Evaluating Continuous Queries with Inconsistency Annotations

Samuele Langhi
samuele.langhi@univ-lyon1.fr
Lyon 1 University
Lyon, France

Angela Bonifati
angela.bonifati@univ-lyon1.fr
Lyon 1 University
Lyon, France

Riccardo Tommasini
riccardo.tommasini@insa-lyon.fr
INSA Lyon
Lyon, France

ABSTRACT

Continuous Queries (CQs) run indefinitely, processing infinite data streams and producing continuous outputs. They commonly use window functions to segment streams into finite chunks for computation. Ensuring data integrity in CQs is challenging, involving, for example, streaming joins for binary constraints. Current methods, like dropping or repairing inconsistent data, can harm throughput and increase latency. This paper proposes a novel approach using provenance-based techniques to map violations in input streams to CQ results with minimal overhead. This ensures continuous data flow and maintains the analytical integrity of CQs. Our study explores the feasibility and efficiency of this method, addressing a significant gap in applying provenance techniques to streaming data. While provenance-based techniques have proven effective for static data, their application in streaming contexts remains unexplored. Our solution addresses this gap, achieving a stable throughput across increasingly demanding memory loads wrt to the baselines, spacing between a 10% increase for medium-sized buffers (i.e., the windows), up to 80% for heavier loads. Moreover, results show the minimal impact of annotation (up to 25%) in the total execution runtime, demonstrating the effectiveness of our graph-based approach.

PVLDB Reference Format:
Samuele Langhi, Angela Bonifati, and Riccardo Tommasini. Evaluating Continuous Queries with Inconsistency Annotations. PVLDB, 18(5): 1321 - 1334, 2025.
doi:10.14778/3718057.3718062

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/semlanghi/cosqa.

1 INTRODUCTION

Continuous Queries (CQ) differ from classical DBMS queries because they are issued once and run until explicitly terminated [8]. CQs consume one or more infinite data streams and produce another as output. Thus, they are evaluated under Continuous Semantics [46]. There are many ways to achieve continuous semantics [8], the most common is using window functions [48], i.e., specialized operators that chunk the input streams into finite portions where it is possible to compute. As part of the query definition, windows are user-defined to capture phenomena occurring on the stream.

Proceedings of the VLDB Endowment, Vol. 18, No. 5 ISSN 2150-8097.
doi:10.14778/3718057.3718062

In DBMS, integrity checks are performed before the query evaluation by excluding or repairing tuples. On the other hand, CQs are issued once and evaluated periodically according to their window function [31]. Thus, enforcing data integrity while answering CQs requires determining what records violate a given IC over *infinite* data *during* the query execution. Intuitively, the mise-en-place of a versatile approach for detecting and handling constraint violations on the fly impacts the query performance and the results. If the constraint concerns a single tuple, e.g., Schema Constraint [19], violations can be detected using stateless queries. Conversely, binary constraints, like Primary Keys [23], require a self-join over the streaming data, i.e., an expensive operation.

Detection is not the only issue. Dropping inconsistent records, as done in DBMS, negatively impacts throughput. Conversely, repairing records using data mining techniques [44] is time-sensitive and, thus, has an impact on latency [44]. Moreover, neither of the approaches takes into account the standing nature of continuous queries. Indeed, they are applied before the (windowed) query execution permanently changes the input stream and, thus, they lose the mapping between input and output, with the risk of missing answers [39] or inaccurate analysis.

This paper advocates for a third way that requires deriving a mapping between the violations in the input stream and the continuous query results. In particular, we propose to map the integrity violations detected in the input streams to the continuous query results using how-provenance annotations. Although the approach is particularly suitable for streaming applications as it keeps data flowing [45], it was only applied to static data [26]. The main challenge for streaming lies in the detection process, which, if unoptimized, can severely harm the performance.

Our investigation builds on the work of Issa et al. [26] in the static context, representing streaming constraint violations as provenance annotations and integrating them within a novel provenance framework for window-based continuous queries. We show the correctness of the approach by drawing a commuting diagram, as it is done for similar streaming properties such as snapshot reducibility [30]. To evaluate the generality of our framework and empirically validate the performance of our prototype, we focus on three families of unary and binary constraints that have been applied over streams: (i) Primary keys (PK) enforce uniqueness in relational data, theyhave been used to optimised streaming joins in continuous queries [23] and for deduplication [18]. (ii) Schema Constraints (SH) [19] ensure the structural and domain integrity of streams' records, e.g., that attributes are present or types/thresholds are respected. (iii) Speed Constraints (SC) [44] are denial constraints used in domains like traffic or electric grid monitoring that limit the variation speed of a given value between two instants. Such constraints prevent abrupt, unrealistic changes that could indicate sensor failures or domain-specific anomalies.

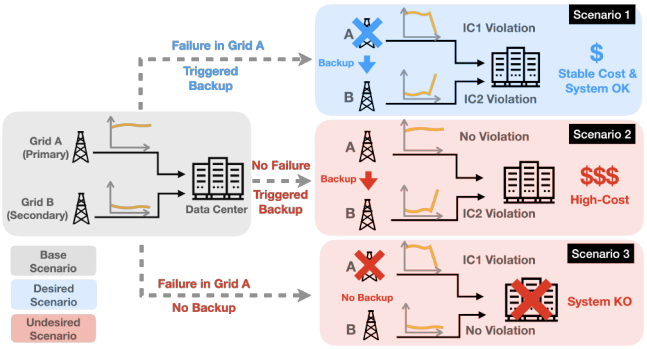


Figure 1: Scenarios that violate constraints IC1 and IC2 when they hint at a given behaviour of the electric system.

In summary, the paper provides the following contributions:

- A novel provenance framework based on semirings [24], in particular the polynomial semirings (the most general), that lifts windowing at the semiring level. Moreover, we define a stream positive algebra of operators for consistency-aware continuous query answering, which allows annotation manipulation.
- A method for efficiently detecting and propagating constraint violations in CQs. We show that a naïve solution using classic streaming operators impacts the query complexity that is quadratic in the window size. Our approach, InkStream, uses a list-based annotating solution that outperforms the naive one for any type of constraint. Also, we propose a graph-based approach that leverages constraint transitivity to optimize the annotation process, achieving linear-time complexity in the number of violations in a window.
- We implemented the naive solution, Inkstream, and its optimisation in Kafka Streams [43]. Moreover, we include an additional baseline based on Inca [26] to show the inadequacy of non-streaming solutions. Our evaluation considers five datasets, six continuous queries, and a qualitative study. We measure throughput, total runtime duration, and scalability, varying the stream inconsistencies percentage. Our experiments show minimal overhead in throughput wrt processing without violation mapping and better performance than the baseline [26]. Moreover, we prove the versatility of our inconsistency management mechanism wrt to other techniques, e.g., repair [44].

Outline. Section 2 presents the paper's running example. Section 3 formalizes the problem and introduces the preliminary concepts. Section 4 explains the formal framework based on provenance semirings for mapping stream inconsistencies to the continuous query results. Section 5 shows how these concepts are integrated with modern streaming solutions. Section 6 discusses the results of our experimental evaluation. Section 7 contains the related work on provenance, integrity over streams, and consistent query answering. Finally, Section 8 presents the conclusion and future work.

2 MOTIVATING EXAMPLE

FEG, a fictional electricity provider, wants to monitor the cumulative usage cost of two grids, denoted A and B, while correctly managing possible malfunctions. The data comes as a relational stream, i.e., an infinite sequence of time-ordered records. In particular, the *Consumption* stream's schema is ⟨`UUID`, `area`, `consA`, `consB`, `ts`⟩, with `area` being the geographical area, e.g., US, `consA` and `consB` being instantaneous consumption, `UUID` is a unique identifier generated by the monitoring system, and `ts` the timestamp.

```
SELECT area,sum(consA)*1.2+sum(consB)*1.5,ts
FROM Consumption [RANGE 5 minutes SLIDE 2 minutes]
WHERE consA >= 0 AND consB >= 0
GROUP BY area;
```

Listing 1: *Consumption* stream Monitoring: the cumulative costs by area within 5 min every 2 min.

Table 1 shows an example of the stream. Listing 1 shows the monitoring query, which processes the *Consumption* stream; it groups records by area and computes the total price of electricity consumption over the last 5 minutes. The total price is calculated by summing up all instantaneous consumption for the two grids and multiplying each sum with the respective consumption/dollars ratio, i.e., 1.2 for grid A and 1.5 for grid B.

FEG strives to reactively back up malfunctions in both power grids. More specifically, Figure 1 presents three possible malfunctions scenarios, where the system must be able to discern between cases where the malfunction is correctly backed up (blue rectangle), to the one where the backup system is faulted (red rectangle). To achieve this goal, they adopt Speed Constraints [44], like those outlined in rules IC1 and IC2, on data records. These constraints help identify irregular "spikes" (positive or negative) in consumption [44]. Moreover, IC3 avoids the even unlikely `UUID` collisions [27]. For the following definition, we reference records with letters x and y within a sample stream S, represented in Table 1.

$$\forall x, y \in S \ -2 \leq \frac{x.consA - y.consA}{x.ts - y.ts} \leq 2 \qquad \text{(IC1)}$$

$$\forall x, y \in S \ -2 \leq \frac{x.consB - y.consB}{x.ts - y.ts} \leq 2 \qquad \text{(IC2)}$$

$$\forall x, y \in S \ x.UUID \neq y.UUID \qquad \text{(IC3)}$$

As shown in Figure 1, the three scenarios correspond to three violation patterns. Notably, repairing the related integrity violations like in [44] is not advisable, since data would be modified to respect the constraint, thus neglecting a more comprehensive analysis of the violations. Indeed, simultaneous violations of IC1 and IC2 hint at desired behaviors of the system (cf. Figure 1). Therefore, it is essential to encode integrity violations in provenance metadata and provide end users with access.

Such integrity violations and the respective violating records should be clearly identifiable in order to determine where to encode the provenance metadata. This guarantee is not always provided for ternary constraints such as the following:

$$IC' : \neg\exists x, y, z(x.UUID = y.UUID) \wedge (y.UUID = z.UUID)$$

When validated over sample stream S from Table 1, the constraint will consider all triples that contain records r_7 and r_8 as inconsistent. Moreover, all records in the stream would be considered in the set of violating triples, thus leading to ambiguity in the violation detection. In such cases, determining how to encode violation metadata is indeed not uniquely determined, and falls beyond the scope of this work (see Section 8). Consequently, in the rest of the paper we focus on the classes of unary and binary constraints, e.g., IC3 and IC1.

Table 1: Sample stream S. In red the sudden changes (IC1/IC2) and in orange UUID collisions (IC3).

UUID	area	ts	consA	consB	*id*
74fcf75a...	Europe	0	8	2	r_1
...	...	...	...	...	...
51361676...	US	2	8	2	r_6
d8f490c1...	Europe	3	8	2	r_7
d8f490c1...	US	3	8	2	r_8
05d6efc8...	Europe	4	5	2	r_9
c0a93dda...	US	4	5	5	r_{10}
d1183d9b...	Europe	5	2	2	r_{11}
edfaed34...	US	5	3	7	r_{12}
f4e29872...	Europe	6	0	2	r_{13}
e3c97cc3...	US	6	0	10	r_{14}
f4e29872...	Europe	7	0	2	r_{15}
...	...	...	...	...	...

3 PROBLEM FORMULATION

This section formalizes the problem that we address in this work, i.e., constraint violation mapping from input streams to continuous query answers. The section introduces various notations used throughout the paper.

