StreamQRE: Modular Specification and Efficient Evaluation of Quantitative Queries over Streaming Data

Konstantinos Mamouras  Mukund Raghothaman
Rajeev Alur  Zachary G. Ives  Sanjeev Khanna
University of Pennsylvania, USA
{mamouras, rmukund, alur, zives, sanjeev}@cis.upenn.edu

Abstract
Real-time decision making in emerging IoT applications typically relies on computing quantitative summaries of large data streams in an efficient and incremental manner. To simplify the task of programming the desired logic, we propose StreamQRE, which provides natural and high-level constructs for processing streaming data. Our language has a novel integration of linguistic constructs from two distinct programming paradigms: streaming extensions of relational query languages and quantitative extensions of regular expressions. The former allows the programmer to employ relational constructs to partition the input data by keys and to integrate data streams from different sources, while the latter can be used to exploit the logical hierarchy in the input stream for modular specifications.

We first present the core language with a small set of combinators, formal semantics, and a decidable type system. We then show how to express a number of common patterns with illustrative examples. Our compilation algorithm translates the high-level query into a streaming algorithm with precise complexity bounds on per-item processing time and total memory footprint. We also show how to integrate approximation algorithms into our framework. We report on an implementation in Java, and evaluate it with respect to existing high-performance engines for processing streaming data. Our experimental evaluation shows that (1) StreamQRE allows more natural and succinct specification of queries compared to existing frameworks, (2) the throughput of our implementation is higher than comparable systems (for example, two-to-four times greater than RxJava), and (3) the approximation algorithms supported by our implementation can lead to substantial memory savings.

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

Keywords data stream processing, Quantitative Regular Expressions

1. Introduction
The last few years have witnessed an explosion of IoT systems in applications such as smart buildings, wearable devices, and healthcare [9]. A key component of an effective IoT system is the ability to make decisions in real-time in response to data it receives. For instance, a gateway router in a smart home should detect and respond in a timely manner to security threats based on monitored network traffic, and a healthcare system should issue alerts in real-time based on measurements collected from all the devices for all the monitored patients. While the exact logic for making decisions in different applications requires domain-specific insights, it typically relies on computing quantitative summaries of large data streams in an efficient and incremental manner. Programming the desired logic as a deployable implementation is challenging due to the enormous volume of data and hard constraints on available memory and response time.

The motivation for our work is to assist IoT programmers: the proposed language StreamQRE (pronounced Stream-Query) makes the task of specifying the desired decision-making logic simpler by providing natural and high-level declarative constructs for processing streaming data, and the proposed compiler and runtime system facilitates deployment with guarantees on memory footprint and per-item processing time. The StreamQRE language extends quantitative regular expressions—an extension of classical regular expressions for associating numerical values with strings [11], with constructs typical in extensions of relational query languages for handling streaming data [5, 7, 15, 17, 34, 36, 38, 45]. The

* This work was supported by NSF Expeditions award CCF 1138996, NSF awards CCF-1617851 and IIS-1447470, NSF awards ACI-1547360, ACI-1640813, and NIH grant U01-EB020954.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
PLDI ’17, June 18–23, 2017, Barcelona, Spain
ACM. 978-1-4503-4988-8/17/06...$15.00
http://dx.doi.org/10.1145/3062341.3062369

The operators split reduce construct that partitions the input data stream into a set of substreams, one per key, and returns a relation. Finally, StreamQRE supports both sequential and key-based partitioning. Expressing the same query in some cases, while some queries are best expressed by intermingling the two views.

Compilation into streaming processor with guaranteed complexity bounds. The StreamQRE compiler translates a query to a single-pass streaming algorithm. The regular operators associate an unambiguous parse tree with every prefix of the input stream. The typing rules allow the algorithm to maintain only a constant (in the length of the data stream) number of potential parse tree alternatives. The hierarchical sequential iterators can be evaluated naturally and efficiently using a stack. Since exact computation of operations such as the median of a sequence of values and computing sum over sliding windows requires linear space for exact computation, we show (with supporting implementation) how to integrate approximate computation of subexpressions in the query evaluation. Implementing the synchronization semantics of multiple threads of computation created during the evaluation of key-based partitioning also requires care. For a subclass of StreamQRE expressions, we give a theoretical guarantee of $O(1)$ memory footprint and $O(1)$ per-item processing time.

Experimental performance evaluation. We compared our StreamQRE implementation with three open-source popular streaming engines RxJava, Esper, and Flink, and found that the theoretical guarantees of our compiler indeed translate to better performance in practice: the throughput of the StreamQRE engine is 2 to 4 times higher than RxJava, 6 to 75 times higher than Esper, and 10 to 140 times higher than Flink. We also show that the approximation algorithms supported by our implementation can lead to substantial memory savings. Finally, StreamQRE supports both sequential iteration and key-based partitioning as high-level programming constructs leading to alternative expressions of the same query with substantially different performance, thus opening new opportunities for query optimization, even beyond those used in database and stream processing engines.

Organization. The remaining paper is organized as follows. Section 2 informally introduces the key StreamQRE constructs using an illustrative example regarding monitoring patient measurements. The language definition is presented in Section 3. The StreamQRE compiler is described in Section 4. Experimental evaluation of the Java implementation of StreamQRE is given in Section 5. Section 6 describes related work, and the contributions and future work are summarized in Section 7.

2. Overview

As a motivating example, suppose that a patient is being monitored for episodes of a physiological condition such as epilepsy, and the data stream consists of four types of events: (1) An event $E$ marking the beginning of an episode, (2) a time-stamped measurement $M(ts, val)$ by a sensor, (3) an event $E$ marking the end of an episode, (4) and an event $D$ marking the end of a day. Given such an input data
stream, suppose we want to specify a policy \( f \) that outputs every day, the maximum over all episodes during that day, of the average of all measurements during an episode. A suitable abstraction is to impart a hierarchical structure to the stream:

![Hierarchical structure diagram]

The data stream is a sequence of days (illustrated as diamonds), where each day is a sequence of episodes (illustrated as rectangles), and each episode is a sequence of corresponding measurements (shown as circles) between a begin \( B \) marker (shown as an opening bracket) and an end \( E \) marker (shown as a closing bracket). The regular expression \((B \times M^* \times E)^* \times D\) over the event types \( B, M, E \) and \( D \) specifies naturally the desired hierarchical structure. For simplicity, we assume that episodes do not span day markers.

The policy \( f \) thus describes a hierarchical computation that follows the structure of this decomposition of the stream: the summary of each episode (pattern \( B \times M^* \times E \) is an aggregation of the measurements (pattern \( M \)) it contains, and similarly the summary of each day (pattern \( (B \times M^* \times E)^* \times D \)) is an aggregation of the summaries of the episodes it contains. In order for the policy to be fully specified, the hierarchical decomposition (parse tree) of the stream has to be unique. Otherwise, the summary would not be uniquely determined and the policy would be ambiguous. To guarantee uniqueness of parsing at compile time, each policy \( f \) describes a symbolic unambiguous regular expression, called its rate, which allows for at most one way of decomposing the input stream. The qualifier symbolic means that the alphabets (data types) can be of unbounded size, and that unary predicates are used to specify classes of letters (data items) \[18\]. The use of regular rates implies decidability of unambiguity. Even better, there are efficiently checkable typing rules that guarantee unambiguity for all policies \[19\][20][40].

