Frontier: Resilient Edge Processing for the Internet of Things

Dan O'Keeffe
Imperial College London
dokeeffe@imperial.ac.uk

Theodoros Salonidis
IBM T.J. Watson Research Center
tsaloni@us.ibm.com

Peter Pietzuch
Imperial College London
prp@imperial.ac.uk

ABSTRACT

In an *edge* deployment model, Internet-of-Things (IoT) applications, e.g. for building automation or video surveillance, must process data *locally* on IoT devices without relying on permanent connectivity to a cloud backend. The ability to harness the combined resources of multiple IoT devices for computation is influenced by the quality of wireless network connectivity. An open challenge is how practical edge-based IoT applications can be realised that are robust to changes in network bandwidth between IoT devices, due to interference and intermittent connectivity.

We present *Frontier*, a distributed and resilient edge processing platform for IoT devices. The key idea is to express data-intensive IoT applications as continuous data-parallel streaming queries and to improve query throughput in an unreliable wireless network by exploiting *network path diversity*: a query includes *operator replicas* at different IoT nodes, which increases possible network paths for data. Frontier dynamically routes stream data to operator replicas based on network path conditions. Nodes probe path throughput and use *backpressure stream routing* to decide on transmission rates, while exploiting multiple operator replicas for data-parallelism. If a node loses network connectivity, a *transient disconnection recovery* mechanism reprocesses the lost data. Our experimental evaluation of Frontier shows that network path diversity improves throughput by 1.3×–2.8× for different IoT applications, while being resilient to intermittent network connectivity.

PVLDB Reference Format:
Dan O'Keeffe, Theodoros Salonidis, and Peter Pietzuch. Frontier: Resilient Edge Processing for the Internet of Things. *PVLDB*, 11 (10): 1178-1191, 2018.
DOI: https://doi.org/10.14778/3231751.3231767

1 Introduction

The sensing and compute capabilities of Internet-of-Things (IoT) devices, including embedded computers, set-top boxes, and mobile devices, have increased to the point at which they can perform complex analysis of sensed data without further infrastructure support. Example applications include in-situ video analysis for security surveillance [22, 47], image object tracking [50], and machine learning for home or industrial building automation [17, 15].

Proceedings of the VLDB Endowment, Vol. 11, No. 10

DOI: https://doi.org/10.14778/3231751.3231767

In such an *edge deployment model*, IoT devices do not require connectivity to a cloud backend to perform useful work [9, 23]. For many IoT applications offloading all processing to remote servers is infeasible or undesirable because the backhaul network connectivity is bandwidth-constrained, has high latency, or is simply unavailable. In addition, IoT devices that rely on cloud backend services risk becoming obsolete when those services are withdrawn for commercial reasons [25, 52]. Edge-based IoT applications can also assuage privacy concerns by analysing sensitive data locally.

As IoT data generation increases exponentially [29], data processing often becomes too resource-intensive to execute on a single IoT device. In this paper, we advocate that data processing should combine the resources of multiple devices available at the edge using a wireless network. This can be achieved by interconnecting them using a wireless network based on WiFi, Zigbee [67], or BlueTooth [12, 63]. In contrast to wired networks, wireless networks experience dynamically fluctuating bandwidth and connectivity between nodes due to factors such as environmental conditions, signal attenuation, interference and wireless channel contention. Both the IoT device limitations and wireless network properties make processing of IoT data at the edge challenging.

The goal of our work is to design an edge-based IoT data processing approach that (a) leverages the compute abilities of multiple IoT devices in order to process data with data-parallelism; and (b) accounts for the unreliable network conditions in wireless networks. We observe that data-intensive IoT applications can be expressed as continuous streaming queries over data streams [6]. Our core idea is to increase *network path diversity* for such queries during distributed execution by adding operator replicas. With a larger number of possible network paths for data, the system becomes more resilient to changes in wireless network conditions and can therefore achieve higher throughput using data-parallel processing

We describe **Frontier**, a distributed and resilient data-parallel edge processing platform for IoT devices. Frontier combines several new features to cooperatively execute stream queries on multiple IoT devices interconnected through a wireless mesh network:

(1) Replicated dataflow graphs. A stream query in Frontier is defined as a *replicated dataflow graph* (RDG)—a dataflow graph in which n-ary processing operators are replicated across multiple nodes. Replicated operators can be stateless or stateful. When routing data along the RDG, Frontier makes dynamic decisions about which downstream replica to send batches to. As a result, the RDG can increase the probability of using a viable network path. If multiple paths exist, the RDG utilises all of them, increasing the share of the available network bandwidth. The RDG also performs data-parallel processing of batches using multiple operator replicas.

(2) Backpressure stream routing. Frontier routes batches through the RDG using a novel network-aware *stream backpressure routing* (BSR) algorithm. BSR is inspired by the backpressure algorithm in wireless networks, which has been proven to be throughput-

optimal in theory [58] and in practice [35]. In BSR, each operator replica routes batches to downstream replicas using weights that reflect the congestion and quality of downstream network paths as well as processing rate at downstream replicas. Congestion is measured by the difference in queue backlogs between an upstream and downstream replica of RDG edge. For multi-input operators, such as stream joints, BSR maintains consistency using a distributed synchronisation mechanism, which coordinates all upstream replicas to send their batches to the same downstream stateful replica.

(3) Selective network-aware replay. Nodes in Frontier can temporarily become disconnected due to wireless interference or when moving out–range. To reduce the impact of network changes, Frontier uses decentralised *selective network-aware replay* (SNAP): each node stores processed batches for retransmission in its send buffer, and selectively acknowledges received batches to its upstream nodes. The received acknowledgements enable nodes to infer the set of batches that are still being processed by downstream operators.

When contact with a downstream operator replica is lost, the upstream node resends the unprocessed batches to other operator replicas. In contrast to prior approaches [26], SNAP minimises unnecessary re-transmissions during recovery, and supports out-of-order processing of batches that are transmitted via changing network paths. Frontier asynchronously creates new operator replicas when a disconnected replica is lost permanently.

We evaluate Frontier through a prototype implementation using a Raspberry Pi testbed and in emulated wireless mesh networks. Our results, with a range of IoT applications realised as stream queries, show that Frontier can increase throughput by 1.3×–2.8× and performance per watt by 1.2×–3.4× with respect to a number of baseline approaches, while remaining robust against network dynamics due to transient failures and mobility.

2 Edge IoT Applications

2.1 IoT application scenarios

IoT applications in many domains, including in-situ video analysis for security surveillance [22, 47], image object tracking [50], and machine learning for home or industrial automation [17, 15], must process large amounts of data continuously. While traditionally such applications relied on cloud infrastructure for processing, the increased CPU and memory capabilities of modern IoT devices make entirely autonomous *edge-based IoT applications* feasible.

As an example, consider a security IoT application for automated face recognition, as shown in Fig. 1. The goal is to continuously processes video frames from a set of camera sources and provide near instantaneous notifications when persons of interests are observed. This application is similar to existing home surveillance products such as Nest Cam [22] and Netatmo [43].

Logically the application consists of several data processing components: (i) one or more *data sources* continuously capture video frames from cameras and (ii) send them to a *face detector* component. This component analyses each frame and outputs detected faces with frame data to (iii) a *face recogniser* component. The face recogniser compares input faces to a database of persons of interest, and (iv) provides notifications with faces and matched names to a *data sink*, which displays the results to the user.

Due to its resource requirements, this application is unlikely to execute on a single IoT device with high processing throughput—current products therefore rely on a cloud back-end for face recognition [22, 43]. To retain the robustness, low latency and privacy benefits of an edge deployment model, we combine the resources of multiple devices at the edge into a distributed system.

Typically, edge IoT devices are interconnected using a wireless network based on the IEEE 802.11 [30], Zigbee [67], or the emerging BlueTooth mesh networking [12] standards. In contrast to wired

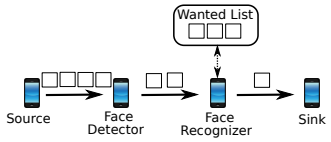


Figure 1: IoT application for face recognition

network connectivity, this introduces several challenging properties such as asymmetric and changing connectivity between devices and dynamically fluctuating network bandwidth due to environmental factors such as interference, contention, and attenuation. Interference due to hidden and exposed terminal problems also makes flow and congestion control difficult during data processing [33].

2.2 Requirements

Based on the above scenario, we describe the requirements for an edge-based IoT processing platform. In §2.3, we then explain why existing solutions fail to meet all requirements.

R1: Stream data model. IoT applications, as described in §2.1, are typically sensor-driven and thus *continuously* analyse incoming data in real-time. A natural model for such IoT applications is a *stream data model* in which where applications are expressed as continuous queries over data streams [6]. An edge-based IoT processing platform therefore should support a stream data model.

R2: Data-parallel processing. IoT applications, such as the face recognition application above, are data-intensive. Processing such volumes of data in near real-time with high throughput is often not possible on a single IoT device. Instead it requires the combined resources of multiple devices, exploiting their parallelism. An edge-based IoT processing platform should therefore support distributed *data-parallel processing* whereby multiple IoT devices each process a subset of the input data in parallel. In the above scenario, multiple devices can process different video frames each.

R3: Adaptability to network conditions. To support distributed data-parallel processing, an edge IoT platform must exchange data over a wireless network. As mentioned in §2.1, wireless network bandwidth is dynamic due to interference and mobility. To use the combined processing resources of different IoT devices efficiently, an edge-based IoT processing platform must be *network-aware*, i.e. react and adapt to changing network conditions.

R4: Recovery from transient network failures. An IoT platform that offers strong fault tolerance guarantees, e.g. by masking transient failures, simplifies application development. For the above face recognition application, fault tolerance ensures that faces are still recognised even when individual IoT devices fail or become unreachable. *Transient network failures* are more common in wireless networks than in wired networks due to interference and mobility. An edge-based IoT platform must therefore recover from such transient network failures gracefully and efficiently.

2.3 Existing IoT processing solutions

Next we survey existing solutions relevant to edge-based IoT data processing, proposed across a number of domains. Table 1 summarises how they meet the requirements outlined in §2.2.

Cloud-based IoT data processing. Today many computationally-intensive IoT application leverage remote cloud resources for data processing [22, 43] because a remote cloud can offer effectively unlimited resources. Such solutions, however, require a permanent network link to a remote cloud backend: as a result, they suffer from the constrained bandwidth available to a remote cloud, introduce additional network latency (violating **R1** above), and cannot offer any service when the remote connectivity is interrupted (violating **R4**). Offloading private or security-sensitive data to third-party cloud providers may also introduce privacy issues [19].

Table 1: Existing approaches to edge IoT processing

Approach	Stream model (R1)	Data-parallel (R2)	Network adaptivity (R3)	Network failure recovery (R4)
Cloud	✗	✗	✗	✗
Centralised edge	✗	✗	✗	✗
WSN	✗|✓	✗|✓	✗|✓	✗
Cluster stream processing	✓	✓	✗	✗
Frontier	✓	✓	✓	✓

Centralised edge-based processing. A trend for cloud-based IoT service providers is to add support for edge-based IoT services that can operate without connectivity to a cloud backend [9, 23, 56]. A *hub device* at the edge can support control-plane operations, such as configuration of IoT devices. The hub device can also collect data from other IoT devices and perform data processing. These solutions, however, are limited by the computational capabilities of the hub device: in particular, they cannot support distributed data-parallel processing across multiple devices (violating **R2**). All devices are assumed to have connectivity to the hub device, which is often not the case in wireless environments (violating **R3**) and introduces a single point-of-failure (violating **R4**).

