TPLP: Page 1–18.

doi:10.1017/S1471068426100556

*A Datalog Framework for Conflict-Free Replicated Data Types**

ELENA YANAKIEVA and ANNETTE BIENIUSA
RPTU, Germany
(*e-mails:* elena.yanakieva@cs.rptu.de, bieniusa@cs.uni-kl.de)

STEFANIA DUMBRAVA
ENSIIE, France
Inria Research Centre of Paris, France
IRIF, France
SAMOVAR, France
(*e-mails:* stefania.dumbrava@ensiie.fr)

submitted 29 May 2026 UTC; revised 29 May 2026 UTC; accepted 1 June 2026 UTC

Abstract

Distributed applications increasingly support local-first collaboration over shared data, allowing multiple users to perform updates concurrently without global coordination. Such collaboration requires careful design to capture the intended semantics of the concurrent interactions. We introduce a declarative framework for specifying and reasoning about the semantics of conflict-free replicated data types (CRDTs) and CRDT-based applications in Datalog. The framework models CRDT semantics as executable logic programs over operation contexts, making concurrency explicit and compositional, and thus amenable to automated analysis. As one application, we use property-based testing to compare implementations. To the best of our knowledge, this is the first work to systematically use Datalog as a foundation for prototyping and analyzing complex CRDTs and their compositions. We evaluate our methodology using a collaborative graph data editing case study and report experimentation results assessing correctness validation and scalability with an increasing number of operations and replicas.

Keywords: logic programming, conflict-free replicated data types, local-first, CRDTs

1 Introduction

Distributed applications increasingly support collaboration over shared data, allowing multiple users to update the same data concurrently. This raises a fundamental question: when concurrent updates conflict, to which state should the application converge? To ensure predictable and well-defined behavior, developers must therefore specify application-specific conflict semantics. From a semantic perspective, the core challenge

* This work was partially supported by the VERDI grant ANR-24-CE25-1109 (Dumbrava).

is, thus, not only convergence itself but also the lack of explicit, analyzable specifications that explain why a particular concurrent outcome arises.

This challenge is amplified by the need for high availability. To enable offline work and low-latency local updates, collaborative systems must weaken consistency guarantees (Gilbert and Lynch 2002) and reconcile concurrent updates afterwards. Conflict-free Replicated Data Types (CRDTs) address this need by providing replicated data structures with deterministic convergence and SEC (Shapiro *et al.* 2011). However, building applications from CRDTs remains difficult in practice. Developers typically combine multiple data structures, but conflict-resolution policies do not compose predictably. Subtle concurrency patterns can cause otherwise correct CRDTs to converge to states that violate the intended application semantics. While CRDTs guarantee deterministic convergence, they do not guarantee that composed CRDTs converge to a state consistent with the intended application semantics.

Our work addresses this problem by providing a declarative modeling framework and tool that supports developers in exploring the composability of CRDT semantics.

Related work. Dedalus (Alvaro *et al.* (2010)) introduces a temporal Datalog-based formalism for reasoning about replicated mutable state and asynchronous communication in long-running distributed systems, whereas our focus is on executable semantic specifications for replicated data types and their compositions. Building on these ideas, the Bloom language (Alvaro *et al.* (2011)) proposes a declarative framework for reasoning about distributed programs through monotonic programming over sets, assuming a fixed merge operator (union) and a containment-based partial order.

Subsequent systems explored deterministic and coordination-free programming models using lattice-based structures. LVars (Kuper and Newton (2013)) introduced deterministic-by-construction parallelism over lattices, while Lasp (Meiklejohn and Roy (2015)) proposed a functional programming model for composing state-based CRDTs into larger computations that satisfy SEC. More recently, CALM theorem (Laddad *et al.* (2022)) established that logically monotonic programs achieve consistency without coordination, reinforcing Datalog as a natural foundation for such reasoning.

The Peepul approach (Soundarapandian *et al.* (2022)) certifies MRDTs with machine-checked convergence proofs via F* and SMT solving, targeting individual CRDT correctness rather than application-level semantics across compositions. Their approach complements ours: while they derive verified implementations, we make semantic mismatches between intended behavior and CRDT compositions explicit, executable, and falsifiable.

The work in Pandey *et al.* (2025) analyzes the challenges of applying replicated data types to property graph databases in local-first settings, identifying how graph-specific constraints interact with concurrent updates; our framework provides a concrete methodology for specifying and testing such interactions.

Our approach. To avoid unintended behavior, developers need to be able to state the intended semantics of an application independently of a particular CRDT design. We propose a workflow in which developers first write a declarative, executable reference specification of the application semantics and then construct an implementation via CRDT composition. Our framework supports systematic testing: given generated concurrent executions, it checks whether a composite realization converges to the same state

as the reference specification using property-based testing. While formal verification of distributed applications remains expert-level and costly, executable specifications enable lightweight, systematic testing readily usable during design.

Datalog is central to our work. First, it provides a concise and declarative way to express CRDT semantics as relations over events and visibility. Second, its monotonic, bottom-up evaluation model aligns naturally with CRDT semantics, where state is derived deterministically from a growing set of observed events. Third, Datalog specifications are executable, enabling systematic testing without committing to a concrete distributed implementation.

We implement the framework in CRDTLog, a Datalog-based system built on the Soufflé engine. CRDTLog provides a library of declarative specifications for basic, nested, and composite CRDTs, as well as tools to generate operation contexts and compare semantic outcomes. *To the best of our knowledge, this is the first work to systematically use Datalog as a foundation for prototyping and analyzing complex CRDTs and their compositions.* We make the following contributions:

- We present an extended declarative framework for specifying the semantics of CRDTs and collaborative applications based on operation contexts.
- We implement the framework in Soufflé and provide a library of classical, nested, and composite CRDT specifications, including two CRDT-based composite realizations of collaborative graph applications.
- We present a case study on collaborative graphs that demonstrates how abstract semantic specifications support CRDT composition choices and expose non-obvious concurrency behaviors.
- We evaluate the approach empirically, assessing semantic equivalence between abstract specifications and composite realizations across a set of generated executions.

Although our prototype is implemented using the Soufflé Datalog engine, the approach itself is not tied to a specific system. CRDTlog relies on language features, such as support for recursion, stratified negation, and aggregates which are provided by a wide range of open-source Datalog engines (e.g., DLV, Nemo, Logica) as well as by Prolog systems with tabling. Soufflé was chosen for its performance and mature tooling, but any engine integrating these features can serve as a backend for our framework. To provide a better playground for designers, we implemented our framework also in DDlog (Ryzhyk and Budiu (2019)), an incremental Datalog engine that allows users to interactively test use cases. A formal user study is beyond the current scope and is deferred to future work.

Additional proofs, and experimental details are provided in the supplemental material.

