READ MODE · ANSWER FIRST

How partial upserts work across distributed databases

Compare MongoDB, DynamoDB, PostgreSQL, Cassandra and Redis strategies for partial upserts, field merges, conflicts and retries.

CHEAT SHEET · 6 ESSENTIAL IDEAS

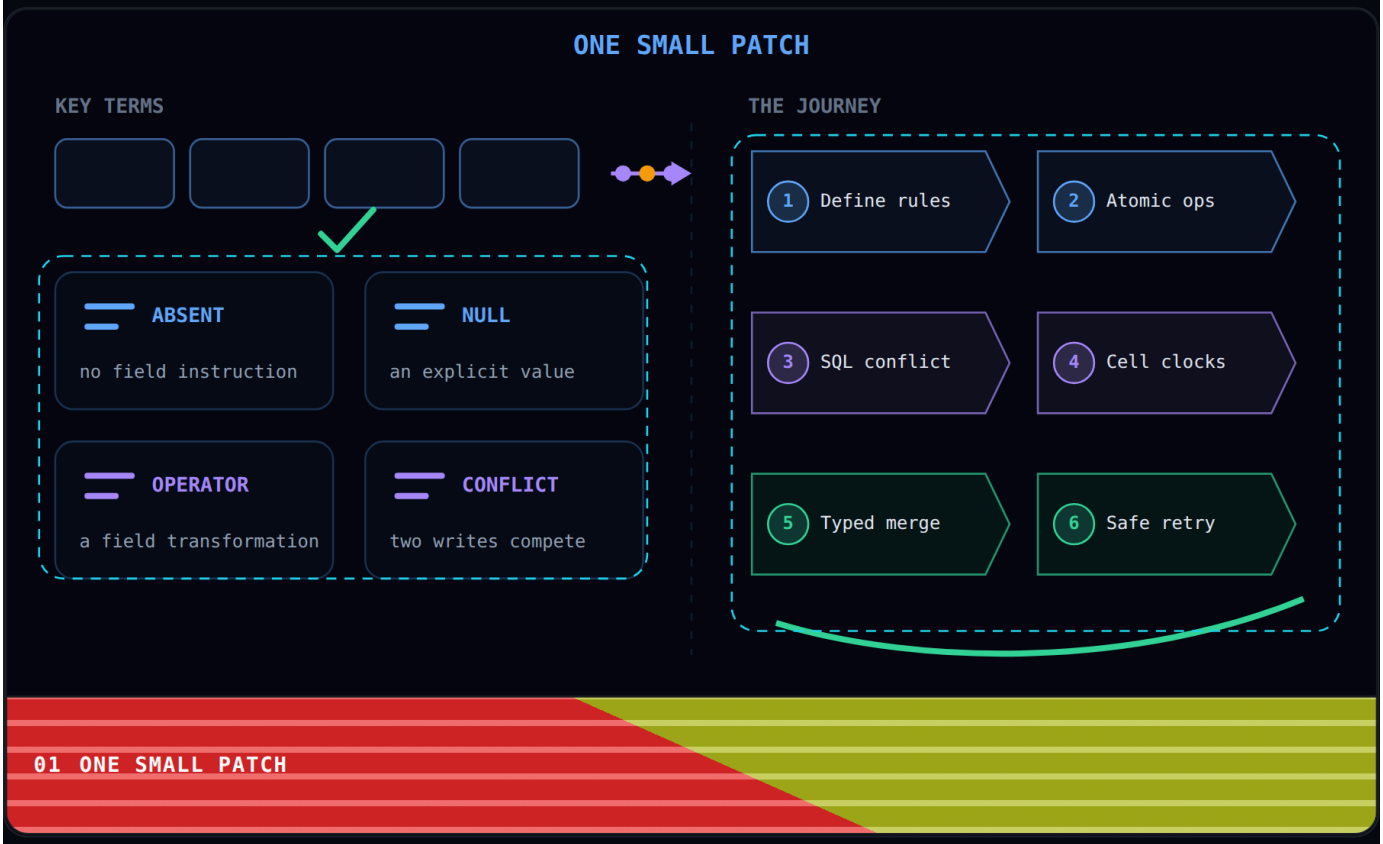
The whole story in 6 lines

Partial upserts are safe when merge meaning, conflict scope and retry behavior are designed together.

