0️⃣

실시간 게임 순위표

문제 이해 및 설계 범위 확정

비기능 요구사항

- 점수 업데트 실시간으로 순위표 반영
- 일반적인 확장성, 사용성 및 안정성 요구사항

개략적 규모 추정

- 게임을 하는 사용자가 24시간 동안 고르게 분포한다고 가정
- DAU가 500만 명인 게임 = 초당 평균 50명의 사용자가 게임을 플레이함
- 사용량이 균등하지 않음
  - 북미 지역 기준 저녁 시간이 피크 시간대일 가능성이 높음
  - 최대 부하는 평균의 5배라고 가정 = 초당 최대 250명의 사용자를 감당해야 함
- 사용자 점수 QPS = 하루 평균 10개의 게임을 플레이한다고 가정 = 50 * 10 = ~500의 최대 5배 = 2500
- 상위 10명 순위표 가져오기 QPS = 각 사용자가 하루에 한 번 게임을 열고 상위 10명 순위표는 사용자가 처음 게임을 열 때만 표시한다고 가정 = 50

개략적 설계안 제시 및 동의 구하기

API 설계

POST	/v1/scores	사용자가 게임에 승리하면 순위표에서 사용자의 순위를 갱신함
		게임 서버에서만 호출할 수 있는 내부 API
			매개변수
			{
			"user_id": "",
			"points": 0
			}

GET	/v1/scores	순위표에서 상위 10명의 플레이어를 가져옴	응답
			{
			"data": [
			"user_id": "",
			"user_name": "",
			"rank": 0,
			"score": 0
			]
			}

GET	/v1/scores/{:user_id}	특정 사용자의 순위를 가져옴	응답
			{
			"user_info": {
			"user_id": "",
			"rank": 0,
			"score": 0
			}
			}

개략적 설계안

게임 서비스

사용자가 게임을 플레이할 수 있도록 함

순위표 서비스

순위표를 생성하고 표시

1. 사용자가 게임에서 승리하면 클라이언트는 게임 서비스에 요청
2. 게임 서비스는 해당 승리가 정당하고 유효한 것인지 검증
3. 순위표 서비스에 점수 갱신 요청
4. 순위표 서비스는 순위표 저장소에 기록된 해당 사용자의 점수를 갱신
5. 해당 사용자의 클라이언트는 순위표 서비스에 데이터 직접 요청
    a. 상위 10명 순위표
    b. 해당 사용자 순위

클라이언트가 순위표 서비스와 직접 통신해야 하나?

- 클라이언트가 점수를 정하는 방식

    - 사용자가 프락시를 설치하고 점수를 마음대로 바꾸는 중간자 공격을 할 수 있기 때문에 보안상 안전하지 않음

  ⇒ 점수는 서버가 설정해야 함

- 온라인 포커처럼 서버가 게임 전반을 통솔하는 경우에는 클라이언트가 점수를 설정하기 위해 게임 서버를 명시적으로 호출할 필요가 없을 수도 있음에 유의
    - 게임 서버가 모든 게임 로직을 처리하고, 게임이 언제 끝나는지 알기 때문에 클라이언트의 개입 없이도 점수를 정할 수 있음

게임 서비스와 순위표 서버 사이에 메시지 큐가 필요한가?

- 점수가 어떻게 사용되는지에 따라 다름
- 점수 데이터가 다른 곳에서도 이용되거나 여러 기능을 지원해야 함 ⇒ 카프카에 데이터를 넣는 것이 합리적
    - 분석 서비스, 푸시 알림 서비스 등 여러 소비자가 동일한 데이터를 사용할 수 있기 때문에
    - 다른 플레이어에게 점수가 바뀌었음을 알려야 하는 순번제 게임이나 멀티플레이어 게임에서는 더욱 필요
- 면접관이 명시적으로 요청한 사항은 아니기 때문에 본 설계안에서는 메시지 큐를 포함시키지 않음

