Incremental Stream Query Deployment under Continuous Infrastructure Changes in the Cloud-Edge Continuum

Ankit Chaudhary
BIFOLD, TU Berlin
ankit.chaudhary@tu-berlin.de

Felix Lang
TU Berlin
f.lang@tu-berlin.de

Danila Ferents
TU Berlin
danila.ferents@campus.tu-berlin.de

Nils L. Schubert
BIFOLD, TU Berlin
nils.schubert@dima.tu-berlin.de

Varun Pandey
Technische Universität Nürnberg
varun.pandey@utn.de

Jeyhun Karimov
Microsoft Corporation
jkarimov@microsoft.com

Steffen Zeuch
BIFOLD, TU Berlin
steffen.zeuch@tu-berlin.de

Kaustubh Beedkar
Indian Institute of Technology Delhi
kbeedkar@cse.iitd.ac.in

Volker Markl
BIFOLD, TU Berlin, DFKI
volker.markl@tu-berlin.de

ABSTRACT

Distributed data stream processing engines (DSPEs) operating over the cloud-edge continuum must deploy data processing operators across a distributed infrastructure. However, the volatile nature of these infrastructure nodes—where devices frequently join, leave, or move—can invalidate existing query operator-to-topology node mappings, leading to interruptions in query execution and potential data loss. To ensure continuous processing while maintaining correctness, DSPEs must dynamically adapt these mappings and redeploy (part of) affected queries.

In this paper, we introduce incremental stream query deployment (`ISQD`), a framework that efficiently redeploys queries affected by topology changes. ISQD employs a greedy strategy to identify and redeploy only affected operators. It uses ad-hoc queries to migrate operator state seamlessly, and leverages reconfiguration markers to synchronize the redeployment process. Our evaluation shows that `ISQD` achieves up to 7.5× lower deployment latency and up to 39× lower event time latency compared to state-of-the-art approaches, even under high-frequency topology changes.

PVLDB Reference Format:
Ankit Chaudhary, Felix Lang, Danila Ferents, Nils L. Schubert, Varun Pandey, Jeyhun Karimov, Steffen Zeuch, Kaustubh Beedkar, and Volker Markl. Incremental Stream Query Deployment under Continuous Infrastructure Changes in the Cloud-Edge Continuum. PVLDB, 19(2): 210-223, 2025.
doi:10.14778/3773749.3773759

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://anonymous.4open.science/r/nebulastream-isqd/ and https://github.com/nebulastream/topology-change-generator.

1 INTRODUCTION

Massively distributed applications, such as smart mobility, fleet management, predictive maintenance, video surveillance, or connected cars [1, 2, 4, 17, 23, 25], increasingly demand near real-time analytics (i.e., latency < 100ms). These applications consume data streams from thousands of geo-distributed and mobile devices that are located outside cloud data centers [14, 46].

State-of-the-art (SOTA) approaches commonly rely on cloud-centric stream processing pipelines. These involve transferring data to the cloud, (generally) persisting it in queues (e.g., Apache Kafka or blob stores), and subsequently consuming it via stream processing engines for analytics. For example, recent studies show that user-to-cloud (`ping`) latency can range from 20 to 250 ms [13]. In addition, Apache Kafka, despite its popularity, has frequently been identified as a performance bottleneck during data retrieval [24, 26, 32, 53], while blob storage access latency can range from 50 to 800 ms [16]. Moreover, a recent work [9, 11, 53] has shown that cloud-centric DSPEs do not effectively utilize the available sources. These limitations result in increased end-to-end latency, network congestion, high data transfer cost, and increased energy consumption, making cloud-based streaming pipelines inadequate.

To mitigate these drawbacks, a new class of DSPEs [34, 36, 40, 52] was designed to push analytical computations (filter, aggregation, join, etc) on the edge and end devices in a single unified system. Combining expandable cloud with existing edge resources offers advantages over traditional cloud-only processing, such as reduced processing latency [51], efficient network and compute utilization [52], and reduced energy consumption [47].

DSPEs utilize *operator placement* as a crucial step during query optimization [6]. Operator placement maps query operators to infrastructure nodes [5–7, 10]. The mapped operators are subsequently deployed in a serial and holistic manner using the deployment mechanism of the underlying DSPE [8]. However, in a dynamic infrastructure such as sensor-edge-cloud, DSPEs face challenges to ensure valid deployments during the lifetime of query execution.

DPSEs must consider the *volatility in the sensor-edge-cloud infrastructure* [36, 52, 54] as a result of frequent connection/disconnection, failure, or relocation of the devices, unlike robust cloud-only infrastructure [34]. For example, IoT/sensor devices (e.g., trains, cars) use cellular modems to connect to the nearest edge data centers, depending on their proximity and signal strength [41]. The mobility of these devices can also cause them to approach another edge data center, thereby changing the optimal edge data center location to the new

Proceedings of the VLDB Endowment, Vol. 19, No. 2 ISSN 2150-8097.
doi:10.14778/3773749.3773759

one [12], resulting in topology changes. These topology changes result in the interruption or failure of running queries and the invalidation of existing operator placements and deployments [8]. To tackle this, DSPEs must identify affected queries, perform replacement, redeployment, and reconfiguration within a reasonable time while preventing tuple or state loss.

Several works have addressed dynamic workload redeployment and reconfiguration in response to sudden stream changes (e.g., tuple rate, key distribution) [22, 33, 50], but they mainly target stable, cloud-only infrastructures that support proactive planning [15]. Recent works have also proposed solutions for volatile sensor-edge-cloud infrastructure. Falcon [34] launches new instances of affected operators on globally accessible cloud nodes and uses special components to route tuples from mobile devices to the old edge node (where the data source was connected before relocation) and cloud nodes. This results in increased CPU and network bandwidth utilization, additional processing latency, and increased system complexity due to special routing components. `ISQP` [8] proposes an efficient approach to incrementally and concurrently re-optimize operator mappings to keep them valid in a volatile topology, but does not address redeploying these mappings. Overall, a solution is required to redeploy queries in a volatile sensor-edge-cloud infrastructure efficiently. We further motivate this requirement using a real-world application.

Smart Mobility Application. *Consider a real-time public transport application that computes the number of passengers on trains and their distribution across coaches [39]. Each train has passenger counting units that record the number of passengers, bikes, or strollers boarding and alighting at each stop. A stream query processes this data to display real-time occupancy information at the next train station or joins the occupancy data from incoming and outgoing trains at a transfer station to identify potential overcrowding situations. While the trains can perform local computations (filter, transformation, partial aggregation), more holistic computations (joins, global aggregation) need to be placed on an edge or cloud node to access holistic data.*

To assess the impact of topology changes on query processing latency, we emulated a smart mobility scenario. We simulated a circular train line, similar to Berlin's S41/S42, with 40 trains running simultaneously. Ten continuous queries were deployed, each processing data from a group of four trains. Under stable conditions, when trains are stationary and no topology changes occur, the DSPE maintains an average event time latency (excludes window size [26]) of 5.7 ms.

At the 20th second of the experiment, we introduced mobility by emulating the movement of all trains for 120 seconds, ending at the 140th second. As the trains circulate, they frequently disconnect and reconnect between base stations. This results in 10 reconnects per second—i.e., 10 topology changes per second—each affecting all 10 running queries.

Due to these frequent topology changes, the DSPE continuously adjusts operator placement and redeploys affected queries. However, this persistent redeployment process overwhelms the system, causing it to stop processing data beyond the 27th second. The optimizer re-triggers operator re-placement and redeployment every second, consuming resources that would otherwise be used for actual data processing. Consequently, latency-sensitive downstream applications experience significant disruptions in data delivery. Similar scenarios arise in connected car applications [23], which can generate even larger volumes of data per hour across numerous mobile vehicles [29, 42].

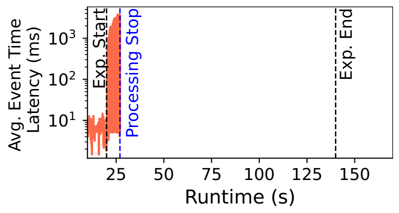

Figure 1: Impact of topology changes on event time latency.

This example shows the need for an efficient deployment mechanism that can handle high-frequency unplanned topology changes in sensor-edge-cloud while maintaining continuous processing and correctness. To this end, a DSPE must address the following challenges.

C1: Resume interrupted queries while keeping deployment latency low. To resume interrupted queries, SOTA DSPEs: (1.) undeploys operators of the affected queries from the topology, (2.) performs operator re-placement, (3.) redeploys *all* operators on the topology, and (4.) resumes processing from last checkpoint. However, when the topology changes often, the affected queries continuously invoke expensive undeploy, optimize, and redeploy phases of the optimizer to resume execution. This adversely affects the latency of deploying updated placements resulting in higher event latency as little data gets processed (cf. Figure 1). Thus, a DSPE must minimize deployment latency under frequent topology changes.

C2: Resuming stateful queries in a dynamic and hierarchical infrastructure. An edge infrastructure enables *near-data* processing, i.e., processing closer to the sensors. This allows for several optimization opportunities; for instance, a DSPE can compute aggregations closer to the sensor nodes to reduce processing latency and minimize data transfer. Generally, when migrating stateful operators due to a topology change, their state must also be migrated. This migration is typically done by DSPEs through state transfer or state replication using specialized components [15, 22]. Furthermore, peer-to-peer protocols are widely used for state transfer; yet, they are not suitable for use in a hierarchical network topology. In a dynamic and hierarchical topology, it is difficult to predict the location of mobile device reconnections, complicating the identification of nodes for state replication. Therefore, another critical challenge is to perform state migration in a dynamic and hierarchical infrastructure.

In this paper, we propose *Incremental Stream Query Deployment* (`ISQD`) to efficiently redeploy queries affected by unplanned topology changes. `ISQD` is designed for dynamic sensor-edge-cloud infrastructures that ensures uninterrupted operation similar to *cloud* environments [18] while enabling near-data processing on volatile *sensor-edge* infrastructures [34, 36]. To reduce the deployment latency (**C1**), `ISQD` concurrently identifies the minimum set of operators to redeploy for all affected queries while continuing to process and buffer data at the unaffected operators. This avoids unnecessary undeployments and redeployments of unaffected operators, helping to reduce deployment latency. To address state migration (**C2**), `ISQD` deploys ad-hoc stream queries to perform state transfer between topology nodes. This enables `ISQD` to perform state migration in a hierarchical and dynamic infrastructure using existing data processing capability of the DSPE. It further simplifies system design by eliminating the need for additional components and protocols.

We implement `ISQD` in the state-of-the-art DSPE NebulaStream [52] and show that `ISQD` keeps up with a high rate of topology changes while ensuring low query deployment and event-time latencies relative to baseline approaches. ISQD reduces deployment and event-time latencies by up to 7.5× and 39×, respectively, relative to the strongest baselines, while accommodating high-frequency topology changes. In sum, we make the following major contributions after discussing preliminaries in Sec. 2:

- We present ISQD, a framework that keeps deployment latency low under continuous changes to the physical topology (Sec. 3).
- ISQD computes a minimum set of redeployments to resume the affected query and reduce the overall deployment latency (Sec. 4).
- We introduce *ad-hoc state migration queries* to handle the redeployment of stateful operators in a dynamic and hierarchical infrastructure (Sec. 5).
- We present a protocol that combines deployment contexts and ad-hoc state migration queries to allow the resumption of queries interrupted due to topology changes (Sec. 6).
- We conduct detailed evaluation of ISQD with state-of-the-art baselines. We show that ISQD reduces overall deployment latency and keeps the event time latency to a minimum (Sec. 7).

