맹준영_8, 9장

목표

- SELECT에 대한 이해

▼ 8.1 SELECT는 SQL의 심장부

- SELECT는 릴레이션이 연산 단위
- 한 개 이상의 릴레이션을 조합해 새로운 릴레이션을 뽑음
- 릴레이션 : 테이블 → 테이블의 데이터 조회는 "SELECT”만 가능

SELECT의 기본 구조

```
SELECT 컬럼 목록   # Projection
FROM 테이블의 목록  # Product
WHERE 검색 조건    # Restrict
```

- SELECT는 Projection, Product, Restrict의 연산이 한번에 이루어짐
  - Projection : 속성을 선택하는 연산
  - Product : 해당하는 릴레이션의 곱집합
  - Restrict : 특정 조건에 맞는 튜플을 포함한 릴레이션 반환 연산
- SELECT 내 연산의 논리적인 순서 : Product → Restrict → Projection

💡 RDB는 내부적으로 다양한 최적화를 수행하기 때문에 실행 순서와 연산의 논리적인 순서가 항상 같지 않을 수 있다!

- “실행 계획, 인덱스는 RDB가 SQL을 어떻게 실행할까?”에 대한 구현(물리적인) 관점
- “SELECT를 통해 어떤 결과를 얻을 것인가?”에 대한 논리적인 관점
- 구현 관점과 논리적인 관점은 다른 의미로 구분되어야함

▼ 8.2 SELECT의 다양한 모습

집계함수

- 구문이 동일해도 select list (프로젝션 컬럼 목록)에 집계함수가 포함된 경우 SELECT의 결과 전체가 집계결과가 된다.
- 집계함수의 유무에 따라 결과 행의 의미에 변화가 생김.
- 두 쿼리 모두 함수가 사용되고 있지만 일반 함수인지 집계 함수인지 얼핏 봐서 알아채기 어려운 점이 SELECT의 까따로운 점 중 하나

```
# select list에 함수가 사용된 경우 - Restrict의 조건에 해당하는 컬럼이 모두 반환
SELECT CONCAT(NAME, ':', DEPARTMENT)
FROM STUDENTS
WHERE DEPARTEMNT = 'DB';

# select list에 집계함수가 사용된 경우 - 구문은 동일하나 결과 행이 1건이다.
SELECT COUNT(*)
FROM STUDENTS
WHERE DEPARTMENT = 'DB';
```

COUNT의 특수성

- COUNT는 WHERE 절에 일치하는 행이 없는 경우 0을 반환
- COUNT를 제외한 다른 집계함수는 일치하는 행이 없는 경우 NULL 반환

```
# 일반 집계함수 사용 경우 - 0학년은 존재하지 않으므로 행을 반환 X
SELECT AVG(age)
FROM STUDENTS
WHERE GRADE = 0;

>> 결과 : NULL

# COUNT 사용 경우 - 0학년은 존재하지 않으므로 0을 반환
SELECT COUNT(age)
FROM STUDENTS
WHERE GRADE = 0;

>> 결과 : 0
```

GROUP BY를 이용한 집계의 서식

- GROUP BY : 테이블의 특정 컬럼을 통해 집계하고 싶은 겨우 사용
  - “SELECT가 집계를 나타내는 것”에 대해 직관적으로 알려줌
- HAVING : 집계 결과에 대해 조건을 지정하는 구문

> 💡 HAVING, WHERE의 구분
>
> WHERE : 집계의 대상이 되는 행의 조건 , 집계 전 원래의 행에 사용되는 조건
> HAVING : 집계 결과에 대한 조건, **GROUP BY에 지정된 컬럼, 집계함수의 결과만 사용 가능**

```
SELECT DEPARTMENT, COUNT(*)
FROM STUDENTS
WHERE GRADE IN (1,2)    # 집계 전 원래의 행에 조건 지정
GROUP BY DEPARTMENT     # 특정 컬럼 지정
HAVING COUNT(*) <= 30;  # 집계 결과에 대해 조건 지정
```