Existing query languages such as CQL \[15\] and CEDR \[18\] indeed use regular expressions, and other complex forms of pattern matching and sliding windows, to select events. However, the selected events are collected as a set without any intrinsic temporal ordering. If the desired summary of an episode is a set-aggregator (such as the average of all measurements), then the relational languages suffice. But suppose the diagnosis depends on the average value of the \( \text{piecewise-linear interpolation} \) of the sampled measurements. Such a computation is easy to specify as a stateful streaming algorithm that iterates over the sequence of data items, but cannot be expressed directly in existing relational query engines (and would instead need to be specified by a user-defined aggregate function that cannot be optimized by the compiler). The list-iteration scheme provided by the combinator fold : \( B \times (B \times A \rightarrow B) \times A^* \rightarrow B \) (where \( B \) is the aggregate type, and \( A \) is the type of the elements of the list), which is very common in functional programming languages, is sufficient for expressing iterative computations.

State can be introduced in relational languages using recursion, but this is neither natural, nor necessary, nor conducive to efficient compilation to a streaming algorithm. Recent work on quantitative regular expressions (QRE) \[11\] shows how to tightly integrate regular expressions with numerical computations, and allows a natural expression of sequential iterators. QREs and other such proposals in runtime monitoring and quantitative formal verification, however, do not provide relational abstractions.

In our example, suppose now the data stream consists of measurements for multiple patients, hence every episode marker and measurement has a patient identifier:

\[
B(pId) \quad M(pId, ts, val) \quad E(pId) \quad D
\]

where \( pId \) is the unique identifier for a patient. Suppose we have written a query \( f \) processing the data stream for a single patient that outputs the desired quantitative summary at the end of each day as discussed earlier. If the daily summary computed by \( f \) exceeds a fixed threshold value, then we say that the patient has had a \textit{critical day}. Suppose we want to output each day the set of patients for whom the two past consecutive days have been critical. Given \( f \), we can first construct the query \( g \) which partitions the input stream by patient identifiers, applies \( f \) to each sub-stream, collects the results of the form \((pId, daySummary)\) in a set, and selects those patients whose daily summary indicates a critical day. The high-level query then can be specified by supplying the input stream both to \( g \) and a version of \( g \) shifted by a day, and intersect the outputs of the two. Such computation can be most naturally specified in existing relational query languages, but not in state-based formalisms. This \textit{key-based partitioning} operation is our analog of the map-reduce operation \[27][28\], but it raises semantic and implementation questions that have to do with the sequential nature of the data stream: reducing must be sequence-aware, and we need a mechanism to indicate when to emit the output.

The proposed query language StreamQRE draws upon features of relational languages for continuous query processing and quantitative extensions of regular expressions and related formalisms. Unlike previous proposals, it allows the arbitrary mixing of relation-based and sequence-based operations.

### 3. The StreamQRE Language

In this section we introduce the formal syntax and semantics of the core StreamQRE language. Figure \[1\] summarizes the formalism, which we will use to program several common stream transformations, such as stream filtering, projection, and windowing. Several detailed examples that illustrate the use of StreamQRE can be found in \[12\].

**Basic types & operations.** Since our query language combines regular expressions with quantitative operations, we first choose a typed signature which specifies the basic data
types and operations for manipulating them. We fix a collection of basic types, and we write $A, B, \ldots$ to range over them. This collection contains the type $\text{Bool}$ of boolean values, and the unit type $\text{Unit}$ whose unique inhabitant is denoted by $\text{def}$. It is also closed under the cartesian product operation $\times$ for forming pairs of values. Typical examples of basic types are the natural numbers $\mathbb{N}$, the integers $\mathbb{Z}$, and the real numbers $\mathbb{R}$. We also fix a collection of basic operations on the basic types, for example the $k$-ary operation $\text{op} : A_1 \times \cdots \times A_k \to A$. The identity function on $D$ is written as $\text{id}_D : D \to D$, and the operations $\pi_1 : A \times B \to A$ and $\pi_2 : A \times B \to B$ are the left and right projection respectively. We assume that the collection of operations contains all identities and projections, and is closed under pairing and function composition. We write $!_A : A \to \text{Unit}$ for the unique function from $A$ to $\text{Unit}$.

**Predicates.** For every basic type $D$, we fix a collection of atomic predicates, so that the satisfiability of their Boolean combinations is decidable. We write $\varphi : D \to \text{Bool}$ to indicate that $\varphi$ is a predicate on $D$, and we denote by $\text{true}_D : D \to \text{Bool}$ the always-true predicate. The requirement for decidability of satisfiability for predicates is necessary for the query typing rules that we will present later. The satisfiability checks can be delegated to an SMT solver [26].

**Example 1.** For the example stream of monitored patients described in [2], the data type $\text{DP}$ is the tagged (disjoint) union:

$$\text{DP} = \{ \text{E}(p) \mid p \in \text{PID} \} \cup \{ \text{M}(p, t, v) \mid p \in \text{PID}, t \in T, v \in V \},$$

where $\text{PID}$ is the set of patient identifiers, $T$ is the set of timestamps, and $V$ is the set of scalars for the measurements.

**Relational types and operations.** We consider relation types of the form $\text{Rel}(A)$, where $A$ is a basic type. The elements of $\text{Rel}(A)$ are multisets over $A$. To express the well-known operations select, project and join from relational algebra [6], it suffices to consider the operation of cartesian product and the higher-order functions $\text{map}$ and $\text{filter}$.

$$\text{prod}_{A,B} : \text{Rel}(A) \times \text{Rel}(B) \to \text{Rel}(A \times B)$$

$$\text{map}(\text{op}) : \text{Rel}(A) \to \text{Rel}(B), \text{where } \text{op} : A \to B$$

$$\text{filter}(\varphi) : \text{Rel}(A) \to \text{Rel}(A), \text{where } \varphi : A \to \text{Bool}$$

We also consider standard set-aggregators of relational query languages (e.g., sum, count, minimum, maximum, average), which we model as operations of type $\text{Rel}(A) \to A$. We write $\text{Map}(K, V)$ for the subtype of $\text{Rel}(K \times V)$ which consists of (partial) maps from $K$ to $V$.

**Symbolic Regexes.** For a type $D$, we define the symbolic regular expressions over $D$ [46], denoted $\text{RE}(D)$, as

$$r ::= \varphi | r \sqcup r | r \cdot r | r^*,$$

where $\varphi$ ranges over predicates on $D$. The concatenation symbol $\cdot$ is sometimes omitted, i.e. we write $rs$ instead of $r \cdot s$. The expression $r^*$ (iteration at least once) abbreviates $r \cdot r^*$. We interpret an expression $r : \text{RE}(D)$ as a set $[r] \subseteq D^*$ of strings over $D$. We put $[\varnothing] = \{ d \in D \mid \varphi(d) \text{ is true} \}$, and this extends to all expressions in the usual way.

The notion of unambiguity for regular expressions [19] is a way of formalizing the requirement of uniqueness of parsing. The languages $L_1$ and $L_2$ are said to be unambiguously concatenatable if for every word $w \in L_1 \cdot L_2$ there are unique $w_1 \in L_1$ and $w_2 \in L_2$ with $w = w_1 \cdot w_2$. The language $L$ is said to be unambiguously iterable if for every word $w \in L^*$ there is a unique integer $n \geq 0$ and unique $w_i \in L$ with $w = w_1 \cdot \ldots \cdot w_n$. The definitions of unambiguous concatenatability and unambiguous iterability extend to regular expressions in the obvious way. Now, a regular expression is said to be unambiguous if it satisfies the following: (1) $r, s$ are disjoint for every subexpression $r \sqcup s$, (2) $r, s$ are unambiguously concatenatable for every subexpression $rs$, and (3) $r$ is unambiguously iterable for every subexpression $r^*$.

