

# RIoT Bench: A Real-time IoT Benchmark for Distributed Stream Processing Platforms

Anshu Shukla, Shilpa Chaturvedi and Yogesh Simmhan

*Department of Computational and Data Sciences*

*Indian Institute of Science (IISc), Bangalore 560012, India*

*Email: shukla@grads.cds.iisc.ac.in, shilpa@grads.cds.iisc.ac.in, simmhan@cds.iisc.ac.in*

## Abstract

The Internet of Things (IoT) is an emerging technology paradigm where millions of sensors and actuators help monitor and manage, physical, environmental and human systems in real-time. The inherent closed-loop responsiveness and decision making of IoT applications make them ideal candidates for using low latency and scalable stream processing platforms. Distributed Stream Processing Systems (DSPS) hosted on Cloud data-centers are becoming the vital engine for real-time data processing and analytics in any IoT software architecture. But the efficacy and performance of contemporary DSPS have not been rigorously studied for IoT applications and data streams. Here, we develop *RIoT Bench*, a Real-time IoT Benchmark suite, along with performance metrics, to evaluate DSPS for streaming IoT applications. The benchmark includes 27 common IoT tasks classified across various functional categories and implemented as reusable micro-benchmarks. Further, we propose four IoT application benchmarks composed from these tasks, and that leverage various dataflow semantics of DSPS. The applications are based on common IoT patterns for data pre-processing, statistical summarization and predictive analytics. These are coupled with four stream workloads sourced from real IoT observations on smart cities and fitness, with peak streams rates that range from 500 – 10,000 *messages/sec* and diverse frequency distributions. We validate the RIoT Bench suite for the popular Apache Storm DSPS on the Microsoft Azure public Cloud, and present empirical observations. This suite can be used by DSPS researchers for performance analysis and resource scheduling, and by IoT practitioners to evaluate DSPS platforms.

## 1 Introduction

Internet of Things (IoT) is a technology paradigm wherein ubiquitous sensors numbering in the billions will be able to *monitor* physical infrastructure and environment, human beings and virtual entities in real-time, *process* both real-time

and historic observations, and *take actions* that improve the efficiency and reliability of systems, or the comfort and lifestyle of society. The technology building blocks for IoT have been ramping up over for a decade, with research into pervasive and ubiquitous computing [43], and sensor networks [17] forming precursors. Recent growth in the capabilities of high-speed mobile (e.g., 3G/4G) and *ad hoc* (e.g., Bluetooth) networks [15], smart phones and devices, affordable sensing and crowd-sourced data collection [1], Cloud data-centers, and Big Data analytics platforms have all converged to the current inflection point for IoT.

Existing IoT deployments in vertical domains such as *Smart Power Grids* [36] and *health and fitness monitoring* [41] already have millions of sensing and actuation points that constantly stream observations and trigger responses. The IoT stack for such domains is tightly integrated to serve specific needs, but typically operates on a closed-loop *Observe Orient Decide Act (OODA)* cycle [33], where sensors communicate time-series observations of the (physical or human) system to the Cloud for analysis, and the resulting analytics drives recommendations that are enacted on the system to improve it, which is again observed and so on. In fact, this *closed-loop responsiveness* is one of the essential and distinguishing design characteristics of IoT applications, compared to other Big Data domains.

This low-latency cycle makes it necessary to process data streaming from sensors at fine spatial and temporal scales, in *real-time*, to derive actionable intelligence. In particular, this streaming analytics has to be done at massive scales (millions of sensors, thousands of events per second) from across distributed sensors, requiring large computational resources. *Cloud computing* offers a natural platform for scalable processing of the observations at globally distributed data centers, and sending a feedback response to the IoT system at the edge.

Recent *Big Data platforms* like Apache Storm [39], Spark Streaming [42] and Flink [13] provide an intuitive dataflow programming model for composing such streaming applications, with a scalable, low-latency execution engine designed for commodity clusters and Clouds. These *Distributed Stream Processing Systems (DSPS)* are becoming essential components of any IoT stack to support online analytics and decision-making for IoT applications. DSPS provide the ability to compose a dataflow graph of user-defined tasks that can process a continuous stream of opaque messages on distributed resources. This flexibility allows DSPS to incorporate a wide variety of business logic for real-time processing and online analytics necessary for a diverse and emerging domain like IoT. In fact, reference IoT solutions from Cloud providers like Amazon AWS<sup>1</sup> and Microsoft Azure<sup>2</sup> include their proprietary stream and event processing engines as part of the IoT software architecture.

Shared-memory stream processing systems [16, 17] have been investigated over a decade back for wireless sensor networks, with benchmarks such as *Linear Road* [6] being proposed. But there has not been a detailed review of, or benchmarks for, *distributed* stream processing for IoT. IoT encompasses mul-

---

<sup>1</sup><https://aws.amazon.com/iot/how-it-works/>

<sup>2</sup><https://www.microsoft.com/en-in/server-cloud/internet-of-things/overview.aspx>

tiple domains, and applications go well beyond traditional social network and web traffic workloads for which DSPS were designed for [39]. They include a swathe of generalizable tasks for data pre-processing, statistical summarization and predictive analytics, as well as analytics for specific IoT application areas like Smart Transportation or health. As such, the efficacy and performance of contemporary DSPS have not been rigorously studied for *IoT applications and data streams*. One reason is the absence of a well-defined IoT benchmark that realistically captures the domain features, exercises the unique compositional capabilities of DSPS, and validates them on real data streams. We address this gap in this paper.

This paper extends our prior published work, significantly increasing both the breadth and depth of the benchmark suite [35]. We add 14 new tasks to the earlier 13 tasks, including in new categories; two new streaming dataflow applications, besides updating the earlier two as well; and two new data workloads from the smart grid and personal fitness domains. We also include support for spatial scaling to increase the number of sensor streams, in addition to the temporal scaling used earlier to increase the stream rates. These make our benchmark comprehensive.

Specifically, we make the following contributions in this article:

1. We classify different *characteristics* of streaming applications, their composition semantics, and their data sources, in § 3.
2. Then, in § 4, we propose *categories of tasks* that are essential for IoT applications and the *key features of input data streams* they operate upon.
3. We identify *performance metrics* of DSPS that are necessary to meet the latency and scalability needs of streaming IoT applications, in § 5.
4. We propose the **RIoTBench** real-time IoT benchmark for DSPS based on representative *micro-benchmark tasks*, drawn from the above categories, in § 6. We design four reference *IoT applications* that span Data pre-processing, Statistical analytics and Predictive Analytics, and are composed from these tasks. We also identify *four real-world streams* with different distributions as workloads on which to evaluate them.
5. Lastly, we validate the proposed benchmark suite for the popular *Apache Storm* DSPS, and present empirical results for the same in § 7.

Our contributions benefit two classes of audience. One, for *developers and users in IoT domains*, RIoTBench offers a set of realistic IoT tasks and applications that they can customize and configure to help evaluate candidate DSPS platforms for their performance and scalability needs. Two, for *researchers on Big Data*, it provides a reference micro and application benchmark, along with datasets, that can be used as a baseline to uniformly compare the impact of their research advances in resource management, scalability and resiliency for DSPS on the emerging IoT domain.

## 2 Background and Related Work

Stream processing systems allow users to compose applications as a dataflow graph, with task vertices having some user-defined logic and streaming edges passing messages between the tasks. The systems then run the applications continuously over incoming data streams. Early Data Stream Management Systems (DSMS) extended Database Management Systems (DBMS) to support by sensor network applications, that have similarities to IoT [8,14,18]. They supported continuous query languages with operators such as join and aggregation similar to SQL, but with a temporal dimension using time and tuple window operations. These have been extended to distributed implementations [9,11] and, more recently, complex event processing (CEP) engines for detecting sequences and patterns [19].

Contemporary Distributed Stream Processing Systems (DSPS) like Apache Storm, Spark Streaming, Flink and Yahoo S4 [13,32,39,42] were designed using Big Data fundamentals – running on commodity clusters and Clouds, offering weak scaling, ensuring robustness, and supporting fast data processing over thousands of events per second. Unlike DSMS, DSPS do not support native query operators and instead allow users to plug in their own logic composed as dataflows that are executed on a cluster. Event processing and querying can be higher-level abstractions on top of these <sup>3</sup>. While developed for web and social network applications, such fast data platforms have found use in financial markets, astronomy, and particle physics. IoT is one of the more recent domains to consider them.

There are design and architectural differences even within DSPS, which we highlight as part of our characterization. The types of programming semantics supported can vary, and determines the flexibility in composition. Spark Streaming uses micro-batch processing in contrast to per-tuple processing in Storm, with consequences trade-offs between latency and throughput. As a result, it is important to qualitatively and quantitatively evaluate these frameworks for specific application domains, and the distributed platform they target. Understanding the common set of feature dimensions and performance metrics, in addition to the actual IoT benchmark definitions, is necessary for fair comparison across the DSPS. We discuss these later for DSPS operating on Clouds to support IoT applications.

### 2.1 DSPS Benchmarks

Work on DSMS spawned the *Linear Road Benchmark (LRB)* [6] that was proposed as an application benchmark. In the scenario, the DSMS had to evaluate toll and traffic queries over event streams from a virtual toll collection and traffic monitoring system. This has parallels with current smart transportation scenarios. However, there have been few studies or community efforts on benchmarking DSPS, other than individual evaluation of research prototypes

---

<sup>3</sup>Apache Trident, <http://storm.apache.org/releases/1.0.1/Trident-tutorial.html>

against popular DSPS like Storm or Spark. These efforts define their own measures of success – typically limited to throughput and latency – and use generic workloads such as the Enron email dataset with empty operations (NoOps) as micro-benchmark to compare InfoSphere Streams [30] and Storm.

*SparkBench* [3] is a framework-specific benchmark for Apache Spark, and includes four categories of applications from domains spanning Graph computation and SQL queries, with one on streaming applications supported by Spark Streaming. The benchmark metrics include CPU, memory, disk and network IO, with the goal of identifying tuning parameters to improve Spark’s performance. *CEPBen* [28] evaluates the performance of CEP systems based of the functional behavior of queries. It shows the degree of complexity of CEP operations like filter, transform and pattern detection. The evaluation metrics consider event processing latency, but ignore network overheads and CPU utilization. Further, CEP applications rely on a declarative query syntax to match event patterns rather than a dataflow composition based on user-logic provided by DSPS.

