2019-2 [406.426B_001 DMA]

Team Project 2 – DB implementation and query processing

12조 박소정, 이준현, 이창민, 한승우

DB implementation and Query processing

- Using Python and MySQL

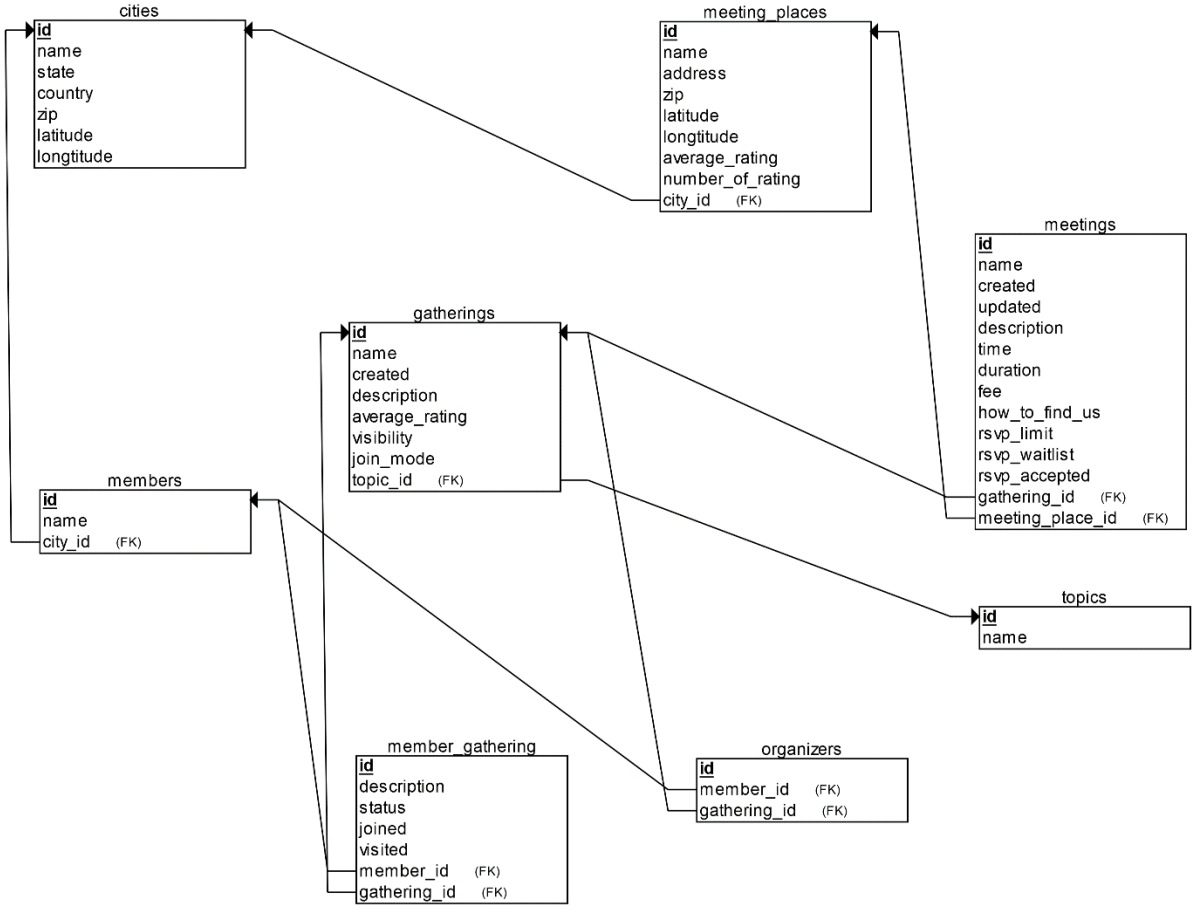
서 울 대 학 교
SEOUL NATIONAL UNIVERSITY

목차

1. 서론

1.1. 문제 정의

본 프로젝트는 특정 사이트 A의 사용자, 그룹, 기타 관련 데이터를 이에 적합한 데이터베이스 스키마를 설계한 후 DB 테이블을 생성하고, 데이터를 입력하며 이를 활용하는 프로그램을 구현하는 것을 목적으로 한다. 주어진 데이터베이스 스키마의 형태는 다음과 같다. 본 프로젝트의 세부 목적은 아래 표현된 스키마의 형태에 따라 python의 mysql.connector 패키지를 이용하여 MySQL 서버에 database를 만들고 각각의 relation에 해당하는 table을 만들고, 주어진 csv 데이터를 업로드 한 후 R2-R6의 쿼리문을 작성하여 결과 테이블을 출력하는 것에 있다.



1.2. 제약조건에 대한 설명

제약조건에 대한 설명은 Project 2 Requirements에 자세하게 나와있다. mysql.connector 이외의 패키지를 불러오지 않고 csv 파일을 불러오는 것과, 각각의 쿼리를 nested 형태로 하나의 SQL 문장(R5제외)으로 작성하는 것이 핵심 제약조건이다.

2. 본론

2.1. R1 코드 설명

R1 코드는 크게 세가지 부분으로 나뉜다 우선, DMA_team12의 이름을 가지는 schema를 생성하고, 데이터를 저장하는 테이블을 생성한다. 이 때 테이블의 각 attribute의 data type에 대한 지정과 nullable에 대한 여부를 같이 지정해주어야 한다. 두 번째로 만들어진 테이블 형태에 가지고 있던 csv 파일의 데이터 값을 저장해야 한다. 마지막으로, 프로그램 실행 및 수정으로 referential integrity 가 손상되지 않도록 foreign key 설정을 해줘야 한다.

