StreamSight: A Query-Driven Framework for Streaming Analytics in Edge Computing

Zacharias Georgiou, Moysis Symeonides, Demetris Trihinas, George Pallis, Marios D. Dikaiakos

Department of Computer Science
University of Cyprus
Email: { zgeorg03, msymeo03, trihinas, gpallis, mdd }@cs.ucy.ac.cy

***Abstract*—Edge computing is the emerging architectural paradigm extending cloud technologies to the logical extremes of the network for on-demand and delay-sensitive services. However, once service placement on edge-enabling resources has been dealt with, a new challenge arises: how to process enormous volumes of streaming data to provide query-driven analytics while still satisfying the delay-critical servicing requirements. To overcome this challenge we introduce StreamSight, a framework for edge-enabled IoT services which provides a rich and declarative query model abstraction for expressing complex analytics on monitoring data streams and then dynamically compiling these queries into stream processing jobs for continuous execution on distributed processing engines. To overcome the resource restrictive barriers in edge computing deployments, StreamSight outputs the query execution plan so that intermediate results are reused and not continuously recomputed. In turn, StreamSight enables users to express various optimization strategies (e.g., approximate answers, query prioritization) and constraints (e.g., sample size, error-bounds) so that delay-sensitive requirements relevant to the deployment are not violated. We evaluate our framework on Apache Spark with real-world workloads and show that leveraging StreamSight can significantly increase performance by at least $4\times$ while still satisfying all accuracy guarantees.**

***Index Terms*—Edge Computing, Cloud Computing, Big Data, Stream Processing, Query Execution**

I. Introduction

With the prevalence of the Internet of Things (IoT) as the dominating technology to monitor and understand the physical world, inevitably both the number of internet-enabled "things" and the amount of IoT data are exploding. Reports put the number of connected devices surpassing 20 Billion by 2020 while these devices will produce more than 500 ZB of data [1]. In respect to this, IoT services naturally offload intensive analytic jobs to the cloud, as the computing power of the cloud outclasses the resource capabilities of mobile and remote IoT devices [2]. However, with the growing volume of data generated at the logical extremes of the network, and the fact that network bandwidth is simply not scaling at the same speed as with computing power, data mitigation is becoming a bottleneck constraining the cloud computing paradigm for delay-sensitive IoT services [3]. Thus, it seems only inevitable that data must be processed at the edge of the network for shorter response times, more efficient processing and significantly less network pressure.

Edge computing refers to the enabling technologies allowing computation to be performed at the logical extremes of the network, such as on downstream data, on behalf of cloud services, and upstream data, on behalf of IoT services [4]. The rationale of edge computing is that computing should happen at the proximity of the data source with the "edge" constituting any computing and network resources along the path between data sources and the cloud. In this context, sensory data are converted from raw signals to contextually relevant information in proximity of the data source. To this end, a series of frameworks, such as Hadoop, Apache Spark and Storm, have contributed to "democratizing" big data analytics by hiding most of the complexity related to machine communication, scheduling, and fault tolerance [5]. Thus, as the compute capabilities of edge servers scale [6], IoT service providers are now embracing big data frameworks to extract analytic insights from edge environments although these frameworks are designed for homogeneous machine clusters usually found in the cloud rather than in edge realms [7].

However, queries on IoT data usually have near real-time requirements that need the latency between data generation and query response to be bounded [8]. Once service placement on edge-enabling resources has been dealt with [9] [10], a new challenge arises: *How to process enormous volumes of streaming data to provide query-driven analytic insights while also minimizing response times?* Naively deploying distributed processing engines without acknowledging the unique characteristics of the "edge" is inefficient and can be error-prone. For instance, IoT settings usually have high load influxes (e.g., road congestion and vehicle safety surveillance). In turn, edge servers can be geographically distributed across wide areas of coverage. This means a huge communication penalty for data exchange to produce intermediate results required for continuously executing complex analytic queries. Consider also the case where queries are not envisioned beforehand, but are rather exploratory and submitted ad-hoc by platform operators (e.g., one may ask what is the difference in current traffic flow for a city segment compared to an hour ago). However, this implies specific knowledge of the programming model of the underlying processing engine and (usually) requires multiple lines of code just for a single query (e.g., 26 lines of code in Spark for the above query). Therefore, the implementation of abstractions, which are able to model knowledge extraction from data streams supporting a wide range of exploratory queries, is still an open research challenge [11] [12].

The focal point of our work is to address these limitations by introducing StreamSight: a practical framework for monitored IoT services aimed at simplifying the specification, compilation and execution of analytic queries on distributed processing engines deployed in edge computing environments.

The main contributions of this work are:

- A declarative query modeling framework, called StreamSight, which abstracts complex analytic insights definition from the compute capabilities of distributed stream processing engines. Thanks to StreamSight, data scientists and platform operators can compose complex analytic queries without any knowledge of the programmable model of the underlying distributed processing engine by solely using high-level directives and expressions. In turn, analytic insight descriptions can also be reused without any alterations and executed on multiple and different underlying processing engines.
- A compilation unit for query execution that transparently maps and compiles the modeled analytic insights descriptions into stream processing jobs for execution on the underlying distributed processing engine. To make efficient use of available resources, the query execution has been optimized so that results are output in time on the (limited) resources provided to the edge computing setting. To this end, the compilation unit creates a query execution plan that allows the reuse of intermediate results, avoiding continuous re-computations, while users are allowed to: (i) prioritize query execution so results of high importance are always output in time (e.g., high load influx), while low priority insights are scheduled when resources are available; (ii) request query enforcement on a sample of the data stream for indicative but in time responses; and (iii) define constraints such as the maximum tolerable upper error bounds for approximate query responses.
- A reference implementation for edge computing settings integrating Apache Spark as the underlying distributed processing engine. To illustrate the effectiveness of our framework, we introduce a thorough evaluation using real-world data and actual queries of interest from Intelligent Transportation Services and Vehicular Networks. Our results are reproducible, the reference implementation is available online[1], while users can take advantage of our Docker Compose extension to deploy the reference implementation and auto-submit their analytic queries without any infrastructure re-engineering.

The rest of the paper is structured as follows: Section II presents the related work. Section III introduces StreamSight, and Section IV describes the algorithmic process. Section V presents our evaluation and Section VI concludes the paper.

