Twitter System Design - High-Level Design (HLD)

Table of Contents

Problem Statement

Design a microblogging and social networking platform like Twitter that allows users to:

- Post short messages (tweets) up to 280 characters
- Follow other users
- View a personalized timeline
- Like, retweet, and reply to tweets
- Search for tweets, users, and hashtags
- Send direct messages
- View trending topics

Scale Requirements

- **400 million daily active users (DAU)**
- **500 million tweets per day**
- **Billions of timeline requests per day**
- **Average tweet size: 200 bytes (text)**
- **Read-heavy system (1000:1 read-to-write ratio)**
- **Real-time requirements** for timeline updates

Functional Requirements

Must Have (P0)

1. **User Management**
    - User registration and authentication
    - Profile management (bio, profile picture, verification)

    - User search by username/display name

2. **Tweet Operations**
    - Post tweets (280 character limit)
    - Delete tweets
    - Like/unlike tweets
    - Retweet/undo retweet
    - Quote tweet (retweet with comment)
    - Reply to tweets (threaded conversations)

3. **Social Features**
    - Follow/unfollow users
    - Block users
    - Mute users (hide content without unfollowing)
    - View followers/following lists

4. **Timeline**
    - Home timeline (tweets from followed users)
    - User timeline (user's own tweets)
    - Mentions timeline (tweets mentioning user)
    - Chronological and algorithmic sorting

5. **Discovery**
    - Trending topics/hashtags
    - Search tweets by keyword, hashtag
    - Search users
    - Suggested users to follow

Nice to Have (P1)

- Direct messaging between users
- Tweet bookmarking
- Lists (curated groups of users)
- Spaces (audio conversations)
- Twitter Blue (premium features)
- Tweet editing
- Polls
- Threads (connected tweet series)

Non-Functional Requirements

Performance

- **Timeline load time**: < 200ms for p99
- **Tweet post time**: < 100ms

- **Search latency**: < 300ms
- **Timeline updates**: Near real-time (< 5 seconds)

Scalability

- Handle 400M DAU
- Support 500M tweets/day (~6K writes/second)
- Process 500B timeline requests/day (~6M reads/second)
- Support traffic spikes during major events (10x normal)

Availability

- **99.99% uptime** (4 nines)
- Multi-region deployment
- Graceful degradation during failures

Consistency

- **Eventual consistency** for timelines (acceptable)
- **Strong consistency** for tweets and user data
- **Causal consistency** for reply chains

Security

- OAuth 2.0 authentication
- Rate limiting (prevent spam, API abuse)
- Content moderation
- DDOS protection

Capacity Estimation

Traffic Estimates

```
Daily Active Users (DAU): 400M
Tweets per day: 500M
Average timeline refreshes per user: 10/day

Read requests/day: 400M × 10 × 50 tweets = 200B
Read requests/second: 200B / 86400 ≈ 2.3M QPS
Peak read QPS (5x average): ~11.5M QPS

Write requests/day: 500M tweets
Write requests/second: 500M / 86400 ≈ 6K QPS
Peak write QPS (10x during events): ~60K QPS
```

Storage Estimates

```
Average tweet size: 200 bytes (text only)
With metadata (user_id, timestamp, etc.): 500 bytes
Photos/videos (20% of tweets): Additional storage

Daily tweet storage:
  Text: 500M × 500 bytes = 250 GB
  Media (20%): 100M × 500 KB avg = 50 TB
  Total: ~50 TB/day

Yearly storage: 50 TB × 365 = 18.25 PB/year
Storage for 5 years: 91 PB
```

Bandwidth Estimates

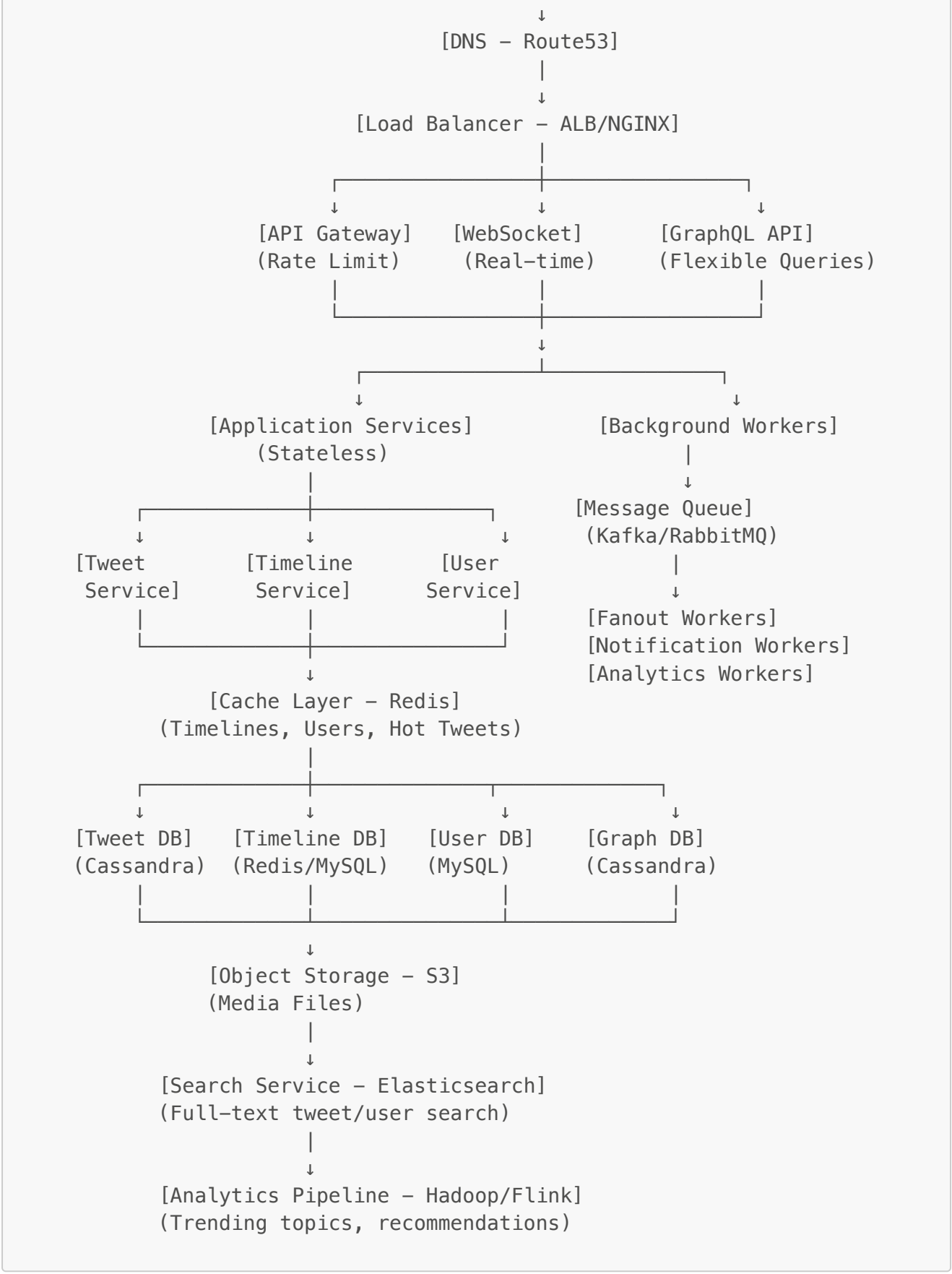
```
Incoming bandwidth:
  50 TB/day / 86400 seconds = 578 MB/s

Outgoing bandwidth (1000:1 read ratio):
  578 MB/s × 1000 = 578 GB/s
```

