*$ " to mean $*$ values for each attribute of the schema not already listed.

group G in a grouping and a punctuation scheme $\mathcal{P}$, $\mathcal{P}$ is a *cover* for G if each data item in G matches some punctuation in $\mathcal{P}$. G is *compact* relative to $\mathcal{P}$ if some finite subset of $\mathcal{P}$ also forms a cover for G. If all groups of a grouping $\mathbf{G}$ are compact relative to $\mathcal{P}$, we say that $\mathbf{G}$ is compact relative to $\mathcal{P}$.

For example, consider groupings based on that in Fig. 3. The grouping

$$\mathbf{G}_h = \big\{\{< a, b, v, h, m > | a \in \mathbb{Z}, b \in \mathbb{Z}, v \in \mathbb{R}, m \in [0, 59]\} | h \in \mathbb{Z}\big\}$$

is compact relative to the punctuation scheme

$$\mathcal{P}_B^h = \{< *, *, *, h, * > | h \in \mathbb{Z}\}$$

since a finite number of punctuations (one in this case) from $\mathcal{P}_B^h$ will match each group in $\mathbf{G}_h$. However, the grouping $\mathbf{G}_b = \big\{\{< a, b, v, h, m > | a \in \mathbb{Z}, v \in \mathbb{R}, h \in \mathbb{Z}, m \in [0, 59]\} | b \in \mathbb{Z}\big\}$ is not compact relative to $\mathcal{P}_B^h$. Our basic strategy is to show that, if a grouping of interest for an operator is compact relative to its input punctuation scheme, then that punctuation scheme benefits the operator. Further, if the groupings of interest for all operators in a query are compact relative to their input punctuation schemes, then the entire query benefits.

4 Enabling Query Operators

We first consider the kinds of punctuation schemes that unblock unary operators. For a unary operator O, $\mathcal{P}_I[O]$ denotes the input punctuation scheme and $\mathcal{P}_R[O]$ the output (result) punctuation scheme. An input punctuation scheme $\mathcal{P}_I[O]$ *enables* an output punctuation $p \in \mathcal{P}_R[O]$ if p can be propagated after some finite subset of $\mathcal{P}_I[O]$ has arrived. Further, an input punctuation scheme $\mathcal{P}_I[O]$ *enables* $\mathcal{P}_R[O]$ if, for every $p \in \mathcal{P}_R[O]$, $\mathcal{P}_I[O]$ enables p. If $\mathcal{P}_I[O]$ enables $\mathcal{P}_R[O]$, then we know that all result data items contained in interpretations of $\mathcal{P}_R[O]$ will be output. If $\mathcal{P}_R[O]$ is complete, we know that O is enabled.

4.1 The Definition of $Preimage$

We need a map from a punctuation p in $\mathcal{P}_R[O]$ to a subspace T of the input dataspace D_I such that, if T is compact relative to $\mathcal{P}_I[O]$, then p will eventually be emitted. We call this map $preimage$, defined for specific operators in Table 2. (We use *dupelim* to refer to the operator that removes duplicates.) Intuitively, for any punctuation $p \in \mathcal{P}_R[O]$, $preimage[O](p)$ tells us what subspace must be covered by input punctuations before O can safely emit p. Put another way, $preimage[O](p)$ returns that subspace of the input that contributes to output data items that match p.

These are the definitions we will consider in this work. We can choose an alternate definition of $preimage$ for a given operator, such as $preimage[\sigma](p) = \sigma(\mathcal{I}(p))$, as long as Theorem 1 (given in the next section) holds.

4.2 Enabling Punctuations from Unary Operators

We want to determine if an input punctuation scheme $\mathcal{P}_I[O]$ enables an output punctuation scheme $\mathcal{P}_R[O]$ for a unary operator O. The following theorem specifies when a given punctuation in $\mathcal{P}_R[O]$ can be emitted by O. We will later use that result to show when $\mathcal{P}_R[O]$ is enabled by $\mathcal{P}_I[O]$.

Theorem 1. *For unary operator O, grammatical input stream S, input punctuation scheme $\mathcal{P}_I[O]$, and output punctuation scheme $\mathcal{P}_R[O]$, $p_r \in \mathcal{P}_R[O]$ can be emitted if $preimage[O](p_r)$ is compact relative to $\mathcal{P}_I[O]$.*

A detailed proof can be found elsewhere [7]. The proof strategy is to consider $P \subseteq \mathcal{P}_I[O]$ that is compact over $preimage[O](p_r)$, where all punctuations in P have arrived. As discussed in Section 2, the pass invariant for O formally defines what data items can be output due to punctuations. We use the pass invariant for O to show that all data items in $preimage[O](p_r)$ have been output and, therefore, p_r can be emitted.

For example, Query 1 uses the group-by operator $\mathcal{G}_h^{count(*)}$. A possible output punctuation scheme for $\mathcal{G}_h^{count(*)}$ could be $\mathcal{P}_O^h = \{< h, * > | h \in \mathbb{Z}\}$. An input punctuation scheme that is compact relative to the grouping on h, such as $\mathcal{P}_B^h = \{< *, *, *, h, * > | h \in \mathbb{Z}\}$, would enable $\mathcal{P}_O^h$ for $\mathcal{G}_h^{count(*)}$.

We use this result to show the required properties of $\mathcal{P}_I[O]$ that will enable $\mathcal{P}_R[O]$ in the following theorem.

Theorem 2. *For a unary operator O, let*

$$\mathbf{S}_g = \{preimage[O](p_r) | p_r \in \mathcal{P}_R[O]\}.$$

If $\mathbf{S}_g$ is compact relative to $\mathcal{P}_I[O]$, then $\mathcal{P}_I[O]$ enables $\mathcal{P}_R[O]$.

Proof. Suppose $\mathbf{S}_g$ is compact relative to $\mathcal{P}_I[O]$. Then,

$$\begin{aligned}
&\forall G \in \mathbf{S}_g, G \text{ is compact relative to } \mathcal{P}_I[O] \\
&\Rightarrow \forall G \in \{preimage[O](p_r) | p_r \in \mathcal{P}_R[O]\}, G \text{ is} \\
&\quad \text{compact relative to } \mathcal{P}_I[O] \\
&\Rightarrow \forall p_r \in \mathcal{P}_R[O], preimage[O](p_r) \text{ is compact} \\
&\quad \text{relative to } \mathcal{P}_I[O] \\
&\Rightarrow [\text{by Theorem 1}]\ \forall p_r \in \mathcal{P}_R[O], p_r \text{ will be emitted} \\
&\Rightarrow \forall p_r \in \mathcal{P}_R[O], \mathcal{P}_I[O] \text{ enables } p_r \\
&\Rightarrow \mathcal{P}_I[O] \text{ enables } \mathcal{P}_R[O].
\end{aligned}$$

□

That is, if the collection of preimages for each punctuation in $\mathcal{P}_R[O]$ is compact relative to $\mathcal{P}_I[O]$, then $\mathcal{P}_R[O]$ is

