Incremental Stream Query Placement in Massively Distributed and Volatile Infrastructures

Ankit Chaudhary[1,2], Kaustubh Beedkar[3], Jeyhun Karimov[4], Felix Lang[1], Steffen Zeuch[1,2], Volker Markl[1,2,5]
BIFOLD[1], TU Berlin[2], IIT Delhi[3], Ververica GmbH[4], DFKI GmbH[5]
firstname.lastname@(tu-berlin.de[1,2], cse.iitd.ac.in[3], ververica.com[4], dfki.de[5])

PREPRINT

***Abstract*—More and more data is produced outside the cloud by edge devices that provide basic processing capabilities. This trend enables a new class of data management systems that use both edge and cloud infrastructures for efficient data processing. Such systems push down operations by placing query operators close to the data-producing devices. A key challenge for these systems is handling the evolution of continuous queries and the dynamic changes in the infrastructure. In particular, frequent arrival or removal of queries and potential volatility of the infrastructure might invalidate or reduce the efficiency of previous operator placement decisions and thus might lead to constant, expensive re-optimizations of queries. These changes require new solutions for operator placement, which adjust existing placement decisions upon changes to the queries and infrastructure. In this paper, we propose `ISQP`, a framework that keeps the operator placements valid under query and infrastructure changes. `ISQP` performs a fine-grained identification of invalid operator placements and takes concurrent, incremental placement decisions to reduce the optimization time. `ISQP` works for arbitrary placement strategies, making it a general-purpose framework. Our evaluations show that `ISQP` reduces the optimization overhead by one order of magnitude compared to the baseline.**

***Index Terms*—IoT, Stream Processing, Operator Placement**

I. Introduction

Massively distributed applications, such as smart mobility, surveillance, delivery services, or traffic management, are gaining popularity due to their ability to improve quality of life [1], [2], [3], [4], [5], [6], [7]. These applications need low-latency [8], [9], [10], energy efficient [11], [12], [13], privacy-preserving computations [14], [11] over numerous data streams produced by geo-distributed and mobile devices outside the cloud data centers. However, the traditional approach of transferring data to robust and scalable cloud data centers for processing [15], [16] cannot satisfy these requirements due to its high latency, increased network resource consumption, and data movement outside permissible zones. These limitations have led to the emergence of a new class of DSPEs that operate over a unified *sensor-edge-cloud* infrastructure at a massive scale and manage the execution of thousands of queries by pushing down operations outside cloud data centers [17], [18], [19], [20].

In such systems, the *physical topology* of this large, heterogeneous, distributed infrastructure is typically modeled as a large graph of *device nodes* with different characteristics (e.g., different computational capabilities, storage, energy consumption) connected by *communication edges* with varying capacities (e.g., varying bandwidth, latency, energy consumption). The continuous queries running in the system can be modeled as a set of dataflow plans, where each plan consists of operator nodes denoting either data sources or operations on the data and edges denoting the flow of data between the operator nodes. We can merge dataflow plans with common operators and refer to this merged set of concurrently running plans as the *global query plan* (`GQP`) in the rest of the paper.

```
Q1:bus1.union
    (bus2).window().map(PassengerLoadUDF).sink()
Q2:bus1.union(bus2).window().map(ODMarixUDF).sink()
Q3:bus1.union(bus2).window().map(ODMarixUDF).sink()
Q4:bus1.union(bus2).window()
    .map(TelemetryUdf).map(PassengerLoadUdf).sink()
```

Listing 1: Queries serving smart mobility applications.

Operator placement is a crucial step during the query optimization in a DSPE that maps query operators to the physical topology nodes before their execution [21], [22]. DSPEs operating over a massively distributed edge-cloud infrastructure have to account for two unique characteristics: *the arrival or removal of stream queries* [23], [24], and *volatility in the underlying infrastructure* [17], [14], [25]. These characteristics generate frequent *change events* (`CE`s) that affect running queries (i.e., addition or removal of queries) and the topology (i.e., registration or de-registration of devices and their communication edges). Under these frequent `CE`s, the placement mappings of operators can be missing, invalid, or inefficient. As a result, `CE`s frequently interrupt, delay, or cause the running queries to fail. Throughout this paper, we use the term "invalid" to denote inefficient, missing, or invalid placement mappings, as the handling of all cases is identical. Next, we highlight the effect of `CE`s on a real-world application.

Smart City Platform: Figure 1(a) illustrates the topology of a distributed edge-cloud infrastructure in a smart city, consisting of heterogeneous, geo-distributed devices (for now, ignore the red arrow) [5], [26]. The topology includes two buses and a speed camera (N_1–N_3), which are connected via intermediate edge devices (N_4–N_8) to cloud devices (N_9 and N_{10}).[1] The public transport system in a smart city can generate data streams containing information on vehicle location, ticketing, types of objects entering or exiting, and performance parameters [1], [27], [28]. These data streams are processed to support real-time applications such as crowd management, adaptive scheduling, and breakage detection [29].

Initially, three queries—Q_1, Q_2, and Q_3 from Listing 1—were deployed to run smart mobility applications. Q_1 is a long-running query implemented by the engineering team to count people entering or exiting buses and to compute passenger load [30]. Queries Q_2 and

[1]In practice, such infrastructure contains thousands of mobile devices (e.g., buses, taxis, or trains) and fixed devices (e.g., cameras or weather stations).

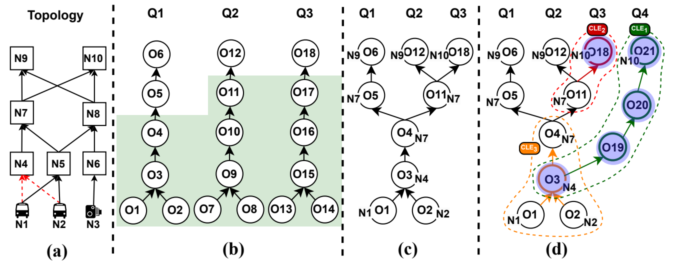

Fig. 1: (a) Example topology; (b) Example queries; (c) A global query plan with initial operator mapping; (d) Invalid mapping upon re-connection of mobile nodes N_1 and N_2, removal of query Q_3, and addition of query Q_4.

$\mathtt{Q_3}$ are short-running exploratory queries submitted simultaneously by the data analyst team to evaluate the performance of their origin-destination matrix algorithm capturing the flow of passengers [31]. Figure 1(b) illustrates the individual query plans of these three queries, which the DSPE merges into a `GQP`, as shown in Figure 1(c).[2] Each operator's label N_x indicates its execution locations.

During runtime, the DSPE concurrently (or within a short period) receives three `CE`s. $\mathtt{CE_1}$ adds a new short-running query $\mathtt{Q_4}$ for the duration of a public event, which analyzes telemetry data and passenger load of buses to detect traffic congestion, breakdown probability, and passenger load, thereby adjusting bus frequencies to prevent overcrowding adaptively [32], [33]. $\mathtt{CE_2}$ requests the removal of the short-running query $\mathtt{Q_3}$. $\mathtt{CE_3}$ is a node migration alert for the re-connection of buses, represented by $\mathtt{N_1}$ and $\mathtt{N_2}$, from $\mathtt{N_4}$ to $\mathtt{N_5}$.

Applying the above `CE`s to the `GQP` at runtime invalidates the operator mappings. Figure 1(d) shows the `GQP` (for now, ignore the dotted lines) after applying the `CE`s. Specifically, adding Q_4 results in operators O_{19}, O_{20}, and O_{21} lacking placement mappings (cf. green nodes). Removing query Q_3 results in operator O_{18} having an invalid mapping (cf. red node). The re-connection of nodes N_1 and N_2 results in operator O_3 being placed on a node without a connection to the upstream operators (cf. orange node).

A traditional approach [25] handles an invalid operator placement by performing `CE`s serially, one at a time. Since each `CE` can potentially affect many queries, the traditional approach requires removing and performing operator (re-)placement of affected queries, one query at a time. However, under frequent query and physical topology changes, this approach can quickly become a bottleneck, as the overall optimization time is influenced by the number of operators to be placed and the number of nodes in the topology [34], [35], [22]. Consequently, this approach delays the deployment of newly arriving queries and the resumption of interrupted queries, making the DSPE impractical for latency-sensitive applications.

An improvement over the traditional approach is to *jointly* handle a batch of `CE`s to (1) identify the operators affected by query or topology changes and (2) perform concurrent (re-)placement of the affected operators rather than the entire affected query plans. We refer to this improved approach as *incremental and concurrent stream query placement* (ISQP). We show in Sec. VII-B that `ISQP` reduces the optimization overhead by up to 23× and execution latency by up to 880× compared to the traditional approach and thus making DSPEs capable of operating in dynamic environments.

[2]For brevity, our example `GQP` comprises only one disconnected plan, but in actual there can be thousands of such plans.

ISQP, however, entails the following challenges: **C1: Identify invalid operator placement mappings among thousands of queries.** A DSPE can receive arbitrary `CE`s that affect operator placement of existing or new queries. Scaling `ISQP` across large topologies and query graphs under frequent `CE` changes is particularly challenging. **C2: Compute query deltas from affected operators to perform incremental amendments.** `CE`s can affect prior placement decisions for multiple queries and operators. Affected operators may be amended independently or require their placements to be performed together (dependent). Thus, identifying dependencies among affected operators to compute delta across all queries affected by `CE`s to perform incremental amendment remains challenging. **C3: Perform concurrent amendments to reduce overall optimization time.** Typically, DSPEs perform centralized, query-at-a-time placement [36], [37], [38], [17]. Looking jointly at all affected queries and providing concurrent amendments is a challenge but necessary for both high throughput, low latency processing of `CE`s and better performance of `ISQP`.

Existing research has studied the operator placement problem for geo-distributed environments [39], [35], [40], [34], [21], [41], [42]. These works apply cost-based, centralized, or decentralized methods to optimize operator placement for various goals. However, only a few approaches consider dynamicity [21], [41], [39] and only for changing data characteristics. An alternate approach to handle infrastructure dynamism is to replicate operators on multiple nodes [14]. However, this requires custom logic for de-duplicating processed data and consumes more network and energy resources. Overall, no existing work proposes a solution for amending an existing operator placement due to `CE`s in a massively distributed, heterogeneous, volatile environment.

`ISQP` maintains valid operator placement in a DSPE under high-frequency `CE`s with minimal optimization overhead. `ISQP` processes `CE`s in batches to identify invalid placements. First, it records affected placements by combining operator metadata with *change log entries*, which efficiently allows identifying impacted operators among thousands of queries (C1). Then, `ISQP` computes *query deltas* to manage placement dependencies across concurrent `CE`s, reducing operators and change log entries processed, enabling incremental amendments (C2). Finally, `ISQP` treats these amendments as transactions, using concurrency control to avoid conflicts in placement (C3).

In summary, after discussing preliminaries in Sec. II, we make the following major contributions:

- We present `ISQP`, a framework that keeps operator placement valid under continuous changes to concurrently running queries and physical topology (Sec. III).
- We propose the mechanism to process `CE`s and store affected operators as change logs for efficient identification (Sec. IV).
- We introduce query deltas that encapsulate non-overlapping change logs and enable incremental amendments (Sec. V).
- We present how concurrency control mechanisms can be applied for parallel processing of query deltas (Sec. VI).
- We present a detailed analysis to show that `ISQP` reduces optimization overhead up to a factor of 23.3 (Sec. VII).

PREPRINT

II. Preliminaries

a) Physical Topology: We denote the underlying physical infrastructure by a directed graph $T=(N,L)$ 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.

b) Global Query Plan: The Global Query Plan (GQP) is 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. The global query plan captures all queries concurrently running in the DSPE. If queries share common sub-expressions, their plans may have been connected in the global query graph [24]. However, queries that do not share common operators result in disconnected plans in the GQP. These disconnected plans can be processed independently from each other during ISQP.

