An Introduction to the StreamQRE Language

Rajeev ALUR and Konstantinos MAMOURAS
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104, USA

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. We give here an introduction to the StreamQRE language, which has recently been proposed for the purpose of simplifying the task of programming the desired logic in such stream processing applications. StreamQRE provides natural and high-level constructs for processing streaming data, and it offers 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.

Keywords. data stream processing, Quantitative Regular Expressions

Introduction

The last few years have witnessed an explosion of IoT systems in applications such as smart buildings, wearable devices, and healthcare [1]. 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 recently proposed language StreamQRE [2] (pronounced StreamQuery) is meant to assist IoT programmers: it 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 clas-

sical regular expressions for associating numerical values with strings [3], with constructs typical in extensions of relational query languages for handling streaming data [4,5,6,7,8,9,10,11]. The novel integration of linguistic constructs allows the programmer to impart to the input data stream a logical hierarchical structure (for instance, view patient data as a sequence of episodes and view network traffic as a sequence of Voice-over-IP sessions) and also employ relational constructs to partition the input data by keys (e.g., patient identifiers and IP addresses).

The basic object in the language is a *streaming query*, which is modeled as a partial function from sequences of input data items to an output value (which can be a relation). We present the syntax and semantics of the StreamQRE language with type-theoretic foundations. In particular, each streaming query has an associated *rate* that captures its domain, that is, as it reads the input data stream, the prefixes that trigger the production of the output. In the StreamQRE calculus, the rates are required to be *regular*, captured by symbolic regular expressions, and the theoretical foundations of symbolic automata [12] lead to decision procedures for constructing well-typed expressions. Regular rates also generalize the concept of *punctuations* in streaming database literature [6].

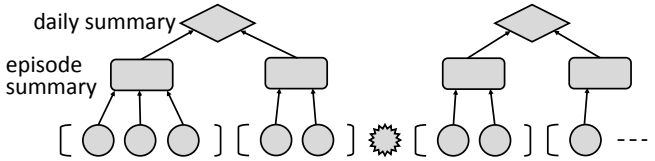
The StreamQRE language has a small set of core combinators with clear semantics. An atomic query processes individual items. The constructs `split` and `iter` are quantitative analogs of concatenation and Kleene-iteration, and integrate hierarchical pattern matching with familiar sequential iteration over a list of values. The global choice operator `or` allows selection between two expressions with disjoint rates. The combination operator `combine` allows combining output values produced by multiple expressions with equivalent rates processing input data stream in parallel. The key-based partitioning operator `map-collect` is a generalization of the widely used map-reduce construct that partitions the input data stream into a set of sub-streams, one per key, and returns a relation. Finally, the streaming composition operator $\gg$ streams the sequence of outputs produced by one expression as an input stream to another, allowing construction of pipelines of operators.

The core StreamQRE constructs can be used to define a number of derived patterns that are useful in practice, such as *tumbling* and *sliding windows* [6], selection, and filtering. The language has been implemented as a Java library that supports the basic and derived constructs [13]. We show how to program in StreamQRE using an illustrative example regarding monitoring patient measurements, the recent Yahoo Streaming Benchmark for advertisement-related events [14], and the NEXMark Benchmark for auction bids [15]. These examples illustrate how hierarchically nested iterators and global case analysis facilitate modular stateful sequential programming, and key-based partitioning and relational operators facilitate traditional relational programming. The two styles offer alternatives for expressing the same query in some cases, while some queries are best expressed by intermingling the two views.

Organization. The remaining paper is organized as follows. Section 1 introduces the syntax and semantics of the StreamQRE language, and explains how each construct is used with an illustrative example regarding monitoring patient measurements. Section 2 shows how to use StreamQRE to program some common stream transformation. Section 3 presents some example queries for the Yahoo Streaming Benchmark, and Section 4 gives queries for the NEXMark Benchmark.

1. The StreamQRE Language

As a motivating example, suppose that a patient is being monitored for episodes of a physiological condition such as epilepsy [16], and the data stream consists of four types of events: (1) An event `B` marking the beginning of an episode, (2) a time-stamped measurement $\texttt{M}(\texttt{ts},\texttt{val})$ 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:



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 nonempty 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 end-of-day marker is shown as a star. The regular expression $((\texttt{B}\cdot\texttt{M}^{+}\cdot\texttt{E})^{*}\cdot\texttt{D})^{*}$ 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 $\texttt{B}\cdot\texttt{M}^{+}\cdot\texttt{E}$) is an aggregation of the measurements (pattern `M`) it contains, and similarly the summary of each day (pattern $(\texttt{B}\cdot\texttt{M}^{+}\cdot\texttt{E})^{*}\cdot\texttt{D}$) 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) [12]. The use of regular rates implies decidability of unambiguity. Even better, there are efficiently checkable typing rules that guarantee unambiguity for all policies [17,18,19].

To define quantitative queries, we first choose a typed signature which describes 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 `Bool` of boolean values, and the unit type `Ut` whose unique inhabitant is denoted by `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 `Nat`, the integers `Int`, and the real numbers `R`.

We also fix a collection of *basic operations* on the basic types, for example the k-ary operation $op : A_1 \times \cdots \times A_k \to B$. The identity function on D is written as $id_D : D \to D$, 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 pro-

jections, and is closed under pairing and function composition. To describe derived operations we use a variant of lambda notation that is similar to Java's lambda expressions [20]. That is, we write $(A\ x)\ \texttt{->}\ t(x)$ to mean $\lambda x{:}A.t(x)$ and $(A\ x,\ B\ y,\ C\ z)\ \texttt{->}\ t(x,y,z)$ to mean $\lambda x{:}A, y{:}B, z{:}C.t(x,y,z)$. For example, the identity function on D is $(D\ x)\ \texttt{->}\ x$, the left projection on $A \times B$ is $(A\ x,\ B\ y)\ \texttt{->}\ x$, the right projection on $A \times B$ is $(A\ x,\ B\ y)\ \texttt{->}\ y$, and $(D\ x)\ \texttt{->}\ \texttt{def}$ is the unique function from D to $\texttt{Ut}$. We will typically use lambda expressions in the context of queries from which the types of the input variables can be inferred, so we will omit them as in $(x,y)\ \texttt{->}\ x$.

For every basic type D, assume that we have fixed a collection of *atomic predicates*, so that the satisfiability of their Boolean combinations (built up using the Boolean operations: and, or, not) is decidable. We write $\varphi : D \to \texttt{Bool}$ to indicate that φ is a predicate on D, and we denote by $\mathit{true}_D : D \to \texttt{Bool}$ the predicate that is always true. The predicate $((\texttt{Int}\ x)\ \texttt{->}\ x > 0) : \texttt{Int} \to \texttt{Bool}$ is true of the strictly positive integers.

Example 1. For the example patient-monitoring stream described previously, suppose that we now allow the stream to contain information for several patients. The data type $\texttt{D}_\texttt{P}$ for this multiple-patient monitoring stream is the tagged (disjoint) union:

$$\texttt{D}_\texttt{P} = \{\texttt{D}\} \cup \{\texttt{B}(p), \texttt{E}(p) \mid p \in \texttt{PID}\} \cup \{\texttt{M}(p,t,v) \mid p \in \texttt{PID},\ t \in \texttt{T}, \text{ and } v \in \texttt{V}\},$$

where PID is the set of patient identifiers, T is the set of timestamps, and V is the set of scalars for the measurements. The projection functions $\texttt{typ} : \texttt{D}_\texttt{P} \to \{\texttt{D},\texttt{B},\texttt{E},\texttt{M}\}$, $\texttt{pId} : \texttt{D}_\texttt{P} \to \texttt{PID}$, $\texttt{ts} : \texttt{D}_\texttt{P} \to \texttt{T}$ and $\texttt{val} : \texttt{D}_\texttt{P} \to \texttt{V}$ get the type, patient identifier, timestamp, and value of a data item respectively (when undefined, the functions simply return some default value). For a data item $x \in \texttt{D}_\texttt{P}$, we write $x.\texttt{typ}$, $x.\texttt{pId}$, $x.\texttt{ts}$ and $x.\texttt{val}$ to denote the application of these functions.

Symbolic regular expressions. For a type D, we define the set of *symbolic regular expressions over* D [21], denoted $\texttt{RE}\langle D\rangle$, with the following grammar:

$$\begin{array}{lll} r ::= \varphi \mid & & \text{[predicate on } D\text{]} \\ \quad \varepsilon \mid & & \text{[empty sequence]} \\ \quad r \sqcup r \mid & & \text{[nondeterministic choice]} \\ \quad r \cdot r \mid & & \text{[concatenation]} \\ \quad r^*. & & \text{[iteration]} \end{array}$$

The concatenation symbol $\cdot$ is sometimes omitted, that is, we write rs instead of $r \cdot s$. The expression r^+ (iteration at least once) abbreviates $r \cdot r^*$. We write $r : \texttt{RE}\langle D\rangle$ to indicate the r is a regular expression over D. Every expression $r : \texttt{RE}\langle D\rangle$ is interpreted as a set $[\![r]\!] \subseteq D^*$ of finite sequences over D.

$$[\![\varphi]\!] \triangleq \{d \in D \mid \varphi(d) = \mathit{true}\} \qquad [\![r \sqcup s]\!] \triangleq [\![r]\!] \cup [\![s]\!] \qquad [\![r^*]\!] \triangleq \bigcup_{n \geq 0} [\![r]\!]^n$$
$$[\![\varepsilon]\!] \triangleq \{\varepsilon\} \qquad [\![r \cdot s]\!] \triangleq [\![r]\!] \cdot [\![s]\!]$$

For subsets $X, Y \subseteq D^*$, we define $X \cdot Y = \{uv \mid u \in X \text{ and } v \in Y\}$ and X^n is given by induction on n as follows: $X^0 = \{\varepsilon\}$ and $X^{n+1} = X^n \cdot X$. Two expressions are said to be *equivalent* is they denote the same language.

	data item	stream seen so far	current output (if any)
		ε	$f(\varepsilon)$
	d_1	d_1	$f(d_1)$
	d_2	$d_1\,d_2$	$f(d_1\,d_2)$
time ↓	d_3	$d_1\,d_2\,d_3$	$f(d_1\,d_2\,d_3)$
	d_4	$d_1\,d_2\,d_3\,d_4$	$f(d_1\,d_2\,d_3\,d_4)$
	d_5	$d_1\,d_2\,d_3\,d_4\,d_5$	$f(d_1\,d_2\,d_3\,d_4\,d_5)$
	$\ldots$		

Figure 1. A streaming transformation f specifies the output for every prefix of the stream.

Example 2. The symbolic regular expression $((\texttt{Nat}\ x)\texttt{->}\mathit{true})^* \cdot ((\texttt{Nat}\ x)\texttt{->}x > 0)$ over the type `Nat` of natural numbers denotes all sequences that end with a strictly positive number.

Unambiguity. The notion of unambiguity for regular expressions [17] is a way of formalizing the requirement of uniqueness of parsing. The languages L_1, L_2 are said to be *unambiguously concatenable* 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 $w = w_1 w_2$. 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 $w = w_1 \cdots w_n$. The definitions of unambiguous concatenability 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. For every subexpression $e_1 \sqcup e_2$, e_1 and e_2 are *disjoint*.
2. For every subexpression $e_1 \cdot e_2$, e_1 and e_2 are *unambiguously concatenable*.
3. For every subexpression e^*, e is *unambiguously iterable*.

Checking whether a regular expression is unambiguous can be done in polynomial time. For the case of symbolic regular expressions this results still holds, under the assumption that satisfiability of the predicates can be decided in unit time [3].

