BatchIT: Intelligent and Efficient Batching for IoT Workloads at the Edge

Guoxi Wang, Ryan Hildebrant, Andrew Chio, Sharad Mehrotra, Nalini Venkatasubramanian
Dept. of Computer Science, University of California, Irvine
guoxiw1@uci.edu, rhildebr@uci.edu, achio@uci.edu, sharad@ics.uci.edu, nalini@ics.uci.edu

***Abstract*—Next-generation stream processing systems for community scale IoT applications must handle complex non-functional needs, e.g. scalability of input, reliability/timeliness of communication and privacy/security of captured data. In many IoT settings, efficiently batching complex workflows remains challenging in resource-constrained environments. High data rates, combined with real-time processing needs for applications, have pointed to the need for efficient edge stream processing techniques. In this work, we focus on designing scalable edge stream processing workflows in real-world IoT deployments where performance and privacy are key concerns. Initial efforts have revealed that privacy policy execution/enforcement at the edge for intensive workloads is prohibitively expensive. Thus, we leverage intelligent batching techniques to enhance the performance and throughput of streaming in IoT smart spaces. We introduce BATCHIT, a processing middleware based on a smart batching strategy that optimizes the trade-off between batching delay and the end-to-end delay requirements of IoT applications. Through experiments with a deployed system we demonstrate that BATCHIT outperforms several approaches, including micro-batching and EdgeWise, while reducing computation overhead.**

***Index Terms*—Edge Computing, Stream Processing, Privacy**

I. INTRODUCTION

The widespread availability of low-cost sensing devices and ubiquitous network connectivity has led to the emergence of pervasive monitoring of physical spaces. This has attracted the interest of infrastructure providers, who are increasingly adopting IoT technologies to collect and analyze data for improving operational efficiency [1]. The adoption of IoT devices in building and facility management systems is gaining traction, with benefits such as enhanced energy efficiency [2].

With real-time IoT data, there is a growing need to scalably handle high volumes and sporadic arrival patterns to ensure timely and accurate analysis on workloads. Robust execution of IoT applications requires reliable and non-lossy workflows to capture and process data. IoT data from smart infrastructures reflect human activities and behaviors, which can contain sensitive and personally identifiable information [3], [4]. Such information can be misused by data consumers for unintended purposes, leading to potential privacy violations. Numerous privacy operations require substantial computational resources, and applications using live data have strict performance and latency standards, such as Service Level Agreements (SLAs).

Existing stream processing engines, e.g., Apache Kafka, Storm, Flink, and Spark Streaming enable the real-time processing of high-volume data streams at the edge. The next generation of IoT stream processing systems must consider approaches that explicitly consider privacy/security, along with performance and scalability. Architectures must support flexible integration of diverse mechanisms and policies to achieve a balance between performance, privacy/security at scale [5].

This paper addresses these composite concerns by focusing on designing scalable and efficient stream processing techniques with privacy in mind. We envision a general model for IoT data sharing in physical spaces involving three key components: data providers, data consumers, and data subjects [6]. Data providers generate and store IoT data, data consumers use this data to develop innovative services, and data subjects have control over their personal data and can choose to share it with specific data consumers and establish access policies.

Given the potential for data misuse by service providers, it is essential to limit data sharing [7]. This can be achieved by implementing privacy-preserving mechanisms, such as anonymization, aggregation, data encryption, and policy-based access, during data processing and sharing [8]. The requirements for privacy differ among applications and data subjects; the heavy I/O demands of the workload, combined with policy checks at the individual data subject level, result in performance challenges. A solution is to perform privacy computation at the edge, within the trust boundary of the data provider. This allows data providers to enforce data privacy policies and ensure that sensitive information is protected from unauthorized access or misuse [5].

In this paper, we propose a novel IoT data processing middleware called BATCHIT that integrates a range of novel methods at the edge. Recent work has shown the value and feasibility of data processing at the edge without connectivity to a cloud backend [9]–[14]. Our system leverages edge computing to enable efficient and scalable processing of IoT data while also ensuring data privacy and security. We model an edge environment as a stream processing engine that ingests raw data and processes it through a workflow, which we represent as a directed acyclic graph (DAG) [10], [11].

To enable privacy, BATCHIT applies a workflow of privacy operators to the data stream. We address the performance challenge by implementing a smart batching strategy that optimizes the trade-off between the batching delay and the end-to-end delay requirement posed by IoT applications. Our system computes appropriate batch sizes for data processing operations based on the computational capabilities of edge devices and delay requirements. We verify our results on a large-scale deployed system across a University campus to collect WiFi access data. Our experimental evaluation shows that BATCHIT achieves superior performance while meeting privacy needs, maintaining service-level performance agreements (SLA), and increasing throughput as compared to existing approaches. Key contributions in this work include:

- Measurement studies that indicate the need for intelligent batching in privacy-enabled IoT workflows (§2)
- Formulation of the intelligent batching problem for I/O intensive IoT workloads and efficient heuristics (§3)
- Design of efficient and practical batch-aware worker scheduling techniques deployment at the edge (§4)
- Design, implementation, and validation of BATCHIT in a long-running campus-wide deployment (§5)

II. Enabling Scalable Edge Streaming

To understand the challenges involved in efficiently processing IoT data-sharing workflows, we conducted a preliminary real-world deployment of IoT-enabled applications in at the University of California, Irvine campus with multiple instrumented buildings, or "smart spaces". We collaborated with the university's network infrastructure team, campus information security and privacy committees, and multiple university departments to design and develop a testbed with application workloads.

