ISE: A High Performance System for Processing Data Streams

Paolo Cappellari[1], Soon Ae Chun[1] and Mark Roantree[2]
[1]*City University of New York, New York, U.S.A.*
[2]*Insight Centre for Data Analytics, School of Computing, Dublin City University, Dublin, Ireland*

Keywords: Data Stream Processing, High-performance Computing, Low-latency, Distributed Systems.

Abstract: Many organizations require the ability to manage high-volume high-speed streaming data to perform analysis and other tasks in real-time. In this work, we present the Information Streaming Engine, a high-performance data stream processing system capable of scaling to high data volumes while maintaining very low-latency. The Information Streaming Engine adopts a declarative approach which enables processing and manipulation of data streams in a simple manner. Our evaluation demonstrates the high levels of performance achieved when compared to existing systems.

1 INTRODUCTION

Processing data originating from streaming sources has become a requirement for modern organizations. Driven by the need for more timely results, and having to deal with an increasing availability of real-time information, companies are looking at ways of including data stream processing into daily operations, including real-time alerts, real-time analytics and as support for data integration. To develop a robust platform for streaming applications, there are three broad requirements:

- The system must be capable of managing high volumes of streaming data even when the rate at which data generated is extremely high;
- results of steaming calculations, on which organization base decisions, should be available as soon as possible, ideally in real-time;
- designers must be proficient in parallel computation or high-performance programming.

Existing streaming systems mainly focus on the problems of scalability, fault-tolerance, flexibility, and on the performance of individual operations, e.g. (Akidau et al., 2013; Zaharia et al., 2012; Balazinska et al., 2008; Teubner and Müller, 2011). Our contribution is the delivery of a scalable, low-latency data stream processing system. Unlike other research in this area, we focus on the provision of a complete and comprehensive solution for the rapid development and execution of scalable high-performance, low-latency, real-time data stream processing applications. We provide a declaration-based application development environment that requires no prior knowledge of parallel computation or high-performance programming. A comprehensive evaluation is provided to demonstrate the performance of our system.

In the reminder of this paper, we will refer to our proposed solution as Information Streaming Engine, or ISE. ISE follows a similar design philosophy of other distributed stream processing systems. As with other approaches, ISE can scale seamlessly to use available computational resources, parallelizing the data processing, but it differentiates from other approaches in a number of ways: (i) it offers built-in operators optimized for high-performance environments that go beyond the semantics of relational operators; (ii) targets high-performance both in terms of hardware and software, by supporting the best performing hardware while reducing the software layers separating the computation from the execution; (iii) allows rapid development of high-performance streaming application by shielding users from the (tricky) details of the parallel computing paradigm and the high-performance environments, and (iv) application and cluster specification details are exposed and visible as simple text files (rather than code in some programming language).

The paper is organized as follows. In Sec. 2, we define the constructs and properties of our approach. In Sec. 3, we present the architecture of system. In Sec. 4, we illustrate how to specify data processing. In Sec. 5, we discuss the experimental setting and the performance results. In Sec. 6, we provide a compar-

Cappellari, P., Chun, S. and Roantree, M.
ISE: A High Performance System for Processing Data Streams.
DOI: 10.5220/0005938000130024
In *Proceedings of the 5th International Conference on Data Management Technologies and Applications (DATA 2016)*, pages 13-24
ISBN: 978-989-758-193-9

ison of our approach against other works. Finally, in Sec. 7 we present our conclusions.

2 STREAMING MODEL

Streaming sources generate a (potentially infinite) sequence of data elements, comprising the data stream. A *data element* describes a single event from the input source point of view, such as a sensor reading, a social media post, a stock price change, etc.

In this section, we describe the ISE model through its constructs, operators and properties. Data stream processing systems run continuously: once data processing is launched it never ends, indefinitely waiting for (new) data to be available for processing (Carney et al., 2002; Chandrasekaran and Franklin, 2002; Madden et al., 2002). Results are also generated continuously. For example, an organization may want to monitor posts on social media mentioning their products (e.g. Twitter) and retain (i.e. filter) only those posts containing one of their products in the post text; then further data manipulation can be applied to realize products' statistics, real-time alerts, analytics, etc.

2.1 Model Constructs

In general, stream data applications consist of a sequence of data manipulation operations, where each operation performs a basic transformation to a data element, and passes results to the operation next in the sequence. Examples of basic transformations are: filtering data elements outside our scope of interest, calculating data aggregates over time, correlating data from different streams, or enriching data elements with contextual information from a local repository. When multiple basic transformations are chained together in a pipeline fashion, they create sophisticated, complex, transformations. More generally, a complex transformation is a workflow, where multiple pipelines coexist and are combined. These workflows can be represented as direct acyclic graphs (DAGs) and are also referred to as data flow diagrams or topologies (Carney et al., 2002). The ISE model was developed using the following constructs:

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 Twitter stream, a status update is an event; each status update has multiple values associated with it, including: the status update text, the user-id of the author, the location, and others[1].

Stream. A stream is a sequence, potentially infinite, of events described in data elements, that is, tuples. Tuples in a stream conform to a (known) schema, that is: the values in each tuple in the same stream are instances of a known set of attributes, each having a specific data type. For instance, tuples generated from the status update stream on Twitter have all the same structure, although different values.

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 offered in ISE are detailed later in this section. For now, we discuss two important parameters that are associated with each operator: *parallelism* and *protocol*.

Parallelism. The parallelism defines how many instances of an operator collaborate to realize a data processing step. In order to process large amounts of data, the processing must be distributed across multiple computational resources (cores, CPUs, machines). In a topology, each operator has its own degree of parallelism.

