# Windowed Conflict-Free Replicated Data Types

Wu Tianxing
KTH Royal Institute of Technology
Stockholm, Sweden
tianxing@kth.se

Jonas Spenger
KTH Royal Institute of Technology
Stockholm, Sweden
jspenger@kth.se

Philipp Haller
KTH Royal Institute of Technology
Stockholm, Sweden
phaller@kth.se

## Abstract

Globally shared state and deterministic results without blocking coordination in a distributed stream processing system are typically unavailable. Conflict-free replicated data type (CRDT) has partially resolved this problem of having an eventual consistent globally shared state, yet failed to guarantee a deterministic result and is difficult to reason about. In this paper, we propose a new solution to the problem called windowed CRDT, providing a deterministic result with minimum blocking besides normal CRDT advantages. We formally proved that it can be constructed from any CRDT as well as accessed and updated by the user at will, while still guaranteeing a deterministic processing result.

***Keywords:*** Conflict-free Replicated Data Types, Stream Processing Systems, Window-Based Computations, Confluence

## 1 Introduction

Globally shared state within stateful dataflow streaming programs is typically not available [24]. This is due to its conflicting nature with the reactivity of the streaming event processing model; events should be processed immediately, and blocking should not happen.

Instead, if necessary, a global state is typically emulated through aggregation networks. A simple approach is to send all updates to a single physical instance, whereas more sophisticated methods leverage aggregation trees. Conflict-free replicated data types (CRDTs) are a way to introduce a globally shared state without the need for blocking coordination on the event processing level. Instead, the coordination happens off the critical path. However, CRDTs are nondeterministic and provide only strong eventual consistency, making it difficult to reason about programs using them [9]. In other words, CRDTs may yield different values given different processing orders. Furthermore, strong eventual consistency does not apply to the streaming model, in which the system may continue processing events indefinitely. In this setting, strong eventual consistency does not imply that eventually, all replicas will have the same state. The nondeterminism and the lack of stronger consistency guarantees make it difficult to integrate CRDTs into stateful dataflow systems that provide exactly-once processing guarantees.

Towards this issue, we consider a new type of CRDT, so-called windowed CRDTs. Windowed CRDTs provide all the benefits CRDTs have plus confluence, which guarantees independence between output and processing order. By allowing to construction of windowed CRDTs from any CRDT and arbitrary windows, it enables the user to set synchronization points at will and introduce data dependency from the latter window to previous windows without compromising deterministic property. Moreover, a window also represents causally consistent snapshots of the system states. This could be utilized to provide failure transparent recovery guarantees.

### Contributions

In summary, this paper makes the following contributions.

- We define windowed CRDTs together with construction from arbitrary state-based CRDTs (Section 4).
- We provide a small-step operational semantics of windowed CRDTs within a stream processing system (Sections 5, 6).
- We prove that the given semantics is confluent and satisfies strong eventual consistency (Section 7).
- We provide an algorithm for transparent failure recovery of windowed CRDTs (Section 8).
- We evaluate the performance of our implementation with failure recovery in Scala 3 using the Apache Pekko [20] actor model framework (Section 9).

## 2 Stream Processing System and Window-Based Computations

### 2.1 Stream Processing Systems (SPS)

A streaming processing system [24, 34] contains multiple parallel processors, operating on incoming data streams. Data streams can be bounded, but more often, unbounded and are generated as time goes [7]. Typically, these data streams are structured as a pair of labels and real data, which is fed to different processors based on labels. Streaming processing is getting growing attention as it is important to modern social media platforms like YouTube, Facebook etc [8].

SPS has been extensively researched and widely used, some examples are Apache Flink [2, 19], KAR [35], Portals [33] etc.

### 2.2 Globally Shared State in SPSs

Traditionally, a globally shared state is achieved via an external service, typically a database. This approach has always been a widely discussed topic in the database community [15]. However, modern SPSs such as MapReduce [**?** ] usually require massive scale-out, which makes this approach unacceptable due to high communication cost, risk

of bringing the whole system down by a single failure in the database and database's computation power limitation.

More recent attempts focus on using distributed snapshotting protocols to achieve such goals [11]. This usually involves analyzing the graph of dataflows and some algorithms are only restricted to graphs with specific properties [14] or interfering execution flows [26].

### 2.3 Window-Based Computations

Windows in SPSs can separate unbounded streams into bounded streams and perform calculations based on those bounded streams [37]. Some typical usages are window aggregation [23] and producing timely responses. It is also used as a periodic checkpoint in systems, where deliveries of data are out of order, while still retaining some kind of order during computation.

## 3 Conflict-Free Replicated Data Types (CRDTs)

### 3.1 Overview

In the context of a distributed system, the ultimate goal is to provide a system that has consistency (C), availability(A), and partition tolerance (P). However, this is generally not possible, as the CAP theorem states [22]. CRDTs, originally designed as a solution for concurrent editing [28, 30], are found useful to partially address this problem by relaxing the requirement of consistency to strong eventual consistency [31]. In general, CRDTs have the following characteristics.

1. Each node holds a replica and performs local updates without synchronization.
2. Replicas can be sent to other replicas and perform a merge operation.
3. Idempotent is guaranteed such that if the same merge is performed twice it shall not change anything.

CRDTs are widely used in many research and commercial projects including Redis [6], Walter [32], SwiftCloud [29], AntidoteDB [3] etc.

There are two kinds of CRDTs, state-based CRDT and operation-based CRDT. The difference is that state-based CRDTs do synchronization by sending the whole state, while operation-based ones only send modifications. However, operation-based CRDTs require exactly-once delivery of updates to guarantee consistency. In contrast to common sense, they are equivalent [31]. Thus, in this paper, we only consider state-based CRDTs for simplicity.

### 3.2 Formal Definition

A state-based object satisfies strong eventual consistency if it is a monotonic semilattice object [31]. Thus, consider the definition of CRDTs.

**Definition 1** (CRDT). *A state-based object, equipped with partial order $\leq$, denoted by $(S, \leq, s0, q, u, m)$, that has the following properties, is a monotonic semi-lattice and subsequently a CRDT.*

1. *Set $S$ of payload values forms a semilattice ordered by $\leq$.*
2. *Merging state $s$ with remote state $s'$ computes the LUB of the two states, i.e., $s \cdot m(s') = s \sqcup s'$.*
3. *State is monotonically non-decreasing across updates, i.e., $s \leq s \cdot u$.*

For simplicity and to avoid confusion in notations, we also denote $s \cdot m(s')$ as $s \vee s'$ from this point.

### 3.3 Instances of CRDTs

A typical example of a CRDT object is an add-only set [5].

**Definition 2.** ***(Add-Only Set).*** *An add-only set is defined as $(S, \leq, s^0, q, u, m)$ where*

$$s \vee s' = s \cup s'$$

$$s \cdot u = s \cup u$$

$$s^0 = \emptyset$$

$$s \leq s' \iff s \subseteq s$$

Given any CRDT as values, a map is also a CRDT [27].

