근접성 서비스 (240103)

💡 현재 위치에서 가까운 _____ 찾기

1️⃣ 설계 범위 확정

검색 반경

- 사용자 지정?
- 검색 결과 없는 경우
  - 시스템이 검색 반경 자동 확장?
- 최대 허용 반경?

사업장 정보

- 어떻게 관리?
  - 소유주가 직접 CUD
  - 변경 결과 실시간 반영?

기타

- 이동 중에 사용하는 경우
  - 검색 반경 자동 갱신?

2️⃣ 요구사항 정리

기능 요구사항

1. 사용자의 위치(위도, 경도)와 검색 반경 정보에 매치되는 **사업장 목록 반환**
2. 사업장 소유주는 **사업장 정보를 CUD** 가능
3. 변경된 사업장 정보는 실시간 반영되지 않음
4. 고객은 **사업장 상세 정보 조회** 가능

비기능 요구사항

낮은 응답 지연

데이터 보호

고가용성

3️⃣ 개략적 규모 추정

💡 **DAU**는 1억명, **등록된 사업장 수**는 2억개, 각 사용자는 **매일 5회 검색**한다고 가정

QPS

```
1억명 * 5회 / 86400초
```

계산을 쉽게 하기 위해 하루를 100000초라고 가정

```
1억명 * 5회 / 100000초 = 5000
```

⇒ **QPS : 5000**

4️⃣ 개략적 설계

API 설계

사용자 관련

API	설명
GET /v1/search/nearby	특정 검색 기준에 맞는 사업장 목록 반환

필드	설명	자료형
latitude	위도	decimal
longitude	경도	decimal
radius	(optional) 반경(default 5km)	int

```
{
    "total": 10,
    "businesses": [{business object}]
}
```

사업장 관련

API	설명
GET /v1/businesses/:id	특정 사업장의 상세 정보 반환
POST /v1/businesses	새로운 사업장 추가
PUT /v1/businesses/:id	사업장 상세 정보 갱신
	특정 사업장 정보 삭제

DELETE
/v1/businesses/:id

데이터 모델

읽기/쓰기 비율

읽기 > 쓰기

- 주변 사업장 검색, 사업장 정보 확인 기능 이용 빈도가 높기 때문

⇒ 읽기 연산이 압도적인 시스템은 관계형 데이터베이스가 바람직함!

데이터 스키마

business(사업장)	**지리적 위치 색인**
(PK) business_id	쫌이따.. 😉
address	
city	
state	
country	
latitude	
longitude	

개략적 설계안

시스템은 위치 기반 서비스(LBS)와 사업장 관련 서비스로 구분된다.

로드밸런서

유입 트래픽을 여러 서비스에 분산시키는 컴포넌트

단일 entry point를 지정하고 URL 경로를 분석하여 `LBS or 사업장 서비스`로 트래픽 전달

LBS

- **읽기 요청만** 발생하는 서비스
- QPS가 높음
  - 특정 시간 / 인구 밀집 지역일수록 높음

- 무상태 서비스

사업장 서비스

- 사업장 소유주가 사업장 정보를 CUD
  - 기본적으로 **쓰기 요청**
  - QPS 높지 않음
- 고객이 사업장 정보를 조회
  - 특정 시간에 QPS 높음
- 무상태 서비스

> 💡 **LBS와 사업장 서비스의 규모 확장성**
>
> 둘 다 무상태 서비스이므로 특정 시간 몰리는 트래픽에는 자동으로 서버를 추가 / 유휴 시간대에는 서버를 삭제하여 대응 가능
> 클라우드 서비스라면 여러 AZ에 서버를 두어
> **시스템 가용성을 높일 수 있음**

데이터베이스 클러스터

- 주-부 데이터베이스 형태
  - 주: 쓰기 요청 처리
  - 부: 읽기 요청 처리
- 주 데이터베이스 → 부 데이터베이스 데이터 복제 시간 지연으로 인해 차이가 있을 수 있음

주변 사업장 검색 알고리즘

- 레디스 지오해시
- PostGIS extension을 설치한 postgres
- …

⇒ 어떤 방안이 있고 어떻게 동작하는지 설명하자.

