Example Catalogue for Reactive Views

Contents

1 Simple Per-Key Views **1**

2 Enriched-State Views **3**

3 Set and Index Views **4**

4 Windowed and Session-Based Views **5**

5 History and Ordered-State Patterns **6**

6 Graph and Relational Incremental Maintenance **7**

7 Business Metrics and UI-Composed Summaries **9**

1 Simple Per-Key Views

Example 1.1 (Active members per group service (Skip docs)). *Input collection* `memberships : GroupId × UserId`*, plus a separate view or flag indicating which users are active. Define a derived collection* `activeMembers : GroupId → int` *by mapping only active* `(groupId, userId)` *pairs and using a count reducer per* `GroupId`*. On insertion of an active membership, increment the corresponding group's count; on removal or when a user becomes inactive, decrement. This yields per-group active-member counts for use in admin dashboards or access-control views.*

Example 1.2 (Total sales by category service (Skip blog)). *Input collection* `sales : SaleId × Sale`*, where* `Sale` *includes* `categoryId` *and* `amount`*. Define a resource* `categoryTotals : CategoryId → Money` *computed by mapping each sale to* `(categoryId, amount)` *and reducing per category with a sum reducer. New sales add* `amount` *to the appropriate category total; corrections or deletions subtract the previous amount, keeping real-time revenue or inventory value per category.*

Example 1.3 (Portfolio value by sector service (Skip blog)). *Input collection* `positions : PositionId × Position`*, where* `Position` *includes* `sector : SectorId`*,* `shares`*, and* `price`*. A derived collection* `sectorValue : SectorId → Money` *is defined by mapping each position to* `(sector, shares * price)` *and summing per sector. On position updates (change in shares or price) or position insert/delete, the service applies the delta in* `shares * price` *to the corresponding sector, maintaining live sector-level portfolio values.*

Example 1.4 (Global active-user count service (conceptual Skip example)). *Input collection* `users : UserId × UserState`*, where* `UserState` *includes an* `isActive` *flag. Define a single-key collection* `activeCount : Unit → int` *by mapping each active user to key* `()` *and using a count*

reducer. When a user becomes active, the service increments `activeCount[()]`*; when they become inactive or are deleted, it decrements, providing a live global active-user metric.*

Example 1.5 (Max value per key service (Skip helper `Max`))**.** *Input collection* `measurements : KeyId × Value`*. Per key, the service maintains a single-number accumulator storing the current maximum value. On add of* `(k,v)`*, it sets* `max[k] := max(max[k], v)`*; on remove, if* `v` *is not equal to the stored maximum, no change is needed, otherwise the service recomputes the maximum from remaining values for* `k`*. The derived view* `maxPerKey : KeyId → Value` *exposes these maxima.*

Example 1.6 (Min value per key service (Skip helper `Min`))**.** *Analogous to the max service, with accumulator* `min[k]` *initialized to a top element. On add, it updates* `min[k] := min(min[k], v)`*; on remove of a non-minimum value, it does nothing; on removal of the current minimum, it recomputes the minimum from the remaining values for that key. The resource* `minPerKey : KeyId → Value` *supports queries like "lowest price per product" or "earliest timestamp per stream".*

Example 1.7 (Continuous count per key service (Kafka Streams KTable style))**.** *Input collection* `events : KeyId × Event`*. The service maintains* `counts : KeyId → int` *where each new event with key* `k` *increments* `counts[k]`*. If deletions or tombstones are modeled, removing an event decrements* `counts[k]`*. This corresponds to classic word-count or per-key event-count services in KTable-like APIs.*

Example 1.8 (Per-window sum service (Flink, Spark, Beam variants))**.** *Input collection* `values : (KeyId, WindowId) × Number`*, where* `WindowId` *encodes a time bucket or logical window. Define a view* `windowSum : (KeyId, WindowId) → Number` *that, for each key/window pair, maintains the sum of all values mapped to that pair using a simple additive reducer. Window creation and expiration are handled by separate logic that manages the* `WindowIds`*; within each window bucket, the aggregator is a pure per-key fold.*

