Leaderboard System Design - High-Level Design Focus

Table of Contents

1. Problem Statement & Requirements
2. High-Level Architecture
3. Core Components
4. Data Model & Storage Design
5. Real-time Updates & Data Flow
6. Ranking Algorithms
7. Distributed System Design
8. Scalability & Performance
9. Caching Strategy
10. Fault Tolerance & Reliability
11. Design Trade-offs
12. Real-World Case Studies

Problem Statement & Requirements

Problem Definition

Design a scalable leaderboard system that can:

- Handle millions of users with real-time score updates
- Support multiple leaderboard types (global, regional, friends)
- Provide fast rank lookups (< 100ms)
- Handle 50,000+ score updates/second
- Display top N players efficiently
- Support time-based leaderboards (daily, weekly, monthly, all-time)

Functional Requirements

1. Core Features

- **Score Updates**: Users can update their scores in real-time
- **Rank Queries**: Get a user's current rank instantly
- **Top N Queries**: Retrieve top N users (e.g., top 100)
- **Range Queries**: Get users ranked between M and N
- **User's Neighborhood**: Show users around a specific user's rank
- **Multiple Leaderboards**: Global, friends, regional, by game level

2. Leaderboard Types

```
Time-based:
├── Real-time (continuous)
├── Daily (resets at midnight)
├── Weekly (resets Monday 00:00)
├── Monthly (resets 1st of month)
└── All-time (never resets)

Scope-based:
├── Global (all users)
├── Regional (by country/continent)
├── Friends (social graph)
├── Guild/Team (group-based)
└── Game Level (per level/stage)
```

3. Advanced Features

- **Historical Snapshots**: View past leaderboard states
- **Percentile Ranks**: "You're in top 5% of players"
- **Achievements**: Milestone-based badges
- **Decay System**: Scores decay over time (for engagement)
- **Tie-breaking**: Handle identical scores fairly

Non-Functional Requirements

1. Performance

- **Score Update Latency**: < 50ms (P95)
- **Rank Query Latency**: < 100ms (P95)
- **Top N Query Latency**: < 200ms (P95)
- **Throughput**: Support 50K+ updates/sec

2. Scalability

- **Users**: 100M+ registered users
- **Active Users**: 10M+ DAU
- **Concurrent Updates**: 50K+ simultaneous score updates
- **Read-Heavy**: 100:1 read-to-write ratio

3. Availability & Reliability

- **Uptime**: 99.9% availability
- **Data Durability**: No score loss
- **Consistency**: Eventual consistency acceptable (< 1 second lag)
- **Fault Tolerance**: Handle node failures gracefully

4. Cost Efficiency

- **Storage**: Optimize for large user base
- **Compute**: Balance real-time vs batch processing
- **Network**: Minimize cross-region data transfer

Capacity Estimation

```
User Base:
├── Total Users: 100M
├── Daily Active Users (DAU): 10M
├── Concurrent Users (Peak): 1M
└── Games per User per Day: 5

Score Updates:
├── Daily Updates: 10M × 5 = 50M
├── Average Updates/Sec: 50M / 86400 ≈ 580
├── Peak Updates/Sec: 580 × 10 = 5,800
└── Safety Margin (10x): 58,000 updates/sec capacity

Read Operations:
├── Rank Checks per Game: 2
├── Top 100 Views: 1 per game
├── Daily Reads: 50M × 3 = 150M
├── Average Reads/Sec: 150M / 86400 ≈ 1,736
├── Peak Reads/Sec: 17,360
└── Safety Margin: 173,600 reads/sec capacity

Storage Requirements:
├── Per User Record: 50 bytes (user_id: 16, score: 8, timestamp: 8,
metadata: 18)
├── Global Leaderboard: 100M × 50 = 5GB
├── Daily/Weekly/Monthly: 3 × 5GB = 15GB
├── Regional (10 regions): 10 × 5GB = 50GB
├── Friends (avg 100 friends): Complex, stored in social graph
├── Historical Snapshots: 365 × 5GB = 1.825TB/year
└── Total with replicas (3x): ~6TB active + ~5.5TB archive

Server Requirements:
├── Redis Nodes (score storage): 20 nodes × 64GB = 1.28TB memory
├── Application Servers: 50 servers for processing
├── PostgreSQL (metadata): 3 servers with replication
└── Message Queue (Kafka): 5 brokers

Monthly Cost (AWS):
├── Redis (ElastiCache): ~$10,000
├── EC2 Application Servers: ~$5,000
├── RDS PostgreSQL: ~$2,000
├── Kafka (MSK): ~$1,500
├── S3 Storage (archives): ~$500
├── Data Transfer: ~$1,000
└── Total: ~$20,000/month
```