Memory Estimates (Caching)

```
Cache hot timelines (20% of active users):
  400M × 0.2 = 80M users
  Assume 50 tweets × 500 bytes = 25 KB per timeline
  Memory needed: 80M × 25 KB = 2 TB

Cache hot tweets (recent 2 hours):
  6K QPS × 7200 seconds = 43M tweets
  43M × 500 bytes = 21.5 GB

Cache user profiles (active users):
  80M users × 2 KB = 160 GB

Total cache: ~2.2 TB distributed across 50 servers
```

High-Level Architecture

```
                              [Users - Mobile/Web]
                                       |
                                       ↓
                            [CDN - CloudFront/Fastly]
                           (Images, Videos, Avatars)
                                       |
```



Core Components

1. Load Balancer

Purpose: Distribute traffic across application servers

Configuration:

- **Layer 7 (HTTP/HTTPS)** load balancing
- **Geographic routing** (route to nearest datacenter)
- **Health checks** every 10 seconds
- **SSL termination** at load balancer
- **Connection draining** for graceful shutdowns

Technology: AWS ALB, NGINX, HAProxy

Capacity: Each LB handles ~50K concurrent connections

2. API Gateway

Purpose: Single entry point, authentication, rate limiting

Responsibilities:

- **Authentication**: JWT token validation
- **Rate Limiting**:

```
Per User:
- Tweets: 300/3 hours (100/hour)
- Follows: 400/day
- Likes: 1000/day
- Retweets: 1000/day

Per IP:
- Registration: 10/hour
- Login: 20/hour
- API calls: 15 requests/15 minutes (unauthenticated)
```

- **Request validation** and transformation
- **API versioning** (v1, v2)
- **Metrics collection**

Technology: Kong, AWS API Gateway, custom Express middleware

3. Tweet Service

Purpose: Handle all tweet-related operations

Operations:

- Create tweet (validate length, extract hashtags/mentions)
- Delete tweet
- Get tweet by ID
- Get tweet metadata (likes, retweets, replies count)

Database: Cassandra (write-heavy, time-series data)

Caching: Redis for hot tweets (recent 2 hours)

4. Timeline Service

Purpose: Generate personalized timelines

Challenges:

- **Fan-out problem**: A user with 10M followers generates 10M writes
- **Hot users**: Celebrities create massive write amplification
- **Real-time updates**: Users expect instant timeline updates

Solution: Hybrid fan-out (detailed in Deep Dives section)

5. User Service

Purpose: Manage user accounts and profiles

Operations:

- User registration/authentication
- Profile CRUD
- Follow/unfollow operations
- Block/mute operations

Database: MySQL (ACID transactions for user data)

Caching: Redis for user profiles (1 hour TTL)

6. Search Service

Purpose: Real-time tweet and user search

Features:

- Full-text search on tweets
- Autocomplete for usernames
- Hashtag search
- Advanced filters (from user, date range, engagement)

Technology: Elasticsearch cluster (10+ nodes)

Indexing Strategy:

- Near real-time indexing (< 1 second latency)
- Index tweet text, username, hashtags
- Rank by recency + engagement

7. Notification Service

Purpose: Real-time push notifications

Types:

- Likes, retweets, replies
- New followers
- Mentions in tweets
- Direct messages

Channels:

- WebSocket (for active users)
- Push notifications (APNs/FCM)
- Email (batched, daily digest)

Deduplication: Aggregate similar notifications

8. Trending Topics Service

Purpose: Identify and rank trending hashtags

Algorithm:

```
Trend Score = (Tweet Volume) × (Velocity) × (Novelty)

Where:
- Tweet Volume: Number of tweets with hashtag in last hour
- Velocity: Rate of change (tweets/minute)
- Novelty: Boost for newly emerging topics
```

Update Frequency: Every 5 minutes

Technology: Apache Storm for real-time stream processing

Database Design

1. Tweet Database (Cassandra)

Why Cassandra?

- Write-heavy workload (500M tweets/day)
- Time-series data (sorted by timestamp)
- High availability (no single point of failure)
- Linear scalability

Schema:

```
CREATE TABLE tweets (
    tweet_id uuid PRIMARY KEY,
    user_id uuid,
    content text,
    created_at timestamp,
    type text,  -- ORIGINAL, RETWEET, QUOTE_TWEET, REPLY
    reply_to_tweet_id uuid,
    retweet_of_tweet_id uuid,
    hashtags set<text>,
    mentioned_users set<text>,
    media_urls list<text>,
    like_count counter,
    retweet_count counter,
    reply_count counter
);

CREATE INDEX ON tweets (user_id);
CREATE INDEX ON tweets (created_at);
```

Partitioning: By tweet_id (consistent hashing)

Replication: 3 replicas per datacenter

2. Timeline Database (Redis + MySQL)

Redis (Hot Data - Last 2 days):

```
Timeline Key: "timeline:user:{user_id}"
Type: Sorted Set (ZSET)
Score: Timestamp
Value: Tweet ID

ZADD timeline:user:123 1704672000 tweet-abc
ZADD timeline:user:123 1704672100 tweet-def

Get timeline: ZREVRANGE timeline:user:123 0 49
```

MySQL (Cold Data - Archive):

```
CREATE TABLE timelines (
    user_id CHAR(36),
    tweet_id CHAR(36),
    created_at TIMESTAMP,
    INDEX idx_user_created (user_id, created_at DESC)
) PARTITION BY RANGE (UNIX_TIMESTAMP(created_at));
```