**Definition 3.** ***(Merge Map).***
*Given a CRDT object $(S, \leq, s^0, q', u', m')$, a merge map is defined as a tuple $(M \equiv \{k \mapsto s\}, \leq, m^0, q, u, m)$ where*

$$\begin{aligned} m \vee m' = &\{k \mapsto m(k) \vee m'(k) | \forall k \in dom(m) \cap dom(m')\} \\ &\cup \{k \mapsto m(k) | \forall k \in dom(m) \backslash dom(m')\} \\ &\cup \{k \mapsto m'(k) | \forall k \in dom(m') \backslash dom(m)\} \end{aligned}$$

$$m \cdot u = m \cup u$$

$$m \leq m' \iff dom(m) \subseteq dom(m') \ \ \& \ \ \forall k, m(k) \leq m'(k)$$

$$m^0 = \{\}$$

*Here, we define map look up operation for map $m \equiv \{k \mapsto s\}$ as*

$$m(k) = \begin{cases} \emptyset & if\, k \notin dom(m) \\ \{s\} & if\, k \in dom(m) \end{cases}$$

### 3.4 Guarantees and Drawbacks

CRDTs guarantee strong eventual consistency, which allows every processor to reach the same state eventually. Some algorithm like distributed counter and text collaboration systems already utilized CRDTs [28, 30], where such a property is used to guarantee every end client eventually have the same text displayed.

However, even with strong eventual consistency, CRDTs can not guarantee the final result is independent of execution order. This kind of non-determinism is the main problem that

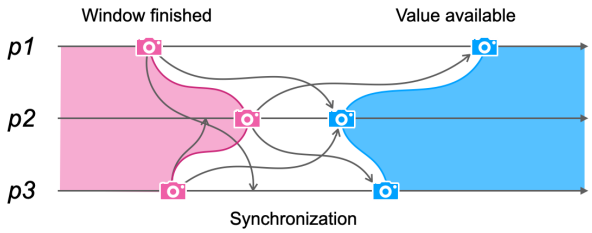


**Figure 1.** Phases one window in Windowed CRDTs

our paper is aiming to resolve. Furthermore, in the context of SPSs, the ever-ongoing updates could cause processors to never reach a consistent state and make the guarantee not very useful. In our paper, we use a new concept of strong eventual consistency for windows (Section 7.3) to provide a stronger guarantee for it.

# 4 Windowed CRDTs

## 4.1 Overview

A CRDT has many limitations upon its usage. Its data depends on the execution order of the distributed system. Thus, any update of the CRDT based on the current CRDT value may not be consistent among different execution orders. Furthermore, its guarantee of eventual consistency may never happen in a stream processing scenario. To address these problems, we introduce windows as checkpoints during the process. With checkpoints, we can make sure that after a stage is finished for everyone, it will no longer be modified and eventually converge. The restriction can thus be relaxed to depend on only completed windows or local modifications.

The phases of each window in windowed CRDTs are demonstrated in Figure 1. A window is first finished by every processor and then starts the synchronization via the CRDT mechanism. And after that, the window value would be available to each processor.

## 4.2 Stream Processing System Model

Consider a scenario of a stream processing system.

There are a set of $n$ stream processors $p_1, p_2, \cdots, p_n$. Each processor may crash at any time without being noticed. For simplicity, we assume that there is no fault recovery mechanism.

Each processor has a stream of events delivered to it in order. At any point, the processor may decide that it has finished the previous window and move into a new window.

All processors share a global state with windowed CRDT, the user program may interact with windowed CRDT via provided interfaces. Also, the user program is pure or deterministic such that it always behaves the same given the same input.

## 4.3 Guarantees

Here we will provide a conceptual definition and a formal definition is given later (Section 7).

- **Strong eventual consistency**: Eventually, once all replicas have received the same updates, they converge to the same state.
- **Strong eventual consistency for windows**: Eventually, once all replicas have received all updates for a window, the replicas will have the same window state.
- **Confluence for windows**: Given the same set of events, the eventual value of a window is always the same, regardless of the processing order.

## 4.4 Windowed CRDT Object

To make the window query deterministic, a naive approach would be trying to take a snapshot exactly when the window is just processed. However, such a moment could have never existed, as processors would not wait for other processors to finish the current window. Thus, what is needed here is a way to reconstruct the snapshot as if all processors stop after the window.

**Definition 4.** ***(Windowed CRDT Object).*** *Given a CRDT $S$, a corresponding windowed CRDT object is defined as a tuple $(S_{local}, F, Gp, w)$ where*

- *Local state $S_{local}$ is a record of local operations on the underlying CRDT that is not affected by the merge operation.*
- *Finished flag $F \equiv \{p \mapsto w\}$ is a map from processor ID to its latest completed window number.*
- *Global progress $Gp \equiv \{w \mapsto S\}$ is a map from the window number to its staged CRDT.*
- *Window number $w$ is the current window being processed (equals to the latest completed window number plus one). We assume that the window number starts from 0.*

In principle, a finished window's value would be a join of all processors' local states when they finished the given window. $S_{local}$ and $w$ are not affected by other processors and are always accessible for the processor while $Gp$ and $F$ are not.

**Definition 5.** ***(CRDT Operation for Windowed CRDT).*** *For any windowed CRDT object $\sigma$, define its CRDT operation as $(\sigma \equiv (S_{local}, F, Gp, w), \leq, \sigma^0, q, u, m)$ where*

$$(S_{local}, F, Gp, w) \leq (S'_{local}, F', Gp', w') \iff F \leq F', Gp \leq Gp'$$

$$\sigma^0 = (s^0, \{\}, \{\}, 0)$$

$$(S_{local}, F, Gp, w) \vee (S'_{local}, F', Gp', w') = (S_{local}, F \sqcup F', Gp \sqcup Gp', w)$$

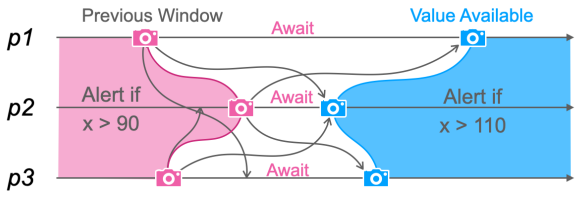


**Figure 2.** Update alerting rules with Windowed CRDT

*Note here that $Gp \sqcup Gp'$ will require a merge strategy for the integer. A simple max-win strategy is used here and we always pick the larger one as the merge result.*

### 4.5 Synchronization and Determinism

To guarantee a stable result regardless of processing order, CRDT's update must be independent of its value, which is a very strong restriction. In windowed CRDT, such restriction can be relaxed so that the update can only depend on past windows but not current in-progress windows. Thus, the program can make use of the value of the CRDT and its synchronization mechanism.

The synchronization in windowed CRDTs happens automatically when a window is finished. The processors will notify each other with a merge message. This process happens concurrently and does not block any message processing. However, when a processor requires some window's value, the processor will have to wait until enough merge messages arrive if it is not available yet so that it can decide the value for that window.

One may consider a *tryAwait* operation in windowed CRDT. However, this will break the guarantee of confluence, as information on merge status is leaked to the program and will result in non-determinism.

### 4.6 Example

Consider a rule-based alerting system [17] using windowed CRDT. A website is trying to prevent DDoS attacks. It has 3 servers and wants to detect anomalies in the amount of visitors. The rule is that every window contains 10,000 messages and if the number of visitors is 10% larger than the average value of the last window, it will send an alert to the IT department. Figure 2 shows how a window would be functioning in this situation.

Before the window, old rules are applied and when each processor reaches the window, it stops and waits for the value to be available. After it is available, the rule is updated for each processor, and processing is continued.

