공간데이터 저장기

우아한테크코스 7기 BE 돔푸

목차

1부 : 바보 시절

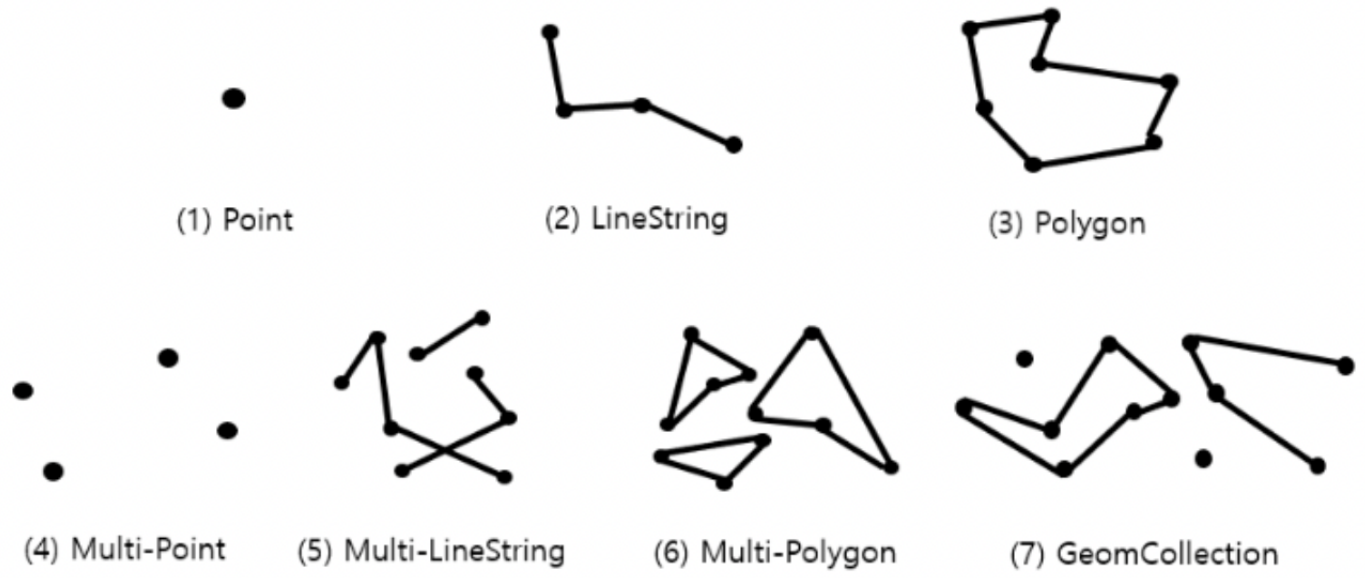
공간인덱스에 대한 학습없이 직관으로 개발하던 시기입니다.
‘얘네는 바보인가?’ 정도의 생각이 드시면 잘 듣고 계시는 겁니다.

먼저 도메인을 잘 이해하시고,
‘나라면 어떻게 저장했을까?’란 상상을 하며 들으시면 재밌으실 겁니다.

공간데이터란?

: **지구상의 특정 기하학적 형태**와 그 속성

- 점 : 교촌치킨
- 선 : 교촌치킨 → 집까지의 배달 경로
- 면 : 교촌치킨의 배달 가능 반경
- 이외에도 많은 종류가 존재한다.



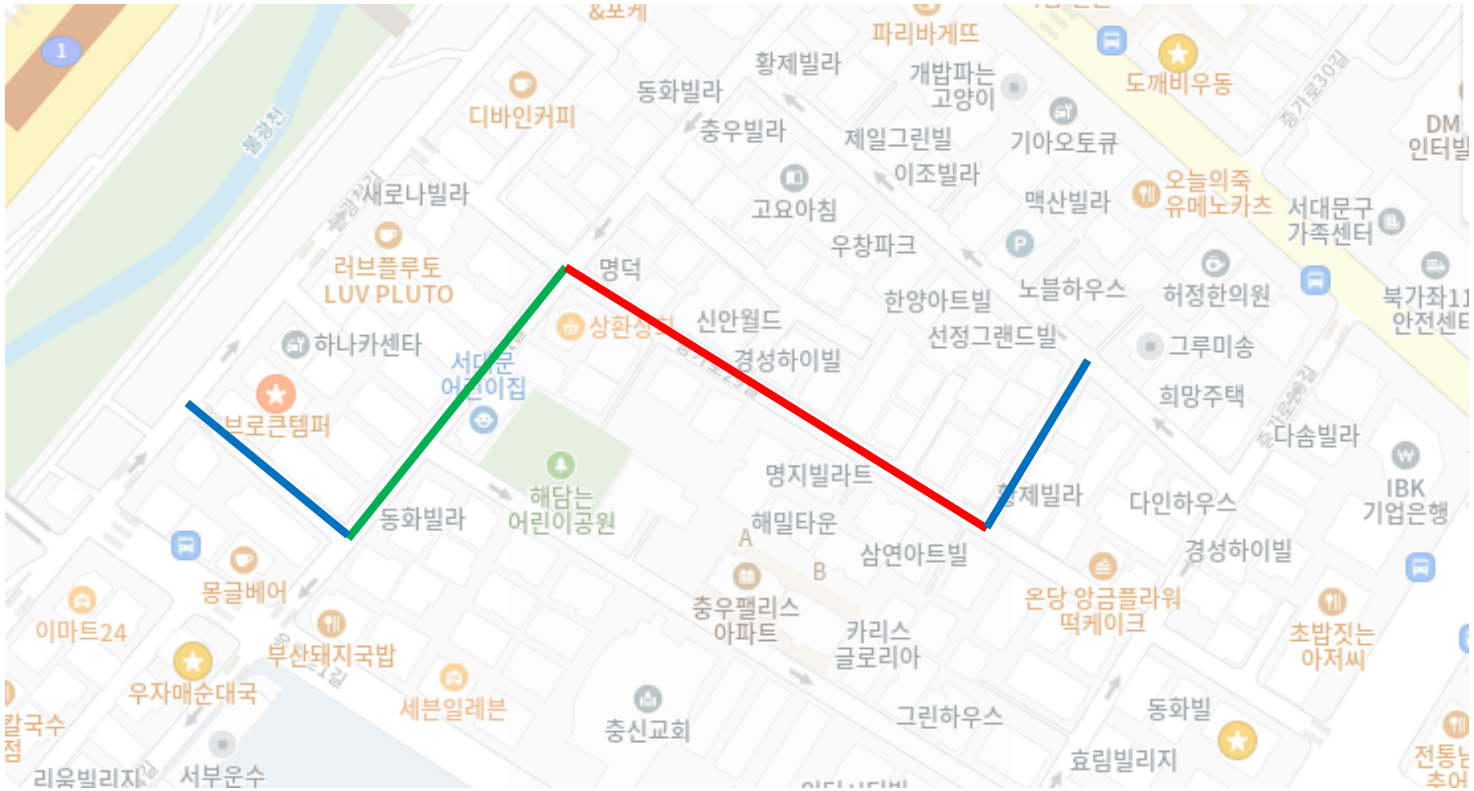
런세권의 도메인 모델

```
Course {
  id: String
  name: String
  roadType: String
  inclineSummary: String
  length: Integer
  difficulty: String
  segments: List<Segment>
}
```

```
Segment {
  inclineType: String
  lines: List<GeoLine>
}
```

```
GeoLine {
  start: Coordinate
  end: Coordinate
}
```

```
Coordinate {
  latitude: Double
  longitude: Double
  elevation: Double
}
```



도메인모델에 충실하게 저장하기

```
Course {
  id: String
  name: String
  roadType: String
  inclineSummary: String
  length: Integer
  difficulty: String
  segments: List<Segment>
}
```

```
Segment {
  inclineType: String
  lines: List<GeoLine>
}
```

```
GeoLine {
  start: Coordinate
  end: Coordinate
}
```

```
Coordinate {
  latitude: Double
  longitude: Double
  elevation: Double
}
```

COURSES	
id	VARCHAR
name	VARCHAR
road_type	VARCHAR
incline_summary	VARCHAR
length	INT
difficulty	VARCHAR

SEGMENTS	
id	VARCHAR
incline_type	VARCHAR
course_id	VARCHAR

GEOLINES	
incline_type	VARCHAR
start_latitude	DOUBLE
start_longitude	DOUBLE
start_elevation	DOUBLE
end_latitude	DOUBLE
end_longitude	DOUBLE
end_elevation	DOUBLE
segment_id	VARCHAR

도메인모델에 충실하게 저장하기

: 대량의 row를 반환해야 했으며, 목표 속도에 도달하지 못했다.

1개 Course → 평균 20개 Segment

1개 Segment → 평균 500개 GeoLine

⇒ 1개 Course 에서 **평균 10000개 row 반환**

+ 테이블을 나눌 이유가 없었다.
  모든 Course 이하의 데이터는 생명주기가 완전히 동일하다.

JSON 타입으로 저장하기

```
Course {
  id: String
  name: String
  roadType: String
  inclineSummary: String
  length: Integer
  difficulty: String
  segments: List<Segment>
}
```

```
Segment {
  inclineType: String
  lines: List<GeoLine>
}
```

```
GeoLine {
  start: Coordinate
  end: Coordinate
}
```

```
Coordinate {
  latitude: Double
  longitude: Double
  elevation: Double
}
```

COURSES	
id	VARCHAR
name	VARCHAR
road_type	VARCHAR
incline_summary	VARCHAR
length	INT
difficulty	VARCHAR
geo_data	JSON

```
[
  {
    “inclineType”: “오르막”,
    “geoLines”: [
      {
        “start”: [10, 20, 30],
        “end”: [10, 20, 30]
      },
      {
        “start”: [10, 20, 30],
        “end”: [10, 20, 30]
      }, …
    ]
  }, …
]
```

JSON 타입으로 저장하기

: 조인 연산이 없고, row 수가 적으므로 목표 속도에는 도달했다.

하지만 결국 '모두 찾아와서 애플리케이션에서 필터링' 해야 했다.

코스가 늘어나며, 특히 데이터 크기가 괴랄한 트레일 러닝 코스가 추가되면서
속도는 저하될 것이 분명했다...

러닝 코스 평균 크기 : 1MB, 서울 지역 예상 개수 500개
⇒ **매번 500MB 조회**

2부 : 공부 후

일단 급한 불은 꺼놓고, 공부 후에 적용한 시기입니다.
실질적으로 추후 사용할 지식은 여기서 나옵니다.

MySQL과 MongoDB에서 공간데이터와 공간인덱스를 다루는 법을 알아봅시다.

공간인덱스란?

: **공간데이터를 빠르게 쿼리**하기 위한 특수 인덱스

무서워하지 않아도 된다. 결국 어떠한 Tree 구조를 만들고,
이를 통해 로그 스케일로 탐색 속도를 줄이는 것이다.

데이터베이스마다 구현이 다른데, 이는 부록에서 다룬다.

MySQL에서 공간인덱스 활용하기

공간데이터 타입을 활용하고, **공간인덱스**를 생성하면 된다.

COURSES	
id	VARCHAR
name	VARCHAR
road_type	VARCHAR
incline_summary	VARCHAR
length	INT
difficulty	VARCHAR
segments	MULTILINESTRING

```
(
  (10 10 10, 20 20 20, 30 30 30),
  (30 30 30, 40 40 40, 50 50 50)
)
```

```
CREATE SPATIAL INDEX idx_segments ON COURSES(segments)
```

MySQL에서 공간인덱스 활용하기

COURSES	
id	VARCHAR
name	VARCHAR
road_type	VARCHAR
incline_summary	VARCHAR
length	INT
difficulty	VARCHAR
segments	MULTILINESTRING

오르막/내리막 정보를 데이터베이스에 저장할 수 없다.