데이터 모델

관계형 데이터베이스

- 가장 간단한 방안
- 규모 확장성이 그다지 중요하지 않고 사용자 수가 많지 않은 경우
- ex) 사용자가 경연에서 승리하면 신규 사용자에게는 1점을 주고, 기존 사용자에게는 기존 점수에 +1

leaderboard	순위표 테이블
user_id	varchar
score	int

    - 실제로 사용할 때는 game_id, 타임스탬프 등의 추가 정보가 있을 것
    - 순위를 조회할 때는 점수를 기준으로 내림차순

    - 특정 사용자의 순위를 조회할 때는 점수를 기준으로 정렬할 수 몇 번째인지 찾으면 됨
- 데이터가 많아지면 성능이 나빠짐
    - 순위를 파악하려면 모든 사용자의 정보를 조회해야 함
    - SQL 데이터베이스는 지속적으로 변화하는 대량의 정보를 신속하게 처리하지 못함
    - 순위가 지속적으로 바뀌기 때문에 캐싱 도입도 불가능
- index를 추가하고 LIMIT 절을 사용하는 방법
    - 규모 확정성이 좋지 않음
    - 특정 사용자의 순위를 조회하려면 결국에는 모든 사용자의 점수를 다 읽어야 함
    - 순위표 상단에 있지 않은 사용자의 순위를 간단하게 찾을 수 없음

Redis

- 메모리에서 동작하기 때문에 빠른 읽기 및 쓰기 가능
- 순위표 시스템 설계 문제를 해결하는 데 이상적인 정렬 집합 자료형 제공

정렬 집합

- 집합과 유사한 자료형
- 정렬 집합에 저장된 각 원소는 점수에 연결되어 있음
    - 집합 내 원소는 고유해야 하지만 같은 점수는 있을 수도 있음
    - 점수는 정렬 집합 내 원소를 오름차순 정렬하는 데 이용
- 내부적으로 해시 테이블과 스킵 리스트라는 두 가지 자료 구조를 사용함
    - 해시 테이블: 사용자의 점수를 저장할 때 사용
    - 스킵 리스트: 특정 점수를 딴 사용자들의 목록을 가져올 때 사용
- 삽입이나 갱신 연산을 할 때 모든 원소가 올바른 위치에 자동으로 배치됨
- 새 원소를 추가하거나 기존 원소를 검색하는 연산의 시간 복잡도가 O(log(n))이기 때문에 관계형 데이터베이스보다 성능이 좋음

스킵 리스트

- 빠른 검색을 가능하게 하는 자료 구조
- 정렬된 연결 리스트에 다단계 색인을 두는 구조
- 근간은 정렬된 단방향 연결 리스트
    - 삽입, 삭제, 검색 연산을 실행하는 시간 복잡도 = O(n)
    - 이진 검색 알고리즘처럼 중간 지점에 더 빨리 도달할 수 있도록 하면 시간 복잡도를 줄일 수 있음
        - 중간 노드를 하나씩 건너뛰는 1차 색인 추가한 다음, 1차 색인 노드를 하나씩 건너뛰는 2차 색인 추가
        - 새로운 색인을 추가할 때마다 이전 차수의 노드를 하나씩 건너뛸 수 있도록 하는 것
        - 노드 사이의 거리가 n-1이 되면 더 이상 색인은 추가하지 않음
    - 데이터 양이 적을 때는 스킵 리스트의 속도 개선 효과가 분명하지 않음

Redis 정렬 집합을 사용한 구현에 사용할 Redis 연산

- ZADD
    - 기존에 없던 사용자를 집합에 삽입
    - 기존 사용자의 경우에는 점수를 업데이트
    - O(log(n))
- ZINCRBY
    - 사용자 점수를 지정된 값만큼 증가
    - 집합에 없는 사용자의 점수는 0에서 시작한다고 가정
    - O(log(n))
- ZRANGE/ZREVRANGE
    - 점수에 따라 정렬된 사용자 중에 특정 범위에 드는 사용자들을 가져옴
    - O(log(n)+m), m = 가져온 항목 수 / n = 정렬 집합 크기
- ZRANK/ZREVRANK
    - 오름차순/내림차순 정렬하였을 때 특정 사용자의 위치를 가져옴
    - O(log(n))