Example 1.9 (Aggregated materialized view service (Materialize SQL `GROUP BY`))**.** *Input base collection* `Sales(productId, amount)`*. Define a derived view* `ProductTotals : ProductId → Money` *corresponding to* `SELECT productId, SUM(amount) FROM Sales GROUP BY productId`*. On insert of a sale for* `p`*, the service increments* `ProductTotals[p]` *by* `amount`*; on delete or retraction, it decrements by the same amount. This is a direct per-key sum over the* `Sales` *collection, mirroring Materialize's incrementally maintained group-by.*

Example 1.10 (FRP event-counter service (Reactive-banana, Yampa, Elm `foldp`))**.** *Input collection* `clicks : CounterId × unit` *representing button-click events (or other discrete events) keyed by component or counter identifier. The service maintains* `clickCount : CounterId → int`*, with each event incrementing the corresponding counter. In languages like Elm or Yampa this is expressed as* `foldp (+1) 0` *over an event stream; in a Skip-style service it is a per-key count reducer over the* `clicks` *collection.*

Example 1.11 (Cart totals and sums service (Redux/UI frameworks))**.** *Input collection* `cartItems : UserId × CartItem`*, where* `CartItem` *has fields* `productId`*,* `quantity`*, and* `unitPrice`*. Define a resource* `cartTotal : UserId → Money` *by mapping each cart item to* `(userId, quantity * unitPrice)` *and reducing with a sum per user. Front-end frameworks typically compute this in selectors or computed properties; this service maintains the same quantities reactively on the backend.*

Example 1.12 (Per-player score service (React `useReducer` example generalized))**.** *Input collection* `scoreEvents : PlayerId × int` *where each entry represents a score delta (e.g. +1) for*

a player. The service maintains `scores : PlayerId → int`*, incrementing* `scores[p]` *by the event's delta for each* `(p,delta)` *and decrementing if negative deltas are supported. This captures multi-player scoreboards or leaderboards as simple per-player counters.*

Example 1.13 (Vertex-degree counting service (incremental graph systems))**.** *Input collection* `edges : EdgeId × (src : NodeId, dst : NodeId)`*. The service derives a view* `degree : NodeId → int` *that counts incident edges per node. On insertion of edge* `(u,v)`*, it increments* `degree[u]` *and* `degree[v]` *(for undirected graphs) or just* `degree[v]` *(for in-degree in directed graphs); on deletion it decrements accordingly. This models the simplest incremental graph metric as a per-key count.*

2 Enriched-State Views

Example 2.1 (Average rating per item (Skip conceptual service))**.** *Input collection* `ratings : ItemId × Rating`*, where* `Rating` *has a numeric field* `score`*. Define a reducer with accumulator state* `(sum : float, count : int)` *per* `ItemId`*; on add of a rating with score* s*, update to* `(sum + s, count + 1)`*, and on remove update to* `(sum - s, count - 1)` *when* `count > 1`*, or signal fallback when* `count = 1`*. Expose a derived view* `avgRating : ItemId → float` *that maps each accumulator to* `sum / count` *when* `count > 0`*.*

Example 2.2 (Histogram / frequency distribution per key (Skip conceptual service))**.** *Input collection* `events : KeyId × Value`*, with a fixed bucketization function* `bucket : Value → BucketId`*. Define an accumulator* `hist : BucketId → int` *for each* `KeyId`*; on add of value* `v`*, increment* `hist[bucket(v)]`*, and on remove decrement* `hist[bucket(v)]`*, deleting buckets whose count drops to zero. The exposed resource* `histograms : KeyId → Map<BucketId, int>` *provides per-key histograms suitable for dashboards (e.g. distribution of response times or purchase amounts).*

Example 2.3 (Distinct count with reference counts (Skip conceptual service))**.** *Input collection* `events : KeyId × Value`*. Define accumulator state* `freq : Value → int` *per key; on add of value* `v`*, set* `freq[v] := freq[v] + 1` *(defaulting from zero), and on remove set* `freq[v] := freq[v] - 1`*, deleting entries whose frequency becomes zero. The view* `distinctCount : KeyId → int` *returns, for each key, the number of entries in* `freq` *(i.e. values with positive frequency), giving an exact per-key distinct count that supports removals.*

