IBM Streams Processing Language: Analyzing Big Data in motion

M. Hirzel
H. Andrade
B. Gedik
G. Jacques-Silva
R. Khandekar
V. Kumar
M. Mendell
H. Nasgaard
S. Schneider
R. Soulé
K.-L. Wu

The IBM Streams Processing Language (SPL) is the programming language for IBM InfoSphere® Streams, a platform for analyzing Big Data in motion. By "Big Data in motion," we mean continuous data streams at high data-transfer rates. InfoSphere Streams processes such data with both high throughput and short response times. To meet these performance demands, it deploys each application on a cluster of commodity servers. SPL abstracts away the complexity of the distributed system, instead exposing a simple graph-of-operators view to the user. SPL has several innovations relative to prior streaming languages. For performance and code reuse, SPL provides a code-generation interface to C++ and Java®. To facilitate writing well-structured and concise applications, SPL provides higher-order composite operators that modularize stream sub-graphs. Finally, to enable static checking while exposing optimization opportunities, SPL provides a strong type system and user-defined operator models. This paper provides a language overview, describes the implementation including optimizations such as fusion, and explains the rationale behind the language design.

Introduction

This paper provides an overview of the IBM Streams Processing Language (SPL) [1] for InfoSphere* Streams [2]. *Stream processing* is a programming paradigm that processes continuous data streams via a stream graph. A *stream* is a sequence of data items, and a *continuous stream* is (at least conceptually) an infinite sequence of data items. A *stream graph* is a directed graph in which each edge is a stream and each vertex is an operator instance. *Operators* are stream transformers, sources, or sinks. Stream processing is important because it can analyze large volumes of data immediately as it is being produced. Continuous data streams are ubiquitous: they arise in telecommunications, health care, financial trading, and transportation, among other domains. Timely analysis of such streams can be profitable (in finance) and can even save lives (in health care). Furthermore, often the data volume is so high that it cannot be stored to disk or sent over slow network links before being processed. Instead, a streaming application can analyze

continuous data streams immediately, reducing large-volume input streams to low-volume output streams for further storage, communication, or action.

Aside from the above-mentioned motivation from the application perspective, stream processing is also motivated by current technology trends. Hardware manufacturers are increasingly building multi-core chips, and vast clusters of servers are becoming more common. All this intra-machine and inter-machine parallelism is difficult to exploit with conventional programming paradigms that are based on the von Neumann model with shared state and complex control flow and synchronization. Much recent work has attempted to address this difficulty. In stream processing, each vertex of the stream graph can run in parallel on a different core of a chip or a different server of a cluster, subject only to data availability. This makes it easier to exploit the nested levels of hardware parallelism, which is important for handling massive data streams or performing sophisticated online analytics.

A streaming language such as SPL makes it easier to write streaming applications. The alternative is to either create applications graphically, by editing a visual instead of a

```
1.  composite BargainFinder {
2.   type
3.     TradeOrQuote  = tuple<rstring sym, timestamp ts, decimal64 price, decimal64 vol>;
4.     PreVwap       = tuple<rstring sym, timestamp ts, decimal64 priceVol, decimal64 vol>;
5.     Vwap          = tuple<rstring sym, timestamp ts, decimal64 vwap>;
6.     Bargain       = tuple<rstring sym, timestamp ts, decimal64 index>;
7.   graph
8.     stream<TradeOrQuote> Trades = TCPSource() {
9.       param   role            : server;
10.              port            : 40000u;
11.     }
12.     stream<PreVwap> PreVwaps = Aggregate(Trades) {
13.       window  Trades          : sliding, delta(ts, 60.0), count(1), partitioned;
14.       param   partitionBy     : sym;
15.       output  PreVwaps        : priceVol = Sum(price*vol), vol = Sum(vol);
16.     }
17.     stream<Vwap> Vwaps = Functor(PreVwaps) {
18.       output  Vwaps           : vwap = priceVol / vol;
19.     }
20.     stream<TradeOrQuote> Quotes = TCPSource() {
21.       param   role            : server;
22.              port            : 40001u;
23.     }
24.     stream<Bargain> Bargains = Join(Vwaps; Quotes) {
25.       window  Vwaps           : sliding, count(1), partitioned;
26.               Quotes          : sliding, count(0);
27.       param   equalityLHS     : Vwaps.sym;
28.               equalityRHS     : Quotes.sym;
29.               partitionByLHS  : Vwaps.sym;
30.       output  Bargains        : index = vwap > price
31.                                          ? price * exp(vwap - price) : 0d;
32.     }
33.     () as Sink = TCPSink(Bargains) {
34.       param   role            : client;
35.               address         : "10.0.0.2";
36.               port            : 40002u;
37.     }
38.  }
```
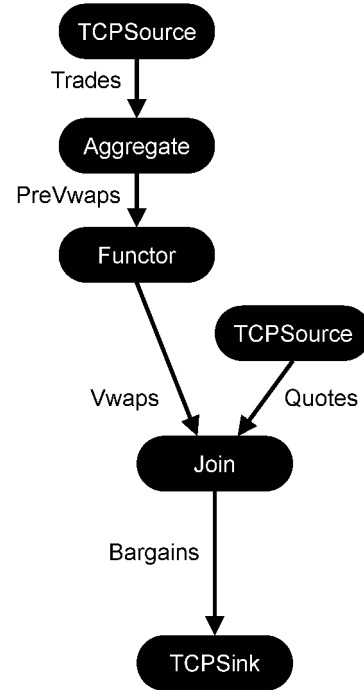

Figure 1

Streams Processing Language (SPL) program for computing the VWAP (volume-weighted average price) of a stream of stock trades and identifying bargains in a stream of stock quotes.

textual representation of the stream graph, or to use a non-streaming programming language with streaming libraries. Both approaches have drawbacks. First, providing only a graphical approach restricts expressiveness (i.e., some information that can be easily expressed as text is missing in typical graphical representations) and, in any case, is usually implemented internally by a streaming intermediate language [3]. Second, the library approach is typically less concise and affords fewer opportunities for error checking and optimization than a dedicated language [4]. In contrast, a streaming language can offer more expressiveness, error checking, and performance. The next subsection of the introduction makes the concept of stream processing concrete by introducing SPL via an example, and the subsection after this places SPL in context by reviewing related efforts in industry and academia.

The particulars: SPL overview

The central concepts of SPL are streams and operators. An SPL application corresponds to a stream graph. Operators without input streams are sources, and have their own thread of control. Most other operators fire only when there is at least one data item on one of their input streams. More specifically, when an operator fires in SPL, it executes a piece of code to handle an input data item. An operator firing consumes one input data item and can submit any number of data items on output streams. A stream delivers data items from one operator to another in the order they were submitted.

