Stream Processing Languages and Abstractions

Martin Hirzel and Guillaume Baudart

1 Synonyms

Stream processing language, streaming language, continuous dataflow language

2 Definition

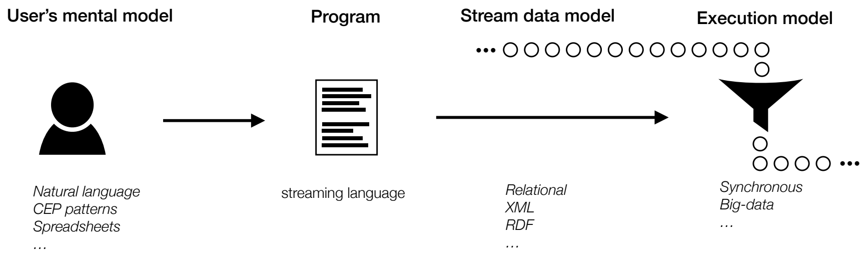

Fig. 1 Stream processing languages.

A *stream processing language* is a programming language for specifying streaming applications. Here, a *stream* is an unbounded sequence of data items, and a *streaming application* is a computer program that continuously consumes input streams and produces output streams. This article surveys recent streaming lan-

Martin Hirzel
IBM Research, Yorktown Heights, NY, USA, e-mail: hirzel@us.ibm.com

Guillaume Baudart
IBM Research, Yorktown Heights, NY, , USA, e-mail: guillaume.baudart@ibm.com

guages designed around the user's mental model, the stream data model, or the execution model, as illustrated in Fig. 1. In addition to specific languages, this article also discusses *abstractions* for stream processing, which are high-level language constructs that make it easy to express common stream processing tasks.

3 Overview

Continuous data streams arise from many directions, including sensors, communications, and commerce. Stream processing helps when low-latency responses are of the essence or when streams are too big to store for offline analysis. Programmers can of course write streaming applications in a general-purpose language without resorting to a dedicated domain-specific language (DSL) for streaming. However, using a streaming language makes code easier to read, write, understand, reason about, modularize, and optimize. Indeed, a suitable streaming language can help developers conceive of a solution to their streaming problems.

This article provides definitions, surveys concepts, and offers pointers for more in-depth study of recent streaming languages. The interested reader may also want to refer to earlier papers for historic perspective: the 1997 survey by Stephens focuses on streaming languages [39], the 2002 survey by Babcock et al. focuses on approximate streaming algorithms [5], and the 2004 survey by Johnston et al. addresses dataflow languages, where streaming is a special case of dataflow [24].

The central abstractions of stream processing are streams, operators, and stream graphs. A *stream* is an unbounded sequence of data items, for example, position readings from a delivery truck. A streaming *operator* is a stream transformer that transforms input streams to output streams. From the perspective of a streaming application, an operator can also have zero input streams (*source*) or zero output streams (*sink*). Finally, a *stream graph* is a directed graph whose nodes are operators and whose edges are streams. Some literature assumes that the shape of stream graphs is restricted, e.g. acyclic, but this article makes no such assumption. While only some streaming languages make the stream graph explicit, others use it as an intermediate representation. For example, the query plan generated from streaming SQL dialects is a stream graph.

The field of streaming languages is diverse and fast-moving. To understand where that diversity comes from, it is instructional to classify streaming languages by their raison d'être. Some streaming languages are based on the attitude that since streams are data-in-motion, data is most central, and the language should be built around a data model (relational, XML, RDF). Other streaming languages focus on the execution model for processing the dataflows efficiently, by enforcing timing constraints or exploiting distributed hardware (synchronous, big-data). A third class of streaming languages focus more on enabling the end user to develop streaming applications in high-level or familiar abstractions (complex events, spreadsheets, or even natural language). Section 4 surveys languages in each of these classes, and Section 5 gives concrete examples for two languages.

4 Findings

This section gives an overview of the field of stream processing languages by surveying eight prominent approaches. Each approach is exemplified by one concrete language. The approaches are grouped along the lines of the previous section into approaches driven by the data model, by the execution model, or by the target user.

4.1 Data-Model Driven Streaming Languages

The success of the **relational** data model for database systems has inspired streaming dialects of the SQL database language. These dialects benefit from developers' familiarity with SQL and from its relational algebra underpinnings. A prominent example is the CQL language, which complements standard relational operators with operators to transform streams into relations and vice versa [2]. CQL lends itself to strong static typing, c.f. Figure 10 of [37]. Efforts towards standardizing streaming SQL focused on clarifying semantic corner-cases [23].