Protocol. The protocol defines how tuples are passed between the instances of contiguous operators in a topology. For instance, depending on the nature of the transformation, a tuple can be passed to just one instance or to all instances of the next operator in the topology. ISE 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 of the downstream node in a round-robin fashion. Round-robin distributes the data *evenly* 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. However, it may lead to uneven usage of downstream resources. With the **broadcast** routing strategy, every output tuple from a single instance of an upstream node, is copied to all instances of the downstream node. This mechanism is useful in multiple scenarios, such as synchronizing or sharing specific pieces of data

[1]Twitter object field guide: https://dev.twitter.com/overview/api/tweets

among all processes. Details on topology specification and implementation are provided in Sec. 4.

Topology. A topology describes how the data stream flows from the input source(s) through the operators to the output. It is modeled as a DAG, where nodes represent operators, and edges describes how tuples move between operators.

ISE provides a declaration-based approach to data stream processing development using the constructs listed above. Users create a **topology** specification using a set of built-in operators optimized for high-performance distributed environments. Specifying a transformation or a topology does not require any programming. Furthermore, all details of a topology, its transformations, parallelism and resources are exposed in specification (text) files. Higher level tools, such as GUI or languages (e.g. (Falt et al., 2014)), can be used to generate, edit and maintain such specifications. The objective of this section is not to propose another language, but to expose the lower-level details of how topologies are specified when provided as input to an data streaming processing engine.

2.2 Operators

This section presents some of the operators offered in ISE. The portfolio of operators is wide enough to enable designers to realize very complex transformations. The rationale for providing a set of built-in operators is: (i) application designers can focus on the transformation workflow and ignore the operator's implementation details; (ii) the operator semantic is guaranteed and consistent across the whole system, and (iii) each operator's implementation delivers the best possible performance; and (iv) new operators can be developed and added to the system, if and when necessary. ISE 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. Examples included in ISE are:

- Projection: Reduces the stream to a subset of its attributes; similar to the SQL projection operator.
- Selection: Using a potentially complex condition, it splits the input stream in two: one stream containing tuples satisfying the condition, the other stream with those not satisfying the condition.
- Function: Implements multiple operations including: adding constants or a sequence attribute to the stream; text-to/from-date conversion; math (addition, division, modulo, etc.) and string functions (find, contains, index-of, etc.) between one or more attributes in the tuple.
- Tokenizer: Breaks a value from an input tuple into multiple parts (tokens) based on a pattern used for matching. It creates a new output tuple for each token where the output tuple is a copy of the input tuples and the matched tuple.
- Delay: Holds each tuple for a specified interval before forwarding it unaltered.

The **Aggregate** operator groups and summarizes tuples from the input stream. It implements the SQL-like aggregations: average, sum, max, min and count. The operator requires a *window* definition that specifies when and for how long (or how many) tuples from the stream to include in the aggregation.

The **Join** operator correlates tuples from two streams according to a join condition. This is similar to the relational join: when values from two tuples, each from a different stream, satisfy a (potentially complex) condition, the input tuples are merged to produce a new tuple in the output stream. Similar to Aggregate, this operator requires the definition of a window specifying what 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. Currently, ISE supports the following:

- DB_Statement: executes an arbitrary SQL statement against a database.
- Lookup: retrieves data matching a key value from the current tuple from a lookup datastore (either a file or a database).

The **Interface** operator enables ISE 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 ISE (e.g. other topologies). Currently, ISE provides:

- Twitter_Connector: Receives data from the Twitter stream.
- Salesforce_Connector: Receives data from an organization's Salesforce stream.

- Http_Connection: Retrieves data from a generic HTTP end-point.
- ZMQ_in, ZMQ_out: Receives/Delivers streams from/to ZeroMQ[2] end-points, a brokerless high-performance message passing system. ZeroMQ end-points are used to expose result data from a topology to consumer applications, including other topologies.

The **FormatConverter** provides data format conversion between the tuple and other formats when processing data within a topology. ISE offers XML and JSON conversion, where values to extract or to format in either XML or JSON can be specified via an XPath-like syntax.

The **Control** operator is used to control the status and perform administrative actions on individual operators or on the a whole topology. Some of the control messages are: sync all operators, get operator status, force operator cache flush, etc.

Additional operators can be added as part of a Utility category to support special operations. Currently, the ISE has the geo-coding operator, to transform a location name into latitude and longitude.

2.3 ISE Sample Application

Fig. 1 illustrates a sample topology analyzing Twitter. Here, the rounded rectangles represent the nodes in the topology. Edges between nodes describe how the stream's tuples flow from one transformation to the next. The operator applied by each node 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 node. On top of each node, 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 tuples satisfying the condition; the dashed edges are the stream of tuples *not* satisfying the condition. When one or the other output stream from Selection is not used, the edge is not shown in the illustration.

The topology in Fig. 1 describes a streaming application that analyzes the Twitter stream by producing three output streams: one providing tweets generated by users from a known list, a second providing tweets mentioning products (from a known list) enriched with geo-location information, and a third providing the number of times the product is mentioned, grouped by category. Data flows into the topology by the `Twitter Connector` operator that connects to the Twitter stream and delivers each tweet to the next operator in the chain. Tweets are provided in JSON format and thus, tuples are passed to the `JSON Parser` operator that convert data from JSON into a tuple format. The next step removes some attributes that are not necessary for the application, before creating two replicas of the stream: one stream focusing on user identifiers, the other with products.