As disconnected plans are quite common in real-world workloads and drastically impact efficiency, we explicitly model this in ISQP. Therefore, we define the GQP as the set $G=\{G_1\cup\dots\cup G_n\}$ of disconnected query plans (DQPs). Each G_i has its own disjoint set of operators O_i and disjoint data flow edges E_i, such that $\mathcal{O}=O_1\cup\dots\cup O_n$ and $\mathcal{E}=E_1\cup\dots\cup E_n$, and for all $i,j\in[1,...,n]$ $i\neq j$: $E_i\cap E_j=\emptyset$ and $O_i\cap O_j=\emptyset$.

Operator Placement Mapping: An operator placement $\mathscr{P}_{\mathcal{G},T}\subseteq\mathcal{O}\times N$ is a mapping relation that relates each operator of the GQP $\mathcal{G}$ to a (set of) node(s) of the physical topology T, and vice versa, i.e., for an operator $o\in\mathcal{O}$ of $\mathcal{G}$ and for a node $n\in N$ of T:[3]

$$\mathscr{P}_{\mathcal{G},T}(o)=\{n\,|\,n\in N \text{ and } (o,n)\in\mathscr{P}_{\mathcal{G},T}\}$$
$$\mathscr{P}^{-1}_{\mathcal{G},T}(n)=\{o\,|\,o\in\mathcal{O} \text{ and } (o,n)\in\mathscr{P}_{\mathcal{G},T}\}$$

Definition 1 (Valid Operator Placement Mapping): Given a GQP $\mathcal{G}$ and a physical topology T, we say that the mapping $\mathscr{P}_{\mathcal{G},T}$ is valid if and only if:

1) For each $o\in\mathcal{O}$ there exists $n\in N$ such that $(o,n)\in\mathscr{P}_{\mathcal{G},T}$.
2) For any two operators $o,o'\in\mathcal{O}$ and nodes $n,n'\in N$, the mappings $(o,n),(o',n')\in\mathscr{P}_{\mathcal{G},T}$ if $o\rightarrow o'\in\mathcal{E}$ and either $n\rightarrow n'\in L$ or $n=n'$.
3) If $o\notin\mathcal{O}$ then $(o,n)\notin\mathscr{P}_{\mathcal{G},T}$ for each $n\in N$.
4) If $n\notin N$ then $(o,n)\notin\mathscr{P}_{\mathcal{G},T}$ for each $o\in\mathcal{O}$.

The above conditions ensure that operator placement mapping is both complete and sound. In particular, conditions (1) and (2) ensure that every operator in the GQP must be mapped to a node in the physical topology and that the mapping respects the topology of both GQP and T. Condition (3) ensures that inactive query operators are not mapped to any node in T, and (4) ensures that no query operator is mapped to an inactive topology node (e.g., due to failure). For example, Figure 1(c) shows a valid operator mapping for the example GQP where we label operators as Nx to denote their topology node mappings.

DSPEs must continuously handle external events by adding or removing queries in the GQP and registering or de-registering links and nodes in the underlying volatile infrastructure T. We refer to the former as *query change events* ($\mathtt{CE_Q}$) and the latter as *topology change events* ($\mathtt{CE_T}$). Let $\Delta\mathcal{G}$ and ΔT denote the change in GQP and topology due to CEs. These events, consequently, render the placement mapping $\mathscr{P}_{\mathcal{G}+\Delta\mathcal{G},T+\Delta T}$ invalid. For example,

[3]wlog, we assume that $|\mathscr{P}_{\mathcal{G},T}(o)|=1$.

Symbol/Acronym	Description
GQP	Global query plan
DQP	Disjoint query plan
S(o)	State of an operator
$\mathtt{CE_{Q+}}$, $\mathtt{CE_{Q-}}$, $\mathtt{CE_{L+}}$, $\mathtt{CE_{L-}}$, $\mathtt{CE_{N+}}$, $\mathtt{CE_{N-}}$, and $\mathtt{CE_P}$	Query addition, query removal, link addition, link removal, node addition, node removal, and placement change events resp.
CLE	Change log entry
ΔQ	Subgraph of DQP containing operators with invalid placements

TABLE I: Table of symbols used in the paper.

in Figure 1(d), adding a new query violates condition (1) (cf. green nodes), removing query Q_3 violates condition (3) (cf. red node), and the re-connection of nodes N_1 and N_2 violates condition (2) (cf. orange node).

Existing approaches in DSPEs handle invalid mapping by holistically optimizing GQP using cost-based or heuristics-based operator placement algorithms (OP), i.e., when GQP and topology changes from $\mathcal{G}$ and T to $\mathcal{G}+\Delta\mathcal{G}$ and $T+\Delta T$, they compute $\mathscr{P}_{\mathcal{G}+\Delta\mathcal{G},T+\Delta T}=\mathtt{OP}(\mathcal{G}+\Delta\mathcal{G},T+\Delta T)$. This results in significant optimization overhead and affects the latency of currently deployed queries. In this paper, we seek to incrementally compute $\mathscr{P}_{\mathcal{G}+\Delta\mathcal{G},T+\Delta T}$ as $\mathscr{P}_{\mathcal{G},T}+\mathtt{OP}(\Delta\mathcal{G},\Delta T)$ under continuous query and infrastructure changes.

Discussion: Many valid mappings could exist for a given $\mathcal{G}$ and T. We seek to incrementally find a mapping that minimizes the GQP (re)optimization time. This paper focuses on jointly handling concurrent changes to the GQP and topology. We incrementally and concurrently compute a valid mapping while allowing plugging in both cost-based and heuristics-based operator placement algorithms.

III. System Overview

This section gives an overview of ISQP, which allows incremental computation of the invalid operator placement mapping $\mathscr{P}_{\mathcal{G},T}$. Figure 2 shows ISQP's internal components (inside the green box) and their integration within a DSPE.

The DSPE uses the query interface 1 to receive requests to run new or remove existing queries ($\mathtt{CE_Q}$ events) and a monitoring component 2 to receive requests to update the underlying infrastructure by adding or removing nodes or edges of the physical topology ($\mathtt{CE_T}$ events). These components use a queue to register continuously the received $\mathtt{CE_Q}$ and $\mathtt{CE_T}$ events. The DSPE uses a query optimizer to perform logical and physical rewrites to a query plan, performs common sub-expression identification to merge queries, and deploys the updated queries by distributing the operators on the physical topology.

ISQP consumes $\mathtt{CE_Q}$ and $\mathtt{CE_T}$ events jointly, i.e., in a batch-at-a-time fashion, to identify and fix operator placement mappings that become invalid due to consumed CEs. Note that the batches can be constructed based on a fixed number of change events, a given time interval, or a combination thereof. Finally, ISQP calls the undeployment 3 and the deployment 4 components to undeploy old or deploy newly placed operators, respectively. Next, we describe the components and workflow of ISQP.

ISQP consists of four main components: (1) Catalog; (2) Change Event Applicator; (3) Delta Computer; and (4) Operator Placement Amenders. The *catalog* maintains the physical

PREPRINT

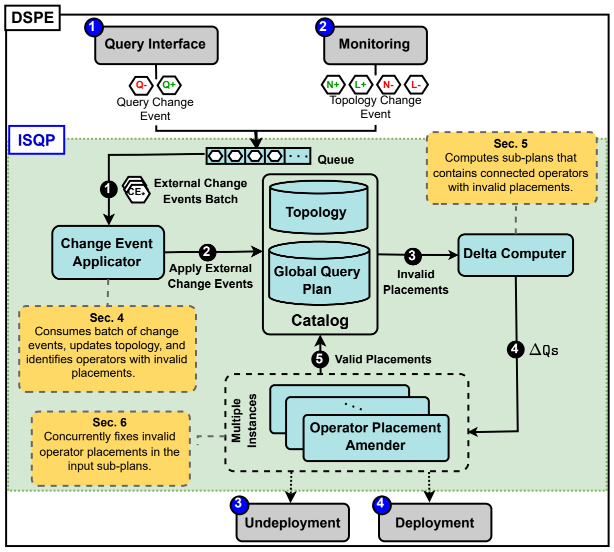
DSPE
1
Query Interface
2
Monitoring
Q-
Q+
Query Change Event
N+
L+
N-
L-
Topology Change Event
ISQP
Queue
Sec. 5
Computes sub-plans that contains connected operators with invalid placements.
1
CE*
External Change Events Batch
Topology
Change Event Applicator
2
Apply External Change Events
Global Query Plan
Catalog
3
Invalid Placements
Delta Computer
Sec. 4
Consumes batch of change events, updates topology, and identifies operators with invalid placements.
4
ΔQs
5
Valid Placements
Multiple Instances
Operator Placement Amender
Sec. 6
Concurrently fixes invalid operator placements in the input sub-plans.
3
Undeployment
4
Deployment

Fig. 2: System overview of ISQP.

topology T and the GQP. For each DQP in the GQP, we maintain a *change log* to record the operators for which any of conditions (1)–(4) of Definition 1 do not hold. These change logs allow ISQP to limit the number of operators to fix and, thus, the operations that need to be performed and allow incrementally computing $\mathscr{P}_{\mathcal{G},T}$. We present details about the internals of change logs in Sec. IV.

The *change event applicator* is responsible for updating the catalog and identifying the affected operators. First, it fetches a batch of CEs comprising of $\texttt{CE}_{\texttt{Q}}$ and $\texttt{CE}_{\texttt{T}}$ events (❶). Second, it applies these events to the topology graph T (to reflect the current state of the infrastructure) and then to the GQP (to add new or remove existing queries). During this process, the applicator records in change logs all operators whose placement mappings become invalid (❷). We present details about CEs and their impact on catalogs in Sec. IV-A and Sec. IV-B. Next, the delta computer computes the delta to fix the invalid placement mapping.

The *delta computer* computes sub-plans containing connected operators with invalid placements by analyzing the change logs (❸). These plan increments are called *query deltas* (ΔQs) in ISQP. A ΔQ allows ISQP to take fine-grained actions to fix only the invalid placement mappings instead of looking at the whole query plan, thus reducing the overall optimization time. Finally, the delta computer can compute multiple ΔQ that can be processed concurrently. We present details about the query deltas and how they are computed in Sec. V.

ISQP initiates a set number (configurable) of *operator placement amender* instances to concurrently process ΔQs (❹). Each instance fixes invalid placements within a query delta in two phases. First, it removes invalid placements to free resources (removal phase). Then, it applies the DSPE's operator placement algorithm [35], [22] to place operators of the query delta onto suitable nodes in the updated topology (addition phase). This approach lets ISQP (1) use existing placement algorithms without modification, (2) incorporate new algorithms seamlessly, and (3) integrate easily within a DSPE. During both phases, each instance updates node resources. Since multiple instances may access common topology nodes, ISQP employs concurrency control to ensure consistency, allowing safe, concurrent use of placement algorithms without modification. Details of the operator placement amender and concurrency control are provided in Sec. VI.

Once a ΔQ is processed, each placement amender instance updates the GQP in the catalog with the newly computed operator placement mappings for all affected operators (❺). Once all ΔQs get processed, the change event applicator repeats the process for the next batch of CEs. Note that each of the four components of ISQP performs the processing sequentially as they are interdependent. For example, the change event applicator updates the topology and the state of operators, while an operator placement amender instance reads this updated topology to fix invalid operator placements. This results in amenders producing valid operator placements based on up-to-date topology information (cf. Definition 1). Next, we detail the components of ISQP and our design decisions.

IV. Change Event Applicator

The change event applicator applies a batch of CEs to the topology and the GQP to identify invalid operator placements. These CEs can have varying effects on the operators of a DQP component of the GQP. ISQP uses a collection of metadata and a change log to record all affected operators. This section discusses the type of CEs, their effect on operator placements, the metadata used to mark these effects, and the change log used to collect affected operators (Sec IV-A). Finally, we present details on how the change event applicator populates the change logs for different CEs (Sec IV-B).

