

# Fault Tolerance Placement in the Internet of Things

ANASTASIYA KOZAR, Technische Universität Berlin, Germany

BONAVENTURA DEL MONTE\*, Observe Inc., USA

STEFFEN ZEUCH, Technische Universität Berlin, Germany

VOLKER MARKL, Technische Universität Berlin, DFKI GmbH, Germany

Today's IoT applications exploit the capabilities of three different computation environments: sensors, edge, and cloud. Ensuring fault tolerance at the edge level presents unique challenges due to complex network hierarchies and the presence of resource-constrained computing devices. In contrast to the Cloud, the Edge lacks high availability standards and a persistent upstream backup. To ensure reliability, fault tolerance mechanisms have to be deployed on the edge devices along with processing operators competing for available resources. However, existing operator placement strategies are not aware of fault tolerance resource requirements, and existing fault tolerance approaches are not aware of available resources. This miscommunication in resource-constrained environments like the Edge leads to underprovisioning and failures.

In this paper, we present a resource-aware fault-tolerance approach that takes the unique characteristics of the Edge into account to provide reliable stream processing. To this end, we model fault tolerance as an operator placement problem that uses multi-objective optimization to decide where to backup data. As opposed to existing approaches that treat operator placement and fault tolerance as two separate steps, we combine them and showcase that this is especially important for low-end edge devices. Overall, our approach effectively mitigates potential failures and outperforms state-of-the-art fault tolerance approaches by up to an order of magnitude in throughput.

CCS Concepts: • **Information systems** → **Stream management**.

Additional Key Words and Phrases: Data Management, Stream Processing, Fault Tolerance

## ACM Reference Format:

Anastasiya Kozar, Bonaventura Del Monte, Steffen Zeuch, and Volker Markl. 2024. Fault Tolerance Placement in the Internet of Things. *Proc. ACM Manag. Data* 2, 3 (SIGMOD), Article 138 (June 2024), 29 pages. <https://doi.org/10.1145/3654941>

## 1 INTRODUCTION

The Internet of Things (IoT) is an evolving paradigm that is involved in myriad areas such as 5G [67], the Electrical Grid Industry [44], Autonomous Vehicle Technology [70], Smart Cities [38], and Healthcare [37]. The number of devices that build up the IoT is projected to reach 27 billion by 2025 [39]. Furthermore, Gartner predicts that 75% of enterprise-generated data will be created and processed outside a traditional data center or Cloud [10]. The fast-growing number of interconnected devices motivated the emergence of unified sensor-edge-cloud (USEC) systems [59, 69, 81]. These systems

---

\*The work was carried while being employed at Technische Universität Berlin.

---

Authors' addresses: Anastasiya Kozar, Technische Universität Berlin, Berlin, Germany, [anastasiya.kozar@tu-berlin.de](mailto:anastasiya.kozar@tu-berlin.de); Bonaventura Del Monte, Observe Inc., Berlin, USA, [venturadelmonte@gmail.com](mailto:venturadelmonte@gmail.com); Steffen Zeuch, Technische Universität Berlin, Berlin, Germany, [steffen.zeuch@tu-berlin.de](mailto:steffen.zeuch@tu-berlin.de); Volker Markl, Technische Universität Berlin, DFKI GmbH, Berlin, Germany, [volker.markl@tu-berlin.de](mailto:volker.markl@tu-berlin.de).

---

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM 2836-6573/2024/6-ART138

<https://doi.org/10.1145/3654941>

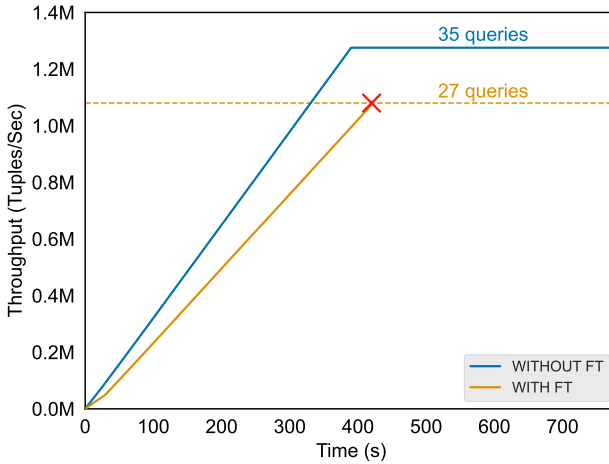


Fig. 1. Comparison of stateful query deployment with and without fault tolerance in Flink.

evolve from a cloud-centric approach by pushing down the computation to the resource-constrained and unreliable edge devices.

Ensuring reliable processing in such a heterogeneous environment is challenging due to deep network hierarchies, network unreliability, and resource-constrained computing nodes. Fault tolerance in the Cloud benefits from high availability standards and persistent upstream backups, such as implemented in Kafka [64]. In contrast to cloud environments, the Edge lacks a universally accessible storage infrastructure, face limitations in resource scalability, and are constrained by the processing power of edge devices. To guarantee reliability, USEC systems deploy fault tolerance on every system device, decreasing the available resources of edge devices. However, Stream Processing Engines (SPEs) place streaming operators on these devices independently, completely overlooking the resource consumption resulting from fault tolerance.

In Figure 1, we illustrate an example where unawareness of fault tolerance leads to underprovisioning, ultimately resulting in a system failure. In this experiment, we submit stateful queries from a cluster monitoring workload [2] to Flink [23] with ■ and without ■ fault tolerance running. The experiment runs on eight Raspberry Pi devices with 2GB RAM connected to a middle-sized server with 7GB RAM. When we disable fault tolerance, we successfully deploy up to 35 queries. However, when we enable fault tolerance, we only successfully deploy up to 26 queries. The 27th query leads to a system failure because the device runs out of memory, which happens due to the wrong estimation of the operator placement algorithm in Flink. The estimation is inaccurate as it does not consider that fault tolerance occupies additional memory. Even worse, in a USEC environment, where transient failures are common, the deployment process would not stop after the failure of a single worker. The master node would have continued deploying queries, terminating resource-constrained edge devices one by one. Overall, this experiment highlights two main problems of running fault tolerance in USEC environments: **P1**: running fault tolerance on every device reduces a significant part of available resources from processing, and **P2**: ignorance of fault tolerance costs results in underprovisioning.

In this paper, we present a holistic fault tolerance placement (FTP) that treats fault tolerance as a first-class operator and places it selectively as a solution to the underlying Multi-objective optimization (MOO) problem. FTP scores potential fault tolerance placements (P2) based on two conflicting goals: utilized resources and provided reliability. Once FTP identifies the optimal placement, it propagates estimated resource requirements to the operator placement strategy (P1). To ensure FTP's compatibility with both heuristic and cost-based operator placement approaches, we divide

it into Naive FTP (NFTP) and Multi-objective FTP (MFTP). NFTP simplifies the optimization problem by focusing on a single objective and uses heuristic resource estimates. In contrast, MFTP leverages cost-based upper-bound resource estimates and hardware characteristics to compute placement scores. Additionally, we introduce MFTP-H, which employs hyperparameter tuning and runtime adaptation to fine-tune resource usage in response to system demands.

We evaluate FTP approaches in NebulaStream (NES), which is a USEC system that explores hardware-tailored code compilation and a highly dynamic execution model [31, 36, 81]. Our evaluation demonstrates that NES, along with our efficient implementation of fault tolerance, achieves up to  $\times 67$  throughput compared to other state-of-the-art solutions. Employing FTPs to NES prevents potential underprovisioning and failures. Moreover, it shows that FTPs require less additional placement decision time by an order of magnitude than incorporating the reliability constraint into existing operator placement strategies. Finally, MFTP-H helps improving the overall system throughput by up to 18% and the number of stateful queries deployed by 30% compared to existing fault tolerance placement approaches. In summary, we make the following contributions:

- We model the FTP problem as a Pareto set and propose single- and multi-objective solutions (Section 3).
- We define a cost space for every fault tolerance approach and introduce an additional parameter to estimate reliability (Section 4).
- We describe naive heuristic-based and elaborative cost-based FTP solutions with adaptive optimizations (Section 5).
- We demonstrate that FTPs avoid underprovisioning and adaptive optimizations improve system resource management (Section 6).

## 2 BACKGROUND

In the following section, we introduce concepts of data streaming (Section 2.1). Additionally, we discuss existing operator placement (Section 2.2) and recovery approaches (Section 2.3).

### 2.1 Stream Processing

SPEs handle infinite sequences of data streams ( $s \in S$ ) composed of relational tuples ( $t \in T$ ) belonging to the same logical type [24]. Each tuple ( $t = (\sigma, \pi)$ ) includes a timestamp ( $\sigma \in N^+$ ) and a payload ( $\pi$ ) assigned by either an external physical or an internal logical clock. Tuples traverse SPEs in a potentially out-of-order manner ( $\forall t_i, t_j$  with  $i < j$ ,  $\sigma_i < \sigma_j$  is not guaranteed), due to variations in event timestamps.

Queries in these systems adhere to the streaming relational model [16] and encapsulate sets of logical operators distributed across interconnected physical nodes ( $n \in N$ ) via data channels. Operators ( $o \in O$ ) process data within SPEs, generating output tuples based on execution semantics. Notably, there are two special operators: sources ( $c \in C$ ) producing data streams and sinks ( $f \in F$ ) consuming them. In neighboring nodes, the one closer to the source is considered the upstream node [40].

### 2.2 Operator Placement Strategies

Over the last decades, plenty of operator placement strategies were developed explicitly targeting SPEs. These strategies aim to optimize the Quality of Service (QoS) metrics depending on a system setup and user requirements. From the methodological perspective, the operator placement strategies can be grouped into the following classes:

**Heuristic Methods.** This class of operator placement strategies is widely adopted in open-source industrial frameworks [15, 18, 22, 45] tailored to specific environments and workloads. The heuristics methods vary from simple first-fit, round-robin, and bottom-up algorithms to resource-aware, multi-layer heuristics [29, 42, 60, 61, 63].

**Cost-based Approaches.** Cost-based methods, including search-based, greedy, and graph-theoretic approaches, aim to optimize system properties, which are typically expressed within a cost space. This cost space integrates various parameters, including queueing delay, execution, transmission, enactment, and migration cost in the optimization process [3, 17, 56, 62]. Search-based operator placement strategies find an optimal placement using local search, tabu search, and simulated annealing [42, 73]. Greedy approaches employ cost-based optimization to enhance different QoS metrics [14, 19, 79] through the use of efficient heuristics. Finally, graph-theoretic methods leverage cost-based techniques, employing weighted dataflow graphs to enhance various QoS metrics [48, 82].

### 2.3 Recovery Approaches

Different SPEs have varying high-availability requirements, characterized by processing guarantees such as none, at-least-once, at-most-once, and exactly-once. Existing fault tolerance solutions, including Upstream Backup, Passive Standby, and Active Standby, treat these guarantees as QoS metrics.

**Upstream Backup.** One of the most common fault tolerance techniques in distributed stream processing systems is data buffering at the upstream nodes, acting as backups for downstream nodes [11, 26, 74]. They log elements in their output queues until downstream nodes process them entirely. In case of a failure, upstream nodes resend stored tuples. This method relies on the assumption of monotonically increasing timestamps and uses punctuation techniques like low watermarks to manage late data. [12].

**Passive Standby.** Systems such as Flink, Spark, and Heron support the passive standby approach [46, 55, 57, 66, 76, 77]. In this method, primary nodes periodically send their state to secondary nodes. In the event of a primary node failure, secondary nodes load the latest state and resume processing. State includes input, local states of stateful operators, and node output. Synchronization is achieved through state updates during global checkpointing [40].

**Active Standby.** SPEs following the active standby approach [20, 27, 41, 49], designate secondary nodes as hot standby states. These secondary nodes process elements in parallel with primary nodes. Secondary nodes log output and take over in the case of primary node failure, sending their output to downstream operators.

## 3 FAULT TOLERANCE PLACEMENT PROBLEM

Highly volatile disaggregated environments with heterogeneous devices change priorities in objectives that are considered for operator placement [21]. In particular, data delivery becomes the main target as the core SPE requirement, i.e., supplying users with the data, cannot be guaranteed in such unreliable environments. To tackle this problem, we integrate a dedicated fault tolerance operator that ensures data reliability. This operator can be placed on a device before or after the processing operators to preserve the data state. Similar to operator placement, we present a placement problem for a fault tolerance operator as a choice of an optimal ratio between utilized resources and provided level of reliability (Section 3.1). We then propose single-objective constraint-based (Section 3.2) and multi-objective cost-based solutions (Section 3.3) to find the optimal trade-off.