3. User Database (MySQL)

Schema:

```
CREATE TABLE users (
    user_id CHAR(36) PRIMARY KEY,
    username VARCHAR(20) UNIQUE NOT NULL,
    email VARCHAR(255) UNIQUE NOT NULL,
    display_name VARCHAR(50),
    bio VARCHAR(160),
    profile_picture_url VARCHAR(500),
    is_verified BOOLEAN DEFAULT FALSE,
    follower_count INT DEFAULT 0,
    following_count INT DEFAULT 0,
    tweet_count INT DEFAULT 0,
    created_at TIMESTAMP,
    updated_at TIMESTAMP,
    INDEX idx_username (username),
    INDEX idx_email (email)
);

CREATE TABLE follows (
    follower_id CHAR(36),
    followee_id CHAR(36),
    created_at TIMESTAMP,
    PRIMARY KEY (follower_id, followee_id),
    INDEX idx_followee (followee_id)  -- Fast follower lookup
);
```

Sharding: By user_id (consistent hashing)

Replication: Master-slave (1 master, 3 read replicas)

4. Social Graph Database (Cassandra)

Why Separate Graph DB?

- Fast fan-out for timeline generation
- Optimized for relationship queries
- Denormalized for read performance

Schema:

```
-- Follower list (who follows this user)
CREATE TABLE followers (
    user_id uuid,
    follower_id uuid,
    created_at timestamp,
    PRIMARY KEY (user_id, follower_id)
```

```
);

-- Following list (who this user follows)
CREATE TABLE following (
    user_id uuid,
    followee_id uuid,
    created_at timestamp,
    PRIMARY KEY (user_id, followee_id)
);
```

5. Engagement Database (Cassandra)

Schema:

```
CREATE TABLE likes (
    tweet_id uuid,
    user_id uuid,
    created_at timestamp,
    PRIMARY KEY (tweet_id, user_id)
);

CREATE TABLE retweets (
    tweet_id uuid,
    user_id uuid,
    created_at timestamp,
    PRIMARY KEY (tweet_id, user_id)
);

CREATE TABLE replies (
    parent_tweet_id uuid,
    reply_tweet_id uuid,
    created_at timestamp,
    PRIMARY KEY (parent_tweet_id, reply_tweet_id)
);
```

API Design

RESTful API Endpoints

Authentication

```
POST   /api/v1/auth/register
POST   /api/v1/auth/login
POST   /api/v1/auth/logout
POST   /api/v1/auth/refresh
```

User APIs

```
GET    /api/v1/users/{username}
PUT    /api/v1/users/{username}
POST   /api/v1/users/{username}/follow
DELETE /api/v1/users/{username}/follow
GET    /api/v1/users/{username}/followers?cursor={cursor}&limit=20
GET    /api/v1/users/{username}/following?cursor={cursor}&limit=20
POST   /api/v1/users/{username}/block
DELETE /api/v1/users/{username}/block
POST   /api/v1/users/{username}/mute
DELETE /api/v1/users/{username}/mute
```

Tweet APIs

```
POST   /api/v1/tweets
DELETE /api/v1/tweets/{tweet_id}
GET    /api/v1/tweets/{tweet_id}
POST   /api/v1/tweets/{tweet_id}/like
DELETE /api/v1/tweets/{tweet_id}/like
POST   /api/v1/tweets/{tweet_id}/retweet
DELETE /api/v1/tweets/{tweet_id}/retweet
POST   /api/v1/tweets/{tweet_id}/quote
POST   /api/v1/tweets/{tweet_id}/reply
GET    /api/v1/tweets/{tweet_id}/replies?cursor={cursor}&limit=20
```

Timeline APIs

```
GET    /api/v1/timelines/home?cursor={cursor}&limit=50
GET    /api/v1/timelines/user/{username}?cursor={cursor}&limit=50
GET    /api/v1/timelines/mentions?cursor={cursor}&limit=50
```

Search APIs

```
GET    /api/v1/search/tweets?q={query}&cursor={cursor}&limit=20
GET    /api/v1/search/users?q={query}&cursor={cursor}&limit=20
GET    /api/v1/search/hashtags?q={query}&limit=10
```

Trending APIs

```
GET    /api/v1/trends/global
GET    /api/v1/trends/location/{woeid}
```

API Response Format

```
{
  "data": {
    "tweet_id": "1234567890",
    "user": {
      "username": "alice",
      "display_name": "Alice Smith",
      "profile_picture_url": "https://cdn.twitter.com/...",
      "is_verified": true
    },
    "content": "Excited about system design! #Engineering",
    "created_at": "2025-01-08T13:00:00Z",
    "metrics": {
      "like_count": 1523,
      "retweet_count": 342,
      "reply_count": 89,
      "view_count": 50234
    },
    "entities": {
      "hashtags": ["Engineering"],
      "mentions": [],
      "urls": [],
      "media": []
    }
  },
  "meta": {
    "cursor": "next_page_token_xyz",
    "result_count": 50
  }
}
```

Deep Dives

1. Timeline Generation (Fan-out Problem)

Challenge: When user tweets, how to update millions of followers' timelines?

Approach 1: Fan-out on Write (Push)

Process:

```
1. User posts tweet
2. System writes to Tweet DB
3. Fanout service retrieves follower list
4. Push tweet to each follower's timeline (Redis)
5. Followers read pre-computed timelines
```

Pros:

- Fast read (O(1) from cache)
- Real-time timeline updates
- Simple read logic

Cons:

- **Celebrity problem**: User with 10M followers = 10M writes
- Wasted computation for inactive users
- High write amplification
- Slow tweet posting for celebrities

Approach 2: Fan-out on Read (Pull)

Process:

```
1. User posts tweet (stored in DB)
2. Follower requests timeline
3. System queries tweets from all followed users
4. Merge and sort in real-time
5. Return to user
```

Pros:

- Fast write (single DB write)
- No wasted computation
- Works for any follower count

Cons:

- Slow read (query many users)
- Complex merge logic
- High database load
- Not real-time

Hybrid Approach (Twitter's Solution)

For Regular Users (< 1M followers):

- **Fan-out on write** (push model)