High-Level Architecture

System Overview

```
┌──────────────────────────────────────────────────────────────────────┐
│                              CLIENTS                                  │
│  Mobile Apps | Web Browsers | Game Clients | Third-party Apps        │
└──────────────────────────────┬───────────────────────────────────────┘
                               │
                               ▼
┌──────────────────────────────────────────────────────────────────────┐
│                         CDN / Edge Layer                              │
│  - Static Assets (images, CSS, JS)                                    │
│  - Cached Top 100 Leaderboard Pages                                  │
│  - Regional Edge Locations                                            │
└──────────────────────────────┬───────────────────────────────────────┘
                               │
                               ▼
┌──────────────────────────────────────────────────────────────────────┐
│                      Load Balancer (Global)                           │
│  - Geographic routing
│
│  - SSL termination
│
│  - Health checks
│
│  - Rate limiting
│
└──────────────────────────────┬───────────────────────────────────────┘
                               │
                ┌──────────────┼──────────────┐
                │              │              │
                ▼              ▼              ▼
┌──────────────────────────────────────────────────────────────────────┐
│                       API Gateway Cluster                             │
│  ┌────────────────────────────────────────────────────────────────┐  │
│  │  Leaderboard Service (Stateless)                                 │  │
│  │  - REST API endpoints                                            │  │
│  │  - GraphQL API (advanced queries)                              │  │
│  │  - WebSocket connections (real-time updates)                   │  │
│  │  - Authentication & authorization                                │  │
│  │  - Request validation                                            │  │
│  └────────────────────────────────────────────────────────────────┘  │
└──────────────────────────────┬───────────────────────────────────────┘
                               │
           ┌───────────────────┼───────────────────┐
           │                   │                   │
           ▼                   ▼                   ▼
┌─────────────────┐  ┌─────────────────┐  ┌─────────────────┐
│  Score Update   │  │   Rank Query    │  │  Top N Query    │
```

```
│     Service     │  │     Service     │  │     Service     │
└────────┬────────┘  └────────┬────────┘  └────────┬────────┘
         │                    │                    │
         │                    │                    │
         ▼                    ▼                    ▼
┌─────────────────────────────────────────────────────────────┐
│                    Message Queue (Kafka)                      │
│  Topics:                                                      │
│  ├── score-updates (partitioned by user_id)                  │
│  ├── rank-queries (for analytics)                            │
│  ├── leaderboard-snapshots (for historical data)             │
│  └── user-events (achievements, milestones)                  │
└──────────────────────────┬──────────────────────────────────┘
                           │
             ┌─────────────┼─────────────┐
             │             │             │
             ▼             ▼             ▼
┌─────────────────────────────────────────────────────────────┐
│                 Primary Data Store (Redis Cluster)            │
│                                                               │
│                                                               │
│  ┌─────────────────────────────────────────────────────────┐ │
│  │ Sorted Sets (ZSETs) - Core Leaderboard Storage          │ │
│  │                                                         │ │
│  │                                                         │ │
│  │ Global Leaderboard:                                       │ │
│  │   Key: "leaderboard:global:alltime"                     │ │
│  │   Members: {user_id: score}                             │ │
│  │   Sorted by score descending                            │ │
│  │                                                         │ │
│  │                                                         │ │
│  │ Daily Leaderboard:                                        │ │
│  │   Key: "leaderboard:daily:2024-01-15"                   │ │
│  │   TTL: 7 days (for historical reference)                │ │
│  │                                                         │ │
│  │                                                         │ │
│  │ Regional Leaderboard:                                     │ │
│  │   Key: "leaderboard:region:us:alltime"                  │ │
│  │                                                         │ │
│  │                                                         │ │
│  │ Friends Leaderboard:                                      │ │
│  │   Key: "leaderboard:friends:user_123:weekly"            │ │
│  └─────────────────────────────────────────────────────────┘ │
│                                                               │
│                                                               │
│  Sharding Strategy: Consistent hashing by leaderboard_type   │
│  Replication: Master-Slave (async replication)               │
│  Persistence: RDB snapshots + AOF logs                       │
└──────────────────────────┬──────────────────────────────────┘
                           │
                           ▼
┌─────────────────────────────────────────────────────────────┐
│               Secondary Data Store (PostgreSQL)               │
```

```
│  ┌─────────────────────────────────────────────────────────────┐  │
│  │  User Metadata:                                              │ │
│  │    user_id, username, email, region, created_at            │  │
│  │                                                              │
│
│  │  Historical Rankings:                                        │ │
│  │    snapshot_id, user_id, rank, score, timestamp            │  │
│  │    (Partitioned by timestamp)                                │ │
│  │                                                              │
│
│  │  Leaderboard Configuration:                                  │ │
│  │    leaderboard_id, type, reset_schedule, rules             │  │
│  │                                                              │
│
│  │  Achievements & Milestones:                                  │ │
│  │    achievement_id, user_id, leaderboard_id, earned_at      │  │
│  └─────────────────────────────────────────────────────────────┘  │
│
│
│  Primary-Replica Setup (3 nodes)                                   │
│  Read Replicas for Analytics                                       │
└────────────────────────────────────────────────────────────────────┘

┌────────────────────────────────────────────────────────────────────┐
│                     Background Jobs (Workers)                       │
│  ┌─────────────────────────────────────────────────────────────┐  │
│  │  Scheduled Tasks:                                            │ │
│  │  ├── Daily Leaderboard Reset (00:00 UTC)                    │  │
│  │  ├── Weekly Leaderboard Reset (Monday 00:00)                │  │
│  │  ├── Monthly Snapshot Generation                            │  │
│  │  ├── Historical Data Archival (move to S3)                  │  │
│  │  ├── Achievement Detection & Notification                   │  │
│  │  ├── Inactive User Cleanup                                  │  │
│  │  └── Leaderboard Analytics Computation                      │  │
│  └─────────────────────────────────────────────────────────────┘  │
└────────────────────────────────────────────────────────────────────┘

┌────────────────────────────────────────────────────────────────────┐
│                       Cache Layer (Redis)                           │
│  ┌─────────────────────────────────────────────────────────────┐  │
│  │  L1 Cache (Application Memory):                              │ │
│  │    - Top 100 cached for 10 seconds                          │  │
│  │    - User's rank cached for 30 seconds                      │  │
│  │    - Leaderboard metadata cached for 5 minutes              │  │
│  │                                                              │
│
│  │  L2 Cache (Redis):                                           │ │
│  │    - Top 1000 cached for 1 minute                           │  │
│  │    - Regional top 100 cached for 1 minute                   │  │
│  │    - Friends leaderboard cached for 1 minute                │  │
│  │    - User profile data cached for 10 minutes                │  │
│  └─────────────────────────────────────────────────────────────┘  │
└────────────────────────────────────────────────────────────────────┘
```

```
┌─────────────────────────────────────────────────────────────────────┐
│                Archive Storage (Amazon S3 / GCS)                    │
│  - Historical leaderboard snapshots (daily exports)                 │
│  - Analytics data (user behavior, trends)                           │
│  - Audit logs (compliance)                                          │
│  - Backup data (disaster recovery)                                  │
└─────────────────────────────────────────────────────────────────────┘

┌─────────────────────────────────────────────────────────────────────┐
│              Monitoring & Observability Stack                       │
│  ┌───────────────────────────────────────────────────────────────┐  │
│  │ Metrics (Prometheus + Grafana):                               │  │
│  │   - Score update throughput                                   │  │
│  │   - Query latency (P50, P95, P99)                             │  │
│  │   - Cache hit rates                                           │  │
│  │   - Redis memory usage                                        │  │
│  │   - Error rates by endpoint                                   │  │
│  │                                                               │  │
│  │ Logs (ELK Stack):                                             │  │
│  │   - Application logs                                          │  │
│  │   - Access logs                                               │  │
│  │   - Error logs with stack traces                              │  │
│  │                                                               │  │
│  │ Tracing (Jaeger / Zipkin):                                    │  │
│  │   - Request flow across services                              │  │
│  │   - Performance bottleneck identification                     │  │
│  │                                                               │  │
│  │ Alerting (PagerDuty / Opsgenie):                              │  │
│  │   - High error rates                                          │  │
│  │   - Latency spikes                                            │  │
│  │   - Service unavailability                                    │  │
│  │   - Redis memory threshold breaches                           │  │
│  └───────────────────────────────────────────────────────────────┘  │
└─────────────────────────────────────────────────────────────────────┘
```