### 3.1 Problem Statement

The USEC environment encapsulates a tree-shaped topology with processing devices that vary from small battery-powered sensors over system-on-a-chip devices to high-end rack-scale servers, making every path unique in terms of resources and reliability [21, 81]. To ensure data delivery in accordance with system resources, fault tolerance has to be placed selectively on a per-path basis. We present the FTP problem in three steps: 1) determining the placement path, 2) determining optimal candidates, and 3) defining the optimal number of devices to run the fault tolerance. The solution to the FTP problem requires locating an optimal mapping of one logical fault tolerance operator to many physical

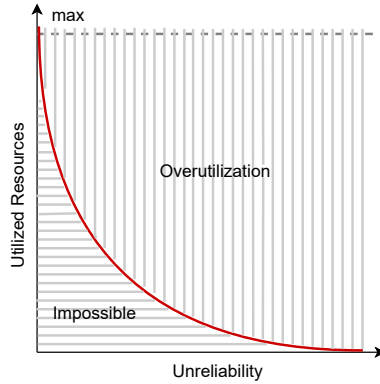


Fig. 2. Resource-reliability trade-off as a Pareto Set.

instances deployed on devices. Finding an optimum is complicated by the need to optimize multiple, possibly conflicting, objectives  $d \in D$ , such as minimizing unreliability, maximizes required resources, and minimizing the utilized resources maximizes unreliability. Similarly to the placement of processing operators [71], an optimal placement can be described as a function  $F$  of  $x$  that maximizes the score of an individual placement. Given the  $P([N])$  is a set of all combinations of devices in topology with  $N$  nodes and set of objectives  $\langle d_1, d_2, \dots, d_n \rangle$ , the score for placement on  $k$  devices can be described as:

$$\underset{x}{\operatorname{argmax}} F(x), \quad x = \langle d_1, d_2, \dots, d_n \rangle, \quad k \in P([N]) \tag{1}$$

where

$$F(x) = \begin{bmatrix} F_1(x) \\ \dots \\ \dots \\ F_k(x) \end{bmatrix}$$

In Equation 1, the maximization of function  $F(x)$  over all device combinations in a topology with  $N$  nodes, shapes the optimal balance required for fault tolerance. Enhancing system reliability to tolerate network partitions in an USEC system means increasing the number of utilized resources, i.e., node replicas. This scenario illustrates the conflict identified by the CAP theorem: a USEC system striving to provide reliability (akin to availability or consistency) in the face of potential network partitions requires significant resource investment. For instance, a USEC system requires replicas of every device to support availability or redundant data paths to keep replicas consistent. Considering that increasing the number of replicas cannot increase system unreliability (device reliability cannot be negative), and fault tolerance cannot decrease utilized resources as it requires redundancy to reproduce data in case of failure, these objectives are co-dependent and grow in opposite directions. Figure 2 visualizes this trade-off as a space of points with unreliability on the x-axis and utilized resources on the y-axis. This space is the Pareto Set, where some points represent a more optimal ratio than others, i.e., dominate others.

In a USEC environment, the dominant points are restricted by the number of resources on each device and the maximum possible reliability, i.e., accumulative probability of failure of all devices in one path. Figure 2 shows the set of Pareto Optimal points as a red line that forms the Pareto Front. The set of points to the right of the line presents an overutilization of resources for the same level of reliability. At the same time, all the points to the left of the line represent impossible trade-offs due to resource and reliability constraints. Therefore, the presented fault tolerance placement problem falls into the class of multi-objective optimization (MOO) problems with multiple, possibly conflicting, objectives.

Tackling the presented MOO problem is challenging due to data scarcity, diverse workloads, and specific user reliability constraints. Conventional supervised learning methods rely on substantial data volumes. Because of the limited datasets that integrate both reliability and resource utilization, these approaches are not feasible for solving the FTP problem. The collection of such data is complicated as the USEC environment includes devices that don't fail for decades. The extensive diversity and scale of potential workloads across heterogeneous devices make full-scale simulation impractical, while ad-hoc simulation for individual workloads significantly increases end-to-end latency.

### 3.2 Single-objective Optimization

The Equation 1 is a mathematical representation of the MOO problem [71], where every  $F_k(x)$  encapsulates two subfunctions with contradicting objectives. Let the function  $f_k^U(x)$  represent utilized resources and  $f_k^R(x)$  the level of reliability achieved by deploying the fault tolerance operator on the  $k$ th combination of devices. Then, every  $F_k(x)$  from Equation 1 can be rewritten as:

$$\begin{cases} F_k(x) = (f_k^U(x), f_k^R(x)) \\ f_k^U(x) \rightarrow \min \\ f_k^R(x) \rightarrow \max \end{cases} \quad (2)$$

The multi-objective utility function  $F_k(x)$  in Equation 2 can be transformed into a linear objective function by replacing resource minimization with the no over-exceeding constraint C. The key objective of the FTP approach, thus, switches to maximizing the reliability level based on available resources (Equation 3). An alternative solution is saving as many resources as possible, providing sufficient reliability for a given query.

$$\begin{cases} F_k(x) = (f_k^U(x), f_k^R(x)) \\ f_k^U(x) \leq C \\ f_k^R(x) \rightarrow \max \end{cases} \quad (3)$$

The single objective solution finds one optimal point for a given Pareto Set. However, the resulting solution finds only a sub-optimal solution as it restricts one of the objectives. It focuses only on one objective, sacrificing the contradictory one. Thus, we present a more elaborate approach that finds all possible points of the Pareto Front.

### 3.3 Multi-objective Optimization

Many approaches have been proposed that do not constraint the objectives providing a set of optimal ratios: Weighted Sum (WS) [52], Evolutionary Methods (EV) [33], Normalized Constraints (NC) [53], and Multi-objective Bayesian Optimization (MOBO). In this section, we examine the premises of the MOO solution and select one of the existing methods that finds the entire set of Pareto Optimal points.

A key requirement for the desired MOO solution is to incorporate user-defined reliability constraints within the reliability objective. Users typically specify reliability levels as processing guarantees, as part of query configurations [41]. Since the utility function in Equation 1 involves continuous objectives, these discrete reliability specifications should be incorporated by the MOO solution. Additionally, the diverse IoT use cases necessitate a more fine-granular approach to specify resource utilization. For example, Healthcare and Autonomous Vehicle Technology demand more processing resources, while Smart Cities and the Electrical Grid Industry require extensive network resources for data transmission. Therefore, the discrete objectives should be adapted not just for reliability but also for different types of resources to reflect their varying importance across different scenarios.

While utilized resources and reliability serve as the primary objectives to prevent underprovisioning, it is essential that the solution can be expanded to encompass typical QoS metrics in USEC

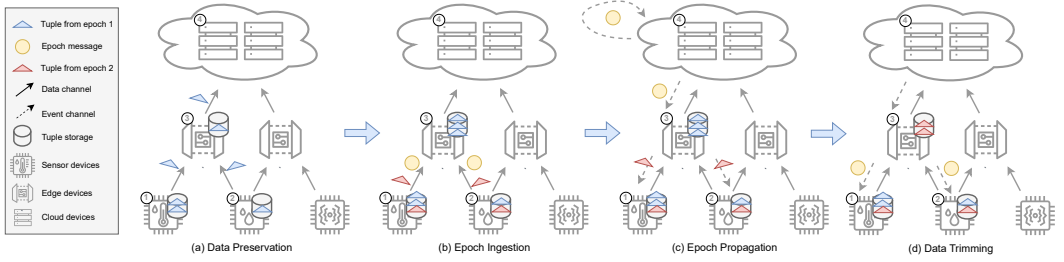


Fig. 3. State preservation pattern.

systems, such as energy consumption or processing latency [54]. For instance, USEC systems can enhance their battery life by avoiding the deployment of resource-intensive tasks, such as filtering at the sensor level. However, this approach increases the amount of data that needs to be stored on these devices for fault tolerance, necessitating additional memory resources. Similarly, the fastest path in terms of latency can be the most unreliable, resulting in the latency-reliability trade-off. Overall, the desired MOO solution should incorporate discrete objectives and be efficient and scalable. However, approaches like NC lacks comprehensive exploration [71], while EV and MOBO can be computationally demanding in the case of scalability [28, 68, 78]. Moreover, none of these methods incorporate discrete coefficients, preferring continuous objectives. Having user-defined prioritization of objectives, we obtained the WS approach, allowing us to integrate coefficients as weights and extend the sum to include additional objectives.

$$F_k = \sum_{k \in P([N])} w_i d_k^i, \quad \sum_i \|w_i\| = 1, \quad i = 1, \dots, n \quad (4)$$

Given  $N$  devices, the Equation 4 presents the utility function  $F_k$  as a weighted sum of all objectives  $d_1, \dots, d_n$  for a given combination  $k \in P([N])$  in a system with  $N$  devices. The objectives used in the function are normalized costs for a given fault tolerance placement. In the case of contradictory objectives, we add the minus sign to one of the objectives, transforming it into the maximization problem. In the next section, we investigate the suitability of existing approaches to follow a generalized assessment of fault tolerance costs that can be used in Equation 4.

#### 4 FAULT TOLERANCE COST ESTIMATION

In this section, we elaborate on the resource-reliability trade-off by defining cost models to determine the cost of fault tolerance operators. We achieve this by defining a common procedure of state preservation that is shared across common fault tolerance approaches developed for SPEs.

Existing fault tolerance approaches require storing data temporarily to ensure data reproducibility in case of failure. Dealing with infinite data streams and limited memory, these approaches have to identify no longer needed data and periodically trim it. A trimming frequency is typically defined using an epoch [18, 22], i.e., time- or count-based interval. In Figure 3, we introduce a common procedure to preserve and release data, i.e., state. This procedure consists of four phases: 1) data preservation, 2) epoch ingestion, 3) epoch propagation, and 4) data trimming. Figure 3 provides an example of a hierarchical topology within the USEC environment, where differently equipped devices at each level contribute to data processing. A query is deployed on devices 1, 2, 3, and 4 and replicated on devices 1, 2, and 3 to ensure reliability. Data flow from sensors (nodes 1 and 2) over an edge device (node 3) directly to the cloud (node 4). In Figure 3a devices 1, 2, and 3 preserve tuples in their tuple storage before shipping them to the downstream nodes. In the next phase, the epoch (trimming) message is ingested in the data stream (see Figure 3b). All the tuples that arrive later than this message belong to the next epoch. Once the trimming message reaches the final sink, device

4 provides a safe-to-trim timestamp and propagates it via event channels to the upstream nodes (see Figure 3c). Finally, in Figure 3d, the epoch message reaches device 3 and launches trimming of all tuples that were produced before the epoch message.

The Upstream Backup and Passive Standby approach, introduced in Section 2, follow this pattern completely, utilizing disk or memory for storing data and the network for sending trimming messages. In contrast, Active Standby additionally requires processing resources due to operator replication. Yet, the storing and trimming pattern stays the same. This section introduces cost models for memory, network, and processing (Section 4.1). Additionally, we introduce a cost model to assess path reliability that considers topology heterogeneity (Section 4.2).

#### 4.1 Estimation of Resource Utilization

Common recovery methods require the same resources in different degrees, typically utilizing the remaining resources not consumed by the processing operators. The cost models presented further estimate network, memory, and processing resources based on the state preservation procedure described in Figure 3. Since our goal is not to underprovision the resources of an individual device, these metrics serve as upper boundaries of required resources, whereas the actual costs can be lower. They are hardware-oblivious and highly depend on the epoch length, as shown in the Evaluation.

**Network.** The network cost of a fault tolerance operator originates from trimming messages that are regularly propagated along the topology (Figure 3d). The frequency of these messages is defined by the epoch length  $EL$ . As mentioned earlier, the epoch length can be set in time or in the number of tuples that left the system. The smaller  $EL$ , the more often trimming messages are sent. Independently of how the  $EL$  is defined, the frequency of trimming messages depends on the arrival rate of tuples. If tuples arrive at the final node with a rate of  $I$ , and messages are generated for every  $EL$  tuples, then the actual message frequency is  $\frac{I}{EL}$ . The resulting ratio is multiplied by the tuple size  $T_s$  to get the network bandwidth penalty.

The Equation 5 represents an upper bound estimation for  $k$  devices. Currently, we assume a fixed ingestion rate due to a tuple storage serving as a backpressure mechanism, guaranteeing the upper bound. However, the equation can be extended by the output selectivity of a given query. Including 1) the fraction of emitted tuples by total tuples received by the operator, and 2) the tuple size modification factor of the operator, i.e., the factor at which an operator changes the size of input tuples.

$$N_k = T_s \sum_{i=1}^k \frac{I_i}{EL} \quad (5)$$

**Memory.** Memory cost in fault tolerance systems is determined by the storage needed for operators to preserve tuples until a trimming message arrives, which then trims tuples from the previous epoch. Under ideal conditions with no new buffer ingestion, storage can hold a maximum of  $EL$  tuples, where  $EL * T_s$  represents the total memory used, with  $T_s$  being the tuple size. However, the dynamic nature of USEC environments, as shown in Figure 3d, results in continued tuple ingestion for the next epoch during trimming message transmission. The additional memory requirement accounts for the network delay  $D$  and the ingestion rate  $I$ . Considering the roundtrip network delay  $D(i,k)$  from device  $i$  to device  $k$ , the total stored tuples can be calculated as  $EL + 2 * D(i,k) * I_i$ . Equation 6 provides an upper boundary estimation of memory usage for  $k$  devices.

$$M_K = T_s \sum_{i=1}^k (EL + 2D(i,k)I_i) \quad (6)$$

**Processing.** The processing cost in Passive Standby and Upstream Backup is mainly represented by data trimming in tuple storages. However, in the Active Standby approach, the main processing



costs come from replicas of primary operators and are equivalent to the processing costs of primary operators. As shown in the Evaluation, the processing costs of producing and forwarding a trimming message can be neglected since they are relatively small compared to other costs.

