A Scalable Platform for Low-Latency Real-Time Analytics of Streaming Data

Paolo Cappellari[1(✉)], Mark Roantree[2], and Soon Ae Chun[1]

[1] City University of New York, New York, USA
{paolo.cappellari,soon.chun}@csi.cuny.edu
[2] School of Computing, Insight Centre for Data Analytics,
Dublin City University, Dublin, Ireland
mark.roantree@cs.dcu.ie

Abstract. The ability to process high-volume high-speed streaming data from different data sources is critical for modern organizations to gain insights for business decisions. In this research, we present the streaming analytics platform (SDAP), which provides a set of operators to specify the process of stream data transformations and analytics. SDAP adopts a declarative approach to model and design, delivering analytics capabilities through the combination of a set of primitive operators in a simple manner. The model includes a topology to design streaming analytics specifications using a set of atomic data manipulation operators. Our evaluation demonstrates that SDAP is capable of maintaining low-latency while scaling to a cloud of distributed computing nodes, and providing easier process design and execution of streaming analytics.

Keywords: Data stream processing · High-performance computing · Low-latency · Distributed systems

1 Introduction

In their quest for competitive advantage, extending data analysis to include streaming data sources has become a requirement for the majority of organizations. Driven by the need for more timely results and having to deal with an increasing availability of real-time data sources, companies are investing in integrating data streaming processing systems in their applications stack. Real-time data processing can help multiple application domains, such as stock trading, new product monitoring, fraud detection and regulatory compliance monitoring, supporting situation awareness and decision making with real-time alerts and real-time analytics. Real-time data processing and analytics requires flexible integration of live data captured from different sources that would otherwise be lost, with traditional data from enterprise storage repositories (e. g. data warehouse).

Existing streaming systems mainly focus on the problems of scalability, fault-tolerance, flexibility, and performance of individual operations, e.g.

© Springer International Publishing AG 2017
C. Francalanci and M. Helfert (Eds.): DATA 2016, CCIS 737, pp. 1–24, 2017.
DOI: 10.1007/978-3-319-62911-7_1

[1,2,26,29]. Our approach to building a high-performance data stream processing engine was influenced by the growing need of more timely information by organizations and the success of streaming systems such as Yahoo S4 [22], Storm [27], Sonora [6], and Spark-Streaming [29]. We observed that none of the modern systems target low-latency and high-performance while also providing an easy way of developing streaming applications for non-expert users. Unlike these systems and other related research, we focus on the provision of a complete and comprehensive solution for the rapid development, execution and management of scalable high-performance, low-latency, stream analytics applications.

1.1 Motivation and Case Study

To illustrate the complex tasks involved in a stream analytics process, we use a scenario which seeks to understand the performance of the bike utilization in multiple locations within a city, trying to monitor the trend of the performance data and comparing usage with bike usage in other cities. Assume a scenario where a town planner needs to know various performance indicators, such as whether bicycles are parked in specific docking stations located across the city are utilized to an acceptable level. This requires the constant monitoring of data from the Bike Sharing Systems (BSS). Every 60 s, the BSS reports the status of each station, which include the number of bikes docked at each station. The goal is to calculate the performance of the BSS as the number of bikes in utilization against the total number of bikes available in the system, in order to identify stations with lower than predicted usage or stations with high usage that require expansion. In addition, the manager is also interested in the performance of bike sharing program in other cities, to gain a direct comparison among different BSS.

1.2 Contribution

In this paper, we improve on our previous work [3] and we present the streaming data analytics platform (SDAP), which provides a set of operators to specify the process of stream data transformations and analytics, together with its execution environment. SDAP adopts a declarative approach to both modeling and designing a streaming analytics system using combinations of primitive operators in a straightforward manner.

SDAP is aimed to be a robust platform for flexible design of streaming analytics applications that addresses the following broad requirements:

- The processing engine can manage high volumes of streaming data even when the rate at which data generated is extremely high;
- Results of steaming analytics and processing, on which organizations base decisions, are available as soon as possible;
- It supports designers of analytical processes by abstracting from the underlying parallel computation or high-performance programming;
- It is easy to develop, maintain and optimize the analytical applications.

The contribution of this research can be summarized as follows:

- Streaming analytics model. SDAP provides a set of operators to support a declaration-based analytics development environment.
- Streaming analytics application specification. SDAP provides users without prior knowledge of parallel computation or high-performance. programming, the tools to easily specify a ‘topology’, which describes the analytics process.
- High-performance topology execution. SDAP delivers a platform that exploits the best performing hardware and software to execute a topology, while also efficiently managing the resource computation underlying data stores and parallel processing.

A comprehensive evaluation demonstrates the performance of our system, both in terms of latency and ease of development. SDAP presents the lowest latency among the compared systems with the same low latency maintained when scaling to a large number of computational resources. Compared to similar systems, SDAP is different because it provides each of the following characteristics: (i) it offers built-in operators optimized for parallel computation; (ii) it was designed to deliver the best latency performance by exploiting the high-performance hardware and software libraries; (iii) it is easy to use, since users are not required to have programming skills; and (iv) it enables rapid development, since applications are specified in a declarative way, where users link built-in operations in a pipeline fashion.

The paper is organized as follows. In Sect. 2, we provide a comparison of our approach against other works. Section 3 we demonstrate a use case realized by using SDAP, which is used as a running example throughout the paper. In Sect. 4, we define the modeling of our platform, including the constructs and the primitives. Section 5, discusses the platform’s architecture. In Sect. 6, we discuss the experimental setting, the performance results, and the ease of usage compared to a popular alternative. Finally, in Sect. 7 we present our conclusions.

2 Related Research

Research projects such as S4 [22], IBM InfoSphere Streams [15], and Storm [27], are considered event-based streaming systems as they process each tuple as soon as they become available. While this is a requirement for low-latency system, these research projects do not address latency or high-performance directly. The S4 system, for instance, provides a programming model similar to Map-Reduce, where data is routed from one operation to the next on the basis of key values. In comparison to SDAP, their approach limits the ability of the designer in the development of generic streaming applications. Storm [27] offers a set of primitives to develop topologies. In brief, a set of constructs are provided to route data between operations, similar to our approach in SDAP. However, the developer must provide the implementation at each step in the topology and thus, requiring development effort and expertise in parallel programming. This is not the case in SDAP, where built-in operations are provided, so that designers

can focus on the creation of the topology rather than on the *implementation* of the operators. IBM InfoSphere Streams [15] follows an approach similar to Storm but also offers a set of predefined operations. In fact, designers can assemble operations in workflows, very much like in SDAP. However, the InfoSphere Streams approach puts the focus on quality of service of the topologies, rather than on latency performance.

A system that specifically targets low latency stream processing is Google's MillWheel [1]. As with the systems described above, MillWheel adopts an event based design and in this case, data manipulation operations are specified in a topology fashion. Similarly to S4, the computation paradigm is based on a key model: data is routed between computation resources on the basis of the value a key holds in the data. As it is the case for S4, this paradigm facilitates the evaluation of operations requiring grouping (on the same key), and only guarantee pure distributed parallelism using different keys. SDAP offers greater flexibility in this respect as the SDAP designer can choose whether or not to base the computation on keys. Moreover, as shown in our evaluation, SDAP delivers superior performance.