Key Architectural Decisions

1. Redis Sorted Sets as Primary Storage

- **Why**: Native support for ranked data structures (ZSET)
- **Capabilities**:
  - O(log N) for score updates
  - O(log N) for rank lookups
  - O(log N + M) for range queries
  - Atomic operations (no race conditions)
- **Trade-off**: In-memory only → requires careful capacity planning

2. Separate Read and Write Paths

- **Write Path**: Score updates → Kafka → Redis
  - Asynchronous processing
  - Buffering during spikes
  - Retry mechanisms
- **Read Path**: Direct Redis queries
  - Fast, synchronous responses
  - Heavy caching
- **Trade-off**: Slight write delay (< 1s) for better read performance

3. Multi-Tier Caching

```
Request → L1 (In-Memory) → L2 (Redis Cache) → Redis ZSET → PostgreSQL
           ↓ Hit (90%)      ↓ Hit (9%)         ↓ Hit (1%)     ↓ Rare
         0.1ms latency     1ms latency       10ms latency   50ms latency
```

4. Time-Based Leaderboard Management

```
Strategy: Separate Redis keys with TTL

Daily:   "lb:daily:2024-01-15"   (TTL: 7 days)
Weekly:  "lb:weekly:2024-W03"    (TTL: 4 weeks)
Monthly: "lb:monthly:2024-01"    (TTL: 12 months)
All-time: "lb:global:alltime"    (No TTL)

Reset Mechanism:
- New key created at reset time
- Old key kept for historical reference
- Archived to S3 after TTL expiry
```

5. Horizontal Partitioning Strategy

```
Partition by Leaderboard Type:
├── Shard 1: Global leaderboards
├── Shard 2: Regional leaderboards (Americas)
├── Shard 3: Regional leaderboards (EMEA)
├── Shard 4: Regional leaderboards (APAC)
└── Shard 5-20: Friends & custom leaderboards

Benefits:
✓ Isolate load from different leaderboard types
✓ Scale each type independently
✓ Better fault isolation
```

Core Components

1. Score Update Service

Purpose: Process score updates with validation and conflict resolution

```
Component Architecture:

┌─────────────────────────────────────────────────────────┐
│                    Score Update API                      │
│  POST /leaderboard/update                                │
│  {                                                       │
│    "user_id": "user_123",                                │
│    "leaderboard_id": "global_alltime",                   │
│    "score": 1500,                                        │
│    "metadata": {                                         │
│      "game_id": "g456",                                  │
│      "timestamp": 1640995200,                            │
│      "session_id": "s789"                                │
│    }                                                     │
│  }                                                       │
└─────────────────────────────────────────────────────────┘
         ↓
┌─────────────────────────────────────────────────────────┐
│                 Validation Layer                         │
│  ├── Authentication: Valid user token                    │
│  ├── Authorization: User owns this score                 │
│  ├── Score Range: Within valid bounds                    │
│  ├── Rate Limiting: Max updates per user                 │
│  ├── Duplicate Detection: Prevent replays                │
│  └── Game Rule Validation: Score achievable              │
└─────────────────────────────────────────────────────────┘
         ↓
┌─────────────────────────────────────────────────────────┐
│              Conflict Resolution Logic                   │
│                                                          │
│  Strategy Options:                                       │
│  ├── HIGHEST: Keep maximum score                         │
│  ├── LATEST: Keep most recent score                      │
│  ├── SUM: Add scores together                            │
│  ├── INCREMENT: Add delta to current                     │
│  └── CUSTOM: Game-specific logic                         │
│                                                          │
│  Example (HIGHEST):                                      │
│    Current Score: 1200                                   │
│    New Score: 1500                                       │
│    Result: 1500 (new score higher, update)               │
└─────────────────────────────────────────────────────────┘
         ↓
```

```
┌──────────────────────────────────────────────────────────────┐
│              Publish to Kafka                                  │
│  Topic: "score-updates"                                        │
│  Partition Key: user_id (ensures ordering)                   │
│  Message:                                                      │
│  {                                                             │
│    "user_id": "user_123",                                    │
│    "leaderboard_id": "global_alltime",                       │
│    "score": 1500,                                            │
│    "previous_score": 1200,                                   │
│    "timestamp": 1640995200,                                  │
│    "operation": "UPDATE"                                     │
│  }                                                             │
└──────────────────────────────────────────────────────────────┘
         ↓
┌──────────────────────────────────────────────────────────────┐
│           Kafka Consumer (Worker Nodes)                        │
│  ├── Consume in batches (100 messages)                       │
│  ├── Group by leaderboard_id                                 │
│  ├── Execute Redis pipeline                                  │
│  └── Commit offset after success                             │
└──────────────────────────────────────────────────────────────┘
         ↓
┌──────────────────────────────────────────────────────────────┐
│            Update Redis Sorted Set                             │
│                                                                │
│  Redis Command:                                                │
│    ZADD leaderboard:global:alltime 1500 user_123            │
│                                                                │
│  Complexity: O(log N) where N = total users                  │
│  Performance: ~0.1ms for 100M users                          │
│                                                                │
│  Atomic Operations (Lua Script):                             │
│    local old_score = redis.call('ZSCORE', key, member)      │
│    if old_score == nil or new_score > old_score then        │
│      redis.call('ZADD', key, new_score, member)             │
│      return {old_score, new_score}                           │
│    end                                                         │
│    return {old_score, old_score}                             │
└──────────────────────────────────────────────────────────────┘
         ↓
┌──────────────────────────────────────────────────────────────┐
│         Invalidate Related Caches                              │
│  ├── User's rank cache                                       │
│  ├── Top 100 if user was in top                             │
│  ├── User's neighborhood view                                │
│  └── Friends leaderboard containing user                     │
└──────────────────────────────────────────────────────────────┘
         ↓
┌──────────────────────────────────────────────────────────────┐
│       Trigger Side Effects (Async)                             │
│  ├── Achievement Detection: Did user hit milestone?          │
│  ├── Notification: Send rank change alerts                   │
```

```
│  ├── Analytics: Log score change event                        │
│  └── Audit: Record for compliance                             │
└───────────────────────────────────────────────────────────────┘
```

