Distributed Counters

Here's a clean, production-ready way to do a **distributed counter** in NestJS microservices using **Redis (sharded counters)**, **Kafka (events + batching)**, and **Prisma (durable storage)**. I'll show you two layers:

1. fast, write-heavy **Redis sharded counters**
2. durable, eventually consistent **DB aggregates** updated by a **Kafka consumer**

Use both. Reads hit Redis; DB is your source of truth for analytics/backups.

---

1) Data model (Prisma)

``

```
model VideoLikeAggregate {
    videoId    String   @id
    likeCount  Int      @default(0)
    updatedAt  DateTime @updatedAt
}
```

You don't store every click here in real time. You'll periodically upsert aggregates.

---

2) Key design (Redis): Intra Entity Sharding

For **each videoId**, we maintain **N shards** (say 64). Each shard is just a Redis key holding an integer counter.

So each video has a set of 64 counters that counts the number of likes for that video independently.

Let say a video has id: **video123**, so corresponding to this video we have these counters:

`vc:video123:0;vc:video123:1;vc:video123:2;vc:video123:3; ... vc:video123:63`

i.e, We have 64 likes counter shard for each video...

🔄 Flow Recap

1. **When a user likes a video**
   - Pick one shard index (0–63) (at random, and each shard is equally likely to be picked).
   - Increment only that shard's counter.
     Example:
     `vc:video123:37 → INCRBY 1`
2. **When you need the total likes for the video**
   - Read all 64 shard counters:
     `vc:video123:0 … vc:video123:63`
   - Sum them → gives the total likes.
     Example:
     `vc:video123:0 = 100 vc:video123:1 = 150 ... vc:video123:63 = 90 --------- ----------- total = sum = 6423 likes`

❓ Why bother with shards?

- If you only had **one counter key per video**, all increments would go to that key.
- Under high load (say **100k likes/sec** on a trending video), Redis would see all writes hitting one key → **hotspot**. (as a key in redis is present on a single node only, so that node will be overloaded with so many increment requests)
- Shards spread the writes across 64 different keys → reduces contention & scales horizontally.
- As these 64 keys for a given video will be spread across our redis cluster.

🧮 Overhead

- Reads: a bit more expensive since you `MGET` 64 values and sum them. But Redis is fast, and summing 64 ints is trivial compared to handling millions of writes.
- Writes: almost as cheap as a normal `INCR`, since you only increment one shard.

⚡So yes:

- **Per video → 64 counters, distributed across our redis cluster.**
- **Total likes = sum of those counters.**

NOTE:

If you use **Redis Cluster** but only store a **single counter key per video** (like `vc:{videoId}`):

- That one key is assigned to **a single hash slot**.
- That hash slot belongs to **one Redis node** in the cluster.
- All `INCRBY` for that video hammer the same node.
- So the "hot video" becomes a **hotspot** and can easily saturate CPU or network on that node, even though the rest of the cluster is idle.

Why this bottlenecks

- **Redis Cluster doesn't auto-distribute a single key's load — it just distributes different keys across nodes.**
- So if you don't shard **within the entity (video)**, the cluster won't help you under extreme write load for a single hot key.

Example

Say you have a cluster with 3 nodes:

- `vc:video1` → goes to Node A
- `vc:video2` → goes to Node B
- `vc:video3` → goes to Node C

If traffic is **evenly distributed across videos**, all good ✅.
But if **video1 is trending**, 90% of traffic hammers Node A → boom 💥.

How sharding fixes it

Instead of one key per video, you keep **multiple shards per video**:

`vc:video1:0 → Node A vc:video1:1 → Node B vc:video1:2 → Node C ...`

Now:

- Likes for video1 are distributed across multiple nodes.

- Each node handles a slice of the traffic.
- Reads just sum up those shards.

This is why **intra-entity sharding** (like 64 counters per video) is necessary if you expect skewed traffic (e.g., one viral video with millions of likes/sec).

✅ **So yes:** if you don't shard, a single hot key will bottleneck a Redis cluster. Sharding lets you spread the load *within* that entity.

