Event Processing across Edge and the Cloud for Internet of Things Applications

Nithyashri Govindarajan[1], Yogesh Simmhan[2], Nitin Jamadagni[3] and Prasant Misra[4]
[1]Birla Institute of Technology and Science, Pilani
[2]Supercomputer Education and Research Centre, Indian Institute of Science, Bangalore
[3]National Institute of Technology, Surathkal
[4]Robert Bosch Centre for Cyber Physical Systems, Indian Institute of Science, Bangalore
f2011746@pilani.bits-pilani.ac.in[1], simmhan@serc.iisc.in[2],
nitin.jamadagni@gmail.com[3], prasant.misra@rbccps.org[4]

ABSTRACT

The rapid growth of sensing devices has opened up complex event processing (CEP) for real-time analytics in Internet of Things (IoT) Applications. While CEP has traditionally been centralized, the increasing capabilities of edge devices like smart phones, and the operational needs of low latency and privacy makes it desirable to use both edge and the Cloud for distributed CEP, the former often serving as event sources. This paper motivates the need for real-time analytics across edge and the Cloud, formalizes an optimization problem for bi-partitioning a CEP query pipeline based on IoT application needs, and proposes an initial solution.

1. INTRODUCTION

The rapid proliferation of sensing elements, both in the physical space (such as RFID tags, sensors, smart phones) and virtual space (crowd-sourced data, social networks, virtual agents) is generating immense volumes of data. Often, these data sources are continuous in nature, contributing to the velocity aspect of Big Data. This is best exemplified by the fast evolving Internet of Things (IoT) domain [1] where 50 billion or more devices are expected to be interconnected through the Internet. Extracting meaningful information from these rapid streams of data requires both existing analytics models, architectures and platforms to be extended, and novel ones to be defined. In this regard, Complex Event Processing (CEP) is a key data processing approach that facilitates easy specification of event patterns, and their fast detection on high traffic event streams, to derive actionable intelligence in real-time [1, 6].

CEP engines accept continuous queries that are defined over event tuples and detect patterns within and across events that arrive on a single or on multiple event streams. They help translate low level information collected from heterogeneous sources at high rates into "complex events" that identify situations of interest [8]. Traditional research and development of CEP engines has been on single-machine, shared memory platforms [5, 7]. There, event sources are often moved centrally into Cloud data centers where integrated processing of CEP queries takes place [2] [3].

In the new generation of IoT applications, the event sources are often on edge devices like wireless sensors, smart power meters, and smart phones, many of which are starting to have non-trivial processing capabilities (e.g., Arduino and Raspberry Pi boards for sensor network gateways, Android Smart Phones). Given the need for rapid decision-making on event pattern detection and the operational constraints , it makes sense to leverage the processing capabilities of the edge devices rather than rely solely on centralized clouds. Such requirements are often seen in healthcare applications and public utilities in Smart Cities [10].

The focus of our work is to enable real-time analytics on distributed systems that span across the edge and the Cloud. As such, there is growing interest on distributed CEP [2]. Like others [3], our aim is to model the distributed query processing of blended streams across dynamic systems, without centralization on the Cloud. Similarly, our goal is to minimize the end-to-end latency for executing the query pipeline by minimizing the overheads in the entire system, and not just on the edge devices or the Cloud.

However, we have several unique aspects. One is the use of a distributed IoT infrastructure with heterogeneous resources with varied computational capabilities for executing CEP pipelines. Secondly, CEP engines available on the diverse resources have varying query expressivity that limits their ability to handle specific query predicates. Lastly, privacy constraints are enforced on event streams. These varied Quality of Service (QoS) requirements and IoT infrastructure conditions are taken into consideration while minimizing the latency for detecting patterns.

In the rest of the paper, we describe an IoT scenario for water analytics and an architecture for distributed CEP across a smart phone platform and the Cloud (§ 2). Further, we motivate the need for intelligent partitioning of CEP query pipelines to meet constraints unique to the IoT domain, and formalize it as an optimization problem (§ 3). We

[1]#IoTH: The Internet of Things and Humans, April 16, 2014, http://radar.oreilly.com/2014/04/ioth-the-internet-of-things-and-humans.html

[2]Amazon AWS Kinesis, http://aws.amazon.com/kinesis/

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
The 20th International Conference on Management of Data (COMAD), 17th-19th Dec 2014 at Hyderabad, India.
Copyright ©2014 Computer Society of India (CSI).

propose initial solutions for this optimization problem, using both brute force and dynamic programming approaches (§ 4), and then discuss future extensions to this work (§ 5).