On the user identification stream, additional steps are required. The selection step filters tweets (tuples) not authored from the users provided in a list; the JSON encoder converts each tuple into a JSON format; and finally, the ZMQ-Endpoint makes the stream available to consumer applications via ZeroMQ.

On the product stream, additional transformations also occur. It is worth noting the *lookup*, *aggregation* and *geo-coding* functions. The lookup function enriches each tuple by adding the category a product belongs to; where a mapping product-to-category must be provided. The aggregation function counts how many times a product is mentioned at regular intervals; the geo-tagging function converts a location name into latitude and longitude where they are not already present in the tuple.

3 THE ISE ARCHITECTURE

ISE was conceived with high-performance environments in mind. Nevertheless, ISE can run on a wide variety of computational resources, from a standard desktop to a shared memory or shared nothing system to a geographically distributed cluster. Fig. 2 shows the main ISE components: the Resource Manager, Clustering, the Data Operators, Monitoring, the System Interface, the Application Specification and the Resource Configuration repositories.

The transformation operators as well as the user space components have been developed from scratch in c/c++. Components such as the resource manager, the computation clustering and the monitoring tool, are based on the following open source libraries: Slurm(Slurm, 2015), MVAPICH2(MVAPICH2, The Ohio State University, 2015), and Ganglia(Ganglia, 2015), all of which are well established projects in high-performance computing.

At the core of the system there is the Clustering component, which can be decomposed in two subcomponents: (i) parallelism and data movement, and (ii) data operations. The first takes care of managing parallel processes and moving data between them; the second implements the data processing operation, enforcing the semantics of these operations across the parallel processes. With regards to the former,

[2]ZeroMQ http://zeromq.org/

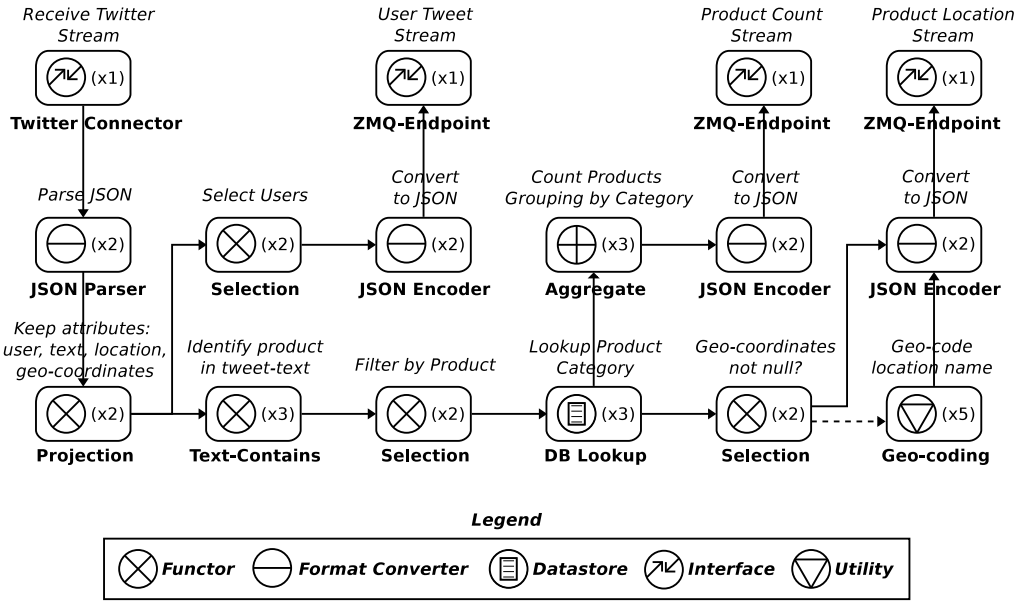

Figure 1: Topology for Twitter Case Study.

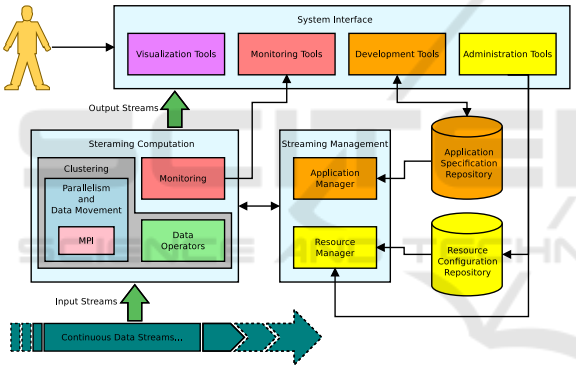

Figure 2: Architecture: Logical View.

we adopt an implementation of the Message Passing Interface (MPI), specifically MVAPICH2, in order to optimize the advantage of using these high-performance environments. MPI is a message passing system designed to support the development of parallel distributed applications. MPI is designed to achieve high performance, scalability, and portability. MVAPICH2 distinguishes itself from the other MPI implementations because it is one of the best performing and because of its support for the most recent and performing hardware, such as Infiniband(Infiniband, 2015), a high-performance inter-connector, designed to be scalable and featuring high throughput and low latency. More precisely, the Parallelism and Data Movement component builds on top of Phish (Plimpton and Shead, 2014), that in turns uses MPI. Phish is library to develop topologies composed of executables, providing an abstraction on parallelism and message delivery. Together with Phish, ISE has a Data Operators component which offers a set of built-in operations, including selection, projection, join, etc.