**Observation 2.** Suppose $r$ and $s$ are unambiguous symbolic regexes over $D$. Let $\Sigma$ be the set of truth assignments for the base predicates of $r$ and $s$. Assuming that satisfiability of the predicates can be decided in unit time, the problems

1. Are $r$ and $s$ disjoint?
2. Are $r$ and $s$ unambiguously concatenatable?
3. Is $r$ unambiguously iterable?
4. Is $r$ contained in $s$?

are decidable in polynomial time in $|r|, |s|, |\Sigma|$ [19] [26] [40]. In particular, checking whether a regular expression is unambiguous can be done in polynomial time.

**Remark 3 (Witnesses of Ambiguity).** Consider the computation: "summarize a patient episode with at least one high-risk measurement". This computation is analogous to the regex-matching problem: "identify sequences over $a$ and $b$ with at least one occurrence of $b$". The regex $(a \sqcup \epsilon)b(a \sqcup \epsilon)^*$ is very natural to write, but is ambiguous. The procedure for checking unambiguity (see Observation 2) can flag it as ambiguous, and it can also give a minimum-length trace witnessing the ambiguity: the sequence $bb$ can be parsed in two ways, either as $\epsilon \cdot b \cdot b$ or as $b \cdot b \cdot \epsilon$. Generally, this procedure can pinpoint both the ambiguous subexpression and the smallest examples that prove ambiguity. For the given example, the expression is equivalent to the unambiguous $a^*b(a + b)^*$.

**Streaming functions.** The basic semantic object in our calculus is a partial map $f$ of type $D^* \to C$ (we use the arrow $\to$ to indicate partiality), where $D$ is the type of the input elements and $C$ the type of the outputs. We call these objects streaming functions. They describe general transformations of unbounded streams by specifying the output $f(w)$ (if any) on the stream $w$ seen so far. For example, suppose we want to describe a filtering transformation on streams of scalars of type $V$, where only the nonnegative scalars are retained and the negative ones are filtered out. The streaming function $f : V^* \to V$ that describes this transformation is defined on the sequences $\{ v_1 v_2 \ldots v_n \mid n \geq 1 \text{ and } v_n \geq 0 \}$ and the value is the last scalar of the sequence, i.e. the current item.
### Streaming Queries

We now introduce formally the language of **Streaming Quantitative Regular Expressions (QREs)** for representing stream transformations. For brevity, we also call these expressions **queries**. A denotational semantics will be given in terms of streaming functions. The denotations satisfy the additional property that their domains are regular sets over the input data type. The rate of a query $f$, written $R(f)$, is a symbolic regular expression that denotes the domain of the interpretation of $f$, that is, the set of stream prefixes for which $f$ is defined: $R(f) = \text{dom}(\llbracket f \rrbracket)$. We say that a sequence $w$ of data items matches the query $f$ if it belongs to $\text{dom}(\llbracket f \rrbracket)$. The definition of the query language has to be given simultaneously with the definition of rates (by mutual induction), since the query constructs have typing restrictions that involve the rates. We annotate a query $f$ with a type $\text{QRE}(D, C)$ to denote that the input stream has elements of type $D$ and the outputs are of type $C$. The full formal definition of the syntax and semantics of streaming queries is given in Figure 1. The decidability of type checking is established in Observation 2.

#### Example 4 (Rate of a Query)

In the patient monitoring example, the statistical summary of a patient’s measurements should be output at the end of each day, and thus, depends only on the types of events in a regular manner. The rate in this case is the regular expression $((B \cdot M^* \cdot E)^* \cdot D)^*$. 

#### Atomic queries

The basic building blocks of queries are expressions that describe the processing of a single data item. Suppose $\varphi : D \to \text{Bool}$ is a predicate over the data item type $D$ and $\text{op} : D \to C$ is an operation from $D$ to the output type $C$. Then, the atomic query $\text{atom}(\varphi, \text{op}) : \text{QRE}(D, C)$, with rate $\varphi$, is defined on single-item streams that satisfy the predicate $\varphi$. The output is the value of $\text{op}$ on the input element. It is common for $\text{op}$ to be the identity function, and $\varphi$ to be the always-true predicate. So, we abbreviate the query $\text{atom}(\varphi, \text{id}_D)$ by $\text{atom}(\varphi)$, and $\text{atom}(\text{true}_D)$ by $\text{atom}()$.

#### Iteration

Suppose the query $f : \text{QRE}(D, A)$ describes a computation that we want to iterate over consecutive non-overlapping subsequences of the input stream, in order to aggregate the produced values (of type $A$) sequentially using an aggregator $\text{op} : B \times A \to B$. More specifically, we split the input stream $w$ into subsequences $w = w_1w_2\ldots w_n$, where each $w_i$ matches $f$. We apply the computation $f$ to each of the $w_i$, thus producing the output values $a_1a_2\ldots a_n$ with $a_i = \llbracket f \rrbracket w_i$. Finally, we combine these results using the list iterator $\text{fold}$ with start value $\text{init} \in B$ and aggregation operation $\text{op} : B \times A \to B$ by foldng the list of values $a_1a_2\ldots a_n$. This can be formalized with the combinator $\text{fold} : B \times (B \times A) \to B^*$, which takes an initial value $b \in B$ and a stepping map $\text{op} : B \times A \to B$, and iterates through a sequence of values of type $A$:

$$\text{fold}(b, w) = b$$
$$\text{fold}(b, \text{op}, w) = \text{op}(\text{fold}(b, \text{op}, w), a)$$

for all sequences $\gamma \in A^*$ and all values $a \in A$. For example, $\text{fold}(b, \text{op}, a_1a_2a_3) = \text{op}(\text{op}(b, a_1), a_2), a_3)$.

The query $h = \text{iter}(f, \text{init}, \text{step}) : \text{QRE}(D, B)$ describes the computation of the previous paragraph. In order for $h$ to be well-defined as a function, every input stream $w$ that matches $h$ must be uniquely decomposable into $w = w_1w_2\ldots w_n$. 

---

**Figure 1.** Streaming Quantitative Regular Expressions (QREs): Syntax and denotational semantics with streaming functions.
with each \( w_i \) matching \( f \). This requirement can be expressed equivalently as: the rate \( R(f) \) is unambiguously iterable.

These sequential iterators can be nested imparting a hierarchical structure to the input data stream facilitating modular programming. In the single-patient monitoring stream, for example, we can associate an iterator with the episode nodes to summarize the sequence of measurements in an episode, and another iterator with the day nodes to summarize the sequence of episodes during a day.

**Combination and application.** Assume the queries \( f \) and \( g \) describe stream transformations with outputs of type \( A \) and \( B \) respectively that process the same set of input sequences, and \( op \) is an operation of type \( A \times B \rightarrow C \). The query \( \text{combine}(f, g, op) \) describes the computation where the input is processed according to both \( f \) and \( g \) in parallel and their results are combined using \( op \). Of course, this computation is meaningful only when both \( f \) and \( g \) are defined on the input sequence. So, we demand w.l.o.g. that the rates of \( f \) and \( g \) are equivalent. This binary combination construct generalizes to an arbitrary number of queries. For example, we write \( \text{combine}(f, g, h, op) \) for the ternary variant. In particular, we write \( \text{apply}(f, op) \) for the case of one argument.

