Stream Processing Languages in the Big Data Era

Martin Hirzel
IBM Research, USA
hirzel@us.ibm.com

Guillaume Baudart
IBM Research, USA
Guillaume.Baudart@ibm.com

Angela Bonifati
Lyon 1 University, France
angela.bonifati@univ-lyon1.fr

Emanuele Della Valle
Politecnico di Milano, Italy
emanuele.dellavalle@polimi.it

Sherif Sakr
University of Tartu, Estonia
sherif.sakr@ut.ee

Akrivi Vlachou
University of Piraeus, Greece
avlachou@aueb.gr

ABSTRACT

This paper is a survey of recent stream processing languages, which are programming languages for writing applications that analyze data streams. Data streams, or continuous data flows, have been around for decades. But with the advent of the big-data era, the size of data streams has increased dramatically. Analyzing big data streams yields immense advantages across all sectors of our society. To analyze streams, one needs to write a stream processing application. This paper showcases several languages designed for this purpose, articulates underlying principles, and outlines open challenges.

1. INTRODUCTION

We have entered the big-data era: the world is awash with data, and more data is being produced every second of every day. Data analytics solutions must contend with data being *big* both in the static-data sense of an ocean of many bytes and in the streaming sense of a firehose of many bytes-per-second. In fact, driven by the realization that static data is merely a snapshot of parts of a data stream, the data technology industry is focusing increasingly on data-in-motion. Analyzing the stream instead of the ocean yields more timely insights and saves storage resources [6].

Stream processing languages facilitate the development of stream processing applications. Streaming languages simplify common coding tasks and make code more readable and maintainable, and their compilers catch programming mistakes and apply optimizing code transformations. The landscape of streaming languages is diverse and lacks broadly accepted standards. Stephens [79] and Johnston et al. [56] published surveys on stream processing languages in 1997 and 2004. Much has happened since then, from database-inspired streaming languages to the rise of big data and beyond. Our survey continues where prior surveys left off, focusing on streaming languages in the big-data era.

A *stream* is a sequence of data items, and the length of a stream is conceptually infinite, in the sense that waiting for it to end is ill-defined [70]. A streaming application is a computer program that consumes and produces streams. A stream processing language is a domain-specific language designed for expressing streaming applications. The goal of a stream processing language is to strike a balance between the three requirements of *performance*, *generality*, and *productivity*. Performance is about answering high-throughput input streams with low-latency output streams. Generality is about making it possible to handle a variety of processing needs and data formats. And productivity is about enabling developers to write good code quickly.

Traditionally, programming languages have been characterized by their paradigm, including imperative, functional, declarative, object-oriented, etc. However, for streaming languages, the paradigm is not the most important characteristic; most streaming languages are more-or-less declarative. More important characteristics include the data model (e.g., relational, XML, RDF), execution model (e.g., synchronous, big-data), and target domain and users (e.g., event detection, reasoning, end-users). Section 2 surveys languages based on these characteristics. Section 3 generalizes from individual languages to extract recurring concepts and principles. Section 4 does the inverse: instead of looking at what most streaming languages have in common, it explores what most streaming languages lack. Finally, Section 5 concludes our paper.

2. STREAM PROCESSING LANGUAGES

There is much diversity in stream processing languages, stemming from different primary objectives, data models, and ways of thinking. This section surveys eight styles of stream processing languages. Each subsection introduces one of these styles using an exemplary language, followed by a brief discussion of important other languages of the same style.

2.1 Relational Streaming

```
1 Select IStream( Max(len) As mxl,
2                 MaxCount(len) As num,
3                 ArgMax(len, caller ) As who )
4 From Calls[Range 24 Hours Slide 1 Minute]
```

Figure 1: CQL code example.

In 2004, Arasu et al. at Stanford introduced CQL (for Continuous Query Language) [11]. CQL has been designed as an SQL-based declarative language for implementing continuous queries against streams of data, such as the LinearRoad benchmark [10]. The design was influenced by the TelegraphCQ system, which proposed an SQL-based language with a focus on expressive windowing constructs [29]. Figure 1 illustrates a CQL code example that uses a time-based sliding window (per minute within the last 24 hours) over phone calls to return the maximum phone call length along with its count and caller information.

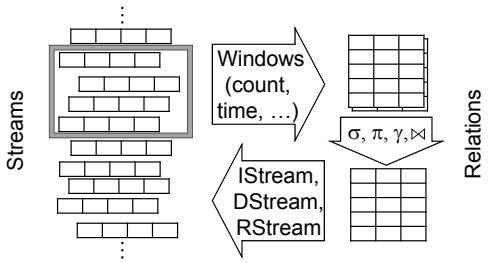

Figure 2: CQL algebra operators.

The semantics of CQL are based on two phases of data, *streams* and *relations*. As Figure 2 illustrates, CQL supports three classes of operators over these types. First, *stream-to-relation* operators freeze a stream into a relation. These operators are based on *windows* that, at any point of time, contain a historical snapshot of a recent portion of the stream. CQL includes time-based and tuple-based windows, both with optional partitioning. Second, *relation-to-relation* operators turn relations into another relation. These operators are expressed using standard SQL syntax and come from traditional relational algebra, such as select (σ), project (π), group-by-aggregate (γ), and join ($\bowtie$). Third, *relation-to-stream* operators thaw a relation back into a stream. CQL supports three operators of this class: `IStream`, `DStream`, and `RStream` (to capture inserts, deletes, or the entire relation).

Streaming SQL dialects were preceded by temporal relational models such as the one by Jensen and Snodgrass in the early '90s [55]. In their model, each temporal relation has two main dimensions: a valid time record and transaction time. Besides TelegraphCQ, another CQL predecessor was GSQL [35]. In addition to the standard SQL operators (e.g., σ, π, γ, $\bowtie$), GSQL supported a *merge* operator that combines streams from multiple sources in order as specified by ordered attributes. GSQL supported joins as long as it could determine a window from ordered attributes and join predicates.

CQL has influenced the design of many systems, for example, StreamInsight [5] and StreamBase [76]. Jain et al. described an approach to unify two different proposed SQL extensions for streams [54]. The first, by Oracle, was CQL-based and used a time-based execution model that could model simultaneity. The second, by StreamBase, used a tuple-based execution model that provided a way to react to primitive events as soon as they are seen by the system. SECRET goes beyond Jain et al.'s work to comprehensively understand the results of various window-based queries (e.g., time- and tuple-based windows) [19]. Zou et al. showed how to turn a stream of queries into a stream query by streaming their parameters [89]. Chandramouli et al. presented TiMR, which implemented temporal queries over the MapReduce framework [27]. And finally, Soulé et al. [77] presented a type system and small-step operational semantics for CQL via translation to the Brooklet stream-processing calculus [78].

2.2 Synchronous Dataflow

```
1 node tracker (speed, limit : int) returns (t: int);
2 var x: bool; cpt: int when x;
3 let
4   x = (speed > limit);
5   cpt = counter((0, 1) when x);
6   t = current(cpt);
7 tel
```

Figure 3: Lustre code example.

