Data-Trace Types for Distributed Stream Processing Systems

Konstantinos Mamouras
Rice University
mamouras@rice.edu

Caleb Stanford
University of Pennsylvania
castan@cis.upenn.edu

Rajeev Alur
University of Pennsylvania
alur@cis.upenn.edu

Zachary G. Ives
University of Pennsylvania
zives@cis.upenn.edu

Val Tannen
University of Pennsylvania
val@cis.upenn.edu

Abstract

Distributed architectures for efficient processing of streaming data are increasingly critical to modern information processing systems. The goal of this paper is to develop type-based programming abstractions that facilitate correct and efficient deployment of a logical specification of the desired computation on such architectures. In the proposed model, each communication link has an associated type specifying tagged data items along with a dependency relation over tags that captures the logical partial ordering constraints over data items. The semantics of a (distributed) stream processing system is then a function from input data traces to output data traces, where a data trace is an equivalence class of sequences of data items induced by the dependency relation. This *data-trace transduction* model generalizes both acyclic synchronous data-flow and relational query processors, and can specify computations over data streams with a rich variety of partial ordering and synchronization characteristics. We then describe a set of programming templates for data-trace transductions: abstractions corresponding to common stream processing tasks. Our system automatically maps these high-level programs to a given topology on the distributed implementation platform Apache Storm while preserving the semantics. Our experimental evaluation shows that (1) while automatic parallelization deployed by existing systems may not preserve semantics, particularly when the computation is sensitive to the ordering of data items, our programming abstractions allow a natural specification of the query that contains a mix of ordering constraints while guaranteeing correct deployment, and (2) the throughput of the automatically compiled distributed code is comparable to that of hand-crafted distributed implementations.

CCS Concepts • **Information systems** → **Data streams**; *Stream management*; • **Theory of computation** → *Streaming models*; • **Software and its engineering** → *General programming languages.*

ACM Reference Format:
Konstantinos Mamouras, Caleb Stanford, Rajeev Alur, Zachary G. Ives, and Val Tannen. 2019. Data-Trace Types for Distributed Stream Processing Systems. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19), June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3314221.3314580

1 Introduction

Modern information processing systems increasingly demand the ability to continuously process incoming data streams in a timely manner. Distributed stream processing architectures such as Apache Storm [26], Twitter's Heron [36, 52], Apache Spark Streaming [25, 54], Google's MillWheel [5], Apache Flink [18, 23] and Apache Samza [24, 44] provide platforms suitable for efficient deployment of such systems. The focus of the existing systems has been mainly on providing high throughput, load balancing, load shedding, fault tolerance and recovery. Less developed, however, is a *semantics* for streaming computations that enables one to reason formally about the *correctness* of implementations and distributed deployments with respect to a specification, even in the presence of disorder in the input. This is especially important because—as we will discuss in section 2—parallelization and distribution can cause spurious orderings of the data items, and it is therefore necessary to have a formal way of reasoning about these effects. The goal of this paper is to develop high-level abstractions for distributed stream processing by relying on a type discipline that is suitable for specifying computations and that can be the basis for correct and efficient deployment.

Physically, streams are linearly ordered, of course, and computations consume one item at a time. However, this is only one of many possible *logical* views of streaming data.

PLDI '19, June 22–26, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6712-7/19/06.
https://doi.org/10.1145/3314221.3314580

Indeed, assuming a strict linear order over input items is not the ideal abstraction for computation specification, for two reasons. First, in an actual implementation, there may be no meaningful logical way to impose a linear ordering among items arriving at different processing nodes. Second, for many computations it suffices to view the input logically as a *relation*, that is, a *bag* of unordered data items. Such lack of ordering often has computational benefits for optimization and/or parallelization of the implementation. Between linear ordering at one extreme and lack of ordering at the other we have the large space of *partial orders* and capturing these orders is the main focus of our type discipline.

We use *partially ordered multisets* (pomsets), a structure studied extensively in concurrency theory [45]. Pomsets generalize both sequences and bags, as well as sequences of bags, bags of sequences, etc., and we have found them sufficient and appropriate for our formal development. To specify the types that capture these partial orders as well as a logical type-consistent semantics for stream computations, we model—inspired by the definition of *Mazurkiewicz traces* [41] in concurrency theory—input and output streams as *data traces*. We assume that each data item consists of a *tag* and a value of a basic data type associated with this tag. The ordering of items is specified by a (symmetric) *dependency relation* over the set of tags. Two sequences of data items are considered equivalent if one can be obtained from the other by repeatedly commuting two adjacent items with independent tags, and a data trace is an equivalence class of such sequences. A *data-trace type* is given by a tag alphabet, a type of values for each tag, and a dependency relation.

For instance, when all the tags are mutually dependent, a sequence of items represents only itself, and when all the tags are mutually independent, a sequence of items represents the bag of items it contains. A suitable choice of tags along with the associated dependency relation, allows us to model streams with a rich variety of logical ordering and synchronization characteristics. As another example, consider a system that implements *key-based partitioning* by mapping a linearly ordered input sequence to a *set* of linearly ordered sub-streams, one per key. To model such a system the output items corresponding to distinct keys should be unordered. For this purpose, we allow the output items to have their own tags along with a dependency relation over these tags, and a sequence of outputs produced by the system is interpreted as the corresponding data trace. This representation can be easily presented programmatically and is also easily related to physical realizations.

While a system processes the input in a specific order by consuming items one by one in a streaming manner, it is required to interpret the input sequence as a data trace, that is, outputs produced while processing two equivalent input sequences should be equivalent. Formally, this means that a stream processor defines a *function from input data traces to output data traces*. Such a *data-trace transduction* is the proposed semantic model for distributed stream processing systems, and is a generalization of existing models in literature such as acyclic Kahn process networks [34, 37] and streaming extensions of database query languages [14, 38]. Our formal model is described in section 3.

In section 4 we propose a programming model where the overall computation is given as an acyclic dataflow graph, where every communication link is annotated with a *data-trace type* that specifies the ordering characteristics of the stream flowing through the link. In order to make the type annotation easier for the application developer, we restrict our framework to data-trace types that have two features: (1) the traces contain linearly ordered periodic *synchronization markers* for triggering the output of blocking operations and forcing progress (similar to the punctuations of [38] or the heartbeats of [47]), and (2) the data items of traces are viewed as key-value pairs in order to expose opportunities for key-based data parallelism. To ensure that each individual computational element of the dataflow graph respects the data-trace types of its input and output channels, we provide a set of *operator templates* for constraining the computation appropriately. For example, when the input data-trace type specifies that the items are unordered, their processing is described by a *commutative monoid* (a structure with an identity element and an associative, commutative binary operation), which guarantees that the output is independent of the order in which the items are processed.

When a programmer uses these typed abstractions to describe an application, she secures the global guarantee that the overall computation has a well-defined semantics as a data-trace transduction, and therefore its behavior is predictable and independent of any arbitrary data item interleaving that is imposed by the network or the distribution system (Theorem 4.2). Moreover, the operator templates allow for data parallelism that always preserves the semantics of the original specification (Theorem 4.3, Corollary 4.4).

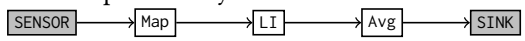
We have implemented data-trace types, operator templates and typed dataflow DAGs in Java as an embedded domain-specific language. Our system compiles the specification of the computation into a "topology" [27] that can be executed using Storm (see section 5). In section 6 we present an experimental evaluation where we address the following questions: (1) Is the code that our framework generates as efficient as a handcrafted implementation? (2) Does our framework facilitate the development of complex streaming applications? To answer the first question, we used a slight variant of the Yahoo Streaming Benchmark [33] and compared a generated implementation (that we built using our typed abstractions) against a handcrafted one. The experimental results show very similar performance. This provides evidence that our approach ***does not impose a computational overhead***, while offering guarantees of type correctness, predictable behavior, and preservation of semantics when data parallelism is introduced. To address the second

question, we consider a significant case study on prediction of power usage. This case study is inspired by the DEBS'14 Grand Challenge [20], which we adapted to incorporate a more realistic prediction technique based on machine learning. This application requires a mix of ordering constraints over data items. Our system automatically deals with low-level ordering and synchronization, so our programming effort was focused on the power prediction itself.

2 Motivation

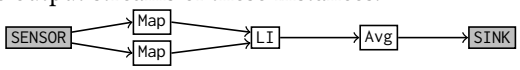
Many popular distributed stream processing systems—such as Storm [26], Heron [36, 52] and Samza [24, 44]—allow the programmer to express a streaming computation as a dataflow graph, where the processing performed by each node is described in a general-purpose language such as Java or Scala. During compilation and deployment, this dataflow graph is mapped to physical nodes and processes.

As a simple example, suppose that we want to process a stream of sensor measurements and calculate every 10 seconds the average of all measurements seen so far. We assume that the sensor generates the data items in increasing timestamp order, but the time series may have missing data points. The processing pipeline consists of three stages: (1) `Map` deserializes the incoming messages and retains only the scalar value and timestamp (i.e., discards any additional metadata), (2) LI performs linear interpolation to fill in the missing data points, and (3) Avg computes the average historical value and emits an update every 10 seconds.



The above pipeline can be programmed conveniently in Storm by providing the implementations of each node `Map`, `LI`, Avg and describing the connections between them.

The implementation described previously exposes pipeline parallelism, and thus suggests a multi-process or distributed execution where each stage of the pipeline computes as an independent process. In the case where the sensor produces messages at a very high rate, the computationally expensive deserialization stage `Map` becomes a bottleneck. In order to deal with such bottlenecks, Storm provides a facility for data parallelism by allowing the programmer to explicitly specify the creation of several parallel instances of the `Map` node. It handles automatically the splitting and balancing of the input stream across these instances, as well as the merging of the output streams of these instances.



The problem, however, with this data parallelization transformation is that it ***does not preserve the semantics*** of the original pipeline. The issue is that the linear interpolation stage `LI` relies on receiving the data elements in increasing order of timestamps. Unfortunately, when Storm merges the output streams of the two `Map` instances it introduces some arbitrary interleaving that may violate this precondition. This introduces nondeterminism to the system that causes the outputs to be unpredictable and therefore not reproducible without modifications to the computation.

Typically, a practical way to deal with these problems is to generate sequence numbers and attach them to stream elements in order to recover their order later (if they get out of order). However, this increases the size of data items. Moreover, it imposes a linear order, even in cases where a partial order is sufficient. For example, synchronization markers can be used to impose a partial order more efficiently than attaching sequence numbers. In general, many such practical fixes make the programs harder to debug, maintain, and modify correctly and thus less reliable.

In contrast, in order to facilitate ***semantically sound*** parallelization transformations and eliminate behaviors that rely on spurious ordering of the data items, our approach relies on *data-trace types* that classify the streams according to their partial ordering characteristics. For example, we can declare that the connection from `Map` to `LI` is "linearly ordered", and this would indicate that the parallelization transformation of the previous paragraph is not sound because it causes the reordering of data items flowing through that channel. Alternatively, the implementation `LI` could be replaced by a new implementation, denoted `Sort-LI`, that can handle a disordered input by sorting it first according to time-stamps. Then, the connection channel between `Map` and `Sort-LI` can be declared to be "unordered", which enables sound data parallelization for the `Map` stage. Assuming that all connections are typed, the problem now arises of whether the computation nodes are consistent with these input/output partial ordering types. We propose later in section 4 a way of structuring the code for each node according to a set of *templates*, so that it respects the types of its input/output channels.