Window-Based Continuous Queries. As common for modern streaming systems, we assume continuous queries can be represented as Direct Acyclic Graphs (DAG) of operators, e.g., in Kafka Stream (our engine of choice) [43]. In particular, we consider CQs expressed over relational streams (cf Definition 3.1) as a combination of the Selection, Projection, Union and Join with a time-based window function to limit the query scope. Due to the lack of space, we refer to [30] for a formal definition of such operators for relational streams, while we describe window functions below.

Definition 3.1. A data stream is an infinite sequence of records $r = (\omega, \tau)$, where ω is a tuple in Ω, and τ is a timestamp in $\mathcal{T}$. Events in the stream are ordered by their timestamps, denoted as $r.ts$.

Time-Based Window Functions. Given a time domain $\mathcal{T}$, windows are defined as functions that, given a timestamp, return an interval, i.e., $W : \mathcal{T} \rightarrow \mathcal{T} \times \mathcal{T}$ where $\mathcal{T}$ is a set of timestamps. We denote the set of all possible intervals as $\mathcal{W}$ and $\mathcal{Q}^W$ a continuous query that adopts a given window function W, and with $\mathcal{Q}^W(S)$ the *result stream* of that query evaluated on input stream S. Operationally, time-based sliding windows are defined as a tuple (ι, β) where ι represents the size and β the slide [16], e.g., respectively 5 and 2 minutes in Listing 1. Applying a window function W to a stream S assigns each record of S to an interval $\mathbb{W}$, and $S[\mathbb{W}]$ defines a finite portion of S records within an interval $\mathbb{W}$.

Definition 3.2. Given a stream S, its evaluation against a Window-based Continuous Query $\mathcal{Q}^W$ is defined as the same query evaluated over all intervals $\mathbb{W}$ returned by W ($\mathbb{W} \leftarrow W$). More formally,

$$\mathcal{Q}^W(S) = \bigcup_{\mathbb{W} \leftarrow W} \mathcal{Q}(S[\mathbb{W}])$$

Integrity constraints (IC) are logical rules that ensure data consistency within a database. In this work, we will cover three types of ICs, i.e., Speed Constraints (SC) [44], e.g., IC1 and IC2, Primary Keys (PK), e.g., IC3, and Schema Constraints (SH) [19].

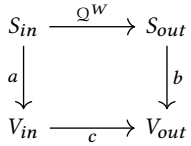


Figure 2: Commuting Diagram for Problem 1: $S_{in/out}$ are the input and output streams, $V_{in/out}$ are the constraint violations associated to $S_{in/out}$ respectively using a and b (to be determined), and c from V_{in} to V_{out} is also to be determined .

For all kinds of IC, their evaluation is conceptually rooted in query answering. In practice, the evaluation of a given constraint IC can be reduced to a query $\mathcal{Q}_{IC}$ that returns the set of violating records [41]. $\mathcal{Q}_{IC}$ is typically expressed using a language like SQL.

```
SELECT C1.*, C2.*
FROM Consumption AS C1, Consumption AS C2
    [RANGE 5 minutes SLIDE 2 minutes]
WHERE (C1.consA-C2.consA)/(C1.ts-C2.ts)<-2 OR
      (C1.consA-C2.consA)/(C1.ts-C2.ts)>2;
```

Listing 2: Query evaluating IC1, window from Listing 1.

Listing 2 shows the window-based CQ derived from IC1 evaluated over the *Consumption* stream. The query self-joins the stream, detecting and searching for records that concur in the violation of constraint IC1 (highlighted in red in Table 1).

Definition 3.3. Given n-ary constraint IC and a stream S, the *violation set* of IC wrt to S, denoted as $V(IC, S)$, is the set of all combinations of records $r_1, ..., r_n \in S$ such that $(r_1, ..., r_n) \in \mathcal{Q}_{IC}(S)$.

This paper advocates for consistency-awareness [26], i.e., given a (set of) constraint(s) defined on the input, we aim at providing a mapping between the violations in the input stream and the CQ results. Below, we formalise the problem using a commuting diagram shown in Figure 2. The intuition is that constraint violations in the input impact the query result, and thus, there should be a function applicable to the query result that can map its consistency back to the consistency of the input. Figure 3 shows why the DBMS-like approach and data repair do not commute. The former filters out the inconsistent records through a preliminary query $\neg\mathcal{Q}_{IC}$ that takes the consistent records. In addition to negatively impacting throughput, this solution renders impossible violation detection directly on the output, i.e., the set V_{out}, as the violation set V_{in} is emptied before the user-defined query processing. Conversely, the latter, data repair [44], makes also impossible V_{out}. The repair strategy R_{IC} leads to irreversible data modifications, which empties V_{in} by construction. Indeed, a function c that makes the diagram commit does not exist (an empty domain makes it not a function). Constraint-specific repairs are usually window-based and may lead to irreversible data modifications based on a finite data portion.

Problem. 1 (Continuous Violation Mapping). *Let S_{in} be a stream, $\mathcal{Q}^W$ be a window-based continuous query with output stream S_{out}, and IC be an integrity constraint. We call V_{in} the set of violations of a constraint IC generated in stream S_{in}. The problem of continuous violation mapping is to map the inconsistency of the query results wrt IC, represented by the violation set V_{out}, with the violations generated by records in S, which translates into identifying the transformation a, b, and c in the commuting diagram in Figure 2.*

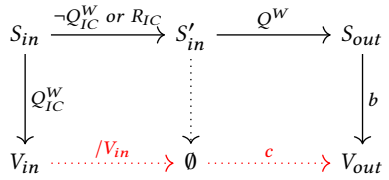


Figure 3: The commuting diagram for the filter-out ($\neg Q^W_{IC}$) and repair (R_{IC}) approaches.

4 CONTINUOUS VIOLATIONS MAPPING

This section introduces our solution to Problem 1 based on query provenance [24]. First, the section recaps the essential notions on query provenance and semirings. Then, it presents our extension to the provenance semiring framework for window-based continuous queries and the operators of the Streaming Positive Algebra. Finally, it draws a commuting diagram that specifies Diagram 2.

4.1 Query Provenance and Semirings

Query provenance takes the form of annotations on the output records based on source ones. That can be as simple as enriching the output with the set of all the contributing records IDs, like is done in data warehousing [14], or complex as in why-provenance [9], where results are annotated with sets of sets of IDs.

Provenance Semirings are used in data management to abstract different provenance models [24]. Formally, a semiring is an algebraic structure of the form $(K, +, \times, 0, 1)$, where $+$ and $\times$ are addition and multiplication over K, e.g., $+ = \cup, - = \cap$ when $K = \mathcal{P}(\Omega)$; Element 0 and 1 are respectively the neutral element of $+$ and $\times$, e.g., $0 = \emptyset, 1 = \Omega$ when $K = \mathcal{P}(\Omega)$. Notably, $(K, +, 0)$ and $(K, \times, 1)$ are commutative monoids and $+$ is distributive over $\times$, i.e., , 0 is the annihilating element for $\times$, i.e., $\forall a \in K, a + 0 = a, a \times 1 = a$, and $a \times 0 = 0$. When used for provenance over databases, each tuples is assciated with an element of the semiring, e.g., $\mathcal{A} : \Omega \rightarrow \mathcal{P}(\Omega)$. Moreover, Green et al. defined the *Positive Relational Algebra* [24] over K-relations, where each relational operator modifies the K-Relation applied on its result tuples, seamlessly integrating provenance semirings within relational queries. This paper focuses on *polynomial semirings*, as they are the most informative form of provenance. The semiring is defined as $(\mathbb{N}[\mathbb{X}], +, \cdot, 0, 1)$, which includes the set of bounded-degree, finite polynomials with natural coefficients and infinite variables [24].

4.2 Provenance for Streaming Integrity

In principle, the representation of constraint violations can be expressed through nested lists, which can be defined within the provenance framework through a multi-set semiring, which is rather

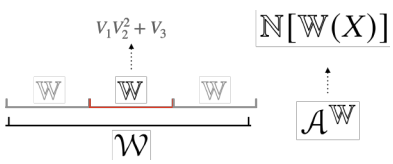


Figure 4: The relationship behind the set of possible windows $\mathcal{W}$, $\mathcal{A}^{\mathbb{W}}$ and the annotation polynomial.

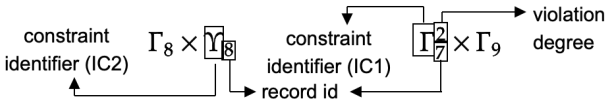
complex. For a more intuitive approach, we rely on the *polynomial semiring* [24]. Thus, the set of polynomial variables $\mathbb{X}$ is an infinite yet countable set of violation variables V_k built from stream S records. More specifically, a variable V_k in the polynomial annotated on record r, indicates a violation related with r_k, i.e., $\neg IC(r, r_k)$.

Nevertheless, a straightforward extension of the polynomial semiring to data streams is not feasible since it does not respect two requirements of semirings [24], i.e., (a) The support of the K-relation must be finite, and a stream S is infinite as per Definition 3.1. (b) Each element in the co-domain of a K-relation is finite, and a record r might violate a constraint with infinite other records. To adjust both problems, we limited the support to each finite subset identified by the window function, introducing an exponent in the polynomial that nullifies variables outside a window scope [32].

These extensions led to the definition of a novel semiring based on the set $\mathbb{N}[\mathbb{W}(\mathbb{X})]$, i.e., the polynomials based on a finite subset of $\mathbb{X}$ defined over records in $S[\mathbb{W}]$, and an alternative to K-relation that we name $\mathbb{W}$-relation (cf Definition 4.1). An essential difference with the non-streaming scenario is that a $\mathbb{W}$-relation is defined for each finite subset of S identified by a given window function W. Such difference affects the algebra as we discuss in Section 4.3.

Definition 4.1. Let $(\mathbb{N}[\mathbb{W}(\mathbb{X})], +, \cdot, 0, 1)$ be the semiring of polynomials with natural coefficients and a finite set of variables. Given a stream S and a time interval $\mathbb{W}$, the $\mathbb{W}$-relation, denoted as $\mathcal{A}^{\mathbb{W}}$, maps each record in $S[\mathbb{W}]$ to a polynomial in $\mathbb{N}[\mathbb{W}(\mathbb{X})]$.

Example. 1. *We can further describe the semiring $\mathbb{N}[\mathbb{W}(\mathbb{X})]$ in the context of our running example. In particular, we will consider the variable set $\mathbb{X}$ as the ensemble of all variables of the form Γ_i, Υ_i and μ_i, where each symbol is associated with a specific constraint, IC1, IC2 or IC3, while each index references a given record r_i. The interval $\mathbb{W}$, preliminary identified by the window from Listing 1, limits those indexes to records with timestamps within $\mathbb{W}$. Given $\mathbb{W} = [0, 5]$, valid polynomials are the following, where exponents are used as constraint-dependent violation degrees (see Definition 4.2):*



The main idea is annotating each record r with monomials containing constraint-specific labels of the form V_k, with k being the index of the record with which the constraint is violated. The window limits the size of the annotation as it approximates any infinite polynomial to a finite one. This approximation is done in two steps through a function α, that goes to zero when the passed record r_k timestamp is not in $\mathbb{W}$ (i.e., $\neg(r_k.ts \in \mathbb{W})$). More formally,

Definition 4.2. $\mathcal{A}^{\mathbb{W}}$ is a function defined on the set of records in a stream S, such that:

$$\mathcal{A}^{\mathbb{W}}(r) = \alpha(r) \cdot \prod_{r_k \in S} V_k^{\alpha(r_k) \cdot \beta(r, r_k)} \text{ with } \alpha(r) = \begin{cases} 1 & r.ts \in \mathbb{W} \\ 0 & otherwise \end{cases}$$

where $\mathbb{W}$ is an interval defined by a window function W (see Section 3), and β is constraint-specific, defined in Table 2.