2. CEP ACROSS EDGE AND THE CLOUD

Our motivating scenario is a Campus Smart Water IoT infrastructure where water level, flow and quality sensors (~ 200) are deployed on water tanks, reservoirs and mainline pipes, and connected to the Internet through Android-based wireless gateway devices and smart phones. The sensors emit events periodically ($\frac{1}{sec} \sim \frac{1}{min}$) and report observations such as the water level in the tank, quality parameters such as total dissolved solids, chlorine content, temperature, and the volume of water flow. Analytics using CEP queries are defined usually as a dataflow pipeline where data pre-processing and quality checks precede aggregations or detection of interesting situations, e.g., water overflow and underflow due to pump operations, leakage of water due to input-output flow imbalance, presence of contaminants, and excess usage relative to campus average. Then, notifications are send to operations managers or water consumers for sustainable water usage.

Given the current scale, the CEP queries can be processed either on the edge devices (here-on called *edge*) or centrally on the Cloud without significant performance impact. But as we scale the system, e.g., to a town or city wide deployment, or by including more number and types of sensors, such as air quality and electricity usage, there is a need to intelligently partition the CEP query execution across the edge and the Cloud to meet these requirements. As we consider location-based sensing of consumer activity, data ownership and security become concerns, and data generators may impose restrictions on moving raw data to the Cloud.

As a proof of concept of executing CEP pipelines across edge and the Cloud, we implement a light-weight CEP engine, *CEPLite*, on the Android platform, complemented by a full-featured *Siddhi* CEP engine on the Cloud [9]. Incoming sensor events on the edge device (Android) are first categorized based on their sensor source and placed into distinct input queues of an EventBus [3] Publish-Subscribe (PubSub) system. Subscribers, in this case the CEPLite engine, register a subscription with the PubSub broker and are notified asynchronously when events arrive. The subscribed events are placed in input streams from which CEPLite Processing Units (PU) poll for events. Each PU is responsible for one query submitted to CEPLite, and operates over specific input event streams in a SIMD manner. The PU's query capability is limited to filter, sequence, count windows and simple aggregation, and it forwards the generated output (complex) events over a stream to the next PU or the Cloud having the next CEP query in the pipeline for processing.

Users submit their CEP query pipeline as a dataflow graph (Fig. 1). A query planner distributes this pipeline across the edge and the Cloud based on several factors, as discussed in § 3. These query segments are registered with CEPLite on the edge device, and Siddhi on the Cloud. The communication between the edge device and the Cloud is done using a REST or a CoAP [4] service on the Cloud. CoAP is intended for resource-constrained IoT devices to communicate over the Internet with low overhead, using UDP and optional multicast, and allows integration with traditional HTTP web services.

[3]http://greenrobot.github.io/EventBus/

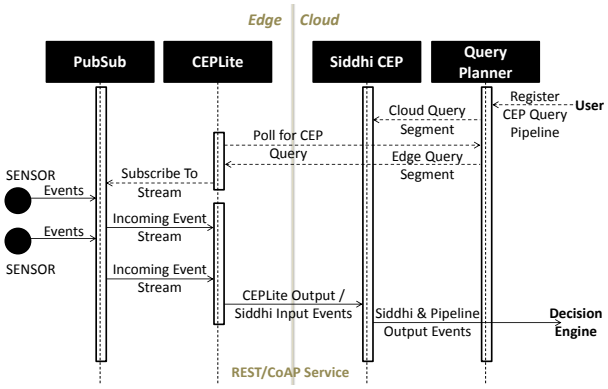

Figure 1: Sequence diagram of interactions between edge and Cloud

3. PROBLEM FORMALIZATION

Preliminaries. The pipeline of queries executing over event streams is represented as a *directed acyclic graph (DAG)* of vertices and edges: $\mathcal{G} = \langle \mathbb{Q}, \mathbb{S} \rangle$. $\mathbb{Q} = \{q_i\}$ is the set of *CEP queries* that are the vertices of the DAG, and $\mathbb{S} = \{s_i \mid s_i = \langle q_j, q_k \rangle,\ q_i, q_j \in \mathbb{Q}\}$, is the set *event streams* that connect the output of query q_i to the input on the next query q_j, and form the *directed* edges of the DAG. Edges indicate the data dependency between the queries, and events produced from the output of a query pass along the stream as input to the next query.