Figure 1 shows an example bargain-finding financial application, with the SPL code on the left and the equivalent stream graph on the right. The code in the figure uses bold font for SPL keywords; we also follow this convention

for SPL keywords throughout this paper. The graph depicts each vertex next to the corresponding operator invocation in the code. All operator invocations have the same high-level syntax. The syntax is declarative, because it only describes *what* each operator does, but not *how*. The following discussion explains the invocation of the Aggregate operator as an example for that syntax. Line 12 in Figure 1 is

$$\mathbf{stream}\langle\text{PreVwap}\rangle\text{PreVwaps} = \text{Aggregate}(\text{Trades})$$

and it creates the PreVwaps stream, with data items of type PreVwap, by invoking the Aggregate operator on the Trades stream. VWAP stands for volume-weighted average price, and the PreVwaps stream carries preliminary information for calculating the VWAP. This example has one input and one output stream, but in general, an operator invocation can have zero or more input and output streams. The following lines of code are indented, because they configure the same operator invocation that started in Line 12. In Line 13, we have:

$$\mathbf{window}\ \text{Trades} : \text{sliding}, \text{delta}(\text{ts}, 60.0), \text{count}(l), \text{partitioned};$$

and this specifies that this instance of the Aggregate operator works on sliding windows of size delta(ts, 60.0). A *window* stores recent data items from a stream, and a *sliding* window retires the oldest data items incrementally as new data items are added. Identifier "ts" refers to a timestamp attribute of stream Trades, and delta(ts, 60.0) creates a window in which timestamps are at most 60 seconds apart, in other words, a 1-minute window. The count(1) means that the window slides by one data item at a time. The "partitioned" at the end of the window clause means that the window is maintained separately for each different value of the partition key, in this case the "sym" attribute as specified by the "partitionBy" parameter in Line 14:

$$\mathbf{param}\quad \text{partitionBy} : \text{sym};$$

Identifier "sym" refers to a string attribute of the Trades stream, which contains a stock symbol, for example, "IBM." In this case, the corresponding data for each stock is aggregated separately. Partitioning can be used for optimization, as we shall see later in this paper. Finally, Line 15 is

$$\mathbf{output}\quad \text{PreVwaps} :$$
$$\text{priceVol} = \text{Sum}(\text{price} * \text{vol}), \text{vol} = \text{Sum}(\text{vol});$$

and it assigns the attributes of data items on the output stream. Output attribute "priceVol" is the sum, for each data item in the window, of the price × vol products; and output attribute "vol" is the sum of the input volumes. In this example, Sum is an operator-specific intrinsic output function. For convenience, the Aggregate operator forwards other attributes that are not explicitly mentioned in the output

clause unmodified from input to output, if they appear in both the incoming and outgoing types.

In general, the operator determines which subset of the clauses **window**, **param**, and **output** is needed, and what the parameters in the **param** clause mean. This design gives SPL a combination of generality and consistency. For generality, SPL supports a wide variety of operators, which can also be user-defined. For consistency, all operator invocations follow the same basic syntax.

Based on the description so far, we can now understand the entire **graph** clause in Lines 7 to 37 of Figure 1. In essence, this application reads Trades from a TCP (Transmission Control Protocol) connection, computes preliminary aggregate information on trades in the PreVwaps stream, computes the VWAP, joins it with the Quotes stream from another TCP source, and computes a bargain index as its output.

At the outermost level, an SPL application consists of a main composite operator—in this case, the BargainFinder operator (Line 1). Aside from the **graph** clause, a composite operator can also contain type definitions (Lines 2–6). For example, Line 4,

$$\text{PreVwap} = \mathbf{tuple}\langle\mathbf{rstring}\ \text{sym}, \mathbf{timestamp}\ \text{ts},$$
$$\mathbf{decimal64}\ \text{priceVol}, \mathbf{decimal64}\ \text{vol}\rangle;$$

defines the PreVwap type as a tuple type with four attributes: sym, ts, priceVol, and vol. In Figure 1, all tuple types only contain attributes of primitive types. However, in general, a tuple attribute in SPL can hold any type, such as a list, a map, or even another tuple. The hierarchical type system of SPL allows programmers to select the stream granularity they find most natural.

The bigger picture: Related technologies
There are two main approaches to processing Big Data (i.e., massive amounts of data): batch processing and stream processing. The distinction is that in batch processing, there is a finite amount of input data, whereas in stream processing, streams are assumed to be "infinite." Batch processing in its pure form reports results only when all computation is done, whereas stream processing produces incremental results as soon as they are ready. This means that batch processing is associated with different optimization goals (e.g., it need not optimize for latency of intermediate results). Furthermore, finiteness affords batch processing the opportunity to wait until one computation stage is complete before starting another, enabling it to perform operations such as globally sorting all intermediate results. Prominent recent examples of batch processing for Big Data include MapReduce [5], Dryad [6], and various languages implemented on top of MapReduce, such as Pig Latin [7] and Jaql [8]. Hybrids between the extremes of batch and streaming exist, such as MapReduce Online [9] and

DBToaster [10]. Batch processing can be emulated by stream processing, because both paradigms are implemented by data-flow graphs [11]. The following paragraphs describe the dominant technologies for stream processing: synchronous data flow, relational streaming, and complex event processing.

In the *synchronous data flow* (SDF) paradigm, each operator has a fixed rate of output data items per input data item. In other words, an operator consumes M input data items and produces N output data items each time the operator fires, where M and N are fixed and known statically. Prominent examples for SDF programming languages include Lustre** [12], Esterel [13], and StreamIt [14]. SDF has the advantage that the compiler can exploit static data rates for optimizations: it can pre-compute a repeating schedule to avoid runtime scheduling overheads, and it can pre-compute buffer sizes to avoid runtime dynamic allocation overheads. On the downside, SDF is inflexible; in many applications, data rates are not static but dynamic, for instance, in time-based windows or data-dependent filtering. SPL is not an SDF language because it allows dynamic data rates by default.

Another technology for stream processing is *relational streaming*. Relational streaming adopts the relational model from databases: a relation is a bag (i.e., an unordered collection with duplicates) of tuples and a tuple consists of named flat attributes. This enables relational streaming systems to adopt the relational operators (select, project, aggregate, join, etc.) from databases, with well-defined semantics and well-developed optimizations. Prominent examples of relational streaming systems include TelegraphCQ [15], the STREAM system underlying CQL (Continuous Query Language) [16], Aurora [17] and Borealis [18], StreamInsight [19], and SPADE (Stream Processing Application Declarative Engine) [20]. SPL supports relational streaming, because it uses tuples as a central data type, and provides relational operators in its standard library. However, SPL is more general than this: tuples can nest, topologies can be cyclic, the library contains many more operators, and users can define their own, fully general, new operators.

