IBM Research Report

SPL: An Extensible Language for Distributed Stream Processing

Martin Hirzel, Scott Schneider
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598
USA

Bugra Gedik
Bilkent University
Ankara 06800
Turkey
SPL: An Extensible Language for Distributed Stream Processing

MARTIN HIRZEL, IBM T.J. Watson Research Center
SCOTT SCHNEIDER, IBM T.J. Watson Research Center
BUĞRA GEDİK, Bilkent University

Big data is revolutionizing how all sectors of our economy do business, including telco, transportation, medical, and finance. Big data comes in two flavors: data at rest and data in motion. Processing data in motion is stream processing. Stream processing for big data analytics often requires scale that can only be delivered by a distributed system, exploiting parallelism on many hosts and many cores. To address this need, IBM built InfoSphere® Streams, a distributed stream processing platform. Early customer experience with Streams uncovered that another core requirement is extensibility, since customers want to build high-performance domain-specific operators for use in their streaming applications. Based on these two core requirements of distribution and extensibility, we designed and implemented a stream processing language called SPL. This paper describes SPL with an emphasis on the language design, distributed runtime, and extensibility mechanism. SPL is now the gateway for the Streams platform, used by our internal (research) and external (industry) customers for stream processing in a broad range of application domains.

Categories and Subject Descriptors: D.3.4 [Programming Languages]: Processors—compilers
General Terms: Languages
Additional Key Words and Phrases: Stream Processing

1. INTRODUCTION

The problem statement for this paper is to design a streaming language for big data. The characteristic features of big data are commonly known as the 3 Vs: volume, velocity, and variety. Handling data at high volume requires a cluster of machines to exploit compute and storage beyond that of a shared-memory multi-core. The velocity requirement is central to streaming, where data must be processed at high throughput and low latency. Data comes in a variety of structured and unstructured formats, creating a demand for streaming operators that parse and convert data on the fly. This paper explores programming language techniques for addressing these 3 Vs.

The databases community has addressed streaming by extending SQL (e.g., CQL [Arasu et al. 2006]). SQL-based streaming languages have tidy semantics, but focus on classic relational operators. We argue that properly addressing variety re-
quires a language that is extensible with arbitrary operators. Where the programming languages community has dealt with streaming, it focused mostly on synchronous dataflow (SDF [Lee and Messerschmitt 1987], e.g., StreamIt [Gordon et al. 2006]). While SDF offers attractive static guarantees, those come with restrictions on dynamism and topology. Most importantly, SDF interacts poorly with native code in a non-streaming language. Native code is central to the variety requirement. Hence, some recent streaming systems use libraries instead of languages to implement their programming model [Zaharia et al. 2013; Toshniwal et al. 2014].

This paper describes the SPL language. SPL supports distribution on a cluster and extension with new operators. It serves as the programming language for IBM InfoSphere Streams (Streams for short)—a commercial distributed stream processing platform. An earlier language for Streams was SPADE, which centered around built-in relational operators with limited support for user-defined operators [Gedik et al. 2008]; in contrast, SPL offers a general code-generation framework for all operators. An earlier paper about SPL offered a high-level overview [Hirzel et al. 2013]; in contrast, this paper presents the full language design, along with case studies and the details on extensibility.

To facilitate distribution, SPL operators communicate only via streams. The language avoids shared state or even any centralized execution scheduling. The source code offers a logical abstraction that hides distribution, and the runtime is in charge of mapping from this logical level to the distributed hardware at hand. This mapping offers many optimization opportunities, which users can influence if they so wish, or the system can automatically optimize. The SPL source code describes the stream graph and configures operators declaratively. The extension mechanism allows developers to define new operators that offer a declarative interface at the SPL language level, but use code-generation templates for native code at the implementation level. An operator model specifies an interface and properties that enable the SPL compiler to do static checking and optimization in the presence of generated native code.

This paper describes the novel features that set SPL apart:

— Language-level graph abstractions and restrictions on data and control dependencies that facilitate distribution.
— A uniform high-level declarative syntax for all operator invocations, including those of user-defined operators.
— An extension mechanism, where operators are mini-compilers generating customized native code.

SPL has had success both commercially and academically. Commercially, SPL is used by customers for a wide variety of application domains [Biem et al. 2010b,a; Bouillet et al. 2012; Kienzler et al. 2012; Park et al. 2012; Sow et al. 2012; LogMon 2014; Zou et al. 2011]. Academically, several papers are based on new stream processing techniques that were first prototyped on a research branch of the SPL compiler [De Pauw et al. 2010; Gedik et al. 2008, 2014; Hirzel 2012; Hirzel and Gedik 2012; Khandekar et al. 2009; Mendell et al. 2012; Schneider et al. 2012; Tang and Gedik 2013]. While those papers describe facets of SPL in isolation, this paper describes the language in its entirety.

2. LANGUAGE OVERVIEW

This section explains language features and provides the rationale for the more surprising design choices.
SPL: An Extensible Language for Distributed Stream Processing

2.1. Stream Graphs

Stream graphs as a programming model are both easy to understand for users, and lend themselves to a parallel and distributed implementation. SPL encourages programmers to think of their applications as graphs by dedicating syntax to this concept.

Fig. 1 shows an example stream graph alongside the corresponding SPL code. Each edge is a stream (a conceptually infinite sequence of data items), and each vertex is an operator instance. The program enriches streams of patient identifiers from two TCP sources with patient profiles from a database source, and sends the resulting stream to a TCP sink. Enrichment here simply means a join of data in motion (Ids) with data at rest (Profiles). One tweak is that the Ctrl stream from ProfileEnricher back to Switch delays the Ids while the Profiles are being initialized.

The same operator can be instantiated multiple times in the same stream graph (e.g., there are two instances of TCPSource). Operator instances are named by output streams, or using an as id clause (e.g., Snk).

The point where a stream connects to an operator is a port. Operators can have zero, one, or multiple input and output ports. Multiple streams can arrive on the same input port (e.g., both Src0 and Src1 arrive on the same input port of the Switch instance), which merges them in arrival order. Syntactically, SPL separates ports by semicolons and streams converging on the same port by commas. A stream from an output port can be used multiple times, yielding copies of the same sequence of data items.

When a data item arrives at an input port, the corresponding operator instance fires. Since firings have no central schedule, they maximize concurrency and minimize distributed coordination. Most operators are passive between firings, but there are also self-activating operators, including sources (operators without input ports). When an operator instance fires, it consumes the data item that triggered the firing, and produces zero or more data items on output ports. Selectivity is the number of output data items per input data item. Selectivity is often dynamic and unknowable for the compiler. For example, many SPL applications use data-dependent filtering, parsing, or time-based aggregation.

SPL operators can be stateful, remembering information between firings. While most SPL applications have some stateful operator instances, many operators are stateless. In contrast to operator-local state, SPL offers no features for sharing state between op-
operators. This omission facilitates distribution and avoids race conditions or deadlocks from shared state.

There are alternatives to SPL's execution model of firing operators each time a data item arrives on any input port. Operators in Kahn networks wait on a specific port whereas SPL operators wait on all ports [Kahn 1974]. In synchronous dataflow, one firing can consume multiple data items [Lee and Messerschmitt 1987]. And in CQL (an SQL dialect for streaming), each firing consumes one data item per input port [Arasu et al. 2006]. SPL's execution model is general in that operator developers can emulate any of the other models by a combination of state and blocking.

2.2. Streams

All inter-operator communication happens via streams. As mentioned earlier, a stream is a sequence of data items. A data item is either a tuple or a punctuation. A tuple is a value of a tuple type, which has named attributes, similar to a C struct, a Pascal record, or a database row (but potentially nested). For example, stream Profiles in Fig. 1 carries tuples of type Patient. A punctuation is a control signal marking a position in a stream. Streams are ordered, and window punctuations are commonly used by programmers to group subsequences of tuples into a window. Final punctuations signal that the job is about to shut down.

As seen in Fig. 1, each output port of an operator instance defines a stream, which can then feed into input ports of other operator instances. To address the full variety of application requirements, SPL poses no restrictions on the resulting topology. SPL allows multiple sources (e.g., primary input vs. control input), multiple sinks (e.g., primary output vs. log data), and even cycles (e.g., to send back control messages). To help avoid potential problems with cycles, SPL supports control ports: an operator firing on a control port is not supposed to submit output data items. The compiler warns when a cycle does not end in a control port.

The snapshot of a stream graph edge at a given point in time can be viewed as a FIFO buffer of in-transit data items. SPL does not specify how this buffer is implemented, or how much time each data item spends in it, except that order is preserved. This enables a flexible placement of operator instances on threads, processes, and hosts: in the general case, SPL runs on a distributed system without centralized scheduling. Downstream operators indirectly throttle the processing rate of upstream operators via back-pressure.

The SPL compiler does not statically know bounds on buffer sizes. The SPL runtime does impose a fixed capacity on buffers, and when buffers fill up, they exert back-pressure. Execution models where an operator is picky about which input port to receive data from, while blocking other ports even if they have data available, can cause deadlocks [Li et al. 2010]. In SPL's execution model, operators fire when data is available on any port. Therefore, in SPL, deadlocks can only happen when users emulate other execution models via blocking operators whose firings can block for an indeterminate amount of time [Xu et al. 2013]. This is rarely a problem in practice, and can be resolved using SPL's interactive debugger [De Pauw et al. 2010].

2.3. Operator Invocations

An operator is a reusable and configurable stream transformer. An operator invocation is the source code that configures an operator to yield an operator instance in the stream graph. Operator invocations have five optional clauses: logic, window, param, output, and config. The first four of these clauses affect operator semantics; this section offers examples and explanations for them. The last clause, config, contains non-functional directives to the compiler or runtime system to influence optimization decisions or debugging support. The available directives are implementation specific; Sec-
Example 1. Fig. 2 is an example application that reads stock bids from an external source, computes aggregate statistics for those bids, and streams those statistics to an external sink. We focus on the framed operator invocation, which performs the aggregation. Line 6 is the operator invocation head, declaring the output stream type (BidStat) and name (BidStats), the operator to be invoked (Custom), and the stream in the input port (Bids). The operator invocation contains a logic clause with two sub-clauses. The state subclause defines variables that are locally scoped to the operator invocation and whose lifetime is that of the entire application. The onTuple subclause defines code to be executed for each tuple arriving on the specified input port. In this case, it incrementally updates the aggregate statistics and submits them to the output port BidStats. While many SPL operators support the logic clause, it is most commonly used on Custom. The Custom operator is special in that it allows programmers to directly call submit from within its logic clause. Calls to submit send data items to the specified output port. While most operators implement core functionality in C++ or Java, Custom is a blank slate for writing logic directly in SPL. The logic clause also supports an onPunct subclause for specifying code to execute upon receiving a punctuation on an input port.

