02장 데이터 모델과 질의 언어

관계형 데이터베이스

비관계형 데이터베이스 NoSQL

NoSQL 중 문서 데이터베이스(Document Database)

: 데이터가 문서 자체에 포함되어 있고 문서끼리 관계가 거의 없는 사용 사례를 대상

NoSQL 중 그래프 데이터베이스(Graph Database)

: 모든 것이 잠재적으로 관련 있다는 사용 사례를 대상

스키마(Schema)

쓰기 스키마(Schema-on-Write) vs 읽기 스키마(Schema-on-Read)

쓰기 스키마(Schmea-on-Write)

: 관계형 데이터베이스의 전통적인 접근 방식으로 명시적인 스키마를 의미

읽기 스키마(Schema-on-Read)

: 암묵적인 스키마를 의미하며 NoSQL의 스키마리스(Schemaless) 는 이를 의미

명시적 스키마의 단점

: 스키마 변경에 따른 데이터 마이그레이션(Migration) 과정에서 성능 저하 발생

데이터 질의어

명령형

: 프로그래밍 언어와 유사한 방식으로 순서가 존재

선언형

: 관계 대수 구조를 사용하여 결과에 집중

조건 분기로 살펴보는 선언형 사고 와 명령형 사고

`Items` 테이블

```
+--------------+----------+------+-----+----------+-------+
| Field        | Type     | Null | Key | Default  | Extra |
+--------------+----------+------+-----+----------+-------+
| item_id      | int      | NO   | PRI | NULL     |       |
| year         | int      | NO   | PRI | NULL     |       |
| item_name    | char(32) | NO   |     | NULL     |       |
| price_tax_ex | int      | NO   |     | NULL     |       |
| price_tax_in | int      | NO   |     | NULL     |       |
+--------------+----------+------+-----+----------+-------+
```

`year` 필드의 값이 `2021` 이하인 경우 `price_tax_ex` 필드를, `2022` 이상인 경우 `price_tax_in` 필드를 사용해야 하는 상황 가정

명령형 사고

: 디스크 접근이 두 번 발생

```
SELECT
    item_id,
    item_name,
    year,
    price_tax_ex AS price_tax
FROM Items
WHERE year < 2022
UNION ALL
SELECT
    item_id,
    item_name,
    year,
    price_tax_in AS price_tax
FROM Items
WHERE year >= 2022;
```

선언형 사고

: 디스크 접근이 한 번 발생

```
SELECT
    item_id,
    item_name,
    year,
    CASE
        WHEN year < 2022 THEN price_tax_ex
        ELSE price_tax_in
    END AS price_tax
FROM Items;
```

맵-리듀스(Map-Reduce)

: 맵(Map)과 리듀스(Reduce) 단계를 반복해서 결과를 도출

맵(Map)

: 분할된 데이터를 처리하는 단계

리듀스(Reduce)

: 맵(Map)을 통해 처리된 결과를 모아 집계하는 단계

맵-리듀스 구조의 단점

: 여러 번 반복해서 결과를 얻어내기 때문에 각 사이클 별 대기 시간이 발생

저장소 지역성(Storage Locality)

: 관계형 데이터베이스에서 다중 결합에 의해 많은 디스크 탐색이 필요할 경우 성능 저하 발생

추가로 이야기해보면 좋은 주제

: 관계형 데이터베이스에서의 UUID와 인덱스

UUID를 인덱스로 갖는 테이블

: 연속되지 않은 페이지 번호로 인한 버퍼 풀(Buffer Pool) 비효율 발생

추가로 이야기해보면 좋은 주제

: SQL에서의 반복문

SQL에서의 반복문

: 관계 조작은 **관계 전체를 모두 조작의 대상으로** 삼는다. 이러한 것의 목적은 **반복을 제외**하는 것이다.

추가로 이야기해보면 좋은 주제

: 관계형 데이터베이스에서의 외부 코드 실행

관계형 데이터베이스에서의 외부 코드 실행

: AWS Aurora 클러스터에서 AWS Lambda 함수 호출

AWS Aurora 클러스터에서 AWS Lambda 함수 호출

```
SELECT
    lambda_async(
        'arn:aws:lambda:...'',
        '{"message": "Hello, world!"}'
    )
;
```

추가로 이야기해보면 좋은 주제

: 방향성 비순환 그래프 (Directed Acyclic Graph, DAG)

DAG의 장점

: 지연 평가(Lazy Evaluation) 를 통해 실행 결과를 요구할 때 데이터 처리가 발생

추가로 이야기해보면 좋은 주제

: 외래 키(Foreign Key, FK) 구현 계층

물리적 외래 키와 논리적 외래 키

외래 키를 사용하지 않는 GitHub의 사례

물리적 외래 키

: 데이터베이스에서 `FOREIGN KEY` 제약 조건을 활용하여 물리적인 제약 조건을 지정

```
CREATE TABLE student (
    id CHAR(8) NOT NULL,
    major_id CHAR(8) NOT NULL,

    PRIMARY KEY(id),

    FOREIGN KEY (major_id)
        REFERENCES major(id)
        ON UPDATE CASCADE ON DELETE RESTRICT
);
```

논리적 외래 키

: 애플리케이션에서 조건문 등을 활용하여 논리적인 제약 조건을 지정

```
if input_major_id not in major_ids:
    raise ValueError("Input Major ID must be in major table.")
```

GitHub에서 외래 키를 사용하지 않는 이유

1. 샤딩(Sharding)의 용이성
2. 삽입(`INSERT`) 및 삭제(`DELETE`) 쿼리에 대한 성능 향상
3. 온라인 스키마 마이그레이션과의 호환성

샤딩(Sharding)의 용이성

: 외래 키로 연결된 여러 테이블 분산에 어려움 발생

삽입 및 삭제 쿼리에 대한 성능 향상

: 삽입 및 삭제가 수행될 경우 외래 키에 대한 인덱스 탐색 및 트리 정렬이 발생

온라인 스키마 마이그레이션과의 호환성

: 물리적인 제약 조건 때문에 서로 연결된 테이블 구조 변경에 어려움이 발생

추가로 이야기해보면 좋은 주제

: AWS S3 스토리지 저장소를 데이터베이스로 활용