Journal of Universal Computer Science, vol. 16, no. 20 (2010), 3073-3101
submitted: 1/7/10, accepted: 13/9/10, appeared: 1/11/10 © J.UCS

Support for Schema Evolution in Data Stream Management Systems

James F. Terwilliger
(Microsoft Corporation, Redmond, Washington, USA
james.terwilliger@microsoft.com)

Rafael J. Fernández-Moctezuma
(Portland State University, Portland, Oregon, USA
rfernand@cs.pdx.edu)

Lois M. L. Delcambre
(Portland State University, Portland, Oregon, USA
lmd@cs.pdx.edu)

David Maier
(Portland State University, Portland, Oregon, USA
maier@cs.pdx.edu)

Abstract: Unlike Database Management Systems (DBMSs), Data Stream Management Systems (DSMSs) do not evaluate queries over static data sets — rather, they continuously produce result streams to standing queries, and often operate in a context where any interruption can lead to data loss. Support for schema evolution in such an environment is currently unaddressed. In this work we address evolution in DSMSs by introducing a new element to streams, called an *accent*, that precedes and describes an evolution. We characterize how a subset of commonly used query operators in DSMS act on and propagate accents with respect to three evolution primitives: Add Attribute, Drop Attribute, and Alter Data.
Key Words: data streams, schema evolution
Category: H.2.1, H.2.4, H.2.8

1 Introduction

Similar to traditional Database Management Systems (DBMSs), Data Stream Management Systems (DSMSs) allow users to declaratively construct and execute queries. Unlike a DBMS, which allows a user to issue a query over a persistent database instance and retrieve a single answer, a DSMS continuously produces results from a *standing* query as new data flows through the system. Data often arrives faster than it can be persisted in a streaming environment, so any time a standing query ceases operation, data are potentially lost.

Data streams are typically treated as unbounded with a consistent schema, but assuming that the schema for a stream never changes is unreasonable. With current DSMS technologies, supporting schema evolution entails bringing down a standing query and instantiating an evolved version of the query against the new schema. In the presence of stateful operators, either state must be propagated from the old query to the evolved query to resume computation, or query answers derived from old state might be lost, in addition to the data that arrived and was not processed during downtime. We address the problem of schema evolution in DSMSs by describing the semantics of a system where queries can automatically respond to evolution and thus continue query evaluation without interruption.

Traditional challenges in DSMSs such as low-latency result production and efficient use of resources have been addressed by adding *markers* in the stream that signal progress, i.e. that signal some notion that all tuples that match a given condition in an unbounded stream have already been seen. Examples include CTIs in CEDR [Barga et al. 2007], heartbeats in Gigascope [Johnson et al. 2005], and punctuations in NiagaraST [Li et al. 2008, Tucker et al. 2003]. We see an opportunity to use embedded markers in a stream to support data and schema evolution. In this paper, we introduce markers called *accents* into a data stream alongside tuples. Traditional query operators produce a query answer by operating on the input relation(s), but stream query operators additionally operate on the aforementioned markers. We continue this trend here by defining stream query operators that can process data and schema evolution accents.

The goal is to have the operators that can appear in a standing query that was instantiated before an accent was issued remain active during and after the processing of the accent. The following challenges arise for each operator: (1) Determine whether the operator can adapt to the schema changes in an accent, and (2) define and implement the processing of each accent for the operator, minimizing blocking and state accumulation while adapting to the evolution. The definition of the processing for each operator includes the local processing of the accent; local processing may involve modifying the query operator parameters, modifying the data stored in local state, preparing to modify future incoming data arriving from input streams, and emitting accents at the proper time to be processed by downstream query operators.

We support evolution that affects only a subset of the stream; for instance, we allow adding new attributes to a stream only for tuples that meet a specified condition. This capability allows evolution to occur gradually, allowing data from different sources to evolve at their own pace (e.g., a firmware upgrade distributed to a collection of sensor sources, which cannot be guaranteed to complete simultaneously).

This paper is organized as follows: Section 2 defines the evolution problem in the context of data streams. Section 3 introduces our approach to expressing evolution needs. Section 4 defines the formal semantics of evolution for a meaningful subset of relational operators. In Section 5, we introduce notions of correctness that allow us to reason about whether accent-aware queries produce proper results. Section 6 introduces a mechanism called *operator signatures* that allow us to reason about the properties of operators at a high level. We discuss related work in Section 7 and outline opportunities for future work in Section 8. This paper is an extended version of a paper that introduced the basic idea behind accents [Fernández-Moctezuma et al. 2009]; the unique contributions of this paper beyond that work are the introduction and formalization of operator correctness, the work on signatures, as well as more complete descriptions of the considered operators.

2 The Evolution Problem Exposed

Our examples use the schema `sensors(ts, s, t)`, `placement(s, l)` to describe a stream of temperature readings from a sensor network and the placement of those sensors. A monotonically increasing attribute `ts` represents timestamps, `s` uniquely identifies a sensor in the network, `l` is a sensor location, and `t` is a reading in Fahrenheit. Consider the following standing query in extended Relational Algebra:

$$\gamma^{AVERAGE}_{\{l,wid\},t}(\mathcal{W}^{5\text{ minutes}}_{ts,wid}(sensors \bowtie_{sensors.s=placement.s} placement))$$

This query joins the two data streams `sensors` and `placement` on `s`, then assigns each resulting tuple a window identifier, `wid`, (for tumbling windows of length 5 minutes) based on the value of the `ts` attribute. Finally, it computes the average temperature per location and window id. We explore several evolution scenarios and how they affect this query.

Example 1. A firmware update changes temperature units. Sensors receive a firmware update and temperature is now reported in Celsius. Note that sensors may not receive the update simultaneously. Consider the example in Figure 1(a), where Sensor #2 has implemented the firmware update as of time `ts = 5`, but none of the other sensors have (tuples in Celsius are underlined in the figure). The content of one window includes tuples with both Fahrenheit and Celsius, and the aggregation in the query produces inconsistent results. Thus, in addition to propagating the desired change through the operators (and the output), we also need to account for subsets of the stream having different semantics. Operators need to know which sensor readings need to be scaled back to Fahrenheit or scaled forward to Celsius. Figure 1(b) shows the same stream but with a marker embedded in the stream indicating the point at which the temperature units change. The marker describes the following information:

- The sensor whose data semantics have changed ($s = 2$).
- Two functions $y \to (y - 32) \times (5/9)$ and $y \to y \times (9/5) + 32$ indicating formulae that can be applied to compare tuples that occur after the marker against those that appear before it. More details appear in Section 3.

Note that, inside the stream, the data evolution marker propagates through several of the operators, right up to the aggregation operator. If at some point, Sensor #1 also switches to Celsius, that marker could propagate through the query as well. However, the aggregate operator now knows that all sensors have switched to Celsius (assuming that it knows the domain of sensors to be $\{1, 2\}$) and can emit its own marker to say all results from that point forward will be in Celsius. The operator would, from that point forward, receive and emit all data in Celsius.

Example 2. A firmware update extends the schema. Sensors now produce `pressure` information in addition to temperature, resulting in a new schema: `sensors(ts,s,t,p)`. The query shown above can either continue to produce the same results by projecting out the new attribute, or the query can be modified to produce averages for pressure, as well as temperature. In general, as a response to evolution, a query may be (1) unaffected, (2) adapted to accommodate the evolution, or (3) rendered unable to execute correctly. As with Example 1, the stream may contain a mix of old and new data.

3 Modeling Streams and Evolution

To support schema flexibility, we redefine streams to have optional attributes in a similar fashion to Lorel [Abiteboul et al. 1997] or RDF [RDF 2004]. We define a tuple in a data stream to be a partial function t from the global set of possible attribute symbols $\mathcal{C}$ to a set of attribute values $\mathcal{V}$. Let X be an element in $\mathcal{C}$. If $t(X)$ is defined, the tuple t has the value $t(X)$ for attribute X; if $t(X)$ is undefined, it is semantically equivalent to evaluating to $\perp$ (null) for attribute X. We represent a tuple as a set of attribute-value pairs enclosed by angle brackets. The tuple with value 1 for A and value 2 for B, and $\perp$ for all other attributes is `<A:1, B:2>`.

According to the previous definition, a tuple can accommodate any attribute in $\mathcal{C}$. In practice, a schema R restricts the attributes that any tuple in a stream with schema R presents. Formally, we say a schema R is a list of attributes from $\mathcal{C}$, a list that evolves over time based on constructs to be introduced in this section but has some definitive starting point at the beginning of query operation.

A *stream* is a (possibly infinite) list T of tuples. The stream has an intrinsic order based on the arrival of tuples into the system, and may not necessarily correlate with order with respect to an attribute such as timestamp. A *prefix (substream)* of stream T is a finite set of tuples that appear consecutively in T.

We consider three evolution primitives in data streams:

– Add Attribute $+(A)$: attribute A is added to tuples in an incoming stream S — specifically, adding an attribute A to the schema of S.