Synchronous dataflow (SDF) languages were introduced to ease the design of real-time embedded systems. They allow programmers to write a well-defined deterministic specification of the system. It is then possible to test, verify, and generate embedded code. The first dataflow synchronous languages Lustre [25] (Caspi and Halbwachs) and Signal [60] (Le Guernic, Benveniste, and Gautier) were proposed in France in the late 1980s. A dataflow synchronous program is a set of equations defining streams of values. Time proceeds by discrete logical steps, and at each step, the program computes the value of each stream depending on its inputs and possibly previously computed values. This approach is reminiscent of block diagrams, a popular notation to describe control systems. Figure 3 presents a Lustre code example that tracks the num-

ber of times the speed of a vehicle exceeds the speed limit. The counter cpt starts with 0 and is incremented by 1 each time the current speed exceed the current limit (when x). The return value t maintains the last computed value of cpt between two occurrences of x (current(cpt)).

Compared to the other languages presented here, SDF languages are relatively low level and target embedded controllers. The focus is on compiling efficient code that executes in bounded memory with a predictable execution time. In particular, this imposes that the schedule and communication rates can be statically computed by the compiler. Additional static analyses reject programs with potential initialization or causality issues. Compilers produce imperative code that can be executed in a control loop without communication buffers triggered by external events or on a periodic signal (e.g., every millisecond). The link between logical and real time is left to the designer of the system.

The dataflow synchronous approach has inspired multiple languages: Lucid Synchrone [73] combines the dataflow synchronous approach with functional features à la ML, StreamIt [80] focuses on efficient parallel processing of large streaming applications, and Zélus [20] is a Lustre-like language extended with ordinary differential equations to define continuous-time dynamics. Lustre is also the backbone of the industrial language and compiler Scade [34] routinely used to program embedded controllers in many critical applications.

2.3 Big-Data Streaming

```
1 stream<float64 len, rstring caller > Calls = CallsSrc() {}
2 type Stat = tuple<float64 len, int32 num, rstring who>;
3 stream<Stat> Stats = Aggregate(Calls) {
4     window Calls: sliding , time(24.0*60.0*60.0), time(60.0);
5     output Stats: len = Max(Calls.len),
6                   num = MaxCount(Calls.len),
7                   who = ArgMax(Calls.len, Calls. caller );
8 }
```

Figure 4: SPL code example.

The need to handle diverse data and processing requirements at scale motivated several recent big-data streaming languages and systems [3, 4, 24, 26, 51, 59, 69, 81, 87]. Each of them makes it easy to integrate operators written in general-purpose languages and to parallelize them on clusters of multicore computers. Hirzel et al. introduced the SPL language as part of the IBM Streams product in 2010 [50, 51]. Figure 4 shows an example for a similar use-case as Figure 1. Line 1 defines a stream Calls by invoking an operator CallsSrc, and Lines 3-8 define a stream Stats by invoking an operator Aggregate. An SPL program explicitly specifies a directed graph of stream edges and operator nodes. Streams carry tuples; in the examples, tuple attributes contain primitive values, but in general, they can also contain compound values such as other tuples or lists. Operators create and transform streams; operators are defined by users or libraries, not built into the language. Operators can be further configured upon invocation, for example, with windows or output assignments. To facilitate distribution, SPL's semantics are defined to require minimal synchronization between operators [77].

Like SPL, the core concept of other languages for big-data streaming is also that of a directed graph of streams and operators. This graph is an evolution of the query plan of earlier stream-relational systems. In fact, one can view Aurora [2], Borealis [1], and Spade [47] as the evolutionary links between relational and big-data streaming languages. They still focused on relational operators while already encouraging developers to explicitly code graphs.

Unlike SPL, which is a stand-alone language, later big-data streaming systems offer languages that are embedded in a general-purpose host language, typically Java. MillWheel focused on key-based partitioned parallelism and semi-automatic handling of out-of-order data [3]. Naiad focused on supporting both streaming and iterative batch analytics [69], using elaborate timestamps and a LINQ-based surface language [66]. Spark Streaming emulated streaming by repeated computations on immutable in-memory data batches [87]. Storm offered at-least-once semantics via buffering and acknowledgements [81]. Trill used batching to improve throughput and offered an extensible aggregation framework [26]. Heron displaced Storm by adding several improvements, such as a back-pressure mechanism [59]. Beam picks up where MillWheel left off, giving programmers ways to reconcile event time and processing time [4]. And finally, Flink focuses on supporting both real-time streaming and batch analytics [24].

All of the above-listed big-data streaming systems offer embedded languages for specifying more-or-less explicit stream graphs. An embedded language is an advanced library or framework that makes heavy use of host-language abstractions such as lambdas, generics, and local variable type inference. For instance, LINQ integrates SQL-inspired query syntax in a general-purpose language [66]. Embedded languages offer simple interoperability with their host language, as well as leveraging host-language tools and skills [52]. On the downside, since they are not self-contained, they are hard to isolate clearly from the host language, inhibiting debugging, optimization, and standardization.

2.4 Complex Event Processing

```
1  stream<Alert> Alerts = MatchRegex(Calls) {
2    param
3      partitionBy : caller ;
4      predicates : {
5        tooFarTooFast =
6              geoDist( First (loc ), Last(loc )) >= 10.0
7          && timeDist(First(ts ), Last(ts )) <= 60.0; };
8      pattern     : ".+ tooFarTooFast";
9    output
10     Alerts : who=caller, where=Last(loc), when=Last(ts);
11 }
```

Figure 5: CEP example.

Complex event processing (CEP) uses patterns over simple events to detect higher-level, *complex*, events that may comprise multiple simple events. CEP can be considered either as an alternative to stream processing or as a special case of stream processing. The latter consideration has led to the definition of CEP operators in streaming languages. For example, the MatchRegex [49] operator implements CEP in the library of the SPL language [51] (Section 2.3). MatchRegex was introduced by Hirzel in 2012, influenced by the MATCH-RECOGNIZE proposal for extending ANSI SQL [88]. Compared to its SQL counterpart, MatchRegex is simplified, syntactically concise, and easy to deploy as a library operator. MatchRegex is implemented via code generation and translates to an automaton for space- and time-efficient incremental computation of aggregates. However, it omits other functionalities beyond pattern matching, such as joins and reporting tasks. Figure 5 shows an example for detecting a complex event when simple phone-call events occur over 10 miles apart within 60 seconds. Line 8 defines the regular expression, where the period (.) matches any simple event; the plus (+) indicates at-least-once repetition; and tooFarTooFast is a simple event defined via a predicate in Lines 5–7. The First and Last functions reference corresponding simple events in the overall match: in this case, the start of the sequence matched by .+ and the simple event matched by tooFarTooFast.

One of the earliest languages for complex event queries on real-time streams was SASE [86]. The language was designed to translate to algebraic operators, but did not yet support aggregation or regular expressions with Kleene closure, as used in Figure 5. The Cayuga Event Language offered aggregation and Kleene closure, but did so in a handcrafted syntax instead of familiar regular expression syntax [42]. The closest predecessor of MatchRegex was the MATCH-RECOGNIZE proposal for extending SQL with pattern recognition in relational rows [88]. It used regular-expression syntax as well as aggregations. Like MatchRegex, it is embedded in a host language that supports orthogonal features via operators such as joins. Another take on CEP using regular expressions was EventScript [33], which allowed the patterns to be interspersed with action blocks. While most CEP pattern matching is inherently sequential, Chandramouli et al. generalized it for out-of-order data streams [28], a topic further discussed in Section 4.1.