A special case of stream processing is *complex event processing* (CEP). CEP terminology refers to data items in input streams as *raw events* and to data items in output streams as *composite* (or *derived*) *events*. A CEP system uses patterns to inspect sequences of raw events and then generates a composite event for each match, for example, when a stock price first peaks and then dips below a threshold. Prominent CEP systems include NiagaraCQ [21], SASE (Stream-based and Shared Event processing) [22], Cayuga [23], IBM WebSphere* Operational Decision Management (WODM) [24], Progress Apama [25], and TIBCO BusinessEvents [26]. Etzion and Niblett provide an introduction to CEP [27]. SPL is a streaming language, not a CEP language, because it does not primarily focus on matching patterns over event sequences. However, it is possible to implement CEP in SPL by wrapping a CEP matching engine in an operator [28].

Other contemporary streaming systems akin to InfoSphere Streams include StreamBase [3], S4 [4], and Storm [29]. As mentioned, SPL differentiates InfoSphere Streams from those systems by providing a declarative language while, at the same time, supporting fully general user-defined operators.

Contributions and paper organization
Compared with the related work we outlined, SPL makes three primary contributions. The following three sections of this paper explain each of these contributions. First, SPL distinguishes itself by its approach for interfacing with non-streaming languages. Specifically, SPL takes a code-generation approach, where fast streaming operators are implemented by emitting optimized code in a non-streaming language. Second, SPL distinguishes itself by how it provides modularity for large streaming applications. Specifically, the higher-order composite operators of SPL encapsulate parametric stream sub-graphs. Third, SPL distinguishes itself by its strong type system and operator models. Together, they enable users to write modularized, large, and efficient streaming applications, which are essential for developing big-data analytics.

Interfacing with C++ and Java code
Some SPL applications need only operators from the standard library of SPL, but other applications need their own new operators. For instance, the example in Figure 1 uses only standard library operators (TCPSource, Aggregate, Functor, Join, and TCPSink). However, imagine that the input arrives in XML (Extensible Markup Language) format. Because there are already powerful XML libraries for non-streaming languages such as C++ or Java, an XML operator for SPL is best implemented in C++ or Java as well, so it can reuse those libraries [30]. For this particular functionality, one could argue that such an operator could be added as a built-in feature. However, in general, there are so many possible features that it is better to have a general extensibility framework available to the SPL user.

There are four design goals for creating an interface between SPL and native C++ or Java code. The first goal is generality: users should be able to write new operators that are just as powerful as those in the standard library. The second goal is consistency: all operators should "look and feel" the same in an SPL application. For example, the SPL code in Figure 1 uses several different operators, but the high-level syntax is the same in all cases. In SPL, this syntax does not change whether the operator is part of the library or written by users. The third goal is legacy code reuse: SPL operators can internally call code written in other languages.
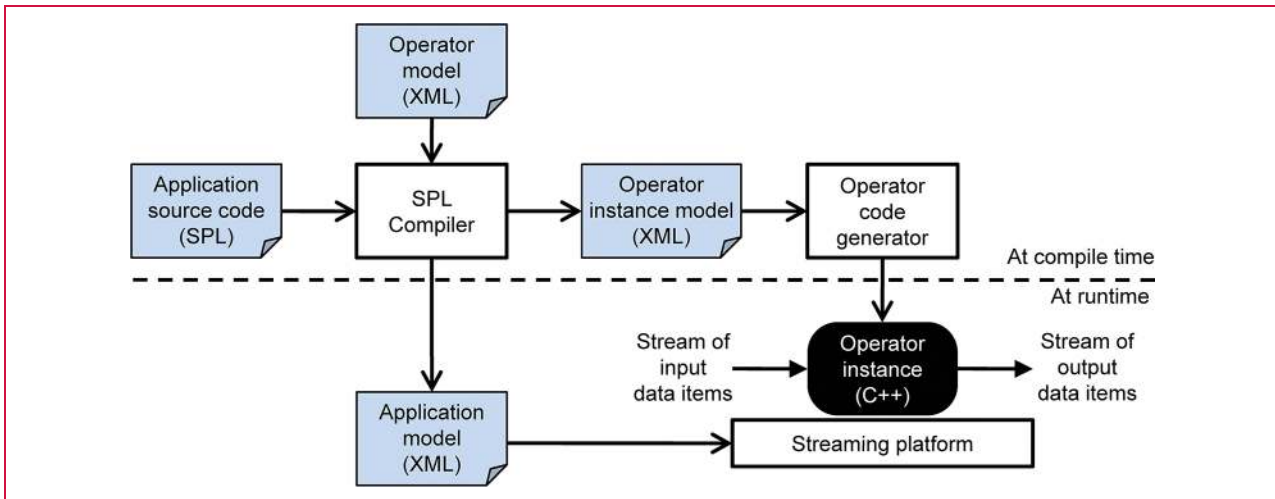
Figure 2

Compiling and running a Streams Processing Language (SPL) application.

The fourth goal is flexibility and performance. In general, operators can be configured in several different ways. For instance, Figure 1 uses the Aggregate operator with a delta-based window to compute sums, but other applications can use the same operator with other window configurations (e.g., count-based) or compute different kinds of aggregations (e.g., average or maximum). SPL supports these flexible configurations with high performance by allowing the operator definition to generate different code depending on how the operator invocation is configured. Using code generation means that the decisions that can be made at compile time do not incur run-time overhead.

In SPL, an operator written in C++ or Java is called a *primitive operator*, in contrast to the composite operators that encapsulate sub-graphs and are the subject of the next section. **Figure 2** illustrates the code-generation approach of SPL for primitive operators. Each primitive operator definition consists of two artifacts: an *operator model* and an *operator code generator*. The operator model (top of Figure 2) is an XML document that describes the operator to the SPL compiler [31]. For example, the operator model for the Aggregate operator indicates that this operator is windowed, and has an optional partitionBy parameter, and can output a variety of aggregations via one of several operator-internal intrinsic functions such as Sum, Max, and Average. Later sections of this paper will also explore how to use the operator model for optimization. The operator code generator (top right of Figure 2) is responsible for generating the C++ code for a particular invocation of the operator in the SPL code. For example, the code generator for the Aggregate operator emits the necessary data structures for incrementally maintaining the required

aggregation state as each data item enters or leaves the window.

