Benchmarking Distributed Stream Processing Platforms for IoT Applications

Anshu Shukla(✉) and Yogesh Simmhan

Indian Institute of Science, Bangalore, India
shukla@grads.cds.iisc.ac.in, simmhan@cds.iisc.ac.in

Abstract. Internet of Things (IoT) is a technology paradigm where millions of sensors monitor, and help inform or manage, physical, environmental and human systems in real-time. The inherent closed-loop responsiveness and decision making of IoT applications makes them ideal candidates for using low latency and scalable stream processing platforms. Distributed Stream Processing Systems (DSPS) are becoming essential components of any IoT stack, but the efficacy and performance of contemporary DSPS have not been rigorously studied for IoT data streams and applications. Here, we develop a benchmark suite and performance metrics to evaluate DSPS for streaming IoT applications. The benchmark includes 13 common IoT tasks classified across functional categories and forming micro-benchmarks, and two IoT applications for statistical summarization and predictive analytics that leverage various dataflow patterns of DSPS. These are coupled with stream workloads from real IoT observations on smart cities. We validate the benchmark for the popular Apache Storm DSPS, and present the results.

Keywords: Stream processing · Benchmark · Workload · Internet of Things · Smart cities · Fast data · Big Data · Velocity · Distributed systems

1 Introduction

Internet of Things (IoT) is a technology paradigm where ubiquitous sensors numbering in the billions will be able to monitor physical infrastructure, humans 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. Besides affordable sensing and pervasive communications, Cloud and Big Data platforms have contributed to this rapid growth.

Currently, the IoT applications are often manifest in vertical domains, such as demand-response optimization and outage management in *smart grids* [5], or fitness and sleep tracking by *smart watches and health bands* [19]. 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, where sensors communicate time-series observations of the system to a Cloud data center for analysis, and the analytics drive recommendations that are enacted on, or

R. Nambiar and M. Poess (Eds.): TPCTC 2016, LNCS 10080, pp. 90–106, 2017.
DOI: 10.1007/978-3-319-54334-5_7

notified to, the system to improve it, which is again observed and so on. In fact, this *closed-loop* responsiveness is an essential characteristic of IoT applications.

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. This complements *Fog Computing* that puts the onus on edge devices to collaboratively collect, process and analyze data with low latency by reduced reliability.

Recent *Big Data platforms* like Storm [18], Flink [2] and Spark [20] provide an intuitive programming model for composing and executing scalable streaming applications on commodity clusters and Clouds. These *Distributed Stream Processing Systems (DSPS)* are becoming essential components of any IoT stack to support online analytics for IoT applications. In fact, reference IoT solutions from Cloud providers[1,2] include their own stream and event processing engines.

Shared-memory stream processing systems [9] have been investigated for wireless sensor networks, with community benchmarks being proposed [6]. But there has not been a detailed review of, or benchmarks for, *distributed* stream processing for IoT domains. In particular, the efficacy of contemporary DSPS, originally designed for web and social network traffic [18], have not been rigorously studied for *IoT data streams and applications*. We address this gap here.

We develop a benchmark suite for DSPS to evaluate their effectiveness for streaming IoT applications. The proposed workload is based on common building-block tasks observed in various IoT domains for real-time decision making, and the input streams are sourced from real IoT observations from smart cities.

Specifically, we make the following contributions:

1. We classify different characteristics of streaming applications and their data sources, in Sect. 3. We propose categories of tasks that are essential for IoT applications and the key features that are present in their input data streams.
2. We identify performance metrics of DSPS that are necessary to meet the latency and scalability needs of streaming IoT applications, in Sect. 4.
3. We propose an IoT Benchmark for DSPS based on representative *micro-benchmark tasks*, drawn from the above categories, in Sect. 5. Further, we design two reference IoT applications – for *statistical analytics* and *predictive analytics* – composed from these tasks. We also offer real-world streams with different distributions on which to evaluate them.
4. We run the benchmark for the popular Apache Storm DSPS, and present empirical results for the same in Sect. 6.