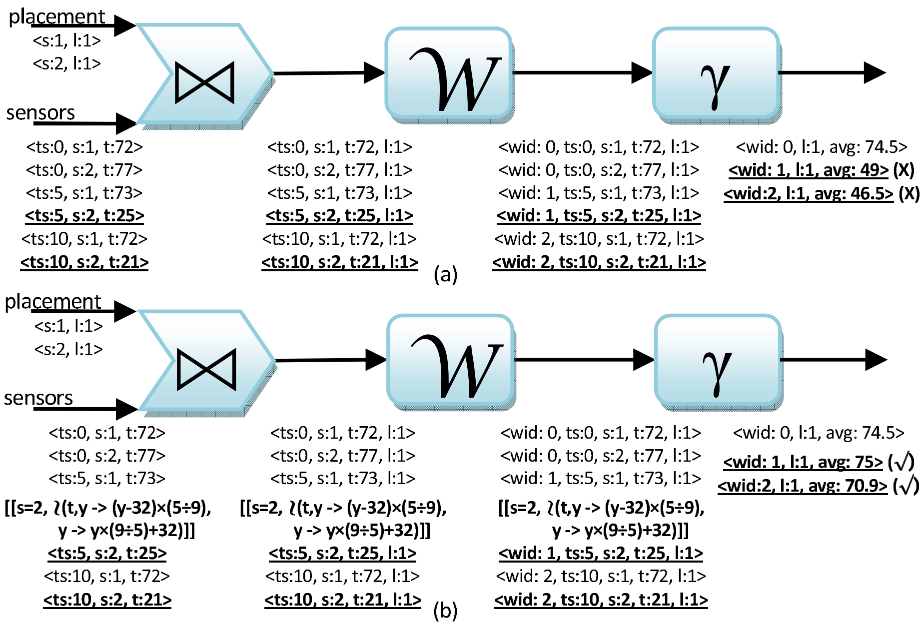

Figure 1: Query plan with sample tuples from Example 1. (a) A change in domain in temperature readings from sensor 2 (tuples in bold) propagates through the query leading to an erroneous average indicated by an X. (b) Schema evolution descriptions (see Section 3), indicated by entries with double braces, indicates the shift in domain allowing operators to handle the evolution and compute the correct average indicated by a checkmark.

– Drop Attribute $-(A)$: attribute A is removed from an incoming stream (more accurately, ensures that future tuples are undefined on attribute A).

– Alter Data $\wr(A, \alpha, \beta)$: data in attribute A is altered by the invertible, order-preserving function α whose inverse is β. Order-preserving means that if $x < y$, $\alpha(x) < \alpha(y)$. We further assume that the functions are affine, so that the functions may commute with common aggregates (the sum aggregate can be derived from average and count aggregates, which interact well with affine maps). In practice, this property is satisfied by the two most common functions that we envision — scaling and translating. Clearly, there are functions that one might consider using in an Alter Data primitive that are invertible but neither order-preserving nor affine, such as a function that translates strings into an abbreviated form or vice versa.

We refer to the actions $+$, $-$, and $\wr$ as *evolution primitives*. To model evolution in a stream, we define an *accent* as a *description* coupled with an evolution primitive. A description is a partial function d from the set of attribute symbols $\mathcal{C}$ to $\mathcal{P} \times \mathcal{V}$, where $\mathcal{P}$ is the set of simple comparators $\{=, \neq, <, \leq, >, \geq\}$. For example, $d(A) = (<, 10)$ is interpreted as the predicate $A < 10$. We disallow the evaluation of symbols not present in a given schema, hence, $d(\perp) = (<, 10)$ is meaningless. For readability, we use the traditional infix notation in our examples in this paper. The description d itself is the conjunction of all predicates corresponding to the defined inputs of d. We denote an accent as $[[d, e]]$, where d is a description and e is an evolution primitive, for example $[[X < 10, +(Y)]]$. An *accented stream* $\tilde{S}$ is a stream of tuples and accents, where accents anticipate and describe evolutions. That is, an accent precedes all data that conforms to the new schema or data representation indicated by the evolution primitive.

In this paper, we only consider accents whose evolution primitive e refers to a column on which the description d is not defined. Thus, as an example, the following accent is not allowed because of the potential ambiguity and the complexity of the processing required: $[[amount < 100, \wr(amount, y \rightarrow y \times 1.1, y \rightarrow y/1.1)]]$.

Accents represent evolution of schema and data on the subset of tuples in a stream that satisfy the predicates in the description. In Figure 1(b), the accent $[[s = 2, \wr(t, y \rightarrow (y - 32) \times (5/9), y \rightarrow y \times (9/5) + 32]]$ indicates that temperature values t from Sensor #2 have been translated from Fahrenheit to Celsius (the α function). In the example, the *join* and *window* operators simply propagate the accent because the temperature attribute is not involved in the join or windowing condition. By propagating the accent immediately upon receipt, the join and windowing operators guarantee anticipatory placement of accents in the stream. The aggregate operator γ applies the inverse function β to data from Sensor #2 (until such time as all sensors have been updated to report Celsius) to produce the correct aggregation values. No accent is propagated after γ in the example as no other accents have arrived from the stream at that point; however, should accents arrive subsequently altering the temperature on all other sensors (i.e., enough accent descriptions to cover the domain of attribute s), the window operator could then issue an accent to its output and start emitting tuples in Celsius.

An accent $a = [[d, e]]$ establishes, for tuples that satisfy description d, a relationship between tuples that occur before a versus after it in the stream:

- If e is an Add Attribute primitive $+(X)$, then any tuple t that follows accent a in the stream and satisfy d may be defined for $t(X)$, while tuples before a are undefined for $t(X)$ (up to any point where primitive $-(X)$ appears earlier than a in the stream, if present).
- If e is a Drop Attribute primitive $-(X)$, then $t(X)$ must be undefined for any tuple t that follows accent a in the stream that satisfies d (up until a corresponding Add Attribute primitive occurs, if any).

– If e is an Alter Data primitive $\wr(X, \alpha, \beta)$, then tuples that follow accent a in the stream and satisfy d have their values for attribute X translated by α. Let t and t' be tuples in the stream that match description d. Recall that order of arrival induces an ordering on tuples. Assume $t < a < t'$. Then, $t \equiv t'$ if $\forall_{B \in \mathcal{C}, B \neq X}(t(B) = t'(B)) \wedge (t(X) = \beta(t'(X))$ (or equivalently $\alpha(t(X)) = t'(X))$).

An accented stream is *valid* if its placement of Add or Drop Attribute accents respect the properties specified above. For instance, one cannot place two accents that add the same attribute for the same description in a stream unless one finds a corresponding Drop Attribute accent between them. The primitive $\wr$ describes how to reason about equality over tuple values, e.g.., it induces an equivalence relation over tuples relative to their location. For instance, consider the finite substream in Figure 1 (b). Any service operating on the stream, whether it be a query operator or an application reading a query output, may treat the temperature value at `ts = 0` and the temperature value at `ts = 5` for `s = 2` as equal, since $(77 - 32) \times (5/9) = 25$ (converting 77 Fahrenheit to 25 Celsius).

4 Modeling Stream Operator Behavior

Most DSMSs support a variety of query operators that accept streams as input and produce streams as output. For instance, for any finite substream of tuples T as input to the operator $\sigma_{C<V}$ (where $C \in \mathcal{C}$, the set of possible attributes), the operator produces as output the finite substream $\{t \in T | t(C) < V\}$. In this section, we describe the semantics of stream operators when they work on accented streams; in particular, we describe how and when accents are output. We also detail the circumstances under which operators fail to support a particular accent, *i.e.*, when a query operator fails to support an evolution.

Our description of how operators process accents includes a discussion of the state associated with the management of accents. We provide some ideas of how state can be managed in this section. Note that efficient management of state in DSMSs (for example, in stateful operators) is enabled by markers in the stream [Barga et al. 2007, Johnson et al. 2005, Li et al. 2008, Tucker et al. 2003]. Our work on using those markers to assist in accent state management is still a work in progress and is discussed briefly in Section 8.

We use the notation $\tilde{S}$ to denote accented streams and $\tilde{S}_O$ to denote an operator's accented output stream. Here, we define how six well-known stream operators react the presence of accents, including how they propagate accents.

We use a notional algebra consisting of the following operators: Select, Project, Union, Join, Window, and Aggregate. Our notional algebra exemplifies the most common relational operators found in stream systems, namely, stateless relational operations (such as filtering), stateful relational operations (such as join), and blocking operations (such as average). While stream algebras are richer than the subset presented (for example, speculative systems use a cleanse operator [Barga et al. 2007]), the presented subset illustrates the use of our framework. The specific operator implementations are not the only correct ones. We have chosen these to minimize stateful behavior and simplify discussion. In our notation, the arrow symbol "$\rightarrow$" represents logical implication.

Select ($\sigma_{C\theta V}\tilde{S}$) Select operator on a single predicate $C\theta V$ with attribute C, comparator θ, and value V on stream $\tilde{S}$. The attribute named in the selection condition C is *necessary* to the Select operator's functioning. This restriction means that C must be present in the schema at the time the query was issued. Responses to various accents are as follows:

* $[[d, +(X)]]$: If $X \neq C$, propagate $[[d, +(X)]]$ to $\tilde{S}_O$. If $X = C$, the accent is not propagated since the attribute X must already exist. Practically speaking, as the attribute must already exist for the operator to function, so we do not expect such an accent to arrive.

* $[[d, -(X)]]$: Regardless of description d, if $X = C$, Select fails to support the evolution, and the query aborts execution. If $X \neq C$, output $[[d, -(X)]]$ to $\tilde{S}_O$.

* $[[d, \wr(X, \alpha, \beta)]]$: If $X = C$, adjust the selection predicate to be $C\theta(\alpha(V))$ for tuples described by d (leaving it as $C\theta V$ for all others), and propagate the accent $[[d, \wr(X, \alpha, \beta)]]$ to $\tilde{S}_O$. If $X \neq C$, no adjustment is necessary and $[[d, \wr(X, \alpha, \beta)]]$ is propagated to $\tilde{S}_O$.

Project ($\Pi_{\mathbf{C}}\tilde{S}$) Project tuples on the attribute set $\mathbf{C}$. The project operator may seem trivial at a glance, but in fact is sensitive to accents with a description that refers to attributes projected out, which we define as $\mathbf{R} = \mathcal{C} - \mathbf{C}$; $\mathcal{C}$ is the set of all possible attribute symbols. We define the operation of project assuming no mechanism exists to map descriptions referring to $\mathbf{R}$ to descriptions referring to $\mathbf{C}$ — we relax this definition in Section 6:

* $[[d, +(X)]]$: If d refers only to attributes in $\mathbf{C}$ and $X \in \mathbf{C}$, output $[[d, +(X)]]$ to $\tilde{S}_O$. If d refers to attributes in $\mathbf{R}$, abort. Otherwise, no accent is output to $\tilde{S}_O$. Notice that this assumes that the project attribute list $\mathbf{C}$ may include attributes that may not exist in the original schema of the stream, allowing one to specify a project operator in a query that may anticipate the arrival of additional attributes.