*StreamBench* [29] is the closest work that partially addresses our goals. The authors propose 7 micro-benchmarks on 4 different synthetic workload suites generated from real-time web logs and network traffic to evaluate DSPS. Metrics including performance, durability and fault tolerance are proposed. The benchmark covers different dataflow composition patterns and common tasks like grep and wordcount, and compare Storm and Spark Streaming.

The paper, while addressing the gap that existed in generalizable benchmarks DSPS, still falls short on several counts. It focuses on micro-benchmarks and does not consider larger applications with more tasks and complex structures. Design patterns like duplicates and round-robin, and selectivity ratios are not explicitly considered. The benchmark does not cover a broad range of realistic input data rates either. We address these gaps. At the same time, we do not emphasize durability or fault-tolerance metrics in our study, through these metrics can be added.

In contrast to these DSPS benchmarks, RIoT Bench offers relevant micro- and application-level benchmarks for evaluating DSPS, specifically for *IoT workloads* for which such platforms are increasingly being used. Our benchmark is designed to be *platform-agnostic*, *simple* to implement and execute within diverse DSPS, and *representative* of both the application logic and the data stream workloads observed in IoT domains. This allows for the performance of DSPS to be independently and reproducibly verified for IoT applications.

## 2.2 Big Data and IoT Benchmarks

There has been a slew of Big Data benchmarks that have been developed recently in the context of processing high volume (i.e., MapReduce-style) and enterprise/web data that complement our work. *Hibench* [25] is a workload suite for evaluating Hadoop with popular micro-benchmarks like Sort, WordCount and TeraSort, MapReduce applications like Nutch Indexing and PageRank, and machine learning algorithms like K-means Clustering. *BigDataBench* [22] analyzes workloads from social network and search engines, and analytics algorithms

like Support Vector Machine (SVM) over structured, semi-structured and unstructured web data. Both these benchmarks are general purpose workloads that do not target any specific domain, but MapReduce platforms at large.

*BigBench* [23] uses a synthetic data generator to simulate enterprise data found in online retail businesses. It combines structured data generation from the TPC-DS benchmark [31], semi-structured data on user clicks, and unstructured data from online product reviews. Queries cover data *velocity* by processing periodic refreshes that feed into the data store, *variety* by including free-text user reviews, and *volume* by querying over a large web log of clicks. We take a similar approach for benchmarking fast data platforms, targeting the IoT domain specifically and using real public data streams.

*Chronos* [24] is a recent work to generate and simulate streams for benchmarking. Their aim is to generate realistic input data streams with a distribution similar to given sample events. They use elastic infrastructure to generate events at high rates, and validate their work for telecom, advertising and stock market data. Their work is complementary to ours, as we propose dataflow patterns and applications, as well as representative datasets as part of benchmarks which are run at their native and scaled rates. Chronos can be used to stress these benchmarks further with larger inputs and faster rates.

There has been some recent work on benchmarking IoT applications. In particular, the generating large volumes of synthetic sensor data with realistic values is challenging, yet required for benchmarking. *IoTAbench* [7] provides a scalable synthetic generator of time-series datasets. It uses a Markov chain model for scaling the time series with a limited number of inputs such that important statistical properties of the stream is retained in the generated data. They have demonstrated this for smart meter data. The benchmark also includes six SQL queries to evaluate the performance of different query platforms on the generated dataset. Their emphasis is on the data characteristics and content, which supplements our focus on evaluating the runtime aspects of the DSPS platform.

*CityBench* [4] is a benchmark to evaluate RDF stream processing systems. They include different generation patterns for smart city data, such as traffic vehicles, parking, weather, pollution, cultural and library events, with changing event rates and playback speeds. They propose fixed set of semantic queries over this dataset, with concurrent execution of queries and sensor streams. Here, the target platform is different (RDF database), but in a spirit as our work.

Benchmarks for IoT hardware is also becoming important. *IoT-Connect* [40] is an Industry-Standard Benchmark for Embedded Systems to analyze the behavior of micro-controllers with various connectivity interfaces like Bluetooth, Thread, LoRa, and WiFi. It also provides methods to determine the energy consumption for IoT devices.

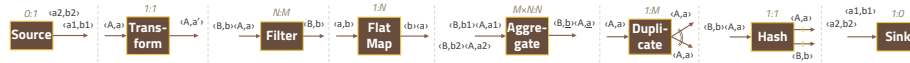


Figure 1: Common task patterns and semantics in streaming applications.

### 3 Characteristics of DSPS Applications and Streams

In this section, we review the common application composition capabilities of DSPS, and the dimensions of the streaming applications that affect their performance on DSPS. These semantics help define and describe streaming IoT applications based on DSPS capabilities.

#### 3.1 Dataflow Composition Semantics

DSPS applications are composed as a *dataflow graph*, where vertices are user provided *tasks* and directed edges refer to *streams of messages* that can pass between them. The graph need not be acyclic. Tasks in the dataflows can execute zero or more times, and a task execution usually depends on data-dependency semantics, i.e, when “adequate” inputs are available, the task executes. However, there are also more nuanced patterns that are supported by DSPS that we discuss. *Messages* (or events or tuples) from/to the stream are consumed/produced by the tasks. DSPS typically treat the messages as opaque content, and only the user logic may interpret the message content. However, DSPS may assign identifiers to messages for fault-tolerance and delivery guarantees, and some message attributes may be explicitly exposed as part of the application composition for the DSPS to route messages to downstream tasks.

*Selectivity ratio*, also called *gain*, is the average number of output messages emitted by a task on consuming a unit input message, expressed as  $\sigma = \text{input rate} : \text{output rate}$ . Based on this, one can assess whether a task amplifies or attenuates the incoming message rate. It is important to consider this while designing benchmarks as it can have a multiplicative impact on downstream tasks.

There are message generation, consumption and routing semantics associated with tasks and their composition. Fig. 1 captures the basic *composition patterns* supported by modern DSPS. **Source** tasks have only outgoing edge(s), and these tasks encapsulate user logic to generate or receive the input messages that are passed to the dataflow. Likewise, **Sink** tasks have only incoming edge(s) and these react to the output messages from the application, say, by storing it or sending an external notification.

**Transform** tasks, sometimes called *Map*<sup>4</sup>, generate one output message for every input message received ( $\sigma = 1 : 1$ ). Their user logic performs a transformation on the message, such as changing the units or projecting only a subset of attribute values. **Filter** tasks allow only a subset of messages that

<sup>4</sup>Spark Programming Guide, <http://spark.apache.org/docs/latest/programming-guide.html>

they receive to pass through, optionally performing a transformation on them ( $\sigma = N : M$ ,  $N \geq M$ ). Conversely, a **FlatMap** consumes one message and emits multiple messages ( $\sigma = 1 : N$ ). An **Aggregate** pattern consumes a *window* of messages, with the window width provided as a *count* or a *time* duration, and generates one or more messages that is an aggregation over each message window ( $\sigma = N : 1$ ). Specific DSPS may expose additional dataflow patterns as well.

When a task has multiple outgoing edges, routing semantics on the dataflow decide if an output message is *duplicated* onto all the edges, or just one downstream task is selected for delivery, either based on a *round robin* behavior or using a *hash function* on an attribute in the outgoing message to decide the target task. Similarly, multiple incoming streams arriving at a task may be *merged* into a single interleaved message stream for the task. Or alternatively, the messages coming on each incoming stream may be conjugated, based on order of arrival or an attribute exposed in each message, to form a *joined* stream of messages. Other custom DSPS routing semantics may exist too.

There are additional dimensions of the streaming dataflow that can determine its performance on a DSPS. Tasks may be *data parallel*, in which case, it can be allocated concurrent resources (threads, cores) to process messages in parallel by different instances the task. This is typically possible for tasks that do not maintain state across multiple messages. The *number of tasks* in the dataflow graph indicates the size of the streaming application. Tasks are mapped to computing resources, and depending of their degree of parallelism and resource usage, it determines the cores/VMs required for executing the application. The *length of the dataflow* is the latency of the critical (i.e., longest) path through the dataflow graph, if the graph does not have cycles. This gives an estimate of the expected latency for each message and also influences the number of network hops a message on the critical path has to take in the cluster.

### 3.2 Data Stream Characteristics

We list a few characteristics of the input data streams that impact the runtime performance of streaming applications, and help classify IoT message streams.

The *input throughput* in messages/sec is the cumulative frequency at which messages enter the source tasks of the dataflow. Input throughputs can vary by application domain, and are determined both by the number of streams of messages and their individual rates. This combined with the dataflow selectivity will impact the load on the dataflow and its individual tasks, and determine the output throughput.

*Throughput distribution* captures the variation of input throughput over time. In real-world settings, the input data rate is usually not constant and DSPS need to adapt to this. There may be several common data rate distributions besides a *uniform* one. There can be *bursts* of data coming from a single sensor, or a coordinated set of sensors. A *saw-tooth* behavior is seen in the ramp-up/-down before/after specific events. A *Normal* distribution can occur



for diurnal (day vs. night) stream sources, with *bi-modal* variations capturing peaks during the morning and evening periods of human activity.

Lastly, the *message size* provides the average size of each message, in bytes. Often, the messages sizes remain constant for structured messages arriving from specific sensor or observation types, but may vary for free-text input streams or those that interleave messages of different types. This size help assess the communication cost of transferring messages in the dataflow.

## 4 Characteristics of IoT Applications and Streams

Here, we categorize IoT tasks, applications and data streams used within DSPS, based on the domain requirements. These, together with the patterns and semantics discussed in the previous section, offer a search space for defining dataflows and workloads that meaningfully and comprehensively validate IoT applications on DSPS.

