#cs #study

목차

1) 효율적 쿼리

서브 쿼리와 조인

- 서브 쿼리(subquery) : 다른 SQL 문이 포함된 SQL

```
SELECT
    users.username,
    (SELECT COUNT(*) FROM posts WHERE posts.user_id = users.user_id)
AS post_count
FROM users;
```

  - () 속에 들어간 내용이 서브쿼리이며 해당 서브쿼리의 결과를 post_count에 나열하는 것
- 조인(join) : 2개의 테이블을 하나로 합치는 것
  - inner join : 테이블 A와 테이블 B의 교집합만 반환
  - left outer : 테이블 A의 모든 레코드를 반환하고 테이블 B 레코드는 테이블 A의 교집합인 레코드만 정상이고 나머지는 모두 NULL
  - right outer : left outer와 달리 테이블 B 기준
  - full outer : 테이블 A와 테이블 B의 합집합

```
SELECT coustomers.name, customers.age, customers.email, orders.id,
orders.product_id, orders.quantity, orders.amount
FROM customers O JOIN orders ON customers.id = orders.customer_id;
```

  - 위 쿼리문의 O에 들어가는 단어에 따라 어떤 결과가 나올 지 보자
    - INNER
      - customers 테이블의 id와 orders.customer_id가 같은 레코드만 합친다
    - LEFT OUTER
      - customers 테이블의 모든 항목을 선택한다
      - customers 테이블 기준으로 orders 테이블 레코드를 합친다
      - customers 테이블 기준으로 customers.id != orders.customer_id인 경우에는 해당 레코드의 orders 필드들은 NULL로 채운다
    - RIGHT OUTER

        - orders 테이블의 모든 항목을 선택한다
        - orders 테이블 기준으로 customers 테이블 레코드를 합친다
        - orders 테이블 기준으로 orders.customer_id != customers.id인 경우에는 해당 레코드의 customers 필드들은 NULL로 채운다
    - FULL OUTER
        - orders와 customers 테이블의 모든 항목을 선택한다
        - customers.id == orders.customer_id인 레코드는 모든 값을 기존 필드값 대로 넣는다
        - customers.id != orders.customer_id인 레코드는 부족한 필드값만 NULL로 채운다
  - FULL OUTER는 구현이 안 된 경우가 있으므로, 보통 UNION을 써서 LEFT OUTER와 RIGHT OUTER 결과물을 결합한다

뷰

- 뷰(view)는 SELECT 문의 결과로 만들어진 가상의 테이블이다

```
CREATE VIEW viewName AS SELECT문;
```

    - 위의 쿼리 구문에서 SELECT 문은 사실상, 서브쿼리처럼 동작한다. 하지만 실제 서브쿼리와 달리 괄호 안에 구문을 넣지 않는 걸 상기할 필요가 있다
- 뷰를 사용하는 이유
    - 프로그래밍 언어에서 왜 함수를 쓰는 지 생각하면 된다. 복잡한 쿼리문을 또 작성할 필요가 없고 자주 불러오기 때문에 쓰는 것.
- 사용 시 주의점
    - 조회는 쉽게 할 수 있지만, 삽입/수정/삭제 연산이 까다롭다

인덱스

- 검색 속도 향상을 목적으로 만드는 하나 이상의 테이블 필드에 대한 자료구조. 즉, 특정 레코드를 쉽게 찾을 수 있게 각각 색인으로 등록하는 것이다.
- 클러스터형 인덱스(clustered index) : 테이블 당 하나씩 만드는 인덱스. 기본 키가 이에 해당한다. 만약, 기본 키가 없으면 NOT NULL과 UNIQUE 제약이 있는 필드를 클러스터형 인덱스로 간주한다.
- 세컨더리 인덱스(secondary index) : 테이블 당 여러개가 존재할 수 있다. 그렇기에 클러스터형 인덱스보다 검색이 느리다.

```
-- 세컨더리 인덱스 생성
CREATE INDEX indexName ON tableName (field);

-- 인덱스 조회
SHOW INDEX FROM tableName;
```

```
-- 인덱스 삭제
DROP INDEX indexName FROM tableName;
```

- 인덱스 제작 시 고려할 점
  - 인덱스는 생성하는 데 인덱스를 위해 별도의 메모리 공간을 확보하고 생성 시간도 필요하다.
  - 조회는 빠를 수 있지만 삽입/삭제/수정 시에는 인덱스 처리하는 시간이 추가로 들어간다.

