# Local Reactive Combinators and Relational Algebra with Aggregates

**Abstract**

The Skip runtime provides reactive bindings with structural operators on key-value collections: entry-wise maps, key-range slices, prefixes, finite merges, and per-key reducers.

We ask: *what is missing from Skip to match the expressivity of relational algebra (RA) with aggregates?* A catalogue of 48 representative examples—including DBToaster-style incremental views, per-group aggregates, and multi-way joins—reveals that joins are pervasive. Skip supports them via a *map-with-lookup* idiom: computing "active members per group" or "revenue by region" works by looking up related entries during a map operation. However, *anti-joins* ("entries with no matching partner") are absent from all 48 examples—not because they are rare, but because Skip cannot express them. Useful reactive patterns like orphan detection (orders with no customer), unacknowledged-alert dashboards, or unassigned-ticket queues all require filtering based on absence of keys, which Skip lacks.

We identify an anti-join combinator, filterNotMatchingOn, as the missing operator, describe how to implement it, and prove that adding it achieves *expressive equivalence* with RA: the extended combinator algebra can express exactly the RA queries (selection, projection, union, difference, product, join, grouping), and vice versa. The equivalence is constructive—we give explicit compilation functions in both directions.

This characterization provides a precise expressivity benchmark for Skip's reactive system, and connects it to classical locality results from finite model theory (Gaifman's theorem), ensuring bounded update propagation. A mechanized Lean 4 proof covers the core equivalence.

## 1 Introduction

This note asks a practical question: *what is missing from the Skip reactive runtime to match the expressivity of relational algebra?*

Skip provides reactive bindings with structural operators on key-value collections: entry-wise maps, key-range slices, prefixes, finite merges, and per-key reducers. These operators are "always safe": they do not maintain hidden state, are insensitive to update order, and admit straightforward incremental implementations.

Examining 48 representative reactive service examples, we find that *joins* are pervasive. Skip supports all of these via a map-with-lookup idiom, where the map function receives a context parameter providing read access to other collections.

However, *anti-joins* ("entries with no matching partner") are absent from all 48 examples. Reactive services often need such patterns:

- **Orphan detection**: orders with no matching customer record (data-integrity alerts).
- **Unacknowledged alerts**: alerts with no entry in an acknowledgments table (pending-item dashboards).
- **Unassigned tickets**: support tickets with no entry in assignments (queue management).

These cannot be implemented in Skip: there is no set-difference operator, no anti-join primitive, and no way to filter one collection based on the absence of keys in another.

**Contributions.** We identify an anti-join combinator, filterNotMatchingOn, as the single missing operator, describe how to implement it in Skip (Section 6.3), and prove that adding it achieves expressive equivalence with relational algebra extended with aggregates. Specifically:

- each Skip combinator is definable by an RA expression, and
- every RA query (including difference) compiles to combinators.

This provides a precise expressivity benchmark for Skip's reactive system.

We also discuss the connection to first-order logic, which provides access to classical locality principles from finite model theory (Gaifman's theorem), ensuring bounded update propagation.

## 2 Collections as Relational Structures

Fix two non-empty sets:

- a set of *keys* $K$, equipped with a total order $\leq_K$; and

- a set of *values* $V$.

In Skip, keys are JSON values with a fixed total order $\leq_{\text{json}}$, and collections are finite, but we keep the presentation abstract.

**Definition 1** (Collection)**.** *A* collection *over* $(K, V)$ *is a finite subset* $R \subseteq K \times V$*. We write* $R(k, v)$ *as shorthand for* $(k, v) \in R$*.*

We regard collections as relations in the standard database sense: a collection $R \subseteq K \times V$ is a binary relation with attributes for the key and value components. The key order $\leq_K$ is available as an auxiliary binary relation for use in selections and aggregate computations.

In this setting, a *view* of a collection is simply another relation $R' \subseteq K' \times V'$ (possibly over different key / value types) that is definable from $(K, V, R, \leq_K)$ using relational algebra operators.

## 3 Relational Algebra with Aggregates

We work with standard relational algebra extended with aggregates, using the following operators:

- **Selection** $\sigma_P(R)$: keeps tuples satisfying predicate $P$;
- **Projection** $\pi_A(R)$: projects onto attributes $A$;
- **Renaming** $\rho_{a/b}(R)$: renames attribute $b$ to $a$;
- **Union** $R_1 \cup R_2$: set union of compatible relations;
- **Difference** $R_1 - R_2$: set difference;
- **Cartesian product** $R_1 \times R_2$: all pairs of tuples;
- **Natural join** $R_1 \bowtie R_2$: join on common attributes.