Moreover, β is constraint-specific and defined in Table 2. SHs and PK have the simplest $\mathbb{W}$-relation, which evaluates the violation as either present or not, i.e., each violation has a maximum

Table 2: The $\mathbb{W}$-relations for each of the analysed constraints.

Constraint	$\mathbb{W}$-Relation (β)
Primary Keys (*PK*)	$\beta(r, r_k) = 1$ if $(r, r_k) \in V(PK, S)$
Schema Constraints (*SH*)	$\beta(r, r_k) = 1$ if $(r) \in V(SH, S)$
Speed Constraints (*SC*)	$\beta(r, r_k) = \epsilon_k(r)$ if $(r, r_k) \in V(SC, S)$

degree of 1 and a minimum of 0. Conversely, one can include a fine-grain violation degree for SCs as exponents, e.g., the difference ϵ_k between the annotated record value and its minimum change repair [44]. Example 2 shows such calculus for the example from Section 2. Notably, the degree encoding is domain-specific, e.g., one can alternatively use the time distance between the two violating records can be used as a decay parameter of the violation as in [12].

Constraint Symmetry. The binary constraints considered are all based on symmetric relations, e.g., iff r_1 violates a PK in combination with r_2, the inverse is also true. Symmetry, in combination with the order of arrival of the violating records [4, 16], allows avoiding to annotate both records involved in a violation. In practice, *when two records generate a violation, we annotate the most recent one. To determine the violations generated by each new record we validate it with past records, i.e., the Backward Context (BC) [32]. Such interpretation is motivated for PKs, where the earliest violating element remains consistent at the time of its arrival, while the latest one triggers the violation.* Alternatively, one can annotate with respect to future records, i.e., the Forward Context (FC). Table 3 shows both annotations, i.e., BC ($\mathcal{A}$) and FC ($\mathcal{A}_{FC}$) for the considered stream portion. The amount of information captured by both methods is equal, with each violation label having a counterpart with the same exponent. Moreover, one can reconstruct $\mathcal{A}^{\mathbb{W}}_{FC}$ from $\mathcal{A}^{\mathbb{W}}$ through the record indexing, e.g., iff $\Gamma_7^2 \in \mathcal{A}^{\mathbb{W}}(r_{13})$ then $\Gamma_{13}^2 \in \mathcal{A}^{\mathbb{W}}_{FC}(r_7)$.

Despite the methods being equivalent, FC-annotation is not compatible with some state-of-the-art stream processing engines, e.g., Kafka Streams [43], since it would require to wait all necessary future records before annotating, ultimately requiring custom watermarking and possibly cause unintended delays [6]. Moreover, BC-annotation allows each record to be annotated immediately upon consumption and passed down the pipeline without retracting the annotation later, offering latency gain. For these reasons, we leave the FC annotation investigation for future works.

Table 3: Records from stream S with their annotations. Violations of IC1, IC2 and IC3 are respectively denoted Γ, Υ, μ.

UUID	area	ts	cA	cB	*id*	$\mathcal{A}^{\mathbb{W}}$	$\mathcal{A}^{\mathbb{W}}_{FC}$
d8f49..	Europe	3	8	2	r_7	1	$\mu_8 \times \Gamma_7 \times \Gamma_{11}^2 \times \Gamma_{13}^2$
d8f49..	US	3	8	2	r_8	μ_7	$\Gamma_{10} \times \Upsilon_{10} \times \Gamma_{12} \times \Upsilon_{12} \times \Gamma_{14}^2 \times \Upsilon_{14}^2$
05d6e..	Europe	4	5	2	r_9	Γ_7	$\Gamma_{11} \times \Gamma_{13}$
c0a93..	US	4	5	5	r_{10}	$\Gamma_8 \times \Upsilon_8$	$\Gamma_{14} \times \Upsilon_{14}$
d1183..	Europe	5	2	2	r_{11}	$\Gamma_7^2 \times \Gamma_9$	1
edfae..	US	5	3	7	r_{12}	$\Gamma_8 \times \Upsilon_8$	$\Gamma_{14} \times \Upsilon_{14}$
f4e29..	Europe	6	0	2	r_{13}	$\Gamma_7^2 \times \Gamma_9$	1
e3c97..	US	6	0	10	r_{14}	$\Gamma_8^2 \times \Gamma_{10} \times \Gamma_{12} \times \Upsilon_8^2 \times \Upsilon_{10} \times \Upsilon_{12}$	1

Example. 2. *Table 3 shows how a finite portion of a stream can be annotated wrt the violation of constraints IC1, IC2, and IC3. Record r_7 is consistent within interval $\mathbb{W} = [0, 3]$, and thus annotated with 1 (cf. Definition 4.2). Additionally, constraint IC3 is violated by r_7 and r_8 since they have the same* UUID. *Such violation is detected at the arrival of r_8, which is annotated with μ_7. Record r_{10} violates both IC1 and IC2 when combined with record r_8. Record r_{11}, on the other hand, violates IC1, but in combination with two distinct records, i.e., r_7 (Γ_7^2) and r_9 (Γ_9). Thus, given an interval $\mathbb{W} = [0, 4]$, r_{10} and r_{11} will be respectively annotated with the polynomials from Example 1, i.e., $\Gamma_8 \times \Upsilon_8$ and $\Gamma_7^2 \times \Gamma_9$. As seen in the annotation, r_{11} and r_7 present a degree of violation of 2, which corresponds to a higher exponent, calculated through the following formula*

$$\epsilon_7(r_{11}) = \underbrace{r_{11}.consB}_{\text{actual } r_{11}} - \underbrace{r_7.consB + 2(r_{11}.ts - r_7.ts)}_{\text{minimum repair of } r_{11} \text{ wrt } r_7} = 2 \rightarrow \Gamma_7^2$$

4.3 Streaming Positive Algebra

In the following, we provide the operator definitions for the Streaming Positive Algebra ($\mathcal{SA}^+$), which describes the rules for combining annotations. The algebra operators must be closed under a specific set, that we identify the set of $\mathbb{W}$-relations, over all possible $\mathbb{W}$.

SELECTION. Given a predicate $P : (\Omega, \mathcal{T}) \rightarrow \{0, 1\}$, selection is the application of the $\mathbb{W}$-relations multiplied by the selection predicate.

$$(\sigma^+_P \mathcal{A}^{\mathbb{W}})(r) = \mathcal{A}^{\mathbb{W}}(r) \cdot P(r)$$

PROJECTION. Let A and U be sets of attributes such that $A \subset U$. Given a $\mathbb{W}$-relation $\pi_A \mathcal{A}^{\mathbb{W}}$ defined over a stream schema A, and $\mathcal{A}^{\mathbb{W}}$ defined over record tuples with schema U is

$$(\pi^+_A \mathcal{A}^{\mathbb{W}})(r) = \sum_{r = \pi_A r' \wedge \mathcal{A}^{\mathbb{W}}(r') \neq 0} \mathcal{A}^{\mathbb{W}}(r')$$

Under set semantics, projection may cause records to collapse. In such cases, we shall sum of all annotations of the collapsed records.

JOIN. Given two annotation policies $\mathcal{A}_1^{\mathbb{W}}, \mathcal{A}_2^{\mathbb{W}}$, and records $r_1 = (\omega_1, \tau_1), r_2 = (\omega_2, \tau_2), r = r_1 \bowtie r_2$, the annotation is

$$(\mathcal{A}_1^{\mathbb{W}_1} \bowtie^+ \mathcal{A}_2^{\mathbb{W}_2})(r) = \mathcal{A}_{res}^{\mathbb{W}_3}(r) = \mathcal{A}_1^{\mathbb{W}_1}(r_1) \cdot \mathcal{A}_2^{\mathbb{W}_2}(r_2)$$

$$\text{with } \mathbb{W}_3 = [min(\mathbb{W}_1.o, \mathbb{W}_2.o), max(\mathbb{W}_1.c, \mathbb{W}_2.c)]$$

As a derivative of the Cartesian product, the final annotation is the product of the two input annotations when performing a join.

Example. 3. *Projecting over attribute* consB *causes the collapse of different records with the same value, e.g., records r_7 and r_8 have both value 2 and the same timestamp. Conversely, joining two records with the same* area*, e.g., r_8 and r_9, the resulting annotation is the multiplication of the joined ones. Thus, given $\mathbb{W} = [1, 3]$*

$$\pi^+_{\text{consB}} \mathcal{A}^{\mathbb{W}}(\langle 2 \rangle) = 1 + \mu_7 \qquad (\mathcal{A}^{\mathbb{W}} \bowtie^+ \mathcal{A}^{\mathbb{W}})(r_8 \bowtie r_9) = \mu_7 \times \Gamma_7$$

Both functions $(\mathcal{A}^{\mathbb{W}} \bowtie^+ \mathcal{A}^{\mathbb{W}})$ and $\pi^+_{\text{consB}} \mathcal{A}^{\mathbb{W}}$, are derived from the operators applied on the data, respectively a projection and a join.

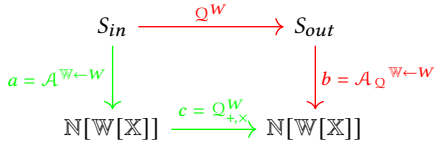


Figure 5: Commuting Diagram for Problem 1 solution (green path): transformation a is a $\mathbb{W}$–relation, b is the same $\mathbb{W}$–relation transformed through $\mathcal{SA}^+$, and c is an operation on polynomials derived from $\mathcal{SA}^+$.

4.4 Mapping Correctness

Figure 5 re-elaborate the diagram from Figure 2 assigning the transformation and interpreting the violation sets V_{in} and V_{out} as polynomials ($\mathbb{N}[\mathbb{W}]$). In green we highlight the path followed by our solution to map the input stream violations and the query results.

As explained in Section 3, Q^W is the user-provided window-based continuous query, with window function W that produces S_{out} from one or more input streams. Moreover, $\mathcal{A}^{\mathbb{W}\leftarrow W}$ is the $\mathbb{W}$–relation as defined in Definition 4.2, applied directly over each element of each input stream. We use the notation $\mathbb{W} \leftarrow W$ to indicate that intervals $\mathbb{W}$ determine the polynomial expression derived from the window function W. Please note that the window function is the same as the user query.

The commutativity of the diagram depends on the definition of the reminder b and c transformations. The former is $\mathcal{A}^{\mathbb{W}}_Q$, which corresponds to the $\mathbb{W}$–relation derived by applying the stream positive algebra to the user query operators, as shown in Example 3. The latter is $Q^W_{(+,\times)}$, representing the enriched query execution on the semiring operations, which combines the polynomials into one.

Figure 6 illustrates how the two commuting paths from the diagram can be implemented in a streaming pipeline. In practice, implementing the output annotator function $\mathcal{A}^{\mathbb{W}\leftarrow W}_Q$, derived through $\mathcal{SA}^+$ operators, requires access to the contributing input tuples. To obtain the output, the query needs to be first executed. Then, each result in *Sout* is joined with the contributing input records in S_{in} (based on custom tuple ids). Finally, we apply the output annotator function $\mathcal{A}^{\mathbb{W}\leftarrow W}_Q$. However, this process is rather impractical, as it introduces considerable complexity to the operation ($O(m \times n)$).

Our approach still calls for more optimization. Indeed, the annotation function $\mathcal{A}^{\mathbb{W}}$ has a complexity up to $O(n^2)$ for binary constraints due to the need of a self-join. This operation adds relevant overhead regarding runtime, as we later show in the experiments (cf. Section 6). In the following, we describe a graph-based annotation method (Section 5.1) and the implementation of positive algebra operators through dataflow ones (Section 5.2).

5 CONSISTENCY-AWARE QUERYING

This section presents how to integrate consistency annotations into an SPE. The processing workflow (cf Figure 7) requires a set of constraints *IC* and a CQ Q^W. The window function in Q^W combines with *IC* into the annotation module that transforms an input stream into an annotated stream. Conversely, the propagation module processes Q^W using consistency-aware query operators.