In general, multiple streaming topologies run on a cluster. A clusters can accommodate multiple topologies that run concurrently. However, since resources are limited, there is the need for managing and monitoring such resources. Components Resource Manager and Resource Monitoring achieve these goals. The Resource Manager is built on top of Slurm (Slurm, 2015). Slurm is 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 facilitates the management of available computational resources and the allocation and deployment of streaming topologies. Among the features it provides, is topology relocation to other resources and a fault-tolerance mechanism. Ganglia (Ganglia, 2015) is our resource monitoring system: it is highly scalable and works in high-performance computing settings.

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

4 TOPOLOGY SPECIFICATION

This section illustrates how topologies are specified in ISE. The specification is composed of three main parts: the specification of the DAG composing the topology; the operator configuration for each node in the topology; and a set of deployment options to specify the execution model. ISE accepts topology specification as text files. These files are exposed, thus can be edited directly or via higher level tools.

4.1 Topology Syntax

A topology is modeled as a DAG. To define a DAG composing a topology in ISE we need to specify: (i) the list of all nodes in the DAG, each with its associated operator, (ii) the connections between the nodes, and the associated protocol; and (iii) the degree of parallelism of each node.

With reference to Fig. 1, Listing 1 shows an excerpt of the topology specification to illustrate how nodes in a DAG are declared in ISE. The excerpt focuses on the bottom right part of Fig. 1, specifically on nodes: `Geo-coordinates not null?`, `Lookup Product Category`, `Geo-code location name`, and the right-most `Convert to JSON`. For convenience, above nodes are renamed to `add_product_category`, `has_geo_info`, `geo_code`, and `product_json_encode`, respectively.

```
## Product stream, nodes
...
operator
   add_product_category
   datastore-mpi
   ${product_category_lookup_conf}
operator
   has_geo_info
   functor-mpi
   ${has_geo_info_conf}
operator
   geo_code
   geocode-mpi
   ${geo_code_conf}
operator
   product_json_encode
   format_converter-mpi
   ${product_json_encode_conf}
...
```

Listing 1: List of Nodes in a Topology.

Each instruction in the topology specification follows the expression syntax outlined in Definition 1.

Definition 1 (Node).

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

This can be explained as follows:

- `operator` is a ISE keyword that declares the presence of a node in the DAG, and indicates the beginning of the definition of such node;
- `node-label` specifies the mnemonic name to associate with the node being defined. A node name is used within the topology specification to refer to the specific node when defining connections with other nodes and parallelism;
- `operator-executable` specifies the operator associated with the node; specifically, it indicates the (relative) path to the operator's executable;
- `node-configuration` specifies the behaviour of the operator, that is the operator's configuration; specifically, it is the (relative) path to a file containing the operator's configuration details for the node being defined.

Details on how to specify an operator's configuration are presented in the Sec. 4.2. In Listing 1, line 3 defines a node with label `add_product_category`, that is associated with operator Datastore, whose configuration is in file `product_category_lookup_conf`. It implements a lookup from a database, retrieving the product category associated with a particular product, that enriches each tuple by adding the category a product belongs to. Similarly, line 5 defines node `has_geo_info` that implements a Functor operator whose configuration is in file `has_geo_info_conf`. This node corresponds to `Geo-coordinates not null?`, and it routes tuples to the geo-coding operation or to the JSON converter, depending on whether the tuple has geo-information already or not.

The second part of the topology specification defines how tuples are routed from one node to the other in the topology, that is the edges in the topology DAG. Listing 2 shows an excerpt of the specification illustrating the linking of the nodes in Listing 1.

Definition 2 (Route).

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

Each expression follows the syntax in Definition 2, where:

- `route` is the ISE keyword that denotes the beginning of an edge declaration that will directly connect two nodes in the topology;
- `upstream-node-label` is the label of one node participating in the connection, specifically the upstream node;

- `protocol` specifies a routing strategy between the nodes being connected: *direct*, *round-robin*, *hash* or *broadcast*;
- `downstream-node-label` is the label of the other node participating in the connection, specifically the downstream node; node;
- `port` specifies which port each node will use to send/receive tuples.

The specification in Listing 2 shows the part of the topology in Fig. 1 that links the *product category lookup* and the *geo-coding* steps. Specifically, all tuples from the `add_product_category` node are passed to `has_geo_info` via port number 1 (for each). Then, node `has_geo_info` splits the stream in two: those tuples having geo-data (i.e. latitude and longitude) are passed to node `product_json_encode` via port 1, for the JSON conversion. All remaining tuples are passed to node `geo_code` via port 2, to evaluate the latitude and longitude from a location name (then tuples are sent to the same JSON conversion step). For simplicity of presentation, all edge declarations use the protocol `roundrobin` to exchange tuples between the instances of the involved nodes.

```
## Product stream, connections
...
route
   add_product_category:1
   roundrobin
   has_geo_info:1
route
   has_geo_info:1
   roundrobin
   product_json_encode:1
route
   has_geo_info:2
   roundrobin
   geo_code:1
route
   geo_code:1
   roundrobin
   product_json_encode:1
...
```

Listing 2: Connections Between Nodes in a Topology.

Listing 3 illustrate the final part of a topology specification: the parallelism of each operator (node).

Definition 3 (Parallelism).

```
parallelism <node-label> <degree>
```

The syntax for parallelism is shown in Definition 3, where:

- `parallelism` is the ISE keyword indicating the beginning of a parallelism declaration for a node in the topology;
- `node-label` indicates the node to which the parallelism is applied;
- `degree` specifies how many instances (i.e. runtine processes) to instantiate for the target node.

