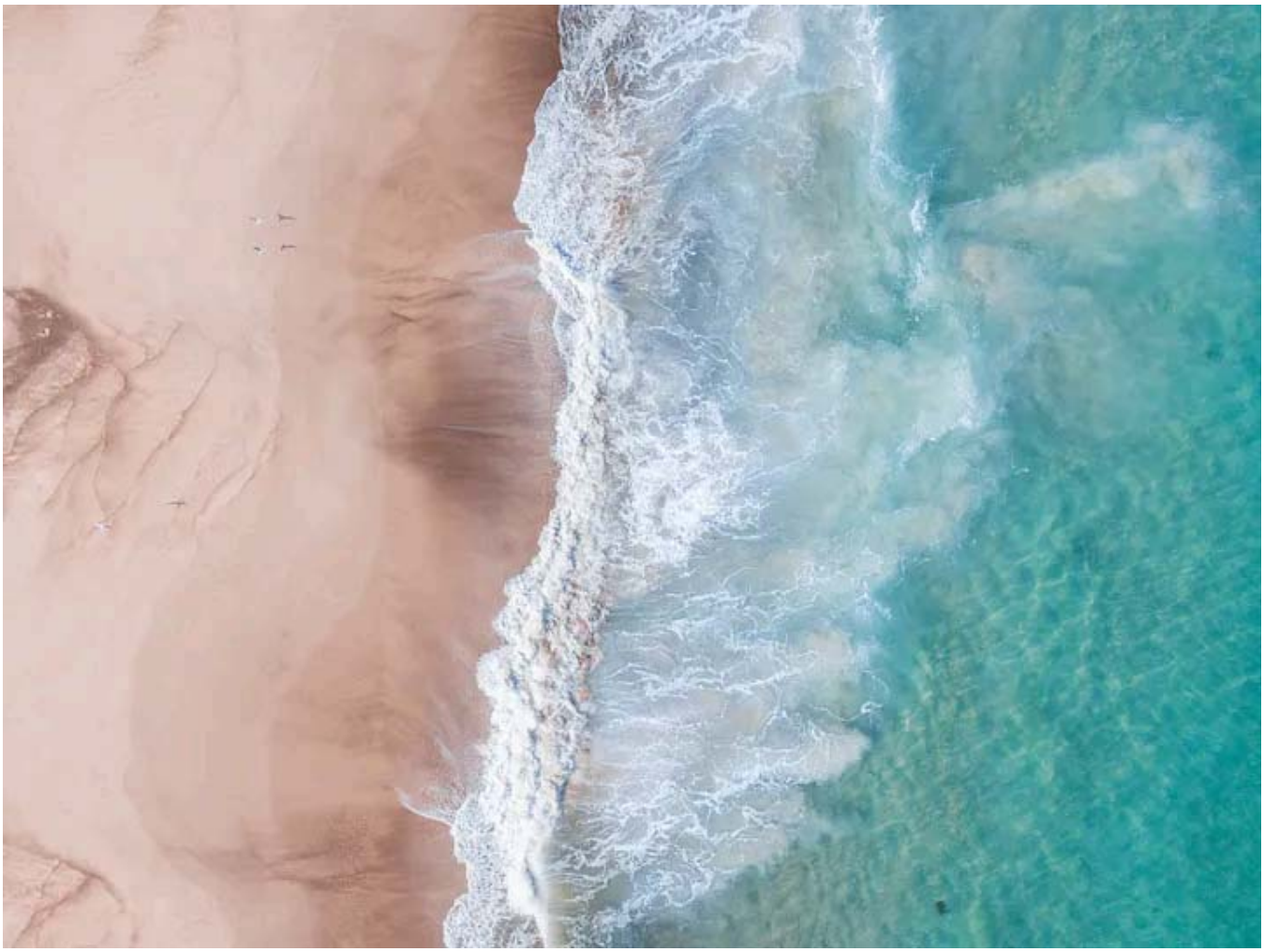


Photo by Samuel Scrimshaw on Unsplash

SQL vs. Pandas: A Comparative Study for Data Analysis-60 Code Snippets

SQL and Pandas Code Snippets

Senthil E · Following
Published in Level Up Coding
26 min read · May 17, 2023

Listen Share More

Introduction:

Introduction to SQL

- SQL, which stands for Structured Query Language.
- It is used for manipulating and retrieving data stored in relational databases.

Introduction to Pandas

- Pandas is a Python package for data manipulation and analysis.
- It provides data structures and functions needed to manipulate structured data. It works well with tabular data from .csv files, SQL databases, or Excel spreadsheets.

Purpose of the Article

- The goal of this article is to compare SQL and pandas, two powerful tools for data manipulation and retrieval.

