# Pandas Comparison with SQL

**In [1]: import** **pandas** **as** **pd**  
  
**In [2]: import** **numpy** **as** **np**

Most of the examples will utilize the tips dataset found within pandas tests. We’ll read the data into a DataFrame called tips and assume we have a database table of the same name and structure.

**In [3]:** url = 'https://raw.github.com/pandas-dev/pandas/master/pandas/tests/data/tips.csv'  
  
**In [4]:** tips = pd.read\_csv(url)  
  
**In [5]:** tips.head() head(3) tail() tail(3)  
**Out[5]:**   
 total\_bill tip gender smoker day time size  
0 16.99 1.01 Female No Sun Dinner 2  
1 10.34 1.66 Male No Sun Dinner 3  
2 21.01 3.50 Male No Sun Dinner 3  
3 23.68 3.31 Male No Sun Dinner 2  
4 24.59 3.61 Female No Sun Dinner 4

## SELECT

In SQL, selection is done using a comma-separated list of columns you’d like to select (or a \* to select all columns):

**SELECT** total\_bill, tip, smoker, time  
**FROM** tips  
**LIMIT** 5;

With pandas, column selection is done by passing a list of column names to your DataFrame:

**In [6]:** tips[['total\_bill', 'tip', 'smoker', 'time']].head(5)  
**Out[6]:**   
 total\_bill tip smoker time  
0 16.99 1.01 No Dinner  
1 10.34 1.66 No Dinner  
2 21.01 3.50 No Dinner  
3 23.68 3.31 No Dinner  
4 24.59 3.61 No Dinner

Calling the DataFrame without the list of column names would display all columns (akin to SQL’s \*).

## WHERE

Filtering in SQL is done via a WHERE clause.

**SELECT** \*  
**FROM** tips  
**WHERE** time = 'Dinner'  
**LIMIT** 5;

DataFrames can be filtered in multiple ways; the most intuitive of which is using [boolean indexing](http://pandas.pydata.org/pandas-docs/stable/indexing.html#boolean-indexing).

**In [7]:** tips[tips['time'] == 'Dinner'].head(5)  
**Out[7]:**   
 total\_bill tip gender smoker day time size  
0 16.99 1.01 Female No Sun Dinner 2  
1 10.34 1.66 Male No Sun Dinner 3  
2 21.01 3.50 Male No Sun Dinner 3  
3 23.68 3.31 Male No Sun Dinner 2  
4 24.59 3.61 Female No Sun Dinner 4

The above statement is simply passing a Series of True/False objects to the DataFrame, returning all rows with True.

**In [8]:** is\_dinner = tips['time'] == 'Dinner'  
  
**In [9]:** is\_dinner.value\_counts()  
**Out[9]:**   
True 176  
False 68  
Name: time, dtype: int64  
  
**In [10]:** tips[is\_dinner].head(5)  
Out[10]:   
 total\_bill tip gender smoker day time size  
0 16.99 1.01 Female No Sun Dinner 2  
1 10.34 1.66 Male No Sun Dinner 3  
2 21.01 3.50 Male No Sun Dinner 3  
3 23.68 3.31 Male No Sun Dinner 2  
4 24.59 3.61 Female No Sun Dinner 4

Just like SQL’s OR and AND, multiple conditions can be passed to a DataFrame using | (OR) and & (AND).