II. Related Work

In this Section, we review frameworks for geo-distributed data processing, edge analytics and declarative query modeling. To the best of our knowledge, other than StreamSight, no

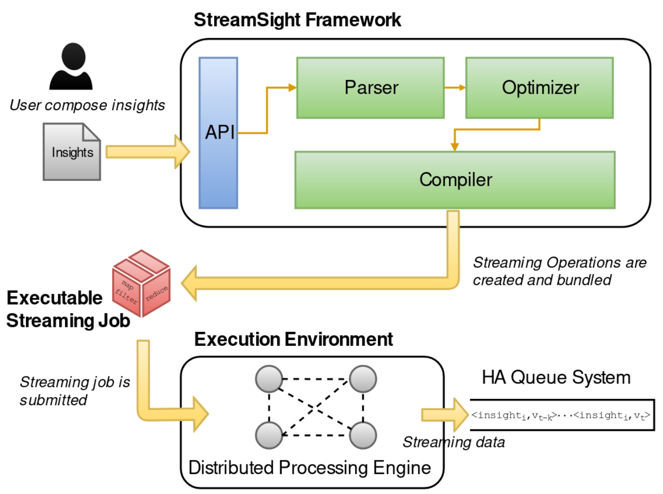

Fig. 1: A High-Level Overview of StreamSight Framework

framework existing incorporates a declarative query interface for modeling, producing and optimizing execution of edge analytics across geo-distributed areas of coverage.

Viswanathan et al. introduce Clarinet [13], a framework which attempts to improve the performance in computing analytic insights formed as SQL queries in geo-distributed data centers. In a similar fashion, Hsieh et al. have developed Gaia [14], a framework which significantly reduces the communication overhead when introducing machine learning queries across multiple data centers. However, both frameworks assume that the computation context of the queries follows a batch processing model and are not tailored to the unique characteristics implied in an edge computing realm.

Gupta et al. [15] introduce Sonata, a framework offering scalable analytic insights for network telemetry. Specifically, network operators express analytic queries through a declarative interface with Sonata making use of the programmable data-plane of network switches for query preprocessing and Spark for query execution. However, Sonata is developed solely for packet-level network telemetry analytics. In turn, Trihinas et al. [16] introduce ADMin, a framework that dynamically adjusts the rate edge devices disseminate data streams for analytics by suppressing data that can be inferred with an estimation model capturing the runtime behavior of the data stream. However, the adaptive strategy is enforced independently per stream although queries may require data from multiple streams. More relevant, but lacking a query expression and compilation model, is the approach followed by Jonathan et al. [17]. In particular, the authors suggest a locality-aware technique for edge servers sharing workloads to maintain the low latency requirements by distributing the load among each server. In turn, Yang et al. [18] introduce a framework for designing fog applications which allows users to define streaming jobs with geographic load partitioning.

In contrast, Satyanarayanan et al. [19] introduce GigaSight, a platform for fog computing analytics. In turn, Cao et al. [20] introduce Codiac, an edge analytics platform that provides data stream pre-processing, summarization and semantic event an-

[1]https://github.com/UCY-LINC-LAB/StreamSight

notation for intelligent transportation services embracing edge servers running Cisco IOx virtualization. However, GigaSight is limited to privacy preserving and video stream processing while Codiac solely targets vehicle traffic analytics. Interestingly, Wen et al. introduce ApproxIoT [21], a data analytics system for approximate computing in IoT services by providing online and hierarchical sampling so that edge resources are exploited to produce approximate outputs with rigorous error bounds. However, the query model is tightly coupled to the Apache Kafka domain specification language and holistic queries (e.g., top-k, kNN) are not supported as approximation can only be exploited for linear queries. Nonetheless, in StreamSight we adopt the hierarchical weighted sampling algorithm introduced by the authors and extend it to support our wide spectrum of supported analytic queries.

III. The StreamSight Framework

A. System Overview

StreamSight supports users in composing analytic queries that are automatically translated and mapped to streaming operations suitable for running on distributed processing engines deployed in wide areas of coverage. This aids both advanced and inexperienced users to abstract and rigorously express complex analytics operations over streaming data, along with query constraints such as sample size and upper error-bounds for query execution to output approximate and in time answers. To date, a user is bounded to follow the procedural programming paradigm made available by the distributed processing engine to specify the sequence of the streaming operations needed to compose the desired query. Despite the flexibility and control offered by this paradigm, the increased complexity can cause unnecessary pains, shifting user attention away from the actual purpose of analytic insight description. Thus, StreamSight adopts a declarative programming paradigm, allowing users to describe analytic insights through a simple and powerful query modeling language.

Figure 1 depicts a high-level and abstract overview of the StreamSight Framework. Users submit, via the **StreamSight API**, ad-hoc queries following the declarative query model which is decoupled from the underlying processing engine. To ease understanding, let us consider a scenario from the intelligent transportation domain. Buses are one of the most common means of transportation and by monitoring bus activity, traffic operators are able to determine road traffic conditions and early detect unfortunate incidents [22]. To this end, buses are equipped with sensors capable of continuously transmitting their location, speed, operating city segment and delay from next stop. For instance, an ad-hoc query that can be submitted to StreamSight by a traffic operator to detect the traffic congestion in each city segment is the following:

```
COMPUTE  ARITHMETIC_MEAN(bus_delay,10 MINUTES)
BY city_segment EVERY 5 SECONDS
```

Insight 1: Computes for a 10min sliding window the average bus delay per city segment with new datapoints considered every 5s.

Having provided the aforementioned query, the **Parser** component immediately decomposes it to validate syntactic correctness, while also detecting circular and antagonizing expressions which result in error-prone and exhaustive query execution. If no errors are detected, the decomposed query is given to the **Optimizer**. This module performs two types of optimizations so as to improve runtime execution: (i) *system-based*, which are automatically applied to detect phases in the stream operation pipeline where intermediate results can be shared instead of recomputed; (ii) *user-defined*, using the *salience*, *sampling*, *error* and *confidence* directives, which are used to output answers in-time. These directives are described in detail in Section IV. For thoroughness, we extend the previous query so that approximate answers are provided to significantly improve response time by executing the query on a sample of the data stream:

```
COMPUTE ARITHMETIC_MEAN(bus_delay,10 MINUTES)
BY city_segment EVERY 5 SECONDS
WITH MAX_ERROR 0.05 AND CONFIDENCE 0.95
```

Insight 2: Computes the average bus delay per city segment on a sample of data with the relative error upper-bounded at 5% in a 95% confidence interval.

The last step is the compilation process, where the **Compiler** recursively traverses the decomposed query structure and maps each expression into the corresponding streaming operations (e.g., map, reduce, filter) of the underlying distributed processing engine. The output is a packaged stream processing job, which contains the pipeline of streaming operations that were automatically generated by the Compiler. When this job is submitted, the distributed processing engine schedules the optimized query execution plan for the required streaming operations on the underlying available resources.

B. Query Model

The StreamSight query model offers users the ability to construct *insights*, namely, new high-level and complex analytics out of raw metric streams. Thus, at the basis, *an insight can be seen as another metric stream that is produced from the composition, transformation and aggregation of multiple metric streams*. Grammar 1 presents in EBNF format the expressivity of the query model language syntax.