### 4.1 Categories of IoT Tasks

IoT covers a broad swathe of domains, many of which are rapidly developing. So, it is not possible to comprehensively capture all possible IoT application scenarios. However, DSPS have clear value in supporting the real-time processing, analytics, decision making and feedback that is intrinsic to most IoT domains. Here, we attempt to categorize these common processing and analytics tasks that are performed over real-time data streams.

**Parse.** Messages are encoded on the wire in a standard text-based or binary representation by the stream sources, and need to be parsed upon arrival at the application. Text formats in particular require string parsing by the tasks, and are also larger in size on the wire. The tasks within the application may themselves retain the incoming format in their streams, or switch to another format or data model, say, by projecting a subset of the fields. They may also annotate and extend the number of fields. Industry-standard formats that are popular for IoT domains include CSV, XML, SenML and JSON text formats, and EXI and CBOR binary formats. For e.g., IETF’s *SenML (Sensor Markup Language)* [26] can define an array of entries, where each entry is an object object that encapsulates attributes and their values, such as the unique identifier for the sensor, the time of measurement, and the current value, with the ability to model repetitions, relative time, etc. SenML serializations into JSON, XML and EXI are possible.

**Filter.** Messages may require to be filtered based on specific attribute values present in them, for data quality checks, to route a subset of message types to a part of the dataflow graph, or as part of their application logic. *Value filters* such as min/max or band-pass filters check the numeric values of different observational fields from the sensors and can drop outliers. Since IoT event rates may be high, more efficient *Bloom filters* are a probabilistic structure that can be used to process large sets of discrete values with low space complexity but

a small fraction of false positives. It can be used to detect invalid sensors or users in an incoming data streams. Filtering over text or media streams is also possible, but requires consideration like using text or video processing libraries.

**Statistical Analytics.** Groups of messages within a sequential time or count window of a stream may require to be aggregated as part of the application. The aggregation function may be *common mathematical operations* like average, count, minimum and maximum. They may also be *higher order statistics* such finding outliers, quartiles, second and third order moments, and counts of distinct elements. Statistical *data cleaning* like linear interpolation or denoising using Kalman filters are common for sensor-based data streams. Some tasks may maintain just local state for the window width (e.g., local average) while others may maintain state across windows (e.g., moving average). When the state size grows, here again approximate aggregation algorithms may be used. *Distinct approximate count* is another such example of statistical tasks where we try to find approximate distinct values present in stream.

**Predictive Analytics.** Predicting future behavior of the system based on past and current messages is an important part of IoT applications. Various statistical and machine-learning algorithms may be employed for predictive analytics over sensor streams. The *predictions* may either use a recent window of messages to estimate the future values over a time or count horizon in future, or train models over streaming messages that are periodically used for predictions over the incoming messages. Even simple techniques like *interpolation* can be useful for replacing empty entries by interpolation over past values. *Classification* algorithms like decision trees, neural networks and naïve Bayes can be trained to map discrete values to a category, which may lead to specific actions taken on the IoT system. External packages like Weka or R may be used by such tasks. The *training* itself can be an online task that is part of a DSPS dataflow. For e.g., ARIMA and linear regression use statistical methods to predict uni- or multi-variate attribute values, respectively. Also trained models can be updated on the fly within such forecasting tasks.

**Pattern Detection.** Another class of tasks are those that identify patterns of behavior over several events. Unlike window aggregation which operate over static window sizes and perform a function over the values, pattern detection matches user-defined predicates on messages that may not be sequential or even span streams, and returns the matched messages. These are often modeled as *state transition automata* or *query graphs*. Common patterns include contiguous or non-contiguous sequence of messages with specific property on each message (e.g., high-low-high pattern over 3 messages), a join over two streams based on a common attribute value, or even semantic matching [44]. Complex Event Processing (CEP) engines like Siddhi [37] may be embedded within the DSPS task to match such patterns.

**Visual Analytics.** Other than automated decision making, IoT applications often generate *charts and animations* for consumption by end-users or system managers. These visual analytics may be performed at the client's browser using libraries like `D3.js`, in which case the processed data stream is aggregated and provided to the users. Alternatively, the streaming application may itself

periodically generate such plots and visualizations as part of the dataflow, to be hosted on the web or pushed to the client. Charting and visualization libraries like `XChart`, `gnuplot` or `matplotlib` may be used for this purpose.

**IO Operations.** Lastly, the IoT dataflow may need to access external storage or messaging services to access/push data into/out of the application. These may be to store or load trained models, archive incoming data streams, access historic data for aggregation and comparison, and subscribe to message streams or publish actions back to the system. These require access to *file storage*, *SQL and NoSQL databases*, and *publish-subscribe messaging systems*. Often, these may be hosted as part of the Cloud platforms themselves like Azure Storage. This also include writing files to local or remote disk, and optionally compressing or uncompressing them. Each of them have their own characteristics in term of latency, peak rate supported and resource usage.

## 4.2 Categories of IoT Applications

The tasks from the above categories, along with other domain-specific tasks, are composed together to form streaming IoT dataflow applications. These domain dataflows themselves fall into specific classes based on common use-case scenarios, and loosely map to the Observe-Orient-Decide-Act (OODA) phases.

**Extract-Transform-Load (ETL) and Archival** applications are front-line “observation” dataflows that receive and pre-process the data streams, and if necessary, archive a copy of the data offline. Pre-processing may perform data format transformations, normalize the units of observations, data quality checks to remove invalid data, interpolate missing data items, and temporally reorder messages arriving from different streams, annotate with the metadata. The pre-processed data may be archived to table storage, and passed onto subsequent dataflow for further analysis.

**Summarization and Visualization** applications perform statistical aggregation and analytics over the data streams to understand the behavior of the IoT system at a coarser granularity. Statistical analytics may include tasks such as finding approximate counts, identifying skewness in data distribution, and using linear regression for online trends. Such summarization can give the high-level pulse of the system, and help “orient” the decision making to the current situation. These tasks are often coupled with visualization tasks in the dataflow to present the summary status to end-users and decision makers.

**Prediction and Pattern Detection** applications use current information and historic models to help determine the future state of the IoT system, and “decide” if any reaction is required. They identify patterns of interest that may indicate the need for a correction, or forecasts based on current behavior that require preemptive actions. For e.g., a trend that indicates an unsustainably growing load on a smart power grid may cause a decision to preemptively shed load, or a detection that the heart-rate from a fitness watch is dangerously high may trigger a decision to reduce physical exertion. Model-based prediction applications are also coupled with batch or online dataflow applications that periodically re-train the models using observed data.

**Classification and Notification** applications determine specific “actions” that are required and communicate them to the IoT system. Decisions may be mapped to specific actions, and the entities in the IoT system that can enact those are notified. These notifications can be delivered using SMS gateways, web service calls, or publish-subscribe brokers. For e.g., the need for load shedding in the power grid may map to notifying specific residents with a request for curtailment, or the need to reduce physical exertion may lead to a treadmill being notified to reduce the speed. The classification or case based reasoning systems may also require model training, like for predictive analytics.

### 4.3 IoT Data Stream Characteristics

IoT data streams are often generated by physical sensors that observe physical systems or the environment. As a result, they are typically time-series data that are generated periodically by the sensors. The sampling rate for these sensors may range from once a day to hundreds per second, depending on the domain. The number of sensors themselves may vary from a few hundred to millions as well. IoT applications like smart power grids can generate high frequency plug load data at thousands of messages/sec from a small cluster of residents, or low frequency data from a large set of sensors, such as in smart transportation or environmental sensing. As a result, we may encounter a wide range of input throughputs from  $10^{-2}$  to  $10^5$  messages/sec. In comparison, streaming web applications like Twitter deal with 6000 tweets/sec from 300M users.

At the same time, this event rate itself may not be uniform across time. Sensors may also be configured to emit data only when there is a change in observed value, rather than unnecessarily transmitting data that has not changed. This helps conserve network bandwidth and power for constrained devices when the observations are slow changing. Further, if data freshness is not critical to the application, they may sample at high rate but transmit at low rates but in a burst mode. E.g. smart meters may collect kWh data at 15 min intervals from millions of residents but report it to the utility only a few times a day, while the FitBit smart watch syncs with the Cloud every few minutes or hours even as data is recorded every few seconds. Message variability also comes into play when human-related activity is being tracked. Diurnal or bimodal event rates are seen with single peaks in the afternoons, or dual peaks in the morning and evening. E.g., sensors at businesses may match the former while traffic flow sensors may match the latter.

There may also be a variety of observation types from the same sensor device, or different sensor devices generating messages. These may appear in the same message as different fields, or as different data streams. This will affect both the message rate and the message size. These sensors usually send well-formed messages rather than free-text messages, using standards like SenML. Hence their sizes are likely to be deterministic if the encoding format is not considered – text formats tend to bloat the size and also introduce size variability when mapping numbers to strings. However, social media like tweets and crowd-sourced data are occasionally used by IoT applications, and these may have

more variability in message sizes.

## 5 Performance Metrics

We identify and formalize commonly-used quantitative performance measures for evaluating DSPS for the IoT workloads.

**Latency.** Latency for a message that is generated by a task is the time in seconds it took for that task to process one or more inputs to generate that message. If  $\sigma = N : M$  is the selectivity for a task  $T$ , the time  $\lambda_M^T$  it took to consume  $N$  messages to *causally produce* those  $M$  output messages is the latency of the  $M$  messages, with the *average latency* per message given by  $\bar{\lambda}^T = \frac{\lambda_M^T}{|M|}$ .

When we consider the average latency  $\bar{\lambda}$  of the dataflow application, it is the average of the time difference between each message consumed at the source tasks and all its causally dependent messages generated at the sink tasks.

The latency per message may vary depending on the input rate, resources allocated to the task, and the type of message being processed. While this task latency is the inverse of the mean throughput, the *end-to-end latency* for the task within a dataflow will also include the network and queuing time to receive a tuple and transmit it downstream.

**Throughput.** The output throughput is the aggregated rate of output messages emitted out of the sink tasks, measured in messages per second. The throughput of a dataflow depends on the input throughput and the selectivity of the dataflow, provided the resource allocation and performance of the DSPS are adequate. Ideally, the output throughput  $\omega^o = \sigma \times \omega^i$ , where  $\omega^i$  is the input throughput for a dataflow with selectivity  $\sigma$ . It is also useful to measure the *peak throughput* that can be supported by a given application, which is the maximum stable rate that can be processed using a fixed quanta of resources.