A. Change Events

Change events: We consider the following CEs: query addition ($\texttt{CE}_{\texttt{Q}+}$) created due to the arrival of new queries; query removal ($\texttt{CE}_{\texttt{Q}-}$) created due to the removal of a running query; node addition ($\texttt{CE}_{\texttt{N}+}$) created due to registration of a new worker node; node removal ($\texttt{CE}_{\texttt{N}-}$) created due to failure or scheduled maintenance of a worker node; link addition ($\texttt{CE}_{\texttt{L}+}$) created due to re-connection of a mobile worker node; and link removal ($\texttt{CE}_{\texttt{L}-}$) created due to the disconnection of mobile worker node. Both $\texttt{CE}_{\texttt{N}+}$ and $\texttt{CE}_{\texttt{L}+}$ do not affect existing operator placements but can present opportunities to improve the existing placement decisions. However, the goal of ISQP is to keep placement mapping valid; therefore, the applicator processes these events only to update the topology and ignores analyzing their effects on the DQPs.

Metadata: CEs have varying effects on operator placements. Therefore, it is necessary to record how an operator is affected and the action required to bring the placements of the affected operator to a valid state. For each operator o, we associate $\mathscr{P}_{\mathcal{G},T}(o)$ as the physical node in which the operator is placed, and *state* $S(o)$ that indicates the current state of the operator. Note that for a newly added operator o, $\mathscr{P}_{\mathcal{G},T}(o) = \emptyset$. The current state of an operator allows for taking appropriate actions to mitigate invalid placements. An operator can be in one of the following states during its lifetime: *ToBePlaced*, *Placed*, *ToBeRePlaced*, *ToBeRemoved*, or *Removed*. Figure 3 shows how the state of an operator changes due to various CEs. Note that $\texttt{CE}_{\texttt{P}}$ event is generated by ISQP after mitigating all invalid placements (cf. Sec. VI). Next, we explain how ISQP records affected operators.

PREPRINT

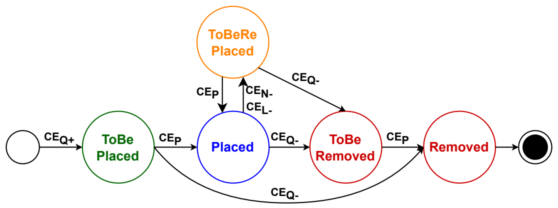

Fig. 3: Operator state transition diagram.

Change Logs: The GQP of a DSPE can contain thousands of DQPs collectively serving hundreds of thousands of merged stream queries [24], [43]. Each DQP contains hundreds of operators collectively serving all merged continuous queries. After applying a change event, ISQP uses *change logs* to capture the operators with invalid placement mappings (cf. Definition 1), i.e., $S(o) \neq$ *Placed* $\vee S(o) \neq$ *Removed*. As the change event applicator can apply several CEs, a change log contains several indexed *change log entries* (CLEs).

We define a CLE as a sub-graph of a DQP containing operators with invalid mappings. In particular, for each operator o in a CLE, the following properties hold at the time of CLE creation[4]: (1) The root operators are either of type sink or are in the state *Placed*; (2) The leaf operators are either of type source or are in the state *Placed*; (3) All operators between leaf and root operators are in the state *ToBePlaced*, *ToBeRePlaced*, or *ToBeRemoved*. For example, Figure 1(d) shows three change log entries (CLE_1 marked with green dashed lines, CLE_2 marked with red dashed lines, and CLE_3 marked with orange dashed lines) after applying CE_{Q+}, CE_{Q-}, and CE_{N-} respectively.

B. Processing Change Events

We present how CLEs are computed for different types of CEs. The change event applicator processes a batch of CEs to update topology and the GQP. This application may result in DQP components of the GQP having operators with invalid placement mappings. The applicator uses operator metadata and CLEs to capture all such operators.

Processing CE_{Q+} event: Upon receiving a CE_{Q+} event, the change event applicator calls the optimizer to find a DQP that shares operators with the input query Q. We use our previous work for sharing identification and refer the authors to the paper for details [24]. If a DQP is found, Q is merged into the DQP by connecting all not shared operators of Q to the shared operator of DQP. The state of only the newly added operators is set to *ToBePlaced* while all shared operators in the DQP remain unchanged as their placements are unaffected. These newly added operators form the sub-graph recorded in the change log of DQP. For example, CLE_1 formed after adding query Q4 in Figure 1(d). If the optimizer finds no DQP that shares operators with the input query, the applicator creates a new DQP. All operators' states in the new DQP are set to *ToBePlaced*. The whole DQP represents the sub-graph with invalid placements and forms the CLE entry.

Processing CLE_{Q-} event: Upon receiving a CLE_{Q-} event, the applicator fetches the DQP hosting the query. Sink operators serving the removed query within the DQP are stored in a downstream operator set (O_{down}) to compute the affected sub-plan. The applicator changes the state of O_{down} operators to *ToBeRemoved* following the state transition diagram in Figure 3. The applicator then recursively iterates over connected upstream operators of all O_{down} operators to mark their state as *ToBeRemoved*. The recursion stops at an operator o where not all downstream operators are in the state *ToBeRemoved*. This indicates that o is shared by other queries and not only serving the removed query. At this point, the operator o is stored in the upstream operator set (O_{up}). The O_{down} and O_{up} are then used to create a new CLE representing the affected sub-graph. For example, CLE_2 in Figure 1(d) represents the sub-graph affected by the removal of the query Q3. Note that the sub-graph stops at operator O6 as it is also serving query Q2.

Processing CE_{L-} event: A CE_{L-} event, apart from impacting the topology, also impacts running DQPs. The applicator updates the topology by removing the link provided in the change event. The applicator then identifies DQPs using the removed link by analyzing the operators placed on the source (N_{Src}) and destination (N_{Dest}) nodes connected by the removed link. If a DQP has operators placed only on one or none of the nodes, the link connecting the two nodes was not used by the DQP. Thus, such DQPs are ignored for further processing. Next, the applicator iterates over the DQPs to find the sub-graph impacted by the link removal. In particular, first, it finds the most upstream operator placed on the N_{Dest} and assigns it to the downstream operator set (O_{down}). The operator in O_{down} represents the most downstream operator unaffected by the link removal. Second, the algorithm finds the most downstream operator placed on the N_{Src} and assigns it to the upstream operator set (O_{up}). The operator in O_{up} represents the most upstream operator unaffected by the link removal. Third, the state of all operators between O_{up} and O_{up} is changed to ToBeRePlaced. Finally, the O_{down} and O_{up} are used to create a new CLE in the DQP.

Processing CE_{N-} event: The applicator treats CE_{N-} as multiple link removals, which allows the applicator to reuse the process used for CE_{L-} event. For example, the node N6 removal in Figure 1(a) is treated as two CE_{L-}s, i.e., (N4, N6) and (N6, N8). After processing the change event, the applicator creates CLE_3. We omit specific details due to space constraints.

V. Delta Computer

Delta computer initiates concurrent mitigation of placements for all collected CLEs. However, CLEs may develop dependencies among each other due to different CEs applied at different times (Sec. V-A). These dependencies prevent concurrent mitigation of CLEs. To address these dependencies, delta computer computes *query deltas* (ΔQs) (Sec. V-B) after analyzing CLEs for dependencies and merging all dependent CLEs (Sec. V-C).

A. Dependencies Among Change Log Entries

ISQP can concurrently process CLEs to mitigate invalid placement mappings. Figure 4(1) shows CLEs of the example GQP from Figure 1(d). However, in some instances, CLEs can create dependency among each other. For example, CLE_1 depends on CLE_3 due to the operator O3. In particular, CLE_1 uses O3 as its pinned upstream operators while CLE_3 marks O3 for re-operator placement. In the processing of CLE_1, the operators are placed such

[4]subsequent CEs may result in violating these properties for existing CLEs.

PREPRINT

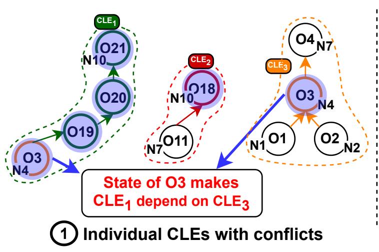

Fig. 4: ① Example change log entries from Figure 1(d), ② Query deltas ($\Delta\mathtt{Q}_1$ and $\Delta\mathtt{Q}_2$) computed by delta computer after merging dependent `CLEs`.

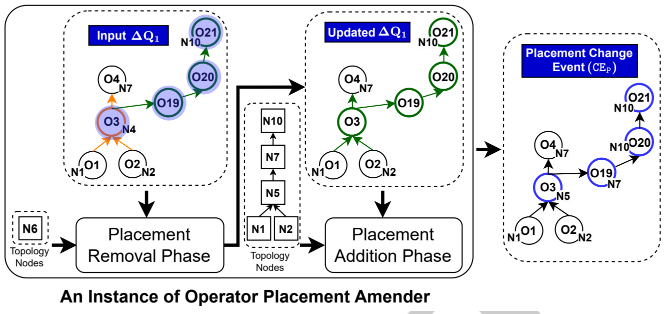

Fig. 5: Expanded view of amender processing $\Delta\mathtt{Q}_1$ from Figure 4. The placement removal phase removes `O4` from `N6`. The placement addition phase adds places `O4`, `O10`, `O11`, and `O12` as highlighted by the blue circles.

that the data from operator `O3` pinned on node `N4` can reach via operators `O19` and `O20` to the operator `O21` pinned on node `N10`. While in the processing of $\mathtt{CLE}_3$, the operator `O3` is replaced from node `N4` to another appropriate node such that the data from operator `O1` pinned on node `N1` and `O2` pinned on node `N2` can reach via `O3` to the operators `O4` on node `N7`. Replacing `O3` to a new node invalidates placement decisions taken while processing `CLE1`. This not only prevents concurrent processing of both `CLEs` but also makes the processing of $\mathtt{CLE}_1$ depend on $\mathtt{CLE}_3$. Thus, concurrent processing of dependent `CLEs` results in inconsistencies. Therefore, we merge all dependent `CLEs` to process them together. This prevents the overhead of finding the right order of dependencies and processing among CLEs.

B. Query Delta

We define a *query delta* ($\Delta\mathtt{Q}$) as a sub-graph of a `DQP` containing operators with placement mapping affected by one or more change events. In particular, for each operator `o` in $\Delta\mathtt{Q}$, the following properties hold: (1) The root operators are either of type sink or are in the state *Placed*. (2) The leaf operators are either of type source or are in the state *Placed*. (3) All operators between leaf and root operators are in the state *"ToBePlaced"*,*"ToBeRePlaced"*, or *"ToBeRemoved"*. These properties allow the root and leaf operators of a $\Delta\mathtt{Q}$ to act as the pinned operators[5]. Thus, the operator amender component needs to fix the placements of only the operators between the root and leaf nodes.

The delta computer generates $\Delta\mathtt{Q}$s to address the issues arising from the dependencies among `CLEs`. To this end, all dependent `CLEs` are merged to compute a single unified $\Delta\mathtt{Q}$. For example, in Figure 4(2) $\Delta\mathtt{Q}_1$ was prepared by merging interdependent `CLE1` and `CLE3`. This results in (1.) all $\Delta\mathtt{Q}s$ become independent of each other, and (2.) enables concurrent processing of $\Delta\mathtt{Q}s$. Next, we present how the delta computer computes $\Delta\mathtt{Q}$s from a collection of `CLEs` before calling placement amender.

C. Delta Computer

The delta computer, first, analyzes `CLEs` of a `DQP` within the `GQP` to detect dependencies. Two or more `CLEs` are dependent if they share operators in the state *ToBePlaced*, *ToBeRemoved*, or *ToBeRePlaced*. For example in Figure 4(1), `CLE1` and `CLE3` share the operator `O4` in state *ToBeRePlaced*. While `CLE2` and `CLE3` also share the operator `O6` it is in the state `Placed`. Thus, `CLE1` and `CLE2` are in conflict but `CLE2` and `CLE3` not. Dependent `CLEs` are merged by unioning their edges and vertices. The delta computer repeats the conflict identification and merging process until all `CLEs` become independent and satisfy the properties described in Sec. V-B. Afterward, the delta computer treats all `CLEs` as $\Delta\mathtt{Q}$s. For example, in Figure 4(2), the dependent `CLE1` and `CLE3` are merged together to represent $\Delta\mathtt{Q}_1$.