An insight at its core is composed by: (i) a **`COMPUTE`** statement applied to a composite expression; (ii) an **`EVERY`** statement denoting the time interval at which the composite is periodically evaluated; and (iii) an optional **`WITH`** statement capturing an **`AND`**-separated list of user-defined optimizations and constraints. A `CompositeExpr` is either a simple expression, denoted as `Expr`, or is recursively constructed via left and right-hand composite expressions operated by a binary operation (e.g., `ADD`, `DIV`). An `Expr` can be an `Aggregate` function (e.g., `MEDIAN(bus_delay)`), a `MetricStream` or a `Number`. Optionally, a `Filter` can be attached to an `Expr` so that left-hand operations are only processed if the filter predicate evaluates to true. As such, a `Filter` is composed from applying on a metric a relational operation and a `CompositeExpr`, with users also able to concatenate

```
⟨Insight⟩          ::= COMPUTE ⟨Composition⟩ EVERY ⟨Interval⟩ [ WITH
                       ⟨Optimizations⟩ ]
⟨Composition⟩      ::= ⟨CompositeExpr⟩ [ BY ⟨Metric⟩ ]
⟨CompositeExpr⟩    ::= ( ⟨CompositeExpr⟩ ⟨BinOp⟩ ⟨CompositeExpr⟩ )
                     | ⟨Expr⟩ | ⟨MapOp⟩ ( ⟨Expr⟩ )
⟨BinOp⟩            ::= ADD | MUL | SUB | DIV
⟨Expr⟩             ::= ⟨Aggregate⟩ [ WHEN ⟨Filters⟩ ] | ⟨MetricStream⟩ |
                       ⟨Number⟩
⟨Aggregate⟩        ::= ⟨WindowedFunc⟩ ( ⟨MetricStream⟩, ⟨Window⟩ )
                     | ⟨AccumFunc⟩ ( ⟨MetricStream⟩ )
⟨WindowedFunc⟩     ::= SUM | COUNT | PRODUCT | ARITHMETIC_MEAN
                       | GEOMETRIC_MEAN | MIN | MAX | VARIANCE |
                       SDEV | MEDIAN | MODE | PERCENTILE[⟨Percent⟩] |
                       TOP_K⟨PositiveInt⟩|
⟨AccumFunc⟩        ::= RUNNING_SDEV | RUNNING_MEAN |
                       RUNNING_MAX | RUNNING_MIN | EWMA[⟨Percent⟩]
                       | PEWMA[⟨Percent⟩]
⟨MetricStream⟩     ::= ⟨Metrics⟩ [ FROM ⟨Membership⟩ ] [WHEN ⟨Filters⟩]
⟨Metrics⟩          ::= ⟨Metric⟩ { , ⟨Metric⟩ }
⟨Membership⟩       ::= ( ⟨MemberID⟩ {, ⟨MemberID⟩ } )
⟨MemberID⟩         ::= ⟨String⟩
⟨Filters⟩          ::= ⟨Filter⟩ { AND ⟨Filter⟩ }
⟨Filter⟩           ::= ⟨Metric⟩ ⟨RelOp⟩ ⟨CompositeExpr⟩
⟨RelOp⟩            ::= '>' | '>=' | '==' | '!=' | '<' | '<='
⟨Metric⟩           ::= ⟨String⟩
⟨Window⟩           ::= ⟨TimePeriod⟩
⟨Interval⟩         ::= ⟨TimePeriod⟩
⟨TimePeriod⟩       ::= ⟨PositiveInt⟩ ⟨TimeUnit⟩
⟨TimeUnit⟩         ::= MILLIS | SECONDS | MINUTES | HOURS
⟨Optimizations⟩    ::= [SALIENCE⟨PositiveInt⟩] [( MAX_ERROR⟨Percent⟩
                       AND CONFIDENCE⟨Percent⟩ | SAMPLE⟨Percent⟩ )]
⟨MapOp⟩            ::= ABS | SQR | SQRT
```

Grammar 1: StreamSight Query Model Language Syntax in EBNF

multiple filters with the `AND` logical operator. In turn, an `Aggregate` is either window-based or accumulated. The difference is in the application of the aggregate on the metric stream. For window-based aggregates, a `window` is needed to denote the time interval of interest for aggregating values, while for accumulated aggregates the result is computed solely based on previous values. Table I provides a description of the currently supported windowed functions, while Table II describes the accumulative operations.

Both windowed and accumulated functions require as input a `MetricStream`, which is composed from a list of `Metrics` and optionally a `Membership` description. As such, other than the bus delay and city segment metrics, the bus stream of our example can emit more metrics to the processing engine to be used as query variables. In turn, the membership description is used when metrics are compiled as aggregates from multiple monitoring sources. When the membership is omitted, measurements satisfying the metric description from all monitoring sources are considered in the insight computation. In the case of Insight 1, as no `FROM` statement is used, the delay of all buses will be used in the computation. What is more, `Filters` can be attached and applied directly to the metric stream. Furthermore, in the composition definition, an optional `BY` statement is available, which permits grouping the expression evaluation based on a given `Metric` key. For Insight 1, "grouping" is applied to compute the average bus delay per city segment.

Function	Description
SUM	The sum of all values within a window
COUNT	The number of values within a window
PRODUCT	The product between all values within a window
MIN	The minimum value of the window
MAX	The maximum value of the window
ARITHMETIC_MEAN	The mean of all values within a window
GEOMETRIC_MEAN	The geometric mean within a window
VARIANCE	The variance of all values within a window
SDEV	The standard deviation of all values within a window
MODE	The most common value within a window
MEDIAN	The median value with a window
PERCENTILE	The value below which a given percentage of measurements fall within a window
TOP_K	The top k values from a window

TABLE I: StreamSight Window-Based Supported Functions

Function	Description
RUNNING_MEAN	The running average of all values in the stream
RUNNING_SDEV	The running standard deviation of the stream
EWMA	The exponential weighted moving average
PEWMA	The probabilistic weighted moving average
RUNNING_MAX	The max value of all values
RUNNING_MIN	The min value of all values

TABLE II: StreamSight Accumulative Supported Functions

Finally, the optional `WITH` statement allows the user to define certain optimization strategies and constraints to improve runtime performance. Specifically, `SALIENCE` denotes the importance of an insight, allowing the user to prioritize query execution over other queries. In turn, `SAMPLE` allows users to specify that query execution can be applied on a percentage of the available measurements so that an approximate answer is given in a fraction of the time required to execute the query on the entire data. StreamSight also supports users to set the `MAX_ERROR` and `CONFIDENCE` which are optimization constraints allowing for the query to be executed on a sample of the data, where the constructed sample must satisfy the aforementioned constraints.

C. Query Model Expressivity

The following examples illustrate the expressivity and main features of the StreamSight query model.

Window Operations: the query model supports the aggregation of values that fall within a time window. The window is the period of time that specifies how far in the past values are considered. Insight 1 illustrates such an example, where the values from a 10 minute time window are aggregated every 5 seconds to compute the average bus delay.

Temporal Compositions: the query model allows the user to compose insights from multiple expressions that may even have different time windows. This is particularly useful when the intention is to reveal the metric stream evolution over time. Insight 3 outputs, per city segment, the ratio between the current 10 minute average delay and the current hourly average which is updated with new data every 5 seconds. When the output is over 1, this means that the particular city segment is experiencing an increase in buses' delay.