---

3) NestJS packages

- `@nestjs/microservices` (Kafka)
- `@nestjs/axios` (if needed)
- `ioredis` (client)
- `@prisma/client` + `prisma`
- `kafkajs` (indirect via Nest) or use Nest's Kafka transport.

---

4) Shared counter util (hash → shard)

```
// libs/counters/src/sharded-hash.ts
import * as crypto from 'crypto';

export function shardFor(id: string, shards = 64): number {
  const h = crypto.createHash('xxhash64' as any) // if not available, use
murmur or sha256
                  .update(id)
                  .digest();
  // fallback: use sha256 and read first 4 bytes as uint32
  const num = h.readUInt32BE ? h.readUInt32BE(0) :
crypto.createHash('sha256').update(id).digest().readUInt32BE(0);
  return num % shards;
}
```

(If `xxhash64` isn't available, the sha256 fallback is fine.)

🎯 Goal

We need to pick **which shard (0–63)** gets incremented for a like/unlike event.
That decision matters because it determines:

- **Load distribution** across shards/nodes.
- **Correctness** (avoiding double-counting or missed counts).

Two strategies

1. Shard by `videoId` only

```
const shard = hash(videoId) % 64;
```

✅ Effect:

- All likes for a video always go to the **same shard**.
- But that's useless — it's basically no sharding at all, since a hot video still hammers one shard → hotspot.
- Good for consistency, bad for scalability.

---

2. Shard by `(videoId + userId)`

```
const shard = hash(videoId + ':' + userId) % 64;
```

✅ Effect:

- Each `(video, user)` pair is mapped to one of 64 shards.
- For a very popular video, different users' likes will spread across different shards.
- This removes the **hot key problem**, because one viral video's traffic is now evenly split across 64 Redis keys (and across nodes in a cluster).
- Reads require summing all shards.

This is the approach I coded in that snippet.

🔑 Why not just random shard?

- If you randomly pick a shard on each like, the same user could end up incrementing multiple shards → double counting.
- By hashing `(videoId + userId)`, we guarantee **same user always maps to the same shard**.

- **Combined with the Redis SET for dedupe (`vlu:{videoId}`), we make sure a user's like increments only once.**

🧮 Example

Say `videoId = v123` , `userId = u99` , `shards = 64

`hash("v123:u99") % 64 = 37`

So user `u99` 's like always increments `vc:v123:37` .
User `u77` might hash to shard `12` .
Over millions of users, load is evenly spread.

✅ Answer to your question

👉 We compute the **shard value (0–63) based on `(videoId + userId)`** , not just the videoId.
This ensures:

- Even distribution of writes for a hot video.
- Same user always maps to the same shard (so no double-counting).

---

5) Atomic Redis ops (Lua) for like/unlike

Why Lua? To guarantee: check dedupe + increment shard in **one atomic op**.

```
-- apps/likes/src/redis/lua.ts
export const LIKE_LUA = `
-- KEYS[1] = user likes set key (vlu:{likes}:{videoId})
-- KEYS[2] = user dislike set key (vlu:{dislikes}:{videoId})
-- KEYS[3] = counter shard key (vc:{likes}:{videoId}:{shard})
-- ARGV[1] = userId
-- return 1 if incremented, 0 if already liked
-- check if the user has already disliked this video or not
-- if the user disliked this video, then remove from dislike set
-- decrement the dislike counter
-- insert in the likes set, and return 1
local fromDisLike = redis.call('SREM', KEYS[2], ARGV[1])
if fromDisLike == 1 then
```

```
    redis.call('INCRBY', KEYS[3], -1)
end
local added = redis.call('SADD', KEYS[1], ARGV[1])
if added == 1 then
  redis.call('INCRBY', KEYS[3], 1)
  return 1
end
return 0
`;

export const UNLIKE_LUA = `
-- reverse: remove from set and decrement if it existed
local removed = redis.call('SREM', KEYS[1], ARGV[1])
if removed == 1 then
  redis.call('INCRBY', KEYS[3], -1)
  return 1
end
return 0
`;
```

```
-- apps/likes/src/redis/lua.ts
export const DISLIKE_LUA = `
-- KEYS[1] = user dislikes set key (vlu:{dislikes}:{videoId})
-- KEYS[2] = user likes set key (vlu:{likes}:{videoId})
-- KEYS[3] = counter shard key (vc:{dislikes}:{videoId}:{shard})
-- ARGV[1] = userId
-- return 1 if incremented, 0 if already disliked
-- check if the user has already liked this video or not
-- if the user liked this video, then remove from liked set
-- decrement the likes counter
-- insert in the dislikes set, and return 1
local fromLike = redis.call('SREM', KEYS[2], ARGV[1])
if fromLike == 1 then
    redis.call('INCRBY', KEYS[3], -1)
end
local added = redis.call('SADD', KEYS[1], ARGV[1])
if added == 1 then
  redis.call('INCRBY', KEYS[3], 1)
  return 1
end
return 0
`;

export const UNDISLIKE_LUA = `
```

```
-- reverse: remove from set and decrement if it existed
local removed = redis.call('SREM', KEYS[1], ARGV[1])
if removed == 1 then
  redis.call('INCRBY', KEYS[3], -1)
  return 1
end
return 0
`;
```