* $[[d, -(X)]]$: If d refers only to attributes in $\mathbf{C}$ and $X \in \mathbf{C}$, output $[[d, -(X)]]$ to $\tilde{S}_O$. If d refers to attributes in $\mathbf{R}$, abort. Otherwise, no accent is output to $\tilde{S}_O$.

* $[[d, \wr(X, \alpha, \beta)]]$: If d refers only to attributes in $\mathbf{C}$ and $X \in \mathbf{C}$, output $[[d, \wr(X, \alpha, \beta)]]$ to $\tilde{S}_O$. If d refers only to attributes in $\mathbf{C}$ and $X \notin \mathbf{C}$, no accent is output. If d refers to attributes in $\mathbf{R}$, then abort.

Union ($\tilde{S} \cup \tilde{T}$) Union of input streams $\tilde{S}$ and $\tilde{T}$. Because tuples are defined as partial functions, there is no need for there to be union compatibility between S and T. To coordinate the processing of accents across input streams, we assume maintenance of input-related state to keep track of the descriptions we have seen on each input. By state we mean maintaining a list of received accents.

* $[[d, +(X)]]$: W.l.o.g., assume the accent is seen on input $\tilde{S}$. If an accent adding attribute X is not in input $\tilde{T}$'s state, add $[[d, +(X)]]$ to the state of input $\tilde{S}$ and propagate $[[d, +(X)]]$ to $\tilde{S}_O$.

If an accent $[[f, +(X)]]$ is in input $\tilde{T}$'s state:

- Replace $[[f, +(X)]]$ with $[[f \wedge \neg d, +(X)]]$ in input $\tilde{T}$'s state — or remove $[[f, +(X)]]$ from $\tilde{T}$'s state if $f \rightarrow d$.
- If $d \not\rightarrow f$, add $[[\neg f \wedge d, +(X)]]$ to $\tilde{S}$'s state and output $[[\neg f \wedge d, +(X)]]$ to output stream $\tilde{S}_O$. The effect is to propagate a new accent to the output that covers tuples that were not described by a previously propagated accent.

For instance, consider a case where accent $[[a = \text{“}FOO'', +(X)]]$ is in input $\tilde{T}$'s state when accent $[[b < 5, +(X)]]$ arrives on $\tilde{S}$. We perform three actions: First, replace the accent in $\tilde{T}$ with $[[a = \text{“}FOO'' \wedge b \geq 5, +(X)]]$, representing the predicate of tuples to watch for on input $\tilde{S}$ with attribute X not yet added. Second, add accent $[[a \neq \text{“}FOO'' \wedge b < 5, +(X)]]$ to the input state of $\tilde{S}$, representing the set of tuples from $\tilde{T}$ that have not yet added X. Finally, propagate accent $[[a \neq \text{“}FOO'' \wedge b < 5, +(X)]]$, indicating the set of tuples that now have attribute X added to the output but were not yet added by the previous accent $[[a = \text{“}FOO'', +(X)]]$.

As a simpler case, consider accent $[[a = \text{“}FOO'', +(X)]]$ in state for $\tilde{T}$ when accent $[[a = \text{“}FOO'', +(X)]]$ arrives on $\tilde{S}$. All new incoming and outgoing tuples with $a = \text{“}FOO''$ have added attribute X, so the accent is removed from state and no additional accent propagates.

* $[[d, -(X)]]$: Union maintains state, but does not propagate a drop attribute accent until it has been seen on both inputs. W.l.o.g., assume the accent is seen on input $\tilde{S}$. If an accent dropping attribute X is not in input $\tilde{T}$'s state, add $[[d, -(X)]]$ to state of input $\tilde{S}$, but propagate nothing.

If an accent $[[f, -(X)]]$ is in input $\tilde{T}$'s state:

- Replace $[[f, -(X)]]$ with $[[f \wedge \neg d, -(X)]]$ in input $\tilde{T}$'s state — or remove it if $f \rightarrow d$.
- If $d \not\rightarrow f$, add $[[\neg f \wedge d, -(X)]]$ to input $\tilde{S}$'s state.
- Output $[[f \wedge d, -(X)]]$ to $\tilde{S}_O$ (describing the set of tuples covered by both descriptions).

* $[[d, \wr(X, \alpha, \beta)]]$: Union also does not propagate an accent with an Alter Data primitive until there is coordination. W.l.o.g., assume the accent is seen on input $\tilde{S}$. If no accent describing an Alter Data on X by α is in input $\tilde{T}$'s state, any subsequent tuple t seen in input $\tilde{S}$ will have its X value replaced with $\beta(t(X))$ in $\tilde{S}_O$. The requirement to do that processing is represented by placing the accent $[[d, \wr(X, \alpha, \beta)]]$ in input $\tilde{S}$'s state.

  If an accent $[[f, \wr(X, \alpha, \beta)]]$ is in input $\tilde{T}$'s state:

  - Replace $[[f, \wr(X, \alpha, \beta)]]$ with $[[f \wedge \neg d, \wr(X, \alpha, \beta)]]$ in input $\tilde{T}$'s state — or remove it if $f \rightarrow d$. The effect of removing or contracting accent $[[f, \wr(X, \alpha, \beta)]]$ from the state of input $\tilde{T}$ is that all subsequent tuples from t will not be altered (unless they match the new description). Hence, such tuples will propagate unaltered to output.

    This scenario is illustrated in depth in Figure 2.
  - If $d \not\rightarrow f$, add $[[\neg f \wedge d, \wr(X, \alpha, \beta)]]$ to input $\tilde{S}$'s state to indicate all subsequent tuples from $\tilde{S}$ that match $\neg f \wedge d$ should be altered.
  - Output $[[f \wedge d, \wr(X, \alpha, \beta)]]$ to $\tilde{S}_O$.

Note: this description of the Union operator is incomplete; to be complete, incoming accents may need to have their descriptions altered based on any Alter Data accents in the operator's state.

Join ($\tilde{S} \bowtie_{\mathcal{J}(C_{\tilde{S}}, C_{\tilde{T}})} \tilde{T}$) Join input streams $\tilde{S}$ and $\tilde{T}$ using equi-join conditions $\mathcal{J}$, which reference attributes $C_{\tilde{S}}$ and $C_{\tilde{T}}$ from streams $\tilde{S}$ and $\tilde{T}$ respectively. Join's behavior on accents that reference the join attributes are operationally the same as Union, i.e., there needs to be synchronization before propagation, and that synchronization is irrespective of the specific join condition $\mathcal{J}$. Similar to Select, dropping attributes named in the join condition causes Join to fail.

* $[[d, +(X)]]$: W.l.o.g., assume the accent is seen on input $\tilde{S}$. If an accent adding attribute X is not in input $\tilde{T}$'s state, add $[[d, +(X)]]$ to state of input $\tilde{S}$ and output $[[d, +(X)]]$ to $\tilde{S}_O$. If an accent adding attribute X

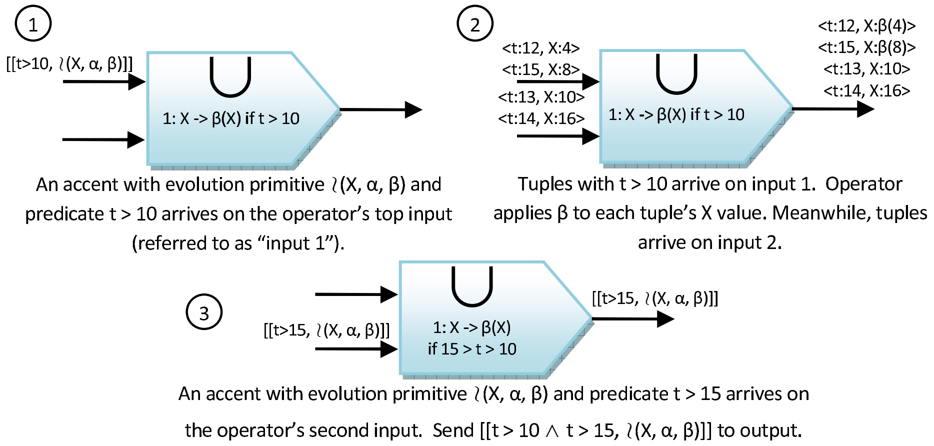

Figure 2: An example of the Union operator responding to accents on both inputs.

is in input $\tilde{T}$'s state ($[[f, +(X)]]$), abort execution (fail), since duplicate attributes (even from distinct streams) are not allowed, and we assume schema disjointness.

* $[[d, -(X)]]$: If $X \in (C_{\tilde{S}} \cup C_{\tilde{T}})$, fail; otherwise, propagate the accent.

* $[[d, \wr(X, \alpha, \beta)]]$: W.l.o.g., assume the accent is seen on input $\tilde{S}$ and no accent describing an Alter Data on X by α is in input $\tilde{T}$'s state. Any subsequent tuple t seen in input $\tilde{S}$ must have its X component replaced by $\beta(t(X))$ before the join occurs. We indicate that action by adding $[[d, \wr(X, \alpha, \beta)]]$ to input $\tilde{S}$'s state. If an accent $[[f, \wr(X, \alpha, \beta)]]$ is in input $\tilde{T}$'s state:

- Replace $[[f, \wr(X, \alpha, \beta)]]$ with $[[f \wedge \neg d, \wr(X, \alpha, \beta)]]$ in input $\tilde{T}$'s state — or remove it if $f \rightarrow d$ — with the effect that the operator will no longer alter t for tuples matching description f.
- If $d \not\rightarrow f$, add $[[\neg f \wedge d, \wr(X, \alpha, \beta)]]$ to input $\tilde{S}$'s state to indicate that subsequent matching tuples on $\tilde{S}$ will be altered by β.
- Output $[[f \wedge d, \wr(X, \alpha, \beta)]]$ to $\tilde{S}_O$.