Wireless sensor networks (WSNs) consist of multiple nodes interconnect by a wireless network and thus must be resilient to intermittent and changing network connectivity (**R3**). Prior work [37, 64] proposed approaches for distributed data processing in WSNs. Targeting resource-constrained wireless sensors, these solutions allow users to define simple filtering and aggregation queries over sensor data, reducing unnecessary communication and prolonging the battery lifetimes of sensors. Given the focus on energy efficient, there is no support for computationally-intensive applications such as video analysis, which would require parallel processing (violating **R2**). In addition, data processing in WSNs typically fails to provide strong fault-tolerance guarantees (violating **R4**).

Cluster stream processing systems [57, 66, 3, 2] offer a data stream model (**R1**) and exploit data-parallelism on a compute cluster (**R2**). These systems, however, are designed to work over a wired network with stable high-bandwidth connectivity, making them unsuitable for wireless networks (violating **R3**).

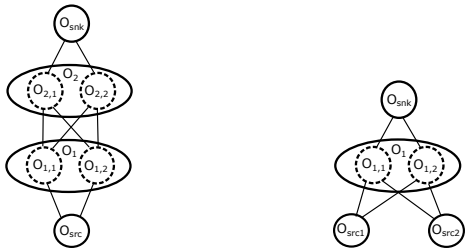
Proposals exist for systems to react to changing network conditions in wired networks: by sending redundant data to multiple workers, it is possible to reduce processing latency [27]; by migrating computation between workers, a system can increase throughput when available network bandwidth changes [46, 13]. These approaches, however, only have limited applicability in wireless broadcast networks because sending redundant data or migrating state uses additional network bandwidth and increases interference.

While distributed stream processing systems can provide strong fault-tolerance guarantees regarding crash-stop node failures [57, 66, 3, 2], they do not cope well with transient network failures (violating **R4**). They depend on network connectivity to a centralised master node for failure recovery. In our edge-based scenario, the master itself may become unavailable due to a network partition.

We introduce the design for **Frontier**, a new edge-based IoT processing platform that meets all of our requirements.

3 Edge Processing with Network Path Diversity

In this section, we first define a stream query model that we use in the remainder of the paper. We then introduce *replicated dataflow graphs* (RDGs), a new model for distributed data processing in wireless networks. RDGs increase *network path diversity*, which permits a data processing system to adapt to network conditions and thus achieve higher resilience and throughout.



(a) With multiple replicated operators **(b)** With replicated join operator

Figure 2: Stream routing over RDGs

3.1 Stream query model

Stream model. A stream $s \in \mathbb{S}$ is an infinite sequence of tuples $t \in \mathbb{T}$. $\mathbb{S}$ and $\mathbb{T}$ denote the sets of all streams and tuples, respectively. A tuple $t = (\tau, p)$ has a timestamp $\tau \in \mathbb{N}^+$ and a payload p. Each timestamp τ is assigned according to a logical clock on entry to the system or an external physical clock if correspondence with wall clock time is required (e.g. when recording event detection times).

Operator model. Tuples are processed by *operators*. An operator O is a *user-defined function* (UDF) that takes *n input streams*, denoted by $I_O = \{s_1, \ldots, s_n\}$, processes their tuples and produces one or more *output streams* O_O. For example, the face recognition component in Fig. 1 may use an operator with a UDF that implements a face recognition algorithm.

In practice, applications require operators that process more than one tuple at a time. In the face recognition application, assuming that each video frame is an input tuple, a user should only be notified when a face is recognised with the highest confidence within a time window. In line with previous work [6], we therefore assume that all operators are *windowed operators*: they process a finite sequence of tuples, or a *window*, from each input stream at a time. Windows are generated according to a *window function* $\omega_s : \mathbb{S} \rightarrow \mathbb{W}^*$ where $\mathbb{W} \subset \mathbb{S}$ is the set of all windows. Each ω_s takes a stream s as input and outputs an infinite sequence of (potentially overlapping) windows $w_s(1), w_s(2), \ldots, w_s(t), \cdots \in \mathbb{W}^*$ where $t \in \mathbb{N}$ is the window sequence number for $w_s(t)$.

Window functions are typically time- or count-based, and they are defined using *size* and *slide* parameters [6]: *size* controls the number of tuples in each window, and *slide* controls the overlap between successive windows.

Query model. A query is a directed acyclic *dataflow graph*, $Q = (\mathbb{O}, \mathbb{S})$ where $\mathbb{O}$ is the set of operators. A stream $s \in \mathbb{S}$ is a directed edge between two operators, $s = (O, O')$ where $\{O, O'\} \subseteq \mathbb{O}$. An operator U is *upstream* to O, $U \in up(O)$, when $\exists (U, O) \in \mathbb{S}$; an operator D is *downstream* to O, $D \in down(O)$, when $\exists (O, D) \in \mathbb{S}$.

A query has two special sets of operators, $U^Q = \{U^Q_1, \ldots, U^Q_n\}$ and $D^Q = \{D^Q_1, \ldots, D^Q_m\}$, which act as the sources and sinks of the n input streams I^Q and m output streams O^Q of Q, respectively. To ensure correctness for queries with time-based window functions, we require the n-tuple of window functions $\omega_Q = \langle \omega_1, \ldots, \omega_n \rangle$ over the n input streams I^Q of Q to order their corresponding input streams by tuple timestamp before generating a new window.

3.2 Replicated Dataflow Graphs

To address the dynamic nature of wireless networks (**R3**), our idea is to add *redundancy* to the query dataflow graph in the form of replicated instances of operators. Each operator is thus presented with the option of sending tuples to multiple operator replicas for processing. We refer to this as a *replicated dataflow graph* (RDG).

More formally, the RDG $\bar{Q}$ for a query Q is a directed acyclic graph that contains for each operator $O \in Q$ a set of $r \in \mathbb{N}^+$ *operator replicas* $\bar{o} = \{o_1, \cdots, o_r\}, o_i \in \bar{\mathbb{O}}$. We refer to r as the *replication*

factor of O. For simplicity, we assume r is the same for all internal operators in Q. We refer to replica o_i of a specific operator O_j as $O_{j,i}$

Network path diversity. An RDG with operator replicas placed at different nodes in a wireless network increases *network path diversity*. Intuitively, when the network path to a particular replica only achieves low throughput, an operator may achieve higher throughput by sending to a different replica over another network path. The example in Fig. 2a shows an RDG with two internal operators, $\{O_1, O_2\}$, each with two replicas. When the path from operator O_{src} to replica $O_{1,1}$ deteriorates, O_{src} can instead route tuples to $O_{1,2}$. Network path diversity has been formalised for multipath routing [53] between a single source and destination. We extend this definition to an RDG.

DEFINITION 1. *RDG placement. Given a wireless network $G = (V,E)$ where V is the set of nodes and E the set of edges, we define a* placement *M for an RDG $\bar{Q}$ as a mapping of each replica $O_{i,j}$ in $\bar{Q}$ to a node $v \in V$.*

DEFINITION 2. *Operator replica combination. For RDG $\bar{Q}$ of query Q, we define an* operator replica combination *$C \in \mathbb{C}$ as a subgraph of $\bar{Q}$ formed by choosing a single replica $O_{i,j}$ for each operator O_i of Q. $\mathbb{C}$ denotes the set of all possible subgraphs of $\bar{Q}$.*

DEFINITION 3. *Pairwise network path diversity. Given a combination C for an RDG $\bar{Q}$, and a placement M of $\bar{Q}$ onto a wireless network $G = (V,E)$, we define a path $P_C(t) \subset E$ as the set of edges in G that the routing algorithm for G uses at time t to send data between all pairs of connected replicas $(O_{i,j}, O_{k,l})$ in C. Given two paths P_a and P_b, we define the pairwise path diversity D_{ab} as:*

$$D_{ab} = 1 - \frac{|P_b \cap P_a|}{|P_a|} \tag{1}$$

DEFINITION 4. *RDG network path diversity. The RDG network path diversity D_{RDG} is defined as:*

$$D_{RDG} = \sum_{C \in \mathbb{C}} D_{min}(P_C) \tag{2}$$

where $D_{min}(P_C)$ is the minimum pairwise path diversity for combination C when evaluated against all combinations in $\mathbb{C}$.

Information about the wireless network topology, if available, can be used to maximise the network path diversity of the initial placement of an RDG. Given the dynamic nature of a wireless network, our goal is not to find an optimal initial placement because this may quickly become outdated. Instead, we aim to introduce sufficient network path diversity through operator replication to adapt to changes at runtime. We therefore focus on random initial placements in our experiments (§6).

3.3 RDG stream routing

Given an RDG with network path diversity, we want to perform data-parallel processing by dividing the input streams of each operator O between its replicas $\bar{o}$ at runtime in a network-aware manner. We refer to this as *stream routing* because tuples are routed over the RDG. The challenge is to achieve good performance while ensuring correctness, i.e. the output streams of the unreplicated query Q must be equivalent to the output streams of the RDG $\bar{Q}$.

Window-based routing. For correctness, window-based operators require a complete window on each input to produce a result. Therefore, stream routing must route at the granularity of windows. For example, in Fig. 2a, if O_{src} routes an outgoing tuple t from stream (O_{src}, O_1) to $O_{1,1}$, it must route all tuples in the same window $w_{(O_{src},O_1)}(1)$ to $O_{1,1}$. Only when the first tuple from the next window $w_{(O_{src},O_1)}(2)$ is available can O_{src} route tuples to $O_{1,2}$ for parallel processing.

Out-of-order processing is necessary to achieve high throughput with parallel operator instances because delays receiving a window from one replica must not block the processing of other windows: a downstream D of an operator O with replicas $\bar{o}$ may receive tuples out-of-order, e.g. due to varying network delays between each replica $o_i \in \bar{o}$ and D.

We observe though that window functions are typically only defined over the source input streams I, and all subsequent operators process tuples derived from a single window in an initial source stream. For example, in our face recognition application, the face detector processes a short window of images and outputs a corresponding window of faces to the face recogniser. In other words, when a non-source operator O processes a window $w_{(U_k,O)}(t)$ from an input stream $(U_k, O) \in I$, it outputs to each downstream D exactly one window $w_{(O,D)}(t)$. Operator O can then perform window-based routing in the same manner as a query source, with each output window inheriting its sequence number t from its corresponding input window. This model of window-based routing allows operators to process windows out-of-order while still ensuring correctness: reordering, if required, can be implemented at the sink using a reorder buffer and window sequence numbers.

