CEPSim: A Simulator for Cloud-Based Complex Event Processing

Wilson A. Higashino*†, Miriam A. M. Capretz*, Luiz F. Bittencourt†
*Dept. of Electrical and Computer Engineering
Western University, London, Canada
Email: {whigashi, mcapretz}@uwo.ca
†Instituto de Computação
Universidade Estadual de Campinas, Campinas, Brazil
Email: {wah, bit}@ic.unicamp.br

***Abstract*—As one of the Vs defining Big Data, data velocity brings many new challenges to traditional data processing approaches. The adoption of cloud environments in complex event processing (CEP) systems is a recent architectural style that aims to overcome these challenges. Validating cloud-based CEP systems at the required Big Data scale, however, is often a laborious, error-prone, and expensive task. This article presents *CEPSim*, a new simulator that has been developed to facilitate this validation process. *CEPSim* extends *CloudSim*, an existing cloud simulator, with an application model based on directed acyclic graphs that is used to represent continuous CEP queries. Once defined, the queries can be simulated in different cloud environments under diverse load conditions. Moreover, *CEPSim* is also customizable with different operator placement and scheduling strategies. These features enable researchers and system architects to experiment with different configurations and strategies and to promote research in this field. Experimental results show that *CEPSim* can successfully simulate existing cloud-based CEP systems.**

***Index Terms*—complex event processing; cloud computing; simulation; Big Data.**

I. Introduction

Recent trends in Web technologies and the proliferation of sensor networks and mobile devices have created massive amounts of data that pose many new challenges to traditional data processing approaches. This new Big Data world is often characterized by 4 Vs that identify these challenges [1]: volume, velocity, variety, and veracity.

The velocity dimension of Big Data refers to how fast data are generated and how fast they must be processed. Therefore, handling velocity requires applications with online processing capabilities of fast and continuous streams of data. From the business perspective, the goal is to obtain insights from these streams and to enable prompt reaction to them [2]. Among the technologies that can be used to achieve this goal, Complex Event Processing (CEP) is one of the most prominent.

CEP-based systems interpret input data as a stream of events and accept user definitions of queries (rules) to derive semantically enriched "complex" events from a series of simpler events. These complex events can then be used to trigger actions or be fed back to the system and potentially originate other events. Because of the enormous scale associated with Big Data, traditional CEP systems have been replaced by newer systems that leverage cloud environments to provide the low latency and scalability required by modern applications [3], [4], [5].

Development of efficient query distribution and scheduling strategies is essential to the good performance of cloud-based CEP systems. Nevertheless, validating these strategies at the required Big Data scale is a research problem *per se*. For instance, setting up large cloud deployments is a laborious and error-prone process. Moreover, cloud environments are subject to variations that make it very difficult to reproduce the environment and conditions of an experiment [6]. Finally, the financial cost and time required by these experiments is often very high and may hamper their execution.

A common approach used to overcome these challenges is to use simulators. Many areas of computer science have been using simulators as a fast and efficient way to experiment with different strategies and algorithms without incurring the effort of running large-scale experiments. This approach has also been applied to cloud computing research, as confirmed by the reasonable number of cloud environment simulators available [7], [8], [9]. Nevertheless, these existing simulators often assume a simplistic application model that cannot represent CEP applications running in the cloud.

This article presents a new simulator called *CEPSim*, which can be used to simulate CEP applications in multi-cloud environments. *CEPSim* extends *CloudSim* [8] with a new application model based on directed acyclic graphs (DAGs) that represent continuous queries processing fast streams of data. Using *CEPSim*, execution of these queries can be simulated in different environments, including private, public, and multiple clouds. In addition, *CEPSim* can also be customized with different operator placement and scheduling strategies. These features enable researchers and system architects to experiment quickly with different configurations and strategies, thus encouraging research in this field.

This article is organized as follows: Section II presents background information and related work. Section III discusses the design of *CEPSim*, and Section IV examines how it has been integrated with *CloudSim*. Experimental results are shown in Section V, followed by conclusions in Section VI.

978-1-4673-7278-7/15 $31.00 © 2015 IEEE
DOI 10.1109/BigDataCongress.2015.34

II. Related Work

A. Complex Event Processing

The basis of CEP was established by the work of Luckham on *Rapide* [10] and later on in his book [11]. At about the same time, the database community developed the first classical Data Stream Processing (DSP) systems, such as *Aurora* [12] and *STREAM* [13]. CEP and DSP technologies share related goals, as both are concerned with processing continuous data flows coming from distributed sources to obtain timely responses to queries [14].

This work adopts a terminology based on the Event Processing Technical Society (EPTS) glossary [15], which originated from the CEP literature. According to the glossary definitions, CEP can be considered a superset of DSP.

B. CEP Query Languages

In CEP systems, users create queries (or rules) that specify how to process the input event streams and derive "complex events". These queries have usually been defined by means of proprietary languages such as Aurora Stream Query Algebra (SQuAl) [12] and CQL [13].

This research uses Directed Acyclic Graphs (DAG) as a language-agnostic representation of CEP queries, which is a natural choice corroborated by many studies in the literature. For instance, many CEP systems also use DAGs to represent user queries. This is the case with *Aurora* [12], *StreamCloud* [5], *Storm* [16], and many others.

Systems that use declarative query languages, on the other hand, transform user queries into query plans to make them "executable", which often leads to structures that can be mapped into DAGs. The *STREAM* system and the CQL language [13] are examples that use this approach, as well as the *TimeStream* [4] project. Finally, some systems are based on pattern-based query languages that cannot be directly mapped to DAGs. Even in this case, however, previous studies [17] have shown that is possible to transform these queries into DAG structures.