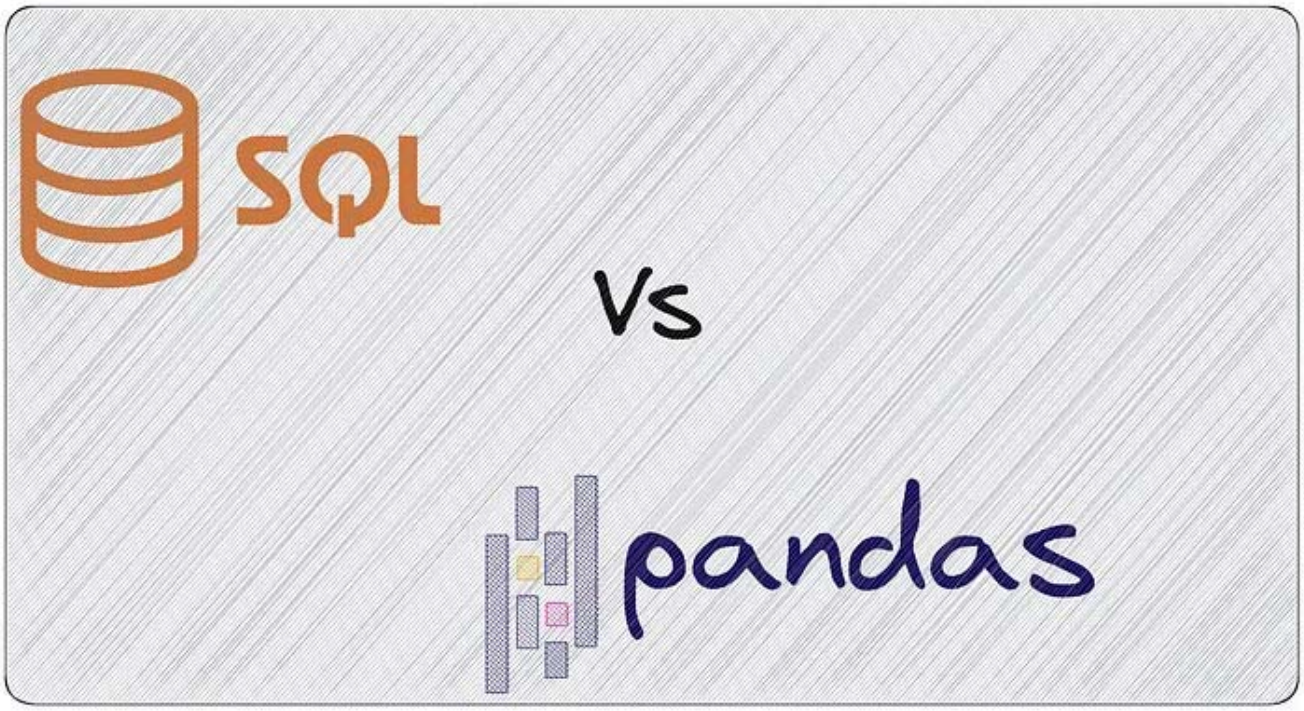


Image by the Author

Background and Basics:

SQL:

- SQL is used for managing and manipulating relational databases.

```
CREATE TABLE Employees (
    ID INT PRIMARY KEY,
    Name VARCHAR(100),
```

```
    Age INT,
    Salary FLOAT
);

INSERT INTO Employees (ID, Name, Age, Salary)
VALUES (1, 'John Doe', 30, 50000.00);
```

Pandas:

- The two primary data structures of pandas are Series (1-dimensional) and DataFrame (2-dimensional).

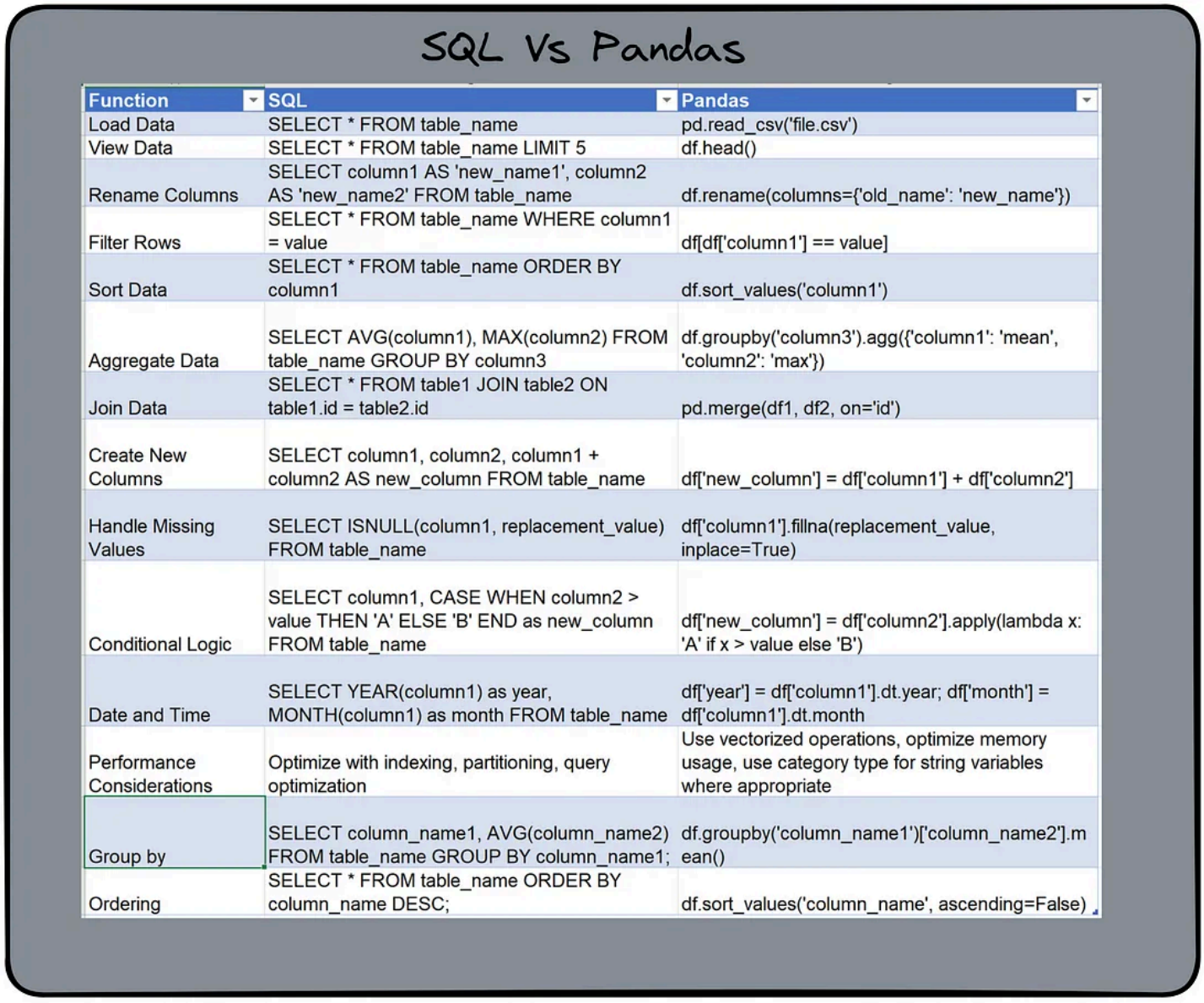
```
import pandas as pd

# Create DataFrame
df = pd.DataFrame({
    'ID': [1],
    'Name': ['John Doe'],
    'Age': [30],
    'Salary': [50000.00]
})

# Display DataFrame
print(df)
```

