Fastcampus Data Science Extension SCHOOL

SQL(2) - SQL with sqlite3

Index

- review
- WHERE
- import, export csv
- temporary table
- SQL with jupyter, sqlite3, pandas

WHERE

```
sqlite> SELECT id, name FROM user WHERE {condition};
```

Comparison Operator	Description
= or ==	Equal
<> or !=	Not Equal
>	Greater Than
>=	Greater Than or Equal
<	Less Than
<=	Less Than or Equal

Comparison Operator	Description
IN ( )	Matches a value in a list ex) IN ('a', 'b', 'c')
NOT	Negates a condition
BETWEEN	Within a range (inclusive) ex) BETWEEN a AND b
IS NULL	NULL value
IS NOT NULL	Non-NULL value
LIKE	Pattern matching with % and _ ex) LIKE '%as%' or 'fa_t'
EXISTS	Condition is met if subquery returns at least one row
AND	All of the conditions that must be met
OR	Any of the conditions that must be met

remove column

```
sqlite> begin transaction;
sqlite> create temporary table user_old_backup(
   ...> id, name, age, lang);
sqlite> .tables
temp.user_old_backup  user_old
sqlite> insert into temp.user_old_backup
   ...> select id, name, age, lang from user_old;
sqlite> drop table user_old;

sqlite> create table user(id, name, age, lang);
sqlite> insert into user
   ...> select * from temp.user_old_backup;
sqlite> drop table temp.user_old_backup;
sqlite> commit;
```

Change schema in sqlite with temporary table

```
sqlite> begin transaction;
sqlite> create temporary table Products_backup(
   ...> ProductID, ProductName, SupplierID, CategoryID,
   Unit, Price);
sqlite> .tables
Categories            OrderDetails          Shippers
Customers             Orders                Suppliers
Employees             Products              temp.Products_backup
sqlite> insert into temp.Products_backup
select ProductID, ProductName, SupplierID, CategoryID,
Unit, Price from Products;
sqlite> drop table Products;
sqlite> create table Products(
   ...> ProductID integer,
   ...> ProductName text,
   ...> SupplierID integer,
   ...> CategoryID integer,
   ...> Unit text,
   ...> Price integer
   ...> );
sqlite> insert into Products select * from temp.Products_backup;
sqlite> drop table temp.Products_backup;
sqlite> commit;
```

import & export data with csv

```
sqlite> .mode csv
sqlite> .import datain.csv another_users
```

```
sqlite> .headers on
sqlite> .mode csv
sqlite> .once ./dataout.csv
sqlite> select * from user;
```

SQL Practice with trySQL Editor

https://www.w3schools.com/sql/trysql.asp?filename=trysql_select_all

1. customers의 모든 데이터를 선택하세요
2. customers에서 Country 가 Germany, France 이거나 도시가 Berlin, London 인 ContactName, Country, City를 선택하세요
3. Suppliers에서 Country가 Germany, France, UK 인 모든 데이터를 선택하세요
4. Products에서 Price 가 10 이하인 ProductName, SupplierID를 선택하세요
5. Products에서 Price가 10 이하인 SupplierID를 가지는 Supplier의 SupplierID, City, Country를 선택하세요

`OrderDetails에서 Quantity가 40개 이상이며, Customers의 CustomerName이 Ernst나 Stop을 포함하는 전체 데이터를 선택하세요.`

Answer

```
SELECT * FROM [Orders]
where
OrderID in (
        SELECT OrderID FROM OrderDetails
            where Quantity > 40)
            and CustomerID in (
                SELECT CustomerID FROM [Customers]
                where CustomerName like '%Ernst%'
                or CustomerName like '%Stop%');
```

SQL with sqlite3, jupyter, pandas

sql with sqlite3, pandas

```
import pandas as pd
import sqlite3 as lite
```

connect with sqlite

```
db = lite.connect()
```

read_sql or execute

- read_sql

```
query = "SELECT * FROM Customers;"
pd.read_sql(query, db)
```

- execute and fetchall

```
cur = db.cursor()
cur.execute(query)
cur.fetchall()
```

show table list

```
query = """
        SELECT name
    FROM sqlite_master
    WHERE
        type = 'table'
    ;
"""
```

show schema

```
query = """
        SELECT sql
    FROM sqlite_master
    WHERE
        type = 'table'
    ;
"""
```

Which is faster?

```
len(pd.read_sql())
```

```
pd.read_sql(count(*))
```

```
time.time()
# script
time.time()
```

sqlite aggregate functions

- count(*)
- count(X)
- sum(X)
- avg(X)
- group_concat(X)
- group_concat(X,Y)
- max(X)
- min(X)

filter with pandas

```
france = df["Country"] == "France"
germany = df["Country"] == "Germany"
paris = df["City"] == "Paris"
df[germany | paris]
df[germany & paris]
```

filter with sql - operator

```
query = """
    select *
    from Customers
    where
        Country = "France"
        and City = "Paris"
    ;
"""
```

sort with pandas

```
products_df = pd.read_sql('select * from Products;', db)
products_df.sort_values('ProductName', ascending=False)\
        [["ProductName", "Price"]]
```

sort with sql - ORDER BY

```
query = """
    select ProductName, Price
    from Products
    order by ProductName desc
    ;
"""
pd.read_sql(query, db)
```

