# Toward Replicated and Asynchronous Data Streams for Edge-Cloud Applications

Owais Qayyum
o.qayyum@uit.no
UiT The Arctic University of Norway
Tromsø, Norway

Weihai Yu
weihai.yu@uit.no
UiT The Arctic University of Norway
Tromsø, Norway

## ABSTRACT

Vast number of computing devices, both in the cloud and at the edge, are continuously generating and consuming large amount of data. Some applications require the data to be always accessible, even when the devices are occasionally offline. We can make the data available by replicating and storing the generated data and query results locally at the devices. The data and results are kept updated when the devices are connected. The challenge, though, is to ensure that the generated data and query results are consistent. We present an approach to replicated and asynchronous data streams that is built on two well-established techniques: data provenance semirings and Conflict-Free Replicated Data Types (CRDTs). Our approach guarantees that the replicated and asynchronously updated data and query results are eventually consistent.

## CCS CONCEPTS

• **Computer systems organization → Availability**; • **Information systems → Relational parallel and distributed DBMSs**; **Database views**; **Stream management**.

## KEYWORDS

Incremental view maintenance, Stream processing, Data provenance, Conflict-Free Replicated Data Type, CRDT

**ACM Reference Format:**
Owais Qayyum and Weihai Yu. 2022. Toward Replicated and Asynchronous Data Streams for Edge-Cloud Applications. In *Proceedings of ACM SAC Conference (SAC'22).* ACM, New York, NY, USA, Article 4, 8 pages.
https: //doi.org/10.1145/3477314.3507687

## 1 INTRODUCTION

Today, vast number of computing devices are generating and consuming large amount of data. The computing devices range from powerful servers in the cloud to laptops, mobile phones and smartwatches at the edge. Some applications require the data to be always accessible, even when the devices are occasionally offline.

For example, an S&R (Search and Rescue) mission may involve multiple devices such as mobile phones, UAVs (Unnamed Aerial Vehicles) etc. Prior to the mission, the S&R team members may prepare the devices with mission-relevant data from their central database in the cloud. During the mission, the devices are kept updated to the latest situation through exchange of data between the devices and with the cloud. However, the devices might be offline due to poor network connectivity. Still, the devices should present up-to-date data whenever necessary.

In this paper, we focus on relational data. The generated data are stored in base relations. Data are consumed with queries. To make the data available at devices, we materialize the queries in terms of materialized views and replicate (base or view) relations. We keep the data up to date in a data-stream fashion.

According to the CAP theorem [4], it is impossible to simultaneously ensure all three desirable properties, namely (C) consistency equivalent to a single up-to-date copy of data, (A) availability of the data for update and (P) tolerance to network partition. Because our applications of interest require data availability during network partition, eventual consistency [5] is the strongest consistency we can achieve.

Stream data processing is a popular research field [10, 15, 17], where the focus has been on high throughput, low latency and scalability on large amount of data. Data are typically processed on a cluster of servers that are assumed to be constantly connected. Our work has a

We present an approach to sharing and querying data through replicated and asynchronous data streams. Generated data and query results (as materialized view) are replicated and stored in local devices. Data and query results are kept incrementally updated like data streams, but asynchronously only when the devices are connected. To make the replicated data and views eventually consistent, we adopt two well-established techniques, namely data provenance semirings [7] and CRDTs (Conflict-Free Replicated Data Types [14]). Our approach ensures eventual consistency of the replicated data and views.

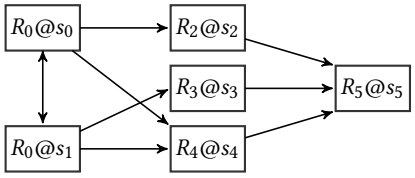
The paper is organized as the following. Section 2 uses an example to illustrate the research problem. Section 3 gives an overview of our approach. Section 4 presents preliminaries on semiring and lattice. Sections 5–7 present the two main elements of our approach, p-semiring and causal-length lattice, along with their properties. Section 8 describes the algorithms and shows that they guarantee eventual consistency. Section 9 presents some experimental results. Section 10 connects our approach to related work. Section 11 concludes.

## 2 PROBLEM STATEMENT

We illustrate the research issues with the example shown in Figure 1.


*SAC'22, April 25 –April 29, 2022, Brno, Czech Republic*

ACM ISBN 978-1-xxx. . . $15.00
https://doi.org/10.1145/3477314.3507687




$R_0(ABC)@s_0$

$R_0(ABC)@s_1 \leftarrow R_0@s_0$

$R_2(AB)@s_2 \leftarrow \pi_{AB}(\sigma_\theta R_0@s_0)$

$R_3(AB)@s_3 \leftarrow \pi_{AB}(\sigma_{\theta'} R_0@s_1)$

$R_4(BC)@s_4 \leftarrow \pi_{BC} R_0@\{s_0, s_1\}$

$R_5(ABC)@s_5 \leftarrow \{R_2@s_2, R_3@s_3\} \bowtie R_4@s_4$

**Figure 1: Base relations and views at different sites**

| $R_0$ | $R_2, R_3$ | $R_4$ | $R_5$ |
|---|---|---|---|
| initial states | | | |
| $\langle a_1, b_1, c_1\rangle$ | $\langle a_1, b_1\rangle$ | $\langle b_1, c_1\rangle$ | $\langle a_1, b_1, c_1\rangle$ |
| after inserting $\langle a_2, b_1, c_1\rangle$ in $R_0$ | | | |
| $\langle a_1, b_1, c_1\rangle$<br>$\langle a_2, b_1, c_1\rangle$ | $\langle a_1, b_1\rangle$<br>$\langle a_2, b_1\rangle$ | $\langle b_1, c_1\rangle$ | $\langle a_1, b_1, c_1\rangle$<br>$\langle a_2, b_1, c_1\rangle$ |
| after deleting $\langle a_2, b_1, c_1\rangle$ from $R_0$ | | | |
| $\langle a_1, b_1, c_1\rangle$ | $\langle a_1, b_1\rangle$ | $\langle b_1, c_1\rangle$ | $\langle a_1, b_1, c_1\rangle$ |

**Table 1: Updates in base relation and views**

In what follows, $s$, $s_1$ etc. are sites (we use the terms sites and devices interchangeably). $R(A, B, C)$, $R(ABC)$, or simply $R$, is a relation schema, where $A$, $B$ and $C$ are attributes. $R@s$ is an instance of $R$ at site $s$. $R@\{s_1, s_2\}$ is an instance of $R$ at site $s_1$ or site $s_2$, when, for example, relation $R$ is replicated at sites $s_1$ and $s_2$. Abusing the terms a little, we write $R$ also as an instance of $R$ when the site of the instance is obvious or unimportant from context.