TABLE 3
State Models for Various Implementations of Query Operators

dupelim	$\mathbf{G}[\delta] = \{\{d\} \mid d \in D_I\}$
group-by	$\mathbf{G}[\mathcal{G}_A^F] = \{\{d \mid \forall r \in R - A, d.r \in D_I(r)\} \mid \forall a \in A, d.a \in D_I(a)\}$
join	$(\mathbf{G}^1[\bowtie_A], \mathbf{G}^2[\bowtie_A])$ where $\mathbf{G}^1[\bowtie_A] = \{\{d \mid \forall r \in R^1 - A, d.r \in D_I^1(r)\} \mid \forall a \in A, d.a \in D_I^1(a)\}$ $\mathbf{G}^2[\bowtie_A] = \{\{d \mid \forall r \in R^2 - A, d.r \in D_I^2(r)\} \mid \forall a \in A, d.a \in D_I^2(a)\}$
intersect	$(\mathbf{G}[\bigcap], \mathbf{G}[\bigcap])$ where $\mathbf{G}[\bigcap] = \{\{d\} \mid d \in D_I\}$
difference	$(\mathbf{G}[-], \mathbf{G}[-])$ where $\mathbf{G}[-] = \{\{d\} \mid d \in D_I\}$

R represents the input schema. For input domain D_I, the domain of attribute a is $D_I(a)$. Superscripts are used to denote specific inputs for binary operators.

enabled by $\mathcal{P}_I[O]$. In this case, we know that all punctuations in $\mathcal{P}_R[O]$ will be emitted and, therefore, all result data items will eventually be output.

The approach for binary operators is similar to that of unary operators and, hence, omitted (and can also be found elsewhere [7]).

5 Cleansing Query Operators

To reason about the state maintained by query operators during query execution, we use groupings to model state for various operators. We show that if the input punctuation schemes are complete and if all groupings for the state for an operator O are compact relative to those schemes, then O will be cleansed.

Our discussion of enabling a query operator O focused on the logical definition of O and did not need to consider the implementation of O. However, because we must model the state maintained during execution to determine whether O is cleansed, we must now consider its implementation. We limit our discussion to well-known implementations of query operators and, when possible, to those that may be suitable for processing nonterminating data streams. For example, the implementation of join that we consider does not block on either input and does not require indexes.

5.1 Modeling State Required for Query Operators

Many query operator implementations use a hash table, keyed on a subset of the attributes of a data item. Clearly, hash table structures conform neatly to groupings, where the hash-key attributes are the grouping attributes. In practice, different hash-key values may hash to the same bucket. Although our model does not exactly conform to the hash buckets, it is sufficient for modeling how state is maintained.

Our models for the state maintained by various operator implementations are shown in Table 3. Operators that maintain no state such as select duplicate-preserving project and union are trivially cleansed and are not listed.

- *DupElim*. Duplicate elimination can be implemented using a hash table, using all attributes as the hash key. Since the hash key includes all attributes, the grouping for this implementation is a collection of singleton sets. Using (A, B, C, D) as an example input schema, the grouping is $\mathbf{G}[\delta] = \{\{< a, b, c, d >\} \mid a \in A, b \in B, c \in C, d \in D\}$.
- *Group-by*. We implement group-by using a hash table, keyed on the group-by attributes. Therefore, we will model its state using a grouping on the group-by attributes. Again using (A, B, C, D), if the group-by attribute is A, then the grouping is $\mathbf{G}[\mathcal{G}_A^f] = \{\{< a, b, c, d > \mid b \in B, c \in C, d \in D\} \mid a \in A\}$. In practice, full data items are not held in state. Instead, only the information required to generate the resulting data items is kept. However, there is a correspondence between that information and the grouping we define.
- *Join*. We use the symmetric hash join [12] implementation for the join operator. Symmetric hash join maintains one hash table for each input, keyed on the join attributes. We model the state required for each input with a grouping, where the grouping attributes are the join attributes for that input. Consider the following two example input streams: S^1 with attributes (A, B, C, D) and S^2 with attributes (D, E, F), where the join condition is $S^1.D = S^2.D$. The grouping is the pair $(\mathbf{G}^1[\bowtie], \mathbf{G}^2[\bowtie])$, where

$$\mathbf{G}^1[\bowtie] = \{\{< a, b, c, d > \mid a \in A, b \in B, c \in C\} \mid d \in D\},$$
$$\mathbf{G}^2[\bowtie] = \{\{< d, e, f > \mid e \in E, f \in F\} \mid d \in D\}.$$

  Note that we model state as a pair of groupings for binary operators. This model contrasts to the *preimage* function, which returned a collection of pairs.
- *Intersect*. Intersect can be implemented as a special case of join on all attributes. Thus, we can model state as for join. The hash keys for the join will all attributes. Thus, the grouping is based on all attributes. Using (A, B, C, D) for both inputs, the grouping pair is $(\mathbf{G}[\bigcap], \mathbf{G}[\bigcap])$, where

$$\mathbf{G}[\cap] = \{\{< a, b, c, d >\} \mid a \in A, b \in B, c \in C, d \in D\}.$$

- *Difference*. We implement difference using a hash table for each input, keyed on all attributes, as in intersect. When some data item arrives from the positive side, we first probe the hash table for the negative side. Using input streams S^1 and S^2 with attributes (A, B, C, D), the two groupings are defined as $(\mathbf{G}[-], \mathbf{G}[-])$, where

$$\mathbf{G}[-] = \{\{< a, b, c, d >\} \mid a \in A, b \in B, c \in C, d \in D\}.$$

5.2 Cleansing Operators with Punctuation Schemes

Using the state models, we can describe the punctuation schemes that cleanse those operators.

Theorem 3. *Given a grammatical stream S, a unary operator O that discards state per the keep invariant for O at the earliest*

possible instant, a state model for O represented as a grouping $\mathbf{G}[O]$*, and an input punctuation scheme* $\mathcal{P}_I[O]$*, if* $G \in \mathbf{G}[O]$ *is compact relative to* $\mathcal{P}_I[O]$*, then all data items held in state for O that also exist in G will eventually be removed.*

Details for the proof for Theorem 3 are found elsewhere [7]. The general proof strategy is to show that any data item d that resides in state belongs to some group in the grouping model for that operator. Since that group is compact relative to the input punctuation scheme, some set of punctuations will eventually arrive that covers the group for d. As mentioned in Section 2, the keep invariant for O formally defines what data items must be held in state for correct output (and, therefore, what can be removed). We use the keep invariant to show that d will then be removed from state.

We extend the results on groups to groupings as a whole, and, therefore, the operator is cleansed by the input punctuation scheme.

Theorem 4. *Given a grammatical stream S, a unary operator O, and a complete punctuation scheme* $\mathcal{P}_I[O]$*, if the state model for O is a grouping* $\mathbf{G}[O]$ *that is compact relative to* $\mathcal{P}_I[O]$*, then* $\mathcal{P}_I[O]$ *cleanses O.*

Proof. We need to show that, for a unary operator, every data item $d \in D_I$ that at some point resides in state will eventually be removed. Since $\mathcal{P}_I[O]$ is complete, there is a punctuation $p \in \mathcal{P}_I[O]$ such that $match(d, p)$. Since S is grammatical, any such punctuation must arrive after d in S. Since d is in state, $d \in G_d$ for some group $G_d \in \mathbf{G}[O]$. Since $\mathbf{G}[O]$ is compact relative to $\mathcal{P}_I[O]$, G_d is also compact relative to $\mathcal{P}_I[O]$. By Theorem 3, all data items in state that are also in G_d can be removed. Therefore, $\mathcal{P}_I[O]$ cleanses O. □

The case for binary operators is similar and can be found elsewhere [7].

