A Formal Semantics for Skip's Reactive `reduce` Combinator

Cristiano Calcagno

November 2025

Abstract

We present a formal semantics for the `reduce` combinator in Skip [1], a reactive programming framework that maintains aggregated views of collections with automatic incremental updates. Skip lets developers define aggregation logic declaratively as pure functions over collections, while the runtime handles event routing, cache invalidation, and state synchronization. Skip exposes reducers as **first-class combinators**: a reducer $R = (\iota, \oplus, \ominus)$ is a reusable, composable value—not a callback tied to a specific engine context—that developers can apply uniformly across arbitrary reactive pipelines. Unlike streaming systems (Flink, Kafka Streams) that provide add/remove callbacks bound to specific operators, and unlike FRP and incremental computation frameworks that hide inverses internally, Skip surfaces this algebraic structure as a user-facing programming construct. We formalize Skip's correctness condition and prove that incremental correctness holds *if and only if* the reducer is well-formed (i.e., $\ominus$ is the inverse of $\oplus$), giving Skip users a precise, user-facing specification for writing correct custom reducers.

1 Introduction

Skip [1] is a reactive programming framework that maintains derived views of collections with automatic incremental updates. When the underlying data changes, Skip efficiently propagates updates to all dependent computations without manual intervention. A central operation in Skip is the `reduce` combinator, which computes a summary (or *view*) for each key in a collection by folding over its associated values—for example, computing the sum, count, or minimum.

The key to efficient updates is that Skip's `reduce` supports *retractions*: when values are removed from a collection, the reducer can incrementally update the accumulated result rather than recomputing from scratch. Skip exposes this capability through a *reducer* abstraction $R = (\iota, \oplus, \ominus)$, where ι is an initial value, $\oplus$ is an add operation, and $\ominus$ is a remove operation. This allows $O(1)$ updates per change, rather than $O(n)$ recomputation.

Declarative programming model. Skip enables developers to write aggregation logic declaratively, composing reducers as pure, reusable values that express *what* should be computed over collections. The Skip runtime maintains derived views and propagates incremental updates, so developers need not reason about event routing, cache invalidation, or state synchronization.

Skip's user-facing combinators. A distinguishing feature of Skip is that reducers are exposed as **user-facing combinators**—first-class programming constructs that developers use directly to build custom reactive services. Rather than being internal implementation details hidden from users, Skip allows developers to define their own reducers for domain-specific aggregations. Skip's documentation specifies an informal correctness condition: the result of applying the runtime's sequence of remove/add calls must equal recomputing from scratch. This design enables extensibility (custom aggregations), composability (reducers combine with other Skip combinators), and—when the condition is satisfied—correctness guarantees.

What distinguishes Skip's design. Three aspects distinguish Skip's reducer from related systems:

1. **First-class combinator, not callback.** Unlike Flink's `retract` methods or Kafka Streams' `Subtractor` interfaces—which are callbacks tied to specific engine contexts—Skip's reducer is a standalone, reusable value applicable to any reactive collection. This enables a declarative programming model where developers compose logic without infrastructure concerns; the reactive runtime handles event propagation and cache management transparently.
2. **User-facing inverse.** Unlike FRP libraries (which provide only append-style folds) and incremental computation frameworks (which handle inverses internally), Skip exposes $\ominus$ as an explicit user-defined operation.
3. **Formal correctness.** While prior work has studied invertible update operators and user-supplied inverses in more specialized settings, to our knowledge no mainstream streaming or reactive system provides a formal specification that application developers can apply when defining their own reducers; this paper fills that gap for Skip.

This paper. We formalize Skip's reducer abstraction and its correctness condition. Our main result (Theorem 2) proves that incremental correctness holds *if and only if* the reducer is well-formed—that is, $\ominus$ is the inverse of $\oplus$. This provides Skip users with a precise specification: satisfy the well-formedness condition, and your custom reducer is guaranteed to work correctly with Skip's incremental update mechanism.

Synthesis of ideas. Our formalization builds on Skip's design and synthesizes ideas from multiple domains, which anchor the rest of the paper:

- From **incremental databases and streaming systems** (distributive, algebraic, holistic aggregates [6]), we carry over the aggregate-class taxonomy and view maintenance perspective, and the insight that invertibility yields $O(1)$ per-change maintenance for suitable aggregates.
- From **Skip's reactive runtime**, we take the user-facing combinator contract: reducers are first-class, composable values whose invertibility the runtime can exploit, and which otherwise fall back safely to recompute.
- From **formal methods**, we provide a complete characterization: not merely sufficient conditions for correctness, but a precise *if-and-only-if* theorem that fully characterizes when Skip's incremental updates are correct.

Contributions. This paper formalizes Skip's `reduce` combinator. We provide:

- A denotational semantics for `reduce` as a derived view (Section 3)
- A formal model of deltas and Skip's incremental update procedure (Section 4)
- A precise well-formedness condition and proof that it is both necessary and sufficient for correctness (Section 5)
- A complexity contract: well-formed reducers admit $O(1)$ per-key updates, while partial reducers fall back to recomputation (Sections 4 and 6)
- Concrete examples including sum, count, and min reducers (Section 6)

We also position Skip's design relative to other streaming and reactive systems (Section 7) and analyze the complexity benefits of incremental updates.

2 Preliminaries

Let K be a set of keys, V a set of values, and A a set of accumulator values. For a set V, we write $\mathcal{M}(V)$ for the set of finite multisets over V; we use $\uplus$ and $\setminus$ for multiset union and multiset difference, respectively, and write $M \subseteq N$ for multisets when every element has multiplicity in M less than or equal to its multiplicity in N.