From Listing 3, we can see that nodes `add_product_category`, `has_geo_info`, `geo_code`, and `product_json_encode` have parallelism $3, 2, 5, 2$, 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, because the lookup and the geo-code operation are expected to be heavier (i.e. slower) than the others, a higher degree of parallelism is required. Note that values in Listing 3 are for illustration purpose. Real-world deployments these values have, in general, much higher values.

```
## Product stream, distribution
...
parallelism add_product_category 3
parallelism has_geo_info 2
parallelism geo_code 5
parallelism product_json_encode 2
...
```

Listing 3: Parallelism of Nodes in a Topology.

4.2 Operator Configuration

This section focuses on how to provide a configuration to a node in a topology. Each operator has a different signature, meaning its executable expects a different set of arguments to represent the operator's behavior at runtime. The first part of the signature is common to all operators: it represents the metadata of the input and output streams. The second part of the signature is operator dependent: each operator has a different set of parameters. Listings 4 and 5 shows snippets of an operator's configuration file for the *Aggregate* node example in Fig. 1. Listing 4 shows the schema of tuples for the input and the output streams. In this case, the stream is "reduced" from four input attributes to just two in output.

Listing 5 shows how the aggregation is specified. The first part specifies the grouping criteria: values from the input streams are grouped on the attribute `ProductCategory`. The second part specifies the aggregate to run; the input attribute; and the output attribute generated. In the case, the node will count the occurrences of `ProductName` for each category group, and provide the result in attribute `count_products`.

```
"in": [
   {"name": "TweetCreationDate"
      , "type": "String"}
   ,{"name": "ProductName"
      , "type": "String"}
   ,{"name": "ProductCategory"
      , "type": "String"}
   ,{"name": "TweetLocation"
      , "type": "String"}
   ]
,"out": [
   {"name": "ProductCategory"
      , "type": "String"}
   ,{"name": "count_products"
      , "type": "Integer"}
   ]
...
```

Listing 4: A Node Input and Ouput Stream Metadata.

```
...
,"groupby": [
   {"attribute": "ProductCategory"}
   ]
,"aggregate": [
   {"input_attribute_name":
      "ProductName"
   ,"operation": "count"
   ,"output_attribute_name":
      "count_products"}
   ]
...
```

Listing 5: An Operator Configuration.

For operators requiring a window definition, there is an additional part in the configuration file, not shown here for the sake of brevity. ISE supports arbitrarily complex windows, including those based on: wall-clock intervals, number of observed tuples, designated progressing attribute in the stream, and external events (e.g. control messages). Designers can use conditions to specify how often to open a window and when to close a window; multiple conditions can be specified and grouped where required.

4.3 Topology Deployment

Once a topology is defined at the *logical* level, it must be associated with the *physical* resources to use at runtime (CPUs, cores, amount of memory, etc.) and how such resources should be used. For instance, it is possible to declare whether a resource should be allocated exclusively or not; what policy to use to distribute the workload over the cluster; how much memory to use, etc. Listing 6 illustrates a snippet of a deployment specification. Directives are prefixed with `#ISE_` and suffixed with either `RM` or `MPI`: the first suffix is a directive to the resource manager; the second is a directive to the MPI library.

```
1  # Topology name
2  #ISE_RM --job-name=UsersAndProducts
3
4  # Cluster partition to use
5  #ISE_RM --partition=development
6
7  # Cluster nodes to allocate
8  #ISE_RM --nodes=3
9
10 # Processes to allocate
11 #ISE_RM --ntasks=13
12
13 # Node sharing policy
14 #ISE_RM --share
15
16 # Core sharing policy
17 #ISE_RM --overcommit
18
19 # Message passing policy
20 #ISE_MPI --interrupt
21 ...
```

Listing 6: Topology Deployment Specification.

The Node sharing policy, Listing 6 line 15, specifies if a node should be allocated exclusively to a topology. This type of configuration is useful where topologies have operators that can exploit the intra-node communication mechanism which is faster. It is also useful when the computation should not be affected by external factors such as another topology taking part of the machine resources. The Core sharing policy, Listing 6 line 18, defines whether a CPU core should run a single process exclusively or not. In high-performance settings, many application default to exclusively allocating one process per core, in order to extract the maximum possible performance. There are many scenarios where such a policy is not ideal, perhaps because it is known that the stream will not require all resources or because computational resources are limited. In such cases, it is preferable to share the core with multiple processes, perhaps belonging to multiple topologies. Note that when resource sharing is enabled in ISE, topologies are not suspended and queued, as in some supercomputing environments: they are always active. However, sharing resources implies that the CPU is subject to the context switch overhead. Note also that ISE does not spawn threads: it only uses processes. This greatly simplifies the computation distribution and management and does not impact performance in Linux environments, where the cost of managing a thread or process is basically equivalent.

The message passing policy, Listing 6 line 21, de-

fines how a receiving process should check whether new data is available for processing. At the conceptual level, message passing can be thought of as a queuing mechanism. This policy specifies if the receiving process should actively check the queue, thus consuming resources, or if it must rely on an interrupt mechanism for notification. In general, the active polling mechanism is more convenient when new messages arrive at a rate that is close to or matching the process maximum throughput. When the number of messages is relatively low compared to the throughput, one may want rely on an interrupt mechanism so that the resources are temporarily released and are utilizable by some other process.

5 EVALUATION

The evaluation of our system focuses on latency and scalability Latency is the interval of time elapsing between the input (solicitation) and the output (response) of a system, a critical metric for real-time data processing systems. For systems like ISE, it is important to scale to large numbers of computational resources while maintaining latency as low as possible.