With these results, we can now determine if a given collection of punctuation schemes will benefit (enable and cleanse) specific queries, as we will show in Section 6. Note, however, we do not address cleansing an operator of punctuation. For example, though the union operator does not maintain data items in state, it must maintain punctuations in state in order to propagate punctuations correctly. Cleansing punctuations is an area for future work.

6 Determining if Punctuation Schemes Benefit Specific Queries

We now have the tools to demonstrate whether a given input punctuation scheme benefits a particular query operator. Now, we consider an entire query plan, composed of an arbitrary combination of query operators. Our approach is to associate input and output punctuation schemes with each operator in the plan, where the output punctuation scheme for an operator is an input punctuation scheme for its parent. If we can prove that each operator in the query plan benefits from its input punctuation(s), then we have demonstrated that the whole query benefits from the input punctuation schemes.

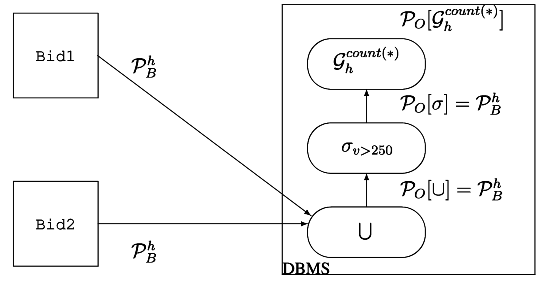

Fig. 4. Punctuation scheme assignments for Query 1.

6.1 Determining if the Example Queries Can Benefit

Consider Query 1, expressed in relational algebra: $Q1 = \mathcal{G}_h^{count(*)}(\sigma_{price>250}(Bid1 \cup Bid2))$. A query plan was given earlier in Fig. 2. Suppose the bid streams ($Bid1$ and $Bid2$) emit the following punctuation scheme: $\mathcal{P}_B^h = \{< *, *, *, h, * > | h \in \mathbb{Z}\}$. $\mathcal{P}_B^h$ benefits Query 1 if all of its operators benefit from their respective input punctuation schemes. Let $\mathcal{P}_O[\mathcal{G}_h^{count(*)}] = \{< h, * > | h \in \mathbb{Z}\}$ be the output punctuation scheme for the query. Fig. 4 shows punctuation scheme assignments for each operator.

We start at the union operator, with input punctuation schemes $\mathcal{P}_B^h$ on both inputs. We can use the definition of $preimage[\cup]$ to set the output punctuation scheme to $\mathcal{P}_B^h$. Similarly, using the definition of $preimage[\sigma]$, we can set the output punctuation scheme for select also to $\mathcal{P}_B^h$.

To show that $\mathcal{P}_B^h$ enables the group-by, we must show that some output punctuation p will be emitted if $preimage[\mathcal{G}_h^{count(*)}](p)$ is compact relative to $\mathcal{P}_B^h$. By definition, $preimage[\mathcal{G}_h^{count(*)}](p) = \mathcal{I}(< *, *, *, p(h), * >)$ As $p(h) \in \Pi_{\mathbb{Z}}$, some $p_b \in \mathcal{P}_B^h$ exists such that

$$\mathcal{I}(< *, *, *, p(h), * >)$$

is covered by p_b. Therefore, $\mathcal{I}(< *, *, *, p(h), * >)$ is compact relative to $\mathcal{P}_B^h$ and, so, $\mathcal{P}_B^h$ enables group-by.

To show that group-by is cleansed by $\mathcal{P}_B^h$, recall that the model for state is a grouping based on the group-by attribute, in this case, h:

$$\mathbf{G}[\mathcal{G}_h^{count(*)}] = \Big\{\{< a, b, v, h, m > | a \in \mathbb{Z}, b \in \mathbb{Z}, v \in \mathbb{R}, m \in [0, 59]\} | h \in \mathbb{Z}\Big\}.$$

It can be shown that $\mathcal{P}_B^h$ is compact relative to $\mathbf{G}[\mathcal{G}_h^{count(*)}]$. Therefore, given some group $G \in \mathbf{G}[\mathcal{G}_h^{count(*)}]$, there must exist some $p \in \mathcal{P}_B^h$ that covers G. Thus, by Theorem 4, $\mathbf{G}[\mathcal{G}_h^{count(*)}]$ is cleansed by $\mathcal{P}_B^h$; hence, $\mathcal{P}_B^h$ benefits group-by.

Therefore, as each operator in Query 1 benefits from its input punctuation ($\mathcal{P}_B^h$ in all cases) and since $\mathcal{P}_O[\mathcal{G}_h^{count(*)}]$ is complete, we have that Query 1 benefits from its input punctuation scheme $\mathcal{P}_B^h$.

In a similar fashion, appropriate punctuations can be shown to benefit Query 2. Suppose the bid stream emits a punctuation scheme $\mathcal{P}_B^a = \{< a, *, *, *, * > | a \in \mathbb{Z}\}$ and the auction stream emits a punctuation scheme

$$\mathcal{P}_A^i = \{< i, *, *, * > | i \in \mathbb{Z}\}.$$

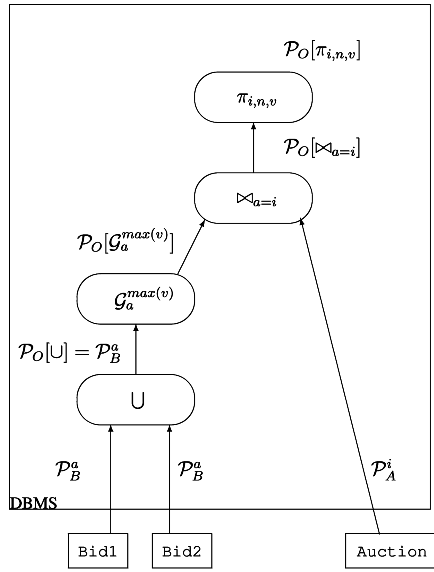

Fig. 5. Punctuation scheme assignments for Query 2.

That is, both streams emit punctuations that are compact relative to groupings defined on auction ID values. Note that such punctuations are realistic. Auctions will end in a fixed amount of time. Let us set the punctuation scheme assignments for the other operators as follows:

$$\begin{aligned} \mathcal{P}_O[\cup] &= \mathcal{P}_B^a, \\ \mathcal{P}_O[\mathcal{G}_a^{max(v)}] &= \{< a, * > | a \in \mathbb{Z}\}, \\ \mathcal{P}_O[\bowtie_{a=i}] &= \{< *, *, i, *, *, * > | i \in \mathbb{Z}\}, \\ \mathcal{P}_O[\pi_{i,n,v}] &= \{< i, *, * > | i \in \mathbb{Z}\}. \end{aligned}$$

Punctuation scheme assignments for Query 2 are illustrated in Fig. 5. The group-by operator can be shown to benefit from $\mathcal{P}_O[\cup]$, and the join operator can be shown to be cleansed by $\mathcal{P}_O[\mathcal{G}_a^{max(v)}]$ and $\mathcal{P}_A^i$ (note that join is trivially enabled). Finally, the project operator for Query 2 can only emit punctuation based on what kinds of punctuation it receives. In this case, it will emit $\mathcal{P}_O[\pi_{i,n,v}]$. Since $\mathcal{P}_O[\pi_{i,n,v}]$ is complete, Query 2 benefits from its input punctuations.