Our system consists of 200 buildings with 2,000 WiFi AP's, which produces several million daily records. In our setting, users can define sets of policies, which results in many unique or overlapping policies. A workflow was developed and deployed at the campus technology operations site where network infrastructure is managed; this workflow implemented **IoT operators** to appropriately process the captured data and routed this to multiple distributed consumers that utilized the data for executing smart IoT applications including room level occupancy, flow monitoring, COVID exposure alerts, building concierge etc. [15]–[18].

A. The Need for Intelligent Batching

In our initial deployment studies, we found that the cost of policy execution and enforcement at the edge for this I/O intensive workload is costly; and scaling to a large number of users in real-time environments was challenging. We found that batching is a natural solution to enhance performance and throughput of streaming and I/O intensive activities [9]. With this intuition in mind, we implemented a simple (naive) approach to support batching of computation. Our preliminary experiments were encouraging and illustrated that the throughput of the system can be significantly improved.

The key issue with batching is that it introduces latency for applications and additional overhead in the system. Different services require different service level agreements (SLAs) and often have strict restrictions on processing delays associated with their data [19], [20]. This implies that we needed different levels of batching in the workflows associated with applications/services. Another challenge to ensure application reliability is that the system works in a stable manner without dropping data items (due to out-of-memory issues). Furthermore, an operator's throughput exhibits unpredictability and varies with the number of input data items for processing. An important research question is that of determining *how to perform batching for complex workloads and workflows in an efficient manner so as to optimize throughput, support bounded latency, ensure reliability and reduce processing overheads.*

Several works study dynamic micro-batching techniques for improving the performance for workload tasks. Recently, [21] created a stream query processing system that allocates network and computational resources at the edge. It considers data locality, resource constraints and operator placement for stream queries. Stout [22] implements dynamic batching for increasing the throughput of cloud applications, but only considers average performance values and the batch parameter cannot be tuned to comply with user-defined SLOs. GrandSLAm [23] uses dynamic batching for processing microservice requests and increases the system throughput while complying with SLOs, which accounts only for synthetic Poisson workloads, and not generic/bursty ones. AWS Batch [24] automatically provides the system with the optimal number of resources based on its workload, but does not support serverless computing yet. Window-based approaches, such as [25] also build workflow graphs and adapt based on network load. However, this creates a large resource burden and is not practical for constrained edge deployments.

Empirical results from our operational system show that increasing the batch size of input data items leads to higher throughput due to amortized processing for each execution (blue line in Fig. 1). The batching profile for an anonymization operator (orange line) indicates that the overall latency at the operator is lower for larger batch sizes due to lowered upfront costs for each execution, but progressively scales sub-linearly due to the cost of aggregation increasing with the number of items in a batch. The goal then is to schedule an operator's execution when the number of data items in its input queue reaches a certain threshold, i.e. a planned batch size.

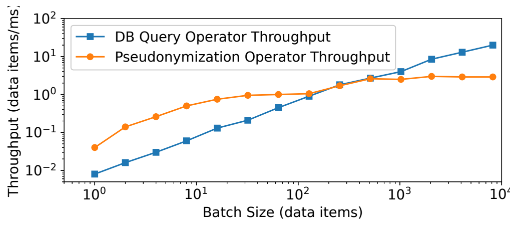


Fig. 1: Profiling batching effectiveness for operators

B. BATCHIT*: System Architecture and Workflow*

The intuition behind BATCHIT is to utilize batching of data items at the operator level to increase system throughput while satisfying the various latency requirements. We address the following questions for effective platform development: (1) What are appropriate batch sizes for operators to balance throughput and latency needs of various data producers and products?; (2) How do we keep track of operator queues and schedule the operator to execute with minimal overhead?; (3) How does the system adapt to the workload dynamicity?

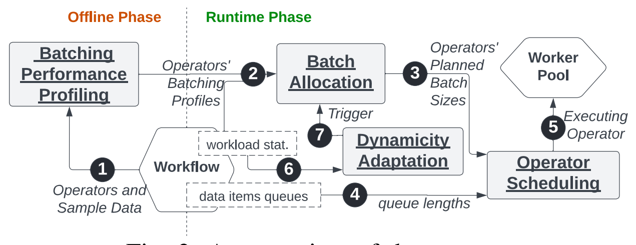


Fig. 2: An overview of the system

BATCHIT features three main components to address the above challenges. As shown in Fig. 2, the **Batch Allocation** module computes adequate batch sizes for operators to balance the throughput and end-to-end latency requirements, considering the operator's batch profile and current workload. The planned batch sizes are provided to the **Operator Scheduling** module, which monitors operator queue lengths in real-time and schedules the operator execution accordingly. The **Dynamicity Adaptation** module checks the data arrival rate at each operator and invokes batching adjustment if the current workload diverges significantly. Together, the above techniques will allow us to dynamically adjust batch sizes based on the computational capabilities of the edge devices and the end-to-end delay requirements of the application.