C. CEP in the Cloud

The recent emergence of cloud computing has been strongly shaping the CEP landscape. For instance, *TimeStream* [4] and *StreamCloud* [5] are CEP systems that use cloud infrastructures as their runtime environments.

Similarly, the discussion around Big Data, and, more specifically the rise of the *MapReduce* platform [18], have also had a great impact on CEP. The prevalence and success of *MapReduce* has motivated many researchers to work on systems that leverage some of its advantages while at the same time trying to overcome its limitations when applied to low-latency processing. *StreamMapReduce* [19], M3 [20], and Twitter's *Storm* [16] are examples of MapReduce-inspired systems aimed at stream processing.

D. Simulator

The use of simulation in research of large-scale cloud applications has motivated the development of a number of simulators such as *GreenCloud* [7], *CloudSim* [8], and *iCanCloud* [9]. None of these, however, can effectively model CEP applications.

GreenCloud [7] targets packet-level simulation and energy consumption of network equipment, which are not the focus of this research.

CloudSim [8], on the other hand, is a well-known cloud computing simulator that can represent various types of environments, including private, public and multiple clouds. It provides customizable virtual machine allocation and provisioning policies and also supports a number of energy consumption and network models. The major drawback of *CloudSim* is its simple application model, which is more appropriate for simulation of independent batch jobs. This work circumvents this limitation with a new model based on DAGs.

Because of its flexibility, *CloudSim* has originated many different extensions in the literature [6], [21], [22]. Garg and Buyya [6] created *NetworkCloudSim*, which extends *CloudSim* with a three-tier network model. Guérout *et al*. [21], on the other hand, focussed on implementing the DVFS model on *CloudSim*. Finally, Grozev and Buyya [22] presented a model for three-tier Web applications and incorporated it into *CloudSim*. These extensions are orthogonal to those presented in this paper because they do not focus on CEP.

The *iCanCloud* simulator [9] is similar to *CloudSim*, but it can also parallelize simulations and it has a GUI to interact with the simulator. The choice of *CloudSim* over *iCanCloud* in this article was motivated by its more mature codebase, the authors' previous experience, and the larger number of extensions available.

III. CEPSim

CEPSim has been built to simulate multiple CEP queries running on multiple virtual machines deployed in cloud environments. Its main design principles are:

- *Generality*: it can simulate different CEP cloud-based systems independently of query definition languages and platform specificities;
- *Extensibility*: it can be extended with different operator placement, operator scheduling, and load shedding algorithms;
- *Multi-Cloud*: it can run the simulation on multiple clouds at the same time;
- *Self-containment*: it can be integrated with cloud simulators other than *CloudSim*.
- *Reuse*: it can reuse capabilities that are present in *CloudSim* and comparable simulators.

Figure 1 shows a component view of *CEPSim*. The *CEPSim Core* component, described in this section, is responsible for most of the CEP-related logic whereas the *CloudSim* component manages the simulation infrastructure and execution. The *CEPSim Integration* component implements the pieces

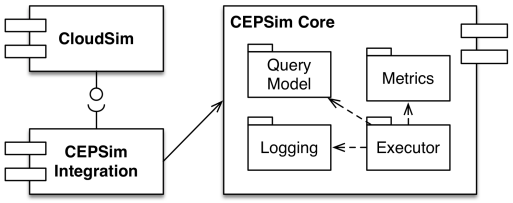


Fig. 1. *CEPSim* components.

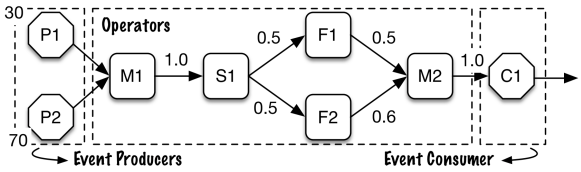


Fig. 2. Query example.

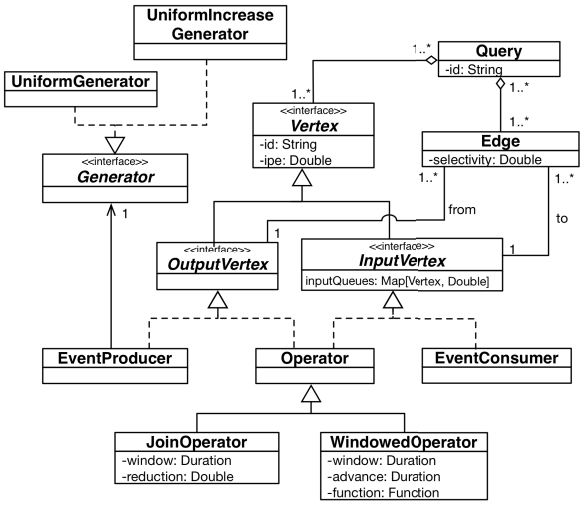


Fig. 3. *CEPSim* class diagram.

necessary to integrate both and is described in Section IV. This design guarantees a loose coupling between *CEPSim* and *CloudSim* and enables future integration with other simulators.

The *CEPSim Core* is composed of four main subcomponents: the *query model*, which contains the base classes used to describe queries; the *query executor*, which implements most of the simulation logic; *logging*, which provides logging facilities for the simulation; and *metrics*, which calculate the metrics of interest to the simulator. The next subsections detail these subcomponents.

A. Query model

In *CEPSim*, each user-defined query Q is represented by a directed acyclic graph (DAG) $G_Q = (V_Q, E_Q)$, in which the vertices V_Q represent query elements and the edges E_Q represent event streams flowing from one element to another. Figure 2 shows an example of a query Q. The numbers on the event producers represent their event generation rates, whereas the numbers on the edges are their selectivities. These concepts are described in detail in the following paragraphs.

