In [1]:
```
import pandas as pd
```

In [2]:
```
sql = pd.read_csv(r"C:\Users\HP\OneDrive\Documents\SQL RAW DATA\dataset_1_202509
```

In [3]:
```
sql
```

Out[3]:

	destination	passanger	weather	temperature	time	coupon	expiration
0	No Urgent Place	Alone	Sunny	55	2PM	Restaurant(<20)	1d
1	No Urgent Place	Friend(s)	Sunny	80	10AM	Coffee House	2h
2	No Urgent Place	Friend(s)	Sunny	80	10AM	Carry out & Take away	2h
3	No Urgent Place	Friend(s)	Sunny	80	2PM	Coffee House	2h
4	No Urgent Place	Friend(s)	Sunny	80	2PM	Coffee House	1d
...	...	...	...	...	...	...	...
12679	Home	Partner	Rainy	55	6PM	Carry out & Take away	1d
12680	Work	Alone	Rainy	55	7AM	Carry out & Take away	1d
12681	Work	Alone	Snowy	30	7AM	Coffee House	1d
12682	Work	Alone	Snowy	30	7AM	Bar	1d
12683	Work	Alone	Sunny	80	7AM	Restaurant(20-50)	2h

12684 rows × 27 columns

In [4]:
```
sql.shape
```

Out[4]:
```
(12684, 27)
```

In [5]:
```
sql.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 12684 entries, 0 to 12683
Data columns (total 27 columns):
 #   Column                Non-Null Count  Dtype 
---  ------                --------------  ----- 
 0   destination           12684 non-null  object
 1   passanger             12684 non-null  object
 2   weather               12684 non-null  object
 3   temperature           12684 non-null  int64 
 4   time                  12684 non-null  object
 5   coupon                12684 non-null  object
 6   expiration            12684 non-null  object
 7   gender                12684 non-null  object
 8   age                   12684 non-null  object
 9   maritalStatus         12684 non-null  object
 10  has_children          12684 non-null  int64 
 11  education             12684 non-null  object
 12  occupation            12684 non-null  object
 13  income                12684 non-null  object
 14  car                   108 non-null    object
 15  Bar                   12577 non-null  object
 16  CoffeeHouse           12467 non-null  object
 17  CarryAway             12533 non-null  object
 18  RestaurantLessThan20  12554 non-null  object
 19  Restaurant20To50      12495 non-null  object
 20  toCoupon_GEQ5min      12684 non-null  int64 
 21  toCoupon_GEQ15min     12684 non-null  int64 
 22  toCoupon_GEQ25min     12684 non-null  int64 
 23  direction_same        12684 non-null  int64 
 24  direction_opp         12684 non-null  int64 
 25  Y                     12684 non-null  int64 
 26  row_count             12684 non-null  int64 
dtypes: int64(9), object(18)
memory usage: 2.6+ MB
```

In [6]:
```
sql.isnull()
```

Out[6]:

	destination	passanger	weather	temperature	time	coupon	expiration	gende
0	False	False	False	False	False	False	False	Fals
1	False	False	False	False	False	False	False	Fals
2	False	False	False	False	False	False	False	Fals
3	False	False	False	False	False	False	False	Fals
4	False	False	False	False	False	False	False	Fals
...	...	...	...	...	...	...	...	
12679	False	False	False	False	False	False	False	Fals
12680	False	False	False	False	False	False	False	Fals
12681	False	False	False	False	False	False	False	Fals
12682	False	False	False	False	False	False	False	Fals
12683	False	False	False	False	False	False	False	Fals

12684 rows × 27 columns

In [7]:
```
sql.isnull().sum()
```

Out[7]:
```
destination               0
passanger                 0
weather                   0
temperature               0
time                      0
coupon                    0
expiration                0
gender                    0
age                       0
maritalStatus             0
has_children              0
education                 0
occupation                0
income                    0
car                   12576
Bar                     107
CoffeeHouse             217
CarryAway               151
RestaurantLessThan20    130
Restaurant20To50        189
toCoupon_GEQ5min          0
toCoupon_GEQ15min         0
toCoupon_GEQ25min         0
direction_same            0
direction_opp             0
Y                         0
row_count                 0
dtype: int64
```