The success of **XML** as a universal exchange format for events and messages has inspired XML-based streaming languages. These languages take advantage of a rich eco-system of XML tools and standards and of the fact that XML documents are self-describing. The languages come in different flavors, from view maintenance over XML updates in NiagaraCQ [14] to languages that process streams where each data item is a (part of an) XML document [16, 30].

The Resource Description Framework (**RDF**) is a versatile data format for integration and reasoning, based on triples of the form $\langle$subject, predicate, object$\rangle$. A popular language for querying static RDF knowledge bases is SPARQL [32], and C-SPARQL [6] extends SPARQL for continuous queries, just like CQL extends SQL. A stream is a sequence of timestamped triples, but a query can also return a graph by emitting multiple triples with the same timestamp.

4.2 Execution-Model Driven Streaming Languages

Dataflow **synchronous** languages [7] were introduced in the late 80s as domain specific languages for the design of embedded control systems. A dataflow synchronous program executes in a succession of discrete steps, and each step is assumed to be instantaneous (the synchronous hypothesis). A programmer writes high-level specifications in the form of stream functions specifying variable values at each step or instant. Section 5.1 illustrates this approach with the language Lustre [11].

Streaming **big data** is motivated by the "4 Vs": a lot of data (volume) streams quickly (velocity) into the system, which must deal with diverse data and functionality (variety) and with uncertainty (veracity). Languages for big data streaming let users specify an explicit stream graph that can be easily distributed with minimal synchronization, and are extensible by operators in widely adopted general-purpose languages. Section 5.2 elaborates on this for the concrete example of SPL [20].

4.3 Target-User Driven Streaming Languages

Complex event processing, or **CEP**, lets users compose events hierarchically to span the gap between low-level and high-level concepts. There are various pattern languages for CEP that compile to finite automatons. Recognizing this, the MATCH-RECOGNIZE SQL extension proposal simply adopts familiar regular expressions as the CEP pattern language [44]. While the SQL basis focuses on a relational model, regular expressions can also be used for CEP in big-data streaming [18].

Since there are many more **spreadsheet** users than software developers, a spreadsheet-based streaming language could reach more target users. Furthermore, spreadsheets are reactive: changes trigger updates to dependent formulas. ActiveSheets hooks up some spreadsheet cells to input or output streams, with normal spreadsheet formulas in between [42]. When the two-dimensional spreadsheet data model is too limiting, it can be augmented with windows and partitioning [19].

A streaming language based on **natural language** might reach the maximum number of target users. However, since natural language is ambiguous, a controlled natural language (CNL) is a better choice [26]. For instance, the language for META is a CNL for specifying event-condition-action rules, temporal predicates, and data types [4]. The data model includes events and entities with nested concepts, and can be shown to be equivalent to the nested-relational model [35].

5 Examples

This section gives details and concrete code examples for two out of the eight approaches for languages surveyed in the previous section: the synchronous dataflow approach exemplified by Lustre [11] and the big-data streaming approach exemplified by SPL [20].

When it comes to implementing streaming languages, there is a spectrum from basic to sophisticated techniques. At the basic end of the spectrum are configuration files in some existing markup format such as XML. The streaming engine interprets the configuration file to construct and then execute a stream graph. An intermediate point is a domain-specific embedded language (EDSL or sometimes DSEL) [21]. As the name implies, an EDSL is a domain-specific language (DSL) that is embedded in some host language, typically a general-purpose language (GPL). The line between simple libraries and EDSLs is blurred, but in general, EDSLs encourage a more idiomatic programming style. Recently, EDSLs have gained popularity as several GPLs have added features that make them more suitable for hosting EDSLs. At the sophisticated end of the spectrum are full-fledged, stand-alone DSLs with their own syntax, compiler, and other tools. While stand-alone streaming DSLs are not embedded in a GPL, they often interface with a GPL, e.g., for user-defined operators.

For clarity of exposition, the following examples use stand-alone streaming languages. Stand-alone languages are the norm for synchronous dataflow, because self-contained code is easier to reason about. On the other hand, for big-data streaming, EDSLs that specify an explicit stream graph are popular, because they are easier to implement. But as the example below illustrates, once implemented, stand-alone languages also have advantages for big-data streaming.