5.1 Annotation

A naive approach for annotating records with integrity violations consists of converting constraints into dataflow topologies, completed with a window operator inherited from the query Q^W from the user. For unary constraints this approach is valid, as they can be validated through stateless dataflow operators. On the other hand, binary constraints require stateful processing, i.e., join and aggregation, which can introduce overheads, as shown in Section 6.

A more tailored approach is designing an ad-hoc stateful operator for efficient annotation wrt binary (and unary) constraints. A straightforward but resource-intensive method is to keep a list of the elements in the annotation window. When a new record arrives, the system iterates through the list. All binary combinations of the newly arrived record and iterated records are validated wrt a given constraint. If a violation is detected, the new record is annotated accordingly. Although this method is able to find all violations, it may be computationally intensive, especially when the percentage of violations is low, as it requires to exhaustively check all combinations of possibly consistent records.

Graph-Based Data Annotation. Alternatively, we can use an evaluation query to build a *provenance graph* as compact storage schemes in provenance annotation [28]. However, while provenance graphs are successfully used in static contexts, they are not natively designed for streaming scenarios. For this reason, we provide a novel graph summary for efficiently tracking data inconsistencies, i.e., *Consistency Graph Summary (CGS)*.

Definition 5.1. A Consistency Graph Summary is defined as a graph where each node corresponds to a record, and each edge (r_i, r_k) represents r_i consistent with r_k and that $r_i.ts > r_k.ts$.

The graph's acyclicity and topological order concerning the time are guaranteed by in-order insertion [40]. Notably, the graph can contain disconnected nodes representing records inconsistent with all the others. Moreover, since the annotation process relies on a $\mathbb{W}$ window, we associate a CGS for each annotation window.

Exploration. For every record, a Breadth-First Search (BFS) navigation is done (cf. Figure 8) following Algorithm 2: a *queue* tracks the nodes to navigate at each iteration. Initially, the *queue* contains only the *roots* (Line 14), i.e., nodes without any incoming edges. The algorithm recursively explores the graph: at each step, the top element of the queue (*CNode*) is removed, and the corresponding constraint is evaluated. If the constraint is respected, the new node and *CNode* are linked, optionally removed from *roots* (Line 6-9). Additionally, the CGS allows us to optimize the traversal if the constraint we are validating is binary and satisfies *consistency transitivity* defined as.

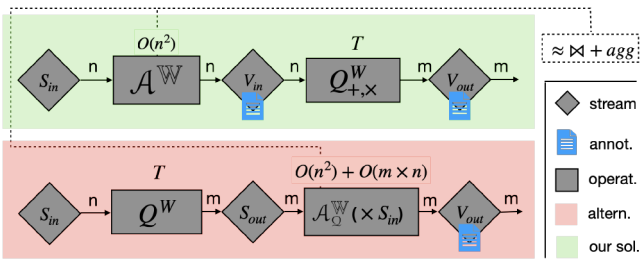


Figure 6: Commuting approaches for Inconsistency Mapping.

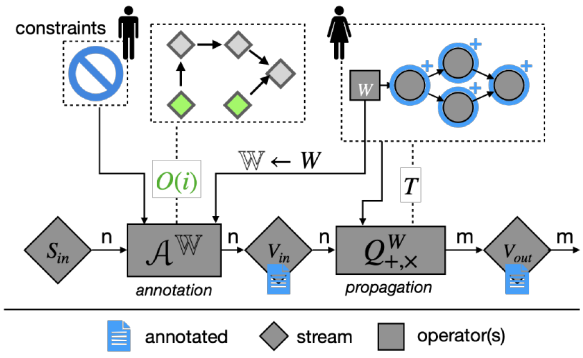


Figure 7: Consistency-Aware Continuous Querying.

Definition 5.2. A binary constraint *IC* is (in)consistency transitive iff: given three ordered records r_x, r_y, r_z, if r_x, r_y and r_y, r_z are (in)consistent wrt *IC*, then also r_x, r_z are (in)consistent wrt *IC*.

Transitivity guarantees that if a path exists between two nodes r_x and r_z in the CGS, then r_x is consistent with r_z. By exploiting this property, we can optimize the traversal without losing correctness. Indeed, when traversing the transitive reduction of a CGS, if a node is consistent, the traversal can stop since we can assume that all successive nodes will be consistent. This property is satisfied by the binary constraints used in this work. Conversely, if the constraint is not satisfied, the added record is annotated, and the nodes targeted by *CNode* (*AdjList*) are added to the queue (Line 13-15). The algorithm terminates when the *queue* is empty and returns the annotated node after adding it to *roots* (Line 2-4).

Maintenance. Algorithm 1 shows the PROCESS method: upon the arrival of a new record, it first clears the multi-buffer (*MB*) from expired windows (Line 2-4). Then, it extracts the time-ordered list of windows that contain the processed record (Line 5) according to a *windowPolicy*. The annotation is based only on the earliest window buffer, (Line 6), but the record is inserted in all the buffers without considering the annotation (Line 7-9). The INSERT performs the annotation through a function that navigates all the connected components in the graph. It follows a breadth-first search (BFS), rather than a depth-first one, because we verify consistency following the time-induced topological order.

Algorithm 1: PROCESS method and INSERT function.

```
   Data: MB Map with entries (window,cgraph), windowPolicy window function
1  Procedure PROCESS(r):
2      foreach window in MB.KEYS do
3          if window.END < r.timestamp then
4              MB.REMOVE(window);
5      aWins ← SELECTWIN(windowPolicy, r);
6      annotNode ← MB.GET(aWins.FIRST).INSERT(r);
7      aWins.REMOVEFIRST;
8      foreach window in aWins do
9          MB.GET(aWins.FIRST).INSERT(r)
10     FORWARD(annotNode);
11 Function INSERT(r):
12     queue ← INITIALIZE();
13     RNode ← NEWNODE(r);
14     queue.APPEND(roots);
15     annotNode ← CGRAPHSEARCH(RNode, queue);
16     return annotNode;
```

Complexity. In the list-based approach, the buffer should be scanned entirely for each arriving record resulting in a complexity of $O(n)$. For the graph-based approach, INSERT has also a worst-case complexity of $O(n)$, with n being the number of nodes in the graph, i.e., the record within the window. The worst case is when a record is inconsistent with all the other records, resulting in a BFS over the whole graph. If *consistency transitivity* holds, the complexity becomes $O(i)$, where i is the number of inconsistent records within the CGS. An Inconsistent Graph Summary (IGS) can used for streams with many inconsistent records: edges now represent the inconsistency between two nodes. In such cases, for those constraints that satisfy *inconsistent transitivity*, i.e., the property described above but for inconsistencies, the complexity becomes $O(c)$, where c is the number of records consistent with the inserted one.

Algorithm 1 implements a multi-buffer state maintenance, also called *bucketing* in other works [49], which is used by many state-of-the-art SPEs [10, 50]. Windows are stored and manipulated as the entries of a map-like structure. Each entry is described as the interval identifying the window (key) and a list of elements (value), allowing the system to neglect a specific routine for deletions. In this context, the INSERT is performed m times, i.e., the number of overlapping windows that contain the input record. Thus, the algorithm has a total time complexity of $O(m \times n)$ in worst-case and $O(m \times (i + r))$ when we assume a lower number of inconsistencies.

5.2 Propagation

The propagation module integrates the Streaming Positive Algebra (cfr. Section 4.3) in dataflow operation, enabling consistency-aware continuous querying. Each operator is instrumented to process two distinct information flows, i.e., data and annotations.

Since dataflow operators are not just relational, we explain our mapping to $\mathcal{SA}^+$ to the corresponding dataflow operator: $\mathcal{A}$-PROJECT performs a projection on the record tuples over a set of attributes A, through a `Map` operator. Their annotation is defined by a projected $\mathbb{W}$-relation (π^+_A), which consists of summing all annotations of collapsed records. To do so, it adopts a time window of size 1 after the projection since the equality of two records implies contemporaneity. $\mathcal{A}$-UNION combines two input streams with the `Merge`. Similar to $\mathcal{A}$-PROJECT, we use a window to sum their annotations

Algorithm 2: The CGRAPHSEARCH function.

```
1  Function CGRAPHSEARCH(RNode, queue):
2      if queue is empty then
3          roots.INSERT(RNode);
4          return RNode;
5      CNode ← queue.DEQUEUE() ;
6      if CHECK(CNode.constraint, RNode) then
7          CONNECT (RNode, CNode);
8          if roots.CONTAINS (CNode) then
9              roots.REMOVE (CNode);
10         if CNode.ISNOTTRANSITIVE then
11             queue.APPEND (CNode.AdjList);
12     else
13         ANNOTATEWITH (RNode, CNode);
14         if CNode.AdjList is not empty then
15             queue.APPEND (CNode.AdjList);
16     return CGRAPHSEARCH (RNode, queue);
```

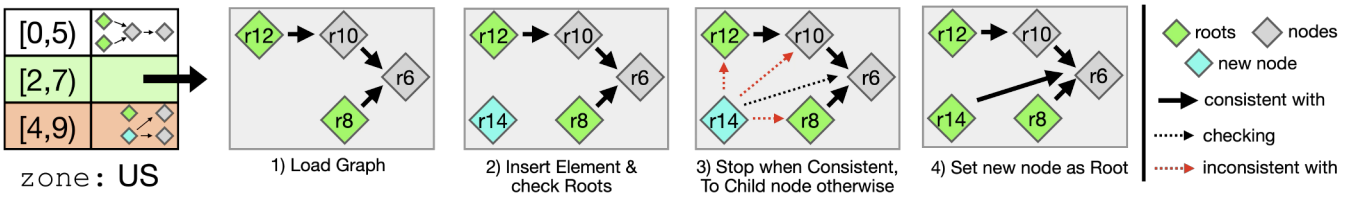


Figure 8: Exploration of a provenance graph upon the arrival of a record.

($\cup^+$). $\mathcal{A}$-SELECT is implemented through a `Filter` operator that discards records not satisfying the given predicate, consequently annihilating its annotation. $\mathcal{A}$-JOIN maps to the homonymous dataflow operation. Notably, `join` uses a user-defined window that is inherited from the annotation operator. Due to the incompatibilities of aggregates with both set and bag semantics [3] within the relational context, we do not include them in $\mathcal{SA}^+$ For demonstrative purposes, however, we added a final $\mathcal{A}$-AGGREGATE operator, which simply sums up all annotations of the windowed records.

Applicability Consideration. To understand the value of provenance annotations, we discuss their applicability. As mentioned earlier, annotations, particularly polynomial ones, have been applied to several tasks [21, 29]. In tracking inconsistency over streams, provenance polynomials offer several opportunities: *Roots Analysis* highlights instances satisfying certain conditions. For example, the *polynomial discriminant* can indicate unique inconsistencies (discriminant <> zero) or suggest redundant ones (discriminant=0). This approach can enable streaming data cleaning operations like [44], but without altering the input. *Quantifying the Degree of Inconsistency* is useful for anomaly detection [26], for instance, in financial analysis. Indeed, polynomials with a higher degree encode more severe inconsistencies, e.g., for SCs, the variables' exponents indicate how much the constraint has been violated. The applicability of more complex degrees [33] can be further investigated in the future. *Simplifying Annotations* through factorization can reveal inconsistency patterns, e.g., the co-violation of IC1 and IC2 in our example is a signal of healthy system. Annotations lift data violations to a symbolic level that could be leveraged for event recognition [13].