- **GROUP BY는 WHERE 절의 조건에 해당하는 행이 없는 경우 그 항목에 관해서는 결과가 표시되지 않음**
- 위의 쿼리에서 GRADE가 1,2인 학생이 없는 경우 DEPARTMENT가 추출되지 않으므로 COUNT(*)가 0임에도 결과가 반환되지 않음
- WHERE 절의 검색 조건에 일치하지 않는 행에 대한 집계(공집합)도 필요할 때 GROUP BY를 사용할 수 없다.
- COUNT 이외의 집계함수는 NULL을 반환하므로 NULL에 대한 대책도 필요하다.
- GROUP BY → HAVING → ORDER BY 순서로 작성

🔥 서브쿼리

- 서브쿼리는 외형은 모두 SELECT이나 서브쿼리의 결과는 스칼라, 행, 테이블과 같은 형태로 자유롭게 변화한다.

테이블 서브쿼리

1. IN, ANY(SOME), ALL 구에 따라서 사용되는 서브쿼리
    a. 서브쿼리의 결과가 한 열이 되는 경우가 많지만, 여러 컬럼을 한 번에 비교 가능

```
SELECT COUNT(*)
FROM COURSE_REGISTRATION
WHERE (DEPARTMENT, COURSE) IN (
  SELECT DEPEARTMENT, COURSE
  FROM COURSES
  WHERE MINIMUM_GRADE >= 2)
```

2. FROM 절의 서브쿼리
    a. 서브쿼리의 결과를 FROM 절에서 일반 테이블처럼 다룬다.
    b. SELECT에 따라서 추가 연산을 수행하거나 다른 테이블과 JOIN하면서 사용한다.

```
SELECT AVG(C)
FROM (
  SELECT COUNT(*) AS C
  FROM STUDENTS
  GROUPBY DEPARTMENT)
```

3. EXISTS 서브쿼리
    a. IN,ANY, ALL 등과 같은 용도이지만 서브 쿼리의 결과 행이 한개라도 존재하는지 아닌지이다.
    b. WHERE 절에 주로 사용되며 select list나 HAVING 절에도 사용할 수 있다.

```
SELECT NAME, DEPARTMENT
FROM STUDENTS
WHERE NOT EXISTS (
  SELECT *
  FROM COURSE_REGISTRATION
  WHERE STUDENT_NAME = STUDENTS.NAME)
```

스칼라 서브쿼리

- 서브쿼리의 결과가 스칼라(1행 1열), 아닌 경우 오류 발생
- WHERE, HAVING, select list에서 스칼라 값을 구하기 위한 목적으로 사용
- 결과가 없는 경우 NULL로 처리, COALESCE() 함수를 이용해 대응 가능

```
# WHERE절에 사용된 스칼라 서브쿼리
SELECT NAME, AGE
FROM STUDENTS S1
WHERE AGE = (
  SELECT MAX(AGE)
  FROM STUDENTS S2)

# HAVING절에 사용된 서브쿼리
SELECT COURSE, COUNT(*)
FROM COURSE_REGISTRATION
GROUP BY COURSE
HAVING COUNT(*) > (
  SELECT AVG(C)
  FROM (
    SELECT COUNT(*) AS C
    FROM COURSE_REGISTRATION
    GROUP BY COURSE) AS T)

# select list에 사용된 서브쿼리
SELECT (
  SELECT AVG(AGE)
  FROM STUDENTS S
  WHERE S.DEPARTMENT = D.DEPARTMENT) AS AGE
FROM DEPARTMENT D
```

행 서브쿼리

- 서브쿼리의 결과가 1행이고 열이 여러개인 경우
- 스칼라와 비교해 컬림이 여러개인 것만 차이가 존재
- 결과가 없는 경우 NULL로 처리, 단일값이 아니므로 COALESCE(0 함수 사용 불가)
- select list에 사용이 불가능
- 스칼라 서브쿼리, 행 서브쿼리 양쪽 모두 서브쿼리의 평가 결과가 여러행이면 오류 발생