[1] https://aws.amazon.com/iot/how-it-works/.
[2] https://www.microsoft.com/en-in/server-cloud/internet-of-things/overview.aspx.

2 Background and Related Work

Early data stream management systems (DSMS) were motivated by sensor network applications, that have similarities to IoT [9]. They supported continuous query languages with operators such as join, aggregators similar to SQL, but with a temporal dimension using windowed-join operations. These have distributed implementations [8] and have evolved to complex event processing (CEP).

Current DSPS like Apache Storm and Apache Spark Streaming [18,20] leverage 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. They do not support native query operators and instead allow users to plug in their own logic composed as dataflow graphs executed across a cluster. 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.

Work on DSMS spawned the Linear Road Benchmark (LRB) [6] that was proposed as an application benchmark. In the scenario, DSMS had to evaluate toll and traffic queries over event streams from a virtual traffic monitoring system, with parallels to current smart transportation. However, there have been few studies or community efforts on benchmarking DSPS, other than research evaluations against popular DSPS like Storm or Spark. These papers define their own metrics of success – typically just throughput and latency – and use generic workloads like the Enron email dataset and custom micro-benchmarks [15].

Stream Bench [14] has proposed 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. It covers different dataflow patterns and common tasks like grep and wordcount. While useful as a generic streaming benchmark, it does not consider aspects unique to IoT applications and streams. SparkBench [3] is specific to Spark, and includes four categories of applications from domains spanning Graph analysis and SQL queries, and one application for Spark Streaming. The benchmark metrics include CPU, memory, disk and network IO, with the goal of identifying tuning parameters to improve Spark's performance.

In contrast, the goal for this paper is to develop 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 data streams observed in IoT domains. This allows for the performance of DSPS to be independently and reproducibly verified for IoT applications.

There has been a slew of Big Data benchmarks for processing high volume (i.e., MapReduce-style) and enterprise/web data, that complement our work. The *Yahoo Cloud Serving Benchmark (YCSB)* [10] was developed to compare different key-value stores on the Cloud. *Hibench* [13] 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. This is a general purpose workload for MapReduce platforms at large. *BigBench* [12] uses a synthetic generator to simulate online retail enterprise data. It combines structured data from the TPC-DS benchmark [16], semi-structured data on user clicks, and unstructured data from product reviews. Queries cover data *velocity* by processing periodic data refreshes, *variety* by including free-text reviews, and *volume* by querying over a large click logs. We take a similar approach to benchmark fast data platforms, targeting the IoT domain and using real public data streams.

There has been some recent work on benchmarking IoT applications. 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 using a Markov chain model for scaling the time series. It uses a limited number of inputs to ensure that important statistical properties of the stream is retained in the generated data. This has been demonstrated for smart meter data. Their emphasis is on the data characteristics and content, which supplements our focus on the systems aspects of the 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.

3 Characteristics of Streaming IoT Applications

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. Subsequently in this section, we also categorize IoT tasks, applications and data streams based on the domain requirements. Together, these offer a search space for defining workloads that meaningfully and comprehensively validate IoT applications on DSPS.

3.1 Dataflow Composition Semantics

DSPS applications are commonly composed as a *dataflow graph*, where vertices are user provided *tasks* and directed edges are refer to *streams of messages* that can pass between them. *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.

Selectivity ratio, also called *gain*, is the number of output messages emitted by a task on consuming a unit input message, expressed as σ = *input rate* : *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.

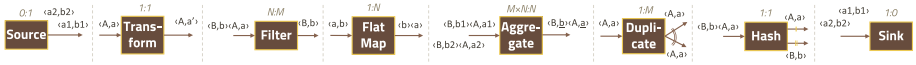


Fig. 1. Common task patterns and semantics in streaming applications.