2.1.1. Database와 테이블 구축

본 프로젝트에서 다룰 database는 A사이트에 관한 것으로, 총 8개의 서로 다른 데이터를 가지고 있는 table을 구축해야한다. 각각의 테이블은 cities, gatherings, meeting_places, meetings, members, member_gathering, organizers, topics이다. 테이블 구축에 관한 내용은 각각의 테이블마다 동일한 과정을 거치기 때문에 첫 번째 cities 테이블을 구축하는 과정만 살펴볼 것이다.

```
cnx = mysql.connector.connect(host=host, user=user, password=password)
cursor = cnx.cursor()
cursor.execute('SET GLOBAL innodb_buffer_pool_size=2*1024*1024*1024;')
```

우선 import 한 mysql.connector를 이용하여 cursor 객체를 생성하고, 많은 데이터를 빠른 속도로 처리하기 위해서 buffer pool size를 키워준다.

```
cursor.execute("""DROP DATABASE IFEXISTS DMA_team12;""")
cursor.execute("""CREATE DATABASE DMA_team12;""")
cursor.execute("""USE DMA_team12;""")
```

이후 DMA_team12라는 이름을 가진 database schema를 생성하되 만약 같은 이름의 schema가 있다면 삭제하고 새로 만든다. 그리고 생성한 db schema를 활성화 시킨다.

```
#cities TABLE 생성
cursor.execute("""
    CREATE TABLE IF NOT EXISTS cities(
    id VARCHAR(22) NOT NULL,
    name VARCHAR(255) NOT NULL,
    state VARCHAR(30) NOT NULL,
    country VARCHAR(30) NOT NULL,
    zip INT NULL,
    latitude FLOAT NULL,
    longitude FLOAT NULL,
    PRIMARY KEY (id));
    """)
```

위의 코드는 cities 테이블을 생성하는 코드이다. cities table은 id, name, state, country, zip, latitude, longitude 이렇게 7개의 attribute를 가지고 있다. id attribute의 경우 해당 테이블에서

primary key이기 때문에 entity integrity를 만족시키기 위해 마지막에 "primary key(id)" 가 추가되었으며, 각각의 attribute를 지정할 때는 datatype과 허용 가능 길이, nullable의 여부가 지정되었다. 각 attribute의 datatype과 length, nullable 여부에 대한 제약조건은 dataset explanation.xlsx 파일을 참고하여 만들었다. 하나만 살펴보면 "id VARCHAR(22) NOT NULL"은 id라는 column은 크기변동가능한 텍스트 datatype으로 최대 22자까지 쓸 수 있으며, NULL값일 수 없다 라는 것을 의미한다.

위의 방식으로 나머지 7개의 테이블에 대한 작업을 진행해주면 Database와 테이블이 생성된다.

2.1.2. Csv 파일 읽어서 데이터 삽입하기

csv파일을 읽어서 데이터를 삽입하는 과정은 다음의 과정으로 요약할 수 있다.

(1) Csv 파일의 한 줄의 value를 테이블에 삽입하는 함수 생성
(2) 가져올 csv 파일을 open
(3) While loop를 활용하여, csv 파일의 data preprocessing 과정을 거치고 한 줄씩 함수를 이용하여 삽입하되, 더 이상 데이터가 없을 때 while loop를 끝냄.

마찬가지로 이 과정 또한 모든 테이블에 대해서 동일한 과정으로 코드가 진행되기 때문에, cities 테이블에 대한 설명으로 나머지 테이블의 설명을 대체할 것이다.

```
sql_cities = """INSERT INTO cities VALUES (%s, %s, %s, %s, %s, %s, %s)"""
```

위의 코드는 formatting을 이용해서 cities 테이블에 csv파일의 값들을 insert하는 과정을 보여주고 있다. sql_cities가 실행되면 csv 파일의 한 row data 가 생성된 cities 테이블에 삽입된다. cities 테이블에는 총 7개의 attribute가 있고 한 줄에 7개의 값이 들어가 있기 때문에 7개의 %s가 들어가 있다.

```
f_cities = open(directory+'cities.csv', 'r', encoding='utf-8')
next(f_cities)
```

f_cities는 cities.csv 파일을 불러오는 것이다. 데이터가 들어 있는 경로인 directory를 input으로 받은 후, 각 데이터 파일의 이름(ex-cities.csv)과 합하여 최종 데이터 경로를 만들어 냈고, 이 파일들을 불러왔다. 그리고 csv file을 불러올 경우 첫 번째 줄은 column명들이므로 next(f_cities)를 수행하여 두 번째 줄부터 데이터를 삽입한다.

```
while True:
    line = f_cities.readline()
    line_list = line.replace('\n', '')
    line_list = line_list.split(',')
    for i, j in enumerate(line_list):
        if line_list[i] == '':
            line_list[i] = None
        try:
            line_list[i] = int(j)
        except:
            pass
   if line:
        cursor.execute(sql_cities, line_list)
   else:
        break
```

위의 While 문은 큰 틀에서

(1) Csv 파일의 한 줄을 불러온다.

(2) 데이터 전처리 과정1을 거친다. (줄바꿈 기호 ₩n 삭제 및 콤마 기준 쪼개어 리스트화)

(3) For 문을 이용하여 데이터 전처리 과정2를 거친다. (csv 파일 데이터 안에 빈 문자열이 있을 경우 None으로 대체, try-except 구문을 활용하였다. Try에서 string 내부의 데이터가 int로 변환될 수 있는 숫자인 경우에는 int값으로 변환. 기타 데이터 타입은 except로 들어가고, pass 구문을 거친다.)

(4) If else 문으로 종료조건(데이터가 없을 때)을 지정해주고 이전 단계에서 정의한 sql_cities를 실행하여 cities 테이블 안에 한 줄 데이터 리스트를 삽입.

이런 구조로 이루어져있다. While loop는 더 이상 불러올 다음 줄의 데이터가 csv 파일에 없을 때까지 돌아가기 때문에 csv 파일의 모든 데이터들이 위에서 정의한 데이터 전처리 과정을 거쳐 원하는 형태로 cities 테이블에 값이 들어가게 된다.

- 추가적으로 meeting_places 테이블의 경우 (3) 번 과정에서 약간의 변형이 필요한데, 그 이유는 meeting_places의 다섯 번째 열인 zip의 경우 값이 float64로 나오는 경우가 있기 때문이다. 이 값은 한번에 int로 변환시키는 것이 어려워 먼저 float형태로 변환시켜주고 다시 int로 바꿔주는 과정이 필요하다. 밑의 코드를 참고하면 for 문 안에서 이전과 다르게 if문을 한번 더 넣어줘서 float 형태로 바꿔주고 int로 바꿔주는 구조를 확인할 수 있다.

