

SQL을 이용한 데이터 분석

5. 트랜잭션과 기타 고급 SQL 문법

한기용

keeyonghan@hotmail.com

Contents

1. 4일차 숙제 리뷰
2. 트랜잭션 소개와 실습
3. 기타 고급 문법 소개와 실습
4. 맺음말

4일차 숙제 리뷰

사용자별 처음과 마지막 채널 찾기

Gross revenue가 가장 큰 사용자 ID 10개 찾기

월별 NPS 계산하기

속제 1: 사용자별로 처음 채널과 마지막 채널 알아내기

- ROW_NUMBER vs. FIRST_VALUE/LAST_VALUE
- 사용자 251번의 시간순으로 봤을 때 첫 번째 채널과 마지막 채널은 무엇인가?
 - 노가다를 하자면 아래 쿼리를 실행해서 처음과 마지막 채널을 보면 된다.

```
SELECT ts, channel
FROM raw_data.user_session_channel usc
JOIN raw_data.session_timestamp st ON usc.sessionid = st.sessionid
WHERE userid = 251
ORDER BY 1
```

- ROW_NUMBER를 이용해서 해보자
 - ROW_NUMBER() OVER (PARTITION BY field1 ORDER BY field2) nn

속제 1 해설 (1) - CTE를 빌딩블록으로 사용

WITH first AS (

SELECT userid, ts, channel, ROW_NUMBER() OVER(PARTITION BY userid ORDER BY ts) seq

FROM raw_data.user_session_channel usc

JOIN raw_data.session_timestamp st ON usc.sessionid = st.sessionid

), last AS (

SELECT userid, ts, channel, ROW_NUMBER() OVER(PARTITION BY userid ORDER BY ts DESC) seq

FROM raw_data.user_session_channel usc

JOIN raw_data.session_timestamp st ON usc.sessionid = st.sessionid

)

SELECT first.userid AS userid, first.channel AS first_channel, last.channel AS last_channel

FROM first

JOIN last ON first.userid = last.userid and last.seq = 1

WHERE first.seq = 1;

속제 1 해설 (2) - JOIN 방식

```
SELECT first.userid AS userid, first.channel AS first_channel, last.channel AS last_channel
FROM (
  SELECT userid, ts, channel, ROW_NUMBER() OVER(PARTITION BY userid ORDER BY ts) seq
  FROM raw_data.user_session_channel usc
  JOIN raw_data.session_timestamp st ON usc.sessionid = st.sessionid
) first
JOIN (
  SELECT userid, ts, channel, ROW_NUMBER() OVER(PARTITION BY userid ORDER BY ts DESC) seq
  FROM raw_data.user_session_channel usc
  JOIN raw_data.session_timestamp st ON usc.sessionid = st.sessionid
) last ON first.userid = last.userid and last.seq = 1
WHERE first.seq = 1;
```

속제 1 해설 (3) - GROUP BY 방식

```
SELECT userid,
```

```
MAX(CASE WHEN rn1 = 1 THEN channel END) first_touch,  
MAX(CASE WHEN rn2 = 1 THEN channel END) last_touch
```

```
FROM (
```

```
    SELECT userid,
```

```
        channel,
```

```
        (ROW_NUMBER() OVER (PARTITION BY usc.userid ORDER BY st.ts asc)) AS rn1,
```

```
        (ROW_NUMBER() OVER (PARTITION BY usc.userid ORDER BY st.ts desc)) AS rn2
```

```
    FROM raw_data.user_session_channel usc
```

```
    JOIN raw_data.session_timestamp st ON usc.sessionid = st.sessionid
```

```
)
```

```
GROUP BY 1;
```

속제 1 해설 (4) - FIRST_VALUE/LAST_VALUE

SELECT **DISTINCT**

A.userid,

FIRST_VALUE(A.channel) over(partition by A.userid order by B.ts

rows between unbounded preceding and unbounded following) AS First_Channel,

LAST_VALUE(A.channel) over(partition by A.userid order by B.ts

rows between unbounded preceding and unbounded following) AS Last_Channel

FROM raw_data.user_session_channel A

LEFT JOIN raw_data.session_timestamp B ON A.sessionid = B.sessionid;

속제 2: Gross Revenue가 가장 큰 UserID 10개 찾기

- user_session_channel과 session_transaction과 session_timestamp 테이블을 사용
- Gross revenue: Refund 포함한 매출