▼ 1. 2차원 검색

주어진 반경으로 그린 원 안에 놓인 사업장을 검색하기

```
SELECT business_id, latitude, longitude
FROM business
WHERE (latitude BETWEEN {:my_lat} - radius AND {:my_lat} +
AND (longitude BETWEEN {:my_long} - radius AND {:my_long} +
```

단점

- 테이블 전부를 읽어야하므로 **비효율적**
  - latitude, longitude에 index를 건다면? 별루
    - 데이터가 2차원적이므로 컬럼별로 가져온 결과도 엄청난 양이다…
    - 두 집합의 교집합을 구해야하므로 비효율적

> 🤔 **2차원 데이터를 한 차원에 대응시키면 안될까?**
>
> 지리적 정보에 인덱스를 만들자!
>
> 지도를 작은 영역으로 분할하고 고속 검색이 가능하도록 하자!
>
> - 해시 기반 방안:
> **균등 격자**, **지오해시**, 카르테시안 계층 등
> - 트리 기반 방안:
> **쿼드트리**, **구글 S2**, R-tree 등

▼ 2. 균등 격자

지도를 작은 격자로 나누기

단점

- 사업장 분포가 균등하지 않음 바다에는 사업장이 없죠
  - 인구 밀집 지역에는 작은 격자를, 나머지는 큰 격자를 사용하면 안될까?
- 주어진 격자의 인접 격자를 찾기 어려움 격자 식별자 할당에 체계가 없기 때문

▼ ⭐ 3. 지오해시

2차원 위도, 경도 데이터를 1차원 문자열로 변환하기

동작 방식

1. 전 세계를 사분면으로 나눔
2. 각각의 격자를 또 다시 사분면으로 나눔
3. 원하는 정밀도를 얻을 때까지 반복
4. base32 표현법으로 표기 (ex: 1001 11010 01001 11111 11110)

지오해시는 12단계 정밀도를 갖고 **4-6 사이 단계**가 적당하다.

최적 정밀도를 정하는 방법은 지정한 반경으로 그린 원을 덮는 최소 크기 격자를 만드는 지오해시 길이를 구하면 된다.

---

edge case 처리하기

1. **격자 가장자리 관련 이슈 (1)**

지오해시는 해시값의 공통 접두어가 긴 격자들이 서로 더 가깝게 놓이도록 보장한다.

하지만 아주 가까운 두 위치가 공통 접두어를 갖지 않을수도 있다.

- 두 지점이 적도의 서로 다른 쪽에 놓이거나
- 자오선상의 다른 반쪽에 놓이는 경우

⇒ 쉽게 말해 다른 사분면에 있게 되는 경우

따라서 아래 SQL문을 사용했을 때 이 이슈가 발생하게 된다.

```
SELECT * FROM geohash_index WHERE geohash LIKE '9q8zn%'
```

2. **격자 가장자리 관련 이슈 (2)**

공통 접두어 길이는 길지만 서로 다른 격자에 놓이는 경우 문제가 발생한다.

현재 격자를 비롯한 인접한 모든 격자의 사업장 정보를 가져옴으로써 해결할 수 있다.

3. **표시할 사업장이 적음**

인접한 격자를 살펴도 표시할 사업장이 부족할 때에도 문제가 발생한다.

1) **주어진 반경 내 사업장만 반환**하거나 2) **검색 반경을 더 키우는** 방법이 있다.

▼ ⭐ 4. 쿼드트리

격자의 내용이 특정 기준을 만족할 때까지 재귀적으로 사분면을 분할하기

ex) 격자 내 사업장 수가 100개 이하가 될때까지 분할

```
public void buildQuadTree(TreeNode node) {
    if (countNumberOfBusinessesInCurrentGrid(node) > 100) {
        node.subdivide();
        for (TreeNode child : node.getChildren()) {
            buildQuadTree(child);
        }
    }
}
```

특징

- 메모리 기반(자료 구조일 뿐 데이터베이스가 아님)
- 각각의 LBS 서버에 존재해야 함

얼마나 많은 메모리가 필요할까?