Multi-input operators. Another challenge to correctness arises when an operator $O \in Q$ is a *multi-input* operator such as a join. For example, in Fig. 2b, O_1 has two inputs, O_{src1} and O_{src2}. If a specific window of tuples $w_{(O_{src1},O_1)}(1)$ in stream (O_{src1}, O_1) should be joined with a specific window $w_{(O_{src2},O_1)}(1)$ in stream (O_{src2}, O_1), O_{src1} and O_{src2} must coordinate to ensure that they route both windows to the same replica of O_1, i.e. either $O_{1,1}$ or $O_{1,2}$.

Window batching. For queries with sliding windows, i.e. where there is overlap in the tuples of adjacent windows, routing windows to different replicas results in redundant communication and processing. To mitigate this overhead, stream routing can group adjacent overlapping windows into *batches*. A batch is defined as $b = ((w_s(t), \ldots, w_s(t+n)), id)$. A batch has an identifier *id* and a payload of n complete windows over a stream $s \in \mathbb{S}$ where $n \in \mathbb{N}$ is a query parameter. Stream routing decisions are then made at the coarser granularity of batches instead of individual windows, and queries can also process batches out-of-order and reorder them at the sink. Batching exposes a trade-off for sliding windows between redundant communication (and computation) and the granularity at which stream routing can adapt to changing network conditions.

3.4 Frontier design

The Frontier architecture that is deployed on each IoT device is shown in Fig. 3. It is divided into three main components:

Processing. Each Frontier IoT device hosts one or more *operator replicas*, where each replica implements a portion of the IoT application's processing logic, defined by an RDG. An operator reads batches of tuples from its *input buffers*. Each input buffer stores batches sent by up to r upstream operator replicas for that input. An operator processes input batches and adds new output batches to the *dispatcher output buffer*, from where the *dispatcher* forwards them downstream. If there are multiple logical downstreams, each has its own dispatcher and corresponding output buffer.

Routing. A dispatcher makes forwarding decisions in collaboration with an *RDG router* component. Based on its current routing state, the router tells its dispatcher the best downstream operator replica for each outgoing batch. The router is designed to be *pluggable*, i.e. it allows a variety of routing strategies to be used. Frontier uses *backpressure stream routing* (BSR) (§4), designed to enable high-throughput data-parallel stream query processing (**R2**), and adapts to changing wireless network conditions (**R3**).

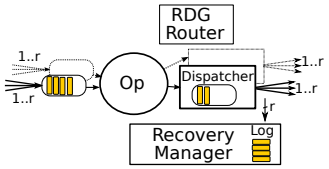


Figure 3: Frontier node architecture

Network disconnection recovery. All batches sent to a particular downstream are buffered in an *output log*. The *recovery manager* receives periodic control updates from each of its downstreams, which it uses to trim the output logs, dispatcher buffer, and input buffers. The recovery manager then computes a control update for each of the operator's upstreams. In the event of a network disconnection, the *recovery manager* performs *selective network-aware replay* (SNAP) (§5) and replays the minimum amount of data needed to ensure correct results (**R4**).

4 Backpressure Stream Routing

To exploit network path diversity in a replicated dataflow graph, we propose *backpressure stream routing* (BSR). BSR is inspired by techniques from backpressure routing in data networks [58, 35], which has been shown to maximise throughput in general networks under the assumption of perfect estimates of routing queue lengths and link capacities, per-tuple scheduling and no wireless interference[1].

4.1 Backpressure model

Next we describe the intuition behind BSR in the context of a query Q with only unary streaming operators (e.g. map, filter). We then discuss how the BSR algorithm can be generalised to handle multi-input operators (e.g. join).

Given a single replicated query Q, each operator replica o_i maintains a variable q_i^Q, which is the number of tuples in its outgoing queue for query Q. (We omit the superscripts when they are clear from the context.) We define $\bar{d}$ to be the set of downstream operator replicas of operator replica o_i, i.e. the vertices of the outgoing edges of o_i in $\bar{Q}$, the RDG of query Q.

Suppose that, if downstream replica $d_j \in \bar{d}$ is selected, the tuples of o_i are transmitted on the network link with transmission rate r_{ij} tuples/sec, processed by replica d_j with a processing rate p_j tuples/sec and placed in the outgoing queue q_j.

The backpressure algorithm is executed periodically for each replica o_i based on estimates of q_i, q_j, r_{ij}, and p_j. The update granularity for these estimates determines the time scale with which the algorithm can change its scheduling decisions.

At each execution step, for each replica o_i, the algorithm computes a weight for each downstream replica:

$$w_{ij} = max(0, (q_i - q_j) \times r_{ij} \times p_j) \quad (3)$$

where r_{ij} is the network transmission rate between o_i and d_j. p_j is the processing rate of d_j, and allows for nodes with heterogeneous processing resources. The term $q_i - q_j$ is the differential backlog over edge (i, j) and represents a gradient of congestion on that edge. The backpressure algorithm at o_i selects the optimal downstream replica $d_j{}^*$, which has the highest positive weight:

$$\mathrm{d_j}^* = argmax_{\mathrm{d_j} \in \bar{d}}(w_{ij}) \mid w_{ij} > 0 \quad (4)$$

Once the downstream operator replica is selected, batches of tuples are sent to that replica until the next scheduling decision.

[1]As we discuss in §7, backpressure routing differs from the flow control mechanism referred to as backpressure in previous stream processing systems [57, 34, 21].

Multi-input operators. The above version of the backpressure algorithm can be executed separately for each replica o_i, but assumes that replicas can make independent routing decisions. Multi-input operators (e.g. join), however, require coordination between upstream replicas to ensure correct routing decisions. For example, in Fig. 2b, replicas of input O_{src1} and input O_{src2} must coordinate to send batches to the same replica of O_1.

BSR generalises backpressure routing to multi-input operators: it (i) *aggregates* backpressure weights and (ii) adds *routing constraints*. If downstream d_j is a replica of a multi-input operator, instead of upstream replica o_i choosing $d_j{}^*$ based on w_{ij}, each d_j computes an aggregate weight w_{agg} over the individual backpressure weights w_{ij} to each of its upstreams o_i. Each d_j feeds back w_{agg} to its upstreams, which route batches to the downstream replica d_j with highest aggregate weight $\bar{w}_{agg}[d_j]$. In the absence of network delays, aggregate weights implicitly coordinate routing decisions because upstreams have the same view of the current aggregate weights $\bar{w}_{agg}$.

In practice, network delays may lead to upstreams with inconsistent views of the current downstream aggregate weights when updates arrive at different times. This could cause batches with the same identifier to be sent to different replicas of a multi-input operator. BSR minimises such inconsistent routing decisions using *routing constraints*. Multi-input operators maintain for each input a *constraint set* of the batch identifiers not yet received on that input but already received on other inputs. The constraint set is sent periodically to all upstream replicas contributing to that input. When making a routing decision, an upstream o_i first checks whether routing constraints exist for any downstream replica d_j, and if so sends the batch to d_j; otherwise, the batch is sent to downstream replica d_j with the highest aggregate weight $\bar{w}_{agg}[d_j]$ as before.

Routing constraints do not completely prevent upstreams from making inconsistent routing decisions because a routing constraint may arrive after a batch has been routed to a different downstream replica. In this case, BSR retransmits the corresponding batch to the downstream from which it received the routing constraint. Our evaluation in §6 shows that this occurs infrequently in practice.

4.2 BSR algorithm

Alg. 1 shows the BSR algorithm executed at each replica o_i. For ease-of-explanation, we assume that each operator has a single downstream D, although D may have several replicas $d_j \in \bar{d}$.

At a high level, the algorithm consists of two data plane functions, processor (line 1) and dispatcher (line 8) that receive, process and forward batches; control handlers periodically exchange routing information (i.e. queue lengths, weight updates, and constraint sets) between upstream and downstream operators (lines 16, 21, 19, 31).

On the data plane, each replica o_i has a processor, which reads batches from o_i's input buffers, processes them, and adds output batches to o_i's output buffer (lines 2–7). The processor waits until a batch with the same identifier exists in each input buffer buf_U (line 3) and then extracts the corresponding batches (lines 4–6). It executes the process function for o_i, passing it $\hat{b}$ with the batches, and adds the output batches to buf_{out}, the output buffer of o_i.

A dispatcher selects the downstream replica to route each output batch to (lines 9–15). The dispatcher first finds the oldest output batch b in buf_{out} (line 10). Next, it computes $\bar{d}_{active}$, the set of *active* downstream replicas. A downstream replica d_j is active if it has a positive aggregate backpressure weight $\bar{w}_{agg}[d_j]$ (line 11). If there is at least one active replica, the dispatcher checks whether there are additional *routing constraints* for b (line 13): if $\bar{d}$ are the replicas of a multi-input operator, a batch with the same identifier as b from an operator other than O may already have been received at one or more of the downstream replicas $d_j \in \bar{d}$. If constraints exist for b, the dispatcher restricts the set of active replicas to include only those

Algorithm 1: BSR algorithm

```
   /* Executed on replica o_i of operator O */
1  function processor()
2      while true do
3          if ∃ id ∈ ℕ⁺ : ∀ U ∈ up(O), ∃ b ∈ buf_U : b.id = id then
4              for U ∈ up(O) do
5                  b̂[U] ← b ∈ buf_U : b.id = id
6                  buf_U ← buf_U \ {b}
7              buf_out ← buf_out ∪ {process(b̂)}
8  function dispatcher()
9      while true do
10         b ← b_min ∈ buf_out : b_min.id = min(ids(buf_out))
11         d̄_active ← {d_j ∈ d̄ : w̄_agg[d_j] > 0}
12         if d̄_active ≠ ∅ then
13             if ∃ d_j ∈ d̄_active : b.id ∈ constraints[d_j] then
14                 d̄_active ← {d_j ∈ d̄_active : b.id ∈ constraints[d_j]}
15             send(b, argmax(d̄_active, w̄_agg))
16 upon SEND(RCTRL, d̄) do /* To downstreams */
17     for d_j ∈ d̄ do
18         send(|buf_out|, d_j)
19 upon RECV(RCTRL, q, u_k) do /* From upstream replica */
20     q̄[u_k] ← q
21 upon SEND(RCTRL, up(O)) do /* To upstreams */
22     for U ∈ up(O) do /* Compute upstream weights */
23         for u_k ∈ ū do w̄[u_k] ← weight(u_k) ;
24     for U ∈up(O) do
25         constraints ← (⋃_{V∈up(O),V≠U} ids(buf_V)) \ ids(buf_U)
26         for u_k ∈ ū do
27             if |up(O)| > 1 then
28                 send((agg(w̄), constraints), u_k)
29             else
30                 send((w̄[u_k], ∅), u_k)
31 upon RECV(RCTRL, (w, constraints), d_j) do /* From downstream
   */
32     w̄_agg[d_j] ← w
33     constraints[d_j] ← constraints
34 function weight(u_k)
35     return (q̄[u_k] − |buf_U| − |buf_out|)×r̄[u_k]×p_{o_i}
```

(line 14). Finally, it sends b to the active downstream replica with the highest aggregate weight (line 15).