5.1 Synchronous Dataflow in Lustre

Synchronous dataflow languages were introduced to ease the design and certification of embedded systems by providing a well-defined mathematical framework that combines a logical notion of time and deterministic concurrency. It is then possible to formally reason about the system, simulate it, prove safety properties, and generate embedded code. The synchronous dataflow language Lustre is the backbone of the industrial language and compiler Scade [15] routinely used to program embedded controllers in many critical applications.

```
node counter (init, incr: int; reset: bool) returns (n: int);
var pn: int;
let
  pn = init -> pre n;
  n = if reset then init else pn + incr;
tel

node tracker (speed, limit: int) returns (t: int);
var x: bool; cpt: int when x;
let
  x = (speed > limit);
  cpt = counter((0, 1, false) when x);
  t = current(cpt);
tel
```

speed	28	29	32	30	44	53	58	48	33	28	29	...
limit	30	30	30	30	55	55	55	30	30	30	30	...
x	F	F	T	F	F	F	T	T	T	F	F	...
cpt			1				2	3	4			...
t	0	0	1	1	1	1	2	3	4	4	4	...

Fig. 2 Lustre code example with a possible execution.

In Lustre a 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. Consider the example of Fig. 2 adapted from [9]. The function *counter* takes three input streams, two integer streams *init* and *incr* and one boolean stream *reset*. It returns the cumulative sum of the values of *incr* initialized with *init* and similarly reset when *reset* is *true* (Line 5). The variable *pn* (Line 4) stores the value of the counter *n* at the previous step using the initialization operator (−>) and the non-initialized delay **pre**.

A stream is not necessarily defined at each step. The clock of a stream is a boolean sequence giving the instants where it is defined. Streams with different clocks can be combined via sampling (**when**) or stuttering (**current**). For instance, the *tracker* function of Fig. 2 tracks the number of times the speed of a vehicle exceeds the speed limit. The **when** operator samples a stream according to a boolean condition. The function *counter* is thus only activated when *x* is *true* (Line 12). The **current** operator completes a stream with the last defined value when it is not present (Line 13). The value of *t* is thus sustained when *x* is *false*. The execution of such a program can be represented as a timeline, called a chronogram (illustrated in Fig. 2), showing the sequence of values taken by its streams at each step.

Specific compilation techniques for synchronous languages exist to generate efficient and reliable code for embedded controllers. Compilers produce imperative code that can be executed in a control loop 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.

Since the seminal dataflow languages Lustre [11] and Signal [27], multiple extensions of the dataflow synchronous model were proposed. Lucid Synchrone [31] combines the dataflow synchronous approach with functional features à la ML, the n-synchronous model [28] relaxes the synchronous hypothesis by allowing communication with bounded buffers, and Zélus [10] is a Lustre-like language extended with ordinary differential equations to define continuous-time dynamics.

Recent efforts focus on the compilation, verification, and test of dataflow synchronous programs. New techniques have been proposed to compile Lustre programs for manycore systems [34] or improve the computation of the Worst Case Execution Time (WCET) of the compiled code [8, 17]. Kind2 [12] is a verification tool based on SMT solvers to model-check Lustre programs and the Vélus compiler [9] tackles the problem of verifying the compiler itself using a proof assistant. Lutin [33] (and its industrial counterpart, the Argosim Stimulus tool [3]) is a DSL to design non-deterministic test scenarios for Lustre programs.

5.2 Big-Data Streaming in SPL

Big-data streaming languages are designed to handle high-throughput streams while at the same time being expressive enough to handle diverse data formats and streaming operators. A popular way to address the requirement of high throughput is to make it easy to execute the streaming application not just on a single core or even a single computer, but on a cluster of computers. And a popular way to address the requirement of high expressiveness is to make it easy for programmers to define new streaming operators, possibly using a different programming language than the stream processing language they use for composing operators into a graph.

SPL is a big-data streaming language designed for distribution and extensibility [20]. It was invented in 2009 and is being actively used in industry [22]. An SPL program is an explicit stream graph of streams and operators. Unlike dataflow synchronous languages, and like other big-data streaming languages, SPL uses only minimal synchronization: an operator can fire whenever there is data available on any of its input ports, following semantics formalized in the Brooklet calculus [38]. Since synchronization across different cores and computers is hard to do efficiently, reducing synchronization simplifies distribution, giving the runtime system more flexibility for which operators to co-locate in the same core or computer. There is no assumption of simultaneity between different operator firings. When downstream operators cannot keep up with the data rate, they implicitly throttle upstream operators via back-pressure.