- Pre-compute timelines asynchronously
- Store in Redis for fast access

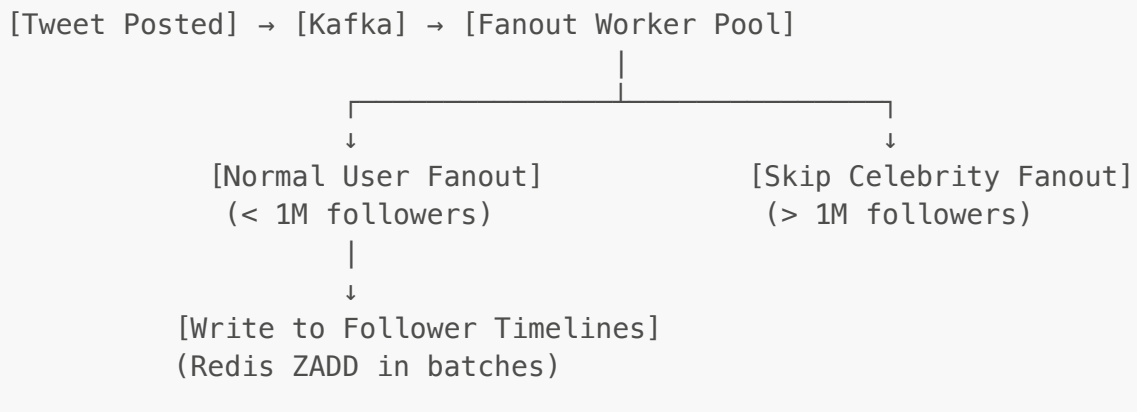
For Celebrities (> 1M followers):

- **Fan-out on read** (pull model)
- Fetch tweets on-demand when followers request
- Cache heavily to reduce DB hits

Implementation:

```
def get_timeline(user_id, limit=50):
    # Part 1: Pre-computed timeline (normal users)
    timeline_tweets = redis.zrevrange(
        f"timeline:{user_id}",
        0,
        limit
    )

    # Part 2: Merge celebrity tweets (on-demand)
    celebrity_following = get_celebrities_following(user_id)
    if celebrity_following:
        celebrity_tweets = fetch_recent_tweets(
            celebrity_following,
            hours=24
        )
        timeline_tweets = merge_and_sort(
            timeline_tweets,
            celebrity_tweets
        )

    # Part 3: Rank by algorithm (if enabled)
    if use_algorithmic_timeline:
        timeline_tweets = rank_tweets(timeline_tweets, user_id)

    return timeline_tweets[:limit]
```

Fan-out Architecture:



2. Tweet Storage & Retrieval

Sharding Strategy:

```
Shard = hash(tweet_id) % num_shards

Why tweet_id not user_id?
- Even distribution (hot users don't overload single shard)
- Better for global timeline/search
- Trade-off: User timeline requires scatter-gather

Optimization for user timeline:
- Maintain user_id → tweet_ids mapping in separate index
- Use Cassandra's wide column for efficient range queries
```

Data Retention:

- **Hot data** (last 7 days): In-memory cache + SSD
- **Warm data** (7-90 days): SSD
- **Cold data** (> 90 days): HDD or archival storage
- **Deleted tweets**: Soft delete with 30-day recovery window

3. Trending Topics Algorithm

Data Collection:

```
[Tweet Stream] → [Kafka] → [Storm/Flink]
                                ↓
                          [Count hashtags per time window]
                                ↓
                          [Calculate trend score]
                                ↓
                          [Update trending cache]
```

Trending Score Formula:

```
Score = (Current Volume / Historical Average) × Time Decay Factor ×
Diversity Factor

Where:
- Current Volume: Tweets in last hour
- Historical Average: Tweets per hour over last 7 days
- Time Decay: 1.0 for last hour, 0.5 for 2 hours ago, etc.
- Diversity Factor: Boost if tweets from many unique users
```

Why This Works:

- Identifies sudden spikes (breaking news)
- Normalizes for naturally popular topics
- Prevents gaming by bot networks
- Time decay keeps trends fresh

Implementation:

```
def calculate_trend_score(hashtag, time_window_hours=1):
    current_count = count_tweets(hashtag, hours=time_window_hours)
    historical_avg = get_historical_average(hashtag, days=7)
    unique_users = count_unique_users(hashtag, hours=time_window_hours)

    # Avoid division by zero
    if historical_avg == 0:
        historical_avg = 1

    volume_ratio = current_count / historical_avg
    diversity_factor = min(unique_users / 100, 2.0)  # Cap at 2x
    time_decay = 1.0  # For simplicity, can add decay

    score = volume_ratio * diversity_factor * time_decay
    return score
```

4. Tweet Search

Elasticsearch Architecture:

```
[Tweet Posted] → [Kafka] → [Search Indexer] → [Elasticsearch]
                                                      |
                                                [Index with mapping]
                                                {
                                                  "content": "text",
                                                  "user_id": "keyword",
                                                  "created_at": "date",
                                                  "hashtags": "keyword",
                                                  "engagement": "integer"
                                                }
```

Search Query Example:

```
{
  "query": {
    "bool": {
      "must": [
        { "match": { "content": "system design" } }
```

```
        ],
        "filter": [
          { "range": { "created_at": { "gte": "now-7d" } } },
          { "term": { "hashtags": "engineering" } }
        ]
      }
    },
    "sort": [
      { "created_at": "desc" }
    ],
    "size": 20
  }
```

Ranking Factors:

1. Recency (70% weight)
2. Engagement (20% weight)
3. User relevance (10% weight)

5. Rate Limiting

Implementation: Token Bucket Algorithm in Redis

Token Bucket Algorithm:

```python
class TokenBucket:
    def __init__(self, capacity, refill_rate):
        self.capacity = capacity  # Max tokens
        self.tokens = capacity
        self.refill_rate = refill_rate  # Tokens per second
        self.last_refill = time.time()

    def consume(self, tokens=1):
        self.refill()
        if self.tokens >= tokens:
            self.tokens -= tokens
            return True
        return False  # Rate limit exceeded

    def refill(self):
        now = time.time()
        elapsed = now - self.last_refill
        new_tokens = elapsed * self.refill_rate
        self.tokens = min(self.capacity, self.tokens + new_tokens)
        self.last_refill = now
```

Redis Implementation:

```python
def is_rate_limited(user_id, action, capacity, rate):
    key = f"rate_limit:{user_id}:{action}"

    # Use Redis for distributed rate limiting
    current = redis.get(key)
    if current is None:
        redis.setex(key, 3600, capacity - 1)
        return False

    if int(current) > 0:
        redis.decr(key)
        return False

    return True  # Rate limited
```

6. Direct Messaging

Architecture:

```
[Sender] → [WebSocket] → [Message Service] → [Cassandra]
                                ↓
                          [Push to Receiver]
                                ↓
                          [WebSocket/Push Notification]
```