All primitive operators in SPL, including both library operators and user-defined operators, employ the same code-generation framework. When the SPL compiler processes the application source code written in SPL (left of Figure 2), it encounters several operator invocations. For each operator invocation, it checks the operator model (top of Figure 2), and emits an operator instance model (middle of Figure 2) that contains all the customization information necessary for the code generator. Then, the SPL compiler runs the code generators for each of the operator instance models, yielding a set of operator instances (bottom right of Figure 2). Aside from using the operator code generators to generate all of the operator instances, the SPL compiler also emits an application model (bottom left of Figure 2). At runtime, the streaming platform (InfoSphere Streams) consults the application model to launch all the operators for the stream graph. Each runtime operator instance is a stream transformer, consuming input data items and producing output data items, with the streaming platform responsible for transport, resilience, and overall management of the runtime environment.

While the ability to write code generators for new operators is powerful, it also assumes a high level of sophistication on the part of the programmer. It means that the operator developer is de-facto writing a small compiler. To facilitate this, SPL provides a templating mechanism. SPL code-generation templates consist of C++ code to-be-generated interspersed with Perl code for customization in much the same way that a PHP (a recursive acronym that stands for PHP: Hypertext Preprocessor) program consists

```
1.  stream<Inputs> Uniques = Custom(Inputs) {
2.    logic
3.      state : {
4.        mutable boolean _first = true;
5.        mutable Inputs _prev = { };
6.      }
7.      onTuple Inputs : {
8.        boolean isUnique = _first || Inputs != _prev;
9.        _first = false;
10.       if (isUnique) {
11.         _prev = Inputs;
12.         submit(Inputs, Uniques);
13.       }
14.     }
15.  }
```
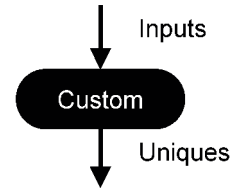
Figure 3

Invocation of the Custom operator with a logic clause.

of HTML (HyperText Markup Language) to be generated interspersed with PHP code for customization. The Perl code is executed at compile time to perform advanced error checks and to generate more or customized C++ code. In cases where no customization is required, this reduces to just writing the C++ code to be generated without any interspersed Perl code. In other words, operator developers do not need to know or care about the code-generation framework except in advanced cases. The C++ runtime application programming interfaces (APIs) provide introspection facilities that are often sufficient; for instance, parameters from the SPL source code are available not only at compile time, but also at runtime. As mentioned, a primitive operator definition consists of two parts: the operator model and the code generator. SPL is distributed with an Eclipse-based environment that makes the operator model easy to write as well.

Finally, in some cases, instead of defining an entirely new operator from scratch, users only want to add some logic to an existing operator, for example, to handle the arrival of a new data item at an input port. For this purpose, SPL syntax supports a **logic** clause alongside the **window**, **param**, and **output** clauses we already saw in Figure 1. The **logic** clause contains handlers written in SPL itself, without the need to write code in a separate, more low-level language. The Custom operator in the standard library has no built-in behavior of its own and instead allows all functionality to reside in the logic clause. **Figure 3** shows an example of using logic in a Custom operator to deduplicate a stream. The **logic state** sub-clause (Line 3) declares variable _first (Line 4) to keep track of whether the current tuple is the first in the stream, and _prev (Line 5) to remember the previous tuple. The **logic onTuple** sub-clause (Line 7) executes each time a tuple arrives on the specified input port. If the current tuple

(Inputs) is first or differs from the previous tuple, it is unique and gets submitted to the output stream (Uniques, Line 12).

The code-generation framework of SPL meets its generality and consistency goals: it has been used for dozens of operators, all of which have the same high-level syntax. Some operators in the standard library are highly customizable, such as Join, which selects different algorithms depending on the type of relational join. Operators reuse legacy code by accessing libraries and platform services in C++ and Java. In extreme cases, the implementation of an operator can include a full-fledged compiler, such as translating CEP patterns to efficient finite state machines [28].