There are message generation, consumption and routing semantics associated with tasks and their dataflow composition. Figure 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 tasks react to the output messages from the application, say, by storing it or sending an external notification.

`Transform` tasks, sometimes called *Map* tasks, 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 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 ($\sigma = N:1$).

When a task has multiple outgoing edges, routing semantics on the dataflow control 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.

Tasks may be *data parallel*, in which case, it may be allocated multiple 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 *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 Input 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 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 may be *bursts* of data coming from a single sensor, or a coordinated set of sensors. A *saw-tooth* behavior may be seen in the ramp-up/-down before/after specific events. *Normal* distribution are seen with diurnal (day vs. night) stream sources, with *bi-modal* variations capturing peaks during the morning and evening periods of human activity.

3.3 Categories of IoT Tasks and Applications

Here, we attempt to categorize common IoT processing and analytics tasks that are performed over real-time data streams to support domain applications.

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. Industry-standard formats that are popular for IoT domains include CSV, XML and JSON text formats, EXI and CBOR binary formats, and serialization protocols like Google's Protocol Buffer and Apache Thrift.

Filter. Messages may require to be filtered based on specific attribute values present in them, as part of 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 and band-pass filters that test an attribute's *numerical value ranges* are common, and are both compact to model and fast to execute. Since IoT event rates may be high, more efficient Bloom filters may also be used to process *discrete values* with low space complexity but with a small fraction of false positives.

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 as 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.

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. The *training* itself can be an online task that is part of an application. For e.g., linear regression use statistics to predict uni- or multivariate attribute values. Classification algorithms like decision trees and neural networks can be trained to map discrete values to a category, which may lead to specific actions taken on the system.

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 returned 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), or a join over two streams based on a common attribute value. Complex Event Processing (CEP) engines [17] may be embedded within the DSPS task to match these 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 either at the client, 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 libraries like D3.js or JFreeChart may be used for this.

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.

The tasks from the above categories, along with other domain-specific tasks, are composed together to form streaming IoT dataflows. 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. 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. 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 succeeded by visualizations tasks in the dataflow to present it to end-users and decision makers.

Prediction and Pattern Detection applications 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 trends based on current behavior that require preemptive actions. For e.g., an unsustainable growing load on a power grid cause load to be shed preemptively, or a detection that the heart-rate from a fitness watch is very high may trigger a treadmill to slow down.

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 be notified. For e.g., the need for load shedding in the power grid may map to specific residents to request the curtailment from, or the need to reduce physical activities may lead to a treadmill being notified to reduce the speed.

3.4 IoT Data Stream Characteristics

IoT data streams are often generated by sensors that observe physical systems or the environment. As a result, they are typically time-series data that are generated periodically. The sampling rate for these sensors may vary 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. As a result, we may encounter a wide range of input throughputs from 10^{-2} to 10^{5} messages/sec.

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. Example smart meters may collecting 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. For e.g., sensors at businesses may match the former while traffic flow sensors may match the latter.

4 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 task is the time in seconds it took for that task to process one or more inputs to generate that message. When we consider the *average latency* $\overline{\lambda}$ 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 ω^i is the input throughput for a dataflow with selectivity σ. 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.

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 between the expected and observed output throughput, defined for a time interval t as, $J_t = \frac{\omega^o - \sigma \times \omega^i}{\sigma \times \overline{\omega^i}}$, 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 $\overline{\omega^i}$. In an ideal case, jitter will tend towards zero.

CPU and Memory Utilization. Streaming IoT dataflows are expected to be resource intensive, and the ability of the DSPS to use distributed resources with minimal overhead is important. This also affects the VM resources used and price to be paid to run the application on the DSPS. 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.

5 Proposed Benchmarks and Workload

We propose IoT benchmark workloads to help evaluate the metrics discussed before for various DSPS. The benchmarks have two parts: the dataflow logic that is executed on the DSPS and the input data streams that they consume.