The queries that receive the initial input event streams into the DAG are called the *source queries*, and are characterized by having no incoming edges (though implicitly, an edge indicating the input stream is incident on the queries). The set of source queries is given by:

$$\mathbb{Q}_{\succ} = \{q_{\succ} \mid \nexists\ s_j = \langle q_i, q_{\succ} \rangle \in \mathbb{S}\ \forall\ q_i \in \mathbb{Q}\}$$

The queries that emit the final output event streams from the DAG are called the *sink queries*, and these have no explicit outgoing edges. The set of sink queries is given by:

$$\mathbb{Q}_{\prec} = \{q_{\prec} \mid \nexists\ s_j = \langle q_{\prec}, q_i \rangle \in \mathbb{S}\ \forall\ q_i \in \mathbb{Q}\}$$

For simplicity, we assume in this paper that there can be multiple source queries, i.e., $|\mathbb{Q}_{\succ}| \geq 1$, but there is only one sink query so that the optimization goals of the pipeline are scoped to outputs from this sink.

We define *paths* in the DAG which represent the logical flow of the events through the streams, from the source queries to a unique sink query. Since we assume a single sink query, we denote its path as p. This path offers an alternative, edge-centric view of the DAG. The following production rules generate the path from *path segments*, where $s_\star \in \mathbb{S}$ and the '$\multimap$' symbol indicates a query that connects streams on either of its sides. The path for the sink query is also the *maximal unique path* for the DAG – it has every stream in the DAG appearing exactly once (maximal) and we assert a canonical ordering for comma-separated parallel streams (unique). By definition, the path is acyclic for a DAG.

$$\begin{aligned} Source &: \varnothing \multimap s_i \\ Sink &: s_i \multimap \varnothing \\ Sequential &: s_i \multimap s_j \\ Parallel &: Predecessor \multimap (Segment, Segment+) \end{aligned}$$

$$
\begin{aligned}
&\multimap Successor\\
Predecessor &: Source \mid Sequential \mid Parallel \mid \varnothing\\
Successor &: Seqential \mid Parallel \mid Sink\\
Segment &: Seqential \mid Parallel\\
p &= \underset{Segment}{\arg\max} \big(\textsc{Length}(Segment)\big)
\end{aligned}
$$

Queries execute on specific *computing resources* and the set of resources available within the IoT infrastructure is given by $\mathbb{R} = \{r_i\}$. We consider two classes of computing resources, *edge* devices such as smart phones and micro-controllers and *Cloud* virtual machines (VMs), each with a specific computing capability as defined later. The mutually exclusive set of edge devices and Cloud VMs are given by $\mathbb{R}^E$ and $\mathbb{R}^C$, respectively. Thus $\mathbb{R} = \mathbb{R}^E \cup \mathbb{R}^C$ and $\mathbb{R}^E \cap \mathbb{R}^C = \varnothing$. A *resource mapping* function indicates the resource on which a query executes:

$$\mathcal{M} : \mathbb{Q} \to \mathbb{R}$$

We define a *privacy function*:

$$\mathcal{P} : \langle \mathbb{S}, \mathbb{R} \rangle \to \{True, False\}$$

that determines if a stream is allowed on a resource. If the privacy for a stream $s_i = \langle q_j, q_k \rangle$ on resource r_l is $False$, this means that neither of the queries q_j or q_k, emitting or consuming the stream, can be executed on resource r_l, i.e.,

$$\mathcal{P}(s_i\langle q_j, q_k\rangle, r_l) = False \implies \mathcal{M}(q_j) \neq r_l \;\&\; \mathcal{M}(q_k) \neq r_l$$

Further we define $\mathbb{A} \subseteq \mathbb{Q}$ as the set of *anonymization queries* such that after a stream is processed by one of them, successive streams along the path have been anonymized and the privacy of such streams on any resource is $True$. For a stream $s_k = \langle q_l, q_m \rangle$ in the path p such that $q_m \in \mathbb{A}$, then $\forall s_j$ *downstream of* $s_k, \mathcal{P}(s_j, \star) = True$, i.e., the streams after the current one up to the sink have been anonymized and their privacy on any resource is $True$.

