How Twitter/X Works

> ***Target Audience**: Fullstack engineers (TS/Go, PostgreSQL, Redis)*
> ***Focus**: Real-time feed, viral spread, scale challenges*
> ***Scale Context**: ~550M MAU, ~200M DAU, ~500M tweets/day*

---

1. Problem Definition (What This System Must Achieve)

Twitter is a public microblogging platform optimized for real-time information dissemination.

Core functional requirements:

- Post tweets (280 chars, media, links, polls)
- Follow/unfollow users (asymmetric relationship)
- Timeline (home feed showing tweets from followed users)
- Replies, retweets, quotes, likes
- Trending topics and hashtags
- Search (tweets, users, hashtags)
- Notifications (mentions, likes, follows)
- Direct messages

Non-functional requirements:

- **Latency**: Timeline load < 300ms (p95), tweet posting < 200ms
- **Scale**: 500M tweets/day = ~6K tweets/sec avg, ~20K/sec peak
- **Availability**: 99.95%+ (5-nines impossible for social media)
- **Real-time**: New tweets appear in followers' timelines within seconds
- **Viral spread**: Handle 100K+ retweets/sec for major events
- **Read-heavy**: 100:1 read/write ratio

What makes this hard:

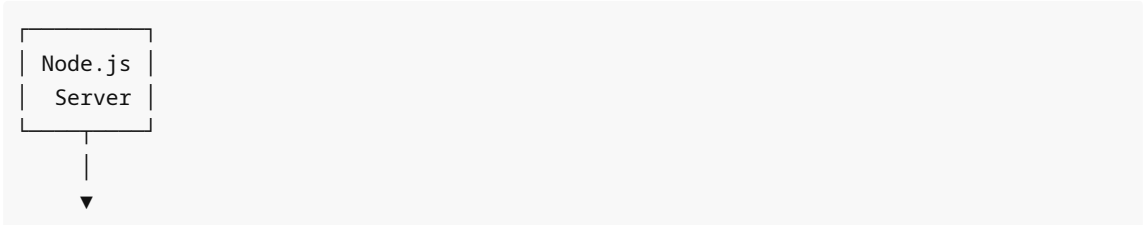
- **Fan-out problem**: Celebrity with 100M followers tweets → 100M timeline updates
- **Real-time delivery**: Push vs pull trade-offs at scale
- **Trending detection**: Identify viral content within minutes
- **Spam/abuse**: Bots, harassment, misinformation at massive scale
- **Global distribution**: Users expect low latency worldwide
- **Hot users**: Taylor Swift tweets → 10M concurrent requests

Unlike LinkedIn (private network), Twitter is **public by default** — this changes everything about caching, access control, and viral spread dynamics.

---

2. Naive Design (And Why It Fails)

The Simple Version



```
┌──────────┐
│Postgres  │
│  (all)   │
└──────────┘
```

Schema:

```
users: { id, username, display_name, bio, followers_count, ... }
follows: { follower_id, followee_id, created_at }
tweets: { id, author_id, content, created_at, retweet_count, like_count }
timeline: computed on-demand
```

Timeline generation (naive):

```
async function getTimeline(userId: string, limit: number) {
  // Get all users I follow
  const following = await db.query(
    'SELECT followee_id FROM follows WHERE follower_id = $1',
    [userId]
  );

  // Get tweets from all followed users
  const tweets = await db.query(
    'SELECT * FROM tweets WHERE author_id = ANY($1) ORDER BY created_at DESC LIMIT
$2',
    [following.map(f => f.followee_id), limit]
  );

  return tweets;
}
```

Why This Breaks

1. Timeline query explodes:

- Average user follows 500 people
- Query scans 500 users' tweets (potentially millions of rows)
- With 200M DAU loading timelines → database melts
- **Result**: p99 latency > 10 seconds

2. Celebrity fan-out kills writes:

- Obama tweets → 132M followers need timeline updates
- Naive approach: Write to 132M users' pre-materialized timelines
- Even at 1K writes/sec, takes 37 hours to complete
- **Result**: Followers don't see tweet for hours/days

3. Viral content creates thundering herd:

- Breaking news tweet gets 50K retweets in 1 minute
- Each retweet triggers timeline updates for retweeter's followers
- Exponential cascade of writes

- **Result**: System grinds to halt during major events

4. Trending detection is impossible:

- Count all hashtag occurrences in real-time?
- Requires scanning every tweet continuously
- **Result**: Can't detect trends, or 30-minute lag

5. Global latency is terrible:

- All users hit single-region database
- User in Tokyo → 200ms to US-East DB
- **Result**: Poor UX for non-US users

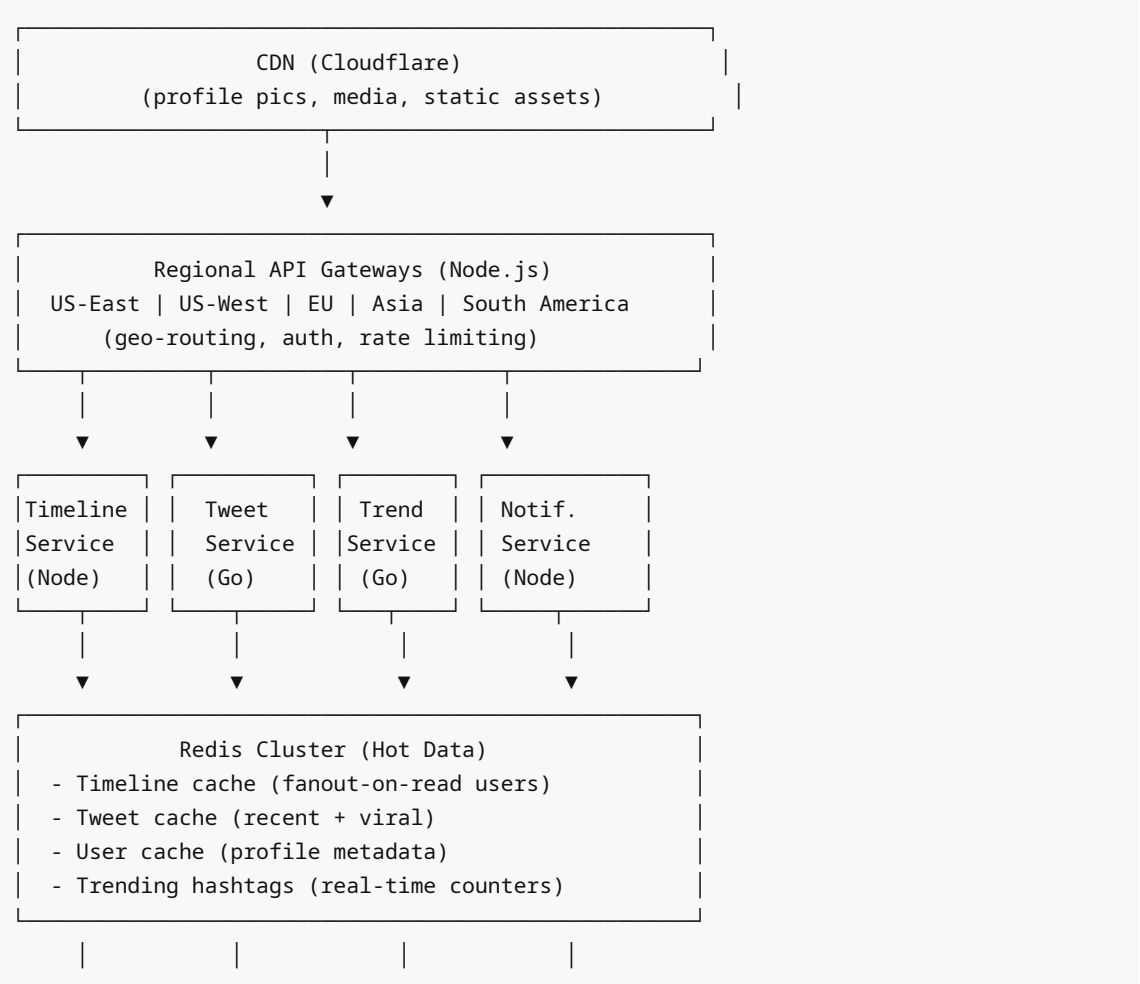
6. Hot key problems:

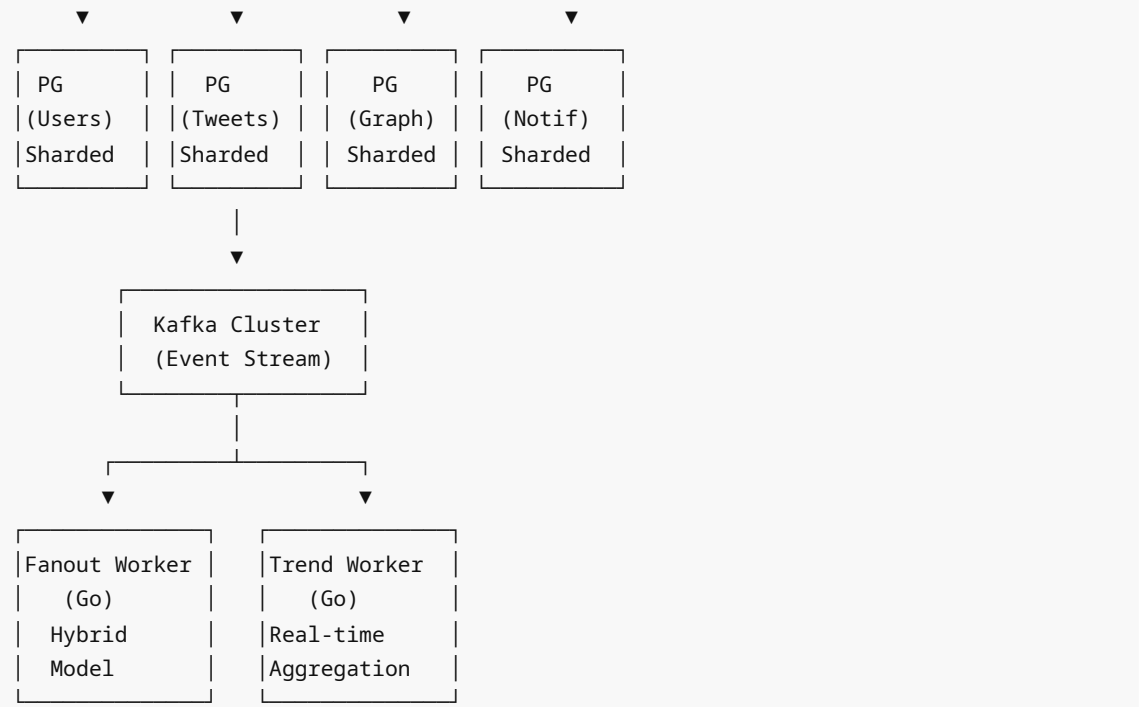
- Taylor Swift's tweet cached in Redis
- 10M concurrent requests hit same key
- Single Redis node saturated
- **Result**: Cache becomes bottleneck

---

3. High-Level Architecture

Component Overview





Service Boundaries

Timeline Service (Node.js):

- Fetch home timeline (cached or computed)
- Implements hybrid fanout model (explained in Section 6)
- Merge fanout-on-write + fanout-on-read
- Pagination and filtering

Tweet Service (Go):

- Create/delete tweets
- Handle media uploads (S3)
- Real-time delivery to followers
- Like/retweet/reply operations
- Chosen Go for: High write throughput, efficient fanout workers

Trend Service (Go):

- Real-time hashtag counting (HyperLogLog)
- Trending topic detection
- Geo-specific trends
- Chosen Go for: CPU-heavy aggregation, low-latency streaming

Notification Service (Node.js):

- Push notifications (likes, mentions, follows)
- WebSocket connections for real-time alerts
- Batching and deduplication

---

4. Core Data Model

PostgreSQL Schema (Sharded)

Users Database (sharded by user_id):

```
interface User {
  id: string; // Snowflake ID (sortable, time-embedded)
  username: string; // unique, indexed
  display_name: string;
  bio: string;
  profile_image_url: string;
  verified: boolean;
  created_at: timestamp;

  // Denormalized counters
  followers_count: number;
  following_count: number;
  tweet_count: number;
}
// Index: (username), (created_at)

// User tier (affects fanout strategy)
interface UserTier {
  user_id: string;
  tier: 'normal' | 'verified' | 'celebrity'; // celebrity = >1M followers
  updated_at: timestamp;
}
```

Social Graph Database (sharded by follower_id):

```
interface Follow {
  follower_id: string; // User who follows
  followee_id: string; // User being followed
  created_at: timestamp;
}
// Composite PK: (follower_id, followee_id)
// Index: (follower_id, created_at) - get who I follow
// Index: (followee_id, created_at) - get who follows me
```

Tweets Database (sharded by author_id):

```
interface Tweet {
  id: string; // Snowflake ID
  author_id: string;
  content: string; // max 280 chars
  media_urls: string[]; // S3 keys

  // Tweet type
  type: 'original' | 'retweet' | 'quote' | 'reply';
  reply_to_tweet_id: string | null;
  retweet_of_tweet_id: string | null;
```

```
  created_at: timestamp;

  // Denormalized engagement (eventually consistent)
  retweet_count: number;
  like_count: number;
  reply_count: number;
  view_count: number;
}
// Index: (author_id, created_at)
// Index: (created_at) for global recent tweets
// Index: (reply_to_tweet_id) for reply threads

interface TweetEngagement {
  tweet_id: string;
  user_id: string;
  type: 'like' | 'retweet' | 'bookmark';
  created_at: timestamp;
}
// Composite PK: (tweet_id, user_id, type)
// Index: (user_id, type, created_at) for user activity
```

Timeline Cache (Redis):

```
// Fanout-on-write timeline (normal users)
// Sorted set: score = tweet timestamp
`timeline:${userId}` → SortedSet<tweetId>

// Recently cached tweets (hot data)
`tweet:${tweetId}` → JSON of tweet object

// User metadata cache
`user:${userId}` → JSON { username, displayName, verified, ... }

// Trending topics (current hour)
`trending:${hour}` → SortedSet<hashtag, score>

// Follower count cache (read-heavy)
`followers:${userId}` → number
```

Consistency Guarantees

Strongly consistent:

- Follow/unfollow operations
- Tweet creation (tweet must be persisted before fanout)
- User credentials

Eventually consistent:

- Engagement counters (likes, retweets)
- Timelines (acceptable delay: 1-2 seconds)

- Trending topics (5-minute lag acceptable)

Immutable:

- Tweets (edit not allowed, only delete)
- Follow history (for audit/analytics)

---

5. Core Workflows

Workflow 1: User Posts a Tweet

Step-by-step:

1. **Client** sends POST request

```
POST /api/v1/tweets
{
  "content": "just setting up my twttr",
  "media": []
}
```

2. **API Gateway** (Node.js):
   - Validates auth token
   - Rate limit check (300 tweets/day for normal users)
   - Forwards to Tweet Service
3. **Tweet Service** (Go) creates tweet:

```
func (s *TweetService) CreateTweet(authorId, content string, media []string)
(*Tweet, error) {
    // Generate Snowflake ID (time-ordered, distributed)
    tweetId := s.snowflake.Generate()

    // Upload media to S3 (if any)
    mediaUrls := s.uploadMedia(media)

    // Persist tweet to Postgres
    tweet := &Tweet{
        Id:        tweetId,
        AuthorId:  authorId,
        Content:   content,
        MediaUrls: mediaUrls,
        Type:      "original",
        CreatedAt: time.Now(),
    }

    _, err := s.db.Exec(
        "INSERT INTO tweets (id, author_id, content, media_urls, type,
created_at) VALUES ($1, $2, $3, $4, $5, $6)",
        tweet.Id, tweet.AuthorId, tweet.Content, pq.Array(tweet.MediaUrls),
tweet.Type, tweet.CreatedAt,
    )
```

```
    if err != nil {
        return nil, err
    }

    // Cache tweet (hot data)
    s.redis.SetEx(ctx, fmt.Sprintf("tweet:%s", tweetId), json.Marshal(tweet),
1*time.Hour)

    // Publish to Kafka for async fanout
    s.kafka.Publish("tweet.created", TweetCreatedEvent{
        TweetId:  tweetId,
        AuthorId: authorId,
        CreatedAt: time.Now(),
    })

    return tweet, nil
}
```