Definition 1 (Collection)**.** *A* collection *is a function $C : K \to \mathcal{M}(V)$. We write $C(k)$ for the multiset of values associated with key k.*

Example 1 (Purchases per user)**.** *As a running example, let* $K = \{\mathit{user1}, \mathit{user2}\}$ *and* $V = \mathbb{Z}$*. We model purchases per user and care about which purchase amounts occurred, not about the order in which they happened. A collection* $C : K \to \mathcal{M}(V)$ *might be given by*

$$C(\mathit{user1}) = \{30, 30, 50\} \quad \textit{and} \quad C(\mathit{user2}) = \{70\}.$$

Here $C(\mathit{user1})$ *is a multiset where* 30 *has multiplicity* 2 *and* 50 *has multiplicity* 1*, meaning that* user1 *made two purchases of \$30 and one purchase of \$50 (in some order). This illustrates that multiple purchases can have the same amount, and that multisets record these multiplicities while abstracting away from order.*

2.1 Commutative Operations

Definition 2 (Pairwise Commutative Operation)**.** *Let* $\star : A \times V \to A$ *be an update operation. We say that* $\star$ *is* pairwise commutative *if*

$$\forall a \in A, v_1, v_2 \in V.\ (a \star v_1) \star v_2 = (a \star v_2) \star v_1.$$

2.2 Folds

Definition 3 (Fold over Sequence for an Operation)**.** *Let* $\star : A \times V \to A$ *be an update operation and let* $s = [v_1, \ldots, v_n]$ *be a finite sequence of elements of* V*. For any* $a \in A$ *we define:*

$$\mathsf{fold}^{\mathsf{seq}}_{\star}(a, []) = a \quad \textit{and} \quad \mathsf{fold}^{\mathsf{seq}}_{\star}(a, v_1 :: s') = \mathsf{fold}^{\mathsf{seq}}_{\star}(a \star v_1, s').$$

When a distinguished initial element $\iota \in A$ *is understood from context, we write* $\mathsf{fold}^{\mathsf{seq}}_{\star}(s)$ *for* $\mathsf{fold}^{\mathsf{seq}}_{\star}(\iota, s)$*.*

Theorem 1 (Characterisation of Multiset Fold)**.** *Let* $\star : A \times V \to A$ *be an update operation. The following are equivalent:*

1. *For all* $a \in A$*,* $M \in \mathcal{M}(V)$ *and any two finite sequences* s_1, s_2 *enumerating* M *(with multiplicity), we have*

   $$\mathsf{fold}^{\mathsf{seq}}_{\star}(a, s_1) = \mathsf{fold}^{\mathsf{seq}}_{\star}(a, s_2).$$

   That is, folding depends only on the multiset of elements, not on their enumeration.

2. *The operation* $\star$ *is pairwise commutative in the sense of Definition 2.*

Proof. Sketch: For ($2 \Rightarrow 1$), one shows first that swapping two adjacent elements in a sequence does not change the fold, using pairwise commutativity of $\star$. Since any permutation of a finite sequence can be written as a product of adjacent transpositions, it follows that the fold depends only on the underlying multiset. For ($1 \Rightarrow 2$), instantiate (1) with the two sequences $[v_1, v_2]$ and $[v_2, v_1]$ enumerating the same multiset $\{v_1, v_2\}$, and expand the definition of the fold to obtain $(a \star v_1) \star v_2 = (a \star v_2) \star v_1$. □

Definition 4 (Fold over Multiset for an Operation). *Let $\star : A \times V \to A$ be pairwise commutative and let $M \in \mathcal{M}(V)$ be finite. For $a \in A$ and any sequence s enumerating M (with multiplicity), we set*

$$\mathsf{fold}_\star(a, M) := \mathsf{fold}^{\mathsf{seq}}_\star(a, s),$$

which is well-defined by the Characterisation of Multiset Fold. If an initial element $\iota \in A$ is fixed, we abbreviate $\mathsf{fold}_\star(M) := \mathsf{fold}_\star(\iota, M)$.

Example 2 (Total spent over a multiset). *Continuing the purchase example, let $A = V = \mathbb{Z}$ and define $\star(a, v) = a + v$ with initial element $\iota = 0$. For the multiset $M = \{30, 30, 50\}$ of purchases for* user1 *we have*

$$\mathsf{fold}_\star(0, M) = 0 + 30 + 30 + 50 = 110,$$

regardless of the order in which the elements of M are enumerated.

Lemma 1 (Fold over Union of Multisets). *Let $\star : A \times V \to A$ be pairwise commutative and let $M, N \in \mathcal{M}(V)$ be finite multisets. Then for all $a \in A$:*

$$\mathsf{fold}_\star(a, M \uplus N) = \mathsf{fold}_\star(\mathsf{fold}_\star(a, M), N).$$

Proof. Choose an enumeration of $M \uplus N$ in which all elements of M appear first, followed by all elements of N. The result then follows immediately from the definition of $\mathsf{fold}^{\mathsf{seq}}_\star$ and the fact that $\mathsf{fold}_\star$ is independent of the particular enumeration. $\square$

Example 3 (Decomposing total spend). *With $\star(a, v) = a + v$ and $\iota = 0$, suppose we split* user1*'s purchases as $M = \{30\}$ and $N = \{30, 50\}$ so that $M \uplus N = \{30, 30, 50\}$. Then*

$$\mathsf{fold}_\star(0, M \uplus N) = 0 + 30 + 30 + 50 = \mathsf{fold}_\star(\mathsf{fold}_\star(0, M), N),$$

showing that we can compute the total spent either directly on $M \uplus N$ or by first aggregating over M and then incrementally folding in the purchases in N.

3 The Reduce Combinator

The `reduce` combinator produces a *view* of a collection by summarizing the values for each key.

Definition 5 (Reduce Combinator). *Let $\oplus : A \times V \to A$ be a pairwise commutative operation and $\iota \in A$ an initial value. Given a collection $C : K \to \mathcal{M}(V)$, the* reduce *combinator produces a view:*

$$\mathsf{reduce}_{\iota,\oplus}(C) : K \to A$$

defined as:

$$\mathsf{reduce}_{\iota,\oplus}(C)(k) = \mathsf{fold}_\oplus(\iota, C(k))$$

That is, for each key k, we fold the operation $\oplus$ over all values in $C(k)$, starting from ι.