(meeting_places 부분의 데이터 처리 코드)

```
for i, j in enumerate(line_list):

    if line_list[i] == '':
        line_list[i] = None

    if i == 4 and line_list[4] != None:
        line_list[i] = int(float(j))

    try:
        line_list[i] = int(j)
    except:
        pass
```

2.1.3. 외래키 설정하기

```
cursor.execute("ALTER TABLE gatherings ADD CONSTRAINT FOREIGN KEY (topic_id)
REFERENCES topics(id);")
```

Table이 완성되면 referential integrity를 만족시키게 하기 위해서 외래키 설정을 추가해줘야 한다. Table을 만들 때 설정해줄 수도 있지만, 그럴 경우 table을 생성시키는 순서도 신경 써야 하는 번거로움이 있기 때문에 맨 마지막에 설정을 통해서 바꿔준다. 이 코드도 예시로 한 개만 설명하면 gatherings table의 topic_id는 topics table의 primary key인 id를 참조하는 외래키로 설정한다는 것을 의미한다.

이렇게 세가지 순서를 통해서 R1을 만족시켰고, 8개의 table을 가진 DB를 효과적으로 생성하였다.

2.2. R2~R5 쿼리 설명

2.2.1. R2 : 2002~2017 사이트 A에서 연도별 사용자들의 활동량 추이

R2에서는 연도별 사용자들의 활동량의 추이를 확인하기 위해, 연도별로 사이트 A에서 새로이 그룹에 가입한 사용자의 수, 생성된 그룹의 수, 개최된 오프라인 모임의 수를 출력하기를 요구하고 있다. 출력된 결과는 연도에 따라 오름차순으로 정렬되어 있어야 하며, column의 순서는 연도, 해당 연도에 새로이 그룹에 가입한 사용자의 수, 해당 연도에 생성된 그룹의 수, 해당 연도에 개최된 오프라인 모임의 수여야 한다는 제약조건을 만족해야 한다.

```
def requirement2(host, user, password):
    cnx = mysql.connector.connect(host=host, user=user, password=password)
    cursor = cnx.cursor()
    cursor.execute('SET GLOBAL innodb_buffer_pool_size=2*1024*1024*1024;')
    cursor.execute('USE DMA_team%02d;' % team)
```

코드 형태는 위와 같이 mysql서버의 host, user, password를 input으로 갖는 함수 형태로 만들었다. 앞에서 살펴봤던 것과 마찬가지로 mysql.connector 패키지를 이용해서 mysql서버에 접근하고 cursor 객체를 만들어 DB에 접근하였다.

이제는 본격적으로 제약조건을 만족하는 결과값을 출력하기 위한 MySQL Query를 살펴볼 것이다.

2.2.1.1. MySQL Query

```
SELECT workson7.Year as Year, IFNULL(NewUser, 0) as NewUser, IFNULL(NewGroup, 0) as NewGroup, IFNULL(NewMeeting,0) as NewMeeting
FROM (SELECT year(joined) as Year, COUNT(*) as NewUser
FROM member_gathering
GROUP BY year(joined)) as workson1
RIGHT JOIN
(SELECT Year, NewGroup, NewMeeting
FROM ((SELECT workson2.Year, NewGroup, NewMeeting
FROM (SELECT year(created) as Year, COUNT(*) as NewGroup
FROM gatherings
GROUP BY year(created)) as workson2
LEFT JOIN (SELECT year(time) as Year, COUNT(*) as NewMeeting
FROM meetings
GROUP BY year(time)) as workson3
ON workson2.Year = workson3.Year)
UNION
(SELECT workson5.Year, NewGroup, NewMeeting
FROM (SELECT year(created) as Year, COUNT(*) as NewGroup
FROM gatherings
GROUP BY year(created)) as workson4
RIGHT JOIN (SELECT year(time) as Year, COUNT(*) as NewMeeting
FROM meetings
GROUP BY year(time)) as workson5
ON workson4.Year = workson5.Year)
) as workson6
) as workson7
ON workson1.Year = workson7.Year
ORDER BY Year
```

2.2.1.2. 코드 설명

코드가 nested query 형태로 이루어져 있어서 복잡하게 보이는데, nested 형태를 풀어 간소하게 표현해보면 다음과 같은 query가 된다.

```
SELECT workson7.Year as Year,
IFNULL(NewUser, 0) as NewUser,
IFNULL(NewGroup, 0) as NewGroup,
IFNULL(NewMeeting, 0) as NewMeeting
FROM workson1
RIGHT JOIN workson7
ON workson1.Year = workson7.Year
ORDER BY Year;
```

여기서 workson1 과 workson7에 대해서 각각 설명을 해보겠다. 먼저 workson7에 대해서 설명하겠다. Workson7는 아래의 쿼리문으로 이루어진 view이다.

```
SELECT Year, NewGroup, NewMeeting
FROM
(
(SELECT workson2.Year, NewGroup, NewMeeting
FROM    (SELECT year(created) as Year, COUNT(*) as NewGroup
         FROM gatherings
         GROUP BY (created)) as workson2
LEFT JOIN (SELECT year(time) as Year, COUNT(*) as NewMeeting
         FROM meetings
         GROUP BY year(time)) as workson3
ON workson2.Year = workson3.Year)
UNION
(SELECT workson2.Year, NewGroup, NewMeeting
FROM    (SELECT year(created) as Year, COUNT(*) as NewGroup
         FROM gatherings
         GROUP BY (created)) as workson4
RIGHT JOIN (SELECT year(time) as Year, COUNT(*) as NewMeeting
```

```
        FROM meetings
        GROUP BY year(time)) as workson5
ON workson4.Year = workson5.Year)) as workson6
) as workson7
```

쿼리문을 보면 알 수 있지만, workson6는 workson2, workson3의 LEFT JOIN으로 나온 view와 workson4, workson5의 RIGHT JOIN으로 나온 view의 UNION형태임을 알 수 있다. 이 후 workson7은 이 view에서 다시 3개의 column을 select 한 view이다. 먼저 workson6에 대해 설명하겠다. workson2와 workson4는 같은 view이고 workson3와 workson5는 같은 view이므로 workson2, workson3에 대해서만 살펴본 후 LEFT JOIN과 RIGHT JOIN, UNION에 대해서 살펴보겠다.

```
SELECT year(created) as Year, COUNT(*) as NewGroup
FROM gatherings
GROUP BY (created) as workson2
```

