Author: Madansingh Deora
Date: 22 Nov 2025
Phamtom Duplicate

Tech Stack: Java 17, Spring Boot, PostgreSQL, Docker, REST APIs

1)Root Cause Analysis

a. Client / network retry behavior

- Client retries (e.g., network timeout) send the same logical operation without a stable idempotency key, or with a different generated ID each retry. Server treats them as different requests and inserts multiple records.

b. Non-atomic "check then insert" logic (race conditions)

- Two concurrent requests both SELECT to check if an item exists, both see "not found", then both INSERT. If the application doesn't rely on an atomic DB constraint, duplicates slip in.

c. Unique constraint gaps and DB semantics

- Unique constraints can be bypassed inadvertently:
    - Nullable unique columns: some DBs allow multiple NULLs, so a nullable "unique id" can allow duplicates.
    - Case-sensitivity / collation differences cause distinct values (User vs user) to pass uniqueness checks.
    - Partial/filtered indexes that don't include all rows.
- Schema drift during migrations: different environments have different indexes.

d. Replication / eventual consistency

- Multi-region writes or async replication (AP mode) can create conflicting inserts on different nodes, and conflict resolution may create duplicates or conflicting rows.

e. Unique ID collisions or mis-generation

- Poor ID generation (non-random small space, fixed seeds) can produce collisions or duplicated IDs across systems (rare with UUIDv4 but possible with bad custom schemes).

f. Batch imports / data migrations

- Import scripts or manual admin processes can insert duplicates if deduplication logic is missing or inconsistent.

g. Multiple uniqueness dimensions

- The "unique ID" enforced in DB might not represent the true logical uniqueness (e.g., unique on external_id but duplicates created because external_id was optional or generated inconsistently).

h. Worker/process crashes and retries

- Background job crashes after creating a record but before marking the job complete; a retry re-creates the same logical entity.

2)Verification Plan

- Track duplicate_count, unique_constraint_violations, insert_conflicts, and idempotency_key_conflicts.
- Expose per-endpoint duplicate rates and per-tenant/region stats. Alert when duplicate rate > threshold.

Logging & tracing

- Attach request-level idempotency_key, client-request-id, and trace-id to logs. Correlate duplicates by these IDs.
- Log INSERT failures and unique-constraint exceptions with full context (payload hash, keys, timestamps).

Immediate DB triggers / audit table

- Add a lightweight trigger that writes all inserts into an audit table; a downstream process scans for same logical key appearing >1 within a time window and raises alerts.

CDC / Stream processing

- Use CDC (Debezium / cloud CDC) to stream DB changes into a stream processor which deduplicates by logical key and emits duplicates to a monitoring topic in real time.

Checksums / Content fingerprint

- For records that should be unique by content, compute a deterministic hash (e.g., SHA256 over canonicalized payload) and index it. Monitor multiple rows with same hash.

Dashboards & anomaly detection

- Dashboards showing duplicates over time, broken down by source (API key, region, client version). Use simple anomaly detection to surface sudden spikes.

3) Prevention Strategy

Definitive rule: Enforce uniqueness at the data layer AND make application logic idempotent.

Application-only checks are necessary but not sufficient.

3.1 Idempotency-key pattern (recommended for API operations)

- Client supplies Idempotency-Key (UUID, GUID) per logical operation.

- Server stores (idempotency_key, request_hash, response_blob, status, created_at) in an idempotency table with a UNIQUE constraint on idempotency_key.
- Handler logic:
    1. Try to insert row (idempotency_key, status='in_progress') (atomic).
    2. If insert succeeds → process operation; on completion update row with status='done' and response_blob.
    3. If insert fails (unique violation) → SELECT the row:
        - if status='done' return stored response_blob (same client should get same response).
        - if status='in_progress' either wait/return 409 or return a specific response indicating processing.
- Use TTL + GC for idempotency rows (e.g., keep result for 24–72 hours).
- Example SQL to create table:

```
CREATE TABLE idempotency_keys (
  idempotency_key TEXT PRIMARY KEY,
  request_hash TEXT,
  response JSONB,
  status TEXT NOT NULL,
  created_at TIMESTAMP WITH TIME ZONE DEFAULT now(),
  updated_at TIMESTAMP WITH TIME ZONE DEFAULT now()
);
```

-

3.2 Atomic DB operations (single-statement upserts)

- Prefer INSERT ... ON CONFLICT DO NOTHING/UPDATE patterns or MERGE to make check-and-insert atomic:

INSERT INTO orders (order_id, payload_hash, ...)

VALUES ($id, $hash, ...)

ON CONFLICT (order_id) DO NOTHING

RETURNING *;

This avoids race windows where multiple processes do SELECT then INSERT.

3.3 Unique constraint design

- Make the true uniqueness explicit:
    - Unique on idempotency_key or the *canonical* payload hash.
    - Composite keys when appropriate: UNIQUE (external_id, source_system, region).
    - Make columns NOT NULL where uniqueness is required—avoid nullable keys that allow multiple NULLs.

    - Ensure collation/case-sensitivity matches logical uniqueness (e.g., use LOWER() in index if case-insensitive).

3.4 Isolation and concurrency control

- Use higher isolation (SERIALIZABLE) only where appropriate—be aware of performance costs.
- Use SELECT ... FOR UPDATE if you need to serialize access to a row.
- Prefer optimistic approach with unique constraints + retry on conflict.

One-line takeaways

- **Prevention = DB atomicity + idempotency keys + canonical keys.**
- **Detection = telemetry + CDC/streaming + alerts.**
- **Enforcement = atomic insert of idempotency key, return stored response on conflict.**