In an entirely different approach, the research ideas presented in [7,11,23,28,29], approach data stream processing by embracing a micro-batch oriented design. These approaches extend the Map-Reduce paradigm and Hadoop systems. Limmat [11] and Google Percolator [23] extend Hadoop by introducing a push-based processing, where data can be pushed into the process and results are computed computed incrementally on top of the current process state, e.g. aggregates for current windows. The main downside of these approaches is latency, which can run into minutes.

The Spark-Streaming [29] and Hadoop Online Prototype (HOP) [7] projects are an attempt to improve the Hadoop process by making it leaner and as a result, faster. When possible, data manipulation is performed directly in main memory without using secondary storage, which makes computation faster. Although these approaches improve performance for real-time analytics support, the micro-batch design creates an intrinsic limit that prevents these types of systems from achieving the same low latency as event-based systems.

Also included in the class of micro-batch systems, although not Map-Reduce oriented, is the Trident [28] system, an extension to Storm that provides higher level operators and other features. Trident suffers from the latency limitation mentioned previously for the micro-batch systems, a problem which we do not have in SDAP. Although both Spark-Streaming and Trident offer a set of predefined operators, developing a topology still requires the development of a program in Java or Scala, which unlike SDAP is a more challenging task because: users have to know the language; and users must validate their code before validating the application itself. SDAP enable designers, not developers, to rapidly develop topologies neglecting all details related to software code development and thus, focusing on the business logic. In addition, SDAP supports complex window definitions which are not available in any of these systems.

There has been much research on developing the performance of individual operators, e.g. [4,5,9,16–18,20,26]. In [12,13], the authors tackle the problem of processing XML data streams. They developed a multidimensional metamodel for constructing XML cubes to perform both direct recursion and indirect recursion analytics. While this approach has similar goals and approach, the SDAP system is designed to scale, adopts an easier to use scripting approach, and can facilitate JSON sources unlike their approach which only uses XML. In fact, none of these research efforts offer a comprehensive solution to maximize performance across all aspects of the streaming network. SDAP, on the other hand, provides a general solution for rapid development of stream analytics for high performance environments.

3 Streaming Analytics Case Study

Figure 1 illustrates a Bike Sharing stream analytics process design using the scenario described in Sect. 1. The data is streamed from bike sharing systems (BSS) in real-time from the cities of New York and Dublin. Here, the rounded rectangles represent the data manipulation steps. Arrows between steps describe how the stream flows from one transformation to the next. The operation applied by each step is depicted with a symbol (see the legend) within the rectangle, along with its degree of parallelism (within parenthesis). The specific operation performed by the operator is detailed with bold text just below each step. On top of each step, an italic text provides a brief explanation of the operation applied in the node. Note that the Selection operator outputs two streams: the solid edge denotes the stream of data satisfying the condition; the dashed edges

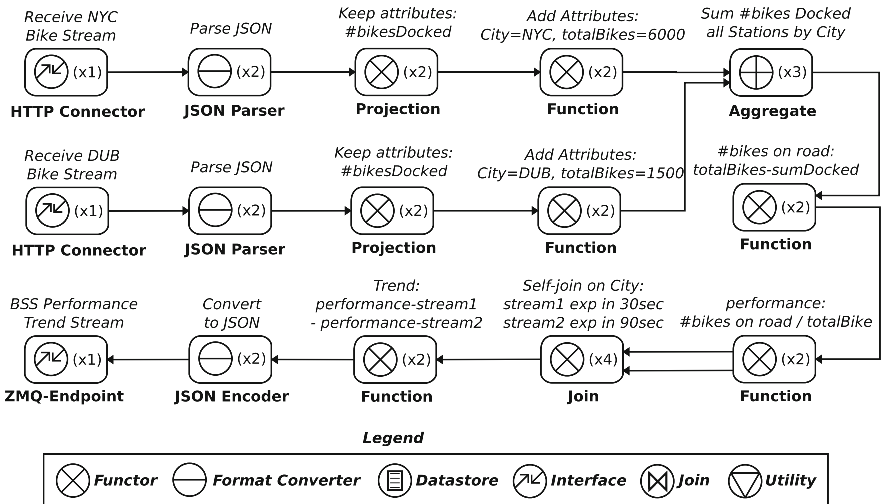

Fig. 1. Topology for bike sharing system case study in SDAP.

are the stream of data *not* satisfying the condition. Where one output stream from Selection is not used, the edge is not shown in the illustration.

The application in Fig. 1 describes an analysis of the bikes stream by generating the performance trend of the input BSS systems. Data flows into the application by the `HTTP Connector` step that connects to the BSS stream and delivers a snapshot of the status of all bike stations in each BSS system. Data is collected on a per minute basis. Station status data is provided in JSON format and thus, it is passed to the `JSON Parser` operator that convert data from JSON into the mapped tuple format. The next step removes attributes unnecessary for the required analysis at hand. Finally, two constants are added to the stream: the city the BSS data is from, and the number of bikes the BSS system has available. Streams from different cities are merged. As part of this process, there are three calculations.

- Available: the number of bikes currently docked across the city, as an aggregation of all docked bikes in each station in the incoming data over the interval of one minute (the data refresh rate);
- InUse: the number of bikes on the road, as the difference between the total bikes available and those docked;
- System Performance: defined as the ratio between the number of bikes on road and the total number of bikes (the more bikes on the road, the better the performance for this topology).

The remaining steps calculate the *trend* of the performance of each system. Performance trend is defined as the difference between two consecutive performances. In order to have two performances values in the same observation, a self-join (same city) is made on the performance stream, where one stream has an expiration time of 30 s (only fresh data being considered), while the second stream has an expiration time of 90 s. This way, a new performance value for a city is coupled with the previous performance value from the same city.

Once the trend is available, the result is converted into JSON format and produced in the output of the application, available to other applications, via a ZeroMQ end-point.

4 Conceptual Model for Streaming Analytics

In general, streaming applications consist of a sequence of data manipulation operations, where each operation performs a basic transformation to a data element, passing the result to the next operator in the sequence. When multiple transformations are chained together in a pipeline fashion, they create sophisticated, complex transformations. A *complex transformation* is a workflow, where *multiple* pipelines are combined. These workflows can be represented as direct acyclic graphs (DAGs) and are also referred to as topologies [4]. The SDAP model enables the construction of complex topologies using the set of constructs described below.

Tuple. A tuple is used to model any data element in a stream. It is composed of a list of values describing the occurrence of an event. For instance, in the BSS stream described later in Sect. 3, each update reports on the status of each station, where each station has an *identifier*, *address*, *status* (operative or not), the *number* of bikes docked, and *geo-location*.