Recently, CEP is also supported by several big-data streaming engines, such as Trill [26], Esper [45], and Flink [24], the latter exhibiting a CEP library since its early 1.0 version. Indeed, the high throughput and low latency nature of these engines make them suitable for CEP's real time analytics.

2.5 XML Streaming

In 2002, Diao et al. [44] presented YFilter, which implemented continuous queries over XML streaming data using a subset of the XPath language [32]. YFilter applied a multi-query optimization that used a single finite state machine to represent and evaluate several XPath expressions. In particular, YFilter exploited commonalities among path queries by merging the common prefixes of the paths so that they were processed at most once. This shared processing improved performance significantly by avoiding redundant processing for duplicate path expressions.

Before YFilter, which processed streams of XML documents, came NiagaraCQ, which processed update streams to existing XML documents [30], borrowing syntax from XML-QL [43]. NiagaraCQ supported incremental evaluation to consider only the changed portion of each updated XML file. It supported two kinds of continuous queries: *change-based* queries, which trigger as soon as new relevant data becomes available, and *timer-based* queries, which trigger only at specified time intervals. XSQ is an XPath-based language for not just filtering but transforming streams of XML documents [71]. And XMLParse is an operator for XML stream transformation in a big-data streaming language [67].

2.6 RDF Streaming

In 2009, Della Valle et al. called the semantic web and AI community to investigate how to represent, manage, and reason on heterogeneous data streams in the presence of expressive domain models (captured by ontologies) [38]. Those communities were still focusing on static knowledge bases, and solutions to incorporate changes were too complex to apply to big data streams. Della Valle et al. pro-

posed to name this new research area *stream reasoning* [41], and the sub-area focused on the semantic web *RDF Stream Processing* (RSP) [82]. This section presents RSP, while Section 2.7 elaborates on stream reasoning.

RSP research extended the semantic web stack [8] to represent heterogeneous streams, continuous queries, and continuous reasoning. Inspired by CQL [11], Della Valle et al. proposed Continuous SPARQL (C-SPARQL, [37]), inspiring multiple extensions [7, 23, 61]. In 2013, a W3C community group[1] was established to define RSP-QL syntax [39] and semantics [40]. In RSP-QL, an *RDF stream* is an unbounded sequence of time-varying graphs $\langle t, \tau \rangle$, where t is an RDF graph and τ is a non-decreasing timestamp. A RSP-QL query is a continuous query on multiple RDF streams and graphs.

```
1  REGISTER STREAM :out
2  AS CONSTRUCT RSTREAM { ?x a :Hub }
3  FROM NAMED WINDOW :lwin
4         ON :in [ RANGE PT120M STEP PT10M]
5  FROM NAMED WINDOW :swin
6         ON :in [ RANGE PT10M STEP PT10M]
7  WHERE {
8    WINDOW :lwin{
9      SELECT ?x ( COUNT(*) AS ?totalLong)
10     WHERE { ?c1 :callee ?x. }
11     GROUP BY ?x }
12   WINDOW :swin{
13     SELECT ?x ( COUNT(*) AS ?totalShort)
14     WHERE { ?c2 :callee ?x. }
15     GROUP BY ?x }
16   GRAPH :bg {?x :hasStandardDeviation ?s }
17   FILTER ((?totalShort − ?totalLong/12)/?s > 2)
18 } GROUP BY ?x
```

Figure 6: RSP-QL example.

Figure 6 illustrates an RSP-QL query that continuously identifies *communication hubs*. The idea is to find callees who appear more frequently than usual. Line 1 registers stream out and Line 2 sends the query result on that stream. Lines 3-6 open a short 10-minute tumbling window swin and a long 2-hour sliding window lwin on the input stream in. Lines 8-11 and 12-15 count the number of calls per callee in the long and short window, respectively. Lines 16-17 fetch the standard deviation of the number of calls for each callee from a static graph, join it with the callees appearing in both windows, and select callees two standard deviations above average.

2.7 Stream Reasoning

Automated reasoning plays a key role in modern information integration where an ontology offers a conceptual view over pre-existing autonomous data sources [63]. In this setting, the reasoner can find answers that are not syntactically present in the data sources, but are deduced from the data and the ontology. This query-answering approach is called ontology-based data access [72].

[1] http://www.w3.org/community/rsp/

```
1  SubObjectPropertyOf(
2     ObjectPropertyChain( :calls :calls ) :gossips
3  )
4  TransitiveObjectProperty( :gossips )
5
6  REGISTER STREAM GossipMeter AS
7  SELECT (count(?x) AS ?impact)
8  FROM NAMED WINDOW :win
9         ON :in [ RANGE PT60M STEP PT10M]
10 WHERE { :Alice :gossips ?x }
```

Figure 7: Stream reasoning example with two ontological axioms and a RSP-QL query.

As RDF is the dominant data model in reasoning for data integration, RDF streaming languages (Section 2.6) bridge the gap between stream processing and ontology-based data integration. Della Valle et al. opened up this direction, showing how continuous reasoning can be reduced to periodic repetition of reasoning over a windowed ontology stream [37]. Figure 7 shows an RSP-QL query that uses reasoning to continuously count how many people :Alice gossips with. Consider an RDF stream with the triples $\langle$:Alice :calls :Bob, $\tau_i\rangle$ and $\langle$:Bob :calls :Carl, $\tau_{i+1}\rangle$. Lines 1-4 define :gossips as the transitive closure of :calls. When the window contains these two triples, the RSP-QL query returns 2, because :Alice :gossips :Bob directly calling him, but the system can also deduce that she :gossips :Carl indirectly via :Bob.

While conceptually simple, this kind of reasoning is hard to do efficiently. Barbieri et al. [14] and Komazec et al. [57] pioneered it optimizing the DRed algorithm observing that in stream processing deletion becomes predictable. The current state-of-the-art is the work of Motik et al. [68].

In parallel, Ren and Pan proposed an alternative approach via truth maintenance systems [74]. Calbimonte et al. exploited ontology-based data access [22]. Heintz et al. developed logic-based spatio-temporal stream reasoning [36]. Anicic et al. bridged stream reasoning with complex event processing grounding both in logic programming [7]. Beck et al. used answer set programming to model expressive stream reasoning tasks [17] in Ticker [18] and Laser [16]. Inductive stream reasoning, i.e., applying machine-learning to RDF streams or to ontology streams, is also an active field [15, 31, 62].

2.8 Streaming for End-Users

We use the term *end-users* to refer to users without particular software development training. Prob-

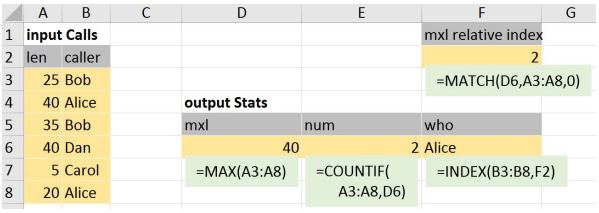

Figure 8: ActiveSheets example.

ably the most successful programming tool for end-users is spreadsheet formulas. And from the early days of VisiCalc in 1979 [21], spreadsheet formulas have been *reactive* in the sense that any changes in their inputs trigger an automatic recomputation of their outputs. Therefore, in 2014, Vaziri et al. designed ActiveSheets, a spreadsheet-based stream programming model [84]. Figure 8 gives an example that implements a similar computation as Figure 1. Cells A3:B8 contain a sliding window of recent call records, which ActiveSheets updates from live input data. Cells D6:F6 contain the output data, (re-)computed using reactive spreadsheet formulas. The formula E6=COUNTIF(A3:A8,D6) counts how many calls in the window are as long as a longest call. The formula F6=INDEX(B3:B8,F2) uses the relative index F2 of the longest len to retrieve the corresponding caller. ActiveSheets was influenced by synchronous dataflow, discussed in Section 2.2. Of course, spreadsheets are not the only approach for end-user programming. For instance, MARIO constructed streaming applications automatically based on search terms [75]. Linehan et al. used a controlled natural language for specifying event processing rules [65]. And TEM used model-driven development based on a spreadsheet [46].