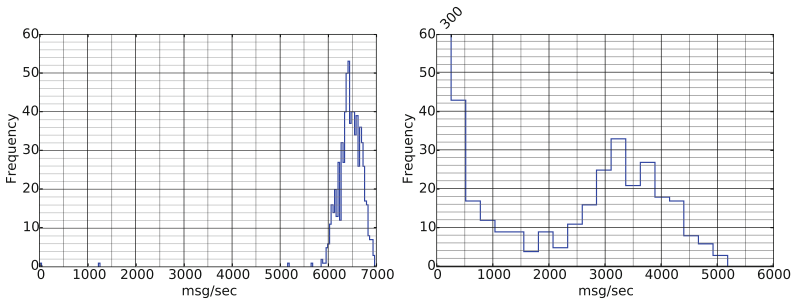
5.1 IoT Input Stream Workloads

Sense your City (CITY) [1]. This is an *urban environmental monitoring* project[3] that crowd-sourced deployment of sensors at 7 cities across 3 continents in 2015, with about 12 sensors per city. Five timestamped observations: temperature, humidity, ambient light, dust and air quality, are reported every minute by a sensor along with the sensor ID and location. Besides urban sensing, this real-world data also captures the vagaries crowd-sourcing for IoT (Table 1).

Table 1. Smart Cities data stream features and rates at 1000× scaling

Dataset	Attributes	Format	Size (bytes)	Peak rate (msg/sec)	Distribution
CITY [1]	9	CSV	100	7,000	Normal
TAXI [11]	10	CSV	191	4,000	Bimodal

We use a single logical stream that combines the data from all 90 sensors. Since practical deployments of environmental sensing can easily extend to thousands of sensors per city, we use a temporal scaling of 1000× the native input rate to simulate a deployment of 90,000 sensors. Figure 2a shows a narrow normal distribution of the event rate centered at 6,400 msg/sec with a peak of 7,000 msg/sec. We use 7 days of data from 27 Jan to 2 Feb, 2015 for our benchmark.



(a) CITY @1000× *msg/sec* (b) TAXI @1000× *msg/sec*

Fig. 2. Frequency distribution of input throughputs for CITY and TAXI streams at 1000× temporal scaling used for the benchmark runs.

NYC Taxi cab (TAXI) [11]. 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[4]. A message is generated when a taxi completes a

[3] http://map.datacanvas.org.
[4] http://www.debs2015.org/call-grand-challenge.html/.

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.

Considering that events may be generated from the GPS sensors periodically rather than only at the end of the trip, we use a temporal scaling factor of 1000× for our workload. 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.

5.2 IoT Micro-benchmarks

We propose a suite of common micro-benchmark tasks that span various IoT categories and types of streaming task patterns as well. Their goal is to evaluate the performance of the DSPS for individual IoT tasks, using the *peak input throughput* that they can sustain on a unit computing resource as the performance measure. This offers a baseline for comparison with other DSPS, as well as when these tasks are used in application benchmarks with variable input rates (Table 2).

Table 2. IoT micro-benchmark tasks with different IoT categories and DSPS patterns

Task name	Code	Category	Pattern	σ ratio	State
XML parsing	XML	Parse	Transform	1:1	No
Bloom filter	BLF	Filter	Filter	1:0/1	No
Average	AVG	Statistical	Aggregate	N:1	Yes
Distinct appox. count	DAC	Statistical	Transform	1:1	Yes
Kalman filter	KAL	Statistical	Transform	1:1	Yes
Second order moment	SOM	Statistical	Transform	1:1	Yes
Decision tree classify	DTC	Predictive	Transform	1:1	No
Multi-variate linear reg.	MLR	Predictive	Transform	1:1	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 query	ATQ	IO	Source/transform	1:1	No
MQTT publish	MQP	IO	Sink	1:1	No

We include a single XML parser as a representative parsing operation within our suite. The Bloom filter is a more practical filter operation for large discrete datasets, and we prefer that to a simple value range filter. We have several statistical analytics and aggregation tasks. These span simple averaging over a single attribute value to and second order moments over time-series values, to Kalman filter for denoising of sensor data and approximate count of distinct values for large discrete attribute values.