Stream. A stream is a sequence of events described by tuples. Tuples in a stream conform to a (known) schema: each tuple value in the same stream are instances of a known set of attributes, each having a specific data type. For instance, tuples generated from bike stations status update on bike sharing system all have the same structure, with potentially different values, as people take and park bikes during the day.

Operator. An operator is a data processing step that processes each tuple received from one (or more) input stream(s) by applying a transformation to the tuple's data to generate a new tuple in the output stream. The operators are described in the following section but for now, we discuss two important parameters that are associated with each operator: *parallelism* and *protocol*.

Parallelism. In a topology, each operator decides its degree of parallelism. Parallelism controls the number of instances of an operator that collaborate to complete a process. In order to process large amounts of data, processing must be distributed across multiple computational resources (cores, CPUs, machines).

Protocol. The protocol defines how tuples are passed between the instances of contiguous operators in a topology. For example, a tuple can be passed to just one instance or to all instances of the next operator in the topology. SDAP supports four routing modes for protocol: *round-robin*, *direct*, *hash* and *broadcast*. In **round-robin** mode, tuples from an upstream node's output port are distributed to all instances, in an even fashion across all the downstream resources. **Direct** mode defines a direct and exclusive connection between *one* instance of the upstream node and *one* instance of the downstream node. This routing strategy is effective when pipelined operators require the same degree of parallelism. The **hash** mode routes tuples on the basis of a (key) value within the tuple itself. This permits an application to collect data having the same key in the same resource. Where this leads to uneven usage of downstream resources, the **broadcast** routing strategy, ensures that every tuple from a single instance of an upstream node, is copied to *all* instances of the downstream node.

Topology. A topology describes a stream analytics workflow, i.e. how the data stream flows from the input source(s) through the combination of primitive operators and sub-topologies to the output. It is modeled as a DAG, where nodes represent operators, and edges describes how tuples move between operators.

4.1 Primitive Operators in SDAP

This section presents a sample of the more important operators in SDAP, which are powerful enough to enable designers to construct very complex transformations. The rationale for providing a set of built-in operators is: (i) application designers focus on the transformation workflow and not implementation details; (ii) semantics are guaranteed and consistent across the entire system; (iii) every operator delivers the best possible performance; and (iv) the system can be extended with new operators as required. SDAP currently offers the following operators: Functor, Aggregate, Join, Sort, Interface, Format Converter, Datastore, Control and Utility.

The **Functor** operator applies a transformation that is confined and local to the tuple currently being processed. Many transformations can be thought as specializations of the Functor operator. SDAP provides *Projection* and *Selection*; *Function* which provides adding constants or a sequence attribute to the stream; text-to/from-date conversion; math (addition, division, modulo, etc.) and string functions.

The **Aggregate** operator groups tuples from the input stream, with an implementation of SQL-like aggregations: average, sum, max, min and count. The operator requires a *window* definition that specifies when and for how long tuples be included in the aggregation.

The **Join** operator is similar to the relational join but requires the definition of a window specifying the tuples from each stream to include in the join evaluation.

The **Sort** operator sorts the tuples within a "chunk" of the input stream in lexicographical order on the specified set of attributes. The number of tuples that comprise the chunk, is specified in a window definition.

The **Datastore** operator enables the stream to interact with a repository to retrieve, lookup, store and update data. The repository can be a database, a text file or an in-memory cache.

The **Interface** operator enables SDAP to create streams of data from external data sources to generate into topologies and to create end-points where processed data can be accessed by consumer applications. Consumer applications can be external or within SDAP (e.g. other topologies). Currently, SDAP can process streams from Twitter, Salesforce, ZeroMQ and generic HTTP end-points.

The **FormatConverter** provides data format conversion between the tuple and other formats when processing data within a topology.

SDAP is extensible and new operators can be added as necessary. Currently, SDAP also includes the following additional operators: **Look-up**, to look-up values from either databases or files; **Geotagging**, to convert name of locations into geo-coordinates; **Delay**, to hold or slow down the elaboration of each tuple by some interval of time; **Heartbeat**, to signal all operators in a topology; **Cache**, to provide a fast memory space where to temporarily hold and share data across the whole topology; and **Tokenizer**, to transforms a text into multiple word tokens.

Some operators, e.g. join, cannot operate on an infinite stream of data: they require the definition of a *window* that cut the stream in "chunks" of data. Windows [4,5,18] are usually defined by specifying a set of constraints on attributes such as time, number of observed events (i.e. received tuples), or values in the stream(s) [16,20]. SDAP supports all of the above and, in addition, allows to define windows on sophisticated constraints involving conditions on both the input and output streams. Section 4.3 presents additional information and an example of a window definition.

4.2 Topology Model

A stream analytics process, called Topology, is modeled as a DAG, a directed acyclic graph, which is defined as in Definition 1:

Definition 1 (Topology). *A topology* $T = \langle N, R, \Sigma_O, \Sigma_R, \Sigma_P, o, r, p\rangle$ *is a five element tuple where* $N = \{n_1, n_2, \ldots\}$ *is a set of Nodes, and* $R = \{r_1, r_2, \ldots\}$ *is a set of Routes,* Σ_O *is the set of operators,* Σ_R *the set of data distribution protocols,* Σ_P *the degree of parallelism, and* o, r *and* p *functions that associates:* $o : N \rightarrow \Sigma_O$ *a node with an operator (and configuration),* $r : R \rightarrow \Sigma_R$ *an edge with a protocol, and* $p : N \rightarrow \Sigma_P$ *a node with a degree of parallelism.*

Each node in the topology specification follows the expression syntax outlined in Definition 2 in [3].

Definition 2 (Node)

```
operator
    <node-label>
    <operator-executable-path>
    <node-configuration-path>
```

Where: `operator` declares a node in the DAG; `node-label` specifies the label for such node, in order to refer to it in other places in the topology definition; `operator-executable` associates the executable with the node; finally, `node-configuration` specifies the arguments to pass to the executable and that configure the behaviour of the operator, e.g. conditions for a filtering criteria.

With reference to Fig. 1, Listing 1.1 shows an excerpt of the topology specification to illustrate how nodes in a DAG are declared in SDAP. The excerpt focuses on the top right part of Fig. 1, specifically on nodes: `Keep Attributes: #bikesDocked`, `Add Attributes: CityN̄YC, total Bikes6̄000` and `Sum #bikes Docked all Station by City`. For convenience, above nodes are renamed to `keep_attributes`, `add_constant`, `sum_docked`, respectively.

Details on how to specify an operator's configuration are presented in the Sect. 4.3. In Listing 1.1, line 11 defines a node with label `sum_docked`, that is associated with operator Aggregation, whose configuration is in file

sum_docked_conf. It implements an aggregation operation, calculating a sum of all docked bikes available at each station, grouping data by city.

```
## Product stream, nodes
...
operator
    keep_attributes_dub
    functor-mpi
    ${keep_attributes_dub_conf}
operator
    add_constants_dub
    functor-mpi
    ${add_constants_dub_conf}
operator
    add_constants_nyc
    functor-mpi
    ${add_constants_nyc_conf}
operator
    sum_docked
    aggregate-mpi
    ${sum_docked_conf}
...
```

Listing 1.1. Specification of nodes in a topology: an example.