Example 4 (Per-user analytics as reduce)**.** *For the purchase collection C above, two common choices of $\oplus$ are:*

- *the* sum *operation $\oplus_{\mathsf{sum}}(a, v) = a + v$ with $\iota_{\mathsf{sum}} = 0$, giving each user their total spend;*
- *the* count *operation $\oplus_{\mathsf{count}}(a, v) = a + 1$ with $\iota_{\mathsf{count}} = 0$, giving each user their number of purchases.*

Then, for example,

$$\mathsf{reduce}_{\iota_{\mathsf{sum}},\oplus_{\mathsf{sum}}}(C)(\mathit{user1}) = 110, \qquad \mathsf{reduce}_{\iota_{\mathsf{count}},\oplus_{\mathsf{count}}}(C)(\mathit{user1}) = 3.$$

The view $\mathsf{reduce}_{\iota,\oplus}(C)$ is a derived collection that depends on C. When C changes, the view must be updated to remain consistent. The next section addresses how to perform these updates efficiently.

4 Incremental Updates

When a collection C changes, the view $\mathsf{reduce}_{\iota,\oplus}(C)$ must be updated. A naïve approach would recompute the fold from scratch for each affected key, requiring $O(n)$ time where n is the size of the multiset. To achieve $O(1)$ updates, we introduce a *remove* operation $\ominus$ that can undo the effect of $\oplus$.

4.1 Reducers

Definition 6 (Reducer)**.** *A* reducer *is a triple $R = (\iota, \oplus, \ominus)$ where $\iota \in A$ is an initial value, and*

$$\oplus, \ominus : A \times V \to A$$

are update operations such that both $\oplus$ and $\ominus$ are pairwise commutative in the sense of Definition 2. We call $\oplus$ the add *operation and $\ominus$ the* remove *operation.*

For a reducer $R = (\iota, \oplus, \ominus)$, we write reduce_R for $\mathsf{reduce}_{\iota,\oplus}$.

Definition 7 (Well-Formed Reducer)**.** *A reducer $R = (\iota, \oplus, \ominus)$ is* well-formed *if $\ominus$ is the* ***inverse*** *of $\oplus$ on reachable accumulator values, that is, for all finite multisets $M \in \mathcal{M}(V)$ and all $v \in V$:*

$$(\mathsf{fold}_{\oplus}(\iota, M) \oplus v) \ominus v = \mathsf{fold}_{\oplus}(\iota, M)$$

Remark 1 (Asymmetric inverse law)**.** *Well-formedness insists on "remove after add" because this is the direction exercised by incremental updates: deltas are applied by first folding in additions and only then subtracting deletions. The reverse law $(a \ominus v) \oplus v = a$ is not guaranteed without extra hypotheses (e.g. cancellativity); requiring only the asymmetric law keeps the definition broad enough to cover partial monoids and multiset subtraction.*

Example 5 (Interpreting well-formedness for sum). *For R_{sum} and any multiset M of purchase amounts and value $v \in \mathbb{Z}$, well-formedness says that if we start from the total spend $\mathsf{fold}_{\oplus}(0, M)$, then adding a purchase of amount v and immediately removing it leaves the total unchanged:*

$$\big(\mathsf{fold}_{\oplus}(0, M) + v\big) - v = \mathsf{fold}_{\oplus}(0, M).$$

This is exactly the intuitive requirement that $\ominus$ undo the effect of $\oplus$ on reachable accumulator states.

In database terminology, a well-formed reducer defines an *invertible distributive aggregate* (see Section 4.2): the fold can be computed over partitions independently (distributive), and individual values can be removed from the accumulated result (invertible).

Remark 2 (Remove-Add Commutativity). *For well-formed reducers where $\oplus$ and $\ominus$ arise from an abelian group action on A, the following property holds automatically:*

$$\forall a \in A, v_1, v_2 \in V.\ (a \ominus v_1) \oplus v_2 = (a \oplus v_2) \ominus v_1$$

This ensures that the order of interleaved adds and removes does not affect the final result. All practical reducers (sum, count, product over commutative groups) satisfy this.

4.2 Aggregate Classes

The database literature [6] classifies aggregates as *distributive*, *algebraic*, or *holistic*. In our setting, a pair $(\iota, \oplus)$ defines a *distributive aggregate* when folding over a union of multisets can be decomposed into folds over the parts.

Definition 8 (Distributive Aggregate). *Let $\oplus : A \times V \to A$ be pairwise commutative and $\iota \in A$. We say that $(\iota, \oplus)$ is a* distributive aggregate *if for all finite multisets $M, N \in \mathcal{M}(V)$:*

$$\mathsf{fold}_{\oplus}(\iota, M \uplus N) = \mathsf{fold}_{\oplus}\big(\mathsf{fold}_{\oplus}(\iota, M), N\big).$$

By Lemma 1, any pair $(\iota, \oplus)$ with pairwise commutative $\oplus$ is a distributive aggregate in this sense. Moreover, a well-formed reducer $R = (\iota, \oplus, \ominus)$ (Definition 7) is precisely an *invertible distributive aggregate*: the aggregate is distributive over partitions of the multiset, and individual contributions can be removed using $\ominus$.

The database literature also defines *algebraic* aggregates [6]: aggregates that can be computed by maintaining a fixed number of distributive aggregates and post-processing their results. For example, average is algebraic because it can be computed from the distributive aggregates sum and count via division. In Skip, this corresponds to using a well-formed reducer with richer accumulator state (e.g., $(\mathit{sum}, \mathit{count})$ pairs) followed by a pointwise mapper to extract the final value. We illustrate this pattern in Section 6.