**Quantitative concatenation.** Suppose that we want to perform two streaming computations in sequence on consecutive non-overlapping parts of the input stream: first execute the query \( f : \text{QRE}(D, A) \), then execute the query \( g : \text{QRE}(D, B) \), and finally combine these two intermediate results using the operation \( op : A \times B \rightarrow C \). More specifically, we split the input stream into two parts \( w = w_1w_2 \), process the first part \( w_1 \) according to \( f \) with output \( f[w_1] \), process the second part \( w_2 \) according to \( g \) with output \( g[w_2] \), and produce the final result \( op(f[w_1], g[w_2]) \) by applying \( op \) to the intermediate results. The query \( \text{split}(f, g, op) : \text{QRE}(D, C) \) describes this computation. In order for this construction to be well-defined as a function, every input \( w \) that matches \( \text{split}(f, g, op) \) must be uniquely decomposable into \( w = w_1w_2 \) with \( w_1 \) matching \( f \) and \( w_2 \) matching \( g \). In other words, the rates of \( f \) and \( g \) must be unambiguously concatenated. The binary \( \text{split} \) construct extends naturally to more than two arguments. For example, the ternary version would be \( \text{split}(f, g, h, op) \).

**Global choice.** Given queries \( f \) and \( g \) of the same type with disjoint rates \( r \) and \( s \), the query \( \text{or}(f, g) \) applies either \( f \) or \( g \) to the input stream depending on which one is defined. The rate of \( \text{or}(f, g) \) is the union \( r \uplus s \). This \( \text{choice} \) construction allows a case analysis based on a global regular property of the input stream. In our patient example, suppose we want to compute a statistic across days, where the contribution of each day is computed differently depending on whether or not a specific physiological event occurs sometime during the day. Then, we can write a query summarizing the daily activity with a rate capturing good days (the ones without any significant event) and a different query with a rate capturing bad days, and iterate over their disjoint union.

---

**Key-based partitioning.** The input data stream for our running example contains measurements from different patients, and suppose we have written a query \( f \) that computes a summary of data items corresponding to a single patient. Then, to compute an aggregate across patients, the most natural way is to partition the input stream by a key, the patient identifier in this case, supply the corresponding projected sub-streams to copies of \( f \), one per key, and collect the set of resulting values. In order to synchronize the per-key computations, we specify a predicate \( \varphi_S : D \rightarrow \text{Bool} \) which defines the synchronization elements. The rest of the elements, which satisfy the negation \( \neg \varphi_S \), are the keyed elements. We typically write \( K \) for the set of keys, and \( \kappa : D \rightarrow K \) for the function that projects the key from an item (the value of \( \kappa \) on synchronization items is irrelevant). For the patient input data type of Example 1, we choose: \( \varphi_S = D \) (i.e., \( \varphi_S(x) \) is true when \( x \) is a day marker) and \( K = \text{PID} \). The partitioning ensures that the synchronization elements are preserved so that the outputs of different copies of \( f \) are synchronized correctly (for example, if each \( f \) outputs a patient summary at the end of the day, then each sub-stream needs to contain all the end-of-day markers). Note that the output of such a composite streaming function is a mapping \( T : \text{Map}(K, C) \) from keys to values, where \( C \) is the output type of \( f \) and \( T(k) \) is the output of the computation of \( f \) for key \( k \). This key-based partitioning operation is our analog of the map-reduce operation [27, 28].

The details of this construction are subtle, and we have to introduce more notation to present the semantics precisely. We describe the partitioning of the input stream using terminology from concurrent programming. For every key \( k \), imagine that there is a thread that receives and processes the sub-stream with the data items that concern \( k \). This includes all synchronization items, and those keyed data items \( x \) for which \( \kappa(x) = k \). So, an item satisfying \( \neg \varphi_S \) is sent to only one thread (as prescribed by the key), but an item satisfying \( \varphi_S \) is sent to all threads. See Figure 2 for an illustration of the partitioning into sub-streams. For a sequence \( w \) of data items, we write \( \text{Keys}(w) \) to denote the set of keys that appear in \( w \), and \( w|_k \) for the subsequence of \( w \) that corresponds to the key \( k \). Each thread computes independently, and the synchronization elements are used for collecting the results of the threads. We specify a symbolic regular expression \( r : \text{RE}(D) \), which enforces a rate of output for the overall computation. For example, if \( r = (\neg D)^* \cdot (D)^2 \) then we intend to have output every other day. The rate should only specify sequences that end in a synchronization item.
Suppose \( f : \text{QRE}(D, C) \) is a query that describes the per-key (i.e., per-thread) computation, and \( r \) is the overall output rate that we want to enforce. Then, the query

\[
\text{map-collect}(\varphi_S, \kappa, f, r) : \text{QRE}(D, \text{Map}\{K, C\})
\]
describes the simultaneous computation for all keys, where the overall output is given whenever the stream matches \( r \). The overall output is the map obtained by collecting the outputs of all threads that match. W.l.o.g. we assume that the rate of \( f \) is contained in \( r \), and that it only contains streams with at least one occurrence of a keyed data item. This can be expressed as the regex inclusion \( \mathcal{R}(f) \subseteq r \setminus \varphi_S^* \), which is equivalent to \( \varphi_S^* \cdot (\neg \varphi_S) \cdot \text{true}_D^* \subseteq \mathcal{R}(f) \subseteq r \). The rate \( r \) should only depend on the occurrence of synchronization elements, a requirement that can be expressed syntactically:

\[
r = s[(\neg \varphi_S)^* \varphi_S / \varphi_S]
\]

for some expression \( s : \mathcal{R}(D) \), where the only predicate that \( s \) is allowed to contain is \( \varphi_S \). We write \( s[\psi/\varphi] \) to denote the result of replacing every occurrence of \( \varphi \) in \( s \) with \( \psi \). When \( r \) satisfies this condition we say that it is well-formed w.r.t. \( \varphi_S \). For example, if \( s = D \) (indicating output at every day marker) and \( \varphi_S = D \), then \( r = s[(\neg \varphi_S)^* \varphi_S / \varphi_S] = ((\neg D)^* D)^* \) is well-formed w.r.t. \( \varphi_S \). All these restrictions that we have imposed on the rates do not affect expressiveness, but are useful for efficient evaluation and type-checking.

Remark 5. The operation of key-based partitioning raises significant challenges for the efficiency of type-checking. In the context of pure regex matching, one could consider either an existential or a universal notion of key-based partitioning:

\[
\text{partition}_\exists(\kappa, r) = \text{RE}(D), \quad \text{partition}_\forall(\kappa, r) : \text{RE}(D)
\]

The expression \( \text{partition}_\exists(\kappa, r) \) accepts nonempty words \( w \) with \( |w|_k \in \{1\} \), where \( k \) is the key of the last element of \( w \). Similarly, the expression \( \text{partition}_\forall(\kappa, r) \) accepts nonempty words \( w \) with \( |w|_k \in \{1\} \) for every key \( k \) that appears in \( w \). Notice that \( \text{partition}_\forall(\kappa, r) \) is the standard construct of disjoint partitioning used in streaming systems (see, for example, the “partition-by” clause in the languages of [49] and [33]). Unfortunately, these natural notions of partitioning and matching give rise to decision problems (such as nonemptiness and unambiguity) that are not easy to compute. The introduction of synchronization elements for the map-collect construct of StreamQRE is meant to solve this problem by enforcing a rate of output that depends only on the occurrence of synchronization elements in a regular way. This simplifies the description of the domain of a map-collect query, and the typing checks become easy.

Streaming composition. A natural operation for query languages over streaming data is streaming composition: given two streaming queries \( f \) and \( g \), \( f \gg g \) represents the computation in which the stream of outputs produced by \( f \) is supplied as the input stream to \( g \). Such a composition is useful in setting up the query as a pipeline of several stages. We allow the operation \( \gg \) to appear only at the top-level of a query. So, a general query is a pipeline of \( \gg \)-free queries. At the top level, no type checking needs to be done for the rates, so we do not define the function \( \mathcal{R} \) for queries \( f \gg g \). With streaming composition we can express non-regular patterns, such as a sequence of increasing numbers.