NOTE:

A mutual exclusivity is required so that a user cannot like/dislike at same time....

🔹 How to model it in Redis

Instead of only one set, you'll maintain **two sets per video**:

- `video:likes:{videoId}` → users who liked
- `video:dislikes:{videoId}` → users who disliked

And still keep **sharded counters** for scalability:

- `video:likes:count:{videoId}:{shard}`
- `video:dislikes:count:{videoId}:{shard}`

🔹 Operations

1. Like

- `SREM video:dislikes:{videoId} userId` (remove from dislikes if present)
- `SADD video:likes:{videoId} userId`
  - If return = `1` → increment likes counter shard
  - If user was removed from dislikes → decrement dislikes counter shard

2. Unlike

- `SREM video:likes:{videoId} userId`
  - If return = `1` → decrement likes counter shard

3. Dislike

- `SREM video:likes:{videoId} userId` (remove from likes if present)
- `SADD video:dislikes:{videoId} userId`
  - If return = `1` → increment dislikes counter shard
  - If user was removed from likes → decrement likes counter shard

4. Undislike

- `SREM video:dislikes:{videoId} userId`
  - If return = `1` → decrement dislikes counter shard

🔹 Why this works

- **No double likes/dislikes** → sets guarantee uniqueness.
- **No like + dislike at the same time** → explicit removal from opposite set.
- **Scalability** → counters are sharded; you don't need to scan sets for counts.
- **Durability** → you can periodically flush Redis sets to a DB for persistence.