뷰

- 관계형 모델은 뷰와 테이블을 구분 X, 테이블과 뷰 모두 릴레이션을 나타내는 것
- 뷰를 이용하면 쿼리가 깔끔해지지만, 백그라운드에서 어떤 처리가 이뤄지는지 보이지 않는다는 단점 존재
- 서브쿼리나 UNION, 집계함수 포함되는 경우 성능 문제 발생 가능성 존재 → 주의 필요

UNION

- SQL 사양 상 결과 집합에 포함된 컬럼 수가 같다면 두 개의 SELECT를 UNION으로 더하기 가능
- UNION으로 더한 두 개의 SELECT는 다른 테이블을 참고하고 있거나 전혀 다른 실행 계획이다.
- 출력 형태는 비슷하다는 공통점은 갖지만, 내용은 다를 가능성이 존재

▼ 8.3 관계형이 아닌 조작

- SELECT는 관계형 모델에 기반하지 않은 조작 연산도 지원하므로 주의가 필요

릴레이션 연산과 SQL

표 8.1 SELECT에서의 관계형 조작

릴레이션의 연산	SELECT로 표현
제한	기본형 : WHERE 절
사영	기본형 : select list
곱집합	기본형 : FROM 절
결합	기본형 : FROM 절
곱	기본형 : FROM 절
합	UNION
차	NOT EXIST 서브쿼리, MINUS
속성명 변경	기본형 : select list
확장	기본형 : select list

정렬

- ORDER BY를 이용한 집합의 정렬은 관계형 모델 기반의 연산이 아니다.
- 집합은 요소의 순서를 갖지 않는다. 행을 정렬해 순열을 붙이는 것은 집합이 아니다.
- SQL에서 ORDER BY는 SELECT 가 아닌 커서의 조작이다.

명시적으로 정의되지 않은 컬럼

- ROWNUM, ROWID는 암묵적으로 사용 가능한 컬럼
- 릴레이션의 튜플은 순서가 없음
- 관계형 모델을 파괴하는 컬럼 → 주의해 다룰 필요성

스토어드 함수 (사용자 정의 함수)

- 스토어드 함수는 절차적으로 작성되어 처리되기 때문에 옵티마이저는 스토어드 함수의 실행 비용을 예측불가
- 스토어드 프로시저 또한 내부는 절차형 프로그래밍 언어로 작성되어 있으므로 문제가 발생할 수 있다.

집계와 GROUP BY

- 집계에 의한 연산 결과는 릴레이션이 아닌 스칼라이다.
- 릴레이션의 연산의 결과는 릴레이션이 되는 클로져 성질이기 때문에 집계는 관계형 모델의 연산에서 벗어난 연산이다.
- SQL에서 GROUP BY와 함께 사용되는 집계함수의 결과는 릴레이션이다.
    - GROUP BY를 이용한 항목별로 집계 데이터를 얻는 조작은 **"요약"**이라 한다.

관계형이 아닌 조작의 취급법

- RDB를 이용한 애플리케이션 개발에서는 관계형이 아닌 연산도 필요하다.
- 관계형 조작과 글렇지 않은 조작의 명확한 구별이 필요
    - 관계형 모델의 범위내에서 조작이 가능하면 관계형이 아닌 조작을 사용하지 않는다.
    - 관계형 모델의 범위에서 작성할 수 없다면 DB 설계를 검토한다.
    - 관계형 모델이 아닌 조작이 필요한 경우 관계형 조작에 대한 로직을 반드시 먼저 실행한다.
- 옵티마이저의 작업은 원래의 쿼리와 같은 결과를 얻을 수 있는 쿼리 중에 최적의 실행 계획이나 가장 실행시간이 짧은 것을 선택
- 릴레이션이 아닌 연산이 쿼리에 포함되는 경우 집합의 연산으로 적용하는 것이 어려워 옵티마이저가 선택할 수 있는 실행 계획이 제한되어 효율성이 떨어진다.