BATCHIT focuses on achieving a balance of throughput. We use an edge server to receive data streams from multiple sources (IoT devices, sensors) where sensor stream is associated with a data subject if it captures personal information. For example, in our Wi-Fi AP association data, the owner of the device that connects to the infrastructure is the data subject. Each data point that enters the stream that is associated with a data subject is transformed through a workflow of data processing and privacy operators to yield a result. We refer to this as a data product, which is in turn shared with service providers. In our model, we assume that data subjects provide data sharing policies upfront and that data policies remained fixed during a time window of the stream. Policies can be updated and changed, but we consider this an offline operation.

III. MODELING BATCHIT COMPONENTS AND EXECUTION

A. Workflows and Operators

A workflow is a rooted DAG $G = (V, E)$, with node $v_j \in V$ denoting a set of *operators* and directed edges $e_{i,j}$ denoting data flow from operator v_i to v_j, as seen in Fig. 3. We define *data items* as the input events, raw sensor readings, and any derivative forms computed afterwards in the system. In general, data items are processed starting at a source operator v_0 representing input data, and ends at sink operators $v_j^* \in V^* \subset V$ representing *data products* that end users/applications require. Each workflow in our model contains a single source, but possibly multiple data product sinks. Each operator v_j also maintains an input data item queue Q that buffers any received data items until a planned batch size is reached. *Selectivity* of an edge $e_{i,j}$ is defined as the ratio of input data items at v_i to output data items at v_j.

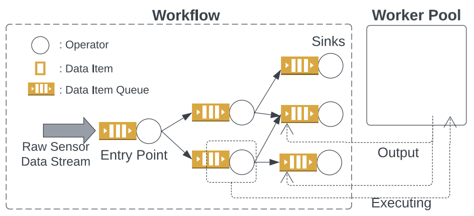


Fig. 3: Example of BATCHIT Workflow.

• *Stateful vs. Stateless Operator*: Maintaining a shared "state" between data items can affect processing and execution times for a stateful operator. Such operators may also not benefit from parallelism since data must be processed sequentially, and this can also present added overhead costs. For example, a differential privacy operator uses a privacy budget ϵ to track and control the cumulative privacy loss incurred during a sequence of data access or computations.

• *Blocking vs. Non-blocking Operator*: Blocking operators, such as anonymization operators that aggregate data, require the complete set of necessary inputs before they can begin processing - this can amplify delays in the workflow, particularly for larger anonymization tasks where the data volume is higher. In contrast, non-blocking operators like those used for anonymizing IoT data identifiers, are designed to process data immediately upon its arrival, avoiding such delays.

B. Worker Execution Model and Worker Pool

BATCHIT represents computing resources using *workers*, which are abstractions for computation and execution in the CPU. Let W be the maximum number of concurrently executing workers, i.e., the system has a worker pool of W workers. When idle, workers are assigned an operator to execute non-preemptively from a processing queue, as shown in Fig. 3. In contrast to dedicating workers for each operator, the worker pool model helps reduce contention and performs better on resource-constrained edge machines.

Batch Sizes: A key factor affecting the efficiency of workers in this model is the batch size of each operator $v_j \in V$. Let $B = \{b_j(t) : j \in 1 \ldots |V|\}$, denote a batch size plan, where $b_j(t)$ is the planned batch size for v_j for a time period t.

Data Arrival and Service Rates: We formalize the data arrival rate $\lambda_j(t)$ as a function of the selectivity $s_{i,j}(t)$ of an edge $e_{i,j}$ at time t. Here, $s_{i,j}(t)$ represents a measurement of the rate of values transferred from one node to another when a batch is processed, The data arrival rate $\lambda_j(t)$ at the operator v_j is then defined recursively in Eq. 1 as the sum of data arrival rates of direct predecessors, weighted by selectivity.

$$\lambda_j(t) = \begin{cases} \lambda_0(t) & \text{if } IsRoot(v_j) \\ \sum_{e_{i,j} \in E} (\lambda_i(t) \cdot s_{i,j}(t)) & \text{otherwise} \end{cases} \quad (1)$$

We next define the service rate $\mu_j(t)$ of operator v_j in Eq. 2, which denotes the speed at which data items of a batch can be processed. Here, $\mu_j(t)$ depends on the batch size $b_j(t)$ and the total delay $D(v_j, t)$ which consists of accumulating the batch, scheduling it to a worker, and processing it (described later).