2 Background: Collaborative data structures

Our work is motivated by *collaborative applications* that require data structures to be shared and updated by multiple users concurrently under weak connectivity and without global coordination. As an example, consider a (directed) graph (V, A) where V is a set of nodes and $A \subseteq V \times V$ is a set of directed edges, as used in a collaborative design canvas or shared knowledge bases. Users may modify the graph by adding (and removing) nodes and edges; to simplify the presentation, we focus here on the structural updates and do

Table 1. *Operations for a directed graph with isolate-delete (ID) semantics; only isolated, non-connected nodes may be removed*

Operation	Precondition	Postcondition	Return value
`hasN`(v)	-	$V' = V \wedge A' = A$	$v \in V$
`hasE`(u,v)	-	$V' = V \wedge A' = A$	$(u, v) \in A$
`addN`(v)	-	$V' = V \cup \{v\} \wedge A' = A$	void
`addE`(u,v)	$u \in V \wedge v \in V$	$A' = A \cup \{(u, v)\} \wedge V' = V$	void
`rmvE`(u,v)	-	$A' = A \setminus \{(u, v)\} \wedge V' = V$	void
`rmvN`(v)	$v \in V \wedge \forall x \in V : (x, v) \notin A \wedge (v, x) \notin A$	$V' = V \setminus \{v\} \wedge A' = A$	void

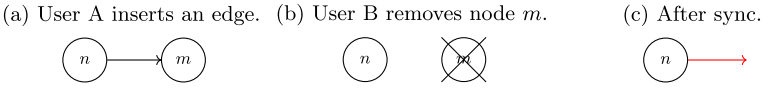


Figure 1. Dangling edge scenario.

not consider modifications of labels. Table 1 summarizes the semantics of such a graph with *isolate-delete (ID)* semantics.

Additionally, we require an application-level invariant, which guarantees the absence of dangling edges: $\forall (u, v) \in A : \; u \in V \wedge v \in V$.

In a single-user, sequential setting, this invariant can be enforced directly by the operations. However, under concurrent updates by multiple users, preconditions are not stable, leading to invariant violations. We illustrate such a violation in Figure 1. Consider a graph with two nodes n and m. User A executes `addE`(n,m) while user B concurrently executes `rmvN`(m). To avoid an invalid graph state, a conflict-resolution strategy is required to preserve the invariant. One possible outcome is to remove m and discard the edge; another is to discard the removal of m. No resolution is universally correct: the appropriate behavior depends on the intended application semantics.

Developers must therefore specify how conflicting operations should be interpreted and combined while preserving availability to enable offline execution and immediate local reasoning. The CAP theorem states that, in the presence of network partitions, strong consistency and availability cannot be guaranteed simultaneously (Gilbert and Lynch (2002)). Consequently, local-first collaborative systems require an explicit and declarative choice of *conflict semantics*.

CRDTs provide a standard approach to implementing highly available replicated state with deterministic convergence (Shapiro *et al.* 2011; Borth *et al.* 2025). Informally, CRDTs guarantee SEC: once all replicas have received the same set of updates, they converge to the same state. Several systems expose CRDTs to application developers, including Yjs (Nicolaescu *et al.* 2015) and Automerge (Kleppmann 2017). These frameworks provide a collection of basic replicated data types, such as sets, maps, sequences, counters, and registers, each equipped with a specific conflict-resolution policy.

Throughout this paper, we refer to common policies for datatypes such as:

- *add-wins* sets: concurrent `add`(v) and `del`(v) leave `v` present;
- *update-wins* maps: concurrent `upd`(k,v) and `rmv`(k) preserve the binding for `k` when the entry is concurrently updated.

These policies are useful primitives, but they are not neutral; each one encodes a specific choice about user intent under concurrency.

Real applications rarely consist of a single basic CRDT. Instead, developers compose and nest CRDTs to obtain richer behavior. However, conflict resolution does not automatically compose intuitively (Kuessner *et al.* 2023): combining CRDTs can introduce unintended behaviors that may only surface under subtle concurrent execution patterns.

This motivates a tool-supported workflow in which developers specify application semantics abstractly, explore alternative CRDT decompositions, and verify that the chosen design matches the intended behavior. Ideally, this relies on an executable specification enabling systematic exploration of concurrent user interactions. The remainder of the paper develops such a framework and implements it in Soufflé (Jordan *et al.* 2016).

Our framework builds on the declarative model of replicated data types by Burckhardt (2014), which defines each data type as a deterministic function over an operation context, in which events and their relation are captured. Note that we generalize the definition of an operation context such that each event can be mapped to a set of operations to model that the atomic execution of all operations of one event as needed in Subsection 3.4.

Definition 1 (Operation context and replicated data type).
Let E be a set of events, Op_X be the set of operations for a replicated data type X and Values be a set of possible return values. An *operation context* is a finite event graph $C = (\mathrm{E}, \mathrm{op}, \mathrm{vis}, \mathrm{ar})$, where $\mathrm{op} : \mathrm{E} \to \mathcal{P}(Op_X)$ is the set of operations of each event. The operations in the set are required to be commutative and associative so that the order of execution does not matter. *vis* is an acyclic relation representing visibility among the elements of E, and ar is a total order representing the arbitration of the elements in E.

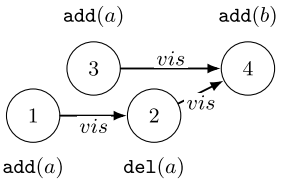
A *replicated data type* $\mathcal{F}_\mathrm{X} : Op_X \times C \to \mathsf{Values}$ is a (deterministic) function that, given an operation o and an operation context C, specifies the expected return value $\mathcal{F}_\mathrm{X}(o, C)$ when performing operation o in context C.

$\mathcal{F}_\mathrm{X}$ is a declarative specification that maps an operation context to the result of each operation. Unlike Burckhardt (2014), our model restricts contexts to update operations. Queries (e.g. `hasN`(v)) are treated as post hoc observers evaluated over the context to obtain the final state. Although concurrency is modeled via the visibility relation, the evaluation of $\mathcal{F}_\mathrm{X}$ is deterministic.

In an add-wins set, an element v is present if there exists an $\texttt{add}(v)$ event e_1 and no $\texttt{del}(v)$ event e_2 that observes it (i.e. e_1 is not visible to e_2). Thus, a delete concurrent with the add does not remove v, capturing add-wins semantics. Formally:

$$\begin{aligned} &\mathcal{F}_{\mathrm{SetAW}}(\texttt{hasV}(v), (\mathrm{E}, \mathrm{op}, \mathrm{vis}, \mathrm{ar})) = \mathrm{true} \\ &\iff \exists e \in E : \texttt{add}(v) \in \mathrm{op}(e) \wedge \neg(\exists e' \in E : \texttt{del}(v) \in \mathrm{op}(e') \wedge e \xrightarrow{\mathrm{vis}} e') \end{aligned} \tag{1}$$

Figure 2 illustrates an example operation context for an add-wins set. The context contains four events: event 1 performs $\texttt{add}(a)$, event 2 $\texttt{del}(a)$, event 3 $\texttt{add}(a)$, and event 4 performs $\texttt{add}(b)$. The visibility relation includes edges $1 \to 2, 3 \to 4$ and $2 \to 4$, meaning that event 2 observes event 1; event 4 observes events 3, 2, and transitively also 1. Event 3 is concurrent with events 1 and 2. As arbitration order is irrelevant for an add-wins set, it is omitted. The resulting set contains a and b, since the concurrent $\texttt{add}(a)$ in event 3 overrules $\texttt{del}(a)$ in event 2.




```
setEvent(1,add,a). setEvent(2,remove,a).
setEvent(3,add,a). setEvent(4,add,b).
vis(1,2). vis(3,4). vis(2,4).
visible(E1,E2) :- vis(E1,E2).
visible(E1,E2) :- visible(E1,E3), vis(E3,E2).
conc(E1,E2) :- setEvent(E1,_,_), setEvent(E2,_,_),
               not visible(E1,E2),
               not visible(E2,E1), E1/=E2.
```

Figure 2. Example operation context for an add-wins set and its Datalog encoding.