*-- tips of more than $5.00 at Dinner meals*  
**SELECT** \*  
**FROM** tips  
**WHERE** time = 'Dinner' **AND** tip > 5.00;

# tips of more than $5.00 at Dinner meals  
**In [11]:** tips[(tips['time'] == 'Dinner') & (tips['tip'] > 5.00)]  
**Out[11]:**   
 total\_bill tip gender smoker day time size  
23 39.42 7.58 Male No Sat Dinner 4  
44 30.40 5.60 Male No Sun Dinner 4  
47 32.40 6.00 Male No Sun Dinner 4  
52 34.81 5.20 Female No Sun Dinner 4  
59 48.27 6.73 Male No Sat Dinner 4  
116 29.93 5.07 Male No Sun Dinner 4  
155 29.85 5.14 Female No Sun Dinner 5  
170 50.81 10.00 Male Yes Sat Dinner 3  
172 7.25 5.15 Male Yes Sun Dinner 2  
181 23.33 5.65 Male Yes Sun Dinner 2  
183 23.17 6.50 Male Yes Sun Dinner 4  
211 25.89 5.16 Male Yes Sat Dinner 4  
212 48.33 9.00 Male No Sat Dinner 4  
214 28.17 6.50 Female Yes Sat Dinner 3  
239 29.03 5.92 Male No Sat Dinner 3

*-- tips by parties of at least 5 diners OR bill total was more than $45*  
**SELECT** \*  
**FROM** tips  
**WHERE** **size** >= 5 **OR** total\_bill > 45;

# tips by parties of at least 5 diners OR bill total was more than $45  
**In [12]:** tips[(tips['size'] >= 5) | (tips['total\_bill'] > 45)]  
**Out[12]:**   
 total\_bill tip gender smoker day time size  
59 48.27 6.73 Male No Sat Dinner 4  
125 29.80 4.20 Female No Thur Lunch 6  
141 34.30 6.70 Male No Thur Lunch 6  
142 41.19 5.00 Male No Thur Lunch 5  
143 27.05 5.00 Female No Thur Lunch 6  
155 29.85 5.14 Female No Sun Dinner 5  
156 48.17 5.00 Male No Sun Dinner 6  
170 50.81 10.00 Male Yes Sat Dinner 3  
182 45.35 3.50 Male Yes Sun Dinner 3  
185 20.69 5.00 Male No Sun Dinner 5  
187 30.46 2.00 Male Yes Sun Dinner 5  
212 48.33 9.00 Male No Sat Dinner 4  
216 28.15 3.00 Male Yes Sat Dinner 5

NULL checking is done using the [**notna()**](https://pandas.pydata.org/pandas-docs/version/0.22.0/generated/pandas.Series.notna.html#pandas.Series.notna) and [**isna()**](https://pandas.pydata.org/pandas-docs/version/0.22.0/generated/pandas.Series.isna.html#pandas.Series.isna) methods.

**In [13]:** frame = pd.DataFrame({'col1': ['A', 'B', np.NaN, 'C', 'D'],  
 **....:**   **....:**   
  
**In [14]:** frame  
**Out[14]:**   
 col1 col2  
0 A F  
1 B NaN  
2 NaN G  
3 C H  
4 D I

Assume we have a table of the same structure as our DataFrame above. We can see only the records where col2 IS NULL with the following query:

**SELECT** \*  
**FROM** frame  
**WHERE** col2 **IS** **NULL**;

**In [15]:** frame[frame['col2'].isna()]  
**Out[15]:**   
 col1 col2  
1 B NaN

Getting items where col1 IS NOT NULL can be done with [**notna()**](https://pandas.pydata.org/pandas-docs/version/0.22.0/generated/pandas.Series.notna.html#pandas.Series.notna).

**SELECT** \*  
**FROM** frame  
**WHERE** col1 **IS** **NOT** **NULL**;

**In [16]:** frame[frame['col1'].notna()]  
**Out[16]:**   
 col1 col2  
0 A F  
1 B NaN  
3 C H  
4 D I

## GROUP BY

In pandas, SQL’s GROUP BY operations are performed using the similarly named [**groupby()**](https://pandas.pydata.org/pandas-docs/version/0.22.0/generated/pandas.DataFrame.groupby.html#pandas.DataFrame.groupby) method. [**groupby()**](https://pandas.pydata.org/pandas-docs/version/0.22.0/generated/pandas.DataFrame.groupby.html#pandas.DataFrame.groupby) typically refers to a process where we’d like to split a dataset into groups, apply some function (typically aggregation) , and then combine the groups together.