Figure 3 shows a diagram of the classes used to represent query graphs in *CEPSim*. The $Vertex$ class is at the top level of the hierarchy, and two subtypes of this class have been identified: $OutputVertex$ and $InputVertex$. The former represents vertices with outgoing edges, and the latter represents vertices with incoming edges. $EventProducer$ describes event producers (sources) and therefore is a subclass of $OutputVertex$ only. Similarly, $EventConsumer$ characterizes event consumers (sinks) and is a subclass of $InputVertex$. Finally, an $Operator$ is both an $OutputVertex$ and an $InputVertex$ because it receives events from some vertices and sends them to others.

1) Vertex: Every $Vertex$ has a unique identifier (id) and an attribute instructions per event (ipe), which represents the number of CPU instructions needed to process a single event. For $EventProducer$s, this attribute estimates the number of instructions required to take events from the system input queues and forward them to query execution. A similar observation applies to $EventConsumer$s: the ipe attribute measures only the effort required to forward the resulting events to external systems or servers that effectively consume the events.

2) Edge: A graph $Edge$ connects an origin $OutputVertex$ with a target $InputVertex$, and has an associated $selectivity$ attribute. The $selectivity$ determines how many of the events that are input to the origin vertex are sent to the target. For example, the outgoing edges of operators that only transform events have $selectivity = 1$, whereas operators that filter or combine events have $selectivity < 1$.

3) Query: A $Query$ is defined as a simple composition of vertices and their associated edges. Similarly to vertices, a query also has an id which uniquely identifies it in the system.

4) Operator: The base $Operator$ class can be used to simulate stateless operators. For example, an *Aurora* filter is an operator that routes events to alternative outputs based on event attributes [12]. This operator is represented in *CEPSim* by an $Operator$ instance op connected to m neighbours op_m. Each edge $(op \rightarrow op_m)$ has a selectivity that determines the percentage of all op input events sent to op_m.

To simulate stateful operators, two $Operator$ subclasses have been created: $JoinOperator$ and $WindowedOperator$.

A $JoinOperator$ is similar to a database join in the sense that it also finds n-tuples of events that satisfy some condition. A $JoinOperator$ has two parameters: a $window$, which controls the window size used to search for the n-tuples, and a $reduction$ factor that determines the percentage of all possible n-tuples that satisfy the join condition. For example: if op is defined as a join operator with two inputs and a window size of one minute, and if 100 events arrived at each input during the last minute, then there are 10 thousand event pairs that could satisfy the join. The $reduction$ factor controls the number of these pairs that are sent to the op output.

The $WindowedOperator$, on the other hand, is used to

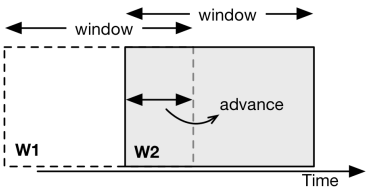


Fig. 4. Window attributes.

simulate operators that process windows of events and combine them in some manner. Typical examples are aggregation operators that count events or calculate the average value of attributes. $WindowedOperator$s behaviour is determined by three main attributes: a $window$ size, an $advance$ duration, and a combination $function$.

Figure 4 illustrates the $window$ and $advance$ concepts. The $window$ specifies the period of time from which the events are taken, and the $advance$ duration defines how the window slides when the previous window closes. Finally, the combination $function$ is defined as:

$$f : \mathbb{R}^m_{\geq 0} \rightarrow \mathbb{R}_{\geq 0} \quad (1)$$

where m is the number of operator predecessors. The function regulates the number of events that are sent to the output given the number of events accumulated in the input. Commonly, it can be defined as a constant $f(x) = 1$, meaning that for each window, only one event is generated (e.g., for counting events).

5) Generator: Every $EventProducer$ is associated with a $Generator$ instance, which defines the number of events generated per simulation interval. *CEPSim* currently contains two different implementations:

- $UniformGenerator$: generates a constant number of events per simulation interval;
- $UniformIncreaseGenerator$: generates a uniformly increasing number of events until it reaches a maximum rate. After this point, this maximum rate is maintained until the end of the simulation.

Other type of generators can be easily added to *CEPSim* by creating extra $Generator$ implementations.

B. Query executor

The main idea of the *CEPSim* simulation algorithm is to keep with each query an internal state representing its execution and, at equally spaced simulation ticks, to update the state of all active queries.

The state of a query on a CEP system is roughly determined by a queue of events associated with each operator's incoming edge (attribute $inputQueues$ from the $InputVertex$ interface). As a simulator, *CEPSim* only tracks the number of events in each queue and does not handle real events. Therefore, to "enqueue" events is equivalent to adding to the queue size counter the number of incoming events. Moreover, *CEPSim* always represents the number of processed events as a floating-point number, which enables "partially processed" events to be exchanged between operators. This characteristic is especially useful to simulate complex operators that require more than one simulation tick to process an event. For example, an operator that requires five simulation ticks to generate an output can be simulated by generating 0.2 events per tick.

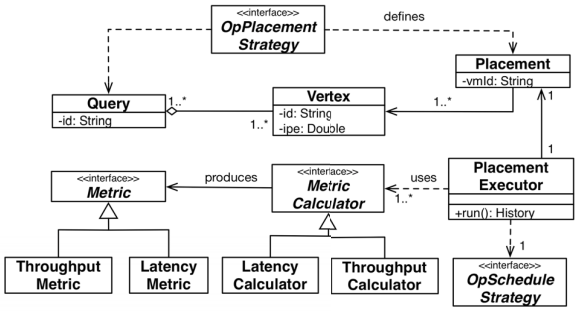


Fig. 5. Query execution class diagram.