We compare our performance to Google's MillWheel(Akidau et al., 2013). MillWheel is the most performing system among those providing a complete data streaming processing solution. We have created a replica of the test scenario used in MillWheel, provided the differences between the platforms. This is a simple two step processing topology where: the first step implements a basic operation; and the second step collects the processed records. We tested the topology with multiple state-less operators, including: `projection` (drop some attributes from the stream), `selection` (evaluate stream values against conditions), `addition` of a constant to the stream (e.g. timestamp), etc. At a minimum, each step performs the following operations:

- Access all the input fields in the incoming record in order to simulate a generic scenario where the designer can pick any value in the stream.
- Perform a basic operation on at least one field of the input record.
- Assemble the output record (each received field) plus a new one if it exists and forward the new record to next step in the topology.

Latency is measured as the time that elapses from when a record is received by the processing step to when a processed record is received by the subsequent collector process. Note that the collector processes are also distributed, thus latency accounts for both the time to process a record and for the time that it takes to the processed record to reach the next step in the topology. Experiments have been conducted on the CUNY's High Performance Computing center. Each node is equipped with 2.2 GHz Intel Sandybridge processors with 12 cores, has 48GB of RAM and uses the Mellanox FDR interconnect.

Fig. 3 shows the latency distribution for record processing and delivery when running over 100 CPUs. Input data is evenly distributed across all CPUs and scaled up as the topology parallelism scales. An analysis of the experimental results shows that the median record latency is 1.2ms, and 95% of the records are processed and delivered within 3 milliseconds.

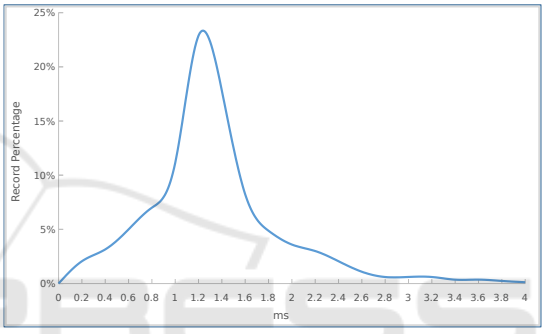

Figure 3: Record Processing and Delivery Latency Percentage.

Compared to Google's MillWheel (Akidau et al., 2013), ISE delivers better performance. In fact, with strong productions and exactly-once disabled in MillWheel, the median record processing time in MillWheel is 3.6 milliseconds (ms) compared to 1.2ms in ISE, approximately one third of the time. Also, if we look at the 95th-percentile latency, MillWheel exhibits a 30ms latency, while ISE requires just 3ms. Basically, ISE performs 3x better than MillWheel (a 200% performance increase) on the median latency test and 10x better on the 95th percentile test (a 900% performance increase).

In cluster environments, the same number of processes can be distributed over a fewer number of machines, each with many cores, or over many machines with fewer cores. Fig. 4 shows the results of studying how the number of machines affects record latency. We have set up a test to run 4 sets of distributed processing, using 8, 24, 48 and 96 CPUs. Let us refer to these sets as Set8, Set24, Set48 and Set96, respectively. Each of the above sets has been run multiple times over a different number of machines, with the condition that as the number of machines increases, the number of *cores per machine* decreases. For in-

stance, Set8 (distribution over 8 CPUs), has been run with the following configurations: 1 machine with 8 cores, 2 machines with 4 cores, 4 machines with 2 cores, and 8 machines with 1 core. The mechanism is similar for the other sets. As illustrated in Fig. 4, the best configuration is when all processes are grouped together on the same machines or when they are highly distributed across different machines.

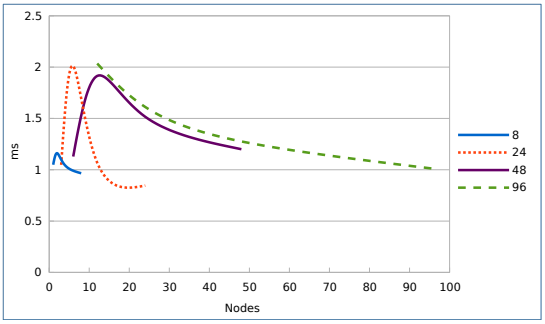

Figure 4: Record Processing and Delivery Latency Time by Number of Nodes Used.

There are many situations in which a streaming topology under-utilizes resources. ISE adapts very well in this scenario, allowing users to specify if and how to share resources at run time. To test resource sharing, we launched multiple instances of the same topology on a single machine, a consumer grade laptop, with an Intel i7 i7-4810MQ, with 4 physical cores and multi-threading (8 threads in total). We were able to run up to 5000 concurrent processes. Sharing resources is useful for low rate data processing scenarios, where multiple topologies can co-exist on the same machines, thus minimizing the amount of hardware resources required.

All performance comparison with MillWheel have been carried out on the basis of the reports from the MillWheel's authors: MillWheel has not been made available, so we were not able to deploy it on the CUNY's HPCC to run direct a comparison.

6 RELATED RESEARCH

Real-time analytic systems are a requirement for most modern organizations, the goals of which are two-fold: provide valuable insights in real-time, and support batch-oriented and data integration tasks.