## 5 Syntax

### 5.1 Overview

To model the interaction between the user program and windowed CRDT, consider a new programming model. The program using windowed CRDTs should be able to access the local states (which is the local modification in windowed CRDTs) and after waiting it should gain access to any passed window's value. The program also needs to be Turing complete. Thus, we introduce a syntax based on value untyped lambda calculus and extended by several primitives that enable the interaction with windowed CRDTs.

$$
\begin{array}{lll}
t & ::= & \textit{terms}: \\
& \mid\ x & \textit{variable} \\
& \mid\ () & \textit{unit} \\
& \mid\ \lambda x.\, t & \textit{abstraction} \\
& \mid\ t\ t & \textit{application} \\
& \mid\ \textit{localState} & \textit{state} \\
& \mid\ \textit{putCRDT}\ t; t & \textit{update} \\
& \mid\ \textit{nextWindow}; t & \textit{next} \\
& \mid\ \textit{await}\ t & \textit{await}
\end{array}
$$

**Figure 3.** Syntax for Windowed CRDT

### 5.2 Syntax

The syntax is shown in Figure 3.

In the syntax, we have some interaction between CRDT and the user program. We use church numerals to encode and decode integers. Some

Details of each rule are as follows:

1. *unit*: Unit value. Any correct lambda should be evaluated to a unit value.
2. *abstraction*: a lambda term.
3. *application*: function application.
4. *state*: Return a tuple of the local modifications $S_l ocal$ and current window number.
5. *update*: Update CRDT with the first parameter and return the second parameter. The update is required to be monotonic.
6. *next*: Make windowed CRDT complete the current window and move to the next one. The parameter will be returned.
7. *await*: Wait until the window's value is ready to be read. The value of that window is returned when the waiting is finished.

Here we say term rule *variable*, *unit*, and *abstraction* is a value type or in normal form since these terms can not be further evaluated.

Some operations like *await* require an integer and others like *putCRDT* and *local* require the ability to encode CRDT as lambda calculus and decode it. See Appendix A for details.

### 5.3 Example

In Section 4.6, we consider one possible application of windowed CRDTs, which is a simple rule-based alerting system. Now with the syntax, we can show the exact term that the

```
let crdt = localState.1
in let window = localState.2
in let crdt' =
   -- Move to next window every 10,000 messages
   if crdt.1.1 % 10000 == 0 and crdt.1.1 /= 0 then
        nextWindow ;
        -- Update the second GCounter to the value of
        -- the first GCounter of the last window
        let lastWindow = (await window).2
        in (crdt.1, crdt.2, lastWindow)
   else
        crdt
in
   -- Send alert if n is too large
   let avg =
    let crdt'' = await (localState.2 - 1)
    in (crdt''.2 - crdt''.3) / 30000
   in alertIf (avg * 1.1) n ;
      -- Update CRDT
      putCRDT ((crdt'.1.1 + 1, crdt'.1.2, crdt'.1.3)
           , crdt'.2 + n
           , crdt'.3) ; ()
```

**Listing 1.** Term for processor 1

system would use to process the incoming messages. Note that some extensions are shown in Appendix A for simplicity.

In this case, we could construct the windowed CRDT as a tuple of CRDTs.

$$(LocalStates, GCounter, GCounter)$$

where

$$\begin{aligned} LocalStates = \\ &( LastWriteWin\ \ Int \\ &, LastWriteWin\ \ Int \\ &, LastWriteWin\ \ Int) \end{aligned}$$

*LocalStates* contains a counter of the number of messages for each processor. The first grow-only counter *GCounter* contains the sum of all messages, and the second *GCounter* contains the sum of all messages excluding the last window. Thus, we can calculate the sum of messages in the last window by subtracting the two counters.

Assume the message is $n$. The term for processor 1 is shown in Listing 1. The terms for other processors can be easily derived.

## 6 Operational Semantics

The semantics for windowed CRDT are shown in Figure 4 and Figure 5.

### 6.1 Overview

Suppose that there are $n$ processors, for a processor $k$ it should have the following fields:

$$\Sigma(k) = (\sigma, i, t)$$

where $\sigma$ is its windowed CRDT state, $i$ is the index of the current message being processed and $t$ is the term to be evaluated.

To model pending merge messages, we introduce a set of messages in transit,

$$\mathcal{T} \equiv \{(\sigma, k)\}$$

where the windowed CRDT $\sigma$ is sent to processor $k$.

There is also a stream of messages that are dispatched to processors, denoted $m_i^j$ where $m_i^j$ is the $j$th event dispatched to the $i$th processor.

$$\mathcal{M} = \begin{Bmatrix} m_1^1 & m_1^2 & \cdots \\ m_2^1 & m_2^2 & \cdots \\ \vdots & \vdots & \ddots \end{Bmatrix}$$

For convenience, we assume that each message is just a user program defined by the syntax. Each message's processing is finished when the current processor term is evaluated to ().

### 6.2 Semantics of Global Evaluation

Figure 4 shows evaluation rules that operate on a global level. The rules are in the following form.

$$\mathcal{M}, \Sigma, \mathcal{T} \to \mathcal{M}, \Sigma', \mathcal{T}'$$

This includes

1. Eval: Apply one-step evaluation on a certain processor. It lifts local evaluation semantics into the global evaluation context.
2. NextMessage: Load the next message to an idle processor with a term () which indicates that it has finished the previous message.
3. Recv: Merge a message from $\mathcal{T}$ to the target processor. A lattice join is carried out.

### 6.3 Semantics of Local Evaluation

Figure 5 shows evaluation rules for processor $k$ 's lambda term. The rules are in the following form and are lifted to the global context via Eval.

$$\mathcal{M}, \Sigma, \mathcal{T}, k \mid t \to \mathcal{M}, \Sigma', \mathcal{T}', k \mid t'$$

This includes

1. App1, AppAbs and App2: Standard lambda calculus evaluation.
2. Await: Wait until the given window's value is ready, this rule can only be applied when the processor has the value (indicated by $\sigma.F$).
3. LocalState: Return a pair of local state $S_{local}$ and window number $\sigma.w$.
4. Update: Put an updated CRDT to $S_{local}$. The $\sqcup$ is not necessary but is used to guarantee the update is monotonic.

$$\textsc{Eval}\ \frac{k \in [1,n] \quad \Sigma(k).t = t \quad \mathcal{M}, \Sigma, \mathcal{T}, k \mid t \xrightarrow{R} \mathcal{M}, \Sigma', \mathcal{T}', k \mid t'}{\mathcal{M}, \Sigma, \mathcal{T} \xrightarrow{Eval(k,R)} \mathcal{M}, [\Sigma(k).t/t']\Sigma', \mathcal{T}'}$$

$$\textsc{NextMessage}\ \frac{k \in [1,n] \quad \Sigma(k) = (\sigma_k, i, ())}{\mathcal{M}, \Sigma, \mathcal{T} \xrightarrow{NextMessage(k)} \mathcal{M}, [\Sigma(k)/(\sigma_k, i+1, m_k^{i+1})]\Sigma, \mathcal{T}}$$

$$\textsc{Recv}\ \frac{(\sigma_i, j) \in \mathcal{T}}{\mathcal{M}, \Sigma, \mathcal{T} \xrightarrow{Recv(\sigma_i,j)} \mathcal{M}, [\Sigma(j).\sigma/\Sigma(j).\sigma \sqcup \sigma_i]\Sigma, \mathcal{T} \backslash \{(\sigma_i, j)\}}$$