1. A partial upsert needs explicit rules for absent fields, nulls and every operator.
2. Document stores can apply field operators atomically inside one item or document.
3. SQL upserts make the merge an explicit conflict handler over old and proposed rows.
4. Sparse cell storage lets independent columns survive while same-cell writes still need a winner.
5. CRDT data types preserve operation intent so replicas can converge without one universal winner.
6. Non-idempotent operations need a request identity, condition or transaction before automatic retry.

STAGE 1 OF 8

One Small Patch



A customer row has four fields, but the next message carries only three changes. The untouched tier must survive while other fields merge differently.

Absent means no instruction arrived. Null is an explicit value, an operator describes a transformation and a conflict means two writes compete.

We will first define the patch, then compare atomic operators, SQL conflict code, cell timestamps, convergent types and safe retry handling.

The roadmap keeps the customer record constant while the merge engine changes. Let us begin by turning the vague word merge into exact field rules.

STAGE 2 OF 8

Define the Merge Contract

02 DEFINE THE MERGE CONTRACT

We begin with one stored customer row. City is Delhi, points equal ten, tags contain early and tier remains silver. Each cell can now receive its own explicit instruction.

The incoming payload is not a replacement row. It says overwrite city, increment points by five and append vip to tags. These verbs matter more than the payload size.

Tier is missing from the payload, while no operator targets it. Missing data and explicit deletion must never be confused. Should the database erase silver or carry it forward?

PAUSE AND PREDICT

What should happen to the absent tier field?