Example 3. Consider the finite alphabet $\Sigma = \{a, b\}$. The regular expression $r = (a + b)^* b (a + b)^*$ denotes the set of sequences with at least one occurrence of b. It is ambiguous, because the subexpressions $(a+b)^* b$ and $(a+b)^*$ are not unambiguously concatenable: the word $w = ababa$ matches r, but there are two different splits $w = ab \cdot aba$ and $w = abab \cdot a$ that witness the ambiguity of parsing. The regular expressions $a^* b (a+b)^*$ and $(a+b)^* b a^*$ are both equivalent to r, and they are unambiguous.

Streaming transformations. The basic object in the StreamQRE language is the *query*, which describes the transformation of an input stream into an output stream. At any given moment in time, only a finite number of data items have arrived, therefore a stream transformation can be modeled as a function from D^*, where D is the type of input data items, to C, where C is the type of the outputs. In other words, the transformation describes how to aggregate the entire stream seen so far into an output value. As the input stream gets extended with more and more items, the emitted outputs form a stream of elements of C. We want to allow for the possibility of not having an output with every new element arrival, therefore a streaming transformation is modeled as a partial function $D^* \rightharpoonup C$. See Figure 1 for an illustration.

Example 4. For example, suppose we want to describe a *filtering* transformation on a stream of integers, where only the nonnegative numbers are retained and the negative numbers are filtered out. The function $f : \texttt{Int}^* \rightharpoonup \texttt{Int}$ that describes this transformation is defined on the nonempty sequences $v_1 v_2 \ldots v_n$ with $v_n \geq 0$, and the value is the last number of the sequence, i.e. the current item.

time ↓

data item	stream seen so far	current output (if any)
	ε	
3	3	3
-2	3 -2	
4	3 -2 4	4
8	3 -2 4 8	8
-1	3 -2 4 8 -1	
5	3 -2 4 8 -1 5	5
...		

The set of stream prefixes for which this function is defined is denoted by the symbolic regular expression $((\texttt{Int}\ x) \texttt{->} \mathit{true})^* \cdot ((\texttt{Int}\ x) \texttt{->} x > 0)$. After having consumed the input stream 3 -2 4 8 -1 5, the overall output stream is 3 4 8 5.

The rate of a transformation describes which stream prefixes trigger the production of output. In some other formalisms, such as transducers [22,23] and synchronous languages [24], output is typically produced at every new data item arrival. In StreamQRE, on the other hard, output does not have to be produced with every new item. For example, after processing n data items, the output stream generally consists of $k \leq n$ items.

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 query represents a streaming transformation whose domain is a regular set over the input data type. The *rate* of a query `f`, written $\mathsf{R}(\texttt{f})$, is a symbolic regular expression that denotes the domain of the transformation that `f` represents. 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 $\texttt{QRE}\langle D, C\rangle$ to denote that the input stream has elements of type D and the outputs are of type C. Figure 2 shows the full formal syntax of streaming queries. The decidability of type checking follows from results in [17,18,19] and it is also discussed in [3].

Example 5 (Rate of a query)**.** In the patient monitoring example described in the beginning of this section, 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 $((\texttt{B} \cdot \texttt{M}^+ \cdot \texttt{E})^* \cdot \texttt{D})^*$. The tag `B` in this expression is abbreviation for the predicate $(x \texttt{->} x.\texttt{typ} = \texttt{B})$. We also write $\neg\texttt{B}$ to stand for the predicate $(x \texttt{->} x.\texttt{typ} \neq \texttt{B})$. Similar abbreviations are considered for the tags `M`, `E` and `D`. We will be using these abbreviations freely from now on, since their meaning is obvious from the context.

$$\frac{\text{satisfiable } \varphi : D \to \texttt{Bool} \qquad op : D \to C}{\begin{array}{c}\texttt{atom}(\varphi,\ op) : \texttt{QRE}\langle D, C\rangle \\ \mathsf{R}(\texttt{atom}(\varphi,\ op)) = \varphi\end{array}}\ (\textbf{\textit{atomic}})$$

$$\frac{c \in C}{\begin{array}{c}\texttt{eps}(c) : \texttt{QRE}\langle D, C\rangle \\ \mathsf{R}(\texttt{eps}(c)) = \varepsilon\end{array}}\ (\textbf{\textit{empty}})$$

$$\frac{\begin{array}{c}\texttt{f} : \texttt{QRE}\langle D, A\rangle \qquad \texttt{g} : \texttt{QRE}\langle D, B\rangle \qquad op : A \times B \to C \\ \mathsf{R}(\texttt{f}) \text{ and } \mathsf{R}(\texttt{g}) \text{ are unambiguously concatenable}\end{array}}{\begin{array}{c}\texttt{split}(\texttt{f},\ \texttt{g},\ op) : \texttt{QRE}\langle D, C\rangle \\ \mathsf{R}(\texttt{split}(\texttt{f},\ \texttt{g},\ op)) = \mathsf{R}(\texttt{f}) \cdot \mathsf{R}(\texttt{g})\end{array}}\ (\textbf{\textit{concatenation}})$$

$$\frac{\begin{array}{c}\texttt{init} : \texttt{QRE}\langle D, B\rangle \qquad \texttt{body} : \texttt{QRE}\langle D, A\rangle \qquad op : B \times A \to B \\ \mathsf{R}(\texttt{body}) \text{ is unambiguously iterable} \\ \mathsf{R}(\texttt{init}) \text{ and } \mathsf{R}(\texttt{body})^* \text{ are unambiguously concatenable}\end{array}}{\begin{array}{c}\texttt{iter}(\texttt{init},\ \texttt{body},\ op) : \texttt{QRE}\langle D, B\rangle \\ \mathsf{R}(\texttt{iter}(\texttt{f},\ \texttt{g},\ op)) = \mathsf{R}(\texttt{f}) \cdot \mathsf{R}(\texttt{g})^*\end{array}}\ (\textbf{\textit{iteration}})$$

$$\frac{\texttt{f} : \texttt{QRE}\langle D, C\rangle \qquad \texttt{g} : \texttt{QRE}\langle D, C\rangle \qquad \mathsf{R}(\texttt{f}) \text{ and } \mathsf{R}(\texttt{g}) \text{ are disjoint}}{\begin{array}{c}\texttt{or}(\texttt{f},\ \texttt{g}) : \texttt{QRE}\langle D, C\rangle \\ \mathsf{R}(\texttt{or}(\texttt{f},\ \texttt{g})) = \mathsf{R}(\texttt{f}) \sqcup \mathsf{R}(\texttt{g})\end{array}}\ (\textbf{\textit{choice}})$$

$$\frac{\texttt{f} : \texttt{QRE}\langle D, A\rangle \qquad op : A \to B}{\begin{array}{c}\texttt{apply}(\texttt{f},\ op) : \texttt{QRE}\langle D, B\rangle \\ \mathsf{R}(\texttt{apply}(\texttt{f})) = \mathsf{R}(\texttt{f})\end{array}}\ (\textbf{\textit{application}})$$

$$\frac{\texttt{f} : \texttt{QRE}\langle D, A\rangle \qquad \texttt{g} : \texttt{QRE}\langle D, B\rangle \qquad op : A \times B \to C \qquad \mathsf{R}(\texttt{f}), \mathsf{R}(\texttt{g}) \text{ equivalent}}{\begin{array}{c}\texttt{combine}(\texttt{f},\ \texttt{g},\ op) : \texttt{QRE}\langle D, C\rangle \\ \mathsf{R}(\texttt{combine}(\texttt{f},\ \texttt{g},\ op)) = \mathsf{R}(\texttt{f})\end{array}}\ (\textbf{\textit{combination}})$$

$$\frac{\texttt{f} : \texttt{QRE}\langle D, C\rangle \qquad \texttt{g} : \texttt{QRE}\langle C, E\rangle}{\texttt{f} \gg \texttt{g} : \texttt{QRE}\langle D, E\rangle}\ (\textbf{\textit{streaming composition}})$$

$$\frac{\begin{array}{c}\varphi_S : D \to \texttt{Bool} \qquad m : D \to K \qquad \texttt{f} : \texttt{QRE}\langle D, C\rangle \qquad r : \texttt{RE}\langle D\rangle \qquad \mathsf{R}(\texttt{f}) \subseteq r \setminus \varphi_S^* \\ r = s[(\neg\varphi_S)^*\varphi_S/\varphi_S], \text{ where the only predicate } s : \texttt{RE}\langle D\rangle \text{ can contain is } \varphi_S\end{array}}{\begin{array}{c}\texttt{map-collect}(\varphi_S,\ m,\ \texttt{f},\ r) : \texttt{QRE}\langle D, \texttt{Map}\langle K, C\rangle\rangle \\ \mathsf{R}(\texttt{map-collect}(\varphi, m, \texttt{f}, r)) = r\end{array}}\ (\textbf{\textit{key-based partitioning}})$$

Figure 2. The syntax of Streaming Quantitative Regular Expressions.

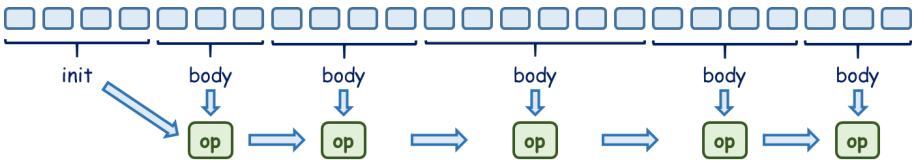
Atomic queries. The basic building blocks of queries are expressions that describe the processing of a single data item. Suppose $\varphi : D \to \texttt{Bool}$ is a predicate over the data item type D and $op : D \to C$ is an operation from D to the output type C. Then, the *atomic query* $\texttt{atom}(\varphi, op) : \texttt{QRE}\langle D, C\rangle$, with rate φ, is defined on single-item streams that satisfy the predicate φ. The output is the value of op on the input element.

Notation: It is very common for op to be the identity function, and φ to be the always-true predicate. So, we abbreviate the query $\texttt{atom}(\varphi,\ (D\ x)\texttt{->}x)$ by $\texttt{atom}(\varphi)$, and the query $\texttt{atom}((D\ x)\texttt{->}\mathit{true})$ by $\texttt{atom}()$.

Example 6. For the stream of monitored patients, the query that matches a single measurement item and returns its value is $\texttt{f} = \texttt{atom}(x\texttt{->}x.\texttt{typ} = \texttt{M},\ x\texttt{->}x.\texttt{val})$. The type of $\texttt{f}$ is $\texttt{QRE}\langle \texttt{D}_\texttt{P}, \texttt{V}\rangle$ and its rate is $\texttt{M}$.

Empty sequence. The query $\texttt{eps}(c) : \texttt{QRE}\langle D, C\rangle$, where c is a value of type C, is only defined on the empty sequence ε and it returns the output c.

Iteration. Suppose that $\texttt{init} : \texttt{QRE}\langle D, B\rangle$ describes the computation for initializing an aggregate value of type B, and $\texttt{body} : \texttt{QRE}\langle D, A\rangle$ describes a computation that we want to iterate over consecutive subsequences of the input stream, in order to aggregate the values (of type A) sequentially using an aggregator $op : B \times A \to B$.