Finally, *holistic* aggregates [6] cannot be computed from bounded intermediate state—they potentially require access to the entire multiset. Examples include:

- **MEDIAN**: needs the full distribution to pick the middle value(s)
- **QUANTILES/PERCENTILES**: similar to median, require global ordering information
- **RANK**: depends on the position of a value within the full sorted dataset

For holistic aggregates, any exact incremental solution must maintain auxiliary state that grows with the data (e.g., the entire multiset or an order-statistic tree) in order to answer updates and queries. Skip can of course support such analyses by using richer data structures or approximations (e.g., quantile sketches), but these fall outside the constant-space, purely algebraic reducer model we formalize in this paper.

4.3 Deltas

We model updates to collections as deltas.

Definition 9 (Delta). *A* delta Δ *for a collection C is a pair* (Δ^+, Δ^-) *where:*

- $\Delta^+ : K \to \mathcal{M}(V)$ *represents added values*
- $\Delta^- : K \to \mathcal{M}(V)$ *represents removed values*

We require that $\Delta^-(k) \subseteq C(k)$ for all k (i.e., we can only remove values that exist in the collection).

Definition 10 (Delta Application). *Given a collection C and a delta $\Delta = (\Delta^+, \Delta^-)$ for C, the updated collection $C \bullet \Delta$ is defined pointwise by:*

$$C \bullet \Delta = \lambda k.\, (C(k) \setminus \Delta^-(k)) \uplus \Delta^+(k)$$

Intuitively, $C \bullet \Delta$ is the collection obtained by first removing all values in Δ^- from C and then adding all values in Δ^+.

Remark 3 (Why remove then add?). *The order here mirrors how the incremental update function works: we take the* existing *accumulation, subtract contributions that are no longer present, and then fold in new contributions. This ensures that deletions target only previously present elements; additions are layered on top of that cleaned state. The following example demonstrates why this order matters when Δ^+ and Δ^- overlap.*

Example 6 (Order matters when Δ^+ and Δ^- overlap). *Let $C(k) = \{a\}$, and suppose $\Delta^-(k) = \{a\}$ (remove the old a) and $\Delta^+(k) = \{a, b\}$ (add back a and a new b). The intended result is $C'(k) = \{a, b\}$: we remove the old a and then add the new multiset. With the remove-then-add definition, we get*

$$(C(k) \setminus \{a\}) \uplus \{a, b\} = \varnothing \uplus \{a, b\} = \{a, b\}.$$

While with multisets both orders may yield the same result, the remove-then-add order is essential when working with sets (where union is $\cup$ and difference is $\setminus$). If we used add-then-remove with sets, we would compute

$$(C(k) \cup \{a, b\}) \setminus \{a\} = \{a, b\} \setminus \{a\} = \{b\},$$

which incorrectly removes the newly added a from $\Delta^+(k)$, giving $\{b\}$ instead of the intended $\{a, b\}$. The remove-then-add order ensures that deletions target only elements that existed in $C(k)$, and additions are applied to the cleaned state.

Example 7 (Changing a user's purchases)**.** *For key $k = user1$ in the purchase collection C, suppose we remove one \$30 purchase and add a new \$20 purchase. This yields a delta with*

$$\Delta^-(k) = \{30\}, \qquad \Delta^+(k) = \{20\}$$

and updated multiset

$$C'(k) = (C(k) \setminus \{30\}) \uplus \{20\} = \{30, 50, 20\},$$

which again records all purchases for user1 *but with one order-irrelevant change in their amounts.*

Remark 4 (Operational Construction of Deltas)**.** *In the Skip runtime, for each key k we compute an* old *multiset $old(k)$ of contributing values and a* new *multiset $new(k)$. The delta is then constructed as:*

$$\Delta^+(k) = new(k) \setminus old(k) \quad and \quad \Delta^-(k) = old(k) \setminus new(k).$$

Note that Δ^+ and Δ^- are disjoint by construction. If C_{old} is the collection with $C_{old}(k) = old(k)$, then $\Delta^-(k) \subseteq C_{old}(k)$, so Δ is a valid delta for C_{old}. Moreover, $C_{old} \bullet \Delta = C_{new}$ where $C_{new}(k) = new(k)$.

4.4 Incremental Update

Definition 11 (Incremental Reduce)**.** *Given the current accumulator value a_k for key k and a delta Δ, the new accumulator is computed as:*

$$\mathsf{update}_R(a_k, \Delta, k) = \mathsf{fold}_\oplus\big(\mathsf{fold}_\ominus(a_k, \Delta^-(k)),\ \Delta^+(k)\big)$$

That is, we first apply all removals $\Delta^-(k)$ to a_k using $\ominus$, and then apply all additions $\Delta^+(k)$ using $\oplus$. This is well-defined since $\oplus$ and $\ominus$ are pairwise commutative. Here $\mathsf{fold}_\ominus$ is the multiset fold induced by $\ominus$, defined exactly as in Section 2; pairwise commutativity of $\ominus$ guarantees it is well-defined.

Example 8 (Updating a user's total spend)**.** *Let R_{sum} be the sum reducer and consider again $k = user1$ with*

$$C(k) = \{30, 30, 50\}, \qquad a_k = \mathsf{reduce}_{R_{\mathsf{sum}}}(C)(k) = 110.$$

Applying the delta above (removing \$30, adding \$20) gives

$$\mathsf{update}_{R_{\mathsf{sum}}}(a_k, \Delta, k) = \mathsf{fold}_\oplus(\mathsf{fold}_\ominus(110, \{30\}), \{20\}) = (110 - 30) + 20 = 100,$$

which matches the total obtained by recomputing from scratch on the updated multiset $C'(k) = \{30, 50, 20\}$.

5 Correctness

We now characterize exactly when incremental updates are correct. At some moment we have an *old* collection C and, for each key k, an accumulator

$$a_k = \mathsf{reduce}_R(C)(k)$$

that agrees with the denotational semantics. A change to the collection is described abstractly by a delta Δ, yielding the *updated* collection

$$C' = C \bullet \Delta.$$

For each key k there are then two ways to obtain the *new* accumulator value:

- *Denotational recompute:* ignore a_k and compute

  $$a'_k = \mathsf{reduce}_R(C')(k),$$

  i.e. start from ι and fold $\oplus$ over the current multiset $C'(k)$.