Example 2.4 (Weighted average per key (Flink-style UDAF as Skip service))**.** *Input collection* `measurements : KeyId × (value : float, weight : float)`*. For each key, use accumulator* `(sumWeights : float, sumWeightedValues : float)`*; on add of* `(v,w)`*, update to* `(sumWeights + w, sumWeightedValues + w * v)`*, and on remove update to* `(sumWeights - w, sumWeightedValues - w * v)` *when valid, otherwise fall back. Expose a view* `weightedAvg : KeyId → float` *defined as* `sumWeightedValues / sumWeights` *when* `sumWeights > 0`*.*

Example 2.5 (Top-2 / Top-K per group (Flink & Materialize-inspired service))**.** *Input collection* `scores : GroupId × (itemId : Id, score : float)`*. Per group, maintain accumulator state as a bounded ordered list of up to* K *pairs* `(itemId, score)`*, sorted descending by score. On add, insert the new pair into the list (evicting the lowest-scoring element if the list exceeds length* K*); on remove, if the removed item is in the list, delete it and optionally track enough extra candidates (e.g. store the top* $K+M$*) or trigger recompute for that group. Expose a resource* `topK : GroupId → array<(Id, float)>` *that returns the current top-K items per group.*

Example 2.6 (Top-N ranking per key (Skip conceptual)). *Generalizing the Top-K pattern, define a service over* `metrics : KeyId × (entityId : Id, score : float)`. *For each* `KeyId`, *maintain a sorted data structure (e.g. a bounded heap or balanced tree) of the top N* `(entityId, score)` *entries. Add operations insert or update entries based on score; remove operations delete entries when they are present, and may trigger a recompute of the per-key top-N if the removed entity was not tracked. The exported resource* `topN : KeyId → array<(Id, float)>` *provides a ranked list per key.*

Example 2.7 (Approximate distinct count with HLL (Flink, Beam, Materialize-inspired service)). *Input collection* `events : KeyId × UserId`. *Per key, the accumulator is a HyperLogLog sketch* `hll`, *initialized empty; on add of user* `u`, *apply the HLL update algorithm to incorporate* `u`, *and on remove either ignore (append-only approximation) or use a more advanced HLL variant if available. The view* `approxDistinct : KeyId → int` *estimates the number of distinct users per key using the HLL cardinality estimate.*

Example 2.8 (Sliding-window averages with sum & count (Kafka Streams, Spark-inspired)). *Input collection* `readings : (SensorId, WindowId) × float`, *where the* `WindowId` *encodes a time bucket or window key. For each* `(SensorId, WindowId)` *pair, maintain accumulator* `(sum : float, count : int)` *updated on add/remove as in the average-rating example. The resource* `windowAvg : (SensorId, WindowId) → float` *reports per-sensor averages for each active window, leaving window management (creating and retiring* `WindowId`*s) to separate logic.*

Example 2.9 (Enriched min/max with secondary state (Skip conceptual)). *Input collection* `values : KeyId × Value`. *For each key, accumulator state extends a simple extremum with secondary information, for example* `(min : Value, secondMin : Value, countMin : int)`. *On add of* `v`, *update the triple appropriately (updating* `min`, `secondMin`, *and* `countMin`*); on remove of* `v`, *decrement* `countMin` *if* `v = min` *and, when* `countMin` *reaches zero, promote* `secondMin` *or trigger recomputation. The exposed view* `minPerKey : KeyId → Value` *returns the current minimum, benefiting from the enriched state to avoid full recomputation in many removal scenarios.*

3 Set and Index Views

Example 3.1 (Groups-per-user index service (Skip docs)). *Input collection* `groupMembers : GroupId × UserId` *encodes membership edges. Define a derived collection* `groupsPerUser : UserId → array<GroupId>` *by mapping each membership* `(g,u)` *to* `(u,g)` *and aggregating per user key so that all group IDs for a given user are collected. On insertion of a membership, the service appends* `g` *to the list for* `u`; *on deletion, it removes* `g` *from that list. This acts as an inverted index from users to the (multi)set of groups they belong to and supports queries like "list all groups for user* `u`*".*

Example 3.2 (Exact distinct count per key service (Skip conceptual)). *Input collection* `events : KeyId × Value`. *For each* `KeyId`, *maintain accumulator state* `freq : Value → int` *as in the enriched-state distinct-count example. The view* `distinctPerKey : KeyId → (distinctValues : array<Value>, count : int)` *exposes both the set of values with positive frequency and its size. On add of* `(k,v)`, *increment* `freq[v]` *for key* `k`, *inserting* `v` *into the exposed set when its frequency becomes positive; on remove, decrement* `freq[v]` *and delete* `v` *from the set when the frequency drops to zero. This yields exact per-key distinctness information suitable for analytics dashboards.*