In [8]:
```
sql.columns
```

Out[8]:
```
Index(['destination', 'passanger', 'weather', 'temperature', 'time', 'coupon',
       'expiration', 'gender', 'age', 'maritalStatus', 'has_children',
       'education', 'occupation', 'income', 'car', 'Bar', 'CoffeeHouse',
       'CarryAway', 'RestaurantLessThan20', 'Restaurant20To50',
       'toCoupon_GEQ5min', 'toCoupon_GEQ15min', 'toCoupon_GEQ25min',
       'direction_same', 'direction_opp', 'Y', 'row_count'],
      dtype='object')
```

In [9]:
```
sql['weather']
```

Out[9]:
```
0        Sunny
1        Sunny
2        Sunny
3        Sunny
4        Sunny
         ...
12679    Rainy
12680    Rainy
12681    Snowy
12682    Snowy
12683    Sunny
Name: weather, Length: 12684, dtype: object
```

In [10]:
```
sql['destination']
```

Out[10]:
```
0        No Urgent Place
1        No Urgent Place
2        No Urgent Place
3        No Urgent Place
4        No Urgent Place
              ...
12679               Home
12680               Work
12681               Work
12682               Work
12683               Work
Name: destination, Length: 12684, dtype: object
```

In [11]:
```
sql[['weather','temperature']]
```

Out[11]:

	weather	temperature
0	Sunny	55
1	Sunny	80
2	Sunny	80
3	Sunny	80
4	Sunny	80
...	...	...
12679	Rainy	55
12680	Rainy	55
12681	Snowy	30
12682	Snowy	30
12683	Sunny	80

12684 rows × 2 columns

In [12]:
```
sql[['time', 'age']]
```

Out[12]:

	time	age
0	2PM	21
1	10AM	21
2	10AM	21
3	2PM	21
4	2PM	21
...	...	...
12679	6PM	26
12680	7AM	26
12681	7AM	26
12682	7AM	26
12683	7AM	26

12684 rows × 2 columns

In [13]:
```
sql.head(10)
```

Out[13]:

	destination	passanger	weather	temperature	time	coupon	expiration	ge
0	No Urgent Place	Alone	Sunny	55	2PM	Restaurant(<20)	1d	Fe
1	No Urgent Place	Friend(s)	Sunny	80	10AM	Coffee House	2h	Fe
2	No Urgent Place	Friend(s)	Sunny	80	10AM	Carry out & Take away	2h	Fe
3	No Urgent Place	Friend(s)	Sunny	80	2PM	Coffee House	2h	Fe
4	No Urgent Place	Friend(s)	Sunny	80	2PM	Coffee House	1d	Fe
5	No Urgent Place	Friend(s)	Sunny	80	6PM	Restaurant(<20)	2h	Fe
6	No Urgent Place	Friend(s)	Sunny	55	2PM	Carry out & Take away	1d	Fe
7	No Urgent Place	Kid(s)	Sunny	80	10AM	Restaurant(<20)	2h	Fe
8	No Urgent Place	Kid(s)	Sunny	80	10AM	Carry out & Take away	2h	Fe
9	No Urgent Place	Kid(s)	Sunny	80	10AM	Bar	1d	Fe

10 rows × 27 columns

In [14]:
```
sql.tail(10)
```

Out[14]:

	destination	passanger	weather	temperature	time	coupon	expiration
12674	Home	Alone	Rainy	55	10PM	Coffee House	2h
12675	Home	Alone	Snowy	30	10PM	Coffee House	2h
12676	Home	Alone	Sunny	80	6PM	Restaurant(20-50)	1d
12677	Home	Partner	Sunny	30	6PM	Restaurant(<20)	1d
12678	Home	Partner	Sunny	30	10PM	Restaurant(<20)	2h
12679	Home	Partner	Rainy	55	6PM	Carry out & Take away	1d
12680	Work	Alone	Rainy	55	7AM	Carry out & Take away	1d
12681	Work	Alone	Snowy	30	7AM	Coffee House	1d
12682	Work	Alone	Snowy	30	7AM	Bar	1d
12683	Work	Alone	Sunny	80	7AM	Restaurant(20-50)	2h