SQL Vs pandas:

Image by the Author



Function	SQL	Pandas
Load Data	SELECT * FROM table_name	pd.read_csv('file.csv')
View Data	SELECT * FROM table_name LIMIT 5	df.head()
Rename Columns	SELECT column1 AS 'new_name1', column2 AS 'new_name2' FROM table_name	df.rename(columns={'old_name': 'new_name'})
Filter Rows	SELECT * FROM table_name WHERE column1 = value	df[df['column1'] == value]
Sort Data	SELECT * FROM table_name ORDER BY column1	df.sort_values('column1')
Aggregate Data	SELECT AVG(column1), MAX(column2) FROM table_name GROUP BY column3	df.groupby('column3').agg({'column1': 'mean', 'column2': 'max'})
Join Data	SELECT * FROM table1 JOIN table2 ON table1.id = table2.id	pd.merge(df1, df2, on='id')
Create New Columns	SELECT column1, column2, column1 + column2 AS new_column FROM table_name	df['new_column'] = df['column1'] + df['column2']
Handle Missing Values	SELECT ISNULL(column1, replacement_value) FROM table_name	df['column1'].fillna(replacement_value, inplace=True)
Conditional Logic	SELECT column1, CASE WHEN column2 > value THEN 'A' ELSE 'B' END as new_column FROM table_name	df['new_column'] = df['column2'].apply(lambda x: 'A' if x > value else 'B')
Date and Time	SELECT YEAR(column1) as year, MONTH(column1) as month FROM table_name	df['year'] = df['column1'].dt.year; df['month'] = df['column1'].dt.month
Performance Considerations	Optimize with indexing, partitioning, query optimization	Use vectorized operations, optimize memory usage, use category type for string variables where appropriate
Group by	SELECT column_name1, AVG(column_name2) FROM table_name GROUP BY column_name1;	df.groupby('column_name1')['column_name2'].mean()
Ordering	SELECT * FROM table_name ORDER BY column_name DESC;	df.sort_values('column_name', ascending=False)

Image by the Author

Data Loading:

Loading data in SQL:

- `INSERT INTO` statement

```
INSERT INTO Employees (ID, Name, Age, Salary)
VALUES (1, 'John Doe', 30, 50000.00),
       (2, 'Jane Doe', 28, 55000.00),
       (3, 'Jim Brown', 35, 60000.00);
```

For loading larger datasets from external files, the `BULK INSERT` command (in SQL Server) or `LOAD DATA INFILE` (in MySQL) is often used.

```
BULK INSERT Employees
FROM 'C:\Data\Employees.csv'
WITH (FIELDTERMINATOR = ',', ROWTERMINATOR = '\n');
```

Loading data in pandas:

In pandas, you can manually create dataframes like so:

```
df = pd.DataFrame({
    'ID': [1, 2, 3],
    'Name': ['John Doe', 'Jane Doe', 'Jim Brown'],
    'Age': [30, 28, 35],
    'Salary': [50000.00, 55000.00, 60000.00]
})
```

But for larger datasets, you'd typically load the data from an external file using the `pd.read_csv()` or `pd.read_excel()` functions:

```
df = pd.read_csv('C:/Data/Employees.csv')
```

SQL is usually preferred when the data is already stored in a relational database.

Image by the Author

Creating, altering, and deleting tables/dataframes:

SQL

In SQL, you can create a table using the `CREATE TABLE` statement, alter it with the `ALTER TABLE` statement, and delete it with the `DROP TABLE` statement.

```
-- Create table
CREATE TABLE Employees (
    ID INT PRIMARY KEY,
```

```
    Name VARCHAR(100),
    Age INT,
    Salary FLOAT
);

-- Alter table
ALTER TABLE Employees
ADD Email VARCHAR(255);

-- Drop table
DROP TABLE Employees;
```

Pandas:

In pandas, you can create a DataFrame using the `pd.DataFrame()` function, alter it by adding or dropping columns, and delete it by using the `del` keyword.

```
# Create DataFrame
df = pd.DataFrame({
    'ID': [1, 2, 3],
    'Name': ['John Doe', 'Jane Doe', 'Jim Brown'],
    'Age': [30, 28, 35],
    'Salary': [50000.00, 55000.00, 60000.00]
})

# Alter DataFrame (add column)
df['Email'] = ['john@example.com', 'jane@example.com', 'jim@example.com']

# Alter DataFrame (drop column)
df = df.drop(columns=['Email'])

# Delete DataFrame
del df
```

Inserting, updating, and deleting records:

SQL:

In SQL, you can insert records into a table using the `INSERT INTO` statement, update them with the `UPDATE` statement, and delete them with the `DELETE` statement.

```
-- Insert record
INSERT INTO Employees (ID, Name, Age, Salary)
VALUES (4, 'Jack Smith', 40, 70000.00);
```

```
-- Update record
UPDATE Employees
SET Salary = 75000.00
WHERE ID = 4;

-- Delete record
DELETE FROM Employees
WHERE ID = 4;
```

Pandas:

In pandas, you can insert records into a DataFrame by appending rows, update them by assigning new values to specific locations and deleting them with the `drop()` function.

```
# Insert record
df = df.append({'ID': 4, 'Name': 'Jack Smith', 'Age': 40, 'Salary': 70000.00},

# Update record
df.loc[df['ID'] == 4, 'Salary'] = 75000.00

# Delete record
df = df[df.ID != 4]
```

Data Querying:

Basic data querying in SQL:

- In SQL, you use the `SELECT` statement to query data from a table.
- You can filter rows using the `WHERE` clause,
- group records with the `GROUP BY` clause,
- and sort results with the `ORDER BY` clause.

```
-- Select records
SELECT * FROM Employees;

-- Filter records
SELECT * FROM Employees WHERE Age > 30;

-- Group records
SELECT Age, COUNT(*) FROM Employees GROUP BY Age;
```

```
-- Sort records
SELECT * FROM Employees ORDER BY Salary DESC;
```