Error Handling:

```
Retry Strategy:
├── Kafka Consumer Retries: 3 attempts with exponential backoff
├── Redis Connection Failures: Circuit breaker pattern
├── Validation Failures: Return 400 Bad Request immediately
└── Idempotency: Use unique transaction IDs to prevent duplicates
```

2. Rank Query Service

Purpose: Fetch user's current rank efficiently

```
API Endpoint:
GET /leaderboard/rank?user_id=user_123&leaderboard_id=global_alltime

Response:
{
  "user_id": "user_123",
  "rank": 1547,
  "score": 1500,
  "total_users": 10000000,
  "percentile": 99.98,
  "rank_change": +23 (compared to yesterday)
}
```

Implementation Flow:

```
┌───────────────────────────────────────────────────────────────┐
│ Step 1: Check L1 Cache (In-Memory)                            │
│   Key: "rank:user_123:global_alltime"                         │
│   TTL: 30 seconds                                              │
│   Hit Rate: ~70%                                               │
│   Latency: 0.1ms                                               │
└───────────────────────────────────────────────────────────────┘
         ↓ (cache miss)
┌───────────────────────────────────────────────────────────────┐
│ Step 2: Query Redis Sorted Set                                │
│   Command: ZREVRANK leaderboard:global:alltime user_123       │
│   Returns: 1546 (0-indexed, so rank is 1547)                 │
│   Complexity: O(log N)                                         │
│   Latency: ~1ms for 100M users                                │
└───────────────────────────────────────────────────────────────┘
```

```
        ↓

┌──────────────────────────────────────────────────────────┐
│ Step 3: Get Score                                        │
│   Command: ZSCORE leaderboard:global:alltime user_123    │
│   Returns: 1500                                          │
│   Complexity: O(1)                                       │
│   Latency: ~0.5ms                                        │
└──────────────────────────────────────────────────────────┘

        ↓

┌──────────────────────────────────────────────────────────┐
│ Step 4: Calculate Percentile                             │
│   Command: ZCARD leaderboard:global:alltime              │
│   Returns: 10000000 (total users)                        │
│   Percentile: (1 - 1547/10000000) × 100 = 99.98%         │
└──────────────────────────────────────────────────────────┘

        ↓

┌──────────────────────────────────────────────────────────┐
│ Step 5: Get Rank Change (Historical)                     │
│   Query: PostgreSQL historical_rankings table            │
│   SELECT rank FROM historical_rankings                   │
│   WHERE user_id = 'user_123'                             │
│   AND leaderboard_id = 'global_alltime'                  │
│   AND date = CURRENT_DATE - 1                            │
│   Result: Previous rank = 1570                           │
│   Change: 1547 - 1570 = -23 (improved by 23 positions)   │
└──────────────────────────────────────────────────────────┘

        ↓

┌──────────────────────────────────────────────────────────┐
│ Step 6: Cache Result                                     │
│   Store in L1 cache with 30-second TTL                   │
│   Return to client                                       │
└──────────────────────────────────────────────────────────┘
```

Optimization: Batch Rank Queries:

```
For multiple users (e.g., friends list):

Naive Approach:
  For each friend:
    ZREVRANK leaderboard friend_id
  Total: N × O(log M) = O(N log M)

Optimized Approach:
  Pipeline all ZREVRANK commands
  Execute in single Redis round trip
  Total: O(N log M) but single network call
  Speedup: 10-100x depending on network latency
```

3. Top N Query Service

Purpose: Retrieve top N players efficiently

```
API Endpoint:
GET /leaderboard/top?leaderboard_id=global_alltime&limit=100&offset=0

Response:
{
  "leaderboard_id": "global_alltime",
  "updated_at": 1640995200,
  "total_users": 10000000,
  "entries": [
    {
      "rank": 1,
      "user_id": "user_999",
      "username": "ProGamer",
      "score": 9999,
      "avatar_url": "...",
      "country": "US"
    },
    ...
  ]
}
```

Implementation Flow:

```
┌──────────────────────────────────────────────────────────────┐
│ Step 1: Check CDN Cache                                      │
│   URL: /leaderboard/top?lb=global&limit=100                  │
│   TTL: 10 seconds                                            │
│   Hit Rate: ~95% (very cacheable)                            │
│   Latency: <10ms (edge location)                             │
└──────────────────────────────────────────────────────────────┘
        ↓ (cache miss)
┌──────────────────────────────────────────────────────────────┐
│ Step 2: Query Redis (Top N)                                  │
│   Command:                                                   │
│     ZREVRANGE leaderboard:global:alltime 0 99 WITHSCORES     │
│                                                              │
│   Returns:                                                   │
│     [user_999, 9999, user_888, 8888, ..., user_100, 1000]    │
│                                                              │
│   Complexity: O(log N + M) where M = 100                     │
│   Latency: ~1-2ms                                            │
└──────────────────────────────────────────────────────────────┘
        ↓
┌──────────────────────────────────────────────────────────────┐
│ Step 3: Enrich with User Data (Batch)                        │
│   Fetch user profiles for all 100 users in parallel:         │
│   - Username, avatar, country from PostgreSQL                │
```

```
│   - Use connection pooling                              │
│   - Cache results for 5 minutes                         │
│   Latency: ~20ms (parallel queries)                     │
└─────────────────────────────────────────────────────────┘
        ↓
┌─────────────────────────────────────────────────────────┐
│ Step 4: Format Response & Cache                         │
│   Combine Redis scores + PostgreSQL user data          │
│   Cache in CDN for 10 seconds                           │
│   Return to client                                       │
│   Total latency: ~30-50ms                                │
└─────────────────────────────────────────────────────────┘
```