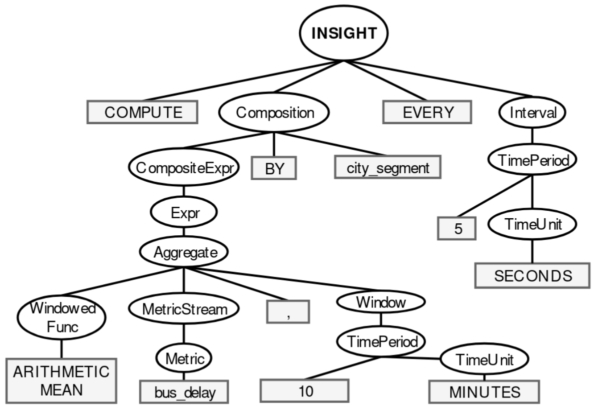

Fig. 2: The Abstract Syntax Tree for Insight 1

```
COMPUTE ( ARITHMETIC_MEAN(bus_delay, 10 MINUTES)
          DIV
          ARITHMETIC_MEAN(bus_delay, 60 MINUTES)
) BY city_segment EVERY 5 SECONDS
```

Insight 3: Computes per city segment the ratio between the current 10 min and the current hour average bus delay

Accumulated Operations: these differ from window operations by not requiring a time sliding window. Instead, results are updated only when new measurements are available, using previously computed data. A prominent example is an Exponential Weighted Moving Average (EWMA), with Insight 4 depicting how the number of passengers is accumulated per bus stop using an EWMA. These operations are often parameterized; in this case a denotes the discounting rate of past observations and is set to 0.85. Insight 4 uses a membership ruling to only aggregate data FROM night bus streams.

```
COMPUTE EWMA[a=0.85](passengers FROM NightBuses)
BY bus_stop EVERY 5 SECONDS
```

Insight 4: Computes an exponential weighted moving average of the number of passengers from night buses per bus stop.

Hybrid Compositions: the query model supports the combination of window and accumulated operations, thus enriching the tool-set of possible analytic operations. A valuable insight constructed from our scenario, is shown in Insight 5. This insight returns the difference of the average delay of the last 10 minutes from the EWMA.

```
COMPUTE (
   ARITHMETIC_MEAN(bus_delay, 10 MINUTES)
   SUB
   EWMA[a=0.65](bus_delay)
) BY city_segment  EVERY 5 SECONDS
```

Insight 5: Computes the difference between the 10 minute average delay and the EWMA per city segment.

Filtered Compositions: these operations support the application of filter predicates on both input and output streams. For example, Insight 6 is used to count, per city segment, the number of times a bus was delayed for more than 2 minutes in a 1 hour sliding window, but outputs results only when the counter is greater than 10. With this query, false positives are reduced with traffic operators only notified to take action when a substantial number of delays are actually reported.

```
COMPUTE (
   COUNT(bus_delay WHEN bus_delay >= 2, 1 HOUR)
) WHEN >= 10 BY city_segment EVERY 1 MINUTES
```

Insight 6: Counts the number of bus delays exceeding 2 min per city segment, but only outputs result if more than 10 delays are detected.

In turn, Insight 7 can be used to detect, per city segment, abnormal bus delays which exceed more than 3 standard deviations from the current mean.

```
COMPUTE bus_delay
WHEN bus_delay > (RUNNING_MEAN(bus_delay)
                 + 3 * RUNNING_SDEV(bus_delay))
EVERY 5 SECONDS BY city_segment
```

Insight 7: Report, per city segment, bus delays exceeding more than 3 standard deviations from the current mean.

User-defined Optimizations: allow users to: (i) prioritize query execution over other queries so that when there is a high load influx, high priority queries are not delayed; and (ii) enforce query execution over a sample of the available measurements to output immediate results. For the latter, users can denote either the exact sample size as a percentage from the available measurements or denote the maximum tolerable relative error and confidence interval the sampling technique must obey when constructing the sample. For example, Insight 8 features a salience of 4 to denote that in the case of a high load influx, it is prioritized over queries with a lower priority (default salience is 0). Thus, high priority queries are executed first, until resource saturation, and other queries are queued until resources are released. In turn, Insight 8 is computed over 20% of the available measurements.

```
COMPUTE ARITHMETIC_MEAN(bus_delay, 10 MINUTES)
BY city_segment EVERY 5 SECONDS
WITH SALIENCE 4 AND SAMPLE 0.20
```

Insight 8: Insight for average bus delay with denoted salience and sampling size.

IV. Algorithmic Implementation

This section provides details of the algorithmic process. We describe the steps for mapping insights into stream operations and elaborate on key aspects of the compilation phase.

Parsing Phase: The initial step that takes as input the insight description from the user and constructs the query model. Specifically, the query model is decomposed and represented as an **Abstract Syntax Tree (AST)** that materializes the syntactic rules of Grammar 1. Figure 2 depicts the AST for Insight 1. Each subtree (inner node) corresponds to a grammar rule while the leaves are the tokens and symbols of the language. An insight is syntactically correct when the tree can be successfully constructed, meaning that the grammar rules have been matched. In the case of a syntax error, the process stops and returns a message to the user containing the identified syntax mistakes. For our prototype, we adopt ANTLR [23], a Java framework for creating the grammar parser. The output of this process contains the constructed ASTs for all analytic queries submitted by the user.

Compilation Phase: This phase constructs the **Query Execution Plan** capturing the pipeline of stream operations

Algorithm 1 Update Insight Stream

```
function updInsight (Composite $comp$, MetricStream $stream$)
    $cached \leftarrow getInsightStreamFromCache(comp)$
    if $cached$ not $null$ then
        return $cached$ //cached insight stream already updated
    end if
    if $comp$ is CompositeExpr then
        $op \leftarrow comp.getOperator()$
        $lStream \leftarrow updInsight(comp.getLeft(), stream)$
        $rStream \leftarrow updInsight(comp.getRight(), stream)$
        $newStream \leftarrow merge(lStream, rStream, op)$
    else if $comp$ is Aggregate then
        $aggr \leftarrow comp.getAggFuntion()$
        $newStream \leftarrow applyAggregate(aggr, stream)$
    else if $comp$ is MetricStream then
        $memb \leftarrow comp.getMembership()$
        $newStream \leftarrow applyMembership(memb, stream)$
    else if $comp$ is Number then
        $num \leftarrow comp.getNumericVal()$
        $newStream \leftarrow genNumericStream(num, stream)$
    end if
    if $comp.hasFilter()$ then
        $op \leftarrow filter.getOperator()$
        $expr \leftarrow comp.getFilterExpr()$
        $fStream \leftarrow updInsight(expr, stream)$
        $newStream \leftarrow applyFilter(newStream, fStream, op)$
    end if
    $addCompositeToCache(comp)$
    return $newStream$
end function
```

that transform raw data into valuable insights based on the AST representation for the given queries. To achieve this, the compilation process requires as input the AST for each query and the initial input stream. The input stream is a logical representation of all the relevant data and metrics streamed by monitoring sources to the processing engine (e.g., bus streams). With this, a query execution plan can be constructed by naively mapping the AST to a pipeline of stream operations, allowing StreamSight to enclose the pipeline to the final query plan for execution by the processing engine.