**Figure 4.** Operational Semantics for Windowed CRDT Object (Global)[1]

$$\textsc{App1}\ \frac{\mathcal{M}, \Sigma, \mathcal{T}, k \mid t_1 \to \mathcal{M}, \Sigma', \mathcal{T}', k \mid t_1'}{\mathcal{M}, \Sigma, \mathcal{T}, k \mid t_1\ t_2 \xrightarrow{App1} \mathcal{M}, \Sigma', \mathcal{T}', k \mid t_1'\ t_2} \qquad \textsc{App2}\ \frac{\mathcal{M}, \Sigma, \mathcal{T}, k \mid t_2 \to \mathcal{M}, \Sigma', \mathcal{T}', k \mid t_2'}{\mathcal{M}, \Sigma, \mathcal{T}, k \mid v_1\ t_2 \xrightarrow{App2} \mathcal{M}, \Sigma', \mathcal{T}', k \mid v_1\ t_2'}$$

$$\textsc{Await}\ \frac{decode\ t = w \quad \Sigma(k).Gp(w) = \sigma_w \quad \forall i \in [1,n], (i,w) \in \Sigma(k).F}{\mathcal{M}, \Sigma, \mathcal{T}, k \mid await\ t \xrightarrow{Await} \mathcal{M}, \Sigma, \mathcal{T}, k \mid encode\ \sigma_w}$$

$$\textsc{AppAbs}\ \frac{}{\mathcal{M}, \Sigma, \mathcal{T}, k \mid \lambda x.\ t_1\ v_2 \xrightarrow{AppAbs} \mathcal{M}, \Sigma, \mathcal{T}, k \mid [x \mapsto v_2]t_1}$$

$$\textsc{NextWindow}\ \frac{\Sigma(k).\sigma = (S_{local}, F, Gp, w) \quad \Sigma' = [\Sigma(k).\sigma/(S_{local}, [F(k)/w], [Gp(w)/Gp(w) \sqcup S_{local}]Gp, w+1)]\Sigma}{\mathcal{M}, \Sigma, \mathcal{T}, k \mid nextWindow; t \xrightarrow{NextWindow} \mathcal{M}, \Sigma', \mathcal{T} \bigcup \{(\Sigma'(k).\sigma, i) \mid i \in [1,n]\}, k \mid t}$$

$$\textsc{LocalState}\ \frac{}{\mathcal{M}, \Sigma, \mathcal{T}, k \mid localState \xrightarrow{LocalState} \mathcal{M}, \Sigma, \mathcal{T}, k \mid encode\ (\Sigma(k).\sigma.S_{local}, \Sigma(k).\sigma.w)}$$

$$\textsc{Update}\ \frac{\Sigma(k).\sigma.S_{local} \leq decode\ t_1}{\mathcal{M}, \Sigma, \mathcal{T}, k \mid putCRDT\ t_1\ t_2 \xrightarrow{Update} \mathcal{M}, [\Sigma(k).\sigma.S_{local}/\Sigma(k).\sigma.S_{local} \sqcup decode\ t_1]\Sigma, \mathcal{T}, k \mid t_2}$$

**Figure 5.** Operational Semantics for Windowed CRDT Object (Local)

5. NextWindow: Finish current window. Finished set $F$ and global progress $Gp$ is updated. The processor will also send a merge message to all processors for synchronization.

## 7 Proof of Guarantees

Note that the semantics may lead to a normal form that is not (), or there could be *await* that can not be completed. We say the system is correct if all messages are evaluated to () within finite steps and *decode* always gets a number. In the following definition, we implicitly assume that **the system is correct**.

### 7.1 Confluence

Regardless of the evaluation order, windowed CRDTs shall always yield the same output. In order words, the evaluation semantics are confluence and commutative, which is shown later.

**Theorem 1.** ***(Confluence of Windowed CRDT Evaluation.)*** *Windowed CRDT evaluation is confluent if the following property holds. For any given $\mathcal{M}, \Sigma, \mathcal{T} \xrightarrow{*} \mathcal{M}, \Sigma_1, \mathcal{T}_1$ and $\mathcal{M}, \Sigma, \mathcal{T} \xrightarrow{*} \mathcal{M}, \Sigma_2, \mathcal{T}_2$ there exists $\Sigma_3$ and $\mathcal{T}_3$ such that*

$$\mathcal{M}, \Sigma_1, \mathcal{T}_1 \xrightarrow{*} \mathcal{M}, \Sigma_3, \mathcal{T}_3$$

$$\mathcal{M}, \Sigma_2, \mathcal{T}_2 \xrightarrow{*} \mathcal{M}, \Sigma_3, \mathcal{T}_3$$

By the definition, we can show that (given well-defined after interchange) if every pair of small-step evaluations is commutative, then any two evaluations are commutative.

**Definition 6.** ***(Commutativity of Windowed CRDT Evaluation)*** *Windowed CRDT evaluation is commutative if for any given $\mathcal{M}, \Sigma, \mathcal{T}$ with two small step evaluation $f$ and $g$*

[1]Here, we use notation $[t/t']\Sigma$ to denote replacing $t$ with $t'$ in variable $\Sigma$.

*where*

$$\mathcal{M}, \Sigma, \mathcal{T} \xrightarrow{f} \mathcal{M}, \Sigma', \mathcal{T}'$$
$$\mathcal{M}, \Sigma', \mathcal{T}' \xrightarrow{g} \mathcal{M}, \Sigma'', \mathcal{T}''$$

*There always is $f \circ g = g \circ f$ if $f \circ g$ and $g \circ f$ are both well-defined.*

*Note here we denote the composition of small step evaluation as $\circ$, where $f \circ g$ means to apply evaluation rule $f$ first and then $g$.*

**Lemma 1.** ***(Sufficient condition for Confluence.)*** *Windowed CRDT evaluation is confluent if it is commutative.*

*Proof.* Without loss of generality, let $F = \{f_n \mid n \in [1, i]\}$ and $G = \{g_n, n \in [1, j]\}$ be two evaluations where $f$ and $g$ are all small step evaluations. There is

$$\mathcal{M}, \Sigma, \mathcal{T} \xrightarrow{f_1 \circ \cdots \circ f_i} \mathcal{M}, \Sigma_1, \mathcal{T}_1$$
$$\mathcal{M}, \Sigma, \mathcal{T} \xrightarrow{g_1 \circ \cdots \circ g_j} \mathcal{M}, \Sigma_2, \mathcal{T}_2$$

To prove our lemma, we could find a common evaluation result $\mathcal{M}, \Sigma_3, \mathcal{T}_3$ that can be reached from both $\mathcal{M}, \Sigma_1, \mathcal{T}_1$ and $\mathcal{M}, \Sigma_2, \mathcal{T}_2$, and a series of small step evaluations $H$ that satisfies

$$\mathcal{M}, \Sigma, \mathcal{T} \xrightarrow{H} \mathcal{M}, \Sigma_3, \mathcal{T}_3$$

The simplest $H$ is the least common multiple of $F$ and $G$. Consider $H = h_1 \circ h_2 \circ \cdots \circ h_k$ such that

$$\forall n_1 \in [1, i] \text{ and } n_2 \in [i, j], f_{n_1} \in H \text{ and } g_{n_2} \in H$$

. To ensure $H$ is well-defined, $H$ needs to satisfy

$$\forall h_{k_1} = f_x, h_{k_2} = f_y \text{ and } x \geq y, k_1 \geq k_2$$
$$\forall h_{k_1} = g_x, h_{k_2} = g_y \text{ and } x \geq y, k_1 \geq k_2$$

One possible way of producing a well-defined $H$ is to let

$$H = deduplicate(f_1 \circ \cdots \circ f_i \circ g_1 \circ \cdots \circ g_j)$$

Since $\forall f, g, f \circ g = g \circ f$, it is trivial to show that by swapping the order in $H$ we can get

$$\mathcal{M}, \Sigma_1, \mathcal{T}_1 \xrightarrow{*} \mathcal{M}, \Sigma_3, \mathcal{T}_3$$
$$\mathcal{M}, \Sigma_2, \mathcal{T}_2 \xrightarrow{*} \mathcal{M}, \Sigma_3, \mathcal{T}_3$$

□

**Lemma 2.** *windowed CRDT is commutative.*

*Proof.* We will analyze case by case.