2) 데이터베이스 설계

ER 데이터그램

- ER diagram(ERD) : 데이터베이스에서 엔티티 관계를 표현하는 설계도

정규화(Normal Form, NF)

- 잠재적인 문제가 발생하지 않도록 테이블 필드를 구성하고 필요할 경우 테이블을 나누는 작업
- 제 1 정규형
  - 모든 속성이 원자 값을 가진다. 즉, 모든 필드가 더는 쪼개질 수 없는 값을 가진다.
- 제 2 정규형
  - 제 1 정규형을 만족하면서, 기본 키가 아닌 모든 필드들이 모든 기본 키에 완전히 종속
  - 종속성
    - 부분 함수 종속성(partial functional dependency) : 기본 키가 아닌 필드가 기본 키의 일부에 종속
    - 완전 함수 종속성(full functional dependency) : 기본 키 전체에 완전히 종속
  - 제 2 정규형은 완전 함수 종속성을 지닌 상태다
- 제 3 정규형
  - 제 2 정규형을 만족하면서, 기본 키가 아닌 모든 필드가 기본 키에 이행적 종속 상태가 없어야 한다
  - 이행적 종속 상태 : 필드 A, B, C가 있을 때, A가 B를 결정하고 B가 C를 결정하면 A가 C를 결정한다. 이 때, A와 C는 이행적 종속 상태다.
- 보이스/코드 정규형(Boyce-Codd Normal Form, BCNF)
  - 제 3 정규형을 만족하면서, 모든 결정자가 후보키다. 결정자는 특정 필드를 식별할 수 있는 필드를 말한다.
- 역정규화(Denormalization)
  - 정규화한 테이블을 다시 합치는 것
  - 정규화의 단점인 파편화로 연산이 복잡하고 많아지는 점을 극복하기 위해 행한다

3) NoSQL

NoSQL 특징

- 키-값 데이터베이스

    - 레코드를 키와 값의 쌍으로 저장한다
    - 종류 : Redis, Memcached 등
    - 특징
        - 레코드 구조가 단순하여 메모리에 저장해 빠른 접근성을 제공한다
        - 메모리에 저장되는 특성을 고려, 보조 데이터베이스로 활용하는 경우가 많음
- 문서지향 데이터베이스
    - 레코드를 문서(document) 단위로 저장하고 관리하는 데이터베이스
    - 문서는 비정형화 레코드 단위이며 JSON이나 XML 형식을 문서로 활용한다
    - 종류 : MongoDB
    - 특징
        - 하나의 레코드 = 하나의 JSON 또는 XML 문서
        - 레코드가 모이면 테이블이 되는 것처럼 문서지향 데이터베이스는 문서가 모여 컬렉션을 이룬다.
- 그래프 데이터베이스
    - 그래프의 노드 형태로 저장하는 데이터베이스
    - 종류 : neo4j
    - 특징
        - 보통, 방향 그래프를 표현하기 위해 사용한다
        - 노드 간의 연결 관계와 방향이 유의미하므로 SNS 친구 관계나 교통망 같은 "관계"가 뚜렷하고 중요한 내용물을 저장하는 데 사용
- 컬럼 패밀리 데이터베이스
    - RDBMS 처럼 행렬이 존존재하고 로우 키를 통해 특정 행을 식별한다
    - RDBMS와 달리 스키마가 고정되어 있지 않아 자유롭게 열 추가 가능. 그리고 정규화나 조인을 사용하지 않음
    - 종류 : Cassandra, HBase