$$\mu_j(t) = b_j(t)/D(v_j, t) \quad (2)$$

Batching, Scheduling and Processing Delay: Let $d_j(t)$ denote the total delay experienced by an operator v_j at the time period t, which consists of: (i) the batching delay $w_b(v_j, t)$ in Eq. 3a, representing the delay for accumulating data items in a batch; (ii) the scheduling delay $w_s(v_j, t)$ in Eq. 3b, representing the delay for allocating the batch to a worker; and (iii) the processing delay $w_p(v_j, t)$ in Eq. 3c, representing the delay in processing the data items of the batch. For an operator v_j, we compute $w_b(v_j, t)$ by dividing the batch size $b_j(t)$ by the data arrival rate $\lambda_j(t)$. After the batch is filled, we estimate a worse-case delay for $w_s(v_j, t)$, which assumes that all items "ahead" of the new batch are processed sequentially. Finally, we assume that $w_p(v_j, t)$ is computed offline in an empirical manner. However, parameters related to the type of operator (i.e., blocking/non-blocking, stateful/stateless) will further affect the processing time. In particular, the processing time of a blocking operator with block size β_i is the maximum between an empirically derived value, and the time taken for β_i data items to arrive, i.e., $\beta_i/\lambda_i(t)$. Similarly, the processing time of a stateful operator is the sum of empirical derived values and the time taken for state initialization $\sigma_j t_{in}$, where σ_j is the state initialization time, and t_{in} is equal to 1 iff the initialization should occur and else 0.

$$w_b(v_j, t) = b_j(t)/\lambda_i(t) \tag{3a}$$

$$w_s(v_j, t) = \sum\nolimits_{i=1}^{|Q(t)|} w_p(v_i) \tag{3b}$$

$$w_p(v_j, t) = \begin{cases} \max(w_p(v_j), \frac{\beta_i}{\lambda_i(t)}) + \sigma_j t_{in} & \text{if } v_j \text{is both} \\ \max(w_p(v_j), \frac{\beta_i}{\lambda_i(t)}) & \text{if } v_j \text{is blocking} \\ w_p(v_j) + \sigma_j t_{in} & \text{if } v_j \text{is stateful} \\ w_p(v_j) & \text{otherwise} \end{cases} \tag{3c}$$

C. The Batch Size Planning Optimization Problem

We first identify two key performance aspects for the system: throughput and end-to-end latency. Then, we formulate the batch size planning optimization problem, which aims to find optimal batch sizes for each operator in the workflow.

The Throughput Requirement: In general, each operator needs to process data items at a service rate larger than its data arrival rate (i.e., throughput). In doing so, the operator avoids accumulating data items the input queue indefinitely, which can lead to data loss or memory errors. We use the concept of traffic intensity measurements in queuing theory to model this requirement. For an operator v_j, its traffic intensity is the ratio of its data arrival rate $\lambda_j(t)$ and its service rate $\mu_j(t)$, measured during a time period t, as shown in 4. To avoid data loss and exponentially increasing queuing latency, we bound $\Lambda_j(t)$ by a user-defined threshold $(1 - \epsilon)$.

$$\Lambda_j(t) = \lambda_j(t)/\mu_j(t) \tag{4}$$

The End-to-end latency Goals: Generating each data product requires raw data to be processed by the operators in the workflow. The end-to-end delay of a data product is then the sum of delays incurred by all the predecessor operators of the corresponding sink node in the workflow, which we denote as $D(v_j, t)$ for the time period t. Eq. 5 defines a recursive function to find the end-to-end latency, which is the sum of the maximal end-to-end delay among all its upstream operators and the delay incurred by the operator itself, $d_j(t)$.

$$\begin{aligned} D(v_j, t) &= \max_{v_i, v_j \in E} D(v_i, t) + d_j(t) \\ \text{where } d_j(t) &= w_b(v_j, t) + w_s(v_j, t) + w_p(v_j, t) \end{aligned} \tag{5}$$

The Optimization Problem: We formulate batch size planning as an optimization problem in 6. We assume that $sla(v_j^*)$ denotes the service-level agreement, which represents an agreed upon time for operator v_j^* to produce a data product. Our formulation is as follows:

$$\arg\min_{B} \left\{ \max_{v_j^* \in V^*} \left\{ \frac{D(v_j^*, t) - sla(v_j^*)}{sla(v_j^*)} \right\} \right\} \tag{6a}$$

subject to:

$$\Lambda_j(t) \leq 1 - \epsilon, \forall t \tag{6b}$$

$$\sum\nolimits_j b_j(t) \leq C, \forall t \tag{6c}$$

The objective function shown in 6a captures the goal of optimizing the latency performance and fairness. Note that any value that does not violate a service level agreement $sla(v_j^*)$ will be negative, while any violating value is positive. Hence, ratio of $D(v_j^*, t) - sla(v_j^*, t)$ to $sla(v_j^*, t)$, for each v_j^* in V^* is the normalized latency violation across all data products. The objective function aims to minimize the largest latency violation among all data products. The constraint in 6b represents the throughput requirement, while the constraint in 6c denotes the system memory limit requirement.

Complexity of Batch Size Determination: The batch size planning problem can be shown to be NP-hard, by reducing from the Online Variable-Sized Bin Packing problem (OVBP) [26]. The reduction considers items arriving over time and a set of resizable bins. Here, items correspond to data products v_j^*, while resizable bins represent operators with dynamic batch sizes b_j. The OVBP objective aims to maximize the space utilization, which is equivalent in BATCHIT as minimizing the maximum service-level agreement violation across all operators. The dynamic adjustment of batch sizes at operator also corresponds with the online nature of OVBP.