In Figure 1, site $s_0$ continuously generates and updates data stored in base relation $R_0$. The data in $R_0$ are replicated at site $s_1$, which may make concurrently updates. Site $s_2$ makes a query on $R_0@s_0$ and site $s_3$ makes a query on $R_0@s_1$. Both $R_2$ at $s_2$ and $R_3$ at $s_3$ have the same attributes $A$ and $B$. Sites $s_4$ makes a query on $R_0$, from either $s_0$ or $s_1$, depending on network connectivity. Site $s_5$ makes a query that joins $R_2$ or $R_3$, depending on the network connectivity with $s_2$ and $s_3$, with $R_4$. The query results are stored locally as materialized views at sites $s_2$, $s_3$, $s_4$ and $s_5$.

The continuously generated and updated data at sites $s_0$ and $s_1$ are propagated to sites $s_2$, $s_3$, $s_4$ and $s_5$, which incrementally update their views locally. The data are propagated asynchronously, since the sites might occasionally get disconnected from one another.

In this example scenario, we are subject to two consistency requirements. The first requirement is that, the replicated instances of the base relation must be kept consistent. We can achieve eventual consistence of the replicated base relations with the help of CRRs (Conflict-free Replicated Relations [16]), which applies CRDTs (Conflict-free Replicated Data Types [14]) to relational data. With CRDT, a site updates its local replica and merges with incoming remote updates, without coordination with other sites. The states of the replicas converge when they have applied the same set of updates (referred to as *strong eventual consistency* in [14]).

The second requirement is that, the materialized views must be eventually consistent with respect to the states of the base relations. In other words, when the updates in the base relations have been propagated to the views, the states of the views should be the same as in a non-distributed setting given the same states of the base relations.

Table 1 shows an example of how the instances of the base relations and the views in Figure 1 get updated, assuming that $\theta = \theta' = \text{True}$ (i.e. no tuple in the base relation is filtered away). Initially the base relation has one tuple $\langle a_1, b_1, c_1\rangle$ and correspondingly each of the views has also one tuple. After the insertion and deletion of a tuple in the base relation, the views get updated accordingly.

Notice that after the insertion of tuple $\langle a_2, b_1, c_1\rangle$ in the base relation $R_0$, view $R_4$ remains unchanged, because tuple $\langle b_1, c_1\rangle$, which is supposed to be inserted, already exists in $R_4$. After the deletion of tuple $\langle a_2, b_1, c_1\rangle$ from $R_0$, $R_4$ still remains unchanged, because the tuple can be derived from $\langle a_1, b_1, c_1\rangle$ that remains in $R_0$. In the deletion case, by only using the state of $R_4$, it is impossible to decide whether the tuple $\langle b_1, c_1\rangle$ should be deleted or not.

Existing approaches to addressing the last issue is to augment the view states with additional metadata. For example, in their seminal work [8], Gupta et al associate each tuple in the view with a counter. In the above example, the counter value of tuple $\langle b_1, c_1\rangle$ in $R_4$ is initially 1. The value becomes 2 after the insertion of $\langle a_2, b_1, c_1\rangle$ and back to 1 after the deletion. Because the value is still a positive number, the tuple remains in the view.

These approaches work well in a non-distributed setting. The situation gets more complicated when the base relations and views are replicated and stored at different sites.

First, when a view receives an update from a base relation, the counter value of the resulting tuple cannot tell whether the view has already applied the same update received from a different replica. For example, if $s_4$ receives the deletion in the example from $s_1$, should it decrement the counter value? The answer depends on whether it has already received the same deletion from $s_0$.

Second, when a site receives two concurrent updates from different replicas, how does the site know which update wins? For example, if sites $s_0$ and $s_1$ get disconnected after the initial state. Site $s_0$ makes the insertion in the example. Concurrently, site $s1$ makes the same insertion and then the deletion. When site $s_4$ receives the insertion from $s_0$ and the deletion from $s_1$, how can $s_4$ tell that the deletion wins?

Third, when a site receives an update from a view, how does it know if some update from another site would influence its local state? For example, if site $s_5$ receives the insertion of $\langle a_2, b_1\rangle$ from $s_2$ and the deletion from $s_3$, how does $s_5$ tell the relationship between these updates? Should $s_5$ insert or delete tuple $\langle a_2, b_1, c_1\rangle$? Notice that $R_2$ and $R_3$ are not replicas and there is no collaboration between site $s_2$ and site $s_3$. Notice also that making a union of $R_2$ and $R_3$ at $s_5$ is not desirable, because receiving the deletion from $s_2$ implies

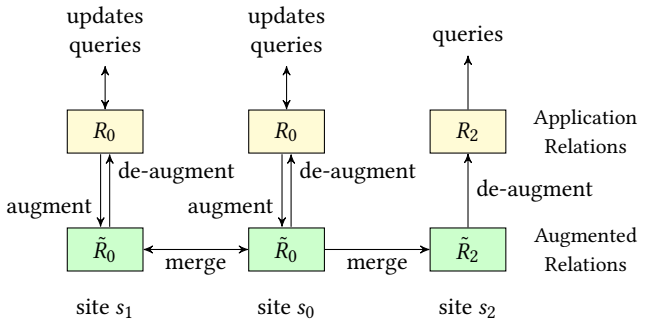



**Figure 2: Application and Augmented Relations**

that $\langle a_2, b_1, c_1\rangle$ has been deleted from $R_0$ even though the deletion has not been propagated from $s_3$ (via $s_1$) yet.

Essentially, we must be able to address the following problems:

(1) How does a tuple in a view relate to the dependent tuples in the base relations? This is a data provenance problem.
(2) How does the provenance relate to the concurrent updates in the base relations? This is a concurrency control problem concerning data provenance.

## 3 APPROACH OVERVIEW

To address the problems stated in Section 2, we adopt two algebraic structures:

- a idempotent commutative semiring for data provenance,
- a causal-length lattice for capturing causality of concurrent updates.

Our system consists of two layers: application-relation (APP) layer and augmented-relation (AUG) layer. Figure 2 shows three sites of the system. The sites correspond to sites $s_0$, $s_1$ and $s_2$ of Figure 1. The APP layer presents relations as a in conventional relational database system. Applications make updates and queries to application relations. The AUG layer associates the application-relation tuples with the meta data based on the two algebraic structures. It is the AUG layer that performs incremental maintenance of views and ensures that the consistency requirements (Section 2) are met.

In Figure 2, applications at sites $s_0$ and $s_1$ make concurrent updates to the base application relation $R_0$. The updates are augmented and stored in the augmented relation $\tilde{R}_0$. When sites $s_0$ and $s_1$ are connected, they send each other their local updates. When receiving a remote update, a site merges the update into its local $\tilde{R}_0$. It then updates the local $R_0$ with the de-augmented update resulted from the merge. Our system guarantees that when $s_0$ and $s_1$ have applied the same set of updates, regardless of the order in which the updates are applied, the two sites have the same $R_0$ and $\tilde{R}_0$ states.