Redis 정렬 집합을 사용한 구현의 동작 원리

- 사용자가 점수를 획득한 경우
  - 매월 새로운 순위표를 위한 정렬 집합을 만들고 이전 순위표는 이력 데이터 저장소로 보냄
  - ZINCRBY를 호출하여 순위표의 사용자 점수를 1만큼 증가시키거나, 아직 순위표 세트에 없는 경우에는 해당 사용자를 순위표 집합에 추가

```
ZINCRBY <키> <증분> <사용자>
ZINCRBY leaderboard_feb_2021 1 'mary1934'
-- 'mary1934' 사용자가 경연에 승리한 경우
```

- 사용자가 순위표 상위 10명을 조회하는 경우
  - ZREVRANK를 호출
  - 각 사용자의 현재 점수도 가져와야 하기 때문에 WITHSCORES 속성도 전달

```
ZREVRANK leaderboard_feb_2021 0 9 WITHSCORES
```

- 사용자가 자기 순위를 조회하는 경우
  - ZREVRANK를 호출
  - 내림차순으로 정렬한 결과를 기준으로 순위를 매겨야 하기 때문에 ZRANK는 사용하지 않음

```
ZREVRANK leaderboard_feb_2021 'mary1934'
```

- 특정 사용자 순위를 기준으로 일정 범위 내 사용자를 질의하는 경우
  - ZREVRANGE를 활용해서 특정한 사용자 전/후 순위 사용자 목록 얻어올 수 있음

```
ZREVRANGE leaderboard_feb_2021 357 365
-- 특정 사용자의 랭크가 361위고, 전/후로 순위 플레이어 4명을 가져오는
```

NoSQL

상세 설계에서 살펴볼 것

저장소 요구사항

- 최소한 사용자 ID와 점수는 저장해야 함
  - 최악의 시나리오: 월간 활성 사용자 2500만 명 모두가 최소 한 번 이상 게임에서 승리해서 모두 월 순위표에 올라야 하는 경우
  - ID가 24자 문자열 / 점수가 16비트 정수 = 26바이트
  - MAU당 순위표 항목이 하나 = 26바이트 * 2500만 = 억 5000만 바이트 or 650MB 저장공간 필요
  - 스킵 리스트 구현에 필요한 오버헤드와 정렬 집합 해시를 고려해 메모리 사용량을 두 배로 늘린다고 해도 최신 Redis 서버 한 대만으로도 데이터를 충분히 저장 가능
- CPU 및 I/O 사용량
  - 개략적 추정치에 따르면 갱신 연산의 최대 QPS = 2500/초
  - 단일 Redis 서버로도 충분히 감당할 수 있음
- 데이터 영속성
  - Redis는 데이터를 디스크에 영속적으로 보관하는 옵션도 지원
  - 디스크에서 데이터를 읽어 대규모 Redis 인스턴스를 재시작하려면 시간이 오래 걸림
  - 보통은 Redis에 읽기 사본을 두는 식으로 구성함
  - 주 서버에 장애가 생기면 읽기 사본을 승격시켜 주 서버로 만들고, 새로운 읽기 사본을 만들어 연결
- 추가적인 성능 최적화 방안
  - 가장 자주 검색되는 상위 10명의 사용자 정보를 캐싱

상세 설계

클라우드 서비스 사용 여부

자체 서비스 이용

- 매월 정렬 집합을 생성하여 해당 기간 순위표를 저장
- 이름 및 프로필 이미지와 같은 사용자 세부 정보 → MySQL 데이터베이스에 저장

- API 서버: 순위 데이터와 더불어 데이터베이스에 저장된 사용자 정보도 같이 조회
- 상위 사용자 10명의 세부 정보 캐싱

AWS 같은 클라우드 서비스 업체 이용

- 아마존 API 게이트웨이
  - RESTful API의 HTTP 엔드포인트를 정의하고 아무 백엔드 서비스에나 연결할 수 있음
- AWS 람다
  - 가장 인기 있는 서버리스 컴퓨팅 플랫폼
  - 서버를 직접 준비하거나 관리할 필요 없이 코드 실행 가능
  - 필요할 때만 실행됨
  - 트래픽에 따라 규모 자동 확장
  - Redis를 호출할 수 있도록 하는 클라이언트를 제공
  - DAU 성장세에 맞춰 자동으로 서비스 규모 확장 가능
- 개략적인 흐름
  1. API 게이트웨이 호출
  2. 게이트웨이가 적절한 람다 함수 호출
  3. 람다 함수는 스토리지 계층(Redis/MySQL)의 명령을 호출하여 얻은 결과를 API 게이트웨이에 반환
  4. API 게이트웨이는 받은 결과를 애플리케이션에 전달