Example 2. The Custom operator is convenient for defining specific logic in-place, and in practice, real SPL applications contain many invocations of the Custom operator. However, the reusability and customizability of such invocations is limited. For exam-
ple, Fig. 2 aggregates over all tuples on a stream during the entire application lifetime. Such unbounded aggregations are rare in practice. More common are aggregations over a particular window of tuples. In fact, computing some kind of aggregation over a particular window of tuples is so common in streaming applications that SPL's standard library defines the Aggregate operator for this purpose. Fig. 3 shows an example invocation of the Aggregate operator, which could replace the framed portion of Fig. 2.

The Aggregate invocation in Fig. 3 shows three additional operator clauses: window, param and output. The Aggregate operator definition is a generic aggregation template, and the configurations in the clauses specialize the invocation for specific behavior.

The window clause in Fig. 3 specifies that the contents of the window should tumble every 3 seconds, and that the window is partitioned. In general, the window clause declares an operator-instance local FIFO buffer of tuples that recently arrived on an input port. Streams are conceptually infinite, but practical programs work on bounded space. Therefore, most streaming languages offer windows, as they are an intuitive way to bound required data [Arasu et al. 2006; Gordon et al. 2006; Zaharia et al. 2013]. A tumbling window clears out its contents between firings. A sliding window evicts only a subset of its contents, making room for new tuples but retaining some old ones. Windows that are partitioned maintain separate buffers and firings for each distinct value of user-specified key attributes.

The param clause in Fig. 3 contains the partitionBy parameter, which specifies the window partitioning key as the ticker attribute. In general, the param clause configures operator-specific parameters. Configuring the param clause is the primary way for programmers to specialize an operator's behavior upon an invocation.

The output clause in Fig. 3 specifies how to assign values to an output tuple’s attributes. Operator definitions determine when to submit new tuples based on the semantics of their operation; for instance, Aggregate submits an output tuple for every aggregation result. The output clause can exist on each output port, and it is how programmers who invoke that operator specialize the resulting tuple. When there is no explicit assignment for an output attribute, the compiler inserts an assignment copying a corresponding input attribute, if the name matches unambiguously and has the same type. The output clause in Fig. 3 also uses two operator-specific intrinsic functions, Sum and ArgMax, which produce the total and the bidder with the highest bid. While calls to operator-specific intrinsics look like ordinary function calls, the operator code generator does not have to implement them that way. For instance, the Aggregate operator implements tumbling-window aggregation incrementally, as opposed to computing the aggregate result in bulk by looping over the window contents.

Example 3. Fig. 4 shows an operator invocation that determines when to make a sale based on joining bids and asks. While the invocation in Fig. 4 contains the same clauses as the invocation in Fig. 3, they configure the different semantics of a different operator. Line 1 defines the output stream Sales by invoking the Join operator, which accepts two input streams, Bids and Asks. The operator instance maintains a window over the Bids stream. Fig. 5 illustrates the semantics. Each time the operator instance receives an Asks tuple, it compares it against each tuple currently in the Bids window by executing the match predicate. For each successful match, the operator instance: assigns attributes of the output tuple using the output clause; forwards values from the input tuple to the output tuple for matching attributes that were unmentioned in the output clause; and submits the output tuple.

The window clause in Fig. 4 is over only one of the input ports, Bids. Unlike the Aggregate invocation in Fig. 3, the Join invocation in Fig. 4 has multiple input ports. Fig. 4 uses a sliding window of tuples whose ts attribute differs by no more than 30 ($delta(ts, 30.0)$), with a sliding granularity of a single tuple ($count(1)$). Windows
make data from one port available during firings on another port, as seen in the interaction between the window clause and the match predicate. Such interaction is necessary for implementing any join-like operation with windows. SPL’s execution model fires operators when a new data item arrives on any particular input port, and correlating data across input ports may require looking at a different port’s window.

The param clause in Fig. 4 passes an expression to the predicate match. The expression for match gets re-executed (possibly multiple times) during each firing to compare the new tuple against tuples in the window. SPL supports different parameter passing modes. In general, the operator implementation determines whether and when such expression parameters execute (in contrast to logic clauses, which always execute at the start of a firing). Besides expression parameters, operators can also declare parameters that are only evaluated once before the application runs. For example, Line 6 in Fig. 4 uses constant values for role and address. And in Fig. 3, the parameter partitionBy accepts a list of tuple attributes to use as keys; the parameter has no concept of executing. The operator model specifies parameter names, types, modes, and multiplicities.

Discussion. The Aggregate operator invocation in Fig. 3 uses the same clauses as the Join operator invocation in Fig. 4. However, despite using the same clauses, they are able to configure different operations in non-trivial ways. The interfaces to both operators are essentially embedded domain-specific languages, in the sense that they borrow host language syntax and types [Hudak 1998]. They implement different semantics, for streaming aggregations and streaming joins, respectively. The design of SPL’s operator invocation syntax and clause structure accomplishes two goals. First, it is uniform across operators: once a user has mastered it, they know how to invoke any
operator. Second, it serves as the foundation of SPL’s extensibility: operators use code generation to specialize their code to the declarative operator configuration.

2.4. Conventional Language Features

While SPL uses new syntax for streams and operators, it borrows syntax from conventional languages for concepts such as expressions, functions, and variable declarations. Here, by conventional, we mean not specific to streaming. SPL reuses features from C and Java (syntax style), SQL (tuples), Python (built-in lists and maps), ML (parametric polymorphism), and others, making it more familiar and thus easier to learn. This reuse also leverages established practices and hard-earned lessons in areas where SPL does not intend to innovate. At the same time, there were frequently many design choices to pick from, and the streaming context informed those decisions.

To address the variety of streaming in big data, besides the usual primitive types (numbers, strings, booleans, etc.), SPL offers four generic type constructors: tuple, list, map, and set. Streams carry tuples, but tuple types can also be used like any other type for variables, parameters, function return values, or even attributes of other tuple types. Lists, maps, and sets are homogeneous collections. Lists are dynamic arrays indexed by integers; maps support efficient associative lookup and are indexed by any key type; and sets are unordered collections without repetition. In stream processing, establishing the exact size of data items can speed up serialization and transport. Therefore, SPL offers bounded variants of its variable-sized string and collection types; e.g., `list<int32>[4]` is the type for lists of up to 4 `int32`s.

SPL is strongly and statically typed to catch as many errors as possible at compile time and avoid dynamic dispatch overheads. On the other hand, SPL’s type constructors make working with types easy. SPL provides JSON-inspired literal syntax for values of each type constructor. SPL uses structural equivalence, because types are often written in-place (e.g., Line 1 of Fig. 4).

A stream type is parameterized with a tuple body. There are two ways to specify a tuple body. One is by a sequence of attributes. For instance, `stream<float64> P2 loc > S` defines a stream S, where each tuple in the stream contains two attributes, `val` and `loc`. If `P2` is itself a tuple type, such as `tuple<int32> x, int32 y >`, this leads to nested tuples. The other way to specify a tuple body is by a sequence of tuple types, where the combined type has all attributes of the individual types, which must be unique. For instance, `stream<P2, tuple<int32 z >> P3s` defines a stream `P3s` with all the attributes of `P2` and an additional attribute `z`. Note that this is not nesting, but type construction via concatenation. Finally, as a shorthand, SPL allows a stream name to refer to its tuple type.

SPL does not offer any pointer types, and as a consequence, no recursive or cyclic types. This design decision has several advantages: there are no null-pointer errors; all values are easy to serialize for transport on streams; SPL offers simple automatic memory management without requiring full-fledged garbage collection; and there is no aliasing, making it easier to curb side effects.

Variables, expressions, statements, and functions in SPL will look familiar to anyone used to C-inspired languages. However, variables and parameters in SPL are immutable by default unless declared with an explicit mutable modifier. An immutable variable or parameter is deep-constant. Functions in SPL are stateless by default unless declared with an explicit stateful modifier. A stateless function cannot read or write non-local data except for its mutable parameters, if any. A simple interprocedural analysis in the SPL compiler checks mutability and statefulness. All function parameters are passed by reference. Note that this only affects semantics for mutable parameters. The compiler checks that actuals passed to mutable formals are never aliased.
Fig. 6: Composite operator definition.

Taken together, the omission of pointer types, the explicit mutability and statefulness declarations, and the prohibition of aliased mutable parameters make it easy to statically pin-point expression side effects. This is useful both for error prevention and for optimization. For example, the SPL compiler statically checks that state written by a statement is not read anywhere elsewhere in the same statement. This prevents statements such as \( \text{return } (x++)/f(x) \); that depend on expression evaluation order.

For programming in the large, SPL provides namespaces and toolkits. An SPL namespace acts similarly to a C++ namespace or a Java package. A toolkit is a separate root directory in the library lookup path, similarly to a classpath component in Java.

2.5. Composite Operators

SPL users think in terms of stream graphs, and doing so is a simple mental model as long as applications do not get too large. Composite operators make stream graphs manageable at scale. A composite operator encapsulates a stream subgraph. The SPL compiler macro-expands composite operator invocations until only a flat graph remains. The vertices of that flat graph are primitive operator instances. Primitive operators are the subject of Section 4. Composite operators make it possible to reuse subgraphs, and offer graph-level modularity. The syntax for invoking composite and primitive operators is the same, except that composite operator invocations never carry logic, window, or output clauses.

Fig. 6 shows an example definition of a composite operator. Composite GenericEnricher declares output port Enriched, input ports In and External, formal parameters $\$Enricher$ and $\$FullData$, and a graph clause that uses the ports and parameters. When the SPL compiler encounters an invocation of GenericEnricher, it checks that the number of ports and the parameter names and kinds match. Then, it replaces the invocation by a copy of the subgraph, while substituting the appropriate actual streams and parameters. This expansion is *hygienic* in the sense of avoiding accidental name capture.