The flow of tuples from operator to operator in the topology is defined along with a routing protocol. Its specification follows Definition 3 from [3].

Definition 3 (Route)

```
route
    <upstream-node-label:port>
    <protocol>
    <downstream-node-label:port>
```

Where: route declares an edge in the topology; upstream-node-label is the label of a node acting as data provider (also called upstream node); analogously, downstream-node-label is the label of the other node participating in the connection, specifically the label of the node receiving data (also called downstream node); protocol specifies how to distribute tuples between the two nodes (e.g. *direct*, *round-robin*, *hash* or *broadcast*); port specifies which port each node will use to send/receive tuples.

The specification in Listing 1.2 shows the part of the topology in Fig. 1 that links the *add_attributes_dub* and the *sum_docked* steps (also in Listing 1.1). Specifically, all tuples from the add_attributes_dub node are passed to node sum_docked via port number 1. Note that *sum_docked* receives data from two upstream nodes, namely streams, one for the data from Dublin, the other for

the data form NYC. All edge declarations use the protocol `roundrobin` to exchange tuples between the instances of the involved nodes.

```
## Product stream, connections
...
route
   add_attributes_dub:1 roundrobin add_constants_dub:1
route
   add_constants_dub:1 roundrobin sum_docked:1
route
   add_constants_nyc:1 roundrobin sum_docked:1
...
```

Listing 1.2. Specification of routes between nodes in a topology: an example.

Lastly, we need to associate each node with a degree of parallelism. The syntax is the following, from [3]:

Definition 4 (Parallelism)

parallelism <node-label> <degree>

Where: `parallelism` declares the parallelism for a node; `node-label` indicates the node in question; and `degree` specifies the degree of parallelism, that is how many runtime process instances have to be instantiated for the node in question.

Listing 1.3 illustrate the final excerpt of the sample topology specification. From Listing 1.3, we can see that nodes `keep_attributes`, `add_constants`, and `sum_docked` have parallelism $2, 2, 3$, respectively. The rationale in choosing a degree of parallelism is based on the amount of data to process and on the cost of the operation. In this example, the first two operations are rather simple, whether the aggregation is actually performing a calculation, thus a higher degree of parallelism. Note that values in Listing 1.3 are for illustration purpose. Real-world deployments these values have, in general, much higher values.

```
## Product stream, distribution
...
parallelism keep_attributes 2
parallelism add_constants 2
parallelism sum_docked 3
...
```

Listing 1.3. Specification of node parallelism: an example.

```
"in": [
    {"name": "timestamp", "type": "double"}
    ,{"name": "docked", "type": "int"}
    ,{"name": "City", "type": "String"}
    ,{"name": "totalBikes", "type": "int"}
    ]
,"out": [
    {"name": "timestamp", "type": "double"}
    ,{"name": "City", "type": "String"}
    ,{"name": "totalBikes", "type": "int"}
    ,{"name": "sumDockedBikes", "type": "int"}
    ]
...
```

Listing 1.4. Schema of input and output streams of a node.

```
...
,"groupby": [
    {"attribute": "City", "attribute": "totalBikes"}
    ]
,"aggregate": [
    {"input_attribute_name": "dockedBikes"
    ,"operation": "sum"
    ,"output_attribute_name": "sumDockedBykes"}
    ]
...
```

Listing 1.5. Detail of the aggregation operator configuration.

4.3 Operator Configuration

In a topology, a node is associated with an executable implementing a specific operator, e.g. `Selection`. The details of the nature of the input and output streams, as well as how to filter incoming tuples is provided in the operator configuration specification file. This specification starts with detailing the schemas of the input and output streams, that is attribute names and types. Then, each operator has its own signature, thus a different set of configuration parameters. Since it is not possible to illustrate the configuration of all operators in SDAP, we focus on just one of them: the `Aggregate`. Listings 1.4 and 1.5 shows excerpt of configuration for the aggregate operator in our example in Fig. 1. Listing 1.4 shows the schema of tuples for the input and the output streams. Listing 1.5 shows the details of aggregation, in this case a *sum*. It can be seen that values from the input streams are grouped by `City` (and `totalBikes`); the aggregations are defined on attribute `dockedBikes`; the results are provided in output attribute `sumDockedBikes`. An attribute `timestamp` is also added to the output stream.

In contrast to the infinite nature of the data stream, the aggregation operator is required to work on a finite set of data. Finite sets of data are defined by

windows. SDAP supports arbitrarily complex windows, including those based on: wall-clock intervals, number of observed tuples, the value of a progressing attribute [19] in the stream (e.g. time), external events (e.g. control messages), and conditional, that is based on values in the stream.

SDAP allows for the following window types: *Interrupt*, *Attribute*, and *Tuple*. With *Interrupt*, the window is defined by an external message: basically, the operator finalize the calculations and release the results only when requested. This options suits operations that have to release data at regular interval of (wall-clock) times. Type *Tuple* models windows defined on the number of input observations, e.g. create windows of 50 consecutive observations each, with a new window starting every 20. Type *Attribute* models window based on a progressing attribute embedded in the stream. The progressing attribute has the characteristic of being monotone in value, that is, increase at a standard interval (e.g. time).

When a basic window is not sufficient, developers can define window borders by condition: the developer can express an arbitrary condition to define when to close or open a new window. This is useful, for instance, to define landmark [10] windows or, more generally, windows whose boundaries depend on values in the data stream. SDAP allows developers to define conditions on attributes from both the input and output streams. An example of when a conditional window may be needed is the following: provide aggregate results immediately when the aggregate value exceeds a specified threshold defined by a literal in a constraint or by another value embedded in the stream. SDAP does not wait for windows to close to evaluate results: new partial, temporary, results are evaluated as new data is received. Thus, temporary results are always current and can be forwarded as part of a subsequent output.

Listing 1.6 continues the presentation of the configuration for the aggregation operator by specifying its window. The listing defines a tumbling window [10] on the progressing attribute *timestamp*. In fact we can see that: the window is of type `Attribute`; that the attribute characterizing the window is the `timestamp`; that a window should close (`window_close`) every 60 s; that values should be forwarded to output (`window_emit`) at the same time the window is closed (i.e. 60 s); that a new window should be created (`window_advance`) 60 s ahead of the previous open one; and that tuples are considered part of the "current" window if they arrive up to half a second after the specified window close limit.

```
...
,"window_type":"Attribute"
,"progressing_attribute":"timestamp"
,"window_close": {"type":"literal","size": 60}
,"window_emit": {"type":"literal","size": 60}
,"window_advance": 60
,"window_delay": 0.5
...
```

Listing 1.6. Specification of a tumbling window configuration.