- Keep silver
- Erase the field

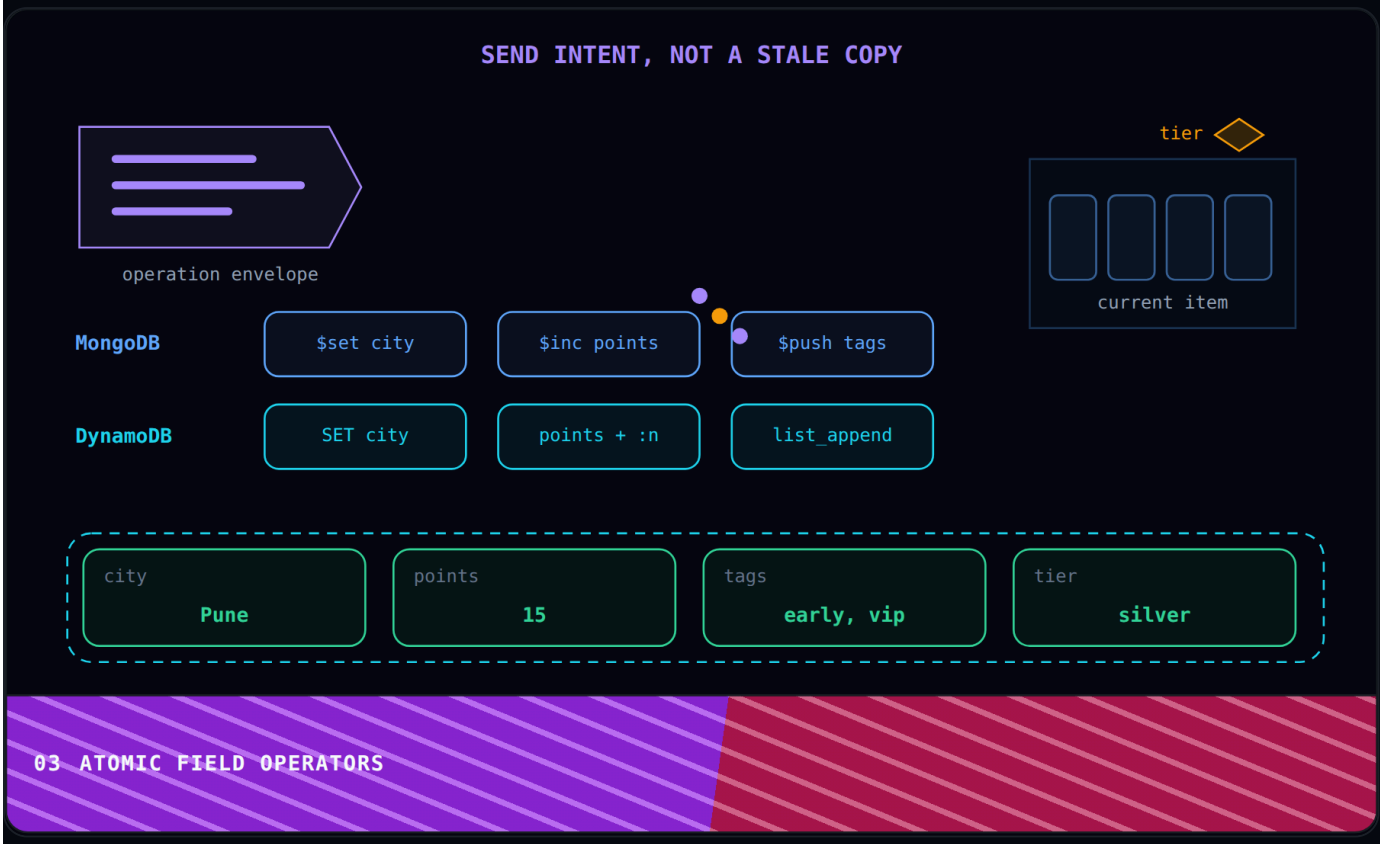
Skip this check

The operators now land on separate cells. Pune replaces Delhi, points become fifteen, vip joins the tag list and silver passes through unchanged. One sparse payload creates four deliberate outcomes.

A partial upsert is therefore a field-level program with an explicit absence rule. Null would need another rule because it arrived explicitly. Next, we will let document databases execute that program atomically.

STAGE 3 OF 8

Atomic Field Operators



03 ATOMIC FIELD OPERATORS

Now that the contract is clear, the client can send operations instead of reading the row and rebuilding it locally. That removes one stale snapshot from the write path.

MongoDB exposes operators such as set, inc and push. DynamoDB UpdateItem uses SET, arithmetic, list_append, ADD and conditional expressions. Both dialects describe intent near the current record.

Another writer changes tier while our patch is traveling. Our payload still names only three target paths. Will a sparse server-side mutation overwrite that unrelated field?

PAUSE AND PREDICT

Does this sparse mutation need to replace the unrelated tier field?