4. **Fanout Worker** (Go) picks up event:

```
func (w *FanoutWorker) HandleTweetCreated(event TweetCreatedEvent) error {
    // Get author's tier
    tier := w.getUserTier(event.AuthorId)

    switch tier {
    case "normal":
        // Fanout-on-write: Push to all followers' timelines
        return w.fanoutToFollowers(event)

    case "celebrity":
        // Fanout-on-read: Don't push, let users pull on demand
        // Just cache the tweet
        return w.cacheHotTweet(event.TweetId)

    case "verified":
        // Hybrid: Push to active followers only
        return w.hybridFanout(event)
    }

    return nil
}

func (w *FanoutWorker) fanoutToFollowers(event TweetCreatedEvent) error {
    // Get followers (from cache or DB)
    followers := w.getFollowers(event.AuthorId)

    // Batch writes to Redis (pipeline for performance)
    pipe := w.redis.Pipeline()

    for _, followerId := range followers {
        // Add tweet to follower's timeline (sorted set)
        pipe.ZAdd(ctx, fmt.Sprintf("timeline:%s", followerId), &redis.Z{
```

```
            Score:  float64(event.CreatedAt.Unix()),
            Member: event.TweetId,
        })

        // Keep only recent 800 tweets (trim older)
        pipe.ZRemRangeByRank(ctx, fmt.Sprintf("timeline:%s", followerId), 0,
-801)
    }

    // Execute all writes at once
    _, err := pipe.Exec(ctx)
    return err
}
```

5. **Client** receives response with tweet ID
    - Optimistic UI update

Failure Handling:

- S3 upload fails → retry 3x, fail request
- DB insert fails → return error, no fanout
- Kafka publish fails → log error, **but don't fail request** (fanout can be retried)
- Redis fanout fails → retry via dead letter queue (eventual consistency)

Workflow 2: User Loads Home Timeline

Core challenge: Some followed users are celebrities (fanout-on-read), some are normal (fanout-on-write)

Step-by-step:

1. **Client** requests timeline

```
GET /api/v1/timeline?limit=20&cursor=abc123
```

2. **Timeline Service** (Node.js) - **Hybrid Approach**:

```
async function getTimeline(userId: string, limit: number, cursor?: string) {
  // Get user's followed accounts
  const following = await getFollowing(userId);

  // Separate normal vs celebrity accounts
  const { normalUsers, celebUsers } = await categorizeFo llowing(following);

  // Two parallel queries:
  const [cachedTweets, celebrityTweets] = await Promise.all([
    // 1. Fanout-on-write: Fetch from pre-materialized timeline
    fetchCachedTimeline(userId, limit),

    // 2. Fanout-on-read: Fetch recent tweets from celebrities
    fetchCelebrityTweets(celebUsers, limit)
  ]);
```

```
  // Merge both lists (sort by timestamp)
  const merged = mergeTweets(cachedTweets, celebrityTweets, limit);

  // Hydrate tweets (get full tweet objects)
  const tweets = await hydrateTweets(merged);

  return {
    tweets,
    nextCursor: generateCursor(tweets[tweets.length - 1].id)
  };
}

async function fetchCachedTimeline(userId: string, limit: number) {
  // Redis sorted set (fanout-on-write results)
  const tweetIds = await redis.zrevrange(
    `timeline:${userId}`,
    0,
    limit - 1
  );
  return tweetIds;
}

async function fetchCelebrityTweets(celebIds: string[], limit: number) {
  // Query DB directly for recent tweets from celebrities
  const tweets = await db.query(
    `SELECT id FROM tweets
     WHERE author_id = ANY($1)
     AND created_at > NOW() - INTERVAL '24 hours'
     ORDER BY created_at DESC
     LIMIT $2`,
    [celebIds, limit]
  );
  return tweets.map(t => t.id);
}

function mergeTweets(cached: string[], celebrity: string[], limit: number) {
  // Decode Snowflake IDs (they contain timestamp)
  const allTweets = [...cached, ...celebrity]
    .map(id => ({ id, timestamp: snowflakeToTimestamp(id) }))
    .sort((a, b) => b.timestamp - a.timestamp)
    .slice(0, limit);

  return allTweets.map(t => t.id);
}

async function hydrateTweets(tweetIds: string[]) {
  // Check Redis cache first
  const cached = await redis.mget(tweetIds.map(id => `tweet:${id}`));

  const tweets = [];
  const missingIds = [];
```

```
  for (let i = 0; i < tweetIds.length; i++) {
    if (cached[i]) {
      tweets.push(JSON.parse(cached[i]));
    } else {
      missingIds.push(tweetIds[i]);
    }
  }

  // Fetch missing from DB
  if (missingIds.length > 0) {
    const dbTweets = await db.query(
      'SELECT * FROM tweets WHERE id = ANY($1)',
      [missingIds]
    );

    tweets.push(...dbTweets);

    // Cache for next time
    for (const tweet of dbTweets) {
      await redis.setex(`tweet:${tweet.id}`, 3600, JSON.stringify(tweet));
    }
  }

  // Sort by timestamp
  return tweets.sort((a, b) => b.created_at - a.created_at);
}
```

Performance:

- Fanout-on-write users: 50ms (Redis lookup)
- Fanout-on-read (celebrities): 150ms (DB query)
- Merge + hydrate: 50ms
- **Total p95**: ~300ms

Workflow 3: User Retweets

Step-by-step:

1. **Client** sends retweet request

```
POST /api/v1/tweets/:tweetId/retweet
```

2. **Tweet Service** creates retweet record:

```
func (s *TweetService) Retweet(userId, originalTweetId string) (*Tweet, error)
{
    // Check if already retweeted (idempotency)
    existing, _ := s.db.Query(
        "SELECT id FROM tweets WHERE author_id = $1 AND retweet_of_tweet_id =
$2",
```

```
        userId, originalTweetId,
    )
    if existing.Next() {
        return nil, errors.New("already retweeted")
    }

    // Create retweet
    retweetId := s.snowflake.Generate()

    _, err := s.db.Exec(
        "INSERT INTO tweets (id, author_id, type, retweet_of_tweet_id,
created_at) VALUES ($1, $2, $3, $4, $5)",
        retweetId, userId, "retweet", originalTweetId, time.Now(),
    )
    if err != nil {
        return nil, err
    }

    // Increment original tweet's retweet counter (async, eventual
consistency)
    s.redis.Incr(ctx, fmt.Sprintf("tweet:%s:retweets", originalTweetId))
    s.kafka.Publish("tweet.retweeted", TweetRetweetedEvent{
        OriginalTweetId: originalTweetId,
        RetweetId:       retweetId,
        UserId:          userId,
    })

    // Fanout retweet to user's followers
    s.kafka.Publish("tweet.created", TweetCreatedEvent{
        TweetId:  retweetId,
        AuthorId: userId,
        CreatedAt: time.Now(),
    })

    return &Tweet{Id: retweetId}, nil
}
```

3. **Background worker** updates counters:

```
func (w *CounterWorker) HandleRetweet(event TweetRetweetedEvent) {
    // Batch updates every 10 seconds
    w.batch = append(w.batch, event)

    if len(w.batch) >= 1000 || time.Since(w.lastFlush) > 10*time.Second {
        w.flushCounters()
    }
}

func (w *CounterWorker) flushCounters() {
    // Bulk update retweet counts
    for tweetId, count := range w.aggregateBatch() {
        w.db.Exec(
```

```
            "UPDATE tweets SET retweet_count = retweet_count + $1 WHERE id =
$2",
            count, tweetId,
        )
    }

    w.batch = nil
    w.lastFlush = time.Now()
}
```

Viral retweet handling:

- If original tweet is from celebrity → retweet also triggers fanout-on-read
- If retweet rate > 1K/sec → deduplicate counter updates
- Trending detection worker picks up signal

Workflow 4: Trending Topics Detection

Real-time hashtag counting:

1. **Tweet created** with hashtag `#BREAKING`
2. **Trend Service** (Go) consumes Kafka stream:

```go
func (t *TrendService) ProcessTweet(tweet Tweet) {
    // Extract hashtags
    hashtags := extractHashtags(tweet.Content)

    for _, tag := range hashtags {
        // Increment HyperLogLog (approximate unique users)
        key := fmt.Sprintf("trend:%s:%d", tag, time.Now().Hour())
        t.redis.PFAdd(ctx, key, tweet.AuthorId)

        // Get unique user count
        count, _ := t.redis.PFCount(ctx, key).Result()

        // Update trending sorted set
        t.redis.ZAdd(ctx, "trending:global", &redis.Z{
            Score:  float64(count),
            Member: tag,
        })
    }
}

func (t *TrendService) GetTrending(limit int) []string {
    // Get top N trending hashtags
    tags, _ := t.redis.ZRevRange(ctx, "trending:global", 0, int64(limit-
1)).Result()
    return tags
}
```

Why HyperLogLog:

- Counts unique users mentioning a hashtag
- Uses only 12KB per hashtag (vs millions for exact sets)
- 1% error rate is acceptable for trending

Trending algorithm:

- Not just raw count (prevents spam/bots)
- Score = unique_users * velocity_multiplier
- Velocity = (current_hour_count / previous_hour_count)
- Filters: Remove banned hashtags, minimum 1K unique users

---

6. API Design

REST Endpoints

Authentication:

```
POST /api/v1/auth/login
POST /api/v1/auth/logout
GET  /api/v1/auth/verify
```

Tweets:

```
POST   /api/v1/tweets
DELETE /api/v1/tweets/:id
GET    /api/v1/tweets/:id
POST   /api/v1/tweets/:id/like
DELETE /api/v1/tweets/:id/like
POST   /api/v1/tweets/:id/retweet
POST   /api/v1/tweets/:id/reply
```

Timeline:

```
GET /api/v1/timeline?limit=20&cursor=xyz
GET /api/v1/users/:userId/tweets?limit=20
```

Social Graph:

```
POST   /api/v1/users/:userId/follow
DELETE /api/v1/users/:userId/follow
GET    /api/v1/users/:userId/followers?limit=20
GET    /api/v1/users/:userId/following?limit=20
```

Trends:

```
GET /api/v1/trends/global
GET /api/v1/trends/:location
```

Search:

```
GET /api/v1/search?q=query&type=tweets|users
```

Cursor-based Pagination

Why not offset-based:

- `LIMIT 1000 OFFSET 10000` scans 11K rows
- Timeline changes during pagination (new tweets)
- Result: Duplicate or missing tweets

Cursor approach:

```
GET /api/v1/timeline?limit=20&cursor=1234567890

Response:
{
  "tweets": [...],
  "nextCursor": "1234567850" // Last tweet's Snowflake ID
}

// Next page:
GET /api/v1/timeline?limit=20&cursor=1234567850
```

Implementation:

```
async function getTimeline(userId: string, limit: number, cursor?: string) {
  let query = 'SELECT * FROM timeline WHERE user_id = $1';
  let params = [userId];

  if (cursor) {
    // Snowflake IDs are time-ordered, so we can use them for pagination
    query += ' AND id < $2';
    params.push(cursor);
  }

  query += ' ORDER BY id DESC LIMIT $' + (params.length + 1);
  params.push(limit);

  const tweets = await db.query(query, params);

  return {
    tweets,
    nextCursor: tweets[tweets.length - 1]?.id
  };
}
```

Idempotency

Like operation:

```
POST /api/v1/tweets/:id/like
```

Server-side ensures idempotency:

```
async function likeTweet(userId: string, tweetId: string) {
  // INSERT ... ON CONFLICT DO NOTHING (Postgres)
  const result = await db.query(
    `INSERT INTO tweet_engagement (tweet_id, user_id, type, created_at)
     VALUES ($1, $2, 'like', NOW())
     ON CONFLICT (tweet_id, user_id, type) DO NOTHING
     RETURNING *`,
    [tweetId, userId]
  );

  if (result.rows.length === 0) {
    return { alreadyLiked: true };
  }

  // Increment counter
  await redis.incr(`tweet:${tweetId}:likes`);

  return { success: true };
}
```

---

7. Snowflake ID Generation

Why Not Auto-increment or UUID?

Auto-increment (1, 2, 3, ...):

- ❌ Reveals tweet volume (competitors can track growth)
- ❌ Requires centralized sequence (single point of failure)
- ❌ Not sortable across shards

UUID (random):

- ❌ Not time-ordered (can't use for pagination)
- ❌ 128 bits (inefficient storage)
- ❌ No embedded metadata

Snowflake ID:

- ✅ 64-bit integer (efficient indexing)
- ✅ Time-ordered (sortable, paginate-able)
- ✅ Globally unique without coordination
- ✅ Encodes timestamp + machine + sequence

Structure

```
| 41 bits: timestamp | 10 bits: machine ID | 12 bits: sequence |
|--------------------------------------------------------|
```

```
Example: 1234567890123456789
- Timestamp: Milliseconds since epoch (Jan 1, 2020)
- Machine ID: Which server generated this (0-1023)
- Sequence: Counter within same millisecond (0-4095)
```

Implementation (Go)

```go
type Snowflake struct {
    epoch      int64 // Custom epoch (Jan 1, 2020)
    machineId  int64 // Unique per server (0-1023)
    sequence   int64
    lastTime   int64
    mu         sync.Mutex
}

func NewSnowflake(machineId int64) *Snowflake {
    return &Snowflake{
        epoch:     1577836800000, // Jan 1, 2020 in ms
        machineId: machineId,
    }
}

func (s *Snowflake) Generate() int64 {
    s.mu.Lock()
    defer s.mu.Unlock()

    now := time.Now().UnixMilli()

    if now == s.lastTime {
        // Same millisecond: increment sequence
        s.sequence = (s.sequence + 1) & 4095 // Mask to 12 bits

        if s.sequence == 0 {
            // Sequence overflow: wait for next millisecond
            for now <= s.lastTime {
                time.Sleep(100 * time.Microsecond)
                now = time.Now().UnixMilli()
            }
        }
    } else {
        s.sequence = 0
    }

    s.lastTime = now

    // Combine fields
    id := ((now - s.epoch) << 22) | (s.machineId << 12) | s.sequence
    return id
}

// Extract timestamp from ID
```

```go
func (s *Snowflake) GetTimestamp(id int64) time.Time {
    ms := (id >> 22) + s.epoch
    return time.UnixMilli(ms)
}
```

Advantages

Pagination:

```sql
// Get tweets older than cursor
SELECT * FROM tweets WHERE id < $1 ORDER BY id DESC LIMIT 20;
```

Time-range queries:

```sql
// Get tweets from last hour
const oneHourAgo = generateSnowflakeForTime(Date.now() - 3600000);
SELECT * FROM tweets WHERE id > ${oneHourAgo};
```

Sharding:

```
// Consistent shard assignment
const shard = snowflakeId % numShards;
```

---

8. Consistency, Ordering & Concurrency

Consistency Model by Feature

Strong consistency:

- Follow/unfollow operations
- Tweet creation (tweet must exist before fanout)
- Account suspension/deletion

Timeline consistency:

- Eventual (1-2 second lag acceptable)
- User sees their own tweet immediately (write-through to cache)

Engagement counters:

- Eventually consistent
- Acceptable to be off by <1% during high load

Follow/Unfollow Race Condition

Problem:

```
User A follows User B
User A unfollows User B (within 100ms)

Timeline worker processes "follow" event → starts fanout
```

```
Timeline worker processes "unfollow" event → but fanout already started
Result: User A still sees User B's tweets
```

Solution: Versioned events

```go
type FollowEvent struct {
    FollowerId string
    FolloweeId string
    Action     string // "follow" or "unfollow"
    Version    int64  // Monotonically increasing
    Timestamp  time.Time
}

func (w *FanoutWorker) ProcessFollowEvent(event FollowEvent) {
    // Get last processed version for this relationship
    key := fmt.Sprintf("follow_version:%s:%s", event.FollowerId, event.FolloweeId)
    lastVersion, _ := w.redis.Get(ctx, key).Int64()

    if event.Version <= lastVersion {
        // Stale event, ignore
        return
    }

    // Process event
    if event.Action == "follow" {
        w.startFanout(event.FolloweeId, event.FollowerId)
    } else {
        w.removeFanout(event.FolloweeId, event.FollowerId)
    }

    // Update version
    w.redis.Set(ctx, key, event.Version, 24*time.Hour)
}
```