3 Types for Data Streams

We will introduce a semantic framework for distributed stream processing systems, where the input and output streams are viewed as partially ordered [13]. Under this view, finite prefixes of streams are represented as *data traces*, and they are classified according to their ordering characteristics using *data-trace types*. The input/output behavior of a stream processing system is modeled as a *data-trace transduction*, which is a monotone function from input data traces to output data traces.

3.1 Data Traces

We use data traces to model streams in which the data items are partially ordered. Data traces generalize sequences (data items are linearly ordered), relations (data items are unordered), and independent stream channels (data items are organized as a collection of linearly ordered subsets). The

concatenation operation and the prefix order on sequences can be generalized naturally to the setting of data traces.

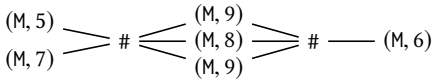
A ***data type*** $A = (\Sigma, (T_\sigma)_{\sigma\in\Sigma})$ consists of a potentially infinite *tag alphabet* Σ and a value type T_σ for every tag $\sigma \in \Sigma$. The set of *elements* of type A, or ***data items***, is equal to $\{(\sigma, d) \mid \sigma \in \Sigma \text{ and } d \in T_\sigma\}$, which we will also denote by A. The set of *sequences* over A is denoted as A^*. A ***dependence relation*** on a tag alphabet Σ is a symmetric binary relation on Σ. We say that the tags σ, τ are *independent* (w.r.t. a dependence relation D) if $(\sigma, \tau) \notin D$. For a data type $A = (\Sigma, (T_\sigma)_{\sigma\in\Sigma})$ and a dependence relation D on Σ, we define the dependence relation that is induced on A by D as $\{((\sigma, d), (\sigma', d')) \in A \times A \mid (\sigma, \sigma') \in D\}$, which we will also denote by D. Define $\equiv_D$ to be the smallest congruence (w.r.t. sequence concatenation) on A^* containing $\{(ab, ba) \in A^* \times A^* \mid (a, b) \notin D\}$. Informally, two sequences are equivalent w.r.t. $\equiv_D$ if one can be obtained from the other by repeatedly commuting adjacent items with independent tags.

Example 3.1. Suppose we want to process a stream that consists of sensor measurements and special symbols that indicate the end of a one-second interval. The data type for this input stream involves the tags $\Sigma = \{\mathtt{M}, \#\}$, where $\mathtt{M}$ indicates a sensor measurement and # is an end-of-second marker. The value sets for these tags are $T_\mathtt{M} = \mathtt{Nat}$ (natural numbers), and $T_\# = \mathtt{Ut}$ is the unit type (singleton). So, the data type $A = (\Sigma, T_\mathtt{M}, T_\#)$ contains measurements $(\mathtt{M}, d)$, where d is a natural number, and the end-of-second symbol #.

The dependence relation $D = \{(\mathtt{M}, \#), (\#, \mathtt{M}), (\#, \#)\}$ says that the tag $\mathtt{M}$ is independent of itself, and therefore consecutive $\mathtt{M}$-tagged items are considered unordered. For example, (M, 5) (M, 5) (M, 8) # (M, 9) and (M, 8) (M, 5) (M, 5) # (M, 9) are equivalent w.r.t. $\equiv_D$.

A ***data-trace type*** is a pair $X = (A, D)$, where A is a data type and D is a dependence relation on the tag alphabet of A. A ***data trace*** of type X is a congruence class of the relation $\equiv_D$. We also write X to denote the set of data traces of type X. Since the equivalence $\equiv_D$ is a congruence w.r.t. sequence concatenation, the operation of concatenation is also well-defined on data traces: $[u] \cdot [v] = [uv]$ for sequences u and v, where $[u]$ is the congruence class of u. We define the relation $\leq$ on the data traces of X as a generalization of the prefix partial order on sequences: for data traces $\mathbf{u}$ and $\mathbf{v}$ of type X, $\mathbf{u} \leq \mathbf{v}$ iff there are $u \in \mathbf{u}$ and $v \in \mathbf{v}$ s.t. $u \leq v$ (i.e., u is a prefix of v). The relation $\leq$ on data traces of a fixed type is a partial order. Since it generalizes the prefix order on sequences (when the congruence classes of $\equiv_D$ are singleton sets), we will call $\leq$ the *prefix order* on data traces.

Example 3.2 (Data Traces)**.** Consider the data-trace type $X = (A, D)$, where A and D are given in Example 3.1. A data trace of X can be represented as a sequence of multisets (bags) of natural numbers and visualized as a partial order on that multiset. The trace corresponding to the sequence of data items (M, 5) (M, 7) # (M, 9) (M, 8) (M, 9) # (M, 6) is visualized as:



where a line from left to right indicates that the item on the right must occur after the item on the left. The end-of-second markers # separate multisets of natural numbers. So, the set of data traces of X has an isomorphic representation as the set $\mathtt{Bag}(\mathtt{Nat})^+$ of nonempty sequences of multisets of natural numbers. In particular, the empty sequence ε is represented as $\emptyset$ and the single-element sequence # is represented as $\emptyset\ \emptyset$.

A singleton tag alphabet can be used to model sequences or multisets over a basic type of values. For the data type given by $\Sigma = \{\sigma\}$ and $T_\sigma = T$ there are two possible dependence relations for Σ, namely $\emptyset$ and $\{(\sigma, \sigma)\}$. The data traces of $(\Sigma, T, \emptyset)$ are multisets over T, which we denote as $\mathtt{Bag}(T)$, and the data traces of $(\Sigma, T, \{(\sigma, \sigma)\})$ are sequences over T.

Example 3.3 (Multiple Input and Output Channels)**.** Suppose we want to model a streaming system with multiple independent input and output channels, where the items within each channel are linearly ordered but the channels are completely independent. This is the setting of (acyclic) *Kahn Process Networks* [34] and the more restricted synchronous dataflow models [17, 37]. We introduce tags $\Sigma_\mathtt{I} = \{\mathtt{I}_1, \ldots, \mathtt{I}_m\}$ for m input channels, and tags $\Sigma_\mathtt{O} = \{\mathtt{O}_1, \ldots, \mathtt{O}_n\}$ for n output channels. The dependence relation for the input consists of all pairs $(\mathtt{I}_i, \mathtt{I}_i)$ with $i = 1, \ldots, m$. This means that for all indexes $i \neq j$ the tags $\mathtt{I}_i$ and $\mathtt{I}_j$ are independent. Similarly, the dependence relation for the output consists of all pairs $(\mathtt{O}_i, \mathtt{O}_i)$ with $i = 1, \ldots, n$. Assume that the value types associated with the input tags are $T_1, \ldots, T_m$, and the value types associated with the output tags are $U_1, \ldots, U_n$. The sets of input and output data traces are (up to a bijection) $T_1^* \times \cdots \times T_m^*$ and $U_1^* \times \cdots \times U_m^*$ respectively.

3.2 Data-String Transductions

In a *sequential implementation* of a stream processor the input is consumed in a sequential fashion, i.e. one item at a time, and the output items are produced in a specific linear order. Such sequential semantics is formally described by *data-string transductions*, which we use as a precursor to defining *data-trace transductions*.

Let A and B be data types. A ***data-string transduction*** with input type A and output type B is a function $f : A^* \to B^*$. A data-string transduction $f : A^* \to B^*$ describes a streaming computation where the input items arrive in a linear order. For an input sequence $u \in A^*$ the value $f(u)$ gives the output items that are emitted right after consuming the sequence u. In other words, $f(u)$ is the output that is triggered by the arrival of the last data item of u. We say that f is a *one-step* description of the computation because it gives the *output increment* that is emitted at every step.

The ***lifting*** of a data-string transduction $f : A^* \to B^*$ is the function $\bar{f} : A^* \to B^*$ that maps a sequence $a_1 a_2 \ldots a_n \in$

A^* to $\bar{f}(a_1a_2\dots a_n) = f(\varepsilon)\cdot f(a_1)\cdot f(a_1a_2)\cdots f(a_1a_2\dots a_n)$. In particular, the definition implies that $\bar{f}(\varepsilon) = f(\varepsilon)$. That is, $\bar{f}$ accumulates the outputs of f for all prefixes of the input. Notice that $\bar{f}$ is *monotone* w.r.t. the prefix order: $u \le v$ implies that $\bar{f}(u) \le \bar{f}(v)$ for all $u, v \in A^*$. The lifting $\bar{f}$ of a data-string transduction f describes a sequential streaming computation in a different but equivalent way. For an input sequence $u \in A^*$ the value $\bar{f}(u)$ is the *cumulative* output of the computation as the stream is extended item by item.

Example 3.4. Suppose the input is a sequence of natural numbers, and we want to define the transformation that outputs the current data item if it is strictly larger than all data items seen so far. We model this as a data-string transduction $f : \mathtt{Nat}^* \to \mathtt{Nat}^*$, given by $f(\varepsilon) = \varepsilon$ and

$$f(a_1\dots a_{n-1}a_n) = \begin{cases} a_n, & \text{if } a_n > a_i \text{ for all } i = 1,\dots,n-1; \\ \varepsilon, & \text{otherwise.} \end{cases}$$

The table below gives the values of f and $\bar{f}$ on input prefixes:

current item	input history	f output	$\bar{f}$ output
	ε	ε	ε
3	3	3	3
1	3 1	ε	3
5	3 1 5	5	3 5
2	3 1 5 2	ε	3 5

Notice that $\bar{f}(3\,1\,5\,2) = f(\varepsilon)\cdot f(3)\cdot f(3\,1)\cdot f(3\,1\,5)\cdot f(3\,1\,5\,2)$.

3.3 Data-Trace Transductions

Data-trace transductions are useful for giving the meaning (semantics) of a stream processing system. Consider the analogy with a functional model of computation: the meaning of a program consists of the input type, the output type, and a mapping that describes the input/output behavior of the program. Correspondingly, the semantics for a stream processing systems consists of: (1) the type X of input data traces, (2) the type Y of output data traces, and (3) a monotone mapping $\beta : X \to Y$ that specifies the cumulative output after having consumed a prefix of the input stream. The monotonicity requirement captures the idea that output items cannot be retracted after they have been omitted. Since β takes an entire input history (data trace) as input, it can model stateful systems, where the output that is emitted at every step depends potentially on the entire input history.

We have already discussed how (monotone) functions from A^* to B^* model sequential stream processors. We will now introduce the formal notion of *consistency*, which captures the intuition that a sequential implementation does not depend on the relative order of any two elements unless the stream type considers them to be relatively ordered.

Definition 3.5 (Consistency)**.** Let $X = (A, D)$ and $Y = (B, E)$ be data-trace types. We say that a data-string transduction $f : A^* \to B^*$ is (X, Y)-*consistent* if $u \equiv_D v$ implies that $\bar{f}(u) \equiv_E \bar{f}(v)$ for all $u, v \in A^*$.

Let $f \in A^* \to B^*$ be a (X, Y)-consistent data-string transduction. The function $\beta : X \to Y$, defined by $\beta([u]) = [\bar{f}(u)]$ for all $u \in A^*$, is called the (X, Y)-*denotation* of f.