More specifically, we split the input stream w into subsequences $w = u\ w_1\ w_2 \ldots\ w_n$, where u matches $\texttt{init}$ and each w_i matches $\texttt{body}$. We apply $\texttt{init}$ to u and $\texttt{body}$ to each of the w_i, thus producing the output values b_0 and $a_1\ a_2\ \cdots\ a_n$ with $b_0 = \texttt{init}(u)$ and $a_i = \texttt{body}(w_i)$. Finaly, we combine these results using the list iterator *left fold* with start value b_0 and aggregation operation $op : B \times A \to B$ by folding the list of values $a_1\ a_2\ \cdots\ a_n$. This can be formalized with the combinator $\text{fold} : B \times (B \times A \to B) \times A^* \to B$, which takes an initial value $b \in B$ and a stepping map $op : B \times A \to B$, and iterates through a sequence of values of type A:

$$\text{fold}(b, op, \varepsilon) = b \qquad\qquad \text{fold}(b, op, \gamma a) = op(\text{fold}(b, op, \gamma), a)$$

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

The query $\texttt{g} = \texttt{iter}(\texttt{init},\ \texttt{body},\ op) : \texttt{QRE}\langle D, B\rangle$ describes the computation of the previous paragraph. In order for g to be well-defined as a function, every input stream w that matches g must be uniquely decomposable into $w = uw_1w_2 \ldots w_n$ with u matching $\texttt{init}$ and each w_i matching $\texttt{body}$. This requirement can be expressed equivalently as follows: the rate $\mathsf{R}(\texttt{body})$ is unambiguously iterable, and the rates $\mathsf{R}(\texttt{init})$, $\mathsf{R}(\texttt{body})^*$ are unambiguously concatenable.

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.

Example 7. For the stream of monitored patients, the query g below matches a sequence of measurements and returns the sum of their values.

$$\texttt{f} : \texttt{QRE}\langle \texttt{D}_\texttt{P}, \texttt{V} \rangle = \texttt{atom}(x \texttt{->} x.\texttt{typ} = \texttt{M},\ x \texttt{->} x.\texttt{val}) \qquad \text{// rate } \texttt{M}$$

$$\texttt{g} : \texttt{QRE}\langle \texttt{D}_\texttt{P}, \texttt{V} \rangle = \texttt{iter}(\texttt{eps}(0),\ \texttt{f},\ (x,y) \texttt{->} x+y) \qquad \text{// rate } \texttt{M}^*$$

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 $A \times B \to C$. The query 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 combine(f, g, h, (x,y,z) -> $op(x,y,z)$) for the ternary variant. In particular, we write apply(f, *op*) for the case of one argument.

Example 8 (Average). For the stream of monitored patients, the query h below matches a nonempty sequence of measurements and returns the average of their values.

$$\texttt{f}_1 : \texttt{QRE}\langle \texttt{D}_\texttt{P}, \texttt{V} \rangle = \texttt{atom}(x \texttt{->} x.\texttt{typ} = \texttt{M},\ x \texttt{->} x.\texttt{val}) \qquad \text{// rate } \texttt{M}$$

$$\texttt{g}_1 : \texttt{QRE}\langle \texttt{D}_\texttt{P}, \texttt{V} \rangle = \texttt{iter}(\texttt{f}_1,\ \texttt{f}_1,\ (x,y) \texttt{->} x+y) \qquad \text{// rate } \texttt{M}^+$$

$$\texttt{f}_2 : \texttt{QRE}\langle \texttt{D}_\texttt{P}, \texttt{V} \rangle = \texttt{atom}(x \texttt{->} x.\texttt{typ} = \texttt{M},\ x \texttt{->} 1) \qquad \text{// rate } \texttt{M}$$

$$\texttt{g}_2 : \texttt{QRE}\langle \texttt{D}_\texttt{P}, \texttt{V} \rangle = \texttt{iter}(\texttt{f}_2,\ \texttt{f}_2,\ (x,y) \texttt{->} x+y) \qquad \text{// rate } \texttt{M}^+$$

$$\texttt{h} : \texttt{QRE}\langle \texttt{D}_\texttt{P}, \texttt{V} \rangle = \texttt{combine}(\texttt{g}_1,\ \texttt{g}_2,\ (x,y) \texttt{->} x/y) \qquad \text{// rate } \texttt{M}^+$$

The query $\texttt{g}_1$ computes the sum of a nonempty sequence of measurements, and the query $\texttt{g}_2$ computes the length of a nonempty sequence of measurements. An alternative implementation uses a single iteration construct and an accumulator that is the pair of the running sum and the running count.

$$\texttt{f}' : \texttt{QRE}\langle \texttt{D}_\texttt{P}, \texttt{V} \times \texttt{V} \rangle = \texttt{atom}(x \texttt{->} x.\texttt{typ} = \texttt{M},\ x \texttt{->} (x.\texttt{val}, 1)) \qquad \text{// rate } \texttt{M}$$

$$\texttt{g}' : \texttt{QRE}\langle \texttt{D}_\texttt{P}, \texttt{V} \times \texttt{V} \rangle = \texttt{iter}(\texttt{f}',\ \texttt{f}',\ (x,y) \texttt{->} x+y) \qquad \text{// rate } \texttt{M}^+$$

$$\texttt{h}' : \texttt{QRE}\langle \texttt{D}_\texttt{P}, \texttt{V} \rangle = \texttt{apply}(\texttt{g}',\ x \texttt{->} \pi_1(x)/\pi_2(x)) \qquad \text{// rate } \texttt{M}^+$$

The $+$ operation in $\texttt{g}'$ is componentwise addition of pairs of values. The query $\texttt{h}'$ computes the average by diving the running sum by the running count.

Example 9 (Standard Deviation). For a sequence $x_1, x_2, \ldots, x_n$ of numbers, the *mean* is the simple average $\mu = (\sum_i x_i)/n$, and the *standard deviation* is $\sigma = (\sum_i (x_i - \mu)^2)/n$. Equivalently, we can calculate the quantity $\sigma \cdot n$ as follows:

$$\sigma \cdot n = \textstyle\sum_i (x_i^2 + \mu^2 - 2\mu x_i) = (\sum_i x_i^2) + n\mu^2 - 2\mu(\sum_i x_i) = (\sum_i x_i^2) - (\sum_i x_i)^2/n.$$

So, both the mean and the standard deviation can be calculated from the quantities $\sum_i x_i$ and $\sum_i x_i^2$ and the count n. This suggests the following query for the streaming computation of the standard deviation for a sequence of patient measurements:

$$\begin{aligned}
\mathtt{f_1} &: \mathtt{QRE}\langle \mathtt{D_P}, \mathtt{V}\rangle = \mathtt{atom}(x \mathtt{->} x.\mathtt{typ} = \mathtt{M},\ x \mathtt{->} 1) && \text{// rate } \mathtt{M}\\
\mathtt{g_1} &: \mathtt{QRE}\langle \mathtt{D_P}, \mathtt{V}\rangle = \mathtt{iter}(\mathtt{f_1},\ \mathtt{f_1},\ (x,y) \mathtt{->} x+y) && \text{// rate } \mathtt{M}^+\\
\mathtt{f_2} &: \mathtt{QRE}\langle \mathtt{D_P}, \mathtt{V}\rangle = \mathtt{atom}(x \mathtt{->} x.\mathtt{typ} = \mathtt{M},\ x \mathtt{->} x.\mathtt{val}) && \text{// rate } \mathtt{M}\\
\mathtt{g_2} &: \mathtt{QRE}\langle \mathtt{D_P}, \mathtt{V}\rangle = \mathtt{iter}(\mathtt{f_2},\ \mathtt{f_2},\ (x,y) \mathtt{->} x+y) && \text{// rate } \mathtt{M}^+\\
\mathtt{f_3} &: \mathtt{QRE}\langle \mathtt{D_P}, \mathtt{V}\rangle = \mathtt{atom}(x \mathtt{->} x.\mathtt{typ} = \mathtt{M},\ x \mathtt{->} x.\mathtt{val} \cdot x.\mathtt{val}) && \text{// rate } \mathtt{M}\\
\mathtt{g_3} &: \mathtt{QRE}\langle \mathtt{D_P}, \mathtt{V}\rangle = \mathtt{iter}(\mathtt{f_3},\ \mathtt{f_3},\ (x,y) \mathtt{->} x+y) && \text{// rate } \mathtt{M}^+\\
\mathtt{h} &: \mathtt{QRE}\langle \mathtt{D_P}, \mathtt{V}\rangle = \mathtt{combine}(\mathtt{g_1},\ \mathtt{g_2},\ \mathtt{g_3},\ (x,y,z) \mathtt{->} z - y^2/x) && \text{// rate } \mathtt{M}^+
\end{aligned}$$

The query $\mathtt{g_1}$ computes the running count, the query $\mathtt{g_2}$ computes the running sum $\sum_i x_i$, and the query $\mathtt{g_3}$ computes the running sum of squares $\sum_i x_i^2$.

Example 10 (Integration). For the single-patient monitoring stream, assume that the diagnosis depends on the average value of the *piecewise-linear interpolant* of the sampled measurements. Computing this quantity corresponds to integrating the interpolant over the interval of the measurements. That is, the quantitative summary of a given sequence (t_1, v_1), (t_2, v_2), …, (t_n, v_n) of timestamped values with $t_1 < t_2 < \cdots < t_n$ is

$$A_n = \frac{1}{(t_n - t_1)} \cdot \sum_{i=1}^{n-1} \frac{(v_i + v_{i+1})(t_{i+1} - t_i)}{2}.$$

To compute the quantity $S_n = \sum_{i=1}^{n-1} (v_i + v_{i+1})(t_{i+1} - t_i)$ incrementally we must maintain the vector $X_n = (t_n, v_n, S_n)$, where t_n is the last timestamp, v_n is the last value, and S_n is the running sum. We then put $X_1 = (t_1, v_1, 0)$ and

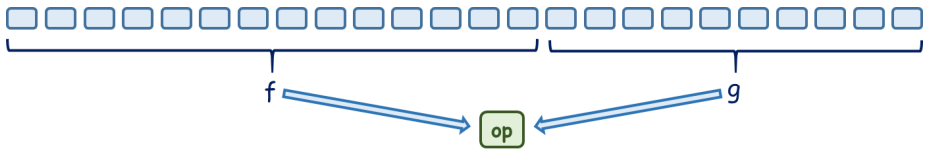
$$X_{n+1} = (t_{n+1},\ v_{n+1},\ S_n + (v_n + v_{n+1})(t_{n+1} - t_n)).$$

From $X_n = (t_n, v_n, S_n)$ and the first timestamp t_1 we can then compute $A_n = S_n / 2(t_n - t_1)$.

$$\begin{aligned}
\mathtt{f} &: \mathtt{QRE}\langle \mathtt{D_P}, \mathtt{D_P}\rangle = \mathtt{atom}(x \mathtt{->} x.\mathtt{typ} = \mathtt{M}) && \text{// rate } \mathtt{M}\\
\mathtt{g_1} &: \mathtt{QRE}\langle \mathtt{D_P}, \mathtt{T}\rangle = \mathtt{iter}(\mathtt{apply}(\mathtt{f},\ x \mathtt{->} x.\mathtt{ts}),\ \mathtt{f},\ (x,y) \mathtt{->} x) && \text{// rate } \mathtt{M}^+\\
op &= (\mathtt{T} \times \mathtt{V} \times \mathtt{V}\ x,\ \mathtt{D_P}\ y) \mathtt{->}\\
&\qquad (y.\mathtt{ts},\ y.\mathtt{val},\ \pi_3(x) + (\pi_2(x) + y.\mathtt{val})(y.\mathtt{ts} - \pi_1(x))\\
\mathtt{g_2} &: \mathtt{QRE}\langle \mathtt{D_P}, \mathtt{T} \times \mathtt{V} \times \mathtt{V}\rangle = \mathtt{iter}(\mathtt{apply}(\mathtt{f},\ x \mathtt{->} (x.\mathtt{ts}, x.\mathtt{val}, 0)),\ \mathtt{f},\ op) && \text{// rate } \mathtt{M}^+\\
op' &= (\mathtt{T}\ x,\ \mathtt{T} \times \mathtt{V} \times \mathtt{V}\ y) \mathtt{->}\\
&\qquad \mathtt{if}\ (\pi_1(y) - x > 0)\ \mathtt{then}\ \pi_3(y)/2(\pi_1(y) - x)\ \mathtt{else}\ \pi_2(y)\\
\mathtt{h} &= \mathtt{combine}(\mathtt{g_1},\ \mathtt{g_2},\ op') && \text{// rate } \mathtt{M}^+
\end{aligned}$$

The query $\mathtt{g_1}$ passes along the first timestamp, and the query $\mathtt{g_2}$ calculates the vector X_n. The top-level query $\mathtt{h}$ calculates the average of the piecewise-linear interpolant when $n \geq 2$, and returns the value v_1 when $n = 1$.