3. PRINCIPLES

The previous section described concrete stream processing languages belonging to several families. This section takes a cross-cutting view and explores concepts that many of these languages have in common by identifying the language design principles behind the concepts. The views and opinions expressed herein are those of the authors and are not meant as the final word. Explicitly articulating principles demystifies the art of language design. We categorize language design principles according to the three requirements from Section 1, namely performance, generality, and productivity.

The **performance** requirement is addressed by streaming language design principles $\mathbf{P_1}$–$\mathbf{P_4}$:

$\mathbf{P_1}$ *Windowing principle.* Windows turn streaming data into static data suitable for optimized static computation. For instance, in CQL, windows produce relations suitable for classic relational algebra [9], optimizable via classic relational rewrite rules (see Figure 2).

$\mathbf{P_2}$ *Partitioning principle.* Key-based partitions enable independent computation over disjoint state, thus simplifying data parallelism. For instance, MatchRegex performs complex event processing separately by partition [49] (see Line 3 of Figure 5). Principles P_1 and P_2 also simplify advanced state management, e.g., in key-value stores for operator migration [48].

$\mathbf{P_3}$ *Stream graph principle.* Streaming applications are graphs of operators that communicate almost exclusively via streams, making them easy to place on different cores or machines. This principle is central to the big-data languages in Section 2.3 such as SPL [51] (see Figure 4).

$\mathbf{P_4}$ *Restriction principle.* The schedules and communication rates in a streaming application are restricted for both performance and safety. For instance, Lustre can be compiled to a simple imperative control loop without communication buffers [25] (see Section 2.2).

The **generality** requirement is addressed by streaming language design principles $\mathbf{P_5}$–$\mathbf{P_8}$:

$\mathbf{P_5}$ *Orthogonality principle.* Basic language features are irredundant and work the same independently of how they are composed. For instance, in CQL, relational-algebra operators are orthogonal to windows [9] (see Section 2.1).

$\mathbf{P_6}$ *No-built-ins principle.* The core language remains slim and regular by enabling extensions in the library. For instance, in SPL, relational operators are not built into the language, but are user-defined in the library instead [51] (see Lines 3–8 of Figure 4).

$\mathbf{P_7}$ *Auto-update principle.* The syntax of conventional non-streaming computation is overloaded to also support reactive computation. For instance, ActiveSheets uses conventional spreadsheet formulas, updating their output when input cells change [84] (see Figure 8). The Lambda or Kappa architectures [58] take this to the extreme by combining batch and streaming outside of the language.

$\mathbf{P_8}$ *General-feature principle.* Similar special-case features are replaced by a single more-general feature. For instance, operator parameters in SPL [51] accept general uninterpreted expressions, including predicates for the special case of CEP [49] (see Lines 4–7 of Figure 5).

The **productivity** requirement is addressed by streaming language design principles $\mathbf{P_9}$–$\mathbf{P_{12}}$:

Language	*Performance*	*Generality*	*Productivity*
CQL	P_1 P_2 P_3	**P_5** P_8	**P_9**
Lustre	P_4	**P_5** P_6 P_7 P_8	**P_9** P_{10} P_{11} P_{12}
SPL	P_1 P_2 P_3	**P_5** P_6 P_8	**P_9** P_{11} P_{12}
MatchRegex	P_2	**P_5** P_6 P_8	**P_9** P_{10} P_{12}
YFilter	P_4	**P_5** P_6	**P_9** P_{10}
RSP-QL	P_1 P_3	**P_5** P_6 P_8	**P_9** P_{10} P_{11}
ActiveSheets	P_1 P_2 P_4	**P_5** P_6 P_7 P_8	**P_9** P_{10}

Table 1: Which of the languages that served as examples in Section 2 satisfy which of the language design principles in Section 3.

P_9 *Familiarity principle.* The syntax of non-streaming features in streaming languages is the same as in non-streaming languages. This makes the streaming language easier to learn. For instance, CQL [11] adopts the select-from-where syntax of SQL (see Figure 1).

P_{10} *Conciseness principle.* The most concise syntax is reserved for the most common tasks. This increases productivity since there is less code to write and read. For instance, regular expressions represent "followed-by" concisely via juxtaposition $e_1\, e_2$ (see Line 8 of Figure 5).

P_{11} *Regularity principle.* Data literals, patterns that match them, and/or declarations all use similar syntax. For instance, RSP-QL uses pattern syntax resembling concrete RDF triples (see Line 10 of Figure 6).

P_{12} *Backward reference principle.* Code direction is consistent with both scope and control dominance, for readability. For example, Lustre declares variables before their use (see Figure 3).

3.1 Principles Summary

Good language design is driven by principles, but it is also an exercise in prioritizing among these principles. For instance, CQL satisfies P_9 (familiarity principle) by adopting SQL's syntax and CQL violates P_{12} (backward reference principle) by adopting SQL's scoping rules. Table 1 summarizes principles by language. Only two of the twelve principles (P_5 and P_9, shown in bold) are uniformly covered, both related to the ease of use of the language (separation of concerns and syntax familiarity). Although some of the languages exhibit fewer principles, Table 1 does not provide a comparative metric for quantifying the coverage of each principle; such a metric would be hard to agree upon. Satisfying more principles does not automatically imply satisfying the associated requirement better. While we formulated the principles from the perspective of streaming languages, we do not claim to have invented them: many are well-known from the design of other programming languages. For instance, the orthogonality principle was a stated aim of the Algol 68 language specification [83]. Now that we have seen concepts that are *present* in most streaming languages, the next section will explore what is commonly *missing or underdeveloped*.

4. WHAT'S NEXT?

In the Big Data era, the need to process and analyze a high volume of data is a fundamental problem. Industry analysts point out that besides volume, there are also challenges in variety, velocity, and veracity [53]. Streaming languages naturally handle volume and velocity of the data, since they are designed to process data in real-time in a streaming way. Thus, in the following, we focus on veracity and variety, since there are more open research challenges in these directions despite much recent progress in streaming languages. In addition, we elaborate on the challenge of adoption, which is an important problem of programming languages in general and of streaming languages in particular.