Pagination Strategy:

```
For large result sets (e.g., top 1000):

Cursor-based:
  GET /leaderboard/top?cursor=user_500&limit=100
  - More consistent with changing data
  - Better for infinite scroll
  - Use ZRANK to find position

Offset-based:
  GET /leaderboard/top?offset=500&limit=100
  - Simpler to implement
  - Better for page numbers
  - May skip entries if data changes
```

4. User Neighborhood Service

Purpose: Show users around a specific user's rank

```
API Endpoint:
GET /leaderboard/neighbors?user_id=user_123&range=10

Response:
{
  "user_id": "user_123",
  "rank": 1547,
  "neighbors": {
    "above": [
      {"rank": 1537, "user_id": "user_abc", "score": 1510},
      ...
      {"rank": 1546, "user_id": "user_xyz", "score": 1501}
    ],
    "current": {
      "rank": 1547,
```

```
      "user_id": "user_123",
      "score": 1500
    },
    "below": [
      {"rank": 1548, "user_id": "user_def", "score": 1499},
      ...
      {"rank": 1557, "user_id": "user_ghi", "score": 1490}
    ]
  }
}
```

Implementation:

```
Redis Command:
  # Get user's rank first
  ZREVRANK leaderboard:global:alltime user_123  → 1546

  # Get users around this rank
  ZREVRANGE leaderboard:global:alltime 1536 1556 WITHSCORES

  Returns 21 users (10 above + user + 10 below)
  Complexity: O(log N + M) where M = 21
```

Data Model & Storage Design

Redis Data Structures

1. Sorted Set (ZSET) - Core Leaderboard

```
Key: "leaderboard:{type}:{scope}:{period}"
Member: user_id
Score: user's score (numeric)

Examples:
┌───────────────────────────────────────────────────────────┐
│ Key: "leaderboard:global:alltime"                         │
│                                                            │
│ user_999 → 9999                                           │
│ user_888 → 8888                                           │
│ user_777 → 7777                                           │
│ ...                                                        │
│ user_123 → 1500                                           │
│ ...                                                        │
│ user_001 → 100                                            │
│                                                            │
│ Total: 100M members, ~5GB memory                          │
└───────────────────────────────────────────────────────────┘
```

```
Operations:
├── ZADD: O(log N) - Add/update score
├── ZREVRANK: O(log N) - Get rank
├── ZREVRANGE: O(log N + M) - Get top M
├── ZSCORE: O(1) - Get score
├── ZCARD: O(1) - Total members
└── ZINCRBY: O(log N) - Increment score
```

2. String - User Rank Cache

```
Key: "rank:{user_id}:{leaderboard_id}"
Value: JSON {rank: 1547, score: 1500, updated_at: 1640995200}
TTL: 30 seconds

Purpose: Cache frequently queried ranks
Memory: ~100 bytes per cached entry
Cache size: 10M active users × 100B = 1GB
```

3. List - Recent Score Updates

```
Key: "recent_updates:{leaderboard_id}"
Value: List of recent score changes
Structure: LPUSH new updates, LTRIM to keep last 1000

Purpose: Real-time activity feed, leaderboard changes
Memory: ~50KB per leaderboard
```

4. Set - Active Users

```
Key: "active_users:{leaderboard_id}:{date}"
Value: Set of user_ids who scored today
TTL: 48 hours

Purpose: Track daily/weekly active users
Memory: 10M users × 16 bytes = 160MB
```

PostgreSQL Schema

```sql
-- Users table (10M-100M rows)
CREATE TABLE users (
    user_id UUID PRIMARY KEY,
```

```sql
    username VARCHAR(50) UNIQUE NOT NULL,
    email VARCHAR(255) UNIQUE NOT NULL,
    country_code CHAR(2),
    region VARCHAR(50),
    avatar_url TEXT,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    last_active_at TIMESTAMP,
    INDEX idx_username (username),
    INDEX idx_country (country_code),
    INDEX idx_last_active (last_active_at)
);

-- Leaderboard configurations (hundreds of rows)
CREATE TABLE leaderboards (
    leaderboard_id VARCHAR(100) PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    description TEXT,
    type VARCHAR(50), -- 'global', 'regional', 'friends', 'custom'
    scope VARCHAR(50), -- 'alltime', 'daily', 'weekly', 'monthly'
    reset_schedule VARCHAR(50), -- cron expression
    score_update_strategy VARCHAR(20), -- 'highest', 'latest', 'sum'
    tie_breaker VARCHAR(20) DEFAULT 'timestamp', -- 'timestamp',
'user_id'
    is_active BOOLEAN DEFAULT true,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Historical rankings (billions of rows, partitioned)
CREATE TABLE historical_rankings (
    id BIGSERIAL,
    snapshot_id VARCHAR(50),
    leaderboard_id VARCHAR(100),
    user_id UUID,
    rank INTEGER,
    score BIGINT,
    snapshot_date DATE,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    PRIMARY KEY (id, snapshot_date),
    INDEX idx_user_leaderboard_date (user_id, leaderboard_id,
snapshot_date),
    INDEX idx_leaderboard_date (leaderboard_id, snapshot_date)
) PARTITION BY RANGE (snapshot_date);

-- Create partitions for each month
CREATE TABLE historical_rankings_2024_01 PARTITION OF
historical_rankings
FOR VALUES FROM ('2024-01-01') TO ('2024-02-01');

-- Achievements (millions of rows)
CREATE TABLE achievements (
    achievement_id VARCHAR(100) PRIMARY KEY,
    name VARCHAR(255),
    description TEXT,
```

```sql
    threshold_type VARCHAR(50), -- 'rank', 'score', 'streak'
    threshold_value INTEGER,
    icon_url TEXT,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE user_achievements (
    id BIGSERIAL PRIMARY KEY,
    user_id UUID NOT NULL,
    achievement_id VARCHAR(100) NOT NULL,
    leaderboard_id VARCHAR(100),
    earned_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    metadata JSONB,
    INDEX idx_user_achievements (user_id),
    INDEX idx_achievement (achievement_id),
    UNIQUE (user_id, achievement_id, leaderboard_id)
);

-- Score audit log (for compliance, billions of rows)
CREATE TABLE score_audit_log (
    id BIGSERIAL PRIMARY KEY,
    user_id UUID NOT NULL,
    leaderboard_id VARCHAR(100),
    old_score BIGINT,
    new_score BIGINT,
    operation VARCHAR(20), -- 'UPDATE', 'DELETE', 'RESET'
    ip_address INET,
    user_agent TEXT,
    session_id VARCHAR(100),
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
) PARTITION BY RANGE (created_at);
```