EXAMPLE. 4. *Table 4 shows the results from Listing 1 and the annotations resulting from $\mathcal{SA}^+$ operators. As already shown in Figure 1, a simultaneous violation of both IC1 and IC2 hints at a correct backup management, as the consumption from grid A decreases suddenly due to a malfunction (violation of IC1), while grid B instantly backs it up increasing the consumption (violation of IC2). To detect such situations, we apply the following function to the annotation, and simplify the annotations in Table 4 wrt the simultaneous presence of Γ (IC1 Viol.) and Υ (IC2 Viol.) in each monomial m within $\mathcal{A}^{\mathbb{W}}(r)$*

$$simplify(\mathcal{A}^{\mathbb{W}}(r)) = \sum_{m \in \mathcal{A}^{\mathbb{W}}(r)} \left(\frac{m}{\prod_{\Gamma_k \Upsilon_k \in m} \Gamma_k \Upsilon_k}\right)$$

6 EVALUATION

This section studies the performance of our framework on different datasets and with different sets of constraints. After introducing the experimental setup, we analyze the qualitative impact of our solution (distance wrt ground-truth), the scalability (average

Table 4: Subset of results from Listing 1. Violations of IC1, IC2 and IC3 are Γ, Υ, μ, respectively. We highlight the simplifications ($simplify$) of two violations of IC1 and IC2, which translates to the presence of both Γ_i and Υ_i, for a generic i.

z	cost	ts	$\mathcal{A}^{\mathbb{W}}(r)$	$simplify$
Eu	59. 4	5	$\underbrace{1}_{r_7} + \underbrace{\Gamma_7}_{r_9}$	$1+\Gamma_7$
US	63.9	5	$\underbrace{\mu_7}_{r_8} + \underbrace{\Gamma_8 \times \Upsilon_8}_{r_{10}}$	μ_7
Eu	42.6	7	$\underbrace{1}_{r_7} + \underbrace{\Gamma_7}_{r_9} + \underbrace{2(\Gamma_7^2 \times \Gamma_9)}_{r_{11}+r_{13}}$	$1+\Gamma_7+2(\Gamma_7^2 \times \Gamma_9)$
US	63.6	7	$\underbrace{\mu_7}_{r_8} + \underbrace{2(\Gamma_8 \times \Upsilon_8)}_{r_{10}+r_{12}} + \underbrace{\Gamma_8^2 \times \Gamma_{10} \times \Gamma_{12} \times \Upsilon_8^2 \times \Upsilon_{10} \times \Upsilon_{12}}_{r_{14}}$	μ_7

throughput) and overhead (runtime) for annotation and processing, varying window sizes and violation density.

Setup. All the experiments were executed on a Macbook Pro 2021, running MacOS (version 13.4.1) and Java (v 17.0.2), with a RAM of 64GB LPDDR5 and a 10-core M1 Max Processor. Our system runs on top of Kafka Streams [43] (v 3.2.2). To simulate a streaming ecosystem, experiments are executed consuming data from Apache Kafka topics (v 3.1.0). Finally, *inca-sc* uses PostgreSQL (v 15.3).

Datasets. We evaluate our solutions over six different datasets: three are real-world datasets, and three are synthetic. The *Electric Grid Dataset* is a synthetic dataset based on the running example from Section 1. We use it for the qualitative and ablation analysis. The *Stock Dataset* [35] is a collection of observations of price updates across the years. Updates arrive daily as $\langle name, price, date \rangle$. We shrank the arrival frequency to 1 second for compatibility with the Kafka retention time of the internal topics used in the queries while preserving the dataset's regular nature. The GPS dataset collects position observations detected through a GPS signal inside a campus. Positions are 2-dimensional coordinates (x, y), each with its timestamp. The dataset contains several wrongfully registered position observations, which can be detected through SCs, as already done in [44]. The Review dataset [17] is a real-world dataset of reviews coming from Metacritic and Rotten Tomatoes. Each review includes a user ID, the title of the reviewed movie/videogame, the number of stars (1 to 10), and the timestamp. Finally, the LinearRoad is a synthetic dataset [5]. Records contain the vehicles speeds and ids (vid) on a highway (xWay) divided into segments.

Queries. We run six different queries, one for the qualitative and ablation studies, and the other five for the performance study. For each query, we consider Primary Keys (PK), Speed Constraints (SC) and Schema Constraints (SH). (I) *StockPearson* calculates the correlation between multiple financial assets. In this context, SCs, PKs

and SH.s monitor asset volatility, key integrity, and schema adherence. (II) *StockCombo* joins assets according to their timestamps, creating real-time asset combinations. It uses the same operators as StockPearson, except the aggregate, and also inherits the same uses for each class of constraints considered. (III) *LinearRoad* is a query that joins vehicle positions wrt their location. SCs detect potential incidents, while PKs and SHs are adopted to verify timely data generation from every vehicle. (IV) *Review* directly monitor a stream of reviews. It uses SCs and polynomial Root Analysis to identify "review bombing", i.e., a high number of reviews with low scores produced to lower the average score of a movie/videogame. Instead, PKs and SHs verify the uniqueness of user reviews for each movie/videogame. (V) *GPS* detects absurd position variations with SCs. PKs are not used due to the absence of a proper key in the dataset tuples. (VI) *ElectricGrid* is the example query (cf. Listing 1).

Implementation Variants. In the performance study, all the queries are evaluated on three variants of our solution [36], implemented on top of single-thread Kafka Streams pipelines. *ink-schema* is a native implementation that annotates tuples according to SHs using Kafka Streams APIs. Both *ink-optim-sc* and *ink-sc* are native implementations that adopt SCs. While *ink-optim-sc* uses the graph from Section 5.1, *ink-sc* uses a list-based data structure. *ink-optim-pk* and *ink-pk* are native implementations adopting PK on graphs, exploiting the property of inconsistent transitivity (cf. Definition 5.2) and lists, respectively. *Naive-sc* and *naive-pk* perform consistency annotation and propagation wrt SCs and PKs with Kafka Streams.

Baselines. Our experiments include two baselines: (I) a Kafka Streams (*noinc*) implementation of the queries without consistency awareness, serving as the upper-bound performance target for our solution. (II) a DBMS-oriented (*inca-sc*) implementation of the consistency-aware framework based on [26]. Without a comparable streaming contribution from the state of the art, we adapted this solution to a continuous scenario by processing each annotation window as a table, managed incrementally for performance reasons. Rather than providing Inca with an entirely new dataset each time, we update it by adding and removing records based on the window's movement. We evaluate the *inca-sc* only on four of the five queries as it does not support aggregates.

6.1 Qualitative study

To highlight the value of our approach, we designed an experiment to show how our solution can identify inconsistencies and related underlining patterns to get closer to the groundtruth.

Scenario. The experiments are conducted on a synthetic dataset based on our running example (cf. data in Table 1 and query in Listing 1). In the first part, there is a malfunction in the monitoring service, as grid A stops working. This should be addressed by reducing the impact of this stoppage on the final price. The second half of the experiment consists of a stoppage of grid A and a backup from grid B. This behaviour is correct and needs no repair.

Figure 9 shows the query results executed with four different consistency management policies and the groundtruth. The *filter* indicates a database-like consistency enforcement, where inconsistent records are removed. This policy underestimates the electricity consumption price, as too many records are excluded. The *repair* adjust the consumption following the technique from [44], wrt IC1 and IC2. Finally, *noinc* indices no inconsistency management. Our solution, i.e., *inkstream*, is the nearest to the groundtruth. By annotating the data with the related violations, we can reason for inconsistencies at a higher level of abstraction. In such cases, we choose to ignore simultaneous violations of IC1 and IC2 by simplifying them, as shown in Table 4, and then using them to repair. Such preliminary simplification produces a cumulative cost more adherent to the groundtruth than the one produced by the *repair*.

6.2 Performance Study

Our second group of experiments measures the scalability of annotation and propagation, varying the window size and the number of inconsistencies in it. We operate under the assumption that *every record can generate a new violation*. This assumption is worst-case and is stricter than real-world scenarios. The second group of experiments measure the scalability wrt the number of violations considered, i.e., the *violation density*. Finally, a third group calculates the overheads of annotation and propagation.

Scalability wrt Window Size. Figure 11 shows all variants and baselines' average ingestion throughput. We calculate the throughput by measuring the total time needed to consume a fixed number of records. The figure is twofold: while moving horizontally varies the datasets, it also distinguishes the experiments for two different window size-slide ratios, i.e., 2 for the plots above (a) and 5 for those below (b). Such a measure highlights the impact of overlapping windows. We repeated the experiment for different annotation windows, i.e., DTW=10/100/1000/10000 seconds.

The *ink-schema* implementation (in gray) shows the best performance, near the one of the *noinc* baseline, with a 5% to 2% difference. This is thanks to the nature of SHs, which can be implemented as native Kafka Streams filters, which guarantee minimum latency, except for LinearRoad. Conversely, systems requiring window-based constraint monitoring excel with a small size/slide ratio, coherently with our complexity analysis (see Section 16). More specifically, *ink-optim-sc* (in **blue**) is the best among variants that implement SCs and PKs, i.e., *ink-optim-pk* (in **brown**), *ink-sc* (in **orange**), *ink-pk* (in **lightblue**), *naive-sc* (in **red**) and *naive-pk* (in **purple**).

In particular, for the simplest GPS and Review *ink-optim-sc*'s throughput decreases by 5% wrt *noinc* baseline (in **black**). Thanks to the CGS, the window size has far less impact in *ink-optim-sc* than all the other solutions compared to *noinc*. Indeed, for *ink-optim-sc*, the complexity depends on the number of inconsistencies and not the size of the windows. The same property is valid for *ink-optim-pk*, which is applied on all datasets except GPS, since it is not a key-based dataset. However, PKs are opposite to SCs because they satisfy *in*consistent transitivity (cf. Definition 5.2). As the experiments are based on or inspired by real-world scenarios, the number of violations is small compared to the amount of records in the window. Consequently, the CGS for PK, whose edges represent inconsistencies between records, presents many isolated nodes. This hinders the performance of *ink-optim-pk*, since the graph navigation over those nodes is comparable to a list iteration. For small windows, *ink-optim-pk* behaves similarly to *ink-optim-sc*. But as the size of the window increases, there is a linear decrease in throughput (the scale is logarithmic in Figure 11), which is comparable to *ink-pk*. Both *ink-optim-pk* and *ink-pk* go a 5% decrease

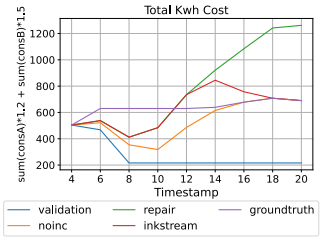


Figure 9: Gap wrt Groundtruth.

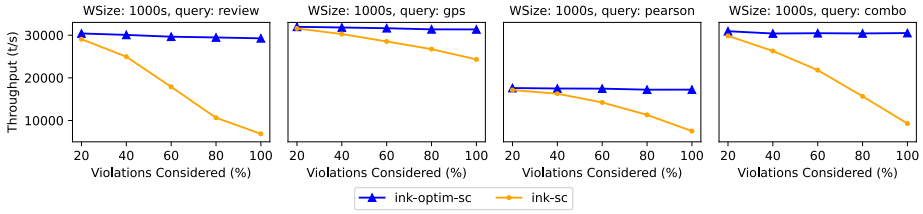


Figure 10: Scalability by Records Number: avg throughput varying validated records (%).