On the control plane, each o_i exchanges *routing control messages* with its neighbouring operators to update the backpressure weights and routing constraints. Periodically, o_i sends a routing control message with its current output queue length to each replica d_j of its downstream operator (lines 17–18). On receiving an output queue length q from a replica of one of its own upstreams, o_i stores q indexed by the upstream replica (line 20).

Upon receiving a queue length update from an upstream replica, o_i sends a new routing control message to each upstream replica (lines 22–30). It first computes a backpressure weight for each upstream replica u_k (lines 22–23), according to the backpressure formula from Eq. (3), modified to include tuples queued at buf_U, the input buffer for upstream U at o_i (line 35). Next, o_i computes the current routing constraints for U as the set of all batch identifiers that are in its input buffer buf_v for any other upstream V (line 25).

After that, o_i sends a routing control message to each upstream replica u_k (lines 26–30). If O is a multi-input operator, the message contains a weight computed as the aggregate of the backpressure weights of all upstreams, in addition to any routing constraints (line 28). As an optimisation, if O is a unary operator, the unaggregated backpressure weights for the corresponding upstream replica are used because no coordination is necessary (line 30). We omit details of further optimisations for unary operators (e.g. with no coordination of upstreams through weight aggregation, backpressure weights can be calculated at upstreams).

Finally, when o_i receives a routing control message from a downstream replica d_j (lines 32–33), o_i saves the associated weight and routing constraints indexed by d_j, from where the dispatcher accesses for routing decision.

4.3 Discussion

Switching granularity. In addition to the rate at which weights are updated, the switching granularity of BSR is limited by the batch size. Intuitively, performance gains are likely highest for smaller windows because they allow for finer granularity switching. For tumbling windows, the batch size could therefore be set to one window of tuples. For sliding windows, batching exposes a trade-off between the overhead of redundant tuples and switching granularity. Given a window with length w and slide s, the overlap between two successive windows is $w-s$. If $w \simeq s$, there is little overlap and b can be small; if $w \gg s$, b should be larger. If the expected speed-up for an RDG with $w = s$ is p, for the same RDG with a smaller slide $s' \ll w$, b should be set to at least $w/(p \times s')$—otherwise the throughput gain does not improve goodput.

Depending on the operator, finer granularity switching may be achieved with *sub-batching*, similar to previous approaches for intra-window parallelisation using *panes* [36]. Sub-batching rewrites an operator into two sub-operators, a *mapper* and a *reducer*, which are replicated as usual: inputs to the mapper split their batches into several non-overlapping sub-batches, each with its own sub-batch identifier, which are routed to mapper replicas independently. Mappers route sub-batches of the same parent batch to the same reducer replica, coordinating routing decisions using BSR's support for multi-input operators. Sub-batching therefore supports finer granularity switching at the cost of extra communication.

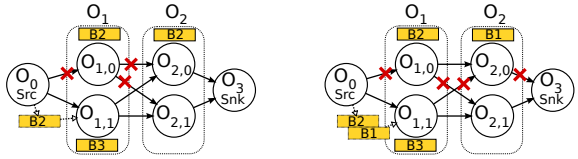
For some operations (e.g. group-by), subdivision of a single output batch into smaller independent output batches based on the values of key attributes may allow for increased parallelism. Frontier supports key-based partitioning by statically dividing the key space into a number of disjoint sub-spaces and creating a separate logical downstream for each sub-space.

Internal window functions. BSR processes batches out-of-order under the assumption that window functions are defined only over query inputs (§3.3). If a window function ω' is defined over an internal stream, e.g. stream (O_1, O_2) in Fig. 2a, each upstream replica may output only partial windows of tuples. Before applying ω', the partial windows of all upstream replicas may need to be combined and ordered. Although a distributed ordering implementation is straightforward for some window functions, Frontier instead currently introduces an additional *reorder* operator R between O_1 and O_2. This splits the query Q into two separate queries Q_1 and Q_2 such that R is a sink for Q_1 and a source for Q_2. This has the down-side that, as R is unreplicated, it reduces network path diversity.

Practical aspects. Backpressure routing is constrained by practical aspects such as accurate estimation of the network transmission rate, processing rate, and wireless interference:

Transmission rate r_{ij}^Q is the maximum achievable transmission capacity of the network path. Our current implementation uses link-state information provided by the network to compute an estimate of the path capacity under the assumption that capacity is inversely proportional to the expected transmission count (ETX [16]).

Processing rate p_j^Q can be directly measured at the downstream replicas. Alternatively, it can be estimated based on models using



(a) Single **(b)** Concurrent

Figure 4: Transient disconnection recovery

a-priori profiles for each operator that relate incoming traffic to processing time, as in our current implementation.

Wireless interference. In the wireless version of the backpressure algorithm, interference is taken into account by solving a complex centralised scheduling problem. As we require distributed operation, we indirectly capture interference by considering congestion (on top of network and processing costs) captured by the weights w_{ij}.

5 Failure Recovery

A stream processing platform designed for the data centre typically provides strong reliability guarantees such as at-least-once or exactly-once delivery [57, 65, 20]. In a wireless network, however, *transient node disconnections* make it challenging to provide strong reliability guarantees with good performance.

Fig. 4a gives an example of a query experiencing a transient network partition, e.g. caused by a node moving out of range temporarily. Having just received batch b_2 from the source, replica $O_{1,0}$ becomes disconnected from its neighbour nodes. Waiting until the source reconnects with $O_{1,0}$ again before processing b_2 would subject b_2 to an excessively long delay.

Frontier must therefore ensure fast recovery from transient node disconnections. Unlike stream processing systems in a data centre, in which typically a centralised master detects node failures and coordinates recovery [57, 65, 20], Frontier's recovery mechanism must be *distributed* because a centralised coordinator could itself become partitioned. Since in practice partitions could cause multiple nodes to disconnect simultaneously, Frontier's recovery mechanisms must also handle *concurrent disconnection* of nodes. To ensure efficient recovery in a wireless network, Frontier's recovery mechanisms must be *network-aware*. Finally, to allow normal operation to resume quickly and to alleviate congestion due to wireless interference, recovery should be *selective* and retransmit the minimum required amount of data to ensure correct results.

5.1 Selective network-aware replay

To handle transient network failures, we describe a new approach called *Selective Network-Aware rePlay* (SNAP), which exploits (i) existing replicas and (ii) their ability to process batches out-of-order.

Network-awareness. In SNAP, tuples are saved in the buffers of upstream nodes. When an upstream detects a disconnected node, it retransmits batches sent to the disconnected node to an alternative replica. For example, in Fig. 4a, the source reroutes batch b_2 through $O_{1,1}$ after $O_{1,0}$ disconnects.

For network-awareness, SNAP uses BSR during normal operation. Batches are rerouted to another replica with the current highest weight. Network-aware replay relies on the ability of replicas to process batches out-of-order. In Fig. 4a, this allows $O_{1,1}$ to process b_2 during recovery, even though it has already received b_3. Recovery solutions such as *upstream backup* [26] do not account for out-of-order processing but require a new node through which all tuples are replayed. Instead, SNAP exploits pre-existing replicas. Permanent failures are handled as in upstream backup by creating new replicas, albeit asynchronously off the critical path, after a longer timeout.

Garbage collection. SNAP uses *acknowledgements* to reclaim buffer space after tuples have been processed to completion by a query. Nodes acknowledge tuples at batch granularity using identifiers.

To guard against concurrent node disconnections, acknowledgements originate at a sink in response to the arrival of a batch. Sink-initiated acknowledgements ensure that, even if multiple intermediate nodes become partitioned, progress can still be made for all batches so long as there is a path in the RDG to the sink through alternative replicas. When an operator replica o_i receives an acknowledgement for a batch from at least one replica of each downstream $D \in \text{down}(O)$, o_i trims the batch from its state. o_i then forwards the acknowledgement to upstreams, which repeat the process.

Selective replay. SNAP must minimise unnecessary work when recovering from transient disconnections. A node should ideally only replay batches currently being processed on disconnected nodes, and not batches previously processed by a disconnected node and forwarded downstream.For example, in Fig. 4a, even if the source previously sent batches b_1 and b_2 to $O_{1,0}$, it should only replay b_2 because b_1 has already been processed and forwarded to $O_{2,0}$. Although acknowledgements help avoid some retransmissions during recovery, they do not apply in this case because b_1 has not yet been received and processed by the sink.

SNAP ensures efficient recovery through the use of batch *heartbeats*. In contrast to acknowledgements, heartbeats indicate batches that have been received downstream but not yet processed to completion. Although these are not safe to discard, they do not need to be replayed during recovery. Heartbeat messages are created at each node recursively and sent to all upstreams. A heartbeat message includes heartbeats for batches held locally in addition to heartbeats received from all downstream replicas.

When a node disconnects from a downstream, it discards all heartbeats received from that downstream. Batches previously sent to the downstream are then replayed *selectively*, depending on whether the node is still receiving heartbeats for a batch from another downstream replica. For example, in Fig. 4a, node $O_{2,0}$ sends a heartbeat for batch b_1 to upstreams $O_{1,0}$ and $O_{1,1}$, which in turn include b_1 in the heartbeats sent to the source. On detecting that $O_{1,0}$ has disconnected, the source does not replay b_1 because it continues to receive heartbeats for it from $O_{2,0}$ through $O_{1,1}$.

Another benefit of heartbeats is that they allow nodes to replay with concurrent failures using only heartbeat messages received from direct downstreams. Fig. 4b shows the same situation as Fig. 4a, except with both nodes $O_{1,0}$ and $O_{2,0}$ disconnected. When $O_{1,1}$ detects that $O_{2,0}$ has disconnected, it discards its heartbeat for batch b_1 and propagates an updated heartbeat message to the source. This triggers the source to replay b_1 in addition to b_2.

Under concurrent node failures, replay of lost batches is initiated from the furthest downstream replica holding a copy of a batch with the same identifier, reducing unnecessary retransmissions.

5.2 SNAP algorithm

Alg. 2 shows the SNAP algorithm. For ease-of-explanation, each operator O has a single downstream D, i.e. $|\text{down}(O)| = 1$), although the downstream may have multiple replicas. It also assumes that the sink has a single upstream but with multiple replicas.

When a replica o_i of operator O receives a new batch b from replica u_k of upstream U, it first checks whether b is a duplicate (lines 1–2). To do this, it computes the set of batch identifiers stored in the local input buffer for U (buf_U), output buffer of batches waiting for dispatch (buf_{out}), and log of previously transmitted tuples (log). If not found locally, b is also discarded if one of the downstreams of o_i has already received it, perhaps via a different replica, as indicated by *acks* and *hbeats*, which are the most recent acknowledgements and heartbeats o_i has received from all downstream replicas $d_j \in \bar{d}$. If not a duplicate, o_i adds b to its input buffer for u (line 3). In addition, if O is a sink, o_i adds $b.id$ to the set of identifiers safe to acknowledge (lines 4–5).