Example 3.3 (Distinct visitors / approximate distinct service (streaming systems)). *Input collection* `visits : (Day, PageId) × UserId` *records page views, keyed by day and page. The service*

maintains, for each `(Day, PageId)` *pair, either: (i) an exact accumulator* `visitors : UserId → int` *(for low-volume cases), from which it derives* `uniqueVisitors : (Day, PageId) → int` *as the number of users with positive frequency; or (ii) a HyperLogLog sketch per* `(Day, PageId)`*, updated on each visit, and exposes* `approxUniqueVisitors : (Day, PageId) → int` *as the HLL cardinality estimate. This models streaming distinct-count queries (e.g. "unique visitors per day per page") seen in Flink/Beam/Materialize, with both exact and approximate variants.*

Example 3.4 (General inverted index and membership view service)**.** *Given a base collection* `relations : LeftId × RightId` *encoding edges of a bipartite relation (e.g. users–groups, documents–terms, products–tags), the service defines two derived collections: (i)* `rightPerLeft : LeftId → array<RightId>`*, collecting all right identifiers associated with a given left identifier; and (ii)* `leftPerRight : RightId → array<LeftId>`*, the inverted index mapping each right identifier to all left identifiers that reference it. On insertion or deletion of a pair* `(l,r)`*, the service symmetrically updates both collections, providing bidirectional membership views that support queries such as "all documents with term* `t`*" or "all tags for product* `p`*".*

4 Windowed and Session-Based Views

Example 4.1 (Sliding time-window aggregate service (Skip conceptual))**.** *Input collection* `events : (KeyId, Timestamp) × Payload`*, with wall-clock or logical timestamps. The service exposes views such as* `lastHourCount : KeyId → int` *or* `lastHourSum : KeyId → Number`*, defined conceptually as counts or sums over events with timestamps in the interval* `[now - 1h, now]`*. An external scheduler or time-based process is responsible for: (i) inserting new events with their timestamps, and (ii) deleting events once they fall outside the window, causing the corresponding per-key reducers to subtract their contributions. This yields sliding-window metrics by coupling per-key reducers with explicit time-based eviction.*

Example 4.2 (Session-based aggregation service (Skip conceptual, Flink/Kafka-inspired))**.** *Input collection* `userEvents : (UserId, Timestamp) × Event`*, ordered by arrival time. Sessions are defined per user by an inactivity gap parameter* `G` *(e.g. 30 minutes). A separate sessionization component maintains a mapping* `sessionId : (UserId, Timestamp) → SessionId` *by grouping consecutive events for a user whose inter-arrival gaps are less than* `G`*. The reactive service then maintains per-session metrics via a collection* `sessionMetrics : SessionId → MetricState` *(e.g. event count, total duration), updated with a simple reducer per* `SessionId`*. As sessions merge or close, the sessionization layer adjusts keys (splitting/merging* `SessionId`*s) and the reducers follow suit, mirroring Flink/Kafka session window counts.*

Example 4.3 (Fixed and sliding window sum/average service (Flink, Kafka Streams, Spark, Beam))**.** *Input collection* `measurements : (KeyId, WindowId) × float`*, where* `WindowId` *encodes a tumbling or sliding time bucket (e.g.* `(startTime, endTime)`*). For each* `(KeyId, WindowId)`*, the service maintains accumulators* `sum` *and* `count` *and exposes views: (i)* `windowSum : (KeyId, WindowId) → float` *and (ii)* `windowAvg : (KeyId, WindowId) → float` *as* `sum / count` *when* `count > 0`*. Windowing logic (assignment of events to one or more* `WindowId`*s and retirement of obsolete windows) is handled externally; within each window bucket, the aggregator is a standard per-key fold.*