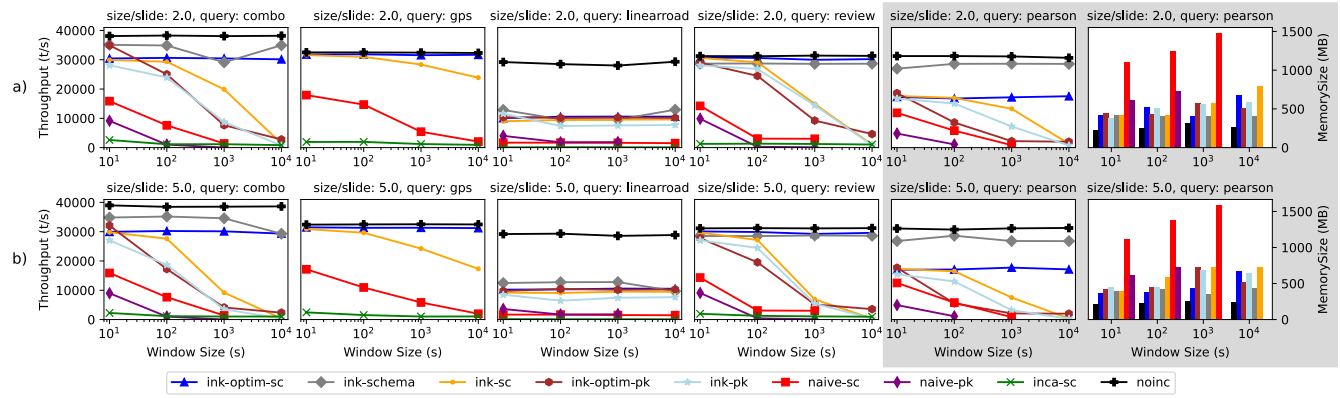


Figure 11: Scalability by Window Size: avg throughput and memory usage, varying annotation window size.

for small windows up to 90% decrease for very large windows. Conversely, *ink-sc* performance starts decreasing by 5% wrt *noinc* for small windows, to 20% decrease (50% in Review) for medium window sizes, to a 30% (up to 90% in Review) for large windows. For *naive-sc* and *naive-pk*, the decrease wrt *noinc* performance starts from 50% and goes down to 90% for GPS query.

For StockCombo, *ink-optim-sc* performs similarly to GPS and Review, although the decrease is more significant (15-20%). Again, the window size significantly impacts the rest of the system's performance, with a 50% to 65% decrease for *ink-sc*, *ink-optim-pk*, and *ink-pk* for middle-sized windows. For very large windows, as expected, we obtain a 90% decrease, which is similar to *naive-sc* and *naive-pk*. In StockPearson, all consistency-aware solutions show the highest overhead, with *ink-optim-sc* remaining the overall best with a 45% decrease in performance against *noinc*. Among all the PK-based solutions, *ink-optim-pk* is the best. However, like other queries, the window size significantly impacts throughput, leading to a decrease of up to 95% if compared with *noinc*.

LinearRoad shows the worst performance for all solutions. Indeed, the dataset contains registrations from almost 5000 vehicles in a very short period. Since annotation windows are maintained per vehicle, each retains a small number of records, thus not impacting the performance as in other queries. Moreover, the high density of records makes propagation performance heavy, even for *ink-schema*. Notably, *ink-optim-sc* and *ink-optim-pk* perform better than *ink-sc* and *ink-pk*. Finally, non-optimized solutions based on SCs, i.e., *ink-sc* and *naive-sc*, are better than those adopting PKs, i.e., *ink-pk* and *naive-pk*, by 5 to 10% as the window grows. This difference highlights how different constraints impact the performances, since SCs produce fewer violations as the time distance between records grows. On the other hand, PK violations are not time-dependent and can grow indefinitely as the window size grows.

The two rightmost graphs illustrate memory usage across various annotation window sizes. For space reasons, we only included the *StockPearson*, being the most complex query out of the five. Due to the inherent fluctuations in Java's garbage collection process, there are memory usage variances. However, focusing on broader trends, the plot reveals an overall increase in memory usage as the window size expands (up to 35-40%). The optimized solutions, i.e., *ink-optim-sc* and *ink-optim-pk*, consistently ranks among the least memory-consuming approaches, particularly when compared to the Kafka Streams baseline, i.e., *naive-sc* and *naive-pk*. These baselines require substantial inter-operator caching to execute the annotations, contributing to higher memory consumption.

Scalability wrt number of Validated Records. Figure 10 reports the average throughput for *ink-optim-sc* and *ink-sc* with varying percentages of considered records in the constraint validation. We used a Gaussian randomizer, configured with different percentage thresholds, to decide whether a record is included in the validation. As a probabilistic approach, we fixed the window size to be large enough (1000 seconds) to have enough samples. While the throughput of *ink-optim-sc* is robust wrt the number of validated records (decrement at most 5% in all the cases), *ink-sc* shows an evident decrement in the performance (up to 70%). Indeed, when low percentages of the records are considered in the validation process, the performance of *ink-optim-sc* and *ink-sc* are almost the same, as *ink-sc* has to iterate over few records. The more the records are considered in the validation, the more elements are added to the list, resulting in a decrease in the performances.

Scalability wrt Number of Violations. Figure 12 shows the average throughput difference between *ink-optim-sc* and *ink-sc* for the number of total violations detected within multiple experiments. To calculate this difference, we picked two real-world datasets and the

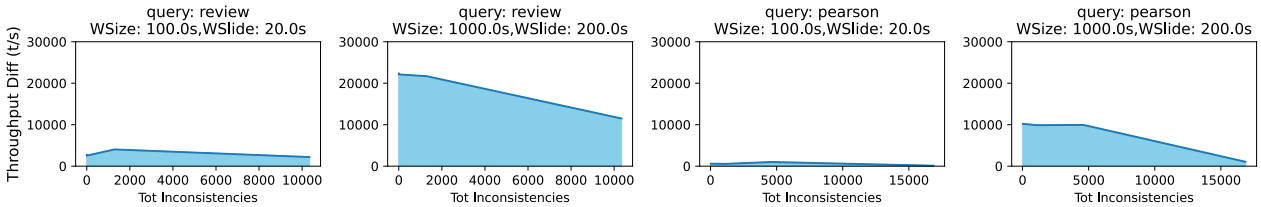


Figure 12: Scalability by Number of Violations: *ink-optim-sc* vs *ink-sc* for Review and StockPearson.

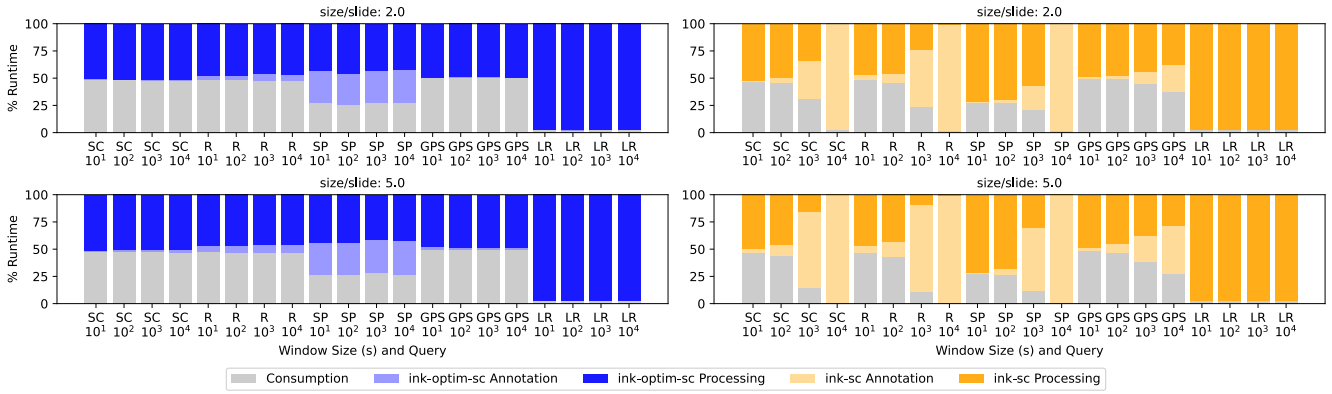


Figure 13: Overhead: Runtime Occupation (%) . Legend: [S]tock[C]ombo, [R]review, [S]tock[P]earson, [L]inear[R]oad, GPS.

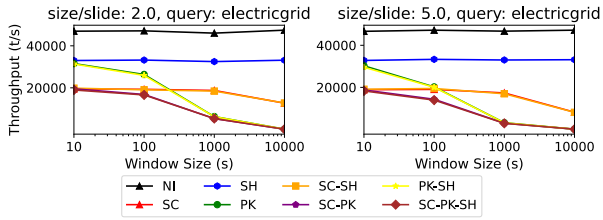


Figure 14: Ablation Study: the impact of SC, PK, SH on average throughput for the *ElectricGrid* query; NI: no constraints.

respective queries, i.e., *Review* and *StockPearson*, while the number of inconsistencies is increased by considering stricter SCs.

The results show a decreasing trend of the throughput difference as we increase the number of violations. The system can consider fewer inconsistencies for smaller windows, resulting in a less evident decrease. Indeed, the decrease is more noticeable for bigger windows (size 1000 seconds), reaching up to a 90% decrease for StockPearson and a 50% decrease for Review. This is compatible with our analysis from Section 16, since the complexity of graph-based annotation is proportional to the number of inconsistencies, behaving as a list-based annotation for a totally inconsistent stream.

Ablation Study. Figure 14 shows how each constraint impacts throughput for *ELectricGrid* query. We chose it as it includes multiple constraints, i.e., SC and PK, to which we add SH. We include the constrain-less execution (NI) as reference for max-throughput.

Evaluating many constraints show a performance degradation while increasing the window size. SHs evaluation concerns a single record, hence their impact on throughput is stable across window sizes. For SC the overhead in throughput is higher, i.e., about 55% lower then NI. However, PKs exhibit alone the highest reduction in throughput, about 15%. The experiment confirms the independence of constraint evaluation. Indeed, their effects on performance can be summed up. When evaluated together, SC and SHs (SC-SH) lead to a more significant drop, with performance declining by about 30% SC and PK are handled through separate execution graphs, while SH uses a dedicated operator, minimizing their overhead. The combined validation of SC and PK leads to a performance drop as window size grows, largely driven by PKs. SHs have minimal additional impact when combined with SC or PK, as their processing overhead remains low. Thus, SC and PK primarily drive the performance drop, while SH contributes only slightly.

Analysis of Overheads. Figure 13 shows the analysis of the annotation and processing overheads in terms of the percentage between the execution time of the whole query (in *ink-optim-sc* and *ink-sc*) and the execution time of a consistency-aware streaming pipeline that only performs annotation. On top of this, to evaluate the consumption overhead introduced by Kafka, we measured the execution time of a Kafka Streams pipeline that simply consumes records. For instance, for an annotation window equal to 10000, the Stock Pearson takes 5461 seconds to execute. Executing the sole annotation phase takes only 5300 seconds, and considering the consumption overhead of roughly 30 seconds, we obtain the final plot column, where most of the time spent (98-99%) is on the annotation. The annotation time represents 10 to 30% for *ink-optim-sc* and 10 to 99% for *ink-sc* of the entire time to perform the queries. In particular, while *ink-optim-sc* stays steady wrt the window size, *ink-sc* annotation time percentage grows up to 99% for high window sizes, representing the real bottleneck of the system. For GPS, Review, and StockCombo, the annotation of *ink-optim-sc* takes roughly 5-10% of the query's runtime, while it takes 30% of the running time in Stock Pearson. On the other hand, *ink-sc* annotation reaches 99% in Combo, Review, and Pearson and 50% of the runtime in GPS.