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

Fig. 3 SPL code example.

Fig. 3 shows an example SPL program. Line 1 defines a stream *Calls* as the output of invoking an operator *CallsSource*. In SPL, streams carry tuples that are strongly and statically typed and whose fields can hold simple numbers or strings as in the example, but can also hold nested lists and tuples. The *CallsSource* operator has no further configuration (empty curly braces); for this example, assume it is user-defined elsewhere. Programmers can define their own operators either in SPL or in other languages such as C++ or Java. Lines 3–8 define a stream *CallStats* by invoking an operator *Aggregate*. The code configures the operator with an input stream *Calls*, with a **window** clause for a 24-hour sliding window with 1-minute granularity, and with an **output** clause. While many operators support these and other clauses, they can also contain code restricted to the operator at hand. The *Aggregate* operator in SPL's standard library supports intrinsic functions for *Max*, *MaxCount*, and various other aggregations. Programmers can extend SPL with new operators that, like *Aggregate*, support various configurations and operator-specific intrinsic functions.

SPL was influenced by earlier big-data streaming systems such as Borealis [1] and TelegraphCQ [13], generalizing them to be less dependent on relational data and more extensible. Various other streaming systems after SPL, such as Storm [41] and Spark Streaming [43], also target big-data streaming. Like SPL, they use explicit stream graphs as their core abstraction, but unlike SPL, they use embedded (not stand-alone) domain-specific languages.

While the examples of Lustre and SPL draw a stark contrast between synchronous dataflow and big-data streaming, there are also commonalities. For instance, the StreamIt language is synchronous, but like big-data streaming languages, it uses an explicit stream graph as its core abstraction [40]. And Soulé et al. show how to reduce the dependence of StreamIt on synchrony [36].

6 Future Directions for Research

The landscape of streaming languages is far from consolidating on any dominant approach. New languages keep coming out to address a variety of open issues. One active area of research is the interaction between streams (data in motion) and state (data at rest). While CQL gave a conceptually clean answer [2], people are debating alternative approaches, such as the Lambda Architecture [29] and the Kappa Architecture [25]. Another active area of research is how to handle uncertainty, such as out-of-order data, missing fields, erroneous sensor readings, approximate algorithms, or faults. On this front, streaming languages have not yet reached the clarity of databases with their ACID properties. When it comes to implementation strategies, there has been a recent surge in embedded domain-specific languages. But while EDSLs have fewer tooling needs and are less intimidating for users familiar with their host language, they are less self-contained and offer less static optimization and error-checking than stand-alone languages. We hope this article inspires innovation in streaming languages that are well-informed by those that came before.

7 Cross-References

Continuous Queries, Languages for Big Data analysis

References

1. Abadi, D.J., Ahmad, Y., Balazinska, M., Cetintemel, U., Cherniack, M., Hwang, J.H., Lindner, W., Maskey, A.S., Rasin, A., Ryvkina, E., Tatbul, N., Xing, Y., Zdonik, S.: The design of the