Both throughput and latency measurements are relevant only under *stable conditions* when the DSPS can sustain a given input rate, i.e., when the latency per message and queue size on the input buffer remain constant and do not increase unsustainably.

**Jitter.** The ideal output throughput may deviate due to variable rate of the input streams, change in the paths taken by the input stream through the dataflow (e.g., at a `Hash` pattern), or performance variability of the DSPS. We use jitter to track the variation in the output throughput from the expected output throughput, defined for a time interval  $t$  as,

$$J_t = \frac{\omega^o - \sigma \times \omega^i}{\sigma \times \bar{\omega}^i}$$

where the numerator is the observed difference between the expected and actual output rate during interval  $t$ , and the denominator is the expected long term average output rate given a long-term average input rate  $\bar{\omega}^i$ . In the case of an ideal DSPS, jitter will tend toward zero, even if there are instantaneous changes in the input rate.

**CPU and Memory Utilization.** Streaming IoT dataflows are expected to be resource intensive, and the ability of the DSPS to use the distributed resources efficiently with minimal overhead is important. This also affects the VM resources and consequent price to be paid to run the application using the given stream processing platform. We track the CPU and memory utilization for the dataflow as the average of the CPU and memory utilization across all the VMs that are being used by the dataflow’s tasks. The per-VM information can also help identify which VMs hosting which tasks are the potential bottlenecks and can benefit from data-parallel scale-out, and cases of over-allocation of resources.

## 6 *RIoTBench* IoT Benchmark Suite

We propose benchmark workloads to help evaluate the metrics discussed before for various DSPS. These benchmarks are in particular targeted for emerging IoT applications, to help them distinguish the capabilities of contemporary DSPS on Cloud computing infrastructure. The benchmarks themselves have two parts, the dataflow logic that is executed on the DSPS and the input data streams that they are executed for. We next discuss our choices for both.

### 6.1 IoT Micro-benchmarks

We propose a suite of common IoT tasks that span various categories we have identified and different streaming task patterns. These tasks form independent micro-benchmarks, and are further composed into application benchmarks later. The goal of the micro-benchmarks is to evaluate the performance of the DSPS for individual IoT tasks, and we measure the *peak input throughput* that they can sustain on a unit computing resource as the performance metric. This offers a baseline for comparison with other DSPS, and can also inform resource scheduling decisions for more complex application dataflows composed using these tasks.

Table 1 lists the different micro-benchmark tasks, and their IoT categories, task patterns, and selectivity. These are grouped by their categories. The *parse* category includes tasks that process standard text formats such as SenML and XML, and convert them to object formats, and also convert from a CSV format to a SenML form with additional semantics. The annotation task appends metadata content to an existing message based on an in-memory lookup for a unique ID present in the tuple. All these parse tasks transform messages from one form to another. The Bloom filter finds practical use in the *filter* category for processing a large, discrete data space. It is trained with a white-list of valid sensor IDs that it will permit. The simple value range filter is used filtering in messages with observation fields that fall within a fixed upper and lower bound.

We have several tasks in the *statistical analytics* category that perform aggregations and transformations. Basic statistics include a simple average of a single attribute’s values over a count window, and a generic accumulator that

Table 1: IoT Micro-benchmark Tasks with different IoT Categories and DSPS Patterns.

Task Name	Code	Category	Pattern	$\sigma$ Ratio	State
Annotate	ANN	Parse	Transform	1:1	No
CsvToSenML	C2S	Parse	Transform	1:1	No
SenML Parsing [26]	SML	Parse	Transform	1:1	No
XML Parsing	XML	Parse	Transform	1:1	No
Bloom Filter [12]	BLF	Filter	Filter	1:0/1	No
Range Filter	RGF	Filter	Filter	1:0/1	No
Accumulator	ACC	Statistical	Aggregate	N:1	Yes
Average	AVG	Statistical	Aggregate	N:1	Yes
Distinct Appox. Count [21]	DAC	Statistical	Transform	1:1	Yes
Kalman Filter [27]	KAL	Statistical	Transform	1:1	Yes
Second Order Moment [5]	SOM	Statistical	Transform	1:1	Yes
Decision Tree Classify [34]	DTC	Predictive	Transform	1:1	No
Decision Tree Train	DTT	Predictive	Aggregate	N:1	No
Interpolation	INP	Predictive	Transform	1:1	Yes
Multi-var. Linear Reg.	MLR	Predictive	Transform	1:1	No
Multi-var. Linear Reg. Train	MLT	Predictive	Aggregate	N:1	No
Sliding Linear Regression	SLR	Predictive	Flat Map	N:M	Yes
Azure Blob D/L	ABD	IO	Source/Transform	1:1	No
Azure Blob U/L	ABU	IO	Sink	1:1	No
Azure Table Lookup	ATL	IO	Source/Transform	1:1	No
Azure Table Range	ATR	IO	Source/Transform	1:1	No
Azure Table Insert	ATI	IO	Transform	1:1	No
MQTT Publish	MQP	IO	Sink	1:1	No
MQTT Subscribe	MQS	IO	Sink	1:1	No
Local Files Zip	LZP	IO	Sink	1:1	No
Remote Files Zip	RZP	IO	Sink	1:1	No
MultiLine Plot [2]	PLT	Visualization	Transform	1:1	No

buffers incoming messages based on a count window for use by other tasks. The second order moments over time-series values is another common statistics we implement. Estimating the frequencies a large range of streaming values can be memory intensive, and distinct approximate count performs a probabilistic count over the incoming messages while conserving memory. Lastly, the Kalman filter we provide is a popular denoising algorithm used for smoothing sensor data values in a time-series.

*Predictive analytics* uses the Weka library to implement several common Machine Learning tasks. A multi-variate linear regression is included to predict one attribute’s numerical value based on the values of one or more other in the message. This has both online training and online prediction tasks. Similarly, the decision tree classifier is used for predict a class based on enumerated field values in the message, and also comes with a training and a classification task. Training for both these models happens over large, batched windows of messages. Interpolation and linear regression are standard techniques used for univariate time-series observation, and are available in the micro-benchmark suite.

We have several *IO* tasks for reading and writing to Microsoft Azure Cloud’s

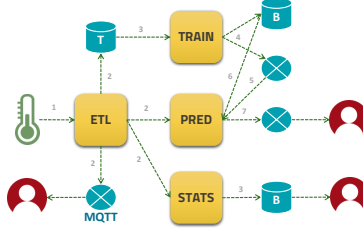


Figure 2: High-level logical interactions between different sensors, benchmark applications, platform services and users

file (blob) storage and NoSQL (table) storage. In addition, the common file operation of compressing a set of files is also available, with the source files being either on local disk or on the network. Publish/subscribe to/from an MQTT publish-subscribe broker for notifications and also included. Lastly, a single exemplar *Visualization* task in the form of a Java XChart plotting library is present to accumulate and generate an image file.

A micro-benchmark dataflow is composed for each of these tasks as a sequence of a source task, the benchmark task and a sink task. As can be seen, these tasks also capture different dataflow patterns such as transform, filter, aggregate, flat map, source and sink.

## 6.2 IoT Application Benchmarks

Application benchmarks are valuable in understanding how non-trivial and meaningful IoT applications behave on DSPS. Application dataflows for a domain are most representative when they are constructed based on real or realistic application logic, rather than synthetic tasks. In case applications use highly-custom logic or proprietary libraries, this may not be feasible or reusable as a community benchmark. However, many of the common IoT tasks we have defined earlier are naturally composable into application benchmarks that satisfy the requirements of a OODA decision making loop.

Fig. 2 shows a high-level use case of such an IoT scenario that is generalizable to many domains such as smart power, transportation and fitness. This is achieved by the interaction between four different application dataflows we propose. Here, input streams from sensors in the domain arrive at a *Extract-Transform-Load (ETL)* dataflow that performs data pre-processing and cleaning on the observations and archives it to Cloud table storage. Further, one output stream is published to the MQTT message broker so that clients interested in real-time monitoring can subscribe to it, while another copy is forked to the second dataflow which performs *Statistical Summarization (STATS)*. This application does higher order aggregation and plotting operations, and stores the generated plots to Cloud blob file storage, from where web-pages can load the visualization files on browsers.

Concurrently, two dataflows support predictive analytics. *Model Training*



(*TRAIN*) periodically loads the archived data from the Cloud table store and trains forecasting models that are stored in the Cloud, and notifies the MQTT broker of an updated model being available. The *Predictive Analytics (PRED)* dataflow subscribes to the broker and downloads the new models from the Cloud, and continuously operates over the pre-processed data stream from ETL to make predictions and classifications that can indicate actions to be taken on the domain. It then notifies the message broker of the predictions, that can independently be subscribed to by a user or device for action.

More specifically, **ETL** (Fig. 3a) ingests incoming data streams in SenML format, performs data filtering of outliers on individual observation types using a Range and Bloom filter, and subsequently interpolates missing values. It then annotates additional meta-data into the observed fields of the message and then inserts the resulting tuples into Azure table storage, while also converting the data back to SenML and publishing it to MQTT. A dummy sink task shown is used for logging purposes.

The **STATS** dataflow (Fig. 3b) parses the input messages that arrive in SenML format – typically from the ETL, but kept separate here for modularity. It then performs three types of statistical analytics in parallel on individual observation fields present in the message: an average over a 10 message window, Kalman filtering to smooth the observation fields followed by a sliding window linear regression, and an approximate count of distinct values that arrive. These three output streams are then grouped for each sensor IDs, plotted and the resulting image files zipped. These three tasks are tightly coupled and we combine them into a single meta-task for manageability, as is common. and the output file is written to Cloud storage for hosting on a portal.

The **TRAIN** (Fig. 3c) application uses a timer to periodically (e.g., for every minute) trigger a model training run. Each run fetches data from the Azure table available since the last run and uses it to train a Linear Regression model. In addition, these fetched tuples are also annotated to allow a Decision Tree classifier to be trained. Both these trained model files are then uploaded to Azure blob storage and their file URLs are published to the MQTT broker.