Retweet Counter Race Condition

Problem:

- Viral tweet: 10K retweets/second
- All workers incrementing same counter
- Database lock contention

Solution: Write-behind with batching

```go
type CounterAggregator struct {
    buffer map[string]int64 // tweetId -> count
    mu     sync.Mutex
}

func (a *CounterAggregator) IncrementRetweet(tweetId string) {
    a.mu.Lock()
    defer a.mu.Unlock()
```

```
    a.buffer[tweetId]++
}

// Flush every 10 seconds
func (a *CounterAggregator) FlushLoop() {
    ticker := time.NewTicker(10 * time.Second)

    for range ticker.C {
        a.mu.Lock()
        snapshot := a.buffer
        a.buffer = make(map[string]int64)
        a.mu.Unlock()

        // Batch update to DB
        for tweetId, count := range snapshot {
            db.Exec(
                "UPDATE tweets SET retweet_count = retweet_count + $1 WHERE id =
$2",
                count, tweetId,
            )
        }
    }
}
```

Trade-off:

- Counters can be stale by up to 10 seconds
- But no lock contention
- 10K writes/sec → 100 batched writes/sec

Timeline Ordering Guarantees

Requirement:

- Tweets must appear in chronological order
- Even if fanout is delayed or reordered

Solution: Snowflake IDs as ordering

- Tweet ID embeds timestamp
- Redis sorted set uses Snowflake ID as score
- Guarantees chronological order without clock sync issues

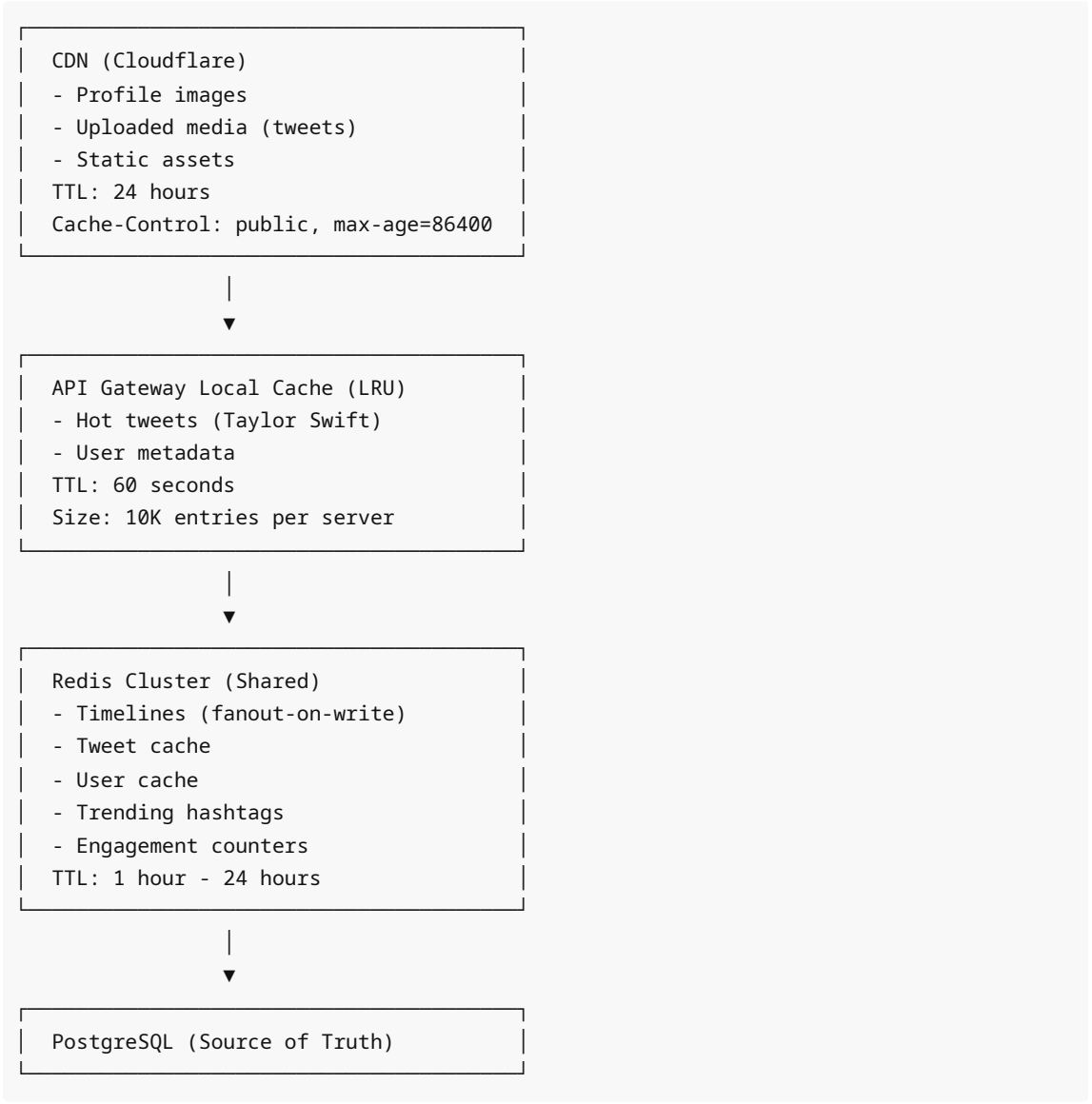
```
// Add to timeline (score = Snowflake ID itself)
redis.ZAdd(ctx, fmt.Sprintf("timeline:%s", userId), &redis.Z{
    Score:  float64(tweetId), // Snowflake ID is time-ordered
    Member: tweetId,
})

// Fetch timeline (automatically sorted by time)
redis.ZRevRange(ctx, fmt.Sprintf("timeline:%s", userId), 0, 19)
```

---

9. Caching Strategy

Layer Architecture



What Gets Cached

Timeline cache (Redis sorted set):

```
Key: `timeline:${userId}`
Value: SortedSet of tweet IDs (score = Snowflake ID)
TTL: 2 hours
Size: 800 tweets per user
Eviction: Keep only latest 800

// Cache timeline
await redis.zadd(`timeline:${userId}`, tweetId, tweetId); // Score = member
```

```
await redis.zremrangebyrank(`timeline:${userId}`, 0, -801); // Keep latest 800
await redis.expire(`timeline:${userId}`, 7200);
```

Tweet object cache (Redis hash):

```
Key: `tweet:${tweetId}`
Value: JSON {id, authorId, content, mediaUrls, createdAt, likeCount, retweetCount}
TTL: 1 hour (normal), 24 hours (viral)

// Detect viral tweets
if (likeCount > 10000) {
  // Extend TTL, replicate to multiple keys
  await redis.expire(`tweet:${tweetId}`, 86400);
}
```

User metadata cache:

```
Key: `user:${userId}`
Value: JSON {id, username, displayName, verified, profileImage}
TTL: 4 hours

// On profile update, invalidate
await redis.del(`user:${userId}`);
```

Follower count cache:

```
Key: `followers:${userId}`
Value: Integer count
TTL: 10 minutes

// Update on follow/unfollow
await redis.incrby(`followers:${userId}`, 1); // or -1
```

Hot Key Mitigation

Problem: Taylor Swift tweets

- 100M followers
- 10M+ concurrent reads within minutes
- Single Redis key = bottleneck

Solution 1: Local in-memory cache

```
import LRU from 'lru-cache';

const localCache = new LRU({
  max: 10000,
  ttl: 60000 // 60 seconds
});
```

```
async function getTweet(tweetId: string) {
  // Check local cache first (no network)
  let tweet = localCache.get(tweetId);
  if (tweet) return tweet;

  // Check Redis
  tweet = await redis.get(`tweet:${tweetId}`);
  if (tweet) {
    localCache.set(tweetId, tweet);
    return tweet;
  }

  // Fetch from DB
  tweet = await db.query('SELECT * FROM tweets WHERE id = $1', [tweetId]);
  await redis.setex(`tweet:${tweetId}`, 3600, JSON.stringify(tweet));
  localCache.set(tweetId, tweet);

  return tweet;
}
```