2 PRELIMINARIES

Physical Topology. We denote the underlying physical infrastructure by a directed graph $\mathbf{T}=(\mathbf{N},\mathbf{L})$ where N is a set of nodes (devices) with compute and memory resources, and L is a set of network links between pairs of nodes.

Global Query Plan. The global query plan (GQP) captures all concurrently running queries in the DSPE. We represent GQP as a directed acyclic graph $\mathcal{G}=(\mathcal{O},\mathcal{E})$ where $\mathcal{O}$ denotes the set of operators, and $\mathcal{E}$ denotes the set of directed data flow edges between operators. If queries share common sub-expressions, the optimizer may have connected their plans in the GQP. However, queries that do not share common operators result in disconnected components in the GQP.

Operator State. Each operator $o \in \mathcal{O}$ is associated with a state $S_t(o)$, which captures the tuples retained by the operator at time t. Formally, the state of an operator is a function: $\mathbf{S_t(o)=W(o,H_t(o))}$ where:

- W is a function that periodically prunes $H_t(o)$ based on its semantics (e.g., time-based or count-based constraints).[1]
- $H_t(o)$ represents the processed tuples retained by o up to time t.

Some operators, such as windowed aggregations or joins, maintain a non-empty state, while stateless operators (e.g., selections, projects) have $S_t(o)=\emptyset$ at all times.

Operator Placement. The *operator placement* determines how operators from the global query plan (GQP) are assigned to nodes in the physical topology. Formally, the *operator placement* $\mathcal{P}_{\mathcal{G},T}$ is a relation $\mathcal{P}_{\mathcal{G},\mathbf{T}} \subseteq \mathcal{O} \times \mathbf{N}$ where:

- $\mathcal{O}$ is the set of operators in the GQP $\mathcal{G}$.
- N is the set of physical topology nodes in the system
- Each pair $(o,n) \in \mathcal{P}_{\mathcal{G},T}$ indicates that operator o is placed on node n.

The mapping defines how the GQP is distributed across the topology, influencing performance, resource utilization, and fault tolerance [10, 30]. Operators may be mapped to a single node or replicated across multiple nodes depending on the DSPEs partitioning and replication strategies.

[1]In practice, $S_t(o)$ can often be derived from $S_{t-1}(o)$. For example, when the window function is associative aggregate function.

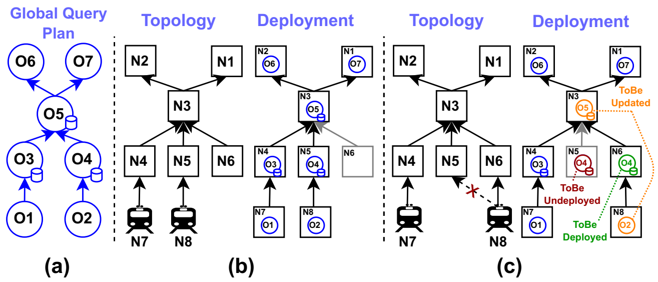

Figure 2: (a) Example global query plan; (b) Example topology and deployed DQPs at t_0; (c) Topology and deployed DQPs at t_1.

Decomposed Query Plan. After the operator placement is performed, the GQP $\mathcal{G}=(\mathcal{O},\mathcal{E})$ is partitioned into decomposed query plans (DQP), each deployed on a specific topology node. A DQP is formally defined as a tuple $\mathbf{D}=(\mathcal{O}_\mathbf{D},\mathcal{E}_\mathbf{D},\mathbf{n_D})$ where:

- $\mathcal{O}_D \subseteq \mathcal{O}$ is a maximal connected subset of operators that are co-located on the same node n_D.
- $\mathcal{E}_D=\{(o_i,o_j) \in \mathcal{E} \mid o_i,o_j \in \mathcal{O}_D\}$ represents the directed data flow edges within the DQP.
- n_D is the topology node where all operators in D are placed, as determined by $\mathcal{P}_{\mathcal{G},T}$.

The global query plan $\mathcal{G}$ is then represented as a collection of decomposed query plans distributed across different topology nodes $\mathcal{D}=\{\mathbf{D_1},\mathbf{D_2},\ldots,\mathbf{D_k}\}$ where each D_i corresponds to a distinct DQP running on a specific topology node.

Example (Decomposed Query Plans). *Figure 2(a) shows an example GQP representing a query computing the number of passengers boarding and alighting trains based on a session window. Operators $O1$ and $O2$ represent the sources, operators $O3$ and $O4$ compute the partial aggregates on passenger count, operator $O5$ computes the holistic aggregation, and operators $O6$ and $O7$ write output to external systems. Figure 2(b) shows the infrastructure topology at time t_0 and all DQPs deployed after performing operator placement of the GQP. For example, the operator $O1$ from the GQP is placed on the mobile node $N7$ and forms the only deployed DQP on that node. Similarly, operators $O2$, $O3$, $O4$, $O5$, $O6$, and $O7$ form the DQPs on topology nodes $N8$, $N4$, $N5$, $N3$, $N2$, and $N1$, respectively.*

Operator Deployment and Reconfiguration in a Dynamic Environment. We first formally define a valid DQP before discussing scenarios leading to invalid DQPs and their reconfiguration.

Valid Decomposed Query Plan. Given a GQP $\mathcal{G}$ and a physical topology T, after performing operator placement $\mathcal{P}_{\mathcal{G},T}$, we say that the set of decomposed query plans $\mathcal{D}$ is valid if and only if the following conditions hold:

(1) Valid Operator Placements in DQP: For each DQP $D \in \mathcal{D}$ deployed on a node n, every operator in D must be placed on n according to the operator placement mapping: $\forall \mathbf{o} \in \mathcal{O}_\mathbf{D}, \quad \exists (\mathbf{o},\mathbf{n}) \in \mathcal{P}_{\mathcal{G},\mathbf{T}}$. This ensures that a DQP does not contain operators that are not assigned to the nodes where it is deployed.
(2) Connectivity Between Distributed Operators: Given two operators $o_1 \in D_1$ and $o_2 \in D_2$ deployed on nodes n_1 and n_2, respectively, if there exists an edge between them in the global query plan ($(o_1,o_2) \in \mathcal{E}$), then there must be a valid communication

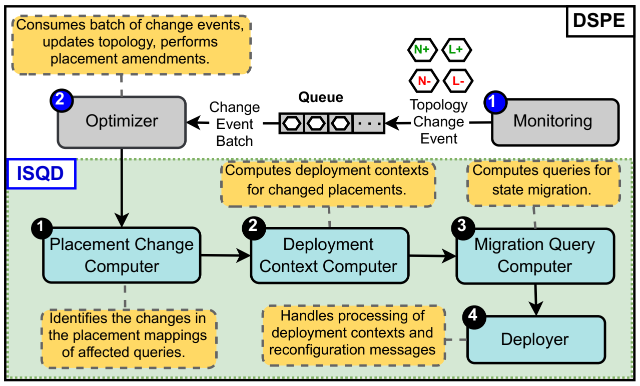
DSPE
Consumes batch of change events, updates topology, performs placement amendments.
N+
L+
N-
L-
Queue
Optimizer
Change Event Batch
Topology Change Event
Monitoring
ISQD
Computes deployment contexts for changed placements.
Computes queries for state migration.
Placement Change Computer
Deployment Context Computer
Migration Query Computer
Identifies the changes in the placement mappings of affected queries.
Handles processing of deployment contexts and reconfiguration messages
Deployer

Figure 3: System overview of ISQD within a DSPE.

path $P(n_1,n_2)$ between the nodes in the physical topology:

$$\exists(o_1,o_2)\in\mathcal{E}\wedge\{(o_1,n_1),(o_2,n_2)\}\in\mathcal{P}_{\mathcal{G},T}\implies\exists P(n_1,n_2)\in T$$

This guarantees that tuples can be correctly transmitted across operators in different DQP instances over the infrastructure.

(3) State Integrity for Query Processing: At any time t, the state of each operator must retain tuples processed since the start of query execution, subject to pruning by the state management function $\mathbf{W}:\forall\mathbf{o}\in\mathcal{O},\mathbf{S_t}(\mathbf{o})=\mathbf{W}(\mathbf{o},\mathbf{H_t}(\mathbf{o}))$ where $H_t(o)$ represents the tuple history and W defines the state management function. This condition ensures that stateful operators (e.g., aggregations, joins) maintain complete and valid state information necessary for correct query results.

Maintaining Valid DQPs During Reconfiguration. The above conditions collectively ensure that each DQP deployed on node n remains valid. Condition (1) ensures correct operator placement, preventing an operator from being executed on an unintended node. Condition (2) enforces connectivity constraints, ensuring tuples are propagated correctly through the infrastructure. Condition (3) guarantees state correctness, preventing state loss during reconfigurations. However, dynamic changes in the infrastructure can invalidate existing DQPs. To reveal this, consider the following example.

Example (Invalid Decomposed Query Plans). *Consider that the DSPE receives a topology change event as the train represented by node* N8 *moves and gets connected to node* N6. *The optimizer adapts the placement and performs redeployment of the* DQP *containing operator* $O4$ *from the node* N5 *to* N6. *Figure 2(c) shows the updated topology and the existing deployments where* DQP*s become invalid (all* DQP*s that are not in blue color). In particular, the* DQP *containing operator* $O5$ *on node* N3 *violates Condition (1) as there exists no placement of operator* $O4$ *on* N5. *The* DQP *containing* $O2$ *violates Condition (2) as it tries to connect to the* DQP *containing the older instance of* $O4$ *on* N5 *but neither the path between the nodes* N8 *and* N5 *exists nor the mapping of placement mapping of* $O4$ *on* N5. *The newly deployed* DQP *containing the operator* $O4$ *violates Condition (3) as the state of the operator does not contain tuples arrived since the start of the* GQP. *The* DQP *on the node* N3 *violates Condition (2) as it receives the tuples from* DQP *on* N5 *containing operator* $O4$ *with invalid mapping.*

Discussion. In this paper, we seek to incrementally un-deploy, redeploy, and reconfigure DQPs when a new operator placement mapping $\mathcal{P}^{\text{new}}_{\mathcal{G},T}$ is computed under continuous infrastructure changes. In particular, our goal is to (1) incrementally migrate operators to ensure that they are executed on the correct topology nodes and that the dataflow paths are (incrementally) reconfigured to preserve operator connectivity and (2) ensure state consistency by devising operator state transfer mechanisms for new operator instances to continue processing without loss of data.

3 SYSTEM OVERVIEW

This section provides an overview of our framework ISQD. ISQD supports *incremental redeployment* of queries impacted by *infrastructure changes* in dynamic *sensor-edge-cloud* environments. It ensures minimal downtime and uninterrupted query execution, making DSPEs resilient to infrastructure fluctuations.