4.1 Veracity

With the evolution of the internet of things and related technologies, many end-user applications require stream processing and analytics. Streaming languages should ensure veracity of the output stream in terms of accuracy, correctness, and completeness of the results. Furthermore, they should not sacrifice performance either, answering high-throughput input streams with low-latency output streams. Veracity in a streaming environment depends on the semantics of the language since the stream is infinite and new results may be added or computed aggregates may change. It is important that the output stream for a given input stream be well-defined based on the streaming language semantics. For example, if the language offers a sliding time window feature, any aggregate should be computed correctly at any time point based on all data within the time window. Stream veracity problems may occur for different reasons. For example, in multi-streaming applications, each stream may be produced by sensors. Errors may occur either in the data itself (e.g., noisy sensor readings) or by delays or data loss during the transfer to the stream processing system. For instance, data may arrive out-of-order because of communication delays or because of the inevitable time drift between independent distributed stream sources. Ideally, the output stream should be accurate, complete, and timely even if errors occur in the input stream. Unfortunately, this is not always feasible.

Why is this important? Veracity of the output of streaming applications is important when high-stakes and irreversible decisions are based on these outputs. In the big-data era, veracity is one of the most important problems even for non-streaming data processing, and stream processing makes veracity even more challenging than in the static case. Streams are dynamic and usually operate in a distributed environment with minimal control over the underlying infrastructure. Such a loosely coupled model can lead to situations where any data source can join and leave on the fly. Moreover, stream-producing sensors have limitations such as processing power, energy level, or memory consumption, which can easily compromise veracity.

How can we measure the challenge? To estimate the robustness of a streaming language implementation to veracity problems, we define as *ground truth* the output stream in the absence of veracity problems (for example data loss or delayed data). Then we can quantify veracity. Let *error* be a function that compares the produced result of an approach with and without veracity problems. An example of an error function is the number of false positives and false negatives. An approach is *robust* for veracity of streaming data if the error scales at most linearly with respect to the size and the error rate of the input stream, while the delay in the latency is bounded and independent of the input size. The streaming language veracity challenge can be broken down into the following measures $\mathbf{C_1}$–$\mathbf{C_3}$:

- $\mathbf{C_1}$ *Fault-tolerance.* A program in the language is robust even if some of its components fail. The language can define different behaviors, for example, at-least-once semantics in Storm [81] or check-pointing in Spark Streaming [87].
- $\mathbf{C_2}$ *Out-of-order handling.* This measure has two facets. First, the streaming language should have clear semantics about the expected result. Second, the streaming language should be robust to out-of-order data and should ensure that the expected output stream is produced with limited latency. Li et al. define out-of-order stream semantics based on low watermarks [64]; Spark Streaming relies on idempotence to handle stragglers [87]; and Beam separates event time from processing time [4].
- $\mathbf{C_3}$ *Inaccurate value handling.* A program in the language is robust even if some of its input data is wrong. The language can help by supporting statistical quality measures [85].

Why is this difficult? In stream processing, data is typically sent on a best-effort basis. As a result, data can be lost, incorrect, arrive out of order, or be approximate. This is exacerbated by the fact that the streaming setting affords limited opportunity to compensate for these issues. Furthermore, the performance requirements of streaming systems encourage the use of approximate computing [12], thus increasing the uncertainty of the data. Also, machine-learning often yields uncertain results due to imperfect generalization. An important aspect of streaming data is ordering, typically characterized by time. The correctness of the response to queries depends on the source of ordering, such as the creation, processing, or delivery time. Stream processing often requires that each piece of data must be processed within a window, which can be characterized by predefined size or temporal constraints. In stream settings, sources typically do not receive control feedback. Consequently, when exceptions occur, recovery must occur at the destination. This reduces the space of possibilities for handling transaction rollbacks and fault tolerance.

4.2 Data Variety

Data variety refers to the presence of different data formats, data types, data semantics, and associated data management solutions in an information system. The term emerged with the advent of Big Data, but the problem of taming variety is well known for machine understanding of unstructured data such as text, images, and video as well as (syntactic, structural, and semantic) interoperability and data integration for structured and semistructured data. There are multiple known solutions to data variety for a moderate number of high-volume data sources. But data variety is still unsolved when there are hundreds of data sources to integrate or when the data to integrate is highly dynamic or streaming (as in this paper).

Why is this important? Increasingly, applications must process heterogeneous data streams in real-time together with large background knowledge bases. Consider the following two examples from [41] (where interested readers can find others).

In the first example, we want to use sensor readings of the last 10 minutes to find electricity-producing turbines that are in a state similar (e.g., Pearson correlated by at least 0.75) to any turbine that subsequently had a critical failure. Here, data variety arises from having tens of turbines of 3-4 different types equipped with different sensors deployed other many years, where more sensors will be deployed in the future. Moreover, in many cases, once an anomaly is detected, the user also needs to retrieve multimedia maintenance instructions and

annotations to complete the diagnosis process.

In the second example, we want to use the latest open traffic information and social media as well as the weather forecast to determine if the users of a mobile mobility app are likely to run into a traffic jam during their commute tonight and how long it will take them to get home. Here, data variety arises from using third-party data sources that are free to evolve in syntax, structure, and semantics.

How can we measure the challenge? The streaming language data variety challenge can be broken down into the following measures $\mathbf{C_4}$–$\mathbf{C_6}$:

- $\mathbf{C_4}$ *Expressive data model.* The data model used to logically represent information is expressive and allows encoding multiple data types, data structures, and data semantics. This is the path investigated by RSP-QL [41, 82].
- $\mathbf{C_5}$ *Multiple representations.* The language can ingest data in multiple representations, offering the programmer a unified set of logical operators while implementing physical operators that work directly on the representations for performance. An example is the most recent evolution of the Streaming Linked Data framework [13].
- $\mathbf{C_6}$ *New sources with new formats.* The language allows adding new sources where data are represented in a format unforeseen when the language was released. This might be accomplished by extending R2RML[2].

Why is this difficult? Deriving value is harder for a system that has to tame data variety than for a system that only has to handle a single well-structured data source. This is because solutions that analyze data require homogeneous well-formed input data, so, when there is data variety, preparing such data requires a number of different data management solutions that take time to perform their part of the processing as well as to coordinate among each others. This time is particularly relevant in stream processing, where answers should be generated with low latency. Even if the time available to answer depends on the application domain (in call centers, routing needs to be decided in subseconds, while in oil operations, dangerous situations must be detected within minutes), traditional batch pipelines for feature extraction and extract-transform-load (ETL) may take so long that the results, when computed, are no longer useful. For this reason, it is still challenging to tame variety in stream processing systems.

[2] `https://www.w3.org/TR/r2rml`

4.3 Adoption

Stream processing languages have an adoption problem. As Section 2 illustrates, there are several families of streaming languages comprising several members each. But no one streaming language has been broadly adopted. The language family receiving the most attention from large technology companies is big-data streaming, including offerings by Google [3], Microsoft [5], IBM [51], and Twitter [81]. However, they all differ. Furthermore, in the pursuit of interoperability and expediency, most big-data streaming languages are not stand-alone but embedded in a host language. While being embedded gives a short-term boost to language development, the entanglement with a host language makes it hard to offer stable and clear semantics. And, if the history of databases is any guide, such stable and clear semantics are useful for agreeing on and consistently implementing a standard. Part of the reason that the relational model for databases displaced its disparate predecessors is its strong mathematical foundation. One of the most-used languages mentioned in this survey is Scade [34], but it is designed for embedded systems and not big-data streaming. Getting broad adoption for a big-data streaming language remains an open challenge.