The **PRED** (Fig. 3d) application subscribes to these notifications and fetches the new model files from the blob store, and updates the downstream prediction tasks. Meanwhile, the dataflow also consumes pre-processed messages streaming in, say from the ETL dataflow, and after parsing it forks it to the decision tree classifier and the multi-variate regression tasks. The classifier assigns messages into classes, such as good, average or poor, based on one or more of their field values, while linear regression predicts a numerical attribute value in the message using several others. The regression task also compares the predicted values against a moving average and estimates the residual error between them. The predicted classes, values and errors are published to the MQTT broker. The Appendix lists configuration parameters and attributes used for relevant tasks in the dataflows for different workloads we benchmark them on.

As such, these applications leverage many of the compositional capabilities of DSPS. The dataflows include *single and dual sources*; tasks that are composed *sequentially, task-parallel* and as *combined* meta-tasks; *stateful and stateless*

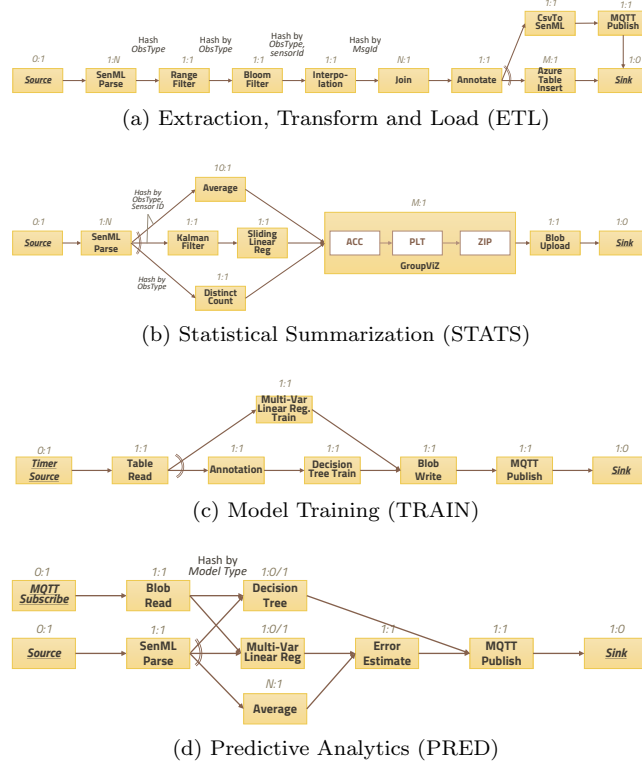


Figure 3: Application benchmarks composed using the micro-benchmark tasks.

tasks; and *data parallel tasks* allowing for concurrent instances. Each message in the data streams contains multiple observation fields, but several of these tasks are applicable only on univariate streams and some are meaningful only from time-series data from individual sources. Thus, the initial parse task for ETL and STATS uses a *flat map* pattern ( $\sigma = 1 : N$ , where  $N$  is number of observational fields) to create observation-specific streams early on. These streams are further passed to task instances, grouped by their observation type and optionally their sensor/meter ID using a *hash pattern*.

### 6.3 IoT Input Stream Workloads

We have identified four real-world IoT data streams available in the public domain as candidates for our benchmarking workload. These correspond to domains within smart cities, which is a major contributor to the growth of IoT, taxi cab services, and personal fitness. Their features are shown in Table 2 and their message rate distribution is in Fig. 4.

**Sense your City (CITY)** [1]. This is an *urban environmental monitoring*

Table 2: Characteristics of Smart Cities data stream workloads used in benchmarks, with temporal and spatial scaling.

Name	Raw Workload				Scaling Factor			Effective Workload	
	Sensors	Attrib.*	Size (bytes)	Distribution	Temporal	Spatial	Effective	Peak Rate (msg/sec)	Sensors
<b>CITY</b> [1]	90	9	380	Uniform	30×	30×	900×	5,000	2,700
<b>FIT</b> [10]	10	26	1,024	Uniform	1×	1×	1×	500	10
<b>GRID</b> [38]	6,435	3	130	Normal	1×	500×	500×	10,000	32,17,500
<b>TAXI</b> [20]	20,355	17	191	Bimodal	1,000×	1×	1,000×	4,000	20,355

\* Every dataset has a minimum of three attributes: sensorId, timestamp and one (or more) observational field(s).

project <sup>5</sup> that has used crowd-sourcing to deploy sensors at 7 cities across 3 continents in 2015, with about 12 sensors per city. Five timestamped observations, outdoor temperature, humidity, ambient light, dust and air quality, are reported every minute by each sensor along with metadata on sensor ID and geolocation. Besides urban sensing, this also captures the vagaries of using crowd-sourcing for large IoT deployments. Data from over 2 months is available. We use a single logical stream that combines the global data from all unique sensors provided in the dataset. Fig. 4a shows a narrow distribution of the message rate, with the peak frequency centered at 5,000 msg/sec.

**NYC Taxi cab (TAXI)** [20]. This offers a stream of *smart transportation* messages that arrive from 2M trips taken in 2013 on 20,355 New York city taxis equipped with GPS. A message is generated when a taxi completes a single trip, and provides the taxi and license details, the start and end coordinates and timestamp, the distance traveled, and the cost, including the taxes and tolls paid. Other similar transportation datasets are also available <sup>6</sup>, though we chose ours based on the richness of the fields. This data has a bi-modal event rate distribution that reflects the morning and evening commutes, with peaks at 300 and 3,200 events/sec. We use 7 days of data from 14-Jan-2013 to 20-Jan-2013 for our benchmark runs.

**Energy dataset (GRID)** [38]. This is a univariate dataset that reports the energy consumption for each smart meter in a pilot smart grid deployment in Ireland. The actual dataset had 6,435 unique sensors and emits a reading every half an hour. Data is available from over 500 days of observations. It shows a normal distribution of data around each half an hour timestamp. The final peak rate of dataset used in the experimental runs is 10,000 events/sec.

**MHealth dataset (FIT)** [10]. The MHEALTH (Mobile HEALTH) dataset consists of body motion and vital signs recordings for ten volunteers of diverse profiles collected when performing physical activities. Sensors measure in different parts of the subject’s body collect acceleration, rate of turn, magnetic field, and ECG data, among others, at a constant rate of 50 Hz. We merge 10 subjects data into a single global stream, with messages having the subject ID as sensor ID. It has a constant rate of 500 events/sec as shown in Fig. 4d.

While these datasets correspond to real values collected from the domain, they are representative samples from even larger datasets that are typically proprietary. In order to capture the real scale of these data streams, we make use

<sup>5</sup><http://map.datacanvas.org>

<sup>6</sup><https://github.com/fivethirtyeight/uber-tlc-foil-response>

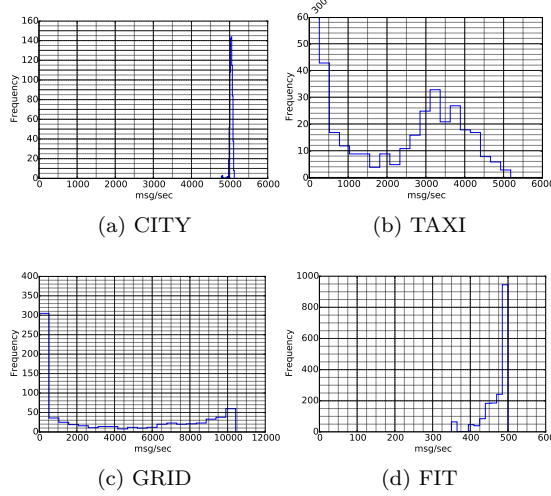


Figure 4: Frequency distribution of input throughput for the four workloads, with the temporal and spatial scaling used for the benchmark runs.

of temporal and spatial scaling. *Temporal scaling* allows us to accelerate the data rate by time-compressing messages that were generated over a longer interval into a smaller one. For e.g., when the CITY data is temporally scaled by  $30\times$ , its original rate per sensor goes from an average of 6 msg/min to 180 msg/min, and 7 days of wall-clock time get reduced to 336 mins of benchmark time. This causes the shape of the distribution in Fig. 4 to be retained, but widens the X Axis. Temporal scaling is relevant when the raw workload data that is available is not representative of the sampling rates that are expected in contemporary IoT sensors and domains. Considering that GPS sensors placed in taxis report their location each second for navigation and monitoring rather than only at the end of the trip, we use a temporal scaling factor of  $1000\times$  for the TAXI workload.

*Spatial scaling*, on the other hand, allow us to simulate a larger number of sensors than available in the raw data. This is necessary when data streams are available only from a small sample of sensors. Here, we consider data streams from the same sensor but during different time windows (e.g., days) to act as if they are from different sensors but at a previous time. This too does not affect the shape of the message rate distribution, but expands the Y Axis. For e.g., in the GRID data, a spatial scaling of  $500\times$  increases the 6,435 smart meters present in original dataset to 3,217,500 unique meters, which is more representative of a city-scale deployment. Similarly, a  $30\times$  spatial scaling in CITY (in addition to the temporal scaling) causes the 336 mins of benchmark time to further reduce to 12 mins of benchmark time, while increasing the sensor count from 90 to 2,700.

These two scaling factors are also shown in Table 2, along with the effective

number of sensors and the peak rate after applying these factors. As we can see, using scaling to create workloads that are representative of real-world scenarios also achieves diversity in the event rate distribution profiles for the input streams, and also offers peak rates that span from 500 to 10,000 msg/sec.

## 7 Evaluation of Proposed Benchmarks

### 7.1 Benchmark Implementation

We implement the 27 micro-benchmarks as generic Java tasks that can consume and produce objects. These tasks are building blocks that can be composed into micro-dataflows and the ETL, STATS, PRED and TRAIN dataflows using any DSPS that is being benchmarked. To validate our proposed benchmark, we compose these dataflows on the Apache Storm open source DSPS, popular for fast-data processing, using its Java APIs. We then run these for the four stream workloads and evaluate them based on the metrics we have defined. The benchmark is available online at <https://github.com/dream-lab/riot-bench>.