Predictive analytics using a multi-variate linear regression model that is trained offline and a sliding window univariate model that is trained online are included. A decision tree machine learning for discrete attribute values is also used for classification, based on offline training. Lastly, we have several IO tasks for reading and writing to Cloud file and NoSQL storage, and to publish to an MQTT publish-subscribe broker for notifications. We see that these tasks capture different dataflow patterns like transform, filter, aggregate and flat map.

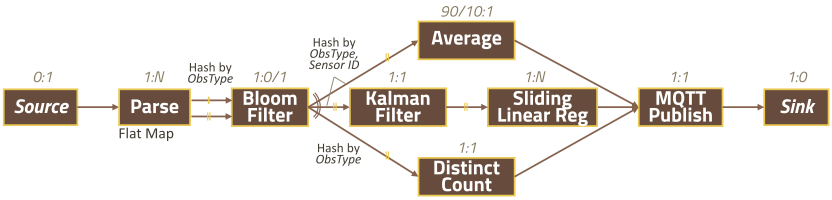
5.3 IoT Application Benchmarks

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 proposed earlier are naturally composable into application benchmarks that satisfy the requirements of a OODA decision making loop.

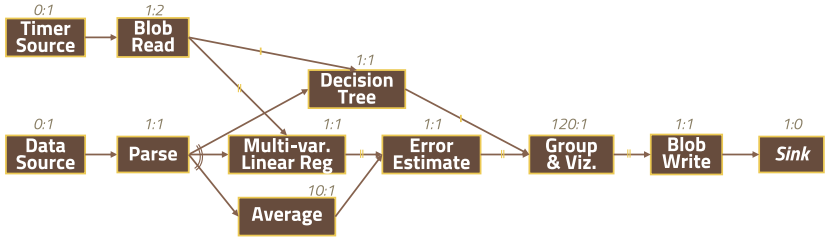
We propose application benchmarks that capture two common IoT scenarios: a *Data pre-processing and Statistical summarization (STATS)* application and a *Predictive Analytics (PRED)* application. STATS (Fig. 3a) ingests incoming data streams, performs data filtering of outliers on individual observation types using a Bloom filter, and then does three concurrent types of statistical analytics on observations from individual sensor/taxi IDs: sliding Average over a 90/10 event window for CITY/TAXI (∼15 min native time window), Kalman filter for smoothing followed by a sliding window linear regression, and an approximate count of distinct readings. The outcomes from these statistics are published by an MQTT task, which can separately be subscribed to and visualized on a client.

The PRED dataflow captures the lifecycle of online prediction and classification to drive visualization and decision making for IoT applications. It parses incoming messages and forks it to a decision tree classifier and a multi-variate regression task. The decision tree uses a trained model to classify messages into classes, such as good, average or poor air quality, based on one or more of their attribute values. The linear regression uses a trained model to predict an attribute value in the message using several others. It then estimates the error $\frac{|p-o|}{\bar{o}}$ between the predicted and observed value, normalized by the sliding average of the observations. These outputs are then grouped and plotted, and the file written to Cloud storage for hosting on a portal. One realistic addition is the use of a separate stream to periodically download newly trained classification and regression models from Cloud storage, and push them to the prediction tasks.

As such, these applications leverage many of the compositional capabilities of DSPS. The dataflows include *single and dual sources*, tasks that are *composed sequentially and in parallel*, *stateful and stateless* tasks, and *data parallel tasks* allowing for concurrent instances. The initial parse task for STATS uses a *flat map* pattern to create observation-specific streams. These are further grouped by their observation type using a *hash pattern* and passed to task instances.



(a) Pre-processing & statistical summarization dataflow (STATS)



(b) Predictive Analytics dataflow (PRED)

Fig. 3. Application benchmarks composed using the micro-benchmark tasks.