## 4.2 K-Safety

To measure system reliability, we redefine the existing property K-Safety [72] to incorporate the heterogeneity of devices. For the first device ( $k = 1$ ), the K-Safety ( $S_k$ ) is simply equal to the reliability of that device, denoted as  $R_k$ . For subsequent devices ( $k > 1$ ), the K-Safety is determined by adding the reliability of the current device ( $R_k$ ) to the product of two factors: 1) the K-Safety of the previous device ( $S_{k-1}$ ), which represents the accumulated fault tolerance up to that point, 2) the necessity of the backup ( $1 - R_k$ ). Equation 7 iteratively calculates the K-Safety level for each device in a chain, taking into account its own reliability and the K-Safety of the devices that precede it. This cumulative measure helps to assess how many devices within the system can fail while still ensuring data preservation.

$$S_k = \begin{cases} R_k, & k = 1 \\ R_k + (1 - R_k) \prod_{i=2}^k S_{i-1}, & k > 1 \end{cases} \quad (7)$$

As the reproducibility of data is not ensured, the USEC systems cannot provide two out of four existing processing guarantees: at-least-once and exactly-once. The remaining ones, none and at-most-once processing guarantees accept data loss. To match different use cases, we enriched the at-most-once processing guarantee with further granularity by creating new reliability levels: LOW, MEDIUM, and HIGH [13, 51]. Each level corresponds to the percentage of devices in one path that participates in fault tolerance. Specifically, LOW represents participation above 25%, MEDIUM above 50%, and HIGH above 75%. These levels allow us to tailor fault tolerance placement to the diverse and dynamic nature of USEC environments, ensuring that processing guarantees are aligned with the reliability requirement of a specific query.

The individual probability of each device can be calculated using the MTBF property. For example, the MTBF value of the Common Future Modular Seeker sensor devices used in navigation, rocket propulsion, fin actuators, etc. [7] is 15,000 hours. In contrast to low-end battery-powered sensors, system-on-a-chip devices like the Raspberry Pi 5.12V DC show a higher MTBF value of 50,000 hours [9]. This value decreases primarily due to power breakdowns or SD card failures [6]. Meanwhile, the MTBF estimate for the Intel Server Board S1200V3RP is around 44 years [5]. The probability that the device is functioning without failures at the time  $t$  can be calculated using the bathtub curve from exponential modeling. Using the MTBF value, the probability  $R_k$  that device  $k$  is operational at time  $t$  is:

$$R_k = e^{-t/MTBF(k)} \quad (8)$$

The resulting probability  $p$  in Equation 8 gives us a value in the range  $[0, 1]$  with a probability that  $k$ th device participating in the query is running at the time  $t$ .

With defined cost models, it appears that deploying fault tolerance closer to the sink appears profitable because devices with more resources typically offer higher reliability compared to smaller devices. However, this approach also means not replicating data from devices closer to the sources, which could lead to potential data loss. Conversely, deploying fault tolerance on small sensor-edge devices exacerbates resource constraints and poses a risk to data consistency, as these devices suffer the most from transient failures.

Overall, the definition of the state preservation procedure exposes the possibility of introducing quantitative estimates to the resource-reliability tradeoff. These estimates improve the precision of available system resources and are used in the following section to solve the FTP problem.

**Algorithm 1** General FTP approach.

---

```

1: function FTP(paths)
2:   for each path ∈ paths do
3:     placement ← FINDPLACEMENT(path)
4:     score ← CALCULATESCORE(placement)
5:     if score > selectedPath.score then
6:       selectedPath.score ← score
7:       selectedPath.placement ← placement
8:       selectedPath.path ← path
9:     for each node ∈ selectedPath.placement do
10:      node.isFaultTolerant(true)
11:      REDUCERESOURCES(node)
12:   return selectedPath.path

```

---

**5 FAULT TOLERANCE PLACEMENT**

To enable effective fault tolerance in USEC environments, operator placement strategies must integrate fault tolerance costs to prevent underprovisioning and prioritize reliability due to non-guaranteed data delivery. Our FTP solutions are designed to prioritize data delivery in query deployment, dynamically adjusting fault tolerance placement and resource allocation for each query to avoid underprovisioning and ensure efficient deployment. For each query, our approaches dynamically adjust fault tolerance placement and prevent resource underprovisioning by propagating the chosen path and necessary resources to the subsequent query deployment stages.

Generally, FTP approaches can be split into two subproblems: 1) finding a path with the best placement based on a score, 2) updating path resources based on the final placement. Algorithm 1 presents a general structure of all FTP algorithms and highlights functions that vary in different algorithms. For every path the FTP approach defines a set of devices with enough resources to run fault tolerance (Line 3). Based on this set, it calculates a score (Line 4). It compares the score to the current maximal score of the selected path and remembers the placement in case the score is larger (Lines 5-8). It updates the system resources by marking the selected nodes as used for fault tolerance, reducing their available resources. The returning value of Algorithm 1 is dependent on the systems data forwarding strategy. Systems that implement data partitioning replicate processing operators over multiple paths [49, 59, 74]. In such systems, FTP approaches can be updated to return  $r$  paths with the best score, where  $r$  is a replication factor.

In the rest of the section, we discuss both heuristic-based and advanced cost-based fault tolerance placements compatible with various operator strategies. We begin with a heuristic-based approach for the SOO problem, using heuristic estimates for placement scoring and resource reduction (Section 5.1). This is followed by a cost-based approach, leveraging cost models from Section 4 for the MOO problem (Section 5.2). Lastly, Section 5.3 introduces runtime adaptations that enhance the cost-based approach by factoring in the unique attributes of individual queries.

**5.1 Naive FTP**

To efficiently solve the operator placement problem, various heuristics have been introduced [29, 42, 60, 61, 63]. However, as the heuristic-based approaches typically exclude fault tolerance from resource estimation, it can lead to underprovisioning and failures in resource-constrained environments (Figure 1) [25]. The NFTP approach avoids underprovisioning and ensures data delivery. NFTP simplifies the FTP problem by transforming it into a SOO problem, maximizing system reliability

**Algorithm 2** The NFTP approach.

---

```

1: function FINDPLACEMENT(path)
2:   for each node  $\in$  path.reverse do
3:     if node.hasEnoughResources() then
4:       placement.add(node)
5:   return placement
6: function CALCULATESCORE(placement)
7:   for each node  $\in$  placement do
8:     score  $\leftarrow$  score + node.distance()
9:   return score
10: function REDUCERESOURCES(node)
11:   node.slots  $\leftarrow$  node.slots - 1

```

---

while treating the resource capacity of a given path as a limiting constraint. After finding an optimal placement, it propagates fault tolerance costs to the operator placement strategy.

Existing operator placement strategies estimate utilized resources using different abstractions, e.g., slots [23]. In the slotted cost model, each infrastructure node has a fixed number of computing slots that can be used for assigning operators, and each operator has a dedicated requirement in terms of slots. In NFTP, each fault tolerance operator is estimated to consume one resource unit, which is equivalent to the simple processing operator cost.

The placement of fault tolerance in the NFTP is performed based on simple heuristics stating the importance of placing fault tolerance as soon as data enters the system to ensure upstream backup (similar to Kafka). In Algorithm 2, NFTP iterates through the nodes in reverse order, starting from the leaf node and moving towards the root (Line 2). The reverse order is motivated by the requirement to satisfy user-defined reliability constraint, which is referred to the number of devices participating in fault tolerance. In case of the lower reliability levels, NFTP still ensures that data is preserved as early as possible. At first, the NFTP approach adds every node that has enough resources to the final placement (Lines 3-5). Then it assigns a score to each node in the placement based on the device's proximity to the leaf node. This score is calculated as the number of hops between the current and the root node (Lines 6-9). After the function returns the placement score and the selected placement, NFTP reduces the required slots from the path capacity (Lines 10-11).

The naive algorithm effectively addresses the challenge of underprovisioning while striving to optimize system reliability. However, it tends to underutilize system resources due to imprecise resource estimation. Thus, in scenarios where the fault tolerance mechanism is lightweight and demands fewer resources than a processing operator, NFTP may underutilize resources, leading to a reduced number of deployed queries.

## 5.2 Multi-objective FTP

Existing cost-based operator strategies involve mathematical modeling and optimization techniques that take into account a wide range of factors, including current resource availability, workload characteristics, and QoS requirements [3, 17, 56, 62]. However, as heuristic-based strategies, they also ignore fault tolerance costs [25], resulting in underprovisioning and failures. To improve fault tolerance estimation and provide compatible costs for the cost-based operator placement, we introduce the MFTP approach.

Our MFTP approach utilizes cost models for network  $N_k$ , memory  $M_k$ , processing  $P_k$ , and K-Safety  $S_k$  introduced in Section 4. These cost models provide upper bound estimates of resources required for fault tolerance. Figure 5 visualizes the memory estimates (Equation 6) compared to the real values.

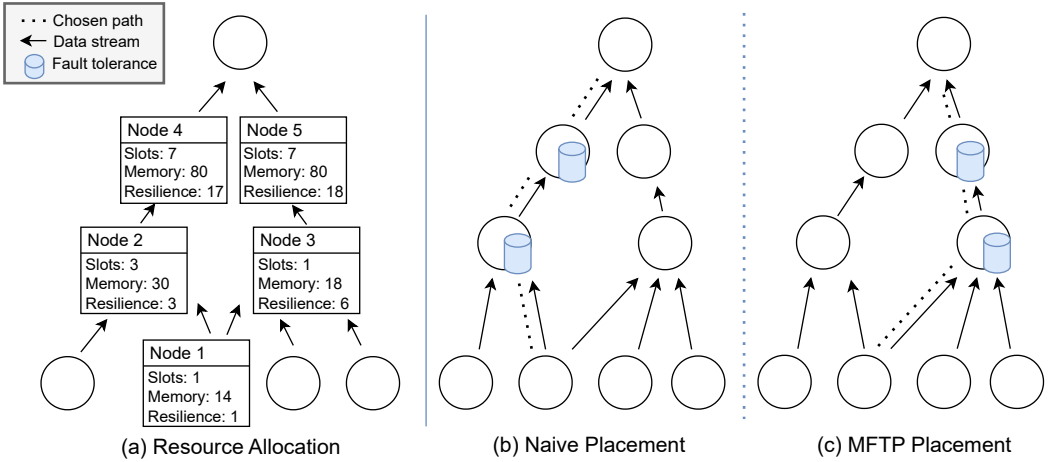


Fig. 4. Example placement using NFTP and MFTP approaches.

Based on these estimates, the MFTP then constructs a cost space for a set of all possible placements and chooses the one with the maximal score. It employs a cost-based utility function that extends Equation 4 to evaluate any fault tolerance placement based on the combination of K-Safety and resource utilization.

In Equation 9, K-Safety is a probabilistic metric and does not require normalization. The resource additives are normalized by including minimum and maximum values of resources per path. The min value equals 0, representing the case when deployment requires all available resources of a given objective. The max value depicts the current path capacity, representing the case when fault tolerance was not deployed on any device.

$$F_k = w_s S_k + w_n \frac{N_{max}^k - N^k}{N_{max}^k - N_{min}^k} + w_m \frac{M_{max}^k - M^k}{M_{max}^k - M_{min}^k} + w_p \frac{p_{max}^k - p^k}{p_{max}^k - p_{min}^k} \quad (9)$$

The complexity of the resulting utility function is exponential and equals to the number of combinations of  $n-1$  devices on one path. To reduce the resolution time, our MFTP approach utilizes a fault tolerance heuristic similar to NFTP. It begins by identifying the nearest device to the source with sufficient resources for fault tolerance (Lines 2-6). For every device, MFTP calculates required resources based on the equations presented in Section 4.1 with  $k$  equivalent to the current device only. Then, it traverses the rest of the path in the opposite direction, iteratively going from the largest device to the smallest (Lines 7-10). This heuristic is based on the observation that larger devices nearer to the root typically have more resources [81]. It saves resolution time without sacrificing Pareto Optimal points. For the resulting placement, the MFTP approach calculates scores using the cost models from Section 4 (Lines 12-22), with reliability being recursive and dependent on all chosen devices. Finally, the MFTP approach reduces the required memory, network, and processing resources from each node that was chosen for placement using individual cost estimates from Section 4.1 (Lines 24-26).

Figure 4 presents the difference between placements of NFTP and MFTP in a simplified example with only memory resources. Given the topology presented in Figure 4a, both algorithms find two paths available: nodes 1-2-4 and 1-3-5. The path with nodes 1-3-5 has a higher level of resilience and is therefore considered optimal. The NFTP approach requires a minimum of two slots from every node: one for fault tolerance and one for processing. While iterating over the path 1-3-5, Nodes 1 and 3 are not added to the set of candidates for placement. Similarly, on the path 1-2-4 the Node 1 is not considered for placement. The overall score of nodes 2, 4 on the path 1-2-4 is 3, while the score of the node 5 on the path

**Algorithm 3** The MFTP approach.

---

```

1: function FINDPLACEMENT(path)
2:   for each node ∈ path.reverse do                                ▷ find initial placement
3:     path.pop()
4:     if node.hasEnoughResources() then
5:       placement.add(node)
6:       break
7:   for each node ∈ path do                                        ▷ proceed with the normal order
8:     if node.hasEnoughResources() then
9:       placement.add(node)
10:  return placement
11: function CALCULATESCORE(placement)
12:  for each node ∈ placement do
13:    memory ← memory + node.calculateMemory()
14:    network ← network + node.calculateNetwork()
15:    processing ← processing + node.calculateProcessing()
16:    currentReliability ← node.calculateReliability()
17:    if node.isSourceNode() then
18:      reliability ← currentReliability
19:    else
20:      reliability ← currentReliability +
21:        (1 - currentReliability) * reliability
22:  return  $w_1 * \text{memory.normalize}() + w_2 * \text{network.normalize}() + w_3 * \text{processing.normalize}() + w_4 * \text{reliability}$ 
23: function REDUCERESOURCES(node)
24:  node.memory ← node.reduceMemory()
25:  node.network ← node.reduceNetwork()
26:  node.processing ← node.reduceProcessing()

```

---

1-3-5 is only 1. Thus, NFTP chooses a suboptimal path 1-2-4 for placement with nodes 2, 4 participating in fault tolerance, as illustrated in Figure 4b. At the same time, the estimation of individual resources allows MFTP to avoid using imprecise slot approximations. Assuming network delay being 0.01s between every two nodes, a tuple size of 0.1Mb, an ingestion rate of 100 tuples/s, and an epoch length of 100 tuples, the required memory for fault tolerance to be placed on Node 3 is  $M_3 = 0.1 * 100 + 2 * 2 * 0.01 * 100 = 14Mb$ . However, the required memory for Node 1 is  $M_1 = 0.1 * 100 + 2 * 3 * 0.01 * 100 = 16Mb$ , meaning that Node 1 is not added by MFTP to the set of placement candidates for both paths. After finalizing the set of candidates, the MFTP approach calculates a score for all possible combinations of devices. For instance, having the system running for one hour and individual reliabilities  $R_5 = e^{-1/18} = 0.95$  and  $R_3 = e^{-1/6} = 0.85$ , the k-safety of devices 3, 5 is  $S_{35} = 0.95 + (1 - 0.95) * 0.85 = 0.99$ . Thus, omitting weights for simplicity, the  $F_{35} = \frac{98 - (16 + 15)}{98 - 0} + 0.99 = 1.68$ . Following similar calculations, MFTP computes a score for the placement 2, 4 with the score 1.61. Finally, it chooses the path 1-3-5 with the highest score and nodes 3, 5 for running fault tolerance (see Figure 4c).