5 SDAP System Architecture

While SDAP runs on a wide range of computational resources, the architecture was designed and implemented as a high performance system. In Fig. 2, the SDAP architecture is illustrated as having seven major components: the Resource Manager, Clustering, Data Operators, Monitoring, System Interface, Application Specification (repository) and the Resource Configuration repositories. Components such as the Resource Manager, Computation, Clustering and Monitoring use Slurm [25], MVAPICH2 [21], and Ganglia [8], as they are established, high performance open source source libraries.

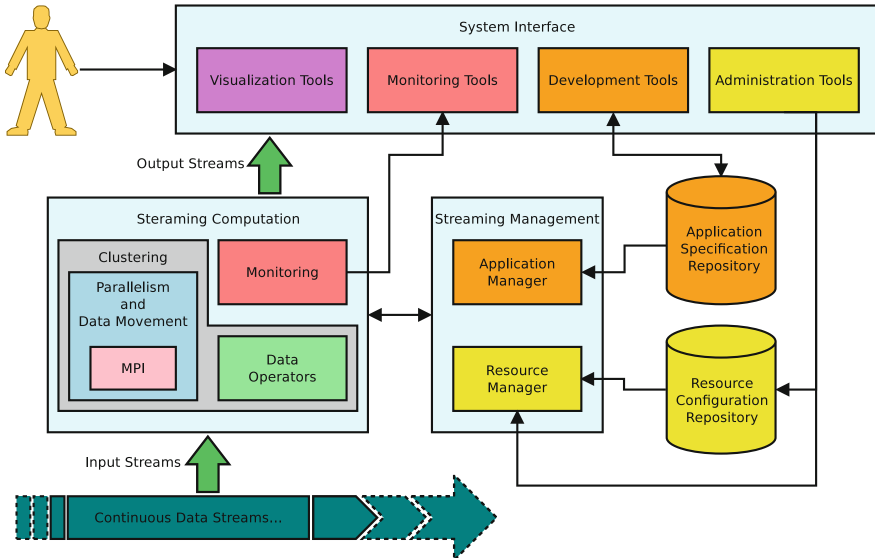

Fig. 2. SDAP architecture: logical view.

5.1 Data Operations

The Clustering component resides at the core of the SDAP architecture and comprises two sub-components: (i) parallelism and data movement and (ii) data operations. The first component manages parallel processes and the movement of data between processes. We adopt an implementation of the Message Passing Interface (MPI), specifically MVAPICH2, in order to optimize these high-performance environments. MPI is designed to achieve high performance, scalability, and portability and MVAPICH2 is one of the best performing implementation. This is mainly due to its support for the most recent and performing hardware, such as Infiniband [14], a high-performance inter-connector, designed

to be scalable and featuring high throughput and low latency. The Parallelism and Data Movement component builds on top of Phish [24], that in turns uses MPI. Phish is library to develop topologies composed of executables, providing an abstraction on parallelism and message delivery. The Data Operations component implements the data processing operation and enforces the operation protocols across the parallel processes. Again using Phish, SDAP has a Data Operators component which offers a set of built-in operations, including selection, projection, join, etc.

5.2 Distribution Management

Figure 3 illustrate the physical architecture of SDAP. The cluster is divided in *Compute* and *Control* nodes. Compute nodes provide computation and are mutually independent while the control node manages and coordinates compute nodes. Specifically, each compute node hosts the operators' executables, to perform data manipulation, and is responsible for forwarding data to the next operator in the topology. The operators' executable are deployed to every compute node so that each node can accommodate any operation specified in a topology. Each compute node also hosts the slave processes of the resource manager and of the resource monitoring. The Resource Manager process maintains the state of available resources and the plan for allocation to each topology (e.g. a single core allocated exclusively or not to an operator of a topology). When the resource manager slave receives a request to allocate or to release a resource, it first checks the state of the resource and then applies requests where possible. The Resource Monitoring process collects resource usage data, i.e. CPU time, memory allocation, etc., for the local node. This data is then forwarded to the master node, where data is aggregated and evaluated. The master node dispatches resource allocation requests and analyzes the resource usage of all compute nodes. Topologies are deployed or recalled using the Application Manager component that, in turns, uses the *master* process of the resource manager to allocate the nodes as per topology specification, when possible. The resource manager Master Process collects and analyzes resource usage data sent by all slave processes residing on compute nodes. Resource usage is provided at both the individual and collective level: a user can analyze details of CPU, memory, and network load for each individual node or for the cluster as a whole. Resource load can be analyzed for any specified time interval.

5.3 Resource Management

As multiple streaming topologies run on a Cluster and since resources are limited, there is a requirement for managing and monitoring resources. The Components Resource Manager and Resource Monitoring components deliver on these requirements. The Resource Manager is built on top of Slurm [25], a high-performance, scalable and fault-tolerant cluster resource manager. Slurm provides functionality to execute, manage and monitor distributed parallel applications and is used in many of the most powerful computers in the world. It

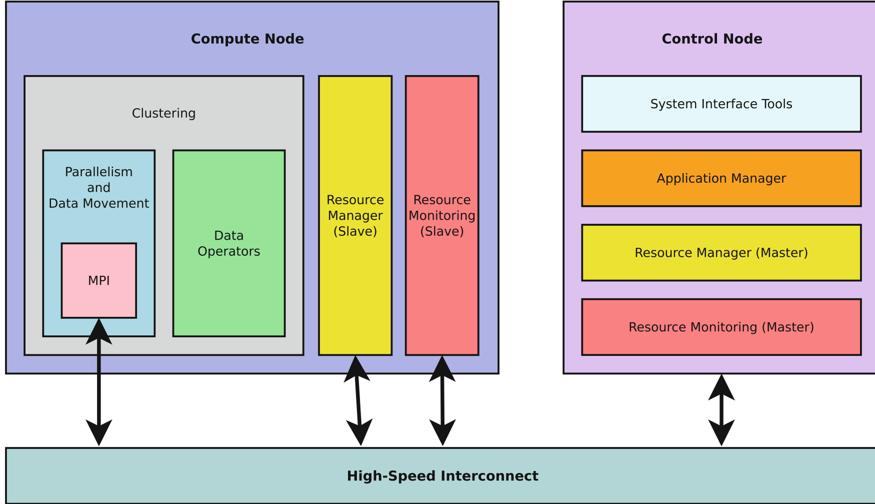

Fig. 3. SDAP architecture: physical view.

facilitates computational resource management and the allocation and deployment of streaming topologies. Among its features are topology relocation (to other resources) and a fault-tolerance mechanism. We have integrated Ganglia [8] as our resource monitoring system as it is highly scalable and works in high-performance environments.

Because the SDAP system was designed for the highest levels of scalability and performance, the resource monitoring and manager components can be deployed in a hierarchical manner, as illustrated in Fig. 4. Such a hierarchical organization of processes facilitates resource allocation requests and monitoring to be distributed over a larger number of processes and thus, avoiding bottlenecks at either the CPU or network level.

For data monitoring and analysis, the processes between the bottom and top of the hierarchy can perform partial aggregations which further reduces the load on the control node.