▼ 8.4 요약

- SELECT는 관계형 조작과 관계형이 아닌 조작이 모두 포함되어 있다.
- SELECT를 이용해 복잡한 로직 처리도 가능하며 SELECT 코드의 이해가 중요하다.

▼ 9.1 이력 데이터의 문제점

- 응용프로그램이 로그 데이터를 테이블에 저장하거나,과거부터 현재에 이르는 데이터를 저장할 때 생성되는 것
- 일반적으로 타임스탬프나 버전 번호와 같이 저장

이력과 관계형 모델의 상성 문제

- 릴레이션은 집합이므로 각 요소간 순서가 존재하지 않는다.
- 이력에는 어느 쪽이 오래된 쪽인지, 새로운 것인지에 관한 순서가 존재한다.
- 이력 데이터는 쉽게 테이블이 커지므로 성능 저하가 나타날 수 있다.

이력 데이터의 예

item	price	start_date	end_date
아령 세트	100000	2010-01-01	9999-12-31
악력기	40000	2013-04-01	2014-03-31
악력기	50000	2014-04-01	9999-12-31
턱걸이 기계	180000	2010-01-01	2011-12-31
턱걸이 기계	200000	2012-01-01	2014-12-31
턱걸이 기계	220000	2015-01-01	9999-12-31

```
# 현재 날짜가 2014-12-31인 경우, 턱걸이 기계의 현재 가격 구하는 쿼리
SELECT PRICE
FROM PRICE_LIST
WHERE ITEM = '턱걸이 기계'
AND NOW() BETWEEN START_DATE AND END_DATE;
```

- 이력 데이터는 릴레이션이 시간축과 직교하지 않는다.
  - 시간에 따라 쿼리의 실행 결과가 변한다.
  - 같은 데이터, 같은 쿼리임에도 시간에 따라 결과가 달라진다.

NULL의 가능성

- end_date는 설계에 따라 NULL이 될 수도 있다.
- NULL이 되는 경우 검색 조건이 복잡해지고 성능에 악영향을 미칠 수 있다.
- 컬럼 값에 NULL이 들어갈 여지가 있는 것은 DB 설계에 문제가 있을 수 있다.

특정 행만 의미가 다르다

item	price	start_date
아령 세트	100000	2010-01-01
악력기	40000	2013-04-01
악력기	50000	2014-04-01
턱걸이 기계	180000	2010-01-01
턱걸이 기계	200000	2012-01-01

- end_date가 존재하지 않고 start_date만 존재하는 경우

```
SELECT PRICE
FROM PRICE_LIST
WHERE ITEM = '턱걸이 기계'
AND START_DATE = (
  SELECT MAX(START_DATE)
  FROM PRICE_LIST
  WHERE ITEM = '턱걸이 기계');
```

- 집계 연산은 관계형 모델을 벗어난 연산이다.
- 각 행의 의미가 균일하지 않은 문제가 발생
  - 릴레이션 : “참이 되는 명제의 집합, 릴레이션 내 튜플은 주어진 명제에 모두 참인 결과”
  - 테이블에는 start_date가 최신인 행 = 현재 가격(참)
  - 이외의 행 = 과거 가격(거짓)
  - 한 명제에 대해 튜플들의 결과가 참/거짓이 구분된다. → 두 개 이상의 릴레이션의 합집합일 가능성 존재

▼ 9.2 이력 데이터에 대한 해결책

릴레이션을 나눈다

price_list

item	price	start_date
아령 세트	100000	2010-01-01
악력기	50000	2014-04-01
턱걸이 기계	200000	2012-01-01

price_list_history

item	price	start_date
악력기	40000	2013-04-01
턱걸이 기계	180000	2010-01-01

- 같은 명제로 평가할 수 없는 튜플은 같은 릴레이션에 포함시키지 않는다.
- 튜플 기준으로 릴레이션을 나눠야 하기 때문에 속성별로 릴레이션을 나누는 정규화는 이러한 필요성을 확인하기 힘들다.