***Case 1:*** $\mathcal{M}, \Sigma, \mathcal{T} \xrightarrow{Eval(i,R_1)} \mathcal{M}, \Sigma', \mathcal{T}' \xrightarrow{Eval(j,R_2)} \mathcal{M}, \Sigma'', \mathcal{T}''$ ***where*** $i \neq j$**.**

Let $\mathcal{M}, \Sigma, \mathcal{T} \xrightarrow{Eval(j,R_2)} \mathcal{M}, \Sigma_1', \mathcal{T}_1' \xrightarrow{Eval(i,R_1)} \mathcal{M}, \Sigma_1'', \mathcal{T}_1''$.

Notice that without Recv, $\mathcal{T}$ is insertion only and thus there will always be $\mathcal{T}'' = \mathcal{T}_1''$

By definition, there is $\Sigma'' = \Sigma_1'' = [\Sigma(i).t/t_i, \Sigma(j).t/t_j]\Sigma$.

Thus, in this case, it is commutative.

***Case 2:*** $\mathcal{M}, \Sigma, \mathcal{T} \xrightarrow{Eval(i,R_1)} \mathcal{M}, \Sigma', \mathcal{T}' \xrightarrow{Eval(i,R_2)} \mathcal{M}, \Sigma'', \mathcal{T}''$**.**

In this case, there is a dependency between the evaluation order. Thus, it will not be well-defined after the interchange of the two steps.

***Case 3:*** $\mathcal{M}, \Sigma, \mathcal{T} \xrightarrow{NextMessage(i)} \mathcal{M}, \Sigma', \mathcal{T} \xrightarrow{*} \mathcal{M}, \Sigma'', \mathcal{T}'$**.**

The analysis of this case is similar to Case 1 and Case 2. The only special case here is that if the order is $Eval(i, *) \rightarrow NextMessage(i)$ it would be undefined after swap due to the dependence. Otherwise, it is obvious to tell that it is commutative.

***Case 4:*** $\mathcal{M}, \Sigma, \mathcal{T} \xrightarrow{Recv(\sigma_1,i)} \mathcal{M}, \Sigma', \mathcal{T}' \xrightarrow{Recv(\sigma_2,j)} \mathcal{M}, \Sigma'', \mathcal{T}''$**.**

Since $\sigma$ is commutative (since $\sigma$ is a semi-join lattice), there is $\sigma = \sigma_i \sqcup \sigma_1 \sqcup \sigma_2 = \sigma_i \sqcup \sigma_2 \sqcup \sigma_1$. Thus, $\Sigma'' = [\sigma_i/\sigma]\Sigma$ regardless of evaluation order.

If $i \neq j$, then it is obvious to be commutative.

***Case 5:*** $\mathcal{M}, \Sigma, \mathcal{T} \xrightarrow{Recv(\sigma,i)} \mathcal{M}, \Sigma', \mathcal{T}' \xrightarrow{Eval(j,R)} \mathcal{M}, \Sigma'', \mathcal{T}''$ ***where*** $i \neq j$**.**

It is trivial to show that this case is commutative, as they do not interfere with each other.

***Case 6:*** $\mathcal{M}, \Sigma, \mathcal{T} \xrightarrow{Recv(\sigma,i)} \mathcal{M}, \Sigma', \mathcal{T}' \xrightarrow{Eval(i,R)} \mathcal{M}, \Sigma'', \mathcal{T}''$**.**

Notice that lattice join will only affect $F$ and $Gp$, thus there are only two case worth considering, namely $Recv(\sigma, i) \rightarrow Eval(i, NextWindow)$ and $Recv(\sigma, i) \rightarrow Eval(i, Await)$.

In the first case, applying the same argument made in Case 4 (idempotency), it is easy to show that this case is commutative. For the second case, if Await needs the Recv to be performed, it is not well-defined after the swap. Otherwise, it is straightforward that it is commutative.

By now, all cases have been proven. □

**Theorem 2.** *Windowed CRDT evaluation is confluent.*

*Proof.* By Lemma 2 and Lemma 1, windowed CRDT evaluation is confluent. □

### 7.2 Strong Eventual Consistency

Assuming a finite amount of messages, eventually, after all messages have been processed and updates delivered, all processors shall have an equivalent state.

**Definition 7.** ***(Strong Eventual Consistency).*** *For all evaluation $\mathcal{M}, \Sigma, \mathcal{T} \xrightarrow{*} \mathcal{M}, \Sigma', \mathcal{T}'$, if $\mathcal{M}, \Sigma', \mathcal{T}'$ is a normal form., then there is always*

$$\forall i, j \in [1, n], \Sigma'(i).\sigma = \Sigma'(j).\sigma$$

**Theorem 3.** *Windowed CRDT's evaluation satisfies strong eventual consistency.*

*Proof.* Consider a special case, every NextWindow is followed by $n$ Recv and there is no SendMerge. Suppose this takes $x$ small step evaluation. Notice that for all $i \in [1, n]$, $\Sigma(i).\sigma = \bigsqcup_{j \in [i,n]} \Sigma(j).\sigma$. Since $\sigma$ is a semi-join lattice, there must be $\sigma_1 = \sigma_2 = \cdots = \sigma_n$.

By Theorem 2, all other possible evaluation paths a rewritten of the above case and thus will yield the same conclusion. □

### 7.3 Strong Eventual Consistency for Windows

With the number of messages being finite or infinite, all processors shall have the same value for a given window after all updates for the window are delivered.

**Definition 8.** ***(Strong Eventual Consistency for Windows).*** *For any initial state $\mathcal{M}, \Sigma, \mathcal{T}$ and $w$, if during evaluation $\mathcal{M}, \Sigma, \mathcal{T} \xrightarrow{*} \mathcal{M}, \Sigma', \mathcal{T}'$ every message inserted by NextWindow for $w$ on each processor has been received via Recv then there is*

$$\forall i, j \in [i, n], \Sigma(i).\sigma.Gp = \Sigma(j).\sigma.Gp$$

**Theorem 4.** *Windowed CRDT's evaluation satisfies strong eventual consistency for windows.*

*Proof.* Given window $w$, let the local state for each processor after completing window $w$ be $S^1_{local}, S^2_{local}, \cdots, S^n_{local}$. By definition of NextWindow and Recv, we have

$$\forall i \in [1, n].\Sigma(i).\sigma.Gp(w) = \bigsqcup_{j \in [1,n]} S^j_{local}$$

This means every processor has the same value for window $w$. □

## 8 Transparent Failure Recovery

### 8.1 Scenario of Failure Recovery

In a distributed system, failures happen at any given time. Failure may happen with some messages being sent to notify other actors but more often quietly without any warning. It is typically difficult to recover a stateful system [18, 21], as failure usually means loss of internal mutable state and also potentially leads to corrupted global state. Failure transparency [36] means each failure is masked in such a way that the user cannot observe the failure, and is typically achieved by taking snapshots of the whole system. An example of snapshotting in a distributed system is Apache Flink's recovery protocol [10, 12, 13].