Basic data querying in pandas:

- In pandas, you can select data by label with the `.loc` function, by position with the `.iloc` function, group records with the `groupby()` function, and sort values with the `sort_values()` function.

```
# Select records
df

# Filter records
df.loc[df['Age'] > 30]

# Group records
df.groupby('Age').size()

# Sort records
df.sort_values(by='Salary', ascending=False)
```

Advanced-Data Manipulation

Joins in SQL and pandas:

SQL:

SQL provides several types of joins including `INNER JOIN`, `LEFT JOIN`, `RIGHT JOIN`, and `FULL JOIN`.

```
-- Assume we have another table 'Departments'
CREATE TABLE Departments (
    DeptID INT PRIMARY KEY,
    DeptName VARCHAR(100),
    EmployeeID INT
);

-- Inner Join
SELECT * FROM Employees
INNER JOIN Departments
ON Employees.ID = Departments.EmployeeID;

-- Left Join
SELECT * FROM Employees
```

```
LEFT JOIN Departments
ON Employees.ID = Departments.EmployeeID;

-- Right Join
SELECT * FROM Employees
RIGHT JOIN Departments
ON Employees.ID = Departments.EmployeeID;

-- Full Join
SELECT * FROM Employees
FULL JOIN Departments
ON Employees.ID = Departments.EmployeeID;
```

Pandas:

Pandas also supports the same types of joins with the `merge()` function.

```
# Assume we have another DataFrame 'departments'
departments = pd.DataFrame({
    'DeptID': [1, 2, 3],
    'DeptName': ['HR', 'Sales', 'Marketing'],
    'EmployeeID': [1, 2, 3]
})

# Inner Join
pd.merge(df, departments, left_on='ID', right_on='EmployeeID', how='inner')

# Left Join
pd.merge(df, departments, left_on='ID', right_on='EmployeeID', how='left')

# Right Join
pd.merge(df, departments, left_on='ID', right_on='EmployeeID', how='right')

# Outer Join
pd.merge(df, departments, left_on='ID', right_on='EmployeeID', how='outer')
```

Aggregations in SQL and pandas:

SQL:

SQL provides several aggregation functions including `COUNT`, `SUM`, `AVG`, `MIN`, and `MAX`.

```
-- Count
SELECT COUNT(*) FROM Employees;
```

```sql
-- Sum
SELECT SUM(Salary) FROM Employees;

-- Average
SELECT AVG(Salary) FROM Employees;

-- Minimum
SELECT MIN(Salary) FROM Employees;

-- Maximum
SELECT MAX(Salary) FROM Employees;
```

Pandas:

Pandas also provides similar aggregation functions.

```python
# Count
df['ID'].count()

# Sum
df['Salary'].sum()

# Average
df['Salary'].mean()

# Minimum
df['Salary'].min()

# Maximum
df['Salary'].max()
```

Subqueries and complex operations in SQL and pandas:

SQL:

In SQL, you can use subqueries to perform complex operations. For example, finding employees with above-average salaries:

```sql
SELECT * FROM Employees
WHERE Salary > (SELECT AVG(Salary) FROM Employees);
```

Pandas:

In pandas, you can achieve similar results with boolean indexing.

```
df[df['Salary'] > df['Salary'].mean()]
```

Handling Null Values

Null value handling in SQL:

SQL provides several ways to handle null values, including the `IS NULL`, `IS NOT NULL`, and `COALESCE` keywords.

```
-- Select records where Age is null
SELECT * FROM Employees WHERE Age IS NULL;

-- Select records where Age is not null
SELECT * FROM Employees WHERE Age IS NOT NULL;

-- Replace null values in the Age column with 0
SELECT COALESCE(Age, 0) FROM Employees;
```

Null value handling in pandas:

Pandas also provides several functions for handling null values, such as `.isnull()`, `.notnull()`, and `.fillna()`.

```
# Select records where Age is null
df[df['Age'].isnull()]

# Select records where Age is not null
df[df['Age'].notnull()]

# Replace null values in the Age column with 0
df['Age'].fillna(0, inplace=True)
```

Conditional Logic

Conditional logic in SQL:

In SQL, the `CASE` statement is used for conditional logic.

```sql
-- Use CASE to categorize employees based on their salaries
SELECT Name, Salary,
CASE
    WHEN Salary < 50000 THEN 'Low'
    WHEN Salary BETWEEN 50000 AND 100000 THEN 'Medium'
    ELSE 'High'
END AS Salary_Category
FROM Employees;
```

Conditional logic in pandas:

In pandas, the `.where()` and `.mask()` functions are used for conditional logic. These functions allow you to replace values where certain conditions are met.


```
df['Salary_Category'] = df['Salary'].mask(df['Salary'] <= 50000, 'Low')
df['Salary_Category'] = df['Salary_Category'].mask(df['Salary_Category'] != 'Lo
```