Site $s_0$ sends its updates to $s_2$, when the two sites are connected. When receiving an update, site $s_2$ incrementally updates the augmented view $\tilde{R}_2$ and updates the application view $R_2$ with a corresponding de-augmented update. Our system guarantees that the state of $R_2$ is the same as in a non-distributed system given the same state of $R_0$. The same is guaranteed for all $R_3$, $R_4$ and $R_5$ in Figure 1, although they are not shown in Figure 2.

## System Model

We assume a distributed system consisting of sites that do not share memory. They maintain durable states. Sites may crash, but will eventually recover to the durable state at the time of the last crash.

A site can send messages to any other site in the system through an asynchronous and unreliable network. There is no upper bound on message delay. The network may discard, reorder or duplicate messages, but it cannot corrupt messages. Through re-sending, messages will eventually be delivered. The implication is that there can be network partitions, but disconnected sites will eventually get connected.

## Limitations

We currently focus on *named conjunctive relational algebra*, or SPJR (Select, Project, Join and Rename) algebra, extended with Union [1]. We leave negation and recursion to future work.

## 4 PRELIMINARIES: SEMIRING AND LATTICE

A *commutative semiring* is an algebraic structure $\langle P, +, \cdot, \mathbf{0}, \mathbf{1}\rangle$ such that:

- $P$ is a set and $\mathbf{0}, \mathbf{1} \in P$.
  In what follows, $p, p_1, p_2, p_3 \in P$.
- $(P, +, \mathbf{0})$ is a commutative monoid:
  - $\mathbf{0}$ is the unit element: $\mathbf{0} + p = p + \mathbf{0} = p$;
  - $+$ is associative: $(p_1 + p_2) + p_3 = p_1 + (p_2 + p_3)$;
  - $+$ is commutative: $p_1 + p_2 = p_2 + p_1$.
- $(P, \cdot, \mathbf{1})$ is a commutative monoid:
  - $\mathbf{1}$ is the unit element: $\mathbf{1} \cdot p = p \cdot \mathbf{1} = p$;
  - $\cdot$ is associative: $(p_1 \cdot p_2) \cdot p_3 = p_1 \cdot (p_2 \cdot p_3)$;
  - $\cdot$ is commutative: $p_1 \cdot p_2 = p_2 \cdot p_1$;
- $\mathbf{0}$ is an absorbing element of $\cdot$: $\mathbf{0} \cdot p = p \cdot \mathbf{0} = \mathbf{0}$.
- $\cdot$ left- and right-distributes over $+$:
  $p_1 \cdot (p_2 + p_3) = p_1 \cdot p_2 + p_1 \cdot p_3$ and
  $(p_1 + p_2) \cdot p_3 = p_1 \cdot p_3 + p_2 \cdot p_3$.

As a common convention, we may write $p_1 \cdot p_2$ as $p_1 p_2$.

A *lattice* is a partially ordered set where every subset of elements have a LUB (Least Upper Bound) and a GLB (Greatest Lower Bound). LUB is also known as a *join* and is written as $\sqcup$. GLB is also known as a *meet* and is written as $\sqcap$.

A lattice is *distributive* if $\sqcap$ distributes over $\sqcup$. That is,

$$p_1 \sqcap (p_2 \sqcup p_3) = p_1 \sqcap p_2 \sqcup p_1 \sqcap p_3$$

A *join-semilattice* is a partially ordered set where every subset of elements have a LUB.

## 5 PROVENANCE SEMIRING

The provenance of a tuple in a view describes how the tuple is derived. We model data provenance with a special semiring called p-semiring, where "p" stands for provenance.

A *p-semiring* is a commutative semiring with additional properties:

- $+$ is idempotent: $p + p = p$;
- $\cdot$ is idempotent: $p \cdot p = p$;
- $+$ applies to empty and infinitive sets: $\sum \emptyset = \mathbf{0}$ and $\sum P = \mathbf{1}$.

| $R(ABC)$ | $\tilde{R}(ABCP)$ | $\tilde{R}_1 \leftarrow \tilde{\pi}_{AB}\tilde{R}$ | $\tilde{R}_2 \leftarrow \tilde{\pi}_{BC}\tilde{R}$ | $\tilde{R}_3 \leftarrow \tilde{\pi}_{AC}(\tilde{R}_1 \tilde{\bowtie} \tilde{R}_2)$ | $\tilde{R}_3(P)$ after reduction |
|---|---|---|---|---|---|
| $\langle a_1, b, c_1\rangle$ | $\langle a_1, b, c_1, t_1\rangle$ | $\langle a_1, b, t_1 + t_2\rangle$ | $\langle b, c_1, t_1\rangle$ | $\langle a_1, c_1, (t_1 + t_2)t_1\rangle$ | $t_1$ |
| $\langle a_1, b, c_2\rangle$ | $\langle a_1, b, c_2, t_2\rangle$ | $\langle a_2, b, t_3\rangle$ | $\langle b, c_2, t_2 + t_3\rangle$ | $\langle a_1, c_2, (t_1 + t_2)(t_2 + t_3)\rangle$ | $t_1t_3 + t_2$ |
| $\langle a_2, b, c_2\rangle$ | $\langle a_2, b, c_2, t_3\rangle$ | | | $\langle a_2, c_1, t_3t_1\rangle$ | $t_1t_3$ |
| | | | | $\langle a_2, c_2, t_3(t_2 + t_3)\rangle$ | $t_3$ |

**Table 2: Example of p-semiring augmented relation instances**

A p-semiring in actually a c-semiring in [3] where $\cdot$ is idempotent.

For a relational database instance, a *p-semiring augmentation* is a function $\text{aug}: X \longrightarrow P$, where $X$ is the set of tuples and $P$ is the set of provenance expressions in a p-semiring. For an application-relation schema $R(ABC)$, the augmented-relation schema is $\tilde{R}(ABCP)$, where $P$ is an attribute for provenance expressions.

