Scaling Strategy for Tinder-like Application

1. System Overview

A Tinder-like application must efficiently handle a **high read-heavy workload**, optimize for **low-latency user interactions**, and ensure **high availability and scalability**. Given the assumptions:

- **80 million MAU (Monthly Active Users)**
- **20 million DAU (Daily Active Users)**
- **100 million Peak DAU** (5x DAU for high availability)
- **QPS (API request estimation):** 3,472 QPS
- **Storage QPS:** 80,000 QPS

This document outlines a **FAANG-worthy scaling strategy** to ensure the application scales seamlessly under peak load conditions.

2. Database Scaling Strategy

Optimized Sharding Strategy

- **Geographical Partitioning**: Users are sharded based on their location (e.g., country/state/city). This ensures that users match with others within their geographical proximity, reducing cross-region data fetch latency.
- **User ID-Based Sharding**: Within a geographical region, further sharding is done using **User ID hashing** to distribute load evenly across database nodes.
- **Hot Users Handling**:
  - Popular profiles (celebrities, influencers) receive excessive matches/messages, leading to data skew.
  - Solution: Store hot users in a **separate high-performance database** and use a **consistent hashing technique**to distribute their profiles across multiple shards dynamically.
  - **Redis-backed caching layer** for frequently accessed hot profiles to minimize database reads.

3. Caching Strategy

Multi-Layered Caching

- **CDN (Content Delivery Network)**: Used for serving **static content** like profile images, JavaScript, CSS.
- **Edge Caching**:
  - Implement **geo-distributed cache** using **CloudFront / Fastly**.

    - Cache **frequent searches** and **match results** at the edge to reduce database hits.
- **Application-Level Caching (Redis / Memcached)**:
    - **Profile Data**: Store **user profile** details in a Redis cache with **TTL-based expiration**.
    - **Match Lists**: Recent match lists are stored in cache to avoid repeated database queries.
    - **Swipe Actions**: Cache last **few hundred swipes** per user in Redis to reduce storage QPS.
- **Write-Through vs. Write-Back**:
    - **Write-Through Cache** for **profile updates**, ensuring consistency with the database.
    - **Write-Back Cache** for non-critical operations like user preferences.

4. Scaling for High Availability

Hybrid Approach: Replication + Sharding

- **Primary-Replica Replication**:
    - Read-heavy queries served via **read replicas** to handle high read QPS.
    - **Eventual consistency** is maintained via async replication to reduce write bottlenecks.
- **Auto-Scaling Database Clusters**:
    - DynamoDB/Cassandra for NoSQL-based user profiles & match preferences.
    - PostgreSQL/MySQL for transactional data like payments.

Load Balancing & Rate Limiting

- **Global Load Balancer (GLB)**:
    - Routes traffic based on region to the nearest data center.
- **API Gateway with Rate Limiting**:
    - Limits API calls based on **subscription plans** (e.g., Tinder Gold users get more requests).

5. Handling Hot Users Problem

- **Separate Hot User Cluster**:
    - Profiles exceeding a threshold of requests per second are **offloaded to separate DB clusters**.
    - **Dedicated in-memory Redis cache** to store hot user profiles.
- **Adaptive Load Distribution**:
    - **AI-based predictive load balancing** to redirect traffic dynamically to less-loaded servers.
- **Bloom Filters**:

    - Prevent redundant requests for the same hot user by storing **recent requests in a Bloom filter**.

6. Optimized Bandwidth & Resource Estimation

- **Outbound Data Estimation**:
    - Profile pictures (300KB avg per user, 7 images per user) + text data.
    - Match list responses, swipe actions, chat messages.
- **CPU & Memory Optimization**:
    - **CPU Cores** required based on request processing time estimation.
    - **Memory-based caching strategies** to reduce database pressure.