Example 4.4 (Session window count service (Flink, Kafka Streams))**.** *Input an event stream* `events : (SessionKey, Timestamp) × Event`*, where* `SessionKey` *might be derived from user or IP. A session manager groups events into session identifiers* `SessId` *based on inactivity gaps*

and emits `(SessId, Event)` *pairs. The service maintains* `sessionCounts : SessId → int`*, incrementing on each event in the session; when a session closes (as determined by the session manager), the final count is retained or moved to a historical collection, and the live* `SessId` *entry is retired. This matches streaming session-window count semantics while keeping counting itself as a simple reducer.*

Example 4.5 (Materialize-style time-bounded active count service)**.** *Input collection* `intervalEvents : KeyId × (startTs : Time, endTs : Time)` *representing validity intervals for each key. The service defines a view* `activeNow : KeyId → int` *that, for the current logical time* `t`*, counts how many intervals for each key satisfy* `startTs ≤ t < endTs`*. Implementation-wise, updates can be modeled as two streams:* `(+1)` *at* `startTs` *and* `(-1)` *at* `endTs`*, aggregated into a time-indexed collection; evaluating at time* `t` *sums all contributions up to* `t`*. This mirrors Materialize queries that filter by* `mz_logical_timestamp()` *to report currently active records.*

Example 4.6 (RxJS-style sliding window and moving-average service)**.** *Input collection* `samples : (StreamId, Timestamp) × float`*. For each* `StreamId`*, the service maintains a bounded buffer of the last* `N` *samples or samples within the last* `T` *units of time, along with a running sum. On insertion of a new sample, it adds the value to the buffer and sum; periodically (or on each insert), it evicts samples older than the configured window (by count or time), subtracting their values from the sum. A view* `movingAvg : StreamId → float` *returns* `sum / bufferSize` *for each stream, replicating RxJS sliding-window moving-average operators in a Skip-style service.*

Example 4.7 (Text input with clear as window delimiter service (Fran/Fruit-style FRP))**.** *Input collections: (i)* `keystrokes : InputId × Char` *for character input events; and (ii)* `clears : InputId × unit` *for explicit clear actions. For each* `InputId`*, the service maintains current text as a string accumulator. On keystroke events, it appends characters to the current text; on a clear event, it resets the text to the empty string, effectively starting a new logical window of aggregation. The view* `currentText : InputId → string` *exposes the text since the last clear, mirroring FRP examples where accumulation is reset by a "clear" signal.*

5 History and Ordered-State Patterns

Example 5.1 (Elm-style undo/redo history service)**.** *Model an application state type* `AppState` *and an input collection* `actions : Unit × Action`*, where* `Action` *encodes user commands (draw, erase, move, etc.). The service maintains a single-key resource* `History : Unit → (past : array<AppState>, present : AppState, future : array<AppState>)`*. On each new* `Action` *applied to* `present`*, the service: (i) appends the old* `present` *to* `past`*, (ii) discards any states in* `future`*, and (iii) computes the new* `present` *by applying the action. Separate control inputs* `undo` *and* `redo` *move the focus backward or forward in the history by shifting one element between* `past`*,* `present`*, and* `future`*, providing time-travel semantics analogous to Elm architecture examples.*

Example 5.2 (Redux-like time-travel state service)**.** *Define a base collection* `commands : Unit × Command` *and a service resource* `Timeline : Unit → (past : array<State>, present : State, future : array<State>)` *for some application* `State` *type. The service offers three operations modeled as updates: (i)* applyCommand*: given a* `Command`*, computes a new* `present` *from the old one, pushes the old* `present` *onto* `past`*, and clears* `future`*; (ii)* undo*: when* `past` *is non-empty, pops the last state from* `past`*, pushes the current* `present` *onto* `future`*, and sets* `present` *to the popped state; (iii)* redo*: symmetrically moves one state from* `future` *back to* `present`*, pushing the old* `present` *onto* `past`*. This mirrors Redux undo/redo recipes where the store tracks a linear history of states.*

Example 5.3 (Svelte-style undoable store service)**.** *Consider an input collection* `updates : Unit × Update`*, where* `Update` *transforms an* `AppState`*. The service keeps a resource* `Undoable : Unit → (history : array<AppState>, index : int)`*, where* `history` *is a non-empty array of snapshots and* `index` *is the current position. On a new update, it: (i) truncates* `history` *to elements* `[0..index]` *(discarding any redo states), (ii) computes a new state by applying the update to* `history[index]`*, appends it to* `history`*, and (iii) sets* `index := index + 1`*. On* `undo`*, when* `index > 0` *it decrements* `index`*; on* `redo`*, when* `index < |history|-1` *it increments* `index`*. The* `currentState` *view is simply* `history[index]`*, matching Svelte undoable store patterns.*