Effect:

- 1M requests to same tweet
- Local cache absorbs 90%+ → 100K to Redis
- Redis load reduced by 10x

Solution 2: Key replication

```
// Write to multiple Redis keys
async function cacheHotTweet(tweet: Tweet) {
  const replicas = 10;
  const promises = [];

  for (let i = 0; i < replicas; i++) {
    promises.push(
      redis.setex(`tweet:${tweet.id}:${i}`, 86400, JSON.stringify(tweet))
    );
  }

  await Promise.all(promises);
}

// Read from random replica
async function getHotTweet(tweetId: string) {
  const replica = Math.floor(Math.random() * 10);
  return redis.get(`tweet:${tweetId}:${replica}`);
}
```

Cache Stampede Prevention

Problem:

- Timeline cache expires for 1M users at same time
- All hit DB simultaneously
- DB overload

Solution: Probabilistic early recomputation

```
async function getTimeline(userId: string) {
  const cacheKey = `timeline:${userId}`;

  // Check TTL
  const ttl = await redis.ttl(cacheKey);

  if (ttl > 0) {
    // Probability of early recomputation increases as TTL decreases
    const recomputeProbability = 1 - (ttl / 7200); // TTL is 2 hours

    if (Math.random() < recomputeProbability) {
      // Trigger async recomputation
      recomputeTimelineAsync(userId);
    }
  }

  // Return cached value
  const cached = await redis.zrevrange(cacheKey, 0, 19);
  if (cached.length > 0) return cached;

  // Cache miss: compute synchronously
  return computeTimeline(userId);
}
```

Cache Invalidation Patterns

1. Time-based (timelines):

- No manual invalidation
- TTL = 2 hours
- Acceptable staleness

2. Event-based (user profiles):

```
// User updates profile
async function updateProfile(userId: string, updates: ProfileUpdate) {
  await db.query('UPDATE users SET ... WHERE id = $1', [userId, ...]);

  // Invalidate cache
  await redis.del(`user:${userId}`);

  // Publish event
  await kafka.publish('user.updated', { userId });
}
```

3. Write-through (user's own timeline):

```
// User posts tweet
async function createTweet(userId: string, content: string) {
  const tweet = await db.query('INSERT INTO tweets ...');

  // Immediately add to user's own timeline cache
  await redis.zadd(`timeline:${userId}`, tweet.id, tweet.id);

  // User sees their tweet instantly (no wait for fanout)
  return tweet;
}
```

---

10. Scaling Strategy

Horizontal Scaling

API Gateway:

- Stateless Node.js servers
- 200+ instances globally
- Auto-scale on CPU (target 70%)
- Geo-routed via Cloudflare

Tweet Service (Go):

- Stateless
- 50-100 instances
- High write throughput
- Sharded Kafka partitions for fanout

Timeline Service (Node.js):

- Stateless
- 100+ instances
- Read-heavy
- Heavy Redis usage

Fanout Workers (Go):

- Consumer group (Kafka)
- Auto-scale based on consumer lag
- 20-50 instances
- Each processes 100-200 tweets/sec

Database Sharding

Tweets table (sharded by author_id):

```
// Route to shard based on author
function getTweetShard(authorId: string): number {
  const hash = xxhash(authorId);
  return hash % 64; // 64 shards
}
```

```
async function getTweetsByAuthor(authorId: string) {
  const shard = getTweetShard(authorId);
  const db = getDBConnection(`tweets_shard_${shard}`);
  return db.query('SELECT * FROM tweets WHERE author_id = $1 ORDER BY created_at
DESC', [authorId]);
}
```

Why shard by author_id:

- Most queries: "Get tweets by user"
- Single-shard operation (no scatter-gather)
- Downside: Celebrity tweets hit same shard (solved via caching)

Social graph (sharded by follower_id):

```
// Shard by follower
function getFollowShard(followerId: string): number {
  return xxhash(followerId) % 32;
}

async function getFollowing(userId: string): Promise<string[]> {
  const shard = getFollowShard(userId);
  const db = getDBConnection(`follows_shard_${shard}`);

  const result = await db.query(
    'SELECT followee_id FROM follows WHERE follower_id = $1',
    [userId]
  );

  return result.map(r => r.followee_id);
}
```

Cross-shard query (get followers of user):

```
async function getFollowers(userId: string): Promise<string[]> {
  // BAD: Would require querying all 32 shards
  // SELECT follower_id FROM follows WHERE followee_id = $1

  // SOLUTION 1: Cache in Redis
  const cached = await redis.smembers(`followers:${userId}`);
  if (cached.length > 0) return cached;

  // SOLUTION 2: Denormalize - maintain separate table
  const db = getDBConnection('followers_denormalized');
  return db.query('SELECT follower_id FROM user_followers WHERE user_id = $1',
[userId]);
}
```

Read Replicas

Setup per shard:

```
Shard 1 Primary (writes)
├── Replica 1 (reads)
├── Replica 2 (reads)
└── Replica 3 (reads)
```

Read routing:

```typescript
// Write → primary
await primaryDB.query('INSERT INTO tweets ...');

// Read → random replica
const replica = replicas[Math.floor(Math.random() * replicas.length)];
await replica.query('SELECT * FROM tweets WHERE id = $1', [tweetId]);
```

Replication lag handling:

```typescript
// Critical reads (user just posted) → read from primary
async function getTweetAfterCreation(tweetId: string, authorId: string) {
  // Use primary to avoid replication lag
  const shard = getTweetShard(authorId);
  const primaryDB = getPrimaryDB(shard);
  return primaryDB.query('SELECT * FROM tweets WHERE id = $1', [tweetId]);
}

// Non-critical reads → replica
async function getBrowseTimeline(userId: string) {
  // Replica lag (100-500ms) is acceptable
  const replica = getRandomReplica();
  return replica.query('SELECT * FROM timeline WHERE user_id = $1', [userId]);
}
```

Redis Cluster Scaling

Setup:

```
Redis Cluster: 16 master nodes, 16 replica nodes
Total: 32 nodes

Hash slot distribution:
Master 0: slots 0-1023
Master 1: slots 1024-2047
...
Master 15: slots 61440-65535
```

Automatic sharding:

```typescript
// Redis client automatically routes keys to correct node
await redis.set(`timeline:user123`, data); // Hashes key, routes to node

// Multi-key operations require same hash slot
```

```
await redis.mget([
  `tweet:123`,  // May be on different nodes
  `tweet:456`   // Client handles transparently
]);
```

Scaling up:

```
# Add new master node to cluster
redis-cli --cluster add-node new-node:6379 existing-node:6379

# Rebalance hash slots
redis-cli --cluster rebalance existing-node:6379
```

---

11. Fault Tolerance & Reliability

Failure Scenarios

1. Database shard failure:

- **Impact**: All users on that shard (1/64 of users) can't read/write tweets
- **Detection**: Health check fails every 10s
- **Mitigation**:
  - Automatic failover to read replica (Patroni/Stolon)
  - Promote replica to primary
  - RPO: ~500ms (replication lag)
  - RTO: ~2 minutes
- **Degradation**: Affected users see stale timelines during failover

2. Redis cluster node failure:

- **Impact**: Keys on that node unavailable (1/16 of data)
- **Detection**: Cluster health check
- **Mitigation**:
  - Automatic failover to replica
  - RTO: ~30 seconds
- **Effect on users**: Brief timeline loading errors, then recovers

3. Kafka partition leader failure:

- **Impact**: Fanout events for that partition delayed
- **Detection**: Consumer lag metric spikes
- **Mitigation**:
  - Kafka auto-elects new leader from ISR
  - RTO: ~10 seconds
- **Effect**: Some timelines delay by 10-20 seconds

4. Fanout worker failure:

- **Impact**: Some followers don't receive tweet immediately
- **Detection**: Consumer stops heartbeating
- **Mitigation**:
  - Kafka rebalances partition to other workers

    - RTO: ~30 seconds
- **Eventual consistency**: All followers receive tweet within 1 minute