6 Evaluation of Proposed Benchmarks

We implement the 13 micro-benchmarks as generic Java tasks that can consume and produce objects[5]. We validate our proposed benchmark by composing and running these dataflows on the popular Apache Storm open source DSPS.

In Storm, each task logic is wrapped by a *bolt* that invokes the task for each incoming tuple and emits response tuples. 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 source task (*spout*) that replays events from a CSV file with a scaling factor. We generate random integers as tuples at a constant peak rate for the micro-benchmarks, and replay the original CITY and TAXI datasets at 1000× scaling for the applications.

We use Apache Storm 1.0.0 running on OpenJDK 1.7 and CentOS, and hosted on Microsoft Azure Cloud Virtual Machines (VMs). For the micro-benchmarks, Storm runs the benchmark task on one exclusive `D1` VM (1-core Intel Xeon E5@2.2 GHz, 3.5 GiB RAM, 50 GiB SSD), while the source and sink tasks and the master service run on a `D8` VM (8-core Intel Xeon E5@2.2 GHz, 28 GiB RAM, 400 GiB SSD). The large VM for the supporting services ensures that they are not the bottleneck when benchmarking the peak task rate on 1 VM. For the STATS and PRED application benchmark, we use `D8` VMs for all the tasks of the dataflow, while reserving additional `D8` VMs to exclusively run the supporting service. Each experiment runs for ∼10 min, which translates to about 7 days of event data for the CITY and TAXI datasets at 1000× scaling[6].

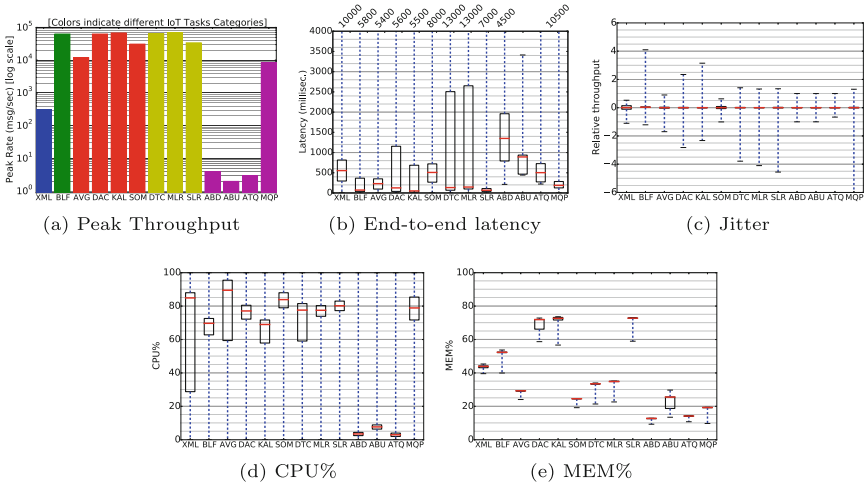
[5] https://github.com/dream-lab/bm-iot.

[6] Application runtime $= \frac{7\,\text{days} \times 24\,\text{h} \times 60\,\text{min} \times 60\,\text{s}}{1000 \times scaling} secs = 10.08\,\text{min}$.

6.1 Micro-benchmark Results

Figure 4 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. 4a 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 BLF, DAC, KAL, DTC and MLR. XML parsing is highly CPU bound and has a peak throughput of only 310 msg/sec, and the Azure operations are I/O bound on the Cloud service and even slower.

The inverse of the peak sustained throughput gives the *mean latency*. 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. 4b) varies widely up to 2,600 ms for Q3. This wide variability could be because of non-uniform task execution times due to which slow executions queue up incoming tuples 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.