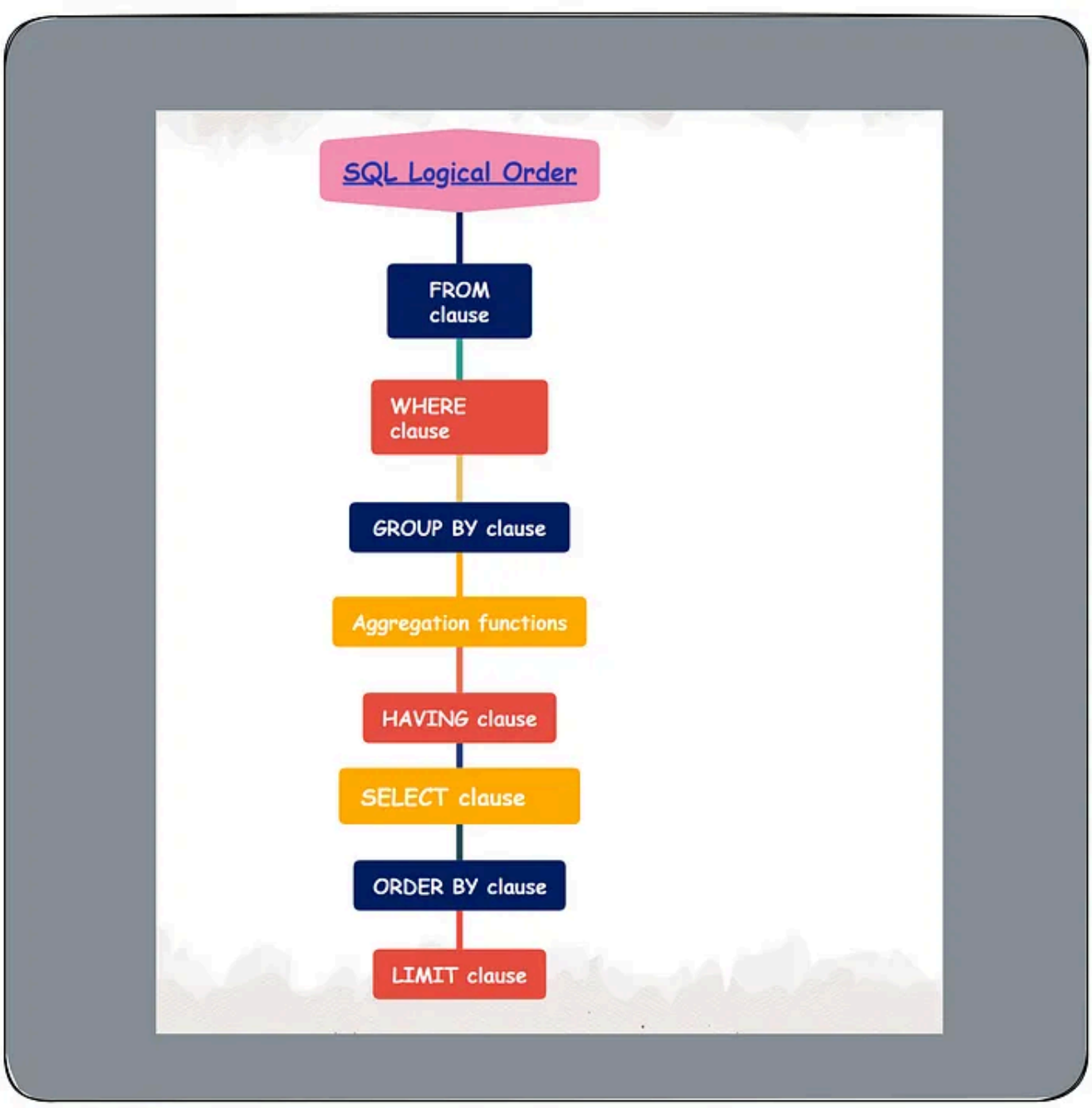


Image by the Author

Date and Time

Handling date and time in SQL:

SQL provides several functions to handle date and time data types, including `GETDATE()`, `DATEADD()`, `DATEDIFF()`, `DAY()`, `MONTH()`, `YEAR()`, etc.

```
-- Get current date and time
SELECT GETDATE() AS CurrentDateTime;

-- Add 1 month to a date
SELECT DATEADD(month, 1, '2023-01-01') AS NewDate;

-- Difference between two dates in days
```

```
SELECT DATEDIFF(day, '2023-01-01', '2023-02-01') AS DiffDays;


SELECT DAY(GETDATE()) AS Day, MONTH(GETDATE()) AS Month, YEAR(GETDATE()) AS Yea
```

Handling date and time in pandas:

Pandas also provides several functions to handle date and time data, such as `to_datetime()`, `date_range()`, and attributes like `.day`, `.month`, `.year`, etc.

```
import pandas as pd

# Convert a string to datetime
df['Date'] = pd.to_datetime(df['Date'])

# Create a date range
dates = pd.date_range('2023-01-01', periods=30)


df['Day'] = df['Date'].dt.day
df['Month'] = df['Date'].dt.month
df['Year'] = df['Date'].dt.year
```

Performance

Performance considerations in SQL:

- Performance improvement through indexing and query optimization.

```
CREATE INDEX idx_employees_name ON Employees (Name);

-- Optimized query: filter before joining
SELECT *
FROM Employees e
JOIN Departments d ON e.DepartmentId = d.Id
WHERE e.Salary > 50000;
```

Performance considerations in pandas:

- Performance improvement through vectorized operations and memory management.

```
import pandas as pd
import numpy as np

# Vectorized operation: add 5000 to each salary
df['Salary'] = df['Salary'] + 5000

# Reduce memory usage by changing data types
df['Salary'] = df['Salary'].astype(np.int32)
```