3.1 System Architecture

Figure 3 illustrates the integration of ISQD within a DSPE and highlights its internal components within the green box. A DSPE consists of the following primary components:

1 **Monitoring Component:** The monitoring component continuously tracks topology events caused by node/link additions or removals. It registers these events in a *queue* for processing.
2 **Optimizer:** The DSPE batches topology change events to improve processing throughput. For each batch, the optimizer (1) identifies queries affected by the infrastructure changes, and (2) updates their operator placement mappings accordingly.

Note that DSPEs can use various placement strategies to adapt operator placements with respect to infrastructure changes [8]. However, ISQD operates independent of the selected placement strategy and focuses only on the deployment and reconfiguration of affected queries. Additionally, we designed ISQD as a central component to reduce system complexity, number of communication and synchronization points, and easy integration within SOTA DSPEs.

3.2 Components of ISQD

We now introduce the main components of ISQD. These components enable ISQD to avoid touching operators that are not impacted by the topology changes and perform fine-grained redeployments of only the necessary operators.

1 **Placement Change Computer:** The *placement change computer* identifies all operator placements affected by topology changes and determines whether they require redeployment. This design allows ISQD to work with arbitrary operator placement algorithm. Instead of redeploying entire queries, it selectively updates only the necessary operators, thus minimizing deployment overhead and reducing latency. The placement change computer invokes the deployment context computer for each affected query to determine the necessary deployment actions concurrently.
2 **Deployment Context Computer:** The deployment context computer determines the deployment mode for each affected operator. In particular, these modes include *un-deploy and redeploy* if the operator moves to a new node or *update* if the operator remains on the same node but requires reconfiguration. Since operators belong to decomposed query plans (DQP) (cf. Sec. 2), the deployment context computer updates DQPs and represents

them within DCs. These *deployment contexts* (DCs) make up the fundamental units for incremental query redeployment. Details about DCs are discussed in Sec. 4. A critical challenge in the redeployment of operators is state migration for stateful operators (e.g., aggregations, joins), as incorrect transfer or missing state can lead to incorrect results or data loss. To this end, ISQD uses ad-hoc state migration queries.

❸ **Migration Query Computer:** The *Migration Query Computer* issues state migration queries, which are ad-hoc queries, to transfer state between nodes during redeployment. ISQD leverages existing DSPE query infrastructure instead of introducing specialized state migration components or modifying peer-to-peer protocols to work in a hierarchical infrastructure. This approach has two key advantages: (i) it simplifies the system design by eliminating the need for a dedicated state migration service, and (ii) it supports dynamic and hierarchical infrastructure (**C2**) by handling arbitrary state migrations. The Migration Query Computer determines how to partition and migrate operator state efficiently and where to deploy auxiliary DCs to facilitate state transfer. Sec. 5 presents more details on state migration.

❹ **Deployer:** The *Deployer* initiates the reconfiguration process by sending computed DCs to the appropriate topology nodes. Since multiple DCs and thus DQPs can be updated simultaneously at different nodes, a critical challenge lies in synchronizing updates to avoid any tuple loss. For example, consider that two connected DQPs (DQP1 $\mapsto$ DQP2) need to be updated. If DQP2 is updated before DQP1 sends its in-flight tuples to DQP2, these in-flight tuples may get lost or processed incorrectly. To prevent this, ISQD introduces a *Reconfiguration Marker Mechanism* that we discuss next.

Reconfiguration Marker Mechanism. The Reconfiguration Marker ensures correct synchronization between deployment actions. Each marker contains: the affected DQP identifier and the action to be performed (un-deploy, deploy, update, etc.). The Deployer inserts reconfiguration markers into the dataflow pipeline to synchronize the execution of DCs. In more detail, when the Deployer receives a batch of DCs, it performs the following steps:(1) it transmits all DCs to their target nodes; (2) it computes a reconfiguration marker based on the DCs; and (3) it inserts the reconfiguration marker into the dataflow pipeline to synchronize the processing of DCs. This ensures that all required redeployment instructions are present before the processing begins. Further details on reconfiguration markers and their processing logic are discussed in Sec. 4 and Sec. 6, respectively.

End-to-end Processing. Overall, when the monitoring component detects topology changes, ISQD reacts to the changes by invoking the optimizer to update operator placement mappings. The Placement Change Computer determines affected operators, while the Deployment Context Computer computes necessary deployment actions. For stateful operators, the Migration Query Computer issues ad-hoc state migration queries, ensuring the correct operator state while initializing the new instance of a migrating stateful operator. Finally, the Deployer executes reconfiguration by deploying updated (DCs) while synchronizing updates using reconfiguration markers to prevent tuple loss. The processing of topology changes, performing operator re-placement, and computation of DCs and RM is done centrally. In contrast, the processing and reconfiguration of updated DQPs deployed using DCs is decentralized at each worker node. Note that ISQD

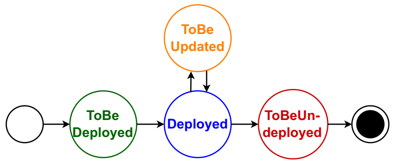

Figure 4: Decomposed query plan state transition diagram.

gracefully handles failures caused by topology changes during redeployment and reconfiguration by reverting the system to its previous stable state. All current and new topology changes are then reconsidered in the subsequent ISQD invocation. Overall, by incrementally identifying and adapting affected queries, ISQD minimizes deployment latency and ensures continuous query execution, making it robust for latency-sensitive applications. This efficient handling of topology changes enables ISQD to keep up with high-throughput topology changes despite being a centralized component.

Discussion. ISQD relies on a centralized monitoring component to collect updated information on the infrastructure topology. As a future direction, decentralized network protocols could be leveraged to detect topology changes more rapidly [47, 48]. Integrating these protocols with ISQD's deployment and reconfiguration strategies may further reduce redeployment and reconfiguration latency. We leave the exploration of such optimizations to future work.

4 DEPLOYMENT CONTEXTS AND RECONFIGURATION MARKERS

First, we explore various scenarios that arise when redeploying a query affected by topology changes and explain how DCs capture these scenarios in Sec. 4.1. Then, we discuss the role of reconfiguration markers alongside DCs and their internals in Sec. 4.2.

4.1 Deployment Context

DQP States. Once the operator re-placement is complete for the queries affected by topology changes, the next step is to compute and deploy DQPs on the topology nodes to resume query execution. However, holistically deploying all DQPs can result in a high deployment latency. ISQD mitigates this by identifying the DQPs affected due to operator re-placement optimization and redeploys them *exclusively*. In particular, ISQD focuses on the redeployment and reconfiguration of DQPs that satisfy one of the following three scenarios: (1) adding new DQPs on the topology nodes; (2) removing existing DQPs in case all operators need to be removed from a topology node; or (3) updating existing DQPs when either new operators are added, or existing operators are removed from the DQPs.

Based on these three scenarios, DQPs can be in one of three states: ToBeDeployed, ToBeUpdated, and ToBeUndeployed (illustrated in Figure 4 with transitions). The deployment contdetailext computer computes a deployment context (DC) for each of the affected DQPs to reflect these states. To this end, ISQD computes placement differences for affected queries and updates only those DQPs with changed placements to minimize deployment time.

The example from Figure 2(c) represents all three states. In particular, the DQP on N6 is marked for ToBeDeployed to start a new instance of operator $O4$, DQPs on N8 and N3 are marked for ToBeUpdated as the DQPs need to send and receive data to and from new DQPs respectively, and the DQP deployed on N5 is marked for ToBeUndeployed as the optimizer removed the placement.

Computing Deployment Contexts. Deployment contexts (DCs) represent the information necessary to redeploy affected DQPs, as the changes in DQPs may require different handling during redeployment. In particular, ISQD analyzes the affected DQP as follows: (1) For each DQP marked for ToBeDeployed, the DC contains the location and the connected operator graph deployed on the node; (2) For each DQP marked for ToBeUpdated, the DC contains the node location, identifier of the existing DQP, and updated connected operator graph; (3) For each DQP marked for ToBeUndeployed, the DC contains the node location, and the identifier of the DQP. Once all DCs are computed, they are transmitted to the appropriate topology nodes. However, concurrently processing these DCs is not trivial and presents several challenges. In the subsequent section, we examine these challenges and explain how ISQD addresses them.

4.2 Reconfiguration Marker

Reconfiguration Marker. DCs encapsulate the instructions required by the topology nodes to reconfigure existing DQPs. These DCs can be transmitted concurrently to the topology nodes. However, processing these instructions concurrently can result in loss of tuples, leading to incorrect results. In Figure 2(c), if the DQP on N3 is updated to receive data from the newly deployed DQP on N6 without accounting for in-flight tuples from the DQP on N5, the in-flight tuples will be lost resulting in incorrect results produced by the query. Therefore, a mechanism is required to synchronize the processing of DCs.

To address this issue, ISQD utilizes a *reconfiguration marker* to establish a synchronization barrier, ensuring that the deployed DCs are processed in the correct order. Marker-based reconfiguration protocols have been proposed to enable the dynamic reconfiguration of stream processing systems [15, 27, 33–35, 45]. A similar concept is employed in Apache Flink [18], where asynchronous snapshots are computed using markers to ensure fault tolerance and failure recovery. ISQD builds upon these ideas and adopts reconfiguration markers to establish synchronization barriers within a reconfigured query.

The core idea is to send the computed DCs to topology nodes, insert a reconfiguration marker into the data flow graph, and use this marker to trigger DC processing. As the marker flows through DQPs, each one checks for instructions to update or terminate itself based on the DCs deployed on the node. The marker then continues downstream to the next DQP, propagating the remaining DCs. Section 6 details how DCs and the reconfiguration marker enable reconfiguration.

The Deployer component analyzes DCs and generates a reconfiguration marker. This marker includes a set of DQP identifiers, actions, and metadata, specifying which DQP needs reconfiguration and how to reconfigure it. As the marker moves through a DQP, the identifier helps determine whether the DQP should respond. The actions and metadata provide the necessary details for processing the marker.

Reconfiguration Actions. A reconfiguration marker can specify one of three actions: *drain*, *update*, and *update then drain* The *drain* action signals a DQP to flush all in-flight tuples and terminate. It is used to safely remove DQPs that the placement optimizer no longer needs, ensuring in-flight data is processed without loss. For example, in Figure 2(c), the marker includes a drain action for the DQP on N5, ensuring its in-flight tuples are correctly handled before termination.

The *update* action indicates that the intended DQP needs to flush and update to a newer DQP included in the deployed DC. This action updates the running DQPs by adding, removing, or updating operators

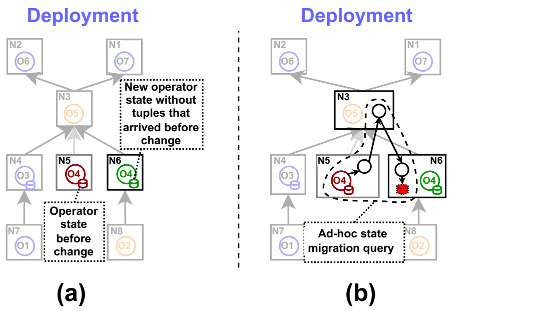
Deployment
N2
O6
N1
O7
N3
O5
New operator state without tuples that arrived before change
N4
O3
N5
O4
N6
O4
Operator state before change
N7
O1
N8
O2
(a)
Deployment
N2
O6
N1
O7
N3
O5
N4
O3
N5
O4
N6
O4
N7
O1
Ad-hoc state migration query
N8
O2
(b)

Figure 5: (a) Issue with reconfiguring DQP with stateful operators. (b) Using an ad-hoc query to perform state migration.

with new information from the optimizer. For example, in Figure 2(c), the reconfiguration marker will include update action for DQPs on nodes N8 and N3. The operators on these nodes must be updated to transmit and receive data from the new DQP on N6 instead of N5.