### 5.3 Runtime Adaptation

The dynamic environment of USEC necessitates continuous adaptation in placement decisions. Static provisioning does not fit continuous, long-running streaming applications, as it almost inevitably leads to an over- or under-provisioned system [43]. Variable workloads and fault tolerance require

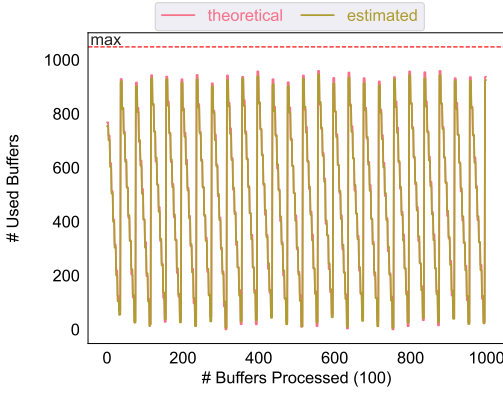


Fig. 5. Estimated vs. actual storage size.

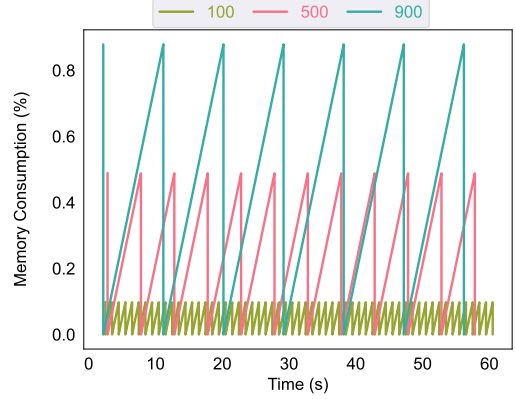


Fig. 6. Memory consumption of different epochs.

different types of resources to different degrees. Constantly optimizing for one type can bring the system to imbalance, resulting in bottlenecks and underprovisioning. Further, we introduce an MFTP-H algorithm that 1) adjusts global fault tolerance parameters to align with current system resources (see Section 5.1) and 2) employs individual query heuristics to iteratively calibrate the weights of the MOO utility function, thus determining the significance of one objective over another (see Section 5.2).

**5.3.1 Hyperparameter Tuning.** As presented in Figure 3, every fault tolerance approach follows the same procedure in terms of resource redundancy. This redundancy cannot be avoided but can be adjusted. Figure 6 shows the memory consumption on one device based on different trimming frequencies. The epoch length is usually a system hyperparameter that is set for the entire system [18, 23, 45]. However, as shown in Figure 6, the appropriate selection of this parameter can aid in deploying fault tolerance on devices with limited resources. For example, setting the epoch to 100 can reduce memory consumption of a fault tolerance approach. Therefore, the correct choice of a hyperparameter helps to distribute resources according to individual system properties. By minimizing the epoch length, we can save memory utilization, or by maximizing it, we can reduce network utilization. As a modification of MFTP, we present MFTP-H, which 1) defines the epoch value on a per-query basis and 2) adapts the epoch length according to the system resource allocation on the arrival of a new query.

Our MFTP-H algorithm alters the epoch parameter iteratively on every newly arrived query. For a given path, it finds the smallest device and its memory and network capacities. Based on these values and the initial resources on the device, it finds the optimal ratio that adjusts the initial epoch value. The value of an epoch cannot become smaller than one. It is important to mention that even if the epoch value equals one, more than one tuple is stored in practice. In Figure 3b-d, we see that we store tuples from the previous epoch along with tuples from the next epoch. This happens because our approach is non-blocking, and while the tuple reaches the final sink and trimming message arrives at the node, more data is processed and stored. Once the new epoch is chosen, we iterate over all nodes on the path and propagate the new value of the epoch.

**5.3.2 Weights Adaptivity.** The utility function presented in Equation 9 is a weighted sum that can be split into two types of objectives with unique weights: reliability and utilized resources. The reliability weight mirrors a user requirement in terms of data importance. For example, if data is unimportant and can be partially lost, then the system does not need to allocate many resources for the fault tolerance of the submitted query. In our approach, we allow users to choose from four classes of reliability: NONE, LOW, MEDIUM, and HIGH. In particular, no fault tolerance for NONE, and at least 25, 50, and

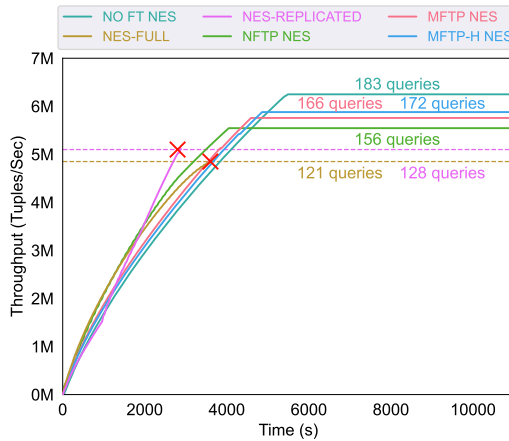


Fig. 7. Multi-Query Scalability.

75% of devices participating in fault tolerance for LOW, MEDIUM, and HIGH. We incorporate these values into the reliability weight and add an additional constraint to ensure user-specified guarantees.

Regarding resource weights, we split adaptation into two steps: initial calibration and runtime adjustment. The initial calibration requires the system administrator to analyze the system resource scarcity. Then, the runtime adaptation depends on the query workload. In our solution, we adapt weights based on simple heuristics: stateful queries are memory-heavy, queries with high selectivity are CPU-heavy and queries with low selectivity are network-heavy. In future work, we see the potential of developing a reinforcement learning model for this analysis and rewarding it with runtime monitoring values.

In summary, our FTP solutions address the unique challenges of fault tolerance placement in USEC environments by prioritizing reliability and efficient resource utilization. These approaches dynamically adapt fault tolerance placement for each query, preventing underprovisioning while ensuring data delivery.

## 6 EVALUATION

In this section, we experimentally evaluate our FTP solutions in NES [47, 81]. In Section 6.1, we introduce our experimental setup. After that, we present four sets of experiments. First, we evaluate the resource efficiency of our approaches in NES and compare it to the other approaches in state-of-the-art SPEs (Sec. 6.2). Second, we analyze the impact of fault tolerance with different hyperparameters on the system’s throughput and latency (Sec. 6.3). Third, we explore the decision time of FTPs on large-scale topologies (Sec. 6.4). Finally, we analyze the recovery latency based on different placements and epoch length (Sec. 6.5).

### 6.1 Experimental Setup

We run our experiments on four types of hardware. *Type A*: a Linux server with an AMD EPYC 7742 2.25GHz CPU (64 physical cores) and 1TB of main memory. *Type B*: a hierarchical cluster of eight Linux servers with 2 Intel Xeon Silver 4216 2.20 GHz CPU (32 physical cores), 500 GB of main memory, and 100 Gbit Infiniband connection. *Type C*: 256 Raspberry Pis united into a Kubernetes cluster. Each node is equipped with two virtual CPUs (vCPUs) at 1.2 GHz, 2 GB RAM, and 10 GB disk space. *Type D*: Raspberry Pi 4 Model B with 1.5 MHz Quad-Core, 2GB RAM, MTBF 50k hours.

To run the experiments, we utilized NES v0.5 and Apache Flink 1.17.1, running on a Kubernetes v1.25 cluster created in Google Cloud with system docker images. In NES, each source node generates data at 900 buffers/sec, with each buffer containing 2048 tuples. Both Flink and NES’s fault tolerance

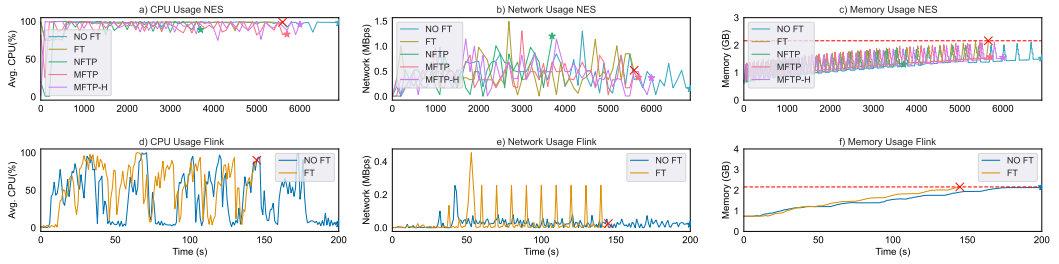


Fig. 8. CPU, network, and memory utilization during stateful queries submission in NES and Flink.

settings include a 100ms trimming frequency (NES measures epoch length in buffers) and a tuple size of 131Kb.rs. The reliability weight of cost-based FTP is LOW (0.25).). The weights of utilized resources are adjusted automatically based on the workload, e.g., 0.5 memory, 0.125 network, and 0.125 processing for memory-heavy workloads (see Section 5.3.2).

**6.1.1 Workloads.** We selected three distinct workloads from the NEXMark benchmark suite [75]:  $NxQ_0$ ,  $NxQ_2$ , and  $NxQ_8$ , to perform an analysis of system resources. This benchmark suite serves as a simulation of a real-time auction platform, encompassing key functionalities such as auctions, bids, and new user events. The suite accommodates both stateless operations (e.g., projection or filtering) and stateful operations (e.g., stream joins or window aggregations). The rationale behind choosing these three workloads is their representation of diverse resource-demanding scenarios, including network-, memory-, and processing-intensive workloads, and their widespread use in the research community. In detail, these queries perform the following workload:

**$NxQ_0$ : Pass Through.**  $NxQ_0$  in the NEXMark benchmark is a network-intensive workload comprising solely of source and sink operators. It tests the system’s capability to handle high data throughput without processing delays.

**$NxQ_2$ : Selection.**  $NxQ_2$  finds bids with specific auction IDs and shows their bid price. This query is CPU-intensive with high ingestion rates and a stateless selectivity operator.

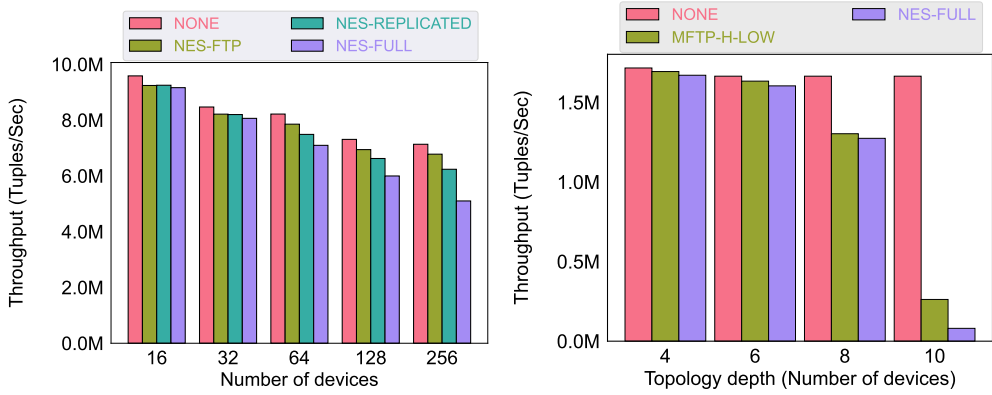
**$NxQ_8$ : Monitor New Users.**  $NxQ_8$  selects people who entered the system and created auctions in the last period using a stateful window operator. This query represents a memory-heavy computation for small IoT devices.

## 6.2 System Comparison

In this section, we study the system throughput and resource utilization under the impact of fault tolerance on hardware with limited resources. To this end, we study multi-query scalability in NES (Sec. 6.2.1), resource utilization (Sec. 6.2.2), and throughput of fault tolerance in NES against state-of-the-art approaches (Sec. 6.2.2).

**6.2.1 Multi-query scalability.** In the first experiment, we evaluate the multi-query scalability of existing fault tolerance placements in an environment with constrained resources. We subsequently deploy stateful query  $NxQ_8$  multiple times on four machines of Type D. All devices run NES, where one machine serves as a Coordinator and the other serve as NES Workers. Each NES Worker is equipped with a source forming a two-layered topology with three available paths. The experiment shows the number of queries deployed and throughput of NES without fault tolerance ■, with NFTP ■, with MFTP ■, and with MFTP-H ■, compared to NES-FULL ■ and NES-REPLICATED ( $r=2$ ) ■. NES-FULL implements fault tolerance on all devices by default, similar to systems with upstream backup or passive standby approaches like Flink [23] or Heron [45]. Conversely, NES-REPLICATED applies fault tolerance on every device across ‘r’ (replication level) paths, similar to systems that implement active standby, such as Frontier [59], TelegraphCQ [27], or Borealis [20].