Data Retention Policy

```
Retention Strategy:

Hot Data (Redis):
├── Current leaderboards: Keep in memory
├── Daily leaderboards: TTL 7 days
├── Weekly leaderboards: TTL 4 weeks
├── Monthly leaderboards: TTL 12 months
└── Cache entries: TTL 30 seconds - 5 minutes

Warm Data (PostgreSQL):
├── Historical rankings: Keep 2 years
├── Partition by month
├── Archive older data to S3
└── Use read replicas for analytics

Cold Data (S3):
```

```
├── Archived historical snapshots: Keep 5 years
├── Compressed (gzip)
├── Parquet format for analytics
└── Glacier for long-term storage (7+ years)

Lifecycle:
Redis (live) → PostgreSQL (2 years) → S3 (5 years) → Glacier (permanent)
```

Real-time Updates & Data Flow

Write Path (Score Update)

```
┌─────────────────────────────────────────────────────────────┐
│ PHASE 1: Client Submission                                   │
└─────────────────────────────────────────────────────────────┘
        Client submits score
        POST /leaderboard/update
        Latency: 0ms (start)
        ↓
┌─────────────────────────────────────────────────────────────┐
│ PHASE 2: API Gateway Processing                             │
│  - Authentication (JWT validation): 5ms                      │
│  - Request validation: 1ms                                   │
│  - Rate limiting check: 1ms                                  │
│  Total: 7ms                                                  │
└─────────────────────────────────────────────────────────────┘

        ↓

┌─────────────────────────────────────────────────────────────┐
│ PHASE 3: Score Update Service                               │
│  - Game rules validation: 3ms                                │
│  - Conflict resolution logic: 2ms                            │
│  - Publish to Kafka: 5ms                                     │
│  - Return 202 Accepted to client: 10ms                      │
│  Total: 20ms (client sees response here)                    │
└─────────────────────────────────────────────────────────────┘

        ↓

┌─────────────────────────────────────────────────────────────┐
│ PHASE 4: Asynchronous Processing (Kafka)                    │
│  - Message queued in Kafka: +1ms                            │
│  - Consumer picks up message: +100-500ms (batching)         │
│  - Process batch (100 messages): 50ms                       │
└─────────────────────────────────────────────────────────────┘

        ↓

┌─────────────────────────────────────────────────────────────┐
│ PHASE 5: Redis Update                                        │
│  - Execute ZADD command: 0.5ms                              │
│  - Replicate to slaves: 1-5ms (async)                      │
│  - Update multiple leaderboards: 2ms                         │
│  Total: 3-8ms                                                │
```

```
└──────────────────────────────────────────────────────────┘
         ↓
┌──────────────────────────────────────────────────────────┐
│ PHASE 6: Side Effects (Parallel)                          │
│  - Cache invalidation: 1ms                                 │
│  - Achievement detection: 10ms                             │
│  - Notification service: 20ms                              │
│  - Analytics logging: 5ms                                  │
│  (All happen in background, don't block)                  │
└──────────────────────────────────────────────────────────┘

Total Perceived Latency (client): ~20ms
Total Actual Processing Time: ~600ms (async)
```

Read Path (Rank Query)

```
┌──────────────────────────────────────────────────────────┐
│ PHASE 1: Request Arrives                                   │
│  GET /leaderboard/rank?user_id=user_123                   │
│  Latency: 0ms (start)                                      │
└──────────────────────────────────────────────────────────┘
         ↓
┌──────────────────────────────────────────────────────────┐
│ PHASE 2: L1 Cache Check (In-Memory)                       │
│  - Lookup in local cache: 0.1ms                           │
│  - Hit rate: 70%                                           │
│  - If HIT: Return immediately                              │
│  Total: 0.1ms (if hit)                                     │
└──────────────────────────────────────────────────────────┘
         ↓ (if miss)
┌──────────────────────────────────────────────────────────┐
│ PHASE 3: Redis Query                                       │
│  - Connection from pool: 0.1ms                             │
│  - ZREVRANK command: 1ms                                   │
│  - ZSCORE command: 0.5ms                                   │
│  - Network roundtrip: 0.5ms                                │
│  Total: 2ms                                                │
└──────────────────────────────────────────────────────────┘
         ↓
┌──────────────────────────────────────────────────────────┐
│ PHASE 4: Historical Data (Optional)                       │
│  - Query PostgreSQL for rank change: 10ms                 │
│  - Use read replica                                        │
│  - Cached query plan                                       │
│  Total: 10ms                                               │
└──────────────────────────────────────────────────────────┘
         ↓
┌──────────────────────────────────────────────────────────┐
│ PHASE 5: Response Assembly                                 │
│  - Format JSON: 0.5ms                                      │
```

```
│  - Cache result in L1: 0.1ms                            │
│  - Return to client: 1ms                                 │
│  Total: 1.5ms                                            │
└─────────────────────────────────────────────────────────┘

Total Latency:
├── Cache hit: 0.1ms
├── Redis only: 2-3ms
└── With historical: 12-15ms

Target: <100ms (P95)
Actual: <15ms (P99) ✓
```