먼저 workson2에 대한 내용이다. 위 형태의 query문은 상당히 기본적인 구조이다. gatherings table의 created column의 year값(year()은 Timestamp타입에서 year값만 뽑아오는 함수이다.)을 기준으로 묶고, year 값의 연도별 행의 개수를 카운트하여 연도별 새로이 만들어진 그룹의 수를 의미하는 NewGroup값을 가져온 view를 의미한다.

```
SELECT year(time) as Year, COUNT(*) as NewMeeting
FROM meetings
GROUP BY year(time)) as workson3
```

위의 쿼리문은 workson3에 대한 내용이다. Workson3는 meetings 테이블에서 time column의 year값을 기준으로 묶고, year값의 연도별 행의 개수를 카운트하여 연도별 개최된 모임의 수를 의미하는 NewMeeting값을 가져온 view를 의미한다.

이제 workson2와 workson3를 이해하였으니, LEFT JOIN과 RIGHT JOIN형태를 살펴본다. 여기서 LEFT, RIGHT JOIN은 두 view를 JOIN 할 때 JOIN 순서가 반대인 관계이므로 LEFT JOIN만 설명하려고 한다. 다시 LEFT JOIN 쿼리를 보면 다음과 같다.

```
(SELECT workson2.Year, NewGroup, NewMeeting
```

```
FROM    (SELECT year(created) as Year, COUNT(*) as NewGroup
         FROM gatherings
         GROUP BY (created)) as workson2
LEFT JOIN (SELECT year(time) as Year, COUNT(*) as NewMeeting
         FROM meetings
         GROUP BY year(time)) as workson3
ON workson2.Year = workson3.Year)
```

여기서 LEFT JOIN은 두 개의 view나 table을 JOIN 할 때 먼저 나온 view나 table의 데이터를 유실하지 않고 JOIN하는 방식이다. 위의 쿼리문에서 NewGroup의 Year값은 2002-2017까지 있지만 NewMeeting의 Year값은 2017, 2018 두 가지가 있다. 여기서 Year을 기준으로 NewGroup에 대한 view가 먼저 나오고 NewMeeting에 대한 view와 LEFT JOIN했으므로 NewGroup Year값의 2002-2017까지의 데이터는 유실되지 않고 NewMeeting의 2002-2016까지의 값이 None으로, 2017의 값은 그대로 채워지는 동시에 2018의 값은 유실된다. 이 두 view를 RIGHT JOIN하면 반대의 연산이 수행된다.

이렇게 LEFT JOIN, RIGHT JOIN을 모두 수행한 후 두 view를 UNION하는 이유를 설명하려고 한다. NewGroup과 NewMeeting의 view를 살펴보면 어떤 연도에는 오프라인 모임이 개최되지 않을 수 있지만 신규가입자는 있을 수 있다. 또 어떤 연도에는 오프라인 모임이 개최되지만 신규가입자가 없을 수 있다. 여기서 INNER JOIN을 사용하면 두 view에 모두 데이터가 존재하지 않는 연도의 데이터는 사라지기 때문에 적절한 형태의 JOIN이 아니다. 맥락을 살펴보았을 때 적절한 형태의 JOIN은 OUTER JOIN의 형태이지만, MySQL에는 OUTER JOIN을 지원하지 않기 때문에 LEFT JOIN과 RIGHT JOIN을 모두 수행한 후 두 view를 UNION하는 방식으로 구현해야 한다.

지금까지 workson6 view 에 대해 설명했고 이후 workson7에 대해 간단하게 보면 다음의 쿼리와 같다. Workson6에서 Year, NewGroup, NewMeeting column을 가져온 view이다.

```
SELECT Year, NewGroup, NewMeeting
FROM workson6 as workson7
```

이 후 workson1과 workson7을 RIGHT JOIN 하면 workson7의 2002-2018연도를 기준으로 workson1의 NewUser의 데이터가 JOIN된다. NewUser의 데이터는 2002-2017까지의 데이터가 있으므로 LEFT, RIGHT JOIN을 UNION해서 OUTER JOIN을 하는 것 보다 workson7 방향으로 한 번만 JOIN 해주는 편이 더 효율적이다. 이를 토대로 다시 R2의 전체 SQL문을 보면 아래와 같다.

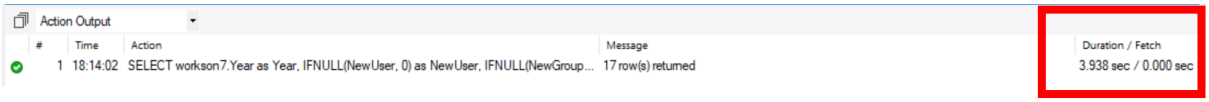
```
SELECT workson7.Year as Year,
IFNULL(NewUser, 0) as NewUser,
IFNULL(NewGroup, 0) as NewGroup,
IFNULL(NewMeeting, 0) as NewMeeting
FROM workson1
RIGHT JOIN workson7
ON workson1.Year = workson7.Year
ORDER BY Year;
```

추가적으로 NewUser, NewGroup, NewMeeting 을 IFNULL(column name, 0)이라고 표현한 이유는 , 특정 연도에 대해서 column의 데이터가 없는 경우 None으로 출력이 되게 되는데, 수를 count하라는 requirement의 의미에 맞게 데이터를 출력하기 위해서 Null일 경우에 0으로 바꾸는 코드를 추가하였다.

2.2.1.3. 실행 결과

	Year	NewUser	NewGroup	NewMeeting
▶	2002	69	52	0
	2003	2150	80	0
	2004	1785	25	0
	2005	2529	36	0
	2006	10819	207	0
	2007	26265	280	0
	2008	41665	310	0
	2009	71118	415	0
	2010	91002	431	0
	2011	139059	573	0
	2012	254796	823	0
	2013	393928	989	0
	2014	556617	1292	0
	2015	1196683	1631	0
	2016	1514295	2191	0
	2017	1591087	3212	1446
	2018	0	0	4102

workbench에서 해당 MySQL구문을 돌렸을 때의 결과값이다.



