Synchronization Schemas

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

Phillip Hilliard
pdh@cis.upenn.edu
University of Pennsylvania

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

Konstantinos Kallas
kallas@seas.upenn.edu
University of Pennsylvania

Konstantinos Mamouras
mamouras@rice.edu
Rice University

Filip Niksic
fniksic@google.com
Google

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

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

Anton Xue
antonxue@seas.upenn.edu
University of Pennsylvania

ABSTRACT

We present a type-theoretic framework for data stream processing for real-time decision making, where the desired computation involves a mix of sequential computation, such as smoothing and detection of peaks and surges, and naturally parallel computation, such as relational operations, key-based partitioning, and map-reduce. Our framework unifies sequential (ordered) and relational (unordered) data models. In particular, we define *synchronization schemas* as types, and *series-parallel streams* (SPS) as objects of these types. A synchronization schema imposes a hierarchical structure over relational types that succinctly captures ordering and synchronization requirements among different kinds of data items. Series-parallel streams naturally model objects such as relations, sequences, sequences of relations, sets of streams indexed by key values, time-based and event-based windows, and more complex structures obtained by nesting of these. We introduce *series-parallel stream transformers* (SPST) as a domain-specific language for modular specification of deterministic transformations over such streams. SPSTs provably specify only monotonic transformations allowing streamability, have a modular structure that can be exploited for correct parallel implementation, and are composable allowing specification of complex queries as a pipeline of transformations.

CCS CONCEPTS

• **Information systems → Data streams**; • **Theory of computation** → *Database theory*; • **Computing methodologies** → *Parallel programming languages.*

KEYWORDS

Stream processing, database query languages, synchronization schemas, parallelism, streams, relations, series-parallel partial orders

ACM Reference Format:
Rajeev Alur, Phillip Hilliard, Zachary G. Ives, Konstantinos Kallas, Konstantinos Mamouras, Filip Niksic, Caleb Stanford, Val Tannen, and Anton Xue. 2021. Synchronization Schemas. In *Proceedings of the 40th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS '21), June 20–25, 2021, Virtual Event, China.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3452021.3458317

1 INTRODUCTION

Emerging applications in the IoT era, in domains such as health monitoring, transportation, and smart homes, require the ability to interact with the physical world using sensors, continuously process collected data, and make decisions in a feedback loop in a timely manner. This requires revisiting models for distributed stream processing — with an emphasis on integrating machine learning and quality checking algorithms into processing pipelines, computing both at the edge and on the cloud, and (rather than simple data acquisition and summarization) providing low-latency feedback to the user on their low-energy device. To facilitate the design and implementation of such applications, stream processing systems should ideally allow programmers to describe the decision logic conveniently and modularly in a high-level query language, and support compilation of queries to distributed platforms with correctness guarantees, low latency, and energy efficiency.

To design query languages and optimizing compilers for streaming computations, we need mathematically precise definitions of data streams and transformations over them. A natural, and classical, definition of a stream is to view it as a (linearly ordered) sequence of data items. A stream transformation, then, is a *monotonic*—with respect to prefix ordering over sequences, function from input streams to output streams. Such transformations compose naturally, and form the basis of a number proposals for event-driven reactive programming (e.g., [69]). However, assuming a strict linear order over input data items is not ideal for two reasons. First, in a distributed implementation, there may be a high overhead to impose a consistent linear ordering among items arriving at different processing nodes. Second, the sequential view hides opportunities for parallel evaluation. For example, consider the computation that partitions the input data into sub-streams based on key values—for example, IP-addresses of source nodes in network traffic, and processes each sub-stream independently. While such key-based

PODS '21, June 20–25, 2021, Virtual Event, China
© 2021 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-8381-3/21/06.
https://doi.org/10.1145/3452021.3458317

partitioning is a common pattern for queries, and can be parallelized naturally, it is not supported in purely sequential languages for streaming computation.

A contrasting formalization of a data stream is to view it as a relation that gets incrementally updated with each new input item. The mature theory and practice of relational query languages and query processing makes such a relational view appealing. However, as noted in the early papers on streaming data processing [10], a purely relational view is inadequate in the streaming context. As an example, consider the computation that, given an input stream of measurements from a monitoring device, emits each minute some aggregate of measurements during the past minute. For specifying this computation, the input stream of measurements needs to be augmented with end-of-minute events. Such events, sometimes called *punctuations* or *synchronization* markers [29, 42, 46, 47, 66], are linearly ordered, and play a central role as triggers to produce outputs of relational queries and keep the state of computation bounded. Consequently, existing query languages typically extend relational query languages with a variety of windowing constructs for selecting sub-streams [13, 36]. This state of the art is not satisfactory for the following reasons. First, consider a scenario where the input stream consists of periodic measurements from a cardiac monitoring device, and the computation involves smoothing of the input signal, detection of peaks, and detection of surge episodes. There is no built-in support for specifying such time-series-dependent sequential computation in SQL-extensions. The user can specify such a computation as low-level user-defined code, but then loses the benefits of automatic parallelization and query optimization. Second, adding different forms of windowing constructs to relational languages is not mathematically satisfying due to lack of generality and compelling semantic foundations. For instance, existing languages typically do not support nested windows, which can be attributed to the mismatch in types of inputs and outputs of the windowing constructs.

We propose to model data streams as particularly structured directed acyclic graphs whose vertices are labeled with data items and correspond to event occurrences, and edges correspond to temporal ordering. More precisely, given a relational schema, that is, relational types for data items, we define *synchronization schemas* as types for input streams, and *series-parallel streams* (SPS) as objects of these types. The base schema $\mathrm{Bag}(\mathcal{H})$, for a set $\mathcal{H}$ of relational types, captures unordered collection of events of type $\mathcal{H}$. Complex schemas are constructed inductively using three rules. $\mathrm{Sync}(\mathcal{H}, S)$ defines a parent relation between events of type $\mathcal{H}$ and events of types appearing in the schema S, and denotes a stream consisting of a sequence of events of type $\mathcal{H}$, that act as synchronization markers, interspersed with sub-streams of type S. $\mathrm{PartitionBy}(K, S)$ partitions a schema S based on a set of keys K, and denotes a set of parallel streams of type S, indexed by the values for the key fields in K. Finally, $\mathrm{Par}(S_1, S_2)$ describes a sibling relation between schemas S_1 and S_2, and corresponds to a parallel composition of streams of types S_1 and S_2. There is a natural (and polynomial-time computable) *relaxation* ordering over synchronization schemas: a schema S is a relaxation of another schema S' if S' requires more pairwise ordering of events than S.

Series-parallel streams can naturally model objects such as relations, sequences, sequences of relations, and time-based or event-based windows, as well as more complex structures obtained by arbitrary nestings of these. The tree-structure imposed by a synchronization schema on data items of different types—such as data items generated by distinct devices, succinctly captures the ordering requirements among them. The synchronization schema then can be viewed as a contract between the implementation producing input events and the query consuming them. For the producer, the events ordered by the schema must arrive in order, which can be enforced using timestamps, while no such synchronization overhead is necessary for unordered events. For the consumer, for events that are unordered according to the schema, the result of the computation should not depend on whether they are processed sequentially in some arbitrary order or in parallel, thus providing a blue-print for safe parallelization.

After we define synchronization schemas and series-parallel streams in Section 2, in Section 3 we introduce *series-parallel stream transformers* (SPST), a domain-specific parallel programming language for specifying transformations over streams. Building upon relational transformers, reduce operators to combine results of partitioned computations, and standard sequential constructs such as list iterators, SPSTs are defined inductively based on the structure of the synchronization schema of the input stream. While defining this model, two aspects particularly require technical care. First, the specification of the computation of an SPST, and its semantics as a transformers over series-parallel streams, should be *modular*. This ensures that once we define a transformer overs streams of type S, it can be used in defining transformers over streams of more complex types with S as a sub-schema. Second, the semantics of an SPST, as a function from input SPS to output SPS, must be *monotonic* with prefix ordering over series-parallel streams. Monotonicity is the essence of streamability, and ensuring it has been a challenge in existing literature on formally defining semantics of streaming computations with punctuations. We achieve modularity and monotonicity by associating two types of semantics, *open* semantics and *closed* semantics with each SPST. The distinction between the two is informally that the latter assumes that the most current sub-streams have been closed by a synchronizing input item triggering the generation of corresponding outputs.

In our proposal, a user can express the desired decision logic as a pipeline of queries, where each query can be compiled into an SPST, such that the input schema of a query is a relaxation of the output schema of the previous one. Note that, in contrast to typical distributed stream processing frameworks which use dataflow graphs for specifying computations, pipelines suffice in our framework since parallelism is built into the types of streams themselves. In Section 5, we present a library of queries that capture some natural transformations over series-parallel streams. These include SPS-analogs of relational operations such as map and filter, windowing constructs to impart hierarchical structure, relational and sequential aggregations, parallel streaming computations of key-based partitioning and MapReduce, complex event processing constructs such as regular-expression-based temporal markers, time-series dependent transformations, and complex forms of joins. In the context of rich literature on stream processing languages, the key benefit of our framework is *composability*: since each query is a deterministic function over series-parallel streams, queries can be composed sequentially in an arbitrary manner (modulo compatibility of types captured by the synchronization schemas). To illustrate this, we show how to modularly express some classical queries from the literature such as NEXMark [65], and show specifications of some complex queries over data about COVID cases that mix relational and temporal constructs [38], and hence are not expressible in current frameworks as high-level queries.

This paper presents synchronization schemas, series-parallel streams, and SPSTs as a mathematical foundation for query languages and query processing for streaming computations. We conclude this paper by discussing directions for future research in Section 6. The next step is to build a prototype implementation with a focus on exploiting parallelism, and evaluating the performance with respect to the existing stream processing systems such as Flink [24]. Series-parallel streams have a mathematically appealing structure, and a principled design of query languages over them needs a theoretical investigation with a focus on expressiveness and guarantees regarding usage of computational resources for streaming parallel evaluation. A direction of research, of particular relevance to IoT applications, concerns query optimization with a focus on distributing the computation to reduce the amount of information exchanged for query evaluation.

2 PARTIALLY ORDERED DATA STREAMS

2.1 Illustrative Example

Suppose a city is trying to monitor the average cost-per-mile being charged by a taxi fleet, and they want to re-estimate this every hour. Towards this goal we have a stream of data items generated by a set of taxis consisting of GPS events describing locations of taxis and RideCompleted events describing fares of completed rides. In order to evaluate such a query, every hour we would need to: (1) use the GPS data to compute the total distance travelled by each taxi that hour, (2) add these distances across all taxis, (3) add the costs of all completed rides during that hour, and (4) output the aggregate cost divided by the aggregate distance.

To implement such a computation, the input stream of GPS and RideCompleted events must be augmented with EndOfHour events that mark the passage of one hour. Such events, called *punctuations* in the literature [29, 42, 46, 47, 66], can be used to trigger the outputs, and allow the input stream to be logically divided into a sequence of windows, where each window consists of GPS and RideCompleted events within the same hour. Let us make a few observations regarding how we can compute the desired output efficiently. First, we can process GPS events for different taxis independently and possibly out of order, but such events for the same taxi must be processed sequentially, in temporal order, to be able to calculate the total distance the taxi travels. Second, we can view RideCompleted events within each hour as a bag (that is, a multiset), and process them in any order. Finally, every time an EndOfHour event appears, it acts as a synchronization marker for all events, and to process it, we need to make sure to combine all the aggregates to produce the correct output.

We can capture all these ordering requirements by viewing the input stream as a *partially ordered* collection of events as shown in Figure 1. In this illustration, squares, circles, and triangles indicate EndOfHour, GPS, and RideCompleted events, respectively, different colors correspond to different taxis, and directed edges denote event ordering. In fact, such a stream is a particular kind of a partial order, namely, a *series-parallel* directed acyclic graph labeled with data items. The *shape* of such an input stream will be captured by a type, which we call a *synchronization schema*, shown in Figure 2. This schema says that the input stream consists of a sequence of substreams punctuated by EndOfHour events, where each substream is a parallel composition of a bag of RideCompleted events and a collection of sequences, one per taxi, of GPS events.

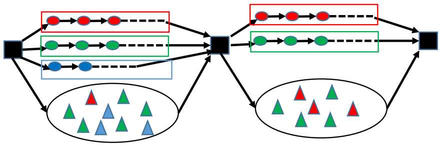

Figure 1: Illustrative series-parallel stream.

The synchronization schema can be viewed as a contract between the implementation generating input events and the query processing them: the events ordered by the schema must arrive in order (which can be enforced using timestamps), and the computation corresponding to the query does not depend on the processing order of events that are independent according to the schema. In this particular case, the schema does not require to order events of different taxis, thus allowing the implementation to ignore reorderings of events across taxis due to, say, network delays. On the other hand, the computation does depend on the temporal ordering of events of the same taxi, and existing stream processing systems do not support mixing such sequential computation with parallelization across taxis in a convenient manner.

Note that a synchronization schema captures only the ordering and synchronization requirements, and not every aspect of the desired computation. For example, if we change the query by replacing the total distance travelled by each taxi by an arbitrary function over the sequence of GPS events of a taxi, the synchronization schema, and the desired shape of the input stream, will stay unchanged.

2.2 Synchronization Schemas

We start by defining some preliminary notions from relational query processing: headers, tuples, and header keys. We assume that tuples, i.e. stream elements, are of finitely many possible types, where each type is given as a relational schema or *header*.

Definition 1 (Headers and tuples). A *header* H consists of a unique *header name* α and *fields* $\langle \alpha_i : \tau_i \rangle$, for $1 \leq i \leq n$, where each α_i is a *field name* and τ_i is a *field type*. A *tuple* x of type H, denoted $x : H$, is of the form $x = (x_1, x_2, \ldots, x_n)$, where each $x_i : \tau_i$. □

When the context is clear, we identify each header H with its name α and likewise each field $\langle \alpha_i : \tau_i \rangle$ with its name α_i. We write $\alpha_i \in H$ to mean that α_i is a field of H, and $x.\alpha_i$ to denote the value of x on α_i. For a set $\mathcal{H}$ of headers, we write $x : \mathcal{H}$ if $x : H$ for some $H \in \mathcal{H}$.

Example 2. For the example introduced in Section 2.1 we have the following three headers for the input data items. As a convention, names of fields start with lowercase, and names of headers start with uppercase. GPS data for each taxi is described by the header `GPS(location: pos, taxiID: int)`, where `pos` indicates the type of a GPS measurement—three dimensional coordinates, for instance. Completed ride data is described by the header `RideCompleted(rideID: int, taxiID: int, passengerID: int, cost: int)`. Finally, end-of-hour events are used to synchronize in time; these are described by the header `EndOfHour(date: date, hour: int)`. □

As explained in Section 2.1, the input stream can be viewed as a partially ordered set of events, each labeled with a tuple of data. The structure of such a stream is captured by a data type that we call *synchronization schemas*. A synchronization schema is a hierarchical forest-like structure that can be seen as an extension of database schemas.

Definition 3 (Synchronization schema)**.** A *synchronization schema* S is inductively defined as follows:

$$S \ ::= \ \mathrm{Bag}(\mathcal{H}) \mid \mathrm{Sync}(\mathcal{H}, S) \mid \mathrm{PartitionBy}(K, S) \mid \mathrm{Par}(S, S),$$

where K denotes a set of field names (partition keys) and $\mathcal{H}$ denotes a set of headers. □

The base rule $\mathrm{Bag}(\mathcal{H})$ defines a stream corresponding to a bag of $\mathcal{H}$-events, that is, events labeled with tuples of type $\mathcal{H}$. To combine the base rule (leafs) in complex schemas we use the other three rules. $\mathrm{Sync}(\mathcal{H}, S)$ defines a parent relation between $\mathcal{H}$-events and S-events—events labeled with any of the headers appearing in the schema S, and denotes a stream consisting of a sequence of $\mathcal{H}$-events that act as synchronization markers, interspersed with substreams of type S. $\mathrm{PartitionBy}(K, S)$ partitions a schema S based on a set of key fields K, and denotes a set of streams of type S, indexed by the values for the keys in K. Finally, $\mathrm{Par}(S_1, S_2)$ describes a sibling relation between schemas S_1 and S_2, and corresponds to a parallel composition of streams of types S_1 and S_2.

In the sequel, we use the following notational abbreviations. When a set of headers consists of a single header H, we write H instead of $\{H\}$. We use $\mathrm{Seq}(\mathcal{H})$ for $\mathrm{Sync}(\mathcal{H}, \mathrm{Bag}(\varnothing))$, where $\varnothing$ is the empty set of headers, and such a schema corresponds to a totally ordered sequence of $\mathcal{H}$-events. Finally, we write $\mathrm{Par}(S_1, S_2, S_3)$ for the parallel composition of three schemas, which is short for $\mathrm{Par}(\mathrm{Par}(S_1, S_2), S_3)$ (note that parallel composition is associative).

Example 4. We can define the schema for the input of the example in Section 2.1 in a bottom up fashion in the following manner. First, $S_1 = \mathrm{PartitionBy}(\mathtt{taxiID}, \mathrm{Seq}(\mathtt{GPS}))$ denotes that `GPS` events are partitioned by the key `taxiID` and are totally ordered for each taxi. Second, $S_2 = \mathrm{Bag}(\mathtt{RideCompleted})$ denotes that `RideCompleted` events are unordered, and can be considered to be a bag. Finally, $S = \mathrm{Sync}(\mathtt{EndOfHour}, \mathrm{Par}(S_1, S_2))$ denotes that `EndOfHour` events synchronize the events in S_1 and S_2, each of which can be processed in parallel as they are independent. It is often helpful to visualize schemas as forests. Figure 2 illustrates the schema for the example. Siblings correspond to the $\mathrm{Par}(S_1, S_2)$ constructor while the rectangular box, labeled with the key fields, corresponds to the $\mathrm{PartitionBy}(K, S)$ constructor. A parent node with several children corresponds to the $\mathrm{Sync}(\mathcal{H}, S)$ constructor. □

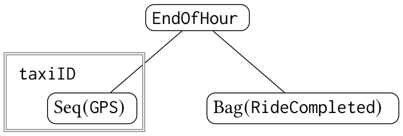

Figure 2: Example synchronization schema.

We additionally require for the remainder of the document that synchronization schemas are *well-formed*, as defined below. First note that the partitioning construct $\mathrm{PartitionBy}(K, S)$ naturally gives rise to a concept of scope for partition keys: for each partition key $k \in K$ and for each header H appearing in the schema S, we say that H is *in the scope of* k.

Definition 5 (Well-formed schema)**.** A *synchronization schema* S is *well-formed* if the following conditions hold: (1) no header H appears in S twice, (2) if a header H is in the scope of a partition key k, then k is a field of H, i.e. $k \in H$, and (3) if a partition schema $\mathrm{PartitionBy}(K, S)$ is in the scope of a partition key k, then $k \notin K$. □

The first condition is necessary for unambiguous parsing, while the latter two ensure that splitting on a key field is meaningful in a given context. Note that it is straightforward to check the conditions necessary for a schema to be well-formed.

2.3 Series-Parallel Streams

Now that we have defined synchronization schemas, which act as types, we can define a natural inductive representation of streams that are values of these types. We call these series-parallel streams, or SPS for short. We have already seen an example of such a stream in Figure 1 informally. Before we formalize the definition, consider the sequence of `GPS` events corresponding to the red taxi. The type of this sequence is $\mathrm{Seq}(\mathtt{GPS})$, but is more specialized since all these events share a common value of the field `taxiID`. We will denote such an instantiation of the schema with common key values as $\mathrm{Seq}(\mathtt{GPS})[\mathtt{taxiID} = \mathtt{red}]$. Such a type can be viewed as *refinement type* of the schema type $\mathrm{Seq}(\mathtt{GPS})$.

If $d : H$ and F is a subset of the fields of H, we write $d|_F$ for the restriction of d to contain only those fields in F. For a set K of partition keys, K can also be considered to be a header containing these keys as its only fields. Then, for a particular tuple v of such a header type K, for a schema S, we use $S[v]$ to denote the refinement of schema S to an instance where all the tuples are required to have key values as specified by v.

We use the following syntactic constructs to capture the structure of the desired series-parallel streams: $[x_1, x_2, \ldots, x_n]$ for a sequence (list), $\{x_1, x_2, \ldots, x_n\}$ for a bag (with standard bag equality semantics), $\langle x_1, x_2 \rangle$ for a pair of x_1 and x_2 where x_1 and x_2 are thought of as parallel instead of sequential, and $v \mapsto x$ to represent a key-indexed value.

Definition 6 (Series-Parallel Streams)**.** Let S be a synchronization schema. A *series-parallel stream* (SPS) $t : S[v]$ for a specific instantiation of key values $v : K$ is inductively defined as follows:

- If $d_i : \mathcal{H}$ such that $d_i|_K = v$ for $i = 1, \ldots, m$, then $t = \{d_1, \ldots d_m\}$ is an SPS of type $\mathrm{Bag}(\mathcal{H})[v]$.

- If $t_1 : S_1[v]$ and $t_2 : S_2[v]$, then $t = \langle t_1, t_2 \rangle$ is an SPS of type $\mathrm{Par}(S_1, S_2)[v]$.
- If $d_i : \mathcal{H}$ such that $d_i|_K = v$ for $i = 1, \dots, m$, and if $t_i : S'[v]$ for $i = 0, 1, \dots, m$, then $t = [t_0, d_1, t_1, \dots, d_m, t_m]$ is an SPS of type $\mathrm{Sync}(\mathcal{H}, S')[v]$.
- Suppose that K' is a set of partition keys disjoint from K, and that $v'_1, v'_2, \dots, v'_m : K'$ are *distinct* instances of key values for K'. Suppose $t_1, t_2, \dots, t_m$ are *nonempty* streams such that $t_i : S'[v_i]$, and let $v_i : K \cup K'$ to the unique valuations such that $v_i|_K = v$ and $v_i|_{K'} = v'_i$, i.e. the extension of v'_i with the key values in v. Then $t = \{v'_1 \mapsto t_1, v'_2 \mapsto t_2, \dots, v'_m \mapsto t_m\}$ is an SPS of type $\mathrm{PartitionBy}(K', S')[v]$. □

We write $t : S$ when $K = \varnothing$ for $t : S[()]$, where $()$ is the empty tuple of type K. When $S[v]$ is clear from the context, we write $\bot : S[v]$ for the empty SPS: this abbreviates $\{\}$ for $S = \mathrm{Bag}(\mathcal{H})$, $\langle \bot, \bot \rangle$ for $S = \mathrm{Par}(S_1, S_2)$, $[\bot]$ for $S = \mathrm{Sync}(\mathcal{H}, S')$, and $\{\}$ for $S = \mathrm{PartitionBy}(K', S')$.

Example 7. Let us revisit the SPS shown in Figure 1 of the schema defined in Example 4. The sequence of GPS events of the red taxi within the first hour is the stream $t_r = [\bot, r_1, \bot, r_2, \dots, r_n, \bot]$, where $\bot$ is the empty stream of type $\mathrm{Bag}(\varnothing)$ and $r_1, r_2, \dots r_n$ are the corresponding events. The streams t_g and t_b of GPS events of the green and blue taxis, respectively, have a similar structure. The stream $t_1 = \{\mathtt{red} \mapsto t_r, \mathtt{green} \mapsto t_g, \mathtt{blue} \mapsto t_b\}$ then captures all the GPS events in the first hour and is of type $S_1 = \mathrm{PartitionBy}(\mathtt{taxiID}, \mathrm{Seq}(\mathtt{GPS}))$. The following is the bag containing all RideCompleted events in the first hour, and is of type $S_2 = \mathrm{Bag}(\mathtt{RideCompleted})$: $t'_1 = \{r'_1, \dots, g'_1, \dots, b'_1, \dots\}$. The streams t_2 and t'_2 are analogous to the streams t_1 and t'_1, respectively, and capture the GPS and RideCompleted events in the second hour. If eoh_1, eoh_2, eoh_3 represent the first three EndOfHour events, then the stream $[\bot, eoh_1, \langle t_1, t'_1 \rangle, eoh_2, \langle t_2, t'_2 \rangle, eoh_3, \bot]$ represents all the events. Its type is $S = \mathrm{Sync}(\mathtt{EndOfHour}, \mathrm{Par}(S_1, S_2))$. □

A few remarks regarding the technical details of this definition are in order. The definition is set up so that a linear sequence of tuples over the headers appearing in a schema can be uniquely parsed (that is, interpreted) as a series-parallel stream (see Proposition 14). In parallel composition case, the order of the components matters, and component sub-streams may be empty. On the other hand, in key-based partitioning, the stream is defined to be a set of non-empty sub-streams, one per key value. To understand the case of nested hierarchical structure, consider the schema $\mathrm{Sync}(A, \mathrm{Sync}(B, \mathrm{Seq}(C)))$. In the corresponding stream, a C-tuple may be followed by an A-tuple without any intervening B-tuples. This requires care to make sure that a synchronizing event is able *close* all open sub-streams corresponding to its descendants.

Finally, we define *concatenation*, denoted $\circ$, of series-parallel streams. This generalizes the notion of concatenation of sequences. Intuitively, if we consider a cut through the series-parallel stream, say, shown in Figure 1, such that the left stream is closed under predecessors (and right stream is closed under successors) then concatenating the left and right substreams should give us the original stream.

Definition 8 (Concatenation and Prefix Ordering for SPS)**.** Let $t, u : S[v]$ be series-parallel streams over the same schema S and key valuation v. The *concatenation* $t \circ u$ is defined inductively on the structure of S:

- If $S = \mathrm{Bag}(\mathcal{H})$, $t = \{d_1, \dots, d_m\}$, and $u = \{e_1, \dots, e_n\}$, then
$$t \circ u = \{d_1, \dots, d_m, e_1, \dots, e_n\}.$$
- If we have $S = \mathrm{Sync}(\mathcal{H}, S')$, $t = [t_0, d_1, t_1, \dots, d_m, t_m]$, and $u = [u_0, e_1, u_1, \dots, e_n, u_n]$, then
$$t \circ u = [t_0, d_1, t_1, \cdots, d_m, (t_m \circ u_0), e_1, u_1, \cdots, e_n, u_n].$$
- If $S = \mathrm{PartitionBy}(K, S')$, then let the overlapping key values between t and u be $v_1, v_2, \dots, v_l$, with additional keys $v'_1, v'_2, \dots, v'_m$ in t only and $v''_1, v''_2, \dots, v''_n$ in u only. If $t = \{v_1 \mapsto t_1, \dots, v_l \mapsto t_l, v'_1 \mapsto t'_1, \dots, v'_m \mapsto t'_m\}$ and $u = \{v_1 \mapsto u_1, \dots, v_l \mapsto u_l, v''_1 \mapsto u'_1, \dots, v''_n \mapsto u'_n\}$, then
$$\begin{aligned} t \circ u = \{\ & v_1 \mapsto t_1 \circ u_1, \dots, v_l \mapsto t_l \circ u_l, \\ & v'_1 \mapsto t'_1, \dots, v'_m \mapsto t'_m, \\ & v''_1 \mapsto u'_1, \dots, v''_n \mapsto u'_n.\ \} \end{aligned}$$
- If $S = \mathrm{Par}(S_1, S_2)$, $t = \langle t_1, t_2 \rangle$, and $u = \langle u_1, u_2 \rangle$, then
$$t \circ u = \langle t_1 \circ u_1, t_2 \circ u_2 \rangle.$$

For $t, u : S[v]$, t is said to be a *prefix* of u, written $t \le u$, if there exists a series-parallel stream $t' : S[v]$ such that $t \circ t'$ equals u. □

Proposition 9. For each type $S[v]$ and for all $t, t', t'' : S[v]$, the following hold: (1) $t \circ \bot = \bot \circ t = t$. (2) $(t \circ t') \circ t'' = t \circ (t' \circ t'')$. (3) $t \le t$. (4) If $t \le t'$ and $t' \le t$, then $t = t'$. (5) If $t \le t'$ and $t' \le t''$, then $t \le t''$. □

2.4 Sequential View

Contrasting with the series-parallel view of streams provided in the previous section, we describe here a sequential view as sequences up to equivalence. To define the equivalence we first define a dependence relation on tuples based on the given synchronization schema: two tuples in the relation are said to be dependent if the order between them is important.

Definition 10 (Dependence Relation)**.** Let S be a synchronization schema and let $\mathbf{headers}(S)$ be all the headers appearing in S. The *dependence relation* is a binary relation on tuples of $\mathbf{headers}(S)$, written $x\, D_S\, y$ for $x, y : \mathbf{headers}(S)$, and defined inductively as follows: (i) if $S = \mathrm{Bag}(\mathcal{H})$, then D_S is the empty set; (ii) if $S = \mathrm{Sync}(\mathcal{H}, S')$, then D_S is $\{\ (x, y) \mid x : \mathcal{H} \text{ or } y : \mathcal{H} \text{ or } x\, D_{S'}\, y\ \}$; (iii) if $S = \mathrm{PartitionBy}(K, S')$, then D_S is $\{\ (x, y) \mid (x\, D_{S'}\, y) \text{ and } x|_K = y|_K\ \}$; and (iv) if $S = \mathrm{Par}(S_1, S_2)$, then D_S is $D_{S_1} \cup D_{S_2}$. □

The dependence relation D_S over the set X of tuples then gives rise to the following equivalence relation on sequences s, s' over X; this equivalence gives an alternative representation of the partial order on input events.

Definition 11 (Equivalent sequences)**.** Let $D \subseteq X \times X$ be a symmetric relation. The equivalence relation $\equiv_D$ over sequences over X is the smallest equivalence relation (i.e. reflexive, symmetric, and transitive) such that (1) commuting independent items: for all $x, y \in X$, if *not* $x\, D\, y$, then $xy \equiv_D yx$; and (2) closure under (sequence) concatenation: for $s_1, s'_1, s_2, s'_2 \in X^*$, if $s_1 \equiv_D s'_1$ and $s_2 \equiv_D s'_2$ then $s_1 s_2 \equiv_D s'_1 s'_2$. For a schema S, two sequences s, s' are *equivalent with respect to* S, written $s \equiv_S s'$, if $s \equiv_{D_S} s'$. □

The structure of D_S reflects the hierarchical series-parallel structure of synchronization schemas. Note that not all binary relations on tuples of **headers**(S) can be represented. In particular, the (symmetric reflexive closure of) the relations $\{(A, B), (B, C), (C, D)\}$ and $\{(A, B), (B, C), (C, D), (D, A)\}$ do not have a hierarchical structure: here there is no way to choose a header out of A, B, C, D to be a root node in the synchronization schema.

We next describe a tight correspondence between sequences and series-parallel streams via dependence relations. First we have to define what it means for a sequence to be a flattening of a series-parallel stream; we then show that sequences up to equivalence exactly correspond to series-parallel streams.

Example 12. Let us revisit the schema in Example 4 (see also an example SPS in Figure 1). Suppose r_1, r_2 are `GPS` events of the red taxi, b_1, b_2 are `GPS` events of the blue taxi, r_1', r_2' are `RideCompleted` events of the red taxi, b' is a `RideCompleted` event of the blue taxi, and *eoh* is an end-of hour event. Following equivalent sequences

$$r_1, r_1', b', b_1, r_2', b_2, r_2, eoh \quad\equiv\quad b_1, b_2, r_1, r_2, b', r_1', r_2', eoh$$

are flattening of the SPS

$$[\langle\{\texttt{red} \mapsto [\bot, r_1, \bot, r_2, \bot],\\ \texttt{blue} \mapsto [\bot, b_1, \bot, b_2, \bot]\}, \{r_1', r_2', b'\}\rangle, eoh, \bot]. \qquad \Box$$

Definition 13 (Flattening)**.** Let S be a synchronization schema, and let t be a series-parallel stream over S. A *flattening* s of t is a sequence of tuples of type **headers**(S) given inductively as follows:

- If $S = \mathrm{Bag}(\mathcal{H})$, then s is a flattening of t if and only if the multiset of events in s equals t.
- If $S = \mathrm{Sync}(\mathcal{H}, S_1)$, and suppose $t = [t_0, d_1, t_1, \cdots, d_m, t_m]$ for some sub-streams t_i. Then s is a flattening of t if and only if $s = s_0 d_1 s_1 \ldots d_m s_m$ for some sequences $s_0, s_1, \ldots, s_m$ where s_i is a flattening of t_i for each i.
- If $S = \mathrm{PartitionBy}(K, S')$, and suppose t is a set with finitely many entries $v_i \mapsto t_i$ for $i = 1, \ldots, m$. Then s is a flattening of t if and only if s is an interleaving of the sequences $s_1, s_2, \ldots, s_m$ where s_i is a flattening of t_i for each i.
- If $S = \mathrm{Par}(S_1, S_2)$, and suppose t is a parallel composition of t_1 and t_2. Then s is a flattening of t if and only if s is an interleaving of the sequences s_1 and s_2 for some s_1, s_2 where s_1 is a flattening of t_1 and s_2 is a flattening of t_2. □

The following proposition formalizes the connection between a series-parallel stream and its flattenings. In particular, given a sequence of tuples, once we fix a schema, there is a unique way to interpret it as a series-parallel stream.

Proposition 14. Let S be a synchronization schema. (1) For every sequence s of tuples of type **headers**(S), there exists a unique (up to equality) $t : S$ such that s is a flattening of t. (2) For all sequences s_1, s_2 of tuples of type **headers**(S) and $t : S$, (a) if $s_1 \equiv_S s_2$ and s_1 is a flattening of t then s_2 is a flattening of t also, and (b) if s_1 and s_2 are both flattenings of t then $s_1 \equiv_S s_2$. □

2.5 Schema Relaxation

We end the section by looking at how dependence relations defined by synchronization schemas relate to each other: in particular, this allows defining what it means for one synchronization schema to be weaker or stronger than another in terms of the ordering requirements it imposes on sequences. Relaxation can be viewed as a sub-typing relation.

Definition 15 (Schema relaxation)**.** For synchronization schemas S_1 and S_2, S_1 is a *relaxation* of S_2, written $S_1 \lesssim S_2$, if **headers**$(S_1) \supseteq$ **headers**(S_2) and for all tuples x, y : **headers**(S_2), if $xD_{S_1}y$ then $xD_{S_2}y$. Two synchronization schemas S_1 and S_2 are *order-equivalent*, denoted $S_1 \sim S_2$, if $D_{S_1} = D_{S_2}$. (Equivalently, if both $S_1 \lesssim S_2$ and $S_2 \lesssim S_1$.) □

Example 16. Revisiting the schema of Figure 2, suppose we want to say that the ordering of `GPS` events of the same taxi is also not important. This can be captured by the schema

$$\mathrm{Sync}(\texttt{EndOfHour}, \mathrm{Par}(\mathrm{PartitionBy}(\texttt{taxiID}, \mathrm{Bag}(\texttt{GPS})), S_2))$$

which is a relaxation of the original schema. Such a schema will restrict the allowed computations, but increase the parallelization opportunities. This revised schema is equivalent to the schema $\mathrm{Sync}(\texttt{EndOfHour}, \mathrm{Bag}(\{\texttt{GPS}, \texttt{RideCompleted}\}))$

Assuming that we only have the GPS events of a single taxi together with the `EndOfHour` tuples, the following two schemas are order-equivalent.

$$\mathrm{Sync}(\texttt{EndOfHour}, \mathrm{Seq}(\texttt{GPS}))$$
$$\mathrm{Seq}(\{\texttt{EndOfHour}, \texttt{GPS}\})$$

This means that they describe the same stream partial orders. Their difference is only relevant for defining hierarchical queries. □

Proposition 17. If $t : S$ and $S' \lesssim S$, then there exists a unique $t' : S'$ such that every flattening of t is a flattening of t'.

Proposition 18. Schema relaxation, that is on two input schemas S_1 and S_2, checking if $S_1 \lesssim S_2$, and consequently checking schema equivalence, is decidable in quadratic time. □

3 SPS-TRANSFORMERS

In this section, we define *SPS-transformers* (SPSTs), a programming language for computations over series-parallel streams. If synchronization schemas are provided as types for the input and output streams of a computation, then an SPST is a (deterministic) function from the input to the output, which respects the structure given by the types. We build on this language in Section 4 to show how it can be used to define typical streaming transformations of interest. Before defining the language constructs, we begin with a discussion of our *design goals*: what properties should computations written in this language satisfy? We identify the following design goals. First, transformations over series-parallel streams should *respect* the **parallelism** in the input and output: parallel input events should be processed in parallel, and parallel input threads should produce parallel output events. For example, given an input which is two streams in parallel, the computation should be written in such a way that the two streams are processed separately, and outputs corresponding to them should be unordered. Second, to allow for the specification of potentially complex computations, we additionally want our language to be **modular**: it should be natural to construct a computation by combining sub-computations. For example, processing a stream of the hierarchical type $\mathrm{Sync}(\mathcal{H}, S)$ should be definable, both syntactically and semantically, in terms of existing computations defined over sub-streams of type S. Finally, any computation in our language should be **streamable**: it should process the input in one pass, producing output incrementally.

To satisfy these design goals, we make the following technical choices. First, to satisfy the parallelism goal, we define SPSTs to have SPS as input and output rather than sequential objects. The input being an SPS allows us to specify the computation to exploit parallelism, and the output being an SPS requires that we respect parallelism when producing output events.

To understand the challenges in defining the semantics due to the interplay between streamability and modularity, consider a transformer P processing hierarchical streams of type $S = \mathrm{Sync}(\mathcal{H}, S')$ that we would like define in terms of a transformer P' processing streams of type S'. Consider an input stream $t = [\bot, d, t']$ of type S for an $\mathcal{H}$-tuple d and stream t' of type S'. Suppose we want to extend the input stream t with a tuple d'. If the type of d' is one of the headers appearing in S', then it really extends the sub-stream t', and should be processed by the transformer P'. For streamability we want to make sure that, while processing d', P' extends the output stream only by adding new items. Formally, this means that the output of P' on the input stream t' should be a prefix of its output on the stream $t' \circ d$. With this motivation, we define such a semantics, which we call *open* semantics, for transformers as functions from input to output streams, and ensure that it is *monotonic* with respect to prefix ordering (see Theorem 25). But now suppose that the item d' is an $\mathcal{H}$-tuple that acts as a synchronization marker for the events in the sub-stream t'. Then to process it, the transformer P' should return, and let the top-level transformer P process the item d'. During this return, the transformer P' can do additional computation and produce additional output items even though the stream it has processed is still t'. This is a typical case when S' corresponds to key-based partitioning, and the arrival of the synchronization marker d' triggers the reduce operation that aggregates the results of the computations of the key-indexed sub-streams of t'. This though requires us to define another semantics of the transformer P' on the input stream t' that extends the open semantics and includes the results of the computation upon return. We call it *closed* semantics to indicate that it is applicable when the current stream is being closed. Note that the result of computation of P on the stream $[\bot, d, t', d', \bot]$ can be described by relying on the closed semantics of P' on the stream t'. In terms of existing work on punctuation, the closed semantics can be thought of as the stream output on a stream terminated by an *end-of-stream* marker.

Finally, an SPST is a function on pairs: it takes an initial value and an input SPS to a final value and an output SPS. We need this for modularity: without the initial value as input, an SPS-transforer on a sub-stream of the input could not be initialized based on the surrounding context. Similarly, the final value (separate from the series of output items produced) can be used to describe a summary of the input stream to be used in the surrounding context when the computation finishes.

We summarize all of these choices in the following definition of the *interface* for an SPST. We also define subtyping for the interface, where the output is relaxed. Each of the language constructs will then implement this interface. In the Appendix, we give an extended example to illustrate the formal definitions in this section.

Definition 19 (SPS-transformer interface)**.** An SPS-transformer (SPST) P has:

- A *type* denoted (X, S, X', S'), where X is the type for the initialization value, S is an input synchronization schema, X' is a type for the final return value, and S' is an output synchronization schema. We write $P : (X, S, X', S')$.
- An *open semantics* denoted $[\![P]\!]_O(x, t) = t'$, where $x : X$ is the initial value, $t : S$ is the input SPS, and $t' : S'$ is the incrementally produced output SPS.
- A *closed semantics* denoted $[\![P]\!]_C(x, t) = (x', t')$, where $x' : X'$ is the initial value, $t : S$ is the input SPS, $x' : X'$ is the final value, and $t' : S'$ is the output SPS. We additionally enforce that the open semantics is a prefix of the closed semantics: $[\![P]\!]_O(x, t) \preceq t'$. □

Definition 20. If $S'' \lesssim S'$ (Definition 15), then (X, S, X', S'') is a *subtype* of (X, S, X', S'). If $P : (X, S, X', S')$ then we also write $P : (X, S, X', S'')$. The open and closed semantics are derived as the unique output stream given by Proposition 17. □

In the remainder of the section, we give one language construct corresponding to each constructor of the input series parallel stream. Some additional notation: for a set of headers $\mathcal{H}$, we write $\mathbf{tup}(\mathcal{H})$ for the set of tuples $x : \mathcal{H}$. For a synchronization schema S, we write $\mathbf{sps}(S)$ for the set of series-parallel streams $t : S$. We write $\mathbf{bag}(X)$ for the set of bags (multisets) of items of type X.

3.1 Relational SPST

We start with the relational SPST, which represents a standard relational operator that can be used to process a bag of items, producing another bag of items. Relational operators are well studied and are commonly defined using SQL and its extensions. Our design choice here is to not impose a particular relational base language or SQL variant; instead, the relational operator is given as two *black-box* functions, which define the open and closed semantics, respectively. We only require that these are functions on bags (i.e. independent of the input order), and that the open semantics is monotone and a prefix of the closed semantics.

Definition 21 (Relational SPST)**.** A relational SPST

$$P : (X, \mathrm{Bag}(\mathcal{H}), X', \mathrm{Bag}(\mathcal{H}'))$$

consists of two fields:

$$P.\mathrm{open} : X \times \mathbf{sps}(\mathrm{Bag}(\mathcal{H})) \to \mathbf{sps}(\mathrm{Bag}(\mathcal{H}'))$$
$$\text{and} \quad P.\mathrm{closed} : X \times \mathbf{sps}(\mathrm{Bag}(\mathcal{H})) \to X' \times \mathbf{sps}(\mathrm{Bag}(\mathcal{H}')).$$

such that (1) P.open is *monotone*: if $r_1 \preceq r_2$, then $P.\mathrm{open}(x, r_1) \preceq P.\mathrm{open}(x, r_2)$; and (2) P.open is a prefix of P.closed: if $P.\mathrm{closed}(x, r) = (x', r')$ then $P.\mathrm{open}(x, r) \preceq r'$. The semantics of P is defined as $[\![P]\!]_O(x, r) = P.\mathrm{open}(x, r)$ and $[\![P]\!]_C(x, r) = P.\mathrm{closed}(x, r)$. □

3.2 Parallel SPST

We now define the inductive SPSTs. An SPST processing inputs of type $\mathrm{Par}(S_1, S_2)$ is composed two SPSTs running in parallel independently. The question here is, can the components SPSTs produce tuples of the same type? The answer is yes, provided such tuples, since they get produced independently, are summarized using a schema $\mathrm{Bag}(O)$, where O is a set of output headers. So the output schema for the parallel SPST will be $\mathrm{Par}(S_1', S_2', \mathrm{Bag}(O))$.

Definition 22 (Parallel SPST)**.** Let S_1, S_2, S_1', S_2' be schemas. A parallel SPST

$$P : (X, \mathrm{Par}(S_1, S_2), X', \mathrm{Par}(S_1', S_2', \mathrm{Bag}(O')))$$

consists of internal types X_1, X_2, X_1', X_2' and four fields:

$$\begin{aligned} P.\mathsf{left} &: (X_1, S_1, X_1', \mathrm{Par}(S_1', \mathrm{Bag}(O'))), \\ P.\mathsf{right} &: (X_2, S_2, X_2', \mathrm{Par}(S_2', \mathrm{Bag}(O'))), \\ P.\mathsf{init} &: X \to X_1 \times X_2, \quad \text{and} \quad P.\mathsf{fin} : X_1' \times X_2' \to X'. \end{aligned}$$

The semantics of P is as follows: if we have that $P.\mathsf{init}(x) = (x_1, x_2)$, $[\![P.\mathsf{left}]\!]_O(x_1, t_1) = \langle t_1', r_1' \rangle$, and $[\![P.\mathsf{right}]\!]_O(x_2, t_2) = \langle t_2', r_2' \rangle$, where $r_1', r_2' : \mathrm{Bag}(O')$, and additionally $[\![P.\mathsf{left}]\!]_C(x_1, t_1) = (x_1', \langle t_1'', r_1'' \rangle)$ and $[\![P.\mathsf{right}]\!]_C(x_2, t_2) = (x_2', \langle t_2'', r_2'' \rangle)$, then

$$\begin{aligned} [\![P]\!]_O(x_1, t_1) &= \langle \langle t_1', t_2' \rangle, r_1' \cup r_2' \rangle \\ [\![P]\!]_C(x_1, t_1) &= (P.\mathsf{fin}(x_1', x_2'), \langle \langle t_1'', t_2'' \rangle, r_1'' \cup r_2'' \rangle). \quad \square \end{aligned}$$

3.3 Hierarchical SPST

When the input schema is $S = \mathrm{Sync}(\mathcal{H}, S_1)$, we want to define the corresponding SPST P parameterized by a sub-SPST from S_1 to S_1'. The SPST P maintains its own state that gets updated sequentially whenever any of the $\mathcal{H}$-tuple is processed, is passed to the sub-SPST when called, and updated when the sub-SPST returns. The output schema of P has the same structure as the input: it is divided into synchronizing events and non-synchronizing events. On input synchronization events, any output tuple may be produced, including a synchronization event; but on input sub-stream events, it would be incorrect to produce an output synchronizing event, as this would not be produced in a consistent order. The distinction between closed and open semantics plays a key role here: synchronizing events, when processed by P, "close" the computation of the sub-SPST. To formalize this inductively, we introduce an auxiliary semantics $[\![P]\!]_{Aux}(y, t)$ where the output is an internal states (rather than a final values), and in which the input stream ends with a d_i event, i.e. the final t_i is $\bot$.

Definition 23 (Hierarchical SPST)**.** Let S_1 and S_1' be schemas, and $\mathcal{H}$ and $\mathcal{H}'$ be a set of input and output headers, respectively. Let $S' = \mathrm{Sync}(\mathcal{H}', S_1')$. A hierarchical SPST

$$P : (X, \mathrm{Sync}(\mathcal{H}, S_1), X', \mathrm{Sync}(\mathcal{H}', S_1'))$$

consists of internal types X_1, X_1', Y and six fields:

$$\begin{aligned} &P.\mathsf{sub} : (X_1, S_1, X_1', S_1'), \\ &P.\mathsf{update} : Y \times \mathbf{tup}(\mathcal{H}) \to Y \times \mathbf{sps}(S'), \\ &P.\mathsf{call} : Y \to X_1, \qquad P.\mathsf{return} : Y \times X_1' \to Y, \\ &P.\mathsf{init} : X \to Y, \qquad \text{and} \quad P.\mathsf{fin} : Y \to X' \times \mathbf{sps}(S'). \end{aligned}$$

The auxiliary semantics of P is denoted $[\![P]\!]_{Aux}(y, t) = (y', t')$, where $y, y' : Y$, and defined inductively *only* for t of the form $[t_0, d_1, t_1, \ldots, d_m, \bot]$. For the base case, $[\![P]\!]_{Aux}(y, \bot) = (y, \bot)$. Then inductively, if $[\![P]\!]_{Aux}(y, t) = (y', t')$, $t_1 : S_1$, and $d : \mathcal{H}$, and if we have $P.\mathsf{call}(y') = x_1$, $[\![P.\mathsf{sub}]\!]_C(x_1, t_1) = (x_1', t_1')$, $P.\mathsf{return}(y', x_1') = y''$, and $P.\mathsf{update}(y'', d) = (y''', t'')$, then $[\![P]\!]_{Aux}(y, t \circ [t_1, d, \bot]) = (y''', t' \circ t_1' \circ t'')$. Given the auxiliary semantics, we define the semantics of P on a trace decomposed as $t \circ [t_1]$, where t ends in an empty sub-trace. Let $P.\mathsf{init}(x) = y$, $[\![P]\!]_{Aux}(y, t) = (y', t')$, and $P.\mathsf{call}(y') = x_1$. Additionally, let $[\![P.\mathsf{sub}]\!]_C(x_1, t_1) = (x_1', t_1')$, $P.\mathsf{return}(y', x_1') = y''$, and $P.\mathsf{fin}(y'') = (x', t'')$. Then:

$$\begin{aligned} [\![P]\!]_O(x, t) &= t' \circ [\![P.\mathsf{sub}]\!]_O(x_1, t_1) \\ [\![P]\!]_C(x, t) &= (x', t' \circ t_1' \circ t''). \quad \square \end{aligned}$$

3.4 Partitioned SPST

Finally, we define SPST for the partition-by case. The idea here is analogous to the parallel composition $\mathrm{Par}(S_1, S_2)$ case: each substream corresponding to a different key value may produce output corresponding to that key value, *or* produce output corresponding to a common bag of tuples O'. The partitioned SPST initializes the state of $P.\mathsf{sub}$ for each key with a non empty series parallel stream and runs the child SPST for each (non-empty) key in parallel. We additionally need an aggregation stage (applicable to the closed semantics only), in which we combine all of the partitioned states using a black-box relational operator $P.\mathsf{agg}$, similar to what was done in the relational SPST base case.

Definition 24 (Partitioned SPST)**.** Let $S = \mathrm{PartitionBy}(K, S_1)$ and $S' = \mathrm{PartitionBy}(K, S_1')$ be schemas, and O' a set of headers. A partitioned SPST

$$P : (X, \mathrm{PartitionBy}(K, S_1), X', \mathrm{Par}(\mathrm{PartitionBy}(K, S_1'), \mathrm{Bag}(O'))$$

consists of internal types X_1, X_1' and three fields:

$$\begin{aligned} &P.\mathsf{sub} : (X_1, S_1, X_1', \mathrm{Par}(S_1', \mathrm{Bag}(O'))), \\ &P.\mathsf{init} : X \times \mathbf{tup}(K) \to X_1, \\ \text{and} \quad &P.\mathsf{agg} : X \times \mathbf{bag}((\mathbf{tup}(K) \times X_1') \to X' \times \mathbf{bag}(\mathbf{tup}(O')). \end{aligned}$$

For the semantics, suppose $t = \{v_1 \mapsto t_1, \ldots, v_m \mapsto t_m\}$, and for $i = 1, \ldots, m$, $P.\mathsf{init}(x, v_i) = x_i$, $[\![P.\mathsf{sub}]\!]_C(x_i, t_i) = (x_i', \langle t_i', r_i' \rangle)$, $P.\mathsf{agg}(x, \{(v_1, x_1), \ldots, (v_m, x_m)\}) = (x', r_0')$, and $[\![P.\mathsf{sub}]\!]_O(x_i, t_i) = \langle t_i'', r_i'' \rangle$. Then

$$\begin{aligned} [\![P]\!]_C(x, t) &= (x', \langle \{v_1 \mapsto t_1', \ldots, v_m \mapsto t_m'\}, r_0' \cup r_1' \cup \cdots \cup r_m' \rangle \\ [\![P]\!]_O(x, t) &= \langle \{v_1 \mapsto t_1'', \ldots, v_m \mapsto t_m''\}, r_1'' \cup \cdots \cup r_m'' \rangle. \quad \square \end{aligned}$$

3.5 Streamability

This brings us to our main theorem about SPSTs, defined using all of the above constructs, which captures the streamability (monotonicity) of the open semantics (proven in the Appendix).

Theorem 25. *Let $P : (X, S, X', S')$ be an SPST. Then P is monotone in the following sense: for any $x : X$ and $t, u : S$, if $t \preceq u$, then $[\![P]\!]_O(x, t) \preceq [\![P]\!]_O(x, u)$.* □

4 SPS QUERIES

In this section we define the notion of an SPS-query, an abstraction over SPS-transformers that can be used to define computation pipelines, and a set of useful SPS-queries that extend queries on both relations and sequences. The set of defined SPS-queries is analogous to a high level query language (such as SQL), the components of which can be composed to define complex queries and do not capture implementation details. SPS-transformers are analogous to an intermediate representation that abstractly describes implementation and can be manipulated by a compiler or optimizer

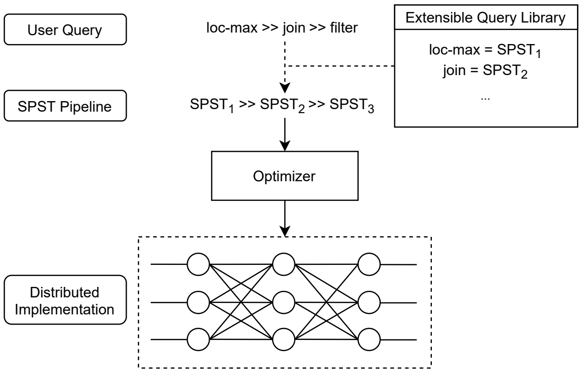

Figure 3: Architecture of an SPS framework.

to produce an efficient concrete implementation. An outline of a framework for SPS-queries is shown in Figure 3; users write SPS-queries, which are implemented as SPSTs, which are then given to an optimizer that produces a parallel implementation. We illustrate the expressive power of the query library by defining a few example queries, some of which are inspired from the literature, and others which showcase the capabilities of our framework.

We first define the *SPS-query* interface. An SPS-query does not use initial and final values, but only interacts with its environment by receiving an SPS as input and producing an SPS as output.

Definition 26 (SPS-Query)**.** Let S and S' be input and output schemas. An *SPS-query* $Q : (S, S')$ is an SPST of type $((), S, (), S')$ where the initial and final values are of type unit. The *evaluation of* Q on input $t : S$, denoted $Q(t)$, is t' where t' is the unique value such that $[\![Q]\!]_C((), t) = ((), t')$. □

As shown in Figure 3, users can sequentially compose SPS-queries using $\gg$ to create a pipeline, as long as the input schema of Q_i is a relaxation of the output schema of Q_{i-1}.

4.1 SPS-Query Library

We start by extending standard operators from the sequential and relational setting to the series parallel setting. This is not meant to be an exhaustive list, but rather an illustration of the capabilities of SPS-queries.

4.1.1 Filter. The filter SPS-query extends both the sequential notion of filtering (drawn from functional languages) as well as the `where` SQL operator. It is parametrized by a predicate that is used to drop all SPS elements for which it does not hold, preserving the hierarchical structure as is. Given a schema S and a predicate function $p : \mathbf{tup}(\mathcal{H}) \to \mathbb{B}$, where $\mathcal{H} = \mathbf{headers}(S)$, filter$(S, p)$ is an SPS-query with interface (S, S). For reference, we show the hierarchical case:

filter$(\mathit{Sync}(\mathcal{H}, S_1), p)$ is the hierarchical SPST P such that P.sub = filter(S_1, p), P.update is the same as in the sequential SPST, and the rest of the components simply return unit.

4.1.2 Map. The map SPS-query extends standard sequential functional maps to the series-parallel setting and can be also used to project fields of a tuple. It is parametrized by a function that transforms a single input tuple to a single output tuple.[1]

Given an input schema S and a mapping function $f : \mathbf{tup}(\mathcal{H}) \to \mathbf{tup}(\mathcal{H}')$, where $\mathcal{H} = \mathbf{headers}(S)$, map$(S, f)$ is an SPS-query with interface (S, S'). The schema S' has the same shape as S but all header sets $\mathcal{H}_i$ in S subexpressions are replaced by header sets $\mathcal{H}'_i$. The output header sets $\mathcal{H}'_i$ must be disjoint, and produced by f when processing the respective input headers, i.e., if $t \in \mathbf{tup}(\mathcal{H}_i), f(t) = t'$ then $t' \in \mathbf{tup}(\mathcal{H}'_i)$. The SPST definition of map is similar to filter, i.e., it is stateless and all the input, and output types are unit. For reference, we show the hierarchical case:

map$(\mathit{Sync}(\mathcal{H}, S_1), f)$ is the hierarchical SPST P such that P.sub = map(S_1, f), P.update$((), x) = ((), [f(x)])$, and the rest of the components return unit.

For the output schema to be well-formed (Definition 5), the following requirement needs to hold for f. If $t_1, t_2 \in \mathbf{tup}(S), f(t_1) = t'_1, f(t_2) = t'_2$, and $t'_1, t'_2 \in \mathcal{H}'_i$, then t_1, t_2 are also in the same header set $\mathcal{H}_i$. Furthermore, f needs to preserve the keys of tuples that were in partitioned in the input, i.e., if H is in the scope of k in S and $t \in \mathbf{tup}(H)$, then $t.k = f(t).k$. The above requirement can be easily satisfied if for example f is defined as the union of a function for each header set $\mathcal{H}$ in each Bag$(\mathcal{H})$, Sync$(\mathcal{H}, S')$ subexpression of the input schema.

4.1.3 Lift. It is often useful to define queries on a subschema, i.e., a subexpression, of the input schema. One such query could be one that processes the bags and sequences at the leafs of the input schema without affecting other parts of the input. We call this higher order query lift, since it lifts a query Q on a subschema by applying it to each substream of the subschema without affecting the other parts of the input stream.

Given an SPS-query $Q : (S_1, S'_1)$, and a schema S such that S_1 is a subschema of S and $(\mathbf{headers}(S) \setminus \mathbf{headers}(S_1)) \cap \mathbf{headers}(S'_1) = \emptyset$, lift$(S, S_1, Q)$ is the query with interface $(S, S[S'_1/S_1])$. The SPST that implements lift simply applies it to each relevant subschema, and keeps the rest of the input intact. For reference, we show two cases of its definition, the base case, where the input schema is the same as the target subschema, and the hierarchical case.

lift(S_1, S_1, Q) is the SPST that implements Q.

lift$(\mathit{Sync}(\mathcal{H}, S), S_1, Q)$ is the hierarchical SPST P such that P.sub = lift(S, S_1, Q), and the update function simply outputs each input tuple, P.update$((), x) = ((), [x])$.

4.1.4 Windowing. Windowing is a way to temporally partition input tuples in logical chunks. Windows are usually either defined using time, e.g., each window corresponds to a day, containing the tuples that happened during that day, or based on the values of the input tuples, e.g., each window corresponds to a stress episode that was identified due to unusually high values of blood pressure in the input stream.

Given a set of headers $\mathcal{H}$, a header $H \in \mathcal{H}$, and a schema $S = \mathrm{Sync}(\mathcal{H}, S_1)$, window$(S, H)$ is the SPS-query with interface (S, S') where:

$$S' = \mathrm{Sync}(H, \mathrm{Sync}(\mathcal{H} \setminus \{H\}, S_1))$$

[1] Returning a single output tuple for each input tuple is the standard map definition, but it could be straightforwadly extended to return multiple output tuples or none (similarly to the flatmap function in functional languages).

The window query is actually a relaxation of the input schema, and therefore it is not implemented as an SPST but is rather handled by the implementation. By combining it with lift, window can be applied to arbitrary synchronization schemas, and not just hierarchical ones.

4.1.5 Event Marking. Windowing restructures the input schema to make the windowing events explicit synchronization events, but for this to happen, we first need to introduce events that act as the window delimiters if they are not already in the stream. One could define several forms of sequential event marking on top of synchronization schemas. For example stress episode delimiters need to be inserted in the stream depending on the values of the input tuples, and the time based delimiters, e.g., `EndOfHour`, need to be inserted in between tuples that happened in different hours. These types of queries are often called complex event processing and here we focus on marking events using symbolic regular expressions.

Given a set of headers $\mathcal{H}$, a symbolic regular expression is defined by the grammar $r ::= \phi|(r \cup r)|(r \cdot r)|(r^*)$, where $\phi : \mathbf{tup}(\mathcal{H}) \to \mathbb{B}$. Interpreting a symbolic regular expression r gives us a predicate on sequences of $\mathcal{H}$-tuples, namely $[\![r]\!] : \mathbf{tup}(\mathcal{H})^* \to \mathbb{B}$. As an example, we could define a symbolic regular expression that identifies whether the value of a field f of a sequence of tuples crossed a threshold h two consecutive times after being below the threshold. Let $\phi_a(t) = \text{True}$ iff $t.f > h$ and let $\phi_b(t) = \neg\phi_a(t)$. Then, we can define the symbolic regex as $\phi_b \cdot \phi_a \cdot \phi_a$.

Building on symbolic regular expressions, we can define a marking query that marks a tuple t of a stream if a prefix ending in t matches a symbolic regex r. For example, if the input is given as $t_1, t_2, t_3, t_4, t_5, t_6, t_7, \ldots$ if r matches t_2, t_3, t_4, and t_3, t_4, t_5, t_6 the query would output $t_1, t_2, t_3, m(t_4), t_5, m(t_6), t_7, \ldots$. Given a set of headers $\mathcal{H}$, a symbolic regular expression r over $\mathcal{H}$, a marking function $m : \mathbf{tup}(\mathcal{H}) \to \mathbf{tup}(\mathcal{H}')$, and a synchronization schema $S = \text{Sync}(\mathcal{H}, S_1)$, where $H' \cap \mathbf{headers}(S_1) = \emptyset$, $\text{mark}(S, r, m)$ is the SPS-query with interface (S, S'), where $S' = \text{Sync}(\mathcal{H} \cup \mathcal{H}', S_1)$.

4.1.6 Aggregations. Aggregate operations that combine several data tuples to a single value are widely used in both the relational setting (SQL offers operations such as min, max, count, etc) and the streaming setting (usually in the form of some folding function). Aggregations can be generalized to the SPS setting, and can be specified using two components: (i) a way to select which part of the stream to aggregate, and (ii) a principled way to combine different parts of the aggregated stream subset. Synchronization schemas provide the first component, since synchronizing tuples are natural merge points that can finalize an aggregation of the sub-stream in between them. Therefore the user only has to define combination functions for different parts of the input stream.

As an example, we will define max, an aggregator that returns the tuple that has the largest value with respect to a function f for each aggregated substream. Since max is commutative and associative, we can define its input schema to be simply $\text{Sync}(\mathcal{H}, \text{Bag}(\mathcal{H}_1))$ without loss of generality, since $\text{Bag}(\mathcal{H}_1)$ relaxes any schema S where $\mathcal{H}_1 = \mathbf{headers}(S)$.

Given two sets of headers $\mathcal{H}, \mathcal{H}_1$, and a value projection function $f : \mathbf{tup}(\mathcal{H}_1) \to \mathbb{N}$, $\max(S, f)$ is the SPS-query with interface $(\text{Sync}(\mathcal{H}, \text{Bag}(\mathcal{H}_1)), \text{Seq}(\mathcal{H}_1))$. Given a relational SPST $\max_r : ((), \text{Bag}(\mathcal{H}_1), \mathbf{tup}(\mathcal{H}_1)^?, \emptyset)$ that simply returns the input tuple with the highest value according to f or $\bot$ if there the input is empty, we can define max as:

$max(S, f)$ is the hierarchical SPST P such that $P.\text{sub} = \max_r$, $P.\text{call}(\_) = ()$, $P.\text{ret}(\_, m) = m$, and $P.\text{update}(m, \_) = (\bot, [m])$.

We could extend the above procedure for aggregators that are not associative and commutative by defining aggregation SPSTs starting from the leafs and combining their values moving up the input schema. Note that we have made some choices when defining max: (i) the synchronizing tuple is dropped completely, assuming that it did not carry any information, e.g., if it was a time-based window marker, and (ii) all the input tuples of headers $\mathcal{H}_1$ are also dropped, assuming that they are not needed later on in the computation. These choices depend on the exact usage of the query, and we could define variants that preserve or drop tuples from the input depending on whether the following queries need them.

4.1.7 Join. Join operations on relations are widely used in queries on relational databases and can be directly lifted to our model by using lift defined above. However, the existence of temporal dimension in our model, i.e., ordering between relations that is induced by synchronizing events, allows us to define more expressive joins. As a concrete instance, we will define a join SPS-query that uses a symbolic regular expression r to match two relations of header H in the input stream and then joins them on the field f.

Consider a synchronization schema $\text{Sync}(\mathcal{H}, \text{Bag}(H))$, a symbolic regular expression r over $\mathcal{H}$, and a field name $f \in H$ to join on. Suppose H' is the type of the result of joining two relations of type H on the field $H.f$. Then, $\text{join}(\mathcal{H}, H, r, f)$ is the SPS-query with the interface $(\text{Sync}(\mathcal{H}, \text{Bag}(H)), \text{Sync}(\mathcal{H}, \text{Bag}(H')))$. The semantics of join is that when processing an input stream $bag_1 \cdot t_1 \cdot bag_2 \cdot t_2 \cdots bag_j \cdot t_j$, it finds the shortest sequence $t_i, \ldots, t_j$ that matches r and then it joins the bag_i and bag_j. If no such sequence exists, it produces an empty bag $\bot$. For example, consider that we want to join the events of relation H with the events of H a week earlier. Then if `EOD` is a predicate that matches tuples of header `EOD` (EndOfDay), we can write $\text{join}(\texttt{EOD}, H, \texttt{EOD}^8, f)$ On the other hand, if we wanted to join events belonging to consecutive sessions (delimited by `SOE` (StartOfSession) and `EOE` (EndOfSession) tuples, we can use the following (where $\cdot$ denotes symbolic regex concatenation) $\text{join}(\{\texttt{SOE}, \texttt{EOE}\}, H, \texttt{EOE} \cdot \texttt{SOE} \cdot \texttt{EOE}, f)$.

Implementation of such join queries as SPST can be done with concrete performance bounds. In particular, it is well-known that a symbolic regular expression can be translated to a (nondeterministic) symbolic finite automaton (SFA) with a linear number of states [68]. Using this result, evaluation of the regular expression part of the join query can be done by keeping track of the most recent matched relation (r_i corresponding to most recent t_i) at each state. This means that evaluation of our join query requires storing at most $O(n)$ relations, where n is the size of the symbolic regular expression (3 in above example). These relations are stored as part of the sequential state in the definition of the hierarchical SPST.

We can define a union query that performs the same symbolic regular expression matching as join and only differs in the binary relation operation that it performs on the matched relations. As an example, we can use union to define sliding event based windows. Assuming that the schema of our input stream is $\text{Sync}(\texttt{EOD}, \text{Bag}(\{H\}))$, we can use $\text{union}(\texttt{EOD}, H, \texttt{EOD} \cdot \texttt{EOD})$ to get a sequence of bags of the events of every two consecutive days, that can then be followed by an aggregation query.

4.1.8 Map Reduce. Programming abstractions that enable parallel processing are commonly used by both the streaming and batch data processing settings. A very common such abstraction is MapReduce [20]. A MapReduce computation first applies a stateless map on all input tuples (see Section 4.1.2), transforming them in preparation for the reduce stage. The reduce stage, then partitions all the output tuples produced by map based on a key, and applies a commutative and associative reduce function on all the tuples with the same key, producing possibly many tuples for each key.

MapReduce computations can be straightforwardly described as SPSTs: a map and a reduce : $(\mathrm{PartitionBy}(K, \mathrm{Bag}(\mathcal{H})), \mathrm{Bag}(\mathcal{H}'))$. Note that the partitioning in the input schema is not necessary and could just be written as $\mathrm{Bag}(\mathcal{H})$ since these two schemas are order equivalent, but we write it to explicitly show the partitioning that is done per key. A composition of map and reduce can then be lifted, so that it is applied to the selected sub-streams of the input stream.

4.1.9 Time Series Maximum. We will now describe loc-max, a query on time series that checks if an element in a sequence is a local maximum. The interesting characteristics of this query is that it requires lookahead, namely it needs to see tuples later down the sequence before processing a tuple. This has the implication that it needs to delay outputting other tuples in the stream, to preserve their order, even though it does not need to modify them.

The behavior of delaying output is captured by $\mathrm{collect}(S)$, an SPS-transformer of type $((), S, S, \emptyset)$ that simply collects all tuples of the SPS and returns them as a final value in their parsed form. Using $\mathrm{collect}(S)$, we can define the time series maximum query. Given a synchronization schema S, a set of headers $\mathcal{H}$, and a function $f : \mathbf{tup}(\mathcal{H}) \to \texttt{int}$, $\text{loc-max}(S, f)$ is an SPS-query with interface (S, S') where $S = \mathrm{Sync}(\mathcal{H}, S_1)$, $S' = \mathrm{Sync}(\mathcal{H}', S_1)$, and the headers in $\mathcal{H}'$ are the headers in $\mathcal{H}$ extended with a boolean field `lmax`, which indicates if they are local maxima. It is defined using the following SPST:

loc-max(S) is the hierarchical SPST P such that, $P.\mathrm{sub} = \mathrm{collect}$, $P.\mathsf{call}(\_) = ()$, and $P.\mathrm{ret}((d_1, s_1, d_2, \bot), s_2) = (d_1, s_1, d_2, s_2)$. Let $d' = \mathrm{lmax}(d_1, d_2, d_3)$ extending d_2 with a `lmax` field that indicates whether it is bigger than both d_1 and d_3, and finally $P.\mathrm{update}((d_1, s_1, d_2, s_2), d_3) = ((d_2, s_2, d_3, \bot), s_1 \circ d')$.

Assuming that the input is $[t_1, d_1, t_2, \ldots]$, where $d_i : \mathbf{tup}(\mathcal{H})$ and $t_i : S[v]$ sps streams, then loc-max always keeps d_{i-1}, d_i in state. When it sees $[t_{i+1}, d_{i+1}]$, then it checks if d_i is a local maximum, i.e., $f(d_{i-1}) < f(d_i)$ and $f(d_i) > f(d_{i+1})$, and outputs $[d_i', t_{i+1}]$.

4.2 Examples

In this section we describe example computations that can be expressed using synchronization schemas and SPS-queries. We start with a query from the NEXMark Benchmark [65] to illustrate how synchronization schemas can be used in existing computations, and we then move to an example that illustrates the benefits of synchronization schemas.

4.2.1 NEXMark Query 7. Figure 4 shows a query from the NEXMark challenge written in CQL. Every 10 minutes the query outputs the item with the highest price in the last 10 minutes. To achieve that, it uses a 10 minute tumbling window on the input stream `Bid`, finds the maximum price in this window, and then joins the items in the window with a price that is equal to the maximum price to find their item identifiers `itemid`.

```
SELECT
  Rstream(B.price, B.itemid)
FROM
  Bid [RANGE 10 MINUTE SLIDE 10 MINUTE] B
WHERE
  B.price =
    (SELECT MAX(B1.price) FROM BID
     [RANGE 10 MINUTE SLIDE 10 MINUTE] B1);
```

Figure 4: The CQL query for NEXMark Query 7 [13]

The input to this query can be described by the $S = \mathrm{Seq}(\texttt{Bid})$ synchronization schema, and its output by the $S' = \mathrm{Seq}(\texttt{Bid'})$ where `Bid'` only contains the `itemid` and `price` fields. Then the query itself, can be written as the following composition (we ignore the schema first arguments of each query):

$$\mathrm{time}(10m) \gg \mathrm{window}(\texttt{10m}) \gg \max((t) \to t.\texttt{price})$$

Let's look at this pipeline one query at a time. The first query $\mathrm{time}(10m)$ takes as input a stream of type $\mathrm{Seq}(\texttt{Bid})$ and interleaves its tuples with tuples of type `10m`, which denote the end of each 10 minute period, producing a stream of type $\mathrm{Seq}(\{\texttt{Bid}, \texttt{10m}\})$. The second query, $\mathrm{window}(\texttt{10m})$, simply produces a stream of type $\mathrm{Sync}(\texttt{10m}, \mathrm{Seq}(\texttt{Bid}))$ and it just used to indicate that the `10m` tuples act as window barriers. The third query, $\max((t) \to t.\texttt{price})$ takes as input a relaxation of the output of the second query, i.e., $\mathrm{Sync}(\texttt{10m}, \mathrm{Bag}(\texttt{Bid}))$, and it aggregates all of the tuples in each substream delimited by a `10m` tuple by keeping the one with the maximum `price`, and then replaces the `10m` tuple with it.

4.2.2 COVID Cases: Top Contributing States. In this subsection we are going to describe a query that showcases both sequential and relational computations. This query computes days that are local maxima with respect to COVID cases in the US, and then identifies which states were the top contributors in consecutive maxima days. Let's assume that the input is of type $\mathrm{Sync}(\texttt{EOD}, \mathrm{Bag}(\texttt{Case}))$ where `EOD` tuples denote the end of a day, and each `Case` tuple corresponds to a single positive COVID test case, including the state where the case happened in the field `stateId`. The complete query is a sequential composition of 5 queries, $Q_1, \ldots, Q_5$ and are described below.

Q1: The first step is to aggregate the cases of each state per day. This can be done by lifting a relational query on $\mathrm{Bag}(\texttt{Case})$ that (1) groups all `Case` tuples with respect to `Case.stateId`, and (2) counts the number of `Case` tuples for each state and adds it to a `cases` field. Such a relational query can be implemented using SQL. This gives us a schema of the form $\mathrm{Sync}(\texttt{EOD}, \mathrm{Bag}(\texttt{StateCases}))$ where `StateCases` contains a `stateId` and a `cases` field.

Q2: The second step is to smoothen the daily cases per state using a 7-day moving average, since cases are often misreported one or two days later than when they actually happened. To achieve this, we need to use an 7-way join, similar to the one described in

Section 4.1.7, that creates a cases field that contains the average of the 7 tuples per state that were joined. This gives us a schema of the form Sync(EOD, Bag(SmoothStateCases)) where StateCases and SmoothStateCases contain the same fields.

Q3: We then perform a *sum* aggregation (similar to max defined in Section 4.1.6), adding a field with the sum of all cases across the US to each EOD tuple. Instead of dropping the aggregated tuples, as we did with max, we keep them all since they will be required by a later query. We now have a schema of the form Sync(EOD, Bag(SmoothStateCases)), where EOD is extended with a cases field.

Q4: In order to identify which days are local maxima with respect to the number of cases, we use the loc-max query defined in Section 4.1.9. The output of this query is a schema of the form Sync(EOD, Bag(SmoothStateCases)), where EOD is extended with a boolean lmax field.

Q5: For each local maximum, we need to filter the states that have contributed the most to the cases of that day. We can achieve this by using a relational query, e.g., in SQL, that finds the highest contributing states, e.g., the ones that are above the median, or the ones that are one standard deviation over the mean, on Bag(SmoothStateCases). The output stream of Q5 is of the form Sync(EOD, Bag(ContribStates)) where the non-empty days are local maxima, and they contain the top contributing states.

Q6: Finally, we need to find the states that belong to the highest contributing states for two consecutive local maxima days. To achieve this, we use the following join query:

$$\text{join}(\texttt{ContribStates}, LM \cdot (\neg LM)^* \cdot LM, \texttt{stateId})$$

where LM matches on EOD tuples that are local maxima, that is when EOD.lmax = True, and the resulting bags of ConsecContribStates tuples only contain states that were among the highest contributors for the current and previous local maximum day.

4.2.3 COVID Cases: Monthly Status of each State. We now turn our attention to a query that was proposed in a recent paper [38] that studied COVID trajectories in the US trying to analyze surge occurrences. They propose a query that finds surge peaks and valleys for each US state, and then use those to determine the number of surges that a state has already passed, and whether they are on the rise or fall. We extend this query to report these results every month, together with the number of cases of the current surge that a state is in. An interesting characteristic of this query (that is supported by our model) is that it is based on event-based windows, i.e., computes an aggregate of the cases per state since its last valley, that are not aligned for different states. That is, surges for different states happen at different times, making this query difficult to describe with traditional windowing operations. Consider input of the form Sync(EOM, PartitionBy(stateId, Seq(Cases))), where EOM denotes the end of month, and Cases denotes the daily cases per state, smoothened using a 7-day average. The first step is to calculate the peaks and valleys for each state, using a query Q_1 that calculates them over a time series. Let S_1 = Seq(Cases) in:

$$\begin{aligned} Q_1 : (&\text{Sync}(\texttt{EOM}, \text{PartitionBy}(\texttt{stateId}, S_1)),\\ &\text{Sync}(\texttt{EOM}, \text{PartitionBy}(\texttt{stateId}, \text{Sync}(\{\texttt{Pk}, \texttt{Vl}\}, S_1)))) \end{aligned}$$

This query is described in [38] and simply identifies peaks and valleys in covid daily cases timeseries. A day is considered a peak if it contains a larger number of cases than its 7 previous and 7 next days. Therefore, this query needs to delay outputting both Cases and EOM tuples, similarly to loc-max, to preserve the output order.

We compose Q_1 with a second query Q_2 that aggregates the input to compute the state of surges for each state.

$$\begin{aligned} Q_2 : (&\text{Sync}(\texttt{EOM}, \text{PartitionBy}(\texttt{stateId}, \text{Sync}(\{\texttt{Pk}, \texttt{Vl}\}, S_1))),\\ &\text{Sync}(\texttt{EOM}, \text{Bag}(\texttt{StateSurgeState}))) \end{aligned}$$

The output header StateSurgeState contains four fields, the key stateId, the number of completed surges, whether the current surge is on a rise or fall, and the total cases of the current surge. We can define Q_2 inductively as an SPST from the bottom up:. The Seq(Cases) SPST simply folds over the cases and returns their sum. The Sync({Pk, Vl}, . . .) SPST has a state containing three components, a number denoting the number of completed surges, a boolean field denoting whether we are on a surge rise or fall, and a number indicating the sum of cases of the current surge, which is updated using substream SPST, and is reset everytime a Vl is encountered. The PartitionBy(stateId, . . .) SPST aggregates all states, outputs them as a bag, and propagates them to the top SPST. The top level SPST does not perform any computation, since EOM tuples are only used as synchronization points for producing output. When calling the underlying SPST, it initializes it with the returned state to continue the computation from the point it was paused.

5 RELATED WORK

There is a rich literature on querying and processing streaming data spanning two decades. Below we describe some representative efforts that have influenced our work.

Streaming database query languages: There is a large body of work on streaming database query languages and systems: Aurora [2] and its successor Borealis [1], STREAM [10], CACQ [49], TelegraphCQ [16], CEDR/StreamInsight [5, 12], and System S [33]. The query languages supported by these systems (for example, CQL [10], SPL [36], and [13]) are typically extensions 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 for richer computations that rely on the temporal ordering. A precise semantics for how to deal with out-of-order streams has been defined using *punctuations* (a type of synchronization markers) [29, 42, 46, 47, 66]. The partial ordering view supported by synchronization schemas 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.

Complex Event Processing: The literature on Complex Event Processing (CEP) is concerned with the recognition of complex patterns over streaming data (see a recent survey [28]). Some representative examples of CEP proposals are SQL-TS [59], SASE [31], Cayuga [14], and the MatchRegex operator [34] for SPL [35]. The patterns are typically given as queries that resemble regular expressions or as automata-based models. In some of these proposals a pattern can depend on the evolution of values: for example, a pattern where the price of a stock is constantly increasing. Such powerful event-selection capabilities supported by CEP languages can be incorporated in our framework in a modular fashion analogous to the marking using regular expressions and time-series maximum as described in Section 4.1.

Distributed Stream Processing: A number of distributed stream processing engines, such as Samza [25, 57], Storm [27, 64], Heron [43, 67], MillWheel [3], Spark Streaming [26, 70], and Flink [15, 24], 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 [4, 23] 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. Such an API provides great power but does not aid the programmer in reasoning about correctness of parallelization. Naiad [54] 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 [46] and deal with certain kinds of disorder, but they cannot encode more general dependencies expressed by synchronization schemas. Systems such as Flink [15, 24] and Naiad [54] support feedback cycles, while we have focused only on pipelines of queries due to the semantic complexities of cycles. General feedback cycles require a complex denotational model involving continuous functions and least fixpoints, as in Kahn process networks [39], and other more restricted forms of feedback (as in [32] and [50]) correspond to unique fixpoints.

Safe Parallelization: Prior work has considered the issue of semantically sound parallelization of streaming applications [37, 60]. The authors of [60] observe that Storm [27, 64] and S4 [55] 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 synchronization events have a similar role to the "pulses" of [60].

Partial-order Models: There is foundational work in concurrency theory dating back to Mazurkiewicz [53], where partially ordered sets of events are called traces. Mazurkiewicz traces have been studied from the viewpoint of algebra, combinatorics, formal languages and automata, and logic [22]. On a related note, partially ordered multisets (called Pomsets) have been proposed to model computations of concurrent systems [58], and a series-parallel stream in fact is a Pomset. Extending relational query languages to partially ordered multisets has been studied in [30], though not in the context of streaming. Viewing a stream transformer as a deterministic function over partially ordered structures was first proposed in [52]. In this work, a partially ordered stream is formalized as a *data trace* whose structure is derived from a dependency relation over events. In this model, a transformer is a function over data traces, and ensuring that it is consistent (that is, the result of the computation does not depend on the order in which independent data items are processed), parallelizable, and monotonic, is up to the programmer. A synchronization schema can be viewed as a specification language for certain kinds of dependencies, and series-parallel streams as data traces with a specific structure. The main benefit of this restricted structure is that it also suggests how to structure the streaming computation leading to the definition of SPSTs with all the desirable properties of modularity, monotonicity, parallelization, and composability.

6 RESEARCH DIRECTIONS

We have presented synchronization schemas, series-parallel streams, and SPSTs as a mathematical foundation for query languages and query processing for streaming computations. To conclude, we discuss directions for future research that can translate these ideas into a high-performance distributed stream processing system with an expressive API for specifying queries and an optimizing compiler with correctness guarantees.

6.1 Exploiting Parallelism in Implementation

The next step in our research agenda is to build a prototype implementation with a focus on exploiting parallelism, and evaluating the performance with respect to the existing stream processing systems such as Flink [15]. Figure 4 shows a plausible architecture for such a framework.

A synchronization schema succinctly captures parallelism opportunities and synchronization requirements that a compiler can exploit. For instance, consider the synchronization schema and the computation described in the illustrative example of Section 2.1. First, the `RideCompleted` events are processed using a relational operator, and existing techniques for high-performance parallel evaluation of relational queries can be directly used for our purpose. Second, the `GPS` events of different taxis can be evaluated in parallel. Implementing such a computation partitioned by keys requires keeping track of active keys, allocating computations corresponding to different keys in parallel and/or distributed architectures with load balancing, and collecting results when needed (while processing the `EndOfHour` event in this specific case). Existing techniques for systems that support Map-Reduce programming framework and stream processing systems supporting windowing constructs suggest solution strategies. Finally, the schema requires that all events must be consistently ordered with respect to `EndOfHour` events, and `GPS` events of the same taxi must be totally ordered. When events are generated at different nodes in a distributed system, imposing such an ordering is a challenge. However, this is a well-studied problem in distributed systems, dating back to Lamport's work on logical clocks [45], and we can build on existing protocols.

If the computation were to treat both `GPS` and `RideCompleted` events within an hour as a bag, then the computation can be expressed naturally with existing stream processing APIs such as Streaming SQL with scalable performance. However, since the order of `GPS` events of the same taxi is important, to express the desired computation using a variant of Streaming SQL, a user has to group `GPS` events of the same taxi per hour using a windowing construct, and then they would need to define a custom function that sorts the `GPS` events based on their timestamp. Our ongoing work [41] shows that such programming limitations are present even in dataflow-based frameworks such as Flink and Timely Dataflow [54]: for example, certain distributed machine learning applications are difficult to realize due to the complex combination of synchronizing and parallel events.

6.2 Theoretical Foundations for SPS-Queries

Foundational understanding of logics and transducers for sequences, relations, and trees has been influential in the design of query languages and optimizing compilers for corresponding data models. For instance, logics and automata over trees provide foundations for processing XML data [56, 61]. Series-parallel streams have an appealing mathematical structure, and deserve an analogous theoretical investigation.

A natural theoretical question related to expressiveness is: Is the model of SPSTs complete, that is, can it express all computable SPS-to-SPS transformations? Answering this question first requires alternative ways of formulating transformations over series-parallel streams. Observe that the hierarchical structure induced by synchronizing events in an SPS is reminiscent of *nested words* with visible markers for calls and returns (or matching open and close parantheses) [8]. The parallel composition of streams corresponds to ordered binary siblings, while the key-based-partitioning over streams corresponds to unordered unranked (that is, unbounded) siblings. Thus, the theories of unordered unranked trees, nested words, and relations can all provide insights for formulating a theory of series-parallel streams and such a theory can lead to a more principled design of query languages.

A challenge related to query evaluation is: Are there classes of query languages—both as acceptors and transformations, over series-parallel streams with guaranteed (parallel) complexity bounds on space and time needed to process the input SPS? The complexity of query evaluation is well understood for relational structures. For sequences, ordered trees, and nested words, monadic-second logic (MSO), automata, and MSO-definable transductions have been well studied, and provide a strong foundation for finite-state computations. Such models cannot capture computation over data (such as aggregates), and a potentially promising foundation is the formalism of *quantitative regular expressions* (QREs), and the corresponding MSO-definable stream transformations and tranducers models, that integrate regular parsing of sequences of data items with a user-defined set of data operations (such as aggregates) with precise bounds on query evaluation [6, 7, 9, 51]. It should be noted that a theoretical understanding of query evaluation for series-parallel streams should focus on *parallel* complexity rather than sequential, opening up new research opportunities.

6.3 Query Optimization

Theoretical foundations in relational algebra, and in particular, semantics-preserving algebraic rewrite rules, have been instrumental in highly effective optimization techniques for relational queries [17, 63]. Query optimization has also been heavily studied for streaming extensions of SQL [21], where notions of windows [10] or punctuation-delimited substreams [66] allow us to leverage the existing relational algebra. Techniques have been developed to adaptively reorder predicates [11], which in fact can be generalized to sliding window joins given the bounded size of windows. Other strategies can also be adapted from traditional query optimization, such as group-by pushdown [18] and aggregate selection [62] and incremental maintenance through derivation tracking [48].

These existing techniques provide useful insights for developing an optimizing compiler for SPS-queries. The combination of relational and temporal operators also opens up new opportunities. As an example of such opportunities, let us revisit the stages Q4 and Q5 of the computation described Section 4.2.2. Query Q4 tags days as local maxima by adding the field `lmax` to each `EOD` tuple. Since the value of this field gets determined only after all the `SmoothStateCases` events of the following day have been processed, and since the next computation stage also refers to `SmoothStateCases` events, such events belonging to the current need to be stored in state so that they can be emitted along with the marked `EOD` tuple. The query Q5 is a relational query that replaces each bag of `SmoothStateCases` with `ContribStates`. It turns out that these two queries commute without changing the semantics. Furthermore, since the number of `ContribStates` tuples is strictly smaller than the number of `SmoothStateCases` events each day, the state that needs to be stored due to delayed output for the local maxima computation is smaller if we commute the two. An optimizing compiler then should rewrite the query by commuting these two stages.

6.4 Testing and Verification

Given the difficulties in ensuring correctness of distributed systems, when used judiciously, formal methods can have impact on addressing challenging design problems in distributed systems. For example, the TLA specification methodology and associated tools have led to verified design and implementation of complex distributed protocols such as Paxos [44], and automated analysis tools are used in checking conformance with respect to security policies at Amazon Web Services [19]. We believe that distributed stream processing systems, such as Flink, Spark Streaming, and Storm, offer a — yet largely unexplored — opportunity for the development of tools for specification, testing, and verification. In particular, while such platforms are increasingly deployed in emerging data-driven computing systems, exploiting the parallelism afforded by them is prone to errors. The first step towards such tools is formalization of the computational model, and this is what the framework of synchronizations schemas and series-parallel streams offers.

Developing a *verified* distributed stream processing system, with performance comparable to the existing systems, is a long term goal. Meanwhile, our formalization can form the basis of tools for testing and debugging of existing systems. In recent work, we developed *DiffStream*, a tool for differential output testing for distributed stream processing systems, that is, checking whether two implementations produce equivalent output streams in response to a given input stream [40]. The notion of equivalence allows reordering of logically independent data items, which in retrospect can be formalized as type checking with respect to the output synchronization schema.

REFERENCES

[1] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stanley Zdonik. The design of the Borealis stream processing engine. In *Proceedings of the 2nd Biennial Conference on Innovative Data Systems Research (CIDR '05)*, pages 277–289, 2005.

[2] Daniel J. Abadi, Don Carney, Ugur Cetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: A new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, 2003.

[3] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. Millwheel: Fault-tolerant stream processing at internet scale. *Proc. VLDB Endow.*, 6(11):1033–1044, August 2013.

[4] 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. 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):1792–1803, August 2015.

[5] Mohamed Ali, Badrish Chandramouli, Jonathan Goldstein, and Roman Schindlauer. The extensibility framework in Microsoft StreamInsight. In *Proceedings of the 27th IEEE International Conference on Data Engineering (ICDE '11)*, pages 1242–1253, 2011.

[6] Rajeev Alur, Dana Fisman, Konstantinos Mamouras, Mukund Raghothaman, and Caleb Stanford. Streamable regular transductions. *Theoretical Computer Science*, 807:15–41, 2020.

[7] Rajeev Alur, Dana Fisman, and Mukund Raghothaman. Regular programming for quantitative properties of data streams. In *Proceedings of the 25th European Symposium on Programming (ESOP '16)*, pages 15–40, 2016.

[8] Rajeev Alur and P. Madhusudan. Adding nesting structure to words. *J. ACM*, 56(3):16:1–16:43, 2009.

[9] Rajeev Alur, Konstantinos Mamouras, and Caleb Stanford. Modular quantitative monitoring. *Proceedings of the ACM on Programming Languages*, 3(POPL):50:1–50:31, January 2019.

[10] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The CQL continuous query language: Semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142, 2006.

[11] Shivnath Babu, Rajeev Motwani, Kamesh Munagala, Itaru Nishizawa, and Jennifer Widom. Adaptive ordering of pipelined stream filters. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 407–418, 2004.

[12] Roger S. Barga, Jonathan Goldstein, Mohamed Ali, and Mingsheng Hong. Consistent streaming through time: A vision for event stream processing. In *Proceedings of the 3rd Biennial Conference on Innovative Data Systems Research (CIDR '07)*, pages 363–374, 2007.

[13] Edmon Begoli, Tyler Akidau, Fabian Hueske, Julian Hyde, Kathryn Knight, and Kenneth Knowles. One sql to rule them all-an efficient and syntactically idiomatic approach to management of streams and tables. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1757–1772, 2019.

[14] Lars Brenna, Alan Demers, Johannes Gehrke, Mingsheng Hong, Joel Ossher, Biswanath Panda, Mirek Riedewald, Mohit Thatte, and Walker White. Cayuga: A high-performance event processing engine. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, SIGMOD '07, pages 1100–1102, 2007.

[15] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache Flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[16] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Sam Madden, Vijayshankar Raman, Fred Reiss, and Mehul Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *Proceedings of the First Biennial Conference on Innovative Data Systems Research (CIDR '03)*, 2003.

[17] Surajit Chaudhuri. An overview of query optimization in relational systems. In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 1-3, 1998, Seattle, Washington, USA*, pages 34–43, 1998.

[18] Surajit Chaudhuri and Kyuseok Shim. Including group-by in query optimization. In *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*, pages 354–366, 1994.

[19] Byron Cook. Formal reasoning about the security of amazon web services. In *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I*, pages 38–47, 2018.

[20] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: a flexible data processing tool. *Communications of the ACM*, 53(1):72–77, 2010.

[21] Amol Deshpande, Zachary Ives, and Vijayshankar Raman. *Adaptive query processing*. Now Publishers Inc, 2007.

[22] Volker Diekert and Grzegorz Rozenberg, editors. *The Book of Traces*. World Scientific, 1995.

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

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

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

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

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

[28] Nikos Giatrakos, Elias Alevizos, Alexander Artikis, Antonios Deligiannakis, and Minos Garofalakis. Complex event recognition in the big data era: a survey. *The VLDB Journal*, 29(1):313–352, 2020.

[29] Michael Grossniklaus, David Maier, James Miller, Sharmadha Moorthy, and Kristin Tufte. Frames: data-driven windows. In *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*, pages 13–24, 2016.

[30] Stéphane Grumbach and Tova Milo. An algebra for pomsets. *Information and Computation*, 150(2):268–306, 1999.

[31] Daniel Gyllstrom, Eugene Wu, Hee-Jin Chae, Yanlei Diao, Patrick Stahlberg, and Gordon Anderson. SASE: Complex event processing over streams. In *Proceedings of the 3rd Biennial Conference on Innovative Data Systems Research (CIDR '07)*, pages 407–411, 2007.

[32] Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, Sep 1991.

[33] 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. IBM Streams Processing Language: Analyzing big data in motion. *IBM Journal of Research and Development*, 57(3/4):7:1–7:11, 2013.

[34] Martin Hirzel. Partition and compose: Parallel complex event processing. In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*, DEBS '12, pages 191–200, New York, NY, USA, 2012. ACM.

[35] Martin Hirzel, Henrique Andrade, Bugra Gedik, Gabriela Jacques-Silva, Rohit Khandekar, Vibhore Kumar, Mark Mendell, Howard Nasgaard, Scott Schneider, Robert Soulé, and Kun-Lung Wu. IBM Streams Processing Language: Analyzing big data in motion. *IBM Journal of Research and Development*, 57(3/4):7:1–7:11, 2013.

[36] Martin Hirzel, Scott Schneider, and Buğra Gedik. SPL: An extensible language for distributed stream processing. *ACM Trans. Program. Lang. Syst.*, 39(1), 2017.

[37] Martin Hirzel, Robert Soulé, Scott Schneider, Buğra Gedik, and Robert Grimm. A catalog of stream processing optimizations. *ACM Computing Surveys (CSUR)*, 46(4):46:1–46:34, March 2014.

[38] Nick James and Max Menzies. COVID-19 in the United States: Trajectories and second surge behavior. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 30(9):091102, 2020.

[39] Gilles Kahn. The semantics of a simple language for parallel programming. *Information Processing*, 74:471–475, 1974.

[40] Konstantinos Kallas, Filip Niksic, Caleb Stanford, and Rajeev Alur. Diffstream: differential output testing for stream processing programs. *Proc. ACM Program. Lang.*, 4(OOPSLA):153:1–153:29, 2020.

[41] Konstantinos Kallas, Filip Niksic, Caleb Stanford, and Rajeev Alur. Stream processing with dependency-guided synchronization. *arXiv preprint arXiv:2104.04512*, abs/2104.04512, 2021.

[42] Sailesh Krishnamurthy, Michael J. Franklin, Jeffrey Davis, Daniel Farina, Pasha Golovko, Alan Li, and Neil Thombre. Continuous analytics over discontinuous streams. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 1081–1092, New York, NY, USA, 2010. ACM.

[43] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthik Ramasamy, and Siddarth Taneja. Twitter Heron: Stream processing at scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 239–250. ACM, 2015.

[44] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[45] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[46] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A. Tucker. 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, pages 311–322. ACM, 2005.

[47] Jin Li, Kristin Tufte, Vladislav Shkapenyuk, Vassilis Papadimos, Theodore Johnson, and David Maier. Out-of-order processing: A new architecture for high-performance stream systems. *Proceedings of the VLDB Endowment*, 1(1):274–288, August 2008.

[48] Mengmeng Liu, Nicholas E Taylor, Wenchao Zhou, Zachary G Ives, and Boon Thau Loo. Recursive computation of regions and connectivity in networks. In *2009 IEEE 25th International Conference on Data Engineering*, pages 1108–1119. IEEE, 2009.

[49] Samuel Madden, Mehul Shah, Joseph M. Hellerstein, and Vijayshankar Raman. Continuously adaptive continuous queries over streams. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, SIGMOD '02, pages 49–60, New York, NY, USA, 2002. ACM.

[50] Konstantinos Mamouras. Semantic foundations for deterministic dataflow and stream processing. In Peter Müller, editor, *Proceedings of the 29th European Symposium on Programming (ESOP '20)*, volume 12075 of *Lecture Notes in Computer Science*, pages 394–427, Berlin, Heidelberg, 2020. Springer.

[51] Konstantinos Mamouras, Mukund Raghothaman, Rajeev Alur, Zachary G. Ives, and Sanjeev Khanna. 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, pages 693–708, New York, NY, USA, 2017. ACM.

[52] Konstantinos Mamouras, Caleb Stanford, Rajeev Alur, Zachary G. Ives, and Val Tannen. Data-trace types for distributed stream processing systems. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 670–685, 2019.

[53] Antoni W. Mazurkiewicz. Trace theory. In *Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986, Part II, Proceedings of an Advanced Course, Bad Honnef, Germany, 8-19 September 1986*, pages 279–324, 1986.

[54] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: A timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 439–455, New York, NY, USA, 2013. ACM.

[55] Leonardo Neumeyer, Bruce Robbins, Anish Nair, and Anand Kesari. S4: Distributed stream computing platform. In *Proceedings of the 2010 IEEE International Conference on Data Mining Workshops*, pages 170–177, Dec 2010.

[56] Frank Neven. Automata, logic, and XML. In *Computer Science Logic, 16th International Workshop, CSL 2002, 11th Annual Conference of the EACSL, Edinburgh, Scotland, UK, September 22-25, 2002, Proceedings*, pages 2–26, 2002.

[57] Shadi A. Noghabi, Kartik Paramasivam, Yi Pan, Navina Ramesh, Jon Bringhurst, Indranil Gupta, and Roy H. Campbell. Samza: Stateful scalable stream processing at LinkedIn. *Proceedings of the VLDB Endowment*, 10(12):1634–1645, August 2017.

[58] Vaughan Pratt. Modeling concurrency with partial orders. *International Journal of Parallel Programming*, 15(1):33–71, Feb 1986.

[59] Reza Sadri, Carlo Zaniolo, Amir Zarkesh, and Jafar Adibi. Expressing and optimizing sequence queries in database systems. *ACM Transactions on Database Systems*, 29(2):282–318, 2004.

[60] Scott Schneider, Martin Hirzel, Buğra Gedik, and Kun-Lung Wu. Safe data parallelism for general streaming. *IEEE Transactions on Computers*, 64(2):504–517, Feb 2015.

[61] Thomas Schwentick. Automata for XML - A survey. *J. Comput. Syst. Sci.*, 73(3):289–315, 2007.

[62] S. Sudarshan and Raghu Ramakrishnan. Aggregation and relevance in deductive databases. In *17th International Conference on Very Large Data Bases, September 3-6, 1991, Barcelona, Catalonia, Spain, Proceedings*, pages 501–511, 1991.

[63] David Toman and Grant Weddell. Fundamentals of physical design and query compilation. *Synthesis Lectures on Data Management*, 3(4):1–124, 2011.

[64] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, and Dmitriy Ryaboy. Storm @ twitter. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 147–156. ACM, 2014.

[65] Pete Tucker, Kristin Tufte, Vassilis Papadimos, and David Maier. NEXMark—a benchmark for queries over data streams. Technical report, Technical report, OGI School of Science & Engineering, 2008.

[66] Peter A. Tucker, David Maier, Tim Sheard, and Leonidas Fegaras. Exploiting punctuation semantics in continuous data streams. *IEEE Transactions on Knowledge and Data Engineering*, 15(3):555–568, 2003.

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

[68] Margus Veanes, Peli De Halleux, and Nikolai Tillmann. Rex: Symbolic regular expression explorer. In *2010 Third International Conference on Software Testing, Verification and Validation*, pages 498–507. IEEE, 2010.

[69] Zhanyong Wan and Paul Hudak. Functional reactive programming from first principles. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Vancouver, Britith Columbia, Canada, June 18-21, 2000*, pages 242–252, 2000.

[70] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 423–438, New York, NY, USA, 2013. ACM.

APPENDIX A: SPST EXAMPLES (SECTION 3)

To illustrate the definition of the various SPST constructs, we continue the example schema from Section 2.1 (Figure 2) as the input type. For the output, suppose we want to produce two kinds of events: `EndOfHour`, representing end-of-hour summaries, and `GPSOutlier`, representing outlier events that should be logged for further investigation. We describe building an SPST with this input and output, building it bottom-up from the structure of the input schema.

We begin with an example of a relational SPST. We describe the transformation on `RideCompleted` events which computes the sum of the costs of all completed hours. The interface of this SPST is $P_1 : ((), \mathrm{Bag}(\texttt{RideCompleted}), \texttt{float}, \emptyset)$: as it consumes a bag of `RideCompleted` events, does not produce any output tuples, but instead we aggregate the sum of the return costs as a single `float`. For this relational base case, the computation can be written using an aggregator in a base relational language such as SQL. Formally, in our framework P_1 is defined by two black-box functions: $P.\mathsf{open}((), r) = \bot$ and $P.\mathsf{closed}((), \{x_1, \ldots, x_m\}) = (x_1 + \cdots + x_m, \bot)$. The former component $P.\mathsf{open}$ indicates that in this case no events are produced incrementally (as the input stream is processed). The latter component $P.\mathsf{closed}$ indicates that the final result of the computation (after the entire input stream is seen) is the sum of all tuples in the input relation.

Next, we describe a simple sequential SPST which processes a linear sequence of `GPS` events. Recall that $\mathrm{Seq}(\mathcal{H})$ is a useful special case of hierarchical synchronization schemas that denote simple sequences, i.e. it is the schema $\mathrm{Sync}(\mathcal{H}, \mathrm{Bag}(\varnothing))$. Suppose we want to compute the distance traveled for a specific taxi given its `GPS` tuples; additionally, suppose we want to produce as output *outlier* `GPS` tuples, rather than including them in the aggregation.

$$P_2 : ((), \mathrm{Seq}(\texttt{GPS}), \texttt{float}, \mathrm{Seq}(\texttt{Outlier}))$$

P_2 keeps the last known location for the taxi and the current distance travelled as its state, and each time it processes a new `GPS` tuple, it updates both. Additionally, if the last known location is too far from the current one (> 1 below) instead of updating state it produces the tuple as output.

$$\begin{aligned}
P_2.\mathsf{update}((\bot, 0), gps) &= ((gps.loc, 0), \bot)\\
P_2.\mathsf{update}((loc, d), gps) &= ((gps.loc, d + dist(gps.loc, loc))\\
&\qquad\qquad \text{if } dist(gps.loc, loc) \leq 1\\
P_2.\mathsf{update}((loc, d), gps) &= ((loc, d), \texttt{Outlier}(loc)) \text{ otherwise}
\end{aligned}$$

Because this is a sequential base case (a special case of hierarchical), $P_2.\mathsf{sub}$, $P_2.\mathsf{call}$, and $P_2.\mathsf{return}$ are trivial with no effect on the state. Finally, $P_2.\mathsf{init}(()) = (\bot, 0)$, and $P_2.\mathsf{fin}(loc, d) = (d, \bot)$.

Next, we define the partitioned SPST that computes the total distance travelled by all taxis (according to the taxi example described in Section 2.1). The interface of the SPST is:

$$P_3 : ((), \mathrm{PartitionBy}(\texttt{taxiID}, \mathrm{Seq}(\texttt{GPS})), \texttt{float}, \mathrm{Bag}(\texttt{Outlier}))$$

since it returns the total distance travelled by all taxis in miles. The child SPST is P_2, i.e., $P_3.\mathsf{sub} = P_2$, However, notice that instead of a sequential output, here the output outliers are a bag: this is because there are multiple keys (taxi IDs), so different key outputs may be unordered. Implicitly, we are relaxing the output of P_2 to be a bag instead of a sequence: this illustrates SPST *subtyping* (Definition 20), in which ordered output events may be reinterpreted as unordered. The interface of our sequential SPST is now $P_2 : \mathrm{Seq}(\mathtt{GPS}), \mathtt{float}, \mathrm{Seq}(\mathtt{Outlier}))$. To fit the SPST definition exactly, we would additionally relax to $\mathrm{Par}(\varnothing, \mathrm{Seq}(\mathtt{Outlier}))$ (to allow both keyed and bag outputs), but we leave this off for presentation; it is just another application of subtyping since the schemas are equivalent. To complete the definition of P_3, the aggregation produces a sum of the distances:

$$P_3.\mathsf{agg}(\_, ds) = (sum(\{d \mid (\_, d) \in ds\}), \bot)$$

and $P_3.\mathsf{init}$ initializes all child SPSTs with the unit value.

At this point, we have a partitioned SPST P_3 for processing the key-partitioned `GPS` stream, and we have a relational SPST P_1 for processing the `RideCompleted` events. In order to combine these into an overall query which also processes the `EndOfHour` synchronizing events, we first need to combine these two streams in parallel. We define an SPST P_4 which divides the aggregate cost by the aggregate distance. Let $S_1 = \mathrm{PartitionBy}(\mathtt{taxiID}, \mathrm{Seq}(\mathtt{GPS}))$ and $S_2 = \mathrm{Bag}(\mathtt{RideCompleted})$. Then the interface of P_4 is:

$$P_4 : ((), \mathrm{Par}(S_1, S_2), \mathtt{float}, \mathrm{Bag}(\mathtt{Outlier})).$$

The SPST calls the underlying SPSTs P_1 and P_3: $P_4.\mathsf{left} = P_3$ and $P_4.\mathsf{right} = P_1$, which return the total distance covered by all taxis and the total cost of all completed rides in that hour, and then simply divides them to return the `float` ride cost per travelled mile, i.e., $P.\mathsf{fin}(dist, cost) = cost/dist$.

Notice that the average value in P_4 is only computed on finalization (after the entire stream is processed). In order to produce the same averages in a streaming manner, we need *synchronization events*, and this leads us to our final step: we complete the input schema in Figure 2 and the example by constructing a hierarchical schema which also processes the `EndOfHour` synchronization events. The schema P_5 which outputs the cost per distance travelled at the end of each hour has the following interface:

$$\begin{aligned} P_5 : ((), &\mathrm{Sync}(\mathtt{EndOfHour}, \mathrm{Par}(S_1, S_2)), \\ &(), \mathrm{Sync}(\mathtt{CostPerMile}, \mathrm{Bag}(\mathtt{Outlier}))) \end{aligned}$$

The SPST calls the underlying SPST P_4, i.e., $P_5.\mathsf{sub} = P_4$, which returns the cost per mile in the last hour as a `float`. P_4 also produces the `Outlier` output events. The internal state Y is the cost per mile from the last substream. The function $P_5.\mathsf{call}$ does not pass anything to P_4, but $P_5.\mathsf{return}$ does consume the final `float` and stores it in the state. Then P_5 simply outputs the `float` when processing an `EndOfHour` tuple:

$$P_5.\mathsf{update}(cpm, \_) = (cpm, \mathtt{CostPerMile}(cpm)).$$

APPENDIX B: PROOFS

Proof of Proposition 14

By induction on S. For $\mathrm{Bag}(\mathcal{H})$, all three conditions follow by the correspondence between a multiset of items and its linearizations. For $\mathrm{Par}(S_1, S_2)$ and for $\mathrm{PartitionBy}(K, S_1)$, we observe that sequences over tuples of $\mathbf{headers}(S)$ are interleavings of events each from a subschema, and all such interleavings are equivalent with respect to $\equiv_S$; conversely $\equiv_S$ only holds between different interleavings of the same two or more sequences up to equivalence, i.e. for parallel composition, if $s \equiv_S s'$ and s is an interleaving of s_1 and s_2 and s'_1 is an interleaving of s'_2, then $s_1 \equiv_{S_1} s'_1$ and $s_2 \equiv_{S_2} s'_2$. The most interesting case is $\mathrm{Sync}(\mathcal{H}, S_1)$. Here, we essentially apply the idea that $(a \cup b)^* = (a^* b)^* a^*$ for languages: in this context a is tuples of $\mathbf{headers}(S')$ and b is tuples of $\mathcal{H}$. So sequences over tuples of $\mathbf{headers}(S)$ decompose into a sequence of subsequences over S' delineated by $\mathcal{H}$ events, where there is one more subsequence than the number of $\mathcal{H}$ events. Since $\mathcal{H}$ events are fully dependent on everything else, this decomposition is not changed by $\equiv$, which can thus be identified with equality on the sequence of $\mathcal{H}$ events together with equivalence on each $\mathbf{headers}(S')$ substream. The definition of flattening reflects this decomposition exactly.

Proof of Proposition 17

We first show uniqueness. Let $t'_1, t'_2 : S'$ such that every flattening of t is a flattening of t'_1 and of t'_2. Since every SPS has at least one flattening, we can choose some particular flattening s of t (a sequence over $\mathbf{headers}(S)$). By uniqueness in Proposition 14-(1), since s is a flattening of both t'_1 and t'_2, $t'_1 = t'_2$.

Now we show existence. As before, choose a particular flattening s of t. Since $\mathbf{headers}(S') \supseteq \mathbf{headers}(S)$, s is also a sequence over $\mathbf{headers}(S')$. Thus by Proposition 14-(1), there exists a $t' : S'$ such that s is a flattening of s. It remains to show that all flattenings of t are flattenings of t'. Given any flattening $\hat{s}$ of t, by Proposition 14-(2b), $s \equiv_{D_S} \hat{s}$. By the definition of relaxation, $s \equiv_{D_{S'}} \hat{s}$. Then by Proposition 14-(2a), $\hat{s}$ is a flattening of t'.

Proof of Proposition 18

We may ignore the headers in S_1 and not in S_2. Then for each pair of headers H, H' in $\mathbf{headers}(S_2)$, we build a formula $\phi^1_{H,H'}$ whose free variables are the fields in H and the fields in H', which describes when $x D_{S_1} x'$ for $x : H$ and $x' : H'$. This is done by expanding out Definition 10. In all case the formula built is either true or false, or derived from a subcase, except in the $\mathrm{PartitionBy}(K, S)$ (and bag) cases where $x|_K = x'|_K$ arises as an atomic formula. Altogether, each $\phi^1_{H,H'}$ is just an atomic formula over the language of equality. We do the same for S_2 to get formulas $\phi^2_{H,H'}$. The problem now becomes checking a set of implications of atomic formulas over the language of equality, which is decidable by checking each implication $\phi^1_{H,H'} \to \phi^2_{H,H'}$ in turn and inspecting the variables τ_i where τ_i is a type present in an attribute $\langle \alpha_i : \tau_i \rangle$ of H or H'. The complexity is quadratic because there are quadratically many pairs H, H'.

Proof of Theorem 25

The proof is by induction on P. We strengthen the hypothesis to additionally show that the open semantics is a prefix of the closed semantics: if $[\![P]\!]_C(x, t) = (x', t')$ then $[\![P]\!]_O(x, t) \leq t'$. In addition to the definition of concatenation $\circ$ and prefix $\leq$, we use that $\leq$ is a partial order (Proposition 9). One of the inductive case is subtyping as given by Definition 20.

- In the relational case, P.open is monotonic and a subset relation of P.closed by assumption.
- In the parallel case, let $t = \langle t_1, t_2 \rangle$ and $u = \langle u_1, u_2 \rangle$, and suppose that we have $[\![P.\text{left}]\!]_O(x_1, t_1) = t'_1$, $[\![P.\text{left}]\!]_O(x_1, u_1) = u'_1$, $[\![P.\text{right}]\!]_O(x_2, t_2) = t'_2$, and $[\![P.\text{right}]\!]_O(x_2, u_2) = u'_2$. Applying the inductive hypothesis, what we need to show is that if $t'_1 \leq u'_1$ and $t'_2 \leq u'_2$, $\langle t'_1, t'_2 \rangle \leq \langle u'_1, u'_2 \rangle$. This follows by unfolding the definition of prefix and underlying concatenation, which works component-wise on $\langle t'_1, t'_2 \rangle$. The exact same reasoning applies to comparing the open and closed semantics.
- In the hierarchical case, our first step is to prove that the auxiliary semantics is monotonic. For this, we only consider when t and u each end in an empty sub-trace $t_m = \bot$ and $u_m = \bot$. This then follows by induction on the trace directly since the output is a sequence and produced one item at a time from the closed semantics of the sub-SPST, using transitivity of $\leq$. Next for the general case, we observe the following: for any trace ending in an empty subtrace t, and any subtraces t_1, u_1 with $t_1 \leq u_1$, the auxiliary semantics on t is a prefix of the open semantics on $t \circ [t_1]$ (by definition), which is a prefix of the open semantics on $t \circ [u_1]$ (by IH), which is a prefix of the auxiliary semantics on t concatenated with closed sub-SPST semantics on u_1 (by definition, IH, and associativity of concatenation), which is a prefix of the auxiliary semantics on $t \circ [u_1, d, \bot]$ for any d (by definition). This chain of prefix relations implies the general monotonicity for $t \leq u$, by induction using transitivity of $\leq$. Also the auxiliary semantics on t concatenated with closed sub-SPST semantics on u_1 is a prefix of the closed semantics on $t \circ [u_1]$ (by definition), which gives that the open semantics is a prefix of closed.
- Next we consider the partition case. For the open semantics, P.agg does not factor in. We consider the output on $t \circ u$ and t in two parts: first the keyed output, and second the relational output. (i) For the keyed output, we need to show that the output on t is a prefix of the output on $t \circ u$. There are three cases here: the key is present in both t and u, present in only t, and present in only u. If present in both, the prefix relation holds by induction hypothesis. If only in t, the output on t and on $t \circ u$ are the same as these SPS are the same for this particular key value. If only in u, the output on t does not contain this particular key value, and so is a prefix of the output on $t \circ u$ taking u' to be the output on u for that key. (ii) For the relational output, we consider the set of key values in t: for each such value, the output on t and on $t \circ u$ produces a relation. We can ignore key values not in t (in $t \circ u$ only) as they only extend the output relation for $t \circ u$. Now we need to show that the relational output on $t \circ u$ is a superset of the relational output on t for each of these keys, which is true by induction hypothesis.
- Finally, we consider the case of subtyping (output schema relaxation). This requires careful application of Definition 13, Proposition 14, and Proposition 17. Using these we derive the following lemma: given a schema, $t' \leq u'$ is equivalent to the following statement: every flattening of t' can be extended to a flattening of u', and every flattening of u' is equivalent to a extension of a flattening of t'.

  Given this lemma, let S and S' be the input and output schemas, and $S'' \lesssim S'$. Let t', u', t'', u'' be the output schemas for t and u: the definition of t'' and u'' is that all flattenings of t' are flattenings of t'', and all flattenings of u' are flattenings of u''. We also know by IH that $t' \leq u'$, which we interpret in terms of flattenings by the lemma. Considering any flattening of t'', first we know it only contains events in $\textbf{headers}(S')$ (because the original schema output was S'), and we can additionally show it is equivalent under S'' to some flattening of t'; this t' then can be extended to a flattening of u', so the flattening of t'' can be extended with the same extension to a flattening of u'', which by the lemma implies $t'' \leq u''$.