In Storm, each task logic is wrapped by a *bolt* that invokes the task for each incoming tuple and emits zero or more response tuples downstream. The dataflow is composed as a *topology* that defines the edges between the bolts, and the *groupings* which determine duplicate or hash semantics. We have implemented a scalable data-parallel event generator that acts as a source task (*spout*). It loads time-series tuples from a given SenML file and replays them as an input stream to the dataflow. While the spatial scaling of the workloads is performed offline as a pre-processing step, our generator can perform temporal scaling online, as it emits the message. We generate random integers as tuples at maximum rate for the micro-benchmarks, and replay the original datasets by scaling their native rates as in table 2 for the application benchmarks, following the earlier distribution.

### 7.2 Experimental Setup

We use Apache Storm 1.0.1 running on OpenJDK 1.7, and hosted on *Ubuntu 14.0* Virtual Machines (VMs) in the Southeast Asia data center of Microsoft Azure public cloud. For the micro-benchmarks, Storm runs the task being benchmarked on one exclusive D1 size VM (1 Intel Xeon E5-2660 core at 2.2 GHz, 3.5 GiB RAM, 50 GiB SSD), while the supporting source and sink tasks and the master service run on a D4 size VM (8 Intel Xeon E5-2660 core at 2.2 GHz cores, 28 GiB RAM, 400 GiB SSD). The larger VM for the supporting tasks and services ensures that they are not the bottleneck, and helps benchmark the peak rate supported by the micro-benchmark task on a single core VM.

For the ETL, STATS, TRAIN and PRED application benchmarks, we use D3 VMs (4 Intel Xeon E5-2660 core at 2.2 GHz cores, 14 GiB RAM, 200 GiB SSD) for all the tasks of the dataflow, while reserving additional D4 VMs to exclusively run the source and sink tasks, and the Storm master service. Storm

Table 3: The number of resources assigned, given as “cores, VMs”, for each application benchmark and workload. Each VM has 4 cores.

App.	CITY	FIT	GRID	TAXI
ETL	11, 3	8, 2	14, 3	10, 3
STATS	27, 7	10, 3	11, 3	32, 8
TRAIN	7, 2	7, 2	N/A <sup>†</sup>	7, 2
PRED	10, 3	9, 3	N/A <sup>†</sup>	9, 3

<sup>†</sup> Benchmarks are not done for the particular applications with the GRID dataset as it is univariate and DTC and MLR tasks require multiple fields.

requires the users to explicitly assign the data parallelism per task, and the total number of resources in the cluster. We determine the number of cores and data parallelism required by each task using a simple resource allocation algorithm, as follows.

We have the peak rate supported by the single-threaded task on a single core as given by the micro-benchmarks, and the peak rate seen for that task for a given application and stream workload by examining the dataflow and selectivity. For tasks where the expected rate in the dataflow is less than its peak rate supported on one core, we assign it an exclusive core and two threads. In cases that are I/O bound (e.g., MQTT, Azure storage) rather than CPU bound, we require multiple task instances on a single core to leverage data parallelism, and sometimes multiple cores as well. Table 3 shows the number of cores and VMs assigned for running the experiments with the applications and stream workloads.

We log the ID and timestamp for each message at the source and the sink tasks in-memory to calculate the latency, throughput and jitter metrics. We also sample the CPU and memory usage on all VMs every 5 secs to plot the utilization metrics. Each experiment runs for  $\sim 10$  mins of wallclock time.

### 7.3 Micro-benchmark Results

Fig. 5 shows plots of the different metrics evaluated for the micro-benchmark tasks on Storm when running at their peak input rate supported on a single D1 VM with one thread. The *peak sustained throughput* per task is shown in Fig. 5a in *log-scale*. We see that most tasks can support 3,000 msg/sec or higher rate, going up to 68,000 msg/sec for ANN, BLF, RGF, ACC, DAC and KAL. XML parsing is highly CPU bound and has a peak throughput of only 310 *msg/sec*. SML parse supports much higher rate than XML with less CPU usage, indicating that it is a better fit for streaming IoT applications than the XML format. DTT and MLT uses WEKA library for model training and supports only 50 and 70 *msg/sec* rate, CPU being the bottleneck. PLT uses the XChart [2] Java charting library and supports only 25 *msg/sec* rate as it is CPU intensive around 70% as shown in 5d at peak rate.

The Azure operations are I/O bound on the Cloud service and slow due to the web service latency. ATR supports only 1 *msg/min*, as the task has to

scan the full table on Azure with, e.g., 753,382 records for the Taxi dataset, to query over non-key attributes on single Azure table partition. Better input rates can be achieved by storing Azure table on multiple partitions with query attributes as partition or row-key. RZP supports 300 *msg/sec* while LZP supports 3,000 *msg/sec* – RZP has to write the Zip file to a remote shared directory while LZP uses a local disk.

The inverse of the peak sustained throughput gives the *mean latency*, and we do not explicitly plot it. However, it is interesting to examine the *end-to-end latency*, calculated as the time taken between emitting a message from the source, having it pass through the benchmarked task, and arrive at the sink task. This is the effective time contributed to the total tuple latency by this task running within Storm, including framework overheads. We see that while the mean latencies should be in sub-milliseconds for the observed throughputs, the box plot for end-to-end latency (Fig. 5b) varies widely up to 2,600 *ms* for Q3, except ACC and INP task. This wide variability could be because of non-uniform task execution times due to which slow executions queue up incoming messages that suffer higher queuing time, such as for DTC and MLR that both use the WEKA library. Or tasks supporting a high input rate in the order of 10,000 *msg/sec*, such as DAC and KAL, may be more sensitive to even small per-tuple overhead of the framework, say, caused by thread contention between the Storm system and worker threads, or queue synchronization.

The Azure tasks that have a lower throughput also have a higher end-to-end latency, but much of which is attributable directly to the task latency. ATR has a latency of 1 *min* due to scanning of the large table. ACC shows wide distribution of latency due to variability in the complexity of operation performed on it. Events associated with a single sensor ID are stored in a time-ordered queue until the threshold count is reached, upon which it extracts all the accumulated values and passes it downstream. MQS shows latency of 1,900 *ms* with no whiskers as the task logic just polls a local queue of messages being populated by the subscribed messages arriving from the broker.

The box-plot for *jitter* (Fig. 5c) shows values close to zero in all cases. This indicates the long-term stability of Storm in processing the tasks at peak rate, without unsustainable queuing of input messages. The wider whiskers indicate the occasional mismatch between the expected and observed output rates. ATR again has a high range for the whiskers as its rate is very low at 1 *msg/min*; thus even minor variation in rate shows high jitter values.

The box plots for CPU utilization (Fig. 5d) shows the single-core VM effectively used at 70% or above in all cases except for the SML, MQS and Azure tasks that are I/O bound. MQS is bounded by the number of threads as single thread is busy in polling the message queue which is not CPU intensive. SML is having low CPU of  $\approx 30\%$ , the reason being that as we are using a JSON representation for SenML which is less CPU intensive as compared to XML. The memory utilization (Fig. 5e) appears to be higher for tasks that support a high throughput, potentially indicating the memory consumed by messages waiting in queue rather than consumed by the task logic itself. MQS shows a high memory usage ( $\approx 50\%$ ) even for a low rate due to buffering of incoming

messages from the broker in a queue that is asynchronously being polled. Similarly, memory for DTT and MLT is  $\approx 45\%$  because a batch of nearly thousand rows is stored in memory for model training triggered by every incoming input message.

## 7.4 Application Results

The ETL and STATS application benchmarks are run for the CITY, FIT, GRID and TAXI stream workloads. TRAIN and PRED are run for CITY, FIT and TAXI datasets and not for GRID because it has only one observation field, and prediction tasks such Decision Tree and Multivariate Linear Classifier uses a combination of fields to predict or classify an observational field. The input rate is as per scaling discussed in table 2 for each dataset.

The *end-to-end latencies* of the applications depend on the sum of the end-to-end latencies of each task in the critical path of the dataflow. For the ETL application, latency values in Fig. 6a remain the same 30 *millisec* for CITY, FIT and TAXI datasets. GRID has a higher variation in latency than others because of its normal distribution of messages over timestamp. The median latency for all the datasets are nearly comparable, with GRID having median latency 50 *millisec* and CITY, TAXI and FIT around 30 *millisec*. The STATS dataflow has latency values in the range of 10 – 40 *millisec*, shown in Fig. 6b, which is higher than ETL and PRED. This is mainly due to the GroupViZ meta-task which batches messages, forming a time-series for plotting, and then accumulating the plots to create a zipped file. Also, its median latency values are highly variable depending on the dataset. The reason is that the accumulation and plotting are done separately for every distinct sensor ID until a fixed count is reached, and hence the latency for the meta-task depends on the content of input messages received.

The TRAIN dataflow’s timer source task simulates the model training trigger every 2 *hours* of original time for CITY, every day for TAXI, and every minute for the FIT dataset. This translates to a benchmark time period of 2 – 5 mins between two source events. The latency values for TRAIN are understandably higher than other applications since it is a batch processing dataflow encoded as a streaming dataflow. The key reason is the Azure Table Range task that scans the full table to fetch rows that were inserted since the last training time. Also, the latency for the CITY dataset in Fig. 6c is larger at 300 *sec* than FIT and TAXI datasets that are at 50 *sec* due to the difference in the table sizes. CITY has 3,629,428 rows in its table while TAXI has 753,382 rows inserted. The PRED topology’s latencies (Fig. 6d) also remain close together at 20 *millisec* for all the datasets. The large range of whiskers for all datasets in PRED is due to DTC and MLR tasks, which exhibit significant variations in their runtimes even for the micro-benchmarks.

The *jitter* is also close to zero in all cases (Fig. 7), indicating a sustainable performance for the application benchmarks. The whiskers for STATS are not visible as the total number of messages at the sink tasks are comparatively fewer than the input messages since the GroupViZ task accumulates many of



the inputs in singleton outputs per sensor ID. Similarly, the whiskers for TRAIN are larger as few messages are emitted from source (max 10 *msg/sec* for FIT) in total, and thus most of the time variation is observed between source and sink rate.