Quantitative concatenation. Suppose that we want to perform two streaming computations in sequence: first execute the query $\mathtt{f} : \mathtt{QRE}\langle D, A\rangle$, then the query $\mathtt{g} : \mathtt{QRE}\langle D, B\rangle$, and finally combine the two results using the operation $op : A \times B \to C$.



More specifically, we split the input stream into two parts $w = w_1 w_2$, process the first part w_1 according to $\mathtt{f}$ with output $\mathtt{f}(w_1)$, process the second part w_2 according to $\mathtt{g}$ with output $\mathtt{g}(w_2)$, and produce the final result $op(\mathtt{f}(w_1), \mathtt{g}(w_2))$ by applying op to the intermediate results.

The query $\mathtt{split}(\mathtt{f},\ \mathtt{g},\ op) : \mathtt{QRE}\langle D, C\rangle$ describes this computation. In order for this construction to be well-defined as a function, every input w that matches $\mathtt{split}(\mathtt{f},\ \mathtt{g},\ op)$ must be uniquely decomposable into $w = w_1 w_2$ with w_1 matching $\mathtt{f}$ and w_2 matching $\mathtt{g}$. In other words, the rates of $\mathtt{f}$ and $\mathtt{g}$ must be unambiguously concatenable.

The binary `split` construct extends naturally to more than two arguments. For example, the ternary version would be $\mathtt{split}(\mathtt{f},\ \mathtt{g},\ \mathtt{h}, (x,y,z) \mathtt{->} op(x,y,z))$.

Example 11. For the stream of monitored patients, we say that a measurement is *high-risk* if its value exceeds 50. The query $\mathtt{h} : \mathtt{QRE}\langle \mathtt{D_P}, \mathtt{V}\rangle$ below matches a sequence of measurements containing at least one high-risk measurement, and returns the maximum value after the last occurrence of a high-risk measurement.

$$\mathtt{f_1} : \mathtt{QRE}\langle \mathtt{D_P}, \mathtt{Ut}\rangle = \mathtt{atom}(x \mathtt{->} x.\mathtt{typ} = \mathtt{M},\ x \mathtt{->} \mathtt{def}) \qquad \text{// rate } \mathtt{M}$$

$$\mathtt{g_1} : \mathtt{QRE}\langle \mathtt{D_P}, \mathtt{Ut}\rangle = \mathtt{iter}(\mathtt{eps}(\mathtt{def}),\ \mathtt{f_1},\ (x,y) \mathtt{->} \mathtt{def}) \qquad \text{// rate } \mathtt{M}^*$$

$$\mathtt{f_2} : \mathtt{QRE}\langle \mathtt{D_P}, \mathtt{Ut}\rangle = \mathtt{atom}(x \mathtt{->} x.\mathtt{typ} = \mathtt{M}\ \mathtt{and}\ x.\mathtt{val} > 50,\ x \mathtt{->} \mathtt{def}) \qquad \text{// } \mathtt{M}(v > 50)$$

$$\mathtt{f_3} : \mathtt{QRE}\langle \mathtt{D_P}, \mathtt{V}\rangle = \mathtt{atom}(x \mathtt{->} x.\mathtt{typ} = \mathtt{M}\ \mathtt{and}\ x.\mathtt{val} \leq 50,\ x \mathtt{->} x.\mathtt{val}) \qquad \text{// } \mathtt{M}(v \leq 50)$$

$$\mathtt{g_3} : \mathtt{QRE}\langle \mathtt{D_P}, \mathtt{V}\rangle = \mathtt{iter}(\mathtt{eps}(-\infty),\ \mathtt{f_3},\ (x,y) \mathtt{->} \max(x,y)) \qquad \text{// } \mathtt{M}(v \leq 50)^*$$

$$\mathtt{h} : \mathtt{QRE}\langle \mathtt{D_P}, \mathtt{V}\rangle = \mathtt{split}(\mathtt{g_1},\ \mathtt{f_2},\ \mathtt{g_3},\ (x,y,z) \mathtt{->} z) \qquad \text{// } \mathtt{M}^* \cdot \mathtt{M}(v > 50) \cdot \mathtt{M}(v \leq 50)^*$$

The query $\mathtt{g_1}$ processes a sequence of measurements and returns nothing, the query $\mathtt{f_2}$ processes a single high-risk measurement and returns nothing, and the query $\mathtt{g_3}$ processes a sequence of non-high-risk measurements and returns the maximum value. The top-level query $\mathtt{h}$ executes $\mathtt{g_1}$, $\mathtt{f_2}$ and $\mathtt{g_3}$ in sequence and returns the output of $\mathtt{g_3}$.

Global choice. Given queries $\mathtt{f}$ and $\mathtt{g}$ of the same type with disjoint rates r and s, the query $\mathtt{or}(\mathtt{f},\mathtt{g})$ applies either $\mathtt{f}$ or $\mathtt{g}$ to the input stream depending on which one is defined. The rate of $\mathtt{or}(\mathtt{f},\mathtt{g})$ is the union $r \sqcup s$. This *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.

Example 12. For the stream of monitored patients, we describe a query that processes a nonempty sequence of episodes (hence, with rate $(\mathtt{B}\cdot\mathtt{M}^+\cdot\mathtt{E})^+$) and outputs at the end of each episode its summary. The summary of the episode is the average of all measurements if there are no high-risk measurements (high-risk: value exceeds 50), otherwise it is the average of only the high-risk measurements. We start with query g, which processes a nonempty sequence of non-high-risk measurements and returns the sum/count:

$$\begin{aligned}
\varphi &: \mathtt{D_P} \to \mathtt{Bool} = x\,\mathtt{->}\,x.\mathtt{typ} = \mathtt{M}\ \mathtt{and}\ x.\mathtt{val} \le 50 && \text{// predicate on } \mathtt{D_P}\\
\mathtt{f} &: \mathtt{QRE}\langle \mathtt{D_P}, \mathtt{V}\times\mathtt{V}\rangle = \mathtt{atom}(\varphi,\ x\,\mathtt{->}\,(x.\mathtt{val}, 1)) && \text{// rate } \mathtt{M}(v\le 50)\\
\mathtt{g} &: \mathtt{QRE}\langle \mathtt{D_P}, \mathtt{V}\times\mathtt{V}\rangle = \mathtt{iter}(\mathtt{f},\ \mathtt{f},\ (x,y)\,\mathtt{->}\,x+y) && \text{// rate } \mathtt{M}(v\le 50)^+
\end{aligned}$$

Similarly, the query h below processes a sequence with at least one high-risk measurement and returns the sum and count of the high-risk measurements. The idea for describing this computation comes from the following observation: the language over the alphabet $\Sigma = \{a,b\}$ that contains at least one occurrence of b is denoted by the ambiguous expression $(a+b)^*b(a+b)^*$, which is equivalent to the unambiguous $a^*(ba^*)^+$.

$$\begin{aligned}
\mathtt{f}_1^* &: \mathtt{QRE}\langle \mathtt{D_P}, \mathtt{Ut}\rangle = \mathtt{iter}(\mathtt{eps}(\mathtt{def}),\ \mathtt{atom}(\varphi),\ (x,y)\,\mathtt{->}\,\mathtt{def}) && \text{// rate } \mathtt{M}(v\le 50)^*\\
\psi &: \mathtt{D_P} \to \mathtt{Bool} = x\,\mathtt{->}\,x.\mathtt{typ} = \mathtt{M}\ \text{and}\ x.\mathtt{val} > 50 && \text{// predicate on } \mathtt{D_P}\\
\mathtt{f}_2 &: \mathtt{QRE}\langle \mathtt{D_P}, \mathtt{V}\times\mathtt{V}\rangle = \mathtt{atom}(\psi,\ x\,\mathtt{->}\,(x.\mathtt{val}, 1)) && \text{// } \mathtt{M}(v > 50)\\
\mathtt{g}' &: \mathtt{QRE}\langle \mathtt{D_P}, \mathtt{V}\times\mathtt{V}\rangle = \mathtt{split}(\mathtt{f}_2,\ \mathtt{f}_1^*,\ (x,y)\,\mathtt{->}\,x) && \text{// } r \triangleq \mathtt{M}(v>50)\cdot\mathtt{M}(v\le 50)^*\\
\mathtt{g}'' &: \mathtt{QRE}\langle \mathtt{D_P}, \mathtt{V}\times\mathtt{V}\rangle = \mathtt{iter}(\mathtt{g}',\ \mathtt{g}',\ (x,y)\,\mathtt{->}\,x+y) && \text{// rate } r^+\\
\mathtt{h} &: \mathtt{QRE}\langle \mathtt{D_P}, \mathtt{V}\times\mathtt{V}\rangle = \mathtt{split}(\mathtt{f}_1^*,\ \mathtt{g}'',\ (x,y)\,\mathtt{->}\,y) && \text{// rate } \mathtt{M}(v\le 50)^*\cdot r^+
\end{aligned}$$

We have written the queries g and h which process sequences of measurements differently based on the occurrence of high-risk measurements. The top-level query $\mathtt{m} : \mathtt{QRE}\langle \mathtt{D_P}, \mathtt{V}\rangle$ is then given below:

$$\begin{aligned}
\mathtt{k} &: \mathtt{QRE}\langle \mathtt{D_P}, \mathtt{V}\times\mathtt{V}\rangle = \mathtt{or}(\mathtt{g},\ \mathtt{h}) && \text{// rate } \mathtt{M}^+\\
\mathtt{f_B} &: \mathtt{QRE}\langle \mathtt{D_P}, \mathtt{D_P}\rangle = \mathtt{atom}(x\,\mathtt{->}\,x.\mathtt{typ} = \mathtt{B}) && \text{// rate B}\\
\mathtt{f_E} &: \mathtt{QRE}\langle \mathtt{D_P}, \mathtt{D_P}\rangle = \mathtt{atom}(x\,\mathtt{->}\,x.\mathtt{typ} = \mathtt{E}) && \text{// rate E}\\
\mathtt{ep} &: \mathtt{QRE}\langle \mathtt{D_P}, \mathtt{V}\rangle = \mathtt{split}(\mathtt{f_B},\ \mathtt{k},\ \mathtt{f_E},\ (x,y,z)\,\mathtt{->}\,\pi_1(y)/\pi_2(y)) && \text{// rate } \mathtt{B}\cdot\mathtt{M}^+\cdot\mathtt{E}\\
\mathtt{m} &: \mathtt{QRE}\langle \mathtt{D_P}, \mathtt{V}\rangle = \mathtt{iter}(\mathtt{ep},\ \mathtt{ep},\ (x,y)\,\mathtt{->}\,y) && \text{// rate } (\mathtt{B}\cdot\mathtt{M}^+\cdot\mathtt{E})^+
\end{aligned}$$

To see why the rate of $\mathtt{or}(\mathtt{g},\mathtt{h})$ is $\mathtt{M}^+$, it suffices to notice that the regular expressions $(a \sqcup b)^+$ and $a^+ \sqcup a^*(ba^*)^+$ are equivalent.