Modularizing via composite operators

Large SPL applications can have stream graphs containing hundreds of operator invocations. For such a large graph to be comprehensible, it is best if it is hierarchically composed from smaller sub-graphs. For this purpose, SPL provides *composite operators*. A composite operator encapsulates a stream sub-graph. When studying an SPL application, one can first look at the composite operator as a single vertex in a collapsed stream graph, and then look at the details by expanding it into the vertices of its own sub-graph. Aside from modularity, composite operators also help with code reuse. If the same sub-graph occurs multiple times (with small variations), each occurrence can be represented by a different invocation of the same composite operator (with parameters). For example, one could have multiple financial applications that all have the same sub-graph for reading a stream of trades and computing their volume-weighted average prices. By placing this sub-graph in a composite, one can avoid code duplication. This is a key SPL feature that enables development of complex Big Data applications.
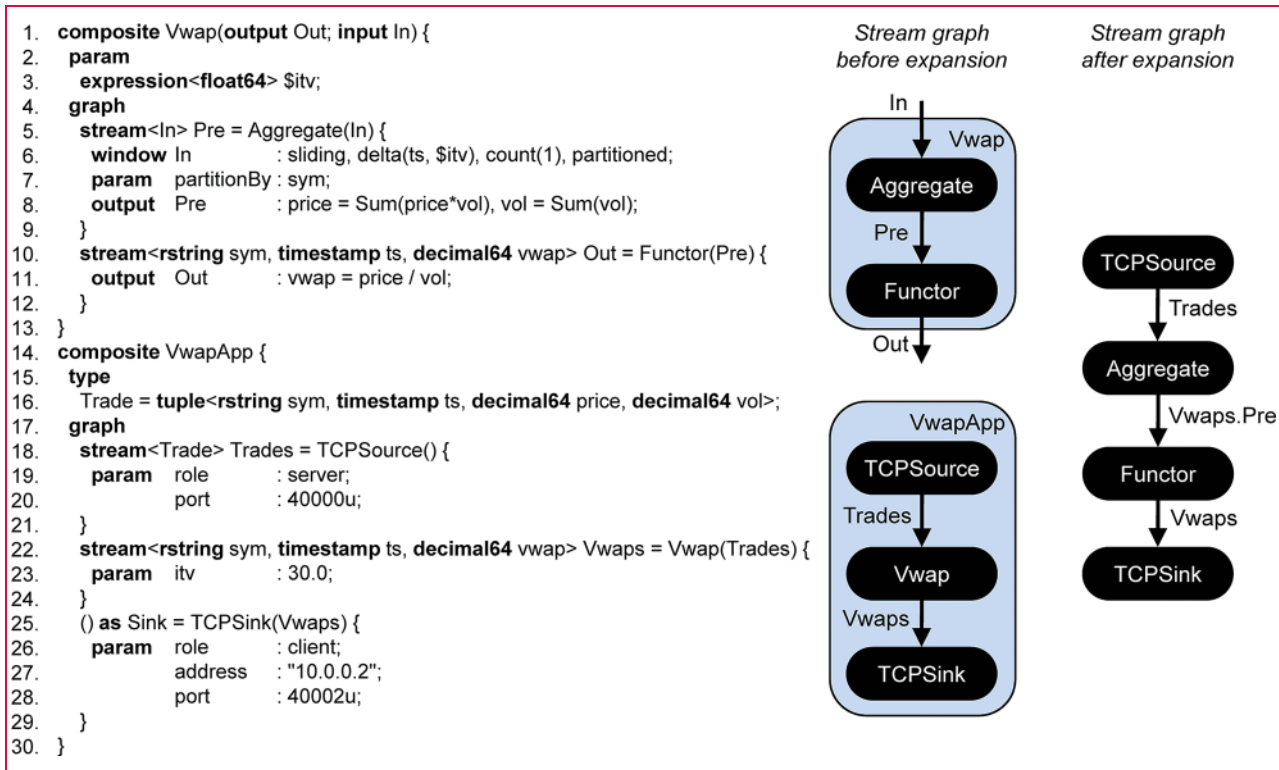
```
1.  composite Vwap(output Out; input In) {
2.    param
3.      expression<float64> $itv;
4.    graph
5.      stream<In> Pre = Aggregate(In) {
6.        window In          : sliding, delta(ts, $itv), count(1), partitioned;
7.        param   partitionBy : sym;
8.        output  Pre          : price = Sum(price*vol), vol = Sum(vol);
9.      }
10.     stream<rstring sym, timestamp ts, decimal64 vwap> Out = Functor(Pre) {
11.       output  Out          : vwap = price / vol;
12.     }
13. }
14. composite VwapApp {
15.   type
16.     Trade = tuple<rstring sym, timestamp ts, decimal64 price, decimal64 vol>;
17.   graph
18.     stream<Trade> Trades = TCPSource() {
19.       param   role : server;
20.               port : 40000u;
21.     }
22.     stream<rstring sym, timestamp ts, decimal64 vwap> Vwaps = Vwap(Trades) {
23.       param   itv  : 30.0;
24.     }
25.     () as Sink = TCPSink(Vwaps) {
26.       param   role    : client;
27.               address : "10.0.0.2";
28.               port    : 40002u;
29.     }
30. }
```

Figure 4

Composite operator expansion for an application that computes the VWAP (volume-weighted average price) of a stream of stock trades.

There are three design goals for composite operators in SPL. As mentioned, the first goal is consistency: composite operators should have the same look-and-feel as primitive operators when they are invoked from an SPL application. The second goal is flexibility: a composite does not need to expand to identical sub-graphs each time, but rather, can have variations, which are controlled by parameters. The third goal is hygiene: composite operator expansion does not change the bindings of SPL names to entities. Hygiene is important for making code easier to understand. Names in a composite operator invocation are scoped such that no expansion is needed to resolve them and such that there is no accidental name capture after expansion. Capturing a name means changing its binding, which is what hygiene prevents [32].

Figure 4 shows an example of composite operator expansion in SPL. The SPL code on the left defines the Vwap composite operator (Lines 1–13), and then uses it in the VwapApp composite operator (Lines 14–30). The stream graphs in the middle show these two composites before expansion, aligned with the corresponding source code. VwapApp is the *main* composite operator of the application, since it has no input or output streams, and it invokes Vwap in Lines 22 to 24. This operator invocation looks just like any other operator invocation: if one examines only Lines 22 to 24, one cannot tell whether Vwap is a primitive operator or a composite operator. Using the same high-level syntax helps consistency. After the SPL compiler expands composite operators, it obtains the stream graph on the far right of Figure 4. That graph is obtained by starting with the main composite and replacing invocations of other composites by their sub-graphs, while renaming streams as necessary to prevent accidental name capture. In this example, Vwap is replaced by Aggregate and Functor, and stream Pre is renamed into Vwaps.Pre. Without that renaming, using the composite twice would cause the name to be ambiguous and unhygienic.

Composite operators are flexible in terms of both types and parameters. The Vwap composite operator from Figure 4 is flexible with respect to the type of its input stream, In. As mentioned, the In stream must have at least the attributes ts, sym, price, and vol, because those are mentioned by the Aggregate operator in Lines 5 to 9. The presence of these attributes is checked after expansion, when the compiler type-checks the expanded code. If they are missing or are used with the wrong type, the compiler reports an error, along with the expansion context to make the error easier to understand. Note, however, that the actual input stream

can contain more attributes. Furthermore, the type of In is used to define the type of Pre in Line 5:

$$\textbf{stream}\langle\text{In}\rangle\text{Pre} = \text{Aggregate}(\text{In})$$

Usually, the $\textbf{stream}\langle T \rangle$ syntax expects a type name for T. However, for convenience, SPL permits use of a stream name where a type name is expected. In this case, the result is the type of the tuples on the stream, making it easy to write operator invocations whose input and output tuples have the same attributes. Aside from reusing types unmodified, SPL also supports defining new tuple types by adding additional attributes to an existing tuple type. For example, $\textbf{type } T_1 = \textbf{tuple}\langle T_2, T_3 \rangle$ defines a new tuple type T_1, which contains the union of all attributes from types T_2 and T_3. The details are in the SPL specification [1].

Figure 4 also contains an example of a parameter being passed to a composite operator. The Vwap composite operator declares the $itv parameter in Line 3 and uses it to define the window size in Line 6. The VwapApp composite operator uses parameter "itv" when it invokes Vwap in Line 23. The mandatory dollar sign at the start of the parameter name in the composite operator definition serves as a reminder to the programmer that this name gets replaced at compile time. While this example only passes a constant to a parameter, in general, operator parameters in SPL can also be unevaluated expressions (e.g., filtering conditions), attributes, functions, types, or even other operators. A composite operator that takes another operator as a parameter is a *higher-order* composite. This feature is useful for making graph shapes reusable without requiring each instance of the sub-graph to contain the exact same operators. Prior work on hierarchically composed stream graphs lacked higher-orderness (e.g., see the prior work in [3], [27]).