6.2 Algorithm for Determining if a Query Can Benefit

Using the examples above, we can formulate a general algorithm to determine if a given query can benefit from available punctuations. In essence, the algorithm is a depth-first traversal algorithm, where each operator requests all available punctuation schemes from its source(s) and determines which of those punctuation schemes will benefit that operator and, of those that do, what the available output streams are. If so, and if all operators benefit, then the query benefits.

Specifically, each query operator will support the iterator `availablePSchemes`. This iterator will first call `availablePSchemes` on its child node(s), then check if the returned punctuation schemes will benefit that operator. If not, then the operator will continue to call that method on its child node(s) until either punctuation schemes are returned that will benefit the operator or a `NULL` is returned, indicating that no more punctuation schemes

```
function availablePSchemes()
    PScheme ps = NULL
    PScheme psOut = NULL

    do
        ps = o.availablePSchemes()
    while ps != NULL AND NOT benefit(ps)

    if ps != NULL
        psOut = determinePScheme(ps)
    return psOut
```

Fig. 6. Algorithm for determining if a specific unary query operator will benefit from available punctuation schemes, where o is the child query operator.

are available. In that case, the operator also returns `NULL` and the query will not benefit from any available punctuation schemes. If punctuation schemes are returned from the child node(s) that benefit the operator, then an output punctuation scheme is generated based on the input punctuation schemes and the propagation invariant for that operator and that punctuation scheme is returned. If the root operator for the query plan returns a non-`NULL` complete punctuation scheme, then the query will benefit from the input punctuation scheme. The algorithm for unary operators is given in Fig. 6. It can be adapted to work for binary operators as well. The method `benefit` returns `true` if the given punctuation scheme benefits that operator. The method `determinePScheme` returns the output punctuation scheme based on the available input punctuation schemes and that operator's punctuation invariant. The definitions of these methods will vary for each operator, based on the propagation invariant for that operator.

7 The Describe Operator

We say that a punctuation *describes* a set of attributes if there exists a specific set of values for those attributes such that the punctuation covers all possible data items with those values. For example, given a grouping $\mathbf{G}$ over some attribute A, a punctuation that covers a group $G \in \mathbf{G}$ is said to *describe* the grouping attributes of G. In our punctuation format, given a set of attributes A in a schema R, a punctuation describes A if the pattern for every attribute in $R - A$ is the wildcard. Consider the punctuation schemes for Bid:

$$\begin{aligned} \mathcal{P}_B^h &= \{< *, *, *, h, * > | h \in \mathbb{Z}\}, \\ \mathcal{P}_B^a &= \{< a, *, *, *, * > | a \in \mathbb{Z}\}. \end{aligned}$$

Punctuations in $\mathcal{P}_B^h$ describe the `hour` attribute (as well as the `hour` and `minute` attributes and any other combination of attributes containing `hour`), and punctuations in $\mathcal{P}_B^a$ describe the `auctionid` attribute (and any combinations containing `auctionid`).

We have seen that operators often have a natural grouping of interest and that only punctuations of a certain form can benefit those operators. Rather than having every operator include a routine to determine if a given punctuation will be beneficial, we factor this functionality into a new operator, called the *describe* operator. The describe operator outputs data items as they arrive and filters out

incoming punctuations that will not help query operators further along in the query tree. In factoring this functionality into a separate operator, we avoid implementing separate code in operators such as join and group by to verify each incoming punctuation. Further, by taking advantage of equivalences involving describe and other operators, we are able to "push" the describe operator down in the query plan in order to filter out punctuations earlier during query execution. Some equivalences of this form have been defined elsewhere [7]. Note that the describe operator can be used to filter out all punctuations when the query does not require punctuations for proper execution (for example, selection queries).

Another function that can optionally be performed by the describe operator is to "build up" new punctuations from incoming punctuations when possible. For example, data items in our Bid stream have an hour attribute and a minute attribute. Suppose a punctuation arrives that matches all possible data items for hour 1 between minutes 0 and 15 ($< *, *, *, 1, [0, 15] >$), then another arrives that matches all possible data items for hour 1 between minutes 15 and 45 ($< *, *, *, 1, [15, 45] >$), and then a third arrives that matches all data items for hour 1 between 45 and 59 ($< *, *, *, 1, [45, 59] >$). From these punctuations, we can infer that all data items for hour 1 have arrived, even though we have not explicitly received that punctuation. In this example, the describe operator can emit a new punctuation that matches all data items for hour 1 ($< *, *, *, 1, * >$).

7.1 Defining Describe and Its Punctuation Invariants

The formal definition for describe is straightforward. We denote the describe operator as $\mathcal{D}_A(S)$, where A is the list of attributes that output punctuations should describe, and S is the input stream. Any data item that arrives is output.

Now, we define the punctuation invariants for describe. The pass and keep invariants for describe are simple. Describe is not a blocking operator. Therefore, the pass invariant for describe does not specify any additional data items to be output due to punctuations. Similarly, because describe does not store data items in state, the keep invariant is trivial (though punctuations may be stored in state).

The main purpose for the describe operator is to propagate punctuation, so the propagation invariant is more complex. Describe manipulates punctuations in one of two ways: First, only those punctuations that will help the query are emitted. Second, new punctuations are built up from incoming punctuations when possible. The first version of describe is relatively easy to define. If a punctuation arrives that describes the desired attributes, then we can pass it on. If not, then ignore it. We use the functions $data$ and $punct$ to separate out the data and the punctuations, respectively, from a stream. Given an input schema R for stream S and $ds = data(S)$ and $ps = puncts(S)$ are the data items and punctuations that have arrived from S, respectively,

$$cprop_{\mathcal{D}_A}(ds, ps) = \{x | x \in ps \wedge (\forall a \in (R - A), x(a) = *)\}, \quad (1)$$

that is, only output punctuations that have a wildcard value (*) for attributes that are not in the set of described attributes A.

Punctuations that alone do not describe the desired attributes may be combined into a new punctuation that does describe those attributes. In the following alternate definition of $cprop$ for describe, we use the function $setCoalesce$ that takes a set of punctuations and builds new valid punctuations from input punctuations:

$$cprop_{\mathcal{D}_A}(ds, ps) = \{x | x \in setCoalesce(ps) \wedge (\forall a \in (R - A), x(a) = *)\}. \quad (2)$$

This definition outputs all punctuations that describe the desired attributes, as well as the punctuations that can be derived from the punctuations received. Both definitions of $cprop$ are valid. Definition 2 comes at an implementation cost. We must keep punctuations in state as they arrive to be able to derive new punctuations when later punctuations arrive, thus making describe a stateful operator. Removing punctuations from state is an area of future work. Definition 1 does not have that added cost.