Examples of recent real-time analytical systems include Google Percolator (Peng and Dabek, 2010), and Limmat (Grinev et al., 2011). The core approach of these systems is to extend the map-reduce paradigm with push-based processing. Basically, new data can be pushed into the process and new output can be computed incrementally on top of the current process state, e.g. aggregates for current windows. A more direct modification of Hadoop is the Hadoop Online Prototype (HOP) (Condie et al., 2010). This approach modifies the Hadoop architecture to allow the execution of pipelined operators. While improving real-time analytic support on map-reduce based solutions, these works still rely on the batch processing paradigm and are thus, inherently limited in pursuing low latency processing. Projects Spark-Streaming(Zaharia et al., 2012) and Twitter Trident(Trident, 2012) can be considered to be in this category as well since they are both micro-batch oriented. However, our ISE approach focuses instead on low-latency and a high-performance environment, neither of which is tackled in these research projects. Spark-Streaming and Trident both offer a set of predefined operators. However, topology design still requires program code development. ISE offers relational and ETL-like operations exposed so that designers do not have to develop programming code which leads to a far quicker deployment of topologies. In addition, ISE supports complex window definitions which are not available in these projects.

S4 (Neumeyer et al., 2010), IBM InfoSphere Streams (InfoSphere streams, 2015), and Twitter Storm (Toshniwal et al., 2014) are systems that specifically target event-based data streaming processing. Unlike the systems described above, these do not rely on batch processing. The S4 streaming system developed at Yahoo!, offers a programming model similar to Map-Reduce. In S4, data is routed from one operation to the next on the basis of key values. IBM InfoSphere Streams offers a set of predefined operation that designers assemble in workflows, similar to ISE. The focus, however, is developing applications where designers have direct control over the quality of the service. Twitter Storm offers a framework to route data through processes, similarly to Phish. ISE offers additional abstractions, such as the data transformation operators, that speed-up topology development time while avoiding the introduction of errors in the operators' implementation and guaranteeing best possible performance.

MillWheel(Akidau et al., 2013) is a Google project for real-time steam processing that specifically targets low-latency performance. This approach relies on a key-based computational model where streams are abstracted as triples composed of a key, a content and a timestamp. Computation is centralized on keys but can be parallelized over different keys. ISE offers greater flexibility in these areas because the designer can choose whether to base the computation on keys or not. Moreover, as shown in our evalu-

ation, ISE delivers better performance.

There is a large collection of work focusing on improving the performance of individual operators. In particular, the join (Carney et al., 2002; Chandrasekaran and Franklin, 2002; Madden et al., 2002; Kang et al., 2003; Motwani et al., 2003; Teubner and Müller, 2011; Gedik et al., 2007) and the aggregate (Li et al., 2005). In contract, the aim of ISE is to develop a platform for defining general streaming computations to be executed in high performance environments. Other research efforts address the broader picture by proposing novel techniques for operators as part of the development of streaming computation frameworks (Abadi et al., 2005; Zaharia et al., 2012).

In (Gui and Roantree, 2013b; Gui and Roantree, 2013a), the approach used for optimizing XML data streams was to build a tree-like topology to model the structure of the incoming XML stream. It introduced a topological cube methodology, built on a multidimensional metamodel for building XML cubes. This research included the ability for recursive analytics, demonstrated using two different forms of recursive structures with both direct recursion and indirect recursion. While this approach has similar goals and approach, the ISE system is designed to scale, adopts an easier to use scripting approach, and can facilitate JSON sources unlike their approach which only uses XML. Systems such as Borealis (Abadi et al., 2005; Balazinska et al., 2008) Medusa (Cherniack et al., 2003), and TelegraphCQ (Chandrasekaran et al., 2003) focus on distributed data processing. However, the workload distribution strategy is at the granularity of query: these systems distribute the workload by allocating queries to individual nodes. While this approach achieves scalability with the number of queries, it is limited in the amount of data an individual query can support, i.e. the parallelism. In recent years, systems have been proposed that address the intra-query scalability problem, such as StreamCloud (Gulisano et al., 2010). While ISE follows a similar design philosophy to that of distributed stream processing systems, it is different in many aspects: it is not limited to relational operators; besides a single query scalability, ISE embraces scalability also from the resource management and monitoring perspective; it targets high-performance environments; and empowers non-expert users to rapidly develop streaming applications.

7 CONCLUSIONS

In this paper, we presented ISE, a high performance framework for developing scalable, low-latency, data streaming applications. Our research provides a high-performance environment and a declarative language for application development based on ISE. The declarative approach allows developers to expose the details of both data processing and resource allocation at both logical and physical levels.

ISE targets high-performance requirements but is also portable to commodity clusters. The framework was tested in terms of latency and scalability performance and compared with other real time processing approaches and our evaluation demonstrated the high performance of ISE when compared to existing streaming environments.

ISE is currently in use in both academic and real-world scenarios. There are a number of topologies analyzing the Twitter stream, ranging from keyword monitoring to searches for specific users, from alerts based on conditions to more classic word counting, word frequency and sentiment analysis. Other topology examples include traffic data monitoring using the numerous bike sharing systems in various cities throughout the world. Traffic data is then used to analyze patterns and make predictions. For instance, we have a topology that in real-time predicts the number of bikes (or available docks for parking) for all stations in the bike sharing network. Such prediction is updated in real-time as fresh data is collected, that is every minute: bikers can decide to reroute depending on their position relative to their destinations.

REFERENCES

Abadi, D. J., Ahmad, Y., Balazinska, M., Çetintemel, U., Cherniack, M., Hwang, J., Lindner, W., Maskey, A., Rasin, A., Ryvkina, E., Tatbul, N., Xing, Y., and Zdonik, S. B. (2005). The design of the borealis stream processing engine. In *CIDR*, pages 277–289.

Akidau, T., Balikov, A., Bekiroglu, K., Chernyak, S., Haberman, J., Lax, R., McVeety, S., Mills, D., Nordstrom, P., and Whittle, S. (2013). Millwheel: Fault-tolerant stream processing at internet scale. *PVLDB*, 6(11):1033–1044.