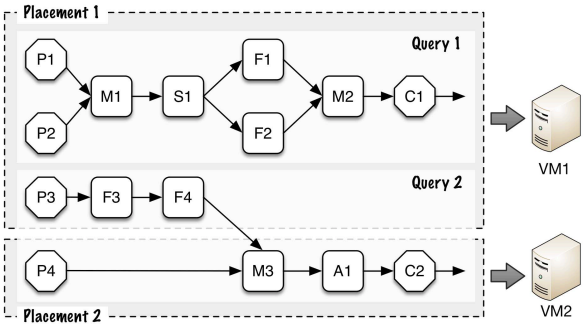


Fig. 6. Placement definitions.

Figure 5 shows the main classes and interfaces developed for query simulation. The first step in a simulation is to define a set of query $Placement$s, in which each $Placement$ maps a set of vertices to the virtual machine on which they will run. Note that the vertices from a query can be mapped to more than one virtual machine, which implies distributed query execution. In addition, a placement can also contain vertices from more than one query. Figure 6 illustrates this concept: $Placement_1$ maps all vertices from $Query_1$ and some from $Query_2$ to Vm_1, whereas $Placement_2$ maps the remaining $Query_2$ vertices to Vm_2. Defining placements for a set of queries is an instance of the operator placement problem, as defined by Lakshmanan *et al.* [23]. *CEPSim* is pluggable and can incorporate different placement algorithms by using alternative implementations of the $OpPlacementStrategy$ interface.

The execution of each $Placement$ is handled by an instance of the $PlacementExecutor$ class, which encapsulates most of the simulation logic. The class has three main methods that constitute the $Placement$ lifecycle: $init$, run, and $finish$.

The $init$ method simply iterates over all vertices belonging to the $Placement$ and initializes each one, whereas the $finish$ method simply frees the resources allocated in the simulation.

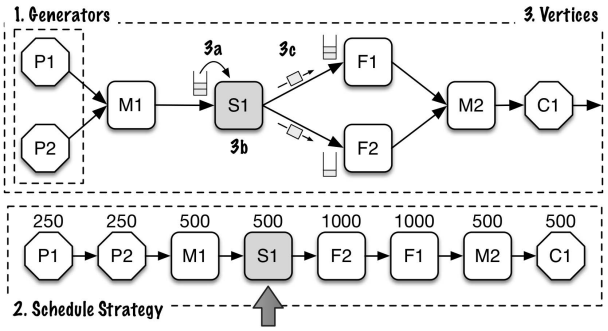


Fig. 7. Execution of a simulation tick.

The run method is invoked at each simulation tick and receives two main parameters: the number of CPU instructions that the placement can use at that tick, and the clock time at which the method has been invoked. Figure 7 shows a schematic representation of the steps executed in a simulation tick, which are:

1) All $Generator$s associated with $EventProducer$s from the placement are activated to determine the number of events that have been generated from the last simulation tick to the current one;
2) The $OpScheduleStrategy$ associated with the $PlacementExecutor$ is invoked to define the order in which the vertices will be run and the number of instructions allocated to each vertex. The default scheduling strategy sorts the placement vertices according to a topological order and allocates a number of instructions proportional to the vertices' ipe attribute. This strategy ensures that each vertex has the chance to process the same number of events at each iteration. Nevertheless, other strategies can also be used by providing alternative $OpScheduleStrategy$ implementations. The bottom of Figure 7 shows an example of a vertex processing sequence determined by a schedule strategy alongside the number of instructions allocated to each vertex.
3) Finally, all vertices are traversed and executed according to the specified order. Each vertex execution is also a multi-step process:

a) The events in the input queues are consumed (step 3a). For $EventProducer$ vertices, the generators are considered as the incoming neighbours. The total number of events consumed is obtained by dividing the number of allocated instructions by the operator ipe. If the operator has more than one input queue, then this total is distributed proportionally to the size of the queues.
b) The operator is "executed", and the output events are generated (step 3b). The number of generated outputs depends on the operator implementation, as described in Section III-A.
c) The generated events are sent to the vertex successors and enqueued in their input queues (step 3c).

TABLE I
CEPSim SIMULATION PARAMETERS.

Name	Description
Simulation interval	Length of the simulation tick
Placement strategy	Algorithm that maps query vertices to virtual machines
Scheduling strategy	(per placement) Algorithm that defines the order on which vertices are traversed and the number of instructions allocated to each
Generator	(per event producer) Function that characterizes the rate at which input events are generated
Queue size	(per vertex) Input queue size

These steps are repeated for all vertices in the sequence returned by the scheduling strategy.

Bounded queues: Most CEP systems limit the size of operator queues to avoid memory overflow and to maintain overall system performance. Because of this characteristic, *CEPSim* also supports the definition of bounded input queues. When using this feature, it is necessary to define the behaviour of the system when new events arrive at an already full queue. Currently, *CEPSim* supports only the application of *backpressure* to the incoming neighbours.

When using *backpressure*, operators inform their predecessors about the maximum number of events accepted for the next iteration at the end of every simulation tick. Then the predecessors limit their output on the next tick (if needed). Nevertheless, when an operator limits its output, it may also accumulate events in its own input queue and, consequently, apply *backpressure* on its own set of predecessors. Ultimately, the backpressure arrives at the event producer generators, which may also choose to discard extraneous events or accumulate them in their own queues.

Table I summarizes the parameters and policies that can be customized by the user and affect the simulation behaviour. The table, however, contains only *CEPSim* specific parameters. Other aspects of the simulation, such as datacentre characteristics and resource allocation policies are defined by *CloudSim*, and their description is out of the scope of this article.