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

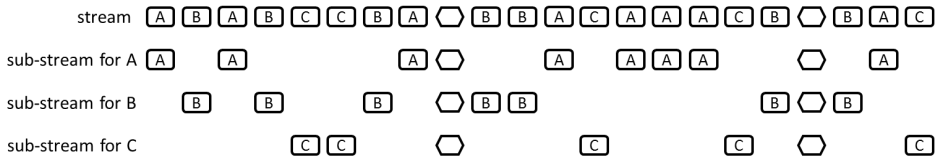


Figure 3. Partitioning a stream into several parallel sub-streams according to a key (letter in box).

identifier in this case, supply the corresponding projected sub-stream to a copy of `f`, one per key, and collect the set of resulting values.

In order to synchronize the per-key computations, we specify a predicate $\varphi_S : D \to \texttt{Bool}$ 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 $map : D \to K$ for the function that projects the key from an item (the value of *map* on synchronization items is irrelevant). For the patient input data type of Example 1 we choose: $\varphi_S = (x \texttt{->} x.\texttt{typ} = \texttt{D})$ and $K = \texttt{PID}$. 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 $T : \texttt{Map}\langle K, C\rangle$ 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 [25] and lends naturally to distributed processing.

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 $map(x) = k$. So, an item of D is sent to only one thread (as prescribed by the key), but an item satisfying φ_S is sent to all threads. See Figure 3 for an illustration of the partitioning into sub-streams. Each thread computes independently, and the synchronization elements are used for collecting the results of the threads. We specify a symbolic regular expression r over D, which *enforces* a rate of output for the overall computation. For example, if $r = (((\neg\texttt{D})^* \cdot \texttt{D})^2)^*$ then we intend to have output every other day. The rate should only specify sequences that end in a synchronization item.

Suppose $\texttt{f} : \texttt{QRE}\langle D, C\rangle$ 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

$$\texttt{map-collect}(\varphi_S,\ map,\ \texttt{f},\ r) : \texttt{QRE}\langle D, \texttt{Map}\langle K, C\rangle\rangle$$

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. The rate r should only depend on the occurrence of synchronization elements, so we demand that $r = s[(\neg\varphi_S)^*\varphi_S/\varphi_S]$ where the only predicate that $s : \texttt{RE}\langle D\rangle$ is allowed to contain is φ_S. We write $s[\psi/\varphi]$ to denote the result of replacing every occurrence of φ in s with ψ. For example, if $s = \texttt{D}^*$ (indicating output at every day marker) and $\varphi_S = \texttt{D}$,

then $r = s[(\neg\varphi_S)^*\varphi_S/\varphi_S] = ((\neg \mathtt{D})^*\mathtt{D})^*$. These restrictions do not affect expressiveness, but are useful for efficient evaluation.

Example 13. Suppose we want to output at the end of each day a table with summaries for the patients that had at least one episode within the day. Assuming that the data stream consists of items for a single patient, we first write the query that produces the episode summary for a single patient:

$$\begin{aligned}
\mathtt{f_M} &: \mathtt{QRE}\langle \mathtt{D_P}, \mathtt{V}\times\mathtt{V}\rangle = \mathtt{atom}(x \mathtt{->} x.\mathtt{typ} = \mathtt{M},\ x \mathtt{->} (x.\mathtt{val}, 1)) && \text{// rate } \mathtt{M}\\
\mathtt{g} &: \mathtt{QRE}\langle \mathtt{D_P}, \mathtt{V}\times\mathtt{V}\rangle = \mathtt{iter}(\mathtt{f},\ \mathtt{f},\ (x,y) \mathtt{->} x+y) && \text{// rate } \mathtt{M}^+\\
\mathtt{f_B} &: \mathtt{QRE}\langle \mathtt{D_P}, \mathtt{D_P}\rangle = \mathtt{atom}(x \mathtt{->} x.\mathtt{typ} = \mathtt{B}) && \text{// rate } \mathtt{B}\\
\mathtt{f_E} &: \mathtt{QRE}\langle \mathtt{D_P}, \mathtt{D_P}\rangle = \mathtt{atom}(x \mathtt{->} x.\mathtt{typ} = \mathtt{E}) && \text{// rate } \mathtt{E}\\
\mathtt{h} &: \mathtt{QRE}\langle \mathtt{D_P}, \mathtt{V}\rangle = \mathtt{split}(\mathtt{f_B},\ \mathtt{g},\ \mathtt{f_E},\ (x,y,z) \mathtt{->} \pi_1(y)/\pi_2(y)) && \text{// rate } \mathtt{B}\cdot\mathtt{M}^+\cdot\mathtt{E}
\end{aligned}$$

The query h, matches a full episode and returns the average of the measurements of the episode. The daily summary is then given by:

$$\begin{aligned}
\mathtt{k} &: \mathtt{QRE}\langle \mathtt{D_P}, \mathtt{V}\rangle = \mathtt{iter}(\mathtt{h},\ \mathtt{h},\ (x,y) \mathtt{->} \max(x,y)) && \text{// rate } (\mathtt{B}\cdot\mathtt{M}^+\cdot\mathtt{E})^+\\
\mathtt{l} &: \mathtt{QRE}\langle \mathtt{D_P}, \mathtt{V}\rangle = \mathtt{split}(\mathtt{k},\ \mathtt{atom}(x \mathtt{->} x.\mathtt{typ} = \mathtt{D}),\ (x,y) \mathtt{->} x) && \text{// rate } (\mathtt{B}\cdot\mathtt{M}^+\cdot\mathtt{E})^+\cdot\mathtt{D}
\end{aligned}$$

The query l matches days with at least one episode and outputs the maximum episode summary. The top-level query for the stream that concerns all patients is then:

$$\begin{aligned}
\mathtt{m} &= \mathtt{map\text{-}collect}(x \mathtt{->} x.\mathtt{typ} = \mathtt{D},\ x \mathtt{->} x.\mathtt{pId},\ \mathtt{l},\ (\neg\mathtt{D})^*\cdot\mathtt{D}) && \text{// rate } (\neg\mathtt{D})^*\cdot\mathtt{D}\\
\mathtt{n} &= \mathtt{iter}(\mathtt{m},\ \mathtt{m},\ (x,y) \mathtt{->} y) && \text{// rate } ((\neg\mathtt{D})^*\cdot\mathtt{D})^+
\end{aligned}$$

The synchronization items for the query $\mathtt{m} : \mathtt{QRE}\langle \mathtt{D_P}, \mathtt{Map}\langle \mathtt{PID}, \mathtt{V}\rangle\rangle$ are the day markers, and the keys are the patient identifiers. The rate of l satisfies the typing restrictions of the definition, because every sequence that matches l contains at least one keyed item and also matches the overall rate $(\neg\mathtt{D})^*\cdot\mathtt{D}$. So, the query m processes a single day and outputs the table of daily summaries for all patients that have had an episode in the day. Finally, the query n iterates m for every consecutive day.

Streaming composition. A natural operation for query languages over streaming data is streaming composition: given two streaming queries f and g, $\mathtt{f} \gg \mathtt{g}$ 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 R for queries $\mathtt{f} \gg \mathtt{g}$.

Example 14. Suppose the input stream concerns a single patient, and we want to compute at the end of each day the minimum and maximum measurement within the day. The first stage of the computation filters out the irrelevant B and E markers:

$$\texttt{f} : \texttt{QRE}\langle \texttt{D}_\texttt{P}, \texttt{D}_\texttt{P} \rangle = \texttt{atom}(x \texttt{->} x.\texttt{typ} = \texttt{M} \text{ or } x.\texttt{typ} = \texttt{D}) \qquad \text{// rate } \texttt{M} \sqcup \texttt{D}$$

$$\texttt{g} : \texttt{QRE}\langle \texttt{D}_\texttt{P}, \texttt{Ut} \rangle = \texttt{iter}(\texttt{eps}(\texttt{def}),\ \texttt{atom}(),\ (x,y) \texttt{->} \texttt{def}) \qquad \text{// rate } (\texttt{M} \sqcup \texttt{D} \sqcup \texttt{B} \sqcup \texttt{E})^*$$

$$\texttt{h} : \texttt{QRE}\langle \texttt{D}_\texttt{P}, \texttt{D}_\texttt{P} \rangle = \texttt{split}(\texttt{g},\ \texttt{f},\ (x,y) \texttt{->} y) \qquad \text{// rate } (\texttt{M} \sqcup \texttt{D} \sqcup \texttt{B} \sqcup \texttt{E})^* \cdot (\texttt{M} \sqcup \texttt{D})$$

The query `h` matches any sequence that ends in a measurement or day marker and returns the last item. Thus, `h` filters out `B` and `E` items. For the second stage of the computation, we assume that the stream consists of only `M` and `D` items.

$$\texttt{k} : \texttt{QRE}\langle \texttt{D}_\texttt{P}, \texttt{V} \rangle = \texttt{atom}(x \texttt{->} x.\texttt{typ} = \texttt{M},\ x \texttt{->} x.\texttt{val}) \qquad \text{// rate } \texttt{M}$$

$$\texttt{l} : \texttt{QRE}\langle \texttt{D}_\texttt{P}, \texttt{V} \rangle = \texttt{iter}(\texttt{eps}(-\infty),\ \texttt{k},\ (x,y) \texttt{->} \max(x,y)) \qquad \text{// rate } \texttt{M}^*$$

$$\texttt{m} : \texttt{QRE}\langle \texttt{D}_\texttt{P}, \texttt{V} \rangle = \texttt{split}(\texttt{l},\ \texttt{atom}(x \texttt{->} x.\texttt{typ} = \texttt{D}),\ (x,y) \texttt{->} x) \qquad \text{// rate } \texttt{M}^* \cdot \texttt{D}$$

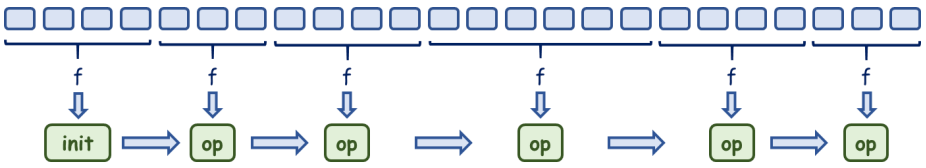
$$\texttt{n} : \texttt{QRE}\langle \texttt{D}_\texttt{P}, \texttt{V} \rangle = \texttt{iter}(\texttt{m},\ \texttt{m},\ (x,y) \texttt{->} y) \qquad \text{// rate } (\texttt{M}^* \cdot \texttt{D})^+$$

The top-level query is then the pipeline $\texttt{h} \gg \texttt{n}$ and its domain is $((\texttt{M} \sqcup \texttt{B} \sqcup \texttt{E})^* \cdot \texttt{D})^+$.

2. Common Patterns

The core language of Figure 2 is expressive enough to describe many common stream transformations. We present below several derived patterns, including stream filtering, stream mapping, and aggregation over windows.

Iteration at least once. Let $\texttt{f} : \texttt{QRE}\langle D, A \rangle$ be a query with output type A, $init : A \to B$ be the initialization function, and $op : B \times A \to B$ be the aggregation function. The query $\texttt{iter}_1(\texttt{f},\ init,\ op)$, 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 $a_0\ a_1\ a_2 \ldots a_n$, i.e. $a_i = \texttt{f}(w_i)$, and combines the results $a_1\ a_2 \ldots a_n$ using the list iterator *left fold* with start value $init(a_0) \in B$ and accumulation operation $op : B \times A \to B$.