With windowed CRDTs, windows can be viewed as an asynchronous and decentralized snapshot. Thus, given there is at least one healthy actor, we could reset any failed actor to its last known valid state, which is the last completed window in the healthy actor.

### 8.2 Guarantees of Failure Recovery with Windowed CRDTs

We provide the following guarantees.

1. Failure is transparent to users.
2. Failure recovery is confluence.

To be specific about confluence, we can define confluence for failure recovery as the following.

**Definition 9.** ***(Confluence for Failure Recovery).*** *A system $\mathcal{F}$ is said to be confluent regarding failure if for any failure*

$$\mathcal{F} \to \mathcal{F}_{failed} \xrightarrow{*} \mathcal{F}_{reocvered}$$

*There exists a $\mathcal{F}'$ such that*

$$\mathcal{F} \xrightarrow{*} \mathcal{F}'$$

$$\mathcal{F}_{recovered} \xrightarrow{*} \mathcal{F}'$$

In such a way, we can guarantee that windowed CRDTs still yield deterministic results regardless of any potential failure.

### 8.3 Additional Requirements

To reconstruct a failed processor, additional information about the position of the message stream for each actor at the end of each window should be stored in $S_{local}$. Also, since we will use window value to replace local value, to guarantee confluence, $S_{local}$ should keep CRDT and local mutable state separately.

$$S_{local} = (index, CRDT, state)$$

And semantics for LocalState should also be adjusted accordingly, as shown in Figure 6.

### 8.4 Algorithm for Failure Transparent Recovery

To reconstruct a failed processor, additional information about the position of the message stream for each actor at the end of each window should be stored in $S_{local}$. Denote this by $S_{local}.index$.

On a failure, when recovering a processor $p_i$, apply Algorithm 1.

**Algorithm 1** Failure Recovery Algorithm

```
1: procedure Recover(i, M, Σ)
2:     j ← random element [1, n] where p_j is alive
3:     w ← min {codom(Σ(j).σ.F)}
4:     Σ' ← Σ
5:     Σ'(i).σ ← (Σ(j).Gp(w), Σ(j).F, Σ(j).Gp, w)
6:     Σ'(i).i ← Σ(i).σ.S_local.index
7:     Σ'(i).t ← M.m_i^{Σ(i).i}
8:     return Σ'
9: end procedure
```

$$\textsc{LocalState}\ \frac{\Sigma(k).\sigma = \sigma_k}{\mathcal{M}, \Sigma, \mathcal{T}, k \mid localState \xrightarrow{LocalState} \mathcal{M}, \Sigma, \mathcal{T}, k \mid encode\ (\sigma_k.S_{local}.state, \sigma_k.S_{local}.i, \sigma_k.w)}$$

**Figure 6.** Adjusted LocalState Semantics for Failure Recovery

The same procedure to prove confluence for windowed CRDTs (Section 7.1) can be easily applied here. For simplicity, we will omit the complete proof here.

# 9 Implementation and Evaluation

## 9.1 Overview

We have implemented a prototype of windowed CRDT with Scala and Pekko [20] framework. Primitives in Figure 4 are implemented as a handle monad. All windowed CRDT features are implemented, as well as a simple failure transparent recovery system. Our code is available on GitHub[2].

## 9.2 Performance Evaluation

**Experiment Goal and Setup.** We conduct a performance measurement on the implemented system. The task is to calculate the sum of a stream of integers, and a grow-only counter [31] is used to construct the windowed CRDT. The experiment is carried out on an 8-core machine and each actor is allocated with a dedicated thread. Each case is measured 5 times to counter random factors. Our goal is to explore the effect of different factors on the system's performance.

In the experiment, we consider the following factors:

1. Number of Windows: Synchronization happens at the end of every window, and the cost of synchronization is $O(n^2)$ for a system of $n$ actors. In our task, synchronization is a relatively expensive operation compared to message processing, even if the system is running fully locally.
2. Number of Awaits: Await is where synchronization is enforced. The actor will be forced to stop processing until the required data is ready.
3. Number of Actors: Different actors can execute tasks concurrently.

**Experiment Design.** To study the effect of different factors, we utilize control variable methods here. We perform 3 groups of experiments in total and for each group, we carry out 3 experiments. A control group contains cases with no windows, 20 windows, and 40 windows while performing await for every window. Two experiment groups, both contain cases with 200 and 400 windows, while one performs await every window and the other one performs await every 10 windows.

In such a way, group 1 and group 3 have the same amount of awaits and group 2 and group 3 have the same amount of windows. All groups are processing the same amount of messages.

**Experiment Result.** The result is shown in Figure 7. One important fact we need to consider is that our software could not fully utilize all 4 cores, as the operating system and other programs would also occupy some computation resources. Consider that we can conclude:

1. **Baseline performance scales almost linearly.** Baseline case does not have any synchronization. It should scale linearly until all cores are occupied and stay there. Our results mostly match the expectations.
2. **Windows has a significant impact on performance, and this impact grows fast with the number of actors.** Comparing the results of group 1 with group 2 and group 3. There is a notable performance decrement as the number of windows grows as well as the number of actors grows. This is expected as the cost of synchronization grows by $O(n)$ and $O(n^2)$ respectively.
3. **Awaits almost do not impact performance.** Comparing group 2 and group 3, the performance impact is insignificant regarding the amount of awaits. This is also expected to await only force synchronization, such that the performance is decided by the slowest actor. In our experiment, all actors have a similar task and there should not be much speed difference.

**In real-world geographically distributed systems.** Real-world distributed systems' synchronization would be much slower. Although in windowed CRDTs, this is happening concurrently and does not interfere with normal processing, the performance can suffer from await operation where synchronization is enforced. This is usually unavoidable due to data dependency in computation, while we advise the developers should always try to minimize the amount of await operations.

# 10 Related Work

Previous work has already considered using CRDTs to approximate statistics about streams of data [1] on sliding windows with sketches [16]. It aims to adapt some common algorithms for stream aggravation such as frequency and entropy [4] to a distributed stream processing system. Some other works also focus on safe shared states in dataflow systems without using CRDTs [25], where safety relies more on a correctly implemented concurrent data structure.

[2]https://github.com/TerenceNg03/Windowed-CRDT

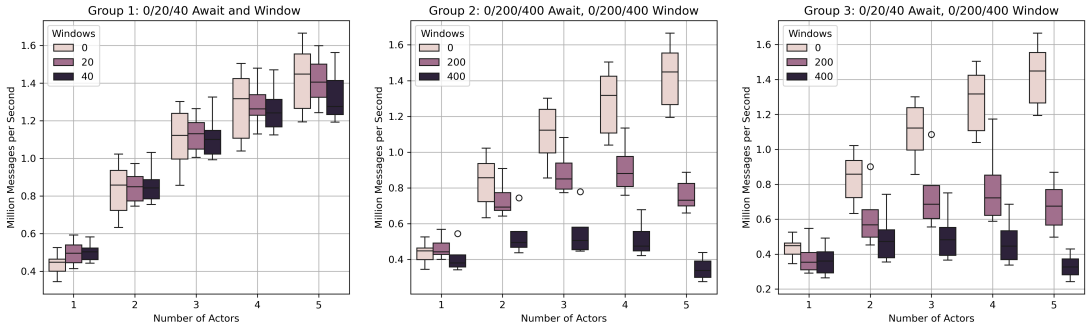


**Figure 7.** Performance Measured on 8 Core Machine, 1 Million Messages per Actor, Repeated 5 times

# 11 Conclusions and Future Work

## 11.1 Conclusions

In conclusion, windowed CRDTs are a new way to utilize globally shared state in a distributed stream processing system, as well as providing deterministic processing results. It is highly efficient since it minimizes the amount of blocking operations. While there are still many more optimizations that can be done, we would expect more results to come out of it.