The *update then drain* action indicates that the target DQP must first update to a new DQP and subsequently terminate by flushing all in-flight tuples. This action briefly updates a DQP to a newer version to perform necessary cleanup tasks. For instance, this action can be used to transfer intermediate data of a DQP to another node for persistence prior to its termination. For brevity, we defer additional details and the application of this action to Sec. 5.

A DC representing a new DQP does not require synchronization via a reconfiguration marker during initial deployment. However, if the new DQP includes a stateful operator previously running on another node, synchronization is necessary. The new operator must wait for the state transfer to complete before starting execution, ensuring compliance with condition (3) in Def. 2.

Discussion. ISQD leverages deployment contexts and reconfiguration markers to minimize redeployments, enabling faster reconfiguration of existing decomposed query plans (**C1**). While ISQD supports incremental deployment, handling query dynamism—i.e., continuous changes in query structure due to operator additions or removals—remains an orthogonal challenge that depends on state-sharing mechanisms [27, 28]. This aspect, which underpins optimizations such as multi-query compute sharing and adaptive resource management, is an important direction for future research.

5 HANDLING STATEFUL OPERATORS

This section details the approach used by ISQD for reconfiguring DQPs with stateful operators. We highlight why deploying new and draining old instances of an operator is insufficient for provisioning a stateful operator (Sec 5.1). Lastly, we present how ISQD uses ad-hoc queries in conjunction with the *update then drain* action to guarantee that no tuple loss occurs, thereby ensuring the correctness of the result after reconfiguration (Sec. 5.2).

5.1 Redeploying Stateful Operators

Redeploying a DQP can involve starting new instances of operators on another node while terminating all old operator instances. Redeployment of a DQP with only stateless operators is relatively simpler, as the new instances of stateless operators can start while the older instances terminate. On the contrary, this does not apply to the DQPs with stateful operators, as the processing of incoming tuples may

Algorithm 1: Processing topology changes

```
Input: $\vec{TC}$
UpdateTopology($\vec{TC}$)
$Q_{affected}$ = FindAffectedQueries(GQP,$\vec{TC}$)
for all Q ∈ $Q_{affected}$ in parallel do
    UpdatePlacement(Q)
    $\Delta\mathcal{P}^{new}_{\mathcal{G},T}$ = ComputePlacementChanges(Q)
    $\vec{DC}$ = ComputeDeploymentContexts($\Delta\mathcal{P}^{new}_{\mathcal{G},T}$)
    $\vec{DC}_{mig}$ = ComputeStateMigrationQueries($\Delta\mathcal{P}^{new}_{\mathcal{G},T}$)
    DeployDeploymentContexts($\vec{DC}$+$\vec{DC}_{mig}$)
    RM = ComputeReconfigurationMarker($\vec{DC}$+$\vec{DC}_{mig}$)
    SendReconfigurationMarker(RM)
```

rely on the results of previously processed tuples (cf. Sec. 2). In particular, stateful operators retain the results of previously processed tuples or computations as internal states. Therefore, when a stateful operator is moved to a new node, its state must be transferred as well in order for the processing to continue correctly.

Figure 5(a) presents an example of DQPs involving stateful operators. The state of operator O4 on the previous node N5 contains the results of tuples processed prior to the change in the underlying topology. However, the state of the operator O4 on the new node N6 is initially empty. Simply starting execution with this new instance of operator O4 will result in an incorrect computation, as the output depends on the computation performed by the operator on N5, rendering the deployment invalid per Def. 2. To address this problem, the operator state must be migrated from the old to the new deployment.

State migration has been widely studied, with existing approaches falling into three categories: state replication, recreation, and transfer [15, 33, 38, 50]. However, these methods are ineffective in sensor-edge-cloud infrastructures because: (1) sparse connectivity makes peer-to-peer state transfer unreliable, and (2) unpredictable mobile device locations hinder effective state replication. ISQD addresses these challenges by using ad-hoc queries for state migration.

5.2 Ad-hoc State Migration Queries

ISQD leverages existing infrastructure and runs ad-hoc queries to perform state migration. Such queries, however, require additional deployment contexts and changes to the reconfiguration marker.

The migration query computer analyzes the computed DCs to identify stateful operators migrating to new nodes (cf. Figure 3). For example, in Figure 5(a), it detects that operator *O*4 is moving from N5 to N6 and generates an ad-hoc query for state migration. This query includes a source operator (on the terminating node) to read the state and a sink operator (on the new node) to write it in binary format for initialization. If intermediate nodes exist, DSPE inserts routing operators as in standard streaming queries. Figure 5(b) shows the ad-hoc query transferring *O*4's state from N5 to N6. The migration query computer also generates additional DCs to deploy these ad-hoc queries.

To prevent data loss during reconfiguration, a DQP must be updated and its state migrated before termination. This involves: (1) updating the DQP to include the ad-hoc query's source operator linked to the stateful operator, and (2) migrating the state to the new node. ISQD achieves this using the update-then-drain action (Section 4.2), which first updates the DQP and then migrates its state. For instance, the DQP on node N5 in Figure 5(a) is updated with the ad-hoc query shown in Figure 5(b); the query is terminated after migration completes.

Algorithm 2: Processing of reconfiguration marker by a DQP

```
Input: RM
if RM.contains(DQP.ID,DQP.Version) then
    RE = RM.get(DQP.ID,DQP.Version)
    switch (RE.action) do
        case Drain do
            flushAndCloseDQP(DQP.ID,DQP.Version)
        case Update do
            flushAndCloseDQP(DQP.ID,DQP.Version)
            startDQP(RE.mData.ID,RE.mData.Version)
        case Update then Drain do
            flushAndCloseDQP(DQP.ID,DQP.Version)
            startDQP(RE.mData.ID,RE.mData.Version)
            flushAndCloseDQP(RE.mData.ID,RE.mData.Version)
dispatch(RM)
```

The current version of ISQD migrates operator states without prioritizing which portion of the state should be transferred first. Meces [21] and Megaphone [22] shows that prioritizing state chunks during migration can reduce the time required to resume processing at the new operator instance. Nonetheless, ISQD can be extended to add such migration priorities or incorporate additional compression techniques [19] to reduce the data transferred during migration.
Discussion. We note that the state migration technique using ad-hoc queries relies on the availability of the node hosting the old instance of the stateful operators. However, this cannot always be guaranteed, making it prone to node failures. ISQD can be extended to handle node failures using two key approaches: *upstream backup* and *state replication* [15, 30]. The current implementation of ISQD uses upstream backups available in the underlying DSPE (cf. Sec. 7.2). However, extending ISQD with an inherent fault-tolerance protocol remains an important direction for future research.

6 RECONFIGURATION PROTOCOL

This section presents how ISQD performs overall incremental deployments of queries interrupted due to topology changes. In particular, ISQD first processes the topology changes to identify and compute DCs using the extended optimizer. Algo. 1 describes the processing done by ISQD at the optimizer. Afterward, ISQD performs the fine-grained reconfigurations and provisioning of updated DQPs deployed using DCs at the topology nodes. Algo. 2 describes the reconfiguration process of DQPs. To explain how ISQD performs the processing, we use the example global query plan, topology, and corresponding deployments from Figure 2. In addition, we refer to Figure 6 to illustrate various processing stages.
Processing Topology Changes. ISQD processes topology changes in batch at-a-time fashion using Algo. 1. When a batch of topology changes ($\vec{TC}$) arrives, ISQD first updates the topology and determines which queries are affected (Lines 1–2). For example, Figure 6(a) shows the topology after the train representing *N*8 disconnects from N5 and connects to N6 and the deployed DQPs. This disruption leads to two immediate issues: (I) The DQP on N8 can no longer forward its locally processed tuples due to the lost connection with downstream N5. To ensure continued operation, it begins buffering the processed tuples. Once the reconfiguration completes, these buffered tuples are forwarded to the new downstream DQP, minimizing the processing latency. (II) This disconnection also causes, albeit indirectly, the DQP on N3 to stagnate. It prevents the stateful operator on N3 from receiving

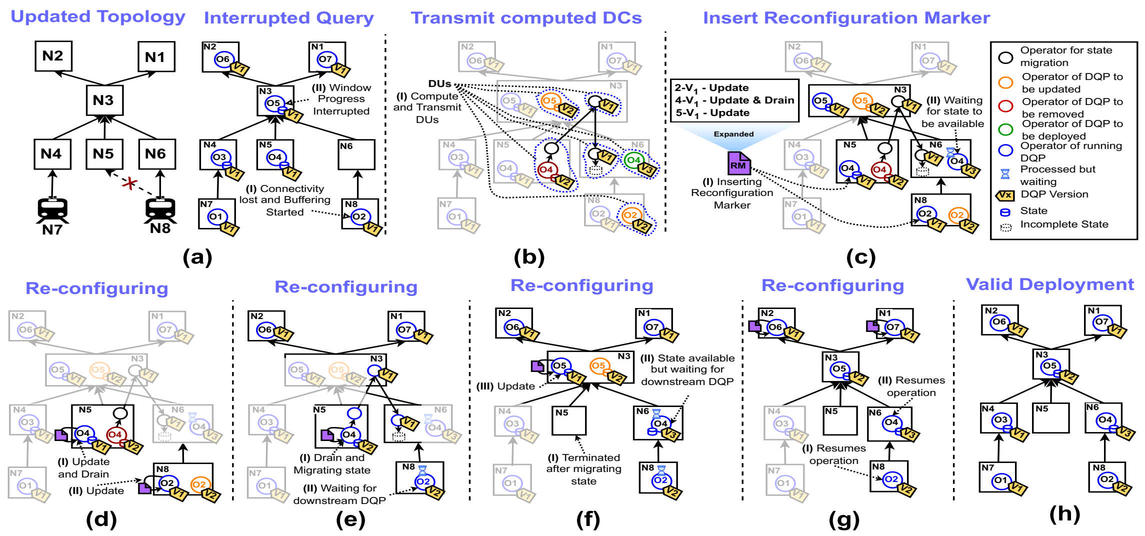
Updated Topology
Interrupted Query
Transmit computed DCs
Insert Reconfiguration Marker
(II) Window Progress Interrupted
(I) Connectivity lost and Buffering Started
DUs
(I) Compute and Transmit DUs
2-V1 - Update
4-V1 - Update & Drain
5-V1 - Update
Expanded
(I) Inserting Reconfiguration Marker
(II) Waiting for state to be available
Operator for state migration
Operator of DQP to be updated
Operator of DQP to be removed
Operator of DQP to be deployed
Operator of running DQP
Processed but waiting
DQP Version
State
Incomplete State
Re-configuring
Valid Deployment
(I) Update and Drain
(II) Update
(I) Drain and Migrating state
(II) Waiting for downstream DQP
(II) State available but waiting for downstream DQP
(III) Update
(I) Terminated after migrating state
(II) Resumes operation
(I) Resumes operation

Figure 6: ISQD processing topology changes and performing reconfiguration.

updated watermarks from N8, which are necessary to trigger the window computation. This results in the disruption of the entire query.