Real-Time Update Notifications (WebSocket)

```
Architecture:

Client ←──WebSocket──→ API Gateway ←──Pub/Sub──→ Redis
                              ↓
                      Subscribed to:
                      - User's rank changes
                      - Friends' score updates
                      - Top 10 changes

Implementation:
┌─────────────────────────────────────────────────────────┐
│ When score updates happen:                               │
│  1. Redis ZADD updates sorted set                       │
│  2. Lua script publishes to Redis Pub/Sub:              │
│     PUBLISH "leaderboard:updates:global" "user_123:1500"│
│  3. API Gateway subscribed to channel                   │
│  4. Forwards to connected WebSocket clients             │
│  5. Client UI updates in real-time                      │
└─────────────────────────────────────────────────────────┘

Message Format:
{
  "type": "rank_change",
  "user_id": "user_123",
  "old_rank": 1570,
  "new_rank": 1547,
  "score": 1500,
  "timestamp": 1640995200
}

Scalability:
├── Use Redis Pub/Sub for broadcasting
├── Sticky sessions for WebSocket connections
├── Connection pooling per server
└── Limit subscriptions per user (max 10 leaderboards)
```

Ranking Algorithms

1. Standard Ranking (Descending by Score)

```
Algorithm: Rank by score, highest first

Example:
┌──────┬───────────┬────────┬──────────┐
│ Rank │ User ID   │ Score  │ Notes    │
├──────┼───────────┼────────┼──────────┤
│  1   │ user_999  │ 9999   │ Highest  │
│  2   │ user_888  │ 8888   │          │
│  3   │ user_777  │ 7777   │          │
│  4   │ user_666  │ 6666   │          │
│  5   │ user_555  │ 5555   │          │
└──────┴───────────┴────────┴──────────┘

Redis Implementation:
  ZREVRANGE leaderboard:global:alltime 0 -1 WITHSCORES
  (Sorted automatically by score descending)

Complexity: O(log N) for insert, O(log N) for rank lookup
```

2. Tie-Breaking Strategies

Problem: Multiple users with same score

```
Scenario:
┌──────┬───────────┬────────┬──────────────┐
│ Rank │ User ID   │ Score  │ Joined Time  │
├──────┼───────────┼────────┼──────────────┤
│  ?   │ user_aaa  │ 1000   │ 10:00:00     │
│  ?   │ user_bbb  │ 1000   │ 10:05:00     │
│  ?   │ user_ccc  │ 1000   │ 10:10:00     │
└──────┴───────────┴────────┴──────────────┘

How to rank them?
```

Strategy A: First Come, First Served

```
User who achieved score first gets higher rank

Implementation:
  Store composite score: score + (1 / timestamp)
```

```
  Example:
    user_aaa: 1000.0000001 (achieved at 10:00:00)
    user_bbb: 1000.0000000.8 (achieved at 10:05:00)
    user_ccc: 1000.0000000.6 (achieved at 10:10:00)

Result:
  Rank 1: user_aaa
  Rank 2: user_bbb
  Rank 3: user_ccc

Pros: Fair, rewards early achievement
Cons: Precision issues with large timestamps
```

Strategy B: Lexicographic (User ID)

```
Break ties by user ID (alphabetically)

Implementation:
  Use Redis ZSET with same score
  Redis maintains insertion order for same scores
  Then sort by user_id for display

Result:
  Rank 1: user_aaa (alphabetically first)
  Rank 2: user_bbb
  Rank 3: user_ccc

Pros: Simple, deterministic, no precision issues
Cons: Arbitrary (user ID doesn't mean much to users)
```

Strategy C: Multiple Criteria (Recommended)

```
Use secondary & tertiary criteria

Primary: Score (highest wins)
Secondary: Timestamp (earlier wins)
Tertiary: User ID (alphabetically)

Implementation:
  Store in Redis: score as primary
  Fetch tied users, sort by timestamp in application
  Final sort by user_id if still tied

Pros: Most fair and intuitive
Cons: Slightly more complex
```

3. Time-Decay Ranking

Purpose: Keep leaderboard fresh, reward recent activity

```
Algorithm: Scores decay over time

Formula:
  effective_score = base_score × decay_factor^days_since_activity

Example (decay_factor = 0.9, decay per day):
┌──────────────┬────────────┬──────┬──────────────────┬────────────┐
│ User         │ Base Score │ Days │ Decay Factor     │ Effective  │
├──────────────┼────────────┼──────┼──────────────────┼────────────┤
│ user_active  │ 1000       │ 0    │ 0.9^0 = 1.0      │ 1000       │
│ user_week    │ 1200       │ 7    │ 0.9^7 = 0.478    │ 574        │
│ user_month   │ 1500       │ 30   │ 0.9^30 = 0.042   │ 63         │
└──────────────┴────────────┴──────┴──────────────────┴────────────┘

Result:
  user_active (1000) ranks higher than user_month (63)
  Despite user_month having higher base score!

Implementation:
  Daily cron job:
    FOR each user IN leaderboard:
      days_inactive = TODAY - last_activity_date
      effective_score = base_score × (0.9 ^ days_inactive)
      UPDATE Redis ZSET with effective_score

Pros: Encourages ongoing engagement
Cons: Computational overhead, may frustrate casual players
```

4. Percentile Ranking

Purpose: Show relative performance ("You're in top 5%")

```
Calculation:
  percentile = (1 - rank / total_users) × 100

Example:
┌──────┬────────────┬───────┬────────────────┐
│ Rank │ Total Users│ %ile  │ Interpretation │
├──────┼────────────┼───────┼────────────────┤
│ 100  │ 10,000,000 │ 99.999│ Top 0.001%     │
│ 1000 │ 10,000,000 │ 99.99 │ Top 0.01%      │
│ 10K  │ 10,000,000 │ 99.9  │ Top 0.1%       │
│ 100K │ 10,000,000 │ 99    │ Top 1%         │
│ 1M   │ 10,000,000 │ 90    │ Top 10%        │
│ 5M   │ 10,000,000 │ 50    │ Median         │
└──────┴────────────┴───────┴────────────────┘
```

```
Display to User:
  "You're ranked 10,543 out of 10M players (top 0.1%)"
  "You're better than 99.9% of players!"

Redis Commands:
  ZREVRANK leaderboard user_id  → rank
  ZCARD leaderboard              → total_users
  Calculate: (1 - rank/total) × 100
```