7.2 Implementation of Describe

In our implementation, the describe operator takes three parameters: The `attributes-to-describe` parameter is required. The `watch-attributes` and `watch-patterns` parameters are optional. The `attribute-to-describe` parameter specifies what punctuations are meaningful. In Query 1, the beneficial punctuations are those that match all data items for a given hour. Therefore, in that case, `attributes-to-describe` is set to `hour`. The `watch-attributes` parameter lists the attributes that can be used to build up punctuations that describe the attributes listed in the `attributes-to-describe` parameter. Each attribute in the `watch-attributes` parameter has a pattern in the `watch-patterns` parameter value defining a range that must be covered by input punctuations in order to generate the new punctuation. Again using Query 1 as an example, we can watch for punctuations that describe the `minute` attribute for a specific hour and, if they cover the range [0, 59] for a particular hour, new punctuation can be generated that describes that hour. Therefore, we set `watch-attributes` to `minute` and `watch-patterns` to the range [0, 59].

The implementation for (1) is easy—simply walk through the attributes of a punctuation and, for all nondescribe attributes, check that the value is the wildcard ($*$). Our current implementation for (2) is somewhat simplistic. We limit the `watch-attributes` parameter to a single attribute. In order to generate new punctuations, we only handle punctuations with range-type values for the `watch-attributes` and wildcard values for all other nondescribe attributes. As punctuations arrive, our goal is to combine ranges for the `watch-attributes` values from multiple punctuations that produce a cover for the `watch-patterns` value. When a cover for the `watch-patterns` has arrived, a new punctuation can be output with wildcard values for attributes not listed in the `attributes-to-describe` parameter value. An example of using the describe operator in Query 1 is shown in Fig. 7.

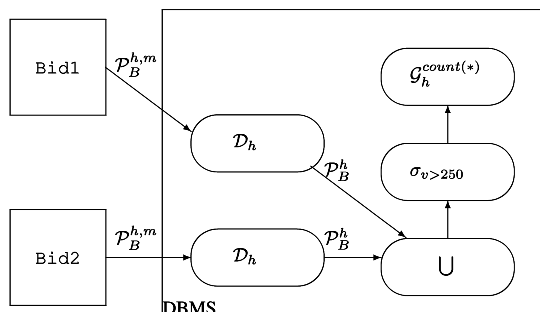

Fig. 7. Use of the describe operator in a query plan to output punctuations on the hour attribute (h), building up from punctuations on the minute attribute (m). $\mathcal{P}_B^{h,m} = \{< *, *, *, h, [m, m+9] > | h \in \mathbb{Z} \wedge m \in \{0, 10, 20, 30, 40, 50\}\}$. $\mathcal{P}_B^h = \{< *, *, *, h, * > | h \in \mathbb{Z}\}$.

We store incoming punctuations in a hash table. The hash key is built by concatenating values for each nondescribe attribute. When a punctuation arrives, we build the hash key and probe the hash table. If the key is not present, we use the range value for the `watch-attribute` as the hash value. If the key is present, we calculate the union of the existing range value with the range value from the punctuation. If that range covers the `watch-pattern`, then we build a punctuation with the wildcard for `watch-attribute`. Otherwise, we rehash the range back into the hash table and continue. The pseudocode for this algorithm is given in Fig. 8.

7.3 Benefit of Describe

We have implemented the describe operator into the NiagaraST query engine, developed based on Niagara [13]. We executed some performance tests to evaluate the performance overhead of punctuations, as well as to determine the effect of adding the describe operator to query plans. A more thorough discussion can be found elsewhere [7]. In short, we did notice that the describe operator can improve overall data throughput for queries that process punctuations. Particularly when the describe operator is able to filter out unwanted punctuations early or build up punctuations for later operators, we see a noticeable improvement in the rate at which data items can be processed. Such results are promising, but more evaluation is an area for future work.

8 Determining Punctuation Schemes for Other Streaming Approaches

Our grouping framework helps us determine if a given set of punctuation schemes benefits a particular query. We have also considered other strategies for querying over data streams: ordered data, windows, and heartbeats. For both ordered data and windows, we posit that punctuations are implicitly embedded in the stream which captures the behavior of those strategies and, therefore, we can define punctuation schemes for those punctuations. In this way, we can compare the kinds of queries each approach can support.

```
handlePunct(DataItem d)
  bool fDescribe = true
  bool fWatch = watch-attr != NULL
  string hashcode

  for each Attribute a in d
    if a.Name not in attrs-to-describe
      if a.Val != '*' then
        fDescribe = false

      if fWatch AND a.Name != watch-attr
        if a.Val != '*' then
          fWatch = false
        hashcode = hashcode + a.Val + ";"

  if fDescribe
    outputPunct(d)
  else if fWatch
    checkWatch(d, hashcode)

checkWatch(DataItem d, HashCode hashcode)
  Range rCurr = hashTable.get(hashcode)
  Range rNew = Range(d.watch-attr.Val)

  if rCurr == null
    hashTable.put(hashcode, rNew)
  else
    rNew = union(rNew, rCurr)
    if intersect(rNew, watch-pattern)
        == watch-pattern
      createAndOutputPunct(d)
      hashTable.remove(hashcode)
    else
      hashTable.put(hashcode, rNew)
```

Fig. 8. Algorithm for handling punctuations in the describe operator.

8.1 Positional and Ordered Data

Sequence database systems [14], [15] rely on metainformation about input sequences to optimize the execution of sequence queries. Many of the techniques used in sequence database systems can be applied to nonterminating data streams. Blocking and stateful operators can be implemented to take advantage of input that arrives in an interesting order. For example, Query 1 uses a group-by operator to group data items by `hour`. If the input arrives sorted on `hour`, when a new `hour` value arrives to the group-by operator, results for the previous `hour` can be output and that state cleared.

The Gigascope system [16] processes streams of network packet data. Operators in Gigascope rely on data that is monotonically nondecreasing on timestamp values. Specifically, the join operator must use a predicate that contains an attribute from each input that is monotonically nondecreasing. The join implementation uses this information to determine when a data item will no longer join with data items from the other join input and can therefore be removed from state (merge is similar). Further, group-by requires that some grouping attribute be monotonically nondecreasing. When a data item arrives whose ordered attribute is greater than any current group, the results for that group can then be output.

Ordered data fits nicely into our punctuation scheme framework. When data arrive in some known order, we

consider each data item d as an implicit punctuation: We know that no data item will arrive after d that precedes d in the sort order. That is, each d is the end of a group, where d is the maximal data item (according to the given order) in the group. Thus, each group in the grouping represents a prefix of the input domain.