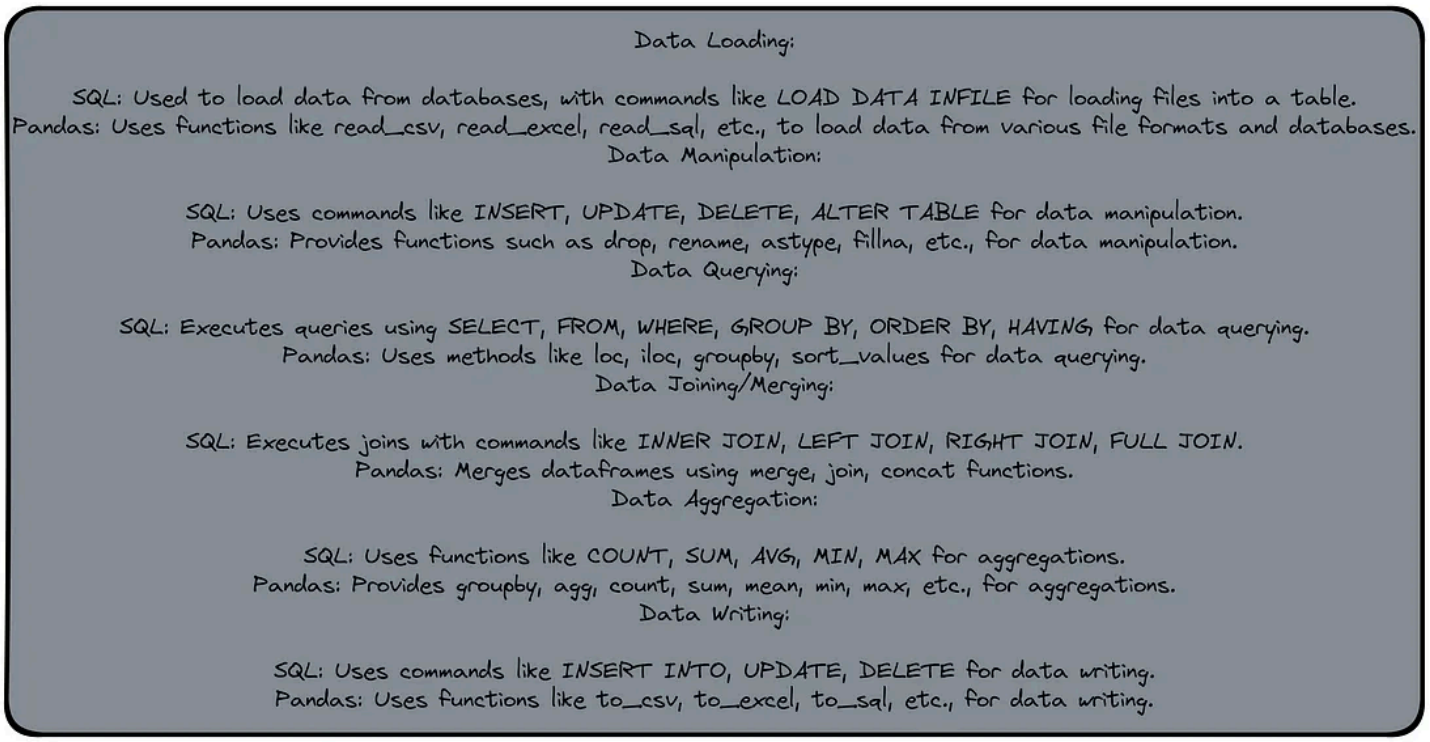


Image by the Author

Appendix: SQL and pandas Cheat Sheet

This cheat sheet provides a side-by-side comparison of common operations in SQL and pandas.

1. Select specific columns from a table (SQL)

```
SELECT name, grade FROM students;
```

1. Select specific columns from a DataFrame (Pandas)

```
df[['name', 'grade']]
```

In SQL, you specify the columns you want to select after the `SELECT` keyword. In pandas, you pass a list of column names to the indexing operator.

2. Filter rows using a condition (SQL)

```
SELECT * FROM students WHERE grade > 80;
```

2. Filter rows using a condition (Pandas)

```
df[df['grade'] > 80]
```

In SQL, you use the `WHERE` clause to filter rows that satisfy a condition. In pandas, you use boolean indexing to filter rows.

3. Sort the result set by a specific column (SQL)

```
SELECT * FROM students ORDER BY grade DESC;
```

3. Sort the DataFrame by a specific column (Pandas)

```
df.sort_values('grade', ascending=False)
```

- In SQL, you use the `ORDER BY` clause to sort the result set by a specific column. You add the `DESC` keyword to sort in descending order.
- In pandas, you use the `sort_values` method to sort the DataFrame by a specific column. Just set the `ascending` parameter to `False` to sort in descending order.

4. Group by a column and compute an aggregate function (SQL)

```
SELECT class, AVG(grade) as average_grade FROM students GROUP BY class;
```

4. Group by a column and compute an aggregate function (Pandas)

```
df.groupby('class')['grade'].mean()
```

- In SQL, you use the `GROUP BY` clause to group the result set by a column, and then you can compute aggregate functions like `AVG` on each group.
- In pandas, you use the `groupby` method to group a column, and then compute aggregate functions like `mean` on each group.

5. Join two tables on a common column (SQL)

```
SELECT students.name, grades.grade
FROM students
INNER JOIN grades ON students.id = grades.student_id;
```

5. Join two DataFrames on a common column (Pandas)

```
pd.merge(students, grades, left_on='id', right_on='student_id', how='inner')[['
```

- `JOIN` clause to combine rows from two or more tables based on a common column.
- `merge` function to combine two DataFrames based on a common column.

6. Select distinct values from a column (SQL)

```
SELECT DISTINCT class FROM students;
```

6. Select distinct values from a column (Pandas)

```
df['class'].unique()
```

- `DISTINCT` keyword to return unique values in the output.
- `unique()` function to get unique values of a Series object.

7. Count the number of rows with a specific condition (SQL)

```
SELECT COUNT(*) FROM students WHERE grade > 80;
```

7. Count the number of rows with a specific condition (Pandas)

```
(df['grade'] > 80).sum()
```

- `COUNT` function to count the number of rows that satisfies a condition.
- In pandas, you apply the condition to the DataFrame and then sum up the resulting boolean Series.

8. Calculate the sum of a column (SQL)

```
SELECT SUM(grade) FROM students;
```

8. Calculate the sum of a column (Pandas)

```
df['grade'].sum()
```

- `SUM` function to calculate the total sum of a numeric column.

- `sum()` function to calculate the total sum of a column.

9. Get the row with the maximum value of a column (SQL)

```
SELECT * FROM students ORDER BY grade DESC LIMIT 1;
```

9. Get the row with the maximum value of a column (Pandas)

```
df.loc[df['grade'].idxmax()]
```

- Order by descending order and use `LIMIT 1` to get the row with the maximum value.
- Use `idxmax()` to get the index of the maximum value, and then use `loc` to select the row.

10. Update values in a table based on a condition (SQL)

```
UPDATE students SET grade = grade + 10 WHERE name = 'John';
```

10. Update values in a DataFrame based on a condition (Pandas)

```
df.loc[df['name'] == 'John', 'grade'] += 10
```

- `UPDATE` statement to modify the values in a table based on a condition.
- `loc` to select rows and columns by label.