1. 어떤 데이터가 쿼드트리에 보관되는가?
    - 말단 노드

이름	크기
격자 식별 위한 좌상단, 우하단 꼭짓점 좌표	32Byte (8Byte * 4)
격자 내부 사업장 ID 목록	(ID 당 8Byte) * 100(한 격자 내 사업장 수 최대값)
합계 : 832Byte	

    - 내부 노드

▼ 이름	크기
격자 식별 위한 좌상단, 우하단 꼭짓점 좌표	32Byte (8Byte * 4)
하위 노드 4개를 가리킬 포인터	32Byte (8Byte * 4)
합계 : 64Byte	

2. 메모리 사용량 계산

- 격자 안에는 최대 100개 사업장이 있을 수 있다.
- 말단 노드의 수는 약 200M / 100 = 200만

- 내부 노드의 수는 2m * 1/3 = 약 67만
- **총 메모리 요구량** = 200만 * 832Byte + 67만 * 64Byte = **약 1.71GB**

  ⇒ 생각보다 메모리를 많이 잡아먹지 않으므로 서버 한 대에 충분히 올릴 수 있다.

전체 쿼드트리 구축에 소요되는 시간?

200만개 사업장 정보를 인덱싱하는 쿼드트리 구축에는 몇 분 정도 소요된다.

쿼드트리로 주변 사업장을 검색하려면?

1. 메모리에 쿼드트리 인덱스 구축
2. 검색 시작점이 포함된 말단 노드를 만날 때까지 루트 노드부터 탐색
   a. 해당 노드에 100개 사업장이 있는 경우 해당 노드만 반환
   b. 아니면 충분한 사업장 수가 확보될 때까지 인접 노드 추가

쿼드트리 운영 시 고려할 점

- 서버 시작시 트리를 구축하면 서버 시작 시간이 길어진다.
  - 서버 소프트웨어를 배포할 때에는 동시에 너무 많은 서버에 배포하지 않도록 주의
- 사업장이 추가/삭제 되었을 때 쿼드 트리 갱신
  - 점진적으로 갱신하기
  - 밤 사이에 캐시를 일괄 갱신하기
    - 수많은 키가 한 번에 무효화되어 캐시 서버에 부하 발생
  - 실시간 갱신
    - 설계가 복잡해짐
    - 여러 스레드가 쿼드트리 자료 구조를 동시 접근한다면…

▼ 5. 구글 S2

메모리 기반, 지구를 힐베르트 곡선이라는 공간 채움 곡선을 사용하여 1차원 색인화하기

힐베르트 곡선

- 인접한 두 지점은 색인화 이후 1차원 공간 내에서도 인접한 위치에 있다.

특징

- 임의 지역에 다양한 수준의 영역 지정 가능

- 영역 지정 알고리즘
    - 최소 수준, 최고 수준, 최대 셀 개수를 지정할 수 있음

지오해시 vs 쿼드트리

지오해시

- 구현/사용 쉬움 (트리 구축 x)
- 지정 반경 내 사업장 검색 지원
- 정밀도 고정 시 격자 크기 고정(동적 지정 x)
- 색인 갱신이 쉬움

쿼드트리

- 구현 어려움 (트리 구축 o)
- k번째 가까운 사업장까지의 목록 구하기 가능
- 인구 밀도에 따라 격자 크기 동적 조정 가능
- 색인 갱신 어려움

5️⃣ 상세 설계

데이터베이스 규모 확장

사업장 테이블

- 샤딩 후보!
- 사업장 ID를 기준으로 샤딩하자

지리 정보 색인 테이블

- 전체 데이터 양은 많지 않음
- 읽기 연산 빈도가 높을 시 샤딩하거나 사본 데이터 서버를 늘릴 수 있음
- 샤딩 로직이 까다로우므로 사본 서버를 두자

캐시

1) 부하는 읽기 중심이고 **2) 데이터베이스 크기는 상대적으로 작아서** 서버 한 대에 수용 가능하다.