5. API Gateway overload:

- **Impact**: High latency, request timeouts
- **Detection**: p99 latency > 3 seconds
- **Mitigation**:
    - Auto-scale (add more instances)
    - Load shedding (reject non-critical requests)
    - Circuit breaker (stop sending to unhealthy backends)
- **User experience**: "Twitter is over capacity" message

6. Cascading failure (thundering herd):

- **Scenario**: Redis cluster fails → all requests hit DB → DB overload → total outage
- **Prevention**:
    - Circuit breaker on DB connections
    - Rate limiting (per-user and global)
    - Graceful degradation (return empty timeline instead of failing)

```
async function getTimelineWithCircuitBreaker(userId: string) {
  try {
    // Try Redis first
    const cached = await circuitBreaker.execute(() =>
      redis.zrevrange(`timeline:${userId}`, 0, 19)
    );

    if (cached.length > 0) return cached;
  } catch (err) {
    logger.error('Redis failed, falling back to DB', {userId, error: err});
  }

  try {
    // Fallback to DB
    return await dbCircuitBreaker.execute(() =>
      buildTimelineFromDB(userId)
    );
  } catch (err) {
    // DB also failed: return empty timeline
    logger.error('DB also failed, returning empty', {userId, error: err});
    return [];
  }
}
```

Partial Failure Handling

Scenario: Tweet created, fanout fails

```
func (s *TweetService) CreateTweet(authorId, content string) (*Tweet, error) {
    // CRITICAL PATH: Persist tweet
```

```
    tweet, err := s.persistTweet(authorId, content)
    if err != nil {
        return nil, err // Fail fast if persistence fails
    }

    // NON-CRITICAL PATH: Fanout (best effort)
    err = s.publishFanoutEvent(tweet)
    if err != nil {
        // Log but don't fail request
        logger.Error("Fanout publish failed", zap.Error(err), zap.String("tweetId",
tweet.Id))

        // Enqueue for retry
        s.retryQueue.Add(RetryTask{
            Type:    "fanout",
            TweetId: tweet.Id,
            Attempt: 1,
        })
    }

    return tweet, nil
}

// Retry worker
func (w *RetryWorker) Process(task RetryTask) error {
    if task.Attempt > 5 {
        // Give up after 5 attempts
        logger.Error("Fanout retry exhausted", zap.String("tweetId", task.TweetId))
        return nil
    }

    err := w.publishFanoutEvent(task.TweetId)
    if err != nil {
        // Exponential backoff
        delay := time.Duration(math.Pow(2, float64(task.Attempt))) * time.Second
        w.retryQueue.AddWithDelay(RetryTask{
            Type:    task.Type,
            TweetId: task.TweetId,
            Attempt: task.Attempt + 1,
        }, delay)
    }

    return nil
}
```

Key principle:

- **Persist critical data first** (tweet exists)
- **Background work is best-effort** (fanout can retry)
- **Never fail user request for non-critical failures**

Idempotency in Fanout

Problem: Fanout worker crashes mid-process, Kafka redelivers

```
func (w *FanoutWorker) ProcessTweet(event TweetCreatedEvent) error {
    // Check if already processed
    key := fmt.Sprintf("fanout_processed:%s", event.TweetId)
    exists, _ := w.redis.Exists(ctx, key).Result()

    if exists > 0 {
        // Already processed, skip
        return nil
    }

    // Do fanout work
    err := w.fanoutToFollowers(event)
    if err != nil {
        return err // Kafka will redeliver
    }

    // Mark as processed (TTL = 7 days)
    w.redis.SetEx(ctx, key, "1", 7*24*time.Hour)

    return nil
}
```

RPO/RTO Targets

Component	RPO	RTO	Strategy
Tweets	0	2 min	Synchronous replication
Follows	0	2 min	Synchronous replication
Likes/retweets	10 sec	2 min	Async replication
Timelines	N/A	30 sec	Rebuild from cache/DB
Trending	N/A	5 min	Recompute from stream

---

12. Observability & Operations

Key Metrics

API Gateway:

```
# Request rate
api_requests_total{endpoint="/api/v1/timeline", status="200"}
api_requests_total{endpoint="/api/v1/timeline", status="500"}

# Latency (histogram with percentiles)
api_latency_seconds{endpoint="/api/v1/timeline", quantile="0.5"}
api_latency_seconds{endpoint="/api/v1/timeline", quantile="0.95"}
api_latency_seconds{endpoint="/api/v1/timeline", quantile="0.99"}
```

```
# Rate limiting
api_rate_limited_total{user_id="...", endpoint="..."}
```

Fanout:

```
fanout_tweets_processed_total
fanout_followers_updated_total
fanout_lag_seconds (time between tweet creation and follower timeline update)
```

Kafka:

```
kafka_consumer_lag{topic="tweet.created", partition="0"}
kafka_message_rate{topic="tweet.created"}
```

Database:

```
db_query_duration_seconds{query="SELECT FROM tweets"}
db_connections_active{shard="1"}
db_replication_lag_seconds{shard="1"}
db_deadlocks_total
```

Redis:

```
redis_commands_total{command="GET"}
redis_hit_rate (hits / (hits + misses))
redis_memory_usage_bytes
redis_evicted_keys_total
redis_keyspace_hits
redis_keyspace_misses
```

Alerts

🔴 **Critical (page on-call):**

- API error rate > 5% for 5 minutes
- Database shard down
- Redis cluster down
- Kafka consumer lag > 1M messages
- Fanout lag > 5 minutes
- p99 latency > 5 seconds

🟡 **Warning (investigate in morning):**

- API p95 latency > 1 second
- Cache hit rate < 85%
- Database replication lag > 10 seconds
- Fanout lag > 1 minute

🟢 **Informational:**

- New deploy completed
- Auto-scaling triggered
- Trending topic detected

Structured Logging

```
logger.info({
  event: 'tweet_created',
  tweetId: 'tweet-123',
  authorId: 'user-456',
  fanoutCount: 5000,
  latency_ms: 45,
  cached: true,
  timestamp: Date.now()
});

logger.error({
  event: 'fanout_failed',
  tweetId: 'tweet-789',
  authorId: 'user-999',
  error: err.message,
  stack: err.stack,
  attempt: 3,
  timestamp: Date.now()
});
```

Distributed Tracing

```
import { trace } from '@opentelemetry/api';

async function getTimeline(userId: string) {
  const span = trace.getTracer('twitter').startSpan('get_timeline');
  span.setAttribute('userId', userId);

  try {
    const cachedSpan = trace.getTracer('twitter').startSpan('redis_get_timeline');
    const cached = await redis.zrevrange(`timeline:${userId}`, 0, 19);
    cachedSpan.end();

    if (cached.length > 0) {
      span.setAttribute('cache_hit', true);

      const hydrateSpan = trace.getTracer('twitter').startSpan('hydrate_tweets');
      const tweets = await hydrateTweets(cached);
      hydrateSpan.end();

      return tweets;
    }

    span.setAttribute('cache_hit', false);

    const buildSpan =
trace.getTracer('twitter').startSpan('build_timeline_from_scratch');
    const tweets = await buildTimelineFromScratch(userId);
```

```
    buildSpan.end();

    return tweets;
  } finally {
    span.end();
  }
}
```

Trace example:

```
get_timeline (580ms)
├─ redis_get_timeline (12ms) ✓ cache_hit=true
├─ hydrate_tweets (420ms)
│  ├─ redis_mget_tweets (80ms) - fetched 15/20
│  ├─ db_query_missing_tweets (180ms) - fetched 5
│  └─ enrich_engagement (160ms)
│     ├─ redis_get_likes (50ms)
│     ├─ redis_get_retweets (50ms)
│     └─ check_user_engagement (60ms)
└─ [end]
```

Debugging Runbook

"My timeline is empty!"

1. Check user's following list:

   ```
   redis-cli> SMEMBERS following:user-123
   ```

2. Check timeline cache:

   ```
   redis-cli> ZRANGE timeline:user-123 0 20 WITHSCORES
   ```

3. Check if user is being fanout to:

   ```
   SELECT * FROM fanout_log WHERE follower_id = 'user-123' ORDER BY created_at
   DESC LIMIT 10;
   ```

4. Manually trigger timeline rebuild:

   ```
   curl -X POST https://api.twitter.com/internal/rebuild_timeline \
     -H "X-Admin-Token: ..." \
     -d '{"userId": "user-123"}'
   ```

"Trending topics not updating!"