A common SQL operation would be getting the count of records in each group throughout a dataset. For instance, a query getting us the number of tips left by gender:

**SELECT** gender, **count**(\*)  
**FROM** tips  
**GROUP** **BY** gender;  
*/\**  
*Female 87*  
*Male 157*  
*\*/*

The pandas equivalent would be:

**In [17]:** tips.groupby('gender').size()  
**Out[17]:**   
gender  
Female 87  
Male 157  
dtype: int64

Notice that in the pandas code we used [**size()**](https://pandas.pydata.org/pandas-docs/version/0.22.0/generated/pandas.core.groupby.DataFrameGroupBy.size.html#pandas.core.groupby.DataFrameGroupBy.size) and not [**count()**](https://pandas.pydata.org/pandas-docs/version/0.22.0/generated/pandas.core.groupby.DataFrameGroupBy.count.html#pandas.core.groupby.DataFrameGroupBy.count). This is because [**count()**](https://pandas.pydata.org/pandas-docs/version/0.22.0/generated/pandas.core.groupby.DataFrameGroupBy.count.html#pandas.core.groupby.DataFrameGroupBy.count) applies the function to each column, returning the number of not null records within each.

**In [18]:** tips.groupby('gender').count()  
**Out[18]:**   
 total\_bill tip smoker day time size  
gender   
Female 87 87 87 87 87 87  
Male 157 157 157 157 157 157

Alternatively, we could have applied the [**count()**](https://pandas.pydata.org/pandas-docs/version/0.22.0/generated/pandas.core.groupby.DataFrameGroupBy.count.html#pandas.core.groupby.DataFrameGroupBy.count) method to an individual column:

**In [19]:** tips.groupby('gender')['total\_bill'].count()  
**Out[19]:**   
gender  
Female 87  
Male 157  
Name: total\_bill, dtype: int64

Multiple functions can also be applied at once. For instance, say we’d like to see how tip amount differs by day of the week - [**agg()**](https://pandas.pydata.org/pandas-docs/version/0.22.0/generated/pandas.core.groupby.DataFrameGroupBy.agg.html#pandas.core.groupby.DataFrameGroupBy.agg) allows you to pass a dictionary to your grouped DataFrame, indicating which functions to apply to specific columns.

**SELECT** **day**, **AVG**(tip), **COUNT**(\*)  
**FROM** tips  
**GROUP** **BY** **day**;  
*/\**  
*Fri 2.734737 19*  
*Sat 2.993103 87*  
*Sun 3.255132 76*  
*Thur 2.771452 62*  
*\*/*

**In [20]:** tips.groupby('day').agg({'tip': np.mean, 'day': np.size})  
**Out[20]:**   
 tip day  
day   
Fri 2.734737 19  
Sat 2.993103 87  
Sun 3.255132 76  
Thur 2.771452 62

Grouping by more than one column is done by passing a list of columns to the [**groupby()**](https://pandas.pydata.org/pandas-docs/version/0.22.0/generated/pandas.DataFrame.groupby.html#pandas.DataFrame.groupby) method.

**SELECT** smoker, **day**, **COUNT**(\*), **AVG**(tip)  
**FROM** tips  
**GROUP** **BY** smoker, **day**;  
*/\**  
*smoker day*  
*No Fri 4 2.812500*  
 *Sat 45 3.102889*  
 *Sun 57 3.167895*  
 *Thur 45 2.673778*  
*Yes Fri 15 2.714000*  
 *Sat 42 2.875476*  
 *Sun 19 3.516842*  
 *Thur 17 3.030000*  
*\*/*