IV. BATCH SIZE DETERMINATION HEURISTIC

There are several difficulties in determining the optimal batch size for an operator v_j at runtime. For example, if the planned batch size for v_j is too small, then the overall throughput performance of the system could suffer. However, using a larger batch size will incur a longer batching delay, which may then result in a SLA violation of a dependent data product. In our approach, we aim to dynamically change the batch sizes for each operator to be as large as possible (to enable higher throughput), without violating SLA constraints.

Alg. 1 selects an optimal batch profile from a set of candidate batches for each operator in the workflow at a time period t. We start by initializing B^* as a mapping of operators

Algorithm 1: Batch Size Expansion Algorithm

```
Input: Cand. batch set $B^{cand} = \{b_i^{cand}\}_{i \in 1 \ldots |B^{cand}|}$, time $t$
Output: Optimal Batch Profile, $B^*$
1 $B^* \leftarrow map()$; $V^{viol} \leftarrow \emptyset$
2 for $v_j \in V$ do
3     for $b_i^{cand} \leftarrow B^{cand}$ do
4         if $\Lambda_j^{b_i^{cand}}(t) > 1 - \epsilon$ then $B^*[v_j] \leftarrow b_i^{cand}$
5     if $v_j$ not in $B^*$ then $V^{viol} \leftarrow V^{viol} \cup \{v_j\}$
  // Check planned batch sizes
6 if $|V^{viol}| = 0$ && $\forall v_j^* \in V^* : D(v_j^*, t) - sla(v_j^*) < 0$,
    using $B^*$ then return $B^*$
7 else return $BatchSizeReduction(B^*, V^{viol})$
```

to their new batch sizes, and V^{viol} as a set of operators for which an appropriate candidate batch size in B^{cand} does not exist (line 1). For each operator v_j, we find the smallest batch size satisfying the condition $\Lambda_j^{b_i^{cand}}(t) > 1-\epsilon$ to ensure sufficient throughput (lines 2-4). Here, we note that $\Lambda_j^{b_i^{cand}}(t)$ denotes the traffic intensity for operator v_j when the candidate batch size b_i^{cand} is applied. If no such batch size exists within B^{cand}, we add v_j to V^{viol} (line 5). Finally, if all operators in V were able to obtain a new batch size B^*, and the resulting configuration had no SLA constraint violations, we return B^* (line 6). Otherwise, it is sent to Alg. 2 to reduce the batch sizes of violating operators (line 7).

In Alg. 2, we take B^* as provided in Alg. 1, the set of operators violating their SLA constraints V^{viol}, and a batch reduction factor γ for reducing batch sizes. We start by sorting violating operators in descending order of violation severity (line 1). Then, we reduce the batch sizes for operators in V^{viol} by a factor of γ, which continues while at least one of the batch sizes has not been reduced to 1. (line 2-4). Then, if the new batch size b_i^{cand} satisfies the traffic intensity constraint $\Lambda_i^{b_j}(t) \leq 1 - \epsilon$, we remove the violating operator from V^{viol}, and set the new batch size in B^* (line 5). If this new modification yields a batching profile configuration that does not violate any of the SLA constraints, then we return B^* as the set of new planned batch sizes (line 6).

Algorithm 2: Batch Size Reduction Algorithm

```
Input: Cand. batch set $B^{cand} = \{b_i^{cand}\}_{i \in 1 \ldots |B^{cand}|}$,
       Violating operators set: $V^{viol}$, Reduction factor $\gamma$
Output: Optimal Batch Profile, $B^*$
1 Sort elements $v_j \in V^{viol}$ by $\frac{b_j}{\lambda_j}$, in descending order
2 while $\exists\ v_j \in V : b_j \neq 1$ do
3     for $v_i \in V^{viol}$ do
4         $b_j \leftarrow b_j * \gamma$ // reduce batch size
5         if $\Lambda^{b_j}(t) \leq 1 - \epsilon$ then $B^*[v_j] \leftarrow b_i^{cand}$;
            $V^{vio} \leftarrow V^{vio} \setminus \{v_j\}$
6         if $\forall\ v_j^* \in V^* : D(v_j^*, t) - sla(v_j^*) < 0$, using $B^*$
            then return $B*$
7 return Error: system is too overloaded
```

V. Worker Scheduling

The worker scheduling model uses the operators' expected batch sizes to select the corresponding batch to execute when there is an idle worker. This selection is based on an earliest deadline first greedy policy, i.e., choose the operator with the earliest deadline. The algorithm relies on two types of queues for scheduling. The batch-ready queue includes the operations that have sufficient data items (i.e., $\geq$ their batch size) while the candidate-ready queue records the operations that are close to (but not) fulfilling their batch size requirement.