10 rows × 27 columns

In [15]:
```
sql['temperature'].unique()
```

Out[15]:
```
array([55, 80, 30])
```

In [16]:
```
sql['temperature'].nunique()
```

Out[16]:
```
3
```

In [17]:
```
sql['passanger'].unique() # the unique parameter gives the information about in
```

Out[17]:
```
array(['Alone', 'Friend(s)', 'Kid(s)', 'Partner'], dtype=object)
```

In [18]:
```
sql['passanger'].nunique()
```

Out[18]:
```
4
```

In [19]:
```
sql.ndim
```

Out[19]:
```
2
```

In [20]:
```
sql.size
```

Out[20]:
```
342468
```

In [21]:
```
sql['weather'].unique()
```

Out[21]:
```
array(['Sunny', 'Rainy', 'Snowy'], dtype=object)
```

In [22]:
```
sql['weather'].nunique() # The nunique parameter give the sum of the attribute p
```

Out[22]: 3

In [23]:
```
sql['destination']
```

Out[23]:
```
0        No Urgent Place
1        No Urgent Place
2        No Urgent Place
3        No Urgent Place
4        No Urgent Place
              ...
12679              Home
12680              Work
12681              Work
12682              Work
12683              Work
Name: destination, Length: 12684, dtype: object
```

In [24]:
```
sql[sql['destination'] == 'Home']
```

Out[24]:

	destination	passanger	weather	temperature	time	coupon	expiration
13	Home	Alone	Sunny	55	6PM	Bar	1c
14	Home	Alone	Sunny	55	6PM	Restaurant(20-50)	1c
15	Home	Alone	Sunny	80	6PM	Coffee House	2h
35	Home	Alone	Sunny	55	6PM	Bar	1c
36	Home	Alone	Sunny	55	6PM	Restaurant(20-50)	1c
...	...	...	...	...	...	...	..
12675	Home	Alone	Snowy	30	10PM	Coffee House	2h
12676	Home	Alone	Sunny	80	6PM	Restaurant(20-50)	1c
12677	Home	Partner	Sunny	30	6PM	Restaurant(<20)	1c
12678	Home	Partner	Sunny	30	10PM	Restaurant(<20)	2h
12679	Home	Partner	Rainy	55	6PM	Carry out & Take away	1c

3237 rows × 27 columns

In [25]:
```
sql[sql['destination'] == 'work']
```

Out[25]:

	destination	passanger	weather	temperature	time	coupon	expiration	gender	ag

0 rows × 27 columns

In [26]:
```
sql['destination'].unique()
```

Out[26]:
```
array(['No Urgent Place', 'Home', 'Work'], dtype=object)
```

In [27]:
```
sql[sql['destination'] == 'No Urgent Place'] # This line gives the how many peop
```

Out[27]:

	destination	passanger	weather	temperature	time	coupon	expiration
0	No Urgent Place	Alone	Sunny	55	2PM	Restaurant(<20)	1c
1	No Urgent Place	Friend(s)	Sunny	80	10AM	Coffee House	2h
2	No Urgent Place	Friend(s)	Sunny	80	10AM	Carry out & Take away	2h
3	No Urgent Place	Friend(s)	Sunny	80	2PM	Coffee House	2h
4	No Urgent Place	Friend(s)	Sunny	80	2PM	Coffee House	1c
...	...	...	...	...	...	...	..
12667	No Urgent Place	Alone	Rainy	55	10AM	Bar	1c
12668	No Urgent Place	Alone	Sunny	80	10AM	Restaurant(<20)	2h
12669	No Urgent Place	Partner	Sunny	30	10AM	Restaurant(20-50)	1c
12670	No Urgent Place	Partner	Rainy	55	6PM	Bar	2h
12671	No Urgent Place	Partner	Snowy	30	10AM	Restaurant(<20)	1c