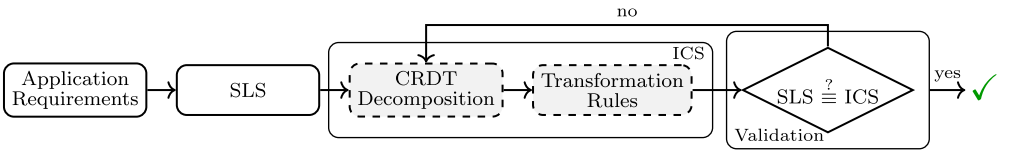


Figure 3. Workflow for deriving and validating CRDT compositions. From application requirements, the user defines the SLS, decomposes it into CRDTs with transformation rules (ICS), and validates equivalence. Mismatches require revising the decomposition.

3 A declarative framework for CRDT specification

We introduce CRDTLog,[1] an implementation of the declarative approach to CRDTs from Section 2. It extends the framework with nested and composite CRDTs, enabling designers to model CRDT-based applications and reason about their semantics. This approach is generic: users can define arbitrary CRDTs with custom conflict resolution strategies. For instance, a delete-wins set, where any observed deletion takes precedence, can be captured by the specification in equation (2).

$$\mathcal{F}_{\text{SetDW}}(\texttt{hasV}(v), (\text{E}, \text{op}, \text{vis}, \text{ar})) = \text{true} \iff \neg(\exists e \in E : \texttt{del}(v) \in \text{op}(e)) \tag{2}$$

CRDTLog is implemented in Soufflé, a Datalog engine with efficient, scalable bottom-up evaluation (Jordan *et al.* 2016). For readability, we present the theory and its Datalog implementation side by side: we represent operation contexts as input relations and derive return values (for queries) as output relations.

Our framework provides a "playground" for the design phase (Figure 3). Developers first write the *SLS*, a high-level specification of the intended application semantics, then propose the *implementation compositional semantics (ICS)*: a CRDT composition with transformation rules mapping application operations to component operations. Both are executable over generated operation contexts, enabling systematic exploration of corner cases and property-based equivalence checks. If equivalence fails, the decomposition is revised and the process iterates. Crucially, the framework isolates the semantic core, that is how conflicts are resolved and how composed CRDTs interact, while abstracting away protocol and engineering choices, which can be deferred until after the semantic design is validated.

Throughout this section, we refer to the collaborative graph introduced in Section 2 and summarized in Table 1 as a running example to illustrate the workflow. The operations of the used datatypes can be found in Table 2.

[1] CRDTLog's code is available at https://github.com/elly-yanakieva/CRDTLog.

Table 2. *Summary of datatype operations*

Datatype	Update operations	Query operations
Set	`add`(v), `del`(v)	`hasV`(v)
Map⟨Set⟩	`upd`(k,`add`(v)), `upd`(k,`del`(v)), `rmv`(k)	`hasKV`(k,v)
Graph	`addN`(n), `rmvN`(n), `addE`(n,m), `rmvE`(n,m)	`hasN`(n), `hasE`(n,m)

```
postVisOp(E1) :- setEvent(E1, _, X), setEvent(E2, "del", X), visible(E1, E2), E1 \= E2.
setState(X)   :- setEvent(E, "add", X), not postVisOp(E).
```

Listing 1. Add-wins set semantics.

3.1 Operation contexts in Datalog

We encode an operation context in CRDTLog using three input relations: one for events and their operations, `vis` for the visibility relation, and `ar` for the arbitration order.

CRDT systems typically derive visibility, a.k.a. happens-before, from metadata such as vector clocks, and arbitration order from timestamps. The arbitration order is used only when semantics require a total order on concurrent events (e.g. last-writer-wins); otherwise, conflict resolution relies solely on visibility.

We illustrate the operation context of the add-wins set from Figure 2. Event facts in Datalog refer to an event identifier, the operation name, and its parameters. For example, event `1` applies `add` with parameter `a`. For readability, the input relation `vis` contains only direct visibility edges, and we compute its transitive closure to obtain full visibility. Events are considered concurrent when they are not connected by the visibility relation.

In general, CRDT correctness is argued via *strong eventual consistency*: replicas that have (eventually) observed the same set of updates must converge to the same state. In our setting, concurrency and update propagation is captured explicitly in the operation context which we assume to be well-formed and identical across replicas once they have observed the same updates. Each data type X is then defined by function $\mathcal{F}_X$ over this context. Consequently, any two replicas that evaluate $\mathcal{F}_X$ on the same operation context derive the same output relations, yielding convergence by construction. Since our Datalog programs implement $\mathcal{F}_X$ as rule evaluation over the relations representing the context, they inherit this determinism directly.

3.2 CRDT encoding in Datalog

In Section 2, we introduced declarative semantics for basic CRDTs, such as the add-wins set (Equation (1)). We express these semantics as Datalog rules that derive output relations representing the observable state at the end of the operation context.

Listing 1 gives the Datalog encoding of the add-wins set. Events are represented by `setEvent(E,O,X)`, where `E` is an event identifier, `O` an operation, and `X` a value. The predicate `setState(X)` computes `hasV(X)` over the operation context. A value `X` is in the set if and only if there exists an `add(X)` that is not canceled by a later `del(X)` (`not postVisOp`).

$$\begin{aligned} \texttt{MapSet.upd}(k, \texttt{add}(v)) &\overset{\mathcal{T}}{\Rightarrow} \{\texttt{KeySet.add}(k), \texttt{ValSet.add}(v)\} \\ \texttt{MapSet.upd}(k, \texttt{del}(v)) &\overset{\mathcal{T}}{\Rightarrow} \{\texttt{KeySet.add}(k), \texttt{ValSet.del}(v)\} \\ \texttt{MapSet.rmv}(k) &\overset{\mathcal{T}}{\Rightarrow} \{\texttt{KeySet.del}(k)\} \end{aligned}$$

Figure 4. Layered transformation for update-wins Map⟨Set⟩.

```
keysEvent(E, "add", K) :- mapSetEvent(E, "upd",  _, K, _).
keysEvent(E, "del", K) :- mapSetEvent(E, "rmv", _, K, _).
setEvent(E, InnerOp, X, K) :- mapSetEvent(E, "upd", InnerOp, K, X), not postVisRemove(E).
mapState(K, X)  :- keysState(K), setState(K, X).
mapState(K, "") :- keysState(K), not setState(K, _).
```

Listing 2. Map⟨Set⟩ semantics expressed in Datalog.

Collaborative applications typically compose basic or nested CRDTs into composite designs (Kuessner *et al.* 2023). Accordingly, CRDTLOG supports nested CRDTs such as maps; our graph case study uses a Map⟨Set⟩.

3.3 Nested CRDTs

Designers obtain nested data types by choosing component CRDTs and writing transformation rules from top-level events to component events. In nested structures, the decomposition is often structural. For example, a Map⟨Set⟩ combines a set that tracks keys and a set of values for each key.