[5]Operators that are to be placed on a pre-defined node.

The delta computer uses a configurable thread pool to process $\Delta\mathtt{Q}$s concurrently. For each $\Delta\mathtt{Q}$, it invokes an instance of operator placement amender and collects the placement change events post-completion. The delta computer continues this process till all $\Delta\mathtt{Q}$s are processed. Finally, the delta computer returns the batch of collected placement change events to the change event applicator to fix all invalid placements. Next, we present details about the operator placement amender and the placement change event.

VI. Operator Placement Amender

The amender is responsible for amending the invalid placement mappings captured by a $\Delta\mathtt{Q}$. Depending on the operator state, the placement amender removes the mappings, finds new mappings, or does both. Thus, the amender processes a $\Delta\mathtt{Q}$ in two phases: placement removal and placement addition (Sec. VI-A). Multiple amenders can perform concurrent amendments of $\Delta\mathtt{Q}$s. However, this concurrent amendment may result in conflicts and an inconsistent topology state. To address this, `ISQP` supports pessimistic and optimistic placement amender strategies for concurrency control (Sec. VI-B). Finally, amenders return placement change events for fixing invalid placement mappings (Sec. VI-C)

A. Placement Amendment

Query deltas can contain operators in different states. For example, $\Delta\mathtt{Q}_1$ in Figure 4 contains some operators in the state ToBePlaced (`O10,O11,O12`) and some in ToBeRePlaced (`O4`). Operators in different states need different mitigation actions. A ToBeRemoved operator needs its placement mapping removed, a ToBePlaced operator needs a new placement mapping, and a ToBeRePlaced operator needs its placement mappings removed before an alternate placement mapping is found. Thus, we first remove all invalid placements before creating new placements to fix operators of a $\Delta\mathtt{Q}$. This not only cleans up all the stopped or paused operators due to query or node/link removal but also enables the reuse of resources such operators occupy. Additionally, placements can be removed and added incrementally, allowing placement amendment to access

PREPRINT

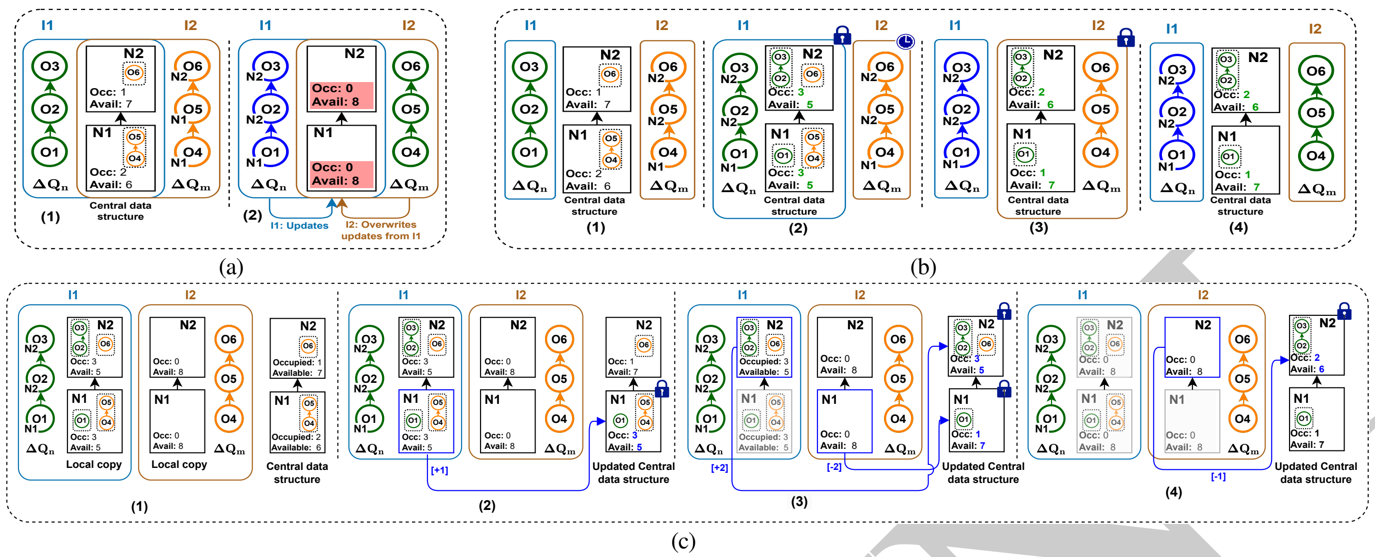
I1
I2
Central data structure
I1: Updates
I2: Overwrites updates from I1
Local copy
Updated Central data structure
(a)
(b)
(c)

Fig. 6: (a) Concurrent placement amendments: (1) Amender instances I1 and I2 concurrently processing ΔQ_m and ΔQ_n using central data structure. (2) Example inconsistent resources; (b) Pessimistic approach: (1) Instances I1 and I2, and common topology nodes. (2) I1 locks the topology nodes while I2 waits. (3) I1 finishes and I2 locks the topology nodes. (4) I1, I2, and the central data structure post-processing; (c) Optimistic approach: (1) I1 and I2 post-processing, the local copies, and central data structure containing topology resources. (2) I1 updates node N1. (3) I1 and I2 update nodes N2 and N1 respectively. (4) I2 updates node N2.

only limited topology nodes and progress in case successive placement addition step fails. Therefore, the amender processes ΔQs in two phases: *placement removal* and *placement addition*.

Placement Removal Phase removes operators in the states ToBeRemoved or ToBeRePlaced, freeing up resources. First, it analyzes the input ΔQ to identify topology nodes where these operators are placed. Second, it retrieves the identified nodes from the catalog. Upon completion, this phase (a) removes current operator mappings, (b) updates operator states from ToBeRemoved or ToBeRePlaced to ToBePlaced, and (c) updates topology nodes with released resources. For instance, Figure 5 illustrates how this phase processes $\Delta Q1$, identifies node `N6`, removes operator `O4`'s placement, changes its state to ToBePlaced (green), and updates `N6`'s resources. Additionally, this phase asynchronously signals the undeployment component to This phase is skipped if a ΔQ has no operators in these states.

Placement Addition Phase finds new placements for operators in the state ToBePlaced. First, this phase uses the pinned root and leaf operators in a ΔQ to select the topology path and identify topology nodes for the placement. For example, in Figure 5, the placement addition phase selects topology path connecting nodes `N4` with `N8` and `N10` (see Figure 1(a)). Second, it uses any configured centralized placement algorithm to find placement mappings for all operators in the state ToBePlaced. Upon successful processing, the addition phase (a.) adds mappings to the operators, (b.) updates the operator state from ToBePlaced to *Placed*, and (c.) updates the topology nodes with occupied resource information. The use of ΔQs instead of whole `DQP` enables `ISQP` to make incremental placement decisions to fix the placement mappings of the affected operators. This design decision allows `ISQP` to use existing placement algorithms without making any changes to their internal implementation. Figure 5 shows the output sub-graph with all newly placed operators shown by blue circles and their placement annotations. This output sub-graph represents the placement change event used to fix invalid or missing placements (Sec. VI-C). Finally, this phase makes asynchronous calls to the deployment component to add all ToBePlaced or ToBeRePlaced operators on the underlying infrastructure. Note that the placement addition phase is skipped for ΔQs that contain no operator in state ToBePlaced.

B. *Concurrent Placement Amendment: Challenges and Solutions*

Challenges: The delta computer invokes multiple instances of placement amenders to concurrently fix ΔQs with invalid placements. This concurrency allows `ISQP` to reduce the overall optimization time. However, DSPEs maintain a centralized data structure to store the number of resources occupied or available at various topology nodes to use them as appropriate placement locations [44], [45]. The placement amender instances concurrently update this central data structure to free (placement removal phase) or occupy (placement addition phase) resources from various topology nodes. This concurrent update can potentially leave the centralized data structure with inconsistent resource information. This inconsistent resource information adversely affects placement decisions as they either lead to over or under-provisioning of topology nodes.

In Figure 6a, we highlight the conflict between two concurrent amender instances (I1 and I2). Particularly, Figure 6a(1) shows I1 and I2 before processing ΔQ_n and ΔQ_m, respectively. For brevity, we keep the two ΔQ*s* simple and do not show their pinned operators. The central data structure shows the occupied and available slots [37] on the topology nodes `N1` and `N2`. We assume each operator takes one slot. The actual number of slots an operator occupies depends on the placement algorithm. I1 updates will be lost due to I2 updates in this example. Figure 6a(2) shows ΔQ_n, ΔQ_m, and the updated topology nodes after processing. The placement addition phase of I1 places ΔQ_n operators on `N1` and `N2`. At the same time, the placement removal phase of I2 removes the placements of ΔQ_m operators from `N1` and `N2`. Next, both instances update the topology nodes with their respective views of the available and occupied slots. This results in the lost updates from I1. Ultimately, this creates a view where nothing appears to be placed on `N1` and `N2`. `ISQP` uses one of the two strategies to prevent these inconsistencies from occurring: (1) pessimistic or (2) optimistic strategy.

Pessimistic Placement Amendment Strategy: The pessimistic strategy uses a two-phase locking strategy (similar to that used in database systems [46]) to prevent concurrent access to topology nodes. First, each instance goes through the growing phase to acquire fine-grained locks over the topology nodes before performing the addition or removal phase. Second, it processes the $\Delta\mathtt{Q}$ and accordingly updates the resources on the locked topology nodes. Finally, post-processing or failure, it goes through the shrinking phase, where the locks are released. Our strategy uses a "back-off and retry" mechanism to prevent deadlock and starvation and ensures a strict order of acquiring locks on the topology nodes [47], [48], [49].

Figure 6b(1) shows the pessimistic approach to processing the two conflicting instances (I1 and I2) from Figure 6a. First, I1 acquires locks on node `N1` to `N2` and, after processing $\Delta\mathtt{Q}_\mathtt{n}$, updates the resources on the locked topology nodes (Figure 6b(2)). While the processing of I1 continues, I2 waits and retries to acquire locks on `N1` and `N2`. Second, I1 releases the locks upon the completion of its processing. I2 acquires the locks on `N1` to `N2` in a strict order and processes $\Delta\mathtt{Q}_\mathtt{m}$ (Figure 6b(3)). Finally, the topology nodes `N1` to `N2` show a correct view that only the operators from $\Delta\mathtt{Q}_\mathtt{n}$ are placed on the topology nodes after the completion of I2 (Figure 6b(4)).

Optimistic Placement Amendment Strategy: The optimistic strategy allows operator placement instances to process $\Delta\mathtt{Q}$s without acquiring exclusive locks over the topology nodes. Similar strategies have been proposed in the literature for the problem of large-scale workload deployments [36], [38]. However, they do not consider incrementally performing placement removal or replacement, a key distinguishing feature of `ISQP`. The optimistic strategy uses an optimistic concurrency control protocol to improve concurrency and prevent inconsistencies. In our setup, we maintain a shared view of the topology data structure and provide the possibility to create local copies that are visible exclusively to an amender instance. In particular, each amender instance operates only on local copies of the topology nodes. Thus avoiding the need to acquire fine-grained locks on the shared data structure. However, post-processing, each instance performs an *optimistic validation* to update the shared data structure atomically.

In the optimistic validation, the local copies and shared view of the topology nodes are compared to verify if sufficient resources are available to complete an amendment phase. This mechanism prevents the probability of validation failure (resulting in lost work) due to the updation of the shared data structure after the start of the amendment phase. We thus call it optimistic validation, which differs from the validation performed in classical optimistic concurrency control protocols.