6283 rows × 27 columns

In [28]:
```
sql.sort_values('coupon')
```

Out[28]:

	destination	passanger	weather	temperature	time	coupon	expiration
11702	Home	Partner	Sunny	30	10PM	Bar	2h
9930	No Urgent Place	Alone	Snowy	30	2PM	Bar	1d
10632	Home	Alone	Rainy	55	6PM	Bar	1d
7997	No Urgent Place	Friend(s)	Rainy	55	10PM	Bar	2h
11166	Work	Alone	Snowy	30	7AM	Bar	1d
...	...	...	...	...	...	...	..
10476	Home	Alone	Sunny	80	6PM	Restaurant(<20)	1d
5447	Home	Alone	Sunny	80	10PM	Restaurant(<20)	2h
10478	Home	Alone	Snowy	30	10PM	Restaurant(<20)	2h
5440	No Urgent Place	Alone	Sunny	80	2PM	Restaurant(<20)	2h
0	No Urgent Place	Alone	Sunny	55	2PM	Restaurant(<20)	1d

12684 rows × 27 columns

df.sort_values() is the method used for sorting.

'coupon' is the name of the column you want to sort by.

ascending=False is the key part. By default, sort_values sorts in ascending order (ascending=True). To get a descending sort, you must explicitly set ascending to False.

In [29]:
```
sql.rename(columns = {'destination': 'Destination'},inplace=True)
```

In [30]:
```
sql
```

Out[30]:

	Destination	passanger	weather	temperature	time	coupon	expiratio
0	No Urgent Place	Alone	Sunny	55	2PM	Restaurant(<20)	1d
1	No Urgent Place	Friend(s)	Sunny	80	10AM	Coffee House	2h
2	No Urgent Place	Friend(s)	Sunny	80	10AM	Carry out & Take away	2h
3	No Urgent Place	Friend(s)	Sunny	80	2PM	Coffee House	2h
4	No Urgent Place	Friend(s)	Sunny	80	2PM	Coffee House	1d
...	...	...	...	...	...	...	.
12679	Home	Partner	Rainy	55	6PM	Carry out & Take away	1d
12680	Work	Alone	Rainy	55	7AM	Carry out & Take away	1d
12681	Work	Alone	Snowy	30	7AM	Coffee House	1d
12682	Work	Alone	Snowy	30	7AM	Bar	1d
12683	Work	Alone	Sunny	80	7AM	Restaurant(20-50)	2h

12684 rows × 27 columns

inplace=True This is an important optional argument.

inplace determines whether the changes are applied directly to the original DataFrame or if a new, modified DataFrame is returned.

When inplace=True, the .rename() operation modifies the DataFrame sql directly. The original DataFrame is permanently changed, and the method returns None.

If you were to use inplace=False (or omit the argument, as False is the default), the .rename() method would return a new DataFrame with the renamed column, while leaving the original DataFrame unchanged. You would typically assign this new DataFrame to a variable, like new_df = df.rename(...).

In [31]:
```
sql.groupby('occupation').size().to_frame('Count').reset_index()
```

Out[31]:

	occupation	Count
0	Architecture & Engineering	175
1	Arts Design Entertainment Sports & Media	629
2	Building & Grounds Cleaning & Maintenance	44
3	Business & Financial	544
4	Community & Social Services	241
5	Computer & Mathematical	1408
6	Construction & Extraction	154
7	Education&Training&Library	943
8	Farming Fishing & Forestry	43
9	Food Preparation & Serving Related	298
10	Healthcare Practitioners & Technical	244
11	Healthcare Support	242
12	Installation Maintenance & Repair	133
13	Legal	219
14	Life Physical Social Science	170
15	Management	838
16	Office & Administrative Support	639
17	Personal Care & Service	175
18	Production Occupations	110
19	Protective Service	175
20	Retired	495
21	Sales & Related	1093
22	Student	1584
23	Transportation & Material Moving	218
24	Unemployed	1870

The code `sql.groupby('occupation').size().to_frame('Count').reset_index()` is not a SQL query. It is a series of chained method calls in the **Python Pandas library**, used for data manipulation. Let's break down each part of this code snippet:

`sql`

This is a placeholder for a Pandas DataFrame. A DataFrame is a data structure that organizes data into a 2D table of rows and columns, similar to a spreadsheet.