However, a naive mapping is extremely inefficient as it results in increased data movement and unnecessary intermediate re-computations. Towards this, StreamSight performs the following to improve the query execution plan. Input stream filtering is performed first to clean the the stream from monitoring sources and metric data not required from any of the given queries. As such, the compilation process selects only the set of metrics that are denoted within the *membership* description of each insight to form the new input stream. Next, StreamSight groups streams that will be used together to update composite expressions in (intermediate) insight computations and present overlapping updating time intervals. Afterwards, the compilation process derives and schedules the application of user optimizations to further reduce the volume of streamed data and segregate identical metrics streams but comprised with different data points (e.g., the same bus stream used in two queries but with different sampling rate applied). Thus, with the application of the above steps early on, the optimized query plan prohibits redundant data to pass for further processing, saving valuable network

Algorithm 2 Reuse Intermediate Insight Streams

```
function getInsightStreamFromCache(Composite $comp$)
    $cached \leftarrow cachedComposites[comp]$ //matching composites
    if $cached$ is $empty$ then
        return $null$ //no matching composite
    end if
    $intrv \leftarrow comp.getInterval()$
    if $cached.intervals[intrv]$ is $empty$ then
        return $null$ //comp in cache but not same interval
    end if
    $uOpts \leftarrow comp.getUserOpts()$
    if $cached.userOpts[uOpts]$ is $empty$ then
        return $null$ //comp in cache but not same optimizations
    end if
    return $cached.getInsightStream(comp, intrv, uOpts)$
end function
```

and computational resources.

Update Insight Stream: Having enforced the above, StreamSight proceeds to define the pipeline of stream operations. This step is a recursive procedure and is introduced in Algorithm 1. The procedure takes as input a composition and the current stream, to derive or update the insights based on the operations denoted in the AST nodes of each query. As presented earlier, a composition can be numerical, a metric stream, an aggregate or a new composite expression. In the case of a composite expression, the algorithm takes the left- and right-hand sides of the expression and recursively applies the same procedure. The result from both sides is then operated using the binary operation taken from the composite expression, forming a new insight stream. The other cases (i.e., aggregate, numerical) act as the terminary base cases of the recursive process. When the recursion reaches an aggregate, it applies the aggregation function and returns the updated insight stream. In turn, numerical expressions generate numeric streams while membership expressions can be applied on metric streams to cleanse the stream from monitoring sources not of interest (e.g., Insight 4). Finally, if a filter is attached to the composite, the algorithm will derive the filter expression and discard datapoints in the current stream not satisfying the filter expression.

Aggregates: These functions are predefined code snippets enforced by stream operators on metric streams to derive a new stream capturing aggregated values for a denoted time-window. Here we show how StreamSight compiles high-level aggregates into low-level streaming operations supported by the programming model of the underlying processing engine. For instance, the `ARITHMETIC_MEAN` will generate the following code for Apache Spark Streaming:

```
stream.map((k,v) -> (k, (v,1)))
      .reduceByKey( ((v1,c1), (v2,c2))
                    -> (v1+v2, c1 + c2] )
      .map( (k, (sum, count)) -> {
            if (count != 0) {
               avg = sum / count
               return (key, avg)
            } else {
               return (key, 0)
            }
      })
```

In the above, the input stream is transformed by mapping each value to a (value, 1) tuple and then reduction is applied to group values and apply the intended aggregate function. Another example is illustrated below for the `RUNNING_MAX` aggregate. The auto-generated code will first derive in parallel the max value between tuple pairs by applying reduction. This value is then compared to the current max with an update emitted, for further reduction, only if the pairwise max exceeds the previously acknowledged max value.

```
stream.reduceByKey(
    (v1, v2) -> MAX(v1,v2)
).updateStateByKey(
    (pairMax, curMax) -> {
         if (pairMax > curMax) {
             return max = pairMax
         }
});
```

Reusing Intermediate Results: StreamSight generates and combines data streams recursively, until the complex insight stream is created. This procedure will take place for every insight submitted by the user. Nonetheless, distributed processing engines, such as Apache Spark, consider and evaluate both analytic queries and composite expressions as independent processes. Thus, execution is completely isolated, even if two or more queries/composites feature pipeline components which operate on the same composite ruling and data. This is illustrated in the example below, where although two insight descriptions feature the 10min arithmetic mean from the same input stream, both pipelines must recompute the aggregate every time. However, this incurs a huge compute penalty from having to execute the same streaming operations on the same data. It also incurs a communication penalty if these tasks are scheduled for execution on multiple and different machines.

```
COMPUTE
  ARITHMETIC_MEAN(bus_delay, 10 MINUTES)
  BY city_segment EVERY 5 SECONDS

COMPUTE(
  ARITHMETIC_MEAN(bus_delay, 10 MINUTES)
  DIV
  ARITHMETIC_MEAN(bus_delay, 60 MINUTES)
) BY city_segment EVERY 5 SECONDS
```

For Apache Spark, users can request for in memory caching of stream results, but for clusters deployed across wide areas of geographic coverage, persistent storage must be enabled (e.g., HDFS) with the analogous compute and disk access penalty. To address this, StreamSight: (i) detects at the compilation phase pipeline operations executed on the same input stream; (ii) parameterizes and enables caching; and (iii) outputs the optimized query execution. Thus, before Algorithm 1 attempts to update a composition, it checks via Algorithm 2 if a cached result exists. However, to determine if a composition is cached, it is not enough to find the same cached expression, but rather both the updating interval and the enforced user optimizations must also match so that the composition operates on the same data. Hence, the execution plan notifies the task scheduler of the underlying processing engine which queries and composite expressions share results. Based on this, the task scheduler at runtime will enforce worker nodes to cache and checkpoint streaming operation results in memory and then broadcast results to the workers nodes that will require them (Fig. 3). Thus, by leveraging StreamSight to coordinate the query optimization process, intermediate result dissemination is enabled which significantly reduces redundant and complex computations (e.g., sorting, joins, etc.).