○

속제 2 해설 (1): GROUP BY

```
SELECT
    userID,
    SUM(amount)
FROM raw_data.session_transaction st
LEFT JOIN raw_data.user_session_channel usc ON st.sessionid = usc.sessionid
GROUP BY 1
ORDER BY 2 DESC
LIMIT 10;
```

속제 2 해설 (2) SUM OVER

```
SELECT DISTINCT
    usc.userid,
    SUM(amount) OVER(PARTITION BY usc.userid)
FROM raw_data.user_session_channel AS usc
JOIN raw_data.session_transaction AS revenue ON revenue.sessionid = usc.sessionid
ORDER BY 2 DESC
LIMIT 10;
```

속제 3: raw_data.nps 테이블을 바탕으로 월별 NPS 계산

- 고객들이 0 (의향 없음) 에서 10 (의향 아주 높음)
 - **detractor (비추천자)** : 0 에서 6
 - **passive (소극자)** : 7이나 8점
 - **promoter (홍보자)** : 9나 10점
- $NPS = \text{promoter 퍼센트} - \text{detractor 퍼센트}$
- 10점 만점으로 '주변에 추천하겠는가?'라는 질문을 기반으로 고객 만족도를 계산
 - 10, 9점 추천하겠다는 고객(promoter)의 비율에서 0-6점의 불평고객(detractor)의 비율을 뺀 것이 NPS

속제 3 해설 (1)

```
SELECT month,  
  ROUND((promoters-detractors)::float/total_count*100, 2) AS overall_nps  
FROM (  
  SELECT LEFT(created, 7) AS month,  
    COUNT(CASE WHEN score >= 9 THEN 1 END) AS promoters,  
    COUNT(CASE WHEN score <= 6 THEN 1 END) AS detractors,  
    COUNT(CASE WHEN score > 6 AND score < 9 THEN 1 END) AS passives,  
    COUNT(1) AS total_count  
  FROM raw_data.nps  
  GROUP BY 1  
  ORDER BY 1  
);
```

2019-01	2.36
2019-02	30.54
2019-03	52.91
2019-04	53
2019-05	54.52
2019-06	65.02
2019-07	64.51
2019-08	67.71
2019-09	37.95
2019-10	53.29
2019-11	61.29
2019-12	65.99

속제 3 해설 (2)

```
SELECT LEFT(created, 7) AS month,  
       ROUND(SUM(CASE  
         WHEN score >= 9 THEN 1  
         WHEN score <= 6 THEN -1 END)::float*100/COUNT(1), 2)  
FROM raw_data.nps  
GROUP BY 1  
ORDER BY 1;
```

2019-01	2.36
2019-02	30.54
2019-03	52.91
2019-04	53
2019-05	54.52
2019-06	65.02
2019-07	64.51
2019-08	67.71
2019-09	37.95
2019-10	53.29
2019-11	61.29
2019-12	65.99

◆ 숙제 실습

❖ 구글 Colab 실습 링크:

- <https://colab.research.google.com/drive/1bbNuJbsk24IXoMRTRkHI-vvteoKVjsyh>

트랜잭션 소개와 실습

트랜잭션이란? (1)

- Atomic하게 실행되어야 하는 SQL들을 묶어서 하나의 작업처럼 처리하는 방법
 - 이는 DDL이나 DML 중 레코드를 수정/추가/삭제한 것에만 의미가 있음.
 - SELECT에는 트랜잭션을 사용할 이유가 없음
 - BEGIN과 END 혹은 BEGIN과 COMMIT 사이에 해당 SQL들을 사용
 - ROLLBACK
- 은행 계좌 이체가 아주 좋은 예
 - 계좌 이체: 인출과 입금의 두 과정으로 이뤄짐
 - 만일 인출은 성공했는데 입금이 실패한다면?
 - 이 두 과정은 동시에 성공하던지 실패해야함 -> Atomic하다는 의미
 - 이런 과정들을 트랜잭션으로 묶어주어야함
 - 조회만 한다면 이는 트랜잭션으로 묶을 이유가 없음

트랜잭션이란? (2)

BEGIN;

A의 계좌로부터 인출;
B의 계좌로 입금;

END;

이 명령어들은 마치 하나의
명령어처럼 처리됨. 다
성공하던지 다 실패하던지 둘중의
하나가 됨

- END와 COMMIT은 동일
- 만일 BEGIN 전의 상태로 돌아가고 싶다면 ROLLBACK 실행
- 이 동작은 commit mode에 따라 달라짐!