The System Interface component includes tools such as: the development environment, result visualization and monitoring and administrative tools. The Application Specification repository maintains all defined topologies, allowing users to store, retrieve and update topology specifications. Finally, the Resource Configuration maintains the configuration of the resources available on the computational cluster.

6 Experiments

In this section, we present our evaluation of SDAP, which consists of two parts: performance, and usability. Performance evaluation focuses on the ability of SDAP to deliver low latency data processing at scale. The usability evaluation

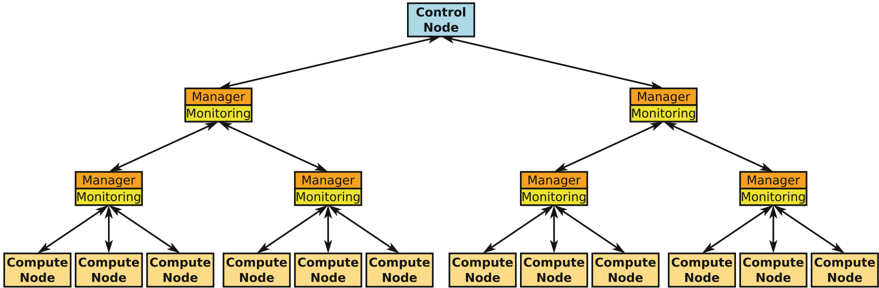

Fig. 4. SDAP architecture: resource and monitor manager scalability.

focuses on the simplicity of use of the tool, compared to popular alternative systems.

6.1 Performance

Latency is a crucial metric for streaming data in a high-performance environment and is defined as the interval of time between the solicitation and the response of a system. For systems targeting analytics on big data, it is important to maintain low latency when the inputs, and subsequent resources, grow to a large number.

We compare SDAP with Apache Storm [27] (version 1.0.2, latest version available at the time of writing) and Google's MillWheel [1]. These systems have been chosen because they adopt the event based data processing paradigm, as with SDAP. In particular, the former explicitly targets low latency performance at a scale; while the latter focused initially on the provision of event based processing primitives and scalability, and has now evolved to deliver low latency in its recent versions.

The common ground on which to compare the systems is a topology composed of the following steps: a data generator, followed by a non blocking stream operation (e.g. select), and a collector. The data generator step generates random data tuples of about 100B each. In implementing the non blocking operation, the stream operation step perform the following tasks: (i) record the timestamp of when the tuple is received, (ii) scan all attributes in the tuple, to emulate an operator worst case scenario where the operation needs to access all data, (iii) attach a timestamp to the tuple, and (iv) forward the tuple in output to the collector step. The collector records the timestamp of the tuple arrival. The two timestamp are used to calculate the intrinsic latency of the system. Specifically, it is the time elapsed between the reception of the input (system solicitation), and the execution and delivery of the data manipulation to the next step in the topology (system response).

Experiments were conducted on the CUNY's High Performance Computing center. Each node is equipped with 2.2 GHz Intel Sandybridge processors with 12 cores, has 48 GB of RAM and uses the Mellanox FDR interconnect. The topology is deployed so that contiguous steps in the topology require inter-host

communication and thus, require the use of the network media to exchange data (i.e. no communication via shared-memory). We conducted the test by distributing the topology over 100 CPUs, scaling the parallelism of the steps as well as the amount of data generated. Results show that SDAP exhibits a median record latency below 1.2 ms and 95% of tuples are delivered within 2.1 ms. In contrast: Storm has a median record latency just below 3.2 ms, and delivers 95% of tuples in just above 3.2 ms; in MillWheel the median tuple latency is 3.6 ms, while the 95th latency percentile is of 30 ms. Figure 5 illustrate the result of our experiment, excluding Google's MillWheel. MillWheel's platform is not available so it was not possible to run an empirical test: our comparison and analysis is based on the author's evaluation in [1].

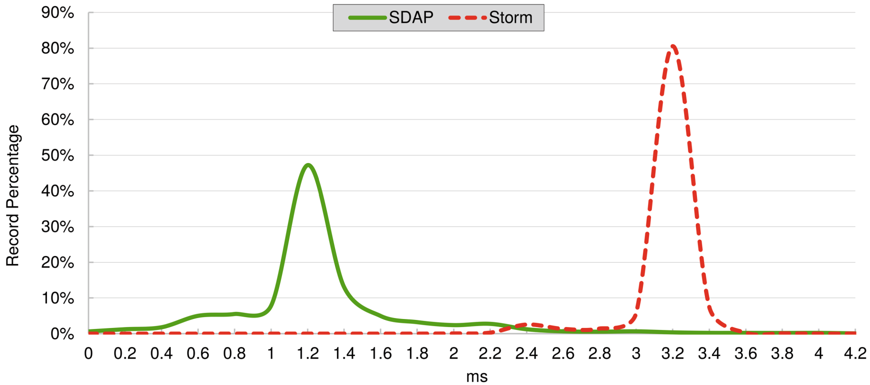

Fig. 5. Tuple processing and delivery latency.

It can be seen that SDAP processes data faster that Storm (and MillWheel) but the latency values are spread close to the median. With respect to Storm, SDAP performs about 3 times better for both the median and the 95th percentile latency. In comparison with MillWheel, SDAP performs 3 times better on the median latency and 10 times better on the 95th percentile latency. Overall, SDAP performs between 200% and 900% better than the other systems.

We also conducted a test to verify how the latency changes when the same number of execution processes are distributed over a small number of machines machines, compared to a large number. We have prepared 4 test scenarios, namely Set8, Set24, Set48, and Set96, see [3]. Each scenario is run on different machine numbers, from the lowest to the highest number of machines that can accommodate the test. For instance, scenario Set8 requires 8 processes with processes run as follows: on the same machine with 8 CPUs; on two machines, using 4 CPUs from each; on four machines, using 2 CPUs from each; and eight machines, using just one CPU from each. For Set96 we started with 12 machines using 8 CPU from each, down to 96 machines using 1 CPU only from each. This was repeated for the remaining configurations. The result of running these scenarios is illustrated in Fig. 6 from [3].

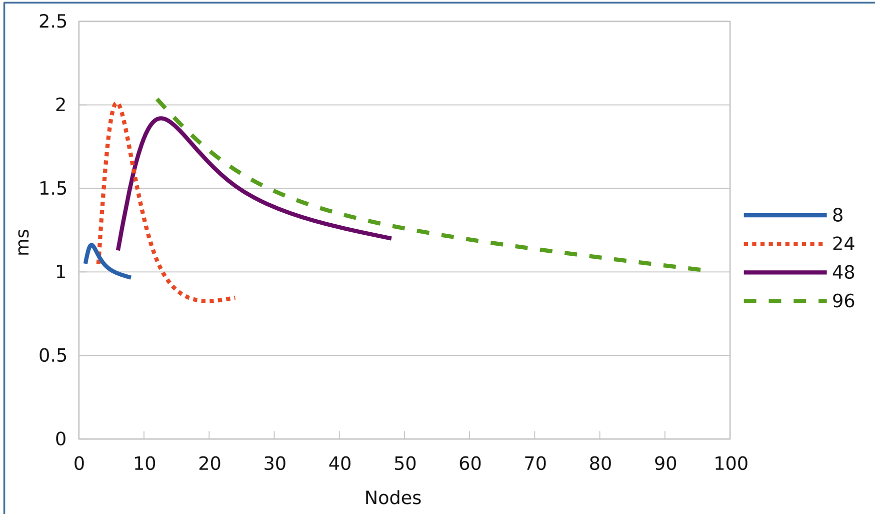

Fig. 6. Tuple processing and delivery latency time by node usage in SDAP.