A *selectivity* function is associated with each query of the pipeline as a statistical measure of the fraction of incoming events satisfying the query and generated as output events. The *selectivity* is denoted by $\gamma(q)$ and is the average number of output events generated per input event. Output events generated by a query are duplicated on all output streams.

The *stream rate* Ω defines the number of events per unit time on a stream. The *incoming stream rate* entering each input query of the pipeline is denoted by $\Omega_{\succ}$. Knowing the incoming stream rates and the selectivity per query in the pipeline, the rate at its output stream s_j generated by query q can be calculated recursively as $\Omega(q) = \gamma(q) \times \sum_i \Omega(s_i)$, where s_i are the input streams to query q. This can then be used to estimate the *outgoing stream rate*, $\Omega_{\prec}$, from the output query for the entire pipeline.

Latency $\mathcal{L}$ is defined as the time for an event to move from one resource to another, over a stream connecting queries present on those resources. Latency is taken to be a constant value for any two resource-pairs at a given maximum event rate Ω^L_{max}. Latency, rather than bandwidth, is often the limiting factor for event processing on commodity networks, and once bandwidth starts becoming the bottleneck, we assume that the pipeline cannot scale on those resources, and, for simplicity, set $\mathcal{L} \to \infty$. Formally:

$$
\begin{aligned}
&\mathcal{L}^{\Omega}(r_i, r_j) = x \mid x \in \mathbf{R}^+ \cup 0 && \text{if } \Omega \leq \Omega^L_{max} \;\&\; i \neq j\\
&\mathcal{L}^{\Omega}(r_i, r_j) = \infty && \text{if } \Omega > \Omega^L_{max} \;\&\; i \neq j\\
&\mathcal{L}^{\Omega}(r_i, r_j) = 0 && \text{if } i = j
\end{aligned}
$$

where $\mathbf{R}^+$ is the set of positive Real Numbers and $r_i, r_j \in \mathbb{R}$.

CEP engine implementations may be limited by the resource platform. If a CEP query (i.e., all its predicates) can be evaluated by an engine on a resource, then the resource's *expressivity* is said to exist for the query. If a query is not supported on a resource, then the expressivity is absent.

Like latency, the *compute time* $\mathcal{C}$ for a query is a constant for a given query q_i on a resource r_j for all $\Omega \leq \Omega^C_{max}$. This max value can be empirically obtained. Beyond this threshold, the compute resources get saturated and the query cannot scale. Since events arrive continuously, the pipeline is unsustainable once saturated. Expressivity of a resource is related to the computational time of the query, since for a query that cannot be express on a resource, its $\mathcal{C} \to \infty$.

$$
\begin{aligned}
&\mathcal{C}^{\Omega}(q_i, r_j) = x \mid x \in \mathbf{R}^+ && \text{if } \Omega \leq \Omega^C_{max} \;\&\; q_i \text{ is expressive on } r_j\\
&\mathcal{C}^{\Omega}(q_i, r_j) = \infty && \text{if } \Omega > \Omega^C_{max} \;\&\; q_i \text{ is expressive on } r_j\\
&\mathcal{C}^{\Omega}(q_i, r_j) = \infty && \text{if } q_i \text{ is } \textit{inexpressive} \text{ on } r_j
\end{aligned}
$$

The total *makespan* $\mathcal{T}$ for the path in the pipeline is the time taken to generate a single event from the output query after processing input events by each query on the resource it is mapped to, and transferring events between resources. For a mapping $\mathcal{M}$ and input rate $\Omega_{\succ}$, the makespan for path p is computed using its rate, latency and compute functions:

$$\mathcal{T} = \sum_{q_i \in p} \mathcal{C}^{\Omega(q_i)}\big(q_i, \mathcal{M}(q_j)\big) + \sum_{q_i \in p} \mathcal{L}^{\Omega(q_i)}\big(\mathcal{M}(q_i), \mathcal{M}(q_j)\big)$$