Example 5.4 (FRP-style resettable accumulator service)**.** *Input consists of two collections:* `events : KeyId × Event` *and* `resets : KeyId × unit`*. For each* `KeyId`*, the service maintains a state* `acc : Accumulator` *(e.g. a running text, count, or other summary) and a* `lastResetTime` *or reset epoch marker. On add of an* `events` *entry for key* `k`*, it updates* `acc[k]` *by folding in the event (e.g. appending text or incrementing a counter); on add of a* `resets` *entry for* `k`*, it resets* `acc[k]` *to its initial value. This yields per-key aggregations over epochs separated by reset events, analogous to FRP text-input-with-clear examples where state is accumulated between explicit clears.*

6 Graph and Relational Incremental Maintenance

Example 6.1 (DBToaster-style incremental SQL view service)**.** *Input base collections correspond to relational tables, for example* `Orders(orderId, customerId, amount)` *and* `Customers(customerId, region)`*. Define a materialized view* `RegionTotals(region → Money)` *that reflects* `SELECT region, SUM(amount) FROM Orders JOIN Customers USING (customerId) GROUP BY region`*. The service maintains: (i) an intermediate keyed collection* `OrderContrib(customerId → Money)` *equal to the sum of* `amount` *per customer; and (ii) the final view* `RegionTotals` *as the sum over* `OrderContrib` *joined with* `Customers`*. On insert of an* `Orders` *row, it increments* `OrderContrib[customerId]` *by* `amount` *and then increments* `RegionTotals[region(customerId)]` *accordingly; on delete, it subtracts the same contributions. This captures the DBToaster idea of maintaining delta views and updating aggregates via plus/minus operations on precomputed partial results, rather than recomputing full joins.*

Example 6.2 (F-IVM-style ring-based analytics service)**.** *Consider a log-processing backend with base collection* `Events(key, payload)`*, where* `payload` *is an element of a user-chosen ring (e.g. numeric sums/products for counts, or more complex structures for ML gradients). Define derived views by interpreting SQL-like queries as expressions over the payload ring: for example, a view* `KeyStats(key)` *whose payload is maintained by ring addition and multiplication as events join or aggregate. On insert of an event, the service adds the event's payload into the appropriate keys; on delete, it subtracts the payload using the ring's additive inverse. This mirrors F-IVM's approach of treating query maintenance as updates in a factorized payload domain with well-defined add/remove operations.*

Example 6.3 (Dynamic acyclic join (Yannakakis-inspired) service)**.** *Suppose base collections* `R(A,B)`*,* `S(B,C)`*, and* `T(C,D)` *participate in an acyclic join query* `Q(A,B,C,D) = R JOIN S JOIN T`*. The service maintains: (i) semi-join filtered projections such as* `R'` *where only tuples with a matching* `B` *in* `S` *are stored; (ii) a join index mapping join-key combinations (e.g.* `(B,C)`*) to participating tuples; and (iii) a materialized view* `Q` *or aggregate over* `Q`*. On insert of a tuple into one relation (say* `R`*), the service traverses the join tree: it finds matching tuples in* `S`*, then in* `T`*, and inserts the resulting joined tuples into* `Q`*; deletions remove just the joined tuples involving the deleted base tuple. This*

specification captures Dynamic Yannakakis' coordinated delta propagation along a join tree without recomputing the entire join.

Example 6.4 (Counting and DRed-style materialized view service)**.** *Given a base relation* `R` *and a derived view defined by a recursive or non-recursive rule (e.g. reachability or a multi-join query), the service maintains: (i) for each derived tuple* `t`*, a count of how many derivations (proofs) support* `t`*; and (ii) the materialized view containing exactly those tuples with positive counts. On insertion of a base fact, the system derives new tuples according to the rules and increments their counts; on deletion, it decrements counts (the Counting algorithm). If a count drops to zero, the tuple is removed from the view; in DRed-style handling of recursion, the system may re-derive some tuples using the remaining facts to ensure no reachable tuples are lost. This combines algebraic inverses (for counts) with selective recomputation.*