## 11.2 Future Work

Windowed CRDTs, by their nature, will record every passed window value. This will cause a memory leak and thus a hit on performance. In our current model, this is unfortunately unable to be addressed. However, if the passed window value is provided via a parameter or reference, it may be possible to implement a distributed garbage collection algorithm upon windowed CRDTs and give a bounded memory usage.

In this paper, only state-based CRDTs are considered. However, in some situations, operation-based CRDTs could be more efficient than state-based ones. Thus, one of the possible future works is to expand the semantics to operation-based CRDTs.

Another thing worth considering in failure recovery is that other processors may have a better recovery point than $p_j$. When a merge message from such a processor $p_{j'}$ is received, we could simply run Algorithm 1 for $p_{j'}$ again, so that $p_i$ 's state is fast-forward to the latest known window and avoids redundant computation.

# Acknowledgments

We acknowledge funding from Digital Future.

# References

[1] Dolev Adas and Roy Friedman. 2021. Sliding Window CRDT Sketches. In *40th International Symposium on Reliable Distributed Systems, SRDS 2021, Chicago, IL, USA, September 20-23, 2021*. IEEE, 288–298. https://doi.org/10.1109/SRDS53918.2021.00036

[2] Adil Akhter, Marios Fragkoulis, and Asterios Katsifodimos. 2019. Stateful Functions as a Service in Action. *Proc. VLDB Endow.* 12, 12 (2019), 1890–1893. https://doi.org/10.14778/3352063.3352092

[3] Deepthi Devaki Akkoorath, Alejandro Z. Tomsic, Manuel Bravo, Zhongmiao Li, Tyler Crain, Annette Bieniusa, Nuno M. Preguiça, and Marc Shapiro. 2016. Cure: Strong Semantics Meets High Availability and Low Latency. In *36th IEEE International Conference on Distributed Computing Systems, ICDCS 2016, Nara, Japan, June 27-30, 2016*. IEEE Computer Society, 405–414. https://doi.org/10.1109/ICDCS.2016.98

[4] Eran Assaf, Ran Ben-Basat, Gil Einziger, and Roy Friedman. 2018. Pay for a Sliding Bloom Filter and Get Counting, Distinct Elements, and Entropy for Free. In *2018 IEEE Conference on Computer Communications, INFOCOM 2018, Honolulu, HI, USA, April 16-19, 2018*. IEEE, 2204–2212. https://doi.org/10.1109/INFOCOM.2018.8485882

[5] Annette Bieniusa, Marek Zawirski, Nuno M. Preguiça, Marc Shapiro, Carlos Baquero, Valter Balegas, and Sérgio Duarte. 2012. An optimized conflict-free replicated set. *CoRR* abs/1210.3368 (2012). arXiv:1210.3368 http://arxiv.org/abs/1210.3368

[6] Cihan Biyikoglu. 2024. Under the Hood: Redis CRDTs. https://redis.io/resources/under-the-hood/ Accessed on 2024-07-25.

[7] Irina Botan, Roozbeh Derakhshan, Nihal Dindar, Laura Haas, Renée J Miller, and Nesime Tatbul. 2010. Secret: A model for analysis of the execution semantics of stream processing systems. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 232–243.

[8] Irina Botan, Roozbeh Derakhshan, Nihal Dindar, Laura M. Haas, Renée J. Miller, and Nesime Tatbul. 2010. SECRET: A Model for Analysis of the Execution Semantics of Stream Processing Systems. *Proc. VLDB Endow.* 3, 1 (2010), 232–243. https://doi.org/10.14778/1920841.1920874

[9] Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. 2014. Replicated data types: specification, verification, optimality. (2014), 271–284. https://doi.org/10.1145/2535838.2535848

[10] Paris Carbone. 2018. *Scalable and Reliable Data Stream Processing*. Ph. D. Dissertation. Royal Institute of Technology, Stockholm, Sweden. https://nbn-resolving.org/urn:nbn:se:kth:diva-233527

[11] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. 2017. State Management in Apache Flink®: Consistent Stateful Distributed Stream Processing. *Proc. VLDB Endow.* 10, 12 (2017), 1718–1729. https://doi.org/10.14778/3137765.3137777

[12] Paris Carbone, Gyula Fóra, Stephan Ewen, Seif Haridi, and Kostas Tzoumas. 2015. Lightweight Asynchronous Snapshots for Distributed Dataflows. *CoRR* abs/1506.08603 (2015). arXiv:1506.08603 http://arxiv.org/abs/1506.08603

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

[14] K. Mani Chandy and Leslie Lamport. 1985. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Trans. Comput. Syst.* 3, 1 (1985), 63–75. https://doi.org/10.1145/214451.214456

[15] Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska, Donald Carney, Ugur Çetintemel, Ying Xing, and Stanley B. Zdonik. 2003. Scalable Distributed Stream Processing. In *First Biennial Conference on Innovative Data Systems Research, CIDR 2003, Asilomar, CA, USA, January 5-8, 2003, Online Proceedings*. www.cidrdb.org. http://www-db.cs.wisc.edu/cidr/cidr2003/program/p23.pdf

[16] Graham Cormode and S. Muthukrishnan. 2005. An improved data stream summary: the count-min sketch and its applications. *J. Algorithms* 55, 1 (2005), 58–75. https://doi.org/10.1016/J.JALGOR.2003.12.001

[17] Nick G. Duffield, Patrick Haffner, Balachander Krishnamurthy, and Haakon Ringberg. 2009. Rule-Based Anomaly Detection on IP Flows. In *INFOCOM 2009. 28th IEEE International Conference on Computer Communications, Joint Conference of the IEEE Computer and Communications Societies, 19-25 April 2009, Rio de Janeiro, Brazil*. IEEE, 424–432. https://doi.org/10.1109/INFCOM.2009.5061947

[18] E. N. Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. 2002. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.* 34, 3 (2002), 375–408. https://doi.org/10.1145/568522.568525

[19] The Apache Software Foundation. 2024. Apache Flink Stateful Functions. https://nightlies.apache.org/flink/flink-statefun-docs-release-3.2/. Accessed on 2024-07-05.

[20] The Apache Software Foundation. 2024. Apache Pekko. https://pekko.apache.org/ Accessed on 2024-07-25.