To capture prefix operators such as `take`, we extend RA with simple *aggregate* operators. We allow grouping and aggregation of the form

$$\gamma_{A;\texttt{count}(*)\to c}(R),$$

which groups $R$ by attributes $A$ and computes a count for each group, storing the result in a new attribute $c$.

**Definition 2** (RA with aggregates)**.** *We write* $\mathrm{RA}[R, \leq_K, \#]$ *for relational algebra with difference and counting aggregates, over a base relation* $R$ *with access to the key order* $\leq_K$*. We say that a binary relation* $R' \subseteq K' \times V'$ *is* RA-definable *from* $(K, V, R, \leq_K)$ *if there exists an expression in* $\mathrm{RA}[R, \leq_K, \#]$ *whose result, when evaluated on the given structure, equals* $R'$*.*

In what follows, each structural combinator will be paired with such a defining RA expression. The key point is that these expressions use only:

- the base relation $R$,
- the key order $\leq_K$ (as a selection predicate),
- finitely many parameters (e.g. slice bounds, mappings), and
- counting aggregates for the prefix combinator.

# 4 Overview of Combinator Operators

We now present the combinator operators of the reactive calculus. These are organised into two groups: the *core operators* available in the Skip runtime bindings, and *extensions* that provide additional expressive power.

**Core operators (Skip bindings).** These operators are directly available in the Skip reactive runtime:

| **Combinator** | **Type** | **Ref.** |
|---|---|---|
| $\mathsf{map}_f$ | $R \to R'$ | 5.1 |
| $\mathsf{slice}_{[a,b]}$ | $R \to R$ | 5.2 |
| $\mathsf{slices}_{\mathcal{I}}$ | $R \to R$ | 5.3 |
| $\mathsf{take}_n$ | $R \to R$ | 5.4 |
| merge | $R^m \to R$ | 5.5 |
| $\mathsf{reduce}_{R_{\mathrm{red}}}$ | $R \to R'$ | 5.6 |

**Extensions.** The following operators extend the core calculus to achieve full RA expressiveness (Section 6):

| **Combinator** | **Type** | **Ref.** |
|---|---|---|
| $\mathsf{joinOn}_{f_1,f_2}$ | $R_1 \times R_2 \to R'$ | 6.2 |
| $\mathsf{filterNotMatchingOn}_{f_1,f_2}$ | $R_1 \times R_2 \to R_1$ | 6.3 |

Each operator is shown to be definable in $\mathrm{RA}[R, \leq_K, \#]$ in the referenced subsection. Note that filter is not listed as it can be implemented via map (by mapping non-matching entries to an empty output).

# 5 Structural Combinators on Collections (Skip Bindings)

This section formalises the core structural operators that are directly available in the Skip reactive runtime bindings. We show that each is definable in $\mathrm{RA}[R, \leq_K, \#]$.

## 5.1 Entry-wise mapping

Let $K', V'$ be sets of keys and values for the output collection.

**Definition 3** (Entry-wise map)**.** *Let $f : K \times V \to K' \times V'$ be a function. Given a collection $R \subseteq K \times V$, we define the mapped collection*

$$\mathsf{map}_f(R) \subseteq K' \times V'$$

*by*

$$\mathsf{map}_f(R)(k', v') \iff \exists k \in K, \exists v \in V.\ R(k, v) \wedge f(k, v) = (k', v').$$

**Lemma 4** (RA-definability of entry-wise map)**.** *For any fixed function $f : K \times V \to K' \times V'$, the relation $\mathsf{map}_f(R)$ is definable in $\mathrm{RA}[R, \leq_K, \#]$ by the expression*

$$\pi_{k',v'}(\rho_{k'/f_1(k,v),v'/f_2(k,v)}(R)),$$

*where $f_1$ and $f_2$ are the component functions of $f$, treated as extended projection expressions.*

This matches the intuitive semantics of a structural `map` on collections: each input entry is transformed independently.