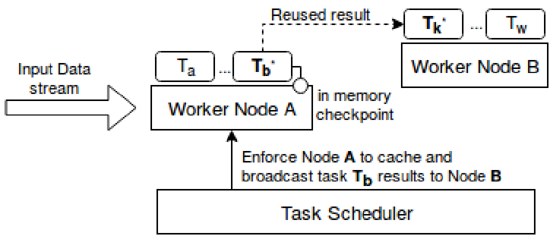

Fig. 3: Reusing Intermediate Results

Sampling: enables the execution of an insight description on a portion of the streamed measurements to receive approximate but in time answers with two popular sampling techniques (*reservoir* and *stratified sampling*). In reservoir sampling, a random sample of fixed size (reservoir) is probabilistically selected without replacement from a dataset of unknown size. This property and the fact that it can be performed online makes it ideal for streaming analytics. However, each measurement is selected with equal probability which can significantly alter the sample statistical quality, if multiple and heterogeneous monitoring sources comprise the input stream. Thus, reservoir sampling is applied only when the user denotes through the insight description the exact sample size.

To overcome this challenge, stratified sampling is also used, where each monitoring source is sampled independently. This reduces the sampling error and improves sample quality, but works only if the statistics of all streams are known (e.g., stream length). However, this assumption is unrealistic in practice. Thus, for StreamSight we adopt the Weighted Hierarchical Reservoir Sampling (WHRS) approach proposed in [21] which combines Reservoir and Stratified Sampling. In this approach, we first "stratify" the input stream on each worker node receiving measurements into monitoring streams. Then, reservoir sampling is applied but with the difference that the reservoir size of each independent monitoring source is dynamically adjusted at runtime. This is achieved through a weighting mechanism, where the significance of each stratified reservoir is periodically updated. Updating the weighting that adjusts the sample size of each reservoir allows StreamSight to adhere to user-defined constraints for sample error-bounds and confidence intervals. Specifically, by notifying the Task Scheduler through the query plan to keep track of the input rate per monitoring stream the weighting per reservoir can be updated efficiently without any additional data movement.

V. Evaluation

In this section, we introduce a comprehensive study to evaluate StreamSight performance when integrated with Apache Spark as the distributed processing engine and configured in an emulated edge computing environment. We realize our testbed on an Openstack private cloud and use Docker to completely isolate and cap compute, memory and network

Workload	Insight ID	LoC (Spark)
Dublin Buses	I_1	14
	I_2	16
	I_4	26
	I_5	38
	I_6	13
	I_7	41
	I_8	14
NYC Taxis	T_1	10
	T_2	22
	T_3	17
	T_4	22

TABLE III: Insights Used in Experiments

resources for deployed edge servers. Specifically, edge servers are configured with 1 VCPU clocked at 2.66GHz, 1 GB RAM and linked with a 2Mbps upload and 16Mbps download wifi interface. We use tc-tool [24] to introduce an artificial network latency of $\sim 15ms$ among edge servers and $\sim 65ms$ between these servers and monitoring sources. We opt for these specific capabilities to resemble actual edge servers deployed in smart transportation services [22]. As our intent is not on service placement, all edge servers are placed in equal network distance. Hence, the distributed processing engine is comprised of 16 worker nodes deployed on the aforementioned edge servers and one master node responsible for job submission and coordination. The master node is configured with 4 VCPUs and 4GB RAM to avoid any potential bottlenecks.

A. Workload and Insight Description

To stress the Spark engine, we develop an IoT workload emulator [2]. The emulator takes as input the *workload* description (e.g., request rate, IoT source type) and the *dataset* to use when monitoring data are not randomly generated. For our evaluation, instead of trivial and random data, we select two publicly available and real-world workloads to truly reveal the strengths of our framework. These workloads are:

- **Dublin Smart City Buses Network [22]**: This workload is comprised of 1 month of data (Jan. 2014) from 968 buses. Each bus is equipped with a GPS tracking device and periodically sends data to the Intelligent Transportation Service. A record in the monitoring stream features 16 metrics, including: bus id, location coordinates, operating city region, etc. Moreover, in each record there is an estimation of the current bus route delay.
- **NYC Taxis [25]**: This workload is comprised of routes from $10,000$ taxis and limousines in New York in 2017. Each vehicle is equipped with GPS tracking to record data for each route, including: passenger number, charged amount, tip amount, payment type, pickup/dropoff location, etc. In total, each record captures 18 metrics.

Table III lists the analytic insights applied for both workloads, along with the lines of code required for an expert developer to compose optimized versions of these queries using the Apache Spark programming model[3]. For the Dublin bus workload we use 7 of the insight descriptions introduced in Section III. For the NYC workload we introduce 4 new insight descriptions, denoted as T_{1-4}, which are of actual interest to taxi companies and drivers for selecting areas of the city to operate. Specifically, T_1 computes the maximum tip amount per city region in a 3 hour sliding window with new data included every 5 seconds. A filter is applied on this insight so that only cash and credit card payments are included. In the dataset these payment types are listed as options 1-2.

```
COMPUTE
  MAX(tip_amount FROM yellow_Cab_List
                 WHEN payment_type <= 2, 3 HOURS)
BY city_region EVERY 5 SECONDS
```

For brevity, we omit depicting $T_2 - T_4$ which adopt the same 3 hour sliding window, but compute: the average tip amount (T_2), the 95^{th} percentile for the tip amount (T_3), and the top 5 tips (T_4). All these insights are computed per city region and include the aforementioned filter.

B. Performance Evaluation

In this scenario we evaluate the performance of StreamSight Framework on an increasing volume of data without allocating any new resources. This scenario is extremely important in edge deployments, since installing new physical servers in-place every time load fluctuates, is usually not an option.

Evaluation Metric: We evaluate the *performance growth* that can be achieved by the examined configurations compared to the *Baseline*, where insight descriptions are deployed to the Spark without any optimizations. To evaluate growth we monitor both the workload *request rate* and the Spark cluster *total delay*. The *total delay* metric includes both the Spark cluster *processing* and *scheduling* time. Processing time, denotes the time required to execute the query on a batch of data points (e.g., sliding window), while scheduling time designates the time from which a batch is queued, up to the time it starts being processed. The cluster is considered stable when the total delay is beneath the batch time, which in our case is $10s$. Otherwise, when the delay is continuously increasing, batches are queued and not processed immediately as the system is unable to keep up and therefore the system is considered unstable.

Configurations: For both datasets, we evaluate the following configurations: (i) Spark with StreamSight deployed to optimize intermediate result reusability; (ii) *WHRS Sampling*, which introduces weighted hierarchical reservoir sampling to Spark; and (iii) Streamsight enabled with WHRS sampling. In regards to workload enforcement, we apply an increasing linear workload through the Workload Emulator by starting our testbed with 50 monitoring sources (buses, taxis) and increasing this number every 3 minutes which, in turn, increases the data input rate to the distributed processing engine.

Results: Figure 4 depicts the total delay for the Baseline and StreamSight configuration when applying the linear workload for the Bus dataset without sampling. We observe that Streamsight can handle 38% more requests per second than the Baseline before the cluster becomes unstable. This translates to a $1.4\times$ performance growth and can be explained by the

[2] https://github.com/dtrihinas/JobEmulator

[3] We only count lines of code for the actual query and omit any stream and cluster configurations.