트랜잭션 커밋 모드: autocommit

- autocommit = True

- 모든 레코드 수정/삭제/추가 작업이 기본적으로 바로 데이터베이스에 쓰여짐. 이를 커밋(Commit)된다고 함.
- 만일 특정 작업을 트랜잭션으로 묶고 싶다면 BEGIN과 END(COMMIT)/ROLLBACK으로 처리

- autocommit = False

- 모든 레코드 수정/삭제/추가 작업이 COMMIT 호출될 때까지 커밋되지 않음

트랜잭션 방식

- Google Colab의 트랜잭션
 - 기본적으로 모든 SQL statement가 바로 커밋됨 (autocommit=True)
 - 이를 바꾸고 싶다면 BEGIN;END; 혹은 BEGIN;COMMIT을 사용 (혹은 ROLLBACK;)
- psycopg2의 트랜잭션
 - autocommit이라는 파라미터로 조절가능
 - autocommit=True가 되면 기본적으로 PostgreSQL의 커밋 모드와 동일
 - autocommit=False가 되면 커넥션 객체의 .commit()과 .rollback()함수로 트랜잭션 조절 가능
 - 무엇을 사용할지는 개인 취향

DELETE FROM vs. TRUNCATE

- DELETE FROM table_name (not DELETE * FROM)
 - 테이블에서 모든 레코드를 삭제
 - vs. DROP TABLE table_name
 - WHERE 사용해 특정 레코드만 삭제 가능:
 - DELETE FROM raw_data.user_session_channel WHERE channel = 'Google'
- TRUNCATE table_name도 테이블에서 모든 레코드를 삭제
 - DELETE FROM은 속도가 느림
 - TRUNCATE이 전체 테이블의 내용 삭제시에는 여러모로 유리
 - 하지만 두가지 단점이 존재
 - TRUNCATE는 WHERE을 지원하지 않음
 - TRUNCATE는 Transaction을 지원하지 않음

◆ 트랜잭션 실습

❖ 구글 Colab 실습 링크:

- <https://colab.research.google.com/drive/1bbNuJbsk24IXoMRTRkHI-vvteoKVjsyh#scrollTo=hvCDVS6FeEbz>

기타 고급 문법 소개와 실습

◆ 알아두면 유용한 SQL 문법들

- ❖ UNION, EXCEPT, INTERSECT

- ❖ COALESCE, NULLIF

- ❖ LISTAGG

- ❖ LAG

- ❖ WINDOW 함수

 - ROW_NUMBER OVER

 - SUM OVER

 - FIRST_VALUE, LAST_VALUE

- ❖ JSON Parsing 함수

UNION, EXCEPT, INTERSECT

- UNION (합 집합)
 - 여러개의 테이블들이나 SELECT 결과를 하나의 결과로 합쳐줌
 - UNION vs. UNION ALL
 - UNION은 중복을 제거
- EXCEPT (MINUS)
 - 하나의 SELECT 결과에서 다른 SELECT 결과를 빼주는 것이 가능
- INTERSECT (교 집합)
 - 여러 개의 SELECT문에서 같은 레코드들만 찾아줌

COALESCE, NULLIF

- COALESCE(Expression1, Expression2, ...):
 - 첫번째 Expression부터 값이 NULL이 아닌 것이 나오면 그 값을 리턴하고 모두 NULL이면 NULL을 리턴한다.
 - NULL값을 다른 값으로 바꾸고 싶을 때 사용한다.
- NULLIF(Expression1, Expression2):
 - Expression1과 Expression2의 값이 같으면 NULL을 리턴한다

LISTAGG (1)

- GROUP BY에서 사용되는 Aggregate 함수 중의 하나
- 사용자 ID별로 채널을 순서대로 리스트:

SELECT

userid,

LISTAGG(channel) WITHIN GROUP (ORDER BY ts) channels

FROM raw_data.user_session_channel usc

JOIN raw_data.session_timestamp st ON usc.sessionid = st.sessionid

GROUP BY 1

LIMIT 10;