Finally, the provenance graph introduces an overhead for small windows. Indeed, *ink-sc* behaves mostly as than *ink-optim-sc* in

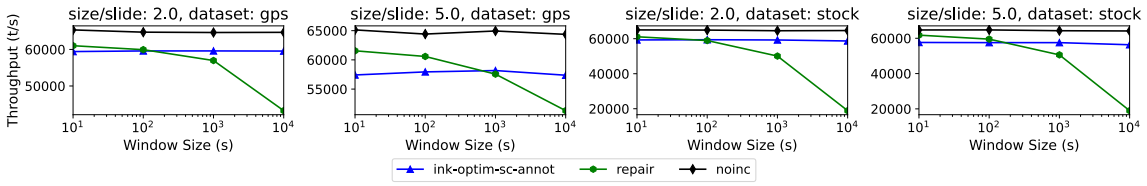


Figure 15: Comparative Study: avg throughput of InkStream annotation over SC, and SCREEN repair technique.

Pearson until the size of the window exceeds 10^3 (10^4) with size/slide ratio 5 (2). Moreover, Figure 13 confirms our hypothesis that the low performance of LinearRoad were due to the dataset nature.

6.3 Comparative Study

This section compares InkStream and SCREEN [44], an SC-based streaming repair technique. To evaluate throughput in similar conditions, we have ported SCREEN [20] into a Kafka Streams operator.

Figure 15 shows that the annotation-based provenance technique outperforms the repair method for the GPS and Stock datasets[1]. As window sizes increase, e.g., from 10 to 1000 seconds and greater, while InkStream maintained a constant throughput, SCREEN performances decrease, from a 10% (for Stock 5% for GPS) up to a 60% (for Stock 20% for GPS) lower than InkStream. Indeed, the annotation approach scales better wrt window size. In contrast, SCREEN performs better for small windows (10 and 100 seconds), with a 1% to 10% greater throughput than InkStream in both Stock and GPS queries. SCREEN's performance degradation for big windows is due to its iterative approach: SCREEN calculates the set of local optimum repairs, whose median becomes the global optimal repair [44]. As window grows, the number of considered elements increases, reducing the throughput. Instead, the graph-based approach of InkStream is more scalable as it avoids unnecessary comparisons.

7 RELATED WORK

This section positions our work in the state-of-the-art. Our work is related to streaming data quality and provenance. We also discuss the work on consistent query answering and consistency measures.

Provenance [9, 11, 24]. Recent works applying provenance to streaming data: *Ariadne* [22] is a provenance-aware extension of [2]. It uses operator instrumentation, i.e., modifying the behaviour of operators, to generate and propagate lineage $Lin(X)$ through operators of CQs. Conversely, its processing is based on relational algebra, making it harder to compare with modern SPEs based on dataflow operations. Our propagation approach follows the same principles for operator instrumentation but in dataflow settings. Instead, *Ananke* [38] adds users with why-provenance ($Why(X)$) to dataflow operators, specifying whether each source tuple can contribute to future results. It is based on the authors' prior work *GeneaLog* [37]. Finally, *Erebus* investigates the aspect of *completeness* [15, 42, 51], relying on why-provenance [38] for explaining the mismatch between actual and expected CQs results.

Consistent Query Answering. Existing works start from the notion of repair, i.e., a database instance fixing constraint violations. Multiple repairs are possible wrt the formal definition of repair, the family of queries, and the type of integrity constraints. Thus, consistent query answering refers to the notion of *certain answers*, i.e., the intersection of the query answers on all possible repairs of the initial database instance. As an alternative approach to data cleaning, consistent query answering could be prohibitive as the number of repairs can be exponentially significant. Moreover, the dynamic aspects of CQA have been largely neglected [7], and theoretical and empirical results are limited [34].

Transactional Stream Processing (TSP)[52] enables continuous computations over streaming data with ACID-compliant transactional correctness while preserving temporal order and low latency/high throughput. Affetti et al.[1] designed *TSpoon* to ensure consistency in the SPE's transactional subgraph (t-graph) by enforcing integrity constraints on individual keys and preventing invalid tuples. Thus, a t-graph remains consistent outside fully committed transactions. This provenance tracking suffices for $\mathbb{B}$ semirings. Ververica's Streaming Ledger is a similar TSP engine.

Consistency-Aware Query Answering focuses on quantifying the inconsistency. In particular, Inca [26] uses polynomial provenance to propagate the constraint violations to query results. In these regards, *Inca* is similar to our work. However, the work does not apply to window-based continuous querying. Designed for static, it quantifies the inconsistency wrt a set of denial constraints over the whole database. We identify Inca as the ideal baseline for measuring the suitability of our approach. Thus, we extend following a similar approach for SPE prototyping as specified in [47]. Our experimental study shows that an adapted solution is unsuitable for streaming, and a native system is required.

8 CONCLUSION

This paper introduced the problem of continuously mapping input stream integrity violations to continuous query results. We design a framework based on provenance semirings that address the problem by enriching the input stream with polynomial annotations. Then, we extended the semiring positive algebra to propagate annotations across CQs operators, i.e., projection and joins. We show the suitability of our approach by deriving constraint violations in the output. We also designed a graph-based optimization for annotation and extended dataflow operators to integrate the algebra.

In the future, we will explore how complex windowing functions, e.g., data-driven windows [25] can impact the annotation and propagation phases. Moreover, we plan to investigate how these policies can be used to provide integrity guarantees over the scoped records. We will extend the covered constraints regarding definition and arity, e.g., denial constraints, and investigate how alternative annotation methods can impact performances.

ACKNOWLEDGMENTS

R. Tommasini and S. Langhi are supported by the French Research Agency under grant agreement nr. ANR-22-CE23-0001 Polyflow.

[1]We excluded other datasets due to space limits, but results are similar.

REFERENCES

[1] Lorenzo Affetti, Alessandro Margara, and Gianpaolo Cugola. 2020. TSpoon: Transactions on a stream processor. *J. Parallel Distributed Comput.* 140 (2020), 65–79. https://doi.org/10.1016/J.JPDC.2020.03.003

[2] Yanif Ahmad, Bradley Berg, Ugur Çetintemel, Mark Humphrey, Jeong-Hyon Hwang, Anjali Jhingran, Anurag Maskey, Olga Papaemmanouil, Alex Rasin, Nesime Tatbul, Wenjuan Xing, Ying Xing, and Stanley B. Zdonik. 2005. Distributed operation in the Borealis stream processing engine. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, June 14-16, 2005*, Fatma Özcan (Ed.). ACM, 882–884. https://doi.org/10.1145/1066157.1066274

[3] Yael Amsterdamer, Daniel Deutch, and Val Tannen. 2011. Provenance for aggregate queries. In *Proceedings of the 30th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2011, June 12-16, 2011, Athens, Greece*, Maurizio Lenzerini and Thomas Schwentick (Eds.). ACM, 153–164. https://doi.org/10.1145/1989284.1989302

[4] Arvind Arasu, Shivnath Babu, and Jennifer Widom. 2006. The CQL continuous query language: semantic foundations and query execution. *VLDB J.* 15, 2 (2006), 121–142. https://doi.org/10.1007/S00778-004-0147-Z

[5] Arvind Arasu, Mitch Cherniack, Eduardo F. Galvez, David Maier, Anurag Maskey, Esther Ryvkina, Michael Stonebraker, and Richard Tibbetts. 2004. Linear Road: A Stream Data Management Benchmark. In *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, VLDB 2004, Toronto, Canada, August 31 - September 3 2004*, Mario A. Nascimento, M. Tamer Özsu, Donald Kossmann, Renée J. Miller, José A. Blakeley, and K. Bernhard Schiefer (Eds.). Morgan Kaufmann, 480–491. https://doi.org/10.1016/B978-012088469-8.50044-9

[6] Edmon Begoli, Tyler Akidau, Slava Chernyak, Fabian Hueske, Kathryn Knight, Kenneth L. Knowles, Daniel Mills, and Dan Sotolongo. 2021. Watermarks in Stream Processing Systems: Semantics and Comparative Analysis of Apache Flink and Google Cloud Dataflow. *Proc. VLDB Endow.* 14, 12 (2021), 3135–3147. https://doi.org/10.14778/3476311.3476389

[7] Leopoldo E. Bertossi. 2021. Second-Order Specifications and Quantifier Elimination for Consistent Query Answering in Databases (Abstract). In *Proceedings of the Second Workshop on Second-Order Quantifier Elimination and Related Topics (SOQE 2021) associated with the 18th International Conference on Principles of Knowledge Representation and Reasoning (KR 2021), Online Event, November 4, 2021 (CEUR Workshop Proceedings)*, Renate A. Schmidt, Christoph Wernhard, and Yizheng Zhao (Eds.), Vol. 3009. CEUR-WS.org, 28–36. https://ceur-ws.org/Vol-3009/abstract1.pdf

[8] Angela Bonifati and Riccardo Tommasini. 2024. An Overview of Continuous Querying in (Modern) Data Systems. In *Companion of the 2024 International Conference on Management of Data, SIGMOD/PODS 2024, Santiago AA, Chile, June 9-15, 2024*, Pablo Barceló, Nayat Sánchez Pi, Alexandra Meliou, and S. Sudarshan (Eds.). ACM, 605–612. https://doi.org/10.1145/3626246.3654679

[9] Peter Buneman, Sanjeev Khanna, and Wang Chiew Tan. 2001. Why and Where: A Characterization of Data Provenance. In *Database Theory - ICDT 2001, 8th International Conference, London, UK, January 4-6, 2001, Proceedings (Lecture Notes in Computer Science)*, Jan Van den Bussche and Victor Vianu (Eds.), Vol. 1973. Springer, 316–330. https://doi.org/10.1007/3-540-44503-X_20

[10] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink™: Stream and Batch Processing in a Single Engine. *IEEE Data Eng. Bull.* 38, 4 (2015), 28–38. http://sites.computer.org/debull/A15dec/p28.pdf

[11] James Cheney, Laura Chiticariu, and Wang Chiew Tan. 2009. Provenance in Databases: Why, How, and Where. *Found. Trends Databases* 1, 4 (2009), 379–474. https://doi.org/10.1561/1900000006

[12] Graham Cormode, Vladislav Shkapenyuk, Divesh Srivastava, and Bojian Xu. 2009. Forward Decay: A Practical Time Decay Model for Streaming Systems. In *Proceedings of the 25th International Conference on Data Engineering, ICDE 2009, March 29 2009 - April 2 2009, Shanghai, China*, Yannis E. Ioannidis, Dik Lun Lee, and Raymond T. Ng (Eds.). IEEE Computer Society, 138–149. https://doi.org/10.1109/ICDE.2009.65

[13] Gianpaolo Cugola and Alessandro Margara. 2012. Processing flows of information: From data stream to complex event processing. *ACM Comput. Surv.* 44, 3 (2012), 15:1–15:62. https://doi.org/10.1145/2187671.2187677

[14] Yingwei Cui, Jennifer Widom, and Janet L. Wiener. 2000. Tracing the lineage of view data in a warehousing environment. *ACM Trans. Database Syst.* 25, 2 (2000), 179–227. https://doi.org/10.1145/357775.357777

[15] Tamraparni Dasu, Rong Duan, and Divesh Srivastava. 2016. Data Quality for Temporal Streams. *IEEE Data Eng. Bull.* 39, 2 (2016), 78–92. http://sites.computer.org/debull/A16june/p78.pdf

[16] Nihal Dindar, Nesime Tatbul, Renée J. Miller, Laura M. Haas, and Irina Botan. 2013. Modeling the execution semantics of stream processing engines with SECRET. *VLDB J.* 22, 4 (2013), 421–446. https://doi.org/10.1007/S00778-012-0297-3

[17] EllieLockhart. [n.d.]. Metacritic—Rotten-Tomatoes-Controversial-Reviews-Dataset: A dataset of both controversial and non-controversial video game and film reviews, created with the intention of studying "review bombing" algorithmically. — github.com. https://github.com/EllieLockhart/Metacritic---Rotten-Tomatoes-Controversial-Reviews-Dataset. [Accessed 12-11-2024].