Definition 3.6 (Data-Trace Transductions)**.** Let $X = (A, D)$ and $Y = (B, E)$ be data-trace types. A ***data-trace transduction*** with input type X and output type Y is a function $\beta : X \to Y$ that is monotone w.r.t. the prefix order on data traces: $\mathbf{u} \le \mathbf{v}$ implies that $\beta(\mathbf{u}) \le \beta(\mathbf{v})$ for all traces $\mathbf{u}, \mathbf{v} \in X$.

Definition 3.5 essentially says that a data-string transduction f is consistent when it gives equivalent cumulative outputs for equivalent input sequences. It is shown in [13] that the set of data-trace transductions from X to Y is equal to the set of (X, Y)-denotations of all (X, Y)-consistent data-string transductions.

Example 3.7 (Deterministic Merge)**.** Consider the streaming computation where two linearly ordered input channels are merged into one. More specifically, this transformation reads items cyclically from the two input channels and passes them unchanged to the output channel. Recall from Example 3.3 that the set of input data traces is essentially $T^* \times T^*$, and the set of output data traces is essentially T^*. The data-trace transduction merge : $T^* \times T^* \to T^*$ is given by:

$$\mathrm{merge}(x_1\dots x_m, y_1\dots y_n) = \begin{cases} x_1\ y_1\ \dots\ x_m\ y_m, & \text{if } m \le n; \\ x_1\ y_1\ \dots\ x_n\ y_n, & \text{if } m > n. \end{cases}$$

Example 3.8 (Key-Based Partitioning)**.** Consider the computation that maps a linearly ordered input sequence of data items of type T (each of which contains a key), to a set of linearly ordered sub-streams, one per key. The function key : $T \to K$ extracts the key from each input value. An input trace is represented as an element of T^*. The output type is specified by the tag alphabet K, value types $T_k = T$ for every key $k \in K$, and the dependence relation $\{(k, k) \mid k \in K\}$. So, an output trace is represented as a K-indexed tuple, that is, a function $K \to T^*$. The data-trace transduction $\mathrm{partition}_{\mathrm{key}} : T^* \to (K \to T^*)$ describes the partitioning of the input stream into sub-streams according to the key extraction map key: $\mathrm{partition}_{\mathrm{key}}(u)(k) = u|_k$ for all $u \in T^*$ and $k \in K$, where $u|_k$ denotes the subsequence of u that consists of all items whose key is equal to k. The implementation of $\mathrm{partition}_{\mathrm{key}}$ can be modeled as a data-string transduction $f : T^* \to (K \times T)^*$, given by $f(\varepsilon) = \varepsilon$ and $f(wx) = (\mathrm{key}(x), x)$ for all $w \in T^*$ and $x \in T$.

Although the computation of *aggregates* (e.g., sum, max, and min) is meaningful for unordered input data (i.e., a bag), if the bag is given as a stream then it is meaningless to produce partial aggregates as the data arrives: any partial aggregate depends on a particular linear order for the input items, which is inconsistent with the notion of unordered input. Therefore, for a computation of relational aggregates in the streaming setting we require that the input contains

linearly ordered *markers* that trigger the emission of output (see [38] for a generalization of this idea). The input can then be viewed as an ordered sequence of bags (each bag is delineated by markers), and it is meaningful to compute at every marker occurrence the aggregate over all items seen so far. Our definition of data-trace transductions captures these subtle aspects of streaming computation with relational data.

Example 3.9. Suppose that the input stream consists of unordered natural numbers and linearly ordered markers #. Consider the computation that emits at every occurence of # the maximum of all numbers seen so far. More specifically, the input type is given by $\Sigma = \{\sigma, \tau\}$, $T_\sigma = \mathtt{Nat}$, $T_\tau = \mathtt{Ut}$ (unit type), and $D = \{(\sigma,\tau),(\tau,\sigma),(\tau,\tau)\}$. So, an input data trace is essentially an element of $\mathrm{Bag}(\mathtt{Nat})^+$, as in Example 3.2. The streaming maximum computation is described by the data-trace transduction $\mathrm{smax} : \mathrm{Bag}(\mathtt{Nat})^+ \to \mathtt{Nat}^*$, where for a sequence of bags $B_1 \ldots B_n$, $\mathrm{smax}(B_1 \ldots B_n) := \max(B_1)\max(B_1 \cup B_2)\ldots\max(B_1 \cup B_2 \cup \cdots \cup B_{n-1})$. In particular, the output does not include the bag of items B_n since the last occurrence of #. The implementation of smax is modeled as a data-string transduction $f : (\mathtt{Nat} \cup \{\#\})^* \to \mathtt{Nat}^*$, which outputs at every # occurrence the maximum number so far. That is, $f(\varepsilon) = \varepsilon$ and

$$f(a_1 \ldots a_n) = \begin{cases} \varepsilon, & \text{if } a_n \in \mathtt{Nat}; \\ \max \text{ of } \{a_1, \ldots, a_n\} \setminus \{\#\}, & \text{if } a_n = \#. \end{cases}$$

for all sequences $a_1 a_2 \ldots a_n \in (\mathtt{Nat} \cup \{\#\})^*$.

4 Type-Consistent Programming

Complex streaming computations can be naturally described as *directed acyclic graphs* (DAGs), where the vertices represent simple operations and the edges represent streams of data. Such a representation explicitly exposes task and pipeline parallelism, and suggests a distributed implementation where every vertex is an independent process and inter-process communication is achieved via FIFO channels.

The semantic framework of section 3, which includes the notions of data-trace types and data-trace transductions, will serve a dual purpose. First, it will allow us to give a formal denotational semantics for streaming computation DAGs that respect the input/output stream types. Second, it will enable reasoning about equivalence and semantics-preserving transformations, such as data parallelization. We will focus here on a subset of data-trace types that emphasizes two crucial elements that are required by practical streaming computations: (1) a notion of *synchronization markers*, and (2) viewing the data items as *key-value pairs* in order to expose opportunities for data parallelization.

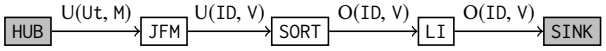
The synchronization markers can be thought of as events that are periodically generated by the input sources. The period is configurable and can be chosen by the application programmer depending on the time-granularity requirements of the computation (e.g. 1 msec, 1 sec, etc). The purpose of the markers is similar to the punctuations of [38] or the heartbeats of [47]. They are used for triggering the output of nonmonotonic operations (e.g., the streaming aggregation of Example 3.9) and making overall progress, as well as for merging streams in a predictable way by aligning them on corresponding markers. These synchronization markers are always assumed to be linearly ordered, and they occur in order of increasing timestamp.

We define two ***data-trace types for key-value pairs:*** *unordered* key-value pairs $\mathsf{U}(K,V)$, and *ordered* key-value pairs $\mathsf{O}(K,V)$. For a set of keys K and a set of values V, let $\mathsf{U}(K,V)$ denote the type with alphabet $K \cup \{\#\}$, values V for every key, values $\mathtt{Nat}$ for the # tag (i.e., marker timestamps), and dependence relation $\{(\#,\#)\} \cup \{(k,\#),(\#,k) \mid k \in K\}$. In other words, $\mathsf{U}(K,V)$ consists of data traces where the marker tags # are linearly ordered and the elements between two such tags are of the form (k,v), where $k \in K$ and $v \in V$, and are completely unordered. We define $\mathsf{O}(K,V)$ similarly, with the difference that the dependence relation also contains $\{(k,k) \mid k \in K\}$. That is, in a data trace of $\mathsf{O}(K,V)$, elements with the same key are linearly ordered between # markers, but there is no order across elements of different keys.

A ***transduction DAG*** is a tuple $(S, N, T, E, \to, \lambda)$ which represents a labelled directed acyclic graph, where: S is the set of *source vertices*, T is the set of *sink vertices*, N is the set of *processing vertices*, E is the set of *edges* (i.e., connections/channels), $\to$ is the *edge relation*, and λ is a *labelling function*. The function λ assigns: (1) a data-trace type to each edge, (2) a data-trace transduction to each processing vertex that respects the input/output types, and (3) names to the source/sink vertices. We require additionally that each source vertex has exactly one outgoing edge, and each sink vertex has exactly one incoming edge.

Next we define the ***denotational semantics of a transduction DAG*** G with source vertices $S_1, \ldots, S_m$ and sink vertices $T_1, \ldots, T_n$. Suppose that e_i is the unique edge emanating from the source vertex S_i (for $i = 1, \ldots, m$), and $\bar{e}_i$ is the unique edge leading to the sink vertex T_i (for $i = 1, \ldots, n$). The graph G denotes a data-trace transduction, where the set of input traces is (up to a bijection) $\prod_{i=1}^m \lambda(e_i)$ and the set of output traces is (up to a bijection) $\prod_{i=1}^n \lambda(\bar{e}_i)$. Given an input trace, we will describe how to obtain the output data trace (representing the entire output history of G on this input trace). We will gradually label every edge e of the DAG with a data trace $\mathbf{u}(e)$. First, label every edge emanating from a source vertex with the corresponding input trace. Then, consider in any order the processing vertices whose incoming edges have already been labeled. For such a vertex n, apply the data-trace transduction $\lambda(n)$ to the input traces and label the outgoing edges with the corresponding output traces. After this process ends, the output is read off from the data traces which label the edges that point to sinks.

Example 4.1 (Time-Series Interpolation)**.** Consider a home IoT system where temperature sensors are installed at a residence. We wish to analyze the sensor time series to create real-time notifications for excessive energy loss through the windows. The sensor time series sometimes have missing data points, and therefore the application requires a pre-processing step to fill in any missing measurements using linear interpolation. We assume that the sensors first send their measurements to a hub, and then the hub propagates them to the stream processing system. The stream that arrives from the hub does not guarantee that the measurements are sent in linear order (e.g., with respect to a timestamp field). Instead, it produces synchronization markers every 10 seconds with the guarantee that all elements with timestamps $< 10 \cdot i$ have been emitted by the time the i-th marker is emitted. That is, the i-th marker can be thought of as a watermark with timestamp $10 \cdot i$. The input stream is a data trace of U(Ut, M), where M is the type of measurements (*id*, *value*, *ts*) consisting of a sensor identifier *id*, a scalar value *value*, and a timestamp *ts*. This is a transduction DAG that describes the pre-processing computation:



The vertex HUB represents the source of sensor measurements, and the vertex SINK represents the destination of the output stream. ID is the type of sensor identifiers, and V is the type of timestamped values (*value*, *ts*). The processing vertexes are described below:

- The stage Join-Filter-Map (JFM) joins the input stream with a table that indicates the location of each sensor, filters out all sensors except for those that are close to windows, and reorganizes the fields of the input tuple.
- Recall the guarantee for the synchronization markers, and notice that it implies the following property for the input traces: for any two input measurements that are separated by at least one marker, the one on the left has a strictly smaller timestamp than the one on the right. The sorting stage SORT sorts for each sensor the measurements that are contained between markers.
- The linear interpolation stage LI considers each sensor independently and fills in any missing data points.

We have described informally the data-trace transductions JFM, SORT and LI. The transduction DAG shown before denotes a data-trace transduction $\mathrm{U}(\mathtt{Ut}, \mathtt{M}) \to \mathrm{O}(\mathtt{ID}, \mathtt{V})$.

The computation performed by a processing node is given in a structured fashion, by completing function definitions of a specified *operator template*. Table 1 shows the three templates that are supported, which encompass both ordered and unordered input streams. Each operator is defined by a sequential implementation, which we describe informally below. This means that each operator can be modeled as a data-string transduction. It can then be proved formally that these data-string transductions are consistent w.r.t. their input/output data-trace types (Definition 3.5). It follows that each operator that is programmed according to the template conventions has a denotation (semantics) as a data-trace transduction of the appropriate type.