**In [21]:** tips.groupby(['smoker', 'day']).agg({'tip': [np.size, np.mean]})  
**Out[21]:**   
 tip   
 size mean  
smoker day   
No Fri 4.0 2.812500  
 Sat 45.0 3.102889  
 Sun 57.0 3.167895  
 Thur 45.0 2.673778  
Yes Fri 15.0 2.714000  
 Sat 42.0 2.875476  
 Sun 19.0 3.516842  
 Thur 17.0 3.030000

## JOIN

JOINs can be performed with [**join()**](https://pandas.pydata.org/pandas-docs/version/0.22.0/generated/pandas.DataFrame.join.html#pandas.DataFrame.join) or [**merge()**](https://pandas.pydata.org/pandas-docs/version/0.22.0/generated/pandas.merge.html#pandas.merge). By default, [**join()**](https://pandas.pydata.org/pandas-docs/version/0.22.0/generated/pandas.DataFrame.join.html#pandas.DataFrame.join) will join the DataFrames on their indices. Each method has parameters allowing you to specify the type of join to perform (LEFT, RIGHT, INNER, FULL) or the columns to join on (column names or indices).

**In [22]:** df1 = pd.DataFrame({'key': ['A', 'B', 'C', 'D'],  
 **....:**  'value': np.random.randn(4)})  
 **....:**   
  
**In [23]:** df2 = pd.DataFrame({'key': ['B', 'D', 'D', 'E'],  
 **....:**  'value': np.random.randn(4)})  
 **....:**

Assume we have two database tables of the same name and structure as our DataFrames.

Now let’s go over the various types of JOINs.

### INNER JOIN

**SELECT** \*  
**FROM** df1  
**INNER** **JOIN** df2  
 **ON** df1.**key** = df2.**key**;

# merge performs an INNER JOIN by default  
**In [24]:** pd.merge(df1, df2, on='key')  
**Out[24]:**   
 key value\_x value\_y  
0 B -0.318214 0.543581  
1 D 2.169960 -0.426067  
2 D 2.169960 1.138079

[**merge()**](https://pandas.pydata.org/pandas-docs/version/0.22.0/generated/pandas.merge.html#pandas.merge) also offers parameters for cases when you’d like to join one DataFrame’s column with another DataFrame’s index.

**In [25]:** indexed\_df2 = df2.set\_index('key')  
  
**In [26]:** pd.merge(df1, indexed\_df2, left\_on='key', right\_index=True)  
**Out[26]:**   
 key value\_x value\_y  
1 B -0.318214 0.543581  
3 D 2.169960 -0.426067  
3 D 2.169960 1.138079

### LEFT OUTER JOIN

*-- show all records from df1*  
**SELECT** \*  
**FROM** df1  
**LEFT** **OUTER** **JOIN** df2  
 **ON** df1.**key** = df2.**key**;

# show all records from df1  
**In [27]:** pd.merge(df1, df2, on='key', how='left')  
**Out[27]:**   
 key value\_x value\_y  
0 A 0.116174 NaN  
1 B -0.318214 0.543581  
2 C 0.285261 NaN  
3 D 2.169960 -0.426067  
4 D 2.169960 1.138079

### RIGHT JOIN

*-- show all records from df2*  
**SELECT** \*  
**FROM** df1  
**RIGHT** **OUTER** **JOIN** df2  
 **ON** df1.**key** = df2.**key**;

# show all records from df2  
**In [28]:** pd.merge(df1, df2, on='key', how='right')  
**Out[28]:**   
 key value\_x value\_y  
0 B -0.318214 0.543581  
1 D 2.169960 -0.426067  
2 D 2.169960 1.138079  
3 E NaN 0.086073

### FULL JOIN

pandas also allows for FULL JOINs, which display both sides of the dataset, whether or not the joined columns find a match. As of writing, FULL JOINs are not supported in all RDBMS (MySQL).