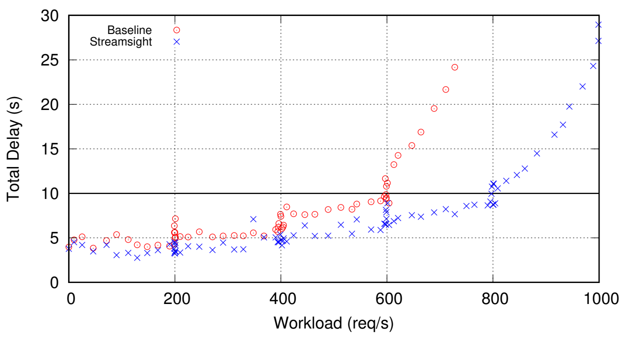

Fig. 4: Bus Workload: Baseline vs StreamSight

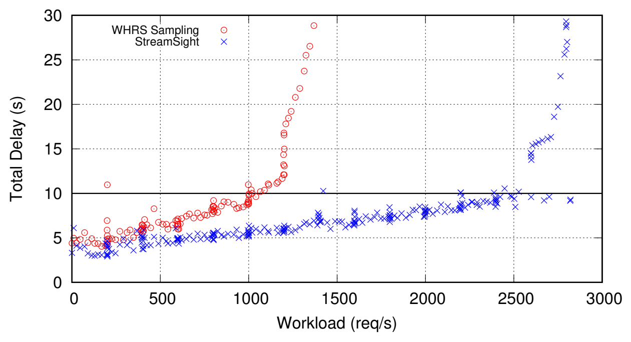

Fig. 5: Bus Workload: WHRS Sampling vs StreamSight

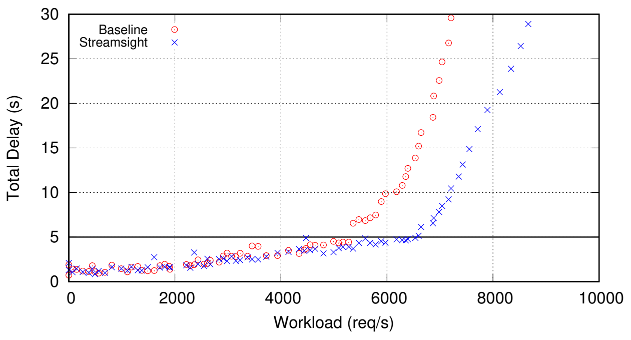

Fig. 6: Taxi Workload: Baseline vs StreamSight

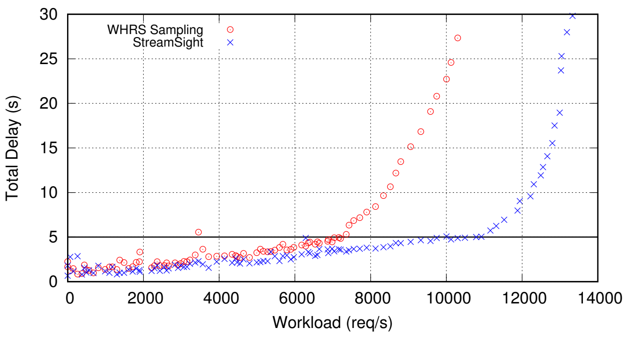

Fig. 7: Taxi Workload: WHRS Sampling vs StreamSight

fact that the reusage of intermediate streams, alleviates the system from further re-computations, thus allowing more load to be handled with the same processing capabilities. Next, Figure 5 shows the total delay for the bus dataset while applying the linear workload and user optimizations take place by enabling sampling with a 20% fixed sample size. In regards to the previous results, WHRS achieves only a $1.7\times$ performance growth compared to the Baseline, although it operates on 20% of the data. On the other hand, StreamSight achieves a $4.3\times$ performance growth from the Baseline and when directly compared to WHRS, we immediately observe that StreamSight is able to handle almost three times more workload. This significantly highlights the importance of not requiring constant data movement across worker nodes for intermediate computations, even when enabling highly efficient approximation techniques.

Similarly, Figures 6 and 7 show the total delay under a linear workload for the NYC dataset. In this setup, Figure 6 depicts the performance growth of StreamSight (no sampling) over the baseline which is $1.3\times$. In turn, for the results depicted in Figure 7, user optimizations are enabled with the max error bounded to 10% with a 95% confidence interval. This resulted to a sample size fluctuating between 24% and 39%. Thus, compared to the Baseline, WHRS was only able to achieve a $1.5\times$ performance growth, while StreamSight managed a $2.4\times$ performance growth. In regards to the Dublin workload, the performance growth is comparably lower. However, this is due to the fact that the number of monitoring sources is significantly higher and the lower sampling rate. Additionally, the query composition overlap ratio is lower for this dataset and choice of queries. The Dublin workload has a 23% overlap between compositions, while the NYC a mere 12% overlap. Still, StreamSight is able to handle almost two times more workload compared to WHRS, primarily due to eliminating redundant sorting operations for T_3 and T_4, which again highlights the importance of reusing intermediate streams.

#Insights	**Overlap Ratio**	**Baseline** (ms)	**StreamSight** (ms)	**Performance Growth Rate**
1	0%	1426	1451	-1.8%
2	0%	2888	2815	2.6%
3	0%	5134	5087	0.9%
4	11%	5985	5171	13.6%
5	23%	8664	7043	19.7%
6	25%	-	7318	$> 26.8\%$

TABLE IV: Reused Results at Different Overlap Ratios

C. *Re-usage of Intermediate Streams*

Motivated by the previous findings, the purpose of this experiment is to explore the effect of reusing intermediate computations at different overlap levels. In this experiment, we fixed the workload input rate for the Dublin dataset and examined insights with different overlap levels.

Evaluation Metric: The average processing time, measured in milliseconds for the Baseline and StreamSight.

Results: The results are depicted in Table IV. The last column shows the performance growth of StreamSight compared to the Baseline. The first three insights have zero overlap between them and the difference between the performance is statistically insignificant. This shows that using StreamSight does not incur a performance overhead, even for expert users capable of deriving optimized versions for insight descriptions. On the contrary, for a 11% overlap the increment of performance is 13.6% and for a 23% overlap we have a 19.7% increment of performance. Finally, for six insights at a 25% overlap,

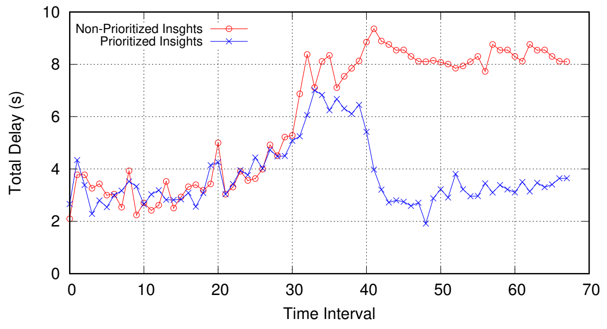

Fig. 8: Priority vs Non-Priority Insights

Streamsight succeeded processing the entire dataset but the Baseline configuration failed. For this reason, we set the Baseline value to the processing time just before the system failed. Thus, for a 25% overlap StreamSight achieved at least a $26,8\%$ increase in performance.