ISQD first spawns threads to process affected queries in parallel (Line 3). For each query, it updates operator placements and computes the resulting configuration changes (Lines 4–5). Since re-placement may modify existing DQPs or create new ones (cf. Sec. 4.1), ISQD identifies the impacted DQPs and generates the necessary DCs (Line 6). It then checks whether any stateful operator has migrated (Line 7); if so, it creates an ad-hoc state-migration query, updates the corresponding DQP, and computes its DQPs and DCs. Finally, all generated DCs are deployed to their worker nodes (Line 8).

Figure 6(b) shows the newly computed and deployed DCs. The DQPs on the N8 and N3 are updated to new versions of DQPs (shown in orange) as they need to send and receive data from new DQPs, respectively. A new DQP for the stateful operator O4 is deployed on N6 (shown in green). Since O4 is stateful, an ad-hoc query is generated to migrate its state from the old instance on N5 to the new one on N6. Accordingly, the DQP on N5 is updated (shown in red) to connect to the DQPs of the ad-hoc query (shown in black). Note, all other DQPs remain unchanged, allowing ISQD to minimize deployment time.

To synchronize the processing of newly deployed DCs, ISQD analyzes all DCs and computes a reconfiguration marker, which is then inserted at all affected leaf DQPs (Lines 9–10). Figure 6(c)(I) shows the reconfiguration marker's entries, listing DQP identifiers, required actions, and related metadata. For instance, DQPs at N2 and N5 (version V1) need updates, while the DQP at N4 (version V1) must be updated and then terminated. The marker is inserted into the leaf DQPs on N5 and N8. Entries for new DQPs to be deployed are not created in the reconfiguration marker, as they can be started asynchronously. However, new DQPs containing migrating stateful operators wait for old states to be available before starting. Figure 6(c)(II) shows that the DQP on N6 with the migrated operator O4 waits for the state migration.

Reconfiguring DQPs. Reconfiguration of new DQPs deployed via DCs occur in a decentralized manner. However, these reconfigurations are synchronized using a reconfiguration marker that is propagated through the dataflow graph. Upon receiving a reconfiguration marker, a DQP checks if its id and version are listed. If so, then it performs the processing based on the action defined in the reconfiguration marker. Otherwise, the DQP propagates the marker downstream.

For example, Figure 6(d) illustrates how DQPs at N5 and N8 process reconfiguration markers. In Figure 6(d)(I), the DQP at N5 detects the action update then drain (Line 9), flushes in-flight tuples, and updates from version V1 to V2, which includes an ad-hoc state migration operator for transferring the state of operator O4 (Lines 10–11). After completing the state transfer, it terminates (Line 12) and passes the reconfiguration marker downstream (Line 13). Similarly, in Figure 6(d)(II), the DQP at N8 detects the action update, flushes in-flight tuples, terminates, and updates to version V2 (Lines 6–8). It then discards the reconfiguration marker, as the V1 instance has no downstream DQPs.

Figure 6(e) shows the updated DQPs on N5 and N8, respectively. (I) The DQP with version V2 at N5 starts to transmit the state of the operator O4 to the newer instance on N6. (II) The updated DQP at N8 waits for the downstream DQP on N6 to start and to connect with it. In the meantime, the DQP at N8 continues the processing and buffers the processed tuples for later transmission. This allows the query to progress even when the reconfiguration is not completed.

Figure 6(f) illustrates the continued processing of the reconfiguration marker and the initialization of DQPs with migrating stateful operators. (I) The DQP on N5, which hosts the old instance of operator O4 along with ad-hoc query DQPs, terminates after completing state transfer. The reconfiguration marker is then forwarded to the downstream DQP on N3. (II) The DQP on N6 initializes operator O4 using the migrated state but waits for its downstream DQP on N3 to start. (III) Meanwhile, the DQP on N3 receives the reconfiguration marker and begins updating from version V1 to V2.

Figure 6(g) illustrates the DQP on N3 updating to version V2 and propagating RMs to downstream DQPs on N2 and N1. Since these downstream DQPs have no reconfiguration entries, they discard the marker and continue normal processing. (I) Simultaneously, the DQP on N6 connects to the updated DQP on N3 and resumes processing. (II) This, in turn, enables the DQP on N8 to resume as well. Figure 6(h) shows the final deployment state post-reconfiguration. This example demonstrates how ISQD incrementally deploys only affected DQPs and uses reconfiguration markers to coordinate their updates.

7 EVALUATION

We experimentally evaluate ISQP using an emulated edge–cloud infrastructure and compared it against SOTA redeployment approaches for queries with both stateless and stateful operators.

7.1 Experimental Setup

We implement ISQD and multiple baselines in NebulaStream [52], a state-of-the-art DSPE, to negate the influence of the underlying DSPE on experiment results.

End-to-End Baselines. We implement the following two baselines to evaluate the performance of using incremental and concurrent deployment strategy used by ISQD: *(1) Holistic Serial Query Redeployment*(HSQD): is the default behavior of NebulaStream and other state-of-the-art DSPEs [52]. This baseline first updates the topology; second identifies the affected queries; third serially performs placement updates for affected queries; and lastly, performs holistic redeployment of the affected queries. It uses the concept of upstream backups and checkpoints to ensure exactly-once guarantee between successive query restarts [30]. *(2) Holistic Concurrent stream Query Redeployment*(HCQD): performs the same steps as HSQD. However, it concurrently performs holistic placement and deployment for the affected queries to reduce the deployment and optimization time.

State Migration Baselines. We implement the following two baselines for evaluating the state migration strategy: *(1) State Recreation* (SR) strategy replays the previously played data streams from the source(s) to the downstream operators. This allows any downstream stateful operator to recreate the state without the need to do state migration. To this end, this strategy makes use of upstream backups and checkpoints to track the progress and replay the stream [30]. *(2) State Transfer* (ST) strategy transfers the snapshot of a pre-built state from the older instance of an operator to the newer instance of the operator. These strategies are commonly used in cloud data centers where a state snapshot is transferred between nodes residing in the same network area [15, 22]. However, for hierarchical infrastructure, two nodes can be connected via multiple hops. To handle this hierarchical nature, we modified the ST approach to transfer the state from a source to a destination node one hop at a time.

Topology. We base our experiment on an emulated infrastructure represented by combining OpenCelliD database [37] and the trajectory data from VBB (a public transport company) [20]. The OpenCelliD and trajectory dataset allows us to represent a dynamic sensor-edge-cloud infrastructure for smart transport use cases. In this infrastructure, IoT devices onboard the trains dynamically change their locations and the intermediate node to which they are connected as the train moves. We use the open-source tool from our recent publication that combines OpenCelliD and trajectory data to generate topology changes [8, 43] and simulate them [44]. These topology changes, in turn, impact the query processing of the data from IoT devices onboard the moving trains.

Evaluation Metrics. We evaluate the performance of ISQD and the baselines using the following metrics: *Event time latency:* The time from tuple creation to its eviction after processing by the DSPE [26]; *Aggregated deployment latency:* The total time spent handling topology changes, identifying affected queries, and redeploying them to resume execution; *State availability time:* The time required to make the operator's state available at the new instance to resume processing.

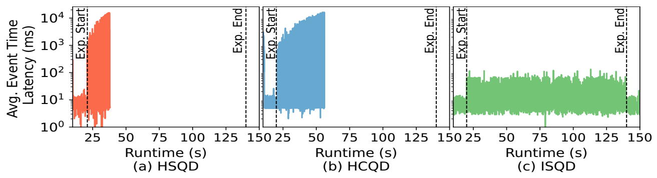

Figure 7: Effect of different strategies on the event time latency of running queries.

7.2 Experiments

This section summarizes extensive analysis of ISQD, HSQD, and HCQD. We vary the following parameters during our analysis: (a) *number of sources per query* to increase the size of affected queries, (b) *number of mobile sources per query* to increase the number of affected operators within queries, and (c) *rate of topology changes* to evaluate the performance of different approaches under stress. For all experiments (unless stated otherwise), we use a server with an AMD EPYC 7742 CPU and 1 TB of RAM. We set up a hierarchical infrastructure topology based on mobility data as discussed in Sec. 7.1.

7.2.1 ***Analyzing Deployment and Execution Latency:*** In this experiment, we analyze the deployment and event time latency incurred when using HSQD, HCQD, and ISQD for handling topology changes. We initialize the experiment by deploying 10 queries with stateless operators (maps and filters) that process data from a varying number of moving trains.

(1) Analyzing event time latency. We first perform a detailed analysis of event time latency incurred in different approaches while keeping all three parameters mentioned above constant.

Setup. We deploy 10 queries, each with 16 sources and 1 mobile source, under a topology change rate of 1s. Each experiment runs for 150s and is repeated three times. The run begins with a 20s stabilization phase, followed by 120s of continuous topology changes affecting all queries, and ends with a 10s recovery period. This setup ensures a consistent and controlled evaluation under dynamism.

Results. Figure 7 presents the evaluation of the three approaches. During the initial 20s, when no topology changes occur, the average event time latency remains around 5.7ms. However, during the subsequent 120s of continuous topology changes, both HSQD and HCQD struggle to keep up. As a result, all queries stop processing after 39s with HSQD and 58s with HCQD. Throughout this period, HSQD incurs an average event time latency of 238 ms, while HCQD experiences a significantly higher latency of 1306 ms. In contrast, ISQD successfully handles all topology changes within 12s-14s, i.e., 10.9× lower than HSQD and 2 to 7.5× lower than HCQD, and shows event time latency of 6.1ms, i.e., 39× and 214× lower than HSQD and HCQD respectively.

Discussion. This experiment shows that both HSQD and HCQD fail to handle all topology changes within the runtime. Their failure stems from repeatedly triggering full placement and redeployment between changes, leaving no time for data processing. HSQD fails earlier due to its sequential execution, while HCQD benefits from concurrency and lasts slightly longer. In contrast, ISQD detects that only

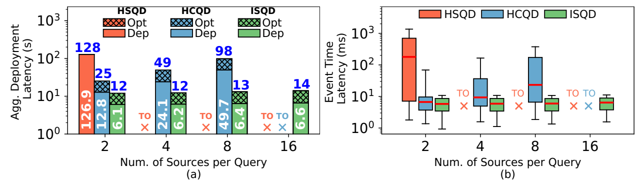
Figure 8: Impact of varying num of sources on (a) Agg. deployment latency; (b) Event time latency.

1 of 16 sources is affected and applies incremental changes, preserving unaffected operators. This allows ISQD to reduce deployment time and maintain continuous data processing.

(2) Varying the number of sources per query. We evaluate how query plan size impacts performance by increasing the number of sources per query. Large queries adversely affect deployment latency.

Setup. We maintain 1 mobile source per query and set the topology change rate to 1s. We vary the number of sources per query, increasing the size of query plans. In particular, we vary the number of sources from 2 to 16 and observe the aggregated deployment latency, the total topology changes, and the event time latency.