하지만 segments 정보에서 이를 다시 계산할 수 있었기 때문에, 괜찮았다.

```
(
  (10 10 10, 20 20 20, 30 30 30),
  (30 30 30, 40 40 40, 50 50 50)
)
```

```
CREATE SPATIAL INDEX idx_segments ON COURSES(segments)
```

MySQL에서 공간인덱스 활용하기

COURSES	
id	VARCHAR
name	VARCHAR
road_type	VARCHAR
incline_summary	VARCHAR
length	INT
difficulty	VARCHAR
segments	MULTILINESTRING

원하는 쿼리 : 기준점 기준 1KM 이내의 모든 코스 조회

1. 기준점이 중심이 되는 폭 1KM의 정사각형을 그린다.
2. MySQL의 ST_Within 연산을 활용하여 내부의 모든 코스데이터를 로드한다.
3. 코스데이터와 점 사이 거리를 ST_Distance 연산으로 계산하여 필터링한다.

```
SET @center = ST_GeomFromText('POINT(37.4979 127.0276)', 4326);

SELECT * FROM courses
WHERE
  ST_Within(segments, ST_Envelope(ST_Buffer(@center, 500)))
AND
  ST_Distance(@center, segments) ⩽ 500;
```

MySQL에서 공간인덱스 활용하기

: 확실히 빠르다.

하지만 억지로 RDB를 활용하는 듯한 느낌을 지우지 못했다.
- 테이블을 제대로 활용하지 못하는 도메인 구조
- 무결성/정합성 등의 트랜잭션 규칙 필요 X

다른 데이터베이스?

1. Course 이하 데이터의 생명주기가 동일하다 .(같이 생성/수정/삭제)
2. 데이터가 깊은 중첩구조를 가진다.
3. 공간인덱스를 활용한 빠른 쿼리가 필요하다.

위 사실들을 바탕으로 더 좋은 데이터베이스를 찾았고,
DocumentDB가 해답이었다.

그 중에 MongoDB가 가장 문서화가 잘 되어 있었기 때문에 채택했다.

MongoDB에서 공간인덱스 활용하기

GeoJSON 형태로 저장하고, **공간인덱스**를 생성하면 된다.

```
{
  "_id": "695a30cae28d2e28d7ee0c83",
  "name": "예시코스",
  "roadType": "트랙",
  "inclineSummary": "REPEATING_HILLS",
  "segments":
  {
    "type": "MultiLineString",
    "coordinates": [
      [
        [10, 10, 10],
        [20, 20, 20],
        [30, 30, 30]
      ],
      [
        [30, 30, 30],
        [40, 40, 40],
        [50, 50, 50]
      ]
    ]
  },
  "length": 500.0,
  "difficulty": "쉬움"
}
```

```
db.courses.createIndex({ segments: "2dsphere" })
```

MongoDB에서 공간인덱스 활용하기

```
{
  "_id": "695a30cae28d2e28d7ee0c83",
  "name": "예시코스",
  "roadType": "트랙",
  "inclineSummary": "REPEATING_HILLS",
  "segments":
  {
    "type": "MultiLineString",
    "coordinates": [
      [
        [10, 10, 10],
        [20, 20, 20],
        [30, 30, 30]
      ],
      [
        [30, 30, 30],
        [40, 40, 40],
        [50, 50, 50]
      ]
    ]
  },
  "length": 500.0,
  "difficulty": "쉬움"
}
```

원하는 쿼리 : 기준점 기준 1KM 이내의 모든 코스 조회

→ MongoDB의 $near 연산을 활용하여 조회한다.

```
db.courses.find({
  "segments": {
    $near: {
      $geometry: {
        type: "Point",
        coordinates: [127.0276, 37.4979]
      },
      $maxDistance: 1000
    }
  }
});
```

MySQL vs MongoDB

: MongoDB가 더 빨랐다.

MongoDB가 중첩된 데이터를 저장하기에 더 적합하면서도,
속도마저 더 빨랐다.

... 왜?

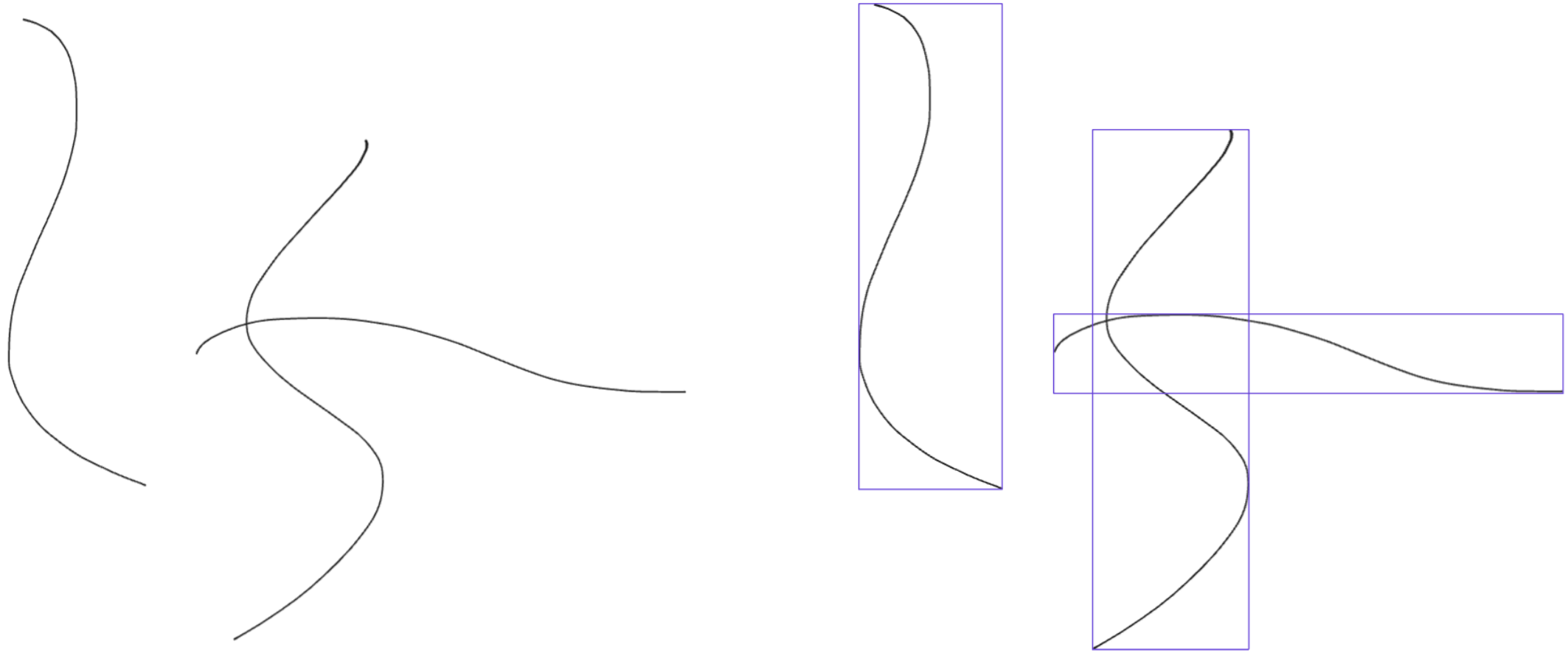
부록 : 공간인덱스 구현

MySQL이 왜 MongoDB에 비해 속도가 느렸는지 알아보는 시간입니다.

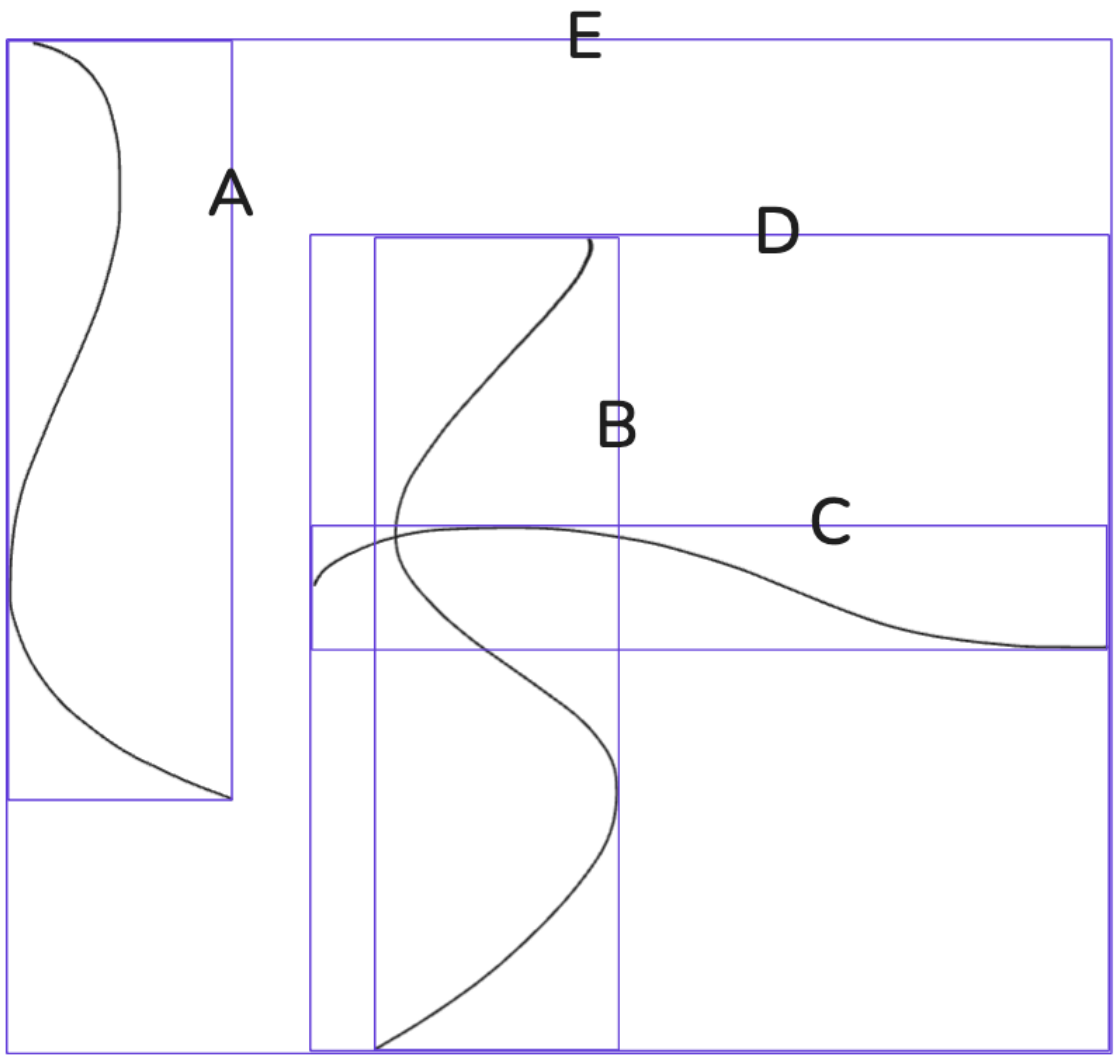
어려울 수 있는데, 최대한 쉽게 설명드리겠습니다.