`.groupby('occupation')`

This is the first step in a "split-apply-combine" operation.

- The `groupby()` method splits the DataFrame into smaller groups based on unique values in the specified column.
- In this case, it groups all rows that have the same 'occupation' together.

`.size()`

This is an aggregation method applied to the grouped object.

- `size()` counts the number of rows (or elements) in each group. It is similar to `count()` but it includes missing values (`NaN`s) and works on the entire group, not on a specific column.
- The result of this operation is a Pandas **Series**, where the index is the unique 'occupation' values and the values are the counts for each occupation.

`.to_frame('Count')`

- The `size()` method returns a Series, which can sometimes be difficult to work with for further operations.
- The `.to_frame()` method converts this Series into a DataFrame.
- `'Count'` is an optional argument that names the new column containing the counts. Without this argument, the column would be given a default name (like `0` or `size`).

`.reset_index()`

- After a `groupby()` operation, the grouped column ('occupation' in this case) becomes the index of the resulting Series or DataFrame.
- The `reset_index()` method converts this index back into a regular column, so that 'occupation' is no longer the index and the DataFrame has a default integer-based index (starting from 0).

Overall, the code performs the following task:

It counts the number of occurrences of each unique value in the 'occupation' column of a DataFrame. The final result is a new DataFrame with two columns: 'occupation' and 'Count', which shows the count of each occupation.

This is a very common pattern in data analysis to get a frequency count of categorical data.

In [32]:
```
sql.groupby('weather')['temperature'].mean().to_frame('avg_tem').reset_index()
```

Out[32]:

	weather	avg_tem
0	Rainy	55.000000
1	Snowy	30.000000
2	Sunny	68.946271

In [33]:
```
sql.groupby('weather')['temperature'].size().to_frame('Count_temp').reset_index(
```

Out[33]:

	weather	Count_temp
0	Rainy	1210
1	Snowy	1405
2	Sunny	10069

['temperature'].size() ['temperature'] selects the 'temperature' column from each of the groups created in the previous step.

.size() is an aggregation method. It counts the number of rows (or elements) within each group. The result is a Pandas Series where the index is the unique 'weather' values and the values are the counts for each weather type.

In [34]:
```
sql.groupby('weather')['temperature'].nunique().to_frame('count_distinct_temp').
```

Out[34]:

	weather	count_distinct_temp
0	Rainy	1
1	Snowy	1
2	Sunny	3

['temperature']: After grouping, this selects the 'temperature' column from each of those newly created groups.

.nunique(): This is an aggregation function. For each group (each unique 'weather' type), it counts the number of unique values in the 'temperature' column. For example, if a 'rainy' group has temperatures [10, 12, 10, 15], the nunique() result for this group would be 3 (for the unique values 10, 12, and 15).

In [35]:
```
sql.groupby('weather')['temperature'].sum().to_frame('sum_temp').reset_index()
```

Out[35]:

	weather	sum_temp
0	Rainy	66550
1	Snowy	42150
2	Sunny	694220

In [36]:
```
sql.groupby('weather')['temperature'].min().to_frame('min_temp').reset_index()
```

Out[36]:

	weather	min_temp
0	Rainy	55
1	Snowy	30
2	Sunny	30

In [37]:
```
sql.groupby('weather')['temperature'].max().to_frame('min_temp').reset_index()
```

Out[37]:

	weather	min_temp
0	Rainy	55
1	Snowy	30
2	Sunny	80

In [38]:
```
sql.groupby('occupation').filter(lambda x: x['occupation'].iloc[0] ==
'Student').groupby('occupation').size()
```

Out[38]:
```
occupation
Student    1584
dtype: int64
```

The Python code `sql.groupby('occupation').filter(lambda x: x['occupation'].iloc[0] == 'Student').groupby('occupation').size()` is a pandas-like code snippet that performs a series of operations on a DataFrame object, likely named `sql`.