Alg. 3 descibes the process of scheduling a batch to a worker. It accepts a queue of batches Q and the current time t as input. If the queue is not empty (i.e., there are batch-ready items in Q), then it selects the batch with the earliest deadline and the item is removed from Q and returned (line 3-4). Otherwise, if there are no items in the queue (i.e., none of the batches are completely full), it selects the batch that is the most full. Let C be a set of potential batches that could be scheduled to a worker. Note that only the blocking characteristic of an operator will prevent its selection at this stage. Hence, all blocking operators v_j whose current batch size is larger than the block requirement β_j, as well as any non-blocking operator are added to C (line 8-9). Then, we select the batch that has the highest percentage of its batch filled to be allocated to the worker (line 10-11).

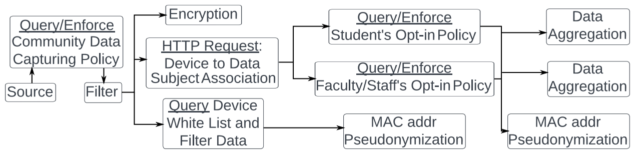
Algorithm 3: Operator Scheduling Algorithm

```
Input: Queue $Q$, Time $t$
Output: Batch to schedule on a worker
   // Queue has batches ready for processing
1  if $|Q| \neq 0$ then
2      $batch \leftarrow \arg\min\{\min_{v_j \in V} sla(v_j^*) - t\}$
3      $Q.remove(batch)$
4      return $batch$
   // schedule candidate if no ready batches
5  $C \leftarrow \emptyset$
6  for $v_j \in V$ do
7      $currBatch \leftarrow$ current batch of $v_j$
8      if $IsBlockingOp(v_j) \wedge |currBatch| \geq \beta_j \vee$
9          $\neg IsBlockingOp(v_j)$ then $C \leftarrow C \cup \{v_j\}$
   // x is the batch associated with $v_j$
10 $batch \leftarrow \arg\max v_j \in C \frac{|x|}{b_j}$
11 return batch
```

VI. Validation and Experiments

This section presents BatchIT's experimental results, including its performance comparison with typical stream processing and micro-batching methods, focusing on throughput and latency. It explores BatchIT's scalability, adaptability, and the effects of key configuration settings, particularly in customizing batching strategies for varied input requirements. The experiments involved a workflow encompassing access control, encryption, data aggregation, and pseudoanonymization, detailed in Fig. 4.

Comparison with Baseline Approaches: To evaluate the techniques used in BatchIT, we compare with the following



Query/Enforce Community Data Capturing Policy
Source
Filter
Encryption
HTTP Request: Device to Data Subject Association
Query Device White List and Filter Data
Query/Enforce Student's Opt-in Policy
Query/Enforce Faculty/Staff's Opt-in Policy
MAC addr Pseudonymization
Data Aggregation
Data Aggregation
MAC addr Pseudonymization

Fig. 4: Wi-Fi Access Point Association Data Workflow

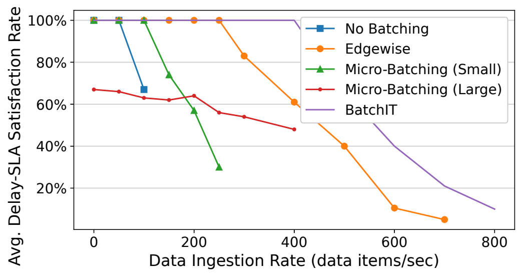


Fig. 5: Throughput and latency performance in Wi-Fi data

baseline techniques: (1) *No Batching*: The batch size is set to 1 for all operators. Whenever an operator has data-item/s in its input data queue, the operator is scheduled to be executed with the first data in the queue; (2) *Micro-batching*: Similar to Spark Streaming, the system works in two phases: batching and processing; (3) *Edgewise*: A stream processing engine designed specifically for edge computing hardware [10].

Overall Throughput and Latency Performance: In our experiments, we used a Raspberry Pi 4, with a quad-core A57 ARM V8 processor with 8GB of RAM as our edge system. We differentiated the processing size of micro-batch systems to show that map-reduce architectures do not work with large batch sizes. In terms of performance, an existing edge processing engine, Edgewise [10] is the most capable of the techniques we tested for handling an increasing rate of traffic besides BATCHIT. However, our system is capable of maintaining a 100% SLA Satisfaction rate for an ingestion rate that is 50% greater than EdgeWise as shown in Fig. 5.

Fig. 5 shows the results of the Average Delay-SLA satisfaction rate with respect to an increasing data ingestion rate on the same edge system across the four approaches. Because there are different latency goals across each data product in a workflow, the latency performance is measured as the average delay-SLA satisfaction rate among all data products. The rapid decay rate of each curve indicates when the data ingestion rate (arrival rate) surpasses the defined threshold $(1 - \epsilon)$, resulting in a steep decrease in SLA satisfaction. Hence, the number of data items buffered in the system increases indefinitely, eventually leading to a memory overflow error that crashes the system in each experiment.

Resource Tuning: In our experiments, we found that increasing the RAM capacity of the edge device leads to a near-linear improvement in its throughput capability. Moreover, it is evident that higher tolerance values are associated with higher throughput rates. In BATCHIT, we implement workers using a Java thread-pool. We tried multiple configurations of workers on our edge device, but found that there was a significant drop-off when scaling past 6 threads, which can be seen in Fig. 7b. As such, all our experiments considered a worker pool of 6.