Table 1. Operator templates for data-trace transductions.

```
U(K, V): unordered key-value pairs between markers
O(K, V): for every key, ordered values between markers
```

```
Type parameters: K, V, L, W
OpStateless: transduction U(K, V) → U(L, W)
Ut onItem(K key, V value) { }
Ut onMarker(Marker m) { }
```

```
Type parameters: K, V, W, S
OpKeyedOrdered: transduction O(K, V) → O(K, W)
S initialState() { }
S onItem(S state, K key, V value) { }
S onMarker(S state, K key, Marker m) { }
// Restriction: Output items preserve the input key.
```

```
Type parameters: K, V, L, W, S, A
OpKeyedUnordered: transduction U(K, V) → U(L, W)
A in(K key, V value) { }
A id() { } // identity for combine
A combine(A x, A y) { } // associative, commutative
S initialState() { }
S updateState(S oldState, A agg) { }
Ut onItem(S lastState, K key, V value) { }
Ut onMarker(S newState, K key, Marker m) { }
// Restriction: in, id, combine, initialState, and
// updateState are all pure functions.
```

OpStateless: The simplest template concerns *stateless* computations, where only the current input event—not the input history—determines the output. The programmer fills in two function definitions: (1) onItem for processing key-value pairs, and (2) onMarker for processing synchronization markers. The functions have no output (the output type is Ut, i.e. the unit type) and their only side-effect is emitting output key-value pairs to the output channel by invoking `emit(outputKey, outputValue)`.

OpKeyedOrdered: Assuming that the input is ordered per key, this template describes a stateful computation for each key independently that is order-dependent. The programmer fills in three function definitions: (1) initialState for obtaining the initial state, (2) onItem for processing a key-value pair and updating the state, and (3) onMarker for processing a synchronization marker and updating the state. The functions have output S, which is the type of the data structure for representing the state. As for stateless computations, the functions allow the side-effect of emitting output key-value pairs to the output channel. This template requires a crucial *restriction* for maintaining the order for the output: every occurrence of emit must preserve the input key. If this restriction is violated, e.g. by projecting out the key, then the output cannot be viewed as being ordered.

OpKeyedUnordered: Assuming that the input is unordered, this template describes a stateful computation for each key independently. Recall that the synchronization markers are ordered, but the key-value pairs between markers are *unordered*. To guarantee that the computation does not depend on some arbitrary linear ordering of the key-value pairs, their processing does not update the state. Instead, the key-value pairs between two consecutive markers are aggregated using

Table 2. Examples of data-trace transductions.

```
M = { id: ID, scalar: Float, ts: Int }
V = { scalar: Float, ts: Int }
joinFilterMap: OpStateless U(Ut, M) → U(ID, V)
Ut onItem(Ut key, M value) {
    if (location(value.id) = "window")
}       emit(value.id, (value.scalar, value.ts))
Ut onMarker(Marker m) { }
linearInterpolation: OpKeyedOrdered O(ID, V) → O(ID, V)
Precondition: items arrive in order of increasing timestamp
V initialState() { return nil }
V onItem(V state, ID key, V value) {
    if (state == nil) then // first element
        emit(key, value)
    else // not the first element
        Float x = state.scalar
        Int dt = value.ts - state.ts
        for i = 1 ... dt do
            Float y = x + i * (value.scalar - x) / dt
            emit(key, (y, state.ts + i))
}   return value
V onMarker(V state, ID key, Marker m) { return state }
maxOfAvgPerID: OpKeyedUnordered U(ID, V) → U(ID, V)
AvgPair = { sum: Float, count: Nat }
AvgPair in(ID key, V value) { return (value.scalar, 1) }
AvgPair id() { return (0.0, 0) }
AvgPair combine(AvgPair x, AvgPair y) {
}   return (x.sum + y.sum, x.count + y.count)
Float initialState() { return -infinity }
Float updateState(Float oldState, AvgPair agg) {
}   return max(oldState, agg.sum / agg.count)
Ut onItem(Float lastState, ID key, V value) { }
Ut onMarker(Float newState, ID key, Marker m) {
}   emit(key, (newState, m.timestamp - 1))
```

the operation of a *commutative monoid* A: the programmer specifies an identity element `id()`, and a binary operation `combine()` that must be *associative* and *commutative*. Whenever the next synchronization marker is seen, `updateState` is used to incorporate the aggregate (of type A) into the state (of type S) and then `onMarker` is invoked to (potentially) emit output. The behavior `onItem` may depend on the last snapshot of the state, i.e. the one that was formed at the last marker. The functions `onItem` and `onMarker` are allowed to emit output data items (but not markers), but the rest of the functions must be pure (i.e., no side-effects).

Table 2 shows how some streaming computations (which are based on the setting of Example 4.1) can be programmed using the operator templates of Table 1. The first example is the stateless computation `joinFilterMap`, which retains the measurements of temperature sensors that are placed near windows. The second example is the per-sensor ordered stateful computation `linearInterpolation`, which fills in the missing data points of a sensor time series by performing linear interpolation. The last example is the per-sensor unordered (between markers) stateful computation that takes the average of the measurements between markers and reports the maximum over all the averages so far.

Theorem 4.2. Every streaming computation defined using the operator templates of Table 1 is consistent w.r.t. its input/output type (see Definition 3.5).

Table 3. Implementation of `OpKeyedUnordered`.

```
R = { agg: A, state: S } // record type
Map<K,R> stateMap = ∅ // state map
S startS = initialState() // state when key is first seen
next(K key, V value) { // process data item
    R r = stateMap.get(key)
    if (r == nil) then // first time key is seen
        r = { agg = id(), state = startS }
        onItem(r.state, key, value)
    r.agg = combine(r.agg, in(key, value))
    stateMap.update(key, r)
}
next(Marker m) { // process marker
    for each (key, r) in stateMap do:
        r.state = updateState(r.state, r.agg)
        r.agg = id()
        stateMap.update(key, r)
        onMarker(r.state, key, m)
    startS = updateState(startS, id())
    emit(m)
}
```

Proof. We will prove the case of the `OpKeyedUnordered` template, since it is the most interesting one, and we will omit the rest. A template `OpKeyedUnordered<K, V, L, W, S, A>` describes a data-string transduction $f : A^* \to B^*$, where:

$$A = (K \times V) \cup (\{\#\} \times \mathtt{Nat}) \quad B = (L \times W) \cup (\{\#\} \times \mathtt{Nat})$$

This data-string transduction was informally described earlier and is defined operationally by the pseudocode shown in Table 3. The streaming algorithm of Table 3 maintains a per-key store and also tracks the state that should be given to keys that have not been encountered yet.

We write M for the memory of the streaming algorithm of Table 3. The function $next : M \times A \to M$ describes how the algorithm updates its memory every time it consumes an element. We also write $next : M \times A^* \to M$ to denote the function that describes how the algorithm updates the memory after consuming a sequence of elements. If $a_1, a_2 \in A$ are key-value pairs, then we have $a_1 a_2 \equiv a_2 a_1$. It is easy to see that $next(m, a_1 a_2) = next(m, a_2 a_1)$ for every $m \in M$. If the items a_1 and a_2 have the same key, then the property holds because of the associativity and commutativity of `combine`. If the items a_1 and a_2 have different keys, then the property holds because different keys cause the modification of disjoint parts of the memory. It follows by an inductive argument (on the construction of $\equiv$) that $next(m, u) = next(m, v)$ for all $m \in M$ and $u, v \in A^*$ with $u \equiv v$.

Suppose now that $out : M \times A \to B^*$ gives the output generated by the algorithm when it consumes a single element. We lift this function to $out : M \times A^* \to B^*$ as follows: $out(m, \varepsilon) = \varepsilon$ and $out(m, ua) = out(m, u) \cdot out(next(m, u), a)$ for all $m \in M$, $u \in A^*$ and $a \in A$. The crucial observation is that for every key-value item $(k, v) \in (K \times V)$, the value $out(m, (k, v))$ depends only on the part of memory that holds the state for k, which we denote by $m[k].\mathtt{state}$. Moreover, this part of the memory does not get modified when key-value pairs are processed. For memories $m_1, m_1 \in M$ and key k, we write $m_1 \equiv_k m_2$ when $m_1[k].\mathtt{state} = m_2[k].\mathtt{state}$. Our previous two observations can now be written formally as $m_1 \equiv_k m_2 \Rightarrow out(m_1, (k, v)) = out(m_2, (k, v))$ and $m \equiv_{k'}$

$next(m,(k,v))$ for all $m, m_1, m_2 \in M$, all $k, k' \in K$, and every $v \in V$. For key-value pairs $a_1, a_2 \in (K \times V)$ we have that

$$out(m, a_1a_2) = out(m, a_1) \cdot out(next(m, a_1), a_2) \in (L \times W)^*$$
$$out(m, a_2a_1) = out(m, a_2) \cdot out(next(m, a_2), a_1) \in (L \times W)^*$$

and by virtue of the properties discussed previously we obtain that $out(m, a_1a_2) \equiv out(m, a_2a_1)$. By an inductive argument on the construction of $\equiv$, we can generalize this property to: $out(m, u) \equiv out(m, v)$ for every memory $m \in M$ and all sequences $u, v \in A^*$ with $u \equiv v$.

In order to establish the consistency property we have to show that: $u \equiv v$ implies $\bar{f}(u) \equiv \bar{f}(v)$ for all $u, v \in A^*$. We have that $\bar{f}(u) = out(m_0, u)$, where m_0 is the initial memory for the algorithm. From $u \equiv v$ and earlier results we conclude that $\bar{f}(u) = out(m_0, u) \equiv out(m_0, v) = \bar{f}(v)$. □

The templates of Table 1 define data-trace transductions with only one input channel and output channel. The operation *merge*, which we denote by MRG or M, combines several input streams into one by aligning them on synchronization markers and taking the union of the key-value pairs that are in corresponding blocks. We consider two variants of merge, which we will not distinguish notationally. The first one has unordered input channels with the same input keys and values, i.e. $\mathtt{MRG} : \mathsf{U}(K,V) \times \cdots \times \mathsf{U}(K,V) \to \mathsf{U}(K,V)$. The second variant of merge has ordered input channels with pairwise disjoint sets of input keys $K_1, K_2, \ldots, K_n$, so we write $\mathtt{MRG} : \mathsf{O}(K_1,V) \times \cdots \times \mathsf{O}(K_n,V) \to \mathsf{O}(K_1 \cup \cdots \cup K_n, V)$.

To enable parallelization, we also need to consider operations that split one input stream into several output streams. The *round-robin* splitter, denoted $\mathtt{RR} : \mathsf{U}(K,V) \to \mathsf{U}(K,V) \times \cdots \times \mathsf{U}(K,V)$, sends every input key-value pair to one output channel by cycling through them and sends a synchronization marker to *all* output channels. The *hash-n* splitter, denoted HASH or $\mathtt{H} : \mathsf{U}(K,V) \to \mathsf{U}(K_{0/n},V) \times \cdots \times \mathsf{U}(K_{n-1/n},V)$ sends a key-value pair (k,v) with $k \in K_{i/n}$ with the i-th output channel where $K_{i/n} = \{k \in K \mid hash(k) = i \pmod n\}$. We write K_i instead of $K_{i/n}$ when no confusion arises. As for the round-robin splitter, H sends a synchronization marker to *all* output channels. The ordered version $\mathtt{H} : \mathsf{O}(K,V) \to \mathsf{O}(K_0,V) \times \cdots \times \mathsf{O}(K_{n-1},V)$ behaves similarly.