Example 6.5 (Differential dataflow / DBSP-style weighted collections)**.** *Model a collection as a mapping* `Key` $\to$ `(Value, weight` $\in \mathbb{Z}$`)`*, where each base update is encoded as a small multiset of weighted records (e.g. +1 for insertion, -1 for deletion). Define a service that maintains one or more derived collections (joins, group-bys, filters) by algebraically combining and canceling these weights along a dataflow graph. Each operator (e.g. join, map, reduce) specifies how to transform input weights into output weights; for example, a group-by sum view computes, per key, the weighted sum of contributions. The service's update procedure simply applies incoming weighted updates and recomputes affected downstream weights, removing any records whose cumulative weight becomes zero.*

Example 6.6 (Unacknowledged alerts as a streaming anti-join)**.** *Base collections* `Alerts(alertId, userId, severity, payload)` *and* `Acks(alertId, ackTime)` *model alert events and acknowledgments. Define a derived view* `Outstanding(alertId` $\to$ `Alert)` *containing exactly those alerts that have not yet been acknowledged, corresponding to the relational query* `Alerts LEFT ANTI JOIN Acks USING (alertId)` *or* `SELECT * FROM Alerts a WHERE NOT EXISTS (SELECT 1 FROM Acks k WHERE k.alertId = a.alertId)`*. On insertion of an alert with id* `a`*, if no acknowledgment exists for* `a`*, the service inserts* `Alerts[a]` *into* `Outstanding`*; on deletion of an alert, it removes* `Outstanding[a]` *if present. On insertion of an acknowledgment for* `a`*, it removes* `Outstanding[a]` *(if the corresponding alert still exists); if acknowledgments themselves can be deleted, removing an ack for* `a` *causes the service to re-insert* `Alerts[a]` *into* `Outstanding`*. This captures a common "unmatched entries" pattern in streaming systems, where the view is maintained as a continuously updated anti-join between a base relation of open items and a relation of completion events.*

Example 6.7 (Foreign-key violation and orphan detection via set difference)**.** *Consider base collections* `Parents(parentId, ...)` *and* `Children(childId, parentId, ...)` *representing a referential-integrity relationship. Define a derived view* `Orphans(childId` $\to$ `Child)` *whose keys are exactly those children whose* `parentId` *is not present in* `Parents`*, corresponding to the set difference* `Children` $\setminus$ `(Children JOIN Parents)` *or the query* `SELECT * FROM Children c WHERE NOT EXISTS (SELECT 1 FROM Parents p WHERE p.parentId = c.parentId)`*. On insertion of a child* `c`*, if no matching parent exists, the service inserts* `c` *into* `Orphans`*; on insertion of a parent* `p`*, it scans children with* `parentId = p.parentId` *and removes them from* `Orphans`*. Similarly, deleting a parent* `p` *adds all remaining children with* `parentId = p.parentId` *to* `Orphans`*, while deleting a child removes it from* `Orphans` *if present. This specifies a streaming foreign-key-violation monitor as a maintained set difference between base collections, a pattern commonly supported via anti-join or* `NOT EXISTS` *in incremental view-maintenance systems.*

Example 6.8 (Incremental graph metrics service (Ingress, GraphBolt-style))**.** *Input collection* `Edges : (src : NodeId, dst : NodeId)` *and optionally per-node attributes. The service maintains per-node views such as: (i)* `degree : NodeId → int`*, counting incident edges via a per-node count reducer; (ii)* `rank : NodeId → float`*, where each node's rank is the sum of neighbor contributions (e.g. PageRank-style updates) maintained by per-node reducers over incoming edges; and (iii) neighborhood summaries (e.g. average neighbor attribute) using enriched-state reducers per node. On edge insert, the service updates the relevant nodes' accumulators (e.g. increment degree for both endpoints, add contributions to* `rank`*); on edge delete, it applies inverse updates. This specification reflects vertex-centric systems where user-defined inverse functions or algebraic structure enable efficient incremental graph metrics.*