Results. Figure 8(a) and (b) show the aggregated deployment and event-time latencies for different approaches as the number of data sources increases. Aggregated latency is further broken down into placement optimization (shaded) and deployment time.

HSQD reaches 128s latency with two sources and times out beyond that. HCQD starts at 25s with two sources and rises to 98s with eight sources, failing to complete all topology changes (72%) at 16 sources. In contrast, ISQD maintains stable latency (12–14s) across all source counts, processes all topology changes, and keeps the 95th percentile of event-time latency under 10ms.

Discussion. This experiment demonstrates that ISQD outperforms HSQD and HCQD by selectively redeploying only the operators affected by topology changes, rather than applying a holistic approach. Since only one source per query moves per topology change, ISQD focuses on redeploying plans only for the affected source. As a result, it achieves the lowest deployment latency, successfully processes all topology changes, and maintains minimal event time latency, outperforming the other approaches.

7.2.2 *Analyzing Performance Under Stress.*

This experiment stresses redeployment and reconfiguration approaches by varying topology change rates and the number of mobile sources. Alongside stateless queries, we evaluate stateful queries with join operators, which involve 2× more total and mobile sources compared to stateless queries due to the binary nature of joins.

(1) Varying number of mobile sources per query. This experiment examines the impact of increasing the number of operators affected by topology changes across both stateless and stateful queries.

Setup. For stateless queries, we set the number of sources per query to 8, and for stateful queries, we set the number of sources to 16. We fix the topology change rate to 2s. To control the number of affected operators, we vary the mobile sources from 1 to 8 for stateless and 2 to 16 for stateful queries.

Results. Figure 9 shows deployment and event-time latencies for stateless queries (a, b) and stateful queries (c, d) as the number of mobile sources varies. HSQD fails to handle topology changes for both query types, regardless of source count. HCQD maintains a 37s deployment latency for stateless queries but fails for stateful ones,

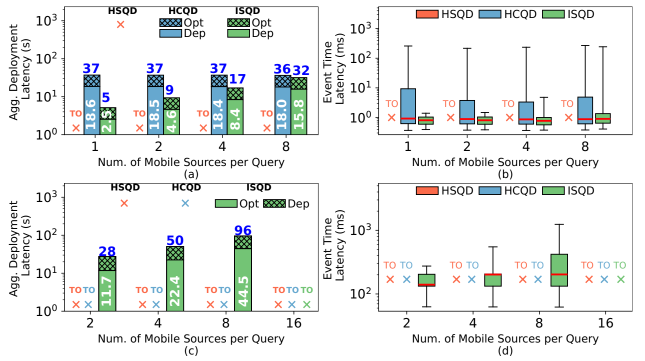
Figure 9: Impact of varying mobile sources on (a) Agg. deployment latency and (b) Event time latency for stateless queries; (c) Agg. deployment latency and (d) Event time latency for stateful queries.

similar to HSQD. For stateless queries, HCQD maintains a stable median event-time latency (1ms), while the 75th percentile varies between 3ms to 9ms as the number of mobile sources increases

ISQD, in contrast, achieves the lowest deployment latency among all baselines but shows an increase with the number of mobile sources. For stateless queries, the latency advantage drops to just 10%, while for stateful queries, ISQD times out at higher source counts.

Discussion. This experiment shows that ISQD's deployment latency is impacted by the number of operators affected by topology changes. In the worst case, its latency approaches that of HCQD and may even fail to complete reconfigurations. However, unlike HCQD, the event time latency of ISQD remains unaffected by the number of re-deployments. While HCQD replays tuples from the last checkpoint, ISQD buffers tuples at source operators during interruptions and replays them after redeployment, enabling continuous query progress and smoother execution under topology changes.

(2) Varying rate of topology changes. In this experiment, we evaluate the impact of topology change rates. A higher rate results in more topology changes being added to the queue (cf. Sec. 3). For stateless queries, we deploy 10 queries, with each query consuming data from 8 sources and 1 mobile source. For stateful queries, we deploy 10 queries, with each query consuming data from 16 sources and 2 mobile sources.

Setup. We vary the rate of topology changes from 4s to 0.25s. A lower value of the rate means more frequent topology changes, increasing the processing load on different approaches.

Results. Figure 10 shows the aggregated deployment latency and the event time latency for stateless ((a),(b)) and stateful ((c),(d)) queries as topology changes rates vary. HSQD fails to process topology changes across all rates and query types, except for the 4s stateless queries, within the experiment runtime. In contrast, HCQD maintains an aggregated deployment latency between 18s and 72s till the topology change rate of 1s for stateless queries, after that, it also times out. For stateful queries, it only operates reliably at a 4s rate. As topology change rates increase, event-time latency for stateless queries rises from 1ms to 25ms (median) and 1.09s to 52ms (75th percentile). For stateful queries, the median and 75th percentile latencies reach 341ms and 869ms, respectively, before timeouts at higher rates.

ISQD consistently achieves 7× lower deployment latency than HCQD for stateless and stateful queries. Unlike HSQD and HCQD, ISQD

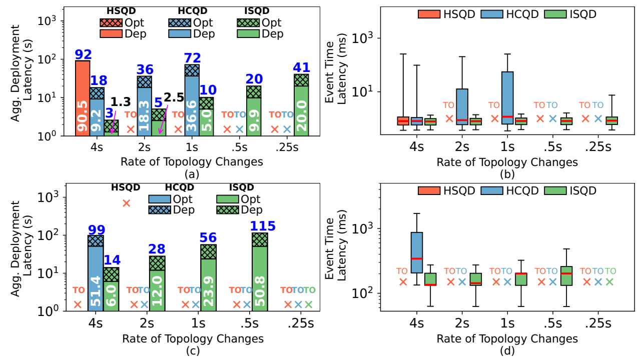
Figure 10: Impact of varying the rate of topology changes on: (a) Deployment latency and (b) Event-time latency for stateless queries; (c) Deployment latency and (d) Event-time latency for stateful queries.

handles all topology changes for stateless queries without failure. For stateful queries, it remains effective until a 0.25s change rate, after which it times out. Event-time latency for stateless queries stays near 1ms regardless of topology rate, while for stateful queries, median latency increases from 134ms to 202ms and the 75th percentile from 203ms to 258ms as the rate increases.

Discussion. In conclusion, ISQD effectively handles high rates of topology changes by redeploying only the affected operators. Its buffering mechanism allows unaffected operators to progress, resulting in lower event-time latency compared to baselines. However, ISQD incurs up to 5.5× higher deployment latency for stateful queries due to the added overhead of state migration during reconfiguration.

7.2.3 *Effect of Topological Complexities.*

We investigate how infrastructure topology complexity impacts the performance of ISQD. We conducted experiments on various topologies, including star/tree, full graph, and circular. As we focus on infrastructure connectivity, we throttle the network to 250 Mbps to highlight the impact of the data transmission path on redeployment and reconfiguration.

Setup. We deployed 10 stateful queries while keeping the number of sources, the number of mobile sources, and the rate of topology changes constant. In each experiment, each query performs analytics over four sources, including 2 sources moving at a rate of 2s.

Results. Our analysis confirms that the node count and network connectivity significantly impact ISQD's performance. Deployment latency is highest in circular topologies (65s), followed by star/tree (55s) and full-graph (52s). In circular topologies, persistent connectivity between intermediate nodes and mobile sources requires more system operator redeployments to reroute data, increasing deployment latency. Full-graph topologies require only single-hop deployments due to full inter-connectivity. Star/tree topologies require, while requiring stateful operator migration due to sparse links, avoid additional hops, resulting in slightly higher deployment latency but lower event-time latency compared to full-graph setups.

Our analysis indicates that greater inter-connectivity often reduces the need for operator redeployment and migration after topology changes. To isolate this effect, we ran experiments on star/tree topologies with 1–16 intermediate nodes. As the number of intermediates increases from 1 to 4, deployment latency rises from 55s to 75s due to (a) more DCs being computed and (b) increased state migration overhead. With 8 intermediates, ISQD fails to complete

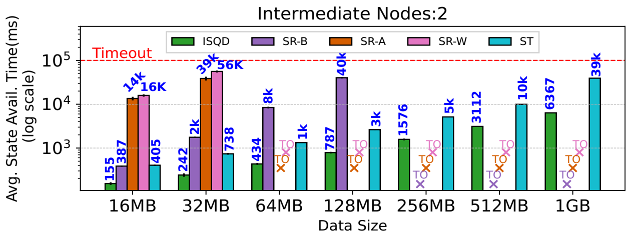
Figure 11: Effect of varying the size of data on the state availability time.

reconfiguration within the experiment time. We omit the plots and only report numbers due to space restrictions.

7.2.4 *Effect of Using Ad-hoc State Migration Queries:*

In this experiment, we compare ISQD's ad-hoc query-based state migration with ST and SR strategies (cf. Sec. 7.1). Additionally, when using SR, a stateful operator may perform extra work by re-triggering windows during stream replay. To assess the impact of window triggering, we define three SR variants: SR-B (Best): No extra work from window triggers, only state creation time. SR-A (Average): Half of the tuple buffers trigger windows, adding some extra work. SR-W (Worst): Every tuple buffer triggers windows, maximizing additional work.

We migrate the state of a join operator containing tuples from two streams and measure the total state availability time. We vary (1) the size of the state and (2) the total number of intermediate nodes between nodes hosting old and new instances of the join operator. Each topology node in these experiments is a server with an Intel(R) Xeon(R) Silver 4216 CPU and 500 GB of RAM.

(1) Varying size of data to be migrated. In this experiment, we vary the size of the state to be migrated from 16 MB to 1 GB. For SR, the state size corresponds to the total size of the tuple buffers that need to be replayed to reconstruct the operator state. To evaluate performance in a hierarchical infrastructure, we configure a NebulaStream cluster with two intermediate worker nodes positioned between the source and destination nodes. For ISQD and ST, these intermediate nodes are located between the worker nodes hosting the old and new instances of a stateful operator. However, for SR, the intermediate nodes are positioned between the worker nodes hosting the upstream operator (which contains the tuples to be replayed) and the stateful operator. Additionally, we set the experiment cutoff to 100s, i.e., if the state is not available within 100s, we mark the experiment as timeout.

Results. Figure 11 shows that both ST and ISQD achieve the fastest average state availability times across all data sizes, with ISQD outperforming ST by 2.6× to 6.2×. SR-W and SR-A perform the worst, timing out beyond 32MB. SR-B handles up to 128MB but is up to 50× slower than ISQD. As data size increases, all methods see longer state availability times due to higher data replay (SR-B, SR-A, and SR-W) or transferred (ST and ISQD).

Discussion. In this experiment, we conclude that ISQD delivers the best performance. For smaller datasets (≤16 MB), SR-B slightly outperforms baselines SR-A, SR-W, and ST as it only needs to reconstruct the state on a new node without experiencing the overhead of window triggers. However, as the data size grows (>16 MB), the overhead of building state from scratch for a larger volume of data exceeds transferring pre-built state (ST and ISQD).