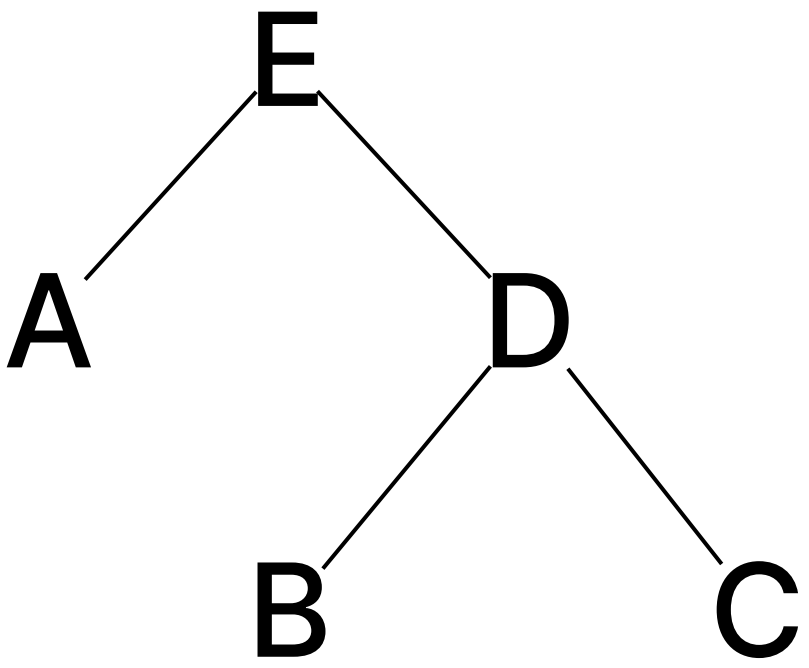
MySQL의 공간인덱스 구현

: **공간데이터를 감싸는 가장 작은 사각형**의 포함 관계를 사용한다.

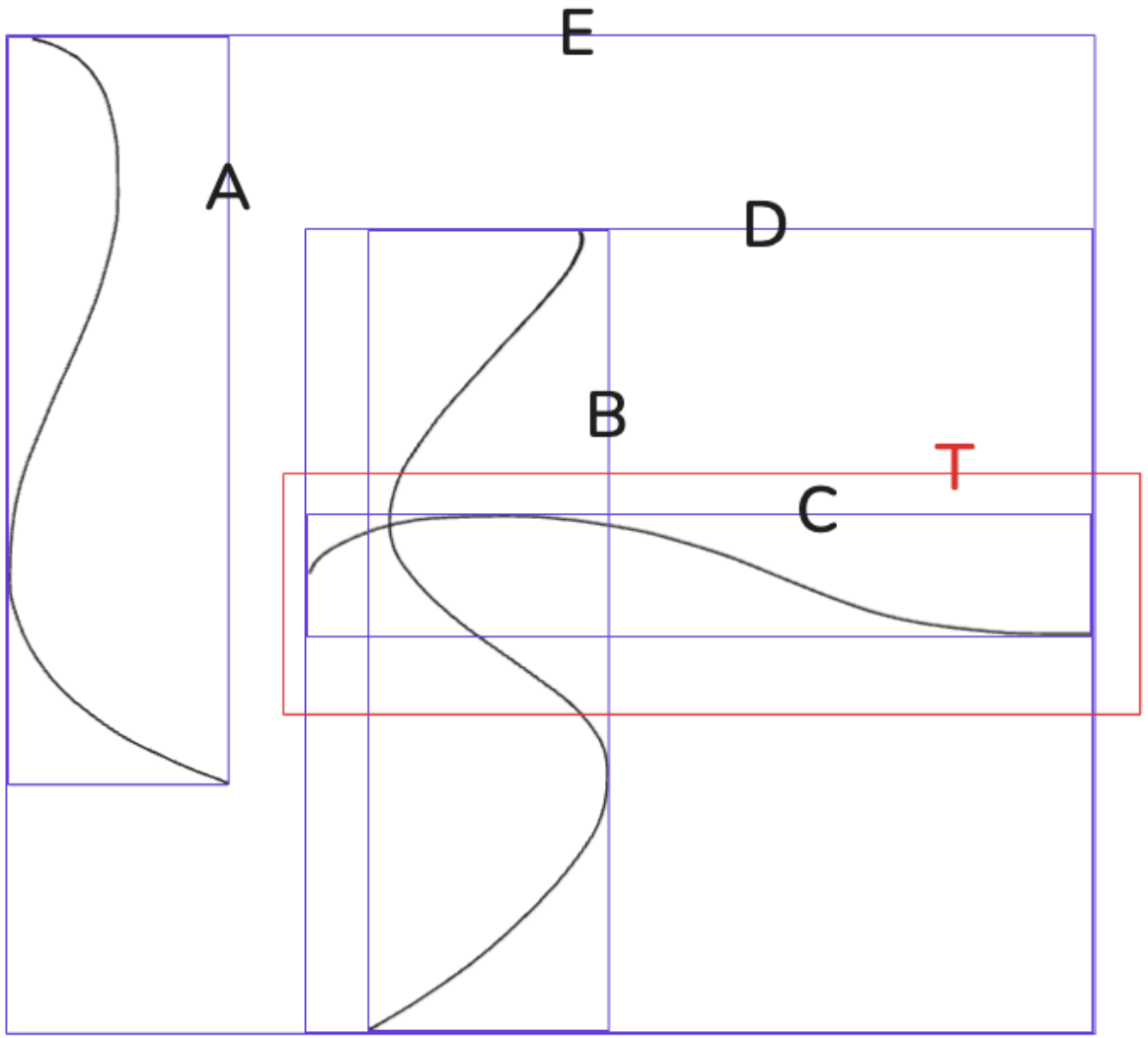


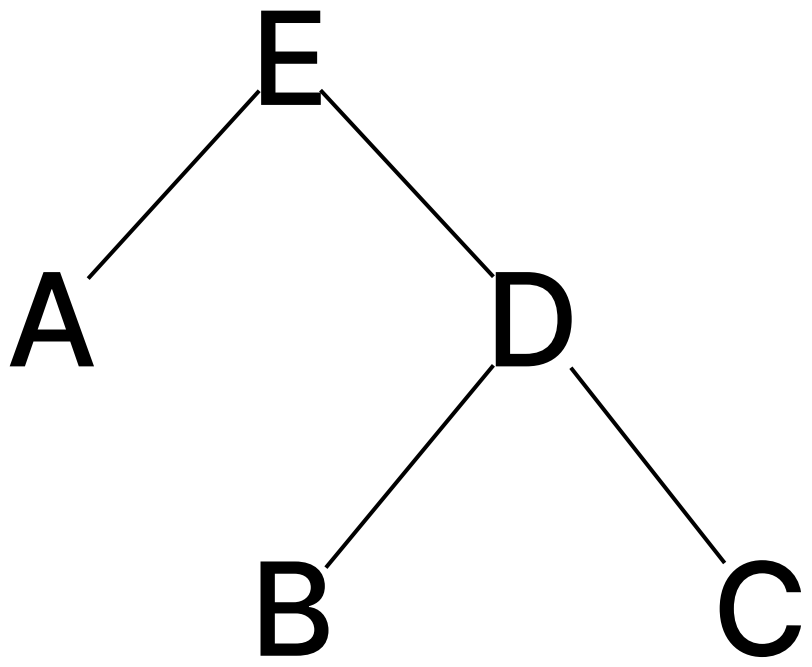
MBR & R-Tree



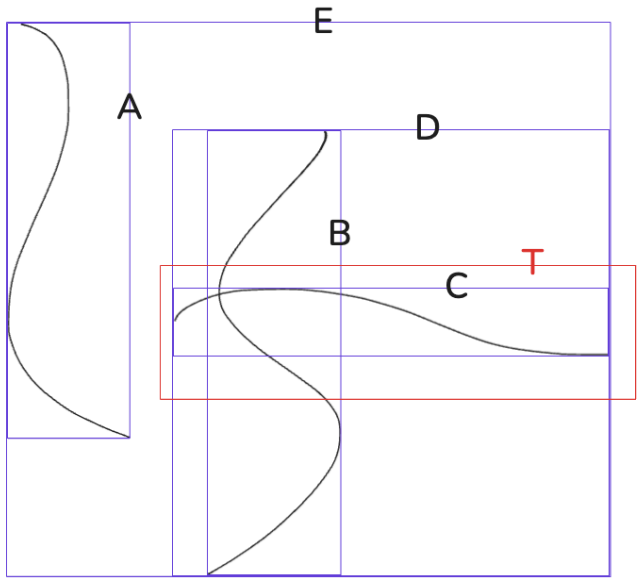


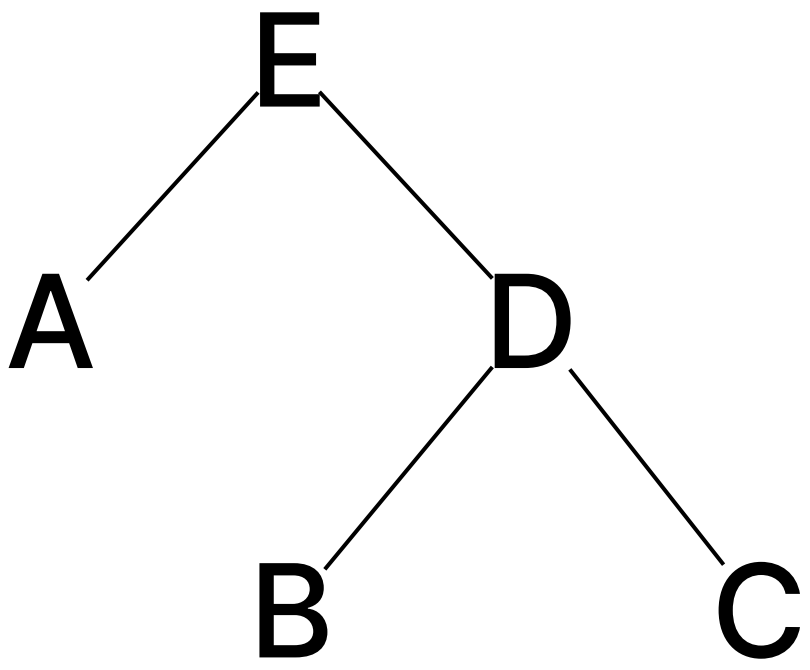
MBR & R-Tree





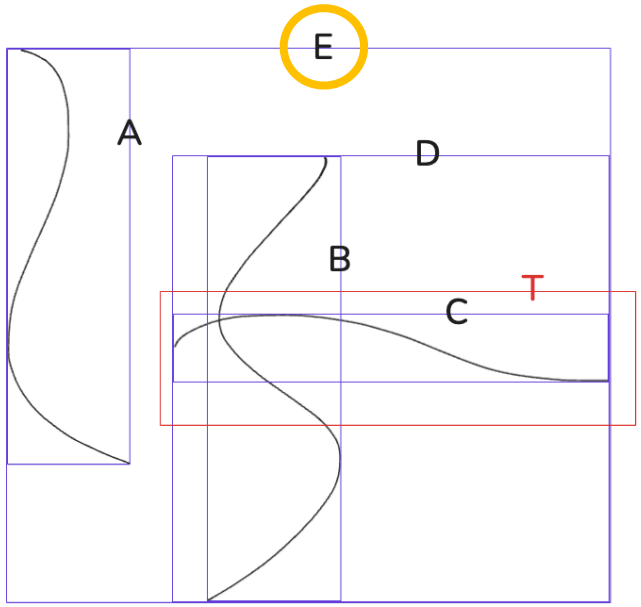
MBR & R-Tree

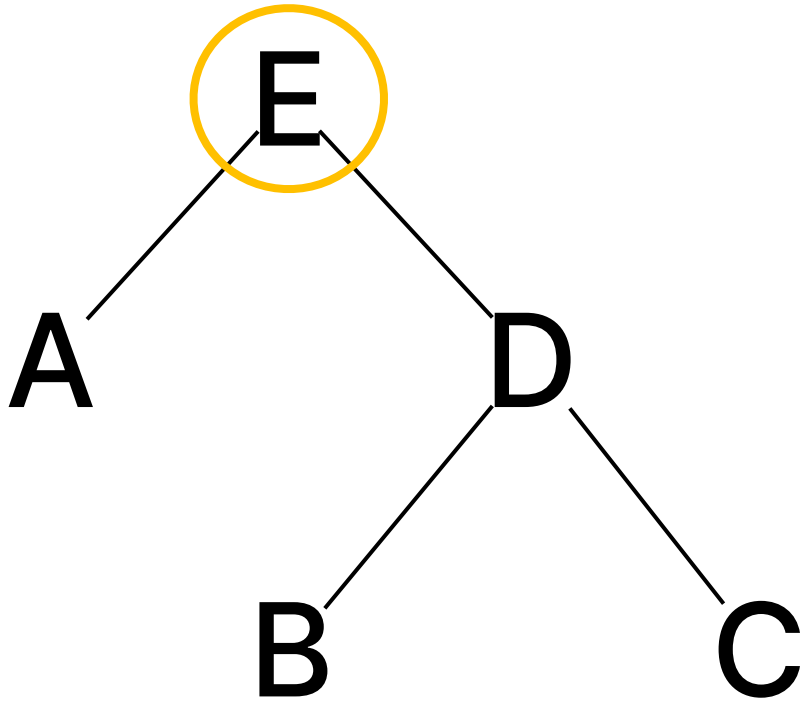




규칙
1. 중간 노드에서는 겹침 여부를 본다.
(겹침은 포함도 아우르는 말)
2. 리프 노드에서는 포함 여부를 본다.