- *Incremental update:* update the old accumulator a_k using the delta by

  $$a'_k = \mathsf{update}_R(a_k, \Delta, k),$$

  i.e. first remove $\Delta^-(k)$ using $\ominus$, then add $\Delta^+(k)$ using $\oplus$.

Definition 12 (Incremental Correctness Property)**.** *A reducer R satisfies the* incremental correctness property *if for all collections C, all valid deltas Δ for C, and all keys k:*

$$\mathsf{reduce}_R(C \bullet \Delta)(k) = \mathsf{update}_R(\mathsf{reduce}_R(C)(k), \Delta, k)$$

Example 9 (Sum as a sanity check)**.** *For the sum reducer R_{sum}, the incremental correctness property simply formalizes the familiar claim that "subtracting the amounts of removed purchases and adding the amounts of new purchases yields the same total spend as recomputing the sum from scratch" for every user and every delta.*

Lemma 2 (Cancellation for a Delta)**.** *Let $R = (\iota, \oplus, \ominus)$ be a well-formed reducer and C a collection. For any valid delta $\Delta = (\Delta^+, \Delta^-)$ for C and key k, writing $M = C(k)$ and $M_0 = M \setminus \Delta^-(k)$, we have:*

$$\mathsf{fold}_\ominus\big(\mathsf{fold}_\oplus(\iota, M), \Delta^-(k)\big) = \mathsf{fold}_\oplus(\iota, M_0).$$

Proof. Write $D = \Delta^-(k)$. We proceed by induction on the size of the multiset D. If $D = \emptyset$, then $\mathsf{fold}_\ominus(\mathsf{fold}_\oplus(\iota, M), D) = \mathsf{fold}_\oplus(\iota, M)$ by definition, and $M_0 = M$, so the claim holds.

For the inductive step, suppose $D = D' \uplus \{v\}$ for some value v and sub-multiset $D' \subseteq M$, and assume the statement holds for D'. Since $M = M_0 \uplus D =$

$M_0 \uplus D' \uplus \{v\}$ and $\oplus$ is pairwise commutative, Lemma 1 (fold over union of multisets) gives:

$$\mathsf{fold}_{\oplus}(\iota, M) = \mathsf{fold}_{\oplus}\big(\mathsf{fold}_{\oplus}(\iota, M_0 \uplus D'), \{v\}\big) = \mathsf{fold}_{\oplus}(\iota, M') \oplus v$$

for the multiset $M' = M_0 \uplus D'$. Applying well-formedness with multiset M' and value v yields

$$(\mathsf{fold}_{\oplus}(\iota, M') \oplus v) \ominus v = \mathsf{fold}_{\oplus}(\iota, M').$$

Since $\mathsf{fold}_{\oplus}(\iota, M) = \mathsf{fold}_{\oplus}(\iota, M') \oplus v$, we have:

$$\mathsf{fold}_{\ominus}(\mathsf{fold}_{\oplus}(\iota, M), \{v\}) = (\mathsf{fold}_{\oplus}(\iota, M') \oplus v) \ominus v = \mathsf{fold}_{\oplus}(\iota, M').$$

Now $M' \setminus D' = (M_0 \uplus D') \setminus D' = M_0$, so by the induction hypothesis applied to M' and D' we obtain

$$\mathsf{fold}_{\ominus}\big(\mathsf{fold}_{\oplus}(\iota, M'), D'\big) = \mathsf{fold}_{\oplus}(\iota, M' \setminus D') = \mathsf{fold}_{\oplus}(\iota, M_0).$$

Since $\ominus$ is pairwise commutative, Lemma 1 applies to $\ominus$ as well, giving:

$$\mathsf{fold}_{\ominus}(\mathsf{fold}_{\oplus}(\iota, M), D) = \mathsf{fold}_{\ominus}(\mathsf{fold}_{\ominus}(\mathsf{fold}_{\oplus}(\iota, M), \{v\}), D'),$$

and using the equalities above,

$$\mathsf{fold}_{\ominus}(\mathsf{fold}_{\ominus}(\mathsf{fold}_{\oplus}(\iota, M), \{v\}), D') = \mathsf{fold}_{\ominus}(\mathsf{fold}_{\oplus}(\iota, M'), D'),$$

and finally

$$\mathsf{fold}_{\ominus}(\mathsf{fold}_{\oplus}(\iota, M'), D') = \mathsf{fold}_{\oplus}(\iota, M_0),$$

which completes the proof. □

Example 10 (Delta cancellation for total spend)**.** *Continuing the purchase example, let R_{sum} be the sum reducer and consider key $k = user1$ with*

$$C(k) = \{30, 30, 50\}.$$

Suppose we remove one \$30 purchase and keep the other two purchases unchanged, so

$$\Delta^-(k) = \{30\}, \qquad \Delta^+(k) = \emptyset,$$

and hence

$$M = C(k), \quad D = \Delta^-(k), \quad M_0 = M \setminus D = \{30, 50\}.$$

Lemma 2 states that

$$\mathsf{fold}_{\ominus}\big(\mathsf{fold}_{\oplus}(0, M), D\big) = \mathsf{fold}_{\oplus}(0, M_0).$$

Indeed, $\mathsf{fold}_{\oplus}(0, M) = 0 + 30 + 30 + 50 = 110$*, so*

$$\mathsf{fold}_{\ominus}(110, \{30\}) = 110 - 30 = 80,$$

while

$$\mathsf{fold}_{\oplus}(0, M_0) = 0 + 30 + 50 = 80.$$

Thus incrementally removing the multiset of cancelled purchases yields exactly the total spend over the remaining purchases.

The following theorem shows that the inverse property is both necessary and sufficient for incremental correctness.