Here is a step-by-step breakdown of what the code is attempting to do:

1. **`sql.groupby('occupation')`**: This first step groups the rows of the DataFrame `sql` based on the unique values in the `'occupation'` column. All rows with the same occupation (e.g., 'Student', 'Engineer', 'Doctor') are put into a single group.
2. **`.filter(lambda x: x['occupation'].iloc[0] == 'Student')`**: This is a filter operation applied to the groups created in the previous step. The `lambda` function is a small, anonymous function that takes a group (`x`) as input.
   - `x['occupation']`: This selects the `'occupation'` column for the current group.
   - `.iloc[0]`: This gets the value of the first row in the `'occupation'` column for that group.
   - `== 'Student'`: This checks if the value is equal to the string `'Student'`.
   - The `.filter()` method then keeps only the groups for which this condition returns `True`. In this specific case, it will keep only the group where the occupation is `'Student'`. This is a somewhat inefficient way to filter for a specific group, as a simple boolean filter or `get_group()` method would be more direct.
3. **`.groupby('occupation')`**: The output of the `.filter()` method is a new DataFrame containing only the rows where the occupation is 'Student'. This step

regroups this new, filtered DataFrame by the `'occupation'` column. Since the filtered DataFrame only contains one unique occupation (`'Student'`), this effectively creates a single group.

4. **`.size()`**: This is an aggregation function applied to the groups. It counts the number of rows (the size) within each group. Since there is only one group (the 'Student' group) remaining, it will return the total count of rows for that group.

In summary, the code effectively counts the total number of rows where the `'occupation'` is `'Student'`.

In [50]:
```
sql1 = pd.read_csv(r"C:\Users\HP\OneDrive\Documents\SQL RAW DATA\dataset_1_20250
```

In [52]:
```
pd.concat([sql, sql1])['destination'].drop_duplicates()
```

Out[52]:
```
0                 NaN
0     No Urgent Place
13               Home
16               Work
Name: destination, dtype: object
```

This code is a single-line command in the `pandas` library for Python. It performs a sequence of operations to find all the unique values in the 'destination' column across two different dataframes, `sql` and `sql1`.

Here is a step-by-step description of what each part of the code does:

1. `pd.concat([sql, sql1])`

- `pd.concat()` is a pandas function used to concatenate or join `pandas` objects (like DataFrames or Series) along a particular axis.
- `[sql, sql1]` is a list containing the two DataFrames you want to combine.
- **Result:** This part of the code vertically stacks the rows of the `sql` and `sql1` dataframes. It creates a new, single DataFrame that contains all the rows from `sql` followed by all the rows from `sql1`.

2. `['destination']`

- This is standard pandas syntax for selecting a column from a DataFrame.
- It is applied to the new, combined DataFrame created in the previous step.
- **Result:** This selects only the column named 'destination' from the concatenated DataFrame. The output of this operation is a `pandas` Series object, which is essentially a single-column array. This Series will contain every destination value from both the `sql` and `sql1` dataframes, including any duplicates.

3. `.drop_duplicates()`

- This is a method called on the `pandas` Series that was created in the previous step.
- Its purpose is to remove duplicate values from the Series.

- By default, it keeps the first occurrence of each value and removes all subsequent duplicates.
- **Result:** The final output is a new `pandas` Series that contains only the unique values from the 'destination' column.

Summary

In essence, the entire command can be summarized as:

"Take the two dataframes, `sql` and `sql1`, combine them into one. Then, from this new combined dataframe, select the `destination` column. Finally, remove all duplicate entries from that column, giving me a clean list of all unique destinations from both original dataframes."