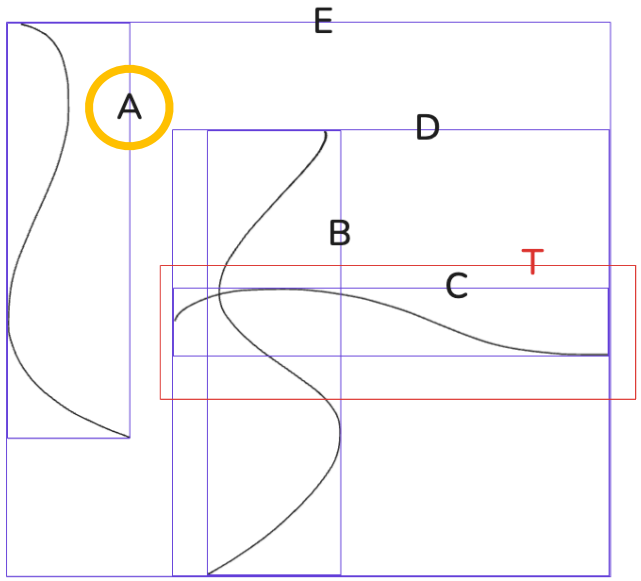
MBR & R-Tree

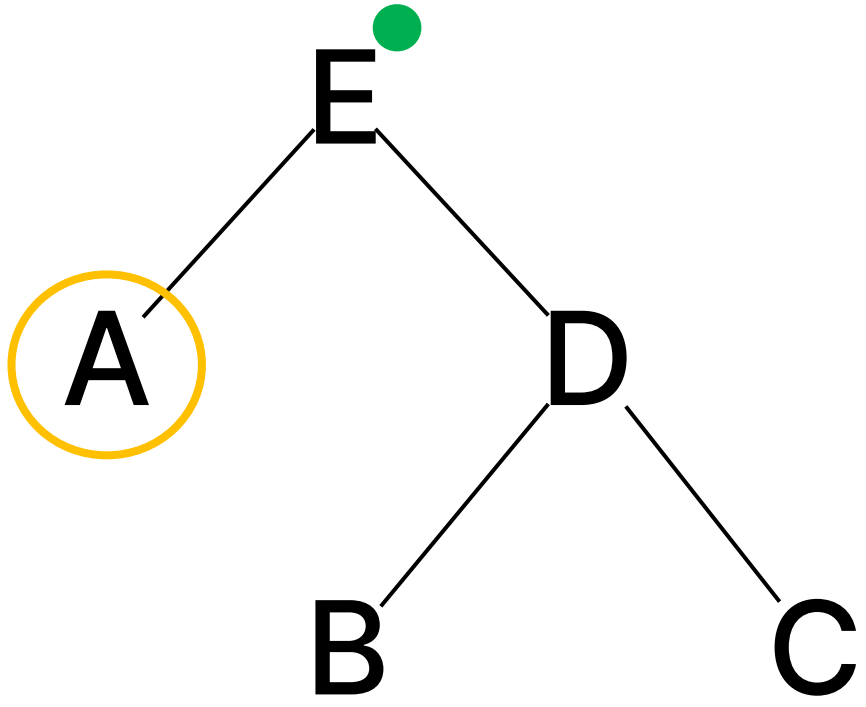




(1) T는 E와 겹친다.

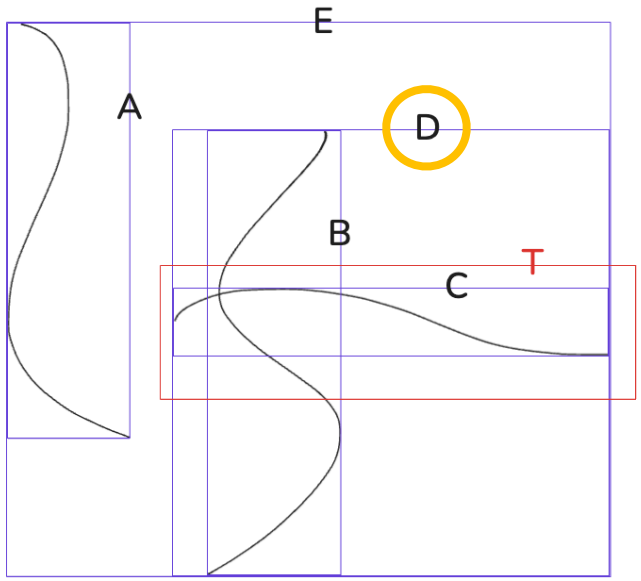
MBR & R-Tree

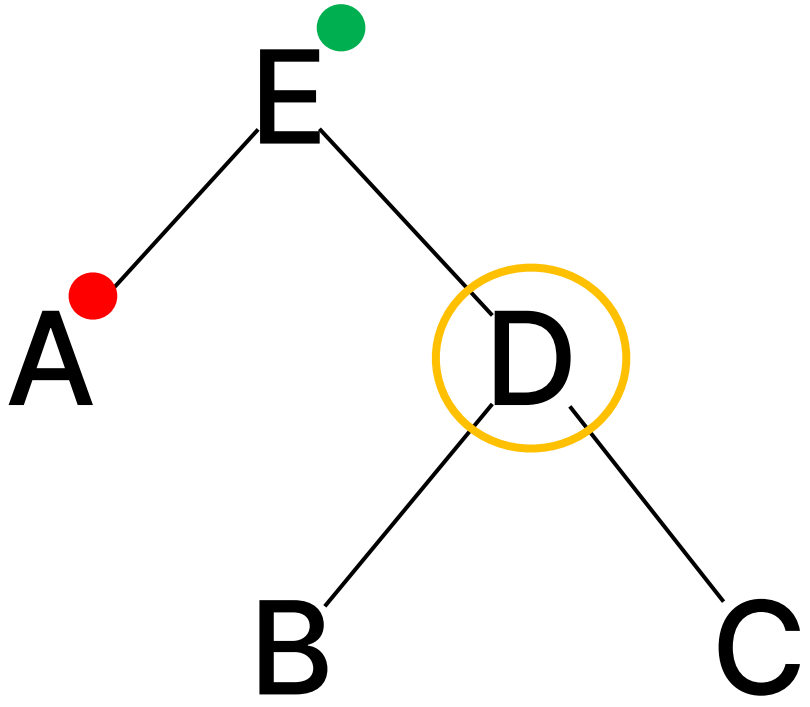




(1) T는 E와 겹친다.
(2) T는 A와 포함되지 않는다.

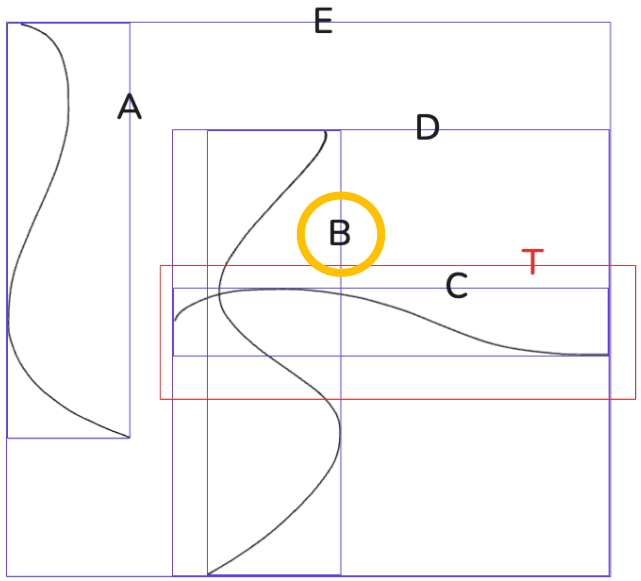
MBR & R-Tree

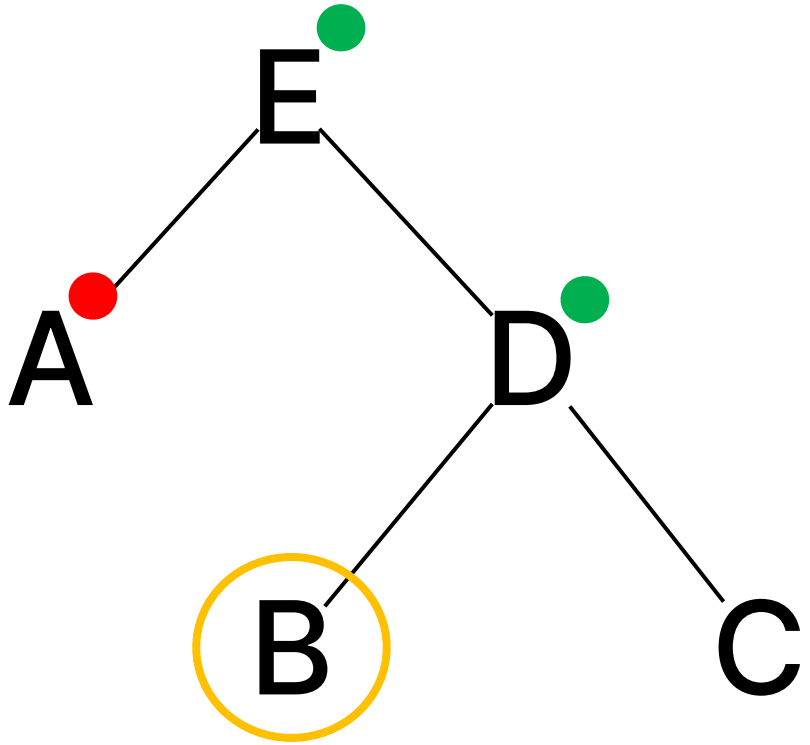




(1) T는 E와 겹친다.
(2) T는 A와 포함되지 않는다.
(3) T는 D와 겹친다.