*-- show all records from both tables*  
**SELECT** \*  
**FROM** df1  
**FULL** **OUTER** **JOIN** df2  
 **ON** df1.**key** = df2.**key**;

# show all records from both frames  
**In [29]:** pd.merge(df1, df2, on='key', how='outer')  
**Out[29]:**   
 key value\_x value\_y  
0 A 0.116174 NaN  
1 B -0.318214 0.543581  
2 C 0.285261 NaN  
3 D 2.169960 -0.426067  
4 D 2.169960 1.138079  
5 E NaN 0.086073

## UNION

UNION ALL can be performed using [**concat()**](https://pandas.pydata.org/pandas-docs/version/0.22.0/generated/pandas.concat.html#pandas.concat).

**In [30]:** df1 = pd.DataFrame({'city': ['Chicago', 'San Francisco', 'New York City'],  
 **....:**  'rank': range(1, 4)})  
 **....:**   
  
**In [31]:** df2 = pd.DataFrame({'city': ['Chicago', 'Boston', 'Los Angeles'],  
 **....:**  'rank': [1, 4, 5]})  
 **....:**

**SELECT** city, rank  
**FROM** df1  
**UNION** **ALL**  
**SELECT** city, rank  
**FROM** df2;  
*/\**  
 *city rank*  
 *Chicago 1*  
*San Francisco 2*  
*New York City 3*  
 *Chicago 1*  
 *Boston 4*  
 *Los Angeles 5*  
*\*/*

**In [32]:** pd.concat([df1, df2])  
**Out[32]:**   
 city rank  
0 Chicago 1  
1 San Francisco 2  
2 New York City 3  
0 Chicago 1  
1 Boston 4  
2 Los Angeles 5

SQL’s UNION is similar to UNION ALL, however UNION will remove duplicate rows.

**SELECT** city, rank  
**FROM** df1  
**UNION**  
**SELECT** city, rank  
**FROM** df2;  
*-- notice that there is only one Chicago record this time*  
*/\**  
 *city rank*  
 *Chicago 1*  
*San Francisco 2*  
*New York City 3*  
 *Boston 4*  
 *Los Angeles 5*  
*\*/*

In pandas, you can use [**concat()**](https://pandas.pydata.org/pandas-docs/version/0.22.0/generated/pandas.concat.html#pandas.concat) in conjunction with [**drop\_duplicates()**](https://pandas.pydata.org/pandas-docs/version/0.22.0/generated/pandas.DataFrame.drop_duplicates.html#pandas.DataFrame.drop_duplicates).

**In [33]:** pd.concat([df1, df2]).drop\_duplicates()  
**Out[33]:**   
 city rank  
0 Chicago 1  
1 San Francisco 2  
2 New York City 3  
1 Boston 4  
2 Los Angeles 5

Xxxxxxxxxxxxx

## Pandas equivalents for some SQL analytic and aggregate functions

### Top N rows with offset

*-- MySQL*  
**SELECT** \* **FROM** tips  
**ORDER** **BY** tip **DESC**  
**LIMIT** 10 **OFFSET** 5;

**In [34]:** tips.nlargest(10+5, columns='tip').tail(10)  
**Out[34]:**   
 total\_bill tip gender smoker day time size  
183 23.17 6.50 Male Yes Sun Dinner 4  
214 28.17 6.50 Female Yes Sat Dinner 3  
47 32.40 6.00 Male No Sun Dinner 4  
239 29.03 5.92 Male No Sat Dinner 3  
88 24.71 5.85 Male No Thur Lunch 2  
181 23.33 5.65 Male Yes Sun Dinner 2  
44 30.40 5.60 Male No Sun Dinner 4  
52 34.81 5.20 Female No Sun Dinner 4  
85 34.83 5.17 Female No Thur Lunch 4  
211 25.89 5.16 Male Yes Sat Dinner 4

### Top N rows per group

*-- Oracle's ROW\_NUMBER() analytic function*  
**SELECT** \* **FROM** (  
 **SELECT**  
 t.\*,  
 ROW\_NUMBER() OVER(PARTITION **BY** **day** **ORDER** **BY** total\_bill **DESC**) **AS** rn  
 **FROM** tips t  
)  
**WHERE** rn < 3  
**ORDER** **BY** **day**, rn;