It can be seen that: (i) the latency performance is quite stable across all configurations, with values supporting the results from the previous experiment; and (ii) the best configuration is when all processes are grouped together on the same machines or when they are highly distributed across different machines. The latter can be as follows: latency is low when all processes are grouped on the same (few) machine(s) because data transfer is (mostly) performed via shared memory (i.e. not via network); latency is also low when the least amount of CPUs is used per machine, because not enough data is exchanged via the network interface which as a result, does not become saturated. Latency is higher when an intermediate number of CPUs are used per machine because the processes generate enough data to flood the network interface while not being able to take advantage of data exchange via shared memory.

6.2 Ease of Development

In this section, we discuss SDAP ease of development, that is, the effort required to develop and maintain a topology. Since MillWheel is not available, and Storm does not provide built-in operators, we have decided to compare SDAP with another popular system: Spark-Streaming. Spark-Streaming provides built-in operators and allows the designer to specify stream applications in a quite succinct manner as Scala programs. While Spark-Streaming supports other languages, Scala has been chosen because it is one of the less verbose and is supported natively.

Let us compare the two systems using a streaming application that must detect tuples that match a set of specified keywords `keywordSeq`. If a tuple

contains a target keyword, it is then forwarded in JSON format to a Kafka end-point. Listing 1.7 shows such application for Spark-Streaming. As we can see, specifying operations such as the cartesian product is rather straightforward. However, even simple operations require a rather verbose specification. To begin with, the developer must select the right libraries to use, such as what package to use for the JSON conversion – omitted in the snippet. Since there are multiple possibilities, the developer is required to study each alternative to determine which one best fits her needs, which takes time. Then, the developer must compose the application. Let us ignore the details of the streams, i.e. the attributes. After the cartesian product operation, line 3, and before checking the keyword match, line 15, the developer must manually open multiple connections to Kafka for each node of the Spark-Streaming cluster. Specifically, the developer opens a connection for each partition of data in the resilient distributed dataset (or RDD, the main data structure in Spark), in an attempt to parallelize the data exchange between the two systems.

```
...
statuses.foreachRDD(rdd => {  //  for each RDD
val cartesian = rdd.cartesian(keywordSeq)

//  for each partition of data, connect to the end-point
cartesian.foreachPartition(partitionOfRecords => {
//  initialize the Kafka producer
    val props = new HashMap[String, Object]()
    setupKafkaProps(props)
    val producer = new KafkaProducer[String, String](props)
//  for each record in partition, check keyword match
  partitionOfRecords.foreach{
    case (status, keywords) => {
//     if a keyword matches, forward to end-point
      if (keywords.map(l => l.toLowerCase()).toSet
        subsetOf status.toLowerCase().split(" ").toSet) {
        val jsonMessage = ("text" -> record.toString)
          ~ ("keywords" -> keywords.toList)
        val jsonMessageString = compact(render(jsonMessage))
//         send message to kafka
        val message = new ProducerRecord[String, String]
          (topicsOutputSet.head, null, jsonMessageString)
        producer.send(message)
...
```

Listing 1.7. Keyword match sample application in spark-streaming.

In contrast, SDAP: (i) has no need to study libraries for inclusion as they are built-in; (ii) the cartesian operator can also be expressed simply but requires no knowledge of a programming language, Scala in this case; (iii) the connection to the end-point is provided by a built-in operator that does not require the

developer to study the inner workings of Scala optimization for Spark-Streaming; and (iv) the set comparison between the record value and a set of keywords can be implemented as a sequence of tokenizer + selection operators. In total, the SDAP would have 4 operators and associated configuration files. Note that the configuration files would be mostly empty, and the in/out stream attributes are automatically populated using the designer portal. Listing 1.8 shows the equivalent topology specification with details of configuration files omitted for the sake of space.

```
...
operator cross_product join-mpi ${join_conf}
operator tokenize_keyword utility-mpi ${tokenizer_conf}
operator keyword_match functor-mpi ${selection_conf}
operator json_encoder_converter-mpi ${json_conf}
operator kafka_endpoint interface-mpi ${kafka_conf}
...
route cross_product:1 roundrobin tokenize_keyword_set:1
route tokenize_keyword_set:1 roundrobin keyword_match:1
route keyword_match:1 roundrobin json_endpoint:1
route keyword_json:1 roundrobin kafka_endpoint:1
```

Listing 1.8. Keyword match equivalent application in SDAP.

It can be observed that the SDAP implementation is easier to read and does not require any previous programming knowledge. In our experience with the SDAP, we have observed that users rapidly familiarize with topology paradigm, with the options of the operators and become power-users capable of developing rather complex transformations.

7 Conclusions

The increasing availability of data provided through online channels has led to an increasing demand to include this form of data in many decision making processes for growing numbers of organizations. The increasing volumes of this data means a greater need for high performance streaming processors. Current systems have been shown to suffer from issues of latency and/or overly complex design and implementation methods. SDAP provides the capability to design and deploy topologies which can scale to very high volumes of data while hiding the complexities of these systems from the designer. Its powerful operators provide a platform for highly complex analytics with SDAP abstracting the underlying management of data and parallel processing. Our evaluation shows SDAP to outperform popular streaming systems such as Storm and MillWheel. Our current research is focused on a few fronts: analysis of application patters, optimization of resource usage, and performance. On one side, we want to exploit the declarative nature of the approach to further simplify the design of stream analytics, and to discover application and resource optimization opportunity.

The visibility and ease of access to the data transformations operation allows to analyze stream analytics design patterns and to optimize the resource allocation. On the other side, we want to further improve performance of the execution engine by including hardware acceleration, e.g. using graphics processing units (GPUs), in the logic of the operators in the context of a high-performance and low-latency environment.

Acknowledgements. This research was supported, in part, from Collective[i] Grant RF-7M617-00-01, the National Science Foundation Grants CNS-0958379,CNS-0855217, ACI-1126113 and the City University of New York High Performance Computing Center at the College of Staten Island.