Composites in SPL satisfy the design goals of consistent syntax, flexible and even higher-order invocations, and hygienic scoping behavior. Since composite operators make sub-graphs reusable, they can reduce the amount of code compared to repeating similar sub-graphs over and over. In one case study, a Telco benchmark required 7,029 lines of SPL code to write [33]. After expanding all composite operators, this yielded 316,725 lines of SPL code, an expansion factor of 45. This application used nested chains of up to six composite operator invocations and contained 33 higher-order composites.

Optimizing via types and operator models
An optimization is *safe* if it does not change the behavior of the program. The program may run faster after the optimization, but it must not exhibit any observable behavior that could not be observed on the unoptimized version as well. It is up to the compiler and runtime system to establish that optimizations are safe. This can be challenging in the presence of user-defined code. Therefore, prior work on

streaming systems either minimizes user-defined code or compromises on safety. Relational streaming systems such as CQL [16] minimize user-defined code by relying mostly on a small set of built-in relational operators. The semantics of these operators is well understood and can be safely exploited for optimization. On the other hand, recent streaming systems such as S4 [4] or Storm [29] maximize user-defined code by compromising on safety. They optimize based on assumptions about user-defined code that they cannot establish automatically and may thus yield surprising behavior. The SPL compiler performs several optimizations in the presence of user-defined code. To establish safety, the SPL compiler relies on data types and operator models.

There are four design goals for types and operator models in SPL. The first goal is static error checking. Diagnosing errors at runtime is difficult in any parallel language, streaming or not. Since streaming applications run forever (at least in theory), and since SPL applications run in parallel on multi-cores and distributed on clusters, diagnosing errors at runtime can be a challenge. Hence, the SPL compiler strives to diagnose errors as early as possible, so the programmer can fix them before runtime. The second goal is to enable optimizations based on information gleaned by the compiler as it processes the application code. The type system and operator models help the SPL compiler use memory more sparingly, keep the cost of serializing data for network communication low, avoid copying data unnecessarily, limit runtime type checks, and even take advantage of more parallelism. The third goal is conciseness: specifying types should not create more boiler-plate code than necessary. (Here, the term *boiler-plate code* refers to code that needs to be written to satisfy the compiler and avoid compiler errors, without contributing much value of its own.) The fourth goal is support for native code. Native code is non-SPL code used from an SPL application, either via primitive operators or via native functions, i.e., functions written in C++ that can be called from SPL. The author of a primitive operator or native function must provide sufficient information to the SPL compiler to enable it to be safely optimized in the context of an SPL application.

SPL type system and optimization
SPL has a strong static type system. The type system is *strong*, because it minimizes *implicit conversions*, where a value of one type is treated as a value of another type without an explicit cast. In addition, the type system is *static*, because all types are known at compile time. SPL supports three collection types: **list**, **set**, and **map**. All three collection types are generic, meaning that, for instance, a list of integers is a different type than a list of strings. SPL users can write generic native functions that work on both, for instance, to perform sorting. All composite types (collections or tuples) in SPL support full nesting. However, SPL intentionally does not provide any pointer types. Instead, all types have copy

semantics. Copy semantics make sense for a distributed streaming language, since data traveling on streams between servers must be copied. Avoiding pointers minimizes aliasing, where two names refer to the same entity. Minimizing aliasing helps optimization, because the compiler can establish more easily whether pieces of code interfere with one another.

In languages such as C, C++, or Java, the programmer can use a keyword to declare a variable constant. That means that the default is mutable (non-constant) variables in these languages, which are harder for the compiler to optimize. SPL flips this default around: variables in SPL are immutable by default unless explicitly declared mutable. Since most variables are, in fact, immutable, this leads to more concise code that is easier to optimize. Of course, the SPL compiler will generate an error message when the code attempts to modify an immutable variable. Aside from variable mutability, SPL also allows functions to be declared as stateful or stateless. The result of a stateless function depends only on its parameters, making it easier to optimize. SPL functions are stateless by default unless explicitly declared stateful. The compiler checks whether the declaration is correct by analyzing whether the function calls other stateful functions.

SPL supports a large number of primitive types. For example, integers include four signed types **int8**, **int16**, **int32**, **int64** and four unsigned types **uint8**, **uint16**, **uint32**, **uint64** at different bit-widths. In addition, there are Booleans, binary and decimal floating-point types at different bit-widths, complex numbers, enumerations, timestamps, 8-bit and 16-bit strings, and blobs (binary large objects). The large number of primitive types gives the programmer more control over data representation. This is good for performance, since the size of data items affects the costs of serialization, cache footprint, and network transport.

SPL operator models and optimization

An operator model is an XML document written by the operator developer and used by the SPL compiler [31]. Figure 2 illustrates how operator models are used during compilation. The SPL compiler uses the operator model for two purposes: first, to check for errors in the operator invocation, and second, to make optimization decisions. The core portion of the operator model declares the *signature* of the operator: the number of input and output ports, names of parameters, restrictions on what kind of arguments the parameters expect, and operator-specific intrinsic functions such as the Sum function for the Aggregate operator. If the operator invocation violates any of these restrictions, the SPL compiler generates an error message.

The operator model includes information on *port mutability*. Specifically, an operator can declare that it does not modify data items arriving on an input port or that it tolerates a downstream operator modifying data items

departing from an output port. The SPL compiler uses this information to minimize the number of copies necessary for correct behavior. The operator model can declare certain input ports as *control ports*. A control port is usually used in a feedback loop and does not trigger further data items. The SPL compiler uses knowledge about control ports to decide when it is safe to *fuse* cyclic sub-graphs without causing deadlocks. The fusion optimization reduces communication costs by combining multiple operators in the same operating-system process or even in the same thread [34]. In the latter case, fusion is a tradeoff between communication cost and pipeline parallelism, i.e., parallel execution of a producer and a consumer operator on different threads. The operator model can specify that an operator provides a *single-threaded context*. In this case, the compiler can optimize by inserting less synchronization.

Finally, in a research (i.e., non-product) branch of InfoSphere Streams, we have prototyped the operator *fission* optimization. Fission replicates an operator instance multiple times to introduce additional parallelism [35]. To be safe, the replicated operator must produce the same data items in the same order as before fission. Our implementation uses sequence numbers to keep track of the order. If the operator is *prolific* (can generate multiple output data items per input data item), this leads to duplicate sequence numbers that make ordering more difficult. Hence, fission uses selectivity information (information about whether the operator is prolific or selective) from the operator model to establish safety. Furthermore, in a distributed system, sharing state is detrimental to performance. Hence, fission also uses state information from the operator model to establish safety. An operator is easy to parallelize if it is either stateless or if each replica can work on a disjoint portion of the state.