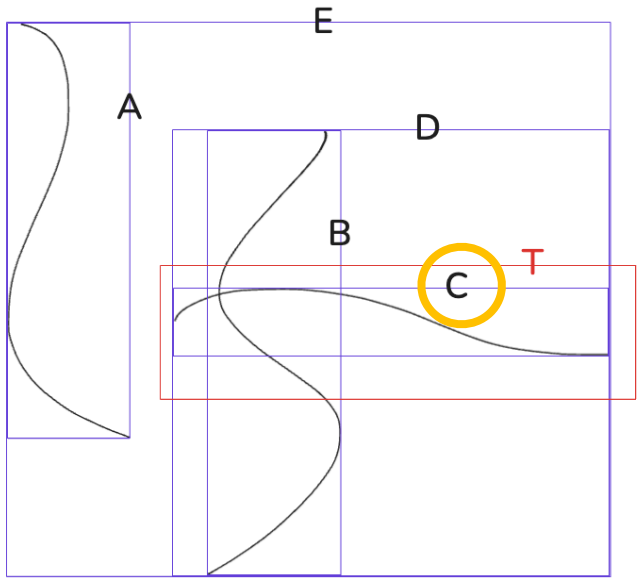
MBR & R-Tree

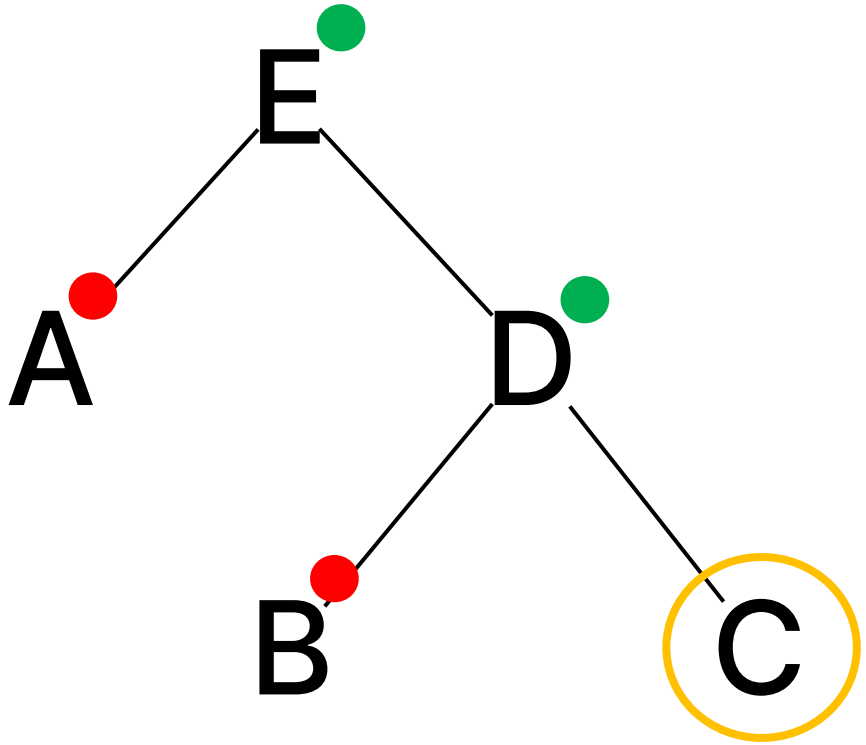




(1) T는 E와 겹친다.
(2) T는 A와 포함되지 않는다.
(3) T는 D와 겹친다.
(4) T는 B와 포함되지 않는다.

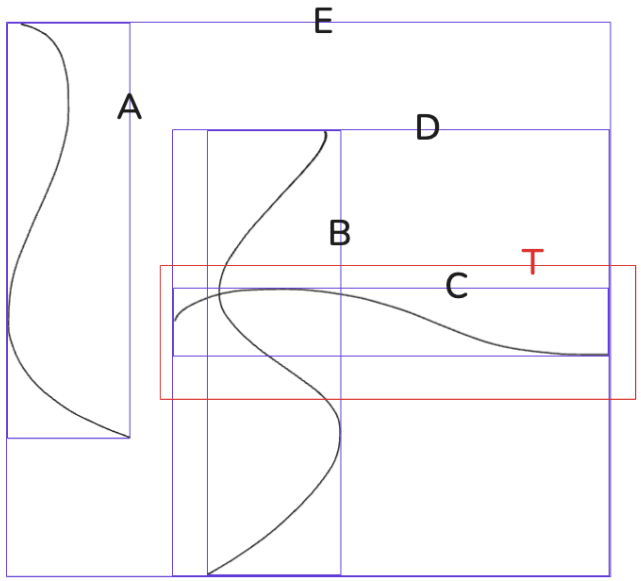
MBR & R-Tree

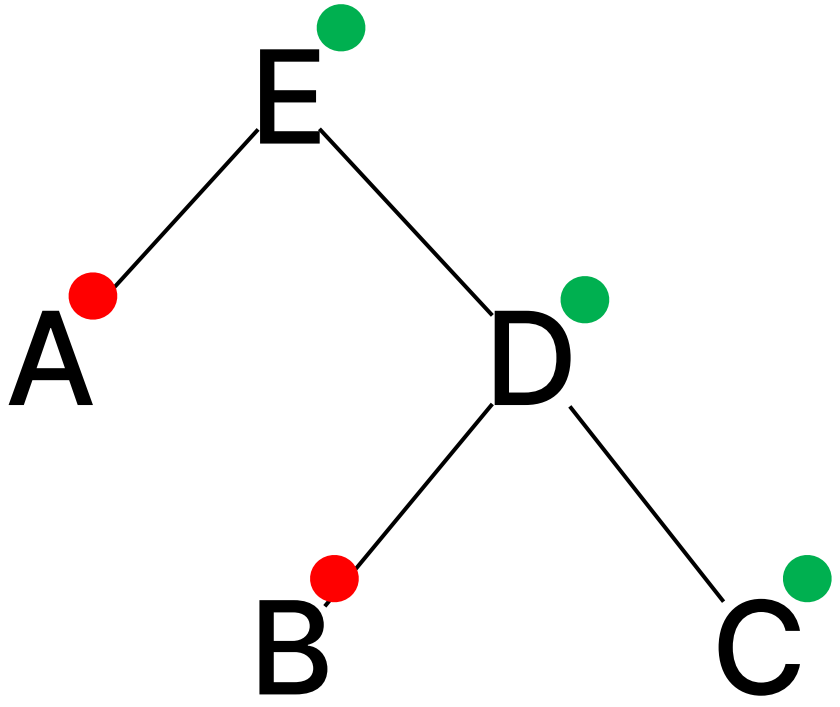




(1) T는 E와 겹친다.
(2) T는 A와 포함되지 않는다.
(3) T는 D와 겹친다.
(4) T는 B와 포함되지 않는다.
(5) T는 C와 포함된다.

MBR & R-Tree

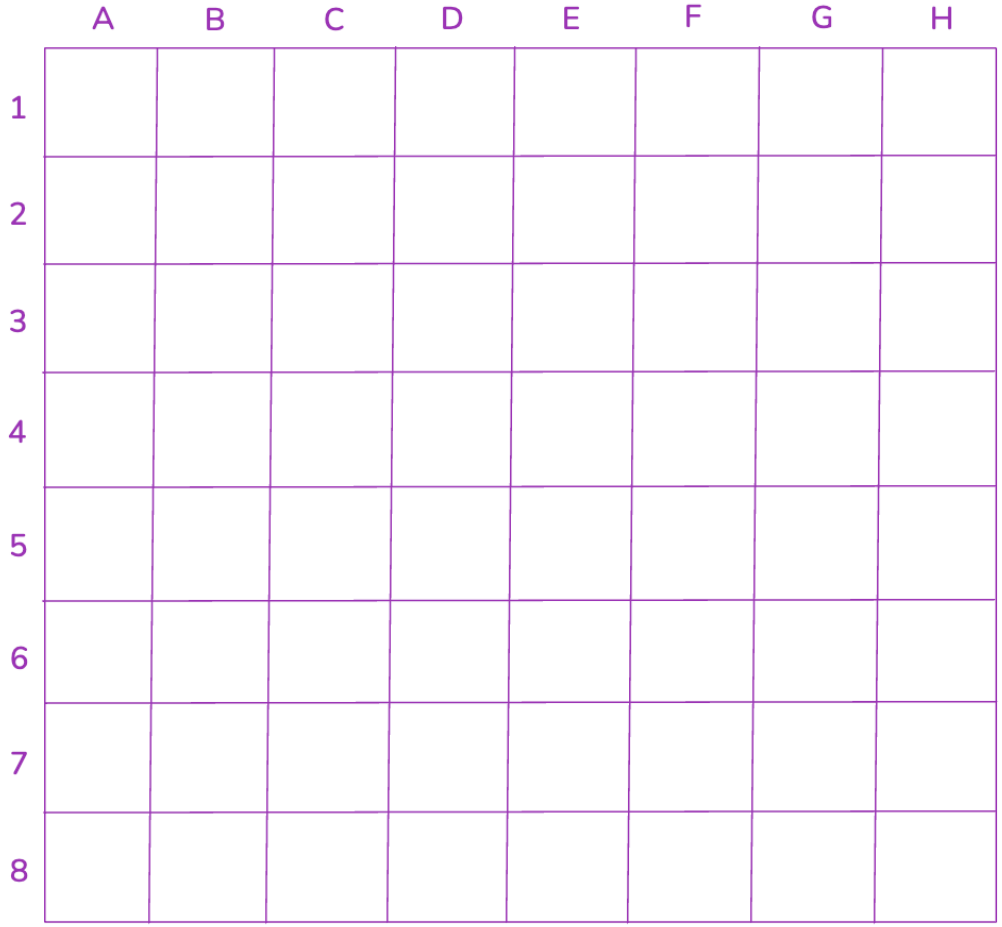


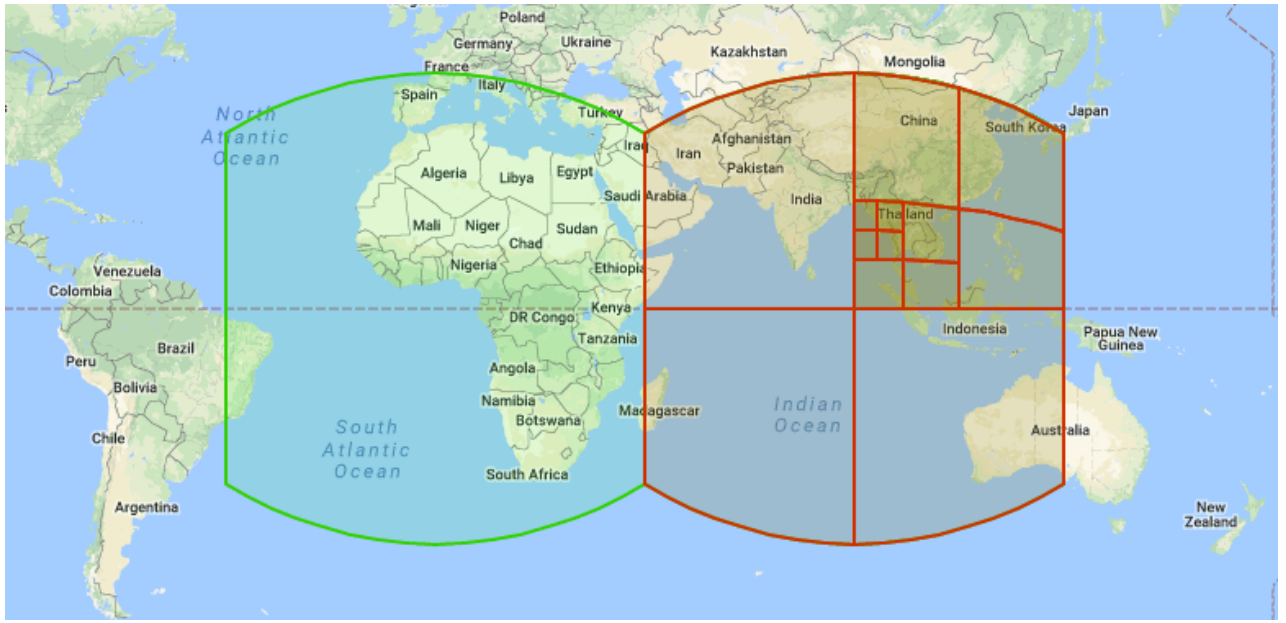


(1) T는 E와 겹친다.
(2) T는 A와 포함되지 않는다.
(3) T는 D와 겹친다.
(4) T는 B와 포함되지 않는다.
(5) T는 C와 포함된다.