References

1. Akidau, T., Balikov, A., Bekiroglu, K., Chernyak, S., Haberman, J., Lax, R., McVeety, S., Mills, D., Nordstrom, P., Whittle, S.: Millwheel: fault-tolerant stream processing at internet scale. PVLDB **6**(11), 1033–1044 (2013). http://www.vldb.org/pvldb/vol6/p1033-akidau.pdf
2. Balazinska, M., Balakrishnan, H., Madden, S., Stonebraker, M.: Fault-tolerance in the borealis distributed stream processing system. ACM Trans. Database Syst. **33**(1), 1–3 (2008). http://doi.acm.org/10.1145/1331904.1331907
3. Cappellari, P., Chun, S.A., Roantree, M.: Ise: a high performance system for processing data streams. In: Proceedings of 5th International Conference on Data Science, Technology and Applications, DATA 2016, Lisbon, Portugal, pp. 13–24, 24–26 July 2016
4. Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Lee, S., Seidman, G., Stonebraker, M., Tatbul, N., Zdonik, S.B.: Monitoring streams - a new class of data management applications. In: Proceedings of 28th International Conference on Very Large Data Bases, VLDB 2002, Hong Kong, China, pp. 215–226, 20–23 August 2002. http://www.vldb.org/conf/2002/S07P02.pdf
5. Chandrasekaran, S., Franklin, M.J.: Streaming queries over streaming data. In: Proceedings of 28th International Conference on Very Large Data Bases, VLDB 2002, Hong Kong, China, pp. 203–214, 20–23 August 2002. http://www.vldb.org/conf/2002/S07P01.pdf
6. Chen, X., Beschastnikh, I., Zhuang, L., Yang, F., Qian, Z., Zhou, L., Shen, G., Shen, J.: Sonora: a platform for continuous mobile-cloud computing. Technical report (2012). https://www.microsoft.com/en-us/research/publication/sonora-a-platform-for-continuous-mobile-cloud-computing/
7. Condie, T., Conway, N., Alvaro, P., Hellerstein, J.M., Gerth, J., Talbot, J., Elmeleegy, K., Sears, R.: Online aggregation and continuous query support in mapreduce. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, pp. 1115–1118, 6–10 June 2010. http://doi.acm.org/10.1145/1807167.1807295
8. Ganglia (2015). http://ganglia.sourceforge.net/. Accessed 15 Nov 2016
9. Gedik, B., Yu, P.S., Bordawekar, R.: Executing stream joins on the cell processor. In: Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, pp. 363–374, 23–27 September 2007. http://www.vldb.org/conf/2007/papers/research/p363-gedik.pdf

10. Gehrke, J., Korn, F., Srivastava, D.: On computing correlated aggregates over continual data streams. In: Mehrotra, S., Sellis, T.K. (eds.) Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data, Santa Barbara, CA, USA, pp. 13–24. ACM, 21–24 May 2001. http://doi.acm.org/10.1145/375663.375665
11. Grinev, M., Grineva, M.P., Hentschel, M., Kossmann, D.: Analytics for the real-time web. PVLDB **4**(12), 1391–1394 (2011). http://www.vldb.org/pvldb/vol4/p1391-grinev.pdf
12. Gui, H., Roantree, M.: Topological XML data cube construction. Int. J. Web Eng. Technol. **8**(4), 347–368 (2013)
13. Gui, H., Roantree, M.: Using a pipeline approach to build data cube for large XML data streams. In: Hong, B., Meng, X., Chen, L., Winiwarter, W., Song, W. (eds.) DASFAA 2013. LNCS, vol. 7827, pp. 59–73. Springer, Heidelberg (2013). doi:10.1007/978-3-642-40270-8_5
14. Infiniband (2015). http://www.infinibandta.org/. Accessed 15 Nov 2016
15. InfoSphere streams (2015). http://www-03.ibm.com/software/products/en/infosphere-streams. Accessed 15 Nov 2016
16. Kang, J., Naughton, J.F., Viglas, S.: Evaluating window joins over unbounded streams. In: Proceedings of the 19th International Conference on Data Engineering, Bangalore, India, pp. 341–352, 5–8 March 2003. doi:10.1109/ICDE.2003.1260804
17. Li, J., Maier, D., Tufte, K., Papadimos, V., Tucker, P.A.: Semantics and evaluation techniques for window aggregates in data streams. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, pp. 311–322, 14–16 June 2005. http://doi.acm.org/10.1145/1066157.1066193
18. Madden, S., Shah, M.A., Hellerstein, J.M., Raman, V.: Continuously adaptive continuous queries over streams. In: Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, pp. 49–60, 3–6 June 2002. http://doi.acm.org/10.1145/564691.564698
19. Maier, D., Li, J., Tucker, P., Tufte, K., Papadimos, V.: Semantics of data streams and operators. In: Eiter, T., Libkin, L. (eds.) ICDT 2005. LNCS, vol. 3363, pp. 37–52. Springer, Heidelberg (2004). doi:10.1007/978-3-540-30570-5_3
20. Motwani, R., Widom, J., Arasu, A., Babcock, B., Babu, S., Datar, M., Manku, G.S., Olston, C., Rosenstein, J., Varma, R.: Query processing, approximation, and resource management in a data stream management system. In: CIDR (2003). http://www-db.cs.wisc.edu/cidr/cidr2003/program/p22.pdf
21. MVAPICH2, The Ohio State University (2015). http://mvapich.cse.ohio-state.edu/. Accessed 15 Nov 2016
22. Neumeyer, L., Robbins, B., Nair, A., Kesari, A.: S4: distributed stream computing platform. In: Proceedings of the 2010 IEEE International Conference on Data Mining Workshops, ICDMW 2010, Washington, DC, USA, pp. 170–177 (2010). IEEE Computer Society. doi:10.1109/ICDMW.2010.172
23. Peng, D., Dabek, F.: Large-scale incremental processing using distributed transactions and notifications. In: Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, Vancouver, BC, Canada, pp. 251–264, 4–6 October 2010. http://www.usenix.org/events/osdi10/tech/full_papers/Peng.pdf
24. Plimpton, S.J., Shead, T.M.: Streaming data analytics via message passing with application to graph algorithms. J. Parallel Distrib. Comput. **74**(8), 2687–2698 (2014). doi:10.1016/j.jpdc.2014.04.001
25. Slurm (2015). http://slurm.schedmd.com/. Accessed 15 Nov 2016

26. Teubner, J., Müller, R.: How soccer players would do stream joins. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, pp. 625–636, 12–16 June 2011. http://doi.acm.org/10.1145/1989323.1989389
27. Toshniwal, A., Taneja, S., Shukla, A., Ramasamy, K., Patel, J.M., Kulkarni, S., Jackson, J., Gade, K., Fu, M., Donham, J., Bhagat, N., Mittal, S., Ryaboy, D.V.: Storm@twitter. In: International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, pp. 147–156, 22–27 June 2014. http://doi.acm.org/10.1145/2588555.2595641
28. Trident (2012). http://storm.apache.org/documentation/Trident-tutorial.html. Accessed 15 Nov 2016
29. Zaharia, M., Das, T., Li, H., Shenker, S., Stoica, I.: Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters. In: 4th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud 2012, Boston, MA, USA, 12–13 June 2012. https://www.usenix.org/conference/hotcloud12/workshop-program/presentation/zaharia