- **No, tier stays separate**
- **Yes, replace the item**

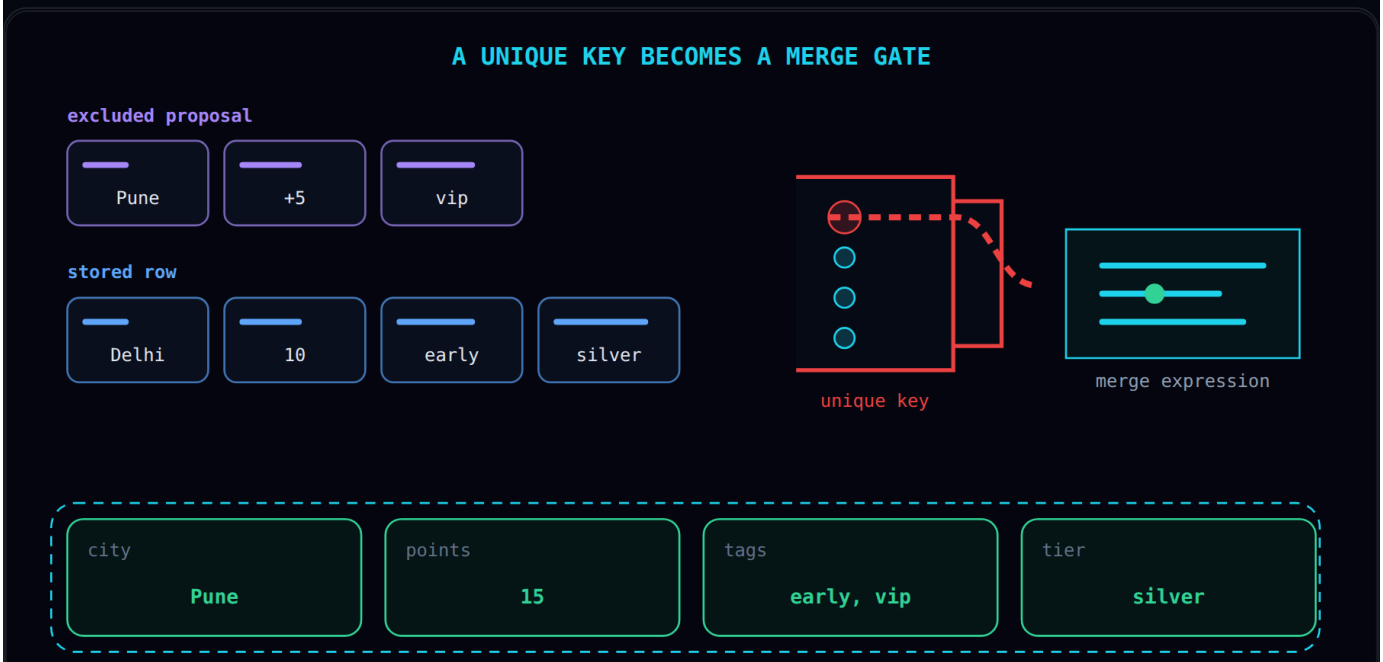
Skip this check

Both operator streams now enter the same current item and produce the merged row. The client never ships a stale full copy back. Unrelated attributes remain outside the mutation.

This strategy is concise and atomic within its documented item boundary, but operator syntax is engine-specific. Cross-item business invariants need a wider database transaction mechanism across records. Next, we will express the merge in SQL.

STAGE 4 OF 8

SQL Conflict Programs



04 SQL CONFLICT PROGRAMS

We just sent operator objects to document stores. PostgreSQL starts from an INSERT and lets a unique index detect the existing account key. That index acts as the conflict referee.

Inside ON CONFLICT DO UPDATE, the table alias exposes stored values. The special excluded row exposes values that the insert proposed. The merge expression can read from both sources.

The handler can combine both inputs with normal SQL expressions. Omitted INSERT columns already received defaults or null before this branch. Which source should supply the untouched tier when the proposal carries none?

PAUSE AND PREDICT

Which source should preserve the untouched tier?