Unlike other streaming languages, SPL supports higher-order composites; e.g., composite GenericEnricher in Fig. 6 takes another operator, $\$Enricher$, as a parameter. Composites can also accept types (such as $\$FullData$ in Fig. 6), values, expressions, or functions as parameters. This broad set of parameters works in concert with automatic attribute forwarding to enable writing highly generic operators. Composite operators can even be entirely structural; a composite operator that only invokes operator parameters defines the structure of a stream graph, but makes no assumptions about the operators themselves. This amount of genericity increases opportunities for subgraph reuse and modularity.
type LogData = tuple<rstring service, uint32 severity, rstring data>;

stream<LogData> Logs = Import() {
  param subscription: service == "mail" ||
    kind == "system";
  filter: severity > 2;
}

() as Sink = LogProcessor(Logs) {}  

stream<LogData> Logins = LoginsProducer() {}  
{}

() as LoginsExp = Export(Logins) {
  param properties = {service = "login",
    kind = "system"};
}

stream<LogData> Mail = MailProducer() {}  
{}

() as MailExp = Export(Mail) {
  param properties = {service = "mail",
    kind = "app"};
}

Fig. 7: Importer application.

Fig. 8: Exporter application.

2.6. Dynamic Application Composition

The shape of an application graph is static: the edges and vertices do not change at runtime. However, users can obtain more dynamic graphs by taking advantage of the fact that Streams is multi-tenant: a Streams instance hosts multiple applications at the same time. SPL provides a feature for cross-application stream edges, called dynamic connections. An application that exports a stream tags it with publication attributes (name-value pairs). An application that imports a stream specifies it with a subscription predicate over publication attributes. The runtime dynamically adds or removes the corresponding edges when applications start or stop.

Figs. 7 and 8 list the SPL code for an example scenario illustrating the use of dynamic application composition features of SPL. There is an importing application (Fig. 7) interested in log streams with specific features and an exporting application (Fig. 8) that produces log streams of potential interest for the importing application. In particular, the importing application is subscribed to streams that are exported with a service property of value "mail" or a kind property of value "system". Furthermore, it specifies that the contents of the subscribed streams are to be filtered, remotely, using the predicate severity > 2. The exporting application is publishing two streams, one with properties service and kind of values "login" and "system", respectively; and another with the same properties, but values "mail" and "app". Both of the exported streams match the subscription of the importer application from Fig. 7. In practice, there could be additional importer and exporter applications, which could come and go dynamically. The SPL runtime is responsible for establishing and severing the connections as needed.

2.7. Putting it All Together

As we have seen, SPL provides syntax for defining graphs of streams and operators, while also offering conventional language features such as types, expressions, and functions. The SPL compiler creates an application containing the stream graph obtained by expanding a main composite operator.

In language design, it is not just important to add certain features, but also to omit others. Besides shared state, pointers, and parameter aliasing, another omitted feature worth mentioning is object-orientedness. SPL strictly separates state from behavior. State and values are passive, as befitting data items on a stream. Behavior resides in operators and functions. This keeps the language simpler.

For a full sample SPL application, see Appendix A.

3. SYSTEMS OVERVIEW

While programmers writing SPL mostly reason about operators, the primary unit from a systems perspective is the processing element (PE). A PE corresponds to an operating system process, and PEs contain one or more operators. PEs have input and output ports that are distinct from operator input and output ports. Each PE input port receives tuples from other PEs and sends them to operator input ports inside itself, and
each PE output port receives tuples from operator output ports inside itself and sends
them to other PEs.

An SPL application in execution comprises one or more PEs, where each PE com-
prises one or more operators. A Streams instance contains all of the runtime services
responsible for launching, running, and coordinating SPL applications.

Making a distinction between the fundamental computational unit in the program-
ing model (operators) and the fundamental system vehicle for execution (PEs) pro-
vides flexibility in how applications can be executed and a high-level abstraction of a
parallel, distributed system.

3.1. Application Life Cycle

The compiler produces two sets of artifacts for the runtime system: the compiled bina-
ries for the PEs, and the Application Description Language file (ADL). The life cycle of
an application starts when a user submits its ADL to the Streams instance.

The ADL contains a logical view of the application, which includes information on
all operators, PEs, types, and post-compilation transformations. The post-compilation
transformations are applied at submission time, and produce the Physical ADL
(PADL). Which operators are in which PEs, and how many input and output ports
each operator has, is fixed at compile time. However, PE input and output ports, and
the connections between operators inside of a PE, are entirely driven by the PADL.

This distinction between the logical view of the application (ADL) and the physical
view of the application (PADL) allows submission time flexibility. The runtime system
is free to transform applications based on submission-time information. We discuss
one of these transformations, fission, in Section 3.2.4.

After submission and initial setup, the SPL application is in execution. Unlike con-
ventional applications, streaming applications are intended to remain running indefi-
nitely. Even if an SPL application is not currently processing data, it is always waiting
and ready for more data to arrive. Hence, users must issue a request to the Streams
instance if they want an SPL application to stop executing.

3.2. User-Controlled Placement

The power of language abstractions such as operators and streams is that they enable
a separation of application logic from system configuration. SPL provides the following
system configuration controls, which are orthogonal to the logic of an application:

Operator placement. Users can direct fusion—how operators are combined into
PEs—with the partitionColocation, partitionExlocation, and partitionIsolation con-
figurations.

Thread placement. Users can introduce new threads with the threadedPort configu-
ration.

Host placement. Users can influence the mapping from PEs to hosts with the host,
hostColocation, hostExlocation, and hostIsolation configurations.

Fission. Users can request fission—replicating subgraphs to exploit data
parallelism—with the @parallel annotation.

3.2.1. Operator Placement. SPL abstracts operator communication as consuming data
items from input streams and emitting data items on output streams. The runtime
implements these abstractions as either function calls or sending data over the network.

Operators in the same PE communicate via function calls: the sending operator’s
data-item submission calls a function associated with the input port of the receiving
operator. In this case, no serialization occurs. In fact, depending on tuple mutation and
graph topology, operators may communicate by simply passing a reference. If not, then
the runtime creates a copy and calls the submission function with that copy.

Operators in different PEs communicate over a network protocol (even if they are
on the same host, and do not actually use the network). The runtime system abstracts
all of the issues related to network communication: resolving addresses, establishing
and maintaining connections, polling, partial transmission, and serializing and dese-
rializing tuples. In fact, because SPL is statically and strongly typed, the compiler can
generate serialization and deserialization routines specialized for each tuple type.

Fusion—grouping operators into PEs—determines the communication profile of an
SPL application. Fusion is, however, orthogonal to the logic of an SPL application.
SPL allows programmers to configure which operators are fused together to form PEs
independently of what computations those operators perform.

The SPL compiler has a profile-directed auto-fusion option, which views fusion
optimization as a graph partitioning problem that minimizes data flow between
PEs [Khandekar et al. 2009].

3.2.2. Thread Placement. Threads in a PE arise from source operators, threaded ports,
and PE input ports, as illustrated by Fig. 9.

Source operators, by definition, have no incoming streams. They introduce new data
items into the application either by creating them from inside the source operator
itself, or by converting them from an outside source. Because source operators are not
driven by any other part of the application, they require their own thread to drive
execution of themselves and downstream operators.

The user can place threaded ports via the threadedPort config. Designating an oper-
ator’s input port as threaded means that a new thread executes the operator starting
from that port. Threads executing upstream operators deposit data items in a queue,
which this threaded port pulls from.

PE input ports receive data items over the network and pass them to operators in-
side of itself. They require a thread to monitor the socket associated with the PE input
port, handle the protocol, and deserialize raw data into structured tuples or punctua-
tions. This same thread delivers data items to operators in the PE and executes those
and other downstream operators.

Threads that originate from source operators and PE input ports execute all down-
stream operators until they encounter either a PE output port or a threaded port.
When threads encounter a PE output port, they send the data item outside of the PE
using the network. When threads encounter a threaded port, they place the data item
in a queue. If the network is busy, or if the queue is full, the thread may block when try-
ing to submit a data item. This blocking naturally leads to back-pressure: if a thread
blocks when trying to submit a data item, it is unable to accept new data items, and
threads trying to submit to it also block. This blocking propagates back to the source.
Back-pressure coordinates execution rates without a centralized scheduler.
Threads execute downstream operators in a depth-first order, but the order in which they execute operators at the same level of the sub-graph is implementation defined.

Fig. 9 shows a PE with all three kinds of threads. Operator A is a source operator, and the thread that executes it, thread 1, also executes operator B. Note that operators A and B are executed sequentially. At operator C, thread 1 encounters a threaded port, so it places the submitted data item in a queue and goes back to executing A and B. Thread 2 retrieves data items from the queue and executes operators C and G. Thread 3 exists because it drives the PE input port; it receives data items from the network and then executes downstream operators D, E, F, and G. After executing E, thread 3 is free to execute either G or F first. Finally, since both threads 2 and 3 execute operator G, its input port requires a lock.

Work in a research prototype investigated automatically determining, at runtime, where to place threaded ports [Tang and Gedik 2013].

3.2.3. Host Placement. At submission time, the runtime system evenly distributes PEs to the hosts in the Streams instance. However, programmers can also control host placement. Operator invocations can carry configs that specify relative constraints, set-based constraints, and absolute constraints. Because PEs are not a language level entity, host placement is specified on operators. Consequently, the host placement constraints for a PE are the union of all of the host placement constraints of the operators contained in that PE. If any operators in a PE have conflicting constraints, the compiler issues an error.

Relative hosts constraints place PEs on the same host or on separate hosts (hostExlocation or hostColocation). A hostpool is a language-level entity that allows programmers to request a set of hosts, with a name at the language level. Operators can then be assigned to that set of hosts through the hostpool name. Hostpools can also contain multiple tags, which are arbitrary strings used by external tooling for naming sets of hosts. Finally, absolute host constraints are host configs that indicate a specific host.

3.2.4. User-Directed Fission. SPL programs naturally expose task and pipeline parallelism. Task parallelism occurs when a stream is consumed by multiple, different operator instances, which then simultaneously process the same data items. Pipeline parallelism occurs when an operator sends data items to an operator in a different thread or PE; pipelined operators can simultaneously process and prepare data items for each other.

Data parallelism in a streaming context means splitting a stream of data items to multiple replicas of the same operator. In the streaming optimizations literature, this process is called fission. Unlike task and pipeline parallelism, data parallelism does not naturally exist in a streaming application. Programmers can introduce data parallelism manually by hard-coding invocations of an operator multiple times and creating and connecting the necessary streams. To alleviate this burden, SPL offers user-directed fission, where programmers request replication of an operator invocation with the @parallel annotation. This is similar to the use of OpenMP [OpenMP 2014] pragmas for parallelization in C, C++, or Fortran. The SPL compiler and runtime do the work of replicating the operator and creating and connecting the streams. The operator can be primitive or composite; if it is composite, the entire subgraph is replicated.