In [58]:
```
sql[sql['passanger'] == 'Alone'][['Destination', 'passanger']]
```

Out[58]:

	Destination	passanger
0	No Urgent Place	Alone
13	Home	Alone
14	Home	Alone
15	Home	Alone
16	Work	Alone
...	...	...
12676	Home	Alone
12680	Work	Alone
12681	Work	Alone
12682	Work	Alone
12683	Work	Alone

7305 rows × 2 columns

"From the sql DataFrame, show me only the Destination and passanger columns for all the rows where the value in the passanger column is 'Alone'

In [59]:
```
sql[sql['weather'].str.startswith('Sun')]
```

Out[59]:

	Destination	passanger	weather	temperature	time	coupon	expiratio
0	No Urgent Place	Alone	Sunny	55	2PM	Restaurant(<20)	1d
1	No Urgent Place	Friend(s)	Sunny	80	10AM	Coffee House	2h
2	No Urgent Place	Friend(s)	Sunny	80	10AM	Carry out & Take away	2h
3	No Urgent Place	Friend(s)	Sunny	80	2PM	Coffee House	2h
4	No Urgent Place	Friend(s)	Sunny	80	2PM	Coffee House	1d
...	...	...	...	...	...	...	.
12673	Home	Alone	Sunny	30	6PM	Carry out & Take away	1d
12676	Home	Alone	Sunny	80	6PM	Restaurant(20-50)	1d
12677	Home	Partner	Sunny	30	6PM	Restaurant(<20)	1d
12678	Home	Partner	Sunny	30	10PM	Restaurant(<20)	2h
12683	Work	Alone	Sunny	80	7AM	Restaurant(20-50)	2h

10069 rows × 27 columns

"From the sql DataFrame, select and return all rows where the text in the weather column starts with 'Sun'.

In [60]:
```
sql[(sql['temperature']>=29) & (sql['temperature']>=75)]['temperature'].unique()
```

Out[60]:
```
array([80])
```

his is a method called on the Series of temperature values.

It returns an array of all the unique values present in that Series, removing any duplicates.

Final Output: A NumPy array containing a list of all the distinct temperature values that were 75 or greater.

In [61]:
```
sql[sql['occupation'].isin(['Sales & Related', 'Management'])][['occupation']]
```

Out[61]:

	occupation
193	Sales & Related
194	Sales & Related
195	Sales & Related
196	Sales & Related
197	Sales & Related
...	...
12679	Sales & Related
12680	Sales & Related
12681	Sales & Related
12682	Sales & Related
12683	Sales & Related

1931 rows × 1 columns

This code is a concise `pandas` command used to filter a DataFrame and then select a specific column from the filtered result. Its purpose is to select all rows where the value in the `'occupation'` column is either `'Sales & Related'` or `'Management'`, and then return only the `'occupation'` column for those rows.

Here is a step-by-step breakdown of how the code works:

1. The Filter: `sql['occupation'].isin(['Sales & Related', 'Management'])`

- `sql['occupation']`: This selects the `'occupation'` column from the `sql` DataFrame.
- `.isin(...)`: This is a powerful `pandas` method that checks for membership. It compares each value in the `'occupation'` Series against the list of values provided (`['Sales & Related', 'Management']`).
- **Result of this step:** This operation creates a **boolean mask** (a Series of `True` and `False` values). A `True` value is placed for every row where the occupation is either `'Sales & Related'` or `'Management'`, and a `False` for all other rows.

2. The Row Selection: `sql[...]`

- The boolean mask from the first step is placed inside the `[]` brackets of the `sql` DataFrame.
- `pandas` uses this mask to filter the DataFrame, keeping only the rows where the mask's value is `True`.
- **Result of this step:** A new DataFrame that contains all original columns but only the rows where the occupation is either `'Sales & Related'` or `'Management'`.

3. The Column Selection: `[['occupation']]`

- This part is applied to the new, filtered DataFrame from the previous step.
- The double brackets `[[]]` are used to select a list of columns.
- **Result of this step:** The final output is a DataFrame that contains only the `'occupation'` column, and its rows are limited to those that passed the initial filter.

In [ ]: