StreamQL: A Query Language for Processing Streaming Time Series

LINGKUN KONG, Rice University, USA
KONSTANTINOS MAMOURAS, Rice University, USA

Real-time data analysis applications increasingly rely on complex streaming computations over time-series data. We propose StreamQL, a language that facilitates the high-level specification of complex analyses over streaming time series. StreamQL is designed as an algebra of stream transformations and provides a collection of combinators for composing them. It integrates three language-based approaches for data stream processing: relational queries, dataflow composition, and temporal formalisms. The relational constructs are useful for specifying simple transformations, aggregations, and the partitioning of data into key-based groups or windows. The dataflow abstractions enable the modular description of a computation as a pipeline of stages or, more generally, as a directed graph of independent tasks. Finally, temporal constructs can be used to specify complex temporal patterns and time-varying computations. These constructs can be composed freely to describe complex streaming computations. We provide a formal denotational semantics for StreamQL using a class of monotone functions over streams. We have implemented StreamQL as a lightweight Java library, which we use to experimentally evaluate our approach. The experiments show that the throughput of our implementation is competitive compared to state-of-the-art streaming engines such as RxJava and Reactor.

CCS Concepts: • **Software and its engineering → Domain specific languages**; • **Theory of computation → Streaming models**.

Additional Key Words and Phrases: data stream processing, denotational semantics

ACM Reference Format:
Lingkun Kong and Konstantinos Mamouras. 2020. StreamQL: A Query Language for Processing Streaming Time Series. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 183 (November 2020), 32 pages. https://doi.org/10.1145/3428251

1 INTRODUCTION

Recent technological advances, such as the Internet of Things (IoT), are causing an enormous proliferation of streaming data, i.e., data that is generated in real-time and at high rates. Such data arise in several application domains, including healthcare monitoring, network traffic monitoring, analysis of financial markets, telecommunications, and smart transportation. There are various proposals for specialized languages, compilers, and runtime systems that deal with the processing of streaming data. Relational database systems and SQL-based languages have been adapted to the streaming setting [Abadi et al. 2003; Abadi et al. 2005; Arasu et al. 2006; Babcock et al. 2002; Chandrasekaran et al. 2003; Motwani et al. 2003]. Several systems have been developed for the distributed processing of data streams that are based on the dataflow model of computation [Kulkarni et al. 2015; Toshniwal et al. 2014; Zaharia et al. 2013]. Languages for detecting complex events in distributed systems, which draw on the theory of regular expressions and finite-state

Authors' addresses: Lingkun Kong, Department of Computer Science, Rice University, USA, klk@rice.edu; Konstantinos Mamouras, Department of Computer Science, Rice University, USA, mamouras@rice.edu.

2475-1421/2020/11-ART183
https://doi.org/10.1145/3428251

automata, have also been proposed [Brenna et al. 2007; Hirzel 2012; Wu et al. 2006; Zemke et al. 2007]. Synchronous dataflow programming languages [Benveniste et al. 2003; Berry and Gonthier 1992; Caspi et al. 1987; Lee and Messerschmitt 1987] have been used for streaming computations in the context of embedded systems. Several formalisms for the runtime verification of reactive systems have been proposed, many of which are based on variants of Temporal Logic and its timed/quantitative extensions [D'Angelo et al. 2005; Deshmukh et al. 2017; Havelund and Roşu 2004; Leucker and Schallhart 2009; Thati and Roşu 2005]. Finally, there is a rich set of languages and systems for reactive programming [Courtney 2001; Elliott and Hudak 1997; Maier and Odersky 2012; Meijer 2012], which focus on the development of event-driven and interactive applications.

While the aforementioned approaches have been successful within the application domains for which they were developed, modern applications require further language support for the high-level specification of processing over *streaming time series*. A streaming time series is a data stream, which consists of a potentially unbounded sequence of data items that arrive in increasing time order. The processing of such data typically involves computations that integrate simple transformations, the detection of patterns, and streaming aggregations. For example, consider the data streams generated by sensors in a real-time health monitoring application (such as heart rhythm and brain activity monitoring). These signals contain noise from various sources. Moreover, they are mostly uneventful and interspersed with episodes of unusual activity that need to be identified and analyzed in a timely manner. So, the monitoring application needs to perform a complex streaming computation that reduces the noise, identifies abnormal patterns in the signals, and summarizes the most important information.

One approach for specifying the processing of streaming time series is to use a low-level imperative programming language such as C or C++. This approach quickly becomes difficult and error-prone, as the overall computation cannot be easily expressed in a modular way. The resulting program contains complex state-manipulating logic and the code is highly entangled (an example that illustrates this point is discussed in Section 3). For this reason, it is desirable to provide language support for assisting the programmer in specifying the application in a modular way by composing simpler computational primitives. However, existing approaches do not provide all the necessary abstractions for specifying such complex computations in a natural and succinct way. For example, streaming SQL and related query languages focus on *relational abstractions*, but provide limited support for computations that rely on the temporal sequencing of events. The synchronous and reactive languages offer *dataflow abstractions*, but are less suitable for the modular specification of complex temporal patterns. The monitoring formalisms that are based on Temporal Logic have some quantitative features (e.g., timestamp comparisons and simple value thresholds), but provide little support for aggregations and signal transformations.

To bridge this gap, we propose a language, called StreamQL (Streaming Query Language), which simplifies the task of specifying complex streaming computations over time series data. In contrast to existing proposals whose basic object is the stream (e.g., `Observable` in Rx [Meijer 2012]), the basic object in StreamQL is the *stream transformation* that describes how an input stream is transformed into an output stream. StreamQL provides a novel integration of several useful programming abstractions for stream processing: (1) ***relational*** constructs (such as filtering, mapping, aggregating, key-based partitioning, and windowing), (2) ***dataflow*** constructs (such as streaming/serial and parallel composition), and (3) ***temporal*** constructs that are inspired from Temporal Logic and regular expressions. StreamQL allows the programmer to specify a streaming analysis in a modular fashion, since its language constructs compose freely.

Design of StreamQL. In StreamQL, a stream transformation is captured syntactically with a *query*. We classify queries according to their *input/output type* in order to guarantee that composite

queries (i.e., queries that result from the composition of simpler queries) are well-formed. We write `f` : `Q`(A, B) to indicate that the query `f` processes an input stream with items of type A and produces an output stream with items of type B. A stream is typically viewed as an unbounded sequence of data items (elements). A key feature of StreamQL is that it generalizes this notion of a stream by allowing the occurrence of a distinguished symbol $\triangleleft$, called *end-of-stream marker*, that signals the termination of the stream. This is useful not only because there are certain streams that indeed terminate (e.g., when reading lines from a text file), but more importantly because it allows us to decompose unbounded streams into finite regions: each finite region can be viewed as a stream that eventually terminates. Such decompositions of streams are essential for the modular description of complex streaming computations. A key design feature of StreamQL is that a query can *halt* (terminate), even before the input stream has terminated. After a query has halted, then our language allows the computation to proceed according to some other queries, thus varying the computation across time. This novel feature of StreamQL enhances modularity by enabling the unrestricted composition of temporal and dataflow/relational operators.

StreamQL has an expressive set of combinators for describing common stream processing primitives, as well as rich forms of composition. The primitive queries `map`, `filter`, `reduce` and `aggr` describe basic streaming operators for transforming, filtering, and aggregating streams. The combinator `groupBy` supports the key-based partitioning of a stream and independent computation over disjoint sub-streams. The windowing combinators `tWindow` (tumbling) and `sWindow` (sliding) facilitate the specification of computations that operate on finite spans of an unbounded data stream. The combinators $\gg$ (streaming/serial composition) and `par` (parallel composition) allow the programmer to describe a complex computation as a directed acyclic graph of independent tasks, which facilitates modular specification and exposes pipeline and task parallelism. The atomic queries `takeUntil`, `skipUntil` and `search` are used to identify simple single-event patterns in a stream. They are inspired from the *Until* connective of Temporal Logic. The combinators `seq` (temporal sequencing) and `iter` (temporal iteration) are useful for describing time-varying analyses and detecting complex temporal patterns. The constructs `seq` and `iter` can be viewed as stream-transforming analogs of concatenation and Kleene's star from regular expressions.

The StreamQL language has a formal *denotational semantics*. A query `f` : `Q`(A, B) represents a monotone function $A^* \cdot \{\varepsilon, \triangleleft\} \longrightarrow B^* \cdot \{\varepsilon, \triangleleft\}$, where * is Kleene's star, ε is the empty string, and $\cdot$ is string concatenation. The monotonicity requirement captures a key requirement of streaming computation: an output item cannot be retracted after it has been emitted to the output. Every combinator of StreamQL has a denotational semantic analog, which provides unambiguous meaning for the entire language and thus validates the language design.

Implementation & Experimental Evaluation. We provide an implementation of StreamQL as a lightweight Java library. We use an explicit mechanism to *reinitialize* or *reset* the streaming computation, which allows us to reuse the allocated memory when performing operations that involve stream decomposition. In addition to the core combinators, the implementation provides support for aggregation (e.g., median and general percentiles), efficient algorithms for sliding windows, signal-processing primitives such as FFT (Fast Fourier Transform), FIR (Finite Impulse Response) filters, IIR (Infinite Impulse Response) filters, and several common stream processing idioms. We have used the library to specify real-word streaming applications on health monitoring.

We compare our StreamQL implementation against three popular open-source streaming engines: RxJava [RxJava 2020], Reactor [Reactor 2020], and Siddhi [Suhothayan et al. 2011]. The experiments show that our implementation consistently performs well when compared to these state-of-the-art streaming engines. In benchmarks with realistic workloads, the throughput of StreamQL is 1.1–10 times higher than RxJava, 1.2–20 times higher than Reactor, and 5–100 times higher than Siddhi.

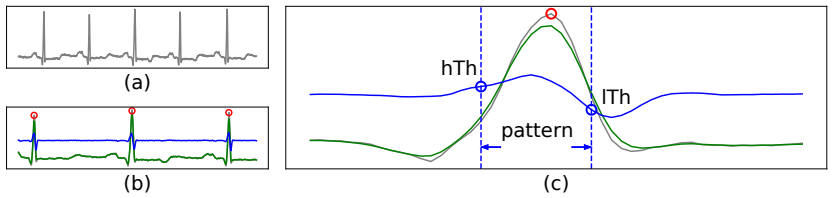


Fig. 1. (a) Electrocardiogram or ECG, (b) ECG with annotated signal peaks, (c) Pattern for peak detection.

Main Contributions. The main contribution of our paper is the identification of language abstractions for processing streaming time series that balance well the following desired properties: (1) they have clear formal semantics, (2) they give rise to an expressive and compositional language, and (3) they enable a very lightweight implementation. At the level of language design, the key choice is to base the language on stream transformations that can potentially halt (instead of nested streams) and combinators on them. With respect to the implementation, a key idea is the introduction of an execution model that is essentially a stream transducer that receives a special control signal for resetting its internal state. We have used this model to provide a compositional implementation of the language that avoids common sources of computational overheads that are present in related streaming languages.

Paper Outline. Section 2 introduces the StreamQL language and shows how it can be used to encode a streaming algorithm for peak detection. Section 3 discusses the expressiveness of StreamQL. The denotational semantics of StreamQL is presented in Section 4. Section 5 describes the Java implementation for StreamQL, and Section 6 presents the experimental evaluation of our implementation. In Section 7, a significant application on ABP (Arterial Blood Pressure) detection is presented. Section 8 reviews related work, and Section 9 summarizes this paper.

2 OVERVIEW OF STREAMQL

As a motivating example for StreamQL, we will consider the processing of cardiac (heart) signal for a patient. This signal is called an electrocardiogram (ECG). We will focus on the problem of *peak detection* in the ECG, which corresponds to the detection of the heartbeat. This problem is one of the most widely studied detection problems in the area of biomedical engineering [Bert-Uwe Köhler 2002; Pan and Tompkins 1985], as it forms the basis of many analyses over cardiac data.

Figure 1(a) shows part of an ECG, which is the electrical cardiac signal recorded on the surface of the skin near the heart. The horizontal axis is time and the vertical axis is voltage. A simple but effective procedure for detecting the peaks consists of three stages: (1) smoothing the signal to eliminate high-frequency noise, (2) taking the derivative of the smoothed signal to calculate the slope, and (3) finding the peaks using both the raw measurements and the derivatives.

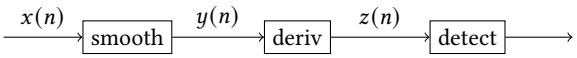


Figure 1(b) shows a short snippet (about 3 seconds) of an ECG signal, where the gray line corresponds to the input time series $x(n)$, the green line is the smoothed data $y(n) = (x(n-2) + 2x(n-1) + 4x(n) + 2x(n+1) + x(n+2))/10$, and the blue line is the derivative $z(n) = y(n) - y(n-1)$. A straightforward algorithm for detecting the peaks is to find the first occurrence (let us say at time i) where the derivative $z(n)$ exceeds a pre-defined threshold `hTh`, followed by the first occurrence after i (let us say at time j) where the derivative $z(n)$ becomes less than a threshold `lTh`. Time point i is located on the ascending slope towards the peak, and time point j is located on the descending slope after the peak. So, the exact peak location can be found by searching for the maximum value

of the original $x(n)$ time series in the interval from i to j. This pattern is illustrated in Figure 1(c). Every time a peak is identified, this detection procedure is reset and repeated.

This motivating example shows that the detection of complex patterns requires the transformation of the data stream (e.g., smoothing and differentiation) to enrich it with extra information. For this reason, the basic concept in the design of StreamQL is the *stream transformation*, which specifies how an input stream is transformed into an output stream. A *query* is a syntactic description of a stream transformation. Every query `f` has a *type* $\mathtt{Q}(A, B)$, where A is the type of input data items, and B is the type of the output data items. We write $\mathtt{f} : \mathtt{Q}(A, B)$ to indicate that `f` is of type $\mathtt{Q}(A, B)$.

A stream is typically viewed as an unbounded sequence of data items. We consider here a generalization of this notion of streams by assuming that the stream can potentially contain an occurrence of a special $\triangleleft$ symbol, called *end-of-stream marker*. We say that a stream is *terminated* if it ends with $\triangleleft$. As we will see later, the introduction of the end-of-stream marker allows us to define stream transformations that operate only on finite parts of the stream, which is useful for the modular specification of complex streaming computations such as time-varying analyses.

We will proceed to present the basic programming constructs of the StreamQL language. We will start with some simple primitives, and we will gradually build up towards the more complex combinators (i.e., composition constructs) of the language. Finally, we will conclude this section with a complete description of the ECG peak detection algorithm.

Map, Filter, Aggregate, and Reduce. Suppose that the input stream is a real-valued discrete-time signal. The type of the input data items is a record type $\mathtt{VT} = \{\mathtt{val} : \mathtt{V}, \mathtt{ts} : \mathtt{T}\}$, where `V` is the type of scalar values (e.g., real numbers) and `T` is the type of time points (e.g., natural numbers). The *map* query $\mathtt{f} = \mathtt{map}(\mathtt{x} \text{ -> } 2 \cdot \mathtt{x.val})$ has type $\mathtt{Q}(\mathtt{VT}, \mathtt{V})$ and represents the transformation that outputs the double of the value of each item. The argument $\mathtt{x} \text{ -> } 2 \cdot \mathtt{x.val}$ is a lambda expression that defines a function of type $\mathtt{VT} \longrightarrow \mathtt{V}$. The *filter* query $\mathtt{g} = \mathtt{filter}(\mathtt{x} \text{ -> } \mathtt{x.val} \geq 2.0)$, of type $\mathtt{Q}(\mathtt{VT}, \mathtt{VT})$, filters out those items with a value less than 2.0 and keeps the rest. The lambda expression $\mathtt{x} \text{ -> } \mathtt{x.val} \geq 2.0$ is a predicate on `VT`. The *aggregation* query $\mathtt{h} = \mathtt{aggr}(0.0, (\mathtt{x}, \mathtt{y}) \text{ -> } \mathtt{x} + \mathtt{y.val}) : \mathtt{Q}(\mathtt{VT}, \mathtt{V})$ represents the running sum of the values in the input stream. The first argument $0.0 : \mathtt{V}$ is the initial aggregate value, and the second argument is a binary function of type $\mathtt{V} \times \mathtt{VT} \longrightarrow \mathtt{V}$ that specifies how to aggregate each input data item. The *reduce* query $\mathtt{k} = \mathtt{reduce}(0.0, (\mathtt{x}, \mathtt{y}) \text{ -> } \mathtt{x} + \mathtt{y.val})$, of type $\mathtt{Q}(\mathtt{VT}, \mathtt{V})$, is similar to the running aggregation query `h`, with the difference that it only emits the total aggregate when the input stream terminates. Table 1 shows the execution of the queries `f`, `g`, `h`, `k`, where time progresses in the left-to-right direction. For a function $\mathtt{op} : A \times A \longrightarrow A$, we also consider the variants $\mathtt{aggr}(\mathtt{op}), \mathtt{reduce}(\mathtt{op}) : \mathtt{Q}(A, A)$, which do not need an initial aggregate (the first item of the input serves this purpose). The most general variants take a function $\mathtt{init} : A \longrightarrow B$ for initialization (using the first item of the input) and an aggregation function $\mathtt{op} : B \times A \longrightarrow B$.

Table 1. Map, Filter, Aggregate, Reduce.

input:	$\langle 2.5, 1\rangle$	$\langle 0.8, 2\rangle$	$\langle 3.5, 3\rangle$	$\langle 0.9, 4\rangle$	$\triangleleft$
f output:	5.0	1.6	7.0	1.8	$\triangleleft$
g output:	$\langle 2.5, 1\rangle$		$\langle 3.5, 3\rangle$		$\triangleleft$
h output:	2.5	3.3	6.8	7.7	$\triangleleft$
k output:					$7.7\ \triangleleft$

Key-based Partitioning. Let us consider an input stream with items of type $\mathtt{IV} = \{\mathtt{id} : \mathtt{ID}, \mathtt{val} : \mathtt{V}\}$, where `ID` is a type of identifiers. Suppose that we have written a query $\mathtt{f} : \mathtt{Q}(\mathtt{IV}, B)$ that computes an aggregate of items with a fixed identifier, i.e. under the assumption that all the items of the input stream have the same identifier. Then, to compute this aggregate across all identifiers, the most natural way is to partition the input stream by a key, the identifier field `id` in this case, and supply the corresponding projected sub-stream to a copy of `f`. This construct is

Table 2. Key-based partitioning.

input:	$\langle a, 3\rangle$	$\langle b, 5\rangle$	$\langle a, 1\rangle$	$\langle c, 2\rangle$	$\langle c, 1\rangle$	$\langle a, 4\rangle$	$\triangleleft$
group a:	$\langle a, 3\rangle$		$\langle a, 1\rangle$			$\langle a, 4\rangle$	$\triangleleft$
group b:		$\langle b, 5\rangle$					$\triangleleft$
group c:				$\langle c, 2\rangle$	$\langle c, 1\rangle$		$\triangleleft$
g output:	3	5	4	2	3	8	$\triangleleft$

called *key-based partitioning* and it is described by the query `g = groupBy(x -> x.id, f) : Q(IV, B)`. The first argument `x -> x.id` is a function of type `IV` $\longrightarrow$ `ID` that specifies the partitioning key, and `f` describes the computation that will be independently performed on each sub-stream. If we choose `f = aggr(0, (x, y) -> x + y.val) : Q(IV, V)` to be a running sum, then the query `g = groupBy(x -> x.id, f) : Q(IV, V)` performs the computation shown in Table 2.

Tumbling & Sliding Windows. The so-called windowing constructs are used to partition an unbounded stream into finite fragments called windows and perform computations on each one of them independently. The *tumbling window* combinator splits the stream into contiguous non-overlapping regions. For a query `f : Q(A, B)` and a natural number $n \geq 1$, the query `tWindow(n, f)` applies `f` to tumbling windows of size n. The *sliding window* combinator splits the stream into overlapping regions. For a query `f : Q(A, B)` and natural numbers n, s with $1 \leq s < n$, the query `sWindow(n, s, f)` applies `f` to windows of size n with a new window starting every s items. Let us consider now the query `f = reduce(0, (x, y)->x+y) : Q(V, V)`, which calculates the total sum of a terminated stream. Table 3 illustrates `tWindow` and `sWindow`. We also provide variants of the windowing constructs that allow the programmer to specify a function `op` : $A^n \longrightarrow B$ to summarize the contents of a window of size n, as in `tWindow(n, op)` and `sWindow(n, s, op)`. For example, the query `sWindow(3, 1, (x, y, z) -> (x + y + z)/3)` computes the sliding (moving) average over windows of size 3.

Table 3. Tumbling and sliding windows.

input:	1	2	3	4	5	6	7	8	◁
`tWindow(2, f)` output:			3		7		11		15 ◁
`tWindow(3, f)` output:				6			15		◁
`sWindow(2, 1, f)` output:			3	5	7	9	11	13	15 ◁
`sWindow(3, 1, f)` output:				6	9	12	15	18	21 ◁
`sWindow(3, 2, f)` output:				6		12		18	◁

Streaming/Serial Composition. A natural construct for streaming computation is to compose queries `f : Q(A, B)` and `g : Q(B, C)` so that the output items produced by `f` are supplied as input to `g`. This is denoted by `pipeline(f, g) : Q(A, C)`, which we abbreviate as `f` $\gg$ `g`. We call this query the *streaming* or *serial composition* of `f` and `g`. This construct generalizes to more than two arguments. It is useful for setting up a complex computation as a pipeline of stages. Consider the queries `f = map(x -> x.val) : Q(VT, V)` and `g = aggr(max) : Q(V, V)`. The query `f` $\gg$ `g : Q(VT, V)` computes the running maximum (see Table 4).

Table 4. Streaming (serial) composition.

input:	$\langle 2.5, 1\rangle$	$\langle 0.8, 2\rangle$	$\langle 3.5, 3\rangle$	$\langle 0.9, 4\rangle$	◁
`f` output:	2.5	0.8	3.5	0.9	◁
`f` $\gg$ `g` output:	2.5	2.5	3.5	3.5	◁

Parallel Composition. We introduce a construct for executing multiple queries in parallel on the same input stream and combining their results. For queries `f` and `g` of type `Q(A, B)`, the query `par(f, g) : Q(A, B)` describes the following computation: The input stream is duplicated with one copy sent to `f` and one copy sent to `g`. The queries `f` and `g` compute in parallel, and their outputs are merged (specifically, interleaved) to produce the final output. Using the running sum query `f = aggr(+)` and the running count query `g = aggr(0, (x, y) -> x + 1)`, both of type `Q(V, V)`, the query `h = par(f, g)` $\gg$ `tWindow(2, (x, y) -> x/y) : Q(V, V)` computes the running average (see Table 5). The `par` construct generalizes to several arguments.

Table 5. Parallel composition

input:	10	20	30	40	50	◁
`f` output:	10	30	60	100	150	◁
`g` output:	1	2	3	4	5	◁
`par(f, g)` output:	10 1	30 2	60 3	100 4	150 5	◁
`h` output:	10	15	20	25	30	◁

Temporal Constructs. As mentioned before, the end-of-stream marker ◁ indicates the end of a stream. When a query emits ◁ to the output we say that it *halts*, because it cannot produce any more output. All the query examples that we have seen so far have the property that they halt exactly when they encounter ◁ in the input stream. By lifting this restriction we can support queries that

can halt early. This is useful (1) for varying a streaming computation as time progresses and (2) for detecting complex temporal patterns.

The query `takeUntil(p) : Q(A, A)`, where `p` is a predicate over A, computes like the identity transformation while there is no occurrence of an item satisfying `p` in the input. When it encounters the first item satisfying `p`, it emits it to the output and halts. A similar query is `take(n) : Q(A, A)`, where $n \geq 1$ is an integer, which echoes the first n items of the input stream to the output and then halts. The query `skipUntil(p) : Q(A, A)`, for a predicate `p` on A, emits no output while the input contains no item satisfying `p`. When the first item satisfying `p` is seen, it emits it to the output and continues to compute like the identity transformation. The query `skip(n) : Q(A, A)`, for an integer $n \geq 1$, emits no output for the first n input items, and then proceeds to echo the rest of the input stream. The query `ignore() : Q(A, A)` emits empty output for all input items and halts for the end-of-stream marker. The query `ignore(n) : Q(A, A)`, for an integer $n \geq 1$, emits no output for the first n input items and then immediately halts. For a predicate `p` on A, the query `search(p) : Q(A, A)` emits no output while it searches for the first occurrence of an item satisfying `p`. When it encounters such an item, it emits it to the output and halts. See Table 6.

Table 6. Take, skip, ignore, search.

input:	1	2	3	4	5	◁	
`takeUntil(x -> x ≥ 4)` output:	1	2	3	4	◁		
`take(3)` output:	1	2	3	◁			
`skipUntil(x -> x ≥ 4)` output:				4	5	◁	
`skip(2)` output:			3	4	5	◁	
`ignore()` output:						◁	
`ignore(3)` output:			◁				
`search(x -> x ≥ 2)` output:		2	◁				

The ***temporal sequencing*** combinator can apply different queries in sequence (i.e., one after the other), thus varying the computation over time. For queries `f` and `g` of type `Q(A, B)`, their temporal sequencing `seq(f, g) : Q(A, B)` computes like `f` until it halts, and then it proceeds to compute like `g`. For example, if `f = search(x -> x ≥ 3)` and `g = takeUntil(x -> x ≤ 2)`, then `seq(f, g)` computes as shown in Table 7.

Table 7. Temporal sequencing.

input:	1	2	3	4	3	2	1	◁
`f` output:			3	◁				
`seq(f, g)` output:			3	4	3	2	◁	

The ***temporal iteration*** combinator can be used to repeat a streaming computation indefinitely. For a query `f : Q(A, B)`, its temporal iteration `iter(f) : Q(A, B)` executes `f` and restarts it every time it halts. This results in an unbounded temporal repetition of the computation that `f` specifies. Now, the iteration of `f` ≫ `g`, where `f = takeUntil(x -> x = 0)` and `g = reduce(0, +)`, computes as shown in Table 8.

Table 8. Temporal iteration.

input:	2	3	0	9	0	1	7	0	3
`f` output:	2	3	0	◁					
`f` ≫ `g` output:			5	◁					
`iter(f` ≫ `g)` output:			5		9			8	

Flatten and Emit. The query `flatten : Q(List(A), A)` processes an input stream whose data items are lists that contain elements of type A, and it propagates list element to the output. For a list `out : List(B)`, the query `emit(out) : Q(A, B)`, specifies the computation that outputs the elements of `out` at the very beginning (before any input items are consumed) and then immediately halts. See Table 9 for examples.

Table 9. Flatten and Emit.

input:	$[a_1, a_2]$	[]	$[a_3]$	[]	◁	
`flatten` output:	a_1	a_2	a_3		◁	

input:		a_1	a_2	...
`emit(`$[b_1, b_2]$`)` output:	b_1 b_2 ◁			

Join. StreamQL provides the constructs `zip`, `zipLast`, and `join` to combine input data from several input sub-streams. Let us consider an input stream with data items from two different sources, in which one is the signal measurement (of type `V`), and the other is the signal identifier (of type `ID`). The input type is `Or(V, ID)`, which means that an input item is either of type `V` or of type `ID`. Then, to annotate the signal measurements with corresponding identifiers as outputs of type `IV = {id : ID, val : V}`, a natural way is to combine the values and the identifiers based

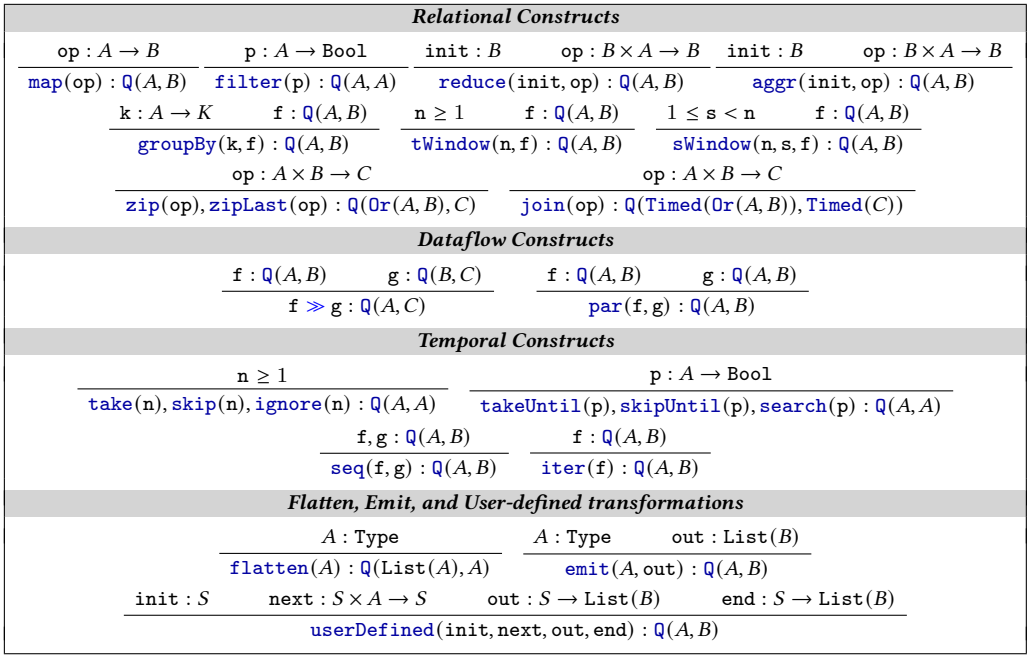


Fig. 2. The Streaming Query Language (StreamQL).

on their order of arrival. StreamQL provides the constructs `zip` and `zipLast`. Given a function `op = (val, id) -> ⟨val, id⟩ : V × ID → IV` that annotates a signal measurement with an identifier, the query `f = zip(op) : Q(Or(V, ID), IV)` combines the measurements and the identifiers one by one with respect to their order of arrival, and the query `g = zipLast(op) : Q(Or(V, ID), IV)` combines the last arrived data items from different categories (See Table 10).

Table 10. Zip and ZipLast.

input: 1.2	a	b	-3.3	c	2.5	$\triangleleft$
val: 1.2			-3.3		2.5	
id:	a	b		c		
f output:	$\langle 1.2, a\rangle$		$\langle -3.3, b\rangle$		$\langle 2.5, c\rangle$	$\triangleleft$
g output:	$\langle 1.2, a\rangle$	$\langle 1.2, b\rangle$	$\langle -3.3, b\rangle$	$\langle -3.3, c\rangle$	$\langle 2.5, c\rangle$	$\triangleleft$

Moreover, StreamQL allows users to assign a *validity interval* to the data item, and it provides the `join` construct to combine data items that have overlapping validity intervals. To assign validity intervals, users need to provide the start/end time of the interval for each input – the input type is specified as $\texttt{Timed}(D) = \{\texttt{data} : D, \texttt{startT} : T, \texttt{endT} : T\}$, where D denotes the type of the data, and T is the type of the time unit (e.g., long integers). Suppose $D = \texttt{Or}(A, B)$ (i.e., the input data is either of type A or type B), given a binary function $\texttt{op} : A \times B \to C$ that combines data, the $\texttt{join}(\texttt{op}) : \texttt{Q}(\texttt{Timed}(\texttt{Or}(A, B)), \texttt{Timed}(C))$ query joins data items with overlapping validity intervals. The output item, of type $\texttt{Timed}(C)$, is also labeled by a validity interval which is the intersection of the validity intervals of the input data.

User-defined Stream Transformations. The construct `userDefined` is used to specify a stream transformation with a transducer (state machine). The query $\texttt{userDefined}(\texttt{init}, \texttt{next}, \texttt{out}, \texttt{end}) : \texttt{Q}(A, B)$ takes four arguments to describe the computation: `init` (of type S) is the initial state of the transducer, $\texttt{next} : S \times A \to S$ is the state transition function, $\texttt{out} : S \to \texttt{List}(B)$ is the output function, and $\texttt{end} : S \to \texttt{List}(B)$ gives the final output (upon termination of the input with $\triangleleft$).

Streaming Peak Detection. Figure 2 summarizes several constructs of StreamQL. In the beginning of this section, we gave a high-level description of a simple streaming algorithm for detecting the peaks in the ECG signal. We will now use StreamQL to provide a complete description of this algorithm, which is a variant of [Moody 2018]. In Section 7, we will present a significant example for processing the Arterial Blood Pressure signal. Suppose that the data stream concerns multiple patients, that is, it is the interleaving of several ECG time series, one for each patient. The type of the input data items is a record type $\texttt{IVT} = \{\texttt{id} : \texttt{ID}, \texttt{val} : \texttt{V}, \texttt{ts} : \texttt{T}\}$, where `ID` is the type of patient identifiers, `V` is the type of scalar values, and `T` is the type of time points. At the top level, the algorithm works by partitioning the input stream into several sub-streams, one for each patient, and performing peak detection for each one of these sub-streams independently. The query $\texttt{groupBy}(\texttt{x -> x.id}, \texttt{findPeak})$ describes this computation, where `findPeak` specifies the peak detection algorithm for a single-patient ECG data stream. This is defined as $\texttt{findPeak} = \texttt{smooth} \gg \texttt{deriv} \gg \texttt{detect}$, which is the composition of three stages: (1) smoothing the signal, (2) computing derivatives, and (3) detecting peaks. The smoothing query $\texttt{smooth} : \texttt{Q}(\texttt{IVT}, \texttt{IVTF})$ has output type `IVTF`, which is the record type `IVT` extended with the component $\texttt{fval} : \texttt{V}$ for storing the smoothed (low-pass filtered) value.

$$\begin{aligned}\texttt{smooth} &= \texttt{sWindow}(5, 1, (\texttt{v}, \texttt{w}, \texttt{x}, \texttt{y}, \texttt{z}) \texttt{ -> expr}), \text{ where}\\ \texttt{expr} &= \langle \texttt{x.id}, \texttt{x.val}, \texttt{x.ts}, \texttt{fval} \rangle : \texttt{IVTF} \text{ and}\\ \texttt{fval} &= (\texttt{v.val} + 2 \cdot \texttt{w.val} + 4 \cdot \texttt{x.val} + 2 \cdot \texttt{y.val} + \texttt{z.val})/10.\end{aligned}$$

The idea is that for a sample `x` at time `x.ts` we consider the window $(\texttt{v}, \texttt{w}, \texttt{x}, \texttt{y}, \texttt{z})$ centered around `x` and calculate a weighted average over the window for the smoothed value. The differentiation query $\texttt{deriv} : \texttt{Q}(\texttt{IVTF}, \texttt{IVTFD})$ calculates discrete derivatives by taking the difference of successive smoothed values. It is implemented as follows:

$$\begin{aligned}\texttt{deriv} &= \texttt{sWindow}(2, 1, (\texttt{x}, \texttt{y}) \texttt{ -> expr}), \text{ where}\\ \texttt{expr} &= \langle \texttt{y.id}, \texttt{y.val}, \texttt{y.ts}, \texttt{y.fval}, \texttt{dval} \rangle : \texttt{IVTFD} \text{ and } \texttt{dval} = \texttt{y.fval} - \texttt{x.fval} : \texttt{V}.\end{aligned}$$

The record type `IVTFD` extends `IVTF` with $\texttt{dval} : \texttt{V}$ for storing the derivative. The detection of the first peak involves searching for the first time point ℓ_1 when `dval` exceeds the threshold `hTh`. The signal interval from this point until the time point r_1 when `dval` falls below the threshold `lTh` contains the first peak. Thus, the signal in the interval $[\ell_1, r_1]$ is streamed to the `argmax` query (see below), which finds the data item with the highest value (in the raw, unfiltered signal). This process is repeated indefinitely in order to detect all peaks:

$$\begin{aligned}\texttt{start} &= \texttt{search}(\texttt{x -> x.dval} > \texttt{hTh})\\ \texttt{take} &= \texttt{takeUntil}(\texttt{x -> x.dval} < \texttt{lTh})\\ \texttt{argmax} &= \texttt{reduce}((\texttt{x}, \texttt{y}) \texttt{ -> } (\texttt{y.val} > \texttt{x.val}) \texttt{ ? y : x})\\ \texttt{detect} &= \texttt{iter}(\texttt{seq}(\texttt{start}, \texttt{take}) \gg \texttt{argmax})\end{aligned}$$

All four queries above are of type $\texttt{Q}(\texttt{IVTFD}, \texttt{IVTFD})$.

3 DISCUSSION OF EXPRESSIVENESS

A natural approach for processing a data stream is to write a program in a low-level imperative programming language such as C. However, this process is tedious and error-prone because the computation cannot be easily expressed in a modular way. The program that specifies the computation typically contains complex state-manipulating logic and the code is heavily entangled. For this reason, several domain-specific languages have been proposed which offer various primitive streaming constructs (e.g., pipelines and sliding windows) in order to assist the programmer in expressing the desired computation. In this section, we will illustrate some of the features of

```
double coef[5] = {0.1, 0.2, 0.4, 0.2, 0.1};
V t[5]; // circular array
int cnt = 0, start = 0;
bool isFReady = false;
void next(V v){
  // smooth the input
  if (cnt < 5) {
    t[cnt++] = v;
  } else {
    t[start] = v;
    start = (start + 1) % 5;
  }
  F f = 0.0;
  if (cnt == 5) {
    // compute the result when t is full
    for (int i = 0; i < 5; i++) {
      f += coef[i] * t[(start + i) % 5];
    }
  }
  if (isFReady) {
    // compute the derivative
    D d = f - lastF;
    lastF = f;
    out(d); // produce output
  } else if (cnt == 5) {
    lastF = f;
    isFReady = true;
  } // else do nothing
}
```

```
double coef[] = {0.1, 0.2, 0.4, 0.2, 0.1};
class FCnt{
  F f; // FIR filtering result
  int cnt;
  FvalCnt(F f, int cnt){
    this.f = f;
    this.cnt = cnt;
  }
}
Observable<D> outputStream = inputStream
  .window(5, 1) // smooth the input
  .flatMap(wnd -> wnd.reduce(
    new FCnt(0.0, 0),
    (pair, v) -> {
      F f = pair.f;
      int cnt = pair.cnt;
      f += v * coef[cnt];
      cnt ++;
      return new FCnt(f, cnt);
    }).map(p -> p.f).toObservable()
  ).window(2, 1) // compute the derivative
  .flatMap(wnd -> wnd.reduce(
    new ArrayList<>(),
    (l, f) -> { // add f into list l
      return List.copyOf(l.add(f));
    }).map(l -> l.size() == 2 ?
                l.get(1) - l.get(0) : null)
    .filter(d -> d != null).toObservable()
  );
```

```
Q<V,F> smooth = sWindow(5, 1, (a, b, c, d, e) -> (a + 2*y + 4*c + 2*d + e) / 10.0);
Q<F,D> deriv = sWindow(2, 1, (a, b) -> b - a);
Q<V,D> query = pipeline(smooth, deriv);
```

Fig. 3. Program for input preprocessing written in C (top-left), RxJava (top-right), and StreamQL (bottom).

StreamQL that facilitate the modular description of streaming computations, particularly for time-series workloads. We will compare StreamQL to both low-level imperative languages (such as C) and domain-specific languages (such as Rx) in the context of a concrete example.

Assume that the input stream consists of signal measurements of type `V` (integer type) which are collected at a fixed frequency. We will consider a computation that is the composition of a smoothing filter and calculating the derivative. We use a low-pass filter to smooth the input into results `f : F` (floating point type), where $\texttt{f} = (\texttt{v}_1 + 2\texttt{v}_2 + 4\texttt{v}_3 + 2\texttt{v}_4 + \texttt{v}_5)/10$ for each five consecutive input items $\texttt{v}_1, \texttt{v}_2, ..., \texttt{v}_5$. Then, we compute the derivative `d : D` (floating point type) where $\texttt{d} = \texttt{f}_2 - \texttt{f}_1$ for every two consecutive smoothed values. The top-left part of Figure 3 shows the algorithm implemented in C. It processes the input stream item by item by calling the `next` function and produces output items by calling the `out` function. We use the circular array `t` to buffer the input for smoothing. We update the array by replacing its oldest element by the incoming input item, and then we apply the coefficients of the low-pass filter (stored in the `coef` array) to the buffered elements to compute the smoothing results. After that, the program computes the derivatives and produces the output. The top-right part of Figure 3 shows the RxJava implementation. RxJava does not provide a sliding window construct that uses circular arrays. To integrate this efficient data structure, a user of the library would need to create a customized operator from scratch. The bottom part of Figure 3 shows the implementation in StreamQL, where the `sWindow` construct allows users to directly aggregate all elements inside the window with efficient built-in data structures.

Now, let us consider an algorithm for detect peaks in a stream of numerical values (suppose they are of type `V`). The algorithm searches for the first value that exceeds the threshold `THRESH`. Then,

```
V peak = -INFINITY;
enum mode { beforePeak, inPeak, afterPeak };
int cnt;
enum mode m = beforePeak;
void next(V v){
  if (m == beforePeak) {
    if (v > THRESH) {
      m = inPeak;
      cnt = PEAK_CNT;
    } // else do nothing
  } else if (m == inPeak) {
    peak = (v > peak) ? v : peak;
    cnt --;
    if (cnt == 0) {
      out(peak); // produce outputs
      m = afterPeak;
      cnt = SILENCE_CNT;
    }
  } else { // m == afterPeak
    cnt --;
    if (cnt == 0) {
      m = beforePeak;
      peak = -INFINITY;
    }
  }
}
```

```
Q<V,V> start = search(v -> v > THRESH);
Q<V,V> take = take(PEAK_CNT);
Q<V,V> max = reduce((x, y) -> (y > x) ? y : x);
Q<V,V> find1 = pipeline(seq(start, take), max);
Q<V,V> silence = ignore(SILENCE_CNT);
Q<V,V> query = iterate(seq(find1, silence));
```

```
enum Mode { beforePeak, inPeak, afterPeak }
class State{
  Mode mode;            int cnt;
  boolean sendOut;      V peak;
  State(Mode m, int c, boolean s, V p) {
    mode = m;           cnt = c;
    sentOut = s;        peak = p;
  }
}
Observable<V> outputStream = derivStream.scan(
  new State(beforePeak, 0, false, -INFINITY),
  (s, v) -> {
    Mode m = s.mode;
    int cnt = s.cnt;
    boolean sendOut = false;
    V peak = s.peak;
    if (m == beforePeak) {
      if (v > THRESH) {
        m = inPeak;
        cnt = PEAK_CNT;
      } // else do nothing
    } else if (m == inPeak) {
      peak = (v > peak) ? v : peak;
      if (-- cnt == 0) {
        sendOut = true;
        m = afterPeak;
        cnt = SILENCE_CNT;
      }
    } else { // m == afterPeak
      if (-- cnt == 0) {
        m = beforePeak;
        peak = -INFINITY;
      }
    }
    return new State(m, cnt, sendOut, peak);
  }).filter(s -> s.sendOut).map(s -> s.peak);
```

Fig. 4. Program for peak detection written in C (top-left), RxJava (right), and StreamQL (bottom-left).

it search for the maximum over the next `#PEAK_CNT` elements, which is considered a peak. After that, the algorithm silences detection for `#SILENCE_CNT` elements to avoid a duplicate detection. This process is repeated indefinitely in order to detect all peaks. The top-left part of Figure 4 shows the C implementation of the algorithm, where the input stream is repeatedly partitioned into three regions: `beforePeak`, `inPeak`, and `afterPeak`. This partitioning is data-dependent as the end of `beforePeak` happens when the value exceeds the threshold. The right part of Figure 4 is the RxJava implementation. Rx provides count/time-based tumbling windows to split up the stream into non-overlapping regions. However, such a decomposition is not data-dependent, as it does not rely on the input values. Moreoever, Rx has no operator like StreamQL's `iter` for repeating the execution of a query (i.e., detection of a single peak) every time it halts. Given the absence of these features, the most convenient way to program the algorithm in RxJava is to use its `scan` operator (similar to `aggr` in StreamQL). This amounts to providing a monolithic imperative implementation of the whole algorithm. The bottom-left part of Figure 4 shows the implementation of the algorithm in StreamQL.

Semantically, Rx and StreamQL can be understood as algebras with combinators. There is a key difference. Rx is an algebra of streams, where the basic objects are streams and the combinators are operations on streams. StreamQL, on the other hand, is an algebra of stream transformations, where its basic objects are stream transformations and the combinators are operations on transformations. More specifically, `Observable` is the basic object in Rx that represents a stream. Rx describes the overall computation as a sequence of transformations to the source `Observable`. The first-class

	StreamQL	Rx	Siddhi	Trill
1. stream filtering	yes	yes	yes	yes
2. stream mapping	yes	yes	yes	yes
3. sequential aggregation	yes	yes	∗	yes
4. key-based partitioning	yes	yes	yes	yes
5. tumbling window	yes	yes	yes	yes
6. sliding window	yes	yes	yes	yes
7. efficient window aggregation	yes	no	yes	yes
8. streaming pipeline	yes	yes	yes	yes
9. relational join	yes	yes	yes	yes
10. temporal sequencing	yes	no	no	no
11. temporal iteration	yes	no	no	no
12. signal processing primitives	yes	no	no	yes
13. regular parsing	∗	no	yes	yes
14. user-defined functions	yes	yes	yes	yes

Fig. 5. Some of the features and streaming constructs supported by StreamQL, Rx, Siddhi and Trill.

object in StreamQL is the stream transformation, which is captured syntactically with a query. StreamQL describes the computation as the composition of sub-computations defined by queries. For example, `iter(f)` is the temporal iteration of a query `f`. So, if Rx is considered first-order then StreamQL is second-order. This explains why `iter` and `seq` are easily integrated in StreamQL but are more difficult to express in Rx.

Figure 5 lists some useful constructs for stream processing and the engines that support them. The streaming operations marked with ∗ in Figure 5 indicate that the library supports such operations, but their use requires additional encoding. For example, for *sequential aggregation*, Siddhi does not have a construct like StreamQL's `aggr`. Instead, it defines a set of fixed aggregations (e.g., sum and average). Other sequential aggregations can be implemented using user-defined functions. Rx does not implement efficient algorithms for *window aggregation*, which is discussed in detail in Section 6. The *temporal sequencing* and the *temporal iteration* constructs (`seq` and `iter` in StreamQL) decompose the stream into sub-streams, and the decomposition is data-dependent. It is not easy to express such computations in Rx, Siddhi and Trill in a modular way. Both StreamQL and Trill provide *signal processing* constructs (e.g., FFT, FIR and IIR filtering) to transform and analyze signals. *Regular parsing* can be encoded in StreamQL since the constructs `seq` and `iter` are essentially stream-transforming analogs of concatenation and Kleene's star from regular expressions. Siddhi and Trill support regular parsing by providing extensions for complex event processing (CEP).

4 SEMANTICS

In this section, we present the denotational semantics of StreamQL using a class of monotone functions. This semantics clarifies the meaning of the language primitives and combinators. The use of monotone functions or other sequence transductions for describing streaming computations has been considered in [Alur et al. 2018; Chattopadhyay and Mamouras 2020; Mamouras et al. 2019; Mamouras and Wang 2020] and in a much more general algebraic setting in [Mamouras 2020].

For a type A, we write A^* to denote the set of finite sequences over A. We write $u \cdot v$ or uv to denote the concatenation of the sequences u and v, and ε for the empty word. Our language allows streams that can terminate. The special symbol $\triangleleft$ indicates the end of a stream. We call $\triangleleft$ the *end-of-stream marker*. Define $A^\dagger = A^* \cdot \{\varepsilon, \triangleleft\} = A^* \cup (A^* \cdot \triangleleft)$, i.e., $A^\dagger$ contains the finite sequences over A that could potentially end with an end-of-stream marker. For sequences $x, y \in A^\dagger$, we write $x \leq y$ if x is a prefix of y, i.e. $xz = y$ for some $z \in A^\dagger$. We say that $\leq$ is the *prefix relation* on sequences. When $x \leq y$, there is a *unique* z with $xz = y$, which we denote by $x^{-1}y$. We write $x < y$ when $x \leq y$ and $x \neq y$. A sequence $x \in A^\dagger$ is said to be *terminated* if it ends with $\triangleleft$.

The input/output behavior of a streaming computation can be described semantically by a function of type $A^\dagger \to B^\dagger$, where A is the type of the input items and B is the type of the output items. If $x \in A^\dagger$ is the prefix of the input stream seen so far, then $f(x) \in B^\dagger$ is the cumulative output that has been emitted after the whole sequence x is processed. As more input data items arrive, the output stream gets extended with more output items. This is captured formally by requiring that the function f is *monotone*: $x \leq y$ implies that $f(x) \leq f(y)$ for every $x, y \in A^\dagger$. A monotone function $f : A^\dagger \to B^\dagger$ is said to be a ***stream transformation***. We write $\mathsf{ST}(A, B)$ to denote the set of all stream transformations with input (resp., output) data items of type A (resp., B).

As mentioned earlier, a stream transformation $f : \mathsf{ST}(A, B)$ specifies the *cumulative output* of a streaming computation, i.e. the total output that has been emitted from the beginning until the entire input prefix is consumed. The computation can be described equivalently by specifying the *incremental output*, i.e. the output increment that is emitted exactly when the last item of an input prefix is consumed. The incremental output of f for the input history xa is equal to $f(x)^{-1}f(xa)$.

input item	input history	incremental output	cumulative output
	ε	0	0
1	1	1	0 1
2	1 2	3	0 1 3
3	1 2 3	6	0 1 3 6
$\triangleleft$	1 2 3 $\triangleleft$	$\triangleleft$	0 1 3 6 $\triangleleft$

The table above illustrates these concepts with the example of calculating the running sum over a stream of integers. Suppose $f : \mathsf{ST}(A, B)$ describes the input/output behavior of a streaming computation in a cumulative fashion, and $\varphi : A^\dagger \to B^\dagger$ describes the same computation in an incremental fashion. Then, f and φ are related in the following way:

$$f(a_1 a_2 \dots a_n) = \varphi(\varepsilon) \cdot \varphi(a_1) \cdot \varphi(a_1 a_2) \cdot \dots \varphi(a_1 a_2 \dots a_n)$$
$$f(a_1 a_2 \dots a_n \triangleleft) = f(a_1 a_2 \dots a_n) \cdot \varphi(a_1 a_2 \dots a_n \triangleleft)$$

for all $a_1 a_2 \dots a_n \in A^*$. Equivalently, we have that

$$\varphi(\varepsilon) = f(\varepsilon) \qquad \varphi(ua) = f(u)^{-1} f(ua) \qquad \varphi(u\triangleleft) = f(u)^{-1} f(u\triangleleft)$$

for all $u \in A^*$ and $a \in A$. Function φ satisfies the following property: if $\varphi(x)$ ends with $\triangleleft$, then $\varphi(y) = \varepsilon$ for all $y \geq x$. This says that when the output stream terminates, no more output data items can be emitted. We write $\partial f : A^\dagger \to B^\dagger$ to denote the incremental version of $f : \mathsf{ST}(A, B)$.

Figure 6 gives the denotational semantics for some core combinators of StreamQL. The definition of the stream transformations $\mathsf{map}(\mathsf{op})$ and $\mathsf{filter}(\mathsf{p})$ are straightforward. The transformations $\mathsf{reduce}(b, \mathsf{op})$ and $\mathsf{aggr}(b, \mathsf{op})$ are both aggregations, but differ in when they give output. Informally, $\mathsf{reduce}(b, \mathsf{op})$ gives the total aggregate when the stream terminates, whereas $\mathsf{aggr}(b, \mathsf{op})$ gives the running aggregate every time a new item arrives. Their definition requires the *fold combinator* $\mathsf{fold} : B \times (B \times A \to B) \times A^* \to B$, given by $\mathsf{fold}(b, \mathsf{op}, \varepsilon) = b$ and $\mathsf{fold}(b, \mathsf{op}, ua) = \mathsf{op}(\mathsf{fold}(b, \mathsf{op}, u), a)$. The *streaming (serial) composition* combinator is given by:

$$\frac{f : \mathsf{ST}(A, B) \qquad g : \mathsf{ST}(B, C)}{f \gg g : \mathsf{ST}(A, C)} \qquad (f \gg g)(a) = g(f(a))$$

We write $\gg$ to denote the composition of functions. For a stream transformation $f : \mathsf{ST}(A, B)$, we write $f \downarrow x$ to indicate that $f(x)$ is terminated, and $f \uparrow x$ to mean that $f(x)$ is not terminated. We say that f *halts* on $x \in A^\dagger$, denoted $f \Downarrow x$, if the following hold: (1) $f(x)$ is terminated, and (2) $f(y)$ is not terminated for every $y < x$. For a sequence $u \in A^*$, we define $(u\triangleleft) \cdot \triangleleft^{-1} = u$ and $u \cdot \triangleleft^{-1} = u$. In other words, $(- \cdot \triangleleft^{-1})$ is the operation that removes the end-of-stream marker from a sequence if it is present. Using this notation, we define the *temporal sequencing* combinator seq, and the *temporal iteration* combinator iter in Figure 6. Notice that $\mathsf{iter}(f)$ is defined under the assumption that $f(\varepsilon)$ is not terminated. This is required, because otherwise the computation of

$$\frac{\mathtt{op} : A \to B}{\begin{array}{r} f = \mathsf{map}(\mathtt{op}) : \mathsf{ST}(A,B) \\ f(\varepsilon) = \varepsilon \\ f(ua) = f(u) \cdot \mathtt{op}(a) \\ f(u\triangleleft) = f(u)\triangleleft \end{array}} \qquad \frac{\mathtt{p} : A \to \mathtt{Bool}}{\begin{array}{r} f = \mathsf{filter}(\mathtt{p}) : \mathsf{ST}(A,A) \\ f(\varepsilon) = \varepsilon,\ f(u\triangleleft) = f(u)\triangleleft \\ f(ua) = f(u) \cdot a, \text{ if } \mathtt{p}(a) = \mathtt{true} \\ f(ua) = f(u), \text{ if } \mathtt{p}(a) = \mathtt{false} \end{array}} \qquad \frac{b : B \qquad \mathtt{op} : B \times A \to B}{\begin{array}{r} f = \mathsf{aggr}(b, \mathtt{op}) : \mathsf{ST}(A,B) \\ f(\varepsilon) = \varepsilon \\ f(ua) = f(u) \cdot \mathsf{fold}(b, \mathtt{op}, ua) \\ f(u\triangleleft) = f(u)\triangleleft \end{array}}$$

$$\frac{b : B \qquad \mathtt{op} : B \times A \to B}{\begin{array}{r} f = \mathsf{reduce}(b, \mathtt{op}) : \mathsf{ST}(A,B) \\ f(u) = \varepsilon \\ f(u\triangleleft) = \mathsf{fold}(b, \mathtt{op}, u)\triangleleft \end{array}} \qquad \frac{A : \mathtt{Type}}{\begin{array}{r} f = \mathsf{flatten}(A) : \mathsf{ST}(\mathsf{List}(A), A) \\ (\partial f)(\varepsilon) = \varepsilon,\ (\partial f)(u\triangleleft) = \triangleleft \\ (\partial f)(ul) = \mathsf{extract}(l) \end{array}}$$

$$\frac{A : \mathtt{Type} \qquad \mathtt{out} : \mathsf{List}(B)}{\begin{array}{r} f = \mathsf{emit}(A, \mathtt{out}) : \mathsf{ST}(A,B) \\ (\partial f)(\varepsilon) = \mathsf{extract}(\mathtt{out})\triangleleft \\ (\partial f)(u) = \varepsilon, \text{ if } |u| > 0, \\ (\partial f)(u\triangleleft) = \varepsilon \end{array}} \qquad \frac{f : \mathsf{ST}(A,B) \qquad f \uparrow \varepsilon}{\begin{array}{r} g = \mathsf{iter}(f) : \mathsf{ST}(A,B) \\ g(x) = f(x), \text{ if } f \uparrow x \\ g(ux) = f(u)\triangleleft^{-1} \cdot g(x), \text{ if } f \Downarrow u \\ g(u\triangleleft) = f(u\triangleleft)\triangleleft^{-1} \cdot f(\varepsilon), \text{ if } f \Downarrow u\triangleleft \end{array}} \qquad \frac{f : \mathsf{ST}(A,B) \qquad g : \mathsf{ST}(A,B)}{\begin{array}{r} h = \mathsf{seq}(f,g) : \mathsf{ST}(A,B) \\ h(x) = f(x), \text{ if } f \uparrow x \\ h(ux) = f(u)\triangleleft^{-1} \cdot g(x), \text{ if } f \Downarrow u \\ h(u\triangleleft) = f(u\triangleleft)\triangleleft^{-1} \cdot g(\varepsilon), \text{ if } f \Downarrow u\triangleleft \end{array}}$$

$$\frac{\mathtt{p} : A \to \mathtt{Bool}}{\begin{array}{r} f = \mathsf{takeUntil}(\mathtt{p}) : \mathsf{ST}(A,A) \\ (\partial f)(\varepsilon) = \varepsilon \text{ and } (\partial f)(u\triangleleft) = \varepsilon \\ (\partial f)(ua) = \varepsilon, \text{ if } \mathtt{p}'(u) = \mathtt{true} \\ (\partial f)(ua) = a, \text{ if } \mathtt{p}'(u) = \mathtt{false} \text{ and } \mathtt{p}(a) = \mathtt{false} \\ (\partial f)(ua) = a\triangleleft, \text{ if } \mathtt{p}'(u) = \mathtt{false} \text{ and } \mathtt{p}(a) = \mathtt{true} \end{array}} \qquad \frac{f : \mathsf{ST}(A,B) \qquad g : \mathsf{ST}(A,B)}{\begin{array}{r} h = \mathsf{par}(f,g) : \mathsf{ST}(A,B) \\ (\partial h)(u) = (\partial f)(u)\triangleleft^{-1} \cdot (\partial g)(u)\triangleleft^{-1}, \text{ if } f \uparrow u \text{ or } g \uparrow u. \\ (\partial h)(u) = (\partial f)(u)\triangleleft^{-1} \cdot (\partial g)(u)\triangleleft^{-1}\triangleleft, \text{ if } f \downarrow u \text{ and } g \Downarrow u. \\ (\partial h)(u) = (\partial f)(u)\triangleleft^{-1} \cdot (\partial g)(u)\triangleleft^{-1}\triangleleft, \text{ if } f \Downarrow u \text{ and } g \downarrow u. \\ (\partial h)(u) = \varepsilon, \text{ otherwise} \end{array}}$$

$$\frac{\mathsf{key} : A \to K \qquad f : \mathsf{ST}(A,B)}{\begin{array}{r} g = \mathsf{groupBy}(\mathsf{key}, f) : \mathsf{ST}(A,B) \\ (\partial g)(\varepsilon) = \varepsilon \\ (\partial g)(ua) = (\partial f)(u|_{\mathsf{key}(a)} \cdot a)\triangleleft^{-1} \\ (\partial g)(u\triangleleft) = \left(\prod_{i=1}^{n} (\partial f)(u|_{k_i}\triangleleft)\triangleleft^{-1}\right)\triangleleft \end{array}} \qquad \frac{n \geq 1 \qquad f : \mathsf{ST}(A,B)}{\begin{array}{r} g = \mathsf{tWindow}(n, f) : \mathsf{ST}(A,B) \\ g(u) = f(u)\triangleleft^{-1}, \text{ if } |u| < n \\ g(u\triangleleft) = g(u)\triangleleft, \text{ if } |u| < n \\ g(ux) = f(u\triangleleft)\triangleleft^{-1} \cdot g(x), \text{ if } |u| = n \end{array}}$$

Fig. 6. Semantics: map, filter, reduce, aggr, seq, iter, emit, flatten, takeUntil, par, groupBy and tWindow.

iter(f) would enter an infinite loop of halting and restarting without consuming any input. The definition of groupBy(key, f) in Figure 6 uses the incremental viewpoint for notational brevity. For a sequence $u \in A^*$ and a key $k \in K$, we write $u|_k$ to denote the subsequence of u that contains the items whose key is equal to k. More formally, $\varepsilon|_k = \varepsilon$, $(ua)|_k = u|_k$ if $\mathsf{key}(a) \neq k$, and $(ua)|_k = u|_k \cdot a$ if $\mathsf{key}(a) = k$. In the third case $(\partial g)(u\triangleleft)$ of the groupBy(key, f) definition, we use $\prod$ as a generalization of concatenation to arbitrarily many arguments. Moreover, $k_1, k_2, \ldots, k_n$ is taken to be the sequence of keys that appear in the sequence u (in their order of appearance). The definition of the transformations emit and flatten both require the *extract combinator* extract : List(A) $\to A^*$, given by extract(nil) $= \varepsilon$ and extract(cons(a, l)) $= a \cdot$ extract(l). In the definition of takeUntil(p), we lift the predicate p on A to the predicate p′ on A^*, where, for a sequence $u \in A^*$, $\mathtt{p}'(u) = \mathtt{true}$ if there exists an item a in u such that $\mathtt{p}(a) = \mathtt{true}$, and $\mathtt{p}'(u) = \mathtt{false}$ if for all a in u such that $\mathtt{p}(a) = \mathtt{false}$. In the definition of the parallel composition, given stream transformations f and g, par(f, g) emits the end-of-stream marker only when f and g have both terminated. Finally, in the definition of tWindow(n, f) and emit(out) in Figure 6, $|u|$ denotes the length of u.

Lemma 4.1 (**Expressive Completeness**). Let $f : \mathsf{ST}(A,B)$ be a stream transformation. If f is computable, then there is a query of type $\mathtt{Q}(A,B)$ that computes it.

Proof. A streaming algorithm for f can be viewed as an automaton $\mathcal{A} = (S, init, next, out)$, where S is a (potentially infinite) state space, $init \in S$ is the initial state, $next : S \times (A \cup \{\triangleleft\}) \to S$

is the state transition function, and $out : S \to B^\dagger$ is the output function. We put $S = A^\dagger$, $init = \varepsilon$, $next(s, a) = sa$, $next(s, \triangleleft) = s\triangleleft$, $out(s) = (\partial f)(s)$, and $out(s\triangleleft) = (\partial f)(s\triangleleft)$ for every $s \in A^*$ and $a \in A$. The execution of $\mathcal{A}$ is an obvious generalization of the execution of finite-state automata. Since f is computable, so are $next$ and out. It remains to show that the execution of $\mathcal{A}$ can be encoded by a query of type $\mathtt{Q}(A, B)$. Let $\delta : S \times A \to S$ be the restriction of $next$ to $S \times A$. Define

$$\mathtt{f} = \mathtt{par}(\mathtt{emit}(A, [init]), \mathtt{aggr}(init, \delta), \mathtt{reduce}(init, \delta) \gg \mathtt{map}(x \texttt{ -> } next(x, \triangleleft)) : \mathtt{Q}(A, S).$$

The query `f` transforms the stream of input items (of type A) into the stream of states (of type S) that the automaton $\mathcal{A}$ goes through. Let $\vartheta : S \to \mathtt{Bool}$ be the function that indicates whether a state is halting or not, that is, for all $s \in S$, $\vartheta(s) = \mathtt{true}$ iff $out(s)$ ends with $\triangleleft$. Then, the query $\mathtt{g} = \mathtt{takeUntil}(\vartheta) : \mathtt{Q}(S, S)$ takes a stream of states and echoes them up until (and including) the first halting state. Let $o : S \to B^*$ be given by $o(s) = out(s) \cdot \triangleleft^{-1}$. The query $\mathtt{h} = \mathtt{flatten}(\mathtt{map}(o)) : \mathtt{Q}(S, B)$ takes a stream of states as inputs and emits the corresponding flattened output. Finally, the query $\mathtt{f} \gg \mathtt{g} \gg \mathtt{h} : \mathtt{Q}(A, B)$ computes the stream transformation f. □

We will use an example to illustrate the construction in the proof of Lemma 4.1. Suppose the input is $\bar{a} = a_1a_2a_3\triangleleft$. Define the states $s_0 = init$, $s_{i+1} = next(s_i, a_{i+1})$, and $t_i = next(s_i, \triangleleft)$. The output of `f` on $\bar{a}$ is $\bar{s} = s_0s_1s_2s_3t_3\triangleleft$. Suppose that s_2 is the first halting state. Then, the output of `g` for input $\bar{s}$ is $\bar{t} = s_0s_1s_2\triangleleft$. Finally, the output of `h` for input $\bar{t}$ is $o(s_0) \cdot o(s_1) \cdot o(s_2) \cdot \triangleleft = f(\bar{a})$.

Significance of Lemma 4.1. Having a formal denotational semantics allows us to pose a well-defined question of whether the query language is expressively complete. It is not difficult to prove the lemma, as the essence of the proof is encoding the execution of a transducer (state machine) that implements the desired stream transformation. Notice, however, that the lemma can fail in subtle ways if some small changes are made to the language. For example, if the constructs `flatten` and `par` are removed, then the language becomes incomplete as all queries produce at most one output item per input item. If the `emit` construct is removed, then no query can emit output at the beginning of the computation (i.e., before the input is seen). So, Lemma 4.1 serves as a sanity check that there is no important omission from the list of the language constructs.

5 JAVA IMPLEMENTATION

In this section, we will describe the implementation of StreamQL. We have chosen to implement StreamQL as an embedded domain-specific language in Java (effectively a Java library) in order to allow for easy integration with user-defined types and operations. The implementation covers all the core constructs we introduced in Section 2 and also provides a rich set of specialized algorithms for real-world applications, such as efficient algorithms for aggregation over windows and a variety of signal processing primitives: FFT (Fast Fourier Transform), Hilbert Transform, FIR (Finite Impulse Response) filters, and IIR (Infinite Impulse Response) filters.

The left part of Figure 7 gives a simple example of a StreamQL program in Java. Given a signal measurement of type `VT` that contains a double value in the field of `val`, the query `sum` of type `Q` computes the sum of the values of the measurements. The method `eval` returns an object that encapsulates the evaluation algorithm for the query. The methods `init` and `next` are used to initialize the memory and consume data items. When the input stream terminates, the `end` method is invoked.

We define two interfaces, `Sink` and `Algo`, to describe the streaming computation in a push-based manner. The `Algo` interface is used to implement stream transformations. The `Sink` interface is similar to the `Observer` interface of Rx. It is used for specifying a sink that consumes a stream. A sink consumes a stream with two methods, `next` and `end`, that are used for stream elements and the end-of-stream marker respectively. The top-right part of Figure 7 shows the definition of

```
// VT is the type of measurements,
// which contains a double value as val
Iterator<VT> stream = ... // input stream
// sink of the output stream
Sink<Double> sink = ...

// sum of the measurements
Q<VT,Double> sum =
    QL.aggr(0.0, (s, vt) -> s + vt.val);
// evaluation of the query
Algo<VT,Double> exe = sum.eval();
// connect the output of query to sink
exe.connect(sink);
// execution loop
exe.init();
while (stream.hasNext()) {
    VT vt = stream.next();
    exe.next(vt);
}
exe.end();
```

```
abstract class Sink<T> {
    // deal with incoming items
    abstract void next(T item);
    // deal with the end-of-stream marker
    abstract void end();
}
```

```
class Printer<T> extends Sink<T>{
    // print each arrived data item
    void next(T item) { print(item); }
    // print "Job done" when input ends
    void end() { print("Job done"); }
}
```

```
abstract class Algo<A,B> extends Sink<A>{
    // connect to a sink
    abstract void connect(Sink<B> sink);
    // initialize or reset the memory
    abstract void init();
}
```

Fig. 7. Example that computes the sum of a nonempty sequence of measurements (left). The Sink interface (top-right). An instance of Sink (mid-right). The Algo interface (bottom-right).

the `Sink` interface in Java, and the mid-right part presents an instance of `Sink` that prints each incoming data item and the end-of-stream marker to the console.

The `Algo` interface is used for describing the evaluation algorithm of a query. An implementation of `Algo` specifies how the input stream is transformed into the output stream. The bottom-right part of Figure 7 shows the definition of the `Algo` interface, which extends the `Sink` interface because it consumes a stream. The `connect` method connects the algorithm to a sink, and the `init` method initializes/resets the state of the algorithm.

RxJava and similar libraries use nested streams (e.g., `Observable`⟨`Observable`⟩ in RxJava) to decompose the input stream into windows or grouped sub-streams. StreamQL, on the other hand, eliminates the overheads introduced by the construction of nested streams. For instance, the left part of Figure 8 presents the algorithm for tumbling windows. Given the size of the window and a sub-query, the tumbling window splits the stream into contiguous non-overlapping windows and applies the sub-query to the data items of each window. We provide an algorithm for tumbling windows with a small memory footprint. The algorithm sends the incoming data items to the sub-query and maintains a counter that records the number of data items in the window. When the current window becomes full, the algorithm resets the internal state of the sub-query. In contrast, to implement a tumbling window, Rx-like libraries construct nested streams to decompose the input stream as several stream objects. Whenever the current window becomes full, a new window is created and it is represented as a stream object. Moreover, to produce the output stream, some additional overhead is introduced to merge (`flatMap` in RxJava) the output sub-streams that are created from the individual windows. The construction of nested streams is a source of overheads. In Section 6, we experimentally validate these overheads and observe that our implementation of tumbling windows is faster compared to Rx-like libraries. Our Java library avoids the overheads of nested streams for all other computations that involve stream decomposition, such as sliding windows and key-based partitioning (group-by construct).

The `Algo` interface facilitates the implementation of the constructs `seq` and `iter`. The windows of Rx and Trill cannot encode these constructs, as `seq` and `iter` decompose the input stream in a data-dependent way. Recall that a query `seq`(`f`, `g`) starts executing as `f` and after `f` terminates it continues executing as `g`. This computation thus splits the input stream into two parts. If Rx was

```
class TWnd<A,B> extends Algo<A,B> {
  private final int size;
  // algorithm of the sub-query
  private final Algo<A,B> algo;
  private Sink<B> sink;
  // counter of items in the window
  private int cnt;
  TWnd(int size, Algo<A,B> algo) {
    this.size = size;
    this.algo = algo;
  }
  void connect(Sink<B> sink) {
    this.sink = sink;
    // sink for the sub-query that transfers
    // its output to the output of TWnd
    Sink<B> subSink = new Sink<B>() {
      void next(B item) { sink.next(item); }
      // discard the end-of-stream marker
      void end() { }
    };
    algo.connect(subSink);
  }
  void init() {  cnt = 0;  }
  void next(A item) {
    // reset algo if old window is full
    if (cnt == 0) {  algo.init();  }
    algo.next(item);
    cnt = (cnt + 1) % size;
    if (cnt == 0) {  algo.end();  }
  }
  void end() {  sink.end();  }
}
```

```
class Seq<A,B> extends Algo<A,B>{
  // algorithms of the sub-queries
  private final Algo<A,B> left;
  private final Algo<A,B> right;
  // pointer of the current active algorithm
  private Algo<A,B> active;
  Seq(Algo<A,B> left, Algo<A,B> right) {
    this.left = left;
    this.right = right;
  }
  void connect(Sink<B> sink) {
    // sink for the left algorithm that
    // activates right when left terminates
    Sink<B> leftSink = new Sink<B>() {
      void next(B item) {
        sink.next(item);
      }
      void end() {
        active = right;
        right.init();
      }
    };
    left.connect(leftSink);
    right.connect(sink);
  }
  void init() {
    active = left;
    left.init();
  }
  void next(A item) {  active.next(item);  }
  void end() {  active.end();  }
}
```

Fig. 8. The Java implementation of the tumbling window (left) and stream sequencing (right) constructs

somehow extended to accommodate these constructs, its design would still incur the overheads associated with the representation of sub-streams as nested stream objects. Figure 8 shows our implementation of the `seq` construct. Notice that all data items are simply routed to the appropriate algorithm/sink without creating any intermediate objects. In the `connect` method, we provide a sink for the `left` algorithm that activates the `right` algorithm once `left` terminates. The implementation of `iter` uses similar ideas.

6 EXPERIMENTAL EVALUATION

We evaluate the performance of our library using four benchmarks: (1) a micro-benchmark that focuses on basic operators, (2) a benchmark for pattern detection in real-time stock market data, (3) the popular NEXMark benchmark [Tucker et al. 2002], and (4) TAQMark for the analysis of high-frequency market data. We compare our implementation with RxJava, Rx.NET, Reactor, Siddhi, and Trill. These are chosen because they are all lightweight and high-performance streaming engines that offer rich APIs and have well-maintained implementations. RxJava, Reactor, and Siddhi are implemented in Java, while Rx.NET and Trill are implemented in .NET. Since there is no .NET implementation of StreamQL, we only test Rx.NET and Trill using the micro benchmark to obtain a very rough comparison, and leave the .NET implementation of StreamQL as future work.

Experimental setup. The experiments were executed in Ubuntu 16.04 LTS on a desktop computer equipped with an Intel Xeon(R) E3-1241 v3 CPU (4 cores) with 16 GB of memory (DDR3 at 1600 MHz). For Java programs, we used version 1.8.0_181-b13 of the JDK, and we set the maximum heap size at 3.5 GB. For .NET programs, we used the NET Core 3.1.100 SDK with C# 8.0. To test the performance of Trill, we set the batch size to 1000 for its columnar representation (as suggested by

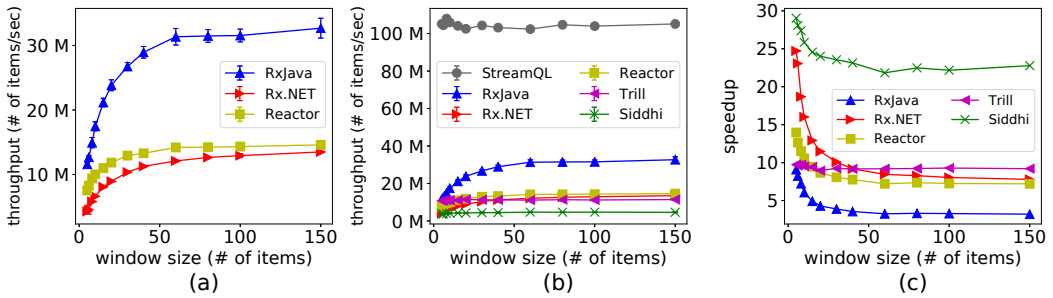


Fig. 9. (a) and (b) show the throughput (y-axis) of twnd-sum queries with different window size (x-axis). (c) shows the throughput speedup (y-axis) of StreamQL comparing to other libraries.

its official documentation [Trill 2020]). All data points in our experiments represent the average of at least five runs, with error bars showing the standard deviation.

Overhead of nested streams. In contrast to Rx-like libraries, StreamQL avoids the overhead brought by the construction of nested streams. To quantify the improvement in performance gained by reducing these overheads, we test the throughput of queries that involve the decomposition of the input stream, which leads to the construction of nested streams in Rx-like libraries. We create an input stream of timestamped integers of the form $\{\texttt{ts}, \texttt{val}\}$ as "$\{1,1\}, \{2,2\}, ..., \{n,n\}$", where both the timestamp `ts` and the data value `val` are integers, and n is set to be 100 million. We use the tumbling window construct to split the input into non-overlapping regions and compute the sum of integers in each region. In our experiments, we measure the throughput of this twnd-sum query with various window sizes, where the window size specifies the number of integers contained in a window, and a small window size leads to the creation of a large number of windows. This leads to the allocation of many nested stream objects in Rx-like libraries.

Figure 9 shows the throughput of queries that sum the integers over tumbling windows of various sizes. Figure 9(a) presents the throughput in libraries (RxJava, Rx.NET, and Reactor) that decompose the input by nested streams. The results indicate that the construction of nested streams is a significant overhead: when the window size is small (e.g., 4), the throughput is 3 times lower than when the window size is large (e.g., 150). This suggests that the intensive construction of nested streams largely decreases the throughput of stream processing. In Figure 9(b), we show the throughput of our StreamQL library along with RxJava, Rx.NET, Reactor, Trill and Siddhi. In comparison to RxJava, Rx.NET, Reactor and Siddhi, the throughput of StreamQL queries remains stable with regards to different size of tumbling windows. Finally, Figure 9(c) presents the throughput speedup of StreamQL with respect to other libraries. By avoiding the construction of nested streams, StreamQL provides significant performance speedup when the input stream is decomposed into a large number of windows, and it is more than 3 times faster than Rx-like libraries even when the size of the window is large (e.g. 150) as StreamQL also eliminates the overhead of flattening nested sub-stream objects. The performance of Trill is stable as its windowing operator works by altering the interval timestamp of each stream element. Trill enriches each raw input data item with a "temporal validity" annotation (interval timestamp) to obtain a stream element of type `StreamEvent`. This choice for the temporal and data model has certain semantic advantages, but it also introduces computational costs to incorporate this additional time information.

To further investigate the overheads, we analyze the memory allocation of the `twnd(sum)` computation for StreamQL, RxJava, and Reactor. We measured the size of total allocated memory on the heap, and we estimated the memory size for intermediate data structures by subtracting

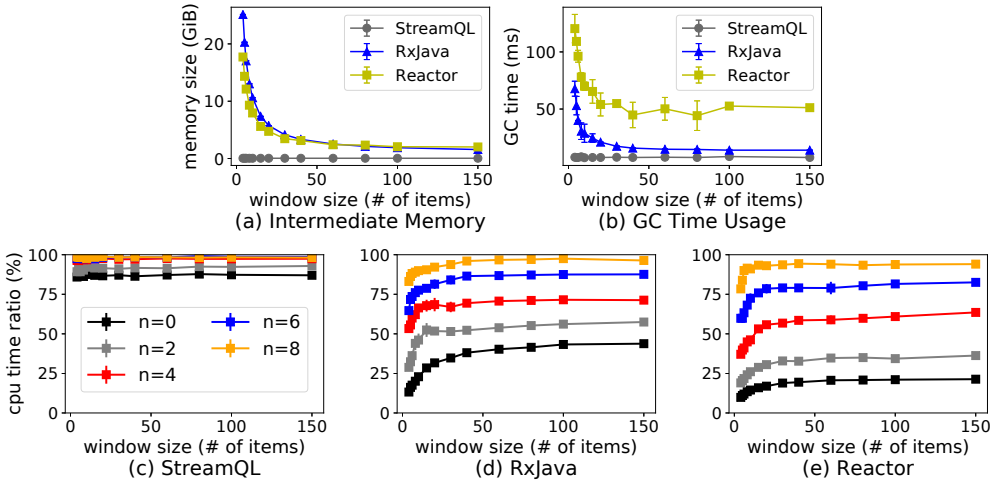


Fig. 10. Figure (a) shows the size of intermediate memory (GiB, y-axis) of twnd-sum queries. Figure (b) shows the garbage collection time (ms, y-axis) of twnd-sum queries. Figure (c), (d), and (e) show the ratio of the execution time on the aggregation calculation to the total execution time for StreamQL, RxJava, and Reactor.

the memory allocated for the input/output streams from the total allocated memory. Figure 10(a) presents the estimation. The StreamQL implementation allocates almost zero additional memory since it uses a counter to record the number of items in the window and maintains the aggregate using a single variable. RxJava and Reactor allocate a significant amount of intermediate memory when the window size is small, which is mainly composed of the nested stream objects (e.g., `InnerObserver` in RxJava), subscription objects (e.g., `UnicastSubject` in RxJava), and internal data buffers (e.g., `SpscLinkedArrayQueue` in RxJava). We also measured the garbage collection (GC) time of the `twnd(sum)` computation. The result is shown in Figure 10(b). In addition, we measured the ratio of GC time to total CPU execution time on the main thread, and we observed this ratio is lower than 1% for StreamQL, RxJava, and Reactor. Moreover, we measured the CPU execution time for the `twnd(sum)` query. We estimated the overheads by measuring the time ratio of the cost of the aggregation calculation to the total execution time (higher ratio indicates lower overheads). To illustrate, we aggregated an integer stream using the function $f(agg, x) = agg + 3^n \cdot x$, where n is an integer that controls the complexity of the computation, and when $n = 0$, the aggregation is exactly the sum computation. Figure 10(c), (d), and (e) show the results for StreamQL, RxJava, and Reactor. In our observation, when the size of the tumbling window is small (10 items) and the computation per item is cheap ($n = 0$), more than 70% of the time cost in RxJava (80% in Reactor) on the main thread is caused by function calls related with the construction of nested streams, which include the creation of the stream objects (e.g., `InnerObservable.create()` in RxJava), the communication between the stream objects and the corresponding data consumers (e.g., `ObservableFlatMap.drain()` in RxJava), and the management of the data subscriptions (e.g., `Subject.create()` in RxJava).

Efficient sliding window aggregation. The StreamQL library provides efficient algorithms for aggregations over sliding windows. When the aggregation is given by a binary function `op`, then efficient algorithms [Hirzel et al. 2017; Li et al. 2005; Tangwongsan et al. 2015] can be given for the special cases where (1) `op` is associative, and (2) `op` is associative and invertible. Consider the aggregation `max`. The implementation of `max` over a sliding window of size n requires a buffer of size n (to store the contents of the window) [Datar et al. 2002]. Every time a new item arrives, the

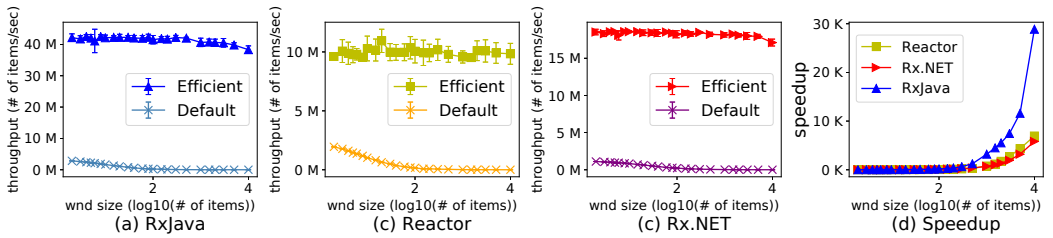


Fig. 11. The left-three figures show the throughput (y-axis) of swnd-sum queries with different window sizes ($\log_{10}$(# of items), x-axis) and a fixed sliding interval in RxJava, Reactor, and Rx.NET. The right-most figure shows the throughput speedup (y-axis) of efficient implementations compared with the default settings.

naive algorithm scans through the entire window to calculate the new maximum, which requires $O(n)$ time. Since `max` is associative, there is a better algorithm, which maintains a tree of partial aggregates and only needs $O(\log n)$ time at each step [Arasu and Widom 2004]. For a function `op` that is invertible (e.g., sum and count) there is an obvious efficient algorithm, which requires $O(1)$ time at each step ("add" the new item, "subtract" the item falling off the window). Libraries such as Trill and Siddhi also provide efficient algorithms for sliding window aggregation. RxJava, Rx.NET, and Reactor, on the other hand, do not incorporate such algorithms. After creating custom constructs for efficient sliding window aggregation in RxJava, Rx.NET, and Reactor, we compare the performance of our customized constructs with the default constructs. We measure the throughput of queries that sum the integers over sliding windows that have a fixed sliding interval (one item) but different lengths, and we show the results in Figure 11. The results suggest that the efficient algorithm is more than 5000 times faster than the default algorithm when the window size is large (e.g., 10,000) and it is about 5 times faster when the size of the window is small.

Remark. To make fair comparisons among StreamQL, Rx, and Reactor, for all queries that involve aggregation over sliding windows in the following benchmarks, we program them using our customized constructs and then test their throughput.

Micro Benchmark. We run several basic streaming computations over an input stream of timestamped integers, and the queries are: `map` selects the value of each input item, `filter` removes items with odd integer values, and `sum` calculates the sum of the values. The qualifiers `tw`, `sw` and `grp` refer to aggregation over tumbling windows, sliding windows, and key-based partitions respectively. The qualifier `gtw`(`gsw`) refers to tumbling (sliding) window aggregation over key-based partitions. All the queries were executed with a stream of timestamped integers. For computations that involve key-based partitioning, we set the key function as $\mathsf{key}(x) = x.\mathtt{val}\ \mathit{mod}\ 100$, and for windows, we always fix the window size to be 100 and the sliding interval to be 1 (if it is a sliding window). Moreover, for sequential aggregation, although StreamQL provides built-in constructs for arithmetical computations, we write queries using primitives to make fair comparisons with libraries that do not provide such features. For example, we use `reduce`(0, (`sum`, `x`)->(`sum+x.val`)) to compute the sum of input values.

The results are shown in Figure 12. (1) StreamQL is 2–100 times faster than Siddhi. The reason for the performance gap is that Siddhi creates complex event objects to ingress the data and queues the data to achieve streaming composition; both of these bring computational overheads. (2) In the comparison to RxJava, for trivial operators (filtering, mapping and aggregation), there is no significant difference between StreamQL and RxJava. For operations that involve tumbling windows and key-based partitioning, StreamQL is about 2–3 times faster than Rx-like libraries, as it eliminates the overheads brought by nested streams. For sliding window operations, we measure

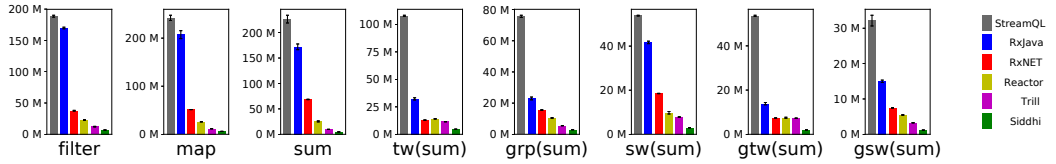


Fig. 12. Throughput (# items/sec, y-axis) of StreamQL, RxJava, Rx.NET, Reactor, Trill, and Siddhi (left to right) in the micro benchmark.

the performance of customized constructs in RxJava to make fair comparisons, where the constructs implement efficient algorithms for sliding window aggregation. Therefore, there is no significant difference between StreamQL and RxJava (without these constructs, StreamQL is more than 100 times faster than RxJava). (3) StreamQL is 3–10 times faster than Reactor. In design, Reactor and Rx share many similarities, and Reactor also suffers from the overheads brought by nested streams. (4) In the comparison to Rx.NET and Trill, the results are largely influenced by the performance gap between the Java framework and the .NET framework. Therefore, we can only have a rough comparison between StreamQL and these two libraries.

Stock Benchmark. The stock benchmark [Agrawal et al. 2008; Chandramouli et al. 2010; Demers et al. 2007; Gyllstrom et al. 2007] uses a synthetic stream of stock quotes that are of the form {`stockId`, `price`, `volume`, `timestamp`}. We consider four families of queries for pattern detection: ***S1*** detects three consecutive quotes whose `volumes` are all above a threshold, ***S2*** detects three consecutive quotes whose `prices` increase continuously, ***S3*** detects five consecutive quotes whose prices fluctuate in a V-pattern (down, down, up, up), and ***S4*** detects price peaks. For every query family there are three variants: ***a.*** concerns a specific stock, ***b.*** detects the pattern for each stock independently, and ***c.*** considers each stock over an 1-minute tumbling window. The experimental results are given in Figure 13. For pattern detection that concerns a specific stock (S1a, S2a, S3a, and S4a), StreamQL is about 10-15 times faster than RxJava and Reactor, and 70-100 times faster than Siddhi. When the computation involves excessive stream partitioning (queries labeled by variants b and c), the total computational cost mostly depends on the cost of stream partitioning, and StreamQL is around 3 times faster than RxJava, 4 times faster than Reactor, and 10-20 times faster than Siddhi since it eliminates the construction of nested streams.

NEXMark. NEXMark [Tucker et al. 2002] is about monitoring an on-line auction system. Its data stream has four kinds of events: `Person` represents the registration of a new user, `Item` indicates the start of an auction for a specified item, `Bid` records a bid made for an auctioned item, and `Close` indicates the end of an auction. We used eight queries: ***N1*** converts the price of each bid to another currency, ***N2*** searches for auctions of a specific set of items, ***N3*** counts the number of bids submitted in the US, ***N4*** calculates the average selling price of items for each auction category, ***N5*** outputs the item with the most bids in the last 10 minutes, ***N6*** computes the average selling price per seller for their last 10 closed auctions, ***N7*** finds the highest bid every 1 minute, and ***N8*** calculates, every 12 hours, the number of new user registrations . Figure 13 shows the experimental results: StreamQL is 1.1–3 times faster than RxJava, 1.5–15 times faster than Reactor, and 5–50 times faster than Siddhi.

TAQ Benchmark. We use data from the NYSE TAQ database [TAQ 2019], which collects real-time *trades* and *quotes* reported on the U.S. Consolidated Tape (where billions of entries are recorded per day). We implemented the following queries: ***T1*** filters out events that are outside normal NYSE hours, ***T2*** computes the running average price for each stock, ***T3*** computes the average price for each stock over a tumbling window, ***T4*** (***T5***) computes the sequence of trading intervals (durations

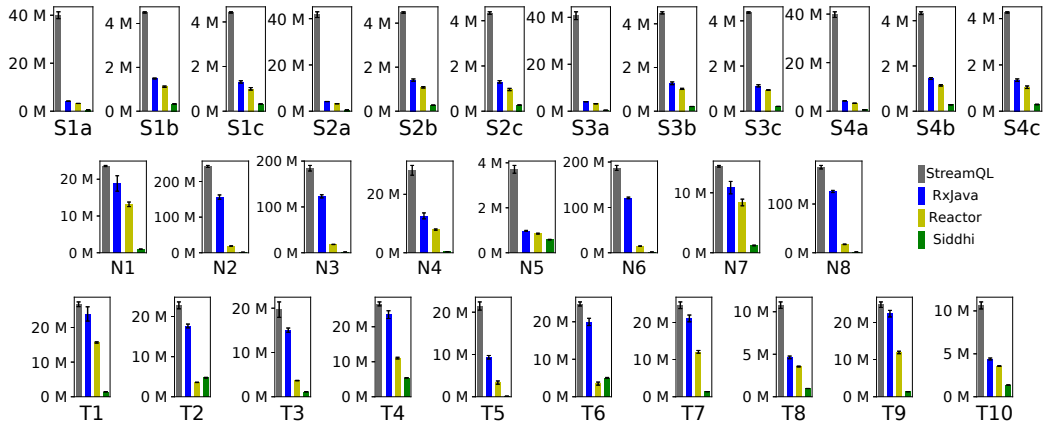


Fig. 13. Throughput (# items/sec) of StreamQL, RxJava, Reactor, and Siddhi (left to right) in the stock benchmark (S1a-S4c), NEXMark (N1-N8), and TAQMark (T1-10).

between consecutive transactions) for a specific (each) stock, ***T6*** counts the number of odd lots (trades with less than 100 shares) for each stock, ***T7*** (***T8***) computes the best bid and offer for a specific (each) stock over a tumbling window, and ***T9*** (***T10***) calculates the true value estimate for a specific (each) stock over a tumbling window. Figure 13 shows the results: StreamQL is generally 1.2–2 times faster than RxJava, 2–10 times faster than Reactor, and 8–100 times faster than Siddhi.

6.1 Multicore Processing

Applications that handle massive data streams have scalability requirements and would therefore benefit from a multicore implementation of the streaming engine. Here we discuss some steps we have taken towards a parallel implementation of StreamQL. We also explain some challenges that the parallelization of StreamQL presents. Due to these challenges, we leave the full development of a parallel StreamQL implementation for future work.

StreamQL allows the arbitrary composition of dataflow/relational operators (whose parallelization is typically easier) and of sequence-dependent temporal operators (which introduce unavoidable synchronization points). For example, pipelines and group-by queries can be easily parallelized. On the other hand, the temporal operators that we provide (e.g., `take`, `search`, `seq`, `iter`) are inherently sequential. If the top-level operators, for example, are nested `iter` and `seq` (e.g., ECG peak detection), then there are frequent synchronization points and hence no benefit from parallelization.

The full elaboration of parallelization in the context of StreamQL language presents unique challenges concerning the preservation of the sequential semantics. We should note that preserving the sequential semantics is crucial in applications with strict requirements of correctness and reproducibility (e.g., healthcare or financial applications). There are certain parallelization patterns where the sequential semantics is preserved without any effort. These include pipelines and queries of the form `groupBy(reduce(op))`. In the general case, however, StreamQL allows order-dependent computations, for which naive parallelization introduces nondeterminism and unpredictability. A relatively simple case concerns *stateless* queries `f`, which can be parallelized by splitting the input stream across several instances of `f` (workers) and then merging the output sub-streams produced by the workers. The merging stage can introduce disorder, but the correct order can be restored by assigning sequence numbers to all data items so that they are reordered appropriately. Such simple order-restoration schemes with sequence numbers [Schneider et al. 2015], punctuations/markers [Mamouras et al. 2019], and timestamps (Trill) are not sufficient for the full StreamQL

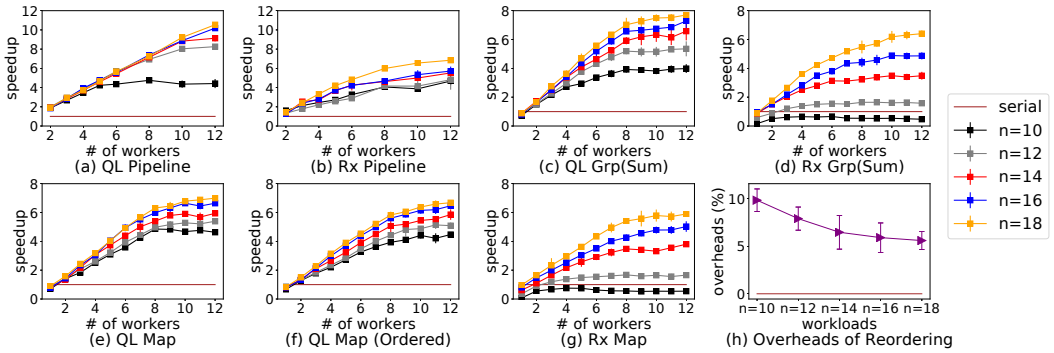


Fig. 14. Comparison of the scalability between StreamQL and RxJava for pipeline, map, and groupBy(sum).

language, because of the deep nesting of order-dependent constructs. For example, $\texttt{par}(\texttt{f}_1, \texttt{f}_2)$ can be parallelized correctly using (scalar) sequence numbers, but $\texttt{par}(\texttt{f}_1 \gg \texttt{par}(\texttt{g}_1, \texttt{g}_2), \texttt{f}_2)$ would require a kind of vector sequence numbers to deal with the nesting of parallelization constructs. StreamIt [Thies et al. 2002] also addresses semantic preservation, but it sidesteps these difficult cases by restricting the language to a "synchronous" subset: it has copy and round-robin splitters but disallows value-based stream splitting. RxJava and Siddhi offer no guarantee of semantics preservation upon parallelization.

StreamQL contains constructs that can be parallelized to exploit multicore architectures. We will describe now a ***preliminary investigation*** of parallelizing `pipeline`, $\texttt{groupBy}(\texttt{reduce}(\texttt{op}))$, and `map`. For these constructs, we provide parallel implementations, and we evaluate the throughput scalability against RxJava. The experiments were executed in Ubuntu 16.04 with an Intel Xeon(R) E7-4830 v2 CPU (10 cores with hyperthreading) and 52 GB of memory. In the experiments, we first measured the throughput of the sequential implementation using StreamQL and RxJava. Then, we measured the throughput of the parallel implementation for a variable number of workers and calculated the speedup compared to the sequential implementation. The parallel implementation has one thread for each worker, as well as two threads for the stream splitter and the output merger. In the experiments, we used a function f, parameterized by n, which is given by $f(x) = 3^n \cdot x$. This allows us to assess scalability as the primitive operation f becomes increasingly compute-heavy (by increasing the parameter n).

Pipeline. To evaluate the performance of the parallel `pipeline`, we consider a pipeline of 120 stages, each of which is $\texttt{map}(f)$. The pipeline stages are partitioned uniformly across the worker threads. For example, when there are 12 workers, each worker handles the computation of 10 consecutive stages. The result is shown in Figure 14(a) and (b).

GroupBy(Reduce). To evaluate the parallel version of $\texttt{groupBy}(\texttt{reduce}(\texttt{op}))$, we assign each input integer x to a partition using the key-extraction function $key(x) = x \bmod 1024$. The aggregation function op is given by $\texttt{op}(agg, x) = agg + f(x)$. Recall that the parameter n controls the computational cost of f and hence op. Each worker thread handles the computation for a subset of the keys. To ensure correctness, we send input items with the same key to the same worker using a splitter thread. Since we are performing a `reduce` operation, the merger thread simply collects the final per-key aggregates when the input stream terminates. Figure 14(c) and (d) show the results for StreamQL and Rx respectively.

Map. We also consider a parallel version of $\texttt{map}(f)$. In our implementation, each worker has an input buffer and an output buffer. The worker reads the data item from the input buffer, processes the data, and sends the result to the output buffer. Different from operations like `pipeline` and

`groupBy(reduce(op))`, to preserve the sequential semantics of the `map` query, we need to ensure the correct order of the output stream by reordering output items. We provide two implementations: (I) one that preserves the sequential semantics, and (II) one that allows the worker threads to output items as they become available (nondeterministic output order). For implementation (I), a round-robin splitter assigns input items to workers, and the merger thread merges the output sub-stream from workers in a round-robin fashion. The results are shown in Figure 14(e), (f), (g).

Based on our experimental results, we observe that, for compute-heavy workloads (e.g., $n = 18$), StreamQL's parallel `pipeline`, `groupBy(reduce(op))`, and `map` queries scale almost linearly for up to 8 worker threads. The processor on which the experiments are executed has 10 cores and 2 threads are assigned to the main thread (input stream generation and splitter) and the thread that merges and consumes the final output. Overall, we observe that the StreamQL is competitive against RxJava in terms of throughput scalability for the three classes of queries that we investigated. We leave for future work a more comprehensive experimental comparison. We have also investigated for `map` queries the overhead that is caused by ensuring the correct output order. As seen in Figure 14(h), the semantics-preserving implementation is 6% to 10% slower than the one that does not preserve the sequential semantics. We leave for future work a more thorough investigation of the overhead caused by other techniques for ensuring semantics-preserving parallelization.

7 CASE STUDY: ABP PULSE DETECTION

We will use StreamQL to specify a streaming algorithm for Arterial Blood Pressure (ABP) pulse detection [O'Rourke 1971; Zong et al. 2003]. This is a complex streaming computation, and is difficult to express with existing languages for stream processing. As we will see, StreamQL allows a natural high-level specification of the algorithm. The use of a streaming query language for medical monitoring applications has been considered in [Abbas et al. 2018, 2019].

The ABP signal is collected from the MIT-BIH Polysomnographic database [Ichimaru and Moody 1999]. The signal measurements are of type $\texttt{VT} = \{\texttt{val} : \texttt{V}, \texttt{ts} : \texttt{T}\}$, where `val` is the value of the signal and `ts` is the timestamp. The signal is uniformly sampled at a frequency of 250 Hz. Figure 15 shows a snippet of an ABP signal containing 3 ABP pulses (around 3 seconds). The ABP waveform contains rich information about the cardiovascular system (e.g., heart rate, systolic, mean, and diastolic arterial pressures). Reliable ABP pulse detection is crucial for extracting this information.

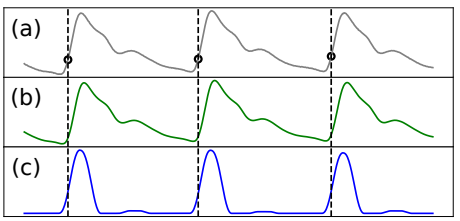
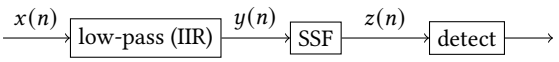


Fig. 15. Examples of (a) raw ABP signal with onset labels, (b) low-pass filtered signal, and (c) SSF signal.

First, the algorithm preprocesses the signal stream using a low-pass IIR filter and a slope sum function (SSF), and then it performs the detection of the pulse onset.



The low-pass filter suppresses high frequency noise, and is defined by $y(n) = 2y(n-1) - y(n-2) + x(n) - 2x(n-5) + x(n-10)$. The SSF is defined by $z(n) = \sum_{0 \le i \le 31} max(0, d(n-i))$, where $d(n) = y(n) - y(n-1)$. It enhances the up-slope of the ABP pulse and restrains the remainder of the pressure waveform. The query $\texttt{getVTP} : \texttt{Q}(\texttt{VT}, \texttt{VTP})$ annotates each item $\{\texttt{val}, \texttt{ts}\}$ of the input stream with an additional component `pval`, which is the result of the preprocessing. The type $\texttt{VTP} = \{\texttt{val} : \texttt{V}, \texttt{ts} : \texttt{T}, \texttt{pval} : \texttt{V}\}$ extends `VT` with this additional component. These preprocessed values have a phase shift of 20 ms (5 samples), which is introduced by low-pass filtering.

The detection of ABP onset is described by the following rules: **R1.** In intervals where the SSF value exceeds a threshold `Thred` (i.e. a tentative pulse), the algorithm selects the first and the maximum SSF values. **R2.** The pulse detection is accepted only if the difference between the first and the maximum SSF values exceeds 100. **R3.** When the pulse is accepted, the algorithm chooses the first sample that crosses the threshold as the onset point. The detected onset is adjusted by 20 ms (5 samples) to compensate for the phase shift of low-pass filtering. **R4.** After an onset is detected, to avoid double detection of the same pulse, the detection falls silent for 300 ms. Figure 16 shows the StreamQL implementation of the detection algorithm.

8 RELATED WORK

There is a large body of work on ***streaming database systems*** such as STREAM [Arasu et al. 2016], Aurora [Abadi et al. 2003], Borealis [Abadi et al. 2005], CACQ [Madden et al. 2002], TelegraphCQ [Chandrasekaran et al. 2003], Niagara [Naughton et al. 2001], Gigascope [Cranor et al. 2003], Nile [Hammad et al. 2004], Microsoft's CEDR [Barga et al. 2007], and StreamInsight [Ali et al. 2009]. The languages supported by these database systems (for example, CQL [Arasu et al. 2006]) are typically variants of SQL with additional streaming constructs for sliding windows. These languages are limited in their ability to perform computations that depend on the order of arrival of data items, such as detecting complex patterns.

```
# preprocess the signal
lowPass = IIR({−1, 2}, {1, 0, 0, 0, 0, −2, 0, 0, 0, 0, 1})
diff = sWindow(2, 1, (x, y) -> y − x)
sum = sWindow(32, 1, reduce((x, y) -> (y > 0) ? (x + y) : x))
ssf = diff ≫ sum
preProc = map(x -> x.val) ≫ lowPass ≫ ssf
getVTP = annotate(preProc, (x, y) -> ⟨x.val, x.ts, y⟩)
# select signal interval containing a peak (R1)
pulse = takeWhen(x -> x.pval > Thred, x -> x.pval < Thred)
# select the first element in interval as the onset sample
# find the measurement with the maximum preprocessed value,
# and store them as a pair ⟨first, max⟩
select = reduce(x -> ⟨x, x⟩,
                ( ⟨f, m⟩, x) -> ⟨f, (x.pval > m.pval) ? x : m⟩)
# examine the detected pulse (R2) and project the onset
getOnset = filterMap( ⟨f, m⟩ -> m.pval − f.pval > 100, ⟨f, m⟩ -> f)
detect1 = getVTP ≫ pulse ≫ select ≫ check ≫ getOnset

rft = skip(75)  # after detecting the ABP onset, apply R4
detectAll = seq(detect1, iter(rft ≫ detect1))
subShift = map(x -> x.ts − 5)  # compensate for phase shift
ABPDetection = detectAll ≫ subShift
```

Fig. 16. StreamQL program for ABP pulse detection.

There is a variety of systems for ***distributed stream processing*** that are based on the distributed dataflow model of computation: S4 [Neumeyer et al. 2010], IBM Streams [Biem et al. 2010], MapReduce Online [Condie et al. 2010], Storm [Toshniwal et al. 2014], Summingbird [Boykin et al. 2014], Heron [Kulkarni et al. 2015], Naiad [Murray et al. 2013], Spark Streaming [Zaharia et al. 2012, 2013], Flink [Carbone et al. 2015], Google's MillWheel [Akidau et al. 2013], Samza [Noghabi et al. 2017], and Beam [Beam 2020]. Many of these systems (including Storm, Heron, and Samza) expose a low-level API for specifying a dataflow graph of operators, where each operator is given as a function for handling events. Other systems (such as IBM Streams, Spark Streaming, Flink, and Beam) provide a higher-level API to describe streaming computations. IBM Streams and Flink provide special operators for pattern detection based on regular expressions. All these systems are engineered to provide high throughput, scalability, load balancing, load shedding, fault tolerance and recovery. Spark Streaming, for example, uses micro-batches to achieve high throughput and offers fault-recovery guarantees. Since StreamQL is implemented as a Java library, it can be used in conjunction with these systems. For example, an individual node of the dataflow graph (called "bolt" in Storm) can be programmed using StreamQL.

Several languages and tools have been proposed for ***Complex Event Processing*** (CEP), which is concerned with the detection of complex patterns over event streams. These languages are typically based on regular expressions [Zemke et al. 2007] and implemented using variants of finite-state automata: SQL-TS [Sadri et al. 2004], SASE [Gyllstrom et al. 2007], Cayuga [Demers et al. 2007], and

SPL MatchRegex [Hirzel 2012]. ZStream [Mei and Madden 2009] uses tree-based query plans instead of automata for query evaluation. Several streaming engines (Trill [Chandramouli et al. 2014], Esper [EsperTech 2006], Siddhi [Suhothayan et al. 2011], Flink [Carbone et al. 2015], Oracle Stream Analytics [Corporation 2019] and IBM Streams [Biem et al. 2010]) provide specialized operators or extensions for CEP. A common problem with CEP implementations is that they need to explore all possible ways in which the input stream could be parsed to match a regular expression, which causes an exponential worst-case blowup in space requirements. StreamQL solves this issue by restricting the nondeterminism of the regular combinators (concatenation and iteration), which results in an implementation with no overhead for pattern detection.

The ***Synchronous Dataflow*** (SDF) languages and models of computation are useful for making explicit the parallelism present in streaming computations that arise in the embedded software domain, including signal processing [Lee and Messerschmitt 1987] and embedded controller design [Benveniste et al. 1991; Berry and Gonthier 1992; Caspi et al. 1987]. These formalisms are restrictions of Kahn's process networks [Kahn 1974; Kahn and MacQueen 1977]. The StreamIt language, in particular, provides a general framework for streaming signal processing with efficient execution on multi-core architectures [Thies et al. 2002]. Compared to the SDF formalisms, StreamQL provides higher-level abstractions that are suitable for a more general class of streaming applications.

There are several ***lightweight streaming engines*** which are implemented as libraries within a general-purpose host language. This facilitates integration with other systems and enables the reuse of existing code for application-specific tasks. Microsoft's Trill [Chandramouli et al. 2014] is a high-performance streaming library that employs a batched-columnar data representation and dynamic compilation. Trill provides mechanisms to handle out-of-order events and provides extensions for pattern matching and signal processing [Chandramouli et al. 2018, 2010; Nikolic et al. 2017]. Esper [EsperTech 2006] and Siddhi [Suhothayan et al. 2011] are lightweight engines for CEP and streaming analytics. They provide rich collections of operators including SQL-based constructs, windows, and pattern matching. Java Stream [Oracle 2014] and Stream Fusion [Kiselyov et al. 2017] provide a simpler streaming API, which can be used for processing static collections of data. StreamQRE [Mamouras et al. 2017] (see also [Alur and Mamouras 2017]) has been proposed as an integration of unambiguous regular expressions with quantitative calculations and other streaming constructs such as streaming composition. The core of StreamQRE is related to transducers with registers that can hold values [Alur et al. 2020, 2017a, 2019] and other related models [Alur et al. 2017b]. KSQL [Jafarpour et al. 2019] is a streaming SQL engine for Apache Kafka, which provides an interactive query interface with lightweight SQL syntax. InfluxDB [InfluxDB 2020] is a database system that is optimized for time-series data. It implements two languages for querying data: (1) InfluxQL, which is based on the syntax of SQL, and (2) Flux [Flux 2020], which has a more functional syntax.

The survey paper [Bainomugisha et al. 2013] explores various approaches for ***reactive programming***. The work on functional reactive programming (FRP) focuses on the transformation of time-varying values (signals). Some representative early languages are Fran [Elliott and Hudak 1997] and Yampa [Nilsson et al. 2003]. Several subsequent frameworks embed FRP in imperative languages, such as Flapjax [Meyerovich et al. 2009], Frappé [Courtney 2001], and Scala.React [Maier and Odersky 2012]. FRP has been used mainly for the development of event-driven and interactive applications such as GUIs. Elm [Czaplicki and Chong 2013] is a practical FRP language focused on easy creation of responsive GUIs. Libraries like Rx [Meijer 2012; ReactiveX 2020], Reactor [Reactor 2020], and Akka Streams [Lightbend 2020] have some similarity to the FRP languages, but deal with streams of events rather than signals. These libraries expose data streams as push-based collections (for example, `Observable` in Rx) and provide APIs for transforming these streams.

StreamQL is related to traditional functional stream processing and ***stream fusion*** in particular. Stream fusion refers to the automatic elimination of intermediate structures (e.g., lists) in the execution of a program. This technique, derived from deforestation [Wadler 1990], has been studied extensively [Coutts et al. 2007; Gibbons 2004; Gill et al. 1993; Johann 2001; Kiselyov et al. 2017; Svenningsson 2002; Takano and Meijer 1995]. In StreamQL, the design of stream transformers (transductions) allows us to avoid the materialization of intermediate streams/lists by using function calls to propagate the elements as they are being generated. Some of StreamQL's constructs (e.g., filter and map) can be understood in the framework of stream fusion. Other StreamQL constructs (e.g., sliding windows and temporal iteration) are not encountered in prior work on stream fusion. It is an interesting question whether these constructs can be incorporated into the framework of stream fusion.

The WaveScope project [Girod et al. 2007] highlights the need to combine event-stream processing with ***signal processing*** for applications that make use of sensor-generated data streams. XStream [Girod et al. 2008] is a streaming engine that was created to serve this goal. TrillDSP [Nikolic et al. 2017] enriches Trill with functionality for signal processing. Our StreamQL library also provides operators for manipulating signals (e.g., support for FFT, FIR and IIR filtering, and so on).

9 CONCLUSION

We have introduced StreamQL, a language that specifies complex streaming computations as combinations of stream transformations. StreamQL integrates relational, dataflow, and temporal language constructs, thus providing an expressive and modular high-level approach for programming streaming analyses. We have implemented StreamQL as a Java library, and we have compared its performance against three popular streaming engines (RxJava, Reactor, and Siddhi) using four benchmarks. In benchmark with realistic workloads, the throughput of the StreamQL library is consistently higher: 1.1–10 times higher than RxJava, 1.2–20 times higher than Reactor, and 5–100 times higher than Siddhi. We have used StreamQL to easily prototype a streaming algorithm for ABP (Arterial Blood Pressure) pulse detection, a complex computation that is difficult to express in other streaming languages.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their constructive comments. This research was supported in part by US National Science Foundation award 2008096.

REFERENCES

D. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, C. Erwin, E. Galvez, M. Hatoun, A. Maskey, A. Rasin, A. Singer, M. Stonebraker, N. Tatbul, Y. Xing, R. Yan, and S. Zdonik. 2003. Aurora: A Data Stream Management System. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data (SIGMOD '03)*. ACM, New York, NY, USA, 666–666. https://doi.org/10.1145/872757.872855

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. 2005. The Design of the Borealis Stream Processing Engine. In *Proceedings of the 2nd Biennial Conference on Innovative Data Systems Research (CIDR '05)*. 277–289. http://cidrdb.org/cidr2005/papers/P23.pdf

Houssam Abbas, Rajeev Alur, Konstantinos Mamouras, Rahul Mangharam, and Alena Rodionova. 2018. Real-time Decision Policies with Predictable Performance. *Proceedings of the IEEE, Special Issue on Design Automation for Cyber-Physical Systems* 106, 9 (2018), 1593–1615. https://doi.org/10.1109/JPROC.2018.2853608

Houssam Abbas, Alena Rodionova, Konstantinos Mamouras, Ezio Bartocci, Scott A. Smolka, and Radu Grosu. 2019. Quantitative Regular Expressions for Arrhythmia Detection. *IEEE/ACM Transactions on Computational Biology and Bioinformatics* 16, 5 (2019), 1586–1597. https://doi.org/10.1109/TCBB.2018.2885274

Jagrati Agrawal, Yanlei Diao, Daniel Gyllstrom, and Neil Immerman. 2008. Efficient Pattern Matching over Event Streams. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD '08)*. ACM, New York,

NY, USA, 147–160. https://doi.org/10.1145/1376616.1376634

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

M. H. Ali, C. Gerea, B. S. Raman, B. Sezgin, T. Tarnavski, T. Verona, P. Wang, P. Zabback, A. Ananthanarayan, A. Kirilov, M. Lu, A. Raizman, R. Krishnan, R. Schindlauer, T. Grabs, S. Bjeletich, B. Chandramouli, J. Goldstein, S. Bhat, Ying Li, V. Di Nicola, X. Wang, David Maier, S. Grell, O. Nano, and I. Santos. 2009. Microsoft CEP Server and Online Behavioral Targeting. *Proceedings of the VLDB Endowment* 2, 2 (2009), 1558–1561. https://doi.org/10.14778/1687553.1687590

Rajeev Alur, Dana Fisman, Konstantinos Mamouras, Mukund Raghothaman, and Caleb Stanford. 2020. Streamable Regular Transductions. *Theoretical Computer Science* 807 (2020), 15–41. https://doi.org/10.1016/j.tcs.2019.11.018

Rajeev Alur and Konstantinos Mamouras. 2017. An Introduction to the StreamQRE Language. *Dependable Software Systems Engineering* 50 (2017), 1–24. https://doi.org/10.3233/978-1-61499-810-5-1

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

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

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

Rajeev Alur, Konstantinos Mamouras, and Dogan Ulus. 2017b. Derivatives of Quantitative Regular Expressions. In *Models, Algorithms, Logics and Tools: Essays Dedicated to Kim Guldstrand Larsen on the Occasion of His 60th Birthday*, Luca Aceto, Giorgio Bacci, Giovanni Bacci, Anna Ingólfsdóttir, Axel Legay, and Radu Mardare (Eds.). Lecture Notes in Computer Science, Vol. 10460. Springer, Cham, 75–95. https://doi.org/10.1007/978-3-319-63121-9_4

Arvind Arasu, Brian Babcock, Shivnath Babu, John Cieslewicz, Mayur Datar, Keith Ito, Rajeev Motwani, Utkarsh Srivastava, and Jennifer Widom. 2016. STREAM: The Stanford Data Stream Management System. In *Data Stream Management: Processing High-Speed Data Streams*, Minos Garofalakis, Johannes Gehrke, and Rajeev Rastogi (Eds.). Springer, Berlin, Heidelberg, 317–336. https://doi.org/10.1007/978-3-540-28608-0_16

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

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

Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. 2002. Models and Issues in Data Stream Systems. In *Proceedings of the Twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS '02)*. ACM, New York, NY, USA, 1–16. https://doi.org/10.1145/543613.543615

Engineer Bainomugisha, Andoni Lombide Carreton, Tom van Cutsem, Stijn Mostinckx, and Wolfgang de Meuter. 2013. A Survey on Reactive Programming. *Comput. Surveys* 45, 4, Article 52 (Aug. 2013), 34 pages. https://doi.org/10.1145/2501654.2501666

Roger S. Barga, Jonathan Goldstein, Mohamed H. Ali, and Mingsheng Hong. 2007. Consistent Streaming Through Time: A Vision for Event Stream Processing. In *CIDR 2007, Third Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 7-10, 2007, Online Proceedings*. www.cidrdb.org, Asilomar, CA, USA, 363–374. http://cidrdb.org/cidr2007/papers/cidr07p42.pdf

Beam 2020. Apache Beam: An advanced unified programming model. https://beam.apache.org/. [Online; Last revised 9 May 2020].

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

Albert Benveniste, Paul Le Guernic, and Christian Jacquemot. 1991. Synchronous Programming with Events and Relations: The SIGNAL Language and its Semantics. *Science of Computer Programming* 16, 2 (1991), 103–149. https://doi.org/10.1016/0167-6423(91)90001-E

Gérard Berry and Georges Gonthier. 1992. The Esterel Synchronous Programming Language: Design, Semantics, Implementation. *Science of Computer Programming* 19, 2 (1992), 87–152. https://doi.org/10.1016/0167-6423(92)90005-V

Reinhold Orglmeister Bert-Uwe Köhler, Carsten Hennig. 2002. The Principles of Software QRS Detection. *IEEE Engineering in Medicine and Biology Magazine* 21, 1 (Jan 2002), 42–57. https://doi.org/10.1109/51.993193

Alain Biem, Eric Bouillet, Hanhua Feng, Anand Ranganathan, Anton Riabov, Olivier Verscheure, Haris Koutsopoulos, and Carlos Moran. 2010. IBM Infosphere Streams for Scalable, Real-time, Intelligent Transportation Services. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD '10)*. ACM, New York, NY, USA, 1093–1104. https://doi.org/10.1145/1807167.1807291

Oscar Boykin, Sam Ritchie, Ian O'Connell, and Jimmy Lin. 2014. Summingbird: A Framework for Integrating Batch and Online MapReduce Computations. *Proceedings of the VLDB Endowment* 7, 13 (Aug. 2014), 1441–1451. https://doi.org/10.14778/2733004.2733016

Lars Brenna, Alan Demers, Johannes Gehrke, Mingsheng Hong, Joel Ossher, Biswanath Panda, Mirek Riedewald, Mohit Thatte, and Walker White. 2007. Cayuga: A High-Performance Event Processing Engine. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data (SIGMOD '07)*. Association for Computing Machinery, New York, NY, USA, 1100–1102. https://doi.org/10.1145/1247480.1247620

Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink: Stream and Batch Processing in a Single Engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36, 4 (2015), 28–38. http://sites.computer.org/debull/A15dec/p28.pdf

Paul Caspi, Daniel Pilaud, Nicholas Halbwachs, and John A. Plaice. 1987. LUSTRE: A Declarative Language for Real-time Programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '87)*. ACM, New York, NY, USA, 178–188. https://doi.org/10.1145/41625.41641

Badrish Chandramouli, Jonathan Goldstein, Mike Barnett, Robert DeLine, Danyel Fisher, John C. Platt, James F. Terwilliger, and John Wernsing. 2014. Trill: A High-performance Incremental Query Processor for Diverse Analytics. *Proceedings of the VLDB Endowment* 8, 4 (2014), 401–412. https://doi.org/10.14778/2735496.2735503

Badrish Chandramouli, Jonathan Goldstein, and Yinan Li. 2018. Impatience Is a Virtue: Revisiting Disorder in High-Performance Log Analytics. In *Proceedings of the IEEE 34th International Conference on Data Engineering (ICDE 2018)*. IEEE, 677–688. https://doi.org/10.1109/ICDE.2018.00067

Badrish Chandramouli, Jonathan Goldstein, and David Maier. 2010. High-performance Dynamic Pattern Matching over Disordered Streams. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 220–231. https://doi.org/10.14778/1920841.1920873

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. 2003. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *Proceedings of the First Biennial Conference on Innovative Data Systems Research (CIDR '03)*. http://cidrdb.org/cidr2003/program/p24.pdf

Agnishom Chattopadhyay and Konstantinos Mamouras. 2020. A Verified Online Monitor for Metric Temporal Logic with Quantitative Semantics. In *Proceedings of the 20th International Conference on Runtime Verification (RV 2020) (Lecture Notes in Computer Science, Vol. 12399)*, Jyotirmoy Deshmukh and Dejan Ničković (Eds.). Springer, Cham, 383–403. https://doi.org/10.1007/978-3-030-60508-7_21

Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. 2010. MapReduce Online. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation* (San Jose, California) *(NSDI'10)*. USENIX Association, USA, 21. https://www.usenix.org/legacy/events/nsdi10/tech/full_papers/condie.pdf

Oracle Corporation. 2019. Oracle Stream Analytics. https://www.oracle.com/middleware/technologies/stream-processing.html. [Online; Accessed November 11, 2019].

Antony Courtney. 2001. Frappé: Functional Reactive Programming in Java. In *Proceedings of the 3rd International Symposium on Practical Aspects of Declarative Languages (PADL '01)*, I. V. Ramakrishnan (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 29–44. https://doi.org/10.1007/3-540-45241-9_3

Duncan Coutts, Roman Leshchinskiy, and Don Stewart. 2007. Stream Fusion: From Lists to Streams to Nothing at All. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming (ICFP '07)*. Association for Computing Machinery, New York, NY, USA, 315–326. https://doi.org/10.1145/1291151.1291199

Chuck Cranor, Theodore Johnson, Oliver Spataschek, and Vladislav Shkapenyuk. 2003. Gigascope: A Stream Database for Network Applications. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data (SIGMOD '03)*. ACM, New York, NY, USA, 647–651. https://doi.org/10.1145/872757.872838

Evan Czaplicki and Stephen Chong. 2013. Asynchronous Functional Reactive Programming for GUIs. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. ACM, New York, NY, USA, 411–422. https://doi.org/10.1145/2491956.2462161

Ben D'Angelo, Sriram Sankaranarayanan, Cesar Sanchez, Will Robinson, Bernd Finkbeiner, Henny B. Sipma, Sandeep Mehrotra, and Zohar Manna. 2005. LOLA: Runtime Monitoring of Synchronous Systems. In *Proceedings of the 12th International Symposium on Temporal Representation and Reasoning (TIME'05)*. IEEE, New York, NY, USA, 166–174. https://doi.org/10.1109/TIME.2005.26

Mayur Datar, Aristides Gionis, Piotr Indyk, and Rajeev Motwani. 2002. Maintaining Stream Statistics over Sliding Windows. *SIAM J. Comput.* 31, 6 (2002), 1794–1813. https://doi.org/10.1137/S0097539701398363

Alan Demers, Johannes Gehrke, Biswanath Panda, Mirek Riedewald, Varun Sharma, and Walker White. 2007. Cayuga: A General Purpose Event Monitoring System. In *Proceedings of the 3rd Biennial Conference on Innovative Data Systems Research (CIDR '07)*. 412–422. http://cidrdb.org/cidr2007/papers/cidr07p47.pdf

Jyotirmoy V. Deshmukh, A. Donzé, Shromona Ghosh, Xiaoqing Jin, Garvit Juniwal, and Sanjit A. Seshia. 2017. Robust Online Monitoring of Signal Temporal Logic. *Formal Methods in System Design* 51, 1 (01 Aug 2017), 5–30. https://doi.org/10.1007/s10703-017-0286-7

Conal Elliott and Paul Hudak. 1997. Functional Reactive Animation. In *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming (ICFP '97)*. ACM, New York, NY, USA, 263–273. https://doi.org/10.1145/258948.258973

Inc EsperTech. 2006. Esper. http://www.espertech.com/esper/. [Online; Accessed April 3, 2019].

Flux 2020. Query and code together with Flux. https://www.influxdata.com/products/flux/. [Online; Accessed 11 August 2020].

Jeremy Gibbons. 2004. Streaming Representation-Changers. In *Mathematics of Program Construction*. Springer, Berlin, Heidelberg, 142–168. https://doi.org/10.1007/978-3-540-27764-4_9

Andrew Gill, John Launchbury, and Simon L. Peyton Jones. 1993. A Short Cut to Deforestation. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture (FPCA '93)*. Association for Computing Machinery, New York, NY, USA, 223–232. https://doi.org/10.1145/165180.165214

Lewis Girod, Yuan Mei, Ryan Newton, Stanislav Rost, Arvind Thiagarajan, Hari Balakrishnan, and Samuel Madden. 2007. The Case for a Signal-Oriented Data Stream Management System. In *Proceedings of the 3rd Biennial Conference on Innovative Data Systems Research (CIDR '07)*. 397–406. http://cidrdb.org/cidr2007/papers/cidr07p45.pdf

Lewis Girod, Yuan Mei, Ryan Newton, Stanislav Rost, Arvind Thiagarajan, Hari Balakrishnan, and Samuel Madden. 2008. XStream: a Signal-Oriented Data Stream Management System. In *2008 IEEE 24th International Conference on Data Engineering*. IEEE, New York, NY, USA, 1180–1189. https://doi.org/10.1109/ICDE.2008.4497527

Daniel Gyllstrom, Eugene Wu, Hee-Jin Chae, Yanlei Diao, Patrick Stahlberg, and Gordon Anderson. 2007. SASE: Complex Event Processing over Streams. In *Proceedings of the 3rd Biennial Conference on Innovative Data Systems Research (CIDR '07)* (Asilomar, CA, USA). 407–411. http://cidrdb.org/cidr2007/papers/cidr07p46.pdf

Moustafa A. Hammad, Mohamed F. Mokbel, Mohamed H. Ali, Walid G. Aref, Ann Christine Catlin, Ahmed K. Elmagarmid, Mohamed Eltabakh, Mohamed G. Elfeky, Thanaa M. Ghanem, Robert Gwadera, Ihab F. Ilyas, Mirette Marzouk, and Xiaopeng Xiong. 2004. Nile: A Query Processing Engine for Data Streams. In *Proceedings of the 20th International Conference on Data Engineering (ICDE '04)*. IEEE, New York, NY, USA, 851–851. https://doi.org/10.1109/ICDE.2004.1320080

Klaus Havelund and Grigore Roşu. 2004. Efficient Monitoring of Safety Properties. *International Journal on Software Tools for Technology Transfer* 6, 2 (2004), 158–173. https://doi.org/10.1007/s10009-003-0117-6

Martin Hirzel. 2012. Partition and Compose: Parallel Complex Event Processing. In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems (DEBS '12)*. ACM, New York, NY, USA, 191–200. https://doi.org/10.1145/2335484.2335506

Martin Hirzel, Scott Schneider, and Kanat Tangwongsan. 2017. Sliding-Window Aggregation Algorithms: Tutorial. In *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems (DEBS '17)*. ACM, New York, NY, USA, 11–14. https://doi.org/10.1145/3093742.3095107

Y. Ichimaru and G. B. Moody. 1999. Development of the polysomnographic database on CD-ROM. *Psychiatry and Clinical Neurosciences* 53, 2 (1999), 175–177. https://doi.org/10.1046/j.1440-1819.1999.00527.x

InfluxDB 2020. InfluxDB: Real-time visibility into stacks, sensors and systems. https://www.influxdata.com/. [Online; Accessed 11 August 2020].

Hojjat Jafarpour, Rohan Desai, and D Guy. 2019. KSQL: Streaming SQL Engine for Apache Kafka.. In *EDBT*. www.OpenProceedings.org, 524–533. https://openproceedings.org/2019/conf/edbt/EDBT19_paper_329.pdf

Patricia Johann. 2001. Short Cut Fusion: Proved and Improved. In *Semantics, Applications, and Implementation of Program Generation*. Springer, Berlin, Heidelberg, 47–71. https://doi.org/10.1007/3-540-44806-3_4

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

Gilles Kahn and David B. MacQueen. 1977. Coroutines and Networks of Parallel Processes. *Information Processing* 77 (1977), 993–998. https://hal.inria.fr/inria-00306565/

Oleg Kiselyov, Aggelos Biboudis, Nick Palladinos, and Yannis Smaragdakis. 2017. Stream Fusion, to Completeness. *SIGPLAN Not.* 52, 1 (Jan. 2017), 285–299. https://doi.org/10.1145/3093333.3009880

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

Edward Ashford Lee and David G. Messerschmitt. 1987. Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing. *IEEE Trans. Comput.* C-36, 1 (Jan 1987), 24–35. https://doi.org/10.1109/TC.1987.5009446

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

Martin Leucker and Christian Schallhart. 2009. A Brief Account of Runtime Verification. *The Journal of Logic and Algebraic Programming* 78, 5 (2009), 293–303. https://doi.org/10.1016/j.jlap.2008.08.004 The 1st Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS'07).

Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A. Tucker. 2005. No Pane, No Gain: Efficient Evaluation of Sliding-window Aggregates over Data Streams. *SIGMOD Rec.* 34, 1 (March 2005), 39–44. https://doi.org/10.1145/1058150.1058158

Inc Lightbend. 2020. Akka Streams. https://akka.io/. [Online; Accessed March 10, 2020].

Samuel Madden, Mehul Shah, Joseph M. Hellerstein, and Vijayshankar Raman. 2002. Continuously Adaptive Continuous Queries over Streams. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data (SIGMOD '02)*. ACM, New York, NY, USA, 49–60. https://doi.org/10.1145/564691.564698

Ingo Maier and Martin Odersky. 2012. *Deprecating the Observer Pattern with Scala.React.* Technical Report. EPFL. 20 pages. http://infoscience.epfl.ch/record/176887

Konstantinos Mamouras. 2020. Semantic Foundations for Deterministic Dataflow and Stream Processing. In *Proceedings of the 29th European Symposium on Programming (ESOP '20) (Lecture Notes in Computer Science, Vol. 12075)*, Peter Müller (Ed.). Springer, Berlin, Heidelberg, 394–427. https://doi.org/10.1007/978-3-030-44914-8_15

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

Konstantinos Mamouras, Caleb Stanford, Rajeev Alur, Zachary G. Ives, and Val Tannen. 2019. Data-Trace Types for Distributed Stream Processing Systems. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. ACM, New York, NY, USA, 670–685. https://doi.org/10.1145/3314221.3314580

Konstantinos Mamouras and Zhifu Wang. 2020. Online Signal Monitoring with Bounded Lag. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2020). https://doi.org/10.1109/TCAD.2020.3013053

Yuan Mei and Samuel Madden. 2009. ZStream: A Cost-based Query Processor for Adaptively Detecting Composite Events. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data (SIGMOD '09)*. ACM, New York, NY, USA, 193–206. https://doi.org/10.1145/1559845.1559867

Erik Meijer. 2012. Your Mouse is a Database. *Commun. ACM* 55, 5 (May 2012), 66–73. https://doi.org/10.1145/2160718.2160735

Leo A. Meyerovich, Arjun Guha, Jacob Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. 2009. Flapjax: A Programming Language for Ajax Applications. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '09)*. ACM, New York, NY, USA, 1–20. https://doi.org/10.1145/1640089.1640091

George B. Moody. 2018. Single-channel QRS detector. https://www.physionet.org/physiotools/wag/sqrs-1.htm. [Online; Last revised 22 March 2018].

Rajeev Motwani, Jennifer Widom, Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Gurmeet Manku, Chris Olston, Justin Rosenstein, and Rohit Varma. 2003. Query Processing, Approximation, and Resource Management in a Data Stream Management System. In *Proceedings of the First Biennial Conference on Innovative Data Systems Research (CIDR '03)*. www.cidrdb.org. http://cidrdb.org/cidr2003/program/p22.pdf

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

Jeffrey F. Naughton, David J. DeWitt, David Maier, Ashraf Aboulnaga, Jianjun Chen, Leonidas Galanis, Jaewoo Kang, Rajasekar Krishnamurthy, Qiong Luo, Naveen Prakash, et al. 2001. The Niagara Internet Query System. *IEEE Data Engineering Bulletin* (2001).

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

Milos Nikolic, Badrish Chandramouli, and Jonathan Goldstein. 2017. Enabling Signal Processing over Data Streams. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17)*. ACM, New York, NY, USA, 95–108. https://doi.org/10.1145/3035918.3035935

Henrik Nilsson, John Peterson, and Paul Hudak. 2003. Functional Hybrid Modeling. In *Practical Aspects of Declarative Languages*. Springer Berlin Heidelberg, Berlin, Heidelberg, 376–390. https://doi.org/10.1007/3-540-36388-2_25

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

Oracle. 2014. Java Stream. https://docs.oracle.com/javase/8/. [Online; Accessed March 31, 2019].

Michael F. O'Rourke. 1971. The arterial pulse in health and disease. *American Heart Journal* 82, 5 (1971), 687 – 702. https://doi.org/10.1016/0002-8703(71)90340-1

J. Pan and W. J. Tompkins. 1985. A Real-Time QRS Detection Algorithm. *IEEE Transactions on Biomedical Engineering* BME-32, 3 (March 1985), 230–236. https://doi.org/10.1109/TBME.1985.325532

ReactiveX. 2020. ReactiveX. http://reactivex.io/. [Online; Accessed March 10, 2020].

Reactor 2020. Project Reactor: Create Efficient Reactive Systems. https://projectreactor.io/. [Online; Accessed 28 March 2020].

RxJava 2020. RxJava: Reactive Extensions for the JVM. Available at https://github.com/ReactiveX/RxJava. [Online; accessed March 10, 2020].

Reza Sadri, Carlo Zaniolo, Amir Zarkesh, and Jafar Adibi. 2004. Expressing and Optimizing Sequence Queries in Database Systems. *ACM Transactions on Database Systems* 29, 2 (2004), 282–318. https://doi.org/10.1145/1005566.1005568

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

Sriskandarajah Suhothayan, Kasun Gajasinghe, Isuru Loku Narangoda, Subash Chaturanga, Srinath Perera, and Vishaka Nanayakkara. 2011. Siddhi: A Second Look at Complex Event Processing Architectures. In *Proceedings of the 2011 ACM Workshop on Gateway Computing Environments (GCE '11)*. ACM, New York, NY, USA, 43–50. https://doi.org/10.1145/2110486.2110493

Josef Svenningsson. 2002. Shortcut Fusion for Accumulating Parameters & Zip-like Functions. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02)*. Association for Computing Machinery, New York, NY, USA, 124–132. https://doi.org/10.1145/581478.581491

Akihiko Takano and Erik Meijer. 1995. Shortcut Deforestation in Calculational Form. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture (FPCA '95)*. Association for Computing Machinery, New York, NY, USA, 306–313. https://doi.org/10.1145/224164.224221

Kanat Tangwongsan, Martin Hirzel, Scott Schneider, and Kun-Lung Wu. 2015. General Incremental Sliding-window Aggregation. *Proc. VLDB Endow.* 8, 7 (Feb. 2015), 702–713. https://doi.org/10.14778/2752939.2752940

TAQ 2019. TAQ Database. https://www.nyse.com/. [Online; Accessed September 6, 2019].

Prasanna Thati and Grigore Roşu. 2005. Monitoring Algorithms for Metric Temporal Logic Specifications. *Electronic Notes in Theoretical Computer Science* 113 (2005), 145–162. https://doi.org/10.1016/j.entcs.2004.01.029 Proceedings of the Fourth Workshop on Runtime Verification (RV 2004).

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

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. 2014. Storm @ Twitter. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD '14)*. ACM, New York, NY, USA, 147–156. https://doi.org/10.1145/2588555.2595641

Trill 2020. Trill Documentation: Best Practices for Using Trill in Real-Time Deployments. Available at https://github.com/microsoft/Trill/blob/master/Documentation/BestPractices.pdf. [Online; accessed March 10, 2020].

Pete Tucker, Kristin Tufte, Vassilis Papadimos, and David Maier. 2002. A benchmark for queries over data streams. http://datalab.cs.pdx.edu/niagara/NEXMark/. [Online; Accessed October 15, 2019].

Philip Wadler. 1990. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science* 73, 2 (1990), 231 – 248. https://doi.org/10.1016/0304-3975(90)90147-A

Eugene Wu, Yanlei Diao, and Shariq Rizvi. 2006. High-Performance Complex Event Processing over Streams. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data (SIGMOD '06)*. Association for Computing Machinery, New York, NY, USA, 407–418. https://doi.org/10.1145/1142473.1142520

Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. USENIX, San Jose, CA, 15–28. https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia

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

Fred Zemke, Andrew Witkowski, Mitch Cherniack, and Latha Colby. 2007. *Pattern Matching in Sequences of Rows*. Technical Report. IBM. ANSI Standard Proposal.

W. Zong, T. Heldt, G. B. Moody, and R. G. Mark. 2003. An open-source algorithm to detect onset of arterial blood pressure pulses. In *Computers in Cardiology, 2003*. IEEE, New York, NY, USA, 259–262. https://doi.org/10.1109/CIC.2003.1291140