The number of *cores and VMs* required for the same application varies with the workload used (Tbl. 3). This is due to the difference in input rate that is processed by tasks for the respective workload, thus requiring different number of cores per task. We also see that the resource allocation strategy is generally liberal, and resources are under-utilized. The CPU utilization for STATS is higher at 20 – 80% than other applications (Fig. 9). This is due to AVG, DAC and GroupViZ tasks requiring higher CPU%, matches with the CPU% required for microbenchmarks. Also memory usage is higher for STATS in comparison to others due to GroupViZ task accumulating the messages and plots in memory before zipping (Fig. 9). The CPU utilisation for TRAIN is fairly small due to the low message rate, and the memory usage is comparatively high at 20% as the large batch of table rows is stored in memory for model training. The CPU utilization for the FIT workload is the least for all the application benchmarks due to the fact that it has the least rate at 500 *msg/sec*, and we have assigned exclusive an core to each of its tasks. TAXI has a low CPU usage, mostly at a 5% median, with a wide box (Figs. 8d, 11c 9d 10c) – this is due to its bi-modal distribution with low input rates at nights, with lower utilization, and high in the day with higher utilization. In general, we see that such a resource under-utilization strongly motivates the need for robust resource allocation strategies for IoT applications on DSPS.

## 8 Conclusion

In this paper, we have proposed *RIoTBench*, a novel benchmark suite for evaluating distributed stream processing systems for Internet of Things applications, which encompasses several emerging domains. Fast data platforms like DSPS are integral for the rapid decision making needs of IoT applications. Our proposed micro and application benchmarks help evaluate their efficacy using common tasks found in IoT domains, as well as fully-functional dataflows for pre-processing, statistical summarization and predictive analytics. These applications naturally fit into the OODA interaction model found in many IoT domains. These benchmarks are combined with four real-world data streams from Smart Grid, Smart Transportation, Urban Sensing and personal fitness domains of IoT, that are further spatially and temporally scaled to recreate the stream profiles of contemporary IoT deployments. The proposed benchmark has been validated for the highly-popular Apache Storm DSPS, and the performance metrics reported.

As future work, we would like to add event pattern detection and notifications as tasks to our benchmark suite to complete the representative categories. The benchmark can also be used to evaluate other popular DSPS such as Apache Spark Streaming and Flink. Incidentally, these tasks and applica-

tions we have provided have real and accurate business logic. Thus, they form a valuable library of tasks that can be used in both generic and IoT streaming applications. We are currently in the process of integrating customized versions of these benchmark applications into the IISc Smart Campus IoT project for smart water and power management <sup>7</sup>.

## Acknowledgments

We acknowledge detailed inputs provided by Tarun Sharma of NVIDIA Corp. and formerly from IISc in preparing this paper. The experiments on Microsoft Azure were supported through a grant from Azure for Research. We thank the reviewers of the Technology Conference on Performance Evaluation & Benchmarking (TPCTC), 2016, for their valuable comments to improve the benchmark suite.

## References

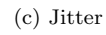
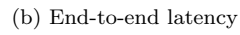
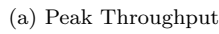
- [1] Sense your city: Data art challenge. Technical report, Data Canvas, 2015.
- [2] Xchart java library. <http://knowm.org/open-source/xchart/>, April 2015.
- [3] Dakshi Agrawal, Ali Butt, Kshitij Doshi, Josep-L Larriba-Pey, Min Li, Frederick R Reiss, Francois Raab, Berni Schiefer, Toyotaro Suzumura, and Yinglong Xia. Sparkbench—a spark performance testing suite. In *TPCTC*, pages 26–44. Springer, 2015.
- [4] Muhammad Intizar Ali, Feng Gao, and Alessandra Mileo. Citybench: a configurable benchmark to evaluate rsp engines using smart city datasets. In *ISWC*, 2015.
- [5] Noga Alon, Yossi Matias, and Mario Szegedy. The space complexity of approximating the frequency moments. In *ACM symposium on Theory of computing*, pages 20–29. ACM, 1996.
- [6] Arvind Arasu, Mitch Cherniack, Eduardo Galvez, David Maier, Anurag S Maskey, Esther Ryzkina, Michael Stonebraker, and Richard Tibbetts. Linear road: a stream data management benchmark. In *VLDB*, 2004.
- [7] Martin Arlitt, Manish Marwah, Gowtham Bellala, Amip Shah, Jeff Healey, and Ben Vandiver. Iotabench: An internet of things analytics benchmark. In *ICPE*, 2015.
- [8] Shivnath Babu and Jennifer Widom. Continuous queries over data streams. *ACM Sigmod Record*, 30(3):109–120, 2001.
- [9] Magdalena Balazinska, Hari Balakrishnan, Samuel R Madden, and Michael Stonebraker. Fault-tolerance in the borealis distributed stream processing system. *ACM TODS*, 2008.
- [10] Oresti Banos, Rafael Garcia, Juan A Holgado-Terriza, Miguel Damas, Hector Pomares, Ignacio Rojas, Alejandro Saez, and Claudia Villalonga. mhealthdroid: a novel framework for agile development of mobile health applications. In *International Workshop on Ambient Assisted Living*, pages 91–98. Springer, 2014.
- [11] Alain Biem, Eric Bouillet, Hanhua Feng, Anand Ranganathan, Anton Riabov, Olivier Verscheure, Haris Koutsopoulos, and Carlos Moran. Ibm infosphere streams for scalable, real-time, intelligent transportation services. In *ACM SIGMOD*, 2010.
- [12] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, July 1970.

---

<sup>7</sup>IISc Smart Campus Project, <http://smartx.cds.iisc.ac.in>

- [13] Paris Carbone, Stephan Ewen, Seif Haridi, Asterios Katsifodimos, Volker Markl, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. *IEEE Data Engineering Bulletin*, page 28, 2015.
- [14] Don Carney, Uğur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Greg Seidman, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Monitoring streams: a new class of data management applications. In *VLDB*, pages 215–226, 2002.
- [15] Patrik Cerwall. Ericsson mobility report. Technical report, Ericsson, 2016.
- [16] Ugur Cetintemel, Daniel Abadi, Yanif Ahmad, Hari Balakrishnan, M Balazinska, M Cherniack, J Hwang, W Lindner, S Madden, A Maskey, et al. The aurora and borealis stream processing engines. *Data Stream Management: Processing High-Speed Data Streams*, Springer-Verlag, pages 1–23, 2006.
- [17] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel Madden, Vijayshankar Raman, Frederick Reiss, and Mehul A. Shah. Telegraphcq: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.
- [18] Jianjun Chen, David J DeWitt, Feng Tian, and Yuan Wang. Niagaraqc: A scalable continuous query system for internet databases. *ACM SIGMOD Record*, 29(2):379–390, 2000.
- [19] Gianpaolo Cugola and Alessandro Margara. Processing flows of information: From data stream to complex event processing. *ACM Comput. Surv.*, 2012.
- [20] Brian Donovan and Daniel B. Work. Using coarse gps data to quantify city-scale transportation system resilience to extreme events. In *Transportation Research Board 94th Annual Meeting*, 2014.
- [21] Marianne Durand and Philippe Flajolet. Loglog counting of large cardinalities. In *European Symposium on Algorithms*, pages 605–617. Springer, 2003.
- [22] Wanling Gao, Yuqing Zhu, Zhen Jia, Chunjie Luo, Lei Wang, Zhiguo Li, Jianfeng Zhan, Yong Qi, Yongqiang He, Shiming Gong, et al. Bigdatabench: a big data benchmark suite from web search engines. *arXiv preprint arXiv:1307.0320*, 2013.
- [23] Ahmad Ghazal, Tilmann Rabl, Mingqing Hu, Francois Raab, Meikel Poess, Alain Crolotte, and Hans-Arno Jacobsen. Bigbench: towards an industry standard benchmark for big data analytics. In *ACM SIGMOD*, 2013.
- [24] Ling Gu, Minqi Zhou, Zhenjie Zhang, Ming-Chien Shan, Aoying Zhou, and Marianne Winslett. Chronos: An elastic parallel framework for stream benchmark generation and simulation. In *ICDE*, 2015.
- [25] Shengsheng Huang, Jie Huang, Jinqian Dai, Tao Xie, and Bo Huang. The hibench benchmark suite: Characterization of the mapreduce-based data analysis. In *IEEE ICDEW*, 2010.
- [26] C. Jennings, Z. Shelby, J. Arkko, A. Keranen, and C. Bormann. Media types for sensor measurement lists (senml). Technical Report draft-ietf-core-senml-04, Internet Engineering Task Force (IETF), 2016.
- [27] Rudolph Emil Kalman. A new approach to linear filtering and prediction problems. *Transactions of the ASME—Journal of Basic Engineering*, 82(Series D):35–45, 1960.
- [28] Chunhui Li and Robert Berry. Cepben: a benchmark for complex event processing systems. In *TPCTC*, pages 125–142. Springer, 2013.
- [29] Ruirui Lu, Gang Wu, Bin Xie, and Jingtong Hu. Stream bench: Towards benchmarking modern distributed stream computing frameworks. In *IEEE/ACM UCC*, 2014.
- [30] Zubair Nabi, Eric Bouillet, Andrew Bainbridge, and Chris Thomas. Of streams and storms. Technical report, IBM, 2014.
- [31] Raghunath Othayoth Nambiar and Meikel Poess. The making of tpc-ds. In *VLDB*, 2006.
- [32] Leonardo Neumeyer, Bruce Robbins, Anish Nair, and Anand Kesari. S4: Distributed stream computing platform. In *IEEE ICDMW*, 2010.