We construct provenance expressions as the following:

- For each tuple in a base application relation, we generate a globally unique id. The id is the provenance of the tuple. For a base application-relation tuple $\langle a, b\rangle$, the augmented-relation tuple is $\langle a, b, t\rangle$, where $t$ is the unique id of $\langle a, b\rangle$.
- For two augmented-relation tuples $\langle a, b, p_1\rangle$ and $\langle b, c, p_2\rangle$, the augmented-relation, as the result of a join, is $\langle a, b, c, p_1p_2\rangle$.
$$\{\langle a, b, p_1\rangle\} \tilde{\bowtie} \{\langle b, c, p_2\rangle\} = \{\langle a, b, c, p_1p_2\rangle\}$$
- For two augmented-relation tuples $\langle a, b, p_1\rangle$ and $\langle a, b, p_2\rangle$, the augmented-relation tuple, as the result of a union, is $\langle a, b, p_1 + p_2\rangle$.
$$\{\langle a, b, p_1\rangle\} \tilde{\cup} \{\langle a, b, p_2\rangle\} = \{\langle a, b, p_1 + p_2\rangle\}$$
- We handle duplicate elimination for a relation project as a union.
$$\tilde{\pi}_A\{\langle a, b_1, p_1\rangle, \langle a, b_2, p_2\rangle\} = \{\langle a, p_1 + p_2\rangle\}$$

Table 2 shows an example of p-semiring augmented relation instances. To understand how p-semiring provenance works, imagine for now that the tuple ids $t_1$ and $t_2$ give truth values of whether the corresponding tuples are present in some base relations, and operations + and $\cdot$ correspond to logical $\vee$ and $\wedge$. Expression $t_1 + t_2$ means that a tuple is present in the view when either tuple $t_1$ or tuple $t_2$ is present in a base relation instance. $t_1t_2$ means that a tuple is present in the view when both tuple $t_1$ and tuple $t_2$ are present in some base relation instances.

Below are some useful properties of p-semiring obtained from [3].

The properties of + allow us to define a partial order $\leq_p$: $p_1 \leq_p p_2$ iff $p_1 + p_2 = p_2$. This partial order has the following important properties:

$$\mathbf{0} \leq_p p_1p_2 \leq_p p_1 \leq_p p_1 + p_2 \leq_p \mathbf{1}$$

To understand the partial order, consider a particular p-semiring, a power set $2^S$, where $\leq_p$ is $\subseteq$, $\mathbf{0}$ is $\emptyset$ and $\mathbf{1}$ is $S$. It is not difficult to see that for $S_1, S_2 \in 2^S$, $\emptyset \subseteq S_1 \cap S_2 \subseteq S_1 \subseteq S_1 \cup S_2 \subseteq S$.

As the maximum element in $P$, $\mathbf{1}$ is an absorbing element of +. That is, $\mathbf{1} + p = p + \mathbf{1} = \mathbf{1}$.

Furthermore, [3] connects p-semiring with distributive lattice:

- + coincides with LUB: $p_1 + p_2 = p_1 \sqcup p_2$.
- $\cdot$ coincides with GLB: $p_1p_2 = p_1 \sqcap p_2$.
- + distributes over $\cdot$: $p_1 + p_2p_3 = (p_1 + p_2)(p_1 + p_3)$
- $\langle P, \leq_p\rangle$ is a distributive lattice.

Finally, there is a useful absorption property: $p_1 + p_1p_2 = p_1$.

We can use this absorption property to reduce provenance expressions. The rightmost column of Table 2 shows the example provenance expressions in $\tilde{R}_3$ after reduction.

## 6 CAUSAL-LENGTH LATTICE

The *causal-length lattice* [16] is defined on $\langle \text{cl}, \leq_{\text{cl}}\rangle$, where

- $\text{cl}: ID \longrightarrow \mathbb{N}$ is a function:
  The domain of the function $\text{dom}(\text{cl}) = ID \subset P$ is the set of tuple ids. $\text{cl}(t) = 0$ if id $t$ is not present in any base relation.
- $\leq_{\text{cl}}$ is a partial order:
  $\text{cl}_1 \leq_{\text{cl}} \text{cl}_2$ iff $\forall t \in ID$: $\text{cl}_1(t) \leq \text{cl}_2(t)$.
- LUB: $(\text{cl}_1 \sqcup \text{cl}_2)(t) = \max(\text{cl}_1(t), \text{cl}_2(t))$.
- GLB: $(\text{cl}_1 \sqcap \text{cl}_2)(t) = \min(\text{cl}_1(t), \text{cl}_2(t))$.

We use the causal-length lattice as the following.

When we insert a tuple in a base application relation for the first time, we generate an id $t$ of the tuple and associate it with causal length 1, i.e. $\text{cl}(t) = 1$. When we delete an existing tuple from a base application relation, we increment the causal length with 1, thus $\text{cl}(t) = 2$. When we insert the tuple back to the application relation, we also increment the causal length with 1, hence $\text{cl}(t) = 3$. The intuition behind this is that, for a given tuple, insertions and deletions occur in turn. When the causal length is an odd number, the tuple is last inserted and therefore is regarded as present in the base application relation. When the causal length is an even number, the tuple is last deleted and therefore is regarded as absent in the base application relation.

Now, consider two sites $s_1$ and $s_2$ concurrently update (insert or delete) the same tuple with id $t$. The updates are regarded as equivalent, if the tuple at the two sites has the same causal length (i.e., $\text{cl}_1(t) = \text{cl}_2(t)$). When the concurrent updates merge, the causal length remains unchanged. If, however, the causal length at one site $s_1$ is greater than the causal length at $s_2$ (i.e., $\text{cl}_1(t) > \text{cl}_2(t)$), we know that $s_1$ has already seen all the equivalent updates of $s_2$. When the concurrent updates merge, the update from $s_1$ wins. The new causal length becomes $\text{cl}_1(t) = \max(\text{cl}_1(t), \text{cl}_2(t))$.

## 7 VALUATION OF PROVENANCE EXPRESSIONS WITH CAUSAL LENGTHS

Now we can valuate provenance expressions with cause lengths, i.e., associate the tuple ids in provenance expressions with their causal lengths. The valuation allows us to answer the question whether a tuple is currently present in an application relation (base

relation or view) given the provenance of the corresponding tuple in the augmented relation.

Function $in?: P \times \mathsf{cl} \to \mathbb{B}$, where $P$ is the set of provenance expressions and cl is the causal-length function, answers the question whether a tuple, given a provenance expression, is currently present in the application relation. We define function in? recursively.

- $\mathsf{in?}(t, \mathsf{cl}) = \mathsf{odd?}(\mathsf{cl}(t))$, where $t$ is a tuple id and function $\mathsf{odd?}: \mathbb{N} \to \mathbb{B}$ tells whether a natural number is odd.
- $\mathsf{in?}(p_1 \cdot p_2, \mathsf{cl}) = \mathsf{in?}(p_1, \mathsf{cl}) \wedge \mathsf{in?}(p_2, \mathsf{cl})$.
- $\mathsf{in?}(p_1 + p_2, \mathsf{cl}) = \mathsf{in?}(p_1, \mathsf{cl}) \vee \mathsf{in?}(p_2, \mathsf{cl})$.