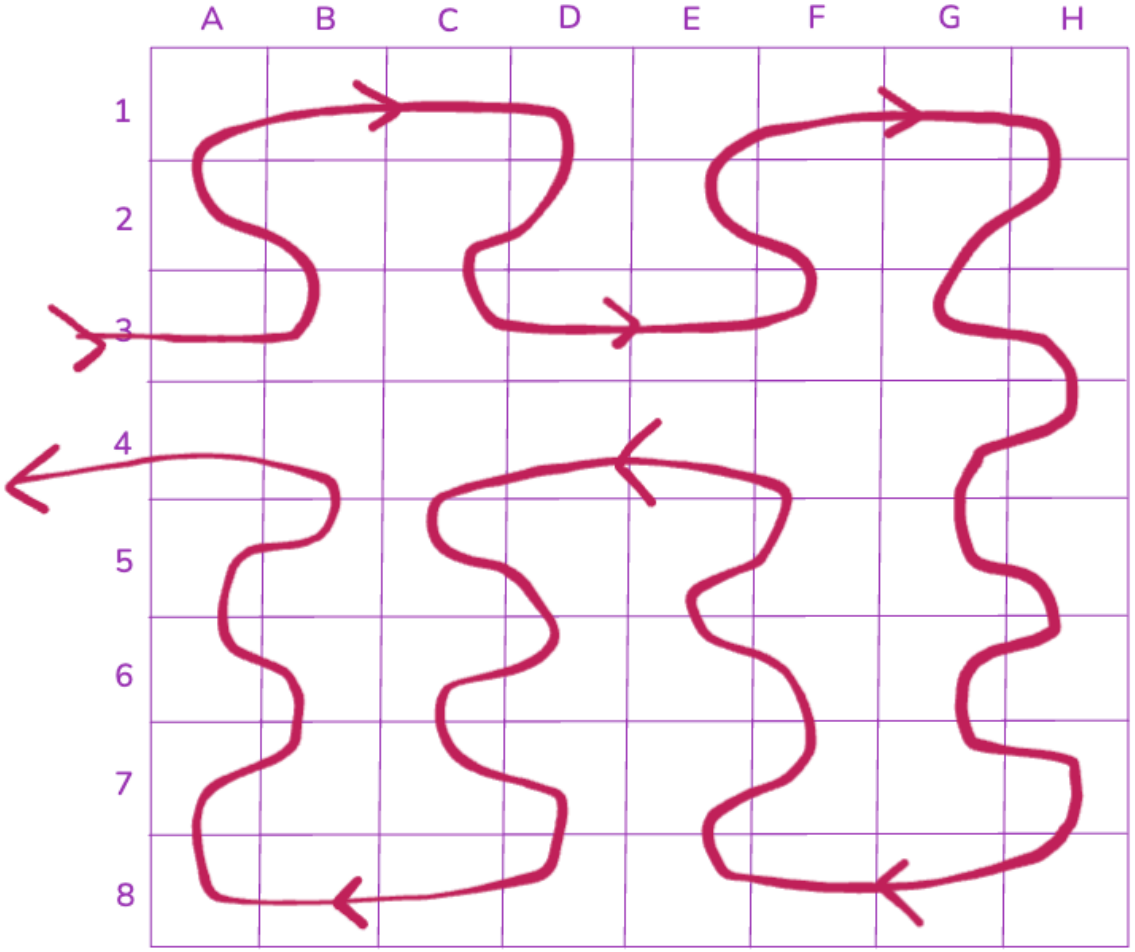
MongoDB의 공간인덱스 구현

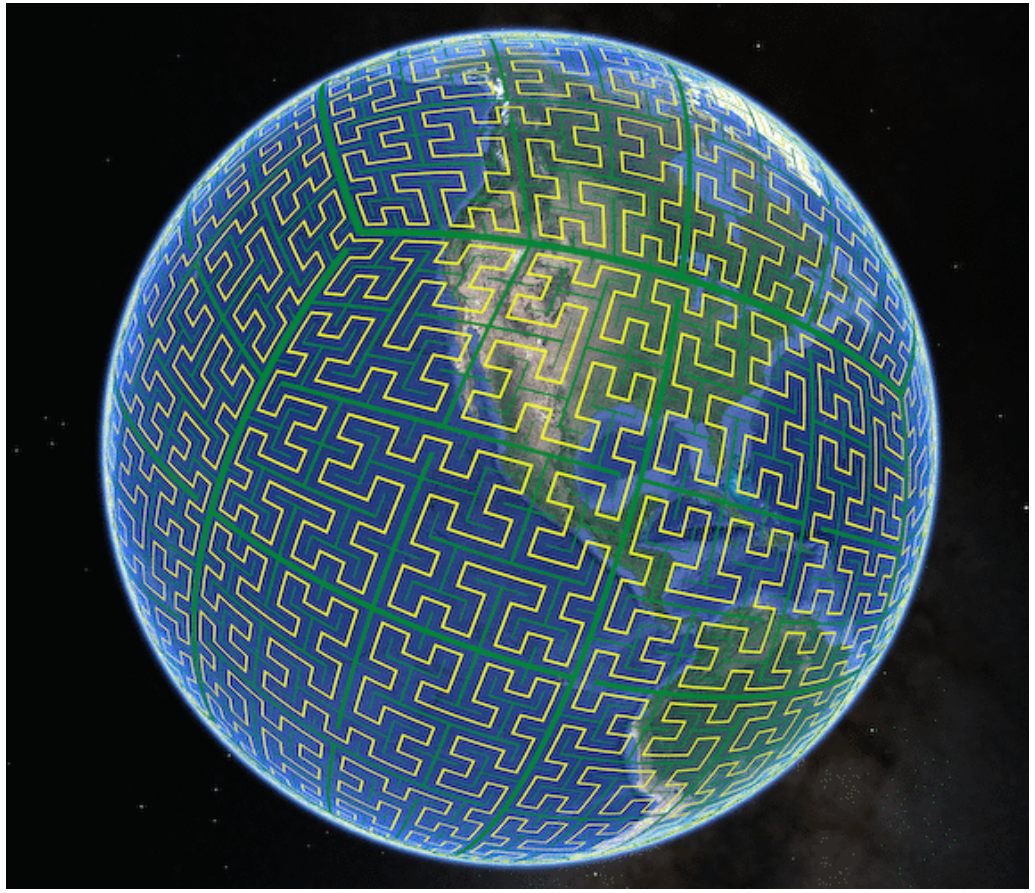
: 지구를 **조각**들로 나누고, 공간 채움 곡선으로 직렬화하여 B+Tree를 구축한다.



