실시간 게임 순위표

🔥 DAU 500만 명, MAU 2500만 명의 게임 순위표를 어떻게 서비스할까?

▼ 1️⃣ 설계 범위 확정

기능 요구사항

- 상위 10명의 랭킹 표시
- 특정 사용자의 순위 표시
- 특정 사용자의 +-4위 사용자 표시

비기능 요구사항

- 점수 업데이트는 실시간으로 랭킹에 반영
- 일반적인 확장성, 가용성, 안정성

규모 추정

- **초당 게임 플레이어 수** = 평균 50명
- **최대 부하** = 평균의 5배 = 250명
- **사용자 점수 획득 QPS** = 50 * 10게임 플레이 = 500
  - 최대 QPS = 500 * 5 = 2500
- **상위 10명 랭킹 조회 QPS** = 랭킹은 처음 접속 시에만 표시한다고 가정, 약 50

▼ 2️⃣ 개략적 설계

API 설계

1. 사용자 순위 갱신 (POST /v1/scores)

게임 서버에서만 호출할 수 있는 내부 API

user_id(사용자 아이디)와 points(획득한 점수) 요청

2. 랭킹 상위 10명 조회 (GET /v1/scores)

3. 특정 사용자 순위 조회 (GET /v1/scores/{:user_id}

user_id(사용자 아이디)로 요청하면 점수와 순위 반환

프로세스

1. 사용자가 게임에서 승리하면 `클라이언트`는 `게임 서비스`에 요청을 보낸다.

   🤔 **클라이언트가 직접 랭킹 서비스와 통신하지 않는 이유?**

   - 사용자가 점수를 마음대로 바꾸는 중간자 공격 방지
   - 게임 서버가 모든 게임 로직을 처리한다면 클라이언트 개입 없이 점수를 정할 수 있다.

2. `게임 서비스`는 승리의 유효성 확인 후 `랭킹 서비스`에 점수 갱신 요청을 보낸다.

   🤔 **게임 서비스 - 순위표 서비스 간 메시지 큐 필요성?**

   - 게임 점수가 여러 기능을 지원한다면 합리적인 선택일지도 ?!

3. `랭킹 서비스`는 `랭킹 저장소`에 기록된 해당 사용자의 점수를 갱신한다.
4. 해당 사용자의 `클라이언트`는 `랭킹 서비스`에 직접 요청하여 데이터를 가져온다.
   - 상위 10명 랭킹
   - 해당 사용자 순위

데이터 모델

RDB

leaderboard
user_id (varchar) score (int)

- **사용자가 승리한 경우** : score를 1씩 증가하여 갱신
- **특정 사용자 순위** : 점수 기준 정렬 후 순위 조회
- **상위 10명 랭킹** : 점수 기준 내림차순 정렬

단점

- 레코드가 수백만 개 수준으로 많아지면 성능이 매우 나쁨
- 실시간성 애플리케이션에 부적합
    - 색인 추가 시 규모 확장성이 좋지 않음
- 데이터의 지속적인 변경으로 캐시 도입 불가

Redis

`sorted set`으로 설계하면 이상적인 해결 가능 !

sorted set이란?

내부적으로 해시 테이블과 스킵 리스트 자료 구조를 사용하는 자료형, **점수의 내림차순으로 정렬**됨

사용자의 점수를 저장하기 위해 `해시 테이블`, 사용자 목록을 저장하기 위해 `스킵 리스트`를 사용

- **스킵 리스트** : 다단계 색인으로 빠른 검색 가능 (RDB보다 성능이 좋다 !)
    - 정렬된 단방향 연결 리스트를 중간 노드를 하나씩 건너뛰는 1차 색인으로 시작해서 n차 색인 노드를 하나씩 건너뛰는 n+1차 색인을 만들어 빠르게 검색할 수 있게 함

Redis 명령어 알아보기

- `ZADD` : 새로운 사용자를 sorted_set에 삽입, 기존 사용자는 점수 업데이트
- `ZINCRBY` : 사용자 점수를 지정된 값만큼 증가
- `ZRANGE` / `ZREVRANGE` : 점수에 따라 정렬된 특정 범위의 사용자 목록 조회
- `ZRANK` / `ZREVRANK` : 점수에 따라 정렬했을 때 특정 사용자 위치 조회

```
# 점수 증가
# ZINCRBY (키) (증가 값) (사용자 아이디)
ZINCRBY leaderboard_mar_2024 1 'cheese'

# 점수 내림차순 특정 범위 목록 조회
# ZREVRANGE (키) (시작 인덱스) (종료 인덱스) (옵션 WITHSCORES = 
ZREVRANGE leaderboard_mar_2024 0 9 WITHSCORES

# 점수 내림차순 특정 사용자 위치 조회
# ZREVRANK (키) (사용자 아이디)
ZREVRANK leaderboard_mar_2024 'cheese'
```

```
# (bonus!) 점수 내림차순 특정 사용자 기준으로 일정 범위 내 목록 조회
ZREVRANGE leaderboard_mar_2024 357 365
```

저장소 요구사항

- **최소 저장 데이터** : 사용자 아이디, 점수
- **최악 시나리오** : MAU 사용자 모두(2500만명)가 모두 랭킹에 있는 경우
  - ID(24byte), 점수(2byte)라고 가정 시 랭킹 항목 당 `26byte` 필요
  - 26 * 2500만 = `650MB` **저장공간 필요** → 단일 레디스 서버로 감당 가능
- **CPU, I/O 사용량** : 최대 QPS(2500) 기준 단일 레디스 서버로 감당 가능
- **데이터 영속성** : 디스크 저장 옵션 시 인스턴스 재시작이 오래 걸리므로 읽기 사본으로 구성
  - RDB로 점수 히스토리를 타임스탬프와 함께 저장하면 인프라 장애 발생 시 레디스 복구에 활용 가능
- **성능 최적화** : 상위 10명 사용자 정보 캐시