Algorithm 2: SNAP algorithm

```
   /* Executed on replica o_i of operator O */
   upon RECV(BATCH, b, u_k) do
1      locals ← ids(buf_U, buf_out, log)
2      if b.id ∉ acks ∪ hbeats ∪ locals then
3          buf_U ← buf_U ∪ {b}
4          if sink(O) then
5              acks ← acks ∪ {b.id}
   upon SEND(RECOVERY-CTRL) do
6      for U in up(O) do
7          locals ← ids(buf_U, buf_out, log)
8          for u_k ∈ ū do
9              send((acks, hbeats ∪ locals), u_k)
   upon RECV(RECOVERY-CTRL,(acks_j, hbeats_j), d_j) do
10     acks ← acks ∪ acks_j
11     hbeats_j ← hbeats_j \ acks
12     hbeats ← (⋃_{d_x ∈ d̄} hbeats_x) \ acks
13     log ← {b ∈ log : b.id ∉ acks} ∪
       {b ∈ buf_out : b.id ∈ hbeats \ acks}
14     buf_out ← {b ∈ buf_out : b.id ∉ acks ∪ hbeats}
15     for U ∈ up(O) do
16         buf_U ← {b ∈ buf_U : b.id ∉ acks}
   upon DISCONNECT(d_j) do
17     hbeats_j ← ∅
18     hbeats ← (⋃_{d_x ∈ d̄} hbeats_x)
19     for b ∈ log : b.id ∉ hbeats do
20         log ← log \ {b}
21         buf_out ← buf_out ∪ {b}
```

Periodically, when a downstream disconnects, o_i sends to each of its upstream replicas a *recovery control* message with its latest acknowledgements and heartbeats (lines 6–9). For each upstream U, o_i computes the set of batch identifiers stored in its input buffer for U, output buffer and output log (line 7). For each replica u_k of U, o_i then sends a recovery control message with the batch identifiers that are safe to acknowledge (*acks*), and heartbeats for the current set of batches stored either locally at o_i (*locals*) or at one of the transitive downstreams of o_i (*hbeats*) (line 9).

On receiving a recovery control message from a downstream replica d_j (lines 10–14), o_i first adds any newly acknowledged batch identifiers to its record of batches safe to trim (*acks*). o_i then stores the latest heartbeats for d_j (line 11), and recomputes the combined set of batches stored at its downstreams, as indicated by the current heartbeats stored for each downstream replica d_x (line 12).

Next, o_i updates its local buffers and log. It discards any batches from its log, output buffer or input buffers that are safe to trim (lines 13–16). In addition, o_i moves any batches awaiting dispatch in its output buffer that have already been received downstream, as indicated by the updated set of heartbeats, to its output log (e.g. if they were retransmitted via a different replica during a network partition) (lines 13–14). Conversely, it also moves any batches remaining in its output log that are no longer known to be *live* at a downstream to its output buffer, and retransmits them (lines 13–14).

The final part describes o_i's behaviour under disconnection of a downstream d_j (lines 17–21). o_i deletes its heartbeats for d_j (line 17), and recomputes the batch identifiers that are still live at a downstream, and hence need not be retransmitted, from the heartbeats stored for each remaining downstream replica (line 18). Finally, o_i moves any batches in its output log that are no longer known to be live to its output buffer for retransmission (lines 19–21).

5.3 Discussion

Consistency. By default, Frontier provides exactly-once delivery with strong consistency, but batches may be delivered out-of-order.

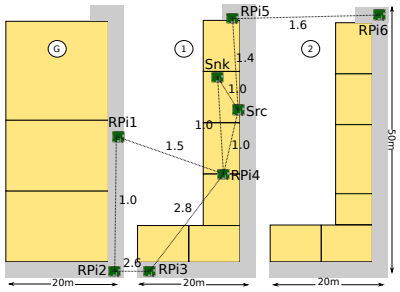
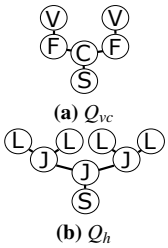


Figure 5: Mesh network deployment with OLSR topology and ETX link weights



(a) Q_{vc} **(b)** Q_h

Figure 6: Evaluation queries

To enforce ordered delivery, sinks may introduce a *reorder buffer* at the cost of extra delay. Sources must then ensure that the total number of unacknowledged batches is less than the size of the reorder buffer. To handle long periods of disconnection between sources and sinks, sources may potentially spill batches to disk, although our prototype implementation does not support this.

Source/sink replication. So far, we have focused on replication of internal operators only, but a failed or partitioned source/sink also affects query execution. Frontier supports their replication, but care must be taken not to affect the correctness of query results.

With replicated sources, the batches produced by different sources with the same identifier are treated interchangeably, even if their contents differ. Thus an acknowledgement for a batch produced by one source replica also acknowledges batches produced with the same identifier by another source replicas. This can be useful for sensing applications that only need to receive a batch from one (or a subset of) several redundant source sensors. Permanent failure of a source may lose batches the source has produced that have not yet been acknowledged at the sink.

Sink replication is suitable when any replica must received a particular batch. For example, in a face recognition application with replicated sink devices belonging to end-users that must verify the recognised face, only one sink must receive each recognised face.

Control message optimisations. Recovery control messages consist conceptually of two sets of batch identifiers, *acks* and *hbeats*. In practice, their network overhead remains low because they contain only batch identifiers and not tuple timestamps. Full acknowledgements and heartbeats are only transmitted on initial connection establishment, with deltas to previous messages otherwise.

Batch identifier sets can also be represented compactly as bit vectors or interval sets with a *low watermark identifier*. The low watermark indicates the greatest identifier for which all lower identifiers can be acknowledged. The *acks* and *hbeats* sets then only contain out-of-order batches with identifiers greater than the watermark.

6 Evaluation

We evaluate Frontier on both real hardware (using Raspberry Pis) and in an emulated wireless network. We explore its perfomance for a variety of queries and network conditions, densities and sizes. Our evaluation answers the following questions: (i) Does Frontier's network-aware processing improve throughput and latency? (ii) Does Frontier improve resilience in the presence of transient failures and mobility? (iii) Do the throughput gains achieved come with an acceptable cost in terms of power consumption?

Our results show that the combination of Frontier's RDGs, backpressure routing, and network disconnection recovery results in a 1.3×–2.8× gain in throughput (with similar or lower latency) depending on the query type and network characteristics.

6.1 Implementation and experimental set-up

The prototype implementation of Frontier is based on SEEP [20], a lightweight Java-based data processing platform, and consists of approximately 25 KLOC. It supports Raspberry Pi, Android and generic Linux devices. If root access is available, the Android version can use multi-hop routing algorithms, such as OLSR [14]; otherwise it uses neighbouring devices over WiFi Direct [61].

Raspberry-Pi deployment. Our evaluation on real hardware uses two mesh network deployments: (i) a network with high path diversity deployed over several floors in an indoor environment (Fig. 5); and (ii) a network with low path diversity in which all nodes are in the same room and directly connected (not shown).

Both networks have 8 nodes including 6 Raspberry Pi 3 Model Bs for hosting intermediate operators, and two Linux machines for hosting source and sink operators, respectively. All nodes are equipped with an 802.11g WiFi adapter (Broadcom 43143), configured in ad-hoc mode. They use OLSR [14] for routing with expected transmission count (ETX) [16] as a cost metric.

Emulation deployment. For our emulator experiments, we use the CORE/EMANE wireless network emulator (version 0.9.2) [1]. For the MAC layer, we use 802.11b ad-hoc mode with a unicast and multicast rate of 11 Mbps, and set the maximum number of MAC layer retries to 7. Unless otherwise stated, we use the default values for all other CORE/EMANE parameters. At the network layer, nodes again use OLSR for routing with ETX.

6.2 Evaluation queries

(1) Distributed face recognition query Q_{fr}. This query implements the distributed face recognition application from Fig. 1, implemented using the OpenCV [45] library.

(2) Video correlation query Q_{vc}. This query correlates faces across multiple video streams in order to reduce false positives (see Fig. 6). Two sources capture frames from video feeds (V) of the same scene from two different cameras. Each source forwards a tuple for each frame to a separate face detection operator (F) that analyses each frame and sends detected faces to a shared face correlation operator (C). It implements a custom *join operator* that recognises faces on each input stream and outputs a match to the sink (S) only if the same face is recognised in both.

(3) Heatmap query Q_h. This query realises a real-time heatmap application (see Fig. 6) that aggregates the movements of mobile users in an area (e.g. members of a sports team [48, 38] or emergency workers in a disaster response scenario). Each query source produces a periodic location report (L) of the time spent in each section of a predefined grid over the area. User reports are aggregated using a distributed tree of binary join operators (J): leaf join operators compute the aggregate time spent by users in each section of the grid; intermediate join operators aggregate the results, until the root operator computes the aggregate time spent by all users in each section and outputs it to the sink (S).

6.3 Throughput and latency

Our first experiments evaluate the benefit of BSR's network-aware processing in terms of query throughput and latency. To model a realistic network deployment, we use packet loss measurements and node locations from the RoofNet trace [11]. For each link in the mesh network, we use the reported average packet loss to precompute the expected pathloss for that link, which EMANE incorporates into its statistical model for 802.11 packet delivery.

Throughput. Fig. 7 shows the throughput for our queries for different replication factors r (error bars indicate the standard error over several runs with random operator placements).

For the face recognition query Q_{fr}, replication increases throughput from 1.6× ($r = 2$) to 2.2× ($r = 4$). Due to wireless interference and switching overhead, increasing r further shows no benefit. For the video correlation query Q_{vc}, the maximum speed-up of 1.7× ($r = 3$) is lower than for Q_{fr} due to the extra coordination overhead with the multi-input correlation operator. The heatmap query Q_h shows a peak gain of 1.3× ($r = 2$). In comparison to Q_{vc}, this query contains two additional join operators—further increasing r yields no benefit due to the coordination overhead. Empirically therefore, the experiments show that the throughput is not usually highly sensitive in the choice of r as long as it is chosen high enough, especially for queries with no or few joins.

For Q_{fr}, we also compare the performance of BSR to several other routing strategies: (i) broadcast (BC) sends a copy of each batch to every replica, similar to the proposal by Hwang et al. [27]; (ii) round-robin (RR) sends an equal number of batches to each replica in round-robin order; (iii) weighted round-robin (WRR) is network-aware and sends batches in proportion to the inverse of the current ETX to each replica; and (iv) power-of-two-choices (P2C) chooses two replicas at random and sends the current batch to the least loaded (as indicated by their current queue length).

As Fig. 8 shows, BSR achieves the best throughput for all replication factors r. The closest performing approach is WRR, over which BSR gives a speedup of 1.1× for $r = 3$. While WRR is also network-aware, we hypothesise that it is more susceptible to interference: it routes at finer granularity and is thus more likely to send to all replicas at once than BSR, which prioritises the current highest weight replica and switches less frequently. This affect is even worse for smaller batch sizes, as we show in Fig. 9 where BSR gives a speedup of 1.64× over WRR for 0.5 KB batches ($r = 4$). In addition BSR weights implicitly include information about each downstream's transitive connectivity and load; WRR relies on the network-layer routing algorithm's ETX to the downstream only. Note that WRR is also not applicable to queries with multi-input operators. The remaining strategies in Fig. 8 are not network-aware, and thus achieve lower throughput. BC has the worst throughput because it sends redundant data to all replicas, causing further interference.