Example 6.9 (Iterative graph algorithms with fixpoints)**.** *For algorithms such as BFS, single-source shortest paths, or label propagation, define: (i) base collections* `Edges(src, dst, weight?)` *and* `InitialSeeds : NodeId` *(e.g. starting nodes); and (ii) a derived per-node collection* `State : NodeId → Value` *(e.g. distance, label) updated iteratively. Each iteration applies local reducer-like updates to* `State` *based on neighbors (e.g. new distance is* `min(current, neighborDistance + weight)`*), but the global algorithm runs until a fixpoint (no state changes) is reached. The service specification therefore includes both the local update rule (a reducer per node) and a fixpoint scheduler that repeatedly applies updates and propagates changes until convergence, distinguishing it from single-step reducer services.*

7 Business Metrics and UI-Composed Summaries

Example 7.1 (Business KPIs in Skip-style services)**.** *A reactive backend for an e-commerce site exposes three KPI resources: (i)* categoryRevenue *with input collection* `sales : CategoryId × Sale`*, where each sale has an* `amount` *field; the resource is a collection* `CategoryId → Money` *maintained by mapping each sale to its category and reducing with a sum reducer over* `amount`*; (ii)* portfolioBySector *with input* `positions : SecurityId × Position`*, where* `Position` *includes* `sector` *and* `shares, price`*; a mapper emits* `(sector, shares * price)` *and a sum reducer computes total portfolio value per sector; (iii)* activeUserCounts *with input* `memberships : GroupId × UserId` *plus a global* `activeUsers : UserId × Status`*; the service defines collections* `ActivePerGroup : GroupId → Int` *(count of active members per group, via a count reducer) and* `ActiveGlobal : Unit → Int` *(total number of active users, by mapping all active users to a single key and counting).*

Example 7.2 (Streaming analytics dashboard service)**.** *A monitoring service ingests an input collection* `requests : ServiceId × RequestEvent`*, where each event records* `serviceId`*, a* `success : bool` *flag, and a timestamp. It exposes resources: (i)* requestThroughput *as* `ServiceId → Int`*, counting total requests per service with a per-key count reducer; (ii)* errorCounts *as* `ServiceId → Int`*, counting failed requests via a mapper that keeps only events with* `success = false` *and a count reducer; (iii)* errorRates *as* `ServiceId → Float`*, computed either by a derived view* `errorCounts / requestThroughput` *or by reducing into an enriched state* `(errors, total)` *per service and projecting the ratio. Optionally, a time-bucketed key (e.g.* `(ServiceId, HourBucket)`*) models per-interval KPIs without introducing an explicit window operator. A related pattern, studied in the anti-join results, is a dashboard of "unacknowledged alerts per service", which maintains a count or list of alerts with no matching acknowledgment event; this corresponds to a service-level aggregation over the streaming anti-join described in the graph/relational examples.*

Example 7.3 (UI-derived business metrics service)**.** *A customer-facing backend maintains input collections* `cartItems : UserId × CartItem` *(where* `CartItem` *has* `productId, quantity, unitPrice`*) and* `reviews : ProductId × Rating` *(where* `Rating` *has a numeric* `score`*). It exposes: (i)* cartTotals : UserId → Money*, mapping each cart item to* `(userId, quantity * unitPrice)` *and reducing with a sum reducer per user; (ii)* averageRating : ProductId → Float*, using an enriched-state reducer that maintains* `(sum, count)` *of scores per product and outputs* `sum / count`*. These resources correspond to totals and averages that front-end frameworks (Redux, Vue, Svelte, MobX) commonly compute in selectors or computed properties, but here are maintained reactively on the server.*

Example 7.4 (Composite metrics and conversion funnels)**.** *A product-analytics service ingests an input collection* `events : UserId × FunnelEvent`*, where each event has a* `stage` ∈ `{visit, signup, addToCart, purchase}` *and an optional* `timestamp`*. The service defines several derived collections: (i)* perStageCounts : Stage → Int*, counting how many distinct users ever reached each stage (e.g. via a set-valued reducer or a distinct-count pattern per stage); (ii)* funnelRatios : StagePair → Float*, computing ratios such as* `signups / visits` *or* `purchases / signups` *by combining the per-stage counts; (iii) optionally* timeBucketedFunnels : (Stage, TimeBucket) → Int *by including a time bucket in the key, enabling per-day or per-hour funnel analysis. These resources together specify a reactive "conversion funnel" service, where all KPIs update automatically as new events arrive.*