Redis 규모 확장

원래 규모인 5백만 DAU의 100배인 5억 DAU를 처리해야 한다고 가정

최악의 경우: 저장 용량은 65GB, 250000 QPS 질의를 처리할 수 있어야 함

⇒ 샤딩이 필요함

데이터 샤딩 방안

- 고정 파티션

    - 순위표에 등장하는 점수의 범위에 따라 파티션을 나누는 방안
    - 순위표 전반에 점수가 고르게 분포되어야 함
        - 그렇지 않다면 각 샤드에 할당되는 점수 범위를 조정하여 고른 분포가 되도록 해야 함
    - 본 설계안에서는 애플리케이션이 샤딩 처리 주체가 된다고 가정
        - 특정 사용자의 점수를 입력하거나 갱신할 때는 해당 사용자가 어느 샤드에 있는지 알아야 함
        - MySQL 질의를 통해 사용자의 현재 점수를 계산하여 알아낼 수 있음
        - 사용자 ID와 점수 사이의 관계를 저장하는 2차 캐시를 통해 성능을 높일 수 있음
        - 사용자의 점수가 높아져서 다른 샤드로 옮겨야 할 때는 기존 샤드에서 해당 사용자를 제거한 다음 새 샤드로 옮겨야 한다는 점 유의
    - 특정 사용자의 순위를 알려면 해당 사용자가 속한 샤드 내 순위뿐 아니라 해당 샤드보다 높은 점수를 커버하는 모든 샤드의 모든 사용자 수를 알아야 함
        - info keyspace 명령을 사용하여 O(1) 시간에 알아낼 수 있음
- 해시 파티션
    - Redis 클러스터를 사용하는 방안
        - 여러 노드에 데이터를 자동으로 샤딩하는 방법을 제공
    - 사용자들의 점수가 특정 대역에 과도하게 모여 있는 경우 효과적
    - 각각의 키가 특정한 해시 슬롯에 속하도록 하는 샤딩 기법을 사용
        - 총 16384개 해시 슬롯이 있음
        - `CRC16(key) % 16384`의 연산을 수행하여 어떤 키가 어느 슬롯에 속하는지 계산함
        - 모든 키를 재분배하지 않아도 쉽게 노드를 추가/삭제 가능
    - 점수를 갱신하려면 해당 사용자의 샤드를 찾아 거기서 해당 사용자 점수를 변경하기만 하면 됨
    - 상위 10명의 플레이어를 검색할 때는 모든 상위 10명을 받아 애플리케이션 내에서 다시 정렬하는 분산-수집 접근법 사용
        - 모든 샤드에 사용자를 질의하는 절차를 병렬화하면 지연 시간을 줄일 수 있음
    - 발생하는 문제

        - 상위 k개의 결과를 반환해야 하는 경우, 각 샤드에서 많은 데이터를 읽고 정렬해야 하기 때문에 지연 시간이 실어짐
        - 가장 느린 파티션에서 데이터를 다 읽고 나서야 질의 결과를 계산할 수 있기 때문에 지연 시간이 길어짐
        - 특정 사용자의 순위를 결정할 간단한 방법이 없음

    ⇒ 고정 파티션 방안을 사용할 것

- Redis 노드 크기 조정
    - 쓰기 작업이 많은 애플리케이션에는 많은 메모리가 필요함
        - 장애에 대비해 스냅숏을 생성할 때 필요한 모든 쓰기 연산을 감당할 수 있어야 하기 때문
    - 성능 벤치마킹을 위해 redis-benchmark라는 도구를 제공함
        - 여러 클라이언트가 동시에 여러 질의를 실행하는 것을 시뮬레이션하여 주어진 하드웨어로 초당 얼마나 많은 요청을 처리할 수 있는지 측정함