In order to convert an unordered trace of type $\mathsf{U}(K,V)$ to an ordered trace of $\mathsf{O}(K,V)$, we also consider the *sorting* data-trace transduction $\mathtt{SORT}^< : \mathsf{U}(K,V) \to \mathsf{O}(K,V)$. The transformation $\mathtt{SORT}^<$ uses the linear order $<$ to impose a total order for every key separately on the key-value pairs between synchronization markers. Even when the stream source is ordered, the parallelization of intermediate processing stages can reorder the key-value pairs between markers in an arbitrary way. So, if a later stage of the processing requires the original ordered view of the data, $\mathtt{SORT}^<$ must be applied immediately prior to that stage.

The templates of Table 1 not only enforce the data-trace type discipline on the input and output channels, but they also expose explicitly opportunities for parallelization and distribution. Computations that are described by the templates `OpKeyedOrdered` and `OpKeyedUnordered` can be parallelized on the basis of keys, and stateless computations can be parallelized arbitrarily.

Theorem 4.3 (Semantics-Preserving Parallelization)**.** Let $\beta : \mathsf{U}(K,V) \to \mathsf{U}(L,W)$, $\gamma : \mathsf{O}(K,V) \to \mathsf{O}(K,W)$, and $\delta : \mathsf{U}(K,V) \to \mathsf{U}(L,W)$ be data-trace transductions that are implemented using the `OpStateless`, `OpKeyedOrdered`, and `OpKeyedUnordered` templates, respectively. Then, we have:

$$\begin{aligned}
\mathtt{MRG} \gg \beta &= (\beta \parallel \cdots \parallel \beta) \gg \mathtt{MRG} \\
\gamma &= \mathtt{HASH} \gg (\gamma \parallel \cdots \parallel \gamma) \gg \mathtt{MRG} \\
\delta &= \mathtt{HASH} \gg (\delta \parallel \cdots \parallel \delta) \gg \mathtt{MRG} \\
\mathtt{SORT} &= \mathtt{HASH} \gg (\mathtt{SORT} \parallel \cdots \parallel \mathtt{SORT}) \gg \mathtt{MRG}
\end{aligned}$$

where $\gg$ denotes streaming composition and $\parallel$ denotes parallel composition [13]:

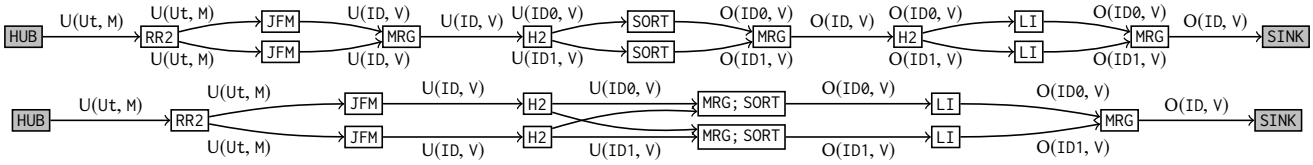
$$\frac{f : X \to Y \quad g : Y \to Z}{f \gg g : X \to Z} \qquad \frac{f : X \to Y \quad g : Z \to W}{f \parallel g : X \times Z \to Y \times W}$$

Proof. First, we observe that all the considered data-trace transductions are well-typed by Theorem 4.2. We will only give the proof for the equation involving β, since the other cases are handled similarly. For simplicity, we ignore the timestamps of the # markers. We can view the traces of $\mathsf{U}(K,V)$ as nonempty sequences of bags of elements of $K \times V$ (recall Example 3.2), i.e. $\mathsf{U}(K,V) = \mathsf{Bag}(K \times V)^+$. Since β is implemented by the template `OpStateless`, there is a function $out : (K \times V) \to \mathsf{Bag}(L \times W)$ that gives the output of β when it processes a single key-value element. Then, we have $\beta(B) = \bigcup_{(k,v) \in B} out(k,v)$ and $\beta(B_1B_2 \ldots B_n) = \beta(B_1)\beta(B_2) \ldots \beta(B_n)$ for all $B, B_1, \ldots, B_n \in \mathsf{Bag}(K \times V)$. Assuming we have m input channels, we obtain:

$$\begin{aligned}
&((\beta \parallel \cdots \parallel \beta) \gg \mathtt{MRG})(B_{11} \ldots B_{1n}, \ldots, B_{m1} \ldots B_{mn}) \\
&= \mathtt{MRG}(\beta(B_{11} \ldots B_{1n}), \ldots, \beta(B_{m1} \ldots B_{mn})) \\
&= \mathtt{MRG}(\beta(B_{11}) \ldots \beta(B_{1n}), \ldots, \beta(B_{m1}) \ldots \beta(B_{mn})) \\
&= (\beta(B_{11}) \cup \cdots \cup \beta(B_{m1})) \ldots (\beta(B_{1n}) \cup \cdots \cup \beta(B_{mn})) \\
&= \beta(B_{11} \cup \cdots \cup B_{m1}) \ldots \beta(B_{1n} \cup \cdots \cup B_{mn}) \\
&= \beta((B_{11} \cup \cdots \cup B_{m1}) \ldots (B_{1n} \cup \cdots \cup B_{mn})) \\
&= (\mathtt{MRG} \gg \beta)(B_{11} \ldots B_{1n}, \ldots, B_{m1} \ldots B_{mn})
\end{aligned}$$

using elementary properties of β and of MRG. So, we conclude that $\mathtt{MRG} \gg \beta = (\beta \parallel \cdots \parallel \beta) \gg \mathtt{MRG}$. □

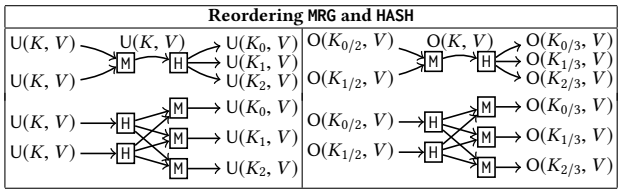
We say that a data-trace transduction $f : \mathsf{U}(K,V) \to \mathsf{U}(K,V) \times \cdots \times \mathsf{U}(K,V)$ is a *splitter* if $f \gg \mathtt{MRG} : \mathsf{U}(K,V) \to \mathsf{U}(K,V)$ is the identity function on data traces (identity transduction). Informally, a splitter splits (partitions) the input stream into several output streams. The data-trace transductions RR and H (defined earlier) are splitters. If SPLIT is a splitter, then Theorem 4.3 implies that for stateless β, $\beta = \mathtt{SPLIT} \gg (\beta \parallel \cdots \parallel \beta) \gg \mathtt{MRG}$.



RR2 (resp., H2) Partitions the input stream into two substreams in a round-robin fashion (resp., based on the hash value of the key).
ID0 (resp., ID1) The subset of identifiers in ID whose hash value is equal to 0 (resp., 1) modulo 2.

Figure 1. A transduction DAG that is equivalent to the one of Example 4.1 and allows data parallelism.

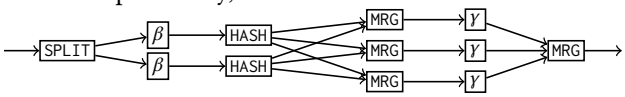
The processing pipeline for the sensor input stream of Example 4.1 can be parallelized. Figure 1 shows two equivalent processing graphs, where every stage of the pipeline is instantiated two times. The input for the JFM stage is partitioned in a round-robin fashion, and the input for the SORT and LI stages is partitioned based on the key (sensor identifier). All the vertices of the graph have a formal denotational semantics as data-trace transductions (Theorem 4.2), which enables a rigorous proof of equivalence for the DAGs of Example 4.1 and Figure 1. The top graph of Figure 1 is obtained from the graph of Example 4.1 by applying the parallelizing transformation rules of Theorem 4.3. The bottom graph of Figure 1 is obtained from the top one using the transformation rules of the following table:



Each box above shows two equivalent transduction DAGs. These rules are specialized to two input channels and three output channels for the sake of easy visualization. They extend in the obvious way to an arbitrary number of input and output channels. The bottom graph of the figure is equivalent to the identity transduction when the number of output channels is equal to the number of input channels, because $\mathtt{HASH}_n : \mathrm{O}(K_{i/n}, V) \to \mathrm{O}(K_{0/n}, V) \times \cdots \times \mathrm{O}(K_{n-1/n}, V)$ sends the entire input stream to the i-th output channel.

Corollary 4.4 (Correctness of Deployment)**.** Let G be a transduction DAG that is built using the operator templates of Table 1. Any deployment of G, regardless of the degree of parallelization, is equivalent to G.

Proof. The idea of the proof is that every deployment can be obtained from the original description G of the computation by applying a sequence of semantics-preserving transformation rules on specific subgraphs. This requires examining several cases. We will limit this proof to one case that illustrates the proof technique, and we will omit the rest since they can be handled with very similar arguments. First of all, we observe that the original graph G (and every graph obtained via transformations) has a denotational semantics in terms of data-trace transductions (Theorem 4.2). Let us examine the case of a subgraph of the form $\beta \gg \gamma$, where β is programmed using `OpStateless` and γ is programmed using `OpKeyedUnordered`. Let `SPLIT` be an arbitrary splitter, which means that $\mathtt{SPLIT} \gg \mathtt{MRG}$ is the identity transduction. Using the equations of Theorem 4.3 we can obtain the equivalent $\mathtt{SPLIT} \gg (\beta \parallel \beta) \gg \mathtt{MRG} \gg \mathtt{HASH} \gg (\gamma \parallel \gamma \parallel \gamma) \gg \mathtt{MRG}$. Using the transformation rules for "reordering MRG and HASH" mentioned previously, we obtain:



Finally, each subgraph $\beta \gg \mathtt{HASH}$ is fused into a single node $\beta;\mathtt{HASH}$, and similarly each subgraph $\mathtt{MRG} \gg \gamma$ is fused into $\mathtt{MRG};\gamma$. These fusion transformations can be easily checked to be semantics-preserving. □

5 Implementation in Apache Storm

In the previous section we proposed an abstraction for describing a distributed streaming computation as a *transduction DAG*, where each processing element is programmed using one of three predefined *templates*. This principled manner of defining computations enforces a data-trace type discipline that disallows operations which depend on a spurious ordering of the data items. Additionally, it enables a number of equivalence-preserving parallelization transformations.

We have implemented a compilation procedure that converts a transduction DAG into a deployment plan for the distributed streaming framework Storm [26]. In Storm, a computation is structured as a DAG (called *topology*) of source vertices called *spouts* and processing/sink vertices called *bolts*. Each vertex (bolt or spout) may be instantiated multiple times, across different physical nodes or CPU threads. In such settings, the connections between vertices specify a data partitioning strategy, which is employed when the vertices are instantiated multiple times. These connections are called *groupings* in Storm's terminology, and the most useful ones are: (1) *shuffle grouping*, which randomly partitions the stream in balanced substreams, (2) *fields grouping*, which partitions the stream on the basis of a key, and (3) *global grouping*, which sends the entire stream to exactly one instance of the target bolt. We refer the reader to [27] for more information on the programming model of Storm.