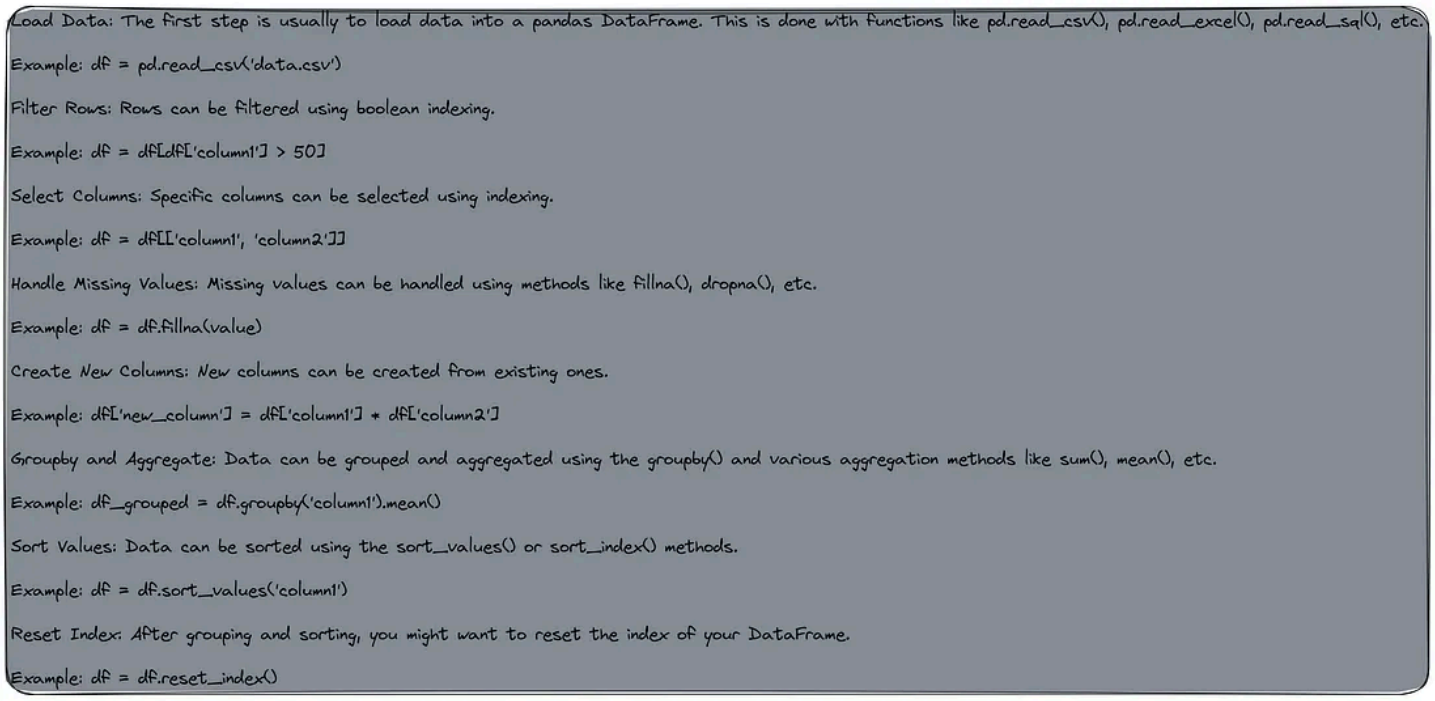


Image by the Author

11. Delete rows (SQL)

```sql
DELETE FROM students WHERE grade < 60;
```

11. Delete rows from a DataFrame (Pandas)

```python
df = df[df['grade'] >= 60]
```

- `DELETE` command to remove rows from a table based on a condition.
- boolean indexing to filter out the rows you want to delete. Creates a new dataframe.

12. Select a range of rows (SQL):

```sql
SELECT * FROM students LIMIT 10 OFFSET 20;
```

12. Select a range of rows (Pandas)

```
df.iloc[20:30]
```

- `LIMIT` and `OFFSET` keywords to select a range of rows. `LIMIT`- the maximum number of rows to return, and `OFFSET` specifies the start position.
- `iloc` indexer to select by integer-based location.

13. Check for NULL values (SQL)

```
SELECT * FROM students WHERE grade IS NULL;
```

13. Check for NULL values (Pandas)

```
df[df['grade'].isnull()]
```

- `IS NULL` or `IS NOT NULL` -SQL.
- `isnull()` or `notnull()` pandas functions

14. Insert a new row into a table (SQL)

```
INSERT INTO students (id, name, grade) VALUES (1, 'John', 85);
```

14. Append a new row to a DataFrame (Pandas)

```
df = df.append({'id': 1, 'name': 'John', 'grade': 85}, ignore_index=True)
```

- `INSERT INTO` command to insert new records into a table.

- `append()` function to add new rows to the end of the DataFrame.

15. Create a new table (SQL)

```
CREATE TABLE new_students (id INT, name VARCHAR(50), grade INT);
```

15. Create a new DataFrame (Pandas)

```
new_students = pd.DataFrame(columns=['id', 'name', 'grade'])
```

- `CREATE TABLE` -to create a new table.
- Instantiate a new DataFrame and specify the column names.

16. Drop a table (SQL)

```
DROP TABLE students;
```

16. Delete a DataFrame (Pandas)

```
del df
```

- `DROP TABLE` command to delete a table.
- `del` statement to delete a variable (in this case, a DataFrame).

17. Use `LIKE` operator to search for a specified pattern in a column (SQL)

```
SELECT * FROM students WHERE name LIKE 'John%';
```

17. Use string methods to search for a specified pattern in a column (Pandas)

```
df[df['name'].str.startswith('John')]
```

- `LIKE` operator is to search for a specified pattern in a column. The percent sign `%` can be used to define wildcards (missing letters) both before and after the pattern.
- Use string methods like `startswith()`, `endswith()`, and `contains()` in pandas.