Suppose the causal-length function for the tuples in Table 2 is $\mathsf{cl} = \{t_1 \mapsto 1, t_2 \mapsto 2, t_3 \mapsto 3\}$, meaning that tuple $t_1$ has been inserted, tuple $t_2$ has been inserted and then deleted, and tuple $t_3$ has been inserted, then deleted and finally inserted back. To see whether tuple $\langle a_1, c_2 \rangle$ is currently present in $R_3$, we evaluate

$$\begin{aligned}\mathsf{in?}(t_1t_3 + t_2, \mathsf{cl}) &= \mathsf{in?}(t_1, \mathsf{cl}) \wedge \mathsf{in?}(t_3, \mathsf{cl}) \vee \mathsf{in?}(t_2, \mathsf{cl}) \\ &= \mathsf{odd?}(\mathsf{cl}(t_1)) \wedge \mathsf{odd?}(\mathsf{cl}(t_3)) \vee \mathsf{odd?}(\mathsf{cl}(t_2)) \\ &= \mathsf{odd?}(1) \wedge \mathsf{odd?}(3) \vee \mathsf{odd?}(2) \\ &= \mathsf{True} \wedge \mathsf{True} \vee \mathsf{False} \\ &= \mathsf{True}\end{aligned}$$

Therefore, tuple $\langle a_1, c_2 \rangle$ is currently present in $R_3$.

## 8 ALGORITHMS

A site maintains, for each application relation $R$ (base relation or view), an instance of $R$ and an instance of augmented relation $\tilde{R}$. For a view resulted from a join: $R = R_1 \bowtie R_2$, the site also maintains $\tilde{R}_1$ and $\tilde{R}_2$.

The site also maintains a relation $CL(ID, L)$ for the causal lengths of the base tuples that it has encountered. In addition, for each incoming or outgoing stream with schema $\tilde{R}(AP)$, the site maintains a relation $\tilde{R}_\Delta(IAP)$ for incoming or outgoing deltas (new updates), where $I$ is an attribute for sequence numbers. Finally, the site maintains an incoming and an outgoing stream of causal-length updates $CL^{in}_\Delta(I, ID, L)$ and $CL^{out}_\Delta(I, ID, L)$.

In the algorithms, we use the function (or mapping) $\tilde{\mathsf{r}}$ to represent provenance states of tuples in the augmented relation $\tilde{R}$ locally at a site. Thus, $\tilde{\mathsf{r}}(\langle a \rangle) = p$ is the provenance part of tuple $\langle a, p \rangle$ in relation $\tilde{R}$. $\tilde{\mathsf{r}}(\langle a \rangle) = \mathbf{0}$ when there does not exist a tuple $\langle a, p \rangle$ in $\tilde{R}$. For an augmented relation $\tilde{R}$ where $\tilde{\mathsf{r}}(\langle a \rangle) = p$, $\tilde{\mathsf{r}}\{\langle a \rangle \mapsto p'\}$ is an update of $\tilde{R}$ such that $\tilde{\mathsf{r}}(\langle a \rangle) = p'$ in the new state of $\tilde{R}$. Similarly, we use functions cl and $\mathsf{cl}_\Delta$ for causal-length states in relations $CL$ and $CL_\Delta$. $\mathsf{cl}(t) = 0$ when id $t$ is not present in relation $CL$. Also, $\mathsf{cl}(\mathbf{0}) = 0$. To simplify presentation, we ignore sequence numbers in $\tilde{\mathsf{r}}_\Delta$ and $\mathsf{cl}_\Delta$.

To insert tuple $\langle a \rangle$ in base application relation $R$ (algorithm 1), a site first inserts $\langle a \rangle$ in $R$ (line 1) and then updates the relations in the AUG layer (lines 2–9).

In the AUG layer, the site obtains from $\tilde{R}$ the provenance of $\langle a \rangle$, which should be a tuple id $t$, and the causal length of the tuple (line 2). If tuple $\langle a \rangle$ does not exist in $\tilde{R}$ (line 3), the site generates a new id and associates the id with $\langle a \rangle$ (line 4). The initial causal length of the tuple is 1 (line 4).

**Algorithm 1:** insert$(R, \langle a \rangle)$

```
1 R ← R ∪ {⟨a⟩};
2 t ← r̃(⟨a⟩); l ← cl(t);
3 if t = 0 then
4 |   t ← newId(); r̃ ← r̃{⟨a⟩ ↦ t}; cl ← cl{t ↦ 1};
5 |   r̃_Δ ← r̃_Δ{⟨a⟩ ↦ t}; cl^out_Δ ← cl^out_Δ{t ↦ 1};
6 else if even?(l) then
7 |   cl ← cl{t ↦ l + 1};
8 |   r̃_Δ ← r̃_Δ{⟨a⟩ ↦ t}; cl^out_Δ ← cl^out_Δ{t ↦ l + 1};
9 end
```

If tuple $\langle a \rangle$ already exists in $\tilde{R}$ and its causal length is an even number (line 6), $\langle a \rangle$ has been inserted and finally deleted. The site re-inserts $\langle a \rangle$ by simply incrementing its causal length with 1.

In both cases, the site updates the outgoing streams accordingly (lines 5 and 6).

If tuple $\langle a \rangle$ exists in $\tilde{R}$ and its causal length is an odd number, $\langle a \rangle$ is already present in $R$ before the insertion, so the insertion has no effect.

**Algorithm 2:** delete$(R, \langle a \rangle)$

```
1 R ← R \ {⟨a⟩};
2 t ← r̃(⟨a⟩); l ← cl(t);
3 if odd?(l) then
4 |   cl ← cl{t ↦ l + 1};
5 |   r̃_Δ ← r̃_Δ{⟨a⟩ ↦ t}; cl^out_Δ ← cl^out_Δ{t ↦ l + 1};
6 end
```

To delete tuple $\langle a \rangle$ from base application relation $R$ (algorithm 2), a site first deletes $\langle a \rangle$ from $R$ (line 1). Then in the AUG layer, it checks if the causal length of $\langle a \rangle$, $l$, is an odd number (line 3). If it is, it simply increments $l$ with 1 (line 4) and updates the outgoing streams (line 5). Otherwise ($l$ is an even number), $\langle a \rangle$ does not exist in $R$ before the deletion, so the deletion has no effect.

With algorithms 1 and 2, the state updates in base augmented relations are *inflationary*. That is, at a given site, for old states $\tilde{\mathsf{r}}$ and cl, and new states $\tilde{\mathsf{r}}'$ and $\mathsf{cl}'$, $\tilde{\mathsf{r}} \leq_p \tilde{\mathsf{r}}'$ (we extend $\leq_p$ to the entire relation $\tilde{\mathsf{r}}$) and $\mathsf{cl} \leq_{\mathsf{cl}} \mathsf{cl}'$.