(a) Wide topology (b) Deep topology  
 Fig. 9. Scalability in wide and deep topologies.

**Results.** Figure 7 presents that NES without fault tolerance supports 65% more queries and achieves 29% higher throughput than NES-FULL. Similarly, NES-REPLICATED reduces the number of deployed queries by 31% and the overall throughput by 18%. Both NES-FULL and NES-REPLICATED placement strategies result in a failure. Employing NFTP allows NES to successfully deploy 156 queries with 5.5M aggregated throughput without failure. Compared to NFTP, the MFTP approach increases the number of deployed queries to 166 and the throughput to 5.7M, which is an increase of 4%. The MFTP-H approach increases the overall number of deployed queries to 172 and the throughput to 5.8M, which is only 2% higher than MFTP.

Independently of the fault tolerance strategy, NES faces underprovisioning and failure. NES-REPLICATED and NES-FULL fail after deploying 127 and 120 queries respectively, due to ignorance of resources utilized by fault tolerance. NFTP deploys fewer queries than MFTP and MFTP-H because of less accurate resource estimation. MFTP’s cost-based estimation successfully deploys 166 queries, while MFTP-H’s runtime adaptation and parameter tuning further increase this to 172 queries. However, it does not significantly improve the throughput (only 2%), as frequent trimming to reduce memory results in a larger processing queue. Ultimately, NFTP, MFTP, and MFTP-H prevent underprovisioning and failure by incorporating fault tolerance costs into the operator placement strategy, with MFTP-H enhancing fault-tolerance query deployment by 30% and throughput by 18% compared to NES-FULL, and by 26% and 13% compared to NES-REPLICATED.

**6.2.2 Resource Utilization.** Figure 8 provides insights into the system resource utilization of a Worker during the query deployment on two machines of Type D with 2 GB memory to reveal the different behavior of our approaches. Both devices run NES or Flink, where one machine serves as a Coordinator (Master) and the second as a Worker. To this end, we subsequently deploy stateful query  $NxQ_8$  multiple times and analyze the CPU, network, and memory utilization of NES with ■ and without ■ fault tolerance, with NFTP ■, with MFTP ■, and with MFTP-H ■, compared to Flink with ■ and without ■ fault tolerance. As NES-FULL and NES-REPLICATED do not perform resource analysis and affect the resources of an individual worker in precisely the same manner, we mention them both as with fault tolerance (FT). The successful query deployment is marked as  $\star$  and failure as  $\times$ .

**CPU Utilization.** NES deploys 64 queries without fault tolerance, but fails at 42 queries with fault tolerance, runs 32 queries with NFTP, 41 queries with MFTP and 47 with MFTP-H. Flink successfully deploys 14 queries without fault tolerance while failing at the 11th query in case of fault tolerance. NES runs longer since it deploys more queries by optimally using resource-constrained hardware. To

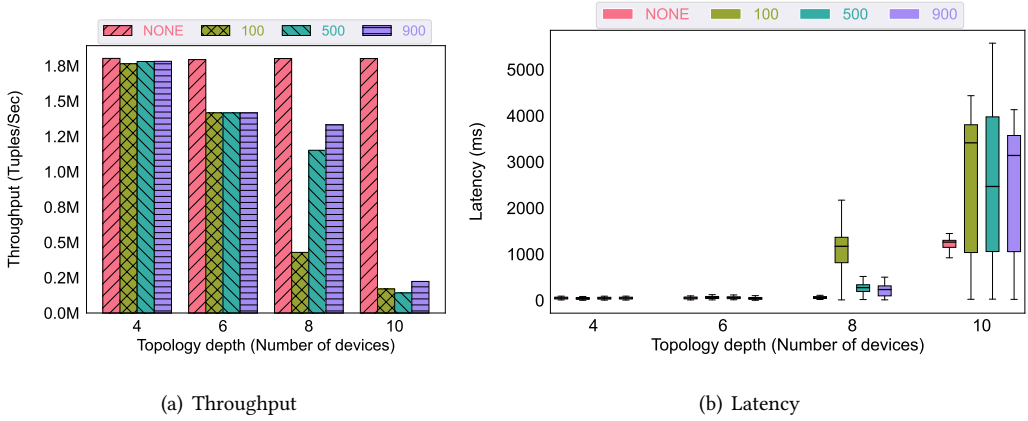


Fig. 10. Varying epoch lengths.

this end, it dynamically schedules processing operators on hardware resources, balancing workload performance requirements and operator compute demands. Figure 8a shows that NES effectively utilizes resources, supporting CPU utilization with and without fault tolerance on an average of 98%. Flink, on average, loads CPU only to 47% with fault tolerance and 40% without (Figure 8d). Although we expected an increased CPU usage in NES due to a higher number of trimming tasks, this effect is mitigated by the low trimming frequency (see Sec. 6.3.1) and the efficient implementation of trimming operations. As a result, our lightweight implementation only induces a negligible overhead.

**Network Utilization.** Flink requires, on average, 25% more network bandwidth with fault tolerance compared to running without fault tolerance, whereas NES requires only 5% more. On average, the network utilization of fault tolerance is higher than without due to trimming messages sent at the end of every epoch. This overhead can be controlled by correctly adjusting fault tolerance parameters. In Figure 8b of NES, we see that MFTP-H utilizes 1% more network capacity than the rest of the approaches. It happens because it reduces epoch length, making the fault tolerance approach trimming more often to save memory.

**Memory Utilization.** Memory usage with fault tolerance is 6% higher on average for NES and 16% for Flink compared to systems running without fault tolerance. Figure 8c reveals that the failure that appeared while deploying stateful queries with fault tolerance came from the device running out of memory.

Fault tolerance requires additional memory resources to store redundant data, ensuring recovery in the event of failure. For instance, in the case of NES, with an epoch length of 128 buffers and a network delay of 0.1s, the expected storage size is estimated at 156 buffers (see Equation 6). Given a buffer size of 131 KB, this translates to an overall additional memory requirement of approximately 20 MB per window. This difference is shown in Figure 8c, where NES requires additional memory with fault tolerance. Additionally, Figure 8c reveals that NFTP stops the query deployment before underprovisioning memory. However, it does it quite far from the memory limit, due to imprecise estimation. MFTP also avoids underprovisioning, stopping the deployment process closer to the memory limit than NFTP. That highlights better resource estimation of MFTP. MFTP-H reaches the memory limit later, requiring 4% less memory than when running the fault tolerance approach without adaptation.

**6.2.3 Scalability.** Figures 9(a) and 9(b) demonstrate the scalability of NES-FULL, NES-REPLICATED and NES-FTP. First, we compare the throughput of NES-FULL, NES-REPLICATED ( $r=10\%$  of devices), FTP approaches, and no fault tolerance in NES on a wide, two-layered flat topology in Figure 9(a). Note that, we merge NFTP, MFTP and MFTP-H to NES-FTP, as applying them to such kind

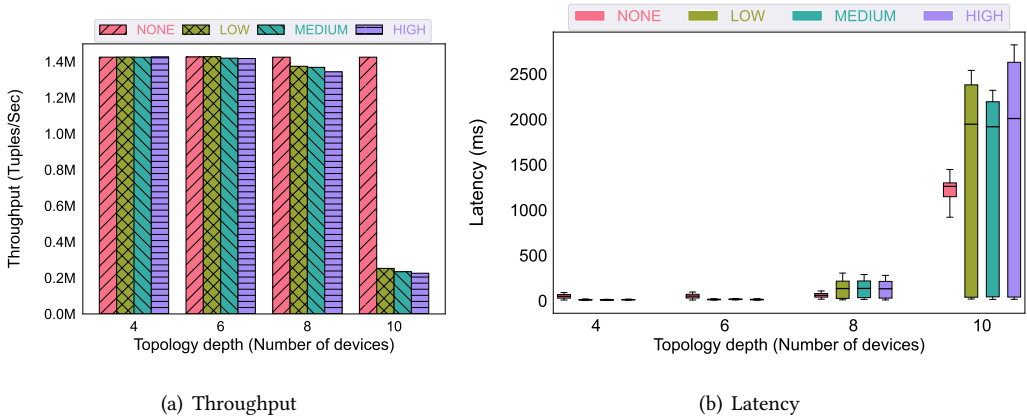


Fig. 11. Varying reliability.

of topology results in the same placement. Second, in Figure 9(b), we compare the throughput of no fault tolerance against MFTP-H with LOW reliability and NES-FULL (as both NES-FULL and NES-REPLICATED place fault tolerance on every device on one path, we show only NES-FULL) on a deep sequential topology (using one path). We run the experiments on Type D hardware with the Coordinator running on a Linux server with 30 vCPU at 1.2 GHz, 30GB RAM, and 10Gb external storage. We run the  $N \times Q_0$  workload and study the throughput of an increasing number of devices.

**Results.** For both topologies, NES-FTP outperforms existing state-of-the-art approaches. In Figure 9(a), FTP induces only a minor overhead and thus reduces the throughput at most by 5%, regardless of the number of devices. In contrast, the throughput difference between NES with NES-REPLICATED and without fault tolerance increases from 10% for 128 devices to 13% for 256 devices. NES-FULL shows a greater reduction in throughput, 18% for 128 devices and 29% for 256 devices. In Figure 9(b), we observe that MFTP-H stays relatively close to NES-FULL for a depth of 4, 6, and 8. Both MFTP-H and NES-FULL reduce system throughput by 24% in comparison to no fault tolerance at a height of 8. Additionally, with a topology depth of 10, NES-FULL reduces throughput from 1.8M to 80k tuples/sec, while MFTP-H reduces it to 262k tuples/sec, resulting in more than 3 times higher throughput. In summary, NES-FTP maintains consistent throughput in wide topologies without scalability issues, while NES-REPLICATED and NES-FULL struggle with scalability as device numbers increase. In deep topologies, fault tolerance reduces throughput beyond a depth of 8 due to increased trimming message delays, but MFTP-H minimizes this overhead more effectively than NES-FULL by dynamically adjusting epoch size and trimming frequency. Overall, NES-FTP demonstrates better scalability in both wide and deep topologies as the system scales, primarily due to deploying fewer replicas and thus reducing system workload compared to other approaches.

**6.2.4 Throughput Comparison.** In this experiment, we evaluate the throughput of Flink [23], Heron [45], Frontier ( $r=2$ ) [59], and NES with LOW (NES-L), MEDIUM (NES-M), and HIGH (NES-H) reliability on 128 devices of Type C using  $N \times Q_2$  workload. To this end, we create a three-layered topology with one source node. We compare NES running NFTP approach for placement, which employs the Upstream Backup fault tolerance approach, with Flink and Heron, both utilizing Passive Standby for fault tolerance. Additionally, we evaluate our approach against Frontier, a fault tolerance solution designed for Edge environments that uses Active Standby.

**Results.** NES achieves a throughput of around 1.8M tuples per second, while other state-of-the-art systems sustain up to 68 times lower throughput. In particular, Frontier achieves 26k, Flink up to 30k, Heron up to 34k tuples/second. Providing the lowest reliability level, NES-L achieves the

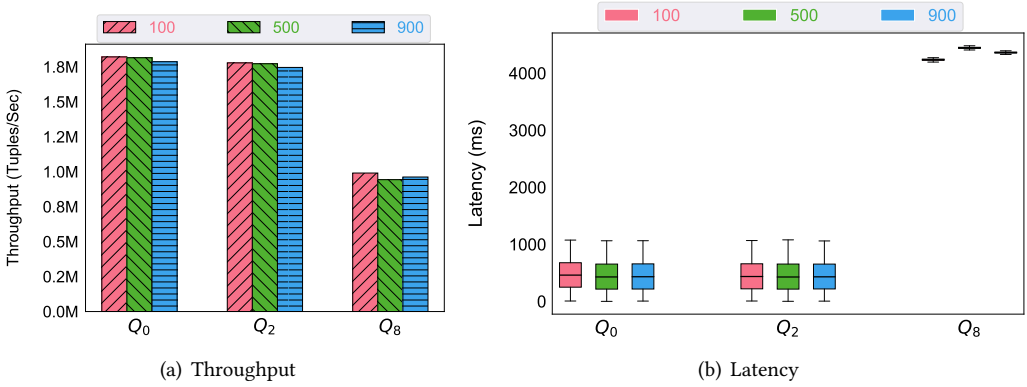


Fig. 12. Varying queries.

highest throughput of 1,78M tuples/sec. NES-M slightly reduces throughput by 0.8%, and the highest reliability level, NES-H, reduces throughput by only 1% from NES-L, sustaining 1,76M tuples/sec.

Parts of the speedup of NES can be devoted to its highly efficient processing engine that provides a holistic stream operator model, which splits logical operators into executable pipelines of reusable sub-operators. To utilize modern hardware most efficiently, the processing engine of NES customizes code generation and compilation to generate efficient code based on customizable sub-operators [30, 34, 35, 58, 65, 83]. In addition, by leveraging our non-blocking lightweight fault tolerance implementation, NES attains an exceptional throughput despite constrained resources. In contrast, Java-based Flink, Heron, and Frontier, running on Raspberry Pis, suffer from resource limitations due to the Java virtual machine (JVM) overhead, significantly constraining the available resources of edge devices.