---

6) Likes API (producer) – emits events and updates Redis fast

```
// apps/likes/src/likes.controller.ts
import { Body, Controller, Post } from '@nestjs/common';
import { LikesService } from './likes.service';

@UseFilters(LikeExceptionFilter)
@LikeServiceControllerMethods()
@Controller()
export class LikesController implements LikeServiceController {
  constructor(private readonly svc: LikesService) {}

  like(likeDto: LikeDto) {
    return this.svc.like(dto.videoId, dto.userId);
  }

  unlike(unlikeDto: UnlikeDto) {
    return this.svc.unlike(dto.videoId, dto.userId);
  }
```

```
  dislike(dislikeDto: LikeDto) {
    return this.svc.dislike(dto.videoId, dto.userId);
  }

  unDislike(unDislikeDto: UnDislikeDto) {
    return this.svc.unDislike(dto.videoId, dto.userId);
  }

}
```

```
// apps/likes/src/likes.service.ts
import { Injectable } from '@nestjs/common';
import Redis from 'ioredis';
import { KafkaService } from './messaging/kafka.service';
import { shardFor } from '@libs/counters/sharded-hash';
import { LIKE_LUA, UNLIKE_LUA } from './redis/lua';

const SHARDS = 64;

@Injectable()
export class LikesService {
  private likeSha?: string;
  private unlikeSha?: string;

  constructor(private readonly redis: Redis, private readonly kafka:
KafkaService) {}

  private userSetKey(videoId: string) {
    return `vlu:${videoId}`;
  }
  private shardKey(videoId: string, shard: number) {
    return `vc:${videoId}:${shard}`;
  }

  private async ensureScripts() {
    if (!this.likeSha) this.likeSha = await this.redis.script('LOAD',
LIKE_LUA);
    if (!this.unlikeSha) this.unlikeSha = await this.redis.script('LOAD',
UNLIKE_LUA);
  }

  async like(videoId: string, userId: string) {
    await this.ensureScripts();
    const shard = shardFor(`${videoId}:${userId}`, SHARDS); // spread write
hotness
```

```
    const res = await this.redis.evalsha(
      this.likeSha!,
      2,
      this.userSetKey(videoId),
      this.shardKey(videoId, shard),
      userId,
    );

    if (res === 1) {
      // fire-and-forget event for durable aggregation
      await this.kafka.emit('video.like', { videoId, userId, delta: +1, ts:
Date.now() });
    }
    return { applied: res === 1 };
  }

  async unlike(videoId: string, userId: string) {
    await this.ensureScripts();
    const shard = shardFor(`${videoId}:${userId}`, SHARDS);
    const res = await this.redis.evalsha(
      this.unlikeSha!,
      2,
      this.userSetKey(videoId),
      this.shardKey(videoId, shard),
      userId,
    );

    if (res === 1) {
      await this.kafka.emit('video.like', { videoId, userId, delta: -1, ts:
Date.now() });
    }
    return { applied: res === 1 };
  }

  // Fast read from Redis: sum shards
  async getCount(videoId: string) {
    const keys = Array.from({ length: SHARDS }, (_, i) =>
this.shardKey(videoId, i));
    const vals = await this.redis.mget(...keys);
    const total = vals.reduce((sum, v) => sum + (v ? parseInt(v, 10) : 0),
0);
    return { videoId, likeCount: total };
  }
}
```

7) Kafka wiring (producer + consumer)

Producer service (used above):

```
// apps/likes/src/messaging/kafka.service.ts
import { Injectable, OnModuleInit } from '@nestjs/common';
import { ClientKafka, Transport } from '@nestjs/microservices';

@Injectable()
export class KafkaService extends ClientKafka implements OnModuleInit {
  constructor() {
    super({
      transport: Transport.KAFKA,
      options: {
        client: { clientId: 'likes-api', brokers: ['kafka:9092'] },
        producer: { allowAutoTopicCreation: true },
      },
    } as any);
  }
  async onModuleInit() {
    await this.connect();
  }
  emit(topic: string, message: any) {
    return super.emit(topic, { value: JSON.stringify(message) });
  }
}
```

Aggregator microservice (consumer → Prisma upsert in batches):

```
// apps/likes-aggregator/src/main.ts
import { NestFactory } from '@nestjs/core';
import { MicroserviceOptions, Transport } from '@nestjs/microservices';
import { AppModule } from './app.module';

async function bootstrap() {
  const app = await NestFactory.createMicroservice<MicroserviceOptions>
(AppModule, {
    transport: Transport.KAFKA,
    options: {
      client: { clientId: 'likes-aggregator', brokers: ['kafka:9092'] },
      consumer: { groupId: 'likes-aggregator-g1' },
    },
  });
  await app.listen();
```

```
}
bootstrap();
```

```
// apps/likes-aggregator/src/likes.consumer.ts
import { Controller } from '@nestjs/common';
import { Ctx, KafkaContext, MessagePattern, Payload } from
'@nestjs/microservices';
import { PrismaClient } from '@prisma/client';

const prisma = new PrismaClient();

// in-memory batch per videoId (you can swap this with Redis streams if
desired)
const pending = new Map<string, number>();
let lastFlush = Date.now();

@Controller()
export class LikesConsumer {
  @MessagePattern('video.like')
  async onLike(@Payload() message: any, @Ctx() ctx: KafkaContext) {
    const { videoId, delta } = JSON.parse(message.value.toString());
    pending.set(videoId, (pending.get(videoId) || 0) + delta);

    const due = pending.size >= 500 || Date.now() - lastFlush > 2000;
    if (due) await this.flush();
  }

  private async flush() {
    const entries = Array.from(pending.entries());
    pending.clear();
    lastFlush = Date.now();

    // Upsert in a transaction
    await prisma.$transaction(
      entries.map(([videoId, delta]) =>
        prisma.videoLikeAggregate.upsert({
          where: { videoId },
          create: { videoId, likeCount: Math.max(0, delta) },
          update: { likeCount: { increment: delta } },
        }),
      ),
    );
  }
}
```