Figure 2 shows a concrete example of ***programming a transduction DAG*** (and thus obtaining a Storm topology)

```
// Source: input stream given by iterator
Iterator<Event<Int, Float>> iterator = new Stream();
Source<Int, Float> source = new Source<>(iterator);
// Processing node 1: filter out the odd keys
Operator<Int, Float, Int, Float> filterOp =
  new OpStateless<Int, Float, Int, Float>() {
    void onItem(KV<Int, Float> item) {
      if (item.key % 2 == 0) this.emit(item); }
    void onMarker(Marker<Int, Float> m) { } };
// Processing node 2: sum per time unit
Operator<Int, Float, Int, Float> sumOp =
  new OpKeyedUnordered<Int, Float, Int, Float, Float, Float>() {
    Float id() { return 0.0; }
    Float in(KV<Int, Float> item) { return item.value; }
    Float combine(Float x, Float y) { return x + y; }
    Float initialState() { return Float.NaN; }
    Float stateUpdate(Float state, Float agg) { return agg; }
    void onItem(Float lastState, KV<Int, Float> item) { }
    void onMarker(Float state, Int key, Marker<Int, Float> m) {
      this.emit(new KV<>(key, state, m.timestamp - 1)); } };
// Sink: prints the output stream
Sink<Int, Float> printer = Sink.defaultPrinter();
// Setting up the transduction DAG
DAG dag = new DAG();
dag.addSource(source);
int par1 = 2; // parallelism hint for filterOp
dag.addOp(filterOp, par1, source); // source ==> filterOp
int par2 = 3; // parallelism hint for sumOp
dag.addOp(sumOp, par2, filterOp); // filterOp ==> sumOp
dag.addSink(printer, sumOp); // sumOp ==> printer
// Check type consistency & create the topology for Storm
StormTopology topology = dag.getStormTopology();
// ... execute Storm topology
```

Figure 2. An extended programming example.

using our framework. In this example, the user first describes the data source using an `Iterator` object and converts it to a `Source` vertex. Then, the operator vertices are described using the templates `OpStateless` and `OpKeyedUnordered`. A transduction DAG is represented as a `DAG` object, which exposes methods for adding new vertices and edges. For example, the method call `dag.addOp(op, par, v1, v2, ...)` adds the vertex `op` to `dag` with the parallelism hint `par`, and it also adds edges from the vertices `v1`, `v2`, ... to `op`. Finally, `dag.getStormTopology()` performs all necessary checks for type consistency and returns a `StormTopology` object that can be passed to Storm for deployment on the cluster.

Our framework ensures that the data-trace types of input, output and intermediate streams are respected. The compilation procedure automatically constructs the glue code for propagating synchronization markers throughout the computation, merging input channels, partitioning output channels, and sorting input channels to enforce a per-key total order on the elements between markers. We use Storm's built-in facilities for the parallelization of individual processing vertices, but we have replaced Storm's "groupings" because they inhibit the propagation of the synchronization markers. For efficiency reasons, we fuse the merging operator (`MRG`) and the sorting operator (`SORT`) with the operator that follows them in order to eliminate unnecessary communication delays.

We chose Storm as the deployment platform because (1) it is a widely adopted "pure streaming" system that is used for many industry workloads, (2) it naturally exposes parallelism and distribution, and (3) it is extensible. Due to its similarity to alternative systems, it would not be difficult to compile transduction DAGs into topologies for these other platforms.

6 Experimental Evaluation

In this section we experimentally evaluate our data-trace type-based framework. We address two questions:

- Can our system generate code that is as efficient as a hand-crafted implementation, while automatically adapting to whatever levels of parallelism are available?
- Does our framework facilitate the development of complex streaming applications?

To answer the first question, we used an extension of the Yahoo Streaming Benchmark [19]. We compared an implementation generated using our framework against a hand-tuned one. To address the second question, we consider a significant case study: the Smart Homes Benchmark [20] used in the Grand Challenge of the DEBS 2014 conference, which we have modified to include a more realistic power prediction technique based on a machine learning model.

Our focus in the experiments is to determine how well stream applications scale. To do this, we used the following ***experimental setup***: We ran our implementation on top of Storm on a cluster of several virtual machines. Each virtual machine has 2 CPUs, 8 GB of memory, and 8 GB of disk each and runs CentOS 7. Across multiple trials and configurations, we measured maximum throughput for each configuration.

Yahoo Streaming Benchmark. The streaming benchmark of Yahoo [19] defines a stream of events that concern the interaction of users with advertisements, and suggests an analytics pipeline to process the stream. There is a fixed set of campaigns and a set of advertisements, where each ad belongs to exactly one campaign. The map from ads to campaigns is stored in a database. Each element of the stream is of the form (`userId`, `pageId`, `adId`, `eventType`, `eventTime`), and it records the interaction of a user with an advertisement, where `eventType` is one of {`view`, `click`, `purchase`}. The component `eventTime` is the timestamp of the event.

The basic benchmark query (as described in [19]) computes, at the end of each second, a map from each campaign to the number of views associated with that campaign within the last 10 seconds. For each event tuple, this involves an expensive database lookup to determine the campaign associated with the advertisement viewed. The reference implementation published with the Yahoo benchmark involves a multi-stage pipeline: (i) *stage 1*: filter view events, project the ad id from each view tuple, and lookup the campaign id of each ad, (ii) *stage 2*: compute for every window the number of events (views) associated with each campaign. The query involes key-based partitioning on only one property, namely the derived campaign id of the event.

To compare the effectiveness of our framework, we next re-implemented this analytics pipeline as a transduction

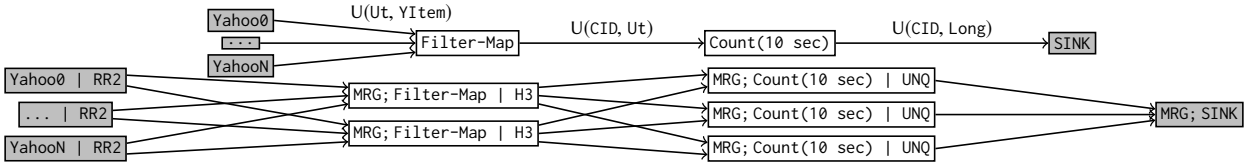


Figure 3. Query IV: Transduction DAG for a variant of the Yahoo Streaming Benchmark [19], and its deployment on Storm with parallelization 2 and 3 for the processing vertices `Filter-Map` and `Count(10 sec)` respectively.

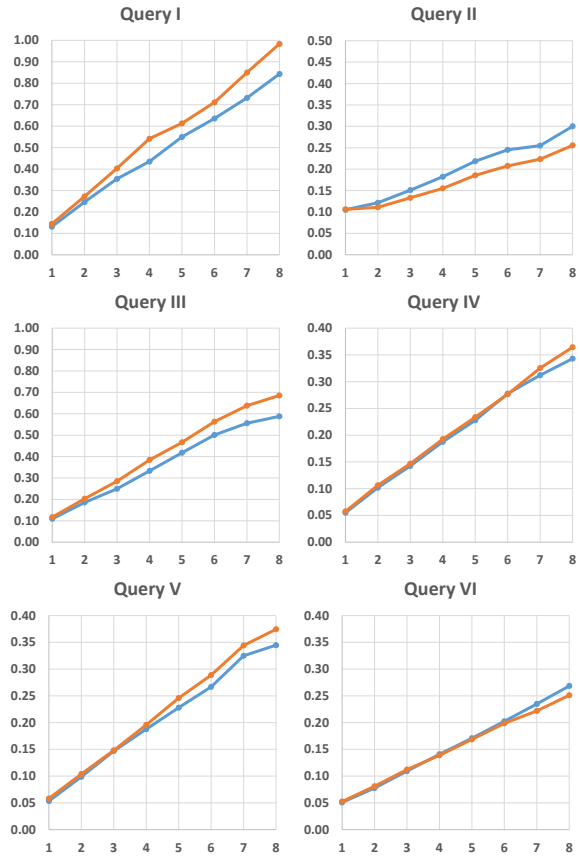


Figure 4. Queries inspired by the Yahoo Streaming Benchmark. The orange (resp., blue) line shows the throughput of the transduction-based (resp., handcrafted) implementation. The horizontal (resp., vertical) axis shows the number of machines (resp., throughput in million tuples/sec).

DAG, where every processing vertex is programmed using a template of Table 1. This is shown in the top graph of Figure 3, where `YItem` is the type of input tuples and `CID` is the type of campaign identifiers. The system is configured so that the stream sources emit synchronization markers at 1 second intervals, i.e. exactly when the timestamps of the tuples cross 1 second boundaries. To evaluate our framework more comprehensively, we have implemented six queries:

- **Query I**: A single-stage stateless computation that enriches the input data items with information from a database (we use Apache Derby [22]).
- **Query II**: A single-stage per-key aggregation, where the intermediate results are persisted in a database.
- **Query III**: A two-stage pipeline that enriches the input stream with location information and then performs a per-location summarization of the entire stream history.
- **Query IV**: A re-implementation of the analytics pipeline of the original Yahoo streaming benchmark (see Figure 3).
- **Query V**: A modification of Query IV, where the per-campaign aggregation is performed over non-overlapping windows (also called *tumbling windows*), instead of the overlapping (or *sliding*) windows of Query IV.
- **Query VI**: A three-stage pipeline that performs a machine learning task. First, it enriches the stream with location information from a database. Then, it performs a per-user feature extraction (i.e., per-key aggregation). Finally, for every location independently it clusters the users periodically using a k-means clustering algorithm.

For every query, we have created a handwritten implementation using the user-level API of Apache Storm, as well as an implementation using our framework of data-trace transductions. Figure 4 shows the experimental comparison of the handcrafted implementations (blue line) and the data-trace-transduction-based implementations (orange line). We have varied the degree of parallelization from 1 up to 8 (shown in the horizontal axis), which correponds to the number of virtual machines assigned to the computation. The vertical axis shows the maximum throughput that can be attained for each configuration. Observe that the hand-written implementation and the generated implementation have similar performance.

The experiments reported in Figure 4 involve compute-heavy operators, but our observations also apply to computationally cheaper operators: our framework incurs a small performance penalty in the range of 0%-20%. In the results for Query I, the generated code is slightly more efficient than the handwritten code (by 10%-15%). This is because we use a routing mechanism that balances the load in a way that minimizes the communication cost, whereas Storm balances the load more evenly across the replicated nodes but incurs a slightly higher communication cost. Overall, we conclude that our framework achieves good performance—despite the higher-level specification and additional typing requirements in the transduction-based code.

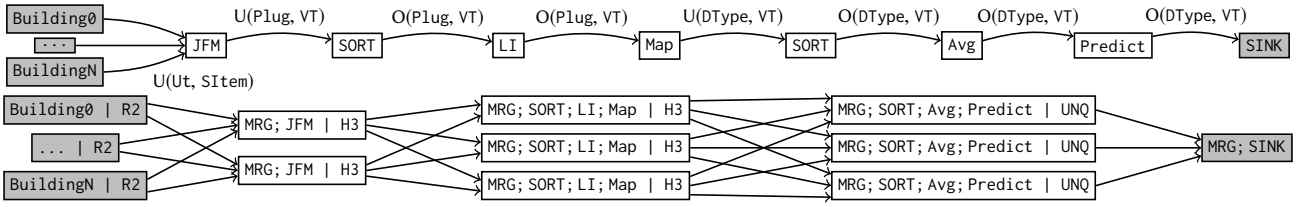


Figure 5. Transduction DAG for a variant of the Smart Home Benchmark [20] of DEBS 2014, and its deployment on Storm.