▼ 3️⃣ 상세 설계

클라우드 서비스

자체 서비스를 이용하는 경우

- 매월 `sorted set`을 생성하여 기간 별 랭킹(사용자, 점수) 저장
- 사용자 세부 정보(이름, 프로필 이미지 등)는 `MySQL`에 저장
- 상위 10명 세부 정보를 `프로필 캐시`에 저장

AWS를 이용하는 경우

`API 게이트웨이`로 HTTP 엔드포인트를 정의하고 `AWS 람다`에 연결

AWS 람다

- 필요할 때에만 실행됨, 트래픽에 따라 자동 규모 확장
- **점수 획득** : 레디스에 ZINCRBY 명령 호출 후 MySQL에 점수 히스토리 저장

- **순위 검색** : 레디스에 ZREVRANGE 명령 호출 후 MySQL에 사용자 세부 정보 조회

Redis 규모 확장

원래 규모의 100배인 5억 DAU 처리하기 → **샤딩**

고정 파티션

랭킹에 등장하는 점수 범위에 따라 파티션 나누기

- 랭킹 전반에 점수가 고르게 분포돼있거나 점수 범위를 조정해서 고른 분포를 만들어야 함
- 사용자 점수 갱신 시 사용자가 어느 샤드에 있는지 알아야 함
  - 1) MySQL 질의를 통해 현재 점수 계산하기
  - 2) 👍🏻 사용자 ID-점수 관계를 지정하는 2차 캐시를 통해 알아내기
- 사용자 점수 변동으로 샤드 이동 시 기존 샤드에서 제거한 후 새 샤드로 옮겨야 함
- 특정 샤드에 속한 모든 사용자 수 조회 : `info keyspace`

해시 파티션

여러 노드에 데이터를 자동 샤딩하는 레디스 클러스터 사용

16384개 해시 슬롯 중 하나에 각각의 키가 속하도록 함 `CRC16(key) % 16384`

- 모든 키를 재분배하지 않아도 클러스터에 쉽게 노드 추가/삭제 가능
- 상위 10명 랭킹 조회 시 **분산-수집 접근법** 필요
  1. 모든 샤드에 대한 상위 10명 랭킹 조회 (병렬화하여 지연 시간 최적화 가능)
  2. 애플리케이션에서 최종 상위 10명 재정렬

단점

- 상위 n명 랭킹 조회를 위해 **분산-수집 접근법** 사용 시 **n이 클수록 지연 시간 증가**
- 특정 사용자의 순위를 조회하기 어려움

→ 결론 : 고정 파티션 쓰자 ~

레디스 노드 크기 조정

쓰기 작업이 많은 애플리케이션은 많은 메모리가 필요함

- 스냅샷 생성 시 필요한 모든 쓰기 연산을 감당해야 하기 때문

- **쓰기 연산이 많은 애플리케이션에는 메모리를 2배 더 할당하는 것이 안전**

성능 벤치마킹을 위해 `redis-benchmark` 도구 제공

- 여러 클라이언트가 동시 질의하는 것을 시뮬레이션
- 주어진 하드웨어로 초당 얼마나 많은 요청을 처리할 수 있는지 측정

대안 : NoSQL

후보 : DynamoDB, 카산드라, MongoDB

쓰기 연산에 최적화되어 있고 효율적 정렬이 가능한 `DynamoDB`를 선택

- **전역 보조 색인**을 제공하여 기본 키 이외의 속성을 활용한 효과적 질의 가능
- **안정 해시**를 사용하여 여러 노드에 데이터를 분산하기 때문에 파티션 키를 월별로 설정하게 되면 **핫 파티션 문제 발생**
  - 쓰기 샤딩 패턴 : 데이터 n개 파티션으로 분할하고 파티션 번호를 파티션 키에 추가
    - 읽기/쓰기 작업을 모두 복잡하게 함
- 🤔 얼마나 많은 파티션을 두어야 하는가?
  - `파티션이 받는 부하`와 `읽기 복잡도` 사이의 트레이드 오프 유의

    같은 달 데이터를 여러 파티션에 분산시키면 ...
    - 한 파티션이 받는 부하는 낮아짐
    - 특정 달의 데이터를 읽기 위해 모든 파티션을 질의한 결과를 합쳐야 함 (**분산-수집 접근법**)

▼ 4️⃣ 마무리

- RDB로 구현하기 vs Redis sorted set으로 구현하기
- 레디스 샤딩해서 규모 확장하기
- NoSQL 이용하기

더 이야기할만한 주제들

더 빠른 조회와 동점자 순위 판정

레디스 해시를 통한 문자열 필드 - 값 사이 대응 관계 저장

1. 랭킹에 표시할 사용자 아이디와 객체 사이의 대응 관계 저장
    - DB 질의 없이 사용자 정보 확인 가능
2. 동점자는 누가 먼저 점수를 받았는지에 따라 순위 판정
    - 사용자 아이디 - 타임스탬프 대응 관계 저장을 통해 오래된 사용자의 순위가 높다고 판정하기

시스템 장애 복구

1. 점수 갱신마다 MySQL에 타임스탬프와 함께 기록
2. 레디스 클러스터 장애 발생 (!)
3. 사용자 별 모든 레코드를 훑으면서 장애 시간 이후의 MySQL 데이터 당 한 번씩 ZINCRBY 호출
4. 복구 완료 ✨