User-directed fission takes advantage of the flexibility introduced by the ADL and PADL (see Section 3.1). Operator replication and stream creation all occur at submission time, which is possible thanks to the separation of application description from execution.
Work in a research prototype explored applying fission automatically, as a compiler optimization [Schneider et al. 2012; Gedik et al. 2014].

3.3. Parallelism in SPL

The ease of exploiting task, pipeline, and data parallelism in a single application is due to the programming model, where operators have independent state and only communicate via streams. The expression of this parallelism, however, is orthogonal to its execution. The system mechanisms that realize this parallelism are threads and PEs. All PEs in an application execute simultaneously as operating system processes. Hence, operators inside of different PEs execute simultaneously. As PEs communicate over the network, they can run on different hosts. Consequently, SPL is a natural means to write a parallel, distributed application.

In general, operators fused into the same PE no longer run in parallel: instead, they execute on the same thread and communicate through function calls. However, we can gain back the lost parallelism by inserting threaded ports between operators. SPL programs, then, are also a natural means to write multi-threaded applications that take advantage of multi-core processors.

The combination of these two system mechanisms (threads and PEs) means that SPL applications can run on a wide range of parallel systems, from large clusters where each host has a modest core count, to single systems with many cores, or any combination of the two extremes. SPL's user controls—operator fusion, threaded ports, host placement, and all kinds of parallelism—gives programmers the ability to adapt to the volume and velocity requirements of their application.

These controls present a tension between throughput and latency, and between lower communication costs and scheduler freedom. Such trade-offs are typical of high-performance parallel/distributed systems. SPL's novelty is that these concerns are orthogonal to an application's logic and architecture.

3.4. Putting it All Together

Fig. 10 shows an example of operator, thread, and host placement, as well as user-directed fission. It uses an invocation of the TCPSource operator to receive data items from an external source that contains a user name and the number of uses from that user. The Filter operator invocation filters out users who have not used the service heavily, and the DBLookup operator invocation returns a tuple with a rich profile for that user. Because the TCPSource may emit many tuples, and the Filter reduces that
number, we fuse them into the same PE with partitionColocation. The operators now communicate through function calls, reducing communication cost and total network traffic. The thread on the Filter’s input port allows it and the TCPSource to exploit pipeline parallelism. The PE containing the TCPSource has a host constraint: it must be on a machine allowed to access the outside network. A hard-coded host assignment places it correctly. The DBLookup operator is expensive, and should not be on the same host as the other operators. A hostExlocation constraint ensures this placement. Finally, the @parallel directive requests two data-parallel copies of this operator.

For a full example of system configuration in an application, including the relationship between the logical and physical view of an application, see Appendix A.2.

4. OPERATORS AND CODE GENERATION

SPL operator development centers around code generation. A new primitive operator is added by writing a code generation template and an operator model (Section 4.1).

The code generation template contains the operator implementation (Section 4.2). The implementation follows an event-driven design, wherein the operator logic is specified by extending a base operator class and overriding relevant data item processing functions. The primary language used for the implementation is C++. However, the code generation templates are not pure C++ code, but instead a mixture of C++ code (the template) and Perl code (the generator). The Perl code is used to generate C++ code. It has access to an operator instance model describing the details of the operator instance for which code is being generated. This way, the operator code can be customized for the operator instance at hand. The SPL compiler optimizes code-generation to accelerate the edit-debug cycle (Section 4.3).

An operator model is a configuration file in XML that describes the constraints on the operator interface and the semantic properties of the operator (Section 4.4). The interface constraints enable the SPL compiler to perform better error checking and diagnostics. The semantic properties enable the compiler and the runtime to better establish safety properties and locate optimization opportunities.

4.1. Operators: Development, Compilation, Execution

Fig. 11 shows the process of compiling and running an SPL application. The figure depicts two kinds of users. One is the application developer who creates streaming applications in SPL, by creating, configuring, and connecting operators. The other is the operator developer, akin to a library developer in general-purpose programming languages, who develops generic, reusable operators.

The SPL compiler takes as input the SPL code as well as the operator models. After expanding all composite operators in the SPL code, the compiler creates a list of all primitive operator instances. It uses the operator models to check the correctness of each operator instance. In case of errors, the operator invocation in the SPL program that has resulted in the problematic operator instance is reported as the cause of the problem. Otherwise, the compiler then generates, for each operator instance, an operator instance model to be used during code generation. These operator instance models are fed to the operator code generators corresponding to their operator kinds. Note that for each operator kind, there is a single code generator, but there could be many operator instance models. An operator’s code generator is itself generated from the code generation template associated with the operator. This step is performed by the SPL toolchain before the operator is registered with the SPL compiler for use in stream programs. Finally, the operator code generator, given the operator instance model, produces the C++ operator code specialized for the operator instance at hand. The generated operator instance code is compiled into a shared library that is loaded
by the SPL runtime for execution. During runtime, the operator instance is initialized using configuration data found in the ADL file, also generated by the SPL compiler.

4.2. Code Generation Templates
SPL's code generation templates bring advantages in the areas of performance, error reporting, and interface flexibility.

Performance. Specialization of generated code based on the operator instance at hand results in better run-time performance. This is because for many tasks code generation avoids run-time introspection of the operator instance model. For example, an operator that parses XML data can use code generation to specialize the code for the particular XPath expressions to extract [Mendell et al. 2012].

Error reporting. The scripting capabilities of code generation templates enable programmatic checking of operator invocations for validity of complex conditions. Since debugging distributed applications is known to be difficult, locating errors at compile-time is beneficial. For example, an operator converting data from an external format can use compile-time programmatic checking to ensure that the output tuple type is compatible with the external format from a schema file specified by an operator parameter.

Interface flexibility. SPL's code-generation capabilities enable the use of mini expression languages for configuring operator parameters and output assignments. This results in highly declarative and reusable operators, and is achieved by allowing operator parameters and output assignments to be SPL expressions that call operator-specific intrinsic functions. While the interfaces of the intrinsics are specified in the operator model, their semantics are implemented as part of the code generation templates. The operator instance model available to the code generation template provides access to a full SPL expression tree for parameters and output assignments. The operator developer can customize the C++ code to be generated by deciding what code to emit for calls to intrinsics. Many generic SPL operators were developed this way, such as rela-
An example. We use the Filter operator to illustrate the performance and error reporting features facilitated by code generation templates. The Filter operator performs selection, that is, it passes only the data items that satisfy a given boolean predicate specified using the filter parameter (see Fig. 10 for an example).

Fig. 12 shows the C++ header file template for the Filter operator. The template contains mixed-mode code: the generator code (Perl) is placed within `<% ... %>` blocks and the generated code (C++) is placed as is. Within the generator code, variable $model holds the operator instance model. It is used to generate code and perform error checking for the operator instance at hand. The first piece of logic in the header template is the verify function, together with the call to it (using $model as the parameter). The verify function is responsible for checking the operator instance configuration for correctness. In particular, it ensures that the input and output ports of the Filter operator have the same type. If not, it prints an error message using code location information coming from the operator instance model. The C++ code in the header template consists of the class definition for the Filter operator. The class contains a single process member function, which is overridden to implement the tuple processing logic for the Filter operator.

Fig. 13 shows the C++ implementation file template for the Filter operator. The definition of the process member function forms the main body of the implementation template. This function simply checks the filter condition over the current data item, and if satisfied, submits the data item to the output port. Rather than interpreting the filter condition, its C++ representation is embedded into the generated code for performance reasons. This is achieved via the call to the getFilterExpr("tuple", $model)
function defined at the top of the file within the generator code block. This function retrieves, from the operator instance model, the C++ expression corresponding to the SPL filter expression specified by the filter parameter and substitutes tuple as the variable name corresponding to the current data item.

4.3. Code Generation Optimizations

To provide a short edit-debug cycle, SPL reduces compilation times via code sharing and incremental compilation.

*Code sharing.* To prevent code bloat, the compiler reduces the number of unique operator instances. This is facilitated by expression rewrite, which is performed by first simplifying the expressions that appear in the operator invocations through constant folding, and then replacing the remaining constant values with placeholders called run-time literals. This way, expressions that have similar structure modulo some constant are brought into a common form. The compiler creates a blueprint operator for each set of operator instances that have the same configuration after expression rewrite. Code sharing is achieved by generating code only for the blueprint operators. The original values of the runtime literals are stored in the ADL file. Operator instances initialize their runtime literal values from the ADL file during load time. An example is a set of TCP source operator instances that differ only in their IP-address parameter.

*Incremental compilation.* Re-compiling the entire application each time there is a change in the source code is costly. This is exacerbated by the long compilation and link times for C++, which is the target language of the operator code generators in SPL. To alleviate this problem, SPL employs incremental compilation. Incremental compilation is facilitated by storing the operator instance model together with the generated code. When re-compiling an SPL application, the current operator instance models are compared against the stored ones. If there is no difference between the two for some of the operator instances, then the code generation is skipped for them. This results in skipping the build of C++ code as well, saving significant time.

4.4. Operator Models

Operator models play two roles in providing extensibility: interface and semantics.

4.4.1. Interface. A fundamental need in developing streaming operators is to define constraints on their interface. While all SPL operator invocations follow the common syntax from Section 2.3, various aspects can be specialized, such as: the arity of the input and output ports, including optional and variable number of ports; the parameter names, their optional/mandatory status, types, expression modes, and expression rewrite permissions; windowing configurations of the input ports; expression modes and expression rewrite permissions of the output ports; and operator-specific intrinsic functions and their signatures. Several of these are worthy of further elaboration:

*Parameter types.* In SPL, an operator parameter can take values with differing types. For instance, the `groupBy` parameter of a streaming relational aggregation operator can take a value of any type. As another example, the key parameter of a streaming sort operator can take a value of any ordered type, that is a type for which a total order is defined over its values. The operator model allows the specification of the parameter type to enable the compiler to type check the parameters and saves the operator developer from performing this check in the code generation template.