Borealis stream processing engine. In: Conference on Innovative Data Systems Research (CIDR), pp. 277–289 (2005)
2. Arasu, A., Babu, S., Widom, J.: The CQL continuous query language: Semantic foundations and query execution. Journal on Very Large Data Bases (VLDB J.) **15**(2), 121–142 (2006)
3. Argosim: Stimulus (2015). http://argosim.com/ (Retrieved November 2017)
4. Arnold, M., Grove, D., Herta, B., Hind, M., Hirzel, M., Iyengar, A., Mandel, L., Saraswat, V., Shinnar, A., Siméon, J., Takeuchi, M., Tardieu, O., Zhang, W.: META: Middleware for events, transactions, and analytics. IBM Journal of Research & Development **60**(2–3), 15:1–15:10 (2016)
5. Babcock, B., Babu, S., Datar, M., Motwani, R., Widom, J.: Models and issues in data stream systems. In: Symposium on Principles of Database Systems (PODS), pp. 1–16 (2002)
6. Barbieri, D.F., Braga, D., Ceri, S., Della Valle, E., Grossniklaus, M.: C-SPARQL: SPARQL for continuous querying. In: Poster at International World Wide Web Conferences (WWW-Poster), pp. 1061–1062 (2009)
7. Benveniste, A., Caspi, P., Edwards, S.A., Halbwachs, N., Le Guernic, P., De Simone, R.: The synchronous languages 12 years later. Proceedings of the IEEE **91**(1), 64–83 (2003)
8. Bonenfant, A., Carrier, F., Cassé, H., Cuenot, P., Claraz, D., Halbwachs, N., Li, H., Maiza, C., De Michiel, M., Mussot, V., Parent-Vigouroux, C., Puaut, I., Raymond, P., Rohou, E., Sotin, P.: When the worst-case execution time estimation gains from the application semantics. In: European Congress on Embedded Real-Time Software and Systems (ERTS2) (2016)
9. Bourke, T., Brun, L., Dagand, P., Leroy, X., Pouzet, M., Rieg, L.: A formally verified compiler for lustre. In: Conference on Programming Language Design and Implementation (PLDI), pp. 586–601 (2017)
10. Bourke, T., Pouzet, M.: Zélus: A synchronous language with odes. In: Conference on Hybrid Systems: Computation and Control (HSCC), pp. 113–118 (2013)
11. Caspi, P., Pilaud, D., Halbwachs, N., Plaice, J.A.: LUSTRE: a declarative language for real-time programming. In: Symposium on Principles of Programming Languages (POPL), pp. 178–188 (1987)
12. Champion, A., Mebsout, A., Sticksel, C., Tinelli, C.: The Kind 2 model checker. In: Conference on Computer Aided Verification (CAV), pp. 510–517 (2016)
13. Chandrasekaran, S., Cooper, O., Deshpande, A., Franklin, M.J., Hellerstein, J.M., Hong, W., Krishnamurthy, S., Madden, S., Raman, V., Reiss, F., Shah, M.A.: TelegraphCQ: Continuous dataflow processing for an uncertain world. In: Conference on Innovative Data Systems Research (CIDR) (2003)
14. Chen, J., DeWitt, D.J., Tian, F., Wang, Y.: NiagaraCQ: A scalable continuous query system for internet databases. In: International Conference on Management of Data (SIGMOD), pp. 379–390 (2000)
15. Colaco, J.L., Pagano, B., Pouzet, M.: Scade 6: A Formal Language for Embedded Critical Software Development. In: International Symposium on Theoretical Aspect of Software Engineering (TASE) (2017)
16. Diao, Y., Fischer, P.M., Franklin, M.J., To, R.: YFilter: Efficient and scalable filtering of XML documents. In: Demo at International Conference on Data Engineering (ICDE-Demo), pp. 341–342 (2002)
17. Forget, J., Boniol, F., Pagetti, C.: Verifying end-to-end real-time constraints on multi-periodic models. In: International Conference on Emerging Technologies And Factory Automation (ETFA) (2017)
18. Hirzel, M.: Partition and compose: Parallel complex event processing. In: Conference on Distributed Event-Based Systems (DEBS), pp. 191–200 (2012)
19. Hirzel, M., Rabbah, R., Suter, P., Tardieu, O., Vaziri, M.: Spreadsheets for stream processing with unbounded windows and partitions. In: Conference on Distributed Event-Based Systems (DEBS), pp. 49–60 (2016)
20. Hirzel, M., Schneider, S., Gedik, B.: SPL: An extensible language for distributed stream processing. Transactions on Programming Languages and Systems (TOPLAS) **39**(1), 5:1–5:39 (2017)