**In [35]:** (tips.assign(rn=tips.sort\_values(['total\_bill'], ascending=False)  
 **....:**  .groupby(['day'])  
 **....:**  .cumcount() + 1)  
 **....:**  .query('rn < 3')  
 **....:**  .sort\_values(['day','rn'])  
 **....:** )  
 **....:**   
**Out[35]:**   
 total\_bill tip gender smoker day time size rn  
95 40.17 4.73 Male Yes Fri Dinner 4 1  
90 28.97 3.00 Male Yes Fri Dinner 2 2  
170 50.81 10.00 Male Yes Sat Dinner 3 1  
212 48.33 9.00 Male No Sat Dinner 4 2  
156 48.17 5.00 Male No Sun Dinner 6 1  
182 45.35 3.50 Male Yes Sun Dinner 3 2  
197 43.11 5.00 Female Yes Thur Lunch 4 1  
142 41.19 5.00 Male No Thur Lunch 5 2

the same using rank(method=’first’) function

**In [36]:** (tips.assign(rnk=tips.groupby(['day'])['total\_bill']  
 **....:**  .rank(method='first', ascending=False))  
 **....:**  .query('rnk < 3')  
 **....:**  .sort\_values(['day','rnk'])  
 **....:** )  
 **....:**   
**Out[36]:**   
 total\_bill tip gender smoker day time size rnk  
95 40.17 4.73 Male Yes Fri Dinner 4 1.0  
90 28.97 3.00 Male Yes Fri Dinner 2 2.0  
170 50.81 10.00 Male Yes Sat Dinner 3 1.0  
212 48.33 9.00 Male No Sat Dinner 4 2.0  
156 48.17 5.00 Male No Sun Dinner 6 1.0  
182 45.35 3.50 Male Yes Sun Dinner 3 2.0  
197 43.11 5.00 Female Yes Thur Lunch 4 1.0  
142 41.19 5.00 Male No Thur Lunch 5 2.0

*-- Oracle's RANK() analytic function*  
**SELECT** \* **FROM** (  
 **SELECT**  
 t.\*,  
 RANK() OVER(PARTITION **BY** gender **ORDER** **BY** tip) **AS** rnk  
 **FROM** tips t  
 **WHERE** tip < 2  
)  
**WHERE** rnk < 3  
**ORDER** **BY** gender, rnk;

Let’s find tips with (rank < 3) per gender group for (tips < 2). Notice that when using rank(method='min')function rnk\_min remains the same for the same tip (as Oracle’s RANK() function)

**In [37]:** (tips[tips['tip'] < 2]  
 **....:**  .assign(rnk\_min=tips.groupby(['gender'])['tip']  
 **....:**  .rank(method='min'))  
 **....:**  .query('rnk\_min < 3')  
 **....:**  .sort\_values(['gender','rnk\_min'])  
 **....:** )  
 **....:**   
**Out[37]:**   
 total\_bill tip gender smoker day time size rnk\_min  
67 3.07 1.00 Female Yes Sat Dinner 1 1.0  
92 5.75 1.00 Female Yes Fri Dinner 2 1.0  
111 7.25 1.00 Female No Sat Dinner 1 1.0  
236 12.60 1.00 Male Yes Sat Dinner 2 1.0  
237 32.83 1.17 Male Yes Sat Dinner 2 2.0

## UPDATE

**UPDATE** tips  
**SET** tip = tip\*2  
**WHERE** tip < 2;

**In [38]:** tips.loc[tips['tip'] < 2, 'tip'] \*= 2

## DELETE

**DELETE** **FROM** tips  
**WHERE** tip > 9;

In pandas we select the rows that should remain, instead of deleting them

**In [39]:** tips = tips.loc[tips['tip'] <= 9]