Case Study: Smart Homes Benchmark. To examine the suitability of our framework for more expressive stream processing applications, we consider a variant of the benchmark used for the "Smart Homes" Internet of Things (IoT) competition of the DEBS 2014 conference [20]. In this benchmark, the input stream consists of measurements produced by smart power plugs. A smart plug is connected to a wall power outlet, and then an electrical device is connected to the plug. This allows the plug sensors to measure quantities that are relevant to power consumption. The deployment of these smart plugs is done across several buildings, each of which contains several units. A smart plug is uniquely identified by three numbers: a building identifier, a unit identifier (which specifies a unit within a building), and a plug identifier (which specifies a plug within a unit). For simplicity, we assume here that the plugs only generate load measurements, i.e. power in Watts. More specifically, every stream event is a tuple with the following components: (i) **`timestamp`**: timestamp of the measurement, (ii) **`value`**: the value of the load measurement (in Watts), (iii) **`plugId`**: identifier that specifies the plug, (iv) **`unitId`**: identifier that specifies the unit, (v) **`buildingId`**: identifier that specifies the building. A plug generates roughly one load measurement for every 2 seconds, but the measurements are not uniformly spaced. There can be gaps in the measurements, as well as many measurements for the same timestamp.

We implement a *load prediction* pipeline in the framework of data-trace transductions. The load prediction is separate for each device type (A/C unit, lights, etc.). The diagram of Figure 5 is a transduction DAG implementing the computation: (i) **`JFM`** (join-filter-map): Join the input stream with information regarding the type of electrical device connected to a plug and retain only a subset of device types. Reorganize the fields of the tuple, separating them into a *plug key* (`plugId`) of type `Plug` and a *timestamped value* of type `VT`. (ii) **`SORT`**: For every plug key, sort the input items (between consecutive markers) by timestamp. (iii) **`LI`**: For every plug key, fill in missing data points using linear interpolation. (iv) **`Map`**: Project every input key (a `Plug` identifier) to the kind of device it is connected to. (v) **`SORT`**: For every device type, sort the input items (between consecutive markers) according to their timestamp. (vi) **`AVG`**: Compute the average load for each device type by averaging all data items with the same key (device type) and timestamp. (vii) **`Predict`**: For every device type and every input value (there is exactly one value per second), predict the total power consumption over the next 10 minutes using the features: current time, current load, and power consumption over the past 1 minute. A decision/regression tree is used for the prediction (`REPTree` of [30]), which has been trained on a subset of the data.

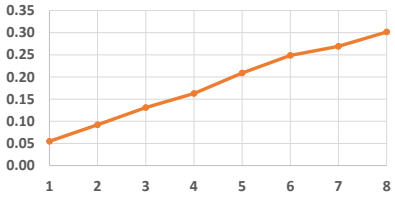


Figure 6. Smart Homes - Energy Prediction: The horizontal (resp., vertical) axis shows the level of parallelization (resp., throughput in million tuples/sec).

Figure 6 shows that by varying the degree of parallelism (number of virtual machines) the computation scales up linearly. As before, we conducted the experiment on a cluster of virtual machines (each with 2 CPUs, 8 GB memory, and 8 GB disk). We conclude from these results that our framework indeed can scale out to high levels of concurrency, even for complex operations such as machine learning inference over streams. Overall, our experiments have demonstrated that our framework can express the complex computations required in both enterprise and IoT streaming applications, and that it can generate an efficient implementation comparable to hand-coded solutions.

7 Related Work

Our programming model is closely related to ***dataflow computation models***. It is a generalization of acyclic *Kahn process networks* (KPNs) [34]. A KPN specifies a finite number of independent linearly ordered input and output channels, and consists of a collection of processes, where each process is a sequential program that can read from its input channels and write to its output channels. *Synchronous Dataflow* [17, 28, 37, 50] is a special case of KPNs, which has been used for specifying and parallelizing streaming programs primarily in the embedded software domain. In a synchronous dataflow graph, each process reads a fixed finite number of items from the input channels and also emits a fixed finite number of items as output. We accommodate a finite number of independent input or output streams, but also allow more complicated dependence relations on the input and output.

In particular, viewing the input or output stream as a bag of events is not possible in KPNs or their restrictions.

There is a large body of work on ***streaming database query languages and systems*** such as Aurora [2] and its successor Borealis [1], STREAM [14], CEDR/StreamInsight [7, 16], and System S [31]. The query language supported by these systems (for example, CQL [14]) is typically an extension of SQL with constructs for sliding windows over data streams. This allows for rich relational queries, including set-aggregations (e.g. sum, max, min, average, count) and joins over multiple data streams, but requires the programmer to resort to user-defined functions in another language for richer computations such as machine learning classification. A precise semantics for how to deal with out-of-order streams has been defined using *punctuations* (a type of synchronization markers) [35, 38, 39, 51]. The partial ordering view supported by data-trace transductions gives the ability to view a stream in many different ways: as a linearly ordered sequence, as a relation, or even as a sequence of relations. This provides a rich framework for classifying disorder, which is useful for describing streaming computations that combine relational with sequence-aware operations. Our implementation supports at the moment only a specific kind of time-based punctuations (i.e., periodic synchronization markers), but our semantic framework can encode more general punctuations. Extending relational query languages to partially ordered multisets has been studied in [29], though not in the context of streaming.

A variety of ***distributed stream processing engines***, including Samza [24, 44], Storm [26], Heron [36, 52], Google's MillWheel [5], Spark Streaming [25, 54], and Flink [18, 23], have achieved widespread use. Spark Streaming and Flink support SQL-style queries or, equivalently, lower-level operations roughly corresponding to the relational algebra underlying SQL. Apache Beam [6, 21] is a programming model that provides relational and window-based abstractions. The other stream engines provide much lower-level abstractions in which the programmer writes event handlers that take tuples, combine the data with windows, and emit results. As with the manually coded Storm implementation used in our experiments, this provides great power but does not aid the programmer in reasoning about correctness. Naiad [42] is a general-purpose distributed dataflow system for performing iterative batch and stream processing. It supports a scheme of logical timestamps for tracking the progress of computations. These timestamps can support the punctuations of [38] and deal with certain kinds of disorder, but they cannot encode more general partial orders. Systems such as Flink [18, 23] and Naiad [42] support feedback cycles, which we do not consider here due to the semantic complexities of cycles: they require a complex denotational model involving continuous functions, as in KPNs [34].

Prior work has considered the issue of ***semantically sound parallelization*** of streaming applications [32, 46]. The authors of [46] observe that Storm [26] and S4 [43] perform unsound parallelizing transformations and propose techniques for exploiting data parallelism without altering the original semantics of the computation. Our framework addresses similar issues, and our markers have a similar role to the "pulses" of [46]. Our approach, however, is based on a type-based discipline for classifying streams and a denotational method for proving the preservation of semantics.

8 Conclusion

We have proposed a type discipline for classifying streams according to their partial ordering characteristics using *data-trace types*. These types are used to annotate the communication links in the dataflow graph that describes a streaming computation. Each vertex of this typed dataflow graph is programmed using a pre-defined set of *templates*, so as to ensure that the code respects the types of the input and output channels. We have implemented this framework in Java and we have provided an automatic procedure for deployment on Apache Storm. We have shown experimentally that our framework can express complex computations required in IoT streaming applications, and that it can produce efficient implementations comparable to hand-coded solutions.

A direction for further work is to enrich the set of type-consistent templates with common patterns. For example, our templates can already express *sliding-window aggregation*, but a specialized template for that purpose would relieve the programmer from the burden of re-discovering and re-implementing efficient sliding-window algorithms (e.g., [15, 48, 49, 53]). Other avenues for future research are to extend the compilation procedure to target streaming frameworks other than Storm, and to automatically perform optimizations that exploit the underlying hardware.

The StreamQRE language [10, 40] consists of a set of programming constructs that allow the combination of streaming computations over linearly-ordered data with static relational operations (i.e., over unordered data). A promising direction for future work is to generalize the language to the setting of partially ordered data streams. StreamQRE is based on a notion of regular stream transformations [8, 9] that admit efficient space-bounded implementations [11, 12], which is a crucial property for applications in resource-constrained environments [3, 4]. It would be interesting to investigate whether a similar notion of regularity can be formulated for the data-trace transductions that we consider here.

Acknowledgments

We would like to thank the anonymous reviewers and Martin Hirzel for their constructive comments. This research was supported in part by US National Science Foundation awards 1763514 and 1640813.

References

[1] D.J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. 2005. The Design of the Borealis Stream Processing Engine. In *Proceedings of the 2nd Biennial Conference on Innovative Data Systems Research (CIDR)*. 277–289.

[2] D. J. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. 2003. Aurora: A New Model and Architecture for Data Stream Management. *The VLDB Journal* 12, 2 (2003), 120–139. https://doi.org/10.1007/s00778-003-0095-z

[3] Houssam Abbas, Rajeev Alur, Konstantinos Mamouras, Rahul Mangharam, and Alena Rodionova. 2018. Real-time Decision Policies with Predictable Performance. *Proc. IEEE* 106, 9 (Sep. 2018), 1593–1615. https://doi.org/10.1109/JPROC.2018.2853608

[4] Houssam Abbas, Alena Rodionova, Konstantinos Mamouras, Ezio Bartocci, Scott A. Smolka, and Radu Grosu. 2018. Quantitative Regular Expressions for Arrhythmia Detection. *To appear in the IEEE/ACM Transactions on Computational Biology and Bioinformatics* (2018). https://doi.org/10.1109/TCBB.2018.2885274

[5] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. 2013. MillWheel: Fault-tolerant Stream Processing at Internet Scale. *Proc. VLDB Endow.* 6, 11 (Aug. 2013), 1033–1044. https://doi.org/10.14778/2536222.2536229

[6] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. 2015. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-scale, Unbounded, Out-of-order Data Processing. *Proceedings of the VLDB Endowment* 8, 12 (Aug. 2015), 1792–1803. https://doi.org/10.14778/2824032.2824076

[7] Mohamed Ali, Badrish Chandramouli, Jonathan Goldstein, and Roman Schindlauer. 2011. The Extensibility Framework in Microsoft StreamInsight. In *Proceedings of the 27th IEEE International Conference on Data Engineering (ICDE '11)*. 1242–1253. https://doi.org/10.1109/ICDE.2011.5767878

[8] Rajeev Alur, Dana Fisman, Konstantinos Mamouras, Mukund Raghothaman, and Caleb Stanford. 2018. Streamable Regular Transductions. *CoRR* abs/1807.03865 (2018). http://arxiv.org/abs/1807.03865

[9] Rajeev Alur, Dana Fisman, and Mukund Raghothaman. 2016. Regular Programming for Quantitative Properties of Data Streams. In *Proceedings of the 25th European Symposium on Programming (ESOP '16)*. 15–40. https://doi.org/10.1007/978-3-662-49498-1_2

[10] Rajeev Alur and Konstantinos Mamouras. 2017. An Introduction to the StreamQRE Language. *Dependable Software Systems Engineering* 50 (2017), 1. https://doi.org/10.3233/978-1-61499-810-5-1