21. Hudak, P.: Modular domain specific languages and tools. In: International Conference on Software Reuse (ICSR), pp. 134–142 (1998)
22. IBM: IBM Streams (2008). https://ibmstreams.github.io (Retrieved November 2017)
23. Jain, N., Mishra, S., Srinivasan, A., Gehrke, J., Widom, J., Balakrishnan, H., Cetintemel, U., Cherniack, M., Tibbets, R., Zdonik, S.: Towards a streaming SQL standard. In: Conference on Very Large Data Bases (VLDB), pp. 1379–1390 (2008)
24. Johnston, W.M., Hanna, J.R.P., Millar, R.J.: Advances in dataflow programming languages. ACM Computing Surveys (CSUR) **36**(1), 1–34 (2004)
25. Kreps, J.: Questioning the Lambda architecture (2014). http://radar.oreilly.com/2014/07/questioning-the-lambda-architecture.html (Retrieved November 2017)
26. Kuhn, T.: A survey and classification of controlled natural languages. Computational Linguistics **40**(1), 121–170 (2014)
27. Le Guernic, P., Gautier, T., Le Borgne, M., Le Maire, C.: Programming real-time applications with SIGNAL. Proceedings of the IEEE **79**(9), 1321–1336 (1991)
28. Mandel, L., Plateau, F., Pouzet, M.: Lucy-n: a n-synchronous extension of Lustre. In: International Conference on Mathematics of Program Construction (MPC), pp. 288–309 (2010)
29. Marz, N.: How to beat the CAP theorem (2011). http://nathanmarz.com/blog/how-to-beat-the-cap-theorem.html (Retrieved November 2017)
30. Mendell, M., Nasgaard, H., Bouillet, E., Hirzel, M., Gedik, B.: Extending a general-purpose streaming system for XML. In: Conference on Extending Database Technology (EDBT), pp. 534–539 (2012)
31. Pouzet, M.: Lucid synchrone, version 3, Tutorial and reference manual (2006)
32. Prud'hommeaux, E., Seaborne, A.: SPARQL query language for RDF. W3C Recommendation (2008). http://www.w3.org/TR/rdf-sparql-query/ (Retrieved November 2017)
33. Raymond, P., Jahier, E.: Lutin Reference manual Version Trilby-1.54 (2013)
34. Rihani, H., Moy, M., Maiza, C., Davis, R.I., Altmeyer, S.: Response time analysis of synchronous data flow programs on a many-core processor. In: Conference on Real-Time Networks and Systems (RTNS), pp. 67–76 (2016)
35. Shinnar, A., Siméon, J., Hirzel, M.: A pattern calculus for rule languages: Expressiveness, compilation, and mechanization. In: European Conference on Object-Oriented Programming (ECOOP), pp. 542–567 (2015)
36. Soulé, R., Gordon, M.I., Amarasinghe, S., Grimm, R., Hirzel, M.: Dynamic expressivity with static optimization for streaming languages. In: Conference on Distributed Event-Based Systems (DEBS), pp. 159–170 (2013)
37. Soulé, R., Hirzel, M., Gedik, B., Grimm, R.: River: An intermediate language for stream processing. Software – Practice and Experience (SP&E) **46**(7), 891–929 (2016)
38. Soulé, R., Hirzel, M., Grimm, R., Gedik, B., Andrade, H., Kumar, V., Wu, K.L.: A universal calculus for stream processing languages. In: European Symposium on Programming (ESOP), pp. 507–528 (2010)
39. Stephens, R.: A survey of stream processing. Acta Informatica **34**(7), 491–541 (1997)
40. Thies, W., Karczmarek, M., Gordon, M., Maze, D., Wong, J., Hoffmann, H., Brown, M., Amarasinghe, S.: StreamIt: A compiler for streaming applications. Tech. Rep. LCS-TM-622, MIT (2002)
41. Toshniwal, A., Taneja, S., Shukla, A., Ramasamy, K., Patel, J.M., Kulkarni, S., Jackson, J., Gade, K., Fu, M., Donham, J., Bhagat, N., Mittal, S., Ryaboy, D.: Storm @Twitter. In: International Conference on Management of Data (SIGMOD), pp. 147–156 (2014)
42. Vaziri, M., Tardieu, O., Rabbah, R., Suter, P., Hirzel, M.: Stream processing with a spreadsheet. In: European Conference on Object-Oriented Programming (ECOOP), pp. 360–384 (2014)
43. Zaharia, M., Das, T., Li, H., Hunter, T., Shenker, S., Stoica, I.: Discretized streams: Fault-tolerant streaming computation at scale. In: Symposium on Operating Systems Principles (SOSP), pp. 423–438 (2013)
44. Zemke, F., Witkowski, A., Cherniak, M., Colby, L.: Pattern matching in sequences of rows. Tech. rep., ANSI Standard Proposal (2007)

Abstract

Stream processing languages are programming languages for writing streaming applications, i.e., computer programs for continuously processing data streams that are conceptually infinite. While such applications could also be written in general-purpose languages, using a domain-specific language for streaming improves programmer productivity. There is a wide variety of stream processing languages catering to different mental models, data models, and execution models. This article surveys recent stream processing languages, along with their core abstractions and design rationale.