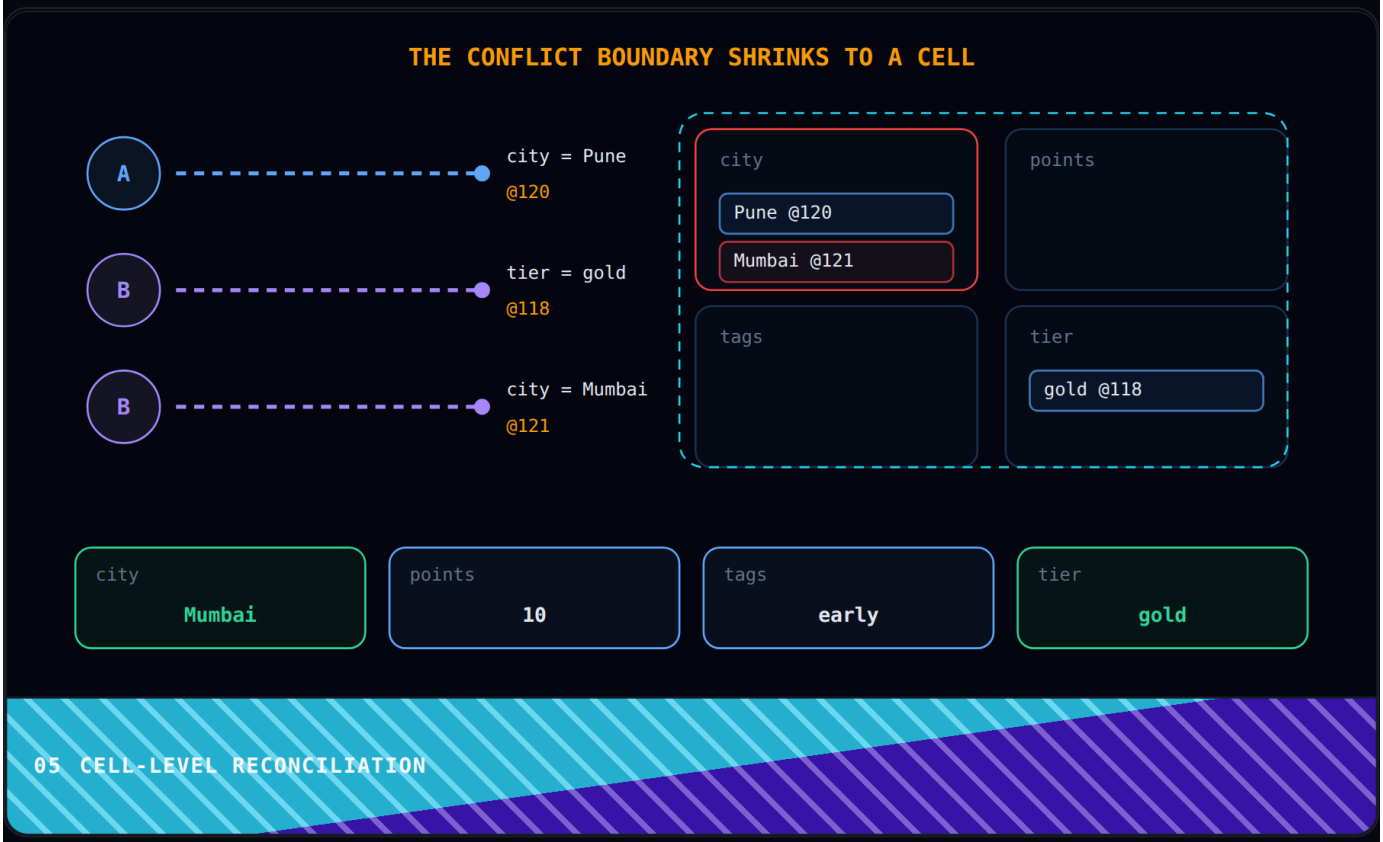
- The stored row
- The proposed row

Skip this check

The conflict branch now computes city from proposed data, points from stored plus delta, tags from concatenation and tier from the stored row. Every field rule is executable SQL.

PostgreSQL guarantees one atomic INSERT or UPDATE outcome, but the developer writes the merge expression. The unique key defines which row can conflict. Next, we will move below rows into individual cells.

Cell-Level Reconciliation



05 CELL-LEVEL RECONCILIATION

Now that row-level strategies are familiar, let us inspect Cassandra. An UPDATE names a primary key and writes only selected non-key columns. A missing row is created without a preliminary check.

Each value carries a write timestamp. One writer can update city while another updates tier, so those cells do not replace each other. Their conflict scopes never overlap.

A later writer also changes city to Mumbai with timestamp 121. This candidate targets the exact cell already holding Pune. Can both city values occupy the same logical cell?

PAUSE AND PREDICT

Which city value survives reconciliation?