MongoDB와 Redis 맛보기

- MongoDB
    - 간단 명령어

```
use databaseName # 데이터베이스 생성/사용
db.createCollection("collectionName") # 컬렉션 생성

show dbs # 데이터베이스 조회
show collections # 컬렉션 조회

db.collectionName.insertOne({key : value, key : value, ...}) # 단일 레코드 삽입
db.collectionName.insertMany([{key: value, ...}, {key: value, ...}, ...]) # 여러 레코드 삽입

db.collectionName.updateOne({key: value}, {$set: {key: value}}) # 단일
```

```
레코드 갱신. 첫번째 {key: value}는 식별 기준이고 뒤 $set은 갱신(없으면 삽입)할 내용물이다
db.collectionName.updateMany({key: { $gte: 20 }, {$set: {status: active}}}) # 여러 레코드 갱신. 여기서는 age가 20이상인 모든 문서의 status 필드를 active로 변경(없으면 삽입)하는 의미
# 만약 updateOne이든 updateMany든 필드를 없애고 싶으면 $set이 아니라 $unset을 사용하면 된다.

db.collectionName.find() # 컬렉션에 삽입된 문서 명세 확인
db.collectionName.find({key : value}) # 컬렉션 문서 중에 특정 키의 특정 값에 해당하는 문서를 찾는다

db.collectionName.deleteOne({key: value}) # 단일 문서 삭제
db.collectionName.deleteMany({age: { $gte: 20 }}) # 여러 문서 삭제
```

- 연산자

```
$eq # 지정한 값과 같은 값을 가진 문서 찾기
$ne # 지정한 값과 같지 않은 값을 가진 문서 찾기
$gt # 지정한 값보다 큰 값을 가진 문서 찾기
$lt # 지정한 값보다 작은 값을 가진 문서 찾기
$lte # 지정한 값보다 작거나 같은 값을 가진 문서 찾기
$and # 모든 조건이 참인 문서 찾기
$or # 하나 이상의 조건이 참인 문서 찾기
$not # 조건이 거짓인 문서 찾기
$exists # 필드의 존재 여부 확인하기
$type # 필드의 자료형 확인하기
$sum # 합계 계산하기
$avg # 평균 계산하기
$min # 최솟값 계산하기
$max # 최댓값 계산하기
```

- Redis
  - 키-값 데이터베이스로 값에 들어갈 수 있는 게 문자열, 리스트, 해시 테이블, 집합 등으로 다양하다.
  - 값으로 문자열을 다룰 시 사용하는 명령어
    - SET : 문자열 값 저장
    - SETNX : 키가 존재하지 않는 경우에만 문자열 저장
    - GET : 문자열 값 조회
    - MGET : 여러 문자열 값 조회
    - DEL : 키 삭제
  - 값으로 리스트를 다룰 시 사용하는 명령어
    - LPUSH : 리스트의 왼쪽에 새로운 요소 추가
    - RPUSH : 리스트의 오른쪽에 새로운 요소 추가
    - LPOP : 리스트의 왼쪽에 요소를 제거 후 반환

- RPOP : 리스트의 오른쪽에서 요소를 제거 후 반환
- LRANGE : 지정된 범위의 요소들을 반환
- LLEN : 리스트 길이 반환

3) 데이터베이스 분할과 샤딩

데이터베이스 분할

- 레코드가 너무 많은 테이블의 경우, 불러오는 데 부하가 심할 수 있다. 이 때, 부하를 줄이기 위해 테이블을 물리적으로 분할하는 걸 데이터베이스 분할(database partitioning)이라 한다.
- 분할 방식
  - 수평적 분할 : 레코드가 과도하게 많은 경우에 행을 기준으로 범위를 산정하여 쪼개서 저장
  - 수직적 분할 : 필드가 과도하게 많은 경우에 열을 기준으로 범위를 산정하여 쪼개서 저장
- 수평적 분할 방법
  - 범위 분할(range partitioning) : 레코드 데이터가 가질 수 있는 범위를 정의하고 해당 범위를 기준으로 데이터를 분할한다
  - 목록 분할(List Partitioning) : 레코드 데이터가 특정 목록에 포함된 값을 가질 경우 해당 레코드를 별도의 테이블로 분할
  - 해시 분할(Hash Partitioning) : 특정 열 데이터에 대한 해시 값을 기준으로 별도의 테이블로 분할
  - 키 분할(Key Partitioning) : 키를 기준으로 별도의 테이블로 분할

데이터베이스 샤딩

- 샤딩(sharding) : 분할된 테이블을 별개의 데이터베이스 서버에 분산하여 저장하는 기술
- 샤드(shard) : 분할되어 저장한 단위