The semantics of streaming composition involves a lifting operator on streaming functions. For a function \( f : D^* \rightarrow C \) we define the partial lifting \( \text{lift}(f) : D^* \rightarrow C^* \), which generates sequences of output values. The domain of \( \text{lift}(f) \) is equal to the domain of \( f \), but \( \text{lift}(f) \) returns the sequence of all outputs that have been emitted so far.

3.1 Derived Stream Transformations

The core language of Figure 1 is expressive enough to describe many common stream transformations. We present below several derived patterns, including filtering, mapping, and aggregation over windows.

Iteration (at least once). Let \( f : \text{QRE}(D, A) \) be a query with output type \( A \), \( \text{init} : B \) be the initialization value, and \( \text{op} : B \times A \rightarrow B \) be the aggregation function. The query \( \text{iter}_1(f, \text{init}, \text{op}) \), with output type \( B \), splits the input stream \( w \) unambiguously into consecutive parts \( w_1 w_2 \ldots w_n \) each of which matches \( f \), applies \( f \) to each \( w_i \) producing a sequence of output values \( a_1 a_2 \ldots a_n \), i.e. \( a_i = f(w_i) \), and combines the results \( a_1 a_2 \ldots a_n \) using the list iterator \( \text{left fold} \) with start value \( \text{init} \) and accumulation operation \( \text{op} \). The construct \( \text{iter}_1 \) can be encoded using \( \text{iter} \) as follows:

\[
\text{iter}_1(f, \text{init}, \text{op}) = \text{split}(\text{iter}(f, \text{init}, \text{op}), f, \text{op})
\]

The type of the query is \( \text{QRE}(D, B) \) and its rate is \( \mathcal{R}(f)^+ \).

Iteration exactly \( n \) times. Let \( n \geq 0 \) and \( f : \text{QRE}(D, A) \) be a query to iterate exactly \( n \) times. The aggregation is specified by the initialization value \( \text{init} \) : \( B \) and the aggregation function \( \text{op} : B \times A \rightarrow B \). The construct \( \text{iter}^n \) describes iteration (and aggregation) exactly \( n \) times, and can be encoded as follows: \( \text{iter}^n(f, \text{init}, \text{op}) = \text{init} \) and

\[
\text{iter}^{n+1}(f, \text{init}, \text{op}) = \text{split}(\text{iter}^n(f, \text{init}, \text{op}), f, \text{op})
\]

The type of \( \text{iter}^n(f, \text{init}, \text{op}) \) is \( \text{QRE}(D, B) \) and its rate it \( \mathcal{R}(f)^n \) (\( n \)-fold concatenation).

Matching without output. Suppose \( r \) is an unambiguous symbolic regex over the data item type \( D \). We want to write a query whose rate is equal to \( r \), but which does not produce any output. This is essentially the same as returning \( \text{def} \). So, we define the regex to query translation function \( \text{match} \):

\[
\text{match}(\varphi) = \text{atom}(\varphi, !D)
\]

\[
\text{match}(r_1 \sqcup r_2) = \text{or}(\text{match}(r_1), \text{match}(r_2))
\]

\[
\text{match}(r_1 \cdot r_2) = \text{split}(\text{match}(r_1), \text{match}(r_2), !\text{left} \times \text{right})
\]

\[
\text{match}(r^*) = \text{iter}(\text{match}(r), \text{def}, !\text{left} \times \text{right})
\]

An easy induction establishes that \( \mathcal{R}(\text{match}(r)) = r \).

Stream filtering. Let \( \varphi \) be a predicate over the type of input data items \( D \). We want to describe the streaming
We can support this primitive, which can be compiled into a query that counts the number of occurrences of a pattern. Let $\text{window}(D)$ be the stream of tuples $(x, c)$ such that $x$ is the occurrence of the pattern at time $c$. Then, we can implement this with the query

$$\text{window}(D) \rightarrow \text{count}(\text{window}(D), \text{atom}(\varphi, \pi_2))$$

The type of $\text{window}(D)$ is $\text{QRE}(D, D)$ and its rate is $\text{true}^\bullet_D \cdot \varphi$.

**Stream mapping.** The mapping of an input stream of type $D$ to an output stream of type $C$ according to the operation $op : D \rightarrow C$ is given by the following query:

$$\text{map}(op) \triangleq \text{split}(\text{match}(\text{true}^\bullet_C), \text{atom}(\text{true}_D, \text{op}), \pi_2)$$

The type of the query is $\text{QRE}(D, C)$ and its rate is $\text{true}^\bullet_D$.

**Pattern-based tumbling windows.** The term tumbling windows is used to describe the splitting of the stream into contiguous non-overlapping regions. Suppose we want to express both the window and the sliding using a pattern. Let $\text{pattern}$ be the query that processes a stream of type $\langle D, C \rangle$ and its rate is $\text{R}(\text{pattern})^\bullet$. The goal of the compilation procedure is to generate an algorithm that computes the intended function. Before presenting the compilation procedure, we should describe more precisely the streaming model of computation. The input stream is a potentially unbounded sequence of elements, and a streaming algorithm computes by consuming the elements of the stream in order, as they become available. The input stream is read incrementally in one pass, which means that past input elements cannot be accessed again. A streaming algorithm $A$ consists of: (1) a possibly infinite state space $S$, which is the set of all possible internal configurations of the algorithm (i.e., its memory), (2) an initial state $\text{init} \in S$, and (3) a transition function $\text{next} : S \times D \rightarrow S \times C$, which consumes the next input element, mutates the current state, and produces an output.

We say that the streaming algorithm $A$ implements the query $f : \text{QRE}(D, C)$ if for every input $w \in D^\bullet$, the output of $A$ after consuming $w$ is equal to $\text{split}(f(w))$ (when defined), and equal to nil when $f(w)$ is undefined. The space and time requirements of a streaming algorithm are typically given as functions of the size $n$ of the stream consumed so far, and possibly also in terms of other significant parameters of the input (e.g., range of numerical values).

The goal of the compilation procedure is to generate an algorithm that uses a small amount of space and processes fast each newly arriving element. Ideally, both the space and time requirements are independent of the length of the data stream. Conceptually, the computation of $\text{split}(f(w))$ amounts to evaluating an expression tree that is of size linear in the length of $w$. This expression tree mirrors the parse tree that corresponds to the rate of $f$, so one crucial challenge is the online computation of the split points in the stream to match the pattern. Suppose we process the query $\text{split}(f, g, op)$ on $w_1w_2w_3w_4$, where the prefixes $w_1$ and $w_2w_3$ match $f$ and the subsequences $w_2w_3$ and $w_3w_4$ match $g$. After having seen the
prefix $w_1 w_2$, we need to maintain the parallel computation of two expression trees, since $f$ has matched twice. The insight is that the number of these parallel computations is bounded and depends only on the size of the query (see also \cite{10} and \cite{11}). The computation of $[[f]] w$ involves, however, more than just parsing: to compute efficiently the intermediate expression trees, they must have a compact representation. Indeed, the intermediate results can be represented by stacks of values, whose size is bounded by the size of the query. This is a crucial part of establishing the efficiency of the proposed evaluation algorithm.