(a) Peak Throughput (b) End-to-end latency (c) Jitter

(d) CPU% (e) MEM%

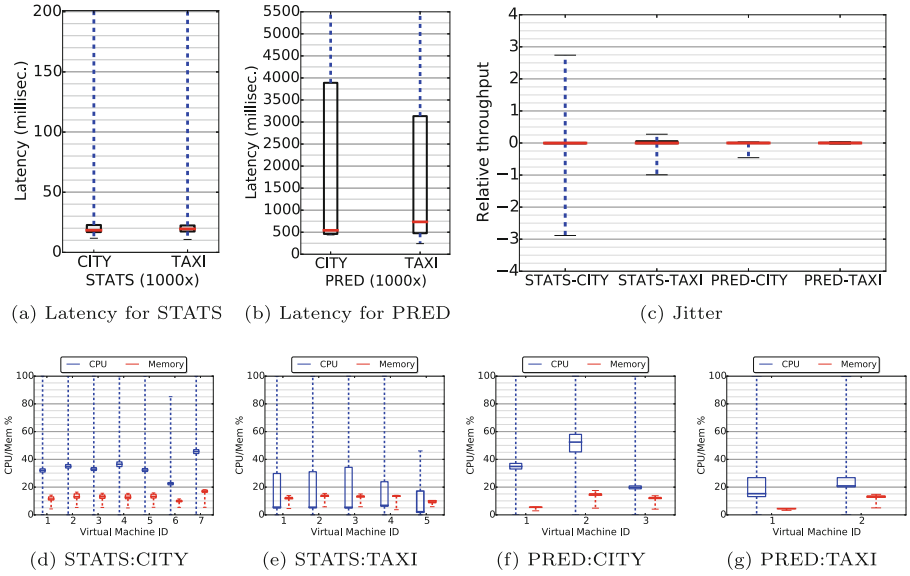
Fig. 4. Performance of micro-benchmark tasks for integer input stream at peak rate.

The box-plot for *jitter* (Fig. 4c) 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.

The box plots for CPU utilization (Fig. 4d) shows the single-core VM effectively used at 70% or above in all cases except for the Azure tasks that are I/O bound. The memory utilization (Fig. 4e) 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.

6.2 Application Results

The STATS and PRED application benchmarks are run for the CITY and TAXI workloads at 1000× their native rates, and the performance plots shown in Fig. 5. 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. The peak rates supported by the tasks in STATS is much higher than the input rates of CITY and TAXI. So the latency box plot for STATS is tightly bound (Fig. 5a) and its median much lower at 20 ms compared to the end-to-end latency of the tasks at their peak rates. The jitter is also close to zero in all cases. So Storm can comfortably support STATS for CITY and TAXI on 7 and 5 VMs, respectively. The distribution of VM CPU utilization is also modest for STATS. CITY has a 35% median with a narrow box (Fig. 5d), while TAXI has a low 5% median



(a) Latency for STATS (b) Latency for PRED (c) Jitter

(d) STATS:CITY (e) STATS:TAXI (f) PRED:CITY (g) PRED:TAXI

Fig. 5. End-to-end latency and Jitter (top), and CPU and Memory utilization (bottom) plots for STATS and PRED application benchmarks on CITY and TAXI workloads.

with a wide box (Fig. 5e) – this is due to its bi-modal distribution with low input rates, hence utilization, at nights, and high rates and utilization in the day.

For PRED, we see that the latency box plot is much wider, and the median end-to-end latency is between 500–700 ms for CITY and TAXI (Fig. 5b). This reflects the variability in task execution times for the WEKA tasks, DTC and MLR, which was observed in the micro-benchmarks too. The Azure blob upload also adds to the absolute increase in the end-to-end time. The jitter however remains close to zero, indicating sustainable performance. The CPU utilization is also higher, reflecting its more complex task logic relative to STATS.

7 Conclusion

In this paper, we have proposed a novel application benchmark for evaluating DSPS for IoT domains. These help evaluate common IoT tasks, as well as fully-functional applications for summarization and predictive analytics using with two real-world workloads from smart cities. The benchmark has been validated for the popular Apache Storm DSPS, and the performance metrics presented.