The optimistic validation, first, iterates over the topology nodes used in the placement. Second, it locks the entry in the central data structure for each iterated topology node. Third, it checks if the number of resources required to complete the amendment phase is available in the shared data structure entry. Note that this check will always pass for the placement removal phase. If the check passes, the validation phase updates the resources in the shared data structure, changes the placement mappings, updates the operators' states, and releases the lock. Otherwise, it terminates further processing by the amender instance and marks it partially successful if any prior topology node iterations were validated.

Figure 6c illustrates the optimistic operator amender strategy. Figure 6c(1) shows I1 and I2 with local copies of topology nodes `N1` and `N2` after processing $\Delta\mathtt{Q}_\mathtt{m}$ and $\Delta\mathtt{Q}_\mathtt{n}$. In addition, it shows the shared data structure containing the topology nodes `N1` and `N2`. I1 starts the validation before I2. In Figure 6c(2), I1 sends the slots (1) occupied on `N1` during the placement addition phase. I1 locks the node `N1` and performs the optimistic validation to check if sufficient slots are available. As the validation succeeds, I1 updates the slots of the shared data structure. In Figure 6c(3), I1 continues the validation phase for `N2` while I2 starts the validation phase for `N1`. Like for `N1`, I1 updates the slots for `N2` in the shared data structure after successful validation. After that, I1 completes its validation. However, I2, during the validation, finds the local and the shared view of the node `N1` different. It uses the released slots (-2) to compute the updated available (7) and occupied (1) slots for the shared copy of `N1`. As the number of updated available slots is non-negative (sufficient resources available), I2 updates the shared data structure for the node `N1`. Figure 6c(4) shows I2 perform validation for `N2`. Similar to `N1`, I2 finds the local and the shared view of `N2` different. I2 used the released slots (-1) to compute the updated occupied (2) and available (6) slots for `N2`. As the number of updated available slots is non-negative, I2 updates the slot values for `N2` in the shared view. After that, I2 also completes its validation.

C. Updating Invalid Placements

Placement amender instances produce placement change events $\mathtt{CE}_\mathtt{P}$) reflecting the fixed operator placement mappings. Similar to a $\Delta\mathtt{Q}$, a $\mathtt{CE}_\mathtt{P}$ represents a sub-graph of connected operators with updated placement information. Figure 5 shows an amender instance producing a $\mathtt{CE}_\mathtt{P}$ after processing $\Delta\mathtt{Q}_1$. If the instance fails or is only partially successful, the $\mathtt{CE}_\mathtt{P}$ contains either no or only partially updated placement information, respectively. This allows a $\mathtt{CE}_\mathtt{P}$ to communicate successful, partially successful, or failed processing of a $\Delta\mathtt{Q}$.

After fixing the invalid placements, each placement amender instance updates the new placements in the catalog (Sec. III). To this end, a placement amender instance iterates over the operators in the $\mathtt{CE}_\mathtt{P}$ and updates the metadata (operator mappings and state) of the corresponding operators in the `GQP`. This marks the completion of the external change event processing. Afterward, `ISQP` fetched the next batch of `CE`s for processing.

VII. Evaluation

We experimentally evaluate `ISQP` using two real-world datasets and emulated sensor-edge-cloud infrastructure. Specifically, we compared `ISQP` with the state-of-the-art operator placement approach, which performs holistic placement with one `CE` at a time model. We further consider another baseline with a batch-at-a-time model. We found that (i) `ISQP` reduces the optimization overhead by 11.2×; (ii) `ISQP` scales linearly as we increase the batch size

PREPRINT

and the number of amender instances; (iii) optimistic strategy outperforms the pessimistic strategy even under a high conflict ratio; and (iv)ISQP limits the number of re-configurations required to resume the interrupted query plans, which reduces the adverse effects on the processing latency as the infrastructure or workloads evolve.

A. Experimental Setup

Baselines: We implement ISQP and the two baselines in NebulaStream [17], a state-of-the-art DSPE. *(1) NES: Holistic operator placement with one CE at a time model:* This is the default behavior of the NebulaStream and other state-of-the-art DSPEs. This baseline first updates the topology (if applicable) and then performs holistic amendments of all affected DQPs, serially. We also extended the baseline to support CE$_{T}$ events. *(2) HSQP: Holistic stream query placement with a batch of CEs at-a-time model:* HSQP performs holistic and concurrent operator placements by first applying a batch of CEs to the topology and the GQP. Then, it concurrently performs holistic amendment of all affected DQPs in the GQP. HSQP allows us to evaluate the benefits of concurrent amendments.
Topology: We base our experiment on two emulated infrastructures: (1) FIT IoT test lab [50], an open access 3-layered sensor-edge-cloud infrastructure test bed used by the research community to conduct experiments [51], [52], [53] involving many small wireless sensor devices, and (2) a combination of OpenCelliD database [54] and the schedule information from a public transport company [55], we refer to this as *public transport use case* in the remaining of section. Both represent a realization of sensor-edge-cloud infrastructure using three layers: cloud devices, intermediate edge devices, and IoT devices.
Change Events: We use the stream query generator from [24] to generate query CEs. The generator produces equivalent stream queries. The optimizer of NebulaStream merges these new queries (CE$_{Q+}$ events) to update DQPs and produce operators with invalid placements. Additionally, we remove queries in fixed intervals to generate CE$_{Q-}$ events during our experiments. We also use synthetic data from our open-source tool that combines OpenCelliD and trains schedule information to generate topology CEs [56]. The tool selects a fixed area, time interval, train trajectories, and cell towers to generate a file containing train movements that represent CE$_{L-}$ and CE$_{L+}$ events.
System setup: We set up the system for each experiment by deploying initial queries (see below for the number of queries). This allows us to represent a DSPE with already running queries in which CEs invalidate the placement mapping. To conduct our experiments, we submit different types of CEs to invalidate the placements of deployed queries either by removing or adding queries or by removing or adding topology links (see Sec. IV).
Methodology: We run each experiment 7 times and report the average optimization time as the time between the arrival of the first CE and the processing of the last CE divided by the number CEs. We use the following parameters to compare the efficiency of NES, HSQP, and ISQP: *(1) Batch Size:* The number of CEs processed at-a-time. *(2) Number of Amender Instances:* The number of threads used for performing concurrent amendment. *(3) Conflict Ratio:* The ratio of CEs that affect operators placed over common topology nodes to the total number of CEs in a batch to be processed.

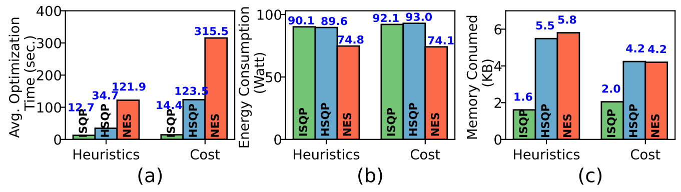

Fig. 7: Effect of incremental and concurrent amendments.

(4) Placement Strategy: The optimistic or pessimistic approach for performing concurrent operator placements (Sec. VI-B).

B. Experiments

This section presents a summary of our micro and macro experiments conducted to perform an in-depth analysis of ISQP. For all experiments (unless stated otherwise), we used a server with an AMD EPYC 7742 CPU and 1 TB of RAM. These experiments assess the impact of incremental and concurrent amendments, the scalability of our approaches, a comparison of pessimistic vs. optimistic amendment strategies, the impact of query CEs, the impact of topology CEs, and impact on the query execution latency (from Sec. VII-B1 to Sec. VII-B6 respectively).

*1) **Effect of Incremental and Concurrent Amendments**:* We measure average optimization time, energy, and memory consumption of ISQP, HSQP, and NES using different types of placement strategies.
Setup: We emulate a FIT IoT test lab infrastructure with 768 nodes (Sec. VII-A). For these experiments, we switched to servers with Intel® Xeon® Gold 6326 CPU and 1 TB of RAM, as the one with AMD do not have performance counters for CPU energy efficiency. The top (cloud) and intermediate (edge) layers contain 64 nodes each. All nodes in the top layer are connected to all nodes in the intermediate layers. The lower (sensor) layer contains 640 nodes representing IoT devices producing data. We divide nodes in the lower layer into groups of 10 nodes each. Each group is then connected to one of the 64 intermediate nodes. We configure HSQP and ISQP with batch size and amender instances of 8 and use only the optimistic amendment strategy. We initialize the system with 1024 queries. Each query consumes data from two distinct sensors, applies filters and maps, and unions both transformed streams before writing the stream to an output sink (located on a top-layer node). We select the number of queries as the multiple of 8, so the CE batches are full. We submit 3072 query addition events (CE$_{Q+}$) with 50% conflict ratio (representing the average case scenario) to the DSPE to perform our evaluations. We used only CE$_{Q+}$ events to prevent the influence of event types on the observations. We conduct this experiment by setting both heuristics and cost-based placement strategies.
Results: Figure 7(a), (b), and (c) shows the average optimization time, energy consumption, and memory consumption of different approaches for both heuristics and cost-base operator placement algorithms. First, ISQP shows a speed up in the optimization time by 2.7× and 8.5× compared to HSQP and 9.5× and 21.9× compared toNES when using heuristics- and cost-based placement algorithms, respectively. Second, the single-threaded NES approach consumed 20% less energy compared to multi-threaded ISQP and HSQP approaches. Third, the memory consumption of ISQP is 3.4× and 2.1× less compared to up to ISQP

PREPRINT

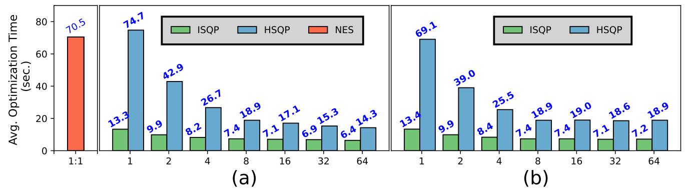

Fig. 8: Scalability w.r.t. (a) batch size and (b) amender instances.

and HSQP approaches when using heuristics- and cost-based placement algorithms, respectively.

Discussion: We show that ISQP achieves the best optimization time and memory consumption respective of the operator placement algorithm. This is due to two reasons. First, ISQP identifies and concurrently amends the placement of only the operators affected by the CEs. This significantly reduces the number of operators that need to be amended and parallelizes the amendment process. In particular, ISQP processes ΔQs containing overall 6144 operators while both HSQP and NES perform placement amendments of ΔQs containing 271872 operators. Second, due to the reduced number of operators, the memory footprint of ISQP also reduces. However, as ISQP uses 8 threads to perform concurrent placement amendments, it shows 20% higher energy consumption compared to NES. This increase can be seen in HSQP as well, which leverages concurrency for placement amendment.

2) ***Scalability:*** We investigate the scalability of ISQP, HSQP, and NES by varying the batch size and amender instances.

Setup: We use the topology setup from the previous experiment. We initialize the DSPE with 4096 queries and then submit another 4096 queries representing $\text{CE}_{\text{Q}+}$. We increased the $\text{CE}_{\text{Q}+}$s to evaluate a larger spectrum of batch size and thread count parameters. We fix the conflict ratio to 50% to represent the average case scenarios. We conducted two experiments: (1) We fixed the amender instances to 8 and varied the batch size from 1 to 64; (2) We fixed the batch size to 8 and varied the amender instances from 1 to 64.

Results: Figure 8(a) shows the average optimization time for NES. As this baseline only processes one event at a time, we fixed the amender instances and batch size to 1. Additionally, the figure shows the results of varying batch sizes for ISQP and HSQP. With increasing batch size, the optimization time reduces by up to 2.2× for ISQP and 5.3× for HSQP. However, no significant performance gain was observed when the batch size exceeded the thread count. Overall, ISQP outperforms NES and HSQP by a factor of 4.4×, even for a batch size of one.