To make this example concrete, consider a Map⟨Set⟩ with $Op_X = \{\texttt{upd}(k, v), \texttt{rmv}(k)\}$, where v can be either $\texttt{add}(x)$ or $\texttt{del}(x)$. The first step is to define the SLS that formalize the behavior of the application as a single object:

$$\begin{aligned} &\mathcal{F}_{\text{MapSet\_SLS}}(\texttt{hasKV}(k, x), (\text{E}, \text{op}, \text{vis}, \text{ar})) = \text{true} \\ &\iff \exists e \in \text{E} : \texttt{upd}(k, \texttt{add}(x)) \in \text{op}(e) \wedge (\neg\exists e' \in \text{E} : \texttt{rmv}(k) \in \text{op}(e') \wedge e \xrightarrow{vis} e') \\ &\quad \wedge (\neg\exists e'' \in \text{E} : \texttt{upd}(k, \texttt{del}(x)) \in \text{op}(e'') \wedge e \xrightarrow{vis} e'') \end{aligned} \tag{3}$$

A Map⟨Set⟩ contains a key `k` and value `x` in the set associated with `k` if it was inserted by an `upd` operation and neither the key nor the value were later removed.

Next, we obtain the ICS for the Map⟨Set⟩. Intuitively, the map contains (k, x) when k is present in the key set and x is present in the value set associated with k. Figure 4 presents the corresponding transformation rules.

$$\begin{aligned} &\mathcal{F}_{\text{MapSet\_ICS}}(\texttt{hasKV}(k, x), (\text{E}, \text{op}, \text{vis}, \text{ar})) = \text{true} \iff \\ &\quad \mathcal{F}_{\text{Set}}(\texttt{hasV}(k), (\text{E}, \text{op}_{\texttt{KeySet}}, \text{vis}, \text{ar})) \wedge \mathcal{F}_{\text{set}_k}(\texttt{hasV}(x), (\text{E}_{\texttt{ValSet}}, \text{op}_{\texttt{ValSet}}, \text{vis}, \text{ar})) \end{aligned} \tag{4}$$

In Datalog, the event facts of the Map⟨Set⟩ `mapSetOp(E,Op,InnerOp,K,X)` consist of the event identifier `E`, the map-level operation `Op`, the inner set operation `InnerOp` and the operation parameters `K` and `X` (Listing 2). `keysEvent` and `setEvent` correspond to the transformation rules. `mapState` computes the final state. Note that a key might exist with an empty value set. For that we use the empty value `''''`.

Soufflé does not support dynamic instantiation of modules at runtime. In particular, a Map⟨Set⟩ cannot allocate a fresh set instance per key during evaluation. We therefore

```
laterVisOpSet(E1, E2) :- setEvent(E1, _, X, S), setEvent(E2, "del", X, S),
                         visible(E1, E2), E1\=E2.
setState(S, X)        :- setEvent(E, "add", X, S), not laterVisOpSet(E, _).
```

Listing 3. Value set semantics expressed for ICS Map⟨Set⟩.

```
conc(E1,E2)      :- graphEvent(E1,_,_,_), graphEvent(E2,_,_,_), not visible(E1,E2),
                    not visible(E2,E1), E1\=E2.
concAddE(E2, K) :- graphEvent(E2,_,_,_), graphEvent(E1, "addE", K, _), conc(E1,E2).
concAddE(E2, K) :- graphEvent(E2,_,_,_), graphEvent(E1, "addE", _, K), conc(E1,E2).
postVisRmvN(E1, K, E2) :- graphEvent(E1, _, _, _), graphEvent(E2, "rmvN", K, _),
                          visible(E1, E2), E1\=E2, not concAddE(E2, K).
nodes(K)         :- graphEvent(E, "addN", K, _), not postVisRmvN(E, K, _).
postVisRmvE(E1) :- graphEvent(E1,_,K1, K2), graphEvent(E2, "rmvE", K1, K2),
                   visible(E1, E2), E1 \= E2.
edges(N, M)      :- graphEvent(E, "addE", N, M), not postVisRmvE(E).
```

Listing 4. SLS semantics of isolate-delete graph.

encode all inner sets within a single indexed set instance, using an additional identifier (the key) to distinguish per-key state. The corresponding code can be found in Listing 3. With this encoding pattern, deeper nesting (e.g. maps of maps) reduces to the repeated use of transformation rules and identifiers.

3.4 Composite CRDTs and CRDT applications

Collaborative applications typically combine multiple CRDTs into composite ones (Kuessner *et al.* 2023). We illustrate the workflow on composite CRDTs using the ID graph application from Table 1. Recall that a node is in the graph if it has been added and not later removed, unless a concurrent `addE` defeats the removal. An edge is present if and only if it has been added and not subsequently removed. We define the corresponding SLS in equation (5).

$$\begin{aligned}
&\mathcal{F}_{\text{IDGraph_SLS}}(\texttt{hasN}(n), \mathrm{E}, \mathrm{op}, \mathrm{vis}, \mathrm{ar}) = \mathrm{true} \iff \exists e \in \mathrm{E} :\ \texttt{addN}(n) \in \mathrm{op}(e) \\
&\wedge\ (\neg\exists e' \in \mathrm{E} :\ \texttt{rmvN}(n) \in \mathrm{op}(e') \wedge e \xrightarrow{\text{vis}} e' \vee\ \exists e', e'' \in \mathrm{E} :\ \texttt{rmvN}(n) \in \mathrm{op}(e') \wedge e \xrightarrow{\text{vis}} e' \\
&\qquad \wedge (\texttt{addE}(n, m) \in \mathrm{op}(e'')\ \vee\ \texttt{addE}(m, n) \in \mathrm{op}(e''))\ \wedge e' \not\xrightarrow{\text{vis}} e'' \wedge e'' \not\xrightarrow{\text{vis}} e') \\
&\mathcal{F}_{\text{IDGraph_SLS}}(\texttt{hasE}(n, m), \mathrm{E}, \mathrm{op}, \mathrm{vis}, \mathrm{ar}) = \mathrm{true} \\
&\iff \exists e \in \mathrm{E} : \texttt{addE}(n, m) \in \mathrm{op}(e) \wedge (\neg\exists e' \in \mathrm{E} : \texttt{rmvE}(n, m) \in \mathrm{op}(e') \wedge e \xrightarrow{\text{vis}} e')
\end{aligned} \tag{5}$$

In the executable SLS in Datalog (Listing 4), we model the input as facts `graphEvent(E, O, V1, V2)`, where `E` is the event identifier, `O` is the operation of the event, and `V1` and `V2` are the parameters of the operation. For a node `n` to be in the resulting graph state `nodes(K)`, there must be an `addN(n)` which is not visible to a `remove` (`not postVisRmvN`); or, if there is a `postVisRmvN`, there exists a concurrent `addE` (`concAddE`), outgoing or incoming from `n`. An edge is in the resulting graph state (`edges(N,M)`) if the edge has been added and it has not been removed later (`postVisRmvE`).

