Gaming Leaderboard - Take home assignment - SDE 1

A **gaming leaderboard system** is a **ranking mechanism** used in multiplayer and competitive games to track and display **player performance** based on their scores, achievements, or other performance metrics. Leaderboards are a key feature in modern games, driving **player engagement, competition, and motivation** by allowing users to compare their performance against others.

In a leaderboard system, players **submit scores** after playing a game, and the system **dynamically updates rankings** based on predefined rules, such as **highest total score, recent performance, or win/loss ratios**.

0. Basic APIs setup

Your task is to implement APIs that allow players to **submit scores, retrieve the top-ranked players, and check their own ranking**.

Note: Any tech stack is fine

Database Structure

```
CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    username VARCHAR(255) UNIQUE NOT NULL,
    join_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
CREATE TABLE game_sessions (
    id SERIAL PRIMARY KEY,
    user_id INT REFERENCES users(id) ON DELETE CASCADE,
    score INT NOT NULL,
    game_mode VARCHAR(50) NOT NULL,
    timestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
CREATE TABLE leaderboard (
    id SERIAL PRIMARY KEY,
    user_id INT REFERENCES users(id) ON DELETE CASCADE,
    total_score INT NOT NULL,
    rank INT
);
```

APIs to Implement

1. **Submit Score (** `POST /api/leaderboard/submit` **)**
   - Accepts `user_id` and `score`.
   - Update score in `game_sessions` table.
2. **Get Leaderboard (** `GET /api/leaderboard/top` **)**
   - Retrieves the **top 10 players** sorted by `total_score`.
3. **Get Player Rank (** `GET /api/leaderboard/rank/{user_id}` **)**
   - Fetches the player's current rank.

Now, you'll take your leaderboard system to the next level by working with millions of game records and optimizing your APIs under real-world conditions. You'll integrate New Relic for monitoring, run a load simulation script to mimic real user behavior, and work towards reducing API latencies while maintaining atomicity under concurrent requests. Additionally, you'll build a simple frontend UI with live updates to showcase how well you can work across the stack. The goal is to test your problem-solving skills, ability to handle pressure, and commitment to delivering a high-performance system.

Don't just focus on solving the assignment—make sure you thoroughly understand the underlying concepts. You'll be asked detailed questions not only about your implementation but also about the various concepts and techniques used throughout the assignment.

1. Setup Database with Large Dataset

Execute the following **SQL queries** to populate the database:

(Reduce the table size if these queries are taking too long to finish, meanwhile plan for the rest of the assignment & get started)

```sql
-- Populate Users Table with 1 Million Records
INSERT INTO users (username)
SELECT 'user_' || generate_series(1, 1000000);
-- Populate Game Sessions with Random Scores
INSERT INTO game_sessions (user_id, score, game_mode, timestamp)
SELECT
    floor(random() * 1000000 + 1)::int,
    floor(random() * 10000 + 1)::int,
    CASE WHEN random() > 0.5 THEN 'solo' ELSE 'team' END,
    NOW() - INTERVAL '1 day' * floor(random() * 365)
FROM generate_series(1, 5000000);
-- Populate Leaderboard by Aggregating Scores
INSERT INTO leaderboard (user_id, total_score, rank)
SELECT user_id, AVG(score) as total_score, RANK() OVER (ORDER BY SUM(score) DESC)
FROM game_sessions
GROUP BY user_id;
```

2. Simulate Real User Usage

Run the following **Python script** to generate continuous leaderboard activity.

```python
import requests
import random
import time
API_BASE_URL = "http://localhost:8000/api/leaderboard"
# Simulate score submission
def submit_score(user_id):
    score = random.randint(100, 10000)
    requests.post(f"{API_BASE_URL}/submit", json={"user_id": user_id, "score": score})
# Fetch top players
def get_top_players():
    response = requests.get(f"{API_BASE_URL}/top")
    return response.json()
# Fetch user rank
def get_user_rank(user_id):
    response = requests.get(f"{API_BASE_URL}/rank/{user_id}")
    return response.json()
if __name__ == "__main__":
    while True:
        user_id = random.randint(1, 1000000)
        submit_score(user_id)
        print(get_top_players())
        print(get_user_rank(user_id))
        time.sleep(random.uniform(0.5, 2))  # Simulate real user interaction
```

3. New Relic for Monitoring

Integrate New Relic (100GB free for new accounts) to monitor API performance. They should:

- Track API latencies under real load.
- Identify bottlenecks and slow database queries.
- Set up alerts for slow response times.

4. Optimize API Latency

Refactor all 3 apis to Reduce latency. Few things to consider are:

- **Using Database Indexing**
- **Implementing Caching**
- **Optimizing Queries**
- **Handling Concurrency**
- **Ensuring data consistency**

5. Ensure Atomicity and Consistency

- Use **transactions** to handle concurrent writes without race conditions.
- Implement **cache invalidation** strategies to ensure up-to-date rankings.
- Guarantee that leaderboard rankings remain **consistent** under high traffic.

6. Build a Simple Frontend UI with Live Updates

Candidates must create a basic frontend interface to display:

- **Top 10 Leaderboard Rankings** (Live-updating)
- **User Rank Lookup**

Evaluation Criteria

- **Bug free working**
- **Code Quality & Efficiency**
- **PRs and change management**
- **Unit tests**
- **Performance Optimization**
- **Data Consistency**
- **Monitoring & Analysis**
- **Basic API Security**
- **Problem-Solving & Ownership**
- **Documentation (HLD/LLD)**
- **Demo**

Final Deliverables

- Backend code.
- Frontend code.
- Performance report with New Relic (dashboard or screenshots).
- Documentation.