Figure 8(b) shows the results for varying amender instances. Increasing amender instances reduces optimization overhead by 2.1× for ISQP and 5.1× for HSQP. However, no significant performance gains were observed as the thread count exceeded the batch size. Overall, ISQP outperforms NES and HSQP by a factor of 4.8×, even for a thread count of one.

Discussion: These experiments show that batch size and amender instances reduce the optimization overhead. First, a larger batch size allows processing more CEs together. In particular, batching allows identifying all operators with invalid placement mappings to compute ΔQs. Second, multiple amender instances allow concurrent processing of ΔQs. This concurrent amendment allows for quickly fixing all invalid placements, leading to a lower optimization time

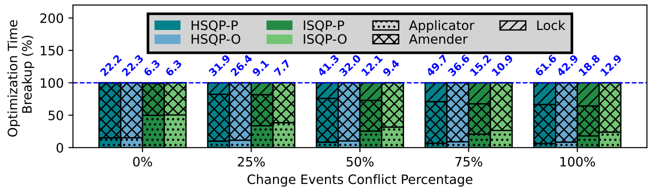

Fig. 9: Impact of pessimistic and optimistic approaches on optimization time with varying conflict ratios.

(Sec. VI-B). Third, ISQP requires a lower optimization overhead even with a small thread count in comparison to NES or HSQP. This is because ISQP operates only over operators with invalid placements instead of the entire query plan. This limits the number of operations the placement amendment component performs, resulting in a lower optimization overhead.

3) ***Pessimistic Vs. Optimistic Amendment Strategies:*** We investigate the impact of optimistic and pessimistic placement amender strategies on the optimization time of ISQP, HSQP, and NES.

Setup: We use the topology setup from experiment Sec. VII-B1. We configure HSQP and ISQP with batch size and amender instances of 8 and use both optimistic (O) and pessimistic (P) placement amendment strategies. We use a heuristic-based placement algorithm that places operators closer to the source [35]. We initialize the system by deploying 1024 queries. We conduct multiple experiments by submitting 3072 queries representing $\text{CE}_{\text{Q}+}$s with varying conflict ratios. Increasing the conflict ratio leads to the usage of common topology nodes for operator placement.

Results: Figure 9 shows the percentage breakdown (change event applicator, lock acquisition, and placement amendment) of the optimization time spent by ISQP and HSQP. First, we observe that the optimization time increases by $2\times - 3\times$ with an increase in the conflict ratio to 100% for both HSQP and ISQP irrespective of concurrency control mechanism (numbers above the bars). Second, we observe that the ratio of time spent by pessimistic approaches (HSQP-P and ISQP-P) for acquiring locks increases with the percentage of conflicts. For 100% conflicts, the time spent acquiring locks increases up to 30% of the overall optimization time. However, optimistic approaches only spend less than 1% of their time in the validation phase across different conflict ratios.

Additionally, we record the abort rate for pessimistic approaches (not shown due to limited space). Here, the abort rate is defined as the total percentage of attempts that failed to acquire locks successfully over topology nodes. Overall, we observe that as the percentage of conflicts among the query change events increases from 0% to 100%, the abort rate for ISQP increases from 0% to 77.6%. However, HSQP shows a constant, high abort rate of close to 99.9% except when the conflict ratio between query change events is 0%.

Discussion: The increase in the conflict ratio adversely affects the optimization time as the placement amender instances try to place affected DQPs on common topology nodes. Common topology nodes lead to contention during the lock acquisition and validation phase of pessimistic and optimistic approaches, respectively. In particular, optimistic approaches do not show a high lock acquisition ratio as they acquire lock only on a single topology node at the time of validation. On the other hand, pessimistic approaches show a very high lock acquisition ratio in the overall optimization time

PREPRINT

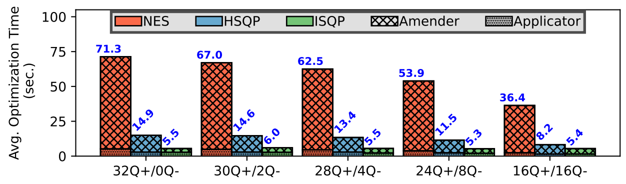

Fig. 10: Impact of changing ratio of $\mathtt{CE}_{\mathtt{Q}+}$ and $\mathtt{CE}_{\mathtt{Q}-}$ events.

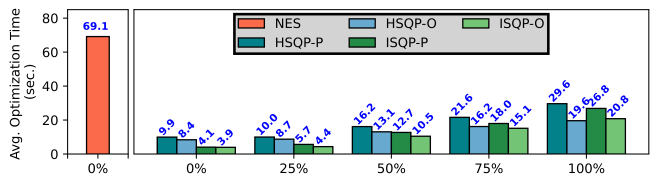

Fig. 11: Impact of varying conflicts for topology CEs.

due to a high abort rate. The main reason for this high abort rate is the use of the "back-off and retry" mechanism that tries to lock all involved nodes in one step (cf. Sec. VI-B). As the percentage of conflict increases, the number of common topology nodes (used in the concurrent processing of affected DQPs) also increases. Therefore, the number of aborts in HSQP (as it performs a holistic placement) is also $10^4\times$ to $10^5\times$ higher compared to ISQP.

4) ***Effect of Query Additions and Removals:*** We investigate the impact of processing a mix of query addition and removal change events on the optimization time of ISQP, HSQP, and NES.

Setup: The topology remains the same as in the previous experiments. We initialize the system with 4096 queries, with a 50% conflict ratio, and process 128 batches of CEs. Each batch contains 32 CEs (a combination of $\mathtt{CE}_{\mathtt{Q}+}$ and $\mathtt{CE}_{\mathtt{Q}-}$). In addition, we set the amender instances to 32 such that ISQP and HSQP can concurrently process all queries affected by the 32 CEs. Note that we chose a larger batch size to experiment with more combinations of $\mathtt{CE}_{\mathtt{Q}+}$ and $\mathtt{CE}_{\mathtt{Q}-}$. We vary the ratio of $\mathtt{CE}_{\mathtt{Q}+}$ and $\mathtt{CE}_{\mathtt{Q}-}$ events within a batch across different experiments and report the aggregated optimization time.

Results: Figure 10 shows the impact of varying the ratio of $\mathtt{CE}_{\mathtt{Q}+}$ and $\mathtt{CE}_{\mathtt{Q}-}$ events on the optimization overhead by different approaches. We observe that ISQP outperforms NES by a factor of $10\times$ and HSQP by a factor of $2\times$. Furthermore, the optimization time reduces as the number of $\mathtt{CE}_{\mathtt{Q}-}$ events increases.

Discussion: This experiment highlights two crucial aspects: First, ISQP incurs low optimization overhead even for a mix of $\mathtt{CE}_{\mathtt{Q}+}$ and $\mathtt{CE}_{\mathtt{Q}-}$ events. This reduced optimization overhead is due to ISQPs' incremental and concurrent processing. Second, increasing $\mathtt{CE}_{\mathtt{Q}-}$ events in a batch reduces overall optimization time because (1) the change event applicator takes less time as it does not perform the query optimization steps to process $\mathtt{CE}_{\mathtt{Q}-}$ events, and (2) the number of operators in the affected DQPs reduces.

5) ***Effect of Topology Change Events:*** We investigate the impact of topology change events on the optimization time.

Setup: We conduct this experiment by simulating an evolving three-layer infrastructure. The infrastructure includes 100 nodes representing suburban trains in the lower layer. Ten of these nodes connect to one of the 10 base stations in the intermediate layer. The top layer consists of 10 nodes (in a private data center) connected to all intermediate nodes.

As trains move, they disconnect from one base station and reconnect to another, generating topology CEs. Specifically, we simulate 4000 pairs of link removal ($\mathtt{CE}_{\mathtt{L}-}$) and addition ($\mathtt{CE}_{\mathtt{L}+}$) events, each representing the disconnection and reconnection of a suburban train from one base station to another during its journey. These pairs are ordered so that only one train moves from one base station to another at a time. For instance, the first 10 pairs of topology CEs depict trains disconnecting from each of the 10 base stations and reconnecting to the neighboring base station. After 100 pairs of such topology CEs, all trains initially covered by one base station at the start of the experiment reconnect to the neighboring base station.It's important to note that node removal ($\mathtt{CE}_{\mathtt{N}-}$) and addition ($\mathtt{CE}_{\mathtt{N}+}$) events can be represented as link removal and addition. Consequently, we conduct this experiment solely using $\mathtt{CE}_{\mathtt{L}-}$ and $\mathtt{CE}_{\mathtt{L}+}$ events.

We report the effect of conflict ratios on the overall optimization time. This allows us to mimic scenarios where topology changes might affect multiple queries using common topology nodes for placements. To this end, we initialize the DSPE with 4096 queries with varying conflict ratios for every experiment. We set the batch size to 10 to process 10 trains moving simultaneously from one base station to another. In addition, we further set amender instances to 10 so that all 10 affected DQPs deployed on each of the moving trains can be processed concurrently.

Results: Figure 11 shows the results of varying conflict ratios for topology CEs: ISQP outperforms NES by $14.8\times$ and HSQP by $2.4\times$. The optimistic approach works well in contrast to the pessimistic approach for both HSQP and ISQP. As the conflict ratio increases, the performance difference between ISQP and HSQP reduces.

Discussion: For a higher conflict ratio, we observe that ISQP takes similar, if not worse, time to optimize as HSQP. This high optimization time for ISQP is due to (1) more operators and topology nodes being involved during placement amendment and (2) the locking contention increases due to common nodes being involved. This is expected as ISQP relies on limited operators processed during placement amendment to improve optimization time.

6) ***Query Execution Latency:*** Lastly, we analyze the impact on running DQPs when a DSPE is subjected to CEs.

Setup: We set up a cluster of NebulaStream to run on a three-layered hierarchical infrastructure of a public transport company based on open-source schedule information and the OpenCelliD database. The bottom layer contains 20 nodes representing the suburban trains producing journey data, which are connected to 10 intermediate nodes representing base stations. The top layer contains only 1 node in the cloud data center, connected to all intermediate nodes. We initialize the system by deploying 100 queries that analyze journey data produced by the suburban trains. In particular, we deploy groups of 10 queries (forming a DQP), each consuming data from 2 distinct suburban trains. To experiment, we submit topology CEs to emulate the movement of 10 suburban trains per second for 60 seconds (1 suburban train from each base station). In particular, after deploying all DQPs, we submit topology CEs from the 20th to 80th second of the total experiment length. These movements interrupt the running DQPs and, thus, affect the processing time latency of in-flight tuples.

Results: Figure 12 shows the impact on the processing time latency

PREPRINT

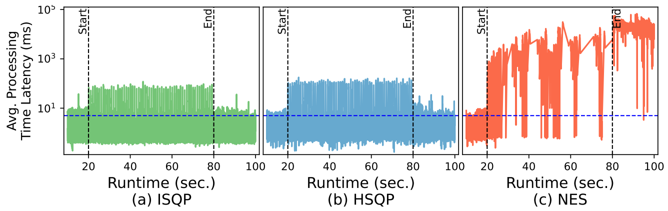

Fig. 12: Impact on the processing time latency.

of running queries. We report the observations from the 10th second of the experiment to discard the cold start of the queries. The maximum processing latency of the deployed DQPs reaches up to 5ms during normal operation (shown with the blue line). As topology CEs start to arrive, the ISQP leads to an average processing latency of 16.7ms (Figure 12(a)). These peaks can be observed every second immediately after concurrently mitigating the invalid operator placements. HSQP leads to an average processing latency of around 33.1ms (Figure 12(b)). However, baseline NES fails to complete the re-operator placement of affected DQPs between successive topology events CEs and leads to an average processing latency of 14701.5ms (Figure 12(c)). Overall, ISQP achieves up to 2× and 880.2× lower average processing latency compared to HSQP and NES, respectively.