Note that, because the join condition $\mathcal{J}$ is an equi-join, we need not alter condition $\mathcal{J}$ on arrival of $\wr$ primitives. Such primitives only propagate when they arrive on both inputs, and thus the attributes must still be equal between the two inputs.

We expect that in practice the set of attributes on which descriptions are defined and the set of attributes on which evolution primitives are defined do not generally overlap. The definition of Join specified here would be complicated by considering cases where those two sets of attributes overlap.

Window ($\mathcal{W}^{w}_{\mathbf{C},wid}\tilde{S}$) Consider a windowing operator for input stream $\tilde{S}$, argument attributes $\mathbf{C}$, output attribute wid, and windowing function $w : \mathbf{C} \rightarrow wid$. The Window operator applies a function w to $t[\mathbf{C}]$ for tuples $t \in \tilde{S}$ to produce a Window ID, most often used for grouping by subsequent aggregate operators. Attributes in $\mathbf{C}$ are *essential* to Window's operation.

* $[[d, +(X)]]$: By our validity conditions, $X \notin \mathbf{C}$ (as with Select, in practice this check is not necessary as the attribute already exists). Thus, Window propagates $[[d, +(X)]]$ to $\tilde{S}_O$.

* $[[d, -(X)]]$: Regardless of description d, if $X \in \mathbf{C}$, Window fails to support the evolution, and the query aborts execution. If $X \notin \mathbf{C}$, Window propagates $[[d, -(X)]]$ to $\tilde{S}_O$.

* $[[d, \wr(X, \alpha, \beta)]]$: If $X \in \mathbf{C}$, adjust the windowing function to operate on $\beta(X)$ for tuples described by d, and the accent is not emitted in $\tilde{S}_O$. If $X \notin \mathbf{C}$, make no adjustment and output $[[d, \wr(X, \alpha, \beta)]]$ to $\tilde{S}_O$.

* In all cases, if the description d in accent $[[d, e]]$ refers to attributes in $\mathbf{C}$, use the windowing function w to reconstruct a new description d' that refers instead to the new windowing attribute wid. Clearly, the original accent should not propagate unaltered because its description refers to attributes that will likely be dropped by a later operation. Accents should thus "move" to the new windowing attribute, which is difficult because the windowing attribute is likely at a lower granularity than the other attributes. Thus, accents are "moved" slightly to accommodate the window boundaries.

  For instance, if an accent with description $t > 10 \wedge region = A$ arrives at a windowing operator that windows on attribute t, replace the description with $wid > z \wedge region = A$, where $z = w(10) + 1$ (i.e., the next window after the window that time value 10 is in). Also, if $e = \wr(X, \alpha, \beta)$, add $[[t > 10 \wedge t \leq b \wedge region = A, \wr(X, \alpha, \beta)]]$ to the input state of $\tilde{S}$ where b is the value of t that marks the boundary between the window z and the previous one. All tuples that match the accent in the state have function β applied as necessary so that the Alter Data accent effectively begins along a window boundary.

Aggregate ($\gamma^{f}_{\mathbf{G},\mathbf{E}}S$) Aggregation operator with aggregate function f, grouping attributes $\mathbf{G}$, and exclusion attributes E on input S. This operator performs a grouping operation, similar to the GROUP BY clause in SQL. The operator partitions the input tuples of stream S according to their values for attributes G, and produces the result of the aggregation function for each partition. Unlike the traditional GROUP BY clause (and also unlike the standard stream query aggregate operator as used in Figure 1), rather than specify the attributes to aggregate (such as the `t` attribute in the example in Section 2) we specify the attributes that should not be aggregated (such as the `ts` and `s` attributes in the same example). In other words, the operator ignores attributes $\mathbf{E}$ (i.e., the excluded attributes) and aggregates on all other non-grouping attributes. This method of specification allows data found in newly added attributes to be aggregated (such as the `pressure` attribute in Example 2 in Section 2):

* $[[d, +(X)]]$: If $X \in \mathbf{E}$, do nothing. If $X \notin \mathbf{E}$, propagate $[[d, +(X)]]$ to $\tilde{S}_O$, adjust internal aggregation to apply the aggregate f to the X attribute of all tuples described by d.

* $[[d, -(X)]]$: If $X \in \mathbf{E}$, do nothing. If $X \in \mathbf{G}$, abort the query because Aggregate fails to support the evolution. If $X \notin \mathbf{G}$, propagate $[[d, -(X)]]$ to $\tilde{S}_O$, and purge any partial aggregates that may appear in state for attribute X that match description d, since that data has effectively been dropped.

* $[[d, \wr(X, \alpha, \beta)]]$: If $X \in \mathbf{E}$, do nothing. If d refers to any attributes but those in $\mathbf{G}$, then abort. Otherwise, propagate the accent to output. Since partial aggregates are being computed, apply $\alpha(X)$ to computations in memory. In other words, tuples that match description d may appear both before and after the accent arrives and has been propagated. Since the accent propagates immediately, any tuples that arrived *before* the accent will now contribute to output that appears *after* the accent. Thus, applying αX to all partial computations assures that all data for a particular aggregation is subject to the post-accent semantics, regardless of the arrival order of tuples and accents. So, for instance, if tuples arrive expressed in Fahrenheit and, mid-computation, an accent arrives changing to Celsius, the operator converts partially aggregated results to Celsius regardless of the units used for each input tuple.

5 Correctness of Accent Propagation and Processing

In Section 4, we described the operational handling of accents by operators in an algebra and the affect this handling has in accent-related state. In this section, we present some notions of correctness. We introduce two such notions

— *propagation correctness* and *point-in-time correctness* — and spend most of the section describing the latter.

Propagation correctness is a sentiment that when an accent enters a query, the operators in that query make a best effort to propagate the accent to query output (short of operator abort or impossibility due to attributes missing in operator output). In addition, query output is valid with respect to the Add and Drop Attribute properties introduced in Section 3. There, we introduced accents as a means to convey schema evolution signals in a stream. With propagation correctness, we also imbue accents with an intent to propagate. Alternative correctness models are possible; for instance, consider a system that, rather than propagating an Alter Data accent, captures them in an operator's state and effectively undoes all of its effects. Returning to the example of Figure 1, operators could absorb Alter Data accents and then transform all subsequent incoming tuples accordingly. Thus, though data may arrive at the first operator as a mixture of Fahrenheit and Celsius, all data would emerge in Fahrenheit — the original semantics for the stream. The data that emerges from the operator is still correct, but even if all sensors switch to Celsius data, the query would still produce data in Fahrenheit. Propagation correctness ensures that the semantics of a query's output evolve as its input evolves, which gives application clients the choice as to how to deal with evolving data.

The operators as defined in Section 4 are correct by construction with respect to propagation. Each operator propagates all accents except when the accent causes an operator to cease functioning or if the accent description or primitive refers to attributes that do not and cannot exist in the operator output. Queries are thus correct in this respect by induction over individual operators.

Point-in-time correctness captures the notion that operators produce correct results with respect to data. We draw inspiration for point-in-time correctness from the CEDR stream system [Barga et al. 2007]. In CEDR, operators can work on data that may be retracted in the future; the correctness of how an operator acts on such data is guaranteed by comparing the output of the operator against what the operator would have output if the input was already in its final state. The definition of a CEDR operator on retraction-free data is then used as a baseline semantics against which the semantics of the operators in the presence of retractions are compared.

We adopt a similar approach with accents; we can prove an operator's behavior on a snapshot of accented input to be correct by comparing it against the operator's behavior on the same snapshot that has been restructured so that it is essentially unaccented, where all tuple data occurs together. We call this restructured stream a *canonical accented stream prefix.*

Let $s = (s_1, s_2, \ldots, s_n)$ represent a finite substream in an accented stream $\tilde{S}$. A *replacement* for that substream is another finite substream $t = (t_1, t_2, \ldots, t_n)$

such that replacing s with t in-place in $\tilde{S}$ does not affect the information content of the stream. By "information content", we mean that $+$ and $-$ primitives still have valid placement, e.g. one cannot add an attribute that already exists. Also, tuples must still respect the equivalence relations that are induced by $\wr$ primitives, that tuples on either side of an Alter Data primitive may be compared via the primitives α function. For instance, $([[d, e]], t)$ is a replacement for $(t, [[d, e]])$ for any tuple t that does not match description d. Two finite substreams of accented streams $\tilde{S}$ and $\tilde{S}'$ are *content equivalent* $\tilde{S} \equiv \tilde{S}'$ if one can transform $\tilde{S}$ into $\tilde{S}'$ using replacements.

A finite substream of an accented stream is *canonical* when all accents with a $+$ primitive appear at the beginning of the substream, followed by all tuples, followed by all accents with a $\wr$ primitive, and ending with all accents with a $-$ primitive. To *canonize* a substream $\tilde{S}$ is to find a canonical substream (denoted as $\hat{S}$) that is content equivalent to the original substream. A canonical substream thus represents a point in time for query output that looks very much like a relation, where the query output is aware of all attributes that exist or have ever existed up to that point and all data has the semantics as it existed at the time of query specification. That the $\wr$ primitives are moved to the end of the substream rather than the beginning is a convenience that makes proofs more straightforward.

We use canonization as part of our formal descriptions; operators do not canonize results as part of their operation, nor do they require canonized input. As mentioned before, we use point-in-time correctness in this paper on a snapshot of accented input, which is not the same as simply taking the current state of a query's output. This distinction is not necessary for many operators, but is important for operators that maintain state. For instance, the aggregation operator technically will not produce results for an aggregation until it has received a clear signal — a punctuation in Niagra or a CTI in CEDR — that the tuples in a given aggregation window have all been accounted for. In other words, an aggregate operator that takes the average of all temperatures from sensor 1 between 3:00 and 4:00 cannot produce a result until it has some guarantee that all such tuples have arrived. The mere presence of a tuple that arrives from 4:05 is insufficient, as the tuples may arrive out of order. A full point-in-time correctness proof for a stateful operator must thus take punctuation into consideration.

