Database Project 2: Implementing a Simple Database Application

2013-11440
CSE 허정수

1. 개요

한 학기 동안 데이터베이스의 정의부터 시작해서, 데이터베이스 모델의 종류, SQL문, 저장 방식, 탐색 방식, indexing, query processing, data mining 등 많은 것을 배웠다. Project 1에서는 SQL문이 어떻게 동작하는지 원리를 파악한 후 parser를 이용해 DDL, DML 명령어를 처리하는 프로그램 만들었다. 하지만 데이터베이스는 결국 실생활에서 데이터를 관리하기 위해 생긴 개념이다. 그렇기 때문에 데이터베이스는 실제 상황을 잘 반영할 수 있어야 한다. Project 2에서는 실제 일어날 수 있는 상황을 가정하고, 알맞게 데이터베이스를 설계하여 이를 이용한 어플리케이션을 만들어본다.

2. 목적

MariaDB를 이용해 간단한 대학교 원서 접수 어플리케이션을 만든다. 이 어플리케이션은 학교와 학생의 정보를 입력 받아 저장할 수 있고, 학교와 학생의 데이터를 이용하여 학생이 학교에 원서 접수를 할 수 있어야 하며, 각 학교별 정원과 내신 가중치를 고려하여 지원자들의 합/불 여부를 예측할 수 있어야 한다.

3. 핵심 알고리즘

먼저 schema를 설계해야 한다. DB를 3개의 table로 구성하였다. 구성은 다음과 같다.

1) University = {id, name, capacity, group, weight, applied, pass_scaled_score, pass_school_score}

: 대학에 대한 기본 데이터 6종류와 합격 커트라인 점수를 추가했다. 학생의 환산 점수(scaled_score = csat_score + weight * school_score)가 커트라인 보다 높으면 합격이고, 환산 점수가 같으면 내신의 비교를 통해 합격 여부를 결정한다.

2) Student = {id, name, csat_score, school_score}

: 학생에 대한 기본 데이터 4종류를 가지고 있다.

3) Apply = {stud_id, univ_id, group, scaled_score}

: 학생의 대학 지원에 대한 정보를 담고 있는 table이다. Group은 대학의 group이고 scaled_score은 각 대학의 weight에 따른 학생의 환산 점수이다.

의미상으로는 University, Student table에서는 id가, Apply table에서는 stud_id와 univ_id가 primary key이고, Apply table의 stud_id와 univ_id는 각각 Student와 University table의 primary key

를 참조하는 foreign key이지만, 따로 DB에 설정을 해주지는 않았다.

정리하자면 데이터베이스의 간단한 다이어그램은 다음과 같다. Attribute의 표시는 생략하였다.

이제 만든 DB를 이용하여 총 12개의 기능을 수행할 수 있어야 한다. 함수 종류별로 설명한다. () 안의 번호는 지금 설명하는 함수와 관련된 기능의 번호들이다(1~12).

1) Main (#12, else)

먼저 메인 함수에서 수행할 기능을 받으면 switch 문으로 알맞은 함수를 실행하도록 했다. 만약 12가 들어오면 "bye!"를 출력하고 프로그램을 종료하도록 하였다. 1~12 외의 범위가 들어오면 적절한 메시지와 함께 아무 명령도 수행하지 않는다.

```
case 12:
    System.out.println("Bye!");
    System.exit(0);
default:
    System.out.println("Invalid action.");
    System.out.println(doubleLine);
```

2) Print (#1, #2, #8, #9, #10, #11)

Print를 하는 6개의 기능은 대학을 출력하는 3개의 기능을 묶고, 학생을 출력하는 3개의 기능을 묶었다. 출력하는 포맷은 묶인 집합끼리 똑같으므로 같은 함수에서 수행하도록 하고, 각 기능은 SQL문을 구성하는데 차이가 있다. 만약 #1 명령을 수행할 때는 `selectSql = "SELECT * FROM university";` 로 간단하게 SQL문을 만들 수 있다. #9는 합격/불합격 여부에 관계없이 학생이 지원한 모든 대학 데이터를 가져오고, #11은 합격할 수 있는 대학 데이터만 가져온다. 그래서 다음과 같이 default로 지원한 모든 대학을 가져오게 만든 후, 만약 #11 명령이라면 그 중에서 합격 커트라인 점수를 넘은 대학만을 선택하도록 where clause를 추가해준다. 또한, 존재하지 않는 student id를 받는 경우는 적절한 메시지와 함께 에러처리를 해준다.

```
switch(code) {
case 1:
    selectSql = "SELECT * FROM university";
    break;
default: // case 9, 11
    System.out.print("Student id: ");
    stud_id = Integer.parseInt(br.readLine());
    checkSql = "SELECT * FROM student WHERE id = " + stud_id;
    checkStmt = conn.prepareStatement(checkSql);
    checkRs = checkStmt.executeQuery();

    if (!checkRs.first()) {
        System.out.println("University " + stud_id + " doesn't exist.");
        System.out.println(doubleLine);
        return;
    } else {
        selectSql = "SELECT * FROM university JOIN apply ON (university.id = univ_id) JOIN student ON (student.id = stud_id) WHERE stud_id = " + stud_id;
    }

    /* case 11 */
    if (code == 11) selectSql += " AND (pass_scaled_score < scaled_score OR (pass_scaled_score = scaled_score AND pass_school_score <= apply.school_score))";
    break;
}
```