### 6.3 Impact of Fault Tolerance

In the following set of experiments, we study the impact of fault tolerance and its parameters on the throughput and latency. To this end, we study the impact of a variable epoch length (Sec. 6.3.1), varying reliability (Sec. 6.3.2), and various queries (Sec. 6.3.3).

**6.3.1 Variable epoch.** In this experiment, we evaluate the scalability of our fault tolerance approach in NES. The coordinator runs on the hardware of Type A and workers on Type B, forming a chained topology. The memory capacity of all workers is restricted to store at most 1024 buffers to simulate a memory-constrained environment, where the buffer size is 131KB. To analyze system behavior with variable epoch lengths, we pick 100, 500, and 900 buffers to represent low, medium, and high memory loads. We compare the throughput and latency of these three epoch lengths against no fault tolerance while increasing the number of nodes in the topology.

**Results.** Figure 10(a) reveals that the throughput remains the same across all epoch lengths for topology depths of 4 and 6 across various fault tolerance epoch lengths. Similar Figure 10(b) shows no difference in latency between no fault tolerance and fault tolerance with three different epoch lengths. For a topology depth of 8, epoch length 100 reduces system throughput by 75% compared to no fault tolerance. Latency increases as well by  $\times 18$  on average from 60 to 1100ms. An epoch length of 500 reduces the system's throughput by 44% and latency by 4x. For a topology depth of 10, all three epochs caused a major reduction in system throughput by around 94% and a latency increase from 1s to 2.5s, which is more than two times higher.

To select the optimal epoch parameter, we propose to use Equation 6 for the closest to the source device. This device experiences the highest load on fault tolerance storage due to the delay in trimming messages. Its memory estimate serves as an upper bound for the required memory dedicated to fault

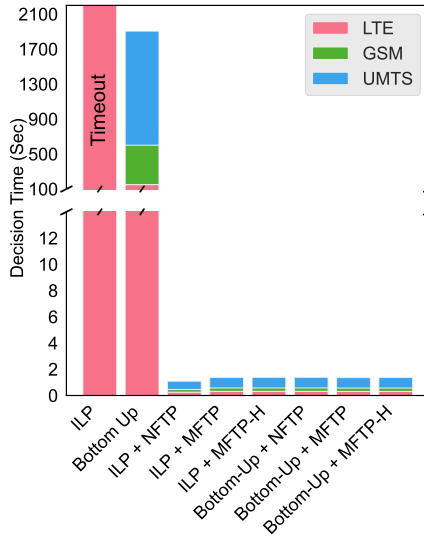


Fig. 13. Decision time for the Berlin cell tower dataset.

tolerance. In this experiment, with the system’s ingestion rate of 700 buffers/second and the round-trip delay of approximately 0.1 seconds between devices, the estimated memory for a topology depth of 8 is 980 buffers. Deviating significantly from this value floods the system with trimming messages, potentially leading to processing bottlenecks, as epoch length 100 demonstrates in Figure 10(b). Similarly, for a topological depth of 10, the estimated memory requirement for the furthest device is around 1260 buffers. Therefore, all considered epoch lengths are sub-optimal for this depth. From this experiment, we conclude that the epoch length has a major impact on the system’s performance, as sub-optimal epoch lengths can flood the system with trimming messages, resulting in decreased performance.

**6.3.2 Varying reliability.** In this experiment, we investigate how different levels of reliability influence system performance. Similar to the previous experiment, we evaluate the scalability of NES with the Coordinator running on the hardware of Type A and Workers on Type B, forming a chained topology. We compare the throughput and latency of three reliability levels, LOW, MEDIUM, and HIGH, with epoch length 900, against no fault tolerance. These levels correlate to the number of devices with fault tolerance operators deployed: more than 25% for LOW, 50% for MEDIUM, and 75% for HIGH.

**Results.** Similar to the previous experiment, Figures 11(a) and 11(b) demonstrate almost no difference in throughput and latency for the topology depth 4 and 6 compared to no fault tolerance. With the topology depth of 8, the throughput of fault tolerance drops by 4%, and the latency increases by more than two times from 60ms to 134ms. Finally, for the topology depth of 10, the throughput drops by 85%, and the latency increases by 30%.

Overall, due to our highly efficient implementation of trimming, there is almost no difference between different reliability levels, i.e., different numbers of nodes participating in fault tolerance. The throughput reduction for the higher topology depth is due to the epoch value of 900, which is sub-optimal for the topology depth of 10 (see Section 6.3.1).

**6.3.3 Various queries.** In this experiment, we assess the impact of epoch sizes 100, 500, and 900 on three types of NEXMark queries:  $NxQ_0$ ,  $NxQ_2$ , and  $NxQ_8$ . To this end, we run NES on 8 nodes formed in a tree topology with a depth of three. We launch the Coordinator on the hardware of Type A and 7 Workers on Type B and measure the throughput and latency at the final sink.

**Results.** Figures 12(a) and 12(b) reveal almost no difference in throughput and latency for all the epoch lengths. The throughput of queries  $NxQ_0$  and  $NxQ_2$  differs by 3%, and latency is similar

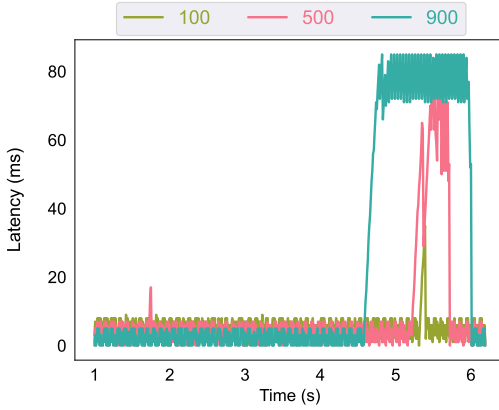


Fig. 14. Recovery latency based on varying epochs.

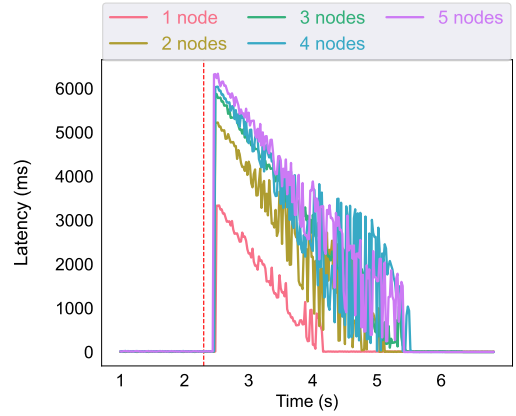


Fig. 15. Recovery latency based on the distance from the failed device.

on average. Due to aggregation,  $NxQ_8$  reduces the throughput by almost 47% compared to  $NxQ_0$ . Additionally, latency in the case of the query  $NxQ_8$  increases due to aggregation times.

The results of this experiment show that our fault tolerance approach works efficiently for a wide range of different queries.

## 6.4 Overhead

In this set of experiments, we estimate the overhead of FTPs in NES as the additional decision time of operator placement.

**Decision time.** To assess the decision time of our proposed algorithms, we conducted an experiment using data from the OpenCellid database [8]. Specifically, we extracted information about LTE, GSM, and UMTS towers located in Berlin, where LTE has 16147, GSM 29757, and UMTS 43602 towers as an example of a running USEC system. Based on each tower type, we construct a global worker topology within NES and measure NFTP, MFTP, and MFTP-H in combination with heuristic Bottom-Up (it pushes computation as much to the edges as possible) and cost-based ILP operator placement strategies. As a baseline to our FTP solutions, we introduced modifications to the ILP and Bottom-Up operator placement strategies that incorporate a reliability constraint into their decision-making processes.


The modified ILP algorithm for the LTE topology required over 5 hours to complete. Similarly, the modified Bottom-Up algorithm, which allocated an additional fault tolerance slot for each operator placed, still took around 22 minutes to decide on placement for the UMTS dataset. Finally, FTP solutions, in combination with operator placement strategies, take around 600ms to perform placement of both processing and fault tolerance operators.

**Results.** In contrast to ILP, Bottom-Up and the FTP approaches focus on a smaller set of objectives and, therefore, require less decision time. Furthermore, the FTP approaches simplify the placement decision-making process for operator placement strategies by condensing it into a single path. Overall, the NFTP, MFTP, and MFTP-H approaches reduce the decision time by an order of magnitude and provide an optimal solution in terms of reliability and operator placement.






## 6.5 Recovery

In the following experiments, we investigate how the main fault tolerance hyperparameter (Section 6.5.1) and fault tolerance placement (Section 6.5.2) affect system latency during failure recovery.

**6.5.1 Impact of varying epoch length on recovery.** To study the impact of different epoch values on processing latency during recovery, we run NES on Type B hardware with  $NxQ_0$  workload. We run

the system with 100 , 500 , and 900  epoch lengths. Every time, we simulate a worker failure and measure the processing latency.

**Results.** Figure 14 shows the longest 1.3s spike for the 900 epoch length, 0.2s for 500, and 20ms for 100. Larger epoch lengths result in storing more buffers, extending the duration of the recovery process. On the one hand, smaller epoch lengths necessitate frequent trimming, which can reduce overall throughput. Therefore, the epoch length serves as a tool to balance system throughput and recovery latency.

*6.5.2 Recovery based on fault tolerance placement.* In Figure 15, we explore the correlation between recovery latency and reliability levels, i.e., how many nodes are between the failed node and the closest node with the fault tolerance operator deployed. To this end, we run NES on Type B hardware with  $NxQ_0$  workload on a chained topology with six nodes. We submit a query five times and fail one node shortly after two seconds. We increase the number of nodes between the failed node and the last node with fault tolerance deployed every time. The distance includes one node , 2 , 3 , 4 , and 5  nodes.

**Results.** The latency spikes due to data getting resend from the upstream backup. Figure 15 demonstrates that with the one-node distance between the failed node and the node with the upstream backup deployed, the latency increases to 3s. Moreover, increasing the distance to two nodes further increases latency to 5s. Conversely, distances 3, 4, and 5 differ in latency by less than 10%, staying around 6s.

Overall, lower reliability levels increase recovery latency due to larger data storage size, highlighting the trade-off between data preservation and recovery speed. For instance, in scenarios with LOW reliability, where only 25% of the devices participate in fault tolerance to preserve data, recovery times can grow, particularly in large topology depths, as data may need to traverse the entire depth in the worst-case scenario. Notably, we observe that placing fault tolerance mechanisms further from the failed device results in longer recovery times. However, when fault tolerance mechanisms are positioned closer to the data source, the risk of data loss decreases. This analysis reveals the important trade-off between recovery time and data preservation that must be considered when optimizing fault tolerance placement.

## 7 RELATED WORK

The fault tolerance approaches developed for SPEs vary based on their targeted environment. As USEC combines Sensor, Edge, and Cloud, we further classify fault tolerance approaches based on these distinct environments.

**Cloud-based SPEs.** Existing cloud-tailored fault tolerance solutions like TelegraphCQ [27], Aurora [11], Flink [22], Rhino [32] and Heron [45] take advantage of virtually unlimited resources, all-to-all connectivity and reliable networks. For instance, Aurora employs Upstream Backup, logging tuples in their output queues at upstream nodes, while TelegraphCQ utilizes Active Standby, allowing secondary nodes to process tuples in parallel with primary nodes. In the case of Flink, it leverages brokers like Kafka for data stream reproducibility and checkpointing to maintain a consistent global state. However, all these solutions do not consider the resources required for fault tolerance and obviously deploy it on every device. While Aurora and Flink analyze resources for load management, they do not estimate the costs associated with fault tolerance during the job deployment. This lack of resource awareness can result in job deployment on devices with insufficient resources, potentially causing underprovisioning issues before the load-balancing algorithm identifies resource shortages.

**Wireless Sensor Networks.** Existing approaches for sensor processing systems like TinyDB [50] and Cougar [80] focus more on resource utilization due to the limited hardware capabilities of sensors and adaptivity to network fluctuations due to the high distributivity of the processing nodes. They enable simple filtering and aggregation queries while considering battery capacities and unreliable

networks. Existing fault-tolerance solutions for Wireless Sensor Networks systems fail to provide strong processing guarantees and cannot ensure reliability for computationally intensive processing.

**Edge-based SPEs.** Systems like Microsoft Azure IoT Edge [4], AWS Greengrass [1], CSA [69], and Frontier [59] function without connectivity to the cloud. In the first three systems, data processing is performed on hub devices, assuming that all nodes are connected to the hub device. These approaches offer fault-tolerance only between hub-devices and the cloud but still require a stable connection between sensors and the hub-device. In addition, they do not reflect the multi-level hierarchical topologies that are typical for USEC environments. As an alternative, Frontier replicates its operators across multiple nodes and maintains dynamic routing of data between replicas. Implementing one of the most demanding fault tolerance mechanisms in the Edge, Frontier still lacks resource estimation for fault tolerance, similar to Microsoft Azure, AWS Greengrass, and CSA.

Overall, all the abovementioned approaches lack awareness of the resources required by fault tolerance. They run in silos on every device, independently of the resource availability and system load. Additionally, none of these approaches adapt fault tolerance parameters during runtime to enhance system resource management.

## 8 CONCLUSION

This paper introduces FTP approaches designed to determine optimal fault tolerance placement and estimate associated costs in large-scale sensor-edge-cloud environments. These approaches emphasize the critical factors of reliability and efficient resource utilization, with the primary goal of avoiding underprovisioning and system failures. Using resource-reliability estimates, our FTP methods identify the optimal path and number of devices for fault tolerance placement. The final placement cost is integrated into the operator placement strategy to avoid underprovisioning. Furthermore, our cost-based FTP solutions support adaptability to dynamic system changes through real-time adjustments and hyperparameter tuning.