결과를 살펴보면 2002년부터 2017년까지 가입자수와 그룹수는 폭발적으로 증가하였다. 오프라

인 미팅의 경우 2002년부터 2016년까지는 한 개도 이루어지지 않고 있다가 2017, 2018년도에만 미팅이 이루어진 것을 확인할 수 있다.

2.2.2. R3 : 호평을 받았지만 오프라인 개최 회수가 적은 만남 장소 홍보

R3에서는 호평을 받았지만 오프라인 모임 개최 횟수가 적은 만남 장소를 소개하는 이벤트를 하기 위해 5회 이상 리뷰를 받고 평균 평점 4.5 이상이지만 오프라인 모임이 3회 이하인 만남 장소들을 찾아서 장소 id, 도시 id, 장소명, 주소를 출력하기를 요구하고 있다. 출력된 결과는 id에 따라 오름차순으로 정렬되어야 하며, column의 순서는 id, 도시 id, 장소명, 주소인 제약조건을 만족시켜야한다.

Cursor 객체를 사용하고 버퍼 용량을 늘려주는 것은 위의 R2부분과 동일하므로 생략하고 바로 SQL 쿼리문 설명으로 넘어가겠다.

2.2.2.1. MySQL Query

```
SELECT MP.id as id, city_id, name, address
FROM meeting_places as MP, (SELECT view1.id as id
        FROM (SELECT id
            FROM meeting_places
            WHERE number_of_rating>=5 AND average_rating>=4.5) as view1
        LEFT JOIN (SELECT meeting_place_id as id
            FROM meetings
            GROUP BY meeting_place_id
            HAVING COUNT(*)>3) as view2
        ON view1.id=view2.id
        WHERE view2.id is NULL) as view3
WHERE MP.id=view3.id
ORDER BY id;
```

2.2.2.2. 코드 설명

R3를 만족시키는 MySQL Query 문은 위와 같다. 위의 query문 또한 복잡한 nested 구조로 되어 있기 때문에 이해를 편하게 하기 위해서 아래와 같이 간소화해보았다.

SELECT MP.id as id, city_id, name, address
FROM meeting_places as MP, view3
WHERE MP.id=view3.id
ORDER BY id

위의 쿼리문을 이해하기 위해서는 view3에 대한 이해가 필요하다. View 3는 다음과 같이 쿼리문으로 표현된다.

```
SELECT view1.id as id
FROM    (SELECT id
         FROM meeting_places
         WHERE number_of_rating>=5 AND average_rating>=4.5) as view1
LEFT JOIN (SELECT meeting_place_id as id
         FROM meetings
         GROUP BY meeting_place_id
         HAVING COUNT(*)>3) as view2
ON view1.id=view2.id
WHERE view2.id is NULL
```

위의 view 3 또한 nested 형태로 되어있기 때문에 이를 다시 한번 간소화해보면 다음과 같이 표시할 수 있다.

```
SELECT view1.id as id
FROM view1
LEFT JOIN view2
ON view1.id=view2.id
WHERE view2.id is NULL
```

View3를 이해하기 위해서는 마찬가지로 view 1과 view2에 대한 이해가 필요하다.

```
SELECT id
FROM meeting_places
WHERE number_of_rating>=5 AND average_rating>=4.5
```

위의 sql 쿼리는 view1에 대한 것으로, meeting_places 테이블에서 리뷰 수가 5개 이상이고 평균 평점이 4.5점 이상인 만남 장소의 id를 가져온 것이다.

```
SELECT meeting_place_id as id
FROM meetings
```

```
GROUP BY meeting_place_id
HAVING COUNT(*)>3
```

위의 sql 쿼리는 view2에 대한 것으로 meetings 테이블에서 meeting place를 기준으로 묶어서 3개 초과의 값을 가진 meeting place의 id를 가져온 것이다.

View1과 view2에 대한 이해를 바탕으로 view3를 다시 이해해보면

```
SELECT view1.id as id
FROM view1
LEFT JOIN view2
ON view1.id=view2.id
WHERE view2.id is NULL
```

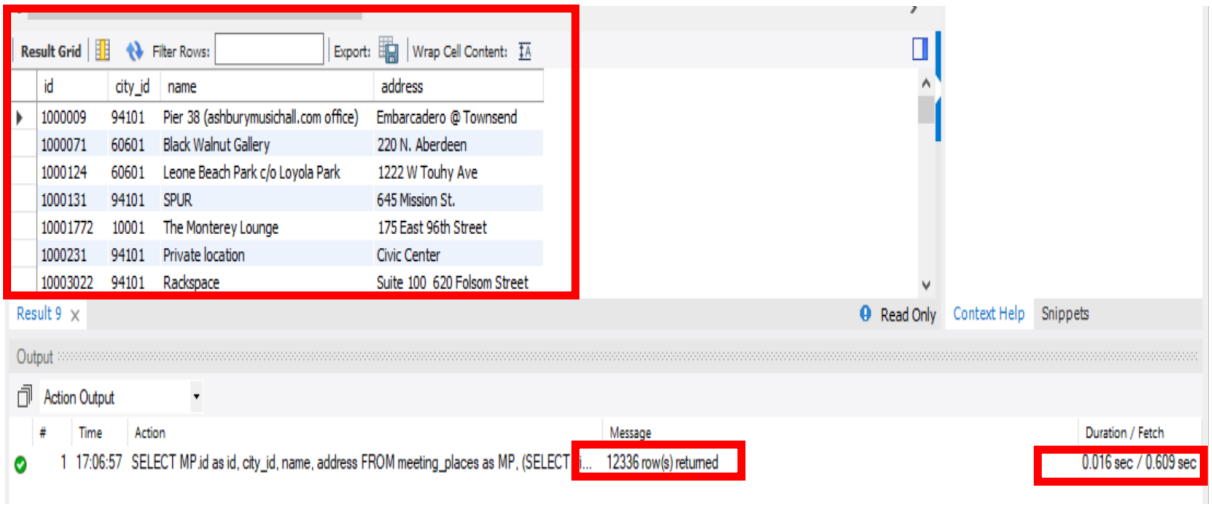
View 3는 View1과 view2를 meeting_place id가 같은 것끼리 left join하고, view2.id가 NULL인 부분에서 view1.id를 뽑아내었다. 즉, view1-view2(차집합)한 것이다. View1에서 만남이 3개 초과인 meeting place의 id들을 제거해 주었으므로 즉 view3는 리뷰 수가 5개 이상이고 평균 평점이 4.5 이상이며, 만남이 3번 이하로 이루어진 만남장소의 id를 가져온다.