학생 데이터를 출력하는 나머지 3개의 기능도 위와 거의 같은 방식으로 처리할 수 있다. 자세한 설명은 생략한다.

3) Insert (#3, #5)

새로운 학교와 학생의 삽입은 쉽게 구현할 수 있다. 주어진 데이터를 받아, 적절한 범위의 값을 가지는지 검사 후 데이터베이스에 삽입한다. 만약 범위를 벗어나는 값이 들어오면 즉시 적절한 메시지를 출력하고 해당 명령의 수행을 종료한다. 학교 레코드를 생성할 때는 지원자가 없으므로 applied, pass_scaled_score, pass_school_score 값은 모두 0으로 초기화 한다. 복잡한 내용이 없으므로 자세한 설명은 생략한다.

4) Apply (#7)

학생이 대학에 지원할 때는 먼저 해당 학생과 학교의 id가 존재하는지 검사한다. 존재하지 않는 id가 들어온다면 즉시 적절한 메시지와 함께 명령을 종료한다. 해당 id가 존재한다면, 해당 학교의 group, weight에 대한 정보를 불러온다. Apply table에서 해당 학생이 지원한 데이터 리스트를 뽑고, 그 중 현재 지원하려 하는 학교와 겹치는 group이 있는지 검사한다. 이미 같은 group의 대학에 지원했다면 지원할 수 없음을 알리는 메시지를 띄운다. 그렇지 않다면 weight을 이용해 환산 점수를 계산하고, 학생id, 학교id, group, 환산 점수 데이터를 가진 레코드를 insert 한다. 중복 group 검사의 편의성을 위해 apply table에 'group' column을 추가했다.

레코드를 insert 했다면, 학교의 입장에서는 지원자의 수에 변동이 생겼고, 그에 따라 합격 커트라인도 바뀔 수 있다. 그러므로 해당 학교는 변동 가능성이 있는 정보들을 update한다. Update 함수들은 아래에서 자세히 설명한다.

```
/* update 'applied' in university table */
updateApplied(conn, univ_id);

/* update 'pass_scaled_score' & 'pass_school_score' in university table */
updatePassScore(conn, univ_id);
```

5) Delete (#4, #6)

레코드를 삭제할 때는 항상 apply table에서 해당 학생이나 학교와 관련된 레코드를 먼저 삭제한

후, university나 student table에서 해당 레코드를 삭제한다. 만약 student를 삭제하는 과정에서 apply table에서 지원 내역을 지운다면, 지원했던 학교의 입장에서는 지원자 수가 감소하고, 합격 커트라인 점수의 변동이 있을 수 있으므로 apply table에 레코드를 insert할 때처럼 해당 학교의 정보들을 update한다.

6) UpdateApplied()

지원자 수를 계산하는 함수로 쿼리 하나로 간단히 구할 수 있다. 구한 값을 쿼리를 통해 다시 update 해준다.

```
String selectSql = "SELECT count(*) FROM apply WHERE univ_id = " + univ_id;

String updateSql = "UPDATE university SET applied = " + applied + " WHERE id =" + univ_id;
```

7) UpdatePassScore()

합격 커트라인 점수를 계산하여 update하는 함수다. 먼저 apply table에서 해당 대학의 지원자 데이터를 모두 가져온다. 이 데이터에는 미리 계산해놓은 환산 점수와 내신 점수가 포함되어 있으며 환산 점수, 내신 점수 순서의 내림차순으로 정렬되어 있다. 즉, 이 순서는 지원자들의 등수이다.