대안: NoSQL

- 쓰기 연산에 최적화되어 있음
- 같은 파티션 내의 항목을 점수에 따라 효율적으로 정렬 가능
- DynamoDb, MongoDb, 카산드라, ...
    - 안정적인 성능과 뛰어난 확정성을 제공하는 완전 관리형 NoSQL 데이터베이스
    - 기본 키 이외의 속성을 활용하여 데이터를 효과적으로 질의할 수 있도록 전역 보조 색인을 제공함
        - 부모 테이블의 속성들로 구성되지만 기본 키는 부모 테이블과 다름

체스 게임 순위표 설계

- 순위표와 사용자 테이블을 비정규화
    - 순위표를 화면에 표시하는 데 필요한 모든 정보를 담고 있음
    - 규모 확장이 어려움
    - 레코드가 많아지면 상위 점수를 찾기 위해 전체 테이블을 뒤져야 하기 때문에 성능이 떨어짐

⇒ game_name#{year-month}를 파티션 키로, 점수를 정렬 키로 사용하여 테이블 전체를 읽어야 하는 일을 피할 수 있음

- 부하가 높을 때 문제임
  - 안정 해시를 사용해 여러 노드에 데이터를 분산
  - 가장 최근 한 달치 데이터가 동일한 파티션에 저장됨 ⇒ 핫 파티션이 됨

⇒ 데이터를 n개 파티션으로 분할하고 파티션 번호를 파티션 키에 추가하는 방식을 사용해서 해결

⇒ 쓰기 샤딩 패턴

- 읽기 및 쓰기 작업을 모두 복잡하게 만듦
- 장단점을 꼼꼼히 따져봐야 함

얼마나 많은 파티션을 두어야 하는가?

- 쓰기 볼륨 or DAU를 기준으로 결정 가능
- 파티션이 받는 부하와 읽기 복잡도 사이에는 타협적인 부분이 있음에 유의
  - 같은 달 데이터를 여러 파티션에 고르게 분산시키면 한 파티션이 받는 부하는 낮아짐
  - 특정한 달의 데이터를 읽으려고 하면 모든 파티션을 질의한 결과를 함쳐야 하기 때문에 구현은 훨씬 복잡함
- 전역 보조 색인은 game_name#{year-month}#p{partition_number}를 파티션 기로, 점수를 정렬 키로 사용하도록 구성
- 같은 파티션 내 데이터는 전부 점수 기준으로 정렬된 n개의 파티션이 만들어짐
- 분산-수집 접근법을 통해 상위 10명의 사용자 조회

파티션 수는 어떻게 정하나?

- 신중한 벤치마킹이 필요함
- 파티션이 많으면 각 파티션의 부하는 줄지만 최종 순위표를 만들기 위해 읽어야 하는 파티션은 많아지기 때문에 복잡성이 증가함
- 사용자의 상대적 순위를 쉽게 정할 수 없지만 사용자 위치의 백분위수를 구하는 것은 가능함

- 규모가 커서 샤딩이 필요한 상황이라면 모든 샤드의 점수 분포는 거의 같다고 가정할 수 있음
    - 이 가장이 사실이라면 각 샤드의 점수 분포를 분석한 결과를 캐싱하는 크론 작업을 만들어 볼 수 있음

마무리

더 빠른 조회 및 동점자 순위 판정 방안

- Redis 해시를 사용하면 문자열 피드와 값 사이의 대응관계를 저장해 둘 수 있음
    - 순위표에 표시할 사용자 ID와 사용자 객체 사이의 대응관계를 저장해서 데이터베이스에 질의하지 않아도 빠르게 사용자 정보 확인 가능
    - 두 사용자의 점수가 같은 경우 누가 먼저 점수를 받았는지에 따라 순위를 매길 수 있음
        - 사용자 ID와 해당 사용자가 마지막으로 승리한 경기의 타임 스탬프 사이의 대응관계를 저장해 동점자가 나오면 기록된 타임스탬프 값이 오래된 사용자 순위가 높다고 처리

시스템 장애 복구

- 사용자가 게임에서 이길 때마다 MySQL 데이터베이스에 타임스탬프와 함께 그 사실을 기록한다는 사실을 활용하는 스크립트를 만들어 간단히 복구 가능
- 사용자별로 모든 레코드를 훑으면서 레코드당 한 번씩 ZINCRBY 호출

⇒ 대규모 장애가 발생했을 때 오프라인 상태에서 순위표 복구 가능