[18] João Esteves, Rosa Maria Costa, Yongluan Zhou, and Ana Almeida. 2023. An exploratory analysis of methods for real-time data deduplication in streaming processes. In *Proceedings of the 17th ACM International Conference on Distributed and Event-based Systems, DEBS 2023, Neuchatel, Switzerland, June 27-30, 2023*, Valerio Schiavoni, Marcelo Pasin, Bettina Kemme, and Etienne Rivière (Eds.). ACM, 91–102. https://doi.org/10.1145/3583678.3596898

[19] Peter M. Fischer, Kyumars Sheykh Esmaili, and Renée J. Miller. 2010. Stream schema: providing and exploiting static metadata for data stream processing. In *EDBT 2010, 13th International Conference on Extending Database Technology, Lausanne, Switzerland, March 22-26, 2010, Proceedings (ACM International Conference Proceeding Series)*, Ioana Manolescu, Stefano Spaccapietra, Jens Teubner, Masaru Kitsuregawa, Alain Léger, Felix Naumann, Anastasia Ailamaki, and Fatma Özcan (Eds.), Vol. 426. ACM, 207–218. https://doi.org/10.1145/1739041.1739068

[20] Apache Foundation. 2023. IoTDB. https://github.com/apache/iotdb/blob/master/library-udf/src/main/java/org/apache/iotdb/library/drepair/util/Screen.java. [Accessed 12-11-2024].

[21] Garima Gaur, Srikanta J. Bedathur, and Arnab Bhattacharya. 2017. Tracking the Impact of Fact Deletions on Knowledge Graph Queries using Provenance Polynomials. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management, CIKM 2017, Singapore, November 06 - 10, 2017*, Ee-Peng Lim, Marianne Winslett, Mark Sanderson, Ada Wai-Chee Fu, Jimeng Sun, J. Shane Culpepper, Eric Lo, Joyce C. Ho, Debora Donato, Rakesh Agrawal, Yu Zheng, Carlos Castillo, Aixin Sun, Vincent S. Tseng, and Chenliang Li (Eds.). ACM, 2079–2082. https://doi.org/10.1145/3132847.3133118

[22] Boris Glavic, Kyumars Sheykh Esmaili, Peter Michael Fischer, and Nesime Tatbul. 2013. Ariadne: managing fine-grained provenance on data streams. In *The 7th ACM International Conference on Distributed Event-Based Systems, DEBS '13, Arlington, TX, USA - June 29 - July 03, 2013*, Sharma Chakravarthy, Susan Darling Urban, Peter R. Pietzuch, and Elke A. Rundensteiner (Eds.). ACM, 39–50. https://doi.org/10.1145/2488222.2488256

[23] Lukasz Golab, Theodore Johnson, Nick Koudas, Divesh Srivastava, and David Toman. 2008. Optimizing away joins on data streams. In *Proceedings of the 2008 International Workshop on Scalable Stream Processing System, SSPS 2008, Nantes, France, March 29, 2008 (ACM International Conference Proceeding Series)*, Byung Suk Lee (Ed.). ACM, 48–57. https://doi.org/10.1145/1379272.1379282

[24] Todd J. Green, Gregory Karvounarakis, and Val Tannen. 2007. Provenance semirings. In *Proceedings of the Twenty-Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 11-13, 2007, Beijing, China*, Leonid Libkin (Ed.). ACM, 31–40. https://doi.org/10.1145/1265530.1265535

[25] Michael Grossniklaus, David Maier, James Miller, Sharmadha Moorthy, and Kristin Tufte. 2016. Frames: data-driven windows. In *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems, DEBS '16, Irvine, CA, USA, June 20 - 24, 2016*, Avigdor Gal, Matthias Weidlich, Vana Kalogeraki, and Nalini Venkasubramanian (Eds.). ACM, 13–24. https://doi.org/10.1145/2933267.2933304

[26] Ousmane Issa, Angela Bonifati, and Farouk Toumani. 2020. Evaluating Top-k Queries with Inconsistency Degrees. *Proc. VLDB Endow.* 13, 11 (2020), 2146–2158. http://www.vldb.org/pvldb/vol13/p2146-issa.pdf

[27] Paulo Jesus, Carlos Baquero, and Paulo Almeida. 2006. ID generation in mobile environments. In *CSMU 2006: Proceedings of the Conference on Mobile and Ubiquitous Systems*, Vol. 6.

[28] Grigoris Karvounarakis and Todd J. Green. 2012. Semiring-annotated data: queries and provenance? *SIGMOD Rec.* 41, 3 (2012), 5–14. https://doi.org/10.1145/2380776.2380778

[29] Grigoris Karvounarakis, Zachary G. Ives, and Val Tannen. 2010. Querying data provenance. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, Ahmed K. Elmagarmid and Divyakant Agrawal (Eds.). ACM, 951–962. https://doi.org/10.1145/1807167.1807269

[30] Jürgen Krämer and Bernhard Seeger. 2009. Semantics and implementation of continuous sliding window queries over data streams. *ACM Trans. Database Syst.* 34, 1 (2009), 4:1–4:49. https://doi.org/10.1145/1508857.1508861

[31] Samuele Langhi, Angela Bonifati, and Riccardo Tommasini. 2024. Towards Streaming Consistency Management. In *40th IEEE International Conference on Data Engineering, ICDE 2024, Utrecht, The Netherlands, May 13-16, 2024*. IEEE, 5663. https://doi.org/10.1109/ICDE60146.2024.00462

[32] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A. Tucker. 2005. Semantics and Evaluation Techniques for Window Aggregates in Data Streams. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, June 14-16, 2005*, Fatma Özcan (Ed.). ACM, 311–322. https://doi.org/10.1145/1066157.1066193

[33] Ester Livshits, Rina Kochirgan, Segev Tsur, Ihab F. Ilyas, Benny Kimelfeld, and Sudeepa Roy. 2021. Properties of Inconsistency Measures for Databases. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh

Srivastava (Eds.). ACM, 1182–1194. https://doi.org/10.1145/3448016.3457310

[34] Andrei Lopatenko and Leopoldo E. Bertossi. 2016. Complexity of Consistent Query Answering in Databases under Cardinality-Based and Incremental Repair Semantics (extended version). *CoRR* abs/1605.07159 (2016). arXiv:1605.07159 http://arxiv.org/abs/1605.07159

[35] Oleh Onyshchak. [n.d.]. Stock Market Dataset. https://www.kaggle.com/datasets/jacksoncrow/stock-market-dataset. [Accessed 12-11-2024].

[36] Issa Ousmane. 2021. INCA. https://github.com/semlanghi/coinca-sc. [Accessed 12-11-2024].

[37] Dimitris Palyvos-Giannas, Vincenzo Gulisano, and Marina Papatriantafilou. 2019. GeneaLog: Fine-grained data streaming provenance in cyber-physical systems. *Parallel Comput.* 89 (2019). https://doi.org/10.1016/J.PARCO.2019.102552

[38] Dimitris Palyvos-Giannas, Bastian Havers, Marina Papatriantafilou, and Vincenzo Gulisano. 2020. Ananke: A Streaming Framework for Live Forward Provenance. *Proc. VLDB Endow.* 14, 3 (2020), 391–403. https://doi.org/10.5555/3430915.3442437

[39] Dimitris Palyvos-Giannas, Katerina Tzompanaki, Marina Papatriantafilou, and Vincenzo Gulisano. 2022. Erebus: Explaining the Outputs of Data Streaming Queries. *Proc. VLDB Endow.* 16, 2 (2022), 230–242. https://doi.org/10.14778/3565816.3565825

[40] Chaoyi Pang, Junhu Wang, Yu Cheng, Hao Lan Zhang, and Tongliang Li. 2015. Topological sorts on DAGs. *Inf. Process. Lett.* 115, 2 (2015), 298–301. https://doi.org/10.1016/J.IPL.2014.09.031

[41] Eduardo H. M. Pena, Eduardo Cunha de Almeida, and Felix Naumann. 2021. Fast Detection of Denial Constraint Violations. *Proc. VLDB Endow.* 15, 4 (2021), 859–871. https://doi.org/10.14778/3503585.3503595

[42] Barna Saha and Divesh Srivastava. 2014. Data quality: The other face of Big Data. In *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014*, Isabel F. Cruz, Elena Ferrari, Yufei Tao, Elisa Bertino, and Goce Trajcevski (Eds.). IEEE Computer Society, 1294–1297. https://doi.org/10.1109/ICDE.2014.6816764

[43] Matthias J. Sax, Guozhang Wang, Matthias Weidlich, and Johann-Christoph Freytag. 2018. Streams and Tables: Two Sides of the Same Coin. In *Proceedings of the International Workshop on Real-Time Business Intelligence and Analytics, BIRTE 2018, Rio de Janeiro, Brazil, August 27, 2018*, Malú Castellanos, Panos K. Chrysanthis, Badrish Chandramouli, and Shimin Chen (Eds.). ACM, 1:1–1:10. https://doi.org/10.1145/3242153.3242155

[44] Shaoxu Song, Aoqian Zhang, Jianmin Wang, and Philip S. Yu. 2015. SCREEN: Stream Data Cleaning under Speed Constraints. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives (Eds.). ACM, 827–841. https://doi.org/10.1145/2723372.2723730

[45] Michael Stonebraker, Ugur Çetintemel, and Stanley B. Zdonik. 2005. The 8 requirements of real-time stream processing. *SIGMOD Rec.* 34, 4 (2005), 42–47. https://doi.org/10.1145/1107499.1107504

[46] Douglas B. Terry, David Goldberg, David A. Nichols, and Brian M. Oki. 1992. Continuous Queries over Append-Only Databases. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 2-5, 1992*, Michael Stonebraker (Ed.). ACM Press, 321–330. https://doi.org/10.1145/130283.130333

[47] Riccardo Tommasini, Pieter Bonte, Femke Ongenae, and Emanuele Della Valle. 2021. RSP4J: An API for RDF Stream Processing. In *The Semantic Web - 18th International Conference, ESWC 2021, Virtual Event, June 6-10, 2021, Proceedings (Lecture Notes in Computer Science)*, Ruben Verborgh, Katja Hose, Heiko Paulheim, Pierre-Antoine Champin, Maria Maleshkova, Óscar Corcho, Petar Ristoski, and Mehwish Alam (Eds.), Vol. 12731. Springer, 565–581. https://doi.org/10.1007/978-3-030-77385-4_34

[48] Juliane Verwiebe, Philipp M. Grulich, Jonas Traub, and Volker Markl. 2023. Survey of window types for aggregation in stream processing systems. *VLDB J.* 32, 5 (2023), 985–1011. https://doi.org/10.1007/S00778-022-00778-6

[49] Juliane Verwiebe, Philipp M. Grulich, Jonas Traub, and Volker Markl. 2023. Survey of window types for aggregation in stream processing systems. *VLDB J.* 32, 5 (2023), 985–1011. https://doi.org/10.1007/S00778-022-00778-6

[50] Guozhang Wang, Joel Koshy, Sriram Subramanian, Kartik Paramasivam, Mammad Zadeh, Neha Narkhede, Jun Rao, Jay Kreps, and Joe Stein. 2015. Building a Replicated Logging System with Apache Kafka. *Proc. VLDB Endow.* 8, 12 (2015), 1654–1655. https://doi.org/10.14778/2824032.2824063

[51] Wenyuan Yu. 2013. *Improving data quality : data consistency, deduplication, currency and accuracy.* Ph.D. Dissertation. University of Edinburgh, UK. https://hdl.handle.net/1842/8899

[52] Shuhao Zhang, Juan Soto, and Volker Markl. 2024. A survey on transactional stream processing. *VLDB J.* 33, 2 (2024), 451–479. https://doi.org/10.1007/S00778-023-00814-Z