## 5.2 Single-range slice

**Definition 5** (Slice)**.** *Given bounds $a, b \in K$ with $a \leq_K b$ and a collection $R \subseteq K \times V$, we define*

$$\mathsf{slice}_{[a,b]}(R) \subseteq K \times V$$

*by*

$$\mathsf{slice}_{[a,b]}(R)(k, v) \iff R(k, v) \wedge a \leq_K k \wedge k \leq_K b.$$

**Lemma 6** (RA-definability of slice)**.** *For any fixed bounds $a, b \in K$, the relation $\mathsf{slice}_{[a,b]}(R)$ is definable in $\mathrm{RA}[R, \leq_K, \#]$ by the expression*

$$\sigma_{a \leq_K k \wedge k \leq_K b}(R).$$

### 5.3 Multi-range slices

**Definition 7** (Multi-range slices)**.** *Let $\mathcal{I} = \{[a_1, b_1], \ldots, [a_n, b_n]\}$ be a finite family of intervals in $K$. Given $R \subseteq K \times V$, we define*

$$\mathsf{slices}_{\mathcal{I}}(R) \subseteq K \times V$$

*by*

$$\mathsf{slices}_{\mathcal{I}}(R)(k, v) \iff R(k, v) \wedge \bigvee_{i=1}^{n} \big(a_i \leq_K k \wedge k \leq_K b_i\big).$$

**Lemma 8** (RA-definability of multi-range slices)**.** *For any fixed finite family of intervals $\mathcal{I}$, the relation $\mathsf{slices}_{\mathcal{I}}(R)$ is definable in $\mathrm{RA}[R, \leq_K, \#]$ by the expression*

$$\bigcup_{i=1}^{n} \sigma_{a_i \leq_K k \wedge k \leq_K b_i}(R),$$

*i.e., the union of the single-interval slices.*

### 5.4 Prefix by key rank

We next formalise a prefix operator analogous to `take` in the reactive calculus: keep the first $n$ keys in the global order, and discard the rest.

**Definition 9** (Key rank)**.** *Let $R \subseteq K \times V$. For $k \in K$, define the* support *of $k$ in $R$ by*

$$\mathsf{supp}_R(k) \iff \exists v.\, R(k, v).$$

*We define the* key rank *of $k$ in $R$ as the natural number*

$$\mathrm{rank}_R(k) := \#\{k' \in K \mid k' <_K k \wedge \mathsf{supp}_R(k')\},$$

*where $\#$ denotes cardinality.*

**Definition 10** (Prefix operator)**.** *Given $n \in \mathbb{N}$ and $R \subseteq K \times V$, we define*

$$\mathsf{take}_n(R) \subseteq K \times V$$

*by*

$$\mathsf{take}_n(R)(k, v) \iff R(k, v) \wedge \mathrm{rank}_R(k) < n.$$

**Lemma 11** (RA+agg-definability of prefix)**.** *For any fixed $n \in \mathbb{N}$, the relation $\mathsf{take}_n(R)$ is definable in $\mathrm{RA}[R, \leq_K, \#]$ as follows:*

1. *Compute the set of distinct keys: $K_R := \pi_k(R)$.*

2. *For each key $k$, count the number of strictly smaller keys: form the relation of pairs $(k, k')$ with $k' <_K k$, join with $K_R$ on the $k'$ component, group by $k$, and count.*

3. *Select keys whose count is less than $n$.*

4. *Semi-join the result with the original relation $R$.*

Thus the `take` combinator on collections aligns with a standard RA-with-aggregates construction on ordered structures.

### 5.5 Finite merge

Finally, we formalise finite merge of collections.

**Definition 12** (Finite merge)**.** *Let $R_1, \ldots, R_m \subseteq K \times V$ be collections. Their merge is the relation*

$$\mathsf{merge}(R_1, \ldots, R_m) \subseteq K \times V$$

*defined by*

$$\mathsf{merge}(R_1, \ldots, R_m)(k, v) \iff \bigvee_{i=1}^{m} R_i(k, v).$$

**Lemma 13** (RA-definability of finite merge)**.** *The merged relation is definable by $R_1 \cup R_2 \cup \cdots \cup R_m$.*

In other words, finite merge corresponds exactly to finite union in relational algebra.

### 5.6 Per-key reduction

The reduce operator aggregates the multiset of values at each key using a user-specified reducer.