For an operator implementation to be point-in-time correct on a snapshot of accented input, it must respect the commutativity diagram in Figure 3 on any input where the query does not abort. The diagram states that for a given set of accented input substreams, the operator produces accented output substreams whose canonical versions are equivalent to the output the operator produces on canonical input. Accents in output streams must respect accent properties, e.g., that following a drop attribute accent in a stream, all tuples matching

$$\begin{tikzcd}
\widetilde{S}_1, \widetilde{S}_2, \ldots, \widetilde{S}_n \arrow[r, "\text{Op}"] \arrow[d, "Canonize"] & \widetilde{T}_1, \widetilde{T}_2, \ldots, \widetilde{T}_m \arrow[d, "Canonize"] \\
\hat{S}_1, \hat{S}_2, \ldots, \hat{S}_n \arrow[r, "\text{Op}"] & \hat{T}_1, \hat{T}_2, \ldots, \hat{T}_m
\end{tikzcd}$$

Figure 3: Commutativity diagram for an operator Op. Top: Accented streams are processed Op on inputs $\tilde{S}_i$ to output accented streams $\tilde{T}_i$. Bottom: Canonical version of the input substreams S_i result in canonical output substreams T_i. Accented substreams can be canonized on both ends to show equivalence.

the accent's description are undefined for the dropped attribute. Our notion of correctness implies that there is no information loss due to accent-aware stream processing.

5.1 Replacement Equivalences

This section details a list of replacement equivalences that can be applied to an accented stream to create another, content-equivalent accented stream by commuting stream elements. Each individual replacement can be proven correct trivially based on the properties of valid streams, which is to say that accents in the stream are faithful to the tuples they represent. For instance, if one finds accent $[[d, +(X)]]$ in the stream, the stream would not be valid if the accent were immediately preceded by a tuple that satisfies d and is defined on X, for in that case the accent does not correctly model the content of the stream.

There are several replacement rules that describe how to commute items in an accented stream:

Commute accent with tuple not matching the accent's description:
$([[d, e]], t) \equiv (t, [[d, e]])$ for any tuple t that does not match description d.

Commute accents with non-overlapping descriptions:
$([[d, e]], [[d', e']]) \equiv ([[d', e']], [[d, e]])$ if $d \wedge d' = false$.

Commute accents with non-overlapping primitive attributes:
$([[d, e]], [[d', e']]) \equiv ([[d', e']], [[d, e]])$ if e and e' do not refer to the same attribute and neither e nor e' are Alter Data primitives.

Commute Add Attribute with tuple not defined on attribute:
$([[d, +(X)]], t) \equiv (t, [[d, +(X)]])$ for any tuple t that matches description d and for which $t(X)$ is undefined.

Commute Drop Attribute with tuple not defined on attribute: $([[d, -(X)]], t) \equiv (t, [[d, -(X)]])$ for any tuple t that matches description d and for which $t(X)$ is undefined.

Commute Alter Data with tuple, data adjusted: $([[d, \wr(X, \alpha, \beta)]], t) \equiv (t', [[d, \wr(X, \alpha, \beta)]])$ for any tuple t that matches description d and $(\forall_{B \in \mathcal{C}, B \neq X}(t(B) = t'(B))) \wedge (\beta(t(X)) = t'(X))$.

Commute Alter Data with accent: $([[d, \wr(X, \alpha, \beta)]], [[d', e]]) \equiv ([[d'', e]], [[d, \wr(X, \alpha, \beta)]])$ if $d' \rightarrow d$, d' is defined on X, and $d'' = d$ except that if $d(X) = (\theta, c)$, $d''(X) = (\theta, \beta(c))$. (Note that if $d' \rightarrow d$, then $d'' \rightarrow d$ as well, so this equivalence can be applied in either direction).

We note that each of these commutations is content preserving by definition. For instance, the Commute Drop Attribute with tuple rule holds because moving a tuple past a Drop Attribute accent does not affect the applicability of the accent — it is still the case that all matching tuples following the accent hold the appropriate property in that they lack the dropped attribute. We also define replacement rules not pertaining to commutativity:

Divide/combine descriptions: $([[d, e]]) \equiv ([[d_1, e]], \ldots, [[d_n, e]])$ if $d = \bigvee_{i \in \{1 \ldots n\}} d_i$, and, for any $i \neq j$, $d_i \wedge d_j = false$). This rule describes how to break an accent into "smaller" accents. In particular, $([[d, e]] \equiv ([[d \wedge (c\theta v), e]], [[d \wedge (c\theta' v), e]])$ for an attribute c, value v, and comparators θ and θ' that are complements of one another (e.g., $<$ and $\geq$).

Annihilation of opposing additions and deletions: $([[d, -(X)]], [[d, +(X)]]) \equiv ([[d, +(X)]], [[d, -(X)]]) \equiv ()$.

Annihilation of Alter Data accents with inverted functions: $([[d, \wr(X, \alpha, \beta)]], [[d, \wr(X, \beta, \alpha)]]) \equiv ()$.

Composition of Alter Data accents: $([[d, \wr(X, \alpha, \beta)]], [[d, \wr(X, \gamma, \delta)]]) \equiv ([[d, \wr(X, \gamma \circ \alpha, \beta \circ \delta)]])$.

5.2 Canonization

The intuition behind content equivalence and canonization is similar to the concepts of conflict equivalence and conflict serializability in transaction management. In transaction management, given a sequence of database actions from several transactions, one can construct a conflict-equivalent schedule by commuting two actions that do not cause a write conflict. The goal of such commuting is to attempt to construct a conflict-equivalent schedule that is grouped by transaction. Constructing a canonized stream is similar in principle in that it uses replacements to group together stream elements by their type.

Unlike conflict serializability, any accented stream can be canonized. To prove this property, we can take any two adjacent accents in a valid accented stream that are not in canonical order — for instance, a tuple occurring before an Add Attribute accent — and use replacements to construct a content equivalent stream with the accents in question replaced with items in canonized order. A full proof means showing that any arbitrary pair of out of order items — whose cases we now enumerate — can be replaced to be in canonical order.

Proposition: Any accented stream can be canonized.

Case 1: Tuple before Add Column: Let $(t, [[d, +(X)]])$ be adjacent items in a valid stream. We know that tuple t is not defined on attribute X if it satisfies description d, or the stream would not be valid (the tuple would have an attribute not yet available to it). Therefore, $(t, [[d, +(X)]]) \equiv ([[d, +(X)]], t)$ by the "Commute accent with tuple not matching the accent's description" replacement if t does not satisfy d, and by the "Commute Add Attribute with tuple not defined on attribute" replacement otherwise.

Case 2: Tuple after Alter Data: Let $([[d, \wr(X, \alpha, \beta)]], t)$ be adjacent items in a valid stream. If t does not satisfy description d or is not defined on attribute X, then $([[d, \wr(X, \alpha, \beta)]], t) \equiv (t, [[d, \wr(X, \alpha, \beta)]])$ by the same replacement rules as Case 1. If t satisfies description d and is defined on X, then $([[d, \wr(X, \alpha, \beta)]], t) \equiv (t', [[d, \wr(X, \alpha, \beta)]])$ by the "Commute Alter Data with tuple, adjusted for function" replacement, where t' is the tuple that results from replacing $t(X)$ with $\beta(t(X))$ in accordance with the replacement rule definition.

Case 3: Tuple after Drop Column: Let $([[d, -(X)]], t)$ be adjacent items in a valid stream. We know that tuple t is not defined on attribute X if it satisfies description d, or the stream would not be valid (the tuple would have an attribute not available to it anymore). Therefore, $([[d, -(X)]], t]]) \equiv (t, [[d, -(X)]])$ by the "Commute accent with tuple not matching the accent's description" replacement if t does not satisfy d, and by the "Commute Drop Attribute with tuple not defined on attribute" replacement otherwise.

Case 4: Drop Attribute before Add Attribute: Let $([[d, -(X)]],$ $[[d', +(Y)]])$ be adjacent accents in a valid stream. If $X = Y$, then neither d nor d' is defined on X. So,
$([[d, -(X)]], [[d', +(X)]])$
$\equiv ([[d \wedge \neg d', -(X)]], [[d \wedge d', -(X)]], [[d \wedge d', +(X)]], [[d' \wedge \neg d, +(X)]])$ (by divide description rule)
$\equiv ([[d \wedge \neg d', -(X)]], [[d' \wedge \neg d, +(X)]])$ (by annihilation rule)

$\equiv ([[d' \wedge \neg d, +(X)]], [[d \wedge \neg d', -(X)]])$ (because descriptions no longer overlap).

If $X \neq Y$, we know that d' cannot be defined on X and have any overlap with d — otherwise, the stream would be invalid because the description d' refers to an attribute that does not exist. Similarly, d cannot be defined on Y and have any overlap with d'. Therefore, $([[d, -(X)]], [[d', +(Y)]]) \equiv ([[d', +(Y)]], [[d, -(X)]])$ by the commute accents with non-overlapping descriptions rule. $\Box$

Case 5: Drop Attribute before Alter Data: Let $([[d, -(X)]], [[d', \wr(Y, \alpha, \beta)]])$ be adjacent accents in a valid stream. If $X = Y$, then $d' \rightarrow \neg d$, since otherwise we would be altering a column that does not exist. Thus, the two accents commute by a single replacement rule.

If $X \neq Y$, either $d' \rightarrow \neg d$ or d' is not defined on X, or the second accent description is invalid.

If $d' \rightarrow \neg d$, then either d is defined on attribute Y or not.

If so, $([[d, -(X)]], [[d', \wr(Y, \alpha, \beta)]]) \equiv ([[d', \wr(Y, \alpha, \beta)]], [[d'', -(X)]])$ where d'' is the description d whose Y component is altered by function α according to the commute Alter Data with accent rule.