[11] Rajeev Alur, Konstantinos Mamouras, and Caleb Stanford. 2017. Automata-Based Stream Processing. In *Proceedings of the 44th International Colloquium on Automata, Languages, and Programming (ICALP '17) (Leibniz International Proceedings in Informatics (LIPIcs))*, Ioannis Chatzigiannakis, Piotr Indyk, Fabian Kuhn, and Anca Muscholl (Eds.), Vol. 80. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 112:1–112:15. https://doi.org/10.4230/LIPIcs.ICALP.2017.112

[12] Rajeev Alur, Konstantinos Mamouras, and Caleb Stanford. 2019. Modular Quantitative Monitoring. *Proceedings of the ACM on Programming Languages* 3, POPL, Article 50 (Jan. 2019), 31 pages. https://doi.org/10.1145/3290363

[13] Rajeev Alur, Konstantinos Mamouras, Caleb Stanford, and Val Tannen. 2018. Interfaces for Stream Processing Systems. In *Principles of Modeling: Essays Dedicated to Edward A. Lee on the Occasion of His 60th Birthday*, Marten Lohstroh, Patricia Derler, and Marjan Sirjani (Eds.). Lecture Notes in Computer Science, Vol. 10760. Springer, Cham, 38–60. https://doi.org/10.1007/978-3-319-95246-8_3

[14] Arvind Arasu, Shivnath Babu, and Jennifer Widom. 2006. The CQL Continuous Query Language: Semantic Foundations and Query Execution. *The VLDB Journal* 15, 2 (2006), 121–142. https://doi.org/10.1007/s00778-004-0147-z

[15] Arvind Arasu and Jennifer Widom. 2004. Resource Sharing in Continuous Sliding-window Aggregates. In *Proceedings of the 30th International Conference on Very Large Data Bases (VLDB '04)*. VLDB Endowment, 336–347. http://dl.acm.org/citation.cfm?id=1316689.1316720

[16] Roger S. Barga, Jonathan Goldstein, Mohamed Ali, and Mingsheng Hong. 2007. Consistent Streaming Through Time: A Vision for Event Stream Processing. In *Proceedings of the 3rd Biennial Conference on Innovative Data Systems Research (CIDR '07)*. 363–374. http://cidrdb.org/cidr2007/papers/cidr07p42.pdf

[17] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. 2003. The Synchronous Languages 12 Years Later. *Proc. IEEE* 91, 1 (2003), 64–83. https://doi.org/10.1109/JPROC.2002.805826

[18] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink: Stream and Batch Processing in a Single Engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36, 4 (2015). http://sites.computer.org/debull/A15dec/p28.pdf

[19] S. Chintapalli, D. Dagit, B. Evans, R. Farivar, T. Graves, M. Holderbaugh, Z. Liu, K. Nusbaum, K. Patil, B. J. Peng, and P. Poulosky. 2016. Benchmarking Streaming Computation Engines: Storm, Flink and Spark Streaming. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 1789–1792. https://doi.org/10.1109/IPDPSW.2016.138

[20] DEBS Conference. 2014. DEBS 2014 Grand Challenge: Smart homes. http://debs.org/debs-2014-smart-homes/. (2014). [Online; accessed November 16, 2018].

[21] Apache Software Foundation. 2019. Apache Beam. https://beam.apache.org/. (2019). [Online; accessed March 31, 2019].

[22] Apache Software Foundation. 2019. Apache Derby. https://db.apache.org/derby/. (2019). [Online; accessed March 31, 2019].

[23] Apache Software Foundation. 2019. Apache Flink. https://flink.apache.org/. (2019). [Online; accessed March 31, 2019].

[24] Apache Software Foundation. 2019. Apache Samza. http://samza.apache.org/. (2019). [Online; accessed March 31, 2019].

[25] Apache Software Foundation. 2019. Apache Spark Streaming. https://spark.apache.org/streaming/. (2019). [Online; accessed March 31, 2019].

[26] Apache Software Foundation. 2019. Apache Storm. http://storm.apache.org/. (2019). [Online; accessed March 31, 2019].

[27] Apache Software Foundation. 2019. Apache Storm: Concepts. http://storm.apache.org/releases/1.2.2/Concepts.html. (2019). [Online; accessed March 31, 2019].

[28] Michael I. Gordon, William Thies, and Saman Amarasinghe. 2006. Exploiting Coarse-grained Task, Data, and Pipeline Parallelism in Stream Programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XII)*. ACM, New York, NY, USA, 151–162. https://doi.org/10.1145/1168857.1168877

[29] Stéphane Grumbach and Tova Milo. 1999. An Algebra for Pomsets. *Information and Computation* 150, 2 (1999), 268–306. https://doi.org/10.1006/inco.1998.2777

[30] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. 2009. The WEKA Data Mining Software: An Update. *SIGKDD Explorations Newsletter* 11, 1 (Nov. 2009), 10–18. https://doi.org/10.1145/1656274.1656278

[31] M. Hirzel, H. Andrade, B. Gedik, G. Jacques-Silva, R. Khandekar, V. Kumar, M. Mendell, H. Nasgaard, S. Schneider, R. Soulé, and K. L. Wu. 2013. IBM Streams Processing Language: Analyzing Big Data in motion. *IBM Journal of Research and Development* 57, 3/4 (2013),

7:1–7:11. https://doi.org/10.1147/JRD.2013.2243535

[32] Martin Hirzel, Robert Soulé, Scott Schneider, Buğra Gedik, and Robert Grimm. 2014. A Catalog of Stream Processing Optimizations. *ACM Computing Surveys (CSUR)* 46, 4, Article 46 (March 2014), 34 pages. https://doi.org/10.1145/2528412

[33] Yahoo Inc. 2017. Reference implementation of the Yahoo Streaming Benchmark. https://github.com/yahoo/streaming-benchmarks. (2017). [Online; accessed March 31, 2019].

[34] Gilles Kahn. 1974. The Semantics of a Simple Language for Parallel Programming. *Information Processing* 74 (1974), 471–475.

[35] Sailesh Krishnamurthy, Michael J. Franklin, Jeffrey Davis, Daniel Farina, Pasha Golovko, Alan Li, and Neil Thombre. 2010. Continuous Analytics over Discontinuous Streams. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD '10)*. ACM, New York, NY, USA, 1081–1092. https://doi.org/10.1145/1807167.1807290

[36] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthik Ramasamy, and Siddarth Taneja. 2015. Twitter Heron: Stream Processing at Scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*. ACM, 239–250. https://doi.org/10.1145/2723372.2742788

[37] Edward A. Lee and David G. Messerschmitt. 1987. Synchronous Data Flow. *Proc. IEEE* 75, 9 (1987), 1235–1245. https://doi.org/10.1109/PROC.1987.13876

[38] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A. Tucker. 2005. Semantics and Evaluation Techniques for Window Aggregates in Data Streams. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data (SIGMOD '05)*. ACM, 311–322. https://doi.org/10.1145/1066157.1066193

[39] Jin Li, Kristin Tufte, Vladislav Shkapenyuk, Vassilis Papadimos, Theodore Johnson, and David Maier. 2008. Out-of-order Processing: A New Architecture for High-performance Stream Systems. *Proceedings of the VLDB Endowment* 1, 1 (Aug. 2008), 274–288. https://doi.org/10.14778/1453856.1453890

[40] Konstantinos Mamouras, Mukund Raghothaman, Rajeev Alur, Zachary G. Ives, and Sanjeev Khanna. 2017. StreamQRE: Modular Specification and Efficient Evaluation of Quantitative Queries over Streaming Data. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '17)*. ACM, New York, NY, USA, 693–708. https://doi.org/10.1145/3062341.3062369

[41] Antoni Mazurkiewicz. 1987. Trace theory. In *Petri Nets: Applications and Relationships to Other Models of Concurrency (LNCS)*, W. Brauer, W. Reisig, and G. Rozenberg (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 278–324. https://doi.org/10.1007/3-540-17906-2_30

[42] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: A Timely Dataflow System. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. ACM, New York, NY, USA, 439–455. https://doi.org/10.1145/2517349.2522738

[43] Leonardo Neumeyer, Bruce Robbins, Anish Nair, and Anand Kesari. 2010. S4: Distributed Stream Computing Platform. In *Proceedings of the 2010 IEEE International Conference on Data Mining Workshops*. 170–177. https://doi.org/10.1109/ICDMW.2010.172

[44] Shadi A. Noghabi, Kartik Paramasivam, Yi Pan, Navina Ramesh, Jon Bringhurst, Indranil Gupta, and Roy H. Campbell. 2017. Samza: Stateful Scalable Stream Processing at LinkedIn. *Proceedings of the VLDB Endowment* 10, 12 (Aug. 2017), 1634–1645. https://doi.org/10.14778/3137765.3137770

[45] Vaughan Pratt. 1986. Modeling Concurrency with Partial Orders. *International Journal of Parallel Programming* 15, 1 (Feb 1986), 33–71. https://doi.org/10.1007/BF01379149

[46] Scott Schneider, Martin Hirzel, Buğra Gedik, and Kun-Lung Wu. 2015. Safe Data Parallelism for General Streaming. *IEEE Trans. Comput.* 64, 2 (Feb 2015), 504–517. https://doi.org/10.1109/TC.2013.221

[47] Utkarsh Srivastava and Jennifer Widom. 2004. Flexible Time Management in Data Stream Systems. In *PODS (PODS '04)*. ACM, New York, NY, USA, 263–274. https://doi.org/10.1145/1055558.1055596

[48] Kanat Tangwongsan, Martin Hirzel, and Scott Schneider. 2017. Low-Latency Sliding-Window Aggregation in Worst-Case Constant Time. In *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems (DEBS '17)*. ACM, New York, NY, USA, 66–77. https://doi.org/10.1145/3093742.3093925

[49] Kanat Tangwongsan, Martin Hirzel, Scott Schneider, and Kun-Lung Wu. 2015. General Incremental Sliding-window Aggregation. *Proceedings of the VLDB Endowment* 8, 7 (2015), 702–713. https://doi.org/10.14778/2752939.2752940

[50] William Thies, Michal Karczmarek, and Saman Amarasinghe. 2002. StreamIt: A Language for Streaming Applications. In *Proceedings of the 11th International Conference on Compiler Construction (CC '02) (Lecture Notes in Computer Science)*, R. Nigel Horspool (Ed.), Vol. 2304. Springer Berlin Heidelberg, Berlin, Heidelberg, 179–196. https://doi.org/10.1007/3-540-45937-5_14

[51] Peter A. Tucker, David Maier, Tim Sheard, and Leonidas Fegaras. 2003. Exploiting Punctuation Semantics in Continuous Data Streams. *IEEE Transactions on Knowledge and Data Engineering* 15, 3 (2003), 555–568. https://doi.org/10.1109/TKDE.2003.1198390

[52] Twitter. 2019. Heron. https://apache.github.io/incubator-heron/. (2019). [Online; accessed March 31, 2019].

[53] Jun Yang and Jennifer Widom. 2003. Incremental Computation and Maintenance of Temporal Aggregates. *The VLDB Journal* 12, 3 (Oct. 2003), 262–283. https://doi.org/10.1007/s00778-003-0107-z

[54] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. 2013. Discretized Streams: Fault-tolerant Streaming Computation at Scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. ACM, New York, NY, USA, 423–438. https://doi.org/10.1145/2517349.2522737