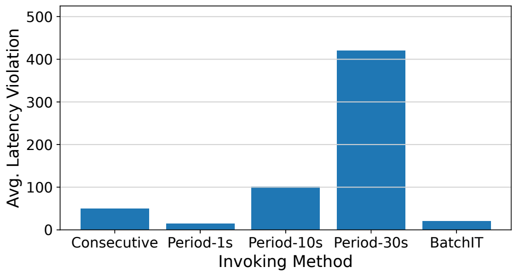


Fig. 6: Different workload dynamicity adaptation policies.

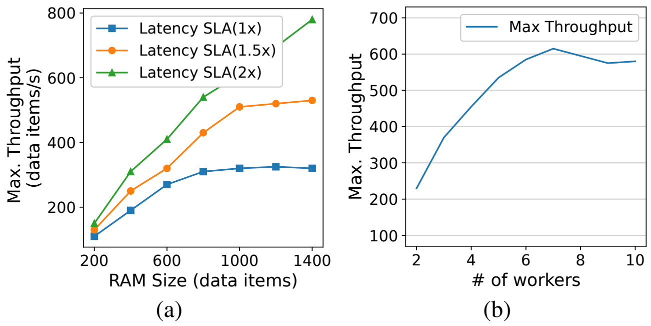


Fig. 7: Resource Tuning for (a) RAM and (b) Workers

Lastly, we explored how often our online dynamic batching approach would execute, as shown in Fig. 6. We tested the following approaches: (1) consecutive checking after each release of a data product, (2) different time window duration's, and (3) the dynamic batch tuning that BATCHIT uses. The interval of periodic checking is based on offline profiling where we view the ingestion patterns of historical data and use data samples to tune the edge device. This offline training occurs once per day and the adjusted sampling rate is applied to the live system after analysis. Our results indicate that the best performing approach in terms of latency violation was checking every second, but this introduces an extremely high overhead cost. As a result, the conditional trigger mechanism in BATCHIT only incurs a slightly higher latency cost, but significantly reduces the amount of computation overhead. The use of the latency tuning makes BATCHIT more scalable, as shown in Fig. 5.

VII. CONCLUSION

In this paper, we introduced BATCHIT, a processing middleware leveraging intelligent fine-grained batch tuning to balance batching-induced delay and end-to-end delay for IoT streaming applications. The key feature of BATCHIT is its ability to dynamically determine the optimal batch size for each operator during runtime, based on predefined thresholds for SLA violations. To design BATCHIT, we thoroughly evaluate various batching strategies, operator categorization techniques, memory models, and worker scheduling approaches. Our evaluation shows the benefit of intelligent batching and latency tuning across a real-world scenario. Furthermore, BATCHIT surpasses existing approaches by effectively maintaining service-level performance agreements and enhancing overall throughput.

Acknowledgment

This work is supported by National Science Foundation NSF Grant No. 1952247, No.2133391 and DARPA under Agreement No. FA8750-16-2-0021. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors.

References

[1] Thinagaran Perumal, Md Nasir Sulaiman, and Chui Yew Leong. Internet of things (iot) enabled water monitoring system. In *2015 IEEE 4th Global Conference on Consumer Electronics (GCCE)*, pages 86–87. IEEE, 2015.

[2] Daniel Minoli, Kazem Sohraby, and Benedict Occhiogrosso. Iot considerations, requirements, and architectures for smart buildings—energy optimization and next-generation building management systems. *IEEE Internet of Things Journal*, 4(1):269–283, 2017.

[3] Mohammad Saiedur Rahaman, Harsh Pare, Jonathan Liono, Flora D Salim, Yongli Ren, Jeffrey Chan, Shaw Kudo, Tim Rawling, and Alex Sinickas. Occuspace: Towards a robust occupancy prediction system for activity based workplace. In *2019 IEEE international conference on pervasive computing and communications workshops (PerCom workshops)*, pages 415–418. IEEE, 2019.

[4] Roberto Yus, Georgios Bouloukakis, Sharad Mehrotra, and Nalini Venkatasubramanian. The semiotic ecosystem: A semantic bridge between iot devices and smart spaces. *ACM Transactions on Internet Technology (TOIT)*, 22(3):1–33, 2022.

[5] Jiale Zhang, Bing Chen, Yanchao Zhao, Xiang Cheng, and Feng Hu. Data security and privacy-preserving in edge computing paradigm: Survey and open issues. *IEEE access*, 6:18209–18237, 2018.

[6] Suparna De, Tarek Elsaleh, Payam Barnaghi, and Stefan Meissner. An internet of things platform for real-world and digital objects. *Scalable Computing: Practice and Experience*, 13(1):45–58, 2012.

[7] Lo'ai Tawalbeh, Fadi Muheidat, Mais Tawalbeh, and Muhannad Quwaider. Iot privacy and security: Challenges and solutions. *Applied Sciences*, 10(12):4102, 2020.