Theorem 2 (Characterization of Incremental Correctness)**.** *Let $R = (\iota, \oplus, \ominus)$ be a reducer (with pairwise commutative $\oplus$ and $\ominus$). The following are equivalent:*

1. *R is well-formed in the sense of Definition 7.*
2. *R satisfies the incremental correctness property.*

Proof. We prove both directions.

(1 $\Rightarrow$ 2): Well-formedness implies correctness.
Assume R is well-formed. Let C be a collection, $\Delta = (\Delta^+, \Delta^-)$ a valid delta for C, and k a key. Write $M = C(k)$ for the old multiset, $M' = C'(k) = (M \setminus \Delta^-(k)) \uplus \Delta^+(k)$ for the new multiset, and $a = \mathsf{fold}_\oplus(\iota, M)$ for the old accumulator.

We must show $\mathsf{fold}_\oplus(\iota, M') = \mathsf{fold}_\oplus(\mathsf{fold}_\ominus(a, \Delta^-(k)), \Delta^+(k))$.

Since $\Delta^-(k) \subseteq M$, we can write $M = M_0 \uplus \Delta^-(k)$ for some multiset M_0. By pairwise commutativity of $\oplus$:

$$a = \mathsf{fold}_\oplus(\iota, M) = \mathsf{fold}_\oplus(\iota, M_0 \uplus \Delta^-(k)) = \mathsf{fold}_\oplus(\mathsf{fold}_\oplus(\iota, M_0), \Delta^-(k))$$

Let $a_0 = \mathsf{fold}_\oplus(\iota, M_0)$. Then $a = \mathsf{fold}_\oplus(a_0, \Delta^-(k))$.

By Lemma 2 we have

$$\mathsf{fold}_\ominus(a, \Delta^-(k)) = \mathsf{fold}_\oplus(\iota, M_0) = a_0.$$

Therefore:

$$\mathsf{fold}_\oplus(\mathsf{fold}_\ominus(a, \Delta^-(k)), \Delta^+(k)) = \mathsf{fold}_\oplus(a_0, \Delta^+(k))$$

Since $M' = M_0 \uplus \Delta^+(k)$:

$$\mathsf{fold}_\oplus(\iota, M') = \mathsf{fold}_\oplus(\mathsf{fold}_\oplus(\iota, M_0), \Delta^+(k)) = \mathsf{fold}_\oplus(a_0, \Delta^+(k))$$

Thus both sides are equal.

(2 $\Rightarrow$ 1): Correctness implies well-formedness.
Assume R satisfies the incremental correctness property. We must show $(a \oplus v) \ominus v = a$ for all $a \in A$ and $v \in V$.

Fix $a \in A$ and $v \in V$. We first establish the property for a of the form $a = \mathsf{fold}_\oplus(\iota, M)$ for some multiset M.

Define a collection C with a single key k where $C(k) = M \uplus \{v\}$. Then:

$$\mathsf{reduce}_R(C)(k) = \mathsf{fold}_\oplus(\iota, M \uplus \{v\}) = \mathsf{fold}_\oplus(\mathsf{fold}_\oplus(\iota, M), \{v\}) = a \oplus v$$

Define delta Δ with $\Delta^-(k) = \{v\}$ and $\Delta^+(k) = \emptyset$. This is valid since $v \in C(k)$. The updated collection is C' with $C'(k) = M$.

By the incremental correctness property:

$$\mathsf{reduce}_R(C')(k) = \mathsf{update}_R(\mathsf{reduce}_R(C)(k), \Delta, k)$$

The left side is:

$$\mathsf{reduce}_R(C')(k) = \mathsf{fold}_\oplus(\iota, M) = a$$

The right side is:

$$\mathsf{update}_R(a \oplus v, \Delta, k) = \mathsf{fold}_\oplus(\mathsf{fold}_\ominus(a \oplus v, \{v\}), \emptyset) = (a \oplus v) \ominus v$$

Therefore $(a \oplus v) \ominus v = a$.

This establishes the inverse property for all a of the form $\mathsf{fold}_\oplus(\iota, M)$, i.e. for all reachable accumulator values, which is exactly the condition in Definition 7.
□

Remark 5. *In many practical reducers (for example sum over integers or product over rationals), every accumulator value is reachable as* $\mathsf{fold}_\oplus(\iota, M)$ *for some multiset* M*, so the definition of well-formedness above coincides with the simpler global inverse law* $(a \oplus v) \ominus v = a$ *for all* $a \in A$*,* $v \in V$*.*

Remark 6 (Partial Reducers in Skip)**.** *Theorem 2 characterizes when incremental updates are correct for well-formed reducers. In Skip's concrete API, the remove operation is allowed to be* partial*: a reducer's remove function can signal "recompute from scratch" (by returning* `None` *in the ReScript bindings) instead of producing an updated accumulator. Such partial reducers handle cases where* $\ominus$ *cannot efficiently invert* $\oplus$*—for example, computing the minimum without maintaining auxiliary state. The runtime responds by recomputing the fold from scratch for that key. Examples of partial reducers appear in Section 6.*

6 Examples

6.1 Sum Reducer

$$R_{\mathsf{sum}} = (0, \lambda(a, v).\, a + v, \lambda(a, v).\, a - v)$$

This reducer is well-formed: addition is commutative, and subtraction is the inverse of addition.

Worked example. Suppose for key k we have $C(k) = \{3, 5, 7\}$, so the current view is $a_k = 0 + 3 + 5 + 7 = 15$. Now suppose value 5 is removed and value 2 is added, giving delta $\Delta^-(k) = \{5\}$ and $\Delta^+(k) = \{2\}$. The incremental update computes:

$$a'_k = (15 - 5) + 2 = 12$$

This matches a full recompute: $0 + 3 + 7 + 2 = 12$.

6.2 Count Reducer

$$R_{\mathsf{count}} = (0, \lambda(a, v).\, a + 1, \lambda(a, v).\, a - 1)$$

This reducer counts the number of values, ignoring their content. It is well-formed since $(a + 1) - 1 = a$.