As discussed before, edge devices are often collocated with the incoming event sources, while the eventual decision making happens on the Cloud. An *edge cut* is a bi-partition of the path into two disjoint path segments. It denotes the logical partitioning of the queries between edge and Cloud resources. Queries in the first path segment execute on the edge, queries in the other path segment execute on the Cloud, and stream(s) at the split move events across these resources. The mapping $\mathcal{M}$ captures this. For edge cuts to form a consistent bi-partition, they should appear on a *Sequential*, *Source*, or *Sink* Segment, or on each sequential segment present in a *Parallel Segment*, recursively.

Optimization Problem Definition. *Given* the query pipeline and its path, the selectivity and anonymization of each query, the resources in the IoT system, the compute time on each resource for a query, the latency between resources, and a given incoming event rate, the optimization problem is to find a edge cut that consistently bi-partitions the path and maps it across different resources such that the makespan time for the pipeline is *minimized*, while ensuring that the privacy *constraint* is satisfied.

$$
\begin{aligned}
\text{Given: } &\mathbb{G}, p, \gamma, \mathbb{A}, \mathbb{R}, \mathcal{C}, \mathcal{L}, \Omega_{\succ}\\
\text{Minimize: } &\mathcal{T}\\
\text{Subject to Constraint: } &\mathcal{P}\\
\text{Output: } &\mathcal{M} : \mathbb{Q} \to \mathbb{R}
\end{aligned}
$$

In the Campus Smart Water project, a sample CEP pipeline performs three queries: data cleaning, temporal aggregation, and detecting water overflow. This forms the dataflow graph, $\mathbb{G}$. The resource r_e is the Android board reporting the water level observations, $\mathcal{L}$ is the network latency in moving this data stream to the Cloud r_c for processing.

Algorithm 1 SplitBruteForce

```
1: procedure SplitBF(EdgeCut EdgeCuts[ ], p, Ω≻)
2:     T_min = ∞, LHS = 0, RHS = 0
3:     for s_x ∈ EdgeCuts[ ] do                ▶ Test each cut
4:                 ▶ Calculate LHS and RHS costs for cut
5:         for s_k = ⟨∅, q≻⟩ to s_x | {s_k = ⟨q_i, q_j⟩ ∈ p} do
6:             LHS = LHS + L^Ω(q_i)(r_e, r_e) + C(q_j, r_e)
7:         end for
8:         for s_k = s_{x+1} to ⟨q≺, ∅⟩ | {s_k = ⟨q_i, q_j⟩ ∈ p} do
9:             RHS = RHS + L^Ω(q_i)(r_c, r_c) + C(q_j, r_c)
10:        end for
11:        T = LHS + RHS + Σ_⟨q_x,⋆⟩∈s_x L^Ω(q_x)(r_e, r_c)
12:        if T < T_min then                   ▶ Found a better cut?
13:            T_min = T, Split = s_x
14:        end if
15:    end for
16:    return ⟨Split, T_min⟩
17: end procedure
```

Algorithm 2 SplitDynamicProgramming

```
1: procedure SplitDP(EdgeCut EdgeCuts[ ], p, Ω)
2:     n = SizeOf(EdgeCuts), T[ ] = float[1..n]
3:     T[1] = Σ_{q_i∈p} C(q_i, r_c)            ▶ Run fully on Cloud
4:     prev = 1
5:     Define Update(prev) : T[i] = T[prev] − C(q_j, r_c)+
       C(q_j, r_e)+L^Ω(q_j)(r_e, r_c)          ▶ Incremental update
6:     for i = 1 to n do
7:         s_i = ⟨q_j, q_k⟩ ∈ EdgeCuts[ ]
8:         if IsSeqSeg(s_i) then               ▶ Cut Sequential
9:             Update(prev)                    ▶ Move q_j to edge
10:        else                                ▶ Cut Parallel
11:            if s_i and s_p rev have same parent stream then
12:                Update(prev)
13:            else
14:                ancestor = Find(EdgeCuts,
                       Parent stream sharing suffix edge cuts)
15:                Update(ancestor)
16:            end if
17:        end if
18:        prev = i
19:    end for
20:    return ⟨EdgeCuts[IndexOf(Min(T))], Min(T)⟩
21: end procedure
```