68 YoutubeGoogleInstagramYoutubeInstagramInstagramInstagramOrganicInstagramYoutube...

LISTAGG (2)

SELECT

userid,

LISTAGG(channel, '->') WITHIN GROUP (ORDER BY ts) channels

FROM raw_data.user_session_channel usc

JOIN raw_data.session_timestamp st ON usc.sessionid = st.sessionid

GROUP BY 1

LIMIT 10;

68 Youtube->Google->Instagram->Youtube->Instagram->Instagram->Instagram->...

WINDOW

- Syntax:
 - function(expression) **OVER** ([**PARTITION BY** expression] [**ORDER BY** expression])
- Useful functions:
 - **ROW_NUMBER**, **FIRST_VALUE**, **LAST_VALUE**, **LAG**
 - Math functions: **AVG**, **SUM**, **COUNT**, **MAX**, **MIN**, **MEDIAN**, **NTH_VALUE**

WINDOW - LAG 함수 (1)

- 어떤 사용자 세션에서 시간순으로 봤을 때
 - 앞 세션의 채널이 무엇인지 알고 싶다면?
 - 혹은 다음 세션의 채널이 무엇인지 알고 싶다면?

userId	sessionId	channel	ts	previous channel
27	a67c8c9a961b4182688768dd9ba015fe	Youtube	2019-05-01 17:04:00	
27	b04c387c8384ca083a71b8da516f65f6	Google	2019-05-02 19:21:30	
27	abebb7c39f4b5e46bbcfab2b565ef32b	Naver	2019-05-03 20:38:41	
27	ab49ef78e2877bfd2c2bfa738e459bf0	Facebook	2019-05-04 21:48:07	
27	f740c8d9c193f16d8a07d3a8a751d13f	Facebook	2019-05-05 18:15:31	

WINDOW - LAG 함수 (2)

-- 이전 채널 찾기

```
SELECT usc.*, st.ts,  
       LAG(channel,1) OVER (PARTITION BY userId ORDER BY ts) prev_channel  
FROM raw_data.user_session_channel usc  
JOIN raw_data.session_timestamp st ON usc.sessionid = st.sessionid  
ORDER BY usc.userid, st.ts
```

-- 다음 채널을 찾으려면?

userId	sessionId	channel	ts	previous channel
27	a67c8c9a961b4182688768dd9ba015fe	Youtube	2019-05-01 17:04:00	
27	b04c387c8384ca083a71b8da516f65f6	Google	2019-05-02 19:21:30	Youtube
27	abebb7c39f4b5e46bbcfab2b565ef32b	Naver	2019-05-03 20:38:41	Google
27	ab49ef78e2877bfd2c2bfa738e459bf0	Facebook	2019-05-04 21:48:07	Naver
27	f740c8d9c193f16d8a07d3a8a751d13f	Facebook	2019-05-05 18:15:31	Facebook

JSON Parsing Functions

- JSON의 포맷을 이미 아는 상황에서만 사용가능한 함수
 - JSON String을 입력으로 받아 특정 필드의 값을 추출가능 (nested 구조 지원)
- 예제) JSON_EXTRACT_PATH_TEXT
 - `SELECT JSON_EXTRACT_PATH_TEXT('{"f2":{"f3":"1"},"f4":{"f5":"99","f6":"star"}}', 'f4', 'f6');`

```
{
  "f2": {
    "f3": "1"
  },
  "f4": {
    "f5": "99",
    "f6": "star"
  }
}
```


◆ SQL 실습

❖ 구글 Colab 실습 링크:

- <https://colab.research.google.com/drive/1bbNuJbsk24IXoMRTRkHI-vvteoKVjsyh#scrollTo=hgGl4yxZB20u>

맺음말

SQL은 모든 데이터 직군에게 필요한 기술
데이터 웨어하우스 vs. 프로덕션 데이터베이스

◆ 배움의 전형적인 패턴

The Learning Curve

여기서 어떻게 하느냐가 아주 중요!

1. 가장 중요한 것은 버티는 힘
 - a. 이걸 즐겨야함 :)
2. 내가 뭘 모르지는지 생각해봐야함
 - a. 내가 어디서 막혔는지 구체적으로 질문할 수 있나?
3. 잘 하는 사람 보고 기죽지 않기