읽기 성능에 병목이 발생한다면 사본 데이터베이스를 증설하는 것이 바람직할지도 … 캐시도 돈이야

캐시 키

가장 직관적인 키는 사용자 위치의 위도, 경도 정보

문제점

- 전화기에서 반환되는 위치 정보는 추정치일뿐…
- 사용자가 이동하면 위도, 경도도 미세하게 변경됨. 대부분의 경우 이 변화는 아무런 의미가 없다.

⇒ 지오해시, 쿼드트리를 사용해서 같은 격자 내 모든 사업장이 같은 해시 값을 갖도록 하자~

캐시 데이터 유형

어떤 데이터를 보관해야 성능을 향상시킬 수 있을까?

key	value
지오해시	해당 격자 내 사업장 ID 목록
사업장 ID	사업장 정보 객체

우리는 이 2개의 레디스 캐시를 각각 지오해시, 사업장 정보라고 부르기로 했어요

격자 내 사업장 ID

사업장 정보는 자주 변경되지 않는다.

- 특정 지오해시에 해당하는 사업장 ID 목록을 미리 계산해서 캐싱하자.
- 사업장 정보가 변경되면 캐시에 보관된 항목을 무효화하자.
- 사용자는 검색 반경을 선택(500m, 1km, 2km, 5km)할 수 있으므로 검색 반경에 따른 검색 결과를 캐싱하자.
- 고가용성을 보장하고 latency를 줄이기 위해 레디스 클러스터를 전 세계에 지역별로 두고 동일한 데이터를 각 지역에 중복해서 저장하자.

region 및 AZ

- 사용자 - 시스템 간 거리를 최소한으로 한다.
- 트래픽을 인구에 따라 고르게 분산한다.
- 각 지역의 사생활 보호법에 맞게 운영한다.

+) 시간대 / 사업장 유형에 따른 검색

> 🤦🏻 : **지금 영업중**인 **카페**만 받아오려면 어떻게 해야하죠?

지오해시, 쿼드트리와 같은 메커니즘을 통해 세계를 작은 격자로 분할하면 결과로 얻어지는 수는 상대적으로 적음

이를 먼저 확보한 후 사업장 정보를 추출해서 필터링하자

최종 아키텍처

주변 반경 500m 내 모든 식당 검색

1. 클라이언트 앱은 사용자의 위도, 경도와 검색 반경(500m)을 로드밸런서로 전송
2. 로드밸런서는 해당 요청을 LBS로 전송
3. LBS는 검색 요건을 만족할 지오해시 길이를 계산
4. LBS는 인접한 지오해시를 계산한 다음 목록에 추가
5. 지오해시 목록에 있는 요소 각각에 대해 캐시 서버(지오해시)를 조회하여 해당 지오해시에 대응하는 모든 사업장 ID 추출
    a. 지오해시별 사업장 ID를 가져오는 연산을 병렬로 수행하여 latency를 줄이자.
6. 반환된 사업장 ID 목록을 가지고 캐시 서버(사업장 정보)를 조회하여 사업장의 상세 정보 취득
7. 상세 정보에 의거하여 사용자와의 거리를 확실하게 계산, 우선순위 선정
8. 클라이언트 앱에 결과 반환

사업자 정보 CRUD

1. 캐시 서버(사업장 정보)를 조회하여 기록 여부 판단
2. 캐싱된 경우 해당 데이터를 반환
3. 없는 경우 데이터베이스 클러스터에서 읽어 캐시에 저장 후 반환
4. 추가/갱신 정보는 밤에 처리

6️⃣ 마무리

오늘 얻은 것

- LBS 서비스가 무엇인가
- 색인 방법
    - 2차원 검색
    - 균등 분할 격자

    - 지오해시
    - 쿼드트리
    - 구글 S2
- 캐시 활용
- 데이터베이스 규모 확장
- 가용성 확장

궁금한 점

- 읽기 연산이 압도적으로 많은 시스템에 왜 RDB가 적합한가?
- 사본 데이터베이스 멋진 말로 뭐라고 하나요? 레플리카?