    - 현재의 가격 릴레이션과 과거의 가격 릴레이션으로 나눈다.
    - 나타나는 문제점
        - 현재부터 과거까지의 이력에서 통계를 찾는 경우 UNION쿼리 실행이 필요
        - 외부키 제약조건을 사용할 수 없으므로 트리거를 사용해야한다.
        - 데이터 부정합이 발생할 수 있다. (현재 데이터가 갱신되는 경우 하나의 트랜잭션 내에서 이뤄지지 않아 과거 데이터에도 적재되는 경우 부정합 문제 발생)

중복행을 허용

- 모든 이력 데이터를 하나의 릴레이션에 저장 후 특정 이력만 새로운 릴레이션에 중복해 저장하는 설계
- 특정 이력 릴레이션에 외부키 제약조건을 지정
- 모든 이력 데이터 릴레이션에 데이터를 우선 삽입
- 모든 이력 데이터 릴레이션은 update 하지 않음
    - update의 경우 새로운 행을 추가하고 특정 이력 릴레이션에서 오래된 행을 삭제
    - update 트리거 사용을 통한 테이블 동기화

대리키

> 대리키
>
> 기본키가 보안을 필요로 하는 속성을 가지고 있거나, 여러 개의 속성으로 구성되어 있어 복잡하거나,
> 기본키로 사용할 속성이 없을 경우에 일련번호 같은 가상의 속성을 생성하여 기본키로 사용하는 키

- 대리키 테이블에 대해 분해한 이력 테이블들이 외부키가 필요
- 분해한 이력 테이블간 튜플의 중복이 존재하면 안됨
- JOIN 연산을 통해 이력 집계
- 이력 테이블의 외래키를 대리키로 사용하기 때문에 기존 후보키가 될 수 있는 속성들은 유니크 제약조건 설정 필요
    - auto increment 값과 같이 일련번호를 사용하는 경우 해당 키 만으로 기본키가 될 수 있기 때문에 나머지 속성은 고유하지 않을 수 있음
    - 유니크 제약조건은 디스크 공간이 낭비되고 제약 확인 위한 오버헤드가 발생

▼ 9.3 이력데이터의 안티 패턴

- 이력 데이터를 다룰 때 피해야 하는 방식

플래그 사용

- 플래그를 사용하면 쿼리는 단순해 보이는 장점
- 플래그로 사용한 컬럼은 카디널리티가 낮아 효율이 높지 않다.
- 키가 아닌 속성(start_date, end_date) → 키가 아닌 속성 (flag) 결정 = 3NF 만족하지 않은 테이블
- 시간에 따라 유효성을 검증할 수 없으므로 정기적으로 플래그 수정 필요

절차형으로 구현하자

- 관계형 모델로 대응할 수 없는 경우 스토어드 프로시저나 함수로 로직을 구현
- 절차형 로직에 의존하면 관계형 모델의 데이터 정합성을 잃어버리는 문제 발생

▼ 9.4 요약

- DB 설계 검토해야하는 현상
    - 상태나 플래그를 나타내는 컬럼이 있는 경우
    - 초깃값이 NULL인 컬럼 존재하는 경우

    - 현재 시각과 비교하는 쿼리가 있는 경우
    - 온라인 트랜잭션 중에 ORDER BY N DESC LIMIT 1, MAX()/MIN()/COUNT()가 사용되는 경우
    - 버전을 나타내는 컬림이 있는 경우
    - INSERT/DELETE보다 UPDATE의 비율이 높은 경우
- 응용프로그램이 데이터를 어떻게 볼것인가? 라는 자의성을 통해 보는 것이 이력데이터 처리의 핵심