The other two SR variants, SR-A and SR-W, incur overhead due to frequent window triggers. SR-A triggered 5331 to 15662 windows,

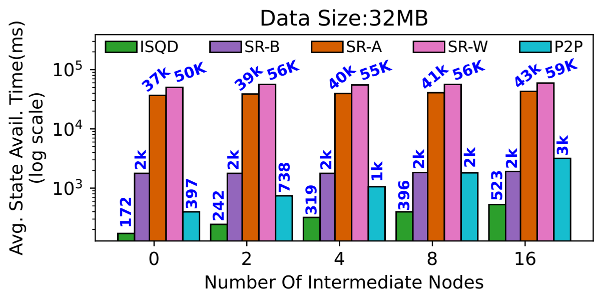

Figure 12: Effect of varying the number of intermediate nodes on the state availability time.

while SR-W triggered 14625 to 62122 windows while processing different amounts of replayed streams. This frequent triggering leads to back-pressure, increasing replay processing time and state availability time. ST suffers from multi-hop data transfer inefficiencies, waiting for the complete state chunk at each hop, which adds to the delays. Additional experiments with varying intermediate nodes and data sizes showed consistent trends across all approaches, but we omit some results due to space constraints.

(2) Varying number of intermediate nodes. In this experiment, we evaluate the impact of varying the number of intermediate worker nodes while keeping the data migration size constant at 32MB. We chose 32 MB as this is the maximum data size for which all baselines work within 100s timeout (cf. Figure 11).

Results. Figure 12 presents the results of our evaluation. We observe that ISQD consistently achieves the fastest state availability time as the number of intermediate worker nodes increases. ISQD demonstrates a significant speedup of 112 to 290× compared to the slowest approach, SR-W. In comparison to the best baseline, ST, it shows a speedup between 2× and 5×. Interestingly, as the number of intermediate nodes grows to 16, we see best baseline ST experiences a slowdown of 50% compared to SR-B.

Discussion. Overall, for ISQD, ST, SR-A, and SR-W, the average state availability time increases as the number of intermediate nodes grows. Since the data size remains constant, the computational effort required to reconstruct the operator state at the new instance does not change. The increase in state availability time is primarily due to the additional network latency introduced by data transfer across multiple hops. However, for ST, the impact is more significant as the state availability time increases from 397 ms to 3s. This approach requires the entire state chunk to be fully received at each hop before forwarding it to the next node, leading to a more pronounced increase in state availability time. Overall, ISQD is the fastest approach among all, as it transmits the state in binary chunks and reassembles them at the destination without triggering windows (unlike SR-B, SR-A, SR-W) or transmitting the entire state one-hop-at-a-time (ST).

8 RELATED WORK

Prior work has shown that sub-/super-aggregation and broadcast joins can enhance processing efficiency [31, 49]. However, these approaches assume stable infrastructures and do not address frequent reconfiguration needs in dynamic sensor-edge-cloud environments. In contrast, ISQD enables efficient query redeployment and reconfiguration under continuous topology changes with minimal overhead. Zhu et al. propose moving state and parallel track strategies to reconfigure states of windowed join operators on the same machine after changing join orders [55]. Wu et al. propose ChronoStream that enables elastic scaling of stateful stream operators by replicating states on multiple cloud nodes to create passive backups [50]. Mai et al. propose *Chi*, a system with a specialized control plane that allows on-the-fly reconfiguration of running stream queries to scale up and down operator parallelism [33]. Similarly, Bartnik et al.[3] propose a checkpoint-based mechanism that stores and restores operator state from external storage during migration. Del Monte et al. extend the previous approach for on-the-fly re-configuring operators with terabytes of state with their system Rhino [15]. Rhino replicates an operator's state to predefined nodes and migrates the operator's execution to them upon observing reduced operator performance. Similarly, Rajadurai et al.[38] propose Glos, which reconfigures stateful/stateless dataflow graphs without downtime, though it requires concurrent execution of old and new instances. Hoffmann et al. and Gu et al. proposed Megaphone [22] and Meces [21], respectively, to enable fine-grained state migration, unlike the single-shot approach in ISQD. Megaphone performs key-by-key migration and optimizes it by batching keys at various granularities. Meces prioritizes state migration based on the arrival order of keys, reducing queueing delays for incoming tuples. Both systems support dynamic scaling in cloud-based DSPEs and rely on multiple operator instances to gradually transfer state. Warnke et al. proposed using network protocols to collect information about the underlying infrastructure topology, and use this information to make decentralized query placement decisions to reduce network transfer between nodes [47, 48].

In contrast to these approaches, ISQD focuses on the redeployment, reconfiguration, and migration of stateful operators within a dynamic edge-cloud continuum. It does so within a hierarchical infrastructure, without prior knowledge of where new instances of operators will be deployed, and without introducing additional components for routing tuples or migrating states.

9 CONCLUSION

We proposed ISQD, a framework that enables DSPEs to efficiently redeploy and reconfigure running queries in response to frequent operator placement changes due to continuously evolving infrastructure topology. ISQD selectively identifies only the operators that require redeployment due to placement adjustments made by the optimizer. It then determines the necessary actions required to redeploy and reconfigure the affected operators incrementally. Additionally, ISQD identifies stateful operators that need to be redeployed and computes ad-hoc queries to migrate their state from one node to another. Our experimental evaluation shows that ISQD incurs up to 7.5× less deployment latency and up to 39× less event time latency compared to the strongest baseline while keeping up with high-frequency topology changes.

ACKNOWLEDGMENT

This work was funded by the German Federal Ministry for Education and Research as BIFOLD - Berlin Institute for the Foundations of Learning and Data (ref. BIFOLD24B) and Software Campus Project ESPAT (01IS23068). Additionally, we want to thank Dr. Stefan Halfpap for his valuable feedback.

REFERENCES

[1] Telia Company AB. 2023. Smart Public Transport. https://business.teliacompany.com/internet-of-things/smart-public-transport. (Accessed on 03/22/2023).

[2] Suliman Abdulmalek, Abdul Nasir, Waheb A Jabbar, Mukarram AM Almuhaya, Anupam Kumar Bairagi, Md Al-Masrur Khan, and Seong-Hoon Kee. 2022. IoT-Based Healthcare-Monitoring System towards Improving Quality of Life: A Review. In *Healthcare*, Vol. 10. MDPI, 1993.

[3] Adrian Bartnik, Bonaventura Del Monte, Tilmann Rabl, and Volker Markl. 2019. On-the-fly reconfiguration of query plans for stateful stream processing engines. *BTW 2019* (2019).

[4] Sebastian Baunsgaard, Matthias Boehm, Ankit Chaudhary, Behrouz Derakhshan, Stefan Geißelsöder, Philipp M. Grulich, Michael Hildebrand, Kevin Innerebner, Volker Markl, Claus Neubauer, Sarah Osterburg, Olga Ovcharenko, Sergey Redyuk, Tobias Rieger, Alireza Rezaei Mahdiraji, Sebastian Benjamin Wrede, and Steffen Zeuch. 2021. ExDRa: Exploratory Data Science on Federated Raw Data. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 2450–2463. https://doi.org/10.1145/3448016.3457549

[5] Boris Jan Bonfils and Philippe Bonnet. 2004. Adaptive and Decentralized Operator Placement for In-Network Query Processing. *Telecommun. Syst.* 26, 2-4 (2004), 389–409. https://doi.org/10.1023/B:TELS.0000029048.24942.65

[6] Valeria Cardellini, Francesco Lo Presti, Matteo Nardelli, and Gabriele Russo Russo. 2022. Runtime Adaptation of Data Stream Processing Systems: The State of the Art. *ACM Comput. Surv.* 54, 11s, Article 237 (sep 2022), 36 pages. https://doi.org/10.1145/3514496

[7] Xenofon Chatziliadis, Eleni Tzirita Zacharatou, Alphan Eracar, Steffen Zeuch, and Volker Markl. 2024. Efficient Placement of Decomposable Aggregation Functions for Stream Processing over Large Geo-Distributed Topologies. *Proc. VLDB Endow.* 17, 6 (2024), 1501–1514. https://www.vldb.org/pvldb/vol17/p1501-chatziliadis.pdf

[8] Ankit Chaudhary, Kaustubh Beedkar, Jeyhun Karimov, Felix Lang, Steffen Zeuch, and Volker Markl. 2025. Incremental Stream Query Placement in Massively Distributed and Volatile Infrastructures. In *41st IEEE International Conference on Data Engineering, ICDE 2025, Hong Kong SAR, China, May 19-23, 2025.* IEEE.

[9] Ankit Chaudhary, Jeyhun Karimov, Steffen Zeuch, and Volker Markl. 2023. Incremental Stream Query Merging. In *Proceedings of the 26th International Conference on Extending Database Technology, EDBT 2023, Ioannina, Greece, March 28-31, 2023.* OpenProceedings.org.

[10] Ankit Chaudhary, Steffen Zeuch, and Volker Markl. 2020. Governor: Operator Placement for a Unified Fog-Cloud Environment. In *Proceedings of the 23rd International Conference on Extending Database Technology, EDBT 2020, Copenhagen, Denmark, March 30 - April 02, 2020*, Angela Bonifati, Yongluan Zhou, Marcos Antonio Vaz Salles, Alexander Böhm, Dan Olteanu, George H. L. Fletcher, Arijit Khan, and Bin Yang (Eds.). OpenProceedings.org, 631–634. https://doi.org/10.5441/002/edbt.2020.81

[11] Ankit Chaudhary, Ninghong Zhu, Laura Mons, Steffen Zeuch, Varun Pandey, and Volker Markl. 2025. Incremental Stream Query Merging In Action. In *Datenbanksysteme für Business, Technologie und Web (BTW 2025)*. Gesellschaft für Informatik, Bonn, 907–915. https://doi.org/10.18420/BTW2025-58

[12] Pedro Cruz, Nadjib Achir, and Aline Carneiro Viana. 2022. On the Edge of the Deployment: A Survey on Multi-access Edge Computing. *ACM Comput. Surv.* 55, 5, Article 99 (Dec. 2022), 34 pages. https://doi.org/10.1145/3529758

[13] The Khang Dang, Nitinder Mohan, Lorenzo Corneo, Aleksandr Zavodovski, Jörg Ott, and Jussi Kangasharju. 2021. Cloudy with a chance of short RTTs: analyzing cloud connectivity in the internet. In *IMC '21: ACM Internet Measurement Conference, Virtual Event, USA, November 2-4, 2021.* ACM, 62–79. https://doi.org/10.1145/3487552.3487854

[14] Andy Davis, Jay Parikh, and William E. Weihl. 2004. Edgecomputing: extending enterprise applications to the edge of the internet. In *Proceedings of the 13th international conference on World Wide Web - Alternate Track Papers & Posters, WWW 2004, New York, NY, USA, May 17-20, 2004.* ACM, 180–187. https://doi.org/10.1145/1013367.1013397

[15] Bonaventura Del Monte, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2020. Rhino: Efficient management of very large distributed state for stream processing engines. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2471–2486.

[16] Dominik Durner, Viktor Leis, and Thomas Neumann. 2023. Exploiting Cloud Object Storage for High-Performance Analytics. *Proc. VLDB Endow.* 16, 11 (2023), 2769–2782. https://doi.org/10.14778/3611479.3611486

[17] Fleetio. [n.d.]. Fleet Management IoT: Benefits & Steps to Enhance Efficiency. https://www.fleetio.com/blog/fleet-iot [Online; accessed 2025-07-29].