Our evaluation highlights the impact of fault tolerance on system performance. It reveals that existing solutions often lead to data loss, device overutilization, and unnecessary resource consumption. In contrast, our FTP approaches effectively mitigate potential failures and significantly reduce placement decision time. Our lightweight implementation of fault tolerance outperforms comparable state-of-the-art approaches in terms of throughput by an order of magnitude. This positions FTP solutions as the cornerstone for efficient resource allocation and the support of reliable continuous data stream processing in heterogeneous environments of massive scale.

## 9 ACKNOWLEDGEMENT

This work was funded by the DFG Priority Program (MA4662/5-2) and the German Federal Ministry for Education and Research as BIFOLD—Berlin Institute for the Foundations of Learning and Data (ref. 01IS18025A and ref. 01IS18037A). We thank the NebulaStream team for their insightful comments and fruitful discussions.



## REFERENCES

- [1] 2007. Amazon AWS Greengrass. Accessed May 2023: <https://aws.amazon.com/greengrass/>.
- [2] 2011. John Wilkes. More Google Cluster Data. Google Research Blog,. Accessed Sep 2023: <https://ai.googleblog.com/2011/11/more-google-cluster-data.html>.
- [3] 2016. Optimal operator placement for distributed stream processing applications. *DEBS 2016 - Proceedings of the 10th ACM International Conference on Distributed and Event-Based Systems*, 69–80. <https://doi.org/10.1145/2933267.2933312>
- [4] 2017. Microsoft Azure IoT Edge. Accessed Jul 2023: <https://azure.microsoft.com/en-us/services/iot-edge/>.
- [5] 2022. Calculated MTBF Estimates. Accessed Mar 2023: [https://www.intel.com/content/dam/support/us/en/documents/motherboards/server/sb/s1200rpcalculatedmtbfestimatesrev1\\_0.pdf](https://www.intel.com/content/dam/support/us/en/documents/motherboards/server/sb/s1200rpcalculatedmtbfestimatesrev1_0.pdf).
- [6] 2022. How Long Will a Raspberry Pi Last? Accessed Dec 2019: <https://raspberrypi.com/how-long-will-a-raspberry-pi-last/>.
- [7] 2022. MBDA CFMS. Accessed May 2023: <https://www.mbda-systems.com/solutions-and-services/subsystems-components/>.
- [8] 2022. OpenCellid. Accessed Apr 2023: <https://www.opencellid.org/>.
- [9] 2022. Raspberry Pi 5.1V DC. Accessed Apr 2023: <https://www.eetgroup.com/en-eu/t5875dv-raspberry-pi-13w-plug-in-power-supply-51v-25a-white-micro-usb-wid-w124475952>.
- [10] 2023. What edge computing means for infrastructure and operations leaders. Accessed Sep 2023: <https://www.gartner.com/smarterwithgartner/what-edge-computing-means-for-infrastructure-and-operations-leaders>.
- [11] D. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, C. Erwin, E. Galvez, M. Hatoun, A. Maskey, A. Rasin, A. Singer, M. Stonebraker, N. Tabul, Y. Xing, R. Yan, and S. Zdonik. 2003. Aurora: a data stream management system. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data (San Diego, California) (SIGMOD '03)*. Association for Computing Machinery, New York, NY, USA, 666. <https://doi.org/10.1145/872757.872855>
- [12] Tyler Akidau, Edmon Begoli, Slava Chernyak, Fabian Hueske, Kathryn Knight, Kenneth Knowles, Daniel Mills, and Dan Sotolongo. 2021. Watermarks in stream processing systems: semantics and comparative analysis of Apache Flink and Google cloud dataflow. *Proc. VLDB Endow.* 14, 12 (jul 2021), 3135–3147. <https://doi.org/10.14778/3476311.3476389>
- [13] C. Albrecht, R. Koch, T. Pionteck, and P. Gloesekoetter. 2009. Towards a Flexible Fault-Tolerant System-on-Chip. In *22th International Conference on Architecture of Computing Systems 2009*. 1–8.
- [14] Leonardo Aniello, Roberto Baldoni, and Leonardo Querzoni. 2013. Adaptive Online Scheduling in Storm. In *Proceedings of the 7th ACM International Conference on Distributed Event-Based Systems (Arlington, Texas, USA) (DEBS '13)*. Association for Computing Machinery, New York, NY, USA, 207–218. <https://doi.org/10.1145/2488222.2488267>
- [15] Amit Shukla Karthik Ramasamy Jignesh M. Patel Sanjeev Kulkarni Jason Jackson Krishna Gade Maosong Fu Jake Donham Nikunj Bhagat Sailesh Mittal Dmitriy Ryaboy Ankit Toshniwal, Siddarth Taneja. 2014. Storm @Twitter. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 147–156. <https://doi.org/10.1145/2588555.2595641>
- [16] Arvind Arasu, Shivnath Babu, and Jennifer Widom. 2004. The CQL Continuous Query Language: Semantic Foundations and Query Execution. *VLDB J.* 2 (03 2004). <https://doi.org/10.1007/s00778-004-0147-z>
- [17] Hamid Reza Arkian, Abolfazl Diyanat, and Atefe Pourkhalili. 2017. MIST: Fog-based data analytics scheme with cost-efficient resource provisioning for IoT crowdsensing applications. *Journal of Network and Computer Applications* 82 (2017), 152–165. <https://doi.org/10.1016/j.jnca.2017.01.012>
- [18] Michael Armbrust, Tathagata Das, Joseph Torres, Burak Yavuz, Shixiong Zhu, Reynold Xin, Ali Ghodsi, Ion Stoica, and Matei Zaharia. 2018. Structured Streaming: A Declarative API for Real-Time Applications in Apache Spark. In *Proceedings of the 2018 International Conference on Management of Data (Houston, TX, USA) (SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 601–613. <https://doi.org/10.1145/3183713.3190664>
- [19] Nathan Backman, Rodrigo Fonseca, and Uunefinedur Çetintemel. 2012. Managing Parallelism for Stream Processing in the Cloud. In *Proceedings of the 1st International Workshop on Hot Topics in Cloud Data Processing (Bern, Switzerland) (HotCDP '12)*. Association for Computing Machinery, New York, NY, USA, Article 1, 5 pages. <https://doi.org/10.1145/2169090.2169091>
- [20] Magdalena Balazinska, Hari Balakrishnan, Samuel R. Madden, and Michael Stonebraker. 2008. Fault-tolerance in the borealis distributed stream processing system. *ACM Trans. Database Syst.* 33, 1, Article 3 (mar 2008), 44 pages. <https://doi.org/10.1145/1331904.1331907>
- [21] Christian Berger, Philipp Eichhammer, Hans P. Reiser, Jörg Domaschka, Franz J. Hauck, and Gerhard Habiger. 2022. A Survey on Resilience in the IoT: Taxonomy, Classification, and Discussion of Resilience Mechanisms. *Comput. Surveys* 54 (9 2022). Issue 7. <https://doi.org/10.1145/3462513>
- [22] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. 2017. State Management in Apache Flink®: Consistent Stateful Distributed Stream Processing. *Proc. VLDB Endow.* 10, 12 (aug 2017), 1718–1729. <https://doi.org/10.14778/3137765.3137777>
- [23] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink™: Stream and Batch Processing in a Single Engine. *IEEE Data Eng. Bull.* 38 (2015), 28–38.

- [24] Paris Carbone, Jonas Traub, Asterios Katsifodimos, Seif Haridi, and Volker Markl. 2016. Cutty: Aggregate Sharing for User-Defined Windows. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management* (Indianapolis, Indiana, USA) (CIKM '16). Association for Computing Machinery, New York, NY, USA, 1201–1210. <https://doi.org/10.1145/2983323.2983807>
- [25] Valeria Cardellini, Francesco Lo Presti, Matteo Nardelli, and Gabriele Russo Russo. 2022. Runtime Adaptation of Data Stream Processing Systems: The State of the Art. *ACM Comput. Surv.* 54, 11s, Article 237 (sep 2022), 36 pages. <https://doi.org/10.1145/3514496>
- [26] Ugur Cetintemel, Jiang Du, Tim Kraska, Samuel Madden, David Maier, John Meehan, Andrew Pavlo, Michael Stonebraker, Erik Sutherland, Nesime Tatbul, Kristin Tuftte, Hao Wang, and Stanley Zdonik. 2014. S-Store: a streaming NewSQL system for big velocity applications. *Proc. VLDB Endow.* 7, 13 (aug 2014), 1633–1636. <https://doi.org/10.14778/2733004.2733048>
- [27] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel R. Madden, Fred Reiss, and Mehul A. Shah. 2003. TelegraphCQ: continuous dataflow processing. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data* (San Diego, California) (SIGMOD '03). Association for Computing Machinery, New York, NY, USA, 668. <https://doi.org/10.1145/872757.872857>
- [28] Rémy Charayron, Thierry Lefebvre, Nathalie Bartoli, and Joseph Morlier. 2023. Towards a multi-fidelity & multi-objective Bayesian optimization efficient algorithm. *Aerospace Science and Technology* 142 (2023), 108673. <https://doi.org/10.1016/j.ast.2023.108673>
- [29] Andreas Chatzistergiou and Stratis D. Viglas. 2014. Fast Heuristics for Near-Optimal Task Allocation in Data Stream Processing over Clusters. In *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management* (Shanghai, China) (CIKM '14). Association for Computing Machinery, New York, NY, USA, 1579–1588. <https://doi.org/10.1145/2661829.2661882>
- [30] Ankit Chaudhary, Steffen Zeuch, and Volker Markl. 2020. Governor: Operator Placement for a Unified Fog-Cloud Environment. (2020), 631–634. <https://doi.org/10.5441/002/EDBT.2020.81>
- [31] Ankit Chaudhary, Steffen Zeuch, Volker Markl, and Jeyhun Karimov. 2023. Incremental Stream Query Merging. In *Proceedings 26th International Conference on Extending Database Technology, EDBT 2023, Ioannina, Greece, March 28-31, 2023*, Julia Stoyanovich, Jens Teubner, Nikos Mamoulis, Evaggelia Pitoura, Jan Mühlrig, Katja Hose, Sourav S. Bhowmick, and Matteo Lissandrini (Eds.). OpenProceedings.org, 604–617. <https://doi.org/10.48786/EDBT.2023.51>
- [32] Bonaventura Del Monte, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2020. Rhino: Efficient Management of Very Large Distributed State for Stream Processing Engines. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) (SIGMOD '20). Association for Computing Machinery, New York, NY, USA, 2471–2486. <https://doi.org/10.1145/3318464.3389723>
- [33] Michael Emmerich and André Deutz. 2018. A tutorial on multiobjective optimization: fundamentals and evolutionary methods. *Natural Computing* 17 (09 2018). <https://doi.org/10.1007/s11047-018-9685-y>
- [34] Philipp M. Grulich, Aljoscha P. Lepping, Dwi Prasetyo Adi Nugroho, Varun Pandey, Bonaventura Del Monte, Steffen Zeuch, and Volker Markl. 2023. Towards Unifying Query Interpretation and Compilation. In *13th Conference on Innovative Data Systems Research, CIDR 2023, Amsterdam, The Netherlands, January 8-11, 2023*. [www.cidrdb.org](https://www.cidrdb.org). <https://www.cidrdb.org/cidr2023/papers/p49-grulich.pdf>
- [35] Philipp M. Grulich, Breß Sebastian, Steffen Zeuch, Jonas Traub, Janis von Bleichert, Zongxiong Chen, Tilmann Rabl, and Volker Markl. 2020. Grizzly: Efficient Stream Processing Through Adaptive Query Compilation. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) (SIGMOD '20). Association for Computing Machinery, New York, NY, USA, 2487–2503. <https://doi.org/10.1145/3318464.3389739>
- [36] Philipp Marian Grulich, Steffen Zeuch, and Volker Markl. 2021. Babelfish: Efficient Execution of Polyglot Queries. *Proc. VLDB Endow.* 15, 2 (2021), 196–210. <https://doi.org/10.14778/3489496.3489501>
- [37] Divya Gupta, Shalli Rani, and Syed Hassan Ahmed Shah. 2022. ICN-Fog Computing for IoT-Based Healthcare. 19–37. <https://doi.org/10.1002/9781119816829.ch2>
- [38] Shahid Sultan Hajam and Shabir Ahmad Sofi. 2021. IoT-Fog architectures in smart city applications: A survey. *China Communications* 18, 11 (2021), 117–140. <https://doi.org/10.23919/JCC.2021.11.009>
- [39] Mark Hung. 2017. *Leading the iot, gartner outsiders on how to lead in a connected world*.
- [40] J.-H. Hwang, M. Balazinska, A. Rasin, U. Cetintemel, M. Stonebraker, and S. Zdonik. 2005. High-availability algorithms for distributed stream processing. In *21st International Conference on Data Engineering (ICDE '05)*. 779–790. <https://doi.org/10.1109/ICDE.2005.72>
- [41] Jeong-Hyon Hwang, Ying Xing, Ugur Cetintemel, and Stan Zdonik. 2007. A Cooperative, Self-Configuring High-Availability Solution for Stream Processing. In *2007 IEEE 23rd International Conference on Data Engineering*. 176–185. <https://doi.org/10.1109/ICDE.2007.367863>
- [42] Gerrit Janßen, Ilya Verbitskiy, Thomas Renner, and Lauritz Thamsen. 2018. Scheduling Stream Processing Tasks on Geo-Distributed Heterogeneous Resources. In *2018 IEEE International Conference on Big Data (Big Data)*. 5159–5164. <https://doi.org/10.1109/BigData.2018.8622651>