Discussion: NES that holistically and serially processes affected queries fails to keep up with the rate of CEs. Thus, increasing the peak processing latency makes the DSPE unusable for latency-sensitive applications. In contrast, HSQP reduces the deployment latency compared to NES, as it performs holistic but concurrent placements of affected DQPs. However, ISQP reduces the deployment latency by 880.2× compared to baseline NES as it places and re-configures only the affected operators to resume interrupted DQPs. Overall, our approach enables latency-sensitive applications to continue operations even in the presence of dynamic infrastructures.

VIII. Related Work

Distributed Stream Processing Engines DSPEs allow the processing of millions of events in a continuous stream. The literature has proposed DSPEs for cloud (Flink [16], Storm [15], Spark [57]), edge (CSA [58], Frontier [59]), and unified edge-cloud (NebulaStream [17]) infrastructures. Unlike the cloud, the edge infrastructure consists of volatile and mobile devices undergoing frequent changes. ISQP focuses on handling these changes in a unified edge-cloud infrastructure by amending affected placement assignments. These amendments allow a DSPE to continue query execution even under frequent infrastructure changes.

Operator Placement: The operator placement problem, particularly for geo-distributed environments, has been well studied [22], [35], [40], [34], [42], [60], [61], [62]. Operator placement allows a DSPE to schedule the execution of queries to satisfy specific service-level goals [35]. These works apply a combination of heuristics, costs, machine learning, and centralized or decentralized methods to optimize operator placement for these goals. ISQP proposes a mechanism that allows easy integration of existing operator placement algorithms into DSPEs operating over edge-cloud infrastructures.

Handling Dynamism using Operator Placement: Some operator placement works have considered dynamism but only in the data characteristics [21], [41]. Luthra et al. [39] proposed a solution that transitions the running plan upon detecting the quality of service violation-based query performance. Another group of works [63], [64], [65], [66], [67], [68], [69], [70], [71] have proposed solutions for handling infrastructure changes by replicating and re-configuring operators on pre-defined nodes in a cloud-only infrastructure. To our knowledge, no existing work proposes a solution for handling invalid placement mapping due to query and infrastructure changes in a unified edge-cloud infrastructure.

Concurrency Control: This is a well-studied problem in database transactions [72], [73], [74]. Chen et al. [75] and Fuchs et al. [76] proposed novel pessimistic and optimistic protocols for handling concurrent updates on a graph database. ISQP treats placement amendments as transactions and, inspired by the above general concurrency control works, performs operator placement amendments concurrently. It leverages the characteristics of the operator placement process to modify optimistic concurrency protocol to define an optimistic validation step.

Large-Scale Workload Schedulers: Our work is also related to workload schedulers [77], [78]. Systems such as Mesos [45] and Hadoop-on-demand [44] were proposed to support concurrent scheduling of workloads by partitioning the cluster. Omega [36] uses an optimistic approach for concurrent workload scheduling in a shared-state environment. Borg [79] uses a similar scheduler to perform the placement of workloads concurrently. [38] extended Omega's approach to perform a fine-grain validation for conflict identification before dispatching the scheduled workloads. Like ISQP, these approaches have visibility over the entire cluster and support concurrent scheduling of the workloads. However, ISQP computes fine-grained query deltas with missing or invalid placements for scheduling. This allows ISQP to minimize the operator placements to amend and thus reduce the overall optimization overhead. In addition, ISQP also supports changes in the underlying infrastructure and fixes placements of affected queries.

IX. Conclusion

We proposed ISQP, a framework that allows DSPEs to keep valid operator placement mapping under continuous query and topology changes. ISQP uses the change event applicator to process external change events and compute change log entries to capture operators whose assignments are affected. These entries allow ISQP to easily identify the affected operators among thousands of queries. Next, using the delta computer, ISQP combines change log entries to reduce redundant operators and compute query deltas to perform incremental placement amendments. To this end, we proposed strategies to concurrent amendment strategies to speed up the optimization process. Our experimental evaluation showed that ISQP incurs 23.3× less optimization overhead and reduces the processing latency by 1093× compared to the baseline while keeping up with high-frequency queries and topology changes.

Acknowledgement. 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. 01IS18025A and 01IS18037A). We thank Dr. Varun Pandey for his valuable discussions and suggestions.

PREPRINT

PREPRINT

REFERENCES

[1] Xovis, "Public transport - automatic passenger counting — xovis," https://www.xovis.com/solutions/transportation, 2024, (Accessed on 07/26/2024).

[2] DataFromSky, "Deep traffic video analysis - datafromsky," https://datafromsky.com/, 2024, (Accessed on 07/19/2024).

[3] S. Abdulmalek, A. Nasir, W. A. Jabbar, M. A. Almuhaya, A. K. Bairagi, M. A.-M. Khan, and S.-H. Kee, "Iot-based healthcare-monitoring system towards improving quality of life: A review," in *Healthcare*, vol. 10, no. 10. MDPI, 2022, p. 1993.

[4] T. C. AB, "Smart public transport," (Accessed on 03/22/2023). [Online]. Available: https://business.teliacompany.com/internet-of-things/smart-public-transport

[5] M. of Digital Development and S. Information, "Smart nation singapore," https://www.smartnation.gov.sg/, (Accessed on 07/26/2024).

[6] D. INSIGHTS, "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, 2022, (Accessed on 03/22/2023).

[7] Y. Fu and C. Soman, "Real-time data infrastructure at uber," in *Proceedings of the 2021 International Conference on Management of Data*, ser. SIGMOD '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 2503–2516. [Online]. Available: https://doi.org/10.1145/3448016.3457552

[8] S. Wang, X. Zhang, Y. Zhang, L. Wang, J. Yang, and W. Wang, "A survey on mobile edge networks: Convergence of computing, caching and communications," *IEEE Access*, vol. 5, pp. 6757–6779, 2017.

[9] C. Avasalcai, B. Zarrin, and S. Dustdar, "Edgeflow—developing and deploying latency-sensitive iot edge applications," *IEEE Internet of Things Journal*, vol. 9, no. 5, pp. 3877–3888, 2022.

[10] W. Zhang, S. Li, L. Liu, Z. Jia, Y. Zhang, and D. Raychaudhuri, "Hetero-edge: Orchestration of real-time vision applications on heterogeneous edge clouds," in *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*. IEEE Press, 2019, p. 1270–1278. [Online]. Available: https://doi.org/10.1109/INFOCOM.2019.8737478

[11] C. Jiang, T. Fan, H. Gao, W. Shi, L. Liu, C. Cérin, and J. Wan, "Energy aware edge computing: A survey," *Computer Communications*, vol. 151, pp. 556–580, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S014036641930831X

[12] J. Hülsmann, J. Traub, and V. Markl, "Demand-based sensor data gathering with multi-query optimization," *Proc. VLDB Endow.*, vol. 13, no. 12, p. 2801–2804, Aug. 2020. [Online]. Available: https://doi.org/10.14778/3415478.3415479

[13] A. Vulimiri, C. Curino, P. B. Godfrey, T. Jungblut, K. Karanasos, J. Padhye, and G. Varghese, "Wanalytics: Geo-distributed analytics for a data intensive world," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 1087–1092. [Online]. Available: https://doi.org/10.1145/2723372.2735365

[14] D. O'Keeffe, T. Salonidis, and P. R. Pietzuch, "Frontier: Resilient edge processing for the internet of things," *Proc. VLDB Endow.*, vol. 11, no. 10, pp. 1178–1191, 2018. [Online]. Available: http://www.vldb.org/pvldb/vol11/p1178-okeeffe.pdf

[15] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy, "Storm@twitter," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 147–156. [Online]. Available: https://doi.org/10.1145/2588555.2595641

[16] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 2015.

[17] S. Zeuch, A. Chaudhary, B. D. Monte, H. Gavriilidis, D. Giouroukis, P. M. Grulich, S. Breß, J. Traub, and V. Markl, "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, 2020. [Online]. Available: http://cidrdb.org/cidr2020/papers/p7-zeuch-cidr20.pdf

[18] L. Tong, Y. Li, and W. Gao, "A hierarchical edge cloud architecture for mobile computing," in *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*, 2016, pp. 1–9.

[19] X. Chatziliadis, E. T. Zacharatou, A. Eracar, S. Zeuch, and V. Markl, "Efficient placement of decomposable aggregation functions for stream processing over large geo-distributed topologies," *Proc. VLDB Endow.*, vol. 17, no. 6, pp. 1501–1514, 2024. [Online]. Available: https://www.vldb.org/pvldb/vol17/p1501-chatziliadis.pdf

[20] G. Bartolomeo, M. Yosofie, S. Bäurle, O. Haluszczynski, N. Mohan, and J. Ott, "Oakestra: A lightweight hierarchical orchestration framework for edge computing," in *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. Boston, MA: USENIX Association, Jul. 2023, pp. 215–231. [Online]. Available: https://www.usenix.org/conference/atc23/presentation/bartolomeo

[21] B. J. Bonfils and P. Bonnet, "Adaptive and decentralized operator placement for in-network query processing," *Telecommun. Syst.*, vol. 26, no. 2-4, pp. 389–409, 2004. [Online]. Available: https://doi.org/10.1023/B:TELS.0000029048.24942.65

[22] V. Cardellini, F. Lo Presti, M. Nardelli, and G. R. Russo, "Runtime adaptation of data stream processing systems: The state of the art," *ACM Comput. Surv.*, vol. 54, no. 11s, sep 2022. [Online]. Available: https://doi.org/10.1145/3514496

[23] J. Karimov, T. Rabl, and V. Markl, "Astream: Ad-hoc shared stream processing," in *Proceedings of the 2019 International Conference on Management of Data*, ser. SIGMOD '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 607–622. [Online]. Available: https://doi.org/10.1145/3299869.3319884

[24] A. Chaudhary, J. Karimov, S. Zeuch, and V. Markl, "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, 2023.

[25] S. Zeuch, E. T. Zacharatou, S. Zhang, X. Chatziliadis, A. Chaudhary, B. D. Monte, D. Giouroukis, P. M. Grulich, A. Ziehn, and V. Markl, "Nebulastream: Complex analytics beyond the cloud," *Open J. Internet Things*, vol. 6, no. 1, pp. 66–81, 2020. [Online]. Available: https://www.ronpub.com/ojiot/OJIOT_2020v6i1n07_Zeuch.html

[26] A. Lepping, H. M. Pham, L. Mons, B. Rueb, P. M. Grulich, A. Chaudhary, S. Zeuch, and V. Markl, "Showcasing data management challenges for future iot applications with nebulastream," *Proc. VLDB Endow.*, vol. 16, no. 12, p. 3930–3933, aug 2023. [Online]. Available: https://doi.org/10.14778/3611540.3611588

[27] I. iris – intelligent sensing NA, "Passenger counting: The benefits of automated systems. — iris intelligent sensing," https://www.iris-sensing.com/us/products/automatic-passenger-counting/, 2024, (Accessed on 07/26/2024).

[28] D. GmbH, "Mobility," https://www.dilax.com/de/themen/mobility, 2023, (Accessed on 10/08/2023).

[29] G. R. Review, "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/, 2023, (Accessed on 07/19/2024).

[30] D. Darsena, G. Gelli, I. Iudice, and F. Verde, "Sensing technologies for crowd management, adaptation, and information dissemination in public transportation systems: A review," *IEEE Sensors Journal*, vol. 23, no. 1, pp. 68–87, 2023.

[31] S. Bera and K. Rao, "Estimation of origin-destination matrix from traffic counts: the state of the art," 2011.

[32] M. S. Kaiser, K. T. Lwin, M. Mahmud, D. Hajializadeh, T. Chaipimonplin, A. Sarhan, and M. A. Hossain, "Advances in crowd analysis for urban applications through urban event detection," *IEEE Transactions on Intelligent Transportation Systems*, vol. 19, no. 10, pp. 3092–3112, 2018.