To validate the results of our Roofnet experiments, we measure the throughput gains for the face recognition query on our Raspberry Pi testbed. Fig. 10a shows how the throughput changes with an increasing replication factor for networks with *high* and *low* path diversity. Similar to the Roofnet results, replication gives a speed-up in mean throughput of 1.6× for $r = 2$, and of 2.8× for $r = 3$ for the sparse network with high path diversity. As expected similar gains are achieved for the dense network with low path diversity, with the absolute throughput higher due to the higher quality network links. We are unable to evaluate higher replication factors due to the limited size of our testbed.

In addition to an unreplicated baseline, we also explore the benefits of BSR compared to the Apache Flink stream processing system [21] (which uses backpressure for flow-control only) on our Raspberry Pi mesh network. Fig. 10b shows the throughput of BSR, round-robin (RR), and Flink for Q_{fr} with $r = 3$. BSR gives a 2.8× speed-up over RR with high path diversity: it can more effectively prioritise replicas with better network connectivity. This throughput gain does not come at the cost of worse performance than RR with low path diversity, where the throughput is similar for both. In comparison to Flink, BSR gives a 2.3× speed-up with high path diversity and similar throughput with low path diversity. With high path diversity, Flink cannot run to completion because it fails due to transient network failures. The Flink results in Fig. 10b with high path diversity therefore give the average throughput up to the point of failure.

The fact that Flink fails during execution shows the importance of transient network failure recovery (**R4**). The Flink worker relies on a master node to coordinate recovery. In some cases, our experiment terminates because the master assumes that the worker itself has failed, and tries to restart it on a new node. Since there are no

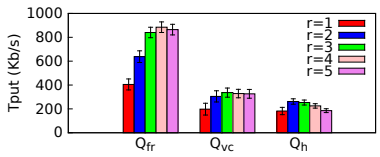


Figure 7: Throughput with varying replication factor r (RoofNet)

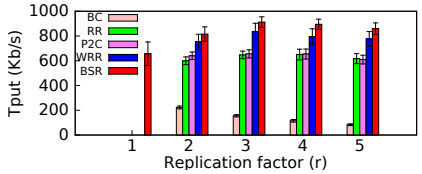


Figure 8: Throughput of BSR compared to baselines (RoofNet)

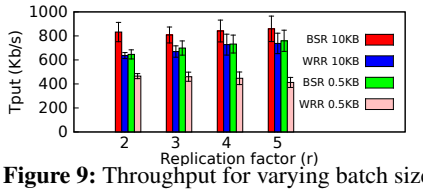
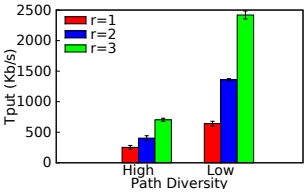
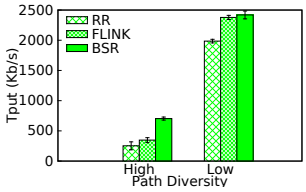


Figure 9: Throughput for varying batch size and replication factor r (Roofnet)



(a) BSR with different r



(b) BSR ($r = 3$) vs. round-robin/Flink

Figure 10: Throughput for Q_{fr} on Raspberry Pis with high/low network path diversity

additional nodes, the master terminates the experiment instead of re-establishing the failed connection or continuing with the remaining workers; in other cases, even though the network connection between two workers fails, the worker connections to the master remain up. Eventually the system stalls because the upstream worker blocks trying to send to the disconnected downstream workers.

We also evaluate the impact on throughput of executing multiple queries, e.g. when there are multiple independent video sources. Fig. 11 shows how throughput for the Roofnet network changes as we increase the number of rate-limited face recognition query sources. Peak throughput is reached for $r = 1$ at 5 sources when the face detector reaches full CPU utilisation. Throughput plateaus between 7–9 queries when the network becomes saturated, with a maximum throughput gain for 9 queries of 1.7× ($r = 3$).

Latency. Fig. 13 shows the latency achieved by each query for the Roofnet experiments. For Q_{fr} with $r = 3$, replication reduces the 95$^{\text{th}}$ percentile latency by up to 17%. For higher replication factors ($r \geqslant 4$), latency starts to grow again as the system becomes saturated, with a 16% increase for $r = 5$. The median latency remains lower than for the unreplicated query though.

For Q_{vc}, 95$^{\text{th}}$ percentile latency is lowest for $r = 2$, which gives a 65% reduction. Similar to Q_{fr}, latency grows again for higher replication factors. The 95$^{\text{th}}$ percentile, however, remains lower than for the unreplicated query for both $r = 3$ and $r = 4$, with a 25% and 20% reduction, respectively.

For Q_h, the absolute latency is higher than for the other queries because finding a replica with good connectivity to multiple sources is harder on average. Nonetheless, replication ($r = 3$) results in an up to 32% reduction in the 95$^{\text{th}}$ percentile latency.

We also compare the latency of BSR to other baselines for Q_{fr} (Fig. 14). BSR and WRR have lower latencies than the remaining strategies, which are not network-aware: BSR and WRR have similar median and 95$^{\text{th}}$ percentile latencies for $r = 3$ and $r = 4$.

Summary. Network-aware stream routing strategies outperform existing approaches on both real and emulated mesh networks, especially with high path diversity, with BSR giving the highest throughput. The benefit for a particular query depends on the number of joins due to the additional coordination overhead.

6.4 Resilience to network dynamicity

Next we explore the benefits of Frontier with increasing network dynamicity due to transient failures. To model transient failures, we divide each experiment run into fixed size slots of 10 seconds and, for each node, disable the network interface in a slot according to a failure probability p_{fail}. To reduce noise, we use a fixed operator placement that exhibits average throughput across all generated placements from the ones in Fig. 7.

Throughput robustness. Fig. 12 shows the throughput for our queries on the Roofnet network as we increase the failure probability. As expected, absolute throughput falls for each replication factor but falls less for higher factors. For Q_{fr}, the speed-up for $r = 4$ increases from 1.2× with a failure rate $p_{fail} = 0.05$ to 6.8× for $p_{fail} = 0.2$, even overtaking $r = 3$ as the best performing replication factor at $p_{fail} = 0.15$. The query Q_{vc} with $r = 4$ gives the best throughput for all failure rates, with $r = 1$ giving a negligible throughput for $p_{fail} \geqslant 0.15$. For the heatmap query Q_h with $r = 3$, we observe the best throughput for all failure rates with $r = 1$, giving no throughput for $p_{fail} \geqslant 0.15$. Thus with higher dynamicity a higher replication factor is justified due to the robustness benefits.

Recovery overhead. We analyse the behaviour of Frontier's disconnection recovery mechanisms, in particular the impact of selective replay. We deploy Q_{fr} with $r = 3$ on a network with 18 nodes arranged in a 3×6 grid. We inject a transient failure at a face detector replica and measure the impact of selective replay on recovery.

Figs. 15 and 16 show the throughput and recovery latency for three set-ups: network bandwidths are adjusted such that the query is network bottlenecked (i) at the source; (ii) at the sink; and (iii) at the sink before recovery and at the remaining replicas of the failed node during recovery.

For the source bottleneck, selective replay has little impact on either recovery time or query throughput because there are few tuples in-flight downstream of the failed node. In contrast, a sink bottleneck before the failure results in many in-flight tuples downstream of the failed node. This leads to a 2.8× increase in redundant retransmissions during recovery in the absence of selective replay, and thus a 1.6× increase in the recovery latency. The impact of selective replay on throughput is modest (12%) because tuples queued downstream of the failed replica are still bottlenecked at the sink. When the query bottleneck shifts from the sink to the replicas of the failed node (e.g. because they have poor connectivity), selective replay increase recovery throughput by 33%.

Summary. The throughput gain of BSR over an unreplicated query increases with network dynamicity due to BSR's ability to exploit network path diversity. The impact of selective replay on failure recovery depends on the number of in-flight tuples and whether the query bottleneck masks the effect of redundant retransmissions.

6.5 Resilience to network mobility

To further illustrate the robustness of Frontier in the face of network dynamicity, our next experiments evaluate performance with node mobility. We emulate networks with node mobility in CORE/EMANE and use BonnMotion [7] to generate mobility traces according to a *steady-state random waypoint* mobility model with a pause time of 2 seconds [40, 41]. At the physical layer, we set the node transmit power to -10.0 dBm, giving an approximate transmission range of 500 m with stable TCP throughput.

Node speed. Fig. 17 shows the throughput for our queries for different replication factors r, as we increase node speed.

For Q_{fr}, the benefits of replication for throughput become apparent as node speed increases, even though the absolute throughput

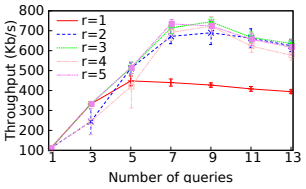
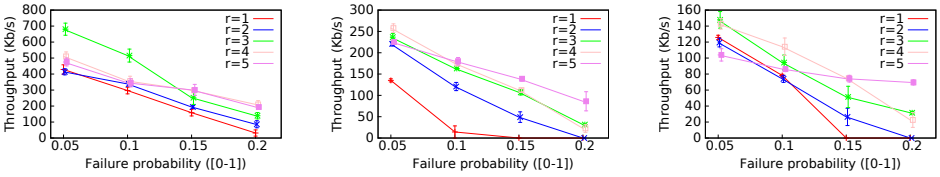


Figure 11: Multi-query scalability



(a) Q_{fr} **(b)** Q_{vc} **(c)** Q_h

Figure 12: Throughput vs. node failures for replication r

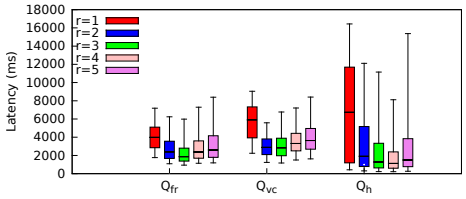


Figure 13: Latency with varying replication factor r (Roofnet) (Datapoints show 5/25/50/75/95$^{\text{th}}$ percentiles.)

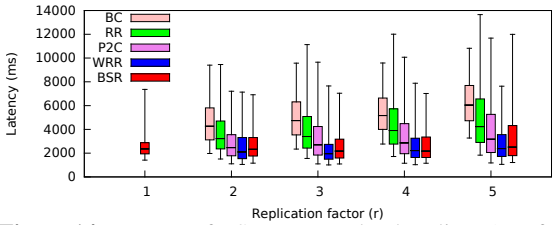


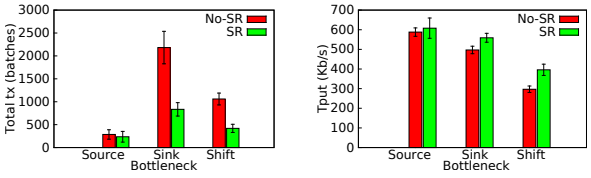
Figure 14: Latency of BSR compared to baselines (Roofnet)

falls (Fig. 17a). With an average speed of 5 m/s, there is a 1.6× and 1.8× improvement in mean throughput for $r = 2$ and $r = 3$, respectively. At 10 m/s, the improvement increases to 2.1× ($r = 2$) and 2.6× ($r = 3$). Even with low mobility (1 m/s), $r = 2$ gives a mean speed-up of 1.3×, and $r = 3$ results in 1.4×. Variance is slightly higher with low mobility because initial operator and node placements have more influence on throughput. Even though variance is higher, relative throughput increases with larger replication factors.