- [43] Vasiliki Kalavri, John Liagouris, Moritz Hoffmann, Desislava Dimitrova, Matthew Forshaw, and Timothy Roscoe. 2018. Three Steps is All You Need: Fast, Accurate, Automatic Scaling Decisions for Distributed Streaming Dataflows. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) (*OSDI'18*). USENIX Association, USA, 783–798.
- [44] Fahad Khan, Muhammad Abu Bakar Siddiqui, Ateeq Ur Rehman, Jawad Khan, Muhammad Tariq Sadiq Adeel Asad, and Adeel Asad. 2020. IoT Based Power Monitoring System for Smart Grid Applications. In *2020 International Conference on Engineering and Emerging Technologies (ICEET)*. 1–5. <https://doi.org/10.1109/ICEET48479.2020.9048229>
- [45] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthik Ramasamy, and Siddarth Taneja. 2015. Twitter Heron: Stream Processing at Scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) (*SIGMOD '15*). Association for Computing Machinery, New York, NY, USA, 239–250. <https://doi.org/10.1145/2723372.2742788>
- [46] YongChul Kwon, Magdalena Balazinska, and Albert Greenberg. 2008. Fault-tolerant stream processing using a distributed, replicated file system. *Proc. VLDB Endow.* 1, 1 (aug 2008), 574–585. <https://doi.org/10.14778/1453856.1453920>
- [47] Aljoscha P. Lepping, Hoang Mi Pham, Laura Mons, Balint Rueb, Philipp M. Grulich, Ankit Chaudhary, Steffen Zeuch, and Volker Markl. 2023. Showcasing Data Management Challenges for Future IoT Applications with NebulaStream. *Proc. VLDB Endow.* 16, 12 (2023), 3930–3933. <https://doi.org/10.14778/3611540.3611588>
- [48] Jian Li, Amol Deshpande, and Samir Khuller. 2009. Minimizing Communication Cost in Distributed Multi-query Processing. In *2009 IEEE 25th International Conference on Data Engineering*. 772–783. <https://doi.org/10.1109/ICDE.2009.85>
- [49] Wei Lin, Haochuan Fan, Zhengping Qian, Junwei Xu, Sen Yang, Jingren Zhou, and Lidong Zhou. 2016. STREAMSCOPE: continuous reliable distributed processing of big data streams. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation* (Santa Clara, CA) (*NSDI'16*). USENIX Association, USA, 439–453.
- [50] Samuel Madden, Michael Franklin, Joseph Hellerstein, and Wei Hong. 2005. TinyDB: An Acquisitional Query Processing System for Sensor Networks. *ACM Transactions on Database Systems* 30 (03 2005), 122–173. <https://doi.org/10.1145/1061318.1061322>
- [51] Marco Marcozzi, Orhan Gemikonakli, Eser Gemikonakli, Enver Ever, and Leonardo Mostarda. 2023. Availability evaluation of IoT systems with Byzantine fault-tolerance for mission-critical applications. *Internet of Things* 23 (2023), 100889. <https://doi.org/10.1016/j.iot.2023.100889>
- [52] R. Marler and Jasbir Arora. 2004. Survey of Multi-Objective Optimization Methods for Engineering. *Structural and Multidisciplinary Optimization* 26 (04 2004), 369–395. <https://doi.org/10.1007/s00158-003-0368-6>
- [53] A. Messac, A. Ismail-Yahaya, and C.A. Mattson. 2003. The normalized normal constraint method for generating the Pareto frontier. *Structural and Multidisciplinary Optimization* 25 (07 2003), 86–98. <https://doi.org/10.1007/s00158-002-0276-1>
- [54] Adrian Michalke, Philipp M. Grulich, Clemens Lutz, Steffen Zeuch, and Volker Markl. 2021. An Energy-Efficient Stream Join for the Internet of Things. In *Proceedings of the 17th International Workshop on Data Management on New Hardware* (Virtual Event, China) (*DAMON '21*). Association for Computing Machinery, New York, NY, USA, Article 8, 6 pages. <https://doi.org/10.1145/3465998.3466005>
- [55] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) (*SOSP '13*). Association for Computing Machinery, New York, NY, USA, 439–455. <https://doi.org/10.1145/2517349.2522738>
- [56] Matteo Nardelli, Valeria Cardellini, Vincenzo Grassi, and Francesco Lo Presti. 2019. Efficient Operator Placement for Distributed Data Stream Processing Applications. *IEEE Transactions on Parallel and Distributed Systems* 30, 8 (2019), 1753–1767. <https://doi.org/10.1109/TPDS.2019.2896115>
- [57] Shadi A. Noghbi, Kartik Paramasivam, Yi Pan, Navina Ramesh, Jon Bringham, Indranil Gupta, and Roy H. Campbell. 2017. Samza: stateful scalable stream processing at LinkedIn. *Proc. VLDB Endow.* 10, 12 (aug 2017), 1634–1645. <https://doi.org/10.14778/3137765.3137770>
- [58] Dwi P. A. Nugroho, Philipp M. Grulich, Steffen Zeuch, Clemens Lutz, Stefano Bortoli, and Volker Markl. 2024. Benchmarking Stream Join Algorithms on GPUs: A Framework and its Application to the State-of-the-art. In *Proceedings 27th International Conference on Extending Database Technology, EDBT 2024, Paestum, Italy, March 25 - March 28*, Letizia Tanca, Qiong Luo, Giuseppe Polese, Loredana Caruccio, Xavier Oriol, and Donatella Firmani (Eds.). OpenProceedings.org, 188–200. <https://doi.org/10.48786/EDBT.2024.17>
- [59] Dan O’Keeffe, Theodoros Salonidis, and Peter Pietzuch. 2018. Frontier: resilient edge processing for the internet of things. *Proc. VLDB Endow.* 11, 10, 1178–1191. <https://doi.org/10.14778/3231751.3231767>
- [60] Boyang Peng, Mohammad Hosseini, Zhihao Hong, Reza Farivar, and Roy Campbell. 2015. R-Storm. In *Proceedings of the 16th Annual Middleware Conference*. ACM. <https://doi.org/10.1145/2814576.2814808>
- [61] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer. 2006. Network-Aware Operator Placement for Stream-Processing Systems. In *22nd International Conference on Data Engineering (ICDE'06)*. 49–49. <https://doi.org/10.1109/ICDE.2006.105>

- [62] Stamatia Rizou, Frank Dürr, and Kurt Rothermel. 2010. Solving the Multi-Operator Placement Problem in Large-Scale Operator Networks. In *2010 Proceedings of 19th International Conference on Computer Communications and Networks*. 1–6. <https://doi.org/10.1109/ICCCN.2010.5560127>
- [63] Hooman Peiro Sajjad, Ken Danniswara, Ahmad Al-Shishtawy, and Vladimir Vlassov. 2016. SpanEdge: Towards Unifying Stream Processing over Central and Near-the-Edge Data Centers. In *2016 IEEE/ACM Symposium on Edge Computing (SEC)*. 168–178. <https://doi.org/10.1109/SEC.2016.17>
- [64] Matthias J. Sax, Guozhang Wang, Matthias Weidlich, and Johann-Christoph Freytag. 2018. Streams and Tables: Two Sides of the Same Coin. In *Proceedings of the International Workshop on Real-Time Business Intelligence and Analytics (Rio de Janeiro, Brazil) (BIRTE '18)*. Association for Computing Machinery, New York, NY, USA, Article 1, 10 pages. <https://doi.org/10.1145/3242153.3242155>
- [65] Nils L Schubert, Philipp M Grulich, Steffen Zeuch, and Volker Markl. 2023. Exploiting Access Pattern Characteristics for Join Reordering. In *Proceedings of the 19th International Workshop on Data Management on New Hardware (Seattle, WA, USA) (DaMoN '23)*. Association for Computing Machinery, New York, NY, USA, 10–18. <https://doi.org/10.1145/3592980.3595304>
- [66] Zoe Sebepeou and Kostas Magoutis. 2011. CEC: Continuous eventual checkpointing for data stream processing operators. 145 – 156. <https://doi.org/10.1109/DSN.2011.5958214>
- [67] Kinza Shafique, Bilal A. Khawaja, Farah Sabir, Sameer Qazi, and Muhammad Mustaqim. 2020. Internet of Things (IoT) for Next-Generation Smart Systems: A Review of Current Challenges, Future Trends and Prospects for Emerging 5G-IoT Scenarios. *IEEE Access* 8 (2020), 23022–23040. <https://doi.org/10.1109/ACCESS.2020.2970118>
- [68] Haris Moazam Sheikh and Philip S. Marcus. 2022. Bayesian optimization for mixed-variable, multi-objective problems. *Struct. Multidiscip. Optim.* 65, 11 (nov 2022), 14 pages. <https://doi.org/10.1007/s00158-022-03382-y>
- [69] Zhitao Shen, Vikram Kumaran, Michael J. Franklin, Sailesh Krishnamurthy, Amit Bhat, Madhu Kumar, Robert Lerche, and Kim Macpherson. 2015. CSA: Streaming Engine for Internet of Things. *IEEE Data Eng. Bull.* 38, 4 (2015), 39–50. <http://sites.computer.org/debull/A15dec/p39.pdf>
- [70] Rathin Chandra Shit and Suraj Sharma. 2018. Localization for Autonomous Vehicle: Analysis of Importance of IoT Network Localization for Autonomous Vehicle Applications. In *2018 International Conference on Applied Electromagnetics, Signal Processing and Communication (AESPC)*, Vol. 1. 1–6. <https://doi.org/10.1109/AESPC44649.2018.9033329>
- [71] Fei Song, Khaled Zaouk, Chenghao Lyu, Arnab Sinha, Qi Fan, Yanlei Diao, and Prashant Shenoy. 2021. Spark-based Cloud Data Analytics using Multi-Objective Optimization. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. 396–407. <https://doi.org/10.1109/ICDE51399.2021.00041>
- [72] Marcelo Sousa and Isil Dillig. 2016. Cartesian Hoare Logic for Verifying K-Safety Properties. *SIGPLAN Not.* 51, 6 (jun 2016), 57–69. <https://doi.org/10.1145/2980983.2908092>
- [73] Ioana Stanoi, George Mihaila, Themis Palpanas, and Christian Lang. 2007. WhiteWater: Distributed Processing of Fast Streams. *IEEE Trans. on Knowl. and Data Eng.* 19, 9 (sep 2007), 1214–1226. <https://doi.org/10.1109/TKDE.2007.1056>
- [74] Li Su and Yongluan Zhou. 2016. Tolerating correlated failures in Massively Parallel Stream Processing Engines. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. 517–528. <https://doi.org/10.1109/ICDE.2016.7498267>
- [75] Peter A. Tucker, Kristin Tufte, Vassilis Papadimos, and David Maier. 2002. NEXMark – A Benchmark for Queries over Data Streams DRAFT. <https://api.semanticscholar.org/CorpusID:18302897>
- [76] Prasang Upadhyaya, YongChul Kwon, and Magdalena Balazinska. 2011. A latency and fault-tolerance optimizer for online parallel query plans. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data (Athens, Greece) (SIGMOD '11)*. Association for Computing Machinery, New York, NY, USA, 241–252. <https://doi.org/10.1145/1989323.1989350>
- [77] Huayong Wang, Li-Shiuan Peh, Emmanouil Koukoumidis, Shao Tao, and Mun Choon Chan. 2012. Meteor Shower: A Reliable Stream Processing System for Commodity Data Centers. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*. 1180–1191. <https://doi.org/10.1109/IPDPS.2012.108>
- [78] Hongyan Wang, Hua Xu, Yuan Yuan, and Zeqiu Zhang. 2022. An adaptive batch Bayesian optimization approach for expensive multi-objective problems. *Information Sciences* 611 (2022), 446–463. <https://doi.org/10.1016/j.ins.2022.08.021>
- [79] Jielong Xu, Zhenhua Chen, Jian Tang, and Sen Su. 2014. T-Storm: Traffic-Aware Online Scheduling in Storm. In *2014 IEEE 34th International Conference on Distributed Computing Systems*. 535–544. <https://doi.org/10.1109/ICDCS.2014.61>
- [80] Yong Yao and Johannes Gehrke. 2002. The cougar approach to in-network query processing in sensor networks. *SIGMOD Rec.* 31, 3 (sep 2002), 9–18. <https://doi.org/10.1145/601858.601861>
- [81] Steffen Zeuch, Ankit Chaudhary, Bonaventura Del Monte, Haralampos Gavrilidis, Dimitrios Giouroukis, Philipp M. Grulich, Sebastian Breß, Jonas Traub, and Volker Markl. 2020. The NebulaStream Platform for Data and Application Management in the Internet of Things. (2020). <http://cidrdb.org/cidr2020/papers/p7-zeuch-cidr20.pdf>
- [82] Qian Zhu and Gagan Agrawal. 2008. Resource Allocation for Distributed Streaming Applications. In *2008 37th International Conference on Parallel Processing*. 414–421. <https://doi.org/10.1109/ICPP.2008.49>

- [83] Ariane Ziehn, Philipp M. Grulich, Steffen Zeuch, and Volker Markl. 2024. Bridging the Gap: Complex Event Processing on Stream Processing Systems. In *Proceedings 27th International Conference on Extending Database Technology, EDBT 2024, Paestum, Italy, March 25 - March 28*, Letizia Tanca, Qiong Luo, Giuseppe Polese, Loredana Caruccio, Xavier Oriol, and Donatella Firmani (Eds.). OpenProceedings.org, 447–460. <https://doi.org/10.48786/EDBT.2024.39>

Received 20 October 2023; revised 19 December 2023; accepted 23 February 2024