최종적으로 간소화한 최종 SQL문을 살펴보면

```
SELECT MP.id as id, city_id, name, address
FROM meeting_places as MP, view3
WHERE MP.id=view3.id
ORDER BY id
```

Id 를 기준으로 meeting_places 테이블과 위에서 정의한 view3 를 inner join하여 만남장소의 id, 도시 id, 장소명, 주소를 출력함으로써 원하는 정보를 얻을 수 있게 되었다.

2.2.2.3. 실행 결과



쿼리문은 위와 같이 실행되었으며, 결과 형태는 위와 같다.

성공적으로 제약조건을 만족시키는 12336개의 만남 장소를 출력할 수 있게 되었다.

2.2.3. R4 : 활동을 많이 하는 가입자들

R4에서는 활동을 많이 하는 가입자들을 찾아 그들의 id, 사는 도시 id, 이름을 출력할 것을 요구하고 있다. '활동을 많이 하는'의 기준을 1년이상 활동한 그룹이 5개 이상인 사람으로 정했고, 활동기간은 마지막 방문 날짜와 가입한 날짜 차이로 정의하였다. Id에 따라 오름차순 정렬과 column 순서는 id, 도시 id, 이름 순서를 지켜야 한다.

2.2.3.1. MySQL Query

```
1 ●  SELECT id, city_id, name
2    FROM members as m
3    WHERE m.id IN (SELECT member_id
4        FROM member_gathering
5        WHERE TIMESTAMPDIFF(YEAR, joined, visited) >= 1
6        GROUP BY member_id
7        HAVING COUNT(gathering_id) >= 5)
8    ORDER BY id;
```

R4는 위와 같은 한번의 nested 구조를 통해서 SQL 문을 작성할 수 있다.

2.2.3.2. 코드 설명

R4는 비교적 간단하기 때문에 바로 Where문에 걸려있는 nested 구조를 설명하고 전체 구조를 살펴보고자 한다.

SELECT member_id
FROM member_gathering
WHERE TIMESTAMPDIFF(YEAR, joined, visited) >= 1
GROUP BY member_id
HAVING COUNT(gathering_id) >= 5
ORDER BY id;

위의 sql문은 nested 된 안쪽 구문의 쿼리문으로 member_gathering 테이블에서 member id를 기준으로 묶었을 때 gathering id가 5개 이상 나오는 것들 중에서 가입했을 때의 시간과 마지막 방문 했을 때의 시간의 차이가 1년 이상인 것의 member id를 골라오는 것이다.

WHERE문을 살펴보면 TIMESTAMPDIFF(YEAR, joined, visited) >=1 이 조금 생소할 수 있는데, TIMESTAMPDIFF는 timestamp 데이터 타입끼리의 차이를 계산하는 것으로 첫 번째 parameter는 어떤 단위를 기준으로 차이를 계산할지를 결정한다. 1년 이상 차이가 나는 것을 골라야 하므로 연도를 기준으로 하여 YEAR을 넣어주었다.

위의 sql문을 view1이라고 한다면 바깥 쪽 구문은 아래와 같은 형태가 된다.

SELECT id, city_id, name
FROM members as m
WHERE m.id IN view1

위 구문은 members 테이블에서 view1의 member_id와 겹치는 가입자의 id와 그 가입자가 사는 도시의 id, 가입자의 이름을 가져오는 것을 의미한다.

2.2.3.3. 실행 결과

위의 구문을 사용하여 가져온 값들의 결과값은 위와 같다. 1년이상 5개이상의 그룹에서 활동을 한 활동량이 많은 고객은 총 59320명이다.

2.2.4. R5 : 등록은 되어 있지만 그룹원이 하나도 없는 그룹들 정리

R5에서는 등록은 되어 있지만 그룹원이 하나도 없는 그룹들을 2014년 이전에 생성된 그룹은 삭제하고 2014년 이후에 생성된 그룹은 홍보하는 식으로 정리하고자 한다. 이를 위해 R1)에서 생성한 table인 gatherings에 데이터형은 VARCHAR(20)이며 제약 사항은 없는 remarks라는 새로운 column을 추가하고 이 remarks에 만약 그룹원이 한 명이라도 있으면 null, 2014년 이전에 생성된 그룹은 "to be deleted"를, 2014년 이후에 생성된 그룹에는 "need promotion"을 저장하기를 요구하고 있다. 또한 그룹원이 없는 그룹들을 찾아 그룹의 id, 그룹명, 생성 날짜, remarks를 출력하기를 요구하고 있다. 출력된 결과는 id에 따라 오름차순으로 정렬되어야 하며, column의 순서는 id, 그룹명, 생성 날짜, remarks여야 한다는 제약조건을 만족해야 한다.

2.2.4.1. MySQL Query

```
1  ALTER TABLE gatherings ADD remarks VARCHAR(20)
```

```
1  UPDATE gatherings SET remarks = CASE
2  WHEN year(created)>=2014 AND id NOT IN (SELECT DISTINCT gathering_id
3      FROM member_gathering)
4      THEN "need promotion"
5  WHEN year(created)<2014 AND id NOT IN (SELECT DISTINCT gathering_id
6      FROM member_gathering)
7      THEN "to be deleted" END;
```

```
1  SELECT id, name, created, remarks
2  FROM gatherings
3  WHERE id NOT IN (SELECT DISTINCT gathering_id
4      FROM member_gathering)
5  ORDER BY id
```

R5는 최대 3개의 SQL 문장으로 작성되어야 한다는 제약에 따라 위와 같이 3개의 문장으로 SQL 문을 작성할 수 있다.

2.2.4.2. 코드 설명

R5는 최대 3개의 SQL 문장으로 작성되어야 하기 때문에 순서대로

1) remarks라는 새로운 column을 추가하는 문장
2) remarks에 값을 update하는 문장
3) 그룹원이 없는 그룹들을 찾아 원하는 결과를 출력하는 문장

으로 나뉘어진다.