D. Insight Prioritization Evaluation

The final experiment aims to evaluate the performance of insights when prioritization is used. For this, we use insights I_1, I_2, I_4 and I_5 from the Dublin dataset, in which I_1 and I_2 are prioritized (`SALIENCE 5`) and fix the workload to an input rate that allows the cluster to remain stable. The initial topology is comprised of 16 workers and at $t = 20$ we double the network latency between workers to cause an overall delay increment in insight computation. Figure 8 depicts the total delay for both prioritized and non-prioritized insights. We observe that up to $t = 20$ the total delay was kept on average below $4s$. When workers are suddenly stressed and for about 12 time intervals, the delay increases for all insights until the distributed processing engine is stabilized. After stabilization, thanks to the StreamSight optimized query execution plan, the Task Scheduler of the processing engine allocates more resources to prioritized insights, with their delay falling back to $4s$, while non priority insights stabilize close to $8s$.

VI. Conclusion

In this paper, we have presented StreamSight, a novel framework for simplifying the specification, compilation and execution of analytic queries on distributed processing engines deployed in edge computing environments. To this end, platform operators can rapidly compose analytic queries with StreamSight automatically compiling and scheduling these queries into stream processing jobs to derive runtime analytic insights. With StreamSight, query definition is decoupled from the programming model of the underlying distributed processing engine, while query execution response time is kept at permissible limits. Our evaluation on Apache Spark, with real-world data and queries of actual interest, shows that performance is significantly improved when intermediate streams are reused. In turn, user-defined optimizations (approximate answers and prioritization) can be decisive for the proper execution of stream processing operations in an edge computing environment with limited resources.

Acknowledgement. This work is partially supported by the EU Commission in terms of Unicorn 731846 (H2020-ICT-2016-1).

References

[1] Ericsson, "Internet of Things 2018 Forecasts," https://www.ericsson.com/en/mobility-report/internet-of-things-forecast, 2018.

[2] B. Zhang, N. Mor, J. Kolb, D. S. Chan, K. Lutz, E. Allman, J. Wawrzynek, E. Lee, and J. Kubiatowicz, "The cloud is not enough: Saving iot from the cloud," in *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 15)*, Santa Clara, CA, Jul. 2015.

[3] D. Trihinas, G. Pallis, and M. Dikaiakos, "Low-Cost Adaptive Monitoring Techniques for the Internet of Things," *IEEE Transactions on Services Computing*, pp. 1–1, 2018.

[4] W. Shi and S. Dustdar, "The promise of edge computing," *Computer*, vol. 49, no. 5, pp. 78–81, May 2016.

[5] K. Kloudas, M. Mamede, N. Preguiça, and R. Rodrigues, "Pixida: Optimizing data parallel jobs in wide-area data analytics," *Proc. VLDB Endow.*, vol. 9, no. 2, pp. 72–83, Oct. 2015.

[6] X. Sun and N. Ansari, "Edgeiot: Mobile edge computing for the internet of things," *IEEE Communications*, vol. 54, no. 12, pp. 22–29, 2016.

[7] J. Ren, H. Guo, C. Xu, and Y. Zhang, "Serving at the edge: A scalable iot architecture based on transparent computing," *IEEE Network*, vol. 31, no. 5, pp. 96–105, 2017.

[8] D. Trihinas, L. F. Chiroque, G. Pallis, A. F. Anta, and M. D. Dikaiakos, "ATMoN: Adapting the "Temporality" in Large-Scale Dynamic Networks," in *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, July 2018, pp. 400–410.

[9] L. Wang, L. Jiao, J. Li, and M. Mhlhuser, "Online resource allocation for arbitrary user mobility in distributed edge clouds," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, June 2017, pp. 1281–1290.

[10] V. D. Maio and I. Brandic, "First hop mobile offloading of dag computations," in *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, May 2018, pp. 83–92.

[11] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 637–646, Oct 2016.

[12] S. Nastic, S. Sehic, M. Vgler, H. L. Truong, and S. Dustdar, "Patricia – a novel programming model for iot applications on cloud platforms," in *2013 IEEE 6th International Conference on Service-Oriented Computing and Applications*, Dec 2013, pp. 53–60.

[13] R. Viswanathan, G. Ananthanarayanan, and A. Akella, "CLARINET: Wan-aware optimization for analytics queries," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, Savannah, GA, 2016, pp. 435–450.

[14] K. Hsieh, A. Harlap, N. Vijaykumar, D. Konomis, G. R. Ganger, P. B. Gibbons, and O. Mutlu, "Gaia: Geo-distributed machine learning approaching LAN speeds," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, 2017, pp. 629–647.

[15] A. Gupta, R. Harrison, M. Canini, N. Feamster, J. Rexford, and W. Willinger, "Sonata: Query-Driven Streaming Network Telemetry," in *Proceedings of SIGCOMM'18*, Aug 2018.

[16] D. Trihinas, G. Pallis, and M. Dikaiakos, "ADMin: adaptive monitoring dissemination for the internet of things," in *IEEE Conference on Computer Communications (INFOCOM 2017)*, Atlanta, USA, May 2017.

[17] A. Jonathan, A. Chandra, and J. Weissman, "Locality-aware load sharing in mobile cloud computing," in *Proceedings of the10th International Conference on Utility and Cloud Computing*, ser. UCC '17. New York, NY, USA: ACM, 2017, pp. 141–150.

[18] S. Yang, "Iot stream processing and analytics in the fog," *IEEE Communications Magazine*, vol. 55, no. 8, pp. 21–27, 2017.

[19] M. Satyanarayanan, P. Simoens, Y. Xiao, P. Pillai, Z. Chen, K. Ha, W. Hu, and B. Amos, "Edge analytics in the internet of things," *IEEE Pervasive Computing*, vol. 14, no. 2, pp. 24–31, Apr.-June 2015.

[20] H. Cao, M. Wachowicz, and S. Cha, "Developing an edge computing platform for real-time descriptive analytics," in *2017 IEEE International Conference on Big Data (Big Data)*, Dec 2017, pp. 4546–4554.

[21] Z. Wen, D. Quoc, P. Bhatotia, R. Chen, and M. Lee, "ApproxIoT: Approximate Analytics for Edge Computing," in *38th IEEE International Conference on Distributed Computing Systems (ICDCS 2018)*, 2018.

[22] Dublin, "Smart City ITS," https://data.smartdublin.ie/, 2018.

[23] ANTLR, "parser generator," http://www.antlr.org, 2018.

[24] B. Hubert *et al.*, "Linux advanced routing & traffic control howto," 2002.

[25] NYC, "Taxi & Limousine Commision," https://goo.gl/X9rCpq, 2018.