*Expression modes.* The expression mode of the parameter specifies the kind of expressions that can be used as the parameter value. Some parameters allow expres-
sions with no limitations, such as the filter parameter of a selection operator. The filter can be any valid SPL expression, as long as it meets the type requirements of the parameter, which in this case requires a boolean expression. Some parameters only allow expressions that do not reference any input tuples. For instance, the modelFile parameter of a data mining scorer operator, which is used to specify the path to the model file to load, cannot take a value that is an expression referencing input tuples. This is because the model file is used during load time and is not dependent on the input tuples. For some parameters, the references to input tuples in the parameter value have to be restricted to a specific input port or ports. In yet another use case, the parameter values must be constant values or expressions that can be folded into constants at compile-time. This is particularly useful for operators whose code generation templates inspect the parameter values at code generation time and generate different code for different values encountered. For instance, the buffer size parameter of a threaded split operator can be used to statically allocate a buffer, if its value is known at code generation time. The expression modes also apply to output attribute assignments, and are similarly specified in the operator model.

**Expression rewrite permissions.** These specify whether the compiler is allowed to rewrite the value of a parameter or output attribute assignment. Expression rewrite is permitted by default to enable the code-sharing optimization. However, if the operator’s code generation template specializes the code depending on an expression’s value, then rewrite should be disallowed. An example is a pattern matching operator that uses the regular expression specified as a string literal to generate a specialized state machine to perform high-performance event matching [Hirzel 2012].