The construct $\texttt{iter}_1$ can be encoded using `iter` as follows:

$$\texttt{iter}_1(\texttt{f},\ init,\ op) \triangleq \texttt{iter}(\texttt{apply}(\texttt{f},\ init),\ \texttt{f},\ op).$$

The type of $\texttt{iter}_1(\texttt{f},\ init,\ op)$ is $\texttt{QRE}\langle D, B \rangle$ and its rate is $\texttt{R}(\texttt{f})^+$. We use the abbreviation $\texttt{iter}_1(\texttt{f}, op)$ for the common case where $A = B$ and $init$ is the identity function.

Matching without output. Suppose r is an unambiguous symbolic regex over the data item type D. The query $\texttt{match}(r)$, whose rate is equal to r, does not produce any output when it matches. This is essentially the same as producing `def` as output for a match. The `match` construct can be encoded as follows:

$$\begin{aligned}
\texttt{match}(\varphi) &\triangleq \texttt{atom}(\varphi,\ x \texttt{->} \texttt{def}) \\
\texttt{match}(r_1 \sqcup r_2) &\triangleq \texttt{or}(\texttt{match}(r_1),\ \texttt{match}(r_2)) \\
\texttt{match}(r_1 \cdot r_2) &\triangleq \texttt{split}(\texttt{match}(r_1),\ \texttt{match}(r_2),\ (x,y) \texttt{->} \texttt{def}) \\
\texttt{match}(r^*) &\triangleq \texttt{iter}(\texttt{eps}(\texttt{def}),\ \texttt{match}(r),\ (x,y) \texttt{->} \texttt{def})
\end{aligned}$$

An easy induction establishes that $\mathsf{R}(\texttt{match}(r)) = r$.

Stream filtering. Let φ be a predicate over the type of input data items D. We want to describe the streaming transformation that filters out all items that do not satisfy φ. We implement this with the query $\texttt{filter}(\varphi)$, which matches all stream prefixes that end with an item satisfying φ.

$$\texttt{filter}(\varphi) \triangleq \texttt{split}(\texttt{match}(\mathit{true}_D^*),\ \texttt{atom}(\varphi),\ (x,y) \texttt{->} y)$$

The type of $\texttt{filter}(\varphi)$ is $\texttt{QRE}\langle D, D\rangle$ and its rate is $\mathit{true}_D^* \cdot \varphi$.

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

$$\texttt{map}(op) \triangleq \texttt{split}(\texttt{match}(\mathit{true}_D^*),\ \texttt{atom}(\mathit{true}_D,\ op),\ (x,y) \texttt{->} y).$$

Its type is $\texttt{QRE}\langle D, C\rangle$ and its rate is $\mathit{true}_D^* \cdot \mathit{true}_D = \mathit{true}_D^+$.

Example 15. Using filtering, mapping and streaming composition we can implement the average of a sequence of scalars with a very common idiom:

$$\begin{aligned}
\texttt{f} &: \texttt{QRE}\langle \texttt{V}, \texttt{V} \times \texttt{V}\rangle = \texttt{map}(x \texttt{->} (x, 1)) && \text{// rate } \texttt{V}^+ \\
\texttt{g} &: \texttt{QRE}\langle \texttt{V} \times \texttt{V}, \texttt{V} \times \texttt{V}\rangle = \texttt{iter}_1(\texttt{atom}(),\ (x,y) \texttt{->} x + y) && \text{// rate } (\texttt{V} \times \texttt{V})^+ \\
\texttt{h} &: \texttt{QRE}\langle \texttt{V} \times \texttt{V}, \texttt{V}\rangle = \texttt{map}(x \texttt{->} \pi_1(x)/\pi_2(x)) && \text{// rate } (\texttt{V} \times \texttt{V})^+
\end{aligned}$$

and the top-level query is the pipeline $\texttt{f} \gg \texttt{g} \gg \texttt{h}$, whose type is $\texttt{QRE}\langle \texttt{V}, \texttt{V}\rangle$.

Iteration exactly n times. Let $n \geq 1$ and $\texttt{f} : \texttt{QRE}\langle D, A\rangle$ be a query to iterate exactly n times. The aggregation is specified by the initialization function $\mathit{init} : A \to B$ (for the first value) and the aggregation function $op : B \times A \to B$. The construct $\texttt{iter}^\texttt{n}$ describes iteration (and aggregation) exactly n times, and can be encoded as follows:

$$\begin{aligned}
\texttt{iter}^\texttt{1}(\texttt{f},\ \mathit{init},\ op) &\triangleq \texttt{apply}(\texttt{f},\ \mathit{init}) \\
\texttt{iter}^{\texttt{n}+\texttt{1}}(\texttt{f},\ \mathit{init},\ op) &\triangleq \texttt{split}(\texttt{iter}^\texttt{n}(\texttt{f},\ \mathit{init},\ op),\ \texttt{f},\ op)
\end{aligned}$$

The type of $\texttt{iter}^\texttt{n}(\texttt{f},\ \mathit{init},\ op)$ is $\texttt{QRE}\langle D, B\rangle$ and its rate it $\mathsf{R}(\texttt{f})^n$ (n-fold concatenation).

Pattern-based tumbling windows. The term tumbling windows is used to describe the splitting of the stream into contiguous non-overlapping regions [6]. Suppose we want to describe the streaming function that iterates `f` at least once and reports the result given by `f` at every match. The following query expresses this behavior:

$$\texttt{iter-last}(\texttt{f}) \triangleq \texttt{iter}(\texttt{f},\ \texttt{f},\ (x,y)\ \texttt{->}\ y).$$

The rate of $\texttt{iter-last}(\texttt{f})$ is equal to $\mathsf{R}(\texttt{f})^+$.

Example 16. Suppose that the query $\texttt{f} : \texttt{QRE}\langle \texttt{D}_\texttt{P}, \texttt{V}\rangle$ has rate $r = (\texttt{B}\cdot\texttt{M}^+\cdot\texttt{E})^*\cdot\texttt{D}$ and computes the daily summary for a single patient. Then, the query $\texttt{iter-last}(\texttt{f})$ has rate r^+ and computes the daily summary at the end of every day. Finally, the query

$$\texttt{iter-last}(\texttt{f}) \gg \texttt{iter}_1(\texttt{atom}(),\ (x,y)\ \texttt{->}\ \max(x,y))$$

computes at the end of every day the maximum daily summary so far. Notice that this query can be equivalently expressed without $\gg$ as $\texttt{iter}_1(\texttt{f},\ (x,y)\ \texttt{->}\ \max(x,y))$.

Sliding windows (slide by pattern). To express a policy such as "output the statistical summary of events in the past ten hours every five minutes" existing relational query languages provide an explicit sliding window primitive [6]. We can support this primitive, which can be compiled into the base language by massaging the input data stream with the introduction of suitable tags (marking five-minute time intervals in this example). The insertion of the tags then allows to express both the window and the sliding using very general regular patterns. Let $n \geq 1$ be the size of the window, and $\texttt{f} : \texttt{QRE}\langle D, A\rangle$ be the query that processes a unit pattern. The aggregation over the window is specified by the function $\mathit{init} : A \to B$ for initialization and the aggregation function $\mathit{op} : B \times A \to B$. We give a query that computes the aggregation over the last n units of the stream (or over all units if the stream has less than n units):

$$\texttt{g} = \texttt{or}(\texttt{iter}^1(\texttt{f},\ \mathit{init},\ \mathit{op}),\ \ldots,\ \texttt{iter}^{\texttt{n}-1}(\texttt{f},\ \mathit{init},\ \mathit{op}))$$
$$\texttt{h} = \texttt{split}(\texttt{match}(\mathsf{R}(\texttt{f})^*),\ \texttt{iter}^{\texttt{n}}(\texttt{f},\ \mathit{init},\ \mathit{op}),\ (x,y)\ \texttt{->}\ y)$$

and $\texttt{wnd}(\texttt{f},\ n,\ \mathit{init},\ \mathit{op}) = \texttt{or}(\texttt{g},\ \texttt{h})$ with rate $\mathsf{R}(\texttt{f})^+$.

3. The Yahoo Streaming Benchmark

The Yahoo Benchmark [14] specifies a stream of advertisement-related events for an analytics pipeline. It specifies a set of campaigns and a set of advertisements, where each ad belongs to exactly one campaign. The static map from ads to campaigns is computed ahead-of-time and stored in memory. Each element of the data stream is of the form

$$(\texttt{userId},\ \texttt{pageId},\ \texttt{adId},\ \texttt{eventType},\ \texttt{eventTime}),$$

indicating the interaction of a user with an advertisement, where `eventType` is one of $\{\texttt{view}, \texttt{click}, \texttt{purchase}\}$. The component `eventTime` is the timestamp of the event.

3.1. First Yahoo query

The basic benchmark query (described in [14]) computes, at the end of every second, a map from each campaign to the number of views associated with that campaign within the last second. For each event tuple, this involves a lookup to determine the campaign associated with the advertisement viewed. The reference implementation published with the Yahoo benchmark involves 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 involes key-based partitioning on only one property, namely the derived campaign id of the event. We present three ways of expressing this query in StreamQRE. We assume w.l.o.g. that the stream also contains events `S` that serve as end-of-second markers.

Implementation (I) We reproduce faithfully the reference implementation of the Yahoo benchmark [14] by constructing the following multi-stage pipeline:

(1) query $\mathtt{g}_1$: filter view and end-of-second events,
(2) query $\mathtt{g}_2$: project the ad id from each view tuple,
(3) query $\mathtt{g}_3$: lookup the campaign id of each advertisement,
(4) query $\mathtt{g}_7$: compute for every one-second window the number of events (views) associated with each campaign.