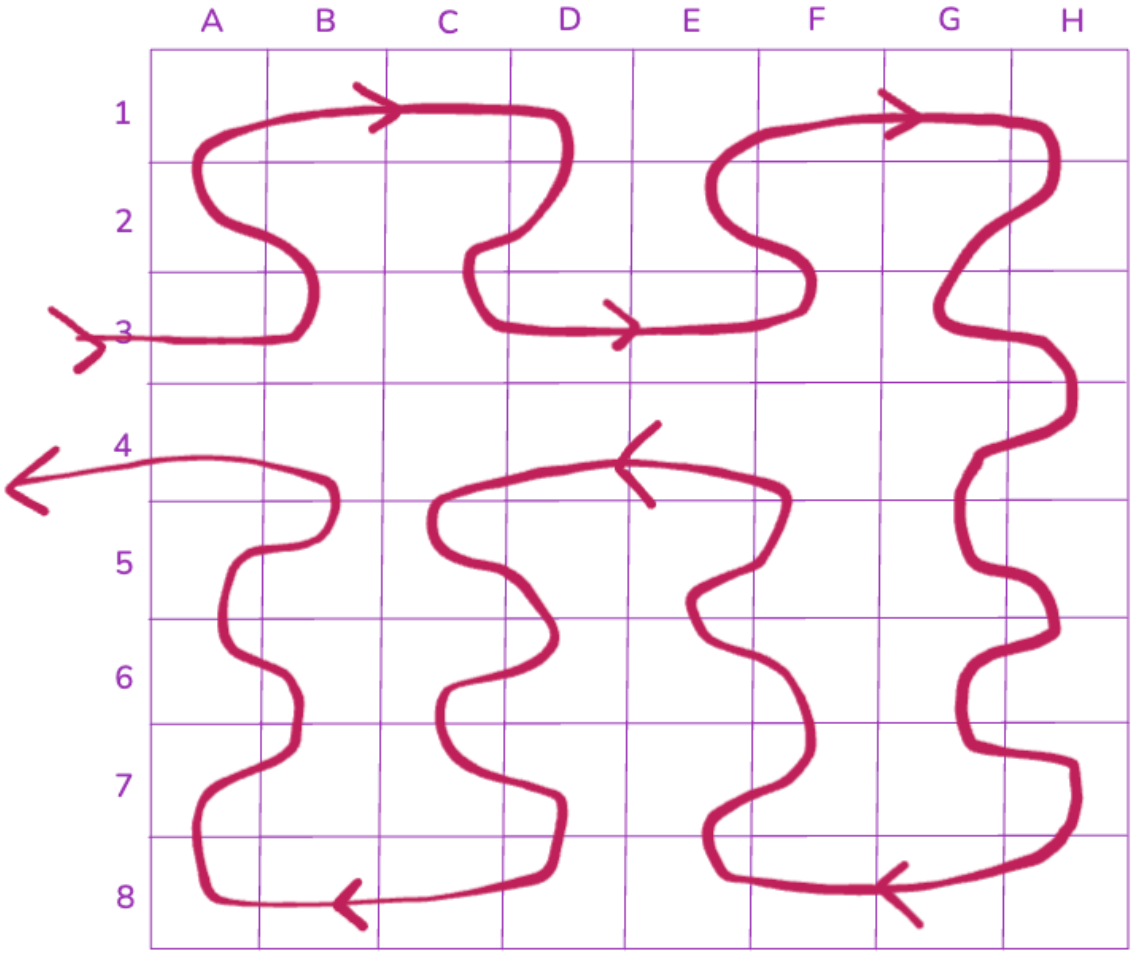


S2 Cell & 공간 채움 곡선 & B+Tree

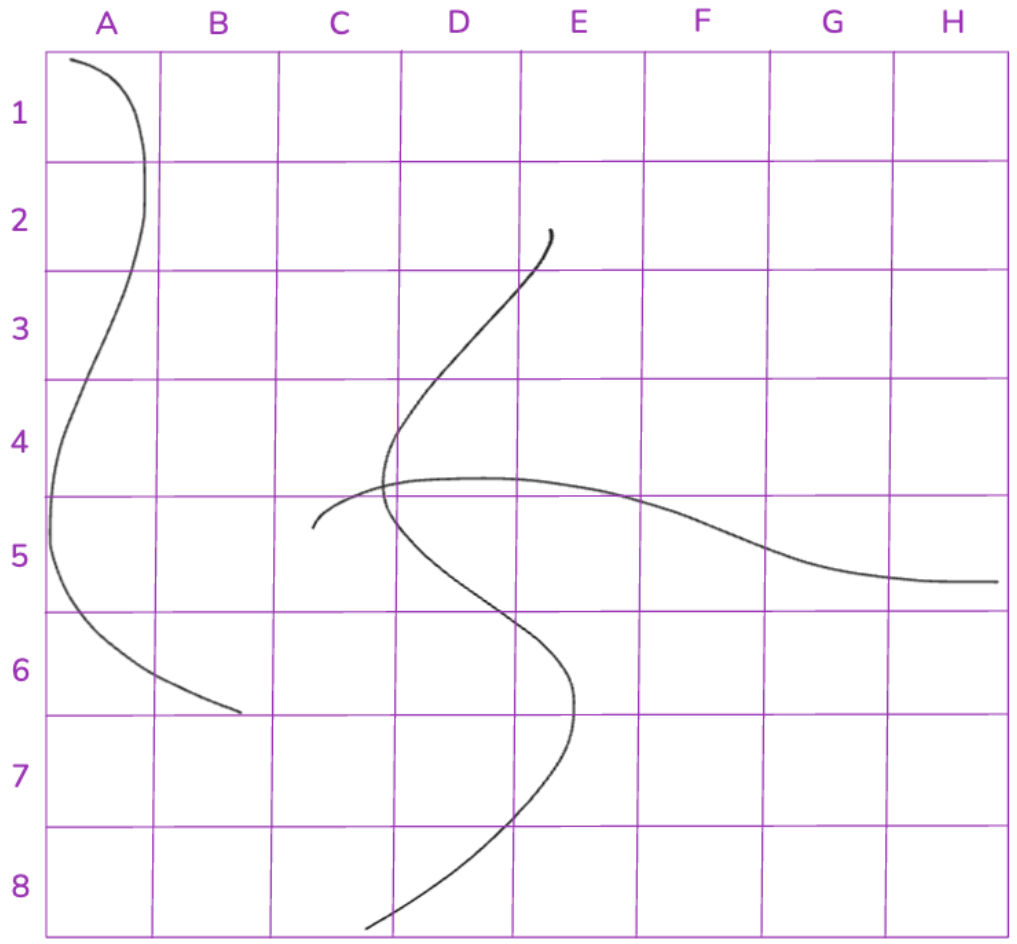


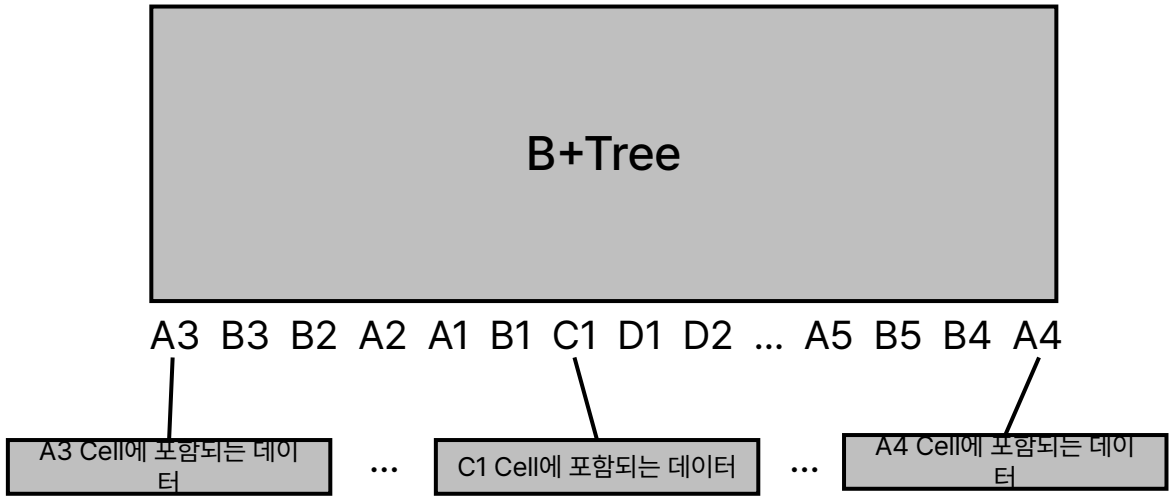


S2 Cell & 공간 채움 곡선 & B+Tree

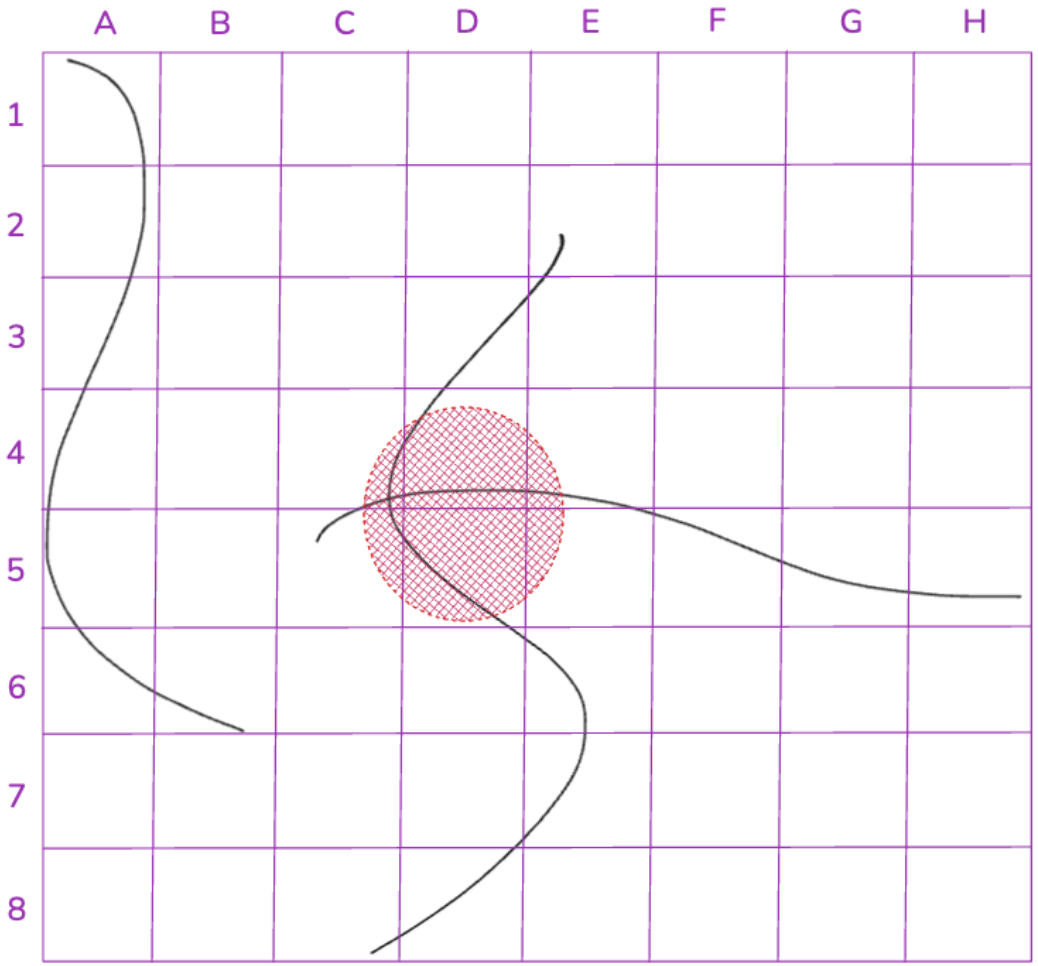


S2 Cell & 공간 채움 곡선 & B+Tree





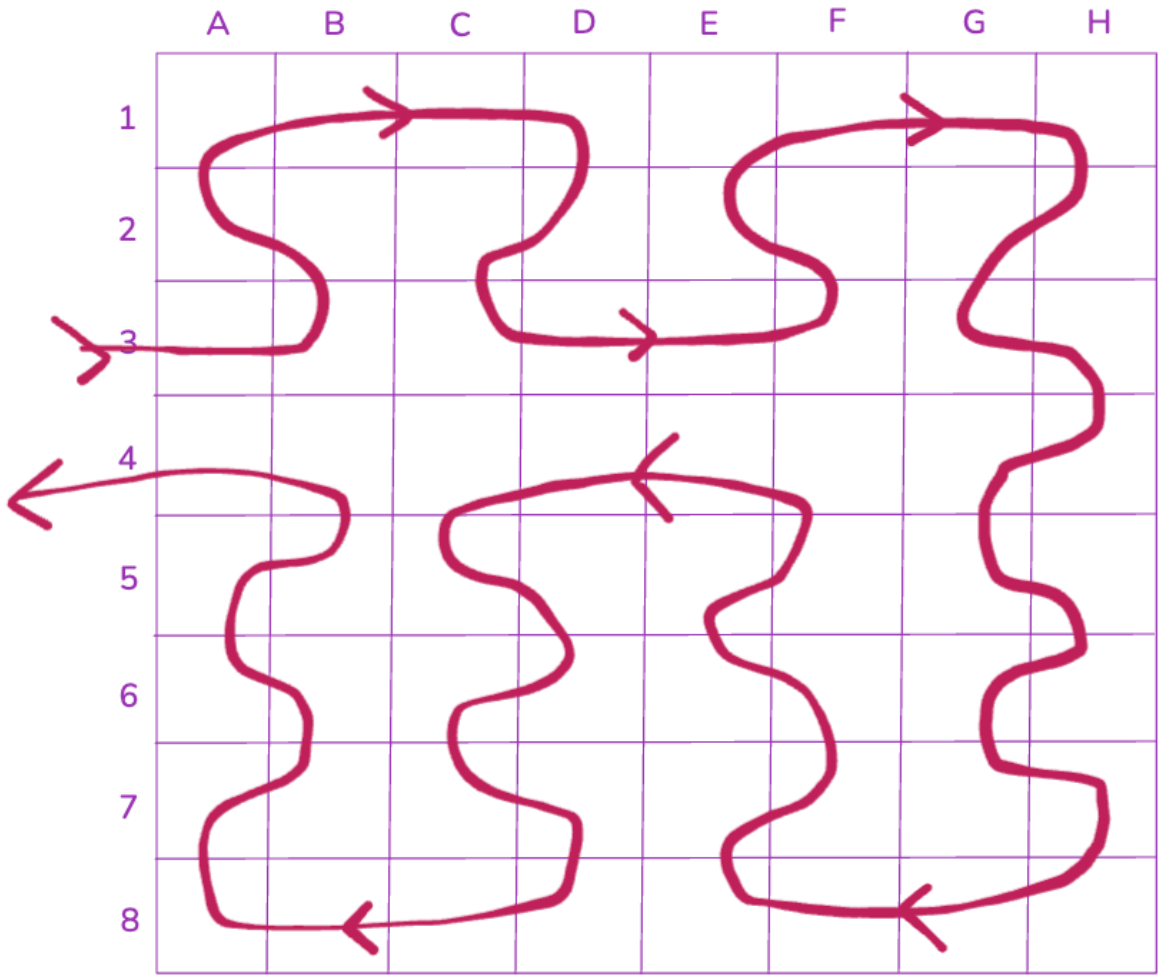
S2 Cell & 공간 채움 곡선 & B+Tree



원 안을 지나가는 공간데이터를 찾고 싶다면,

(1) 원이 포함되는 cell들을 찾는다.
C4, C5, D4, D5, E4, E5

S2 Cell & 공간 채움 곡선 & B+Tree

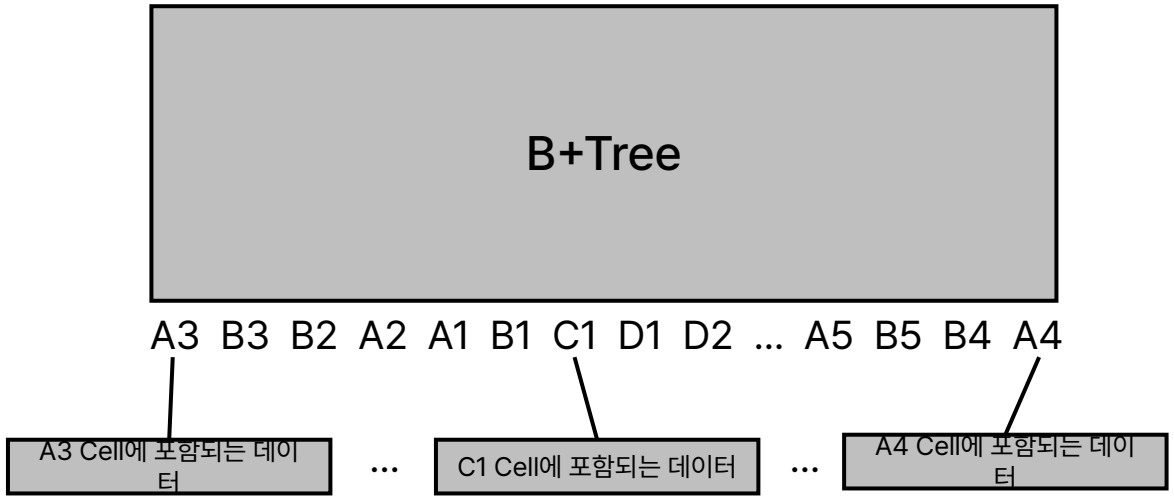


원 안을 지나가는 공간데이터를 찾고 싶다면,

(1) 원이 포함되는 cell들을 찾는다.
C4, C5, D4, D5, E4, E5

(2) 이 cell들을 공간 채움 곡선에 따라 연속된 것끼리 그룹화한다.
E5
E4 → D4 → C4 → C5 → D5

S2 Cell & 공간 채움 곡선 & B+Tree



원 안을 지나가는 공간데이터를 찾고 싶다면,

(1) 원이 포함되는 cell들을 찾는다.
C4, C5, D4, D5, E4, E5

(2) 이 cell들을 공간 채움 곡선에 따라 연속된 것끼리 그룹화한다.
E5
E4 → D4 → C4 → C5 → D5

(3) 범위 검색한다.
E5:E5
E4:D5

MongoDB vs MySQL

그래서 왜 MongoDB가 더 빨랐던 걸까?