Database Schema (Cassandra):

```
CREATE TABLE messages (
    conversation_id uuid,
    message_id timeuuid,
    sender_id uuid,
    receiver_id uuid,
    content text,
    created_at timestamp,
    is_read boolean,
    PRIMARY KEY (conversation_id, message_id)
) WITH CLUSTERING ORDER BY (message_id DESC);

CREATE TABLE conversations (
    conversation_id uuid PRIMARY KEY,
    participant_1_id uuid,
    participant_2_id uuid,
    last_message_id timeuuid,
    updated_at timestamp
);

-- Index for user's conversations
```

```
CREATE INDEX ON conversations (participant_1_id);
CREATE INDEX ON conversations (participant_2_id);
```

Features:

- Real-time delivery via WebSocket
- Read receipts
- Message history pagination
- Typing indicators (ephemeral, not stored)

7. Media Storage

Upload Flow:

```
1. Client requests upload URL
2. API generates pre-signed S3 URL (expires in 15 min)
3. Client uploads directly to S3
4. Client confirms upload with tweet content
5. Background worker processes image:
   - Generate thumbnails (150px, 400px, 1200px)
   - Compress (WebP format)
   - Store processed versions in S3
6. Update tweet with media URLs
7. Invalidate CDN cache if needed
```

Storage Structure:

```
s3://twitter-media/
  ├── images/
  │   ├── original/{tweet_id}.jpg
  │   ├── large/{tweet_id}_1200.webp
  │   ├── medium/{tweet_id}_400.webp
  │   └── thumb/{tweet_id}_150.webp
  └── videos/
      ├── original/{tweet_id}.mp4
      └── processed/{tweet_id}_720p.mp4
```

CDN Strategy:

- CloudFront in front of S3
- Cache hot images (24 hour TTL)
- Push popular content to edge locations
- ~95% cache hit ratio

Scalability & Reliability

Horizontal Scaling

Stateless Services:

- All application servers are stateless
- No session affinity required
- Can scale up/down freely based on load

Auto-scaling Rules:

```
Scale Up:
- CPU > 70% for 5 minutes
- Request queue > 1000

Scale Down:
- CPU < 30% for 15 minutes
- Request queue < 100
- Minimum 10 instances always running
```

Database Scaling:

- **Cassandra**: Add nodes linearly (scales writes)
- **MySQL**: Read replicas (scales reads)
- **Redis**: Redis Cluster with sharding

High Availability

Multi-Region Architecture:

```
Region           Traffic    Purpose
us-east-1        40%        Primary (Americas)
us-west-2        20%        Secondary (Americas)
eu-west-1        25%        Europe
ap-southeast-1   15%        Asia-Pacific

Each region has:
- Complete application stack
- Local database replicas
- Local cache layer
- Cross-region replication for MySQL
```

Failover Strategy:

1. **Automated health checks** every 10 seconds
2. **Circuit breaker** pattern for service failures
3. **Graceful degradation**:
    - Timeline service down → Show cached timeline

- Search down → Disable search, keep core features
- Notification service down → Queue for later delivery

Load Balancing

Multi-Level Load Balancing:

```
Level 1: DNS (Route53)
  - Geographic routing
  - Health-based routing

Level 2: Global Load Balancer
  - Distribute across regions
  - Handle DDOS at edge

Level 3: Application Load Balancer
  - Distribute across app servers
  - SSL termination
  - Health checks
```

Algorithm: Weighted round-robin with health checks

Caching Strategy

Multi-Layer Caching:

L1 - Browser Cache:

- User profiles (5 minutes)
- Static assets (24 hours)

L2 - CDN Cache:

- Media files (7 days)
- User avatars (1 day)

L3 - Application Cache (Redis):

```
Hot Timelines:
  Key: timeline:user:{user_id}
  Type: Sorted Set
  TTL: 2 days

Hot Tweets:
  Key: tweet:{tweet_id}
  Type: Hash
  TTL: 2 hours

User Profiles:
```

```
  Key: user:{user_id}
  Type: Hash
  TTL: 1 hour

Engagement Counts:
  Key: engagement:{tweet_id}
  Type: Hash (like_count, retweet_count, reply_count)
  TTL: 5 minutes
```

Cache Invalidation:

- **Lazy invalidation** for most data
- **Active invalidation** for user profile updates
- **TTL-based** for engagement counts

Detailed Component Interactions

Post Tweet Flow

```
1. User submits tweet via mobile app
2. API Gateway validates JWT token
3. Rate Limiter checks tweet limit (100/hour)
4. Tweet Service:
   - Validates length (≤ 280 chars)
   - Extracts hashtags (#...) and mentions (@...)
   - Generates unique tweet_id (Snowflake ID)
   - Writes to Cassandra
5. Publish event to Kafka: "tweet.created"
6. Return success to user (< 100ms)

Async Processing:
7. Fanout Worker:
   - Check follower count
   - If < 1M: Push to all follower timelines (Redis)
   - If > 1M: Skip fanout (pull on read)
8. Notification Worker:
   - Send notifications to mentioned users
   - Send push notifications to active followers
9. Search Indexer:
   - Index tweet in Elasticsearch
10. Trending Worker:
    - Update hashtag counts
    - Recalculate trending scores
11. Analytics Worker:
    - Log event to data warehouse
```

View Timeline Flow

```
1. User opens Twitter app
2. App calls GET /api/v1/timelines/home
3. API Gateway authenticates user
4. Timeline Service:
   - Check Redis cache for timeline
   - If cache hit: Return immediately (< 50ms)
   - If cache miss:
     a. Get following list from cache/DB
     b. Check which users are celebrities
     c. Fetch normal users' tweets from Redis
     d. Fetch celebrity tweets from Cassandra
     e. Merge and sort by timestamp
     f. Cache result in Redis (2 day TTL)
5. For each tweet:
   - Fetch user data from cache
   - Fetch engagement counts from cache
   - Fetch media URLs from CDN
6. Return paginated response
7. App renders timeline
```

Like Tweet Flow

```
1. User taps heart icon
2. Optimistic UI update (instant feedback)
3. API call: POST /api/v1/tweets/{id}/like
4. Tweet Service:
   - Write to likes table (Cassandra)
   - Increment like count (Redis counter)
   - Publish event to Kafka
5. Notification Worker:
   - Send notification to tweet author
6. Analytics Worker:
   - Update engagement metrics
7. Return success (< 50ms)
```

Search Tweet Flow

```
1. User types in search box
2. Autocomplete: GET /api/v1/search/autocomplete?q={prefix}
   - Query Elasticsearch for prefix matches
   - Return suggestions (< 100ms)
3. User submits search
4. Search Service:
   - Parse query (keywords, filters, operators)
   - Build Elasticsearch query
   - Execute search
   - Rank results by recency + engagement
```

```
5. Return top 20 results with cursor
6. User scrolls: Load more with cursor token
```