C. Logging

In the run method, *CEPSim* maintains a log of all important events occurring during the simulation. Formally, this log is defined for each placement as a list of three types of entries: 1) *Processed*, representing events processed by a vertex; 2) *Sent*, representing events sent to a remote vertex; and 3) *Received*, representing events received from a remote vertex.

Each log entry has the following attributes: 1) t, the simulation tick number; 2) ts, a timestamp measuring the wall clock time elapsed since the beginning of the simulation; 3) id, the vertex identifier; and 4) $processed$, the number of events processed, sent, or received by the vertex. *Sent* tuples also have a $dest$ attribute containing the vertex identifier to which events have been sent, and *Received* tuples have an $orig$ attribute identifying the vertex from which events have been received.

D. Metrics

One of the most important parts of a simulator is how the metrics of interest are calculated. *CEPSim* has an extensible metrics framework which enables new metrics to be added by creating implementations of the $MetricCalculator$ interface (see Figure 5). These implementations are registered on the $PlacementExecutor$ before the simulation and can measure any aspect of the simulation state. Because of their importance, *CEPSim* provides built-in implementation for two metrics: *query latency* and *query throughput*.

The *query latency* metric is calculated for each consumer c and measures the average number of milliseconds from the moment an event arrives at the query to the moment it is consumed by c. Latency calculation is based on the following procedure:

- Each queue U_{uv} connecting a vertex u to a vertex v has an associated timestamp $ts_q(U_{uv})$. This timestamp measures the average simulation time at which the events have arrived at the queue. If v is an event producer, then u is the generator connected to the producer;
- Each queue U_{uv} also has an associated latency $l_q(U_{uv})$ that measures the average latency of events from their producers to vertex v. If v is an event producer, then $l_q(U_{uv}) = 0$;
- When a vertex v emits a new set of events E_v^t at time t, the timestamp and latency of these events are calculated as:

$$ts_e(E_v^t) = \frac{\sum_{i \in pred(v)} ts_q(U_{iv}) \cdot p(U_{iv}, E_v^t)}{\sum_{i \in pred(v)} p(U_{iv}, E_v^t)} \quad (2)$$

$$l_e(E_v^t) = \frac{\sum_{i \in pred(v)} l_q(U_{iv}) \cdot p(U_{iv}, E_v^t)}{\sum_{i \in pred(v)} p(U_{iv}, E_v^t)} + (t - ts_e(E_v^t)) \quad (3)$$

  where $pred(v)$ is the set of predecessor vertices of v and $p(U_{iv}, E_v^t)$ is the number of events consumed from queue U_{iv} to produce E_v^t.
- For each queue U_{vw} that succeeds v, the timestamp $ts_q(U_{vw})$ and latency $l_q(U_{vw})$ are updated as:

$$ts_q(U_{vw}) = \frac{|E_v^t| \cdot t + |U_{vw}| \cdot ts_q(U_{vw})}{|E_v^t| + |U_{vw}|} \quad (4)$$

$$l_q(U_{vw}) = \frac{|E_v^t| \cdot l_e(E_v^t) + |U_{vw}| \cdot l_q(U_{vw})}{|E_v^t| + |U_{vw}|} \quad (5)$$

  where $|E_v^t|$ is the number of events in E_v^t and $|U_{vw}|$ is the number of events in queue U_{vw}.
- The latency of a consumer c is simply the average latency of all events consumed by c:

$$latency(c) = \sum_{t \in T} l_e(E_c^t)/|T| \quad (6)$$

  where T is the set of all timestamps when c has consumed some event.

Query throughput is also calculated for each consumer c as the average number of events processed per second during its lifespan. Formally, the following procedure is executed to calculate this metric:

- Given a query Q, a consumer c has an associated total $tl(c, p_k)$ for each producer $p_k \in Q_p$, where Q_p is the set of all producers from Q. This value is an estimate of the total number of events from p_k that had to be produced to generate c outputs;
- Each queue U_{uv} connecting a vertex u to a vertex v also has an associated total $tl_q(U_{uv}, p_k)$ for each producer $p_k \in Q_p$. This quantity measures the number of events from p_k that had to be produced to originate the events currently in queue U_{uv}.
- When a vertex v emits a new set of events E_v^t at time t, a total for these events and for each producer p_k is calculated as:

$$tl_e(E_v^t, p_k) = \sum_{i \in pred(v)} \left(\frac{p(U_{iv}, E_v^t)}{|U_{iv}|} \cdot tl_q(U_{iv}, p_k) \right) \quad (7)$$

- For each vertex w that succeeds v and each producer $p_k \in Q_p$, the total $tl_q(U_{vw}, p_k)$ is updated as:

$$tl_q(U_{vw}, p_k) = tl_q(U_{vw}, p_k) + tl_e(E_v^t, p_k) \quad (8)$$

- When a consumer c consumes a set of events E_v^t, its total for each producer p_k is updated as:

$$tl(c, p_k) = tl(c, p_k) + tl_e(E_v^t, p_k) \quad (9)$$

- Finally, the throughput of consumer c is calculated as:

$$throughput(c) = \left(\sum_{p_k \in Q_p} \frac{tl(c, p_k)}{|paths(p_k, c)|} \right) / Q.time \quad (10)$$

  where $|paths(p_k, c)|$ represents the number of paths from producer p_k to consumer c and $Q.time$ is the query execution time in seconds.

IV. CloudSim Integration

In accordance with the reuse design principle, *CEPSim* leverages many functionalities provided by *CloudSim* to enable the simulation of CEP queries. For instance, *CloudSim* is used to define the cloud computing environments in which the simulations are run and to select and customize resource allocation algorithms used in these environments. These algorithms implement different strategies for assigning physical servers and CPUs to virtual machines, and for splitting the available processing power among VM processes. In addition, *CloudSim* also provides a discrete simulation framework that is used to control the *CEPSim* main simulation loop.