Considering each data item as an implicit punctuation, we can define a punctuation scheme as follows, where A is the set of input domain attributes, S is set of ordering attributes, $primary(S)$ is the primary attribute of the ordering attributes S, and $domain(a_i)$ is the domain of the attribute a_i:

$$\begin{aligned}\mathcal{P}_S = \{ &< p_1, p_2, \ldots, p_n > \forall a_i \in A, \\ & a_i \neq primary(S) \Rightarrow p_i = *, \\ & a_i = primary(S) \Rightarrow p_i = [\bot, v_i) \\ & \quad \text{where } v_i \in domain(a_i)\}.\end{aligned}$$

That is, each punctuation has wildcard values for the attributes that are not the primary sorting attribute and a range pattern from $\bot$ up to a value for the primary sorting attribute.

For Gigascope, the primary sorting attribute is the timestamp attribute. A data item d serves as a punctuation with wildcard patterns for nontimestamp attributes and a range pattern that matches all values up to the timestamp value for d. (This is a common style of punctuations that we refer to as *linear punctuations*.) Considering d as a punctuation, a blocking operator can output results for groups up to the prefix including d, and a stateful operator can reduce state for those same groups. Since the punctuations are implicit in the data items, in order for punctuations to remain valid, operators must maintain ordered output. Thus, queries containing operators with groupings on the timestamp attribute will benefit from ordered data streams.

Given $\mathcal{P}_S$, we can tell that only queries with groupings of interest based on the sorted attribute will benefit from ordered inputs. Other queries will not benefit and are not appropriate for nonterminating input streams. In our example, Query 1 has a grouping of interest on `hour`. If data items arrive sorted by `hour`, then these systems will be able to execute that query. However, Query 2 has a grouping of interest on `auctionid`. As it is unlikely that the inputs will arrive sorted on `auctionid`, these systems will not be able to execute that query.

8.2 Windows

Many stream processing systems break input data into contiguous subsets, called *windows* [17], [18], [19], [20], [21]. The two most common kinds of windows are

- *tumbling windows*, which break the stream into successive, nonoverlapping subsets of data, and
- *sliding windows*, which break the stream into successive, overlapping subsets of data.

By breaking the stream input into bounded windows, a query operator processes bounded data sets. As each new window arrives, the query is restarted for that new window. Since the window's content is bounded, blocking operators output results before reaching the end of the stream. For example, we could redefine Query 1 to use fixed windows as "Output the number of high-valued bids each hour."

Each window has a first data item and a last data item. Thus, we must have some notion of when the last item for a window has arrived in order to process the window. For example, a window defined on time ends when the time runs out. That point when the window ends can be considered an implicit punctuation.

Li et al. [22] make window participation explicit in a data item using a window identifier (wid). We will use this approach in our discussion for concreteness. Let wid have domain W. Therefore, the punctuation scheme for windows could be

$$\begin{aligned}\mathcal{P}_W = \{ &< p_1, p_2, \ldots, p_n, wid > | \\ & \forall a_i \in A, p_i = *, wid \in W\}.\end{aligned}$$

A punctuation exists in the punctuation scheme for each wid. When a window completes, a punctuation is embedded into the stream stating that the window has closed. A blocking operator can output its results for that window, a stateful operator can remove from state the information for that window, and all operators can emit a punctuation stating that the window is complete. Thus, a query that defines windows over its input streams will benefit from $\mathcal{P}_W$.

We can see from $\mathcal{P}_W$ that queries that have groupings of interest on the window ID will benefit from windows. That is, if the query defines some kind of window, then these kinds of systems can be used over nonterminating streams effectively. Again, since Query 1 has a grouping of interest on the `hour` attribute, a window can be defined over that attribute and window queries will execute successfully.

Query 2 has a grouping of interest on `auctionid` for both the `Auction` and `Bid` streams as those streams participate in the join. A window can be defined for the `auctionid` attribute for the `Auction` stream, but it is impractical to try to define a window based on the `auctionid` over the `Bid` stream. Bids for open auctions will be interspersed with other bids, so each window will contain bids for multiple auctions. Auctions will likely close at different times, so it would not be appropriate to calculate the "current" closing price for all auctions in the window, only those that have actually closed. Clearly, Query 2 cannot be rewritten to use windows effectively.

8.3 Heartbeats

Similar to punctuations are heartbeats, presented by Babu and Widom [23]. The Stanford system requires that all data items be moved from the input manager to the query processor and that they be in order, which they term *progress*. Heartbeats are a mechanism by which progress can be ensured in all cases. They define a heartbeat τ to be an application-defined timestamp over a discrete ordered domain, which we will call $\mathcal{T}$. A heartbeat τ over a stream or set of streams tells the input manager that every subsequent timestamp on those streams will be greater than τ; thus, the input manager can move all buffered data items into the query processor. A heartbeat may be supplied by the stream source or, if such is not the case, it may be determined and supplied by the data stream management

system (DSMS). So long as at least one stream is supplying data, there is progress in the system. However, should all streams pause, there will be no further heartbeats and the input manager will continue to hold the buffered tuples. To avoid this situation, a timeout value $t_{timeout}$ is supplied by the user which will define the maximum amount of time that will be allowed before the DSMS declares a heartbeat and releases data from the input manager.

The entire point of defining progress on the data streams is to make sure that there are never any data items that are buffered indefinitely in the input manager. When a heartbeat is introduced, the input manager knows that the buffered data items may be sent. The input manager also is responsible for ordering any out-of-order data items as order on timestamp values is a requirement for the query processor.

Each heartbeat can be considered a punctuation, stating that no more data items will arrive with a timestamp value greater than the timestamp value of the heartbeat. Thus, we can define a punctuation scheme for heartbeat-style punctuations as follows:

$$\mathcal{P}_T = \{ < p_1, p_2, \ldots, p_n, [\perp, \tau] > | \\ \forall a_i \in A, p_i = *, \tau \in \mathcal{T} \}.$$

When a punctuation from $\mathcal{P}_T$ arrives, blocking operators output data items and stateful operators reduce state. Heartbeats only benefit queries over timestamps. Therefore, this scheme would function well for a time-based query such as in Query 1, but would not be able to process Query 2, where the grouping of interest is on `auctionid`.

8.4 Summary

We have shown how all three common approaches to processing data streams (windows, ordered data, and heartbeats) can be described using punctuations. We described each approach in terms of a punctuation scheme implied by it. The next step is to put this theory into practice. We have begun designing and developing a data stream processing system that will rely on punctuations to implement the various approaches. Such a system will allow us to evaluate the efficiency of our approach as compared to systems that are specifically designed to use other approaches.

9 Related Work

Data stream processing has received a great deal of attention recently in the data management community. Babcock et al. [24] give a good overview.

We have seen that some systems rely on input arrival order to handle queries over nonterminating streams. The Gigascope system [16] is a good example. Sequence data systems [14], [15] and temporal database systems [25] rely on input order. In our example, Query 1 groups on `hour`. If data arrive in an interesting order relative to a query, then these kinds of systems will be able to produce results.

Another approach is to define windows over the input data according to data arrival. *Sliding-window* queries were introduced in temporal database systems [26], [27] and later applied to queries over data streams in the Tangram system [28]. This kind of query has also been called a *moving-window* query. *Tumbling-window* queries (or *fixed-window* queries) are a special case of sliding-window queries. For a tumbling-window query, we alter the request slightly to only output results at the end of the sliding-window period. *Landmark windows* [19] consider all data items in a stream from some landmark forward to calculate a result. *Damped-window* queries [21] are an extension of sliding-window queries. A damped-window query evaluates each window along with previous windows together, where more recent windows make a greater contribution to the results for a window then older windows.