The next step is to derive the *implementation-level compositional semantics* (ICS) by choosing a CRDT decomposition and defining transformation rules (Definition 2). A natural decomposition mirrors a sequential design: a replicated node set (`NodeSet`)

$$
\begin{aligned}
\texttt{IDGraph.addN}(n) &\overset{\mathcal{T}}{\Rightarrow} \{\texttt{NodeSet.add}(n)\} \\
\texttt{IDGraph.rmvN}(n) &\overset{\mathcal{T}}{\Rightarrow} \{\texttt{NodeSet.del}(n)\} \\
\texttt{IDGraph.addE}(n, m) &\overset{\mathcal{T}}{\Rightarrow} \{\texttt{NodeSet.add}(n), \texttt{NodeSet.add}(m), \texttt{EdgeMap.upd}(n, \texttt{add}(m))\} \\
\texttt{IDGraph.rmvE}(n, m) &\overset{\mathcal{T}}{\Rightarrow} \{\texttt{EdgeMap.upd}(n, \texttt{del}(m))\} \\
\texttt{EdgeMap.upd}(n, \texttt{add}(m)) &\overset{\mathcal{T}}{\Rightarrow} \{\texttt{KeySet.add}(n),\ \texttt{ValSet}_n\texttt{.add}(m)\} \\
\texttt{EdgeMap.upd}(n, \texttt{del}(m)) &\overset{\mathcal{T}}{\Rightarrow} \{\texttt{KeySet.add}(n), \texttt{ValSet}_n\texttt{.del}(m)\}
\end{aligned}
$$

Figure 5. Layered transformation for isolate-delete graph.

```
nodesEvent(E, "add", V) :- graphEvent(E, "addN", V, _).
nodesEvent(E, "del", V) :- graphEvent(E, "rmvN", V, _).
nodesEvent(E, "add", V) :- graphEvent(E, "addE", V, _).
nodesEvent(E, "add", V) :- graphEvent(E, "addE", _, V).
edgesMapSetEvent(E, "upd", "add", V1, V2) :- graphEvent(E, "addE", V1, V2).
edgesMapSetEvent(E, "upd", "del", V1, V2) :- graphEvent(E, "rmvE", V1, V2).
```

Listing 5. ICS for isolate-delete graph.

and a map from each node to its outgoing-edge set (`EdgeMap`). The transformation rules (Figure 5) map each application event to one or more component updates. For nodes, `addN(n)` and `rmvN(n)` translate to `add(n)` and `del(n)`; to realize isolate-delete under concurrency, `addE(n,m)` additionally produces `add(n)` and `add(m)` on the `NodeSet`, inducing the conflict needed to win over a concurrent node removal. For edges, outgoing edges are stored in an adjacency map: `addE(n,m)` translates to `upd(n,add(m))` and `rmvE(n,m)` to `upd(n,del(m))`; we use `upd` rather than key removal, which would delete all outgoing edges of `n`. Since `EdgeMap` is a Map⟨Set⟩, its internal transformation rules also apply; Map⟨Set⟩ is fully defined in the extended version.

Definition 2 (Transformation rules).
A *transformation rule* $\mathcal{T} : Op_X \to \mathcal{P}(Op_{X_1} \cup \cdots \cup Op_{X_n})$ maps each operation of a composite data type X to a set of operations over its components $X_1, \ldots, X_n$.

The resulting ICS of the ID graph is given as:

$$
\begin{aligned}
&\mathcal{F}_{\text{IDGraph\_ICS}}(\texttt{hasN}(n), (\text{E}, \text{op}, \text{vis}, \text{ar})) = \text{true} \\
&\quad \Longleftrightarrow \mathcal{F}_{\text{Set}}(\texttt{hasV}(n), (\text{E}_{\texttt{NodeSet}}, \text{op}_{\texttt{NodeSet}}, \text{vis}, \text{ar})) \\
&\mathcal{F}_{\text{IDGraph\_ICS}}(\texttt{hasE}(n, m), (\text{E}, \text{op}, \text{vis}, \text{ar})) = \text{true} \\
&\quad \Longleftrightarrow \mathcal{F}_{\text{MapSet}}(\texttt{hasKV}(n, m), (\text{E}_{\texttt{EdgeMap}}, \text{op}_{\texttt{EdgeMap}}, \text{vis}, \text{ar}))
\end{aligned}
\tag{6}
$$

In the Datalog realization (Listing 5), the input facts have the same signature as the SLS, namely `graphEvent(E, O, V1, V2)`. The resulting state is computed for each component. Note that this implementation may produce keys with empty value sets. We interpret such entries as non-existent edges and ignore them when inspecting the application state.

The framework is compositional at the specification level: composite CRDTs are written as decompositions of simpler ones together with the corresponding transformation rules. The ICS is then mechanically derived from this choice of decomposition. However, Soufflé does not allow multiple independent instances of the same CRDT to share a single set of rules without an explicit discriminator. For nested types, this can be handled

```
postVisRmvN(E, K, S) :- graphEvent(E, _, _, _), graphEvent(S, "rmvN", K, _),
                        visible(E, S), E \= S.
postVisRmvE(E)       :- graphEvent(E, _, K1, K2), graphEvent(S, "rmvE", K1, K2),
                        visible(E, S), E \= S.
edges(N, M) :- graphEvent(E, "addE", N, M), not postVisRmvE(E),
               not postVisRmvN(E, N, _), not postVisRmvN(E, M, _).
```

Listing 6. SLS for DD graph (edge rules).

through a shape-based encoding, where nesting positions are represented as structured scopes. In this approach, a single set module serves all (map) nesting levels, with each instance distinguished by its position in a recursive scope type rather than by duplicated and renamed Datalog rules. For example, a Map⟨Map⟨Set⟩⟩ reuses the same set semantics as a Map⟨Set⟩, with an additional scope level encoding the outer key: no ad-hoc adjustments are required, and the pattern extends to arbitrary nesting depths. In this paper, we opt for the explicit, per-instance encoding (e.g. separate `nodesEvent` and `edgesMapSetEvent` relations) as it maps directly to the formal definitions in Section 2 and makes the correspondence between the specification-level and implementation-level semantics transparent for verification.

CRDT composition is not semantics-neutral: decomposition choices determine which operations conflict and how conflicts are resolved. For graphs, nodes and edges are separate structures but semantically coupled; that is an `addE` may defeat a concurrent `rmvN`, so the transformation rules must intentionally induce or avoid such conflicts.

4 A case study on graph CRDTs

We evaluate CRDTLog on two directed graph variants: the *isolate-delete (ID)* graph from Sections 2 and 3 and a *detach-delete (DD)* graph introduced below, capturing common deletion policies with non-trivial concurrent interactions. We show that CRDTLog yields precise, executable semantics and enables systematic CRDT component selection.

Isolate-delete semantics (ID). We first revisit the ID semantics defined in Section 3.4. A natural implementation-level decomposition (ICS) represents nodes as an add-wins Set and outgoing edges as a Map⟨Set⟩. The transformation layer is straightforward for most operations and maps `addE` to additional `add` updates on the node set to ensure that concurrent edge additions override node removals, as required by the SLS.