**Definition 14** (Per-key reduce)**.** *Let $R \subseteq K \times V$ be a collection and let $R_{\text{red}} = (\mathsf{init}, \mathsf{add}, \mathsf{remove})$ be a reducer specification with accumulator type $A$, where:*

- $\mathsf{init} : A$ *is the initial accumulator value,*
- $\mathsf{add} : A \times V \to A$ *incorporates a value, and*
- $\mathsf{remove} : A \times V \to A$ *removes a value.*

*We define the reduced collection*

$$\mathsf{reduce}_{R_{\text{red}}}(R) \subseteq K \times A$$

*by: for each key $k$, the output value is the result of folding* add *over the multiset $\{v \mid R(k, v)\}$ starting from* init*.*

**Lemma 15** (RA+agg-definability of reduce)**.** *For any reducer $R_{\text{red}}$ whose operations correspond to standard SQL aggregates (count, sum, min, max, etc.), the relation $\mathsf{reduce}_{R_{\text{red}}}(R)$ is definable in $\text{RA}[R, \leq_K, \#]$ using the grouping operator $\gamma$.*

# 6 Completing the Calculus for RA Equivalence

The formal combinator calculus of the previous section (with map, slice, etc. operating on single collections) cannot express joins or set difference. To achieve equivalence with relational algebra, the formal calculus requires operators that relate *two* collections.

In this section we define two such operators:

- joinOn—needed to express $\times$ and $\bowtie$; and
- filterNotMatchingOn—needed to express $-$ (difference).

**Relationship to Skip.** These operators have different status with respect to Skip's actual API:

- joinOn *is* expressible in Skip today via a map-with-lookup idiom (Section 6.1). It appears here to formalize that capability.
- filterNotMatchingOn *is not* expressible in Skip (Section 6.1). It is the genuinely missing operator.

## 6.1 Joins via map-with-lookup in Skip

Before introducing the formal joinOn combinator, we note that the Skip runtime already supports a pattern that achieves the same effect. In Skip, the map function receives not just each $(k, v)$ entry but also a *context* object that provides read access to other collections in the reactive graph.

For example, to join an `orders` collection with a `customers` collection on `customerId`, a Skip programmer writes:

```
orders->map((orderId, order, ctx) => {
  let customer = ctx.customers.getUnique(order.customerId)
  (order.customerId, (order, customer))
})
```

The `ctx.customers.getUnique` call looks up the customer record by key, effectively implementing an equi-join. Skip's reactive runtime tracks these cross-collection dependencies: when an entry in `customers` changes, any `orders` entries that referenced it are automatically re-evaluated.

**Why this is not captured by the formal map.** The formal definition of $\mathsf{map}_f(R)$ in Section 5.1 applies a fixed function $f : K \times V \to K' \times V'$ to each entry of a *single* collection $R$. It does not model the context parameter or cross-collection lookups. As a consequence:

- The formal calculus with only map, slice, take, merge, and reduce cannot express joins between two independent base collections.
- Completeness with respect to RA requires an explicit operator that relates entries from *two* collections.

This motivates the joinOn combinator defined below, which formalises the join capability that Skip users already access via the map-with-lookup idiom.

**Anti-joins appear inexpressible.** While joins are expressible via map-with-lookup, *anti-joins* ("entries of $R_1$ with no matching key in $R_2$") do not appear to be expressible with Skip's current bindings.

Inspecting the Skip API, the `EagerCollection` type provides:

- `map`, `reduce`, `mapReduce` — transformation and aggregation
- `slice`, `slices`, `take` — filtering by key range or count
- `merge` — union of collections

There is no set-difference operator, no anti-join primitive, and no way to filter one collection based on the *absence* of keys in another. The `LazyCompute` mechanism could compute an anti-join on demand, but produces a lazy collection, not an eager reactive one.

The 48-example catalogue contains no "NOT IN" style queries, consistent with this limitation. The filterNotMatchingOn operator defined in Section 6.3 fills this gap with clear, formally specified semantics.

### 6.2 Join on a derived key

Let $K_1, V_1$ and $K_2, V_2$ be key/value types for two input collections, and let $J$ be a set of *join keys*.