Acknowledgement. This work was supported by grants from the Robert Bosch Center for Cyber Physical Systems (RBCCPS) at IISc, DeitY and Microsoft Azure.

References

1. Data Canvas Dataset. http://datacanvas.org/sense-your-city/
2. Apache Flink. https://flink.apache.org/features.html/, April 2015
3. Agrawal, D., et al.: SparkBench – a spark performance testing suite. In: Nambiar, R., Poess, M. (eds.) TPCTC 2015. LNCS, vol. 9508, pp. 26–44. Springer, Cham (2016). doi:10.1007/978-3-319-31409-9_3
4. Ali, M.I., Gao, F., Mileo, A.: CityBench: a configurable benchmark to evaluate RSP engines using smart city datasets. In: Arenas, M., et al. (eds.) ISWC 2015. LNCS, vol. 9367, pp. 374–389. Springer, Heidelberg (2015). doi:10.1007/978-3-319-25010-6_25
5. Aman, S., Simmhan, Y., Prasanna, V.K.: Holistic measures for evaluating prediction models in smart grids. IEEE TKDE **27**(2), 475–488 (2015)
6. Arasu, A., Cherniack, M., Galvez, E., Maier, D., Maskey, A.S., Ryvkina, E., Stonebraker, M., Tibbetts, R.: Linear road: a stream data management benchmark. In: VLDB (2004)
7. Arlitt, M., Marwah, M., Bellala, G., Shah, A., Healey, J., Vandiver, B.: IoTAbench: an internet of things analytics benchmark. In: ICPE (2015)
8. Balazinska, M., Balakrishnan, H., Madden, S.R., Stonebraker, M.: Fault-tolerance in the borealis distributed stream processing system. ACM TODS (2008)
9. Chen, J., DeWitt, D.J., Tian, F., Wang, Y.: Niagaracq: a scalable continuous query system for internet databases. ACM SIGMOD Rec. **29**(2), 379–390 (2000)
10. Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with YCSB. In: ACM SoCC, pp. 143–154. ACM (2010)
11. Donovan, B., Work, D.B.: Using coarse GPS data to quantify city-scale transportation system resilience to extreme events. In: Transportation Research Board 94th Annual Meeting (2014)

12. Ghazal, A., Rabl, T., Hu, M., Raab, F., Poess, M., Crolotte, A., Jacobsen, H.A.: Bigbench: towards an industry standard benchmark for big data analytics. In: ACM SIGMOD (2013)
13. Huang, S., Huang, J., Dai, J., Xie, T., Huang, B.: The Hibench benchmark suite: characterization of the MapReduce-based data analysis. In: IEEE ICDEW (2010)
14. Lu, R., Wu, G., Xie, B., Hu, J.: Stream bench: towards benchmarking modern distributed stream computing frameworks. In: IEEE/ACM UCC (2014)
15. Nabi, Z., Bouillet, E., Bainbridge, A., Thomas, C.: Of streams and storms. Technical report, IBM (2014)
16. Nambiar, R.O., Poess, M.: The making of TPC-DS. In: VLDB (2006)
17. Suhothayan, S., Gajasinghe, K., Loku Narangoda, I., Chaturanga, S., Perera, S., Nanayakkara, V.: Siddhi: a second look at complex event processing architectures. In: ACM Workshop on Gateway Computing Environments (2011)
18. Toshniwal, A., Taneja, S., Shukla, A., Ramasamy, K., Patel, J.M., Kulkarni, S., Jackson, J., Gade, K., Fu, M., Donham, J., et al.: Storm@ twitter. In: ACM SIGMOD, pp. 147–156 (2014)
19. Wolf, G.: The data-driven life. The New York Times Magazine (2010)
20. Zaharia, M., Das, T., Li, H., Shenker, S., Stoica, I.: Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters. In: USENIX Hot Cloud (2012)