[1] 일단 지원자 수가 정원보다 적을 때는 모두 합격할 수 있으므로 커트라인을 0으로 정한다.

```
if (applicantNum <= capacity) { // 1) all pass
    pass_scaled_score = 0;
    pass_school_score = 0;
```

[2] 만약 지원자 수가 정원보다 많다면, 알맞은 커트라인 점수를 계산해야 한다. 정원이 100명이라면, 일단 100등의 점수가 임시 커트라인이다. 그리고 100등과 동점자가 존재하는지 검사한다. 동점자가 존재할 때마다 합격 인원을 늘려가며 몇 명이나 있는지 찾는다.

```
} else { // 2) otherwise
    float tmpScaledScore = scaledScoreSet[capacity-1];
    int tmpSchoolScore = schoolScoreSet[capacity-1];
    int maxCapacity = (int)Math.ceil(capacity * 1.1);
    int passNum = capacity;
    do {
        if (tmpScaledScore == scaledScoreSet[passNum] && tmpSchoolScore == schoolScoreSet[passNum]) {
            passNum++;
        } else {
            break;
        }
    } while (passNum < applicantNum);
```

[2-1] 만약 동점자를 모두 포함한 합격생의 수가 최대 합격자의 수를 초과하지 않는다면, 이 동점자까지 모두 합격할 수 있다. 합격 커트라인은 기존에 있던 100등의 점수와 일치한다.

```
if (passNum <= maxCapacity) { // 2-1) tie students all pass
    /* empty */
```

[2-2] 만약 동점자를 모두 포함한 합격생의 수가 최대 합격자를 초과한다면, 이 동점자들은 모

두 탈락한다. 그러면 다시 100등으로 돌아와 앞쪽으로(99등쪽) 재검사를 한다. 100등과 다른 점수가 나오는 첫 번째 사람의 점수가 커트라인 점수가 된다. 만약 100등과 99등은 동점이고, 98등과는 점수가 다르다면, 98등의 점수가 합격 커트라인이 된다. 만약 100등과 1등이 동점이라면 합격자는 없고, 커트라인 점수를 (1등의 점수) + 1로 한다.

```
} else { // 2-2) tie students all fail
    passNum = capacity-1;
    do {
        if (tmpScaledScore != scaledScoreSet[passNum-1] || tmpSchoolScore != schoolScoreSet[passNum-1]) {
            tmpScaledScore = scaledScoreSet[passNum-1];
            tmpSchoolScore = schoolScoreSet[passNum-1];
            break;
        }
        passNum--;
    } while (passNum > 0);
    if (passNum == 0) {
        tmpScaledScore = scaledScoreSet[0] + 1; // no one can pass!
        tmpSchoolScore = 0; // invalid!
    }
}
pass_scaled_score = tmpScaledScore;
pass_school_score = tmpSchoolScore;
```

구한 값을 다시 해당 학교의 pass_scaled_score와 pass_school_score에 update해준다.

4. 가정한 것들

- 학교와 학생의 name이 제한 길이를 초과했을 시에 특별한 에러처리 없이 최대 길이만큼 truncate 후 삽입한다. 실제 test시에는 길이를 초과하는 input이 존재하지 않는다.

- 에러가 발생하면 즉시 적절한 메시지를 띄우고 현재 명령의 수행을 종료한다.

- 학교와 학생 테이블의 모든 column에 대해 input으로 null이 들어오지 않는다.

- 학교와 학생 테이블의 모든 column에 대해 input으로 올바른 data type만 들어온다.

- Print 형식, 정렬 방식은 자유롭게 한다.

- 올바른 결과만 출력하면 되고, 그 과정은 따지지 않는다.

5. 컴파일과 실행 방법

윈도우 이클립스에서 java파일 컴파일 후 실행하고, console 창으로 테스트 하였다. HeidiSQL 프로그램을 설치하여 데이터베이스에 올바르게 insert, delete, update가 수행되는지 추가로 확인하였다.

6. 프로젝트를 하면서 느낀 점

마지막 프로젝트라 어려울 것이라 예상했는데 생각할게 별로 없어 간단했다. 앞서 했던 프로젝트들도 원리를 이해한다는 점에 있어 중요하지만, 이번 프로젝트도 실제로 SQL을 이용해 프로그램을 만든다는 점에서 중요하다고 생각된다. 그럼에도 불구하고 직전의 프로젝트와 난이도 차이가 너무 많이 나서 아쉽다. 조금 더 복잡한 스펙의 프로젝트가 주어졌으면 하는 아쉬움이 있다.