DD We next consider the DD semantics for directed graphs. This specifies that removing a node also removes all its incident edges, ensuring the absence of dangling edges. The formal definition differs from ID only in the definition of $\texttt{hasE}(u, v)$ (Table 1). DD preserves the node semantics of ID. In particular, a node n is present if it has been added and not subsequently removed, unless the removal is invalidated by a concurrent edge addition involving n. Hence, the definition of $\mathcal{F}_{\text{DDGraph\_SLS}}(\texttt{hasN}(n), \dots)$ is identical to the isolate-delete case (Equation 5). Further, a DD graph contains an edge `(n,m)` if it has been added, has not been subsequently removed, and neither the source nor target node has been removed. The node condition is the new part: if `n` (or `m`) is removed, then all incident edges are removed as well (Equation 7, Listing 6). In the listing, `edges(N,M)` holds for added edges whose endpoints were not removed, while `postVisRmvN` captures

$$
\begin{aligned}
\texttt{DDGraph.rmvN}(n) &\overset{\mathcal{T}}{\Rightarrow} \{\texttt{EdgeSet.del}(n,m), \texttt{EdgeSet.del}(m,n)\},\\
&\quad \text{where } m \text{ is a node to/from which an edge from/to } n \text{ exists.}\\
\texttt{DDGraph.addE}(n,m) &\overset{\mathcal{T}}{\Rightarrow} \{\texttt{EdgeSet.add}(n,m)\}\\
\texttt{DDGraph.rmvE}(n,m) &\overset{\mathcal{T}}{\Rightarrow} \{\texttt{EdgeSet.del}(n,m)\}
\end{aligned}
$$

Figure 6. Transformation rules for detach-delete graph.

```
edgesEvent(E, "add", V1, V2) :- graphEvent(E, "addE", V1, V2).
edgesEvent(E, "del", V1, V2) :- graphEvent(E, "rmvE", V1, V2).
edgesEvent(E, "del", V1, V2) :- graphEvent(E, "rmvN", V1, _), preVisAddOutE(E, V2).
edgesEvent(E, "del", V2, V1) :- graphEvent(E, "rmvN", V1, _), preVisAddInE(E, V2).
preVisAddInE(E, V2)  :- graphEvent(E, _, V1, _), graphEvent(S, "addE", V2, V1),
                        visible(S, E), E\=S.
preVisAddOutE(E, V2) :- graphEvent(E, _, V1, _), graphEvent(S, "addE", V1, V2),
                        visible(S, E), E\=S.
```

Listing 7. ICS for DD graph. Transformation rules for the edges.

concurrent `addE` that invalidate node removals.

$$
\begin{aligned}
&\mathcal{F}_{\text{DDGraph\_SLS}}(\texttt{hasE}(n,m), \mathrm{E}, \mathrm{op}, \mathrm{vis}, \mathrm{ar}) = \mathrm{true} \\
&\iff \exists e \in \mathrm{E} : \ \texttt{addE}(n,m) \in \mathrm{op}(e) \ \wedge \\
&\qquad \neg\exists e' \in E : \ \texttt{rmvE}(n,m) \in \mathrm{op}(e') \wedge e \xrightarrow{\text{vis}} e' \ \wedge (\Gamma(e,n) \wedge \Gamma(e,m)) \\
&\Gamma(e,u) \overset{\text{def}}{=} \neg\exists s \in \mathrm{E} : \ \texttt{rmvN}(u) \in \mathrm{op}(s) \wedge e \xrightarrow{\text{vis}} s
\end{aligned}
\tag{7}
$$

At first glance, one might attempt to reuse the ID implementation strategy, using a set for nodes and a Map⟨Set⟩ for edges. Thus, removing a node `n` would remove `n` from the node set and delete its outgoing edges by removing the key `n` from the edge map. Incoming edges could then be removed by deleting the corresponding entries from the value sets associated with other keys. However, this straightforward approach fails because it introduces unintended conflicts in the map. In particular, a concurrent `rmvE`(n,m) and `rmvN`(n) generate conflicting updates that combine a value-level update (a `upd` or inner-set modification) with a key-level `rmv`. Under the update-wins map semantics, the `upd` may resurrect the key `n`, thereby preserving outgoing edges from `n`. The resulting state contains dangling edges, which violates the DD semantics. To avoid this, we represent the graph with one set for nodes and one set for edges. The edge set stores tuples `(n,m)`. This design eliminates key-level put-versus-remove conflicts since all edge updates are applied uniformly to a single set. The transformations for node operations are unchanged from the ID case. The transformation for edge operations is defined in Figure 6.

$$
\begin{aligned}
&\mathcal{F}_{\text{DDGraph\_ICS}}(\texttt{hasE}(n,m), (\mathrm{E}, \mathrm{op}, \mathrm{vis}, \mathrm{ar})) = \mathrm{true} \\
&\iff \mathcal{F}_{\text{Set}}(\texttt{hasV}(n,m), (\mathrm{E}_{\texttt{EdgeSet}}, \mathrm{op}_{\texttt{EdgeSet}}, \mathrm{vis}, \mathrm{ar}))
\end{aligned}
\tag{8}
$$

Note that the two additional predicates `preVisAddInE` and `preVisAddOutE` identify the edges that are visible during the removal event. The transformation then removes exactly these edges, ensuring that successful node deletions detach all incident edges while preserving the intended concurrency behavior captured by the node semantics.

We establish that SLS and ICS in the DD graph preserve the no-dangling edge invariant. The full proofs are in the supplemental material.

Lemma 1 (Dangling edge safety for the SLS of a DD graph).
For the SLS of the DD graph, the invariant of no dangling edges holds for any valid operation context.

Proof sketch.
A dangling edge requires a removed node n with a present incident edge. If `rmvN`(n) has no concurrent incident `addE`, the endpoint guard $\Gamma(e, n)$ (Equation (7)) is falsified by the observed removal for every incident `addE` event e, so no such edge survives. If a concurrent `addE`(n,m) exists, $\Gamma(e, n)$ holds as the removal is not visible to e, but the node predicate (Equation (5)) also retains n, so the edge is not dangling. □

Lemma 2 (Dangling edge safety for the ICS of a DD graph).
For the ICS of the DD graph, the invariant of no dangling edges holds for any valid operation context.

Proof sketch.
As above, we case-split on events concurrent with `rmvN`(n). Without a concurrent `addE`, the operation translates into `dels` in the underlying set (Figure 6), removing all incident edges with n. With a concurrent `addE`(n,m), the transformed `add`(n) wins over `del` in the nodes set, retaining n and preventing a dangling edge. □

This case study highlights the value of explicit semantic specifications: they expose situations in which a CRDT decomposition enforces conflict resolution incompatible with the intended application semantics. CRDTlog makes this mismatch explicit at the specification level, supporting designers in composing CRDTs with the desired semantics.

5 Experimental evaluation

We compare ICS and SLS for both ID and DD graph applications. Correctness is assessed via property-based testing: for eight configurations, we generate 1K independent executions and verify that SLS and ICS produce identical output graphs. We then benchmark scalability by varying event count, replica count, and graph size. We structure the analysis around the following research questions.

- ***RQ1*** *(Semantic Equivalence).* Do ICS and SLS yield the same observable states for all tested operation contexts?
- ***RQ2*** *(Application Graph Scalability).* How does the size of the application graph affect the cost of semantic evaluation?
- ***RQ3*** *(Concurrency Scalability).* How does increasing replica count and concurrency affect evaluation cost?
- ***RQ4*** *(Event Scalability).* How does evaluation cost scale with increasing events?
- ***RQ5*** *(Incrementality).* How does incremental evaluation impact verification?

Experimental setup. All experimental inputs were automatically generated using Python scripts that enforce each graph application's preconditions. For example, in the isolate-delete variant, `rmvN(n)` is never generated for nodes with incident edges, ensuring that all executions satisfy the assumptions of the model. The primary goal is to validate semantic equivalence (SLS ≡ ICS) via property-based testing, therefore the inputs must be

large and diverse enough to cover corner cases arising from concurrent operations across multiple replicas. While the graph semantics used here are intentionally minimal, the framework is intended for richer application domains (e.g. property graphs, collaborative spreadsheets) where complex operation interleavings are expected. For such scenarios, generating large workloads is necessary to provide adequate coverage.