- [33] Charith Perera, Arkady Zaslavsky, Peter Christen, and Dimitrios Georgakopoulos. Context aware computing for the internet of things: A survey. *Communications Surveys & Tutorials*, 16(1):414–454, 2014.
- [34] J.R. Quinlan. Induction of decision trees. *Machine Learning*, 1(1):81–106, March 1986.
- [35] Anshu Shukla and Yogesh Simmhan. Benchmarking distributed stream processing platforms for iot applications. In *Technology Conference on Performance Evaluation & Benchmarking (TPCTC)*, 2016.
- [36] Yogesh Simmhan, Saima Aman, Alok Kumbhare, Rongyang Liu, Sam Stevens, Qunzhi Zhou, and Viktor Prasanna. Cloud-based software platform for data-driven smart grid management. *IEEE/AIP Computing in Science and Engineering*, July/August:1–11, 2013.
- [37] Sriskandarajah Suhothayan, Kasun Gajasinghe, Isuru Loku Narangoda, Subash Chaturanga, Srinath Perera, and Vishaka Nanayakkara. Siddhi: A second look at complex event processing architectures. In *ACM Workshop on Gateway Computing Environments*, 2011.
- [38] The Research Perspective Ltd. Cer smart metering project. Technical report, Commission for Energy Regulation, Ireland, 2012.
- [39] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, et al. Storm@ twitter. In *ACM SIGMOD*, pages 147–156, 2014.
- [40] Mark Wallis and Brent Wilson. Iot-connect: An industry-standard benchmarks for embedded systems. Technical report, EEMBC, 2017.
- [41] Gary Wolf. The data-driven life. *The New York Times Magazine*, 2010.
- [42] Matei Zaharia, Tathagata Das, Haoyuan Li, Scott Shenker, and Ion Stoica. Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters. In *USENIX Hot Cloud*, 2012.
- [43] A Zaslavsky. Internet of things and ubiquitous sensing. *Computing Now*, 2013.
- [44] Qunzhi Zhou, Yogesh Simmhan, and Viktor Prasanna. Knowledge-infused and consistent complex event processing over real-time and persistent streams. *Future Generation Computer Systems*, 2016.



29

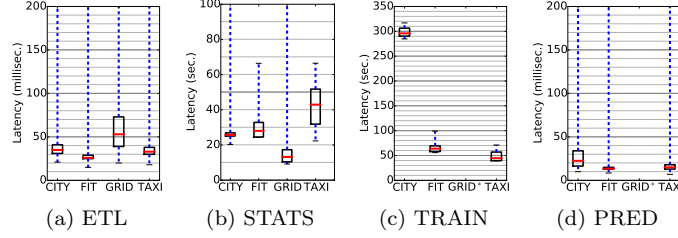


Figure 6: End-to-end latency plots for application benchmarks on workloads. ETL and PRED are in millisec and STATS and TRAIN are in sec. \*TRAIN and PRED are not run for GRID workload as it has only the target field, and no additional field to predict upon.

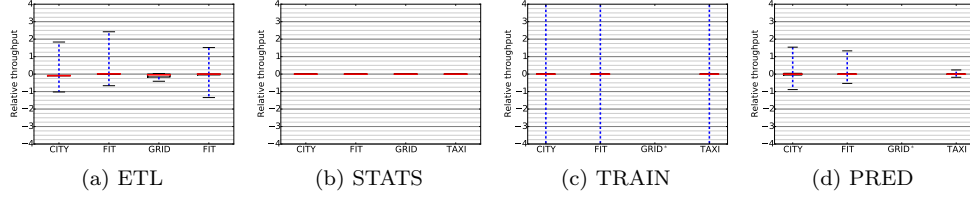


Figure 7: Jitter plots for application benchmarks on workloads. \*TRAIN and PRED are not run for GRID workload as it has only the target field, and no additional field to predict upon.

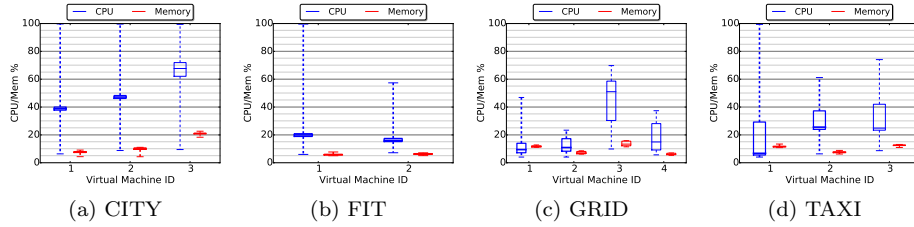


Figure 8: CPU and Memory utilization plots for *ETL* application benchmark on all workloads.

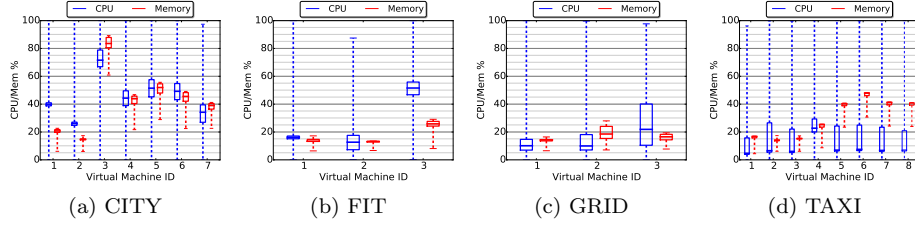


Figure 9: CPU and Memory utilization plots for *STATS* application benchmarks on all workloads.

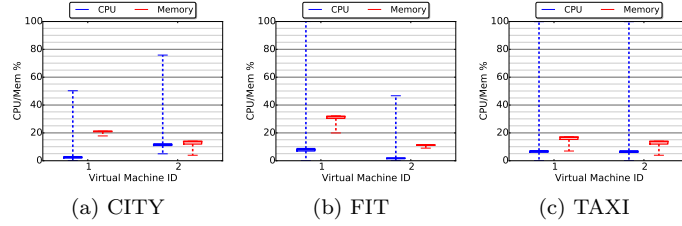


Figure 10: CPU and Memory utilization plots for *TRAIN* application benchmarks three workloads, CITY FIT and TAXI. GRID workload is not used as it has only the target field, and no additional field to predict upon.

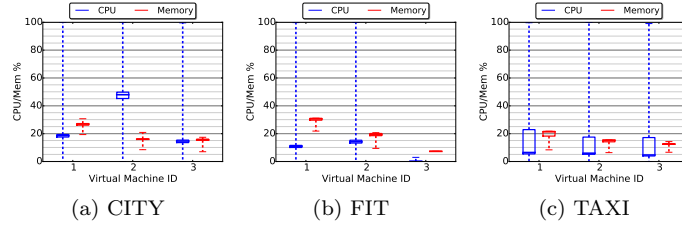


Figure 11: CPU and Memory utilization plots for *PRED* application benchmarks on three workloads, CITY FIT and TAXI. GRID workload is not used as it has only the target field, and no additional field to predict upon.

## A Configurations used in Application Dataflows

Table 4: Attributes Used in Tasks of the ETL Application

Task	CITY	FIT	GRID	TAXI
ANN*	location, sensor type	age, gender	tariff allocation, sme allocation, stimulus allocation	driver name, city, company
BLF	source	N/A †	meterid	taxi identifier
INP	temperature, humidity, light, dust, airquality raw	acceleration chest, arm, ankle X/Y/Z, ecg 1	energyConsumed	N/A ‡
RGF	temperature, humidity, light, dust, airquality raw	acceleration chest, arm, ankle X/Y/Z, ecg 1	energyConsumed	trip time in sec, trip distance, fare amount, surcharge, mta tax, tip amount, tolls amount, total amount

\* Annotation attributes that are added to the dataset by ANN, either provided with the dataset or synthetically

† No fields were used for the particular task with the dataset because the number of unique subjects is very less (10) for FIT thus not requires BLF.

‡ Interpolation of values over different Taxi trips is not meaningful.

Table 5: Attributes Used in Tasks of the STATS Application

Task	CITY	FIT	GRID	TAXI
AVG	temperature, humidity, light, dust, airquality raw	acceleration chest, arm, ankle X/Y/Z, ecg 1/2	energyConsumed	trip time in sec, trip distance, fare amount, surcharge, mta tax, tip amount, tolls amount, total amount
DAC	temperature	ecg 1	energyConsumed	N/A †
SLR	temperature, humidity, light, dust, airquality raw	acceleration chest, arm, ankle X/Y/Z, ecg 1/2	energyConsumed	trip time in sec, trip distance, fare amount, surcharge, mta tax, tip amount, tolls amount, total amount

† No fields were used for the particular task with the dataset because DAC over individual Taxi trips is not meaningful.

Table 6: Attributes Used in Tasks of the PRED Application

Task	CITY	FIT	GRID	TAXI
AVG	airquality raw	ecg 1	N/A †	fare amount
DTC	$\mathcal{F}(\text{temperature, humidity, light, dust, airquality raw}) \rightarrow \{C1   C2   C3   C4\}^*$	$\mathcal{F}(\text{acceleration chest, arm, ankle X/Y/Z, ecg 1}) \rightarrow \{C1   C2   C3   C4\}^*$	N/A †	$\mathcal{F}(\text{trip time in sec, trip distance, fare amount}) \rightarrow \{C1   C2   C3   C4\}^*$
MLR	$\mathcal{F}(\text{temperature, humidity, light}) \rightarrow \text{airquality raw}$	$(\text{acceleration chest, arm, ankle X/Y/Z}) \rightarrow \text{ecg 1}$	N/A †	$\mathcal{F}(\text{trip time in secs, trip distance}) \rightarrow \text{fare amount}$

\* Classes used for prediction by DTC task

† No fields were used for the particular task with the dataset as GRID is univariate whereas DTC and MLR tasks require multiple fields.



Table 7: Attributes Used in Tasks of the TRAIN Application

Task	CITY	FIT	GRID	TAXI
DTT	temperature, humidity, light, airquality raw	humidity, dust, acceleration arm, ankle ecg 1	chest, X/Y/Z, N/A <sup>†</sup>	trip time in sec, trip distance, fare amount
MLT	temperature, humidity, light, airquality raw	humidity, dust, acceleration arm, ankle ecg 1	chest, X/Y/Z, N/A <sup>†</sup>	trip time in sec, trip distance, fare amount

<sup>†</sup> No fields were used for the particular task with the dataset as GRID is univariate whereas DTT and MLT tasks require multiple fields.