[33] B. Zitung, "Narrow platforms: Passengers should only arrive shortly before departure," 2024, (Accessed on 08/03/2024). [Online]. Available: https://www.bz-berlin.de/berlin/mitte/bahnsteige-zu-schmal-fahrgaeste

[34] V. Cardellini, V. Grassi, F. L. Presti, and M. Nardelli, "Optimal operator placement for distributed stream processing applications," in *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems, DEBS '16, Irvine, CA, USA, June 20 - 24, 2016*, A. Gal, M. Weidlich, V. Kalogeraki, and N. Venkasubramanian, Eds. ACM, 2016, pp. 69–80. [Online]. Available: https://doi.org/10.1145/2933267.2933312

[35] A. Chaudhary, S. Zeuch, and V. Markl, "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*, A. Bonifati, Y. Zhou, M. A. V. Salles, A. Böhm, D. Olteanu, G. H. L. Fletcher, A. Khan, and B. Yang, Eds. OpenProceedings.org, 2020, pp. 631–634. [Online]. Available: https://doi.org/10.5441/002/edbt.2020.81

[36] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, "Omega: Flexible, scalable schedulers for large compute clusters," in *Proceedings of the 8th ACM European Conference on Computer Systems*, ser. EuroSys '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 351–364. [Online]. Available: https://doi.org/10.1145/2465351.2465386

[37] A. Flink, "Flink architecture — apache flink," 2023, (Accessed on 12/01/2023). [Online]. Available: https://nightlies.apache.org/flink/flink-docs-master/docs/concepts/flink-architecture/#task-slots-and-resources

[38] L. He, S. Yao, and W. Zhou, "An extended fine-grained conflict detection method for shared-state scheduling in large scale cluster," in *Proceedings of the 1st International Conference on Intelligent Information Processing*, ser. ICIIP '16. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: https://doi.org/10.1145/3028842.3028871

[39] M. Luthra, B. Koldehofe, P. Weisenburger, G. Salvaneschi, and R. Arif, "Tcep: Adapting to dynamic user environments by enabling transitions between operator placement mechanisms," in *Proceedings of the 12th ACM International Conference on Distributed and Event-Based Systems*, ser. DEBS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 136–147. [Online]. Available: https://doi.org/10.1145/3210284.3210292

[40] Y. Huang, Z. Luan, R. He, and D. Qian, "Operator placement with qos constraints for distributed stream processing," in *7th International Conference on Network and Service Management, CNSM 2011, Paris, France, October 24-28, 2011*. IEEE, 2011, pp. 1–7. [Online]. Available: https://ieeexplore.ieee.org/document/6103961/

[41] T. Karnagel, D. Habich, and W. Lehner, "Adaptive work placement for query processing on heterogeneous computing resources," *Proc. VLDB Endow.*, vol. 10, no. 7, pp. 733–744, 2017. [Online]. Available: http://www.vldb.org/pvldb/vol10/p733-karnagel.pdf

[42] A. D. S. Veith, M. D. de Assunção, and L. Lefèvre, "Latency-aware placement of data stream analytics on edge computing," in *Service-Oriented Computing - 16th International Conference, ICSOC 2018, Hangzhou, China, November 12-15, 2018, Proceedings*, ser. Lecture Notes in Computer Science, C. Pahl, M. Vukovic, J. Yin, and Q. Yu, Eds., vol. 11236. Springer, 2018, pp. 215–229. [Online]. Available: https://doi.org/10.1007/978-3-030-03596-9_14

[43] M. Vuppalapati, J. Miron, R. Agarwal, D. Truong, A. Motivala, and T. Cruanes, "Building an elastic query engine on disaggregated storage," in *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*, R. Bhagwan and G. Porter, Eds. USENIX Association, 2020, pp. 449–462. [Online]. Available: https://www.usenix.org/conference/nsdi20/presentation/vuppalapati

[44] Apache, "Hadoop on demand," 2007, (Accessed on 05/04/2024). [Online]. Available: https://svn.apache.org/repos/asf/hadoop/common/tags/release-0.17.1/docs/hod.html

[45] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: a platform for fine-grained resource sharing in the data center," in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'11. USA: USENIX Association, 2011, p. 295–308.

[46] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger, "The notions of consistency and predicate locks in a database system," *Commun. ACM*, vol. 19, no. 11, p. 624–633, nov 1976. [Online]. Available: https://doi.org/10.1145/360363.360369

[47] G. F. Coulouris, J. Dollimore, and T. Kindberg, *Distributed systems: concepts and design*. pearson education, 2005.

[48] S. T. Andrew, *Distributed Operating Systems*. Prentice-Hall International Edition, 1995.

[49] S. T. Andrew and B. Herbert, *Modern operating systems*. Pearson Education, 2015.

[50] C. Adjih, E. Baccelli, E. Fleury, G. Harter, N. Mitton, T. Noel, R. Pissard-Gibollet, F. Saint-Marcel, G. Schreiner, J. Vandaele, and T. Watteyne, "Fit iot-lab: A large scale open experimental iot testbed," in *2015 IEEE 2nd World Forum on Internet of Things (WF-IoT)*, 2015, pp. 459–464.

[51] A. R. Urke, Kure, and K. Øvsthus, "Autonomous flow-based tsch scheduling for heterogeneous traffic patterns: Challenges, design, simulation, and testbed evaluation," *IEEE Open Journal of the Communications Society*, vol. 4, pp. 2357–2372, 2023.

[52] ——, "Experimental evaluation of the layered flow-based autonomous tsch scheduler," *IEEE Access*, vol. 11, pp. 3970–3982, 2023.

[53] X. Chatziliadis, E. T. Zacharatou, A. Eracar, S. Zeuch, and V. Markl, "Efficient placement of decomposable aggregation functions for stream processing over large geo-distributed topologies," *Proc. VLDB Endow.*, vol. 17, no. 6, pp. 1501–1514, 2024. [Online]. Available: https://www.vldb.org/pvldb/vol17/p1501-chatziliadis.pdf

[54] OpenCellid, "Opencellid - largest open database of cell towers and geolocation - by unwired labs," https://opencellid.org/, 2024, (Accessed on 04/22/2024).

[55] V. V. B.-B. GmbH, "Vbb timetable data via gtfs — open data berlin," https://www.vbb.de/vbb-services/api-open-data/datensaetze/, 2024, (Accessed on 07/26/2024).

[56] NebulaStream, "nebulastream/topology-change-generator: This repository contains code to produce a collection of topology changes generated by mobile devices," (Accessed on 08/02/2024). [Online]. Available: https://github.com/nebulastream/topology-change-generator

[57] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets." *HotCloud*, vol. 10, no. 10-10, p. 95, 2010.

[58] Z. Shen, V. Kumaran, M. J. Franklin, S. Krishnamurthy, A. Bhat, M. Kumar, R. Lerche, and K. Macpherson, "Csa: Streaming engine for internet of things." *IEEE Data Eng. Bull.*, vol. 38, no. 4, pp. 39–50, 2015.

[59] D. O'Keeffe, T. Salonidis, and P. Pietzuch, "Frontier: Resilient edge processing for the internet of things," *Proceedings of the VLDB Endowment*, vol. 11, no. 10, pp. 1178–1191, 2018.

[60] P. Agnihotri, B. Koldehofe, C. Binnig, and M. Luthra, "Zero-shot cost models for parallel stream processing," in *Proceedings of the Sixth International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*, 2023, pp. 1–5.

[61] P. Agnihotri, B. Koldehofe, P. Stiegele, R. Heinrich, C. Binnig, and M. Luthra, "Zerotune: Learned zero-shot cost models for parallelism tuning in stream processing," 2024.

[62] R. Heinrich, C. Binnig, H. Kornmayer, and M. Luthra, "Costream: Learned cost models for operator placement in edge-cloud environments," *arXiv preprint arXiv:2403.08444*, 2024.

[63] L. Mai, K. Zeng, R. Potharaju, L. Xu, S. Suh, S. Venkataraman, P. Costa, T. Kim, S. Muthukrishnan, V. Kuppa *et al.*, "Chi: A scalable and programmable control plane for distributed stream processing systems," *Proceedings of the VLDB Endowment*, vol. 11, no. 10, pp. 1303–1316, 2018.

[64] B. Del Monte, S. Zeuch, T. Rabl, and V. Markl, "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*, 2020, pp. 2471–2486.

[65] A. Bartnik, B. Del Monte, T. Rabl, and V. Markl, "On-the-fly reconfiguration of query plans for stateful stream processing engines," *BTW 2019*, 2019.

[66] Y. Zhu, E. A. Rundensteiner, and G. T. Heineman, "Dynamic plan migration for continuous queries over data streams," in *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '04. New York, NY, USA: Association for Computing Machinery, 2004, p. 431–442. [Online]. Available: https://doi.org/10.1145/1007568.1007617

[67] M. Hoffmann, A. Lattuada, F. McSherry, V. Kalavri, J. Liagouris, and T. Roscoe, "Megaphone: Latency-conscious state migration for distributed streaming dataflows," *Proceedings of the VLDB Endowment*, vol. 12, no. 9, pp. 1002–1015, 2019.

[68] Y. Wu and K. Tan, "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*, J. Gehrke, W. Lehner, K. Shim, S. K. Cha, and G. M. Lohman, Eds. IEEE Computer Society, 2015, pp. 723–734. [Online]. Available: https://doi.org/10.1109/ICDE.2015.7113328

[69] S. Rajadurai, J. Bosboom, W. Wong, and S. P. Amarasinghe, "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*, X. Shen, J. Tuck, R. Bianchini, and V. Sarkar, Eds. ACM, 2018, pp. 98–112. [Online]. Available: https://doi.org/10.1145/3173162.3173170

[70] Z. Wang, S. Ni, A. Kumar, and C. Li, "Fries: Fast and consistent runtime reconfiguration in dataflow systems with transactional guarantees (extended version)," *CoRR*, vol. abs/2210.10306, 2022. [Online]. Available: https://doi.org/10.48550/arXiv.2210.10306

[71] E. Volnes, T. Plagemann, and V. Goebel, "To migrate or not to migrate: An analysis of operator migration in distributed stream processing," *IEEE Communications Surveys & Tutorials*, vol. 26, no. 1, pp. 670–705, 2024.

[72] T. Härder, "Observations on optimistic concurrency control schemes," *Information Systems*, vol. 9, no. 2, pp. 111–120, 1984. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0306437984900206

[73] T. Härder and K. Rothermel, "Concurrency control issues in nested transactions," *The VLDB journal*, vol. 2, pp. 39–74, 1993.

[74] P. A. Bernstein and N. Goodman, "Concurrency control in distributed database systems," *ACM Computing Surveys (CSUR)*, vol. 13, no. 2, pp. 185–221, 1981.

[75] H. Chen, C. Li, C. Zheng, C. Huang, J. Fang, J. Cheng, and J. Zhang, "G-tran: A high performance distributed graph database with a decentralized architecture," *Proc. VLDB Endow.*, vol. 15, no. 11, pp. 2545–2558, 2022. [Online]. Available: https://www.vldb.org/pvldb/vol15/p2545-chen.pdf

[76] P. Fuchs, J. Giceva, and D. Margan, "Sortledton: a universal, transactional graph data structure," *Proc. VLDB Endow.*, vol. 15, no. 6, pp. 1173–1186, 2022. [Online]. Available: https://www.vldb.org/pvldb/vol15/p1173-fuchs.pdf

[77] D. B. Jackson, Q. Snell, and M. J. Clement, "Core algorithms of the maui scheduler. in job scheduling strategies for parallel processing." in *Revised Papers from*

DRAFT

the 7th International Workshop on Job Scheduling Strategies for Parallel Processing, ser. JSSPP '01. Berlin, Heidelberg: Springer-Verlag, 2001, p. 87–102.

[78] S. Iqbal, R. Gupta, and Y.-C. Fang, "Planning considerations for job scheduling in hpc clusters," *Dell Power Solutions*, pp. 133–136, 2005.

[79] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at google with borg," in *Proceedings of the Tenth European Conference on Computer Systems*, ser. EuroSys '15. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2741948.2741964

PREPRINT