Worked example. For $C(k) = \{x, y, z\}$, we have $a_k = 3$. If y is removed ($\Delta^- = \{y\}$) and w is added ($\Delta^+ = \{w\}$), then:

$$a'_k = (3 - 1) + 1 = 3$$

6.3 Average Reducer (Algebraic)

The average is a classic example of an *algebraic* aggregate [6]: it can be expressed as a post-processing of a distributive aggregate over richer state. A standard encoding uses accumulator state $A = \mathbb{R} \times \mathbb{N}$ to track sum and count.

Define:

$$R_{\mathsf{avgState}} = \big((0, 0),\ \lambda((s, c), v).\, (s + v, c + 1),\ \lambda((s, c), v).\, (s - v, c - 1)\big)$$

with accumulator state $A = \mathbb{R} \times \mathbb{N}$ tracking (sum, count). On reachable states with $c > 0$, the corresponding average is $\mathtt{avg} = s/c$; when $c = 0$ (empty multiset), the view can be defined as a designated "no value" (e.g., `None`) or 0 depending on the application. This reducer is well-formed: addition and subtraction on sum and increment/decrement on count satisfy the inverse law on all reachable accumulator states, so Theorem 2 applies. In Skip, the average view can be implemented by first using $\mathsf{reduce}_{R_{\mathsf{avgState}}}$ and then applying a pointwise mapper that divides sum by count for each key.

Note that maintaining *only* the average (without count) is insufficient: to update the average when adding a value, one needs to know how many values contributed to the current average. Thus, average is genuinely an algebraic aggregate requiring auxiliary state, unlike sum or count which can be maintained with a single accumulator value.

6.4 Min Reducer (Partial)

The min reducer demonstrates why invertibility is essential for incremental updates. With accumulator $A = \mathbb{R} \cup \{+\infty\}$, the add operation is:

$$\iota = +\infty, \quad \oplus = \lambda(a, v).\, \min(a, v)$$

However, there is no inverse operation $\ominus$ that works in general. Consider $C(k) = \{3, 5\}$ with $a_k = 3$.

- If we remove 5: we need $a'_k = 3$. We could define $(3 \ominus 5) = 3$ (removing a non-minimum has no effect).

- If we remove 3: we need $a'_k = 5$. But from $a_k = 3$ alone, we cannot know that 5 was the second-smallest value!

This shows min is *not* an invertible distributive aggregate: knowing only the accumulated minimum is insufficient to update the result when the minimum itself is removed.

In Skip's implementation, min is handled as a *partial reducer*:

- The remove function signals "cannot update incrementally" (e.g., returns `None`)
- The runtime responds by recomputing from scratch: $\mathsf{fold}_{\min}(\iota, C'(k))$
- Alternatively, one can maintain richer state (e.g., a sorted multiset of all values), making the remove operation invertible on that richer state—but this is no longer a constant-space reducer

7 Related Work

The problem of efficiently maintaining aggregations over changing data has been studied extensively across streaming systems, reactive programming frameworks, and incremental computation. We organize our discussion around a key design dimension: *what operational concerns must developers understand and manage?* Skip's design philosophy—infrastructure abstraction paired with exposed algebraic structure—positions it within a broader movement toward declarative programming models where developers express domain logic without infrastructure concerns [12].

Infrastructure-exposed streaming systems. Traditional streaming systems require developers to understand and work with operational concerns: how data flows through pipelines, when windows trigger, how state is partitioned, and how retractions are processed. These systems expose infrastructure complexity, forcing developers to think about event propagation, state management, and update mechanisms.

Apache Flink (Table API) supports user-defined aggregate functions via classes with methods such as `createAccumulator`, `accumulate`, and (for streaming queries that receive updates) `retract`. These objects are reusable within Flink pipelines, but they are tied to Flink's table runtime and changelog/retraction model; the interface does not present a standalone reducer triple $R = (\iota, \oplus, \ominus)$ with an explicit invertibility law. *Apache Kafka Streams* provides aggregation over KTables by passing two callbacks to `aggregate`: one to add new values and one to remove old values when keys are updated or deleted. These callbacks are defined as part of a particular topology and bound to KTable's update mechanism, rather than as reusable reducer values with a user-facing correctness contract. *Esper CEP* supports `enter`/`leave` callbacks and delivers "new" and "old" results for sliding windows, but the engine itself implements aggregation and retractions; developers react to removals in listeners rather than defining a separate inverse operation. Systems like Microsoft's *Trill* [10] similarly let users define custom aggregations with explicit accumulate and de-accumulate functions, but these remain tied to the streaming query's execution and lack any user-facing formal correctness conditions. In all of these systems, add/remove logic is supplied as configuration for specific operators within a streaming engine, not as a general-purpose, portable combinator abstraction. Moreover, these systems expose their event-driven, retraction-based architectures; developers must understand how retractions propagate, when state is synchronized, and how correctness is ensured.

Apache Beam and *Spark Streaming* take a different approach: they hide user-defined inverse operations entirely, handling retractions internally via recomputation or diffing. While this simplifies the developer interface, it removes the ability to express domain-specific inverse operations, and developers must still reason about operational concerns like windowing, triggering, and state management.

Infrastructure-hidden, structure-hidden systems. At the other extreme, some systems hide both infrastructure complexity and the underlying algebraic structure, working at higher levels of abstraction.

Differential dataflow [2] and *DBSP* [3] provide rigorous foundations for incremental computation using Z-sets and abelian groups. These frameworks internally ensure that all built-in aggregations have inverses, but users typically work at the level of SQL queries or dataflow graphs without explicitly defining $\oplus$ and $\ominus$. *Materialize* [11] is a streaming SQL database built on differential dataflow that exposes incremental view maintenance via SQL, while hiding the underlying use of differential updates. The algebraic structure exists but is hidden; users cannot define custom reducers that exploit invertibility for domain-specific aggregations.