Why is this important? Solving the adoption problem for stream processing languages would yield many benefits. It would encourage students to build marketable skills and give employers a sustainable hiring pipeline. It would raise attention to streaming innovation, benefiting researchers, and to streaming products, benefiting vendors. If most systems adopted more-or-less the same language, they would become easier to benchmark against each other. Other popular programming languages, such as SQL, Java, and JavaScript, flourished when companies competed against each other to provide better implementations of the language. On the downside, focusing on a single language would reduce the diversity of the eco-system, transforming innovation and competition from being broad to being deep. But overall, if the problem of streaming language adoption were solved, we would expect streaming systems to become more robust and faster.

How can we measure the challenge? The streaming language adoption challenge can be broken down into the following measures $\mathbf{C_7}$–$\mathbf{C_9}$:

- $\mathbf{C_7}$ *Widely-used implementation of one language.* One language in the family has at least one implementation that is widely used in practice, for instance, Scade for SDF [34].
- $\mathbf{C_8}$ *Standard proposal or standard.* There are serious efforts towards an official standard, for

Languages	*Veracity*	*Variety*	*Adoption*
Relational	**C2**		**C8** **C9**
Synchronous		**C4**	C7 **C9**
Big Data	C1 **C2**	**C4** C5 C6	C7
CEP	**C2**	**C4**	**C8**
XML		**C4** C6	
Stream Reasoning	C3	**C4** C5 C6	**C8** **C9**
End-user		**C4**	

Table 2: Which of the language families from Section 2 address which of the measures of streaming language challenges in Section 4.

instance, Jain et al. for StreamSQL [54] or MATCH-RECOGNIZE for CEP [88].

C9 *Multiple implementations of same language.* One language in the family has multiple more-or-less compatible implementations, for instance, Lustre [25] and Scade [34] for SDF.

Language adoption is driven not just by the technical merits of the language itself but also by external factors, such as industry support or implementations that are open-source with open governance.

Why is this difficult? Adoption is hard for any programming language, but particularly so for a streaming language. While streaming in general is not new [79], big-data streaming is a relatively recent phenomenon. And big-data streaming, in turn, is driven by several ongoing industry trends, including the internet of things, cloud computing, and artificial intelligence (AI). Since all three of these trends are themselves actively shifting, they provide an unstable ecosystem for streaming languages to evolve. Furthermore, innovation often takes place in a setting where data is assumed to be at rest, as opposed to streaming, where data is in motion. For instance, most AI algorithms work over a fixed training data set, so additional research is necessary to make them work well online. When it comes to streaming languages, there is not even a consensus on what are the most important features to include. For instance, both the veracity and the variety challenge discussed previously have given rise to many feature ideas that have yet to make it into the mainstream. Since people come to streaming research from different perspectives, they sometimes do not even know each other's work, inhibiting adoption. This survey aims to mitigate that problem.

4.4 Challenges Summary

Table 2 summarizes the challenges. Compared to the coverage of principles in Table 1, the coverage of challenges is more sparse and spread out over research prototypes. That is why we tabulated it for language families instead of individual languages. There is much space for future work. The measures highlighted in bold are most covered across all the languages families. The ability to handle a wide variety of data formats appears to be a universal concern. Ultimately, we aim at streaming languages that are both principled and close the gap on all challenges.

5. CONCLUSION

This paper surveys recent stream processing languages. Given their numbers, it appears likely that more will be invented soon. We hope this survey will help the field evolve towards better languages by helping readers understand the state of the art.

Acknowledgements

The idea for this paper was conceived at Dagstuhl Seminar 17441 on "Big Stream Processing Systems". We thank our fellow participants of the seminar and the Dagstuhl staff. A. Bonifati's work was supported by CNRS Mastodons MedClean. S. Sakr's work was funded by the European Regional Development Fund via the Mobilitas Pluss program (grant MOBTT75). A. Vlachou's work was supported by the datAcron project, funded through the European Union's Horizon 2020 research and innovation program (grant 687591).

6. REFERENCES

[1] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. S. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The design of the Borealis stream processing engine. In *Conference on Innovative Data Systems Research (CIDR)*, pages 277–289, 2005.

[2] D. J. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: a new model and architecture for data stream management. *Journal on Very Large Data Bases (VLDB J.)*, 12(2):120–139, Aug. 2003.

[3] T. Akidau, A. Balikov, K. Bekiroglu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. MillWheel: Fault-tolerant stream processing at internet scale. In *Conference on Very Large Data Bases (VLDB) Industrial Track*, pages 734–746, Aug. 2013.

[4] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernandez-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. In *Conference on Very Large Data Bases (VLDB)*, pages 1792–1803, Aug. 2015.

[5] M. H. Ali, C. Gerea, B. Raman, B. Sezgin, T. Tarnavski, T. Verona, P. Wang, P. Zabback, A. Kirilov, A. Ananthanarayan, M. Lu, A. Raizman, R. Krishnan, R. Schindlauer, T. Grabs, S. Bjeletich, B. Chandramouli, J. Goldstein, S. Bhat, Y. Li, V. Di Nicola, X. Wang, D. Maier, I. Santos, O. Nano, and S. Grell. Microsoft CEP server and online behavioral targeting. In *Demo at Conference on Very Large Data Bases (VLDB-Demo)*, pages 1558–1561, 2009.

[6] H. C. Andrade, B. Gedik, and D. S. Turaga. *Fundamentals of stream processing: application design, systems, and analytics*. Cambridge University Press, 2014.

[7] D. Anicic, S. Rudolph, P. Fodor, and N. Stojanovic. Stream reasoning and complex event processing in ETALIS. *Semantic Web*, 3(4):397–407, 2012.

[8] G. Antoniou, P. T. Groth, F. van Harmelen, and R. Hoekstra. *A Semantic Web Primer, 3rd Edition*. MIT Press, 2012.

[9] A. Arasu, S. Babu, and J. Widom. The CQL continuous query language: semantic foundations and query execution. *Journal on Very Large Data Bases (VLDB J.)*, 15(2):121–142, 2006.

[10] A. Arasu, M. Cherniack, E. Galvez, D. Maier, A. S. Maskey, E. Ryvkina, M. Stonebraker, and R. Tibbetts. Linear road: A stream data management benchmark. In *Conference on Very Large Data Bases (VLDB)*, pages 480–491, 2004.

[11] A. Arasu and J. Widom. A denotational semantics for continuous queries over streams and relations. *SIGMOD Record*, 33(3):6–11, 2004.

[12] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Symposium on Principles of Database Systems (PODS)*, pages 1–16, 2002.

[13] M. Balduini, E. D. Valle, and R. Tommasini. SLD revolution: A cheaper, faster yet more accurate streaming linked data framework. In *European Semantic Web Conference (ESWC) Satellite Events*, pages 263–279, 2017.

[14] D. F. Barbieri, D. Braga, S. Ceri, E. Della Valle, and M. Grossniklaus. Incremental reasoning on streams and rich background knowledge. In *Extended Semantic Web Conference (ESWC)*, pages 1–15, 2010.

[15] D. F. Barbieri, D. Braga, S. Ceri, E. Della Valle, Y. Huang, V. Tresp, A. Rettinger, and H. Wermser. Deductive and inductive stream reasoning for semantic social media analytics. *IEEE Intelligent Systems*, 25(6):32–41, 2010.