**Generalized evaluation problem.** We proceed now with some technical details of our approach. In order to define the compilation procedure by recursion on the structure of the query $f : \text{QRE}(D, C)$, we have to generalize the problem. Let Stack be the type of finite stacks that can contain any value of any basic type, and $\Box$ be the empty stack. We assume that Stack is equipped with the following standard operations:

\begin{align*}
\text{push}_A &: Stack \times A \rightarrow Stack \\
\text{pop} &: Stack \rightarrow Stack
\end{align*}

and the operation $\text{getTop}_A : Stack \rightarrow A \cup \{\text{nil}\}$, which maps a stack to the value on the top of it (or nil if the stack is empty). Instead of just input streams $d_1 d_2 d_3 \ldots$ of data items, we consider streams that also contain non-consecutive occurrences of stacks. For example, $s_1 d_1 d_2 s_2 d_4 d_5 s_3 d_6 d_7 d_8 \ldots$ is such a stream. Informally, we think that every stack item marks the beginning of a thread of execution that processes all the data items that follow. Our example stream involves the parallel threads shown on the right. We want an algorithm that simulates the parallel computation of $f$ on the threads. The output on a thread $s w$ must be defined when $[[f]] w$ is defined, and it has to be equal to the stack $\text{push}_A([[f]] w)$. In order for the overall output to be well-defined, we require that at most one active thread gives output. We say that a sequence $w \in (D \cup \text{Stack})^*$ is well-formed for $f$ if for every prefix $w$ there is at most one thread for which $[[f]]$ is defined. So, we describe for every query $f$ an algorithm that solves this generalized problem on well-formed inputs from $(D \cup \text{Stack})^*$.

There are four components to specify: the state space State, the initial state init, the function start that consumes stack items, and the function next that consumes data items (elements of $D$). Figures 3 and 4 show this construction (by recursion on $f$) for the base queries and the constructs iteration, concatenation, and map-collect. We write $A_f$ to denote the streaming generalized evaluation algorithm for $f$. The other constructs are easier to handle, and hence omitted.

**Split.** To see why this generalization of the problem is necessary, let us consider the evaluation of $h = \text{split}(g, \sigma, op)$ on a single thread $\Box d_1 d_2 d_3 d_4 d_5 d_6 d_7 d_8$. The algorithm $A_h$, described in Figure 3, forwards every data item to both $A_f$ and $A_g$. Suppose that there are two prefixes of the input that match $f$: $d_1 d_2 d_3$ and $d_1 d_2 d_3 d_4 d_5$, for which $A_f$ and $A_g$ outputs $a_1$ and $a_2$ respectively. Recall that all queries are unambiguous, therefore it is not possible for both suffixes $d_4 d_5 d_6 d_7 d_8$ and $d_6 d_7 d_8$ to match $g$. We use the stacks of values as a mechanism to propagate the intermediate results of the computations. For our particular example stream, the evaluator $A_h$ ensures that the sequence of items fed to $A_g$ is $d_1 d_2 d_3 [a_1] d_4 d_5 [d_2] d_6 d_7 d_8$. This means that $A_g$ has to

---

**Figure 3.** Compilation for atomic queries, quantitative iteration, and quantitative concatenation.
simulate the concurrent evaluation of multiple suffixes of the data stream. After reading the stream, suppose \( A_k \) accepts the suffix \( s.d \) and reports the value \( b_2 = [g]_{d_0} \). The output of \( A_k \) is then \([b_2, a_2] \), that is, the result of pushing the value \( b_2 \) on the stack \([a_2] \) that started the accepting thread. Finally, \( A_h \) uses the output stack \([b_2, a_2] \) to compute the answer \( op(a_2, b_2) \). This stack-based mechanism of propagating values is crucial for matching the outputs of \( f \) and \( g \) and computing the final value.

**Atomic.** It is instructive to see how the query \( \text{atom}(\varphi, op) \) is implemented in order to satisfy the contract of the generalized evaluation problem. The crucial observation is that there should only be output on consecutive occurrences of the form \( s.d \), where \( s \) is a stack and \( d \) is an item that satisfies the predicate \( \varphi \). To achieve this, we have to record the stack \( s \) given with a \( \text{start}(s) \) invocation, and with the following \( \text{next}(d) \) invocation we return the stack \( s.push(op(d)) \) as output. As seen in Figure 3, the state space of the algorithm consists of elements \( L(in) \), where \( in \) is the stack that initiated the current thread of execution. The initial state is \( L(nil) \), since in the beginning there is no active thread. The function \( \text{start} \) sets the stack for the active thread, and returns \( nil \) because the empty sequence \( \varepsilon \) does not match the query \( \text{atom}(\varphi, op) \). The transition function \( \text{next} \) checks if there is an active thread and if the current item \( d \) satisfies the formula \( \varphi \). If so, it returns the active stack after pushing the result on it.

**Iteration.** Consider now the generalized evaluation algorithm for \( \text{iterate}(f, b_0, \sigma) \), shown in Figure 3. The state space consists of elements \( L(child) \), where \( child \) is a state of \( A_t \). The initial state is \( L(init_2) \), where \( init_2 \) is the initial state of \( A_t \). The idea of the algorithm is to use the stack to store the intermediate results of the aggregation. When \( \text{start}_h \) is invoked with the stack \( s \), we propagate the start call to the algorithm for \( f \) with the stack \( s.push(b_0) \), since the initial value \( b_0 \) is the first intermediate result. Now, the transition on a data item \( d \) involves first sending the item to \( A_t \). The interesting case is when \( A_t \) gives output. The stack \( s \) that it returns contains the value \( a \) given by \( f \) as well as the previous intermediate result \( b \) that we pushed on the stack. From these we can compute the next intermediate result \( \sigma(b, a) \), which we place again on the stack in order to restart \( A_t \).

**Map-Collect.** Suppose that the predicate \( \varphi : D \rightarrow \text{Bool} \) specifies the synchronization elements, \( K \) is a finite set of keys, \( \kappa : D \rightarrow K \) extracts a key from an element, \( f \) describes the per-key processing, and \( r : \text{RE}(D) \) is the rate of output that we want to enforce. In Figure 4, we describe the compilation of \( h = \text{map-collect}(\varphi, \kappa, f, r) \). Each state of \( A_h \) is a pair: the first component is a key-indexed map of states of \( A_t \), and the second component is a state of \( A_t \). The implementation of the function \( \text{next} \) is noteworthy. If the data item is a synchronization element, then it is propagated to all key-threads. If the data item specifies a key, then it is propagated only to the key-thread that it is meant for. The rate determines when to collect the outputs of the key-threads. When the rate accepts, then we collect in a relation the outputs of all key-threads that have available output. A crucial optimization that we have implemented is the collapsing of key-threads that have only seen synchronization elements so far, hence these threads have identical state.

**Theorem 7.** Let \( f \) be a query which involves only finite sets of keys for the occurrences of map-collect in it. Then, the streaming algorithm \( A_t \) that implements \( f \) (as described in Figure 3) satisfies the following: 1. Assume that the values of the basic types appearing in \( f \) require unit space to be stored. Then, \( A_t \) requires space that depends only on the size of \( f \), independent of the length of the stream. 2. Suppose additionally that the basic operations that appear in \( f \) require unit time to be performed. Then, the processing time per element for \( A_t \) depends only on the size of \( f \), and is independent of the length of the stream.