[8] Jean-Sébastien Coron, Avradip Mandal, David Naccache, and Mehdi Tibouchi. Fully homomorphic encryption over the integers with shorter public keys. In *Advances in Cryptology–CRYPTO 2011: 31st Annual Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2011. Proceedings 31*, pages 487–504. Springer, 2011.

[9] Tathagata Das, Yuan Zhong, Ion Stoica, and Scott Shenker. Adaptive stream processing using dynamic batch sizing. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–13, 2014.

[10] Xinwei Fu, Talha Ghaffar, James C Davis, and Dongyoon Lee. Edgewise: a better stream processing engine for the edge. In *USENIX Annual Technical Conference (ATC)*, 2019.

[11] Pinchao Liu, Dilma Da Silva, and Liting Hu. Dart: A scalable and adaptive edge stream processing engine. In *USENIX Annual Technical Conference*, 2021.

[12] Le Xu, Shivaram Venkataraman, Indranil Gupta, Luo Mai, and Rahul Potharaju. Move fast and meet deadlines: Fine-grained real-time stream processing with cameo. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 389–405, 2021.

[13] Daniel Zhang, Nathan Vance, Yang Zhang, Md Tahmid Rashid, and Dong Wang. Edgebatch: Towards ai-empowered optimal task batching in intelligent edge systems. In *2019 IEEE Real-Time Systems Symposium (RTSS)*, pages 366–379. IEEE, 2019.

[14] Zhijing Qin, Grit Denker, Carlo Giannelli, Paolo Bellavista, and Nalini Venkatasubramanian. A software defined networking architecture for the internet-of-things. In *2014 IEEE network operations and management symposium (NOMS)*, pages 1–9. IEEE, 2014.

[15] Yiming Lin, Daokun Jiang, Roberto Yus, Georgios Bouloukakis, Andrew Chio, Sharad Mehrotra, and Nalini Venkatasubramanian. Locater. *Proceedings of the VLDB Endowment*, 14(3):329–341, nov 2020.

[16] Joshua Cao, Jesse Chong, Marissa Lafreniere, Owen Yang, Primal Pappachan, Sharad Mehrotra, and Nalini Venkatasubramanian. The zotbins solution to waste management using internet of things: Poster abstract. In *Proceedings of the 18th Conference on Embedded Networked Sensor Systems*, SenSys '20, page 731–732, New York, NY, USA, 2020. Association for Computing Machinery.

[17] Nisha Panwar, Shantanu Sharma, Guoxi Wang, Sharad Mehrotra, Nalini Venkatasubramanian, Mamadou H. Diallo, and Ardalan Amiri Sani. Iot notary: Attestable sensor data capture in iot environments. *ACM Trans. Internet Things*, 3(1), oct 2021.

[18] Andrew Chio, Daokun Jiang, Peeyush Gupta, Georgios Bouloukakis, Roberto Yus, Sharad Mehrotra, and Nalini Venkatasubramanian. Smartspec: A framework to generate customizable, semantics-based smart space datasets. *Pervasive and Mobile Computing*, page 101809, 2023.

[19] Petar Kochovski, Vlado Stankovski, Sandi Gec, Francescomaria Faticanti, Marco Savi, Domenico Siracusa, and Seungwoo Kum. Smart contracts for service-level agreements in edge-to-cloud computing. *Journal of Grid Computing*, 18:673–690, 2020.

[20] Zhijing Qin, Luca Iannario, Carlo Giannelli, Paolo Bellavista, Grit Denker, and Nalini Venkatasubramanian. Mina: A reflective middleware for managing dynamic multinetwork environments. In *2014 IEEE Network Operations and Management Symposium (NOMS)*, pages 1–4. IEEE, 2014.

[21] Jinlai Xu, Balaji Palanisamy, Qingyang Wang, Heiko Ludwig, and Sandeep Gopisetty. Amnis: Optimized stream processing for edge computing. *Journal of Parallel and Distributed Computing*, 160:49–64, 2022.

[22] John C McCullough, John Dunagan, Alec Wolman, and Alex C Snoeren. Stout: An adaptive interface to scalable cloud storage. In *Proc. of the USENIX Annual Technical Conference–ATC*, pages 47–60, 2010.

[23] Ram Srivatsa Kannan, Lavanya Subramanian, Ashwin Raju, Jeongseob Ahn, Jason Mars, and Lingjia Tang. Grandslam: Guaranteeing slas for jobs in microservices execution frameworks. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–16, 2019.

[24] Adam Richie-Halford and Ariel Rokem. Cloudknot: A python library to run your existing code on aws batch. In *Proceedings of the 17th python in science conference*, pages 8–14, 2018.

[25] Ruben Mayer, Muhammad Adnan Tariq, and Kurt Rothermel. Minimizing communication overhead in window-based parallel complex event processing. In *Proceedings of the 11th ACM International Conference on Distributed and Event-Based Systems*, DEBS '17, page 54–65, New York, NY, USA, 2017. Association for Computing Machinery.

[26] Nancy G Kinnerseley and Michael A Langston. Online variable-sized bin packing. *Discrete Applied Mathematics*, 22(2):143–148, 1988.