Functional reactive programming libraries (Fran, Yampa, Reactive Banana [8]) and UI-oriented frameworks (React, Vue, Svelte, Solid) encapsulate event routing and dependency tracking, exposing declarative combinators like map, filter, and scan/foldp. State evolution is typically described as folds over event streams (e.g., `foldp`, `scan`, `accum`), but these folds are opaque to the runtime: there is no distinguished remove operation or notion of an invertible reducer. As a result, the frameworks can optimize propagation and caching, but not user-specified algebraic inverses for efficient handling of removals.

Incremental computation frameworks (Adapton, Jane Street's Incremental) hide infrastructure complexity via dependency tracking, but also hide the inverse operation; removal is handled internally through dependency graphs rather than explicit algebraic inverses.

Skip's approach: infrastructure-hidden, structure-exposed. Skip's design occupies a distinctive position among these systems: it hides infrastructure complexity while exposing algebraic structure directly to application developers. Developers write aggregation logic declaratively as pure functions over collections, without thinking about event routing, cache invalidation, or state synchronization; the reactive runtime handles these operational concerns automatically. At the same time, Skip exposes the algebraic structure of reducers as first-class combinators: developers define $\oplus$ and $\ominus$ explicitly, enabling domain-specific aggregations with formal correctness guarantees. Several prior systems require user-supplied inverse functions or define formal minus/change operators in their semantic models, but Skip combines a user-facing reducer triple, reuse across multiple reactive contexts, and a formal correctness theorem that connects this triple to the runtime's operational behavior.

This design aligns with a broader trend toward "event-hidden architectures" seen in modern distributed systems [12]. Just as SQL databases hide query execution plans while exposing relational algebra, and just as React hides DOM manipulation while exposing declarative components, Skip hides event queues and reactive runtime internals while exposing mathematical combinators. The runtime remains event-driven under the covers, but developers work at the level of what to compute, not how events propagate.

Skip's reducer $R = (\iota, \oplus, \ominus)$ is a first-class combinator—a reusable, composable value applicable to any reactive collection via reduce_R, not a callback tied to specific engine contexts. This enables developers to define custom aggregations (e.g., domain-specific metrics, business logic) that compose naturally with other Skip combinators, while the runtime automatically handles incremental updates, dependency tracking, and cache management.

Database view maintenance and aggregate classification. The database literature classifies aggregates as *distributive*, *algebraic*, or *holistic* [6]. Skip's well-formed reducers correspond to *invertible distributive aggregates*: the fold can be computed over partitions independently (distributive), and individual contributions can be removed using the inverse operation (invertible). In the database literature, such aggregates are often described as *self-maintainable* or invertible distributive aggregates [6]. Tangwongsan et al. [7] note that "prior work often relies on aggregation functions to be invertible" for efficient sliding-window maintenance. Yin et al. [9] require inverse functions for incremental graph aggregation. These works focus on algorithmic techniques; our contribution is to formalize correctness for Skip's user-facing abstraction, providing developers with a precise specification for writing correct custom reducers.

Our formal contribution. Skip's documentation specifies an informal correctness condition for user-defined reducers: the result of applying the runtime's sequence of remove/add calls must equal recomputing from scratch. Our contribution is to:

1. Formalize this correctness condition as a well-formedness property
2. Prove that correctness holds *if and only if* the property is satisfied (Theorem 2)

3. Connect Skip's design to the theory of invertible distributive aggregates

This gives Skip users a precise specification for writing correct custom reducers, backed by a complete formal characterization. The well-formedness condition is a clear, local specification: developers define $\oplus$ and $\ominus$ explicitly and reason about whether $(a \oplus v) \ominus v = a$ holds for all reachable accumulator states a, rather than about the runtime's operational behavior.

References

[1] Skip Team. *Skip: A Reactive Programming Framework*. `https://github.com/SkipLabs/skip`, 2024.

[2] Frank McSherry, Derek G. Murray, Rebecca Isaacs, and Michael Isard. Differential dataflow. In *Proceedings of CIDR*, 2013.

[3] Mihai Budiu, Tej Chajed, Frank McSherry, Leonid Ryzhyk, and Val Tannen. DBSP: Automatic incremental view maintenance. In *Proceedings of VLDB*, 16(7):1601–1614, 2023.

[4] Erik Meijer. Your mouse is a database. *Communications of the ACM*, 55(5):66–73, 2012.

[5] Umut A. Acar, Guy E. Blelloch, and Robert Harper. Adaptive functional programming. In *Proceedings of POPL*, pages 247–259, 2002.

[6] Ashish Gupta and Inderpal Singh Mumick. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Engineering Bulletin*, 18(2):3–18, 1995.

[7] Kanat Tangwongsan, Martin Hirzel, Scott Schneider, and Kun-Lung Wu. General incremental sliding-window aggregation. *Proceedings of the VLDB Endowment*, 8(7):702–713, 2015.

[8] Conal Elliott and Paul Hudak. Functional reactive animation. In *Proceedings of ICFP*, pages 263–273, 1997.

[9] Shufeng Yin, Huanchen Zhang, Zhengyi Yang, Wentao Han, Wenguang Chen, and Yingxia Shao. GraphBolt: Dependency-driven synchronous processing of streaming graphs. In *Proceedings of ICDE*, 2022.

[10] Badrish Chandramouli, Jonathan Goldstein, Ryan Hill, Mirek Interlandi, Vladimir V. Lychagin, Morgan May, and Ravishankar Ramamurthy. Trill: A high-performance incremental query processor for diverse analytics. *Proceedings of the VLDB Endowment*, 8(4):401–412, 2014.

[11] Materialize, Inc. Materialize: Streaming SQL database. `https://materialize.com`, 2025.

[12] Charles Zedlewski. Event-hidden architectures. SkipLabs Blog, `https://skiplabs.io/blog/event-hidden-arch`, April 2025.