If not defined on Y, the commute holds without description transformation.

If d' is not defined on X, translate $([[d, -(X)]], [[d', \wr(Y, \alpha, \beta)]])$
$\equiv ([[d \wedge d', -(X)]], [[d \wedge \neg d', -(X)]], [[d', \wr(Y, \alpha, \beta)]])$ (by the divide descriptions rule)
$\equiv ([[d \wedge d', -(X)]], [[d', \wr(Y, \alpha, \beta)]], [[d \wedge \neg d', -(X)]])$ (by the commute accents with non-overlapping descriptions rule)
$\equiv ([[d', \wr(Y, \alpha, \beta)]], [[d'' \wedge d', -(X)]], [[d \wedge \neg d', -(X)]])$ (commute accent through Alter Data, altering description). $\Box$

Case 6: Alter Data before Add Column: Let $([[d, \wr(X, \alpha, \beta)]], [[d', +(Y)]])$ be adjacent accents in a valid stream. This case follows the same logic as Case 5. If $X = Y$, then $d' \rightarrow \neg d$, since otherwise we would be altering a column that does not exist. Thus, the two accents commute by a single replacement rule.

If $X \neq Y$, either $d' \rightarrow \neg d$ or d is not defined on Y, or the first accent description is invalid.

If $d' \rightarrow \neg d$, then either d' is defined on attribute X or not.

If so, $([[d, \wr(X, \alpha, \beta)]], [[d', +(Y)]]) \equiv ([[d'', +(Y)]], [[d', \wr(X, \alpha, \beta)]])$ where d'' is the description d' whose X component is altered by function β according to the commute Alter Data with accent rule.

If not defined on X, the commute holds without description transformation.

If d is not defined on Y, translate $([[d, \wr(X, \alpha, \beta)]], [[d', +(Y)]])$
$\equiv ([[d, \wr(X, \alpha, \beta)]], [[d \wedge \neg d', +(Y)]], [[d \wedge d', +(Y)]])$ (by the divide descriptions rule)
$\equiv ([[d, \wr(X, \alpha, \beta)]], [[d \wedge \neg d', +(Y)]], [[d \wedge d', +(Y)]])$ (by the commute accents with non-overlapping descriptions rule)
$\equiv ([[d \wedge \neg d'', +(Y)]], [[d, \wr(X, \alpha, \beta)]], [[d \wedge d', +(Y)]])$ (commute accent through Alter Data, altering description). $\Box$

5.3 Example: Select

With canonization, we can reason about correctness in two ways. First, we can reason about the correctness of data processing — an operator produces correct data if, for a given canonical input, the correct tuples are in the canonical output. Second, we can reason about accent propagation — an operator propagates accents correctly if, for a given canonical input, the correct accents are in the canonical output. Here, we provide a definition of a stream operator — Select — on canonical input, and explain why its operational semantics as presented in Section 4 are correct.

Select ($\sigma_{C\theta V}\hat{S}$): Let $\hat{S}$ be the canonized finite substream at a given point in time for the input of the Select operator. Let $\hat{S} = \mathcal{P} + \mathcal{T} + \mathcal{A} + \mathcal{D}$, where $\mathcal{P}, \mathcal{T}, \mathcal{A}, \mathcal{D}$ are the Add Attribute, tuple, Alter Data, and Drop Attribute items, respectively, in the canonical stream, and $+$ is substream concatenation in order. Then:

$\sigma_{C\theta V}\hat{S} =$ `if` $\exists_i x_i = [[d, -(C)]]$ `then abort`
`else` $\mathcal{P} + \sigma_{C\theta V}(\mathcal{T}) + \mathcal{A} + \mathcal{D}$

In short, the definition of Select fails whenever the C attribute is dropped (as expected), and for all other input streams, produces output where all accents are propagated and the tuple data payload is filtered based on the predicate $C\theta V$ as with the normal operator. This last point is crucial — for the unaccented tuple set, the operator only carries out its relational operation, just as its stream-relational, unaccented counterpart would. Clearly, the operational description of Select from Section 4 will produce the result described above when provided canonical input. The difference between the operational description and the canonical definition lies in processing of Alter Data accents and adjusting the selection predicate for different descriptions, and since all tuples arrive before Alter Data accents on canonical input, the original predicate is the one applied to all incoming tuples.

The only remaining question is whether the operational definition of Select produces correct output. Since canonization occurs as a sequence of replacement steps, first consider the "Commute Alter Data with tuple, adjusted for function" replacement by itself. If $[[d, \wr(X, \alpha, \beta)]], t$ are two adjacent entries in the stream, let $t', [[d, \wr(X, \alpha, \beta)]]$ be the result of applying the replacement rule (replacing

$t(X)$ with $\beta(t(X)))$. Furthermore, let $p = X\theta V$ be the selection predicate for tuples with description d before the Alter Data accent, and $p' = X\theta(\alpha(V))$ be the predicate after the accent due to the operational semantics of Select. Note that $t \rightarrow p' \Leftrightarrow t' \rightarrow p$. The tuple t survives the modified filter exactly when tuple t' survives the original filter.

Therefore, the Select operator satisfies the commutativity diagram of Figure 3 when considering the single replacement. Since this replacement is the only one that has any effect on the Select operator's operation, Select satisfies the commutativity diagram for full canonization by induction on replacements.

6 Declarative Operator Handling of Accents

The similarities and overlap in the various operators in Section 4 is conspicuous. Consider, for example, how similar the definitions of Union and Join are for operating on $\wr$ primitives in Section 4. There are patterns that we can abstract from the operator definitions about operator behavior based on the columns on which the description and the primitive in an accent are defined. This observation suggests that operator functionality can be generalized beyond specific operator internals. We claim that the accent handling and propagation characteristics of an operator can be associated with input attributes, i.e., given an operator O, one can encapsulate its behavior relative to accents via a declarative expression.

In many situations, accents propagate through a stream query operator without alteration. An operator may have three non-trivial behaviors in response to an accent: adjusting the description in an accent and forwarding it, coordinating accents from multiple inputs, or aborting because an accent is unsupportable. We call the set of all input attributes for a given query that share a particular response profile to accents an *attribute profile*. We now present the details of three attribute profiles. Each profile characterizes an operator's behavior in response to an accent referring to specific attributes in a stream, and also specifies a propagation contract that must be fulfilled based on canonical streams: Given an accent in an operator's canonical input, an attribute profile describes which accents, if any, must be present in the operator's canonical output.

6.1 Attribute Profiles

Foundation attributes (F): A Foundation attribute is an attribute that is necessary for a query operator. Foundation attributes must be present in the initial schema of the operator's input streams, and thus can never be added (because they are already present) or dropped (because doing so would eliminate data that is essential for the standing queries). We can describe the action of any operator O on accents whose evolution primitive references a foundation attribute as follows. If $\tilde{S}$ is the input stream to operator O:

– If the operator O receives an accent $[[d, -(X)]]$, regardless of description d, if $X \in F$, the operator (and the query) abort execution.
– If the operator O receives an accent $[[d, +(X)]]$, propagate (one need not check $X \in F$ since the attribute X must already exist).

Merge attributes (M): Merge attributes M for a query are input attributes to n-ary operators that are used by the operator to perform a comparison or computation. A Merge attribute requires the coordination of accents from multiple input streams to produce output. To simplify discussion, we assume that the operator with Merge attributes has two input streams $\tilde{S}_1$ and $\tilde{S}_2$. In terms of operator behavior, we can abstract away the behavior seen in previous operators — namely, Union and Join — that must coordinate data from multiple inputs.

For an operator with merge attributes M, we may describe its handling of accents as follows:

– $[[d, +(X)]]$: W.l.o.g., assume the accent is seen on input $\tilde{S}_1$. If an accent adding attribute $X \in M$ is not in input $\tilde{S}_2$'s state, add $[[d, +(X)]]$ to state of input $\tilde{S}_1$ and output $[[d, +(X)]]$ to $\tilde{S}_O$. If an accent $[[f, +(X)]]$ is in input $\tilde{S}_2$'s state, retrieve its description f, and:
  - Replace $[[f, +(X)]]$ with $[[f \wedge \neg d, +(X)]]$ in input $\tilde{S}_2$'s state — or remove it from state if $f \rightarrow d$
  - If $d \not\rightarrow f$, add $[[\neg f \wedge d, +(X)]]$ to $\tilde{S}_1$'s state and output $[[\neg f \wedge d, +(X)]]$ to output stream $\tilde{S}_O$.
– $[[d, -(X)]]$: Similarly to accents with the primitive $+(X)$, $X \in M$, Union maintains state, but does not propagate a drop attribute accent until it has been seen on both inputs.
– $[[d, \wr(X, \alpha, \beta)]]$: W.l.o.g., assume the accent is seen on input $\tilde{S}_1$ and no accent describing an Alter Data on $X \in M$ by α is in input $\tilde{S}_2$'s state. Any subsequent tuple t seen in input $\tilde{S}_1$ will have its X value replaced with $\beta(t(X))$ in $\tilde{S}_O$, and $[[d, \wr(X, \alpha, \beta)]]$ is added to input $\tilde{S}_1$'s state. If an accent $[[f, \wr(X, \alpha, \beta)]]$ is in input $\tilde{S}_2$'s state, retrieve its description f, and:
  - Replace $[[f, \wr(X, \alpha, \beta)]]$ with $[[f \wedge \neg d, \wr(X, \alpha, \beta)]]$ in input $\tilde{S}_2$'s state — or remove it if $f \rightarrow d$ — and stop altering t for tuples matching description f. This scenario is illustrated in Figure 2.
  - If $d \not\rightarrow f$, add $[[\neg f \wedge d, \wr(X, \alpha, \beta)]]$ to input $\tilde{S}_1$'s state and start altering tuples.
  - Output $[[f \wedge d, \wr(X, \alpha, \beta)]]$ to $\tilde{S}_O$.
  - Alter any tuples or partial computations currently in state if they satisfy description $f \wedge d$ by altering their X attribute by applying function α.