1. Check Kafka consumer lag:

   ```
   kafka-consumer-groups --describe --group trend-processor
   ```

2. Check trend cache:

```
redis-cli> ZREVRANGE trending:global 0 20 WITHSCORES
```

3. Check stream processor health:

```
curl https://trend-service.twitter.com/health
```

---

13. Security & Abuse Prevention

Authentication

OAuth 2.0 flow:

```
User → Login page → Twitter auth server
                        ↓
              Validates credentials
                        ↓
              Issues access_token (JWT)
                        ↓
          User stores token (localStorage)
                        ↓
Every API request includes:
Authorization: Bearer <access_token>
```

Token verification:

```
import jwt from 'jsonwebtoken';

async function authenticateRequest(req: Request): Promise<User> {
  const token = req.headers.authorization?.replace('Bearer ', '');
  if (!token) throw new UnauthorizedError();

  try {
    // Verify JWT
    const payload = jwt.verify(token, process.env.JWT_SECRET);

    // Check if token is revoked (check Redis blacklist)
    const revoked = await redis.sismember('revoked_tokens', token);
    if (revoked) throw new UnauthorizedError();

    // Load user from cache or DB
    return getUser(payload.userId);
  } catch (err) {
    throw new UnauthorizedError();
  }
}
```

Rate Limiting (Token Bucket)

Tiered limits:

```
const RATE_LIMITS = {
  'tweet_create': {
    free: { tokens: 50, refillRate: 50, refillInterval: 86400 }, // 50/day
    verified: { tokens: 300, refillRate: 300, refillInterval: 86400 }, // 300/day
    api: { tokens: 10000, refillRate: 10000, refillInterval: 86400 } // 10K/day
  },
  'follow': {
    free: { tokens: 400, refillRate: 400, refillInterval: 86400 },
    verified: { tokens: 1000, refillRate: 1000, refillInterval: 86400 }
  },
  'api_read': {
    free: { tokens: 1000, refillRate: 1000, refillInterval: 3600 }, // 1K/hour
    verified: { tokens: 5000, refillRate: 5000, refillInterval: 3600 },
    api: { tokens: 100000, refillRate: 100000, refillInterval: 3600 }
  }
};

async function checkRateLimit(userId: string, action: string, tier: string):
Promise<boolean> {
  const limit = RATE_LIMITS[action][tier];
  const key = `rate:${userId}:${action}`;

  const bucket = await redis.get(key);
  const now = Date.now();

  let tokens = limit.tokens;
  let lastRefill = now;

  if (bucket) {
    const data = JSON.parse(bucket);
    const elapsed = (now - data.lastRefill) / 1000;
    const refillAmount = Math.floor(elapsed / (limit.refillInterval /
limit.refillRate));
    tokens = Math.min(data.tokens + refillAmount, limit.tokens);
    lastRefill = data.lastRefill;
  }

  if (tokens < 1) {
    // Rate limited
    return false;
  }

  tokens -= 1;
  await redis.setex(key, limit.refillInterval, JSON.stringify({tokens,
lastRefill}));
```

```
  return true;
}
```

Abuse Detection & Mitigation

1. Bot detection:

```
async function detectBot(userId: string): Promise<boolean> {
  const signals = await Promise.all([
    checkTweetPatterns(userId),   // Same content, rapid fire
    checkAccountAge(userId),      // Newly created
    checkEngagementRate(userId),  // Tweets but no followers
    checkDeviceFingerprint(userId) // Multiple accounts from same IP
  ]);

  const botScore = calculateBotScore(signals);

  if (botScore > 0.8) {
    await redis.sadd('suspected_bots', userId);
    return true;
  }

  return false;
}

// Check for spam patterns
async function checkTweetPatterns(userId: string): Promise<number> {
  const recentTweets = await db.query(
    'SELECT content FROM tweets WHERE author_id = $1 AND created_at > NOW() -
INTERVAL \'1 hour\' ORDER BY created_at DESC LIMIT 50',
    [userId]
  );

  // Calculate similarity between tweets
  const similarities = [];
  for (let i = 0; i < recentTweets.length - 1; i++) {
    const similarity = levenshteinDistance(recentTweets[i].content,
recentTweets[i+1].content);
    similarities.push(similarity);
  }

  const avgSimilarity = similarities.reduce((a, b) => a + b, 0) /
similarities.length;

  // High similarity = likely bot
  return avgSimilarity;
}
```

2. Spam filtering:

```
async function filterSpam(tweet: Tweet): Promise<boolean> {
  const checks = [
    // Check URL reputation
    checkURLReputation(tweet.content),

    // Check for banned keywords
    containsBannedKeywords(tweet.content),

    // ML model prediction
    mlSpamClassifier.predict(tweet.content)
  ];

  const results = await Promise.all(checks);

  if (results.some(r => r.isSpam)) {
    // Soft delete tweet
    await db.query('UPDATE tweets SET status = $1 WHERE id = $2', ['spam_filtered',
tweet.id]);
    return true;
  }

  return false;
}

async function checkURLReputation(content: string): Promise<{isSpam: boolean}> {
  const urls = extractURLs(content);

  for (const url of urls) {
    // Check Google Safe Browsing API
    const response = await
fetch('https://safebrowsing.googleapis.com/v4/threatMatches:find', {
      method: 'POST',
      body: JSON.stringify({
        client: { clientId: 'twitter', clientVersion: '1.0' },
        threatInfo: {
          threatTypes: ['MALWARE', 'SOCIAL_ENGINEERING'],
          platformTypes: ['ANY_PLATFORM'],
          threatEntryTypes: ['URL'],
          threatEntries: [{ url }]
        }
      })
    });

    const data = await response.json();
    if (data.matches) {
      return { isSpam: true };
    }
  }
```

```
  return { isSpam: false };
}
```

3. Misinformation detection:

```
async function flagMisinformation(tweet: Tweet): Promise<void> {
  // Check against known false claims
  const knownFalseClaims = await redis.smembers('false_claims');

  for (const claim of knownFalseClaims) {
    if (tweet.content.includes(claim)) {
      // Add warning label
      await db.query(
        'UPDATE tweets SET labels = array_append(labels, $1) WHERE id = $2',
        ['disputed', tweet.id]
      );

      // Limit viral spread
      await redis.sadd(`limited_reach:${tweet.id}`, '1');
    }
  }
}
```

4. Account compromise detection:

```
async function detectCompromise(userId: string, newTweet: Tweet): Promise<boolean> {
  // Get user's typical behavior
  const profile = await getUserBehaviorProfile(userId);

  const anomalies = [
    // Tweet from unusual location
    isUnusualLocation(newTweet.ipAddress, profile.usualLocations),

    // Different device
    isNewDevice(newTweet.deviceFingerprint, profile.knownDevices),

    // Language change
    isDifferentLanguage(newTweet.content, profile.usualLanguage),

    // Unusual time of day
    isUnusualTime(newTweet.createdAt, profile.activeHours)
  ];

  const anomalyCount = anomalies.filter(a => a).length;

  if (anomalyCount >= 3) {
    // Lock account, require password reset
    await db.query('UPDATE users SET status = $1 WHERE id = $2', ['locked',
userId]);
    await sendSecurityAlert(userId);
    return true;
```

```
  }

  return false;
}
```

Data Privacy & GDPR

Right to deletion:

```
async function deleteUser(userId: string): Promise<void> {
  // 1. Soft delete user
  await db.query('UPDATE users SET deleted_at = NOW(), email = NULL WHERE id = $1',
[userId]);

  // 2. Delete tweets (cascade to likes, retweets)
  await db.query('DELETE FROM tweets WHERE author_id = $1', [userId]);

  // 3. Remove from social graph
  await db.query('DELETE FROM follows WHERE follower_id = $1 OR followee_id = $1',
[userId]);

  // 4. Purge caches
  await redis.del(`user:${userId}`);
  await redis.del(`timeline:${userId}`);
  await redis.del(`following:${userId}`);
  await redis.del(`followers:${userId}`);

  // 5. Async cleanup (S3, analytics, backups)
  await kafka.publish('user.deleted', { userId });
}
```

--- END OF PASS 2 ---