Equivalence tests ran on a machine running Ubuntu 22.04.2 with 11th Gen Intel(R) Core(TM) i7-1165G7 2.80 GHz CPU and 32 GB of memory.

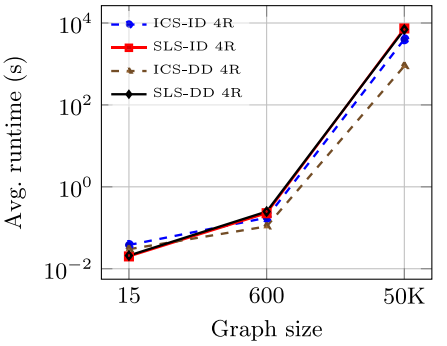
Scalability experiments ran on a CPU-only x86_64 server with two AMD EPYC 7443 processors (1.5–2.85 GHz, boost enabled) and 100 GB of allocated memory. For each configuration, we report mean execution times over 30 independent runs. Both SLS and ICS were evaluated on the same generated inputs.

*Semantic equivalence (**RQ1**).* SLS and ICS compute identical observable states for all tested operation contexts. We evaluated semantic equivalence using a property-based testing approach (Claessen and Hughes 2000), generating randomized operation contexts and comparing the outputs of SLS and ICS. We considered eight configurations, combining two replica counts (5 and 10) with four event volumes (20, 50, 100, and 1K events). For each configuration, we generated 1,000 independent test inputs. All executions terminated successfully, and in every case, the SLS and ICS implementations produced identical results. These results provide empirical evidence that our CRDT compositions faithfully realize the intended semantics for both the ID and DD graph variants. Average execution times per test are reported in the supplemental material.

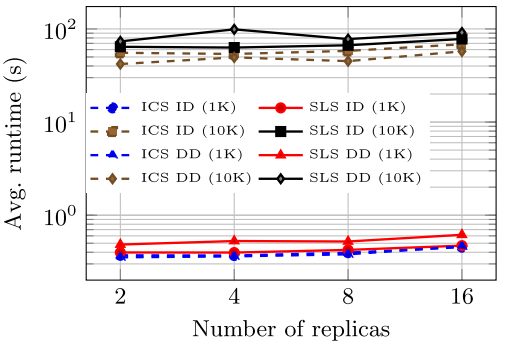
*Scalability (**RQ2**–**RQ5**).* We analyze how semantic evaluation cost scales with operation context size, concurrency, and application graph size. Figure 7 reports runtime as a function of graph size (Figure 7(a)), replica count (Figure 7(b)), and number of events (Figure 7(c)). For concurrency and event scalability, we used the full range of graph operations, whereas for the graph scalability we used only `addN` and `addE` events. Concurrency is modeled by allowing operations to interleave and branch, producing non-trivial visibility structures.

*Application graph scalability (**RQ2**).* Evaluation cost increases with application graph size (nodes and edges), but ICS scales better than SLS for larger graphs. To study graph size scalability, we sampled random graphs with a fixed number of nodes (10, 100, and 1K) using an edge probability of 0.05, yielding an expected number of edges proportional to the graph size ($\approx$ 15, 600, and 50K). This reflects the average number of events. For each graph, we generated an operation context with 4 replicas.

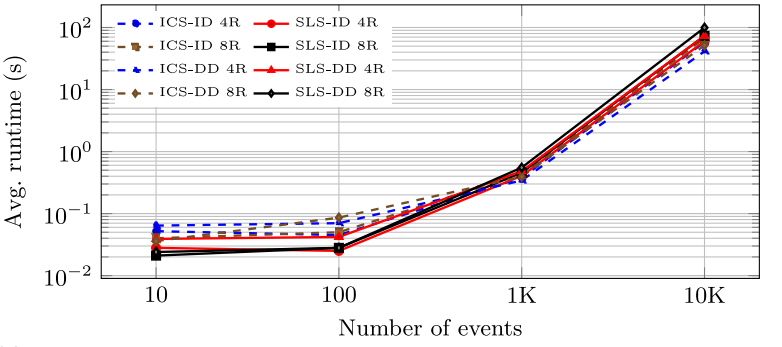
Figure 7(a) shows that for small graphs, ICS implementations are slightly slower than the corresponding SLS specifications due to the overhead of transformation rules. However, as graph size increases (and with it the number of events in the operation context), ICS becomes consistently faster for both applications. At approximately 50K graph entities, the ICS realization is nearly twice as fast as SLS for ID and close to an order of magnitude faster for DD. This behavior reflects the lower rule complexity of the



(a) Scaling with graph size. The number of events is equivalent to the graph size.



(b) Scaling with the number of replicas for 1K and 10K events.



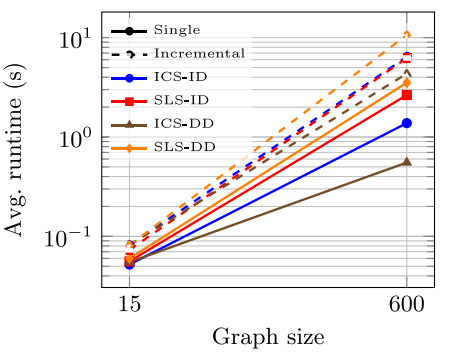
(c) Scaling with the number of events. As the add and remove operations randomly interleave, the graph size varies across the runs.

Figure 7. Scalability of SLS and ICS for isolate-delete (ID) and detach-delete (DD).

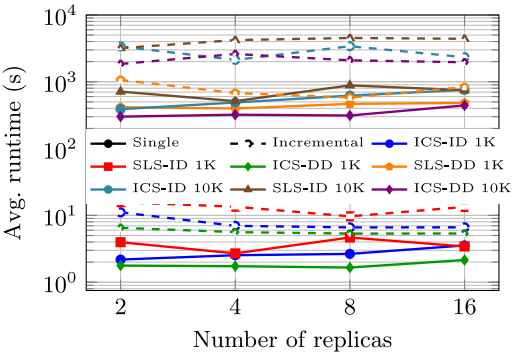
component CRDT semantics used in ICS, which outweighs the transformation overhead once evaluation is dominated by joins over many events.

*Concurrency scalability (**RQ3**).* Increasing the number of replicas only mildly impacts the evaluation cost. We varied the replica count across 2, 4, 8, and 16, holding the total event count fixed at 1K and 10K. As shown in Figure 7(b), increasing the replica count results in only modest runtime growth. In CRDTLog, the number of replicas mainly affects the branching structure of the visibility relation without substantially increasing the complexity of rule evaluation. This suggests that semantic evaluation scales well with respect to concurrency.