One detail that we have overlooked up to this point is how operator state interacts with incoming accents. For instance, consider the case when an accent $[[A < 10, -(X)]]$ arrives on input $\tilde{S}_1$ of a Union operator. We know from Section 4 that a Drop Attribute accent propagates through Union if its description matches an accent from the other input with the same description. However, if the operator had previously seen an accent $[[d, \wr(A, \alpha, \beta)]]$ on the same input, the question becomes: What does it mean for descriptions to "match"? The answer to this question is open and left to future work, though we anticipate this scenario to be rare in practice in that the attributes used in descriptions are likely to be invariants like unique identifiers or temporal attributes, both of whose semantics are unlikely to change over time.

In addition to these behaviors, we can formalize the propagation through an operator of an accent that refers to Merge attributes in terms of the canonical versions of operator input and output. For an operator with Merge attributes M, and with finite input streams $\tilde{S}_1$ and $\tilde{S}_2$ at some given time, with output stream $\tilde{S}_O$ at that time:

- $\forall_{[[d,+(X)]] \in \hat{S}_1}(X \in M \rightarrow \exists_{[[r,+(X)]] \in \hat{S}_o}(d \rightarrow r))$ (Add Attribute primitives are propagated through the operator, though the accent description may become less selective)
- $\forall_{[[d,-(X)]] \in \hat{S}_1}(X \in M \wedge \nexists_q([[q, -(X)]] \in \hat{S}_2)) \rightarrow \nexists_r([[r, -(X)]] \in \hat{S}_O)$ (if Drop Attribute primitives appear on merge attributes from one input but not the other, do not propagate)
- $\forall_{[[d,-(X)]] \in \hat{S}_1}(X \in M \wedge \exists_q([[q, -(X)]] \in \hat{S}_2)) \rightarrow [[d \wedge q, -(X)]] \in \hat{S}_O)$ (if Drop Attribute primitives appear on both inputs, produce an accent with that primitive on the output stream with a predicate at least as selective as the predicates on the input accents' descriptions)
- $\forall e$ (e is an Alter Data primitive), $\forall_{[[p,e]] \in \hat{S}_1} \forall_{[[d,e]] \in \hat{S}_1}(X \in M) \wedge \nexists_q([[q, e]] \in \hat{S}_2) \rightarrow \nexists_r([[r, e]] \in \hat{S}_O)$
- $\forall e$ (e is an Alter Data primitive), $\forall_{[[p,e]] \in \hat{S}_1} \forall_{[[d,e]] \in \hat{S}_1}(X \in M) \wedge \exists_q([[q, e]] \in \hat{S}_2) \rightarrow [[d \wedge q, e]] \in \hat{S}_O)$ (this property and the previous one state to propagate Alter Data primitives similarly to Drop Attribute primitives)

In brief, if an operator has merge attributes, then the operator always propagates Add Attribute primitives on Merge attributes but must receive the same Drop Attribute or Alter Data primitive on all inputs before propagating.

Accumulator attributes (A): An Accumulator attribute is any attribute that cannot participate in any accent description in an operator's output stream. An example is the project operator ($\pi_{\mathbf{X}}$); any attribute $Y \notin \mathbf{X}$ does not appear in the operator output. Accents with descriptions on an Accumulator attribute thus *accumulate* in the operator until they can be grouped together into accents without such description. For example, the descriptions $s = 1 \wedge t < 5$, $s = 1 \wedge t = 5$, and $s = 1 \wedge t > 5$ can be accumulated into a single description $s = 1$ that does not refer to attribute t. For an operator with Accumulator attributes A, input stream $\tilde{S}$, and output stream $\tilde{S}_O$:

– $[[d, +(X)]]$: Add $[[d, +(X)]]$ to operator state. Further:
  - Let d' be the description where, for all attributes $a \notin A$, $d'(a) = d(a)$, and for all attributes $a \in A$, $d'(a)$ is undefined. Description d' is the predicate that is the most selective that also only refers only to non-accumulator attributes and that $d \to d'$.
  - Let $\mathbf{D}$ be the set of descriptions that match an instance of primitive $+(X)$ already in state.
  - Let $d_0 = \bigvee_{x \in \mathbf{D}} x$, which is the predicate that results from unifying all of the descriptions for $+(X)$ in state.
  - If $d \to \neg d_0$, propagate $[[d', +(X)]]$ to output $\tilde{S}_O$. In other words, propagate the Add Attribute accent if it is the accent whose description overlaps d'. This property allows the Add Attribute accent to occur before all possible matching tuples for d' in the output stream.
  - If $d' \to d_0$, remove from state any accent $[[d'', +(X)]]$ where $d'' \to d'$, as the entirety of d' has been covered, and thus we no longer need those entries in state.

– $[[d, -(X)]]$: Add $[[d, -(X)]]$ to operator state. Further:
  - Let d' and d_0 be the same descriptions as in the Add Attribute case.
  - If $d_0 \vee d \equiv d'$, propagate $[[d', -(X)]]$ to output $\tilde{S}_O$. This property says that an operator waits to propagate the Drop Attribute accent until enough accents arrive to cover the entirety of d', ensuring that the accent appears after any tuple or accent that may refer to attribute X. Remove from state any accent $[[d'', -(X)]]$ where $d'' \to d'$.

– $[[d, \wr(X, \alpha, \beta)]]$: Add $[[d, \wr(X, \alpha, \beta)]]$ to operator state, and apply β to the X component of all matching tuples. Further:
  - Let d' and d_0 be the same descriptions as in the Add Attribute case.
  - If $d_0 \vee d \equiv d'$, propagate $[[d', \wr(X, \alpha, \beta)]]$ to output $\tilde{S}_O$. This property says that an operator waits to propagate the Alter Data accent until enough accents arrive to cover the entirety of d'. Remove from state any accent $[[d'', \wr(X, \alpha, \beta)]]$ where $d'' \to d'$. Stop altering tuples for matching tuples, and apply α to the X attribute of any matching tuples or partial computations that may be in the operator's state.

We also formalize the contract for propagating accents using canonical inputs and outputs. For an operator with Accumulator attributes A, and with finite input stream $\tilde{S}$ at some given time and output stream $\tilde{S}_O$ at that time:

- $\forall_{[[d,+(X)]] \in \hat{S}}((d$ refers to attributes in $A) \rightarrow \exists_q [[q,+(X)]] \in \hat{S}_O \wedge (q$ does not refer to attributes in $A) \wedge (d \rightarrow q))$ (Add Attribute primitives propagate through the operator, and the description may become less selective to avoid references to A).
- $(\exists_X \exists_{d_1,d_2,\ldots,d_n} \forall_{i \in \{1\ldots n\}} \exists_{[[d_i,-(X)]] \in \hat{S}}(d_i$ refers to attributes in A$) \wedge (X \in M)$ $\wedge (d = (\bigvee_{i \in \{1\ldots n\}} d_i)$ does not refer to A$)) \rightarrow ([[d,-(X)]] \in \hat{S}_O)$ (if the description attached to a Drop Attribute primitive refers to accumulator attributes, propagate when input description for a given primitive can be rolled up into a single predicate that does not refer to accumulator attributes).
- If $e = \wr(X, \alpha, \beta)$, $(\exists_{[[d,e]] \in \hat{S}} \exists_{d_1,\ldots,d_n} \forall_{i \in \{1\ldots n\}} \exists_{[[d_i,e]] \in \hat{S}}(d_i$ refers to attributes in A$) \wedge (X \in M)(p = (\bigvee_{i \in \{1\ldots n\}} d_i)$ does not refer to A$)) \rightarrow ([[p,e]] \in \hat{S}_O)$ (propagate Alter Data accents like Drop Attribute accents).

6.2 Operator Signatures

The *signature* of an operator is a description of the attribute profiles to which each input attribute belongs. One can specify the signature for an operator as a triple (F, M, A), signifying an operator's Foundation, Merge, and Accumulator attributes. Using a signature, one can encapsulate most — if not all — of an operator's behavior with regards to processing accents. The following list describes the six operators from Section 4 using signatures, followed by any behavior that is required by the operator but not part of the signature:

Select ($\sigma_{C\theta V}\tilde{S}$) has signature $(\{C\}, \emptyset, \emptyset)$, + adjust predicate as Alter Data primitives are seen on attribute C

Project ($\Pi_{\mathbf{C}}\tilde{S}$) has signature $(\emptyset, \emptyset, \mathcal{C} - \mathbf{C})$, + drop all accents whose primitive refers to attributes $\mathcal{C} - \mathbf{C}$ (where $\mathcal{C}$ is the set of all possible attributes)

Union ($\tilde{S} \cup \tilde{T}$) has signature $(\emptyset, \mathcal{C}, \emptyset)$

Join ($\tilde{S} \bowtie_{\mathcal{J}(C_{\tilde{S}}, C_{\tilde{T}})} \tilde{T}$) has signature $(C_{\tilde{S}} \cup C_{\tilde{T}}, C_{\tilde{S}} \cup C_{\tilde{T}}, \emptyset)$

Window ($\mathcal{W}^w_{\mathbf{C},wid}\tilde{S}$) has signature $(\mathbf{C}, \emptyset, \emptyset)$, + descriptions that refer to window attributes $\mathbf{C}$ are transformed into descriptions that refer to wid

Aggregate ($\gamma^f_{\mathbf{G},\mathbf{E}}S$) has signature $(\mathbf{G}, \emptyset, \mathcal{C} - \mathbf{G})$, + drop all accents whose primitive refers to attributes $\mathbf{E}$

One immediate benefit of operator signatures is declarative behavior specification. Note that the Union operator's behavior when interacting with accents is fully specified by signature $(\emptyset, \mathcal{C}, \emptyset)$. We can define the behavior and propagation contract of some other operators just as easily. For instance, the Intersect operator has signature $(\emptyset, \mathcal{C}, \emptyset)$ as well.