4. SOLUTION APPROACH

Edge cuts are locations in the path where the queries can be split between the edge and the Cloud resources. The EdgeCutDetection algorithm (omitted for brevity) scans every stream in the path and decide if it is a candidate for a split. It recursively identifies consistent edge cut candidates in the path, starting with a cut that can happen prior to the source queries (i.e., the entire pipeline is executed fully on the Cloud) to a cut after the sink query (pipeline runs entirely on the edge). The algorithm further limits cuts to those that satisfy the privacy constraints on the streams, by forcing anonymized queries to be on the edge (left hand side, or LHS) or the Cloud (RHS) as needed. Each edge cut includes one or more streams, and the cuts are stored in *EdgeCuts*[ ], an array of linked lists, when every item in the linked list represents a stream in the edge cut. If the linked lists has only one item, it is a cut of a sequential path segment, and multiple items indicate cuts across parallel segments, one item per parallel segment.

We propose two algorithms SplitBruteForce (Alg. 1) and SplitDP (Alg. 2) to obtain the optimal split that minimizes the *makespan*. SplitBruteForce computes $\mathcal{T}$ at all candidate edge cuts by calculating each of their compute times on the edge and the Cloud, and the latency to transfer output events from the edge to the Cloud. It then picks the cut with the least $\mathcal{T}$. While tractable for short pipelines, its time complexity is $O(|\mathbb{S}|^2)$ for even sequential pipelines.

SplitDP uses *Dynamic Programming* to compute the $\mathcal{T}$ assuming the pipeline fully runs in the Cloud, and incrementally moves one query at a time to the edge. It uses memoize to store previously computed values of $\mathcal{T}$ for edge cuts, and reuses it to truncate the computation for incrementally longer cuts. This has a time complexity of $O(|\mathbb{S}|)$.

5. FUTURE WORK

As part of future work, we propose to extend the formalization by allowing multiple edge devices and Cloud VMs for a single pipeline, and using an edge device's power level and efficiency as a QoS factor. We can also introduce *ad hoc* anonymization queries for better privacy control, and allow fine-grained partitioning of a single CEP query rather than between CEP queries. We will also explore additional solutions to the optimization problem, including those that allow dynamic repartitioning based on changing conditions and user needs. The proposed solutions need to be evaluated for practical feasibility and scalability through empirical benchmarks.

6. REFERENCES

[1] A. Adi, D. Botzer et al. Complex event processing for financial services. In *IEEE Services Computing Workshops*, 2006.

[2] M. Akdere, U. Çetintemel, and N. Tatbul. Plan-based complex event detection across distributed sources. *Proc. VLDB Endow.*, 1(1):66–77, Aug. 2008.

[3] W. Z. B. Chandramouli, S. Nath. Supporting distributed feed-following apps over edge devices. *PVLDB*, 6(13), 2014.

[4] C. Bormann, A. P. Castellani, and Z. Shelby. Coap: An application protocol for billions of tiny internet nodes. *IEEE Internet Computing*, 2012.

[5] G. Cugola and A. Margara. Complex event processing with T-REX. *Journal of Systems and Software*, 85(8):1709–1728, 2012.

[6] G. Cugola and A. Margara. Processing flows of information: From data stream to complex event processing. *ACM Computing Surveys*, 44(3), June 2012.

[7] A. Demers, J. Gehrke, and P. Biswanath. Cayuga: A general purpose event monitoring system. In *CIDR*, pages 412–422, 2007.

[8] R. Mayer. Real-time distributed complex event processing for big data scenarios. In *Distributed Event-Based Systems (DEBS) Ph.D. Forum*, 2013.

[9] S. Suhothayan, K. Gajasinghe et al. Siddhi: A second look at complex event processing architectures. In *ACM GCE Workshop*, 2011. http://siddhi.sourceforge.net.

[10] Y. Simmhan, S. Aman et al. Cloud-based software platform for data-driven smart grid management. *Computing in Science and Engineering*, 2013.