[16] H. R. Bazoobandi, H. Beck, and J. Urbani. Expressive stream reasoning with Laser. In *International Semantic Web Conference (ISWC)*, pages 87–103, 2017.

[17] H. Beck, M. Dao-Tran, T. Eiter, and M. Fink. LARS: A logic-based framework for analyzing reasoning over streams. In *Conference on Artificial Intelligence (AAAI)*, pages 1431–1438, 2015.

[18] H. Beck, T. Eiter, and C. Folie. Ticker: A system for incremental ASP-based stream reasoning. *Theory and Practice of Logic Programming (TPLP)*, 17(5-6):744–763, 2017.

[19] I. Botan, R. Derakhshan, N. Dindar, L. Haas, R. J. Miller, and N. Tatbul. SECRET: A model for analysis of the execution semantics of stream processing systems. In *Conference on Very Large Data Bases (VLDB)*, pages 232–243, 2010.

[20] T. Bourke and M. Pouzet. Zélus: A synchronous language with ODEs. In *Conference on Hybrid Systems: Computation and Control (HSCC)*, pages 113–118, 2013.

[21] D. Bricklin and B. Frankston. VisiCalc computer software program, 1979. Reference Manual, Personal Software Inc.

[22] J. Calbimonte, J. Mora, and Ó. Corcho. Query rewriting in RDF stream processing. In *European Semantic Web Conference (ESWC)*, pages 486–502, 2016.

[23] J.-P. Calbimonte, O. Corcho, and A. J. G. Gray. Enabling ontology-based access to streaming data sources. In *International Semantic Web Conference (ISWC)*, pages 96–111, 2010.

[24] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache Flink: Stream and batch processing in a single engine. *IEEE Database Engineering Bulletin*, 36(4):28–38, Dec. 2015.

[25] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. LUSTRE: a declarative language for real-time programming. In *Symposium on Principles of Programming Languages (POPL)*, pages 178–188, 1987.

[26] B. Chandramouli, J. Goldstein, M. Barnett, R. DeLine, D. Fisher, J. C. Platt, J. F. Terwilliger, and J. Wernsing. Trill: A high-performance incremental query processor for diverse analytics. In *Conference on Very Large Data Bases (VLDB)*, pages 401–412, Aug. 2014.

[27] B. Chandramouli, J. Goldstein, and S. Duan. Temporal analytics on big data for web advertising. In *International Conference on Data Engineering (ICDE)*, pages 90–101, 2012.

[28] B. Chandramouli, J. Goldstein, and D. Maier. High-performance dynamic pattern matching over disordered streams. In *Conference on Very Large Data Bases (VLDB)*, pages 220–231, 2010.

[29] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. A. Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *Conference on Innovative Data Systems Research (CIDR)*, pages 668–668, 2003.

[30] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for internet databases. In *International Conference on Management of Data (SIGMOD)*, pages 379–390, 2000.

[31] J. Chen, F. Lécué, J. Z. Pan, and H. Chen. Learning from ontology streams with semantic concept drift. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 957–963, 2017.

[32] J. Clark and S. DeRose. XML path language (XPath) version 1.0. W3C recommendation, W3C, Nov. 1999. http://www.w3.org/TR/1999/REC-xpath-19991116.

[33] N. H. Cohen and K. T. Kalleberg. EventScript: An event-processing language based on regular expressions with actions. In *Conference on Languages, Compiler, and Tool Support for Embedded Systems (LCTES)*, pages 111–120, 2008.

[34] J.-L. Colaco, B. Pagano, and M. Pouzet. Scade 6: A formal language for embedded critical software development. In *Symposium on Theoretical Aspect of Software Engineering (TASE)*, 2017.

[35] C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk. Gigascope: A stream database for network applications. In *International Conference on Management of Data (SIGMOD) Industrial Track*, pages 647–651, 2003.

[36] D. de Leng and F. Heintz. Qualitative spatio-temporal stream reasoning with unobservable intertemporal spatial relations using landmarks. In *Conference on Artificial Intelligence (AAAI)*, pages 957–963, 2016.

[37] E. Della Valle, S. Ceri, D. F. Barbieri, D. Braga, and A. Campi. A first step towards stream reasoning. In *Future Internet Symposium (FIS)*, pages 72–81, 2008.

[38] E. Della Valle, S. Ceri, F. van Harmelen, and D. Fensel. It's a streaming world! Reasoning upon rapidly changing information. *IEEE Intelligent Systems*, 24(6):83–89, 2009.

[39] D. Dell'Aglio, J. Calbimonte, E. Della Valle, and Ó. Corcho. Towards a unified language for RDF stream query processing. In *European Semantic Web Conference (ESWC) Satellite Events*, pages 353–363, 2015.

[40] D. Dell'Aglio, E. Della Valle, J. Calbimonte, and Ó. Corcho. RSP-QL semantics: A unifying query model to explain heterogeneity of RDF stream processing systems. *International Journal on Semantic Web and Information Systems (IJSWIS)*, 10(4):17–44, 2014.

[41] D. DellAglio, E. Della Valle, F. van Harmelen, and A. Bernstein. Stream reasoning: A survey and outlook. *Data Science*, 1(1-2):59–83, 2017.

[42] A. Demers, J. Gehrke, B. Panda, M. Riedewald, V. Sharma, and W. White. Cayuga: A general purpose event monitoring system. In *Conference on Innovative Data Systems Research (CIDR)*, pages 412–422, 2007.

[43] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A query language for XML. *Computer Networks*, 31(11):1155–1169, 1999.

[44] Y. Diao, P. M. Fischer, M. J. Franklin, and R. To. YFilter: Efficient and scalable filtering of XML documents. In *Demo at International Conference on Data Engineering (ICDE-Demo)*, pages 341–342, 2002.

[45] Esper. Esper open source software for streaming analytics, 2018. `http://www.espertech.com/esper/` (Retrieved June 2018).

[46] O. Etzion, F. Fournier, I. Skarbovsky, and B. von Halle. A model driven approach for event processing applications. In *Conference on Distributed Event-Based Systems (DEBS)*, pages 81–92, 2016.

[47] B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, and M. Doo. SPADE: The System S declarative stream processing engine. In *International Conference on Management of Data (SIGMOD)*, pages 1123–1134, 2008.