**Operator-specific intrinsic functions.** Intrinsics can be used to give new meaning to SPL expressions. The signatures of intrinsics are given in the operator model. Since many stream processing operators work with generic types, the intrinsic function signatures support using generics. For instance, the Aggregate operator seen in Fig. 3 supports an ArgMax intrinsic with the signature:

```<ordered TM, any TA> TA ArgMax(TM m, TA a)```

This signature enables the SPL compiler to type-check ArgMax calls in Aggregate operator invocations. It states that ArgMax takes two parameters. The first one has an ordered type and the second one can have any type. Also, the return type is the same as the type of the second parameter. The semantics of the ArgMax intrinsic is up to the operator and its implementation is provided in the code generation template. For instance, for the Aggregate operator, the ArgMax function looks at all the tuples in the operator’s window, computes the value of the first parameter for each tuple, finds the tuple that gives the highest value for it, and returns the value of the second parameter for that tuple. The code generator can incrementalize this computation when the window contents change.

4.4.2. Semantics. It is difficult to statically analyze a code generator to ascertain properties of the C++ code it generates. Therefore, instead of using static analysis, the SPL compiler relies on operator developers to specify such properties in operator models.

**Threading.** Understanding the threading semantics of operators is required to optimize lock acquisition when multiple operators are fused into a single PE. For this purpose, the operator model specifies whether an operator provides a single-threaded context (ST-context) or not. An operator provides an ST-context if (i) it does not perform concurrent data item submissions unless its input processing methods are called concurrently, and (ii) its data item submissions complete before the input processing method call that triggered the submission returns. While many operators, such as a simple filter, provide an ST-context, several do not. An example that violates the first
requirement is a TCP server operator that uses a thread pool to serve connections and submits data items from each thread in the pool. An example that violates the second requirement is a buffering operator that enqueues data items into a buffer and performs submissions using a separate thread of its own. When operators do not provide a ST-context, the SPL runtime may acquire locks on downstream operator instances.

Port mutability. When operators are fused into the same PE, one of the SPL compiler optimizations is to elide copies of tuples flowing from one operator to another. The operator developer can enable this optimization via port mutability properties in the operator model. Input port mutability specifies whether the operator modifies the input tuples as part of its processing. Output port mutability specifies whether the operator relies on the contents of the tuple being unchanged as a result of submission. With these settings specified, the SPL compiler can elide tuple copies when applicable.

Punctuations. Window-punctuation properties are specified on a per-port basis. An input port can be window-punctuation expecting or oblivious. An output port can be window-punctuation generating, preserving, or free. The compiler uses this information to ensure that punctuation-expecting input ports are connected to punctuated streams. As an optimization, it can also drop window punctuations if no downstream input ports are expecting them. For final punctuations, the operator model declares which input ports are to participate in automatic forwarding. For instance, an input port not used for the main data flow may not participate in the forwarding. The runtime uses the forwarding specs to correctly handle application termination.

State. State plays a crucial role in establishing the safety of optimizations that morph the graph, such as fission. The operator model categorizes an operator as either stateless, partitioned stateful, or stateful. The SPL compiler applies auto-fission [Schneider et al. 2012] only in the first two cases. For partitioned stateful operators, auto-fission can be applied given an operator parameter that specifies a partitioning key. The partitioning key promises that the operator maintains independent state for each sub-stream corresponding to a specific value of the partitioning key. The partitioning key is used in the splitter to ensure that the flow is distributed among the parallel channels such that the sub-streams containing tuples with a specific partitioning key value are always sent to the same channel.

Selectivity. Selectivity is the relationship between the number of output tuples generated by an operator per input tuple consumed. In SPL, this is not a static number. The operator model categorizes an operator's selectivity as either one-to-one, one-to-at-most-one, or one-to-many. As an example, for the auto-fission optimization, this information is required for a timely merge—bringing tuples processed by different parallel channels back into their original order. If operators are stateless and one-to-one, then a simple round-robin merge suffices. Otherwise, more involved schemes are needed, such as sequence numbers and pulses [Schneider et al. 2012].

An example. Fig. [14] shows an excerpt from the operator model of the Filter operator. The operator model has four main parts: context, parameters, input ports, and output ports. The context section specifies the general properties of the operator. We see that the filter operator provides a single-threaded context and is stateless. Its selectivity depends on the parameter setting of the operator instance at hand. In particular, if the filter parameter is present, then the selectivity is one-to-at-most-one, otherwise it is one-to-one. The Filter operator supports a single parameter named filter, which is optional. Expression rewrite is allowed on the parameter value, since the operator's code generation template does not conditionally specialize the generated code based on it. The type of the filter parameter must be boolean. The Filter operator
has a single input port that is non-windowed. Its tuple mutation settings are set to false for both input and output ports, since it does not mutate the input data items and directly submits them to the output port. The input window punctuation mode is set to oblivious, as the operator does not rely on window punctuations to operate. The Filter operator has a single output port, for which the auto-assignment mode is set to true, since it forwards unassigned output attributes from the input port. The output window punctuation mode is set to preserving, as the operator just passes the punctuations through.

5. CASE STUDIES
This section offers operator case studies, which illustrate SPL’s features for extensions and code generation, and application case studies, which illustrate usability, performance, and breadth of adoption of the language as a whole.

5.1. Operator Case Studies
This section showcases SPL’s extensibility via operators.

5.1.1. Filter Operator. SPL’s Filter operator uses code generation. The user-supplied filter parameter is an SPL expression on an input tuple. This SPL expression is translated to the equivalent C++ expression, and generated directly in the operator code.

The motivation for using code generation, rather than interpretation, is performance. To quantify the performance difference, Fig. 15 compares the performance of SPL’s Filter operator (codegen) to two alternatives that perform runtime evaluation.
composite FindMShape<input stream<float64> data> Src; output Res) {
  graph
  stream<List<float64> mData> Res = MatchRegex(Src) {
    param pattern: ". up+ down+ up+ down* under";
    predicates: { up = data > First(data) && data > Last(data),
                 down = data >= First(data) && data < Last(data),
                 under = data < First(data) && data < Last(data) }
    output Res: mData = Collect(data);
  }
}

Fig. 16: Flexible pattern matching with MatchRegex.

of a string representation of the expression. The first implementation (eval runtime) reuses part of the existing SPL runtime which allows limited predicate evaluation, and the other (eval compiler) calls into the SPL compiler’s full expression evaluation facilities.

The experiment varies the number of terms in the expression, where each term compares a floating-point tuple attribute to a constant. The terms are combined such that they all must be evaluated at runtime (no short-circuiting). For a single term, the generated code is $3.8 \times$ faster than runtime’s evaluation, and $120 \times$ faster than compiler’s evaluation. For 10 terms, the generated code is $10 \times$ faster than runtime’s evaluation, and over $700 \times$ faster than compiler’s evaluation. This extreme difference in performance is because the generated code can become one instruction per term, while the evaluation at runtime must navigate expression trees. In principle, a JIT could achieve results similar to the generated code, but a JIT for a general purpose streaming language is a research problem unto itself.

5.1.2. MatchRegex Operator. The MatchRegex operator highlights more sophisticated code generation, translating from a pattern via an automaton to optimized C++. It is designed to detect event patterns within a stream. Fig. [16] gives an example use, where the operator is configured to find M-shaped patterns in a data stream containing series of float values.

The MatchRegex invocation in Fig. [16] configures two parameters. The first is the pattern parameter, which specifies a regular expression that represents the event pattern to be detected. The , + , and * characters within the pattern are meta-characters of the regular expression, whereas the tokens up, down, and under reference events. The user defines these events as part of the predicates parameter, whose value is an SPL tuple literal. The attribute names in the tuple literal represent event names and their values represent the detection conditions of the events. These conditions are evaluated on each tuple to detect events, whereas the entire pattern is evaluated over a sequence of events, forming a composite event. The First and Last functions appearing in the event conditions are intrinsics used to access attributes of the first and the last tuples in the current sequence of events being matched. The output clause calls the Collect intrinsic to obtain the list of all tuples in the matched sequence and assigns it to the mData output attribute.

The MatchRegex operator’s implementation uses a number of techniques. The pattern parameter is specified in the operator model as a constant expression of type string, for which expression-rewrite is disallowed. This is because the pattern parameter is used during code generation. First, the pattern is used to generate a finite state machine to recognize matching sequences of tuples. Second, the pattern string is used to check that the event names it references are among the set of attributes defined by the value of the predicates parameter. The generation of a specialized finite state machine provides good performance, whereas the detailed error checking at compile-time improves usability [Hirzel2012].
type Transaction = tuple<rstring id, int32 cost>;
composite FindCustomerTxns(output Out; input stream<rstring data> In) {
  graph
    stream<rstring name, list<Transaction> txns> Out = XMLParse(In) {
      param trigger: "/customer";
      output Out: name = XPath("@name"),
        txns = XPathList("transaction",
          {id=XPath("@id"), cost=(int32)XPath("@cost")});
    }
}

Fig. 17: XMLParse operator showcasing the use of SPL expressions with intrinsic functions for
XML parsing.

<customer name="John">
  <order id="44386" cost="200"/>
  <order id="44756" cost="100"/>
</transaction>
<transaction id="9" cost="600">
  <order id="44812" cost="250"/>
  <order id="44901" cost="350"/>
</transaction>
<customer>

{ name = "John",
  txns = [ { id = "7", cost = 300 },
           { id = "9", cost = 600 } ]
}

Fig. 18: Sample XML segment and its corresponding SPL tuple literal for the XMLParse
transformation specified in Fig. 17.

The predicates parameter value is used in the code generation template to extract
the event detection conditions from the attribute values of the tuple literal. These spec-
ifications are embedded in the generated code to detect events. The intrinsic functions
in the event specifications are rewritten to reference the relevant tuple attribute and/or
its aggregate value corresponding to the intrinsic function’s semantics in the current
partial match. The use of SPL expressions for specification of individual events pro-
vides a great deal of flexibility, syntactic uniformity, and detailed type checking for
free (not performed by the code generation template).

5.1.3. XMLParse operator. The XMLParse operator is used for converting a stream of XML
fragments into a stream of SPL tuples. It is a transformation operator for declaritively
specifying the mapping between XML fragments and SPL tuples. Fig. 17 gives an ex-
ample use of the XMLParse operator. It extracts a subset of the customer information
found in XML fragments received line-by-line on the input stream, and transforms
that into an SPL tuple. Fig. 18 shows a sample XML fragment and the resulting SPL
tuple after performing the transformation given in Fig. 17. The transformation ex-
tracts only the customer name and a list of their transactions.

The XMLParse invocation in the code listing specifies a single parameter, trigger. This
parameter specifies an XPath expression to pinpoint the XML fragments from which to
create tuples—the customer XML elements in this example. The output assignments of
the XMLParse operator use SPL expressions with intrinsic functions to specify the mapping
between the XML data and SPL tuples. The name SPL attribute is assigned via use
of the XPath intrinsic, which takes in as parameter the "@name" XPath expression and
extracts the name attribute from the customer XML element. The txns SPL attribute
is assigned using the XPathList intrinsic, which takes two parameters. The first is an
XPath expression, in this case “transaction”, which is used to locate the XML elements
that will be used to construct the contents of the list. The second parameter is an SPL
expression that specifies the mapping to be used for each list element. The XPathList
intrinsic function has the signature: list<T> XPathList(rstring, T). In this example,
each list element is a tuple with the attributes id and cost, which are both assigned via
the XPath intrinsic. The former is assigned the value of the id XML attribute within the

transaction XML element. The latter is assigned the value of the cost XML attribute, after a cast to an integer.

The XMLParse operator's output assignments are type checked by the compiler to ensure that the type of the extracted data matches the output tuple type. In order to implement the mapping from XML to SPL tuples, the code generation template of the XMLParse operator processes the expression trees of the assignments and replaces the intrinsic functions with the corresponding XPath queries. Importantly, it further converts these XPath queries into an automaton at compile-time. At runtime, the operator instance executes this automaton by reacting to SAX parser events.

Fig. 19 shows that the automaton is orders of magnitude faster than a baseline that executes each XPath individually. The results are presented for the XMark benchmark and for a location-based application we built using the SIRI standard.

The end result [Mendell et al. 2012] is a declarative and high-performance way of mapping XML fragments to SPL tuples—a common data ingest operation in streaming applications.

5.1.4. Other Operators. SPL toolkits built so far include many other interesting uses of the extensibility capabilities we have outlined so far. For instance, the Aggregate operator from the stream relational toolkit relies on intrinsics and code generation not only for providing a list of built-on aggregations, but also for providing an operator-specific extension mechanism through which user-defined aggregations that are specified as functions in the SPL language can be created. The data-mining toolkit provides support for scoring of mining models, wherein model files used as external dependencies form an important role in compile-time error checking. The database-connector toolkit uses similar techniques for schema mapping between tables and streams.