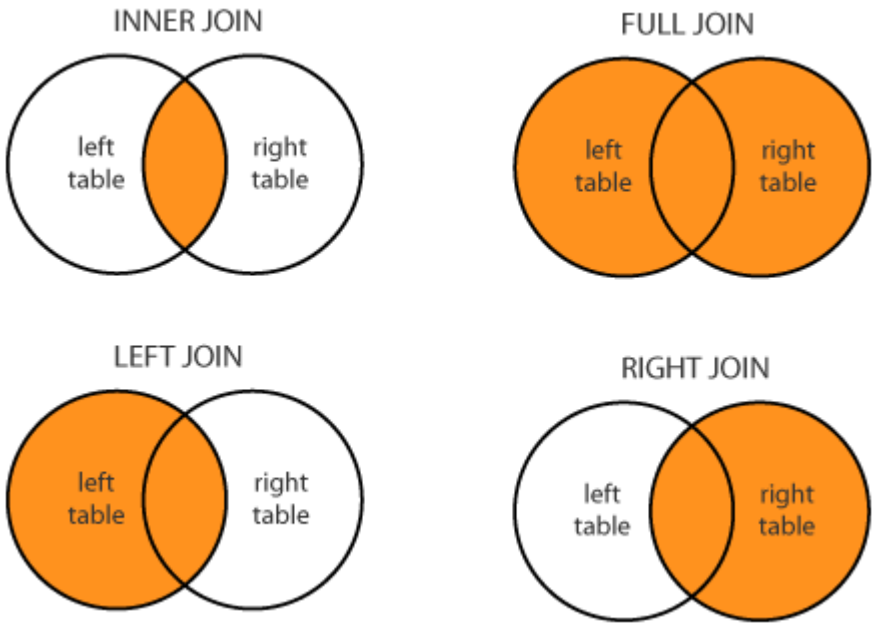
text mining - like

```
# text mining
query = """
    select ProductName, Price
    from Products
    where
        ProductName like "%Ch%"
    ;
"""
pd.read_sql(query, db)
```

join

JOIN

- INNER JOIN: 양쪽의 값 비교 후 조건에 맞는 데이터 병합
- LEFT JOIN: JOIN 왼쪽을 기준으로 오른쪽의 일치하는 데이터 병합
- SELF JOIN: 자기 자신을 병합
- RIGHT JOIN: JOIN 오른쪽을 기준으로 왼쪽의 일치하는 데이터 병합
- FULL OUTER JOIN : 일치하지 않는 값까지 모두 병합
- sqlite는 RIGHT JOIN 과 FULL OUTER JOIN을 지원하지 않습니다.


INNER JOIN
left table
right table
FULL JOIN
left table
right table
LEFT JOIN
left table
right table
RIGHT JOIN
left table
right table

join in pandas - merge

```
integrated_df = orders_df.merge(df, on="CustomerID")\
        [["OrderID", "CustomerID","ContactName","Address"]]
integrated_df.head()
```

join with sql - don't

```
query = """
    select *
    from Customers, Orders
    ;
"""
pd.read_sql(query, db)
```

join with sql - better(1)

```
query = """
    select Orders.OrderID, Orders.CustomerID,
        Customers.ContactName, Customers.Address
    from Customers, Orders
    where
        Customers.CustomerID = Orders.CustomerID
    ;
"""
```

join with sql - better(2)

```
query = """
    select O.OrderID, O.CustomerID, C.ContactName, C.Address
    from Customers C, Orders O
    where
        C.CustomerID = O.CustomerID
    ;
"""
```

join with sql - best

```
query = """
    select O.OrderID, O.CustomerID, C.ContactName, C.Address
    from Customers C
        join Orders O
        on C.CustomerID = O.CustomerID
    ;
"""
```

GROUP BY in pandas

```
date_groups = orders_df.groupby("OrderDate")
date_groups.get_group("1996-07-08")

orders_df["OrderDate"].unique()
```

```
order_count_by_date = pd.DataFrame([
    {
        "OrderDate": OrderDate,
        "Count": len(date_groups.get_group(OrderDate)),
    } for OrderDate in orders_df["OrderDate"].unique()
])
order_count_by_date
```

GROUP BY

```
#sql
query = """
    select count(*), OrderDate
    from Orders
    group by OrderDate
    ;
"""
pd.read_sql(query, db)
```

Having vs where

- 공통점: condition
- where
  - 항상 from 뒤에 위치
  - 모든 필드에 대해 필터링 가능
- having
  - group by 뒤에 위치
  - group by 후 생성된 새로운 테이블에 조건을 줄때

how to use having?

```
query = """
    SELECT
        SUM(d.Quantity) "Count",
        SUM(d.Quantity * p.Price) "Sales",
        ROUND(AVG(d.Quantity * p.Price), 2) "avg",
        SUBSTR(o.OrderDate, 0, 8) "month"
    FROM
        OrderDetails d
        JOIN
            Products p
            ON p.ProductID = d.ProductID
        JOIN
            Orders o
            ON d.OrderID = o.OrderID
    GROUP BY
        substr(o.OrderDate, 0, 8)
    HAVING
        d.Quantity >= 20
    ;
"""
pd.read_sql(query, db)
```

Do It Yourself

`OrderDate`를 조작하여 yyyy-mm 의 형태로 바꾼 컬럼을 추가한 뒤, 연-월 기반의 주문횟수를 sql로 구현하세요

숙제

- 앞서 배운 groupby, join을 활용하여 월간 판매량 합과 평균 구매가격을 sql로 구현하세요.