- **Pune at 120**
- **Mumbai at 121**
- **Both city values**

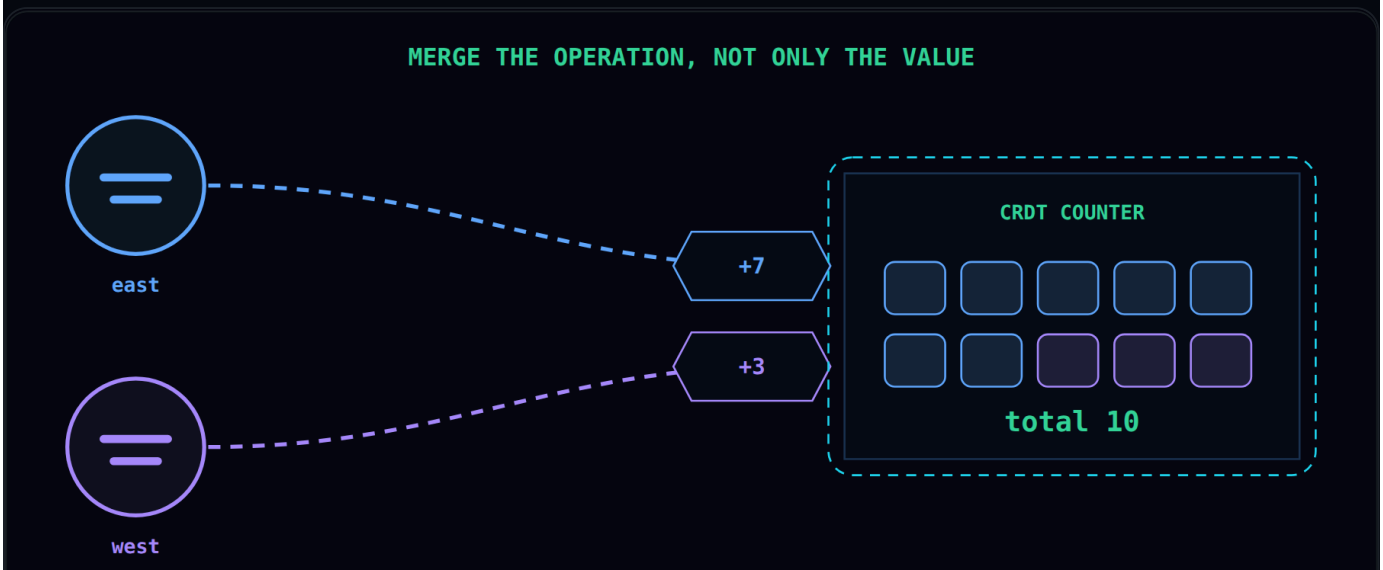
Skip this check

Reconciliation keeps gold from the tier write and Mumbai from the later city write. The winner decision happened per cell, not per row. Points and tags remain untouched.

Sparse cells reduce accidental whole-row loss, but clock order still decides same-cell conflicts. Client-supplied time therefore needs careful discipline. Next, we will preserve operations instead of selecting

one value.

Operation-Aware Data Types



06 OPERATION-AWARE DATA TYPES

We just saw timestamps pick one value for a disputed cell. Active-Active systems face a harder case because distant regions can both accept writes. Network delay can make those writes concurrent.

Redis Active-Active uses CRDT behavior by type. Counter operations retain their increments, while ordinary string overwrites use last-write-wins conflict handling. One rule cannot fit both meanings.

East increments by seven and west increments by three before synchronization. Both operations are valid local work. If one whole number wins, what useful intent disappears?

PAUSE AND PREDICT

What should a convergent counter preserve?

- **Both increments**
- **Only the latest value**

Skip this check

The two operation paths now join into ten rather than discarding one side. A set or map would use its own type-specific convergence rule. The data type carries conflict meaning.

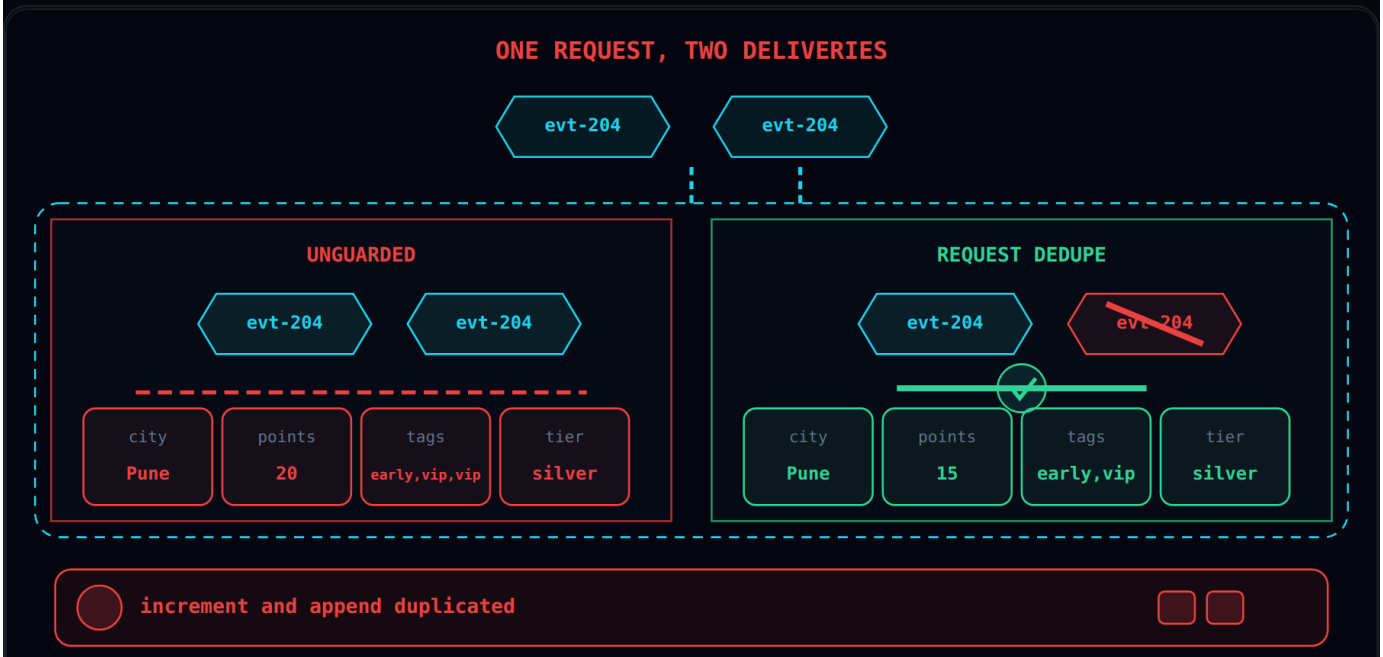
Operation-aware types trade extra metadata and specialized semantics for automatic convergence. Applications must accept the built-in rule for that type. Next, we will handle a different duplicate source created by retries.