Distributed System Design

Challenge 1: Handling Concurrent Updates

Problem: Multiple score updates for same user simultaneously

```
Scenario:
  Time T0:
    Game Server 1: user_123 scores 1000
    Game Server 2: user_123 scores 1200
    Both send updates simultaneously

  Without synchronization:
    Server 1 writes 1000 to Redis
    Server 2 writes 1200 to Redis
    OR vice versa → race condition

  Desired behavior:
    Keep highest score (1200)
```

Solution 1: Atomic Lua Scripts (Recommended)

```lua
-- Store in Redis as a Lua script
-- Ensures atomic compare-and-set operation

local key = KEYS[1]
local member = ARGV[1]
local new_score = tonumber(ARGV[2])
local strategy = ARGV[3] -- 'highest', 'latest', 'sum'

local current_score = redis.call('ZSCORE', key, member)

if current_score == false then
  -- User not in leaderboard, add them
  redis.call('ZADD', key, new_score, member)
  return {nil, new_score, 'added'}
end
```

```lua
current_score = tonumber(current_score)

if strategy == 'highest' then
  if new_score > current_score then
    redis.call('ZADD', key, new_score, member)
    return {current_score, new_score, 'updated'}
  else
    return {current_score, current_score, 'unchanged'}
  end
elseif strategy == 'latest' then
  redis.call('ZADD', key, new_score, member)
  return {current_score, new_score, 'updated'}
elseif strategy == 'sum' then
  local sum = current_score + new_score
  redis.call('ZADD', key, sum, member)
  return {current_score, sum, 'updated'}
end
```

Usage:

```
EVALSHA <script_sha> 1 leaderboard:global:alltime user_123 1200 highest

Returns: [1000, 1200, "updated"]
  → Previous score was 1000
  → New score is 1200
  → Status: updated

Atomic guarantee: No race conditions possible
```

Solution 2: Optimistic Locking

```
Alternative for complex logic:

1. Read current score + version number
   GET leaderboard:user_123:version  → v5
   ZSCORE leaderboard user_123       → 1000

2. Perform logic in application
   new_score = max(1000, 1200) = 1200

3. Conditional write with version check
   WATCH leaderboard:user_123:version
   IF version == v5:
     ZADD leaderboard 1200 user_123
     INCR leaderboard:user_123:version
   ELSE:
     RETRY
```

```
  Pros: Supports complex logic
  Cons: Retries needed, higher latency
```

Challenge 2: Redis Cluster Partitioning

Problem: 100M users don't fit in single Redis instance

Solution: Consistent Hashing with Redis Cluster

```
Redis Cluster Setup:
┌──────────────────────────────────────────────────────────┐
│ 16,384 hash slots distributed across nodes               │
│                                                           │
│ Node 1 (Master): Slots 0-5460     ┌─→ Replica 1          │
│ Node 2 (Master): Slots 5461-10922 ┌─→ Replica 2          │
│ Node 3 (Master): Slots 10923-16383┌─→ Replica 3          │
└──────────────────────────────────────────────────────────┘

Hash Slot Calculation:
  slot = CRC16(key) % 16384

Example:
  Key: "leaderboard:global:alltime"
  Hash: CRC16("leaderboard:global:alltime") = 12345
  Slot: 12345 % 16384 = 12345
  Node: Node 3 (slots 10923-16383)

Routing:
  Client → Redis Cluster
  Cluster returns: MOVED 12345 node3:6379
  Client connects to node3 directly
```

Alternative: Application-Level Sharding

```
Strategy: Partition by leaderboard type

Shard Assignment:
┌──────────────────────────────────────────────────────────┐
│ Shard 1 (Redis Instance): Global leaderboards            │
│   - leaderboard:global:alltime                           │
│   - leaderboard:global:daily                             │
│   - leaderboard:global:weekly                            │
│                                                           │
│ Shard 2-5: Regional leaderboards                         │
│   - leaderboard:region:us:*                              │
│   - leaderboard:region:eu:*                              │
│   - leaderboard:region:asia:*                            │
│                                                           │
```

```
│ Shard 6-20: Friends leaderboards                              │
│   - Partition by user_id hash                                 │
│   - leaderboard:friends:user_*                                │
└───────────────────────────────────────────────────────────────┘

Routing Logic (Application):
  def get_shard(leaderboard_id):
    if leaderboard_id.startswith('global'):
      return shard_1
    elif leaderboard_id.startswith('region'):
      region = extract_region(leaderboard_id)
      return region_shard_map[region]
    else:  # friends
      user_id = extract_user_id(leaderboard_id)
      return hash(user_id) % num_friend_shards + 6

Pros:
  ✓ Isolate load by leaderboard type
  ✓ Scale each type independently
  ✓ Better debuggability

Cons:
  ✗ Application logic complexity
  ✗ Can't use Redis Cluster features
```

Challenge 3: Cross-Region Consistency

Problem: Users in different regions see different ranks

Solution: Multi-Region Architecture

```
Option 1: Regional Independence (Recommended)
┌───────────────────────────────────────────────────────────────┐
│ Each region has own Redis cluster                             │
│                                                                │
│ US-WEST:   leaderboard:global:us-west                         │
│ US-EAST:   leaderboard:global:us-east                         │
│ EU-WEST:   leaderboard:global:eu-west                         │
│                                                                │
│ Async Replication: Every 5 minutes                            │
│   1. Export top 10K from each region                          │
│   2. Merge in central aggregator                              │
│   3. Redistribute merged result                               │
│                                                                │
│ Trade-off:                                                     │
│   Slightly stale cross-region ranks (5 min lag)             │
│   Fast local queries (< 10ms)                                 │
└───────────────────────────────────────────────────────────────┘

Option 2: Global Redis (High Consistency)
```

```
┌──────────────────────────────────────────────────────────────┐
│ Single Redis cluster for global leaderboard                   │
│                                                                 │
│ Primary: US-EAST (central location)                           │
│ Replicas: US-
```