Performance Optimizations

1. Database Query Optimization

Bad: N+1 Query Problem

```
-- Get tweets (1 query)
SELECT * FROM tweets WHERE user_id IN (following_list) LIMIT 50;

-- Then for each tweet, get user data (50 queries!)
SELECT * FROM users WHERE user_id = ?;

-- Then for each tweet, get engagement (50 queries!)
SELECT COUNT(*) FROM likes WHERE tweet_id = ?;
```

Good: Single Query with JOINs

```
SELECT
    t.*,
    u.username, u.display_name, u.profile_picture_url,
    COUNT(DISTINCT l.user_id) as like_count,
    COUNT(DISTINCT r.user_id) as retweet_count
FROM tweets t
JOIN users u ON t.user_id = u.user_id
LEFT JOIN likes l ON t.tweet_id = l.tweet_id
LEFT JOIN retweets r ON t.tweet_id = r.tweet_id
WHERE t.user_id IN (following_list)
GROUP BY t.tweet_id
ORDER BY t.created_at DESC
LIMIT 50;
```

2. Pagination

Cursor-Based Pagination (Better than offset):

```
Why not OFFSET?
- OFFSET 1000000 requires scanning 1M rows
- Performance degrades with large offsets
- Inconsistent results if data changes

Cursor-Based:
- Use last seen tweet_id as cursor
```

```
- WHERE tweet_id > {cursor} ORDER BY created_at DESC
- Consistent, fast, no scanning
```

Implementation:

```python
def get_timeline_page(user_id, cursor=None, limit=50):
    if cursor:
        tweets = db.query("""
            SELECT * FROM timeline_cache
            WHERE user_id = ? AND tweet_id > ?
            ORDER BY created_at DESC
            LIMIT ?
        """, user_id, cursor, limit)
    else:
        tweets = db.query("""
            SELECT * FROM timeline_cache
            WHERE user_id = ?
            ORDER BY created_at DESC
            LIMIT ?
        """, user_id, limit)

    next_cursor = tweets[-1]['tweet_id'] if tweets else None
    return tweets, next_cursor
```

3. Database Indexing

```sql
-- Critical indexes for performance
CREATE INDEX idx_tweets_user_created ON tweets(user_id, created_at
DESC);
CREATE INDEX idx_tweets_created ON tweets(created_at DESC);
CREATE INDEX idx_likes_tweet ON likes(tweet_id);
CREATE INDEX idx_follows_followee ON follows(followee_id);
CREATE INDEX idx_hashtags ON tweets USING GIN(hashtags);  -- PostgreSQL
full-text
```

4. Connection Pooling

- **PgBouncer** for PostgreSQL (pool size: 100)
- **Redis connection pooling** (min: 10, max: 50)
- **HTTP connection pooling** (keep-alive enabled)
- Prevents connection exhaustion

5. Batch Operations

```python
# Bad: Individual writes for each follower
for follower_id in followers:
    redis.zadd(f"timeline:{follower_id}", tweet_id, timestamp)

# Good: Pipeline batching
pipeline = redis.pipeline()
for follower_id in followers:
    pipeline.zadd(f"timeline:{follower_id}", tweet_id, timestamp)
pipeline.execute()  # Single network round-trip
```

Failure Scenarios & Mitigation

Scenario 1: Database Master Failure

Impact: Cannot write tweets/follows
Mitigation:

- Automatic failover to slave (promote to master)
- Use Patroni for orchestration
- Queue writes in Kafka during failover
- RTO: < 30 seconds

Scenario 2: Redis Cluster Failure

Impact: Timeline/cache unavailable, slower reads
Mitigation:

- Redis Sentinel for automatic failover
- Fall back to database queries
- Serve stale cached data if available
- Impact: 5-10x slower response time

Scenario 3: Kafka Queue Backlog

Impact: Delayed fanout, slow notifications
Mitigation:

- Scale up consumer workers
- Prioritize critical events (DMs > likes)
- Drop low-priority events if backlog > threshold
- Alert ops team for manual intervention

Scenario 4: Celebrity Tweet Storm

Impact: Fanout service overwhelmed
Mitigation:

- Rate limit fanout writes (max 10K/sec)

- Use fan-out on read for celebrities
- Progressive fanout over 5-10 minutes
- Cache celebrity's recent tweets heavily

Scenario 5: Search Service Down

Impact: Search functionality unavailable
Mitigation:

- Disable search UI gracefully
- Show cached trending topics
- Core features (timeline, tweet, like) still work
- Rebuild index from database backups

Security

Authentication & Authorization

JWT Tokens:

```
{
  "user_id": "uuid",
  "username": "alice",
  "roles": ["user"],
  "exp": 1704672000,
  "iat": 1704585600
}
```

- Access token: 1 hour expiry
- Refresh token: 7 days expiry
- Rotate refresh tokens on use

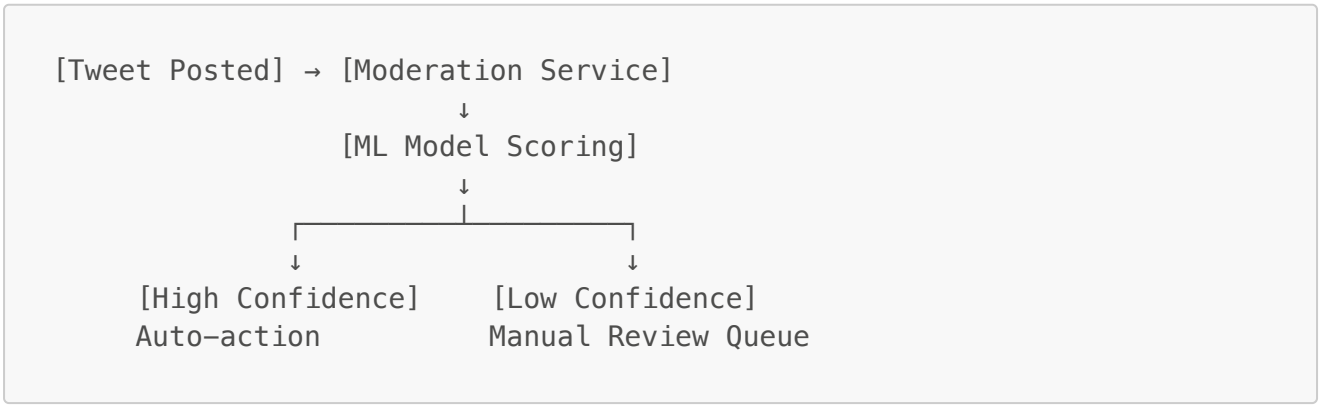
OAuth 2.0 for third-party apps

Content Moderation

Multi-Layer Approach:

1. **Automated**: AI/ML models detect:
    - Hate speech
    - Spam
    - NSFW content
    - Misinformation
2. **User reporting**: Crowd-sourced flagging
3. **Manual review**: Human reviewers for edge cases
4. **Actions**: Warning, suspension, permanent ban

Implementation:



Rate Limiting Implementation

Per-User Limits (Token Bucket):

```
Tweets: 100/hour
Follows: 400/day
Likes: 1000/day
DMs: 500/day
API calls: 1500/15-min window
```

Per-IP Limits (Sliding Window):

```
Registration: 10/hour
Login: 20/hour
Anonymous API: 15/15-min window
```

Implementation in Redis:

```python
def check_rate_limit_sliding_window(user_id, action, limit,
window_seconds):
    key = f"rate:{user_id}:{action}"
    now = time.time()

    # Remove old entries outside window
    redis.zremrangebyscore(key, 0, now - window_seconds)

    # Count current requests
    current_count = redis.zcard(key)

    if current_count < limit:
        # Add current request
        redis.zadd(key, {str(uuid.uuid4()): now})
        redis.expire(key, window_seconds)
        return True  # Allowed
```

```
    return False  # Rate limited
```

Trade-offs & Alternatives

1. SQL vs NoSQL

Chose: Hybrid

- **MySQL** for users (ACID, relationships)
- **Cassandra** for tweets (write-heavy, time-series)
- **Redis** for cache (in-memory speed)

Alternative: Pure MySQL

- Simpler stack
- Easier transactions
- Doesn't scale for Twitter's write volume

2. Push vs Pull Timeline

Chose: Hybrid

- Push for normal users (fast read)
- Pull for celebrities (avoid write amplification)

Alternative: Pure Push

- Consistent experience
- Doesn't work for celebrities
- Too much wasted computation

3. Real-time vs Batch Processing

Chose: Both

- Real-time: Timeline updates, notifications (< 5 sec)
- Batch: Analytics, trending topics (5 min intervals)

Alternative: Pure Real-time

- Better user experience
- Higher infrastructure cost
- More complex systems

4. Monolith vs Microservices

Chose: Microservices

- Independent scaling

- Team autonomy
- Technology flexibility

Alternative: Monolith

- Simpler initially
- Easier debugging
- No network latency between services

System Design Patterns

1. **Fan-out Pattern**

- Distribute single tweet to many timelines
- Used in timeline generation
- Optimized with hybrid push-pull

2. **CQRS (Command Query Responsibility Segregation)**

- Separate write and read models
- Writes to Cassandra, reads from Redis
- Optimized for each use case

3. **Event Sourcing**

- All changes as events in Kafka
- Can replay events to rebuild state
- Useful for audit and analytics

4. **Circuit Breaker**

- Prevent cascading failures
- Fail fast when service down
- Automatic recovery attempts

5. **Bulkhead Pattern**

- Isolate resources by feature
- Separate thread pools for tweets vs DMs
- Prevent one feature from consuming all resources

Monitoring & Observability

Metrics to Track

Application Metrics:

- Request rate (QPS)

- Error rate (%)
- Latency (p50, p95, p99, p999)
- Success rate by endpoint

Business Metrics:

- DAU, WAU, MAU
- Tweets per day
- Engagement rate (likes/tweets)
- Timeline load time

Infrastructure Metrics:

- CPU, memory, disk utilization
- Network I/O
- Database connection pool
- Cache hit ratio

Tools:

- **Metrics**: Prometheus, Datadog, CloudWatch
- **Logging**: ELK Stack (Elasticsearch, Logstash, Kibana)
- **Tracing**: Jaeger, Zipkin
- **Alerting**: PagerDuty

Distributed Tracing

```
[Request] → [API Gateway] → [Tweet Service] → [DB]
   |               |                |               |
   └───────────────┴────────────────┴───────────────┘
                        ↓
              [Trace with span IDs]
              {
                "trace_id": "abc123",
                "spans": [
                  {"service": "api-gateway", "duration": "5ms"},
                  {"service": "tweet-service", "duration": "30ms"},
                  {"service": "cassandra", "duration": "20ms"}
                ]
              }
```

Cost Optimization

Compute Costs

```
Optimization Strategies:
1. Auto-scaling: Scale down 70% during off-peak (save 40%)
```

```
2. Spot instances: Use for non-critical batch jobs (save 70%)
3. Reserved instances: 1-year commit for baseline (save 30%)
4. Right-sizing: Match instance types to workload

Monthly Savings: ~$500K on $2M compute bill
```

Storage Costs

```
Current: 91 PB over 5 years

Optimization:
1. Compress old tweets: Save 50% (Gzip)
2. Archive to Glacier after 1 year: Save 80% on old data
3. Delete spam/bot tweets: Save 10%

Total Savings: ~60% on storage costs
```

Bandwidth Costs

```
Optimization:
1. CDN caching: 95% cache hit = 95% bandwidth savings
2. Image compression (WebP): 30% smaller
3. Response compression (gzip): 70% smaller text
4. Edge caching for API responses

Monthly Savings: ~$200K on $400K bandwidth bill
```

Technology Stack Summary

Layer	Technology	Purpose
CDN	CloudFront, Fastly	Media delivery
Load Balancer	AWS ALB, NGINX	Traffic distribution
API Gateway	Kong, AWS API Gateway	Auth, rate limiting
App Servers	Node.js, Python, Go	Business logic
Cache	Redis Cluster	Hot data storage
SQL Database	MySQL (RDS)	User data
NoSQL Database	Cassandra	Tweets, timeline, graph
Object Storage	Amazon S3	Media files

Layer	Technology	Purpose
Message Queue	Apache Kafka	Event streaming
Search	Elasticsearch	Full-text search
Stream Processing	Apache Storm, Flink	Real-time analytics
Batch Processing	Hadoop, Spark	Trending topics
Monitoring	Prometheus, Datadog	Metrics & alerts
Logging	ELK Stack	Centralized logs
Tracing	Jaeger	Distributed tracing
Container	Docker, Kubernetes	Orchestration

Twitter's Actual Architecture (Historical)

Early Days (2006-2008)

- **Monolithic Ruby on Rails** application
- **MySQL** for all data
- **Memcached** for caching
- Frequent outages due to scalability issues

Evolution (2009-2012)

- Moved to **service-oriented architecture**
- Introduced **Cassandra** for tweets
- Implemented **Gizzard** (custom sharding framework)
- Built **FlockDB** for social graph