One challenge with optimizations based on the operator model is that the compiler must "trust" that the information in the operator model is correct. To this end, we keep the operator model simple. Many programmers can just assemble applications from existing operators, without ever needing to write their own, making use of libraries written by IBM and other vendors [36]. The need to write new operators is further reduced by the **logic** clause and Custom operators mentioned earlier. Only sophisticated programmers write their own operators, with their own operator models. Another challenge with advanced optimizations is that even if they do not change the external behavior of the application, they may change its internal structure, such as replicating operators for fission. This becomes problematic if that internal structure is visualized for debugging purposes [37]. However, this difficulty is hardly unique to SPL or even to streaming languages. Much of the academic literature on debugging non-streaming languages is concerned with debugging optimized code. Our approach is to maintain meta-information alongside the compiled application that a debugger or visualization tool can use to keep track of the

mapping between what the user wrote and what the compiler produced.

Overall, the type system and operator models in SPL help both with static error checking and with enabling optimizations. Furthermore, using immutability and statelessness as the default assumption in the type system improves conciseness. Asking the operator developer to specify an operator model allows the SPL compiler to make safe assumptions about semantic properties of non-SPL native code. SPL on InfoSphere Streams achieves both high throughput (millions of data items per second) and low latency (on the order of microseconds) [38] as a result of a careful implementation of the underlying distributed streaming platform, as well as optimizations in the SPL compiler.

Conclusion

This paper provided an overview of IBM's SPL language and its key features that make it suitable for Big Data analytics. It describes SPL both from the user's perspective, with detailed code examples, and from the implementer's perspective, with substantial contributions relative to prior work in streaming languages and systems. This paper focuses on three primary contributions: code generation for primitive operators, modularity via higher-order composite operators, and innovations that make it easier for the compiler to safely optimize user-defined code. By providing flexible support for primitive and composite operators, SPL allows users to write a broad range of expressive analytics. By supporting efficient code with code generation and optimizations, SPL is suitable for handling big data at massive scales. SPL is the primary means for programming InfoSphere Streams [2] applications. While InfoSphere Streams is a product with a growing user community, SPL is also the subject of ongoing research on languages and optimizations for big data in motion.

References

1. M. Hirzel, H. Andrade, B. Gedik, V. Kumar, G. Losa, M. Mendell, H. Nasgaard, R. Soulé, and K.-L. Wu, "SPL stream processing language specification," IBM Research, Yorktown Heights, NY, USA, Tech. Rep. RC24 897, 2009.
2. IBM Corporation, *InfoSphere Streams*. [Online]. Available: http://www.ibm.com/software/data/infosphere/streams/
3. N. Seyfer, R. Tibbetts, and N. Mishkin, "Capture fields: Modularity in a stream-relational event processing language," in *Proc. Conf. DEBS*, 2011, pp. 15–22.
4. L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, "S4: Distributed stream computing platform," in *2010 IEEE Int. Conf. Data Mining Workshops (ICDMW)*, 2010, pp. 170–177.
5. J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *Proc. Operat. Syst. Des. Impl. (OSDI)*, 2004, p. 10.
6. M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "DRYAD: Distributed data-parallel programs from sequential building blocks," in *Proc. Eur. Conf. Comput. Syst. (EuroSys)*, 2007, pp. 59–72.
7. C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, "Pig Latin: A not-so-foreign language for data processing," in *Proc. Int. Conf. Manage. Data (SIGMOD)*, 2008, pp. 1099–1110.
8. K. S. Beyer, V. Ercegovac, R. Gemulla, A. Balmin, M. Eltabakh, C.-C. Kanne, F. Ozcan, and E. J. Shekita, "Jaql: a scripting language for large scale semistructured data analysis," in *Proc. Conf. Very Large Data Bases (VLDB)*, 2011, pp. 1272–1283.
9. T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears, "MapReduce online," in *Proc. 7th USENIX Conf. Netw. Syst. Des. Impl. (NSDI'10)*, 2010, p. 21.
10. Y. Ahmad and C. Koch, "DBToaster: A SQL compiler for high-performance delta processing in main-memory databases," *Proc. Very Large Data Bases (VLDB-Demo)*, vol. 2, no. 2, pp. 1566–1569, Aug. 2009.
11. R. Soulé, M. Hirzel, R. Grimm, B. Gedik, H. Andrade, V. Kumar, and K.-L. Wu, "A universal calculus for stream processing languages," in *Proc. Eur. Symp. Progr. (ESOP)*, 2010, pp. 507–528.
12. N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous data flow programming language LUSTRE," *Proc. IEEE*, vol. 79, no. 9, pp. 1305–1320, Sep. 1991.
13. G. Berry and G. Gonthier, "The ESTEREL synchronous programming language: Design, semantics, implementation," *Sci. Comput. Programm.*, vol. 19, no. 2, pp. 87–152, Nov. 1992.
14. M. I. Gordon, W. Thies, and S. Amarasinghe, "Exploiting coarse-grained task, data, and pipeline parallelism in stream programs," in *Proc. Arch. Supp. Programm. Lang. Op. Syst. (ASPLOS)*, 2006, pp. 151–162.
15. S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. A. Shah, "TelegraphCQ: Continuous dataflow processing for an uncertain world," in *Proc. Conf. Innov. Data Syst. Res. (CIDR)*, 2003, p. 668.
16. A. Arasu, S. Babu, and J. Widom, "The CQL continuous query language: Semantic foundations and query execution," *VLDB J.*, vol. 15, no. 2, pp. 121–142, Jun. 2006.
17. D. J. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik, "Aurora: A new model and architecture for data stream management," *VLDB J.*, vol. 12, no. 2, pp. 120–139, Aug. 2003.
18. D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. S. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik, "The design of the Borealis stream processing engine," in *Proc. Conf. Innov. Data Syst. Res. (CIDR)*, 2005, pp. 277–289.
19. R. S. Barga, J. Goldstein, M. Ali, and M. Hong, "Consistent streaming through time: A vision for event stream processing," in *Proc. Conf. Innov. Data Syst. Res. (CIDR)*, 2007, pp. 363–373.
20. B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, and M. Doo, "SPADE: The System S declarative stream processing engine," in *Proc. Int. Conf. Manage. Data (SIGMOD)*, 2008, pp. 1123–1134.
21. J. Chen, D. J. DeWitt, F. Tian, and Y. Wang, "NiagaraCQ: A scalable continuous query system for internet databases," in *Proc. Int. Conf. Manage. Data (SIGMOD)*, 2000, pp. 379–390.
22. J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman, "Efficient pattern matching over event streams," in *Proc. Int. Conf. Manage. Data (SIGMOD)*, 2008, pp. 147–160.
23. A. Demers, J. Gehrke, B. Panda, M. Riedewald, V. Sharma, and W. White, "Cayuga: A general purpose event monitoring system," in *Proc. Conf. Innov. Data Syst. Res. (CIDR)*, 2007, pp. 412–422.
24. IBM Corporation, *WebSphere Operational Decision Management*. [Online]. Available: http://www.ibm.com/software/decision-management/operational-decision-management/websphere-operational-decision-management/