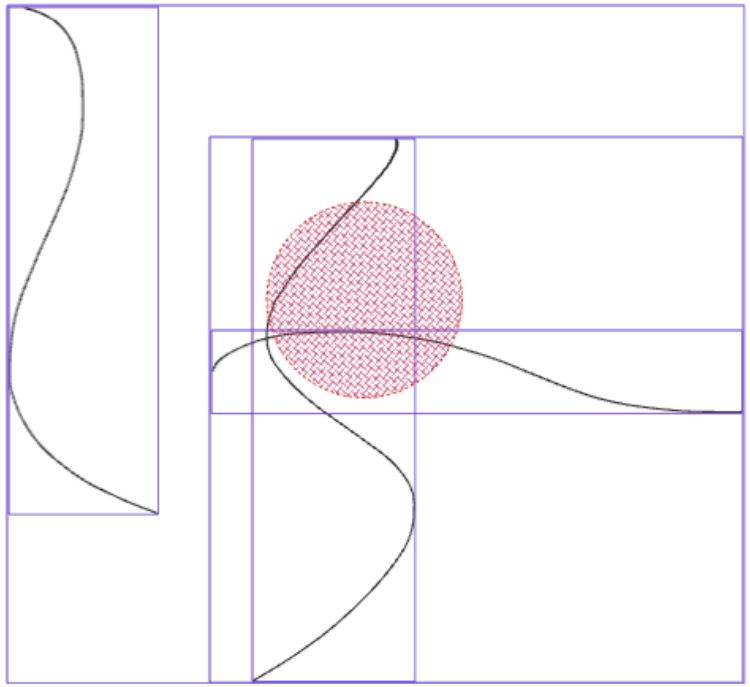
기본 흐름은 매우 유사하다.

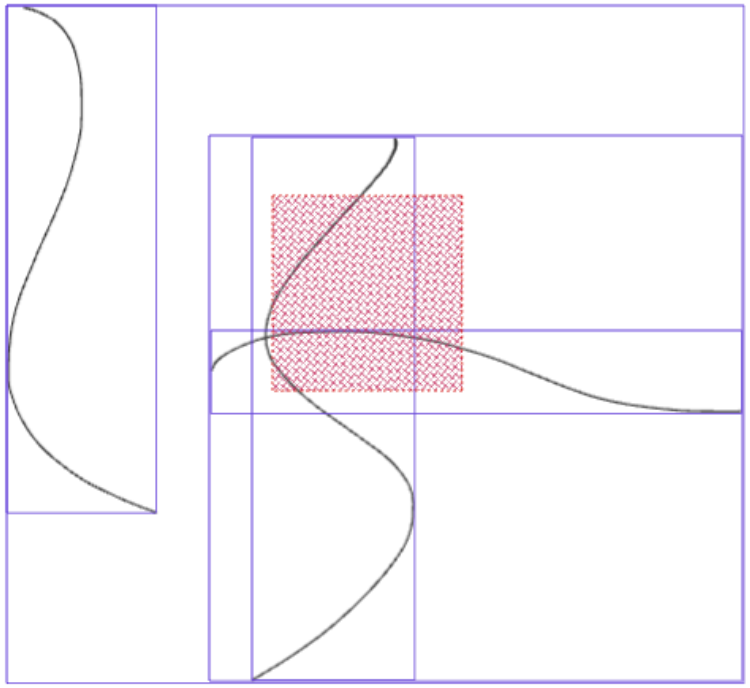
포함될 수 있는 범위 한정
→ 범위에 대한 공간인덱스 적용
→ 실제 거리 계산을 통한 최종 필터링

런세권에서 가장 중요한
‘기준점 기준 1KM 이내의 모든 코스 조회’ 쿼리를 통해 알아보자.

MySQL 정리

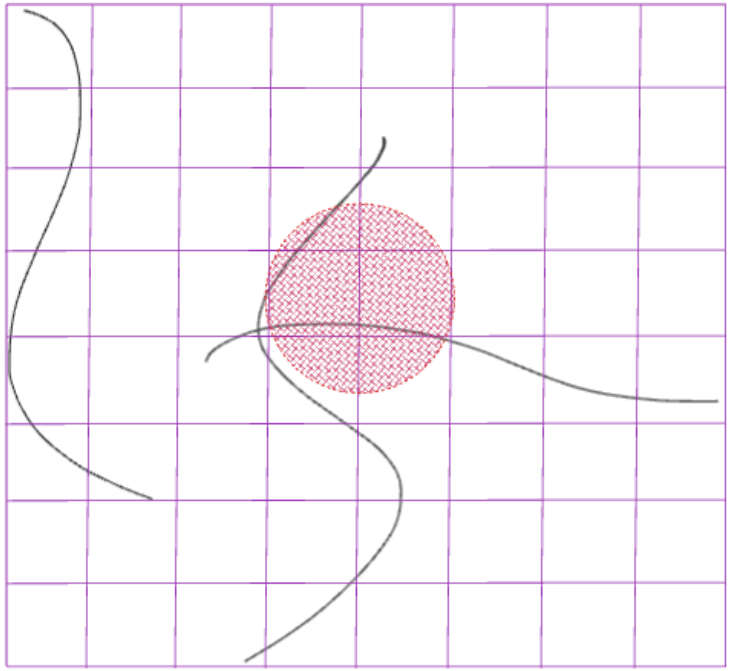
사각형에 대한 인덱싱만 지원된다.
1. 1KM 너비의 **정사각형**을 만든다.
2. 공간인덱스를 통해 해당 정사각형 안의 데이터를 조회한다.
3. 실제 거리를 측정하며 필터링한다.

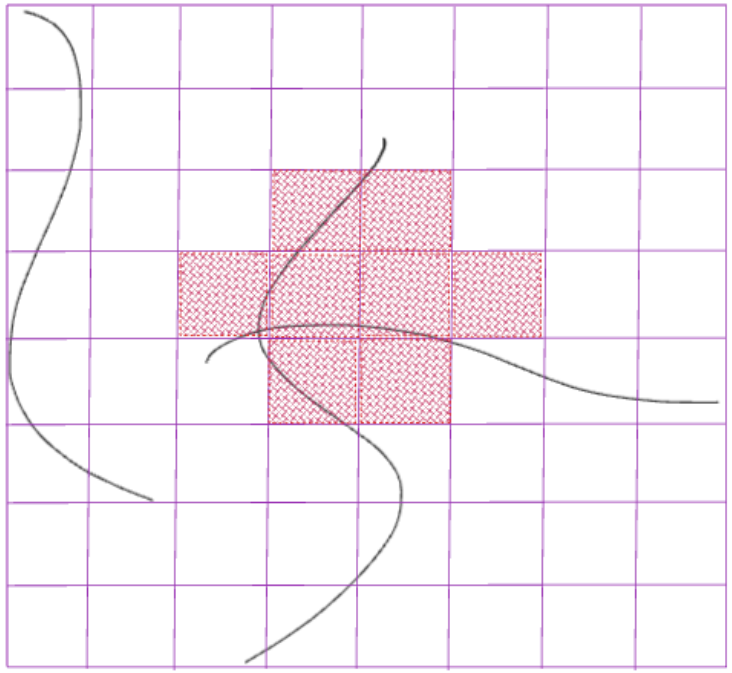




MongoDB 정리

cell에 대하여 인덱싱이 지원된다.
1. 1KM 너비의 **원에 포함되는 cell**들을 계산한다.
2. 공간인덱스를 통해 해당 cell들에 포함되는 데이터를 조회한다.
3. 실제 거리를 측정하며 필터링한다.





MongoDB vs MySQL

MySQL
1. 1KM 너비의 정사각형을 만든다.
2. 공간인덱스를 통해 해당 정사각형 안의 데이터를 조회한다.
3. 실제 거리를 측정하며 필터링한다.

MongoDB
1. 1KM 너비의 원에 포함되는 cell들을 계산한다.
2. 공간인덱스를 통해 해당 cell들에 포함되는 데이터를 조회한다.
3. 실제 거리를 측정하며 필터링한다.

→

MySQL은 단순 정사각형으로 필터링하기 때문에 **전처리가 빠르다.**

MySQL은 정사각형 안의 코스가 모두 조회되기 때문에 **후처리가 많다.**

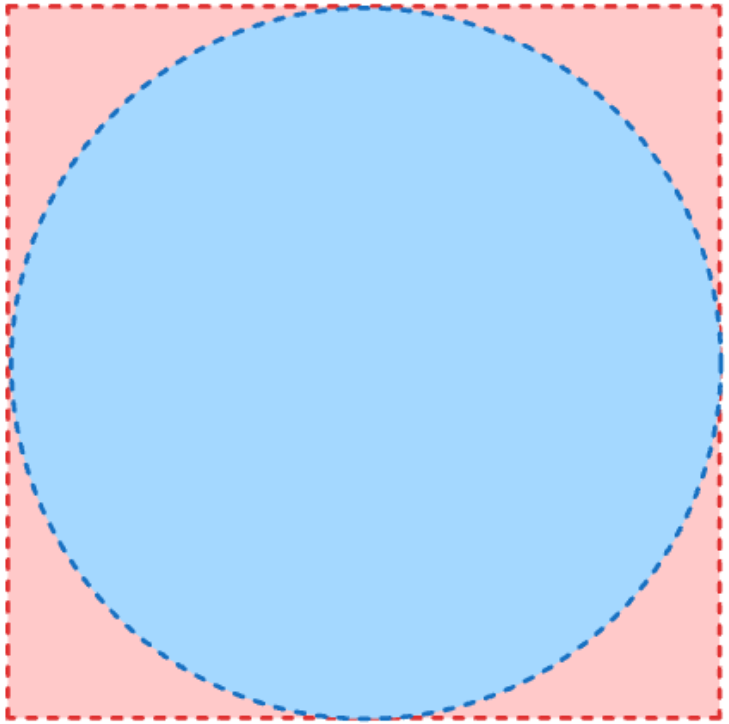
MongoDB는 S2Cell 계산 때문에 **전처리가 느리다.**

MongoDB는 원형에 근사하여 필터링되기 때문에 **후처리가 적다.** (한개의 cell은 1cm^2 정도이다.)

MongoDB vs MySQL

따라서,
데이터 양이 크고 많아서, 후처리 양의 차이가 많을수록 MongoDB가 우세하고
데이터 양이 작고 적어서, 전체 쿼리 시간 중 전처리가 차지하는 비중이 커질수록 MySQL이 우세해진다.

런세권의 데이터가 무겁고 양이 많은 편이었기 때문에, MongoDB가 우세할 수 있었다.



정리

- 런세권 팀은 공간데이터를 저장하기 위하여
  테이블 분리 → JSON → MULTILINESTRING→ MongoDB
  등의 방법들을 시도했다.
- MySQL은 '사각형의 포함 관계'를 기반으로 공간인덱싱을 한다.
- MongoDB는 '작은 cell들의 직렬화'를 기반으로 공간인덱싱을 한다.
- 데이터가 많을수록 MongoDB가 우세해진다.