Li et al. [29] present a guarantee of safety for the join operator using punctuations. That is, their goal is to guarantee that the join operator in a given continuous join query can execute without requiring unbounded state based on knowledge of incoming punctuations. They introduce a simplified version of the punctuated schemes that we present, where each attribute is allowed a wildcard or nonwildcard pattern. Our punctuation schemes are more complete and we show how the punctuation schemes can be applied to many more query operators.

Other systems have used variants of punctuations with positive results. Shkapenyuk et al. [30], for example, show how punctuations can be applied in Gigascope to decrease query memory utilization. Further, Ding et al. [31] present a join algorithm that is optimized to process punctuations, and show that their algorithm outperforms pure window join algorithms in both memory usage and throughput.

10 Conclusions and Future Work

Understanding how punctuations improve the behavior of individual query operators is interesting. However, it is only a building block for a more important problem, namely, whether a given query can benefit from punctuations and, if so, what kinds of punctuations can benefit that query.

To this end, we use punctuation schemes to specify punctuations that may appear from a source. Punctuation schemes are defined using set notation and describe a natural grouping. Further, we have seen that various query operators also have natural groupings of interest. When the groupings defined by punctuation schemes cover the groupings of interest for a query, then the punctuation schemes may benefit the query. We showed how punctuation schemes can be used to define other common approaches to processing data streams.

Further, we have introduced the describe operator specifically to handle punctuations. Instead of each operator analyzing punctuations as they arrive to determine if they can benefit that operator, the describe operator can filter out unwanted punctuations. Further, the describe operator can be "pushed down" the query tree, filtering out unwanted punctuations as early as possible and building up punctuations when desired.

Finally, we presented a top-down algorithm for determining if a given query will benefit from any of the input punctuation schemes available from its inputs. Such an algorithm would be useful for optimizers when determining an appropriate query plan to execute over streams. If one query plan can benefit from incoming punctuations and another plan cannot, then perhaps choosing the first plan would be most appropriate.

Our work here is a starting point. Clearly, some significant issues remain. One is developing a more formal approach for determining necessary input punctuation schemes for a given query. To be effective, such an approach should be able to list multiple possible input punctuation schemes as well as suggest an "optimal" set of punctuation schemes. Optimality could mean the set of punctuation schemes that output data items the fastest, the set of punctuation schemes that minimize state, or some other criterion.

Currently, the placement and values of the parameters for the describe operator is done by hand. Determining those values are very much specific to the application data and the available punctuations. One could imagine a declarative language that defines relationships between attributes in a stream, which could then be used to determine the parameters for the describe operator. An area for future work is how to implement a query optimizer that handles the describe operator.

Our notion of cleansing an operator might be strengthened. An operator is cleansed if any data item that resides in state will eventually be removed from state. A stronger notion is to give a bound for state. This notion might be captured in one of at least three related ways. First, a given data item will be removed within n data items that follow it (similar to one kind of k-constraints [32]). Second, there is a bound on how many data items will be held in state. Finally, there is a bound on how long an item will be held in state. All three are useful, but require more knowledge about the properties of the input streams. For example, knowing order arrival of punctuations and the "skew" (or distance in tuples) between punctuations on different inputs would be useful in determining the maximum amount of time a data item will remain in state. Strengthening our notion of cleansing in these ways could improve how an optimizer chooses a query plan and is an area for future work.

It might make sense to quantify how much a query benefits from a punctuation scheme, rather than making a simple "yes/no" judgment. For example, all operators involved in a given query plan might benefit from punctuations, with the exception of a project operator. Project is neither a blocking nor a stateful operator, but, due to the projected attributes, it is not able to emit punctuations. We would want the query optimizer to ensure that the project operator is executed as near to the top of the query plan as possible, allowing more operators in the query plan to benefit. Further, if some query plan cannot benefit by any punctuation, we would like to determine if an alternative, equivalent query plan exists that can benefit. Finally, we would like to include available punctuation schemes in query optimization. For example, relevant punctuation schemes could be a logical property of subexpressions during query optimization and be used to limit construction of plans to those that can benefit from available input punctuations.

ACKNOWLEDGMENTS

The authors would like to thank Jennifer Widom for her many useful suggestions, Sava Krstic, John Matthews, Kent Jones, Lyle Cochran, and Donna Pierce for the discussions they had related to compactness and topologies, Leonidas Fegaras for discussions he had relating compactness to punctuation schemes and helping to find examples of equivalent queries with different behavior for a punctuation scheme, Vassilis Papadimos, Kristin Tufte, and Jin Li for various discussions on stream queries and feedback on this paper, and comments from the anonymous reviewers. Funding was provided by the US Defense Advanced Research Projects Agency (DARPA) through NAVY/SPAWAR Contract N66001-99-1-8908 and by US National Science Foundation Awards IIS0086002 and IIS0612311. Funding for Paul Stephens was provided by the Office of Career Services and by the Weyerhauser Younger Scholars Program at Whitworth University.

REFERENCES

[1] eBay homepage, http://www.eBay.com/, 2007.
[2] Yahoo! auctions homepage, http://auctions.yahoo.com/, 2007.
[3] J.A. Rodríguez, P. Noriega, C. Sierra, and J. Padget, "FM96.5 A Java-Based Electronic Auction House," *Proc. Int'l Conf. and Exhibition on the Practical Application of Intelligent Agents and Multi-Agent Technology,* pp. 207-224, Apr. 1997.
[4] P.R. Wurman, M.P. Wellman, and W.E. Walsh, "The Michigan Internet AuctionBot: A Configurable Auction Server for Human and Software Agents," *Proc. Second Int'l Conf. Autonomous Agents (Agents '98),* pp. 301-308, May 1998.
[5] A. Arasu, M. Cherniak, E. Galvez, D. Maier, A. Maskey, E. Ryvkina, M. Stonebraker, and R. Tibbets, "Linear Road: A Stream Data Management Benchmark," *Proc. Int'l Conf. Very Large Data Bases,* pp. 480-491, Aug. 2004.
[6] J. Li, D. Maier, V. Papadimos, P. Tucker, and K. Tufte, "NEXMark—A Benchmark for Queries over Data Streams," http://datalab.cs.pdx.edu/niagara/NEXMark/, 2003.
[7] P.A. Tucker, "Punctuated Data Streams," PhD dissertation, OGI School of Science and Eng. at Oregon Health and Science University, Aug. 2005.
[8] P.A. Tucker, D. Maier, T. Sheard, and L. Fegaras, "Exploiting Punctuation Semantics in Continuous Data Streams," *IEEE Trans. Knowledge and Data Eng.,* vol. 15, no. 3, pp. 555-568, May/June 2003.
[9] P.G. Selinger, M.M. Astrahan, D.D. Chamberlin, R.A. Lorie, and T.G. Price, "Access Path Selection in a Relational Database Management System," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* pp. 23-34, May 1979.
[10] R.H. Kasriel, *Undergraduate Topology.* W.B. Saunders, 1971.
[11] W. Rudin, *Principles of Mathematical Analysis.* McGraw Hill, 1964.
[12] A.N. Wilschut and P.M.G. Apers, "Dataflow Query Execution in a Parallel Main-Memory Environment," *Proc. IASTED Int'l Conf. Parallel and Distributed Information Systems,* pp. 68-77, Dec. 1991.
[13] J. Naughton, D. DeWitt, D. Maier, J. Chen, L. Galanis, K. Tufte, J. Kang, Q. Luo, N. Prakash, and F. Tian, "The Niagara Query System," *The IEEE Data Eng. Bull.,* vol. 24, no. 2, pp. 27-33, June 2000.
[14] P. Seshadri, M. Livny, and R. Ramakrishnan, "Sequence Query Processing," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* pp. 430-441, May 1994.
[15] P. Seshadri, M. Livny, and R. Ramakrishnan, "SEQ: A Model for Sequence Databases," *Proc. IEEE Int'l Conf. Data Eng.,* pp. 232-239, Mar. 1995.
[16] T. Johnson, C. Cranor, O. Spatscheck, and V. Shkapenyuk, "Gigascope: A Stream Database for Network Applications," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* pp. 647-651, June 2003.
[17] A. Arasu, S. Babu, and J. Widom, "The CQL Continuous Query Language: Semantic Foundations and Query Execution," *Int'l J. Very Large Data Bases,* vol. 15, no. 2, pp. 121-142, June 2006.
[18] S. Chandrasekaran and M.J. Franklin, "Streaming Queries over Streaming Data," *Proc. Int'l Conf. Very Large Data Bases,* pp. 203-214, Aug. 2002.
[19] J. Gehrke, F. Korn, and D. Srivastava, "On Computing Correlated Aggregates over Continuous Data Streams," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* pp. 13-24, May 2001.