5.2. Application Case Studies

This section gives whole-program SPL examples.

5.2.1. Log-Monitoring Benchmark. This section picks a log-monitoring benchmark for a detailed case study, as it illustrates both usability and performance of SPL. The benchmark consists of 39 applications, each implementing different tasks related to computing real-time operational statistics from streaming log data. These tasks cover parsing, formatting, filtering, enrichment, projection, aggregation, state management, splitting, correlation, and pattern detection. The input data contains operational logs produced from back-end mobile services software of a major telco company. The specification of the benchmark was also provided by the same company. This study showcases two important properties of SPL. First, SPL's higher-order composites simplify development of applications that share high-level topological properties. Second, SPL

\[\text{Fig. 19: XMLParse performance.}\]
applications can reach high throughput and saturate the network while performing non-trivial log processing tasks.

Usability. Each application in the log-monitoring benchmark processes 40 log data streams. These source streams are divided into two groups, and data from pairs of streams belonging to different groups need to be brought together for correlation. The correlated results are then further grouped into four pools and then merged on a per-pool basis. Finally, the per-pool results are combined to generate a global result stream. The SPL-based implementation exploits an important characteristic of these applications: each application’s operator graph follows one of a few high-level topologies, while low-level details are specialized. In particular, each high-level topology contains a common arrangement of how the sources are consumed, parsed, and distributed; how the results are collected and merged; and how sub-graphs are placed on hosts. However, the individual filters, aggregations, correlations, etc. differ for each application. The SPL implementation takes advantage of this structure by organizing the code using two sets of operators:

— First-order composite operators that encapsulate the core application-specific logic.  
— Higher-order composites that encapsulate common high-level topologies. These composites take as parameters other operators with the core application logic and embed them into proper places within the boiler-plate topology.

With this organization, the benchmark has a total of 7,029 lines of code (LOC) in SPL. Part of this is the 4,350 LOC in a toolkit that contains first-order composite operators that are commonly used in different benchmark applications, and more importantly, higher-order composites encapsulating high-level topologies. The remaining 2,679 LOC are in the 39 applications. The application code contains the first-order composites that are embedded into the high-level topologies, as well as the top-level call to the higher-order composite that forms the topology. Note that an average benchmark application has 69 LOC. This is quite small compared to the 8,121 LOC after composite expansion.

In this case study, the LOC between the expanded and unexpanded versions of applications differ by almost 2 orders of magnitude. Further details about higher-order composites in the log-monitoring benchmark can be found in [Hirzel and Gedik 2012].

Performance. This section reports performance results from one of the benchmark applications that performs correlations to compute statistics about dropped events and message latencies. In particular, the application matches send and receive events from 20 pairs of log sources based on an event ID. Next, it locates events for which the
receive log is missing (a lost event), and computes the average latency of events. Further, latencies from different log sources are combined based on a message ID field. Final statistics about the per-message latencies are computed incrementally, and are reported every half-minute.

We run this application on 1–4 hosts, where each host has two 4-core 64-bit Intel Xeon processors running at 2.93 GHz. The 40 log sources are generated by a separate pool of 4 hosts with similar configuration. We experimented with both 1 and 10 Gbit/s network connection between the machines.

Fig. 20 shows the throughput (left y-axis) and number of cores utilized (right y-axis) for different number of hosts, available cores per host, and network configurations. The application reaches a throughput of 3.5 million tuples per second for 4 hosts with 1 Gbit Ethernet, when all 8 cores on each host are available. Interestingly, the number of utilized cores is below the number of available cores for both 2 and 4 host scenarios, implying that the entire network bandwidth is consumed. The results on the 10 Gbit network verify this, showing that the throughput reaches 10.5 million tuples per second, and all 32 cores used are fully utilized. Since the network connection from the client becomes the performance bottleneck, that means no streaming language can outperform SPL on this application on 1 Gbit Ethernet.

5.2.2. Other Applications. SPL has been widely adopted to build sophisticated real-world streaming applications. This section briefly surveys representative case studies (ordered by publication date).

Astronomy Imager takes as input signals from radio antennas, and combines them into an image of the sky [Biem et al. 2010b]. The sheer volume of input data is too high to store on disk. Velocity, on the other hand, is less of a concern, since there are no strict latency requirements.

Stock Trader takes as input stock trades and quotes, and computes trading decisions and prices [Park et al. 2012]. This application had the strictest latency requirements. The authors used InfiniBand network transport and low-level system tweaks to achieve latencies of 23 µs.

Traffic Monitor takes as inputs vehicle location updates and ad-hoc queries, and computes database outputs of the aggregate road network status and ad-hoc visualizations [Biem et al. 2010a]. It motivates why streaming languages must permit diverse application topologies.

Medical Predictor takes as input heart rate, blood pressure, respiration, and other physiological streams, and computes short-term forecasts for them [Sow et al. 2012]. It exemplifies the variety challenge of big-data streaming, since input data is noisy but the stakes to compute accurate outputs are high.

Call Data Mediator takes as input call-data records from telco switches, and performs tasks such as dashboarding, warehousing, billing, etc. [Bouillet et al. 2012]. Its stream graph has around 7,000 operator instances and involves feedback cycles.

Spam Watcher takes as input mobile phone text messages, and classifies them as spam or not spam [Zou et al. 2011]. The classifier itself is derived from a social network, which gets compiled into a state machine for pattern-matching hardware.

Gene Sequencer takes as input DNA fragments, and matches them against a reference genome in a streaming manner [Kienzler et al. 2012]. It is an example of large but finite batch analytics, where volume matters more than velocity.

These applications cover a diverse set of domains. Some were built early and drove
SPL's design. Others were built later and validated that SPL is general and extensible enough for unanticipated use cases. Overall, the applications exhibit different performance requirements, motivating parallelism and distribution, as well as the ability to trade off throughput, latency, and resource utilization.

6. RELATED WORK

This section discusses related work in three categories.

6.1. Stream Processing

This section outlines the main approaches for stream processing in the literature: synchronous dataflow, relational streaming, and complex event processing. SPL draws inspiration from, and expands upon, each of them.

Synchronous dataflow (SDF) is defined by Lee and Messerschmitt as dataflow where the number of data items produced or consumed by each operator per firing is specified a priori [Lee and Messerschmitt 1987]. This a-priori knowledge enables SDF compilers and runtimes to use static schedules. In contrast, while in SPL, each operator firing consumes one data item, the number of data items produced per firing is not known a priori, and is often data-dependent. Data-dependent rates are the common case for big-data applications targeted by SPL, which often include parsing, filtering, aggregation, deduplication, etc. Therefore, SPL's compiler and runtime uses a dynamic and decentralized schedule based on back-pressure.

Lustre [Halbwachs et al. 1991] and Esterel [Berry and Gonthier 1992] are famous SDF languages. In their definition of SDF, a program can be thought of as reacting instantaneously to external events, yielding both functional and temporal determinism. In contrast, in SPL, firings of multiple operators in the same stream graph are not considered a single instantaneous reaction. Allowing operators to fire on their own separate schedules facilitates distribution for scaling over clusters of machines. Lustre and Esterel focus on certified compilation for avionics and cars; in contrast, SPL focuses on big-data applications. The StreamIt SDF language makes the number of data items produced and consumed by each operator firing explicit in the syntax of operator signatures, and puts an emphasis on using that information for optimization and parallelization [Gordon et al. 2006]. Lime is a Java-based SDF language, where the user can lift an ordinary method to turn it into a vertex in a stream graph [Auerbach et al. 2010]. Lime exploits the SDF properties to generate FPGA bit-files. In general, SDF languages provide strong compile-time guarantees, but can only promise those in the absence of foreign-language code. In contrast, extensibility with code in a different language is a core requirement satisfied by SPL.

Relational streaming languages center around built-in operators from database relational algebra. The top-level syntax of these languages tends to be custom-tailored to relational-algebra operators, and is frequently based on SQL. In contrast, SPL delegates support for relational operators to the library. SPL offers the same expressiveness and syntactic power for user-defined operators that it does for library operators. Instead of granting a few operators built-in privileges, SPL ensures the operator development framework is expressive enough for all operators.

NiagaraCQ is an early stream-relational system, and its language is XML-QL, one of the precursors of XQuery [Chen et al. 2000]. NiagaraCQ takes an algebraic query compilation approach, with an emphasis on sharing common subqueries. GigaScope is a SQL-based stream-relational language [Cranor et al. 2003]. One of the challenges with relational operators in a streaming setting is that state might be proportional to the input streams; for instance, as in joins. GigaScope avoids this issue by enforcing constraints over ordering attributes. TelegraphCQ is also based on SQL, but the paper does not emphasize language concerns [Chandrasekaran et al. 2003]. Instead, Tele-
graphCQ emphasizes dynamic optimization and query sharing. Aurora is a stream-relational system where instead of using a language, the programmer creates a stream graph in a GUI [Abadi et al. 2003]. However, Aurora does have its own intermediate language called SQuAl (Stream Query Algebra). Borealis is based on Aurora, and adds distribution and revision messages [Abadi et al. 2005]. The Borealis paper does not discuss language issues.

CQL (the Continuous Query Language) extends SQL with streaming features and has rigorous semantics [Arasu et al. 2006]. Windows play a central role in CQL as the bridge from traditional relational operators to streaming. Spade is the predecessor of SPL with built-in operators for relational algebra [Gedik et al. 2008]. Doing away with built-in operators is one of the major differences between SPL and Spade. CEDR (Complex Event Detection and Response) performs streaming with relational algebra [Barga et al. 2007]. It puts an emphasis on supporting revisions when data items arrive out of order. StreamInsights is based on CEDR, and uses LINQ (Language Integrated Query) for its language [Ali et al. 2011]. Finally, StreamBase is a streaming system with a language called EventFlow that comes in two dialects, one graphical and one textual [Seyfer et al. 2011]. The textual version of EventFlow is SQL-based and has first-order composite operators.

Complex Event Processing (CEP) is another approach in the literature that we classify under the category of stream processing. CEP uses patterns over the ordered sequence of data items (simple events) to discover complex events. While SPL has a CEP operator (see Section 5.1.2 [Hirzel 2012]), SPL supports a broader set of operators rarely found in CEP systems.

The Cayuga CEP system has its own language called CEL (Cayuga Event Language), which it compiles to an NFA [Demers et al. 2007]. The SASE CEP language compiles to a formalism the authors call NFA\(^\ast\), which stands for NFAs with buffers [Agrawal et al. 2008]. The EPL language of Esper is an SQL dialect extended with event pattern-matching features [Esper 2014]. Finally, ODM (Operational Decision Management) is an example of a commercial event processing system [ODM 2014]. ODM is programmed with the JRules production rule language, with a controlled-English syntax, and compiles this to RETE networks. In contrast to CEP languages, which focus on detecting complex events via patterns, SPL is a general-purpose streaming language, with an emphasis on extensibility with new streaming operators.

Of course, there is other work on streaming languages besides these broad categories (SDF, relational streaming, CEP). Hancock is an extension of the C language with constructs for processing a stream one batch at a time [Cortes et al. 2000]. Functional Reactive Programming (FRP) is a paradigm where functions over values are lifted to functions over continuous signals [Hudak et al. 2003]. FRP uses combinators for creating topologies of signal functions. Mario is a system where the user enters tags, and the system generates stream graph that might perform the right computation [Riabov et al. 2008]. Brooklet is a stream calculus that is close to the core of SPL, and the Brooklet paper includes translations from the core of three other dataflow languages (CQL, Sawzall, and StreamIt) to Brooklet [Soulé et al. 2010]. MillWheel is a distributed streaming system with a language for describing stream graphs [Akidau et al. 2013]. In MillWheel, individual operators are implemented in C++, but unlike SPL, it does not offer a code generation framework. Storm is a distributed streaming system that does not have its own language, but instead, offers a Java library for creating stream graphs [Toshniwal et al. 2014]. ActiveSheets uses spreadsheets with Visual Basic formulas to specify streaming computations [Vaziri et al. 2014].

As discussed above, many streaming languages have been proposed before SPL or concurrently with SPL. However, SPL includes novel features not found in other
streaming languages. First, the graph abstractions in SPL are designed to facilitate distribution; most other streaming systems are either not distributed, or lack a language. Second, SPL is designed for extensibility with new operators, which use a uniform syntax but employ code generation for fast and flexible implementation. To learn more about streaming languages, the survey by Stephens offers a historical perspective, centered around classifying languages by their synchrony and determinism [Stephens 1997]. Babcock et al. survey streaming work in databases and algorithms, not languages, and argue that the primary challenge is unbounded memory, motivating windows as a common solution [Babcock et al. 2002]. Johnston et al. complement the earlier surveys by describing streaming languages for hardware description, as well as visual streaming languages [Johnston et al. 2004]. Finally, Hirzel et al. complement earlier surveys with a catalog of optimizations for streaming systems and languages [Hirzel et al. 2014].

6.2. Big Data at Rest