This section describes how *CloudSim* has been extended and integrated with the *CEPSim core* described in the preceding section. The main parts of this extension are depicted in the class diagram of Figure 8. Classes shown in grey are part of *CloudSim*, whereas those shown in white are new classes created to enable the integration.

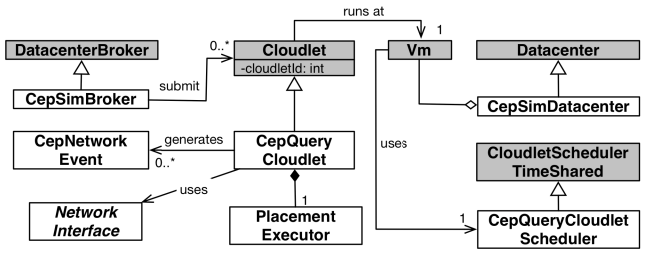


Fig. 8. *CEPSim* integration with *CloudSim*.

A. Standalone queries

First, the *CloudSim* simulator was extended to simulate standalone CEP queries running on a single server. The main part of this extension is the $CepQueryCloudlet$ class, a $Cloudlet$ specialization that encapsulates the CEP placement executor described in Section III-B.

In *CloudSim*, a $Cloudlet$ represents a workload that is submitted and processed by the cloud virtual machines [8]. Normally, a $Cloudlet$ represents a finite computation with a length pre-defined by a fixed number of instructions. It is assumed that these computations are independent and there is no visibility of the $Cloudlet$s internal state other than its expected finish time.

CEP queries, on the other hand, are continuous computations that run indefinitely or for a specific period of time. Moreover, tracking queries internal state during simulation is essential to the analysis of any given CEP system. For example, an input queue that is always full indicates that an operator cannot keep up with the incoming event rate. The $CepQueryCloudlet$ class implements these features, but because it is a $Cloudlet$ specialization it can still be managed by *CloudSim* simulation engine.

During the simulation, a $CepQueryCloudlet$ orchestrates a $PlacementExecutor$ execution by invoking the run lifecycle method at each simulation tick. Both parameters needed by the run method, the number of available CPU instructions and the clock time, are provided by the *CloudSim* simulation engine.

The other main classes created for the integration are:

- $CepSimBroker$: a broker acts as a mediator between SaaS and cloud providers [8]. The $CepSimBroker$ extends the *CloudSim* broker to handle $CepQueryCloudlet$s. It also keeps a mapping of all vertices to the VMs to which they have been allocated.
- $CepSimDatacenter$: this datacentre specialization guarantees that a simulation event is generated at equally spaced intervals. *CloudSim* updates the state of all simulated entities in response to each of these events. Therefore, the shorter the interval, the more accurate the simulation will be, but at a higher computation cost.
- $CepQueryCloudletScheduler$: a scheduler defines how the processing power of a VM is shared among all cloudlets allocated to it [8]. This research extends the time-sharing policy, in which each cloudlet receives a time slice of the VM cores. This policy has been chosen because it is a natural model that reflects current operating system implementations. The extension had to be created because the regular policy handles neither infinite nor duration-based cloudlets.

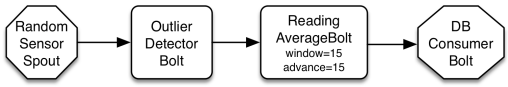


Fig. 9. Storm topology.

B. Networked queries

CloudSim has also been extended to simulate networked (distributed) CEP queries. This extension works as follows:

1) At the end of each $PlacementExecutor$ simulation tick, the $CepQueryCloudlet$ class checks the $Placement$ log for *Sent* log entries. For each entry found, the $CepQueryCloudlet$ invokes the $sendMessage$ method at the $NetworkInterface$.
2) The $sendMessage$ method calculates the delay in sending the events from the target to the destination vertex and schedules a new simulation event at the calculated timestamp. When this simulation event is processed, the sent events are enqueued into the destination vertex input queue.
3) At the next simulation tick, the enqueued events are processed by the corresponding $PlacementExecutor$.

Note that *CloudSim* includes basic network simulation capabilities that are used to calculate the network delay. Nevertheless, more advanced network simulations can be used by creating alternative implementations of *NetworkInterface*.

V. Experiments

This section describes the experiments that have been performed to validate the *CEPSim* simulator. The experiments compare the latency and throughput metrics obtained by running queries on a real CEP / stream processing system (Apache Storm [16]) and by simulating them on *CEPSim*.

Figure 9 shows the Storm query (topology) used in the experiment. A *spout* in the Storm terminology is equivalent to an event producer, whereas a *bolt* is equivalent to an operator. An event consumer in Storm is simply a *bolt* with no outgoing edges. The query in the figure is a use case from Powersmiths' WOW system [24], a sustainability management platform that draws on live measurements of building systems to support energy management.

There are three main steps in this query: the first bolt detects and filters anomalous sensor readings, the second bolt groups readings into windows of 15 seconds and calculates the average, and the final bolt stores the calculated average in a database. The query has been implemented using Java according to the Storm APIs. In addition, it has also been instrumented to obtain reliable latency and throughput measurements.

TABLE II
STORM CLUSTER SPECIFICATION.