We write `ADID` for the type of ad identifiers, `CID` for the type of campaign identifiers, and $\mathtt{D_Y}$ for the data type of the input events.

$$\mathtt{g}_1 : \mathtt{QRE}\langle \mathtt{D_Y}, \mathtt{D_Y} \rangle = \mathtt{filter}(x \mathtt{->} x = \mathtt{S}\ \mathtt{or}\ x.\mathtt{isView})$$

$$\mathtt{g}_2 : \mathtt{QRE}\langle \mathtt{D_Y}, \mathtt{ADID} \cup \{\mathtt{S}\} \rangle = \mathtt{map}(x \mathtt{->} \mathtt{if}\ (x = \mathtt{S})\ \mathtt{then\ S\ else}\ x.\mathtt{adId})$$

$$\mathtt{g}_3 : \mathtt{QRE}\langle \mathtt{ADID} \cup \{\mathtt{S}\}, \mathtt{CID} \cup \{\mathtt{S}\} \rangle = \mathtt{map}(x \mathtt{->} \mathtt{if}\ (x = \mathtt{S})\ \mathtt{then\ S\ else\ lookup}(x.\mathtt{adId}))$$

$$\mathtt{g}_4 : \mathtt{QRE}\langle \mathtt{CID} \cup \{\mathtt{S}\}, \mathtt{Nat} \rangle = \mathtt{iter}(\mathtt{eps}(0),\ \mathtt{atom}(x \mathtt{->} x \neq \mathtt{S}),\ (x,y) \mathtt{->} x+1)$$

$$\mathtt{g}_5 : \mathtt{QRE}\langle \mathtt{CID} \cup \{\mathtt{S}\}, \mathtt{Nat} \rangle = \mathtt{split}(\mathtt{g}_4,\ \mathtt{atom}(x \mathtt{->} x = \mathtt{S}),\ (x,y) \mathtt{->} x)$$

$$\mathtt{g}_6 : \mathtt{QRE}\langle \mathtt{CID} \cup \{\mathtt{S}\}, \mathtt{Nat} \rangle = \mathtt{map\text{-}collect}(x \mathtt{->} x = \mathtt{S},\ x \mathtt{->} x.\mathtt{cId},\ \mathtt{g}_5,\ (\neg \mathtt{S})^* \cdot \mathtt{S})$$

$$\mathtt{g}_7 : \mathtt{QRE}\langle \mathtt{CID} \cup \{\mathtt{S}\}, \mathtt{Nat} \rangle = \mathtt{iter\text{-}last}(\mathtt{g}_6)$$

The function $\mathtt{lookup} : \mathtt{ADID} \to \mathtt{CID}$ models the mapping of an ad to the campaign it belongs to, and $x.\mathtt{isView}$ abbreviates the boolean expression $(x.\mathtt{eventType} = \mathtt{view})$. The auxiliary query $\mathtt{g}_5$, with rate $(\neg \mathtt{S})^* \cdot \mathtt{S}$, calculates the length of event sequences of a single campaign for one second. With $\mathtt{g}_7$ we perform (every second) key-based partitioning based on the campaign id `cId`. The top-level query to compute the number of views for each campaign per second is the pipeline

$$\mathtt{g}_8 = \mathtt{g}_1 \gg \mathtt{g}_2 \gg \mathtt{g}_3 \gg \mathtt{g}_7.$$

Since the domain of $\mathtt{g}_1 \gg \mathtt{g}_2 \gg \mathtt{g}_3$ is $\mathtt{D}_\mathtt{Y}^* \cdot (x \texttt{->} x.\mathtt{isView}\ \mathtt{or}\ x = \mathtt{S})$, and $\mathtt{g}_7$ reports at every S marker, the domain of $\mathtt{g}_8$ is $\mathtt{D}_\mathtt{Y}^* \cdot \mathtt{S}$ (output at the end of every second).

Implementation (II) The stages $\mathtt{g}_1$, $\mathtt{g}_2$ and $\mathtt{g}_3$ of the pipeline of implementation (I) can be collapsed into a single query $\mathtt{h}_3$ as follows:

$$\mathtt{h}_1 : \mathtt{QRE}\langle \mathtt{D}_\mathtt{Y}, \mathtt{CID} \cup \{\mathtt{S}\}\rangle = \mathtt{atom}(x \texttt{->} x.\mathtt{isView},\ x \texttt{->} \mathtt{lookup}(x.\mathtt{adId}))$$

$$\mathtt{h}_2 : \mathtt{QRE}\langle \mathtt{D}_\mathtt{Y}, \mathtt{CID} \cup \{\mathtt{S}\}\rangle = \mathtt{or}(\mathtt{h}_1,\ \mathtt{atom}(x \texttt{->} x = \mathtt{S},\ x \texttt{->} \mathtt{S}))$$

$$\mathtt{h}_3 : \mathtt{QRE}\langle \mathtt{D}_\mathtt{Y}, \mathtt{CID} \cup \{\mathtt{S}\}\rangle = \mathtt{split}(\mathtt{match}(((\mathtt{D}_\mathtt{Y}\ x) \texttt{->} \mathit{true})^*),\ \mathtt{h}_2,\ (x,y) \texttt{->} y)$$

The benchmark query can now be written as a two-stage pipeline $\mathtt{h}_4 = \mathtt{h}_3 \gg \mathtt{g}_7$.

Implementation (III) The previous implementation uses the streaming composition operator to simplify the map-collect part of the query. The channel handling the events of each campaign assumes that all incoming events correspond to views, and therefore simply counts the number of tuples flowing in. We can eliminate the streaming composition by inspecting the event type of each incoming tuple during per-campaign processing:

$$\mathtt{k}_1 : \mathtt{QRE}\langle \mathtt{D}_\mathtt{Y}, \mathtt{Nat}\rangle = \mathtt{atom}(x \texttt{->} x.\mathtt{isView},\ x \texttt{->} 1) \qquad \text{// rate } \mathtt{V} \text{ (view)}$$

$$\mathtt{k}_2 : \mathtt{QRE}\langle \mathtt{D}_\mathtt{Y}, \mathtt{Nat}\rangle = \mathtt{atom}(x \texttt{->} \text{not } (x.\mathtt{isView} \text{ or } x = \mathtt{S}),\ x \texttt{->} 0) \qquad \text{// rate } \neg(\mathtt{V} \sqcup \mathtt{S})$$

$$\mathtt{k}_3 : \mathtt{QRE}\langle \mathtt{D}_\mathtt{Y}, \mathtt{Nat}\rangle = \mathtt{or}(\mathtt{k}_1,\ \mathtt{k}_2) \qquad \text{// rate } \neg\mathtt{S}$$

$$\mathtt{k}_4 : \mathtt{QRE}\langle \mathtt{D}_\mathtt{Y}, \mathtt{Nat}\rangle = \mathtt{iter}(\mathtt{eps}(0),\ \mathtt{k}_3,\ (x,y) \texttt{->} x + y) \qquad \text{// rate } (\neg\mathtt{S})^*$$

$$\mathtt{k}_5 : \mathtt{QRE}\langle \mathtt{D}_\mathtt{Y}, \mathtt{Nat}\rangle = \mathtt{split}(\mathtt{k}_4,\ \mathtt{atom}(x \texttt{->} x = \mathtt{S}),\ (x,y) \texttt{->} x) \qquad \text{// rate } (\neg\mathtt{S})^* \cdot \mathtt{S}$$

So, $\mathtt{k}_5$ counts the number of views in a stream of the form $(\neg\mathtt{S})^* \cdot \mathtt{S}$. We now have a third way of representing the benchmark query:

$$\mathtt{k}_6 : \mathtt{QRE}\langle \mathtt{D}_\mathtt{Y}, \mathtt{Map}\langle \mathtt{CID}, \mathtt{Nat}\rangle\rangle = \mathtt{map\text{-}collect}(x \texttt{->} x = \mathtt{S},\ x \texttt{->} \mathtt{lookup}(x.\mathtt{adId}),\ \mathtt{k}_5,\ (\neg\mathtt{S})^* \cdot \mathtt{S})$$

$$\mathtt{k}_7 : \mathtt{QRE}\langle \mathtt{D}_\mathtt{Y}, \mathtt{Map}\langle \mathtt{CID}, \mathtt{Nat}\rangle\rangle = \mathtt{iter\text{-}last}(\mathtt{k}_6).$$

In implementations (I) and (II) the queries only looked up the campaign ids for view events, while query $\mathtt{k}_7$ computes the campaign id for *each* incoming tuple. It therefore makes more campaign id lookups.

3.2. Second Yahoo query

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.

Assume that we have fixed a specific ad and a specific user, and the stream consists only of events for these. The pattern for *success*, given by the regular expression

$$r = (\neg \mathtt{V})^* \cdot \mathtt{V} \cdot (\neg \mathtt{P})^* \cdot \mathtt{P} \cdot \mathtt{D}_\mathtt{Y}^* \cdot \mathtt{S},$$

indicates that the user purchases the product after seeing the ad for it. For simplicity, we have written `V` to denote the occurrence of view events, and `P` for purchase events. The trailing pattern $\mathtt{D}_\mathtt{Y}^* \cdot \mathtt{S}$ is used for matching until the end of an end-of-second marker.

Now, suppose that we have fixed a specific ad, and we want to compute its *score*: the number of users that have purchased the product after viewing the ad.

$$\begin{aligned} \mathtt{g}_1 : \mathtt{QRE}\langle \mathtt{D}_\mathtt{Y}, \mathtt{Map}\langle \mathtt{UID}, \mathtt{Ut} \rangle \rangle &= \mathtt{map\text{-}collect}(x \mathtt{->} x = \mathtt{S}, \\ &\qquad x \mathtt{->} x.\mathtt{userId},\ \mathtt{match}(r),\ \mathtt{D}_\mathtt{Y}^* \cdot \mathtt{S}) \\ \mathtt{g}_2 : \mathtt{QRE}\langle \mathtt{D}_\mathtt{Y}, \mathtt{Nat} \rangle &= \mathtt{apply}(\mathtt{g}_1,\ x \mathtt{->} x.\mathtt{size}) \end{aligned}$$

where `UID` is the set of user identifiers, and $\mathtt{size} : \mathtt{Map}\langle K, C \rangle \to \mathtt{Nat}$ returns the size of a map data structure (number of keys that are mapped to some value). The rate of $\mathtt{g}_2$ is $\mathtt{D}_\mathtt{Y}^* \cdot \mathtt{S}$, that is, $\mathtt{g}_2$ produces at the end of every second the success score of the ad. Moreover, we observe that the specified rate $\mathtt{D}_\mathtt{Y}^* \cdot \mathtt{S}$ is equivalent to $((\neg \mathtt{S})^* \mathtt{S})^+$.

Given the event stream of a single campaign, we can divide it into sub-streams of individual ads, compute the success score for each one of them, and thus determine the most successful ad of the campaign:

$$\begin{aligned} \mathtt{g}_3 : \mathtt{QRE}\langle \mathtt{D}_\mathtt{Y}, \mathtt{Map}\langle \mathtt{ADID}, \mathtt{Nat} \rangle \rangle &= \mathtt{map\text{-}collect}(x \mathtt{->} x = \mathtt{S},\ x \mathtt{->} x.\mathtt{adId},\ \mathtt{g}_2,\ \mathtt{D}_\mathtt{Y}^* \cdot \mathtt{S}) \\ \mathtt{g}_4 : \mathtt{QRE}\langle \mathtt{D}_\mathtt{Y}, \mathtt{ADID} \times \mathtt{Nat} \rangle &= \mathtt{apply}(\mathtt{g}_3,\ x \mathtt{->} x.\mathrm{argmax}), \end{aligned}$$

where $\mathrm{argmax} : \mathtt{Map}\langle K, C \rangle \to K \times C$ (for a type C that is linearly ordered) calculates the key-value pair (k, c) that has the maximum value. In query $\mathtt{g}_4$, it calculates the pair (`adId`, `score`) for the ad with the maximum score. Finally, we use the `map-collect` construct to map each campaign id to the most successful ad:

$$\mathtt{g}_5 : \mathtt{QRE}\langle \mathtt{D}_\mathtt{Y}, \mathtt{Map}\langle \mathtt{CID}, \mathtt{ADID} \times \mathtt{Nat} \rangle \rangle = \mathtt{map\text{-}collect}(x \mathtt{->} x = \mathtt{S},\ x \mathtt{->} x.\mathtt{cId},\ \mathtt{g}_4,\ \mathtt{D}_\mathtt{Y}^* \cdot \mathtt{S})$$

The rates of $\mathtt{g}_1$, $\mathtt{g}_2$, $\mathtt{g}_3$, $\mathtt{g}_4$ and $\mathtt{g}_5$ are all equal to $\mathtt{D}_\mathtt{Y}^* \cdot \mathtt{S}$. They produce output at every `S` marker occurrence in the stream.

4. The NEXMark Streaming Benchmark

The Niagara Extension to XMark benchmark (NEXMark) [15] 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.