Stream processing deals with data in motion; dealing with large amounts of data at rest is related but different. This section gives a high-level overview of recent work on processing big data at rest, with a focus on languages, especially those that have similarities to streaming languages.

MapReduce executes a map function on data-parallel workers to convert a large but finite set of input data items into intermediate key-value pairs; redistributes these key-value pairs so each key resides on a single machine; and finally executes a reduce function on data-parallel workers to convert the intermediate key-value pairs to output data items [Dean and Ghemawat 2004]. MapReduce has some resemblance to streaming, since both mappers and reducers make a single pass over the data, and typically maintain only a small portion of the data in memory. But there is an important difference, since MapReduce blocks reducers until their input is complete, which is infeasible for infinite streaming input.

The MapReduce system has become a de-facto common runtime system for several big-data processing languages. Sawzall is a language describing a single MapReduce job, using built-in aggregators for the reduce part [Pike et al. 2005]. Other languages compile to an acyclic graph of several MapReduce jobs. PigLatin is inspired by relational algebra, but not SQL-based, and supports nested relations [Olston et al. 2008]. HiveQL is an SQL dialect, extended with explicit syntax for both map and reduce computations [Thusoo et al. 2009]. FlumeJava is a Java library, not a language, whose programming model is that of deferred evaluation over parallel collections [Chambers et al. 2010]. And Jaql uses JSON as its data model, is untyped and interpreted, and supports nested relations [Beyer et al. 2011].

Dryad is a system similar to MapReduce, but designed from the start to facilitate multi-stage execution plans, and is targeted by multiple languages, including DryadLINQ [Yu et al. 2008]. Here, LINQ stands for Language INtegrated Query, embedding SQL-style query constructs in an object-oriented language, such as C#. The query constructs can then in turn be parameterized with object-oriented code, providing natural extensibility.

Neither MapReduce nor DryadLINQ are designed for continuous (infinite) input streams, nor do they attempt to return low-latency (near real-time) responses. But given the wide-spread popularity of MapReduce and similar systems, there have been attempts at generalizing them to also handle continuous streaming data. MapReduce Online is a Java-based library that extends MapReduce with pipelining, and invokes reduce functions periodically on small batches of data [Condie et al. 2010]. Spark Streaming is a Scala-based library centered around the notion of D-streams, which are essentially streams chopped into batches of data items [Zaharia et al. 2013].
Naiad is a C#-based library that focuses on supporting cyclic iterative dataflow for both batch and stream computation [Murray et al. 2013]. In contrast to MapReduce Online, Spark Streaming, and Naiad, SPL is a language rather than just a library. Furthermore, SPL has been designed from the start for data in motion, rather than starting from a batch system and then adding streaming capabilities. This enables it to achieve lower latencies, and gives the programmer finer control over windows, instead of imposing batches on all computation.

6.3. Extensibility

In general-purpose programming languages, extensibility has been extensively studied. Here we survey a few approaches and discuss how they relate to SPL. Given a sufficiently powerful host language, a DSL (domain-specific language) does not need any new syntax, since it can instead just interpret the existing syntax differently. Hudak showed how to accomplish this in Haskell, and coined the term DSEL (domain-specific embedded language) for it [Hudak 1998]. Operator invocations in SPL resemble DSELS, since they use the standard SPL syntax and type system at the surface, but each operator can interpret it differently. In multi-stage programming languages, stage $n$ is a code-generator for stage $n + 1$, and code from different stages is embedded in each other using quote and unquote. MetaML is a multi-stage language based on ML [Taha and Sheard 1997], and 'C is a staged language based on C [Poletto et al. 1999]. Primitive operator definitions in SPL resemble two-stage programs, where the first stage is written in Perl, the second stage is written in C++, and the quote and unquote operators are written <...>.

In contrast to DSELS or homogeneous multi-stage languages, another approach is to embed new syntax into a host language. Scheme, with its minimal syntax based on s-expressions, has been leading in this approach with advanced macro systems. Other languages, including SPL, draw design and implementation lessons from Scheme for their extensibility features. A good description of advanced use cases for Scheme's macro system is the work on “languages as libraries” [Tobin-Hochstadt et al. 2011]. Most languages have more elaborate syntax than Scheme, but there has been research on extending those too, for instance, the Java Syntactic Extender [Bachrach and Playford 2001]. Jeannie takes this to the extreme by embedding Java into C and vice versa [Hirzel and Grimm 2007]. Finally, Marco is a macro system that supports multiple target languages, including C++, while still offering early error detection [Lee et al. 2012]. In contrast to these approaches, extensibility in SPL focuses on operators. SPL programmers need not parse the operator invocation; the operator instance model provides all of the context. This kind of focused extensibility is appropriate for programmers who are experts in their domain, but not in compilers.

While there is a rich body of work on extending general programming languages, there is relatively little work on extending streaming languages. DBToaster uses C++ code generation for incremental view maintenance in a database, but unlike SPL, it focuses on relational algebra only [Ahmad and Koch 2009]. Brooklet accomplishes extensibility by compiling multiple source languages to a single target, as opposed to SPL, which is extensible by generating code for operators [Soulé et al. 2010]. StreamBase offers composite operators, and like composites in SPL, those can process streams with additional fields beyond those explicitly defined [Seyfer et al. 2011]. However, StreamBase composites are first-order, while SPL composites are higher-order. StreamInsights offers extensions in the object-oriented language hosting the LINQ constructs, but new operators are second-class citizens in both syntax and semantics [Ali et al. 2011]. The expressiveness, performance, and uniformity of SPL's extensibility is unmatched among other streaming languages.
7. CONCLUSION

This paper presented the SPL language. SPL is designed around the 3 Vs of big-data streaming. For high data volumes, programs are stream graphs without shared state or central schedule, easy to distribute on clusters of multi-cores. For velocity, SPL operators use code generators to achieve good single-thread performance. And for variety, SPL offers an extensibility framework for operators addressing diverse application domains and data formats.

REFERENCES


Kevin S. Beyer, Vuk Ercegovac, Rainer Gemulla, Andrey Balmin, Mohamed Eltabakh,


Michael I. Gordon, William Thies, and Saman Amarasinghe. 2006. Exploiting coarse-
grained task, data, and pipeline parallelism in stream programs. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 151–162.


neering (ASE). 103–113.


A. SAMPLE APPLICATION

Figs. 22 and 23 show a sample SPL application, SuspectedHosts, with a visualization in Fig. 21. The purpose of the application is to monitor the system messages file on a Linux system (commonly found at /var/log/messages) looking, in real-time, for remote hosts that are trying to break into a host on the current cluster. Computers that face the Internet are typically under constant attack. Malicious hosts try to break in through ssh by constantly trying username and password combinations from large dictionaries. The sshd service logs every attempt in /var/log/messages.

A.1. Application Logic

The main composite of the application, SuspectedHosts, will perform the following steps:

1. Read /var/log/messages in the invocation of the FileSource operator, producing a tuple for each line on the Lines stream (lines 7–11).
2. Parse the line into a structured tuple, but with an unstructured service message on the ParsedLines stream (lines 18–28).
3. Filter out the tuples that do not belong to the sshd service, and do not represent a failed login, with a Filter operator invocation, submitting results to the RawFailures stream (lines 34–38).
4. Further parse the msg field of the tuple, producing the Failures stream (lines 45–54).
5. Look for remote hosts that have failed to login 5 times in the past minute by invoking the SuspectFind composite, producing the RealTime stream (lines 60–64).
6. Publish the suspected malicious hosts over the network via a TCPSink (lines 65–68).

The composite SuspectFind reads a stream of sshd authentication failure tuples, and produces a stream of tuples that represent suspected malicious remote hosts. It is parameterized by the number of attempts ($attempts) and the duration of time ($seconds). It will look for suspects through the following steps:

1. Aggregate $attempts number of failed attempts for each remote host with an Aggregate operator (lines 125–130). Each time it accumulates $attempts failed attempts, it produces a tuple with the maximum and minimum time associated with those attempts.
2. Filter out attempts where the duration of time between the first and last attempt is greater than $seconds number of seconds with a Filter operator (lines 131–133).
3. Compute what the actual duration was with a Functor (lines 134–136).

A.2. Application Configuration

The SuspectedHosts sample application has several instances of system controls independent of the main logic.

A.2.1. User-directed fission. We parallelized both the ParsedLines and Failures operators. Since both of the parsing operations do not maintain state across firings, it is easy to extract data parallelism from them. However, user-directed fission does not maintain tuple ordering, which the logic of the SuspectFind composite depends on. We apply extra logic to put tuples back in order. First, we add sequence numbers to each tuple before the parallel region with the AssignSeqno composite. After the parallel region, we use the OrderedMerge composite to put the tuples back in their proper order.

A.2.2. PEs and threaded ports. We explicitly grouped the operators into 4 PEs, with the labels FileReadingPE, ParsingPE, FailuresPE, and SuspectsPE. (The TCPSink operator invocation is implicitly in an unlabeled PE.) We also placed threaded ports on
computation-heavy operators that appear in a PE with another computation-heavy operator (lines 37 and 129).

Note that only one operator invocation has the label ParsingPE, but that operator is invoked with user-directed fission (line 17). Because no non-parallelized operators are inside of that PE, Streams will create 7 instances of that PE. However, multiple operators have the label FailuresPE, and one of them has user-directed fission. Streams cannot replicate that PE to exploit data parallelism, because there are non-parallelized operators in that PE. Instead, Streams replicates just the parallelized operators, and inserts threaded ports before those operators to gain parallelism.

A.3. Application Execution
The SuspectedHosts application runs continuously, constantly monitoring system messages and reporting suspected malicious remote hosts. It is naturally highly parallel: every operator in a pipeline can execute simultaneously. While the FileSource operator invocation is reading a new tuple from the system messages file, the operators in SuspectFind can look for malicious login attempts in the tuples that the FileSource operator already produced.
composite SuspectedHosts {
  type
  LogLine = tuple<uint64 seqno, timestamp time, rstring hostname, rstring srcv, rstring msg>;
  Failure = tuple<uint64 seqno, timestamp time, rstring uid, rstring euid, rstring tty, rstring rhost, rstring user>;

  graph
  stream<rstring line> Lines = FileSource() {
    param format: line;
    file: "\var/log/messages";
    config placement: partitionColocation("FileReadingPE");
  }
  stream<rstring line, uint64 seqno> LinesSeqno = AssignSeqno(Lines) {
    param Out: tuple<rstring line, uint64 seqno>;
    config placement: partitionColocation("FileReadingPE");
  }

  @parallel(width=7)
  stream<LogLine> ParsedLines = Custom(LinesSeqno) {
    logic onTuple LinesSeqno: {
      list<rstring> tokens = tokenize(line, ",", false);
      rstring time = tokens[2] + ".000";
      timestamp t = timeStringToTimestamp(date, time, false);
      submit({seqno = LinesSeqno.seqno, time = t, hostname = tokens[3], srcv = tokens[4],
        msg = flatten(tokens[5:])}, ParsedLines);
    }
    config placement: partitionColocation("ParsingPE");
  }
  stream<LogLine> ParsedLinesMerged = OrderedMerge(ParsedLines) {
    param key: ParsedLines.seqno;
    config placement: partitionColocation("FailuresPE");
  }

  stream<LogLine> RawFailures = Filter(ParsedLinesMerged) {
    param filter: findFirst(srcv, "sshd", 0) != -1 && findFirst(msg, "authentication failure", 0) != -1;
    config placement: partitionColocation("FailuresPE");
    threadedPort: queue(ParsedLinesMerged, Sys.Wait);
  }
  stream<LogLine> RawFailuresSeqno = AssignSeqno(RawFailures) {
    param Out: LogLine;
    config placement: partitionColocation("FailuresPE");
  }

  @parallel(width=4)
  stream<Failure> Failures = Custom(RawFailuresSeqno) {
    logic onTuple RawFailuresSeqno: {
      list<rstring> tokens = tokenize(msg, ";", false);
      list<rstring> values = tokenize(tokens[1], ",", false);
      submit({seqno = RawFailuresSeqno.seqno, time = RawFailuresSeqno.time, uid = values[2],
        euid = values[4], tty = values[6], rhost = values[9],
        user = size(values) == 12 ? values[11]: ""}, Failures);
    }
    config placement: partitionColocation("FailuresPE");
  }
  stream<Failure> FailuresMerged = OrderedMerge(Failures) {
    param key: Failures.seqno;
    config placement: partitionColocation("SuspectsPE");
  }

  stream<Suspect> RealTime = SuspectFind(FailuresMerged) {
    param attempts: 5u;
    seconds: 60.0;
    config placement: partitionColocation("SuspectsPE");
  }
  } as Sink = TCPSink(RealTime) {
    param role: server;
    port: "http";
}

Fig. 22: The main composite of the SuspectedHosts application.
rstring flatten(list<rstring> lst)
{
mutable rstring str = "";
for (rstring e in lst) {
    str += e + " ";
}
return str;
}

composite AssignSeqno(input In; output Out) {
    param type $Out;
    graph
    stream<$Out> Out = Functor(In) {
        logic state:
        mutable uint64 _count = 0;
    }
    onTuple In: {
        _count = _count + 1;
    }
    output Out: seqno = _count;
}

composite OrderedMerge(input In; output Out) {
    param attribute $key;
    graph
    stream<In> Out = Custom(In) {
        logic state:
        mutable map<uint64, tuple<In>> _tuples;
        mutable uint64 _next = 1;
    }
    onTuple In: {
        if (_next == $key) {
            submit(In, Out);
            ++_next;
            while (_next in _tuples) {
                submit(_tuples[<_next], Out);
                removeM(_tuples, _next);
                ++_next;
            }
        } else {
            _tuples[<_key] = In;
        }
    }
}

type Suspect = tuple<
            timestamp diff, timestamp last, uint32 attempts, rstring rhost, rstring user>
        ;
composite SuspectFind(input Failure; output Diff) {
    param expression<uint32> $attempts;
    param expression<float64> $seconds;
    type FailureRange = tuple<
            timestamp max, timestamp min, rstring rhost, rstring user>
        ;
    graph
    stream<FailureRange> Range = Aggregate(Failure) {
        window Failure: tumbling, count($attempts), partitioned;
        param partitionBy: rhost;
        output Range: max = Max(time), min = Min(time), user = Max(user);
        config threadedPort: queue(Failure, Sys.Wait);
    }
    stream<FailureRange> Cutoff = Filter(Range) {
        param filter: max - min < (timestamp)$seconds;
    }
    stream<Suspect> Diff = Functor(Cutoff) {
        output Diff: diff = max - min, last = max, attempts = $attempts;
    }
}

Fig. 23: Helper functions, composites and types for SuspectedHosts.