**Definition 16** (Join on key)**.** *Let $R_1 \subseteq K_1 \times V_1$ and $R_2 \subseteq K_2 \times V_2$ be collections, and let*

$$f_1 : K_1 \times V_1 \to J, \qquad f_2 : K_2 \times V_2 \to J$$

*be fixed functions (join-key extractors). We define the* join on key *collection*

$$\mathsf{joinOn}(f_1, f_2; R_1, R_2) \subseteq J \times (V_1 \times V_2)$$

*by*

$$\begin{aligned}\mathsf{joinOn}(f_1, f_2; R_1, R_2)(j, (v_1, v_2)) \iff{}& \exists k_1, k_2.\ R_1(k_1, v_1) \wedge R_2(k_2, v_2) \\ & \wedge\ f_1(k_1, v_1) = j \wedge f_2(k_2, v_2) = j.\end{aligned}$$

This operator subsumes both ordinary cartesian product (by taking $J$ to be a singleton) and equi-joins (by taking $f_1$ and $f_2$ to select an existing key component).

**Lemma 17** (RA-definability of join on key)**.** *The join-on-key relation is definable in RA as follows:*

1. *Extend $R_1$ with a new attribute $j_1 := f_1(k_1, v_1)$;*
2. *Extend $R_2$ with a new attribute $j_2 := f_2(k_2, v_2)$;*
3. *Compute the natural join on $j_1 = j_2$;*
4. *Project onto $(j, v_1, v_2)$.*

### 6.3 Filtering entries without matches

We now formalise a combinator that keeps entries from a "left" collection whose join key has no matching entry on the "right".

**Definition 18** (Filter-not-matching on key)**.** *Let $R_1 \subseteq K_1 \times V_1$ and $R_2 \subseteq K_2 \times V_2$ be collections, let $J$ be a set of join keys, and let*

$$f_1 : K_1 \times V_1 \to J, \qquad f_2 : K_2 \times V_2 \to J$$

*be join-key extractors as above. We define the filtered collection*

$$\mathsf{filterNotMatchingOn}(f_1, f_2; R_1, R_2) \subseteq K_1 \times V_1$$

*by: $(k_1, v_1)$ belongs to this collection iff $R_1(k_1, v_1)$ holds and there is no $(k_2, v_2)$ with $R_2(k_2, v_2)$ and $f_1(k_1, v_1) = f_2(k_2, v_2)$.*

Intuitively, this operator keeps precisely those entries of the left collection whose join key has no partner on the right, matching the usual "A except rows that match B on this key" pattern from query languages.

**Lemma 19** (RA-definability of filter-not-matching)**.** *The filter-not-matching relation is definable in RA using difference and anti-join:*

1. *Extend $R_1$ with $j_1 := f_1(k_1, v_1)$;*
2. *Extend $R_2$ with $j_2 := f_2(k_2, v_2)$ and project to get the set of join keys present in $R_2$: $J_2 := \pi_{j_2}(R_2')$;*
3. *Compute the anti-join: $R_1' - (R_1' \ltimes J_2)$, or equivalently, keep rows of $R_1'$ whose $j_1$ is not in $J_2$;*
4. *Project back to $(k_1, v_1)$.*

**Implementation in Skip.** To extend Skip with filterNotMatchingOn, add a method to `EagerCollection`:

```
filterNotMatchingOn<K2,V2,J>(f1, other, f2)
  : EagerCollection<K1, V1>
```

The implementation maintains two indices: (i) $rIdx : J \to \mathbb{N}$, counting entries in `other` per join key; and (ii) $lByKey : J \to \mathcal{P}(K_1 \times V_1)$, grouping entries of `this` by join key.

**Initial computation.** For each $(k_1, v_1) \in R_1$, emit iff $rIdx[f_1(k_1, v_1)] = 0$.

**When `other` changes.** On add of $(k_2, v_2)$: let $j = f_2(k_2, v_2)$; increment $rIdx[j]$; if count went $0 \to 1$, remove $lByKey[j]$ from output. On remove: decrement; if $1 \to 0$, add $lByKey[j]$ to output.

**When `this` changes.** On add of $(k_1, v_1)$: let $j = f_1(k_1, v_1)$; add to $lByKey[j]$; emit iff $rIdx[j] = 0$. On remove: delete from $lByKey[j]$ and output.

This is standard incremental view maintenance for anti-join. Note that filterNotMatchingOn is the right primitive: set difference $R_1 - R_2$ is the special case with identity extractors, while deriving filterNotMatchingOn from difference would require computing a join first (wasteful).

### 6.4 Soundness and completeness

We now collect the previous definability results into a soundness theorem for the extended combinator algebra with respect to relational algebra with aggregates, and state the corresponding completeness result.