[18] Apache Flink. 2023. Flink Architecture | Apache Flink. https://nightlies.apache.org/flink/flink-docs-master/docs/concepts/flink-architecture/#task-slots-and-resources. (Accessed on 12/01/2023).

[19] Haralampos Gavriilidis, Kaustubh Beedkar, Matthias Boehm, and Volker Markl. 2025. Fast and Scalable Data Transfer Across Data Systems. *Proc. ACM Manag. Data* 3, 3, Article 157 (June 2025), 28 pages. https://doi.org/10.1145/3725294

[20] VBB Verkehrsverbund Berlin-Brandenburg GmbH. 2024. VBB timetable data via GTFS | Open data Berlin. https://daten.berlin.de/datensaetze/vbb-fahrplandaten-gtfs. (Accessed on 04/22/2024).

[21] Rong Gu, Han Yin, Weichang Zhong, Chunfeng Yuan, and Yihua Huang. 2022. Meces: Latency-efficient Rescaling via Prioritized State Migration for Stateful Distributed Stream Processing Systems. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 539–556. https://www.usenix.org/conference/atc22/presentation/gu-rong

[22] Moritz Hoffmann, Andrea Lattuada, Frank McSherry, Vasiliki Kalavri, John Liagouris, and Timothy Roscoe. 2019. Megaphone: Latency-conscious state migration for distributed streaming dataflows. *Proceedings of the VLDB Endowment* 12, 9 (2019), 1002–1015.

[23] Otonomo Inc. 2023. Use Cases for Connected Car Data Driver Services | Otonomo. https://otonomo.io/use-cases/. (Accessed on 01/31/2023).

[24] Redpanda Data Inc. 2025. The State of Streaming Data Report. https://www.redpanda.com/resources/state-of-streaming-data-report. (Accessed on 07/20/2025).

[25] DELOITTE INSIGHTS. 2022. Smart cities and digital health | Deloitte Insights. https://www2.deloitte.com/xe/en/insights/focus/smart-city/building-a-smart-city-with-smart-digital-health.html. (Accessed on 03/22/2023).

[26] Jeyhun Karimov, Tilmann Rabl, Asterios Katsifodimos, Roman Samarev, Henri Heiskanen, and Volker Markl. 2018. Benchmarking Distributed Stream Data Processing Systems. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. 1507–1518. https://doi.org/10.1109/ICDE.2018.00169

[27] Jeyhun Karimov, Tilmann Rabl, and Volker Markl. 2019. AJoin: ad-hoc stream joins at scale. *Proceedings of the VLDB Endowment* 13, 4 (2019), 435–448.

[28] Jeyhun Karimov, Tilmann Rabl, and Volker Markl. 2019. AStream: Ad-Hoc Shared Stream Processing. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 607–622. https://doi.org/10.1145/3299869.3319884

[29] Fiodar Kazhamiaka, Matei Zaharia, and Peter Bailis. 2021. Challenges and Opportunities for Autonomous Vehicle Query Systems.. In *CIDR*.

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

[31] Chang Liu, Jiaxing Zhang, Hucheng Zhou, Sean McDirmid, Zhenyu Guo, and Thomas Moscibroda. 2014. Automating Distributed Partial Aggregation. In *Proceedings of the ACM Symposium on Cloud Computing* (Seattle, WA, USA) *(SOCC '14)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/2670979.2670980

[32] Xuhao Luo, Shreesha G. Bhat, Jiyu Hu, Ramnatthan Alagappan, and Aishwarya Ganesan. 2024. LazyLog: A New Shared Log Abstraction for Low-Latency Applications. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles, SOSP 2024, Austin, TX, USA, November 4-6, 2024.* ACM, 296–312. https://doi.org/10.1145/3694715.3695983

[33] Luo Mai, Kai Zeng, Rahul Potharaju, Le Xu, Steve Suh, Shivaram Venkataraman, Paolo Costa, Terry Kim, Saravanan Muthukrishnan, Vamsi Kuppa, et al. 2018. Chi: A scalable and programmable control plane for distributed stream processing systems. *Proceedings of the VLDB Endowment* 11, 10 (2018), 1303–1316.

[34] Pritish Mishra, Nelson Bore, Brian Ramprasad, Myles Thiessen, Moshe Gabel, Alexandre Da Silva Veith, Oana Balmau, and Eyale De Lara. 2024. Falcon: Live Reconfiguration for Stateful Stream Processing on the Edge. In *2024 IEEE/ACM Symposium on Edge Computing (SEC)*. 234–248. https://doi.org/10.1109/SEC62691.2024.00026

[35] J Nogiec and K Trombly-Freytag. 2005. A dynamically reconfigurable data stream processing system. (2005).

[36] Dan O'Keeffe, Theodoros Salonidis, and Peter R. Pietzuch. 2018. Frontier: Resilient Edge Processing for the Internet of Things. *Proc. VLDB Endow.* 11, 10 (2018), 1178–1191. https://doi.org/10.14778/3231751.3231767

[37] OpenCellid. 2024. OpenCelliD - Largest Open Database of Cell Towers & Geolocation - by Unwired Labs. https://opencellid.org/#zoom=16&lat=37.77889&lon=-122.41942. (Accessed on 04/22/2024).

[38] Sumanaruban Rajadurai, Jeffrey Bosboom, Weng-Fai Wong, and Saman P. Amarasinghe. 2018. Gloss: Seamless Live Reconfiguration and Reoptimization of Stream Programs. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2018, Williamsburg, VA, USA, March 24-28, 2018*, Xipeng Shen, James Tuck, Ricardo Bianchini, and Vivek Sarkar (Eds.). ACM, 98–112. https://doi.org/10.1145/3173162.3173170

[39] Global Railway Review. 2023. DB introducing new display system to make travel more convenient. https://www.globalrailwayreview.com/news/140633/db-introducing-new-display-system-to-make-travel-more-convenient/. (Accessed on 07/19/2024).

[40] Zhitao Shen, Vikram Kumaran, Michael J Franklin, Sailesh Krishnamurthy, Amit Bhat, Madhu Kumar, Robert Lerche, and Kim Macpherson. 2015. CSA: Streaming Engine for Internet of Things. *IEEE Data Eng. Bull.* 38, 4 (2015), 39–50.

[41] Suresh Singh, Mike Woo, and C. S. Raghavendra. 1998. Power-Aware Routing in Mobile Ad Hoc Networks. In *Proceedings of the 4th Annual ACM/IEEE International Conference on Mobile Computing and Networking* (Dallas, Texas, USA) *(MobiCom*

'98). Association for Computing Machinery, New York, NY, USA, 181–190. https://doi.org/10.1145/288235.288286

[42] Felix Sterk, David Dann, and Christof Weinhardt. 2022. Monetizing Car Data: A Literature Review on Data-Driven Business Models in the Connected Car Domain.. In *HICSS*. 1–10.

[43] NebulaStream Team. 2025. This repository contains code to produce a collection of topology changes generated by mobile devices. https://github.com/nebulastream/topology-change-generator [Online; accessed 2025-04-01].

[44] NebulaStream Team. 2025. This repository contains code to simulate topology changes. https://github.com/nebulastream/topology-change-simulator [Online; accessed 2025-07-31].

[45] Peter A. Tucker, David Maier, Tim Sheard, and Leonidas Fegaras. 2003. Exploiting punctuation semantics in continuous data streams. *IEEE Transactions on Knowledge and Data Engineering* 15, 3 (2003), 555–568.

[46] Shuo Wang, Xing Zhang, Yan Zhang, Lin Wang, Juwo Yang, and Wenbo Wang. 2017. A Survey on Mobile Edge Networks: Convergence of Computing, Caching and Communications. *IEEE Access* 5 (2017), 6757–6779. https://doi.org/10.1109/ACCESS.2017.2685434

[47] Benjamin Warnke, Stefan Fischer, and Sven Groppe. 2023. Distributed SPARQL queries in collaboration with the routing protocol. In *Proceedings of the 27th International Database Engineered Applications Symposium* (Heraklion, Crete, Greece) *(IDEAS '23)*. Association for Computing Machinery, New York, NY, USA, 99–106. https://doi.org/10.1145/3589462.3589497

[48] Benjamin Warnke, Stefan Fischer, and Sven Groppe. 2023. Using machine learning and routing protocols for optimizing distributed sparql queries in collaboration. *Computers* 12, 10 (2023), 210.

[49] Randall T. Whitman, Bryan G. Marsh, Michael B. Park, and Erik G. Hoel. 2019. Distributed Spatial and Spatio-Temporal Join on Apache Spark. *ACM Trans. Spatial Algorithms Syst.* 5, 1, Article 6 (June 2019), 28 pages. https://doi.org/10.1145/3325135

[50] Yingjun Wu and Kian-Lee Tan. 2015. ChronoStream: Elastic stateful stream computation in the cloud. In *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*, Johannes Gehrke, Wolfgang Lehner, Kyuseok Shim, Sang Kyun Cha, and Guy M. Lohman (Eds.). IEEE Computer Society, 723–734. https://doi.org/10.1109/ICDE.2015.7113328

[51] Mengwei Xu, Zhe Fu, Xiao Ma, Li Zhang, Yanan Li, Feng Qian, Shangguang Wang, Ke Li, Jingyu Yang, and Xuanzhe Liu. 2021. From cloud to edge: a first look at public edge platforms. In *Proceedings of the 21st ACM Internet Measurement Conference* (Virtual Event) *(IMC '21)*. Association for Computing Machinery, New York, NY, USA, 37–53. https://doi.org/10.1145/3487552.3487815

[52] Steffen Zeuch, Ankit Chaudhary, Bonaventura Del Monte, Haralampos Gavriilidis, Dimitrios Giouroukis, Philipp M. Grulich, Sebastian Breß, Jonas Traub, and Volker Markl. 2020. The NebulaStream Platform for Data and Application Management in the Internet of Things. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2020/papers/p7-zeuch-cidr20.pdf

[53] Steffen Zeuch, Bonaventura Del Monte, Jeyhun Karimov, Clemens Lutz, Manuel Renz, Jonas Traub, Sebastian Breß, Tilmann Rabl, and Volker Markl. 2019. Analyzing efficient stream processing on modern hardware. *Proceedings of the VLDB Endowment* 12, 5 (2019), 516–530.

[54] Steffen Zeuch, Eleni Tzirita Zacharatou, Shuhao Zhang, Xenofon Chatziliadis, Ankit Chaudhary, Bonaventura Del Monte, Dimitrios Giouroukis, Philipp M. Grulich, Ariane Ziehn, and Volker Markl. 2020. NebulaStream: Complex Analytics Beyond the Cloud. *Open J. Internet Things* 6, 1 (2020), 66–81. https://www.ronpub.com/ojiot/OJIOT_2020v6i1n07_Zeuch.html

[55] Yali Zhu, Elke A. Rundensteiner, and George T. Heineman. 2004. Dynamic Plan Migration for Continuous Queries over Data Streams. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data* (Paris, France) *(SIGMOD '04)*. Association for Computing Machinery, New York, NY, USA, 431–442. https://doi.org/10.1145/1007568.1007617