Balazinska, M., Balakrishnan, H., Madden, S., and Stonebraker, M. (2008). Fault-tolerance in the borealis distributed stream processing system. *ACM Trans. Database Syst.*, 33(1).

Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Lee, S., Seidman, G., Stonebraker, M., Tatbul, N., and Zdonik, S. B. (2002). Monitoring streams - A new class of data management applications. In *VLDB 2002, Proceedings of 28th International Conference on Very Large Data Bases, August 20-23, 2002, Hong Kong, China*, pages 215–226.

Chandrasekaran, S., Cooper, O., Deshpande, A., Franklin, M. J., Hellerstein, J. M., Hong, W., Krishnamurthy, S., Madden, S., Reiss, F., and Shah, M. A. (2003).

Telegraphcq: Continuous dataflow processing. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003*, page 668.

Chandrasekaran, S. and Franklin, M. J. (2002). Streaming queries over streaming data. In *VLDB 2002, Proceedings of 28th International Conference on Very Large Data Bases, August 20-23, 2002, Hong Kong, China*, pages 203–214.

Cherniack, M., Balakrishnan, H., Balazinska, M., Carney, D., Çetintemel, U., Xing, Y., and Zdonik, S. B. (2003). Scalable distributed stream processing. In *CIDR*.

Condie, T., Conway, N., Alvaro, P., Hellerstein, J. M., Gerth, J., Talbot, J., Elmeleegy, K., and Sears, R. (2010). 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, June 6-10, 2010*, pages 1115–1118.

Falt, Z., Bednárek, D., Krulis, M., Yaghob, J., and Zavoral, F. (2014). Bobolang: a language for parallel streaming applications. In *The 23rd International Symposium on High-Performance Parallel and Distributed Computing, HPDC'14, Vancouver, BC, Canada - June 23 - 27, 2014*, pages 311–314.

Ganglia (2015). Ganglia. http://ganglia.sourceforge.net/. [Online; accessed 24-November-2015].

Gedik, B., Yu, P. S., and Bordawekar, R. (2007). Executing stream joins on the cell processor. In *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*, pages 363–374.

Grinev, M., Grineva, M. P., Hentschel, M., and Kossmann, D. (2011). Analytics for the realtime web. *PVLDB*, 4(12):1391–1394.

Gui, H. and Roantree, M. (2013a). Topological xml data cube construction. *International Journal of Web Engineering and Technology*, 8(4):347–368.

Gui, H. and Roantree, M. (2013b). Using a pipeline approach to build data cube for large xml data streams. In *Database Systems for Advanced Applications*, pages 59–73. Springer Berlin Heidelberg.

Gulisano, V., Jiménez-Peris, R., Patiño-Martínez, M., and Valduriez, P. (2010). Streamcloud: A large scale data streaming system. In *2010 International Conference on Distributed Computing Systems, ICDCS 2010, Genova, Italy, June 21-25, 2010*, pages 126–137.

Infiniband (2015). Infiniband. http://www.infinibandta.org/. [Online; accessed 24-November-2015].

InfoSphere streams (2015). InfoSphere streams. http://www-03.ibm.com/software/products/en/infosphere-streams. [Online; accessed 19-October-2015].

Kang, J., Naughton, J. F., and Viglas, S. (2003). Evaluating window joins over unbounded streams. In *Proceedings of the 19th International Conference on Data Engineering, March 5-8, 2003, Bangalore, India*, pages 341–352.

Li, J., Maier, D., Tufte, K., Papadimos, V., and Tucker, P. A. (2005). 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, June 14-16, 2005*, pages 311–322.

Madden, S., Shah, M. A., Hellerstein, J. M., and Raman, V. (2002). Continuously adaptive continuous queries over streams. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, June 3-6, 2002*, pages 49–60.

Motwani, R., Widom, J., Arasu, A., Babcock, B., Babu, S., Datar, M., Manku, G. S., Olston, C., Rosenstein, J., and Varma, R. (2003). Query processing, approximation, and resource management in a data stream management system. In *CIDR*.

MVAPICH2, The Ohio State University (2015). MVAPICH2, The Ohio State University. http://mvapich.cse.ohio-state.edu/. [Online; accessed 24-November-2015].

Neumeyer, L., Robbins, B., Nair, A., and Kesari, A. (2010). S4: Distributed stream computing platform. In *Proceedings of the 2010 IEEE International Conference on Data Mining Workshops*, ICDMW '10, pages 170–177, Washington, DC, USA. IEEE Computer Society.

Peng, D. and Dabek, F. (2010). Large-scale incremental processing using distributed transactions and notifications. In *9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, October 4-6, 2010, Vancouver, BC, Canada, Proceedings*, pages 251–264.

Plimpton, S. J. and Shead, T. M. (2014). Streaming data analytics via message passing with application to graph algorithms. *J. Parallel Distrib. Comput.*, 74(8):2687–2698.

Slurm (2015). Slurm. http://slurm.schedmd.com/. [Online; accessed 24-November-2015].

Teubner, J. and Müller, R. (2011). How soccer players would do stream joins. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*, pages 625–636.

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., and Ryaboy, D. V. (2014). Storm@twitter. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 147–156.

Trident (2012). Trident. http://storm.apache.org/documentation/Trident-tutorial.html. [Online; accessed 24-November-2015].

Zaharia, M., Das, T., Li, H., Shenker, S., and Stoica, I. (2012). 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'12, Boston, MA, USA, June 12-13, 2012*.