Modern Architecture (2020+)

- **Manhattan** (distributed database)
- **GraphJet** (real-time graph processing)
- **Heron** (stream processing, successor to Storm)
- **Mesos** + **Aurora** (cluster management)

Advanced Features

1. Tweet ID Generation (Snowflake)

Requirements:

- Unique across all shards
- Sortable by time
- 64-bit integer

- Generate 10K+ IDs per second

Snowflake ID Structure (64 bits):

```
| 1 bit | 41 bits | 10 bits | 12 bits |
|unused|timestamp| machine |sequence |
|   0  | ms since| ID (1024|per ms   |
|      | epoch   | machines|(4096)   |
```

Benefits:

- K-sortable (sortable by time)
- Roughly chronological
- No coordination needed
- 4096 IDs per millisecond per machine

Implementation:

```python
class SnowflakeIDGenerator:
    def __init__(self, machine_id):
        self.epoch = 1288834974657  # Twitter epoch
        self.machine_id = machine_id
        self.sequence = 0
        self.last_timestamp = -1

    def generate(self):
        timestamp = int(time.time() * 1000)

        if timestamp == self.last_timestamp:
            self.sequence = (self.sequence + 1) & 4095
            if self.sequence == 0:
                # Wait for next millisecond
                while timestamp <= self.last_timestamp:
                    timestamp = int(time.time() * 1000)
        else:
            self.sequence = 0

        self.last_timestamp = timestamp

        # Combine: timestamp(41) | machine_id(10) | sequence(12)
        id = ((timestamp - self.epoch) << 22) | \
             (self.machine_id << 12) | \
             self.sequence

        return id
```

2. Timeline Ranking Algorithm

Engagement Score:

```
def calculate_engagement_score(tweet):
    likes = tweet.like_count
    retweets = tweet.retweet_count * 2  # Retweets valued higher
    replies = tweet.reply_count * 3     # Replies valued highest

    # Recency decay: newer tweets ranked higher
    hours_old = (now - tweet.created_at).total_seconds() / 3600
    recency_factor = 1 / (1 + hours_old)  # Exponential decay

    # User affinity: boost tweets from frequently engaged users
    affinity = get_user_affinity(viewer_id, tweet.user_id)

    score = (likes + retweets + replies) * recency_factor * affinity
    return score
```

3. Content Recommendation

Collaborative Filtering:

- Users who liked A also liked B
- Train on user-tweet interaction matrix
- Update model daily

Content-Based Filtering:

- Similar to tweets user previously engaged with
- Use TF-IDF for text similarity
- Image similarity for media tweets

Additional Considerations

1. Data Privacy & Compliance

- **GDPR**: Right to deletion, data export
- **CCPA**: California privacy rights
- **Tweet deletion**: Permanent deletion after 30 days
- **Data retention**: 5 years for audit

2. Analytics Pipeline

```
[User Actions] → [Kafka] → [Flink] → [Data Warehouse]
                                        (Redshift)
                                            ↓
                                      [BI Dashboards]
                                      (Tableau, Looker)
```

Metrics Tracked:

- User engagement (DAU, time spent)
- Content metrics (viral tweets, engagement rate)
- A/B test results
- Revenue metrics (if applicable)

3. A/B Testing

- Feature flags for gradual rollouts
- User bucketing (by user_id hash)
- Metrics tracking per experiment
- Statistical significance testing

Interview Talking Points

Key Design Decisions

1. **Why hybrid fan-out?** Solves celebrity problem, balances read/write performance
2. **Why Cassandra for tweets?** Write-heavy, time-series, highly available
3. **Why Redis for timelines?** Fast sorted sets, O(1) access, TTL support
4. **Why separate graph DB?** Optimized for relationship queries, fast fan-out
5. **Why token bucket for rate limiting?** Fair, prevents burst abuse, easy to implement
6. **Why Elasticsearch for search?** Full-text search, fast, scalable
7. **Why Kafka for events?** High throughput, durability, replay capability

Potential Bottlenecks & Solutions

1. **Database write bottleneck** → Cassandra with linear scaling
2. **Timeline generation slow** → Pre-compute with hybrid fan-out
3. **Celebrity problem** → Fan-out on read for high-follower users
4. **Search performance** → Elasticsearch with proper sharding
5. **Rate limiting** → Distributed token bucket in Redis
6. **Real-time updates** → WebSocket + Kafka streaming

References & Further Reading

Official Resources

1. **Twitter Engineering Blog** - https://blog.twitter.com/engineering
2. **Twitter Snowflake ID** - https://blog.twitter.com/engineering/en_us/a/2010/announcing-snowflake
3. **Timelines at Scale** - InfoQ presentation by Twitter
4. **FlockDB** - Twitter's distributed graph database

System Design Articles

1. **System Design Primer** - GitHub (donnemartin)
2. **High Scalability: Twitter Architecture**
3. **How Twitter Stores 250M Tweets/Day Using MySQL**
4. **The Architecture Twitter Uses to Deal with 150M Active Users**

Academic Papers

1. **Snowflake: A Network-Oriented Approach to Unique ID Generation**
2. **Manhattan: Twitter's Real-Time Database**
3. **Cassandra: A Decentralized Structured Storage System**

Books

1. **Designing Data-Intensive Applications** - Martin Kleppmann
2. **System Design Interview** - Alex Xu (Volume 1 & 2)
3. **Building Microservices** - Sam Newman

Appendix

QPS Calculations

```
Tweets/day: 500M
Seconds/day: 86400
Average QPS: 500M / 86400 ≈ 6K

Timeline requests/day: 400M × 10 = 4B
Average read QPS: 4B / 86400 ≈ 46K

During major events (World Cup, Elections):
Peak write QPS: 6K × 10 = 60K
Peak read QPS: 46K × 10 = 460K
```

Storage Growth

```
Year 1: 18.25 PB
Year 2: 36.5 PB  (cumulative)
Year 3: 54.75 PB
Year 4: 73 PB
Year 5: 91.25 PB

With compression & lifecycle policies:
Effective storage: ~55 PB (40% savings)
```

Latency Targets

```
Operation              Target    P99
Post tweet            < 100ms   < 200ms
Load timeline         < 200ms   < 500ms
Like/retweet          < 50ms    < 100ms
Search                < 300ms   < 600ms
Send DM               < 100ms   < 200ms
Load trending topics  < 100ms   < 200ms
```

Document Version: 1.0
Last Updated: January 8, 2025
Author: System Design Interview Prep
Status: Complete & Interview-Ready ✅