ALTER TABLE gatherings ADD remarks VARCHAR(20)

첫 번째로, 위의 SQL문은 gatherings table에 remarks라는 새로운 column을 추가한다. 여기서 데이터형은 주어진 대로 VARCHAR(20)이다.

```
UPDATE gatherings SET remarks = CASE
WHEN year(created)>=2014 AND id NOT IN (SELECT DISTINCT gathering_id
        FROM member_gathering)
THEN "need promotion"
WHEN year(created)<2014 AND id NOT IN (SELECT DISTINCT gathering_id
        FROM member_gathering)
THEN "to be deleted" END;
```

두 번째 SQL문은 remarks에 값을 저장하는 문장이다. 두 개의 WHEN문에 걸려 있는 nested 구조를 설명하고 전체 구조를 살펴보고자 한다.

```
SELECT DISTINCT gathering_id
FROM member_gathering
```

위의 SQL문은 nested된 안쪽 구문의 쿼리문으로 member_gathering 테이블에서 gathering_id를 골라온 것이다. 이때 DISTINCT를 이용하여 중복된 값들을 제거함으로써 그룹원이 한 명이라도 있는 gathering_id만을 골라왔다.

위의 sql문을 view1이라고 한다면 바깥 쪽 구문은 아래와 같은 형태가 된다.

```
UPDATE gatherings SET remarks=CASE
WHEN year(created)>=2014 AND id NOT IN view1 THEN “need promotion”
WHEN year(created)<2014 AND id NOT IN view1 THEN “to be deleted” END
```

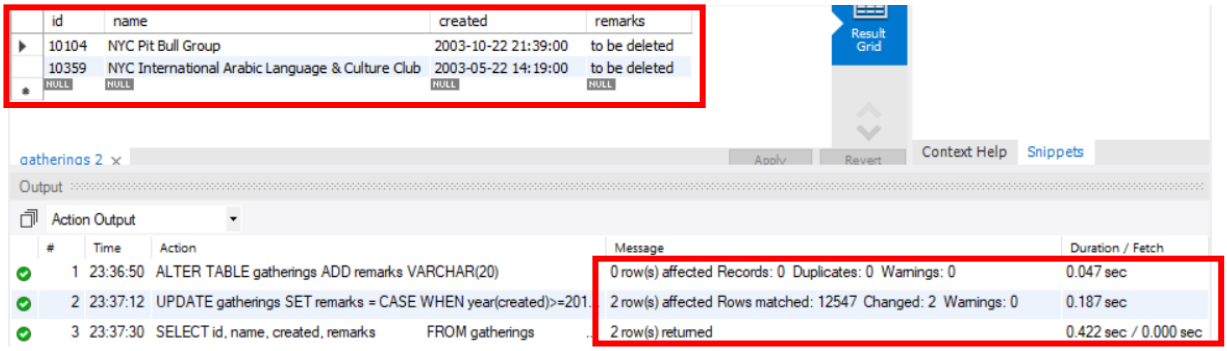
위의 구문은 최종적으로 요구하는 remarks에 값을 저장하게 된다. 첫 번째 WHEN문은 그룹원이 한 명이라도 있는 gathering_id가 아니면서 2014년 이후에 생성된 그룹의 remarks에 “need promotion”을, 두 번째 WHEN문은 마찬가지로 그룹원이 한 명이라도 있는 gathering_id가 아니면서 2014년 이전에 생성된 그룹의 remarks에 “to be deleted”를 저장하게 된다.

```
SELECT id, name, created, remarks
FROM gatherings
```

```
WHERE id NOT IN (SELECT DISTINCT gathering_id
                 FROM member_gathering)
ORDER BY id
```

마지막으로 SQL문은 요구한 결과값을 출력하는 문장이다. 앞에 두번째 쿼리문에서 한 과정과 똑같은 방법으로 그룹원이 한 명이라도 있는 gathering_id를 nested된 구문으로 골라내고 gathering 테이블에서 WHERE문으로 그룹원이 한 명이라도 있는 gathering_id가 아닌 id를 골라내었다. 이를 id에 따라 오름차순으로 정렬하고 그룹의 id, 그룹명, 생성 날짜, remarks를 순서대로 출력함으로써 원하는 정보를 얻을 수 있게 되었다.

2.2.4.3. 실행결과



2.3. R6 정의 및 쿼리 설명

2.3.1. 문제 정의

"사이트 A는 현재 가입자들이 어떤 주제에 관심이 많은지, 어떤 주제에서 양질의 교류가 이루어지고 있는지, 관심사에 대한 거시적인 정보를 파악하기 위해 가입자들에게 좋은 평가를 받은 그룹이 많이 있는 주제를 알아보고자 한다. 이를 위해 주제 별로 평균 평점 4.8이상을 받은 그룹의 개수를 구해 주제 이름과 평균 평점이 4.8이상인 그룹의 개수를 출력하라. 출력된 결과는 그룹의 개수를 기준으로 내림차순으로 정렬되어야 하며 column 순서는 주제 이름, 그룹의 개수여야 한다."

어떤 주제가 평가가 좋은지를 찾기 위해서 우선 그 주제를 가지는 그룹들을 찾았다. 그리고 그 그룹들의 평균 평점이 높다면 그 그룹의 주제도 평가가 좋을 것이라 생각했다. 평균 평점 4.8은 주어진 데이터 값을 보고 임의로 적절한 수치를 정한 것이다.