**Theorem 20** (RA-soundness of the extended combinators)**.** *Let $\mathcal{E}$ be the class of collection-valued expressions built from a base relation $R \subseteq K \times V$ using:*

- *the Skip binding operators* map*,* slice*,* slices*,* take*,* merge*, and* reduce*;*
- *the join operator* joinOn *on derived keys; and*
- *the filter-not-matching operator* filterNotMatchingOn *on derived keys;*

*with fixed function symbols for all key and join-key extractors and fixed numeric parameters for prefixes and ranges. Then for every $E \in \mathcal{E}$ there exists an expression in relational algebra with difference and aggregates over $R$ whose denotation coincides with that of $E$.*

*Proof sketch.* Each primitive combinator has been given an explicit characterisation in terms of the corresponding relational algebra operators (selection, projection/renaming, union, join, and difference) together with simple grouping and aggregates. Closure under composition follows by structural induction on expressions: replacing each primitive occurrence by the corresponding relational algebra fragment yields an equivalent algebra expression for the overall combinator expression. □

**Theorem 21** (RA-completeness of the extended combinators)**.** *Over finite structures $(K, V, R, \leq_K)$ with a total order on keys and counting aggregates as above, every relation definable by an expression in relational algebra with difference and aggregates over $R$ is equivalent to the denotation of some expression $E \in \mathcal{E}$ built from:*

- *the Skip binding operators of Section 5,*
- *the join and filter-not-matching operators of this section.*

*In other words, the extended reactive combinator algebra is expressively complete for relational algebra with difference and aggregates on this class of structures.*

The proof proceeds by the explicit compilation algorithm given in Section 7 below, which translates any relational algebra expression into a combinator expression with the same denotation.

# 7 Algorithmic Compilation from RA to Combinators

We now describe an explicit compilation procedure that turns any expression of $\mathrm{RA}[R, \leq_K, \#]$ into a combinator expression built from the operators of the extended reactive calculus. This gives a constructive witness for the RA-completeness theorem.

## 7.1 Syntax of RA expressions

We use the relational algebra with aggregates $\mathrm{RA}[R, \leq_K, \#]$ defined in Section 3, over a single base relation symbol $R(k, v)$. An RA expression $E$ is built inductively from the base relation $R$ using the standard operators ($\sigma$, $\pi$, $\rho$, $\cup$, $-$, $\times$, $\bowtie$) and the grouping/aggregation operator $\gamma$.

## 7.2 Syntax of combinator expressions

We use the combinator operators defined in Sections 5 and 6 (see the summary tables in Section 4). A combinator expression $E$ is built inductively from a base collection $\mathsf{Base}$ (corresponding to $R$) using these operators. We write $\mathsf{CombExpr}$ for the set of such expressions.

## 7.3 Compilation function: RA to Combinators

We define the compilation function $\mathsf{compile} : \mathrm{RAExpr} \to \mathsf{CombExpr}$ by structural recursion on RA expressions. We use pattern matching notation where $\mathsf{id}(k, v) := (k, v)$ denotes the identity key extractor.

| RA Expression | Compiled to | Combinator Expression |
|---|---|---|
| $R$ | := | $\mathsf{Base}$ |
| $\sigma_P(E)$ | := | $\mathsf{filter}_P(\mathsf{compile}(E))$ |
| $\pi_A(E)$ | := | $\mathsf{map}_{f_A}(\mathsf{compile}(E))$ |
| $\rho_\alpha(E)$ | := | $\mathsf{map}_{f_\alpha}(\mathsf{compile}(E))$ |
| $E_1 \cup E_2$ | := | $\mathsf{merge}(\mathsf{compile}(E_1), \mathsf{compile}(E_2))$ |
| $E_1 - E_2$ | := | $\mathsf{filterNotMatchingOn}(\mathsf{id}, \mathsf{id};$ $\mathsf{compile}(E_1), \mathsf{compile}(E_2))$ |
| $E_1 \times E_2$ | := | $\mathsf{map}_{f_\times}(\mathsf{joinOn}(c, c;$ $\mathsf{compile}(E_1), \mathsf{compile}(E_2)))$ |
| $E_1 \bowtie_\theta E_2$ | := | $\mathsf{map}_{f_\bowtie}(\mathsf{filter}_{P_{\text{res}}}($ $\mathsf{joinOn}(f_1, f_2; \mathsf{compile}(E_1), \mathsf{compile}(E_2))))$ |
| $\gamma_{G;\text{Agg}}(E)$ | := | $\mathsf{map}_{f_\gamma}(\mathsf{reduce}_{R_{\text{red}}}($ $\mathsf{map}_{f_G}(\mathsf{compile}(E))))$ |