[48] B. Gedik, S. Schneider, M. Hirzel, and K.-L. Wu. Elastic

scaling for data stream processing. *Transactions on Parallel and Distributed Systems (TPDS)*, 25(6):1447–1463, June 2014.
[49] M. Hirzel. Partition and compose: Parallel complex event processing. In *Conference on Distributed Event-Based Systems (DEBS)*, pages 191–200, 2012.
[50] M. Hirzel, H. Andrade, B. Gedik, V. Kumar, G. Losa, M. Mendell, H. Nasgaard, R. Soulé, and K.-L. Wu. SPL Streams Processing Language Specification. Technical Report RC24897, IBM Research, 2009.
[51] M. Hirzel, S. Schneider, and B. Gedik. SPL: An extensible language for distributed stream processing. *Transactions on Programming Languages and Systems (TOPLAS)*, 39(1):5:1–5:39, March 2017.
[52] P. Hudak. Modular domain specific languages and tools. In *International Conference on Software Reuse (ICSR)*, pages 134–142, 1998.
[53] H. V. Jagadish, J. Gehrke, A. Labrinidis, Y. Papakonstantinou, J. M. Patel, R. Ramakrishnan, and C. Shahabi. Big data and its technical challenges. *Communications of the ACM (CACM)*, 57(7):86–94, July 2014.
[54] N. Jain, S. Mishra, A. Srinivasan, J. Gehrke, J. Widom, H. Balakrishnan, U. Cetintemel, M. Cherniack, R. Tibbets, and S. Zdonik. Towards a streaming SQL standard. In *Conference on Very Large Data Bases (VLDB)*, pages 1379–1390, 2008.
[55] C. S. Jensen and R. Snodgrass. Temporal specialization and generalization. *Transactions on Knowledge and Data Engineering (TKDE)*, 6(6):954–974, 1994.
[56] W. M. Johnston, J. R. P. Hanna, and R. J. Millar. Advances in dataflow programming languages. *ACM Computing Surveys (CSUR)*, 36(1):1–34, 2004.
[57] S. Komazec, D. Cerri, and D. Fensel. Sparkwave: continuous schema-enhanced pattern matching over RDF data streams. In *Conference on Distributed Event-Based Systems (DEBS)*, pages 58–68, 2012.
[58] J. Kreps. Questioning the lambda architecture, 2014. `http://radar.oreilly.com/2014/07/questioning-the-lambda-architecture.html` (Retrieved June 2018).
[59] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja. Twitter Heron: Stream processing at scale. In *International Conference on Management of Data (SIGMOD)*, pages 239–250, May 2015.
[60] P. Le Guernic, T. Gautier, M. Le Borgne, and C. Le Maire. Programming real-time applications with Signal. *Proceedings of the IEEE*, 79(9):1321–1336, 1991.
[61] D. Le-Phuoc, M. Dao-Tran, M.-D. Pham, P. Boncz, T. Eiter, and M. Fink. Linked stream data processing engines: Facts and figures. In *International Semantic Web Conference (ISWC)*, pages 300–312, 2012.
[62] F. Lécué and J. Z. Pan. Predicting knowledge in an ontology stream. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 2662–2669, 2013.
[63] M. Lenzerini. Data integration: A theoretical perspective. In *Symposium on Principles of Database Systems (PODS)*, pages 233–246, 2002.
[64] J. Li, K. Tufte, V. Shkapenyuk, V. Papadimos, T. Johnson, and D. Maier. Out-of-order processing: A new architecture for high-performance stream systems. In *Conference on Very Large Data Bases (VLDB)*, pages 274–288, 2008.
[65] M. H. Linehan, S. Dehors, E. Rabinovich, and F. Fournier. Controlled English language for production and event processing rules. In *Conference on Distributed Event-Based Systems (DEBS)*, pages 149–158, 2011.
[66] E. Meijer, B. Beckman, and G. M. Bierman. LINQ: Reconciling objects, relations, and XML in the .NET framework. In *International Conference on Management of Data (SIGMOD)*, pages 706–706, 2006.
[67] M. Mendell, H. Nasgaard, E. Bouillet, M. Hirzel, and B. Gedik. Extending a general-purpose streaming system for XML. In *Conference on Extending Database Technology (EDBT)*, pages 534–539, 2012.
[68] B. Motik, Y. Nenov, R. E. F. Piro, and I. Horrocks. Incremental update of datalog materialisation: the backward/forward algorithm. In *Conference on Artificial Intelligence (AAAI)*, pages 1560–1568, 2015.
[69] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A timely dataflow system. In *Symposium on Operating Systems Principles (SOSP)*, pages 439–455, Nov. 2013.
[70] S. Muthukrishnan. Data streams: Algorithms and applications. *Foundations and Trends in Theoretical Computer Science*, 1(2):117–236, 2005.
[71] F. Peng and S. S. Chawathe. XPath queries on streaming data. In *International Conference on Management of Data (SIGMOD)*, pages 431–442, 2003.
[72] A. Poggi, D. Lembo, D. Calvanese, G. De Giacomo, M. Lenzerini, and R. Rosati. Linking data to ontologies. *Journal of Data Semantics*, 10:133–173, 2008.
[73] M. Pouzet. *Lucid synchrone, version 3, Tutorial and reference manual*, 2006.
[74] Y. Ren and J. Z. Pan. Optimising ontology stream reasoning with truth maintenance system. In *Conference on Information and Knowledge Management (CIKM)*, pages 831–836, 2011.
[75] A. V. Riabov, E. Bouillet, M. D. Feblowitz, Z. Liu, and A. Ranganathan. Wishful search: Interactive composition of data mashups. In *International Conference on World Wide Web (WWW)*, pages 775–784, 2008.
[76] N. Seyfer, R. Tibbetts, and N. Mishkin. Capture fields: Modularity in a stream-relational event processing language. In *Conference on Distributed Event-Based Systems (DEBS)*, pages 15–22, 2011.
[77] R. Soulé, M. Hirzel, B. Gedik, and R. Grimm. River: An intermediate language for stream processing. *Software – Practice and Experience*, 46(7):891–929, July 2016.
[78] R. Soulé, M. Hirzel, R. Grimm, B. Gedik, H. Andrade, V. Kumar, and K.-L. Wu. A universal calculus for stream processing languages. In *European Symposium on Programming (ESOP)*, pages 507–528, 2010.
[79] R. Stephens. A survey of stream processing. *Acta Informatica*, 34(7):491–541, 1997.
[80] W. Thies, M. Karczmarek, M. Gordon, D. Maze, J. Wong, H. Hoffmann, M. Brown, and S. Amarasinghe. StreamIt: A compiler for streaming applications. Technical Report LCS-TM-622, MIT, 2002.
[81] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy. Storm @Twitter. In *International Conference on Management of Data (SIGMOD)*, pages 147–156, June 2014.
[82] E. D. Valle, D. Dell'Aglio, and A. Margara. Taming velocity and variety simultaneously in big data with stream reasoning. In *Conference on Distributed Event-Based Systems (DEBS) Tutorial*, pages 394–401, 2016.
[83] A. van Wijngaarden, B. Mailloux, J. Peck, C. Koster, M. Sintzoff, C. Lindsey, L. Meertens, and R. Fisker. *Revised Report on the Algorithmic Language ALGOL 68*. 1975.
[84] M. Vaziri, O. Tardieu, R. Rabbah, P. Suter, and M. Hirzel. Stream processing with a spreadsheet. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 360–384, 2014.
[85] S. Wasserkrug, A. Gal, O. Etzion, and Y. Turchin. Complex event processing over uncertain data. In *Conference on Distributed Event-Based Systems (DEBS)*, pages 253–264, 2008.
[86] E. Wu, Y. Diao, and S. Rizvi. High-performance complex event processing over streams. In *International Conference on Management of Data (SIGMOD)*, pages 407–418, 2006.
[87] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Symposium on Operating Systems Principles (SOSP)*, pages 423–438, Nov. 2013.
[88] F. Zemke, A. Witkowski, M. Cherniak, and L. Colby. Pattern matching in sequences of rows. Technical report, ANSI Standard Proposal, 2007.
[89] Q. Zou, H. Wang, R. Soulé, M. Hirzel, H. Andrade, B. Gedik, and K.-L. Wu. From a stream of relational queries to distributed stream processing. In *Conference on Very Large Data Bases (VLDB) Industrial Track*, pages 1394–1405, 2010.