Using Redis Stream:

```
await this.redis.xadd(
  'video_likes_stream',
  '*',
  'videoId', videoId,
  'userId', userId,
  'delta', res === 1 ? '1' : '-1',
  'ts', Date.now().toString(),
);
```

Consumer:

```
XGROUP CREATE video_likes_stream likes_group $ MKSTREAM
```

```
import Redis from "ioredis";

const redis = new Redis();

async function startConsumer() {
  const consumerName = `likes_consumer_${process.pid}`;

  while (true) {
    // Block for up to 5 seconds waiting for new events
    const streams = await redis.xreadgroup(
      "GROUP", "likes_group", consumerName,
      "BLOCK", 5000,
      "COUNT", 10,
      "STREAMS", "video_likes_stream", ">"
    );

    if (!streams) continue;

    for (const [, messages] of streams) {
      for (const [id, fields] of messages) {
        const data: Record<string, string> = {};
        for (let i = 0; i < fields.length; i += 2) {
          data[fields[i]] = fields[i + 1];
        }

        console.log("Processing like event:", data);

        // TODO: update DB or forward to analytics pipeline
```

```
        // Mark as processed
        await redis.xack("video_likes_stream", "likes_group", id);
      }
    }
  }
}

startConsumer().catch(console.error);
```

Notes:

- Batching reduces DB write load massively.
- In production, protect against negative counts (`Math.max(0, …)` or enforce at read).

---

8) Read path

- API reads **Redis** (sum shards) for low latency.
- Admin/analytics can read **Prisma** `VideoLikeAggregate`.
- If you want exact reads always: read Redis; if cache miss or suspected drift, optionally reconcile with DB on a slow path.

Expose a read endpoint:

```
// apps/likes/src/likes.controller.ts (add)
import { Get, Param } from '@nestjs/common';

@Get(':videoId/count')
get(@Param('videoId') videoId: string) {
  return this.svc.getCount(videoId);
}
```

---

9) Idempotency & double-like protection

- The Lua scripts do **SADD/SREM** on a per-video **user set** before increment/decrement.
- If you can't afford a SET per video (memory), options:
  - **Bloom filter** per video (approx; may block rare legitimate likes), plus periodic hard dedupe.

- **Outbox/Inbox** pattern with a small **"user→video last action"** KV store (TTL) to dedupe at the aggregator.

---

10) Failure & correctness

- **Exactly-once is fantasy** with Kafka. Do **at-least-once** with idempotent aggregation:
  - Keep a Redis `agg:v:{videoId}` incremental bucket; consumer uses `HINCRBY` and flushes snapshot deltas. Or
  - Use **upsert + increment** like above (safe if each event processed once; for duplicates, you'll double-count).
- To handle duplicates, add an **eventId** and a Redis `SETNX processed:{eventId}` with TTL in the consumer. Cheap and effective.

---

11) Observability

- Export counters: Redis shard totals, queue lag, batch sizes, flush latency.
- Alerts if Redis misses surge or Kafka lag spikes.

---

12) Hard numbers / configs that work

- Shards: **64 or 128** per entity handles 50k–200k RPS writes fine.
- Batch flush: every **1–2s** or **500–2k events**, whichever first.
- Redis: use **pipeline** on MGET/MSET freely; we already do MGET for reads.
- Keys TTL: **do not** set TTL for counters; set TTL for processed-event dedupe keys (e.g., 24h).

TL;DR

- **Write**: API → Redis (atomic Lua, sharded) + emit Kafka.
- **Aggregate**: Kafka consumer → batch upserts to Postgres via Prisma.
- **Read**: from Redis (sum shards).
- **Dedupe**: Redis SET (or Bloom) in Lua.

- **Scale**: shards + batching.
- **Consistency**: eventual; good enough for likes/views.