#	CPU	Mem.	Description
1	1 core - Intel Xeon E5-2630 2.6 GHz	512 MB	zookeeper
2	1 core - Intel Xeon E5-2630 2.6 GHz	768 MB	nimbus
3	1 core - Intel Xeon E5-2630 2.6 GHz	2048 MB	workers

Table II describes the cluster of virtual machines (VMs) used in the experiments. VMs #1 and #2 run *zookeeper* and *nimbus*, which coordinates the cluster communication and assigns tasks to workers, respectively. The worker VM is the one which effectively executes the queries. All VMs were deployed on the same physical server (12 cores Intel Xeon E5-2630, 2.6Ghz / 96GB RAM).

A. Setup

The first step of the experiment was to implement the Storm query in the *CEPSim* model. The mapping of Storm queries to *CEPSim* is straightforward because both use DAGs as the underlying query model. As an example, Figure 10 depicts an object diagram of the query as implemented in *CEPSim*. Storm's *spout*s and *bolt*s are mapped to $EventProducer$ and $Operator$ instances respectively. More specifically, the second bolt is mapped to an instance of the $WindowedOperator$ class because it uses the rolling window concept.

Each link connecting two objects in Figure 10 represents an edge of the query graph and is annotated with the corresponding edge selectivity. For example, the operator $outlier$ is connected to $avgWindow$ with $selectivity = 0.95$. This selectivity represents the fact that only 95% of the events processed by $outlier$ are sent to $avgWindow$ (the other 5% are anomalies). Finally, each vertex of the graph also contains the ipe attribute, which estimates the number of instructions needed to process an event. These values were obtained beforehand in a separate experiment which estimated the maximum throughput for each Storm operator. Based on estimations of the processor MIPS and the maximum operator throughput, it is possible to roughly calculate the number of instructions required to process each event.

Following, *CloudSim* was used to create a simulation environment as close as possible to the Storm cluster servers. Processor capacity has been estimated as 2500 MIPS. A simulation interval of 10 ms was used to achieve higher precision.

B. Results

Figure 11 summarizes the results of the experiment. The first graph plots the query latency as a function of the number of sensors sending data to the query. The second graph is similar and depicts the query throughput as a function of the number of sensors. $RandomSensorSpout$ from the Storm query was implemented to send 10 readings each second per sensor, of which 5% are anomalies.

The Storm query was run for every number of sensors for 30 minutes. The values shown in the graphics are the average latency (throughput) for the last 20 minutes. *CEPSim* results were simulated only once, because the results are deterministic.

Generally speaking, *CEPSim* achieved very high accuracy for latency and good accuracy for throughput. The latency error was less than 2% up to 1000 sensors, and even though the error was higher at 1750 sensors, *CEPSim* correctly captured the general shape of the curve and the fact that the query started to overload at this point.

CEPSim also accurately calculated the query throughput, achieving 0% error from 1 to 100 sensors and approximately 10% for 500 and 1000 sensors. The throughput accuracy diminished at higher generation rates, but once again *CEPSim* could show the performance degradation at 1750 sensors.

Further analysis concluded that this divergence in metric values has been mainly caused because *CEPSim* servers were not as overloaded as the real ones during the experiments. By changing the operators' ipe attributes it was possible to better approximate the real behaviour. Therefore, this experiment has also showed the importance of a controlled procedure to estimate the ipe values with higher precision. This limitation will be addressed as a future work.

VI. CONCLUSIONS

This article has presented *CEPSim*, a simulator for cloud-based Complex Event Processing systems. *CEPSim* extends *CloudSim*, an existing cloud computing simulator, with a new application model based on DAGs and an engine that can simulate the execution of CEP queries. Moreover, *CEPSim* enables its user to customize the simulation by creating alternative operator placement and scheduling strategies. Experimental results have shown that *CEPSim* can simulate a real system (Apache Storm) with high precision and accuracy.

By using the simulator, system architects and researchers can experiment with different environment configurations and strategies without incurring the costs of running large-scale tests on cloud environments. In addition, we hope that this simulator can encourage and facilitate research in this field.

As future work, we plan to develop alternative load shedding strategies and a more comprehensive set of experiments, including networked queries, scalability tests, comparison with other CEP systems, and experiments in real cloud environments. Finally, a mechanism will also be implemented that enables queries to arrive and leave during the simulation.

ACKNOWLEDGMENT

This research was supported in part by an NSERC CRD at Western University (CRDPJ 453294-13). Additionally, the authors would like to acknowledge the support provided by Powersmiths.

REFERENCES

[1] F. J. Ohlhorst, *Big Data Analytics: Turning Big Data into Big Money*, 1st ed. Hoboken, NJ, USA: Wiley, 2012.

[2] K. Grolinger, M. Hayes, W. A. Higashino, A. L'Heureux, D. S. Allison, and M. A. M. Capretz, "Challenges for MapReduce in Big Data," in *Services, 2014 IEEE World Congress on*, Jun. 2014, pp. 182–189.

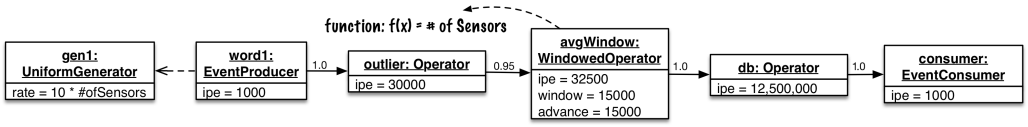
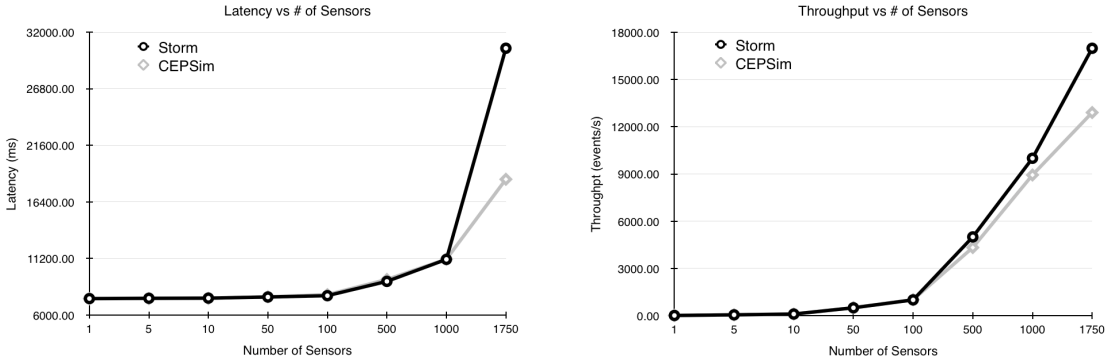


Fig. 10. *CEPSim* object diagram.