For Q_{vc}, replication improves performance with low mobility, with a 1.9× speed-up for $r = 3$ at node speeds of 1 m/s. With greater mobility, throughput converges for r due to the increased coordination overhead of switching between replicas for the correlation operator. In contrast, for the network dynamicity experiment of Fig. 12b, relative throughput increases with network dynamicity for Q_{vc}: with higher failure probabilities, the network effectively becomes sparser, increasing network path diversity; with greater mobility, available path diversity remains the same on average but changes more frequently.

For Q_h, replication continues to give a benefit at higher node speeds compared to Q_{vc}, with a 1.2× speed-up for $r = 3$ at 2.5 m/s, before eventually the throughput converges at 3.0 m/s. Although Q_h has more join operators than Q_{vc}, the smaller tuple size for heatmap location reports in comparison to the images of Q_{vc} allows faster switching between replicas.

Network density. Next we explore for Q_{fr} how changing the size of the emulation area and hence the network density affects throughput for different r. Fig. 18a shows the throughput for different replication factors in a network with 25 nodes and mobility of 5 m/s as the dimensions of the emulation area increase from 900 m to 1800 m. Replication gives approximately a 2× speed-up for a small area, increasing to approximately 4× for an area of 1800 m ×1800 m. The reduction in network density therefore has a higher impact on the unreplicated query. $r = 3$ gives a small increase in throughput over $r = 2$ and has lower variance for larger areas.



(a) Retransmissions during recovery **(b)** Recovery throughput

Figure 15: Recovery throughput for different bottlenecks

Network size. We also investigate how more nodes affect throughput when mobility and the size of the emulation area are fixed. Fig. 18b shows the throughput for different r with a mobility of 5 m/s in a 1200 m ×1200 m area as we increase the number of nodes from 20 to 70. Overall, increasing the number of nodes has a negligible impact on throughput, independent of replication.

Summary. BSR achieves high throughput with mobility, but as speed increases the benefit diminishes more quickly for queries with joins due to the coordination overhead. For fixed mobility, BSR still gives throughput gains with varied network size.

6.6 Power efficiency

To evaluate Frontier's power usage, we use a digital multimeter [49] to measure the power usage of a Raspberry Pi, similar to prior work [32]. When plugged into a wall socket, we measure an idle power usage of 1.3 W. Measuring power usage under increased CPU load reveals an approximately linear relationship, with a maximum power draw of 2.2 W at full utilisation of all four cores (Fig. 19a). For network power usage, we use `iperf` to send UDP traffic at different data rates between two devices connected directly by an 802.11g network link. As Fig. 19b shows, the results are broadly in line with previous measurements [32]: the maximum power usage of network communication is 2.1 W for transmission and 1.6 W for reception, both lower than CPU power usage at full load.

We repeat the experiment from Fig. 10a while recording average CPU utilisation and bytes transmitted/received. Using our power model, we compute the throughput per watt across all nodes (Fig. 20a). Since with lower replication nodes not hosting operators can be powered off, we also consider only active nodes (Fig. 20b).

For the deployment with high path diversity, power usage is dominated by the idle usage of each node, and so the higher throughput of replicated processing leads to gains of up to 2.3× for $r = 3$ network (Fig. 20a). For the low path diversity network, power usage is higher due to the higher data rates and CPU utilisation, but this is offset by the increase in throughput, resulting in a throughput per watt gain of 3.4× for $r = 3$. Furthermore, even when energy usage of non-operator hosting nodes is excluded (Fig. 20b), Frontier achieves throughput per watt gains for $r = 3$ of 1.2× and 1.8× for the high and low path diversity networks, respectively.

Summary. For different Raspberry Pi deployments, BSR achieves comparable or greater throughput per watt than an unreplicated baseline, showing that Frontier efficiently uses energy resources.

7 Related Work

Adaptative stream processing. A plethora of work exists on query planning for stream processing. SODA [62] and SQPR [31] formulate query plans and placement as optimisation problems. We can exploit these approaches to generate an initial operator placement for an RDG, or reconsider placements over long time scales.

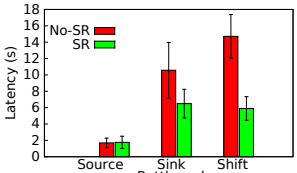
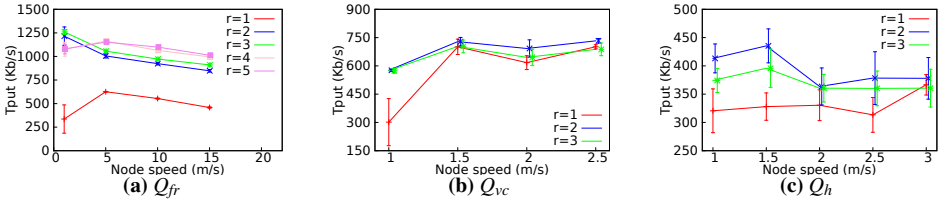
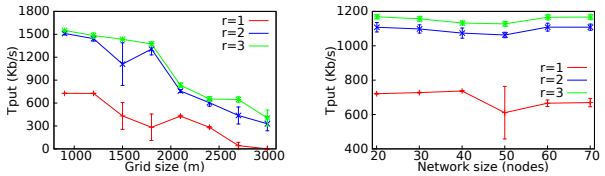


Figure 16: Recovery latency



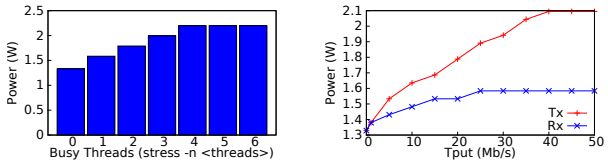
(a) Q_{fr} **(b)** Q_{vc} **(c)** Q_h

Figure 17: Throughput vs. mobility for replication r



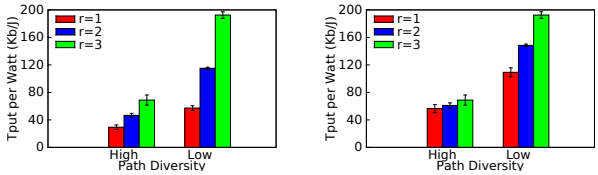
(a) Increasing network area **(b)** Increasing nodes

Figure 18: Throughput vs. network size with mobility 5 m/s



(a) CPU **(b)** Network

Figure 19: Power consumption on Raspberry Pi 3 Model B



(a) Including all nodes **(b)** Operator nodes only

Figure 20: Performance per watt

In the context of wide-area networks, SBON [46] uses a decentralised algorithm to adapt operator placement in a stream query graph to network conditions. In contrast, Frontier creates several replicas of each operator ahead of time and uses BSR to load-balance among them based on network conditions. Jetstream [51] degrades data quality based on network bandwidth between clusters in different data centers. We view data degradation as orthogonal to Frontier.

Techniques to adapt the execution of stream queries to changes in the input data exist: Eddies [8] change the operator order while ensuring correct query results; QueryMesh [42] selects one of multiple pre-computed query plans; JISC [4] reduces the overhead of plan adaptation by lazily recomputing intermediate operator states in the new plan. These approaches execute on a single server or static network environment and thus do not apply to dynamic wireless networks. They also focus on relational queries while we consider queries with UDFs. The adaptive load diffusion techniques of Gu et al. [24] dynamically modify the destination to which data is sent based on network load. Their solution, however, requires a centralised diffusion operator and is not fault tolerant.

Elseidy et al. propose an online algorithm for join operators with large historical state [18]. Their focus though is on data skew, and the proposed mechanism requires expensive state migration between nodes. Nasir et al. propose a distributed algorithm using the power-of-two choices to choose between multiple replicas [39]. It handles key skew for associative operators and requires a separate aggregation step to combine results from different replicas. Query scrambling modifies the scheduling or structure of a query plan based on dynamic access costs during plan execution [5]. While we also adapt a physical query plan, our work addresses a more specific problem: how to route stream data over a replicated dataflow graph in a decentralized manner based on network and processing dynamics. SquirrelJoin [54] uses lazy partitioning to cope with transient network skew, but only supports partitioned joins and needs a central coordinator; Frontier exploits out-of-order batch processing for streaming joins in a distributed fashion.

Stream processing platforms. Existing stream processing systems incorporate backpressure [57, 34, 21] techniques for the purpose of *flow control*. In contrast, Frontier's BSR algorithm is based on throughput-optimal backpressure network routing [58, 35]. Beyond flow control, it dynamically prioritises between replicas in a network-aware manner using explicit queue differentials computed from queue lengths, processing capabilities, and network rates.

Millwheel [2] uses tuple watermarks to support out-of-order processing within a window. Processing early tuples out-of-order on a replica is less beneficial when connectivity to that replica worsens before processing of the window completes. Frontier instead reorders tuples within a batch at query sources and processes out-of-order at the coarser granularity of batches. MobiStreams [59] and Mobile Storm [44] target mobile ad-hoc networks (MANETs), but require cellular connectivity to a backend for control purposes. Frontier avoids dependencies on backend connectivity, which may be unavailable or congested in edge IoT deployments.

Failure recovery. Frontier's disconnection recovery is most related to upstream backup [26]. In contrast to upstream backup, SNAP broadcasts heartbeats to all upstream replicas allowing the *transitive* upstreams of a failed node to compute in a decentralised manner which batches should be retransmitted by which replicas, even with concurrent failures. Other approaches to failure recovery can be divided into replication [26, 55, 10] and checkpoint based [26, 28, 60], but their focus is on single-node crash-stop failures. Frontier could be augmented with a checkpointing mechanism to reduce recovery overhead for stateful operators.

8 Conclusions

We have presented Frontier, an edge-based IoT stream processing system that provides high throughput using the combined resources of multiple IoT devices connected through an unreliable wireless mesh network. It accounts for the dynamic nature of wireless networks by exploiting *network path diversity*: queries are transformed into replicated dataflow graphs with multiple *operator replicas*. Frontier sends to replicas according to a *backpressure stream routing* algorithm that adapts to changing network conditions, while still using replicas in a data-parallel fashion. Frontier provides strong fault tolerance guarantees: it uses *selective network-aware replay* to efficiently recover from transient network failures. Evaluation on a wireless mesh network and in emulation shows that Frontier achieves speed-ups of between 1.3×–2.8× for a range of queries.

Acknowledgements: This research was sponsored by the U.S. Army Research Laboratory and the U.K. Ministry of Defence under Agreement Numbers W911NF-06-3-0001, W911NF-16-3-0001. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Army Research Laboratory, the U.S. Government, the U.K. Ministry of Defence or the U.K. Government. The U.S. and U.K. Governments are authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation hereon.

9 References

[1] J. Ahrenholz. Comparison of CORE network emulation platforms. In *MILCOM*, pages 166–171, 2010.