25. Progress Software, *Progress Apama*. [Online]. Available: http://www.progress.com/en/apama/index.html
26. TIBCO, *TIBCO BusinessEvents*. [Online]. Available: http://www.tibco.com/multimedia/ds-businessevents_tcm8-796.pdf
27. O. Etzion and P. Niblett, *Event Processing in Action*. Greenwich, CT, USA: Manning Publ., 2010.
28. M. Hirzel, "Partition and compose: Parallel complex event processing," in *Proc. Conf. DEBS*, 2012, pp. 191–200.
29. N. Marz, *Storm: Distributed and fault-tolerant real-time computing*. [Online]. Available: http://storm-project.net/
30. M. Mendell, H. Nasgaard, E. Bouillet, M. Hirzel, and B. Gedik, "Extending a general-purpose streaming system for XML," in *Proc. Int. Conf. Extend. Database Technol. (EDBT)*, 2012, pp. 534–539.
31. B. Gedik and H. Andrade, "A model-based framework for building extensible, high performance stream processing middleware and programming language for IBM InfoSphere Streams," *Softw. Pract. Exp. (SP&E)*, vol. 42, no. 11, pp. 1363–1391, Nov. 2012.
32. E. Kohlbecker, D. P. Friedman, M. Felleisen, and B. Duba, "Hygienic macro expansion," in *Proc. LISP Funct. Programm. (LFP)*, 1986, pp. 151–161.
33. M. Hirzel and B. Gedik, "Streams that compose using macros that oblige," in *Proc. Workshop Partial Eval. Progr. Manipul. (PEPM)*, 2012, pp. 141–150.
34. R. Khandekar, K. Hildrum, S. Parekh, D. Rajan, J. Wolf, K.-L. Wu, H. Andrade, and B. Gedik, "COLA: Optimizing stream processing applications via graph partitioning," in *Proc. Int. Middleware Conf. (MIDDLEWARE)*, 2009, pp. 308–327.
35. S. Schneider, M. Hirzel, B. Gedik, and K.-L. Wu, "Auto-parallelizing stateful distributed streaming applications," in *Proc. Int. Conf. Parallel Arch. Compil. Techn. (PACT)*, 2012, pp. 53–64.
36. IBM Corporation, *Streams Exchange: A place for InfoSphere Streams application developers to share code and ideas with others*. [Online]. Available: https://www.ibm.com/developerworks/wikis/display/streams/Home
37. W. De Pauw, M. Letia, B. Gedik, H. Andrade, A. Frenkiel, M. Pfeifer, and D. Sow, "Visual debugging for stream processing applications," in *Proc. Int. Conf. Runtime Verific.*, 2010, pp. 18–35.
38. Y. Park, R. King, S. Nathan, W. Most, and H. Andrade, "Evaluation of a high-volume, low-latency market data processing system implemented with IBM middleware," *Softw. Pract. Exp.*, vol. 42, no. 1, pp. 37–56, Jan. 2012.

Martin Hirzel *IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY 10598 USA (hirzel@us.ibm.com)*. Dr. Hirzel has worked at IBM since 2004, conducting programming languages research. In the SPL team, his focus is on the language design and the compiler front-end.

Henrique Andrade *Goldman Sachs, New York, NY 10282 USA (hcma@hcma.info)*. Dr. Andrade worked at IBM from 2004 to 2010. While at IBM, he co-invented SPADE, the precursor to SPL, and helped launch the SPL project. For a period of time, he was also the runtime architect for InfoSphere Streams.

Buğra Gedik *Department of Computer Engineering, Bilkent University, Bilkent, Ankara 06800, Turkey (bgedik@cs.bilkent.edu.tr)*. Dr. Gedik worked at IBM from 2006 to 2012. While at IBM, he co-invented SPADE and helped launch the SPL project. For a period of time, he also served as the chief architect for InfoSphere Streams.

Gabriela Jacques-Silva *IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY 10598 USA (g.jacques@us.ibm.com)*. Dr. Jacques-Silva joined IBM in 2010. In the SPL team, her focus is on fault tolerance.

Rohit Khandekar *Knight Capital Group, Jersey City, NJ 07310 USA (rkhandekar@gmail.com)*. Dr. Khandekar worked at IBM from 2008 to 2012. In the SPL team, his contributions included the fusion optimizer.

Vibhore Kumar *IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY 10598 USA (vibhorek@us.ibm.com)*. Dr. Kumar joined IBM in 2008. In the SPL team, he is working on distributed shared variables.

Mark Mendell *IBM Canada, Markham, ON, L6G 1C7 Canada (mendell@ca.ibm.com)*. Mr. Mendell has been at IBM since 1991. He worked on C++ compilers and was team lead for the TOBEY optimizer for several years. In the SPL team, his focus is on the compiler back-end and libraries.

Howard Nasgaard *IBM Canada, Markham, ON, L6G 1C7 Canada (nasgaard@ca.ibm.com)*. Mr. Nasgaard has been at IBM since 1977 and has worked on compilers for 17 years. In the SPL team, his focus is on the compiler front-end.

Scott Schneider *IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY 10598 USA (scott.a.s@us.ibm.com)*. Dr. Schneider has been at IBM since 2011. In the SPL team, his focus is on optimizations and, in particular, on auto-parallelization.

Robert Soulé *Cornell University, Department of Computer Science, Ithaca, NY 14850 USA (soule@cs.cornell.edu)*. Dr. Soulé worked at IBM as an intern and a co-op student while he was a student at New York University. In the SPL team, he focused on translating SPADE to SPL and on formalizing the programming model as a calculus.

Kun-Lung Wu *IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY 10598 USA (klwu@us.ibm.com)*. Dr. Wu has worked at IBM for more than 20 years. He manages the SPL team, both on the research side and in product development. Dr. Wu was part of the SPL team from back when it was still called SPADE.