**Remark 8 (Map-Collect).** When the map-collect construct is used at the top level of a query, it is easy to implement key-based partitioning so that the space requirements are \( |K| \cdot (\text{space needed for } f) \), where \( K \) is the set of keys appearing in the stream. The situation is much more complicated when map-collect is nested beneath several regular operators, because then the evaluation algorithm has to explore the possibility of matching it against several (possibly overlapping) subsequences of the stream. Putting a constant bound (on the length of the input stream) on the several concurrent possibilities of matching a nested map-collect hinges on the carefully chosen typing restrictions of Figure 4.
we can overcome this barrier. It is known that an exact
Theorem 7 states space and time guarantees for our evaluation
values. This idea applies to all real numbers (negative, zero,
H holds. The implies, for example, that if
ε ∈ (0, 1), we put apx(x) = (1 + ε)−i−1, where i = ⌊log_{1+ε}(1/x)⌋. For
all x ∈ (0, ∞), the inequality apx(x) ≤ x ≤ apx(x)(1 + ε)
holds. The implies, for example, that if ε = 0.01 then the
approximation error is at most 1% of the actual value.

Our streaming algorithm for computing median queries
maintains a summary H, which is an integer-indexed array
of counts (for discretized values). For an integer i, the entry
H[i] is the count of values in the interval [(1+ε)i, (1+ε)i+1)
that we have seen so far. From this summary, any rank query
can be answered approximately with relative error ε. Suppose
that U is the largest value and L is the smallest (positive)
value appearing in the stream. For asymptotic analysis, we
assume w.l.o.g. that U ≫ 1 and L ≪ 1. Then, the array H
requires \log_{1+ε} U + \log_{1+ε} L^{-1} ≤ ε^{-1} \cdot \log_2(U/L) entries.

We write M_ε(V) for the type of ε-approximate histograms
over scalars V, and 0, for the empty histogram. We support
the operations \text{ins}_ε : M_ε(V) \times V \rightarrow M_ε(V) for value insertion
and \text{mdn}_ε : M_ε(V) \rightarrow V for obtaining the approximate
median. Choosing between exact or approximate computation
amounts to simply using the appropriate data structure. The
approximation guarantees are given concisely as follows:

Theorem 9. For every nonempty sequence w of scalars, we
have that \|f_{ε-axp}\| w ≤ \|f_{ε-axp}\| w ≤ (1+ε) \cdot \|f_{ε-axp}\| w, where
f_{ε-axp} = \text{apply}(\text{iter}(\text{atom}(), 0, \text{ins}), \text{mdn})
and \text{mdn}_ε = \text{apply}(\text{iter}(\text{atom}(), 0, \text{ins}_ε), \text{mdn}_ε)
Assuming that constant space is enough for storing the size
of the stream, the space used by the evaluation algorithm for
f_{ε-axp} is linear in ε⁻¹ and in log(U/L), where U (L) is the
largest (smallest) absolute value appearing in the stream.

When the computation of medians is mixed with other
numerical operations, providing a global approximation guar-
anteest is more subtle. If we restrict attention to non-negative
numbers and the numerical operations +, \min, \max, \text{avg}
and \text{mdn}_ε, then the guarantees of Theorem 9 can be lifted
to all queries involving these operations. However, adding
an ε-approximately computed positive number and an ε-
approximately computed negative number can result in un-
bounded relative error for the result.

Another useful computation that benefits tremendously
from approximation is sliding-window event counting (de-
scribed in \S 3.1). To count the number of events exactly, it
can be shown that space linear in the size of the window is
necessary. The work of [25] shows that a (1+ε)-approximate
estimate can be obtained using space that is linear in ε⁻¹ and
logarithmic in the size of the window. We have incorporated
the approach of [25] in our implementation.

5. Experiments
We have implemented StreamQRE as a Java library in order
to facilitate the easy integration with user-defined types
and operations. Our implementation covers all the core
combinators of Figure 1 and provides optimizations for the
derived operations (stream filtering, mapping, windowing,
event counting, etc.). We have already discussed in Remark 6
that our implementation of sliding windows does not use the
expensive encoding of \S 3.1 but instead allows the use of
efficient algorithms for the manipulation of the window. The
data structure of approximate histograms and the algorithm
for approximate event counting, as described in \S 4.1 are also
supported.

The goals of our experimental evaluation are the follow-
ing: (1) examine if the language constructs of StreamQRE are
sufficiently expressive and flexible to describe useful stream
transformations on realistic workloads, (2) check whether
our compilation procedure produces efficient code that can
achieve high performance, and (3) evaluate whether approxi-
where `eventType` can be encoded using the core regular constructs, but an efficient engine would require additional components indicating the interaction of a user with an advertisement, (and stored in memory. Each element of the data stream is of static map from ads to campaigns is computed ahead-of-time by comparing it against three other streaming engines: RxJava implementation was given. The streaming operations marked supported by StreamQRE, RxJava, Esper and Flink.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1. host language integration</td>
<td>yes</td>
<td>N/A</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>2. streaming composition</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>3. stream filtering</td>
<td>yes</td>
<td>-</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>4. stream mapping</td>
<td>yes</td>
<td>-</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>5. sequential aggregation</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>6. key-based partitioning</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>7. tumbling windows</td>
<td>yes</td>
<td>-</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>8. sliding windows</td>
<td>yes</td>
<td>-</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>9. pattern-based windows</td>
<td>yes</td>
<td>-</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>10. regular parsing</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>11. approximate aggregation</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>12. incremental windows</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>

Figure 5. Some of the features and stream transformations supported by StreamQRE, RxJava, Esper and Flink.

The Yaho Benchmark [22] specifies the Yahoo benchmark involving a multi-stage pipeline: (a) `stage 1`: filter view events, (b) `stage 2`: project the ad id from each view tuple, (c) `stage 3`: lookup the campaign id of each ad, (d) `stage 4`: compute for every one-second window the number of events (views) associated with each campaign. The query invokes key-based partitioning on only one property, namely the derived campaign id of the event.

**Query Y2.** We extend the Yahoo benchmark with a more complex query. An important part of organizing a marketing campaign is quantifying how successful ads are. We define success as the number of users who purchase the product after viewing an ad for it. Our query outputs, at the end of every second, a map from campaigns to the most successful ad of the campaign so far, together with its success score.

**NEXMark benchmark.** The Niagara Extension to XMark benchmark (NEXMark) [43] concerns the monitoring of an on-line auction system such as eBay. Four kinds of events are recorded in the event stream: (a) Person events, which describe the registering of a new person to the auction system, (b) Item events, which mark the start of an auction for a specified item, (c) Close events, which mark the end of an auction for a specified item, and (d) Bid events, which record the bids made for items that are being auctioned.

```
Person(personId, name.ts) Close(itemId, ts)
Item(itemId, sellerId, initPrice, ts, dur, category)
Bid(itemId, bidderId, bidIncrement, ts)
```

Every event contains the field `ts`, which is the timestamp of when the event occurred. Every new auction event (of type `Item`) specifies an initial price `initPrice` for the item, the duration `dur` of the auction, and the category to which the item belongs. Every bid event contains the bid increment, that is, the increment by which the previous bid is raised. So, to find the current bid for an item we need to add the initial price of the item together with all the bid increments for the item so far. We have chosen five queries, which are minor variants of some of the queries of the NEXMark benchmark: (N1) Output at every auction start or close the number of currently open auctions. (N2) Output at every auction close the average closing price of items sold so far. (N3) Output at every auction close the average closing price of items per category. (N4) Output every 10 minutes the highest bid within the last 10 minutes. (N5) Output every 10 minutes the item with the most bids in the last 24 hours.

Queries N2 and N3 involve identifying for every item a pattern-based window that spans all events relating to the auction for that item. These windows are naturally specified by the regular pattern `Item * Bid * Close`. The constructs of StreamQRE are very well suited for this purpose, whereas the encoding of these queries in RxJava, Esper and Flink is arguably awkward and error-prone.