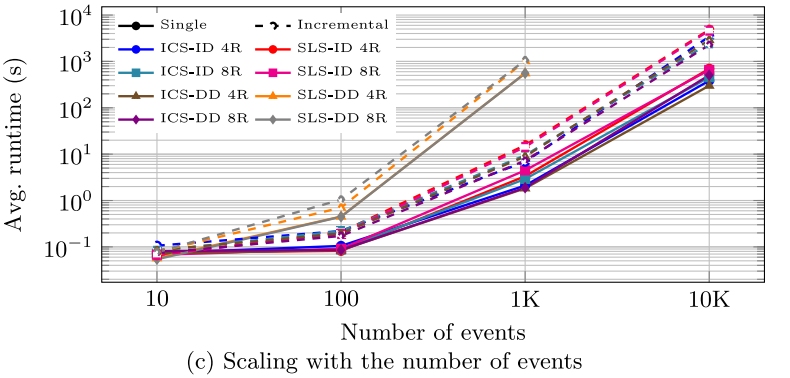
*Event scalability (**RQ4**).* The evaluation cost grows primarily with the number of events in the operation context. As summarized in Figure 7(c), runtime increases steadily as the number of events grows for both graph variants and both semantic encodings. This trend holds largely independently of the number of replicas, indicating that event volume is the dominant factor driving semantic evaluation cost. Unlike in RQ2, the resulting graph size is not held constant here: since add and remove operations are interleaved during generation, graph size varies across runs and is not an independent variable.



(a) Scaling with graph size



(b) Scaling with the number of replicas for 1K and 10K events



(c) Scaling with the number of events

Figure 8. Single vs. incremental SLS and ICS for isolate-delete (ID) and detach-delete (DD).

*Incrementality (**RQ5**).* Our framework can also be instantiated with DDlog, an incremental Datalog engine that supports an interactive workflow: users incrementally add events and observe the resulting CRDT state, aiding scenario exploration and debugging. We evaluate DDlog in single-shot and incremental mode across varying events, replicas, and graph sizes. In incremental mode, events are inserted one at a time, mirroring the interactive use case. Both modes are substantially slower than Soufflé (Figure 8(c)), which is expected. DDlog optimizes for small updates over a stable base, so it pays a constant per-tuple cost for incrementality, even in single-shot mode. Here each new event causes changes to the visibility relation, so incremental evaluation effectively recomputes everything. DDlog remains practical for small to medium contexts (up to $\approx$ 1K events), which suffices for its intended interactive, exploratory use case.

Discussion. Although we do not directly compare ID and DD semantics, their scaling behaviors are notably distinct. DD is substantially faster for large graphs, with the SLS–ICS gap most pronounced. This reflects the underlying representations: ID stores edges in a Map⟨Set⟩, introducing nesting and multiple component-level updates per event, whereas DD uses a single set of edge tuples with at most one component update per event. ICS consistently outperforms SLS for both graphs, as decomposition yields smaller relations with localized joins rather than complex guard predicates over the entire event. The effect

is amplified in DDlog, where the incremental engine benefits from change propagation over many small relations. Thus, decomposition can significantly reduce intermediate joins and improve evaluation time, even when the resulting semantics are identical.

While runtimes grow steeply beyond 1K events, we argue that this is sufficient for property-based testing. Optimizing the underlying Datalog engine is orthogonal to our contribution and could further extend the feasible testing range.

6 Conclusions and perspectives

We introduced CRDTLog, a declarative framework for specifying and reasoning about CRDT semantics over operation contexts, cleanly separating specification-level semantics (SLS) from implementation-level compositional semantics (ICS) via CRDT composition and transformation rules. Both are implemented in Datalog with modular support for basic, nested, and composite CRDTs.

Evaluation on two collaborative graph applications showed that ICS consistently matches SLS across all test configurations. Scalability is driven mainly by execution size, with replica count having limited impact; for larger workloads, ICS often outperforms SLS, making it well suited for large-scale testing and semantic prototyping.

As future work, we plan to formalize CRDTLog in a theorem prover to enable machine-checked proofs of additional semantic properties.

Competing interests

The authors declare no competing interests that are relevant to the content of this article.

Supplementary material

To view supplementary material for this article, please visit https://doi.org/10.1017/S1471068426100556.

References

Alvaro, P., Marczak, W. R., Conway, N., Hellerstein, J. M., Maier, D. and Sears, R. 2010. Dedalus: Datalog in time and space. In Datalog, LNCS, Springer, 262–281. https://doi.org/10.1007/978-3-642-24206-9_16.

Alvaro, P., Conway, N., Hellerstein, J. M. and Marczak, W. R. 2011. Consistency analysis in bloom: A CALM and collected approach. In CIDR, 249–260. www.cidrdb.org

Borth, E., Lersch, P. and Bieniusa, A. 2025. Directed acyclic graph CRDTs. In PaPoC@EuroSys, ACM, 30–37. https://doi.org/10.1145/3721473.3722141.

Burckhardt, S. 2014. Principles of eventual consistency. *Foundations and Trends® in Programming Languages* 1, 1-2, 1–150. https://doi.org/10.1561/2500000011.

Claessen, K. and Hughes, J. 2000. QuickCheck: A lightweight tool for random testing of Haskell programs. In ICFP, ACM, 268–279. https://doi.org/10.1145/351240.351266.

Gilbert, S. and Lynch, N. 2002. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News* 33, 2, 51–59. https://doi.org/10.1145/564585.564601.

Jordan, H., Scholz, B. and Subotic, P. 2016. Soufflé: On synthesis of program analyzers. In CAV (2), LNCS, Springer, 422–430. https://doi.org/10.1007/978-3-319-41540-6_23.

Kleppmann, M. 2017. Automerge. https://github.com/automerge/ (visited: 2026-01).

Kuessner, C., Mogk, R., Wickert, A.-K. and Mezini, M. 2023. Algebraic replicated data types: Programming secure local-first software. *Artifact* 9, 26:-1–26:-4. https://doi.org/10.4230/DARTS.9.2.26.

Kuper, L. and Newton, R. R. 2013. LVars: Lattice-based data structures for deterministic parallelism. In FHPC@ICFP, ACM, 71–84. https://doi.org/10.1145/2502323.2502326.

Laddad, S., Power, C., Milano, M., Cheung, A., Crooks, N. and Hellerstein, J. M. 2022. Keep CALM and CRDT on. *Proceedings of the VLDB Endowment* 16, 4, 856–863. https://doi.org/10.14778/3574245.3574268.

Meiklejohn, C. and Van Roy, P. 2015. Lasp: A language for distributed, coordination-free programming. In PPDP, ACM, 184–195. https://doi.org/10.1145/2790449.2790525.

Nicolaescu, P., Jahns, K., Derntl, M. and Klamma, R. 2015. Yjs: A framework for near real-time P2P shared editing on arbitrary data types. In ICWE, LNCS, Springer, 675–678. https://doi.org/10.1007/978-3-319-19890-3_55.

Pandey, A., Dumbrava, S., Shapiro, M., Ferreira, C., Pereira, M. and Preguiça, N. M. 2025. Towards local-first distributed property graphs. In PaPoC@EuroSys, ACM, 22–29.

Ryzhyk, L. and Budiu, M. 2019. Differential Datalog. In Datalog, CEUR Workshop Proceedings, CEUR-WS.org, 56–67.

Shapiro, M., Preguiça, N. M., Baquero, C. and Zawirski, M. 2011. Conflict-free replicated data types. In SSS, LNCS, Springer, 386–400. https://doi.org/10.1007/978-3-642-24550-3_29.

Soundarapandian, V., Kamath, A., Nagar, K. and Sivaramakrishnan, K. C. 2022. Certified mergeable replicated data types. In PLDI, ACM, 332–347. https://doi.org/10.1145/3519939.3523735.