Another benefit of operator signatures is that they enable the possibility of global behavior settings. Note that the signature definitions of Project and Aggregate have different behavior than their definitions in Section 4. For instance, the project operator was defined assuming that if any part of an accent description d refers to a property that is projected away, then the query aborts. As defined using the signature $(\emptyset, \emptyset, \mathcal{C} - \mathbf{C})$, the project operator has different behavior, namely that such descriptions are accumulated until it can form a description that can propagate without referring to dropped attributes. Both of these definitions have their merits. The original aborting definition has the benefit that the project operator requires no state, with the drawback that fewer accents are supportable. On the other hand, the accumulation definition supports far more accents without bringing down the query, but requires the project operator to maintain state, where the operator did not require any state to operate on non-accented streams. Using signatures, this decision between statelessness and query resilience can be made once, globally, and all operators with a signature that includes accumulator attributes will follow suit.

7 Related Work

To our knowledge, there is no existing work modeling or implementing evolution in DSMSs, although schema evolution has been amply addressed in DBMSs [Rahm and Bernstein 2006]. Schema evolution research often focuses on mitigating the effect of evolutions on artifacts that rely on the schema, such as embedded SQL [Maule et al. 2008] or adjusting schema mappings to address new schemas [Yu and Popa 2005].

Extract-transform-load (ETL) workflows are similar to streaming queries; they are a composition of atomic data transformations (called *activities*) that determine data flow through a system. Unlike streaming queries, ETL workflows do not execute continuously and are typically not as resource-constrained as DSMSs. Papastefanatos et al. addressed schema evolution on an ETL workflow by attaching *policies* to each activity. Policies semi-automatically adjust the activity parameters based on schema evolution primitives that propagate through the activities [Papastefanatos 2007, Papastefanatos et al. 2008]. Unlike our approach, ETL research does not need to address how to maintain uptime of queries or the intermingling of schema evolution with data in the presence of accumulated state.

A final area of related work is schema mapping maintenance, where one "heals" a mapping between schemas S and T when either schema evolves (say, from T to T'), thus creating a new mapping (say, from S to T') that respects the semantics of the original mapping, or failing if impossible. Both-as-View (BAV) describes a schema mapping as a sequence of incremental steps [McBrien and Poulovassilis 2003]. Changes made to a schema become a second sequence of steps that is then composed with the original mapping sequence [McBrien and Poulovassilis 2002]. A similar approach is possible if a mapping is specified using source-to-target tuple-generating dependencies (st-tgds), both when the schema evolution can be reduced into discrete transformations [Velegrakis et al. 2003, Velegrakis et al. 2004] or itself specified as st-tgds [Yu and Popa 2005]. The Guava framework is another approach to mapping evolution [Terwilliger 2009]. A Guava mapping is expressed as a set of algebraic transformations, through which schema evolution primitives expressed against the source schema propagate to the target schema and update it as well.

8 Conclusions and Future Work

We have introduced semantics toward supporting schema evolution in stream systems by introducing the notion of an accented stream. Our contributions include adding markers to announce evolutions to operators in a standing query. Our work on stream query operators suggests we can extend the capability of conceptual models and the systems that implement them by expanding the scope of ordinary query operators to handle evolution. We detailed an initial set of evolutions and the effect they have on common stream operators. We provided a framework that allows stream engines to support schema evolution without bringing down standing queries whenever the evolutions do not render a query meaningless. Our approach supports simultaneous input streams at different stages in their respective evolutions.

Future work will characterize additional operators beyond the six introduced in this paper, and provide more precise definitions of what it means for a stream to have schema in the wake of evolution. Some operational considerations were left out in this paper, such as the effect of Alter Data primitives on other accents in state. Future work will address these issues in more detail, in particular in out-of-order architectures [Li et al. 2008], as well as the impact of accents on query performance and efficient operator implementation.

A key area of future work pertains to state management. We mentioned in Section 4 how the Union operator may remove an accent a from one input's state when accents with the same primitive covering a's description appear on the other input. However, there is already a well-established means of state management employed by many DSMSs, and emerging work related to guarantees regarding state [Fernández-Moctezuma et al. 2010]. In Section 1, we mentioned that various DSMSs have been designed to operate on *punctuated* data streams [Barga et al. 2007, Johnson et al. 2005, Li et al. 2008, Tucker 2005]. For example, consider the following data stream for the schema `s(a,b)`, where punctuations are denoted as descriptions in square brackets ("[]"):

(`<a:1, b:0>`,`<a:1, b:1>`,`[a:<=1]`,`<a:2, b:0>`)

Punctuated data streams follow an important property: no tuple described by a punctuation will be seen in the stream once the punctuation has been seen in the stream. In the example above, no tuple with a value less than or equal to 1 in a will be seen after that punctuation. This property is often exploited in a DSMS to cope with operator state. Several operators maintain information about tuples seen on input in case said information is needed for processing tuples that arrive later. For instance, a Join operator will hold onto tuples from an input indefinitely, just in case it might match a tuple from the other input at some point in the future. Such an operator can exploit punctuation to free state as it knows that no more tuples will arrive to match existing tuples in state.

Future work will extend our current model to operate on accented-punctuated streams, which are streams in which both accents and punctuations appear alongside tuples. This type of stream will use punctuations to track a canonical notion of progress, and accents to anticipate evolution. We plan to exploit punctuation to cleanse state associated with accents. The interaction between punctuations and accents will also be characterized because punctuations may refer to subsets of the stream for which an evolution has occurred.

Acknowledgements

This work is supported by CONACyT México (178258), OTREC, and NSF (0534762, 0612311). We thank Jonathan Goldstein, Badrish Chandramouli, and Kristin A. Tufte for their comments.

References

[Abiteboul et al. 1997] Abiteboul, S.; Quass, D.; McHugh, J.; Widom, J.; Wiener, J. L.: The Lorel Query Language for Semistructured Data. *Int. J. on Digital Libraries*, 1997, 1(1):68–88.

[Barga et al. 2007] Barga, R. S.; Goldstein, J.; Ali, M. H.; Hong, M.: Consistent Streaming Through Time: A Vision for Event Stream Processing. *CIDR 2007*, 363–374.

[Fernández-Moctezuma et al. 2009] Fernández-Moctezuma, R. J.; Terwilliger, J. F.; Delcambre, L. M. L.; Maier, D.: Towards Formal Semantics for Data and Schema Evolution in Data Stream Management Systems. *ETheCom 2009 (ER Workshops 2009)*, 85–94.

[Fernández-Moctezuma et al. 2010] Fernández-Moctezuma, R. J.; Maier, D.; Tufte, K. A.: Towards Execution Guarantees for Stream Queries. *Proceedings of the Third International Workshop on Scalable Stream Processing Systems (SSPS)*, 2010.

[Johnson et al. 2005] Johnson, T.; Muthukrishnan, S.; Shkapenyuk, V.; Spatscheck, O.: A Heartbeat Mechanism and its Application in Gigascope. *VLDB 2005*, 1079–1088.

[Li et al. 2008] Li, J.; Tufte, K.; Shkapenyuk, V.; Papadimos, V.; Johnson, T.; Maier, D.: Out-of-Order Processing: A New Architecture for High-Performance Stream Systems. *Proc. VLDB Endow.*, 2008, 1(1):274–288.

[Maule et al. 2008] Maule, A.; Emmerich, W.; Rosenblum, D. S.: Impact Analysis of Database Schema Changes. *ICSE 2008*, 451–460.

[McBrien and Poulovassilis 2002] McBrien, P.; Poulovassilis, A.: Schema Evolution in Heterogeneous Database Architectures, a Schema Transformation Approach. *CAiSE 2002*, 484–499.

[McBrien and Poulovassilis 2003] McBrien, P.; Poulovassilis, A.: Data Integration by Bi-Directional Schema Transformation rules. *ICDE 2003*, 227–238.

[Papastefanatos 2007] Papastefanatos, G.; Vassiliadis, P.; Simitsis, A.; Vassiliou, Y.: What-if Analysis for Data Warehouse Evolution. *DaWaK 2007*, 23–33.

[Papastefanatos et al. 2008] Papastefanatos, G.; Vassiliadis, P.; Simitsis, A.; Vassiliou, Y.: Design Metrics for Data Warehouse Evolution. *ER 2008*, 440–454.

[Rahm and Bernstein 2006] Rahm, E.; Bernstein, P. A.: An Online Bibliography on Schema Evolution. *SIGMOD Record*, 2006, 35(4):30–31.

[RDF 2004] W3C: Resource Description Framework (RDF). `http://www.w3.org/RDF/`. Publication date: February 10, 2004.

[Terwilliger 2009] Terwilliger, J. F.: *Graphical User Interfaces as Updatable Views*. PhD thesis, Portland State University, Portland, Oregon, USA, 2009.

[Tucker 2005] Tucker, P. A.: *Punctuated Data Streams*. PhD thesis, OGI School of Science & Technology at OHSU, Beaverton, Oregon, USA, 2005.

[Tucker et al. 2003] Tucker, P. A.; Maier, D.; Sheard, T.; Fegoras, L.: Exploiting Punctuation Semantics in Continuous Data Streams. *IEEE Transactions on Knowledge and Data Engineering*, 2003, 15(3):555–568.

[Velegrakis et al. 2003] Velegrakis, Y.; Miller, R. J.; Popa, L.: Mapping Adaptation Under Evolving Schemas. *VLDB 2003*, 584–595.

[Velegrakis et al. 2004] Velegrakis, Y.; Miller, R. J.; Popa, L.: Preserving Mapping Consistency Under Schema Changes. *VLDB J.*, 2004, 13(3):274–293.

[Yu and Popa 2005] Yu, C., Popa, L.: Semantic Adaptation of Schema Mappings When Schemas Evolve. *VLDB 2005*, 1006–1017.