(a) Latency. (b) Throughput.

Fig. 11. Experiment results.

[3] W. A. Higashino, C. Eichler, M. A. M. Capretz, T. Monteil, M. B. F. De Toledo, and P. Stolf, "Query Analyzer and Manager for Complex Event Processing as a Service," in *WETICE Conference, 2014 IEEE 23rd International*, Jun. 2014, pp. 107–109.

[4] Z. Qian, Y. He, C. Su, Z. Wu, H. Zhu, T. Zhang, L. Zhou, Y. Yu, and Z. Zhang, "TimeStream: Reliable Stream Computation in the Cloud," in *Proceedings of the 8th ACM European Conference on Computer Systems*. New York, NY, USA: ACM Press, 2013, p. 1.

[5] V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez, C. Soriente, and P. Valduriez, "StreamCloud: An Elastic and Scalable Data Streaming System," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 12, Dec. 2012, pp. 2351–2365.

[6] S. K. Garg and R. Buyya, "NetworkCloudSim: Modelling Parallel Applications in Cloud Simulations," *2011 Fourth IEEE International Conference on Utility and Cloud Computing*, Dec. 2011, pp. 105–113.

[7] D. Kliazovich, P. Bouvry, and S. U. Khan, "GreenCloud: A packet-level simulator of energy-aware cloud computing data centers," *Journal of Supercomputing*, vol. 62, 2012, pp. 1263–1283.

[8] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. De Rose, and R. Buyya, "CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms," *Software: Practice and Experience*, vol. 41, no. 1, Jan. 2011, pp. 23–50.

[9] A. Núñez, J. L. Vázquez-Poletti, A. C. Caminero, G. G. Castañé, J. Carretero, and I. M. Llorente, "ICanCloud: A Flexible and Scalable Cloud Infrastructure Simulator," *Journal of Grid Computing*, vol. 10, 2012, pp. 185–209.

[10] D. Luckham, "Rapide: A Language and Toolset for Simulation of Distributed Systems by Partial Orderings of Events," Stanford University, Tech. Rep., 1996.

[11] ——, *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*, 1st ed. Addison-Wesley Professional, 2002.

[12] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik, "Aurora: a new model and architecture for data stream management," *The VLDB Journal*, vol. 12, no. 2, Aug. 2003, pp. 120–139.

[13] A. Arasu, S. Babu, and J. Widom, "The CQL continuous query language: semantic foundations and query execution," *The VLDB Journal*, vol. 15, no. 2, Jul. 2005, pp. 121–142.

[14] G. Cugola and A. Margara, "Processing flows of information: from data stream to complex event processing," *ACM Computing Surveys*, vol. 44, no. 3, Jun. 2012, pp. 1–62.

[15] D. Luckham and R. Schulte, "Event Processing Glossary - Version 2.0," Event Processing Technical Society, Tech. Rep. July, 2011.

[16] Storm, "Storm, distributed and fault-tolerant realtime computation." [Online]. Available: http://storm-project.net/

[17] M. Hong, M. Riedewald, C. Koch, J. Gehrke, and A. Demers, "Rule-based multi-query optimization," in *Proceedings of the 12th International Conference on Extending Database Technology Advances in Database Technology - EDBT '09*. New York, NY, USA: ACM Press, 2009, p. 120.

[18] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Communications of the ACM*, vol. 51, no. 1, Jan. 2008, p. 107.

[19] A. Brito, A. Martin, T. Knauth, S. Creutz, D. Becker, S. Weigert, and C. Fetzer, "Scalable and Low-Latency Data Processing with Stream MapReduce," *2011 IEEE Third International Conference on Cloud Computing Technology and Science*, Nov. 2011, pp. 48–58.

[20] A. M. Aly, A. Sallam, B. M. Gnanasekaran, L.-V. Nguyen-Dinh, W. G. Aref, M. Ouzzani, and A. Ghafoor, "M3: Stream Processing on Main-Memory MapReduce," in *2012 IEEE 28th International Conference on Data Engineering*. IEEE, Apr. 2012, pp. 1253–1256.

[21] T. Guérout, T. Monteil, G. Da Costa, R. Neves Calheiros, R. Buyya, and M. Alexandru, "Energy-aware simulation with DVFS," *Simulation Modelling Practice and Theory*, vol. 39, Dec. 2013, pp. 76–91.

[22] N. Grozev and R. Buyya, "Performance Modelling and Simulation of Three-Tier Applications in Cloud and Multi-Cloud Environments," *The Computer Journal*, Sep. 2013.

[23] G. T. Lakshmanan, Y. Li, and R. Strom, "Placement Strategies for Internet-Scale Data Stream Systems," *IEEE Internet Computing*, vol. 12, no. 6, Nov. 2008, pp. 50–60.

[24] Powersmiths, "Powersmiths WOW - Build a more sustainable future." [Online]. Available: http://www.powersmithswow.com/