**Auxiliary functions.** The compilation uses the following auxiliary functions:

- $f_A : (k, v) \mapsto (k', v')$ where $(k', v')$ is the projected tuple containing only attributes in $A$;
- $f_\alpha : (k, v) \mapsto (k', v')$ applies the attribute renaming $\alpha$;
- $c : (k, v) \mapsto ()$ is the constant function (for cartesian product);
- $f_1, f_2$ extract the join key from rows of $E_1, E_2$ respectively;
- $P_{\text{res}}$ is the residual predicate after extracting equi-join conditions from $\theta$;
- $f_\times, f_\bowtie$ package joined pairs into single output rows;
- $f_G : (k, v) \mapsto (g, v')$ re-keys by the grouping attributes $G$;
- $R_{\text{red}}$ is a reducer implementing the aggregates in Agg;
- $f_\gamma$ formats the aggregated result.

**Key insight: Difference via identity keys.** For set difference $E_1 - E_2$, we use $\mathsf{filterNotMatchingOn}$ with *identity* key extractors on both sides. This means $(k, v) \in \mathsf{compile}(E_1 - E_2)$ iff $(k, v) \in \mathsf{compile}(E_1)$ and there is no $(k', v')$ in $\mathsf{compile}(E_2)$ with $(k, v) = (k', v')$—i.e., the entry is not in $E_2$.

**Correctness.** Because compile is defined by case distinction on the top-level constructor and recurses only on strictly smaller subexpressions, it is a well-defined structural recursion. By construction, the denotation of $\mathsf{compile}(E)$ matches that of the original RA expression $E$.

**Lean formalisation.** The Lean formalisation (`ReactiveRel.lean`) proves soundness and completeness theorems for the full set of operators from the paper:

- **RA operators:** $\sigma$ (select), $\pi$ (project), $\rho$ (rename), $\cup$ (union), $-$ (difference), $\times$ (product), $\bowtie$ (join), and $\gamma$ (aggregate).
- **Combinator operators:** map, filter, slice, slices, take, merge, reduce, joinOn, and filterNotMatchingOn.

For tractability, the Lean formalisation makes the following simplifications:

1. *Monomorphic types:* all expressions have fixed key/value types $K \times V \to K \times V$, avoiding universe-level complexity. As a consequence, the compilation of $\times$ (product) is degenerate, and the output-formatting map steps in the compilation of $\bowtie$ (join) and $\gamma$ (aggregate) are omitted.
2. *Singleton aggregation:* the semantics of reduce and $\gamma$ assume at most one value per key, rather than folding over a multiset. The compilation structure is correct for the general case; only the semantic definitions are simplified.

The main theorems `compileCombToRA_sound` and `compileRAToComb_sound` establish that the compilation functions preserve semantics under these simplifications.

# 8 Connection to First-Order Logic and Locality

The classical connection between relational algebra and first-order logic provides the theoretical foundation for understanding locality properties of the reactive combinators. Since relational algebra is expressively equivalent to first-order logic over finite structures, and the extended combinator algebra is complete for relational algebra with aggregates, every combinator expression is semantically equivalent to a first-order formula.

This connection yields locality properties via classical results from finite model theory: first-order queries are *local*, meaning the truth value at a tuple

depends only on a bounded neighborhood. For the reactive combinators, this means updates affect only entries within bounded distance in the key space.

The structural operators are inherently local: `map`, `filter`, and `slice` operate pointwise; `merge` combines independent collections; `take` depends only on key order and a global count. Join and filter-not-matching operators introduce cross-collection dependencies, but these are bounded by the fixed join-key extractors.

Per-key reducers maintain locality by construction: each key's accumulator is updated independently, and the well-formedness laws ensure order-independent, invertible updates.

In summary, locality follows from the expressiveness equivalence with first-order logic and the classical locality properties of first-order queries, providing a theoretical guarantee of bounded update propagation for incremental maintenance in distributed systems.