[2] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. Millwheel: Fault-tolerant stream processing at internet scale. *PVLDB*, 6(11):1033–1044, 2013.

[3] A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, F. Naumann, M. Peters, A. Rheinländer, M. J. Sax, S. Schelter, M. Höger, K. Tzoumas, and D. Warneke. The Stratosphere platform for big data analytics. *PVLDB*, 23(6):939–964, 2014.

[4] A. M. Aly, W. G. Aref, M. Ouzzani, and H. M. Mahmoud. JISC: adaptive stream processing using just-in-time state completion. In *17th International Conference on Extending Database Technology (EDBT).*, pages 73–84, 2014.

[5] L. Amsaleg, M. J. Franklin, A. Tomasic, and T. Urhan. Scrambling query plans to cope with unexpected delays. In *4th International Conference on on Parallel and Distributed Information Systems (DIS)*, pages 208–219, 1996.

[6] A. Arasu, S. Babu, and J. Widom. The CQL continuous query language: Semantic foundations and query execution. *PVLDB*, 15(2):121–142, 2006.

[7] N. Aschenbruck, R. Ernst, E. Gerhards-Padilla, and M. Schwamborn. BonnMotion: A mobility scenario generation and analysis tool. In *3rd International ICST Conference on Simulation Tools and Techniques (SIMUTools)*, pages 51:1–51:10, 2010.

[8] R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. *SIGMOD Rec.*, 29(2):261–272, May 2000.

[9] Microsoft Azure IoT Edge. `https://azure.microsoft.com/en-us/services/iot-edge`, 2017.

[10] M. Balazinska, H. Balakrishnan, S. Madden, and M. Stonebraker. Fault-tolerance in the Borealis distributed stream processing system. In *SIGMOD*, pages 13–24, 2005.

[11] J. Bicket, D. Aguayo, S. Biswas, and R. Morris. Architecture and evaluation of an unplanned 802.11b mesh network. In *MOBICOM*, pages 31–42, 2005.

[12] Bluetooth.com. Bluetooth technology adding mesh networking to spur new wave of innovation. `https://www.bluetooth.com/news/pressreleases/2015/02/24/bluetoothtechnology-adding-mesh-networking-to-spur-new-wave-of-innovation`.

[13] B. J. Bonfils and P. Bonnet. Adaptive and decentralized operator placement for in-network query processing. *Telecommun. Syst.*, 26(2-4):389–409, June 2004.

[14] T. Clausen and P. Jacquet. Optimized link state routing protocol (OLSR). RFC 3626, RFC Editor, October 2003. `http://www.rfc-editor.org/rfc/rfc3626.txt`.

[15] Connode Smart Metering. `http://www.connode.com/solutions`, 2016.

[16] D. S. J. De Couto, D. Aguayo, J. Bicket, and R. Morris. A high-throughput path metric for multi-hop wireless routing. *Wirel. Netw.*, 11(4):419–434, July 2005.

[17] Dust Networks Applications: Industrial Automation. `https://www.linear.com/designtools/wireless_sensor_apps.php#industrial`, 2017.

[18] M. Elseidy, A. Elguindy, A. Vitorovic, and C. Koch. Scalable and adaptive online joins. *PVLDB*, 7(6):441–452, 2014.

[19] Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation). *Official Journal of the European Union*, L119:1–88, May 2016.

[20] R. C. Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch. Integrating scale out and fault tolerance in stream processing using operator state management. In *SIGMOD*, pages 725–736, 2013.

[21] Apache flink. `https://flink.apache.org`.

[22] Google Nest Cam. `https://nest.com/cameras`, 2017.

[23] Amazon AWS Greengrass. `https://aws.amazon.com/greengrass`, 2017.

[24] X. Gu, P. S. Yu, and H. Wang. Adaptive load diffusion for multiway windowed stream joins. In *ICDE*, pages 146–155, 2007.

[25] T. Guardian. Revolv devices bricked as Google's Nest shuts down smart home company. `https://www.theguardian.com/technology/2016/apr/05/revolv-devices-bricked-google-nest-smart-home`.

[26] J.-H. Hwang, M. Balazinska, A. Rasin, U. Cetintemel, M. Stonebraker, and S. Zdonik. High-availability algorithms for distributed stream processing. In *ICDE*, pages 779–790, 2005.

[27] J.-H. Hwang, U. Çetintemel, and S. Zdonik. Fast and reliable stream processing over wide area networks. In *ICDEW '07*, pages 604–613, 2007.

[28] J. H. Hwang, Y. Xing, U. Çetintemel, and S. Zdonik. A cooperative, self-configuring high-availability solution for stream processing. In *ICDE*, pages 176–185, 2007.

[29] IDC. Data Age 2025: The evolution of data to life-critical. `https://www.seagate.com/files/www-content/our-story/trends/files/Seagate-WP-DataAge2025-March-2017.pdf`, 2017.

[30] IEEE Standard for Information technology–Telecommunications and information exchange between systems Local and metropolitan area networks–Specific requirements - Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications. *IEEE Std 802.11-2016 (Revision of IEEE Std 802.11-2012)*, pages 1–3534, Dec 2016.

[31] E. Kalyvianaki, W. Wiesemann, Q. H. Vu, D. Kuhn, and P. Pietzuch. SQPR: Stream query planning with reuse. In *ICDE*, pages 840–851, 2011.

[32] F. Kaup, P. Gottschling, and D. Hausheer. PowerPi: Measuring and modeling the power consumption of the Raspberry Pi. In *39th Annual Conference on Local Computer Networks (LCN)*, pages 236–243, Sept 2014.

[33] S. Khurana, A. Kahol, and A. P. Jayasumana. Effect of hidden terminals on the performance of IEEE 802.11 MAC protocol. In *23rd Annual Conference on Local Computer Networks (LCN)*, pages 12–20, Oct 1998.

[34] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja. Twitter heron: Stream processing at scale. In *SIGMOD*, pages 239–250, 2015.

[35] R. Laufer, T. Salonidis, H. Lundgren, and P. Le Guyadec. A cross-layer backpressure architecture for wireless multihop networks. *IEEE/ACM Trans. Netw.*, 22(2):363–376, Apr. 2014.

[36] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker. No pane, no gain: Efficient evaluation of sliding-window aggregates over data streams. *SIGMOD Rec.*, 34(1):39–44, Mar. 2005.

[37] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong.

Tinydb: An acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.*, 30(1):122–173, Mar. 2005.

[38] C. Mutschler, H. Ziekow, and Z. Jerzak. The DEBS 2013 Grand Challenge. In *DEBS*, pages 289–294, 2013.

[39] M. A. U. Nasir, G. D. F. Morales, D. Garca-Soriano, N. Kourtellis, and M. Serafini. The power of both choices: Practical load balancing for distributed stream processing engines. In *ICDE*, pages 137–148, 2015.

[40] W. Navidi and T. Camp. Stationary distributions for the random waypoint mobility model. *IEEE Transactions on Mobile Computing*, 3(1):99–108, Jan. 2004.

[41] W. Navidi, T. Camp, and N. Bauer. Improving the accuracy of random waypoint simulations through steady-state initialization. *15th International Conference on Modeling and Simulation*, pages 319–326, 2004.

[42] R. Nehme, K. Works, C. Lei, E. Rundensteiner, and E. Bertino. Multi-route query processing and optimization. *Journal of Computer and System Sciences*, 2013.

[43] Netatmo. `https://www.netatmo.com`, 2017.

[44] Q. Ning, C. A. Chen, R. Stoleru, and C. Chen. Mobile storm: Distributed real-time stream processing for mobile clouds. In *2015 IEEE 4th International Conference on Cloud Networking (CloudNet)*, pages 139–145, Oct 2015.

[45] OpenCV. Open source computer vision library. `https://github.com/opencv/opencv`, 2015.

[46] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer. Network-aware operator placement for stream-processing systems. In *ICDE*, page 49, 2006.

[47] Piper: All-in-one wireless security system. `https://getpiper.com/howitworks`, 2016.

[48] Polar Team Pro GPS player tracking. `https://www.polar.com/uk-en/b2b_products/team_sports/team_pro`, 2017.

[49] PortaPow Dual USB Power Monitor V3. `https://www.portablepowersupplies.co.uk/portapow-dual-usb-power-monitor-v3/`.

[50] ProVigial Target Tracking and Analysis. `https://pro-vigil.com/features/video-analytics/object-tracking`, 2017.

[51] A. Rabkin, M. Arye, S. Sen, V. S. Pai, and M. J. Freedman. Aggregation and degradation in Jetstream: Streaming analytics in the wide area. In *NSDI*, pages 275–288, 2014.

[52] Revolv Inc. `http://revolv.com/`.

[53] J. P. Rohrer, A. Jabbar, and J. P. G. Sterbenz. Path diversification: A multipath resilience mechanism. In *7th International Workshop on Design of Reliable Communication Networks*, pages 343–351, Oct 2009.

[54] L. Rupprecht, W. Culhane, and P. Pietzuch. SquirrelJoin: Network-aware distributed join processing with lazy partitioning. *PVLDB*, 10(11):1250–1261, 2017.

[55] M. A. Shah, J. M. Hellerstein, and E. Brewer. Highly available, fault-tolerant, parallel dataflows. In *SIGMOD*, pages 827–838, 2004.

[56] Z. Shen, V. Kumaran, M. J. Franklin, S. Krishnamurthy, A. Bhat, M. Kumar, R. Lerche, and K. Macpherson. CSA: Streaming engine for internet of things. *IEEE Data Eng. Bull.*, 38(4):39–50, 2015.

[57] Storm. `http://github.com/apache/storm/`.

[58] L. Tassiulas and A. Ephremides. Stability properties of constrained queueing systems and scheduling policies for maximum throughput in multihop radio networks. *IEEE Transactions on Automatic Control*, 37(12):1936–1948, Dec 1992.

[59] H. Wang and L.-S. Peh. MobiStreams: A reliable distributed stream processing system for mobile devices. In *IPDPS*, pages 51–60, 2014.

[60] H. Wang, L. S. Peh, E. Koukoumidis, S. Tao, and M. C. Chan. Meteor Shower: A reliable stream processing system for commodity data centers. In *IPDPS*, pages 1180–1191, 2012.

[61] Wi-Fi Direct. `http://www.wi-fi.org/discover-wi-fi/wi-fi-direct`, 2016.

[62] J. Wolf, N. Bansal, K. Hildrum, S. Parekh, et al. SODA: An optimizing scheduler for large-scale stream-based distributed computer systems. In *Middleware*, pages 306–325, 2008.

[63] M. Wolf. With Mesh, Bluetooth strengthens case as key Internet of Things technology. `http://www.forbes.com/sites/michaelwolf/2015/02/26/with-mesh-bluetooth-strengthens-case-as-key-internet-of-things-technology`, 2015.

[64] Y. Yao and J. Gehrke. The Cougar approach to in-network query processing in sensor networks. *SIGMOD Rec.*, 31(3):9–18, Sept. 2002.

[65] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, page 2, 2012.

[66] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *SOSP*, pages 423–438, 2013.

[67] The Zigbee Alliance. `www.zigbee.com`, 2016.