[21] Felix C. Gärtner. 1999. Fundamentals of Fault-Tolerant Distributed Computing in Asynchronous Environments. *ACM Comput. Surv.* 31, 1 (1999), 1–26. https://doi.org/10.1145/311531.311532

[22] Seth Gilbert and Nancy A. Lynch. 2012. Perspectives on the CAP Theorem. *Computer* 45, 2 (2012), 30–36. https://doi.org/10.1109/MC.2011.389

[23] Philipp M. Grulich, Sebastian Breß, Steffen Zeuch, Jonas Traub, Janis von Bleichert, Zongxiong Chen, Tilmann Rabl, and Volker Markl. 2020. Grizzly: Efficient Stream Processing Through Adaptive Query Compilation. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 2487–2503. https://doi.org/10.1145/3318464.3389739

[24] Haruna Isah, Tariq Abughofa, Sazia Mahfuz, Dharmitha Ajerla, Farhana H. Zulkernine, and Shahzad Khan. 2019. A Survey of Distributed Data Stream Processing Frameworks. *IEEE Access* 7 (2019), 154300–154316. https://doi.org/10.1109/ACCESS.2019.2946884

[25] Luca De Martini and Alessandro Margara. 2024. Safe Shared State in Dataflow Systems. In *Proceedings of the 18th ACM International Conference on Distributed and Event-based Systems, DEBS 2024, Villeurbanne, France, June 24-28, 2024*. ACM, 30–41. https://doi.org/10.1145/3629104.3666029

[26] Derek Gordon Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: a timely dataflow system. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, Michael Kaminsky and Mike Dahlin (Eds.). ACM, 439–455. https://doi.org/10.1145/2517349.2522738

[27] Nuno M. Preguiça. 2018. Conflict-free Replicated Data Types: An Overview. *CoRR* abs/1806.10254 (2018). arXiv:1806.10254 http://arxiv.org/abs/1806.10254

[28] Nuno M. Preguiça, Joan Manuel Marquès, Marc Shapiro, and Mihai Letia. 2009. A Commutative Replicated Data Type for Cooperative Editing. In *29th IEEE International Conference on Distributed Computing Systems (ICDCS 2009), 22-26 June 2009, Montreal, Québec, Canada*. IEEE Computer Society, 395–403. https://doi.org/10.1109/ICDCS.2009.20

[29] Nuno M. Preguiça, Marek Zawirski, Annette Bieniusa, Sérgio Duarte, Valter Balegas, Carlos Baquero, and Marc Shapiro. 2014. SwiftCloud: Fault-Tolerant Geo-Replication Integrated all the Way to the Client Machine. In *33rd IEEE International Symposium on Reliable Distributed Systems Workshops, SRDS Workshops 2014, Nara, Japan, October 6-9, 2014*. IEEE Computer Society, 30–33. https://doi.org/10.1109/SRDSW.2014.33

[30] Marc Shapiro and Nuno M. Preguiça. 2007. Designing a commutative replicated data type. *CoRR* abs/0710.1784 (2007). arXiv:0710.1784 http://arxiv.org/abs/0710.1784

[31] Marc Shapiro, Nuno M. Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-Free Replicated Data Types. In *Stabilization, Safety, and Security of Distributed Systems - 13th International Symposium, SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6976)*, Xavier Défago, Franck Petit, and Vincent Villain (Eds.). Springer, 386–400. https://doi.org/10.1007/978-3-642-24550-3_29

[32] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. 2011. Transactional storage for geo-replicated systems. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011*, Ted Wobber and Peter Druschel (Eds.). ACM, 385–400. https://doi.org/10.1145/2043556.2043592

[33] Jonas Spenger, Chengyang Huang, Philipp Haller, and Paris Carbone. 2023. Portals: A Showcase of Multi-Dataflow Stateful Serverless. *Proc. VLDB Endow.* 16, 12 (2023), 4054–4057. https://doi.org/10.14778/3611540.3611619

[34] Robert Stephens. 1997. A Survey of Stream Processing. *Acta Informatica* 34, 7 (1997), 491–541. https://doi.org/10.1007/S002360050095

[35] Olivier Tardieu, David Grove, Gheorghe-Teodor Bercea, Paul Castro, Jaroslaw Cwiklik, and Edward A. Epstein. 2023. Reliable Actors with Retry Orchestration. *Proc. ACM Program. Lang.* 7, PLDI (2023), 1293–1316. https://doi.org/10.1145/3591273

[36] Aleksey Veresov, Jonas Spenger, Paris Carbone, and Philipp Haller. 2024. Failure Transparency in Stateful Dataflow Systems (Technical Report). arXiv:2407.06738 [cs.PL] https://arxiv.org/abs/2407.06738

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


# A Extensions and Encoding for the Syntax

Consider the following definition:

**Definition 10.** ***(Integer Encoding and Decoding)****. Let id be* $\lambda x.\, x$*, define positive integer as church numerals*

$$encode\ n = \lambda f.\, \lambda x.\, f^{\circ n}\ x$$

$$decode\ n = k \ \ where\ nid() \xrightarrow{k} ()$$

Immediately, we have + and − defined.

$$(+) = \lambda m.\, \lambda n.\, \lambda f.\, \lambda x.\, m\ f\ (n\ f\ x)$$

$$pred = \lambda n. \lambda f.\, \lambda x.\, n\ (\lambda g.\, \lambda h.\, h\ (g\ f))\ (\lambda u.\, x)\ (\lambda u.\, u)$$

$$(-) = \lambda m.\, \lambda n.\, (n\ pred)\ m$$

Note that there might be cases in which *decode* will be stuck at a normal form. In which case we can simply terminate it as the program is wrong. In the rest of the paper, we will make the same assumption that the user program is correct.

$$
\begin{array}{lll}
t & ::= & terms: \\
& | & \ldots \\
& | & true \quad true \\
& | & false \quad false \\
& | & if\ t\ then\ t\ else\ t \quad if \\
& | & c \quad integer \\
& | & t + t \quad addition \\
& | & t - t \quad subtraction \\
& | & isZero\ t \quad isZero \\
& | & (t, t, \cdots, t) \quad tuple \\
& | & t.t \quad index \\
& | & let\ x = t\ in\ t \quad let
\end{array}
$$

**Figure 8.** Extension of Windowed CRDT Syntax

For convenience, we could also introduce Boolean and Conditionals in the form of church encoding.

**Definition 11.** ***(Boolean and Conditionals Encoding).*** *Define the following terms:*

$$true = \lambda a.\, \lambda b.\, a$$

$$false = \lambda a.\, \lambda b.\, b$$

$$isZero = \lambda n.\, n\, (\lambda x.\, false)\, true$$

$$if = \lambda p.\, \lambda a.\, \lambda b.\, p\ a\ b$$

We hereby assume that CRDT can be encoded to integers in binary form and ignore the details of the exact mechanism.

Similarly, tuples can also be encoded as

**Definition 12.** ***(Tuple Encoding).*** *Tuples can be encoded as a simplified church-encoded list.*

$$
\begin{aligned}
&encode\ (a_1, a_2, \cdots, a_{n-1}) = \\
&\qquad \lambda x_1.\, \lambda x_2.\, \cdots .\, \lambda x_{n-1}. \\
&\qquad if\ x_1\ then\ a_1, \\
&\qquad else\ if\ x_2\ then\ a_2 \\
&\qquad \cdots \\
&\qquad else\ if\ x_{n-1}\ then\ a_{n-1} \\
&\qquad else\ a_n
\end{aligned}
$$

*For tuple projection, we have*

$$t_1.t_2 = if\ isZero\ t_2\ then\ t_1\ true\ else\ t_1.(t_2 - 1)$$

Furthermore, we can also introduce a let-binding.

Thus, we can extend the syntax shown in Figure 3 by adding the syntax sugar shown in Figure 8.