STAGE 7 OF 8

Retries Change the Algebra

★ If you remember one thing · A repeated delivery leaves overwrite stable but duplicates increment and append unless the request is deduplicated.

Try it in Watch mode. Keep one logical change even when request evt-204 arrives twice.



07 RETRIES CHANGE THE ALGEBRA

We have compared database merge strategies, but distributed clients also retry after timeouts. Request evt-204 now arrives twice with identical field operations. The first network response was simply uncertain.

Repeating overwrite city to Pune produces Pune again. Repeating increment adds five again, while repeating list append creates a second vip entry.

Both deliveries carry the same request identity. Without a remembered guard, can the database know that the second mutation is a retry?

PAUSE AND PREDICT

Can identical field operators reveal a retry by themselves?

- No, identity is separate
- Yes, values are enough

```
Skip this check
```

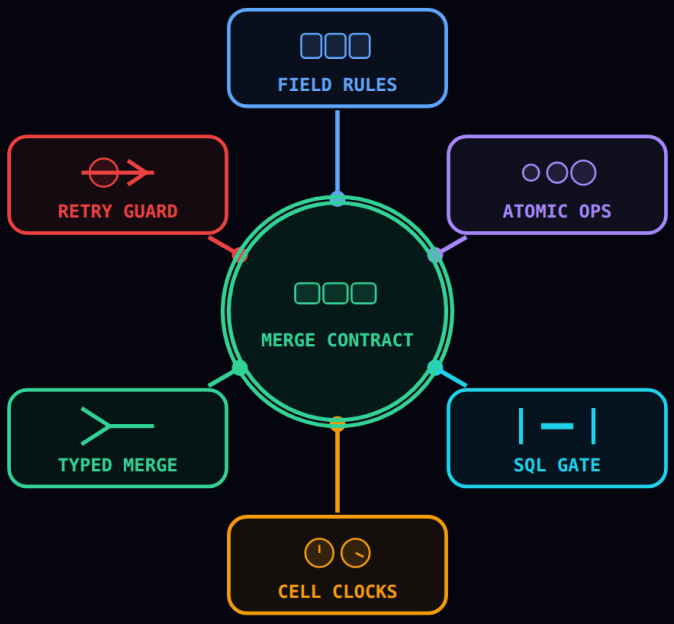
The lanes now diverge. Unguarded delivery ends at twenty points with two vip tags, while request deduplication ends at fifteen and one vip. City remains Pune in both lanes.

Switch the Retry guard control through both choices. Compare the number of applied request tokens, then leave the record with one logical change.

Safe retries combine merge operators with request identity, conditions or transactions. Now let us step back and choose the strategy that matches each field.

STAGE 8 OF 8

Choose the Merge Boundary



08 CHOOSE THE MERGE BOUNDARY

We started by treating the patch as a program. Absence preserved tier while overwrite, increment and append each transformed one field.

Then MongoDB and DynamoDB moved those operations into an atomic item mutation, avoiding a stale client-side reconstruction.

PostgreSQL exposed the stored and proposed rows to one conflict expression, making every merge choice explicit in SQL.

Cassandra showed a smaller conflict boundary. Different columns survived together, while timestamps selected one winner for the same cell.

CRDT-backed data types preserved compatible operation intent across regions, so concurrent counter increments could converge instead of competing.

Finally, retries revealed that overwrite is naturally stable, while increment and append need request identity or another deduplication guard.

The complete map now shares one center. Define field meaning first, choose the conflict boundary next and design retry safety before production traffic arrives.