**Algorithm 3:** merge$(\mathsf{op}, \tilde{\mathsf{r}}, \tilde{\mathsf{r}}'_\Delta, \mathsf{cl}^{in}_\Delta)$

```
1 r̃_new ← mergeRelation(op, r̃, r̃'_Δ, ...); r̃_Δ ← r̃_Δ ⊔ r̃_new;
2 cl ← cl ⊔ cl^in_Δ; cl^out_Δ ← cl^out_Δ ⊔ cl^in_Δ;
3 for ⟨a, p⟩ ∈ r̃_new do
4 |   if in?(p, cl) then R ← R ∪ {⟨a⟩} ;
5 |   else R ← R \ {⟨a⟩} ;
6 end
```

A site merges incoming updates with algorithm 3. First, it merges the augmented relation $\tilde{R}$ with the incoming updates and generates newly merged tuples in $\tilde{\mathsf{r}}_{new}$, which are merged into the outgoing stream $\tilde{\mathsf{r}}_\Delta$ (line 1). The merge with augmented relation $\tilde{R}$ applies

algorithms 4–5 depending on the operation op that produces the relation. The site also merges the incoming updates in causal lengths with $CL$ and $CL_\Delta^{out}$ (line 2). Finally, for every newly merged tuple in $\tilde{r}_{new}$, the site updates the application relation $R$ using the provenance of the tuple valuated with causal lengths (lines-3–6).

**Algorithm 4:** mergeRelation($\pi_A\sigma_\theta, \tilde{r}, \tilde{r}'_\Delta$)

```
1 r̃_new ← ∅;
2 for ⟨a, b, p⟩ ∈ r̃'_Δ do
3     if θ(a, b) then
4         p_new ← r̃(⟨a⟩) + p;
5         r̃ ← r̃{a ↦ p_new}; r̃_new ← r̃_new ∪ ⟨a, p_new⟩;
6     end
7 end
8 return r̃_new;
```

Algorithm 4 handles the merge of incoming updates of augmented relations for both select and project operations. In the algorithm, we assume that $R = \pi_A(\sigma_\theta R'(AB))$. For each incoming update represented with $\langle a, b, p\rangle$, if the selection condition $\theta(a, b)$ is true (line 3), we merge its provenance $p$ with the current provenance of tuple $\langle a\rangle$ (i.e. after the project $\pi_A$) in $\tilde{R}$ (lines 4–5). The provenance of $\langle a\rangle$ before the merge is $\mathbf{0}$ if it does not exist in $\tilde{R}$. We return the set of all newly merged tuples (line 8).

We can use algorithm 4 for replication of both base relations and views, because $R(A) = \pi_A(\sigma_{\mathsf{True}}R(A))$.

**Algorithm 5:** mergeRelation($\bowtie, \tilde{r}, \tilde{r}^1_\Delta, \tilde{r}^1, \tilde{r}^2$)

```
1 r̃¹ ← r̃¹ ⊔ r̃¹_Δ;
2 r̃_new ← ∅;
3 for ⟨a, b, p1⟩ ∈ r̃¹_Δ do
4     for ∀c, p2 : ⟨b, c, p2⟩ ∈ r̃² do
5         r̃_new ← r̃_new ∪ {⟨a, b, c, r̃(⟨a, b, c⟩) + p1p2⟩};
6     end
7 end
8 return r̃_new;
```

For simplicity, we assume in algorithm 5 that $R = R^1(AB) \bowtie R^2(BC)$. The algorithm merges the incoming updates in $R^1$. A more general multi-way join with updates from multiple incoming streams takes the form of semi-naïve evaluation of Datalog programs [1]. We can also combine select and join as in algorithm 4.

In algorithm 5, we first merge the updates with the locally maintained $\tilde{R}^1$ (line 1). Then for each incoming update, represented with $\langle a, b, p_1\rangle$ (line 3), and a matching tuple in $\tilde{R}^2$, represented with $\langle b, c, p_2\rangle$ (line 4), we generate a new tuple $\langle a, b, c\rangle$ with provenance $p_1p_2$, merged with the existing one in $\tilde{R}$ ($\mathbf{0}$ if non-existent) (line 5). Finally, we return the set of all newly merged tuples (line 6).

Since algorithms 3–5 only update $\tilde{R}$ and $CL$ states with the LUP operation $\sqcup$, state merges are commutative:

$$\begin{aligned}&\mathsf{merge}(\mathsf{merge}(\tilde{r}, \tilde{r}_\Delta, \mathsf{cl}_\Delta), \tilde{r}'_\Delta, \mathsf{cl}'_\Delta)\\ &= \mathsf{merge}(\mathsf{merge}(\tilde{r}, \tilde{r}'_\Delta, \mathsf{cl}'_\Delta), \tilde{r}_\Delta, \mathsf{cl}_\Delta)\\ &= \mathsf{merge}(\tilde{r}, \tilde{r} \sqcup \tilde{r}'_\Delta, \mathsf{cl}_\Delta \sqcup \mathsf{cl}'_\Delta)\end{aligned}$$

Therefore, for the same set of updates, merging them in different orders leads to the same resulting state.

State merges are also idempotent:

$$\mathsf{merge}(\mathsf{merge}(\tilde{r}, \tilde{r}_\Delta, \mathsf{cl}_\Delta), \tilde{r}_\Delta, \mathsf{cl}_\Delta) = \mathsf{merge}(\tilde{r}, \tilde{r}_\Delta, \mathsf{cl}_\Delta)$$

A site may receive the same update multiple times via different incoming streams. For example, $R_5@s_5$ in Figure 1 may receive the same update in $R_0@s_0$ via $R_2@s_2$, $R_3@s_3$ and $R_4@s_4$ . The idempotence property guarantees that merging the same update multiple times leads to the same resulting state.

The algorithms has the following consistency results.

Lemma 8.1. *Replicated base augmented relations are eventually consistent.*

Proof. Since updates are inflationary, and both p-semiring and causal-length function are lattices, a replicated base augmented relation is actually a state-based CRDT [14]. The states of a base augmented relation at different replicas eventually converge. Therefore, a replicated base augmented relation is eventually consistent. □

Lemma 8.2. *The augmented views are eventually consistent.*

Proof. Because the merges are commutative and idempotent, the final state of an augmented view depends only on the total set of updates it has merged, regardless of the order and number of times the updates are merged. Therefore the final state of the augmented view is the same as in a non-distributed system. □

Theorem 8.3. *The application relations are eventually consistent.*

Proof. This follows immediately from Lemmas 8.1 and 8.2. □

Corollary 8.4. *Replicated application relations are eventually consistent.*

Proof. Due to Lemma 8.1, replicated base relations are eventually consistent.