2.3.2. MySQL Query

```
1 •  SELECT T.name, COUNT(g.id) as total
2    FROM gatherings as G, topics as T
3    WHERE G.average_rating >= 4.8 AND G.topic_id = T.id
4    GROUP BY T.id
5    ORDER BY total DESC
```

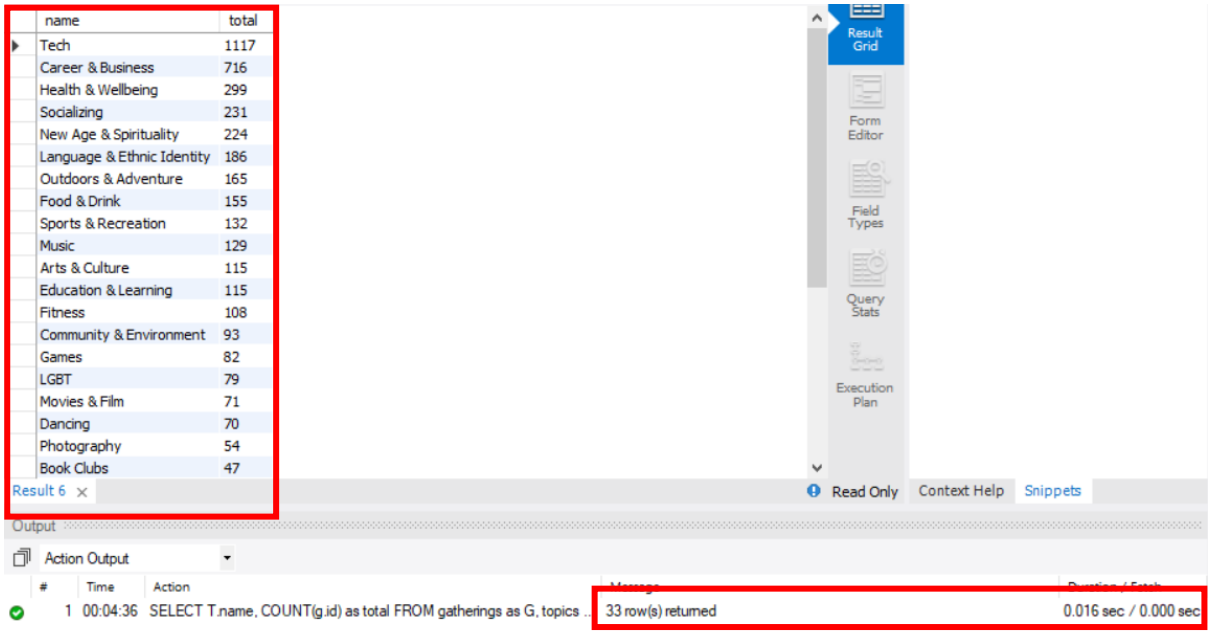
2.3.3. 코드 설명

SELECT T.name, COUNT(g.id) as total
FROM gatherings as G, topics as T
WHERE G.average_rating >= 4.8 AND G.topic_id = T.id
GROUP BY T.id
ORDER BY total DESC;

SQL구문은 비교적 간단하다. 코드가 길어지는 것을 막기 위해 gatherings를 G, topics를 T로 이름 지었다. G와 T 테이블에서 topic_id를 기준으로 inner join을 하되, gatherings table에서의 average_rating 값이 4.8 이상인 것들을 골라왔다. 그리고 이를 T의 id로 그룹화하여 주제 이름과 그 개수를 total이라는 이름의 column으로 출력하여 얻고자 하는 정보를 얻을 수 있게 되었다.

그리고, 어느 주제에 평점이 높은 그룹이 많이 분포되어있는지 확인하기 위해 ORDER BY total DESC 코드를 추가하였다.

2.3.4. 실행결과



name	total
Tech	1117
Career & Business	716
Health & Wellbeing	299
Socializing	231
New Age & Spirituality	224
Language & Ethnic Identity	186
Outdoors & Adventure	165
Food & Drink	155
Sports & Recreation	132
Music	129
Arts & Culture	115
Education & Learning	115
Fitness	108
Community & Environment	93
Games	82
LGBT	79
Movies & Film	71
Dancing	70
Photography	54
Book Clubs	47

실행결과를 살펴보니, Tech 주제에 관하여 평점이 높은 그룹들이 많이 생성되어있음을 알 수 있었다. 그 뒤로 Career &Business, Health & Wellbeing 주제가 따라오고 있음을 알 수 있다.

2.4. R2~R6 결과 출력 코드

```
fopen = open('project2_team%02d_req6.txt' % team, 'w', encoding='utf8')

rows = cursor.fetchall()
for line in rows:
    for i in range(len(line)):
        fopen.write(str(line[i]))
        if i < len(line) - 1: fopen.write(';')
    if line != rows[len(rows) - 1]: fopen.write('\n')

fopen.close()
cursor.close()
```

R2~R6 모두 위의 코드로 결과를 출력하였다.

각각 요구 사항대로 SQL문을 활용하여 cursor에 저장한 모든 데이터들을 fetchall를 이용하여 가져왔다. 이를 반복문을 통해 한 줄씩 읽으며 txt파일에 추가했다. 이 때 line을 입력할 때 column 사이마다 사이에 ;를, 한 줄 입력을 마치고 다음 줄로 넘어갈 때는 줄 바꿈을 해주었다.

마지막 attribute 값을 출력한 후에는 ;를 출력하지 않도록 i<len(line)-1일 때만 ;를 출력했고, 마지막 열을 출력한 후에는 줄바꿈이 일어나지 않도록 했다. 마지막 열을 출력한 후에는 줄바꿈이 일어나지 않도록 Line != rows[len(rows) – 1]을 통해 마지막 열과 다를 때만 줄바꿈을 하게 했는데, 중간에 마지막 열과 같은 열이 생기면 줄바꿈을 안 할 수도 있지만 주어진 문제에서는 이러한 일이 안 생기기 때문에 열마다 줄바꿈을 해주며 원하는 결과값을 출력해낼 수 있었다.

다만, duplicate tuple을 자동으로 제거해 주지 않는 SQL의 특성을 고려한다면 작성한 query이외의 모든 query에 보편적으로 적용되기는 힘들 것이다.

3. 결론

본 프로젝트는 Project Requirement 제약 하에서 R1을 통해 사이트 A의 데이터베이스 스키마를 설계한 후 DB 테이블을 생성하고, R2~R5에서는 주어진 요구사항에 따라 원하는 정보를 출력하며, 마지막으로 R6에서는 자유주제로 새로운 문제를 정의하고 이에 따른 결과를 출력하는 것이 목적이었다. 그 목적에 맞춰 R1에서는 데이터베이스와 테이블을 구축하여 주어진 csv 파일을 읽어 데이터를 삽입하고 마지막으로 외래키를 설정을 해주었다. R2~R6에서는 주어진 요구사항에 맞춰 원하는 결과값을 출력해냈다. 결과적으로 본 프로젝트의 목적을 성공적으로 달성했다.