$\texttt{Person}(\texttt{personId}, \texttt{name}, \texttt{ts})$

$\texttt{Item}(\texttt{itemId}, \texttt{sellerId}, \texttt{initPrice}, \texttt{ts}, \texttt{dur}, \texttt{category})$

$\texttt{Close}(\texttt{itemId}, \texttt{ts})$

$\texttt{Bid}(\texttt{itemId}, \texttt{bidderId}, \texttt{bidIncrement}, \texttt{ts})$

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 will describe two out of the five queries that are considered in [2], and which are minor variants of some of the queries of the NEXMark benchmark:

Query. Calculate the number of currently open auctions. The output should be updated at every auction start and close.

$$\begin{aligned}
\texttt{f} &: \texttt{QRE}\langle \texttt{D}_\texttt{N}, \texttt{D}_\texttt{N} \rangle = \texttt{filter}(x \texttt{->} x.\texttt{isItem or } x.\texttt{isClose}) \\
\texttt{g} &: \texttt{QRE}\langle \texttt{D}_\texttt{N}, \texttt{Nat} \rangle = \texttt{map}(x \texttt{->} \texttt{if } (x.\texttt{isItem}) \texttt{ then } +1 \texttt{ else } -1) \\
\texttt{h} &: \texttt{QRE}\langle \texttt{Nat}, \texttt{Nat} \rangle = \texttt{iter}(\texttt{eps}(0),\ \texttt{atom}(),\ (x,y) \texttt{->} x+y) \\
\texttt{k} &: \texttt{QRE}\langle \texttt{D}_\texttt{N}, \texttt{Nat} \rangle = \texttt{f} \gg \texttt{g} \gg \texttt{h}
\end{aligned}$$

We write $\texttt{D}_\texttt{N}$ for the input data type of the NEXMark benchmark.

Query. Find the item with the most bids in the last 24 hours. The output should be updated every minute. We assume that the stream has end-of-minute markers `M`.

First, suppose that we process a stream consisting only of bid events that ends with an end-of-minute marker. We compute a bid count for every item that appears:

$$\begin{aligned}
\texttt{f} &: \texttt{QRE}\langle \texttt{D}_\texttt{N}, \texttt{Nat} \rangle = \texttt{iter}(\texttt{eps}(0),\ \texttt{atom}(x \texttt{->} x.\texttt{isBid}),\ (x,y) \texttt{->} x+1) \text{ // } \texttt{B}^* \\
\texttt{g} &: \texttt{QRE}\langle \texttt{D}_\texttt{N}, \texttt{Nat} \rangle = \texttt{split}(\texttt{f},\ \texttt{atom}(x \texttt{->} x = \texttt{M}),\ (x,y) \texttt{->} x) \quad \text{// rate } \texttt{B}^* \cdot \texttt{M} \\
\texttt{h} &: \texttt{QRE}\langle \texttt{D}_\texttt{N}, \texttt{Map}\langle \texttt{IID}, \texttt{Nat} \rangle \rangle = \texttt{map-collect}(x \texttt{->} x = \texttt{M}, \\
&\qquad\qquad x \texttt{->} x.\texttt{itemId},\ \texttt{g},\ (\neg\texttt{M})^* \cdot \texttt{M}) \quad \text{// rate } (\neg\texttt{M})^* \cdot \texttt{M} \\
\texttt{k} &: \texttt{QRE}\langle \texttt{D}_\texttt{N}, \texttt{MSet}\langle \texttt{IID} \rangle \rangle = \texttt{apply}(\texttt{h},\ x \texttt{->} x.\texttt{toMSet}()) \quad \text{// rate } (\neg\texttt{M})^* \cdot \texttt{M}
\end{aligned}$$

The type of item identifiers is `IID`, and $\texttt{toMSet} : \texttt{Map}\langle K, \texttt{Nat} \rangle \to \texttt{MSet}\langle K \rangle$ is an operation that turns a map object into a multiset. Now, we write the top-level query:

$$\begin{aligned}
\texttt{l} &: \texttt{QRE}\langle \texttt{D}_\texttt{N}, \texttt{MSet}\langle \texttt{IID} \rangle \rangle = \texttt{wnd}(\texttt{k},\ 24 \cdot 60,\ x \texttt{->} x,\ (x,y) \texttt{->} x \uplus y) \quad \text{// rate } ((\neg\texttt{M})^* \cdot \texttt{M})^+ \\
\texttt{m} &: \texttt{QRE}\langle \texttt{D}_\texttt{N}, \texttt{D}_\texttt{N} \rangle = \texttt{filter}(x \texttt{->} x.\texttt{isBid or } (x = \texttt{M})) \quad \text{// rate } \texttt{D}_\texttt{N}^* \cdot (\texttt{B} \sqcup \texttt{M}) \\
\texttt{n} &: \texttt{QRE}\langle \texttt{D}_\texttt{N}, \texttt{IID} \times \texttt{Nat} \rangle = \texttt{m} \gg \texttt{apply}(\texttt{l},\ x \texttt{->} x.\,\text{argmax})
\end{aligned}$$

We write $\uplus$ to denote multiset union. The function $\text{argmax} : \texttt{MSet}\langle C \rangle \to C \times \texttt{Nat}$ returns the member of the multiset with the highest count.

```
// Process a single measurement: rate M
QRe<DPatients, DPatients> meas =
    Q.atomic(x -> x.isMeasurement(), x -> x);

// Sum of sequence of measurements: rate M*
QRe<DPatients, Double> sum =
    Q.iter(Q.eps(0.0), meas, (x,y) -> x+y.getValue());

// Length of sequence of measurements: rate M*
QRe<DPatients, Integer> count =
    Q.iter(Q.eps(0), meas, (x,y) -> x+1);

// Average of sequence of measurements: rate M*
QRe<DPatients, Double> measAvg =
    Q.combine(sum, count, (x,y) -> x/y);

Iterator<DPatients> stream = ... // input stream

// evaluator for the query
Eval<DPatients, Double> e = measAvg.getEval();

// execution loop
Double output = e.start(); // returns null, if undefined
while (stream.hasNext()) {
    DPatients d = stream.next();
    output = e.next(d); // returns null, if undefined
}
```

Figure 4. Computing the average of a nonempty sequence of measurements.

5. The StreamQRE Library in Java

StreamQRE has been implemented as a Java library [13] in order to facilitate the easy integration with user-defined types and operations. The implementation covers all the core constructs of Figure 2, and also provides optimizations for the derived constructs of Section 2 (stream filtering, stream mapping, sliding windows, etc.).

Figure 4 gives a simple example that illustrates how to program with the StreamQRE Java library. The query `measAvg` describes the computation of the average of a sequence of measurements (for the patient data stream). The method `getEval`, which stands for "get evaluator", is used to obtain an object that encapsulates the evaluation algorithm for the query. On this evaluator object, the methods `start` and `next` are used to initialize the algorithm and consume data items respectively.

6. Conclusion

We have given an introduction to the StreamQRE language [2], a high-level formalism for processing streaming data. The query language integrates two paradigms for pro-

gramming with streams: streaming relational languages with windowing constructs, and state-machine-based models for pattern-matching and performing sequence-aware computations. The language consists of 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 query of the StreamQRE language can be compiled into a streaming algorithm with strong efficiency guarantees [2], both for space usage and processing time per element. An experimental evaluation of StreamQRE is reported in [2], which shows that the StreamQRE implementation is competitive with popular streaming engines such as RxJava [26], Esper [27], and Flink [28].

Acknowledgements

We thank our collaborators Zachary Ives, Sanjeev Khanna and Mukund Raghothaman. This research was supported by NSF Expeditions award CCF 1138996.

References

[1] R. Alur, E. Berger, A. Drobnis, L. Fix, K. Fu, G. Hager, D. Lopresti, K. Nahrstedt, E. Mynatt, S. Patel, J. Rexford, J. Stankovic, and B. Zorn. Systems computing challenges in the Internet of Things. In *Computing Community Consortium Whitepaper*, 2016.

[2] Konstantinos Mamouras, Mukund Raghothaman, Rajeev Alur, Zachary G. Ives, and Sanjeev Khanna. StreamQRE: Modular specification and efficient evaluation of quantitative queries over streaming data. 2017. manuscript.

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

[4] Shivnath Babu and Jennifer Widom. Continuous queries over data streams. *ACM Sigmod Record*, 30(3):109–120, 2001.

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

[6] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A. Tucker. Semantics and evaluation techniques for window aggregates in data streams. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD '05, pages 311–322. ACM, 2005.

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

[8] Barzan Mozafari, Kai Zeng, and Carlo Zaniolo. High-performance complex event processing over XML streams. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 253–264. ACM, 2012.

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

[10] M. Hirzel, H. Andrade, B. Gedik, G. Jacques-Silva, R. Khandekar, V. Kumar, M. Mendell, H. Nasgaard, S. Schneider, R. Soul, and K. L. Wu. IBM Streams Processing Language: Analyzing big data in motion. *IBM Journal of Research and Development*, 57(3/4):7:1–7:11, 2013.

[11] Mandana Vaziri, Olivier Tardieu, Rodric Rabbah, Philippe Suter, and Martin Hirzel. Stream processing with a spreadsheet. In *Proceedings of the 28th European Conference on Object-Oriented Programming (ECOOP '14)*, pages 360–384. Springer Berlin Heidelberg, 2014.

[12] Margus Veanes, Pieter Hooimeijer, Benjamin Livshits, David Molnar, and Nikolaj Bjorner. Symbolic finite state transducers: Algorithms and applications. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '12)*, pages 137–150. ACM, 2012.

[13] StreamQRE library. `http://www.seas.upenn.edu/~mamouras/StreamQRE/`.

[14] Sanket Chintapalli, Derek Dagit, Bobby Evans, Reza Farivar, Thomas Graves, Mark Holderbaugh, Zhuo Liu, Kyle Nusbaum, Kishorkumar Patil, Boyang Jerry Peng, and Paul Poulosky. Benchmarking streaming computation engines: Storm, Flink and Spark streaming. In *First Annual Workshop on Emerging Parallel and Distributed Runtime Systems and Middleware*, 2016.

[15] Pete Tucker, Kristin Tufte, Vassilis Papadimos, and David Maier. NEXMark: A benchmark for queries over data streams, 2002.

[16] Brian Litt and Zachary Ives. The international epilepsy electrophysiology database. In *Proceedings of the Fifth International Workshop on Seizure Prediction*, 2011.

[17] Ronald Book, Shimon Even, Sheila Greibach, and Gene Ott. Ambiguity in graphs and expressions. *IEEE Transactions on Computers*, C-20(2):149–153, 1971.

[18] Richard Edwin Stearns and Harry B. Hunt III. On the equivalence and containment problems for unambiguous regular expressions, regular grammars and finite automata. *SIAM Journal on Computing*, 14(3):598–611, 1985.

[19] Leonardo De Moura and Nikolaj Bjørner. Satisfiability modulo theories: Introduction and applications. *Communications of the ACM*, 54(9):69–77, 2011.

[20] Java's lambda expressions. `https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html`.

[21] Margus Veanes, Peli de Halleux, and Nikolai Tillmann. Rex: Symbolic regular expression explorer. In *Proceedings of the 3rd International Conference on Software Testing, Verification and Validation (ICST '10)*, pages 498–507. IEEE, 2010.

[22] Jacques Sakarovitch. *Elements of Automata Theory*. Cambridge University Press, 2009.

[23] Manfred Droste, Werner Kuich, and Heiko Vogler, editors. *Handbook of Weighted Automata*. Springer, 2009.

[24] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, 2003.

[25] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[26] ReactiveX: An API for asynchronous programming with observable streams. `http://reactivex.io/`.

[27] Esper for Java. `http://www.espertech.com/esper/`.

[28] Apache Flink: Scalable batch and stream data processing. `https://flink.apache.org/`.