A view $R(A)$ replicated at sites $s_1$ and $s_2$ can be regarded as $R@s_1 \leftarrow \pi_A R@s_2$ and $R@s_2 \leftarrow \pi_A R@s_1$. Following Lemma 8.2, replicated views are eventually consistent. □

Since the final states of relation instances are independent of the order and the number of times the incoming updates are merged, the correctness of our system does not require the sequence numbers of the updates. The sequence numbers are useful for avoidance of unnecessary repeat of messages and for garbage collection of update streams.

## 9 EXPERIMENTS

We have implemented the algorithms in Section 8 as well as the classical counting algorithm by Gupta et al [8] for comparison. The purpose of the experiments is to study the performance overhead of our algorithms with regard to the classical non-distributed one. The implementation is in Elixir[1] (version 1.12.3, compiled with Erlang OTS 24). We implemented sites using Elixir GenServer[2] and used ETS[3] (Erlang Term Storage) tables for data storage. We ran the

[1] https://elixir-lang.org
[2] https://hexdocs.pm/elixir/1.12/GenServer.html
[3] https://erlang.org/doc/man/ets.html

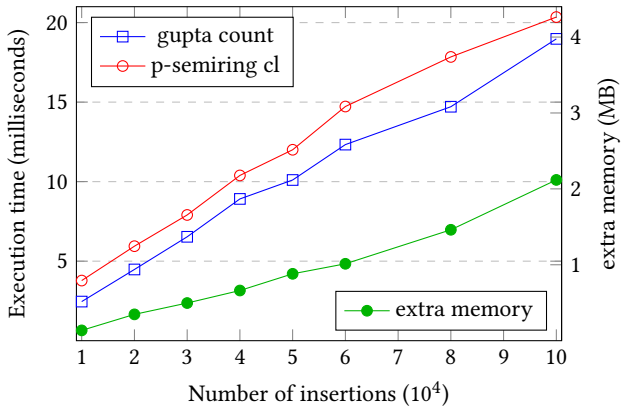



**Figure 3: Project insertions**

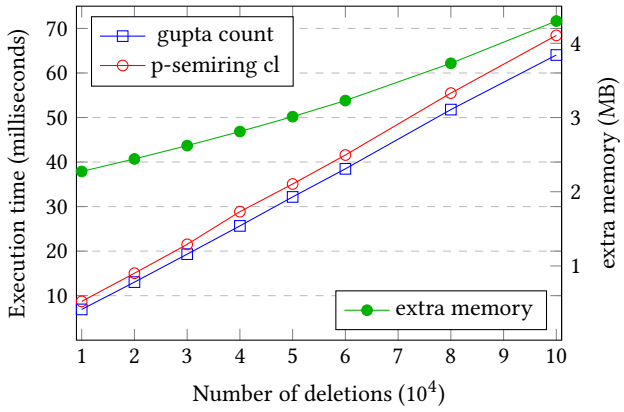



**Figure 4: Project deletions**

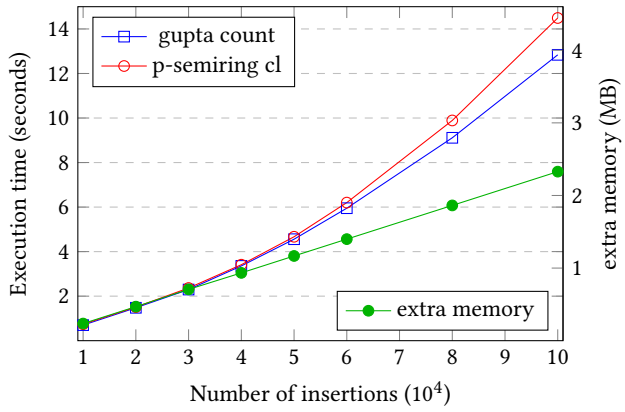



**Figure 5: Join insertions**

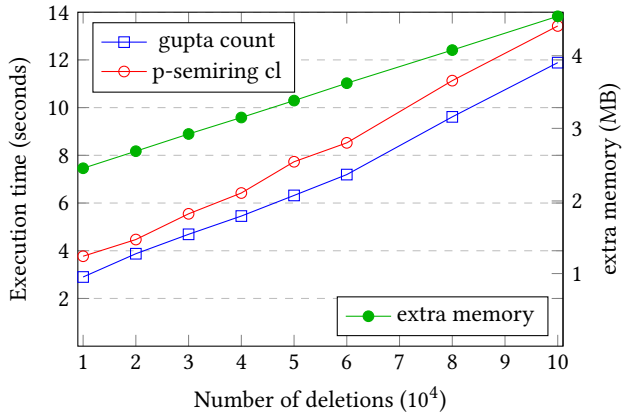



**Figure 6: Join deletions**

experiments on macOS with Apple M1 CPU (8 cores, 3.2GHz) and 16 GB of LPDDR4 (4266MHz) memory.

We use two base relations $R_1(\underline{A}BCDE)$ and $R_2(\underline{K}C)$ for our experiments, where attributes $A$ and $K$ are primary keys of relations $R_1$ and $R_2$ respectively. The experiments maintain two views $\pi_{BC}R_1$ and $R_1 \bowtie R_2$ incrementally. We run the experiments for both insertions and deletions from the incoming streams of the views. Figures 3, 4, 5 and 6 show the experimental results.

In the first experiment (Figure 3), we study the performance of maintaining view $\pi_{BC}R_1$ with incoming insertions. Each run starts with an empty view, and for a given number of insertions (ranging from 10k to 100k), we maintain the view incrementally. We repeat each run ten times with the same insertions. Figure 3 shows the average execution time of the runs and the extra memory overhead of our approach. The extra memory is the difference of totally memory used for the augmented view and for the Gupta counting algorithm. In the final view resulted from 100k insertions, there are around 72k tuples and our algorithm consumes 2.1MB more memory than the Gupta counting algorithm.

In the second experiment (Figure 4), we study the performance of maintaining view $\pi_{BC}R_1$ with incoming deletions. Each run starts with a view with around 72k tuples generated from 100k base tuples, and for a given number of deletions (ranging from 10k to 100k), we maintain the view incrementally. We repeat each run ten times with the same deletions. Figure 4 shows the average execution time of the runs and extra memory overhead. The memory overhead of our approach actually remains unchanged during the runs. The increase in extra memory is due to the fact that, with Gupta counting, tuples are deleted when the counter value of a tuple becomes zero.

In the third experiment (Figure 5), we study the performance of maintaining view $R_1 \bowtie R_2$ with incoming insertions. We fix the size of $R_2$ with 1000 tuples. Each run starts with an empty view, and for a given number of insertions in $R_1$ (ranging from 10k to 100k), we maintain the view incrementally. We repeat each run ten times with the same insertions. Figure 5 shows the average execution time of the runs and the extra memory overhead of our approach. In the final view resulted from 100k insertions, there are around 63k tuples, and our approach consumes 2.4MB more memory than the Gupta counting algorithm.