[20] M. Sullivan and A. Heybey, "Tribeca: A System for Managing Large Databases of Network Traffic," *Proc. USENIX Ann. Technical Conf.*, pp. 13-24, June 1998.
[21] Y. Zhu and D. Shasha, "StatStream: Statistical Monitoring of Thousands of Data Streams in Real Time," *Proc. Int'l Conf. Very Large Data Bases*, pp. 358-369, Aug. 2002.
[22] J. Li, D. Maier, K. Tufte, V. Papadimos, and P.A. Tucker, "Semantics and Evaluation Techniques for Window Aggregates in Data Streams," *Proc. ACM SIGMOD Int'l Conf. Management of Data*, pp. 311-322, June 2005.
[23] S. Babu and J. Widom, "Continuous Queries over Data Streams," *SIGMOD Record*, vol. 30, no. 3, pp. 109-120, Sept. 2001.
[24] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, "Models and Issues in Data Stream Systems," *Proc. ACM SIGACT-SIGMOD-SIGART Symp. Principles of Database Systems*, pp. 1-16, June 2002.
[25] M.D. Soo, "Bibliography on Temporal Databases," *SIGMOD Record*, vol. 20, no. 1, pp. 14-23, 1991.
[26] G. Özsoyoğlu and R.T. Snodgrass, "Temporal and Real-Time Databases: A Survey," *IEEE Trans. Knowledge and Data Eng.*, vol. 7, no. 4, pp. 513-532, Aug. 1995.
[27] A. Segev and A. Shoshani, "Logical Modeling of Temporal Data," *Proc. ACM SIGMOD Int'l Conf. Management of Data*, pp. 454-466, May 1987.
[28] D.S. Parker, R.R. Muntz, and L. Chau, "The Tangram Stream Query Processing System," *Proc. Fifth IEEE Int'l Conf. Data Eng.*, pp. 556-563, Feb. 1989.
[29] H.-G. Li, S. Chen, J. Tatemura, D. Agrawal, K.S. Candan, and W.-P. Hsiung, "Safety Guarantee of Continuous Join Queries over Punctuated Data Streams," *Proc. Int'l Conf. Very Large Data Bases*, pp. 19-30, Sept. 2006.
[30] V. Shkapenyuk, T. Johnson, O. Spatscheck, and S. Muthukrishnan, "A Heartbeat Mechanism and Its Application in Gigascope," *Proc. Int'l Conf. Very Large Data Bases*, pp. 1079-1088, Aug. 2005.
[31] L. Ding, N. Mehta, E.A. Rundensteiner, and G.T. Heineman, "Joining Punctuated Streams," *Proc. Ninth Int'l Conf. Extending Database Technology*, pp. 587-604, Mar. 2004.
[32] S. Babu, U. Srivastava, and J. Widom, "Exploiting k-Constraints to Reduce Memory Overhead in Continuous Queries over Data Streams," *ACM Trans. Database Systems*, vol. 29, no. 3, pp. 545-580, Sept. 2004.

Peter A. Tucker received the BS degree in mathematics and computer science in 1991 from Whitworth College and the PhD degree from Oregon Health and Science University in 2005. He worked for eight years at Microsoft Corp. in software development, holding such roles as software design engineer in testing and software design engineer, before attending the OGI School of Science and Engineering at Oregon Health and Science University. He is currently an assistant professor at Whitworth University in the Department of Math and Computer Science. His current interests include data stream processing, computer ethics, and functional programming. He is a member of the ACM, IEEE, and IEEE Computer Society.

David Maier received the BA (Honors College) degree from the University of Oregon, with a double major in mathematics and computer science and the PhD degree from Princeton University in electrical engineering and computer science. He is currently the Maseeh Professor of Emerging Technologies in the Department of Computer Science at Portland State University. He was previously on the faculties of the OGI School of Science and Engineering (formerly the Oregon Graduate Institute) at the Oregon Health and Science University and of the State University of New York at Stony Brook. He has held visiting positions at l'Institut National de Recherche en Informatique et en Automatique (INRIA, Rocquencourt) and the University of Wisconsin-Madison. His research interests include query processing, object-oriented systems, databases and data-product management for scientific computing, superimposed information systems, and stream data processing. He is a fellow of the ACM and holds the ACM SIGMOD Innovations Award. He is a senior member of the IEEE and a member of the IEEE Computer Society.

Tim Sheard received the PhD degree in computer and information science from the University of Massachusetts, Amherst, in 1985 and is currently a professor of computer science at Portland State University. His research interests include program generation, metaprogramming systems, theorem proving, logical frameworks, type systems, domain specific languages, and patterns for functional programming. He was the general chair of Generative Programming and Component Engineering (GPCE '04) and was organizer of the 2001 ICFP Programming Contest that attracted more than 250 entries from around the world. He is a pioneer in the area of metaprogramming and is the creator of three research artifacts (MetaML, Template Haskell, and the Omega Programming Language) which have a broad influence on the programming language community. He is a member of the IEEE.

Paul Stephens is currently working toward the BS degree in computer science, the BA degree in mathematics, and the BA degree in applied physics at Whitworth University. He has experience in business IT working at UPF Incorporated in Spokane, Washington, where he was active in database management and Web services. Following graduation, he is interested in pursuing graduate studies in natural language processing.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.