**Results.** Figure 6 summarizes the throughput results for the queries of the Yahoo and NEXMark benchmark. Every exper-
We have already mentioned that regular patterns facilitate the implementation of N2 and N3. The ability to extract pattern-based windows of interest from a stream and compute incrementally with their elements distinguishes StreamQRE from other engines. For example, for the patient stream of \( \text{N} \), we can easily extract episodes with at least one extreme measurement and return the average of the measurements after the last occurrence of an extreme one. Programming such realistic queries in RxJava, Esper or Flink would be extremely cumbersome.

6. Related Work

Streaming databases. There is a large body of work on streaming database languages and systems such as Aurora \([5]\), Borealis \([4]\), STREAM \([14]\), and StreamInsight \([7][18]\). The query language supported by these systems (for example, CQL \([15]\)) is typically a version of SQL with additional constructs for sliding windows over data streams. This allows for rich relational queries, including set-aggregations (e.g., sum, maximum, minimum, average, count) and joins over multiple data streams. Such SQL-based languages are, however, severely limited in their ability to express properties and computations that rely on the sequence of the events such as:

\[
\begin{array}{ccccccc}
\text{Y1} & 2.3 & 6.2 & 18 \\
\text{Y2} & 3.6 & 6.7 & 9.8 \\
\text{N1} & 4.3 & 76 & 141 \\
\text{N2} & 2.1 & 22 & 42 \\
\text{N3} & 2.1 & 21 & 42 \\
\text{N4} & 2.0 & 27 & 35 \\
\text{N5} & 2.6 & 18 & 33 \\
\end{array}
\]

\[slo\text{wdown w.r.t. Q} = \frac{R - E}{E} \times 100\%\]

\[\text{Table 1. Performance for queries with approximation.}
\]

\[
\begin{array}{cccccc}
\text{Throughput} & \text{Space (1000 bytes)} & \text{T o U} \\
\text{Exact} & \text{Approx} & \text{Exact} & \text{Approx} & \text{Error} \\
P6 & 8.208 & 8.643 & 1,458 & \text{25.3} & 0.14\% \\
P7 (w = 10^4) & 6.747 & 3.221 & 151 & 9.9 & 0.09\% \\
P7 (w = 10^5) & 5.100 & 2.067 & 1,501 & 14.9 & 0.52\% \\
P7 (w = 10^6) & 4.030 & 1.863 & 15,001 & 19.4 & 0.19\% \\
P7 (w = 2 \cdot 10^6) & 3.411 & 1.857 & 30,001 & 20.9 & 0.24\% \\
P7 (w = 4 \cdot 10^6) & 2.050 & 1.871 & 60,001 & 23.3 & 0.30\% \\
P7 (w = 6 \cdot 10^6) & 1.434 & 1.876 & 90,001 & 23.2 & 0.34\% \\
P7 (w = 8 \cdot 10^6) & 1.028 & 1.864 & 120,001 & 23.7 & 0.03\% \\
P7 (w = 10^7) & 0.826 & 1.841 & 150,001 & 24.3 & 0.36\%
\end{array}
\]
sequence-based pattern-matching, and numerical computation based on list-iteration when the order of the data items is significant (e.g., average of piecewise-linear interpolation). In contrast, our query language allows the combination of relational with sequence-based operations, which poses unique technical challenges that are not addressed in the aforementioned works. Of particular relevance are engines such as IBM’s Stream Processing Language (SPL) [34, 45], ReactiveX [3], Esper [2] and Flink [1], which support user-defined types and operations, and allow for both relational and stateful sequential computation. However, none of these engines provides support for decomposing the stream in a regular fashion and performing incremental computations that reflect the structure of the parse tree, a useful and challenging feature (especially when combined with key-based partitioning) that is central in our work. Another aspect not addressed in any of the aforementioned systems is approximate query evaluation with strong guarantees, for example approximate medians and approximate event-counting over sliding windows. Our use of synchronization elements bears resemblance to the punctuations of [36, 44], which are used to trigger the closing of windows. The setting here is much more general since we allow arbitrary regular patterns of synchronization elements.

**Complex Event Processing (CEP).** The literature on CEP [20, 21, 23, 32, 33, 48, 39] is concerned with the recognition of complex patterns over streaming data. The patterns are typically given as queries that resemble regular expressions or as automata-based models. In some of these proposals a pattern can depend on the evolution of values: for example, a pattern where the price of a stock is constantly increasing. While CEP languages offer powerful event-selection capabilities, they cannot express the hierarchical computations that StreamQRE specifies.

**String transformations.** The language of StreamQRE is closely related to automata- and transducer-based formalisms for the manipulation of strings. There have been several recent works on the subject [10, 13, 24, 29, 35]. The current work draws especially from [11], where the language of Quantitative Regular Expressions (QREs) is proposed. It is a language based on regular expressions, intended for describing simple numerical calculations over streaming data. There are several crucial differences between [11] and the current work. The most important one is that we consider here relational structures and operations, including an explicit construct map-collect for building relations by partitioning the stream on keys and collecting the outputs of the sub-streams. The map-collect combinator presents significant new challenges, both for defining its semantics and for efficient type-checking and evaluation (Remarks 5 and 8). The lack of such a construct from the QREs of [11] implies that the benchmark queries of [5] are inexpressible. Another important technical difference between the present work and [11] is that our iteration scheme is based on list-iteration (the fold combinator), as opposed to a complicated parameter-passing scheme involving variables and algebraic terms. This makes the language much simpler and easier to use. Moreover, it enables efficient evaluation using small stacks of values, as opposed to the arithmetic terms of [11], which require some kind of compression that is dependent on the nature of the primitive operations. Using approximation, we also handle here the challenging median operation, as well as sliding-window event counting [25].

**Streaming algorithms.** Our work addresses issues of streaming computation that are complementary to the algorithmic questions that are tackled by the work on data stream algorithms. The seminal paper of Alon, Matias and Szegedy [8] on the streaming computation of frequency moments gave enormous momentum to the area and spawned a lot of subsequent research. See [39] for a broad exposition of the area. Of particular relevance is the work on the approximate computation of quantile queries (see [30] for a survey). The goal is to return an element of rank \( r \), and the \( \varepsilon \)-approximation relaxation is that the algorithm is allowed to return an element of rank \( \hat{r} \) satisfying the guarantee:

\[
\frac{r}{n} - \varepsilon n \leq \hat{r} \leq r + \varepsilon n,
\]

where \( n \) is the length of the stream. For our application, even if we had an element of rank \( r - 1 \) or \( r + 1 \) (exactly adjacent to \( r \)), its value could be arbitrarily far from the exact value. This means that the above approximation notion is not appropriate for our work. Instead, we want an \( \varepsilon \)-approximate answer that is guaranteed to be close in value to the exact answer. Much of the work within the streaming algorithms literature has focused on approximating specific functions in small space, whereas we investigate here is a wide class of queries with hierarchical nesting of several numerical operations, and we provide a general framework for dealing with such queries.

7. Conclusion

We have introduced the StreamQRE language, a high-level formalism for processing streaming data. Our query language integrates two paradigms for programming with streams: streaming relational languages with windowing constructs, and state-machine-based models for pattern-matching and performing sequence-aware computations. We have presented a small but powerful core language, which has a formal denotational semantics and a decidable type system. The expressiveness of the language has been illustrated by encoding common patterns and programming significant examples. A compilation procedure has been described that translates a query into a streaming algorithm with precise space and time usage guarantees. We have also shown how to incorporate efficient approximation algorithms in our framework, e.g. for computing the median of a collection of numbers and for counting the number of events occurring over large windows of time. A Java implementation of StreamQRE has been provided, and it has been evaluated using two benchmarks: Yahoo [22] and NEXMark [43].
References