In the last experiment (Figure 6), we study the performance of maintaining view $R_1 \bowtie R_2$ with incoming deletions. Again, we fix the size of $R_2$ with 1000 tuples. Each run starts with a view of around 63k tuples resulted from 100k tuples from $R_1$, and for a

given number of deletions in $R_1$ (ranging from 10k to 100k), we maintain the view incrementally. We repeat each run ten times with the same deletions. Figure 6 shows the average execution time of the runs and the extra memory overhead of our approach.

## 10 RELATED WORK

*Incremental view maintenance.* Incremental maintenance of views is a well-studied problem [8, 12] for non-distributed database systems. For example, Gupta et al [8] keeps a count of the number of times a tuple is derived in the view (see also Section 2). Loo et al [11] maintains distributed datalog programs incrementally for FIFO connections. Nigam et al [13] extends the algorithms so that the connections are not necessarily FIFO. Our work is the first that supports eventual consistency when both base relations and views are replicated.

*Stream processing.* The mainstream research in the field of stream data processing [10, 15, 17], has focused on high throughput, low latency and scalability. Data are typically processed on a cluster of servers that are assumed to be constantly connected. Our work focuses more on availability during network partitions. We have not dealt with aggregates and windowed stream processing.

*Eventual consistency and coordination-freeness.* It is challenging to maintain data consistency while data are concurrently updated and consumed during network partition. Coordination-free algorithms with eventual consistence guarantee are therefore particularly attractive.

The CALM (Consistency And Logical Monotonicity) theorem [2, 9] states that a program has an eventually consistent, coordination-free execution strategy if and only if it is expressible in monotonic Dadatlog. Since the updates of our augmented relation instances are inflationary, we extend the applicability of the CALM theorem to systems with deletions.

CRDTs (Conflict Free Replicated Datatypes) [14] guarantee data convergence without coordination. We extend CRR (Conflict-free Replicated Relations) [16] which applies CRDTs to relational data. With the extension, not only do the states of the replicas that communicate with each other converge, but also do the views that might not communicate with each other.

*Data provenance.* Initial study of provenance for relational data can be found in [6]. Green et al [7] presents an elegant framework of provenance semirings. Our provenance semiring (Section 5) is actually c-semiring [3] with an additional property. The theorems in [3] allow us to connect semirings with lattices, hence also provenance with CRDTs.

## 11 CONCLUSION

We have presented replicated and asynchronous data streams, a new approach to sharing and querying data. Our work is particularly motivated by the applications where data and query results must be always available, even when the devices might be occasionally disconnected from the network. We focus on relational data and maintain original data in base relations and query results in materialized views. To make the data and query results always available at local devices, an application can replicate both base relations and materialized views. We keep the data updated, when the devices are connected, through asynchronous streaming and incremental view maintenance. We augment relations with two algebraic structures, p-semiring and causal-length lattice, and show that our algorithms guarantee eventual consistency of base relations and views. We have run experiments to study the performance overhead of our approach with regard to a classical non-distributed one.

Our current work is limited with conjunctive relational algebra extended with union. We leave negation and recursion to future work.

## REFERENCES


[1] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases.* Addison-Wesley.

[2] Tom J. Ameloot, Frank Neven, and Jan Van den Bussche. 2013. Relational transducers for declarative networking. *J. ACM* 60, 2 (2013), 15:1–15:38.

[3] Stefano Bistarelli, Ugo Montanari, and Francesca Rossi. 1997. Semiring-based constraint satisfaction and optimization. *J. ACM* 44, 2 (1997), 201–236.

[4] Eric Brewer. 2010. A Certain Freedom: Thoughts on the CAP Theorem. In *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing* (Zurich, Switzerland) *(PODC '10)*. Association for Computing Machinery, New York, NY, USA, 335. https://doi.org/10.1145/1835698.1835701

[5] Sebastian Burckhardt. 2014. *Principles of Eventual Consistency* (principles of eventual consistency ed.). Foundations and Trends® in Programming Languages, Vol. 1. Now Publishers. 1–150 pages. https://www.microsoft.com/en-us/research/publication/principles-of-eventual-consistency/

[6] 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 (June 2000), 179–227. https://doi.org/10.1145/357775.357777

[7] Todd J. Green, Grigoris Karvounarakis, and Val Tannen. 2007. Provenance Semirings. In *Proceedings of the Twenty-Sixth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems* (Beijing, China) *(PODS '07)*. Association for Computing Machinery, New York, NY, USA, 31–40. https://doi.org/10.1145/1265530.1265535

[8] Ashish Gupta, Inderpal Mumick, and V. Subrahmanian. 1993. Maintaining views incrementally. *Sigmod Record* 22 (06 1993), 157–166. https://doi.org/10.1145/170035.170066

[9] Joseph M. Hellerstein. 2010. The Declarative Imperative: Experiences and Conjectures in Distributed Logic. *SIGMOD Rec.* 39, 1 (Sept. 2010), 5–19. https://doi.org/10.1145/1860702.1860704

[10] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthik Ramasamy, and Siddarth Taneja. 2015. Twitter Heron: Stream Processing at Scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 239–250. https://doi.org/10.1145/2723372.2742788

[11] Boon Thau Loo, Tyson Condie, Minos Garofalakis, David E. Gay, Joseph M. Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. 2009. Declarative Networking. *Commun. ACM* 52, 11 (Nov. 2009), 87–95. https://doi.org/10.1145/1592761.1592785

[12] Boris Motik, Yavor Nenov, Robert Piro, and Ian Horrocks. 2019. Maintenance of datalog materialisations revisited. *Artif. Intell.* 269 (2019), 76–136. https://doi.org/10.1016/j.artint.2018.12.004

[13] Vivek Nigam, Limin Jia, Boon Thau Loo, and Andre Scedrov. 2012. Maintaining Distributed Logic Programs Incrementally. *Comput. Lang. Syst. Struct.* 38, 2 (July 2012), 158–180. https://doi.org/10.1016/j.cl.2012.02.001

[14] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-Free Replicated Data Types. In *Stabilization, Safety, and Security of Distributed Systems*, Xavier Défago, Franck Petit, and Vincent Villain (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 386–400.

[15] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, et al. 2014. Storm@ twitter. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. Association for Computing Machinery, New York, NY, USA, 147–156.

[16] Weihai Yu and Claudia-Lavinia Ignat. 2020. Conflict-Free Replicated Relations for Multi-Synchronous Database Management at Edge. In *IEEE International Conference on Smart Data Services*. IEEE, Beijing, China, 113–121. https://hal.inria.fr/hal-02983557

[17] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: A Unified Engine for Big Data Processing. *Commun. ACM* 59, 11 (Oct. 2016), 56–65. https://doi.org/10.1145/2934664