18. Subquery inside a `SELECT` statement (SQL)

```
SELECT name, (SELECT AVG(grade) FROM grades WHERE grades.student_id = students.
FROM students;
```

18. Apply function to compute derived columns (Pandas)

```
df['average_grade'] = df['id'].apply(lambda x: grades[grades['student_id'] == x
```

- Use a subquery inside a `SELECT` statement to compute derived columns. In this case, for each student, it fetches the average grade from the `grades` table.
- In pandas, you can use the `apply()` function to achieve the same result.

19. Self-Join to compare rows within the same table (SQL)

```
SELECT A.name, B.name, A.grade
FROM students A, students B
WHERE A.grade > B.grade AND B.name = 'John';
```

19. Self-Join to compare rows within the same DataFrame (Pandas)

```
A = df.rename(columns={'name': 'A_name', 'grade': 'A_grade'})
B = df.rename(columns={'name': 'B_name', 'grade': 'B_grade'})
merged = pd.merge(A, B, left_index=True, right_index=True)
merged = merged[merged['A_grade'] > merged['B_grade']]
merged = merged[merged['B_name'] == 'John'][['A_name', 'B_name', 'A_grade']]
```

- In SQL, a self-join is a regular join, but the table is joined with itself.
- In pandas, you can achieve the same by renaming the DataFrame and then merging it with itself based on the index.

20. `HAVING` clause to filter grouped data (SQL)

```
SELECT class, AVG(grade) as average_grade
FROM students
GROUP BY class
HAVING AVG(grade) > 80;
```

20. Filter grouped data (Pandas)

```
grouped = df.groupby('class')['grade'].mean().reset_index()
grouped = grouped[grouped['grade'] > 80]
```

- The `HAVING` -is used to filter (where clause in select) the results of a `GROUP BY` query.
- In pandas, you can achieve the same by performing a group operation and then applying a filter to the result.

21. `UNION` -combine `SELECT` statements (SQL)

```
SELECT name FROM students WHERE grade > 80
UNION
SELECT name FROM teachers;
```

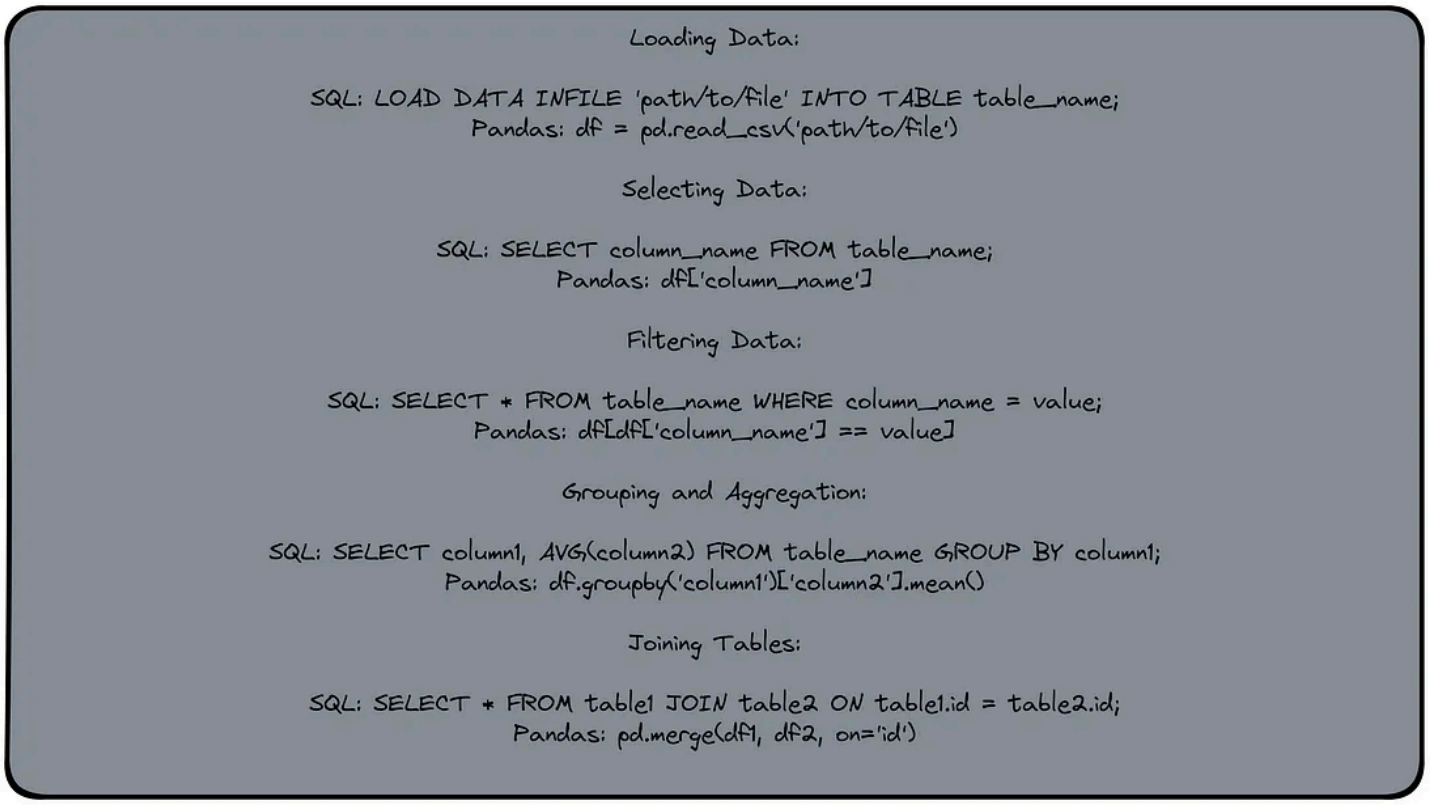


Image by the Author

21. Concatenate two or more DataFrames (Pandas)

```
pd.concat([df[df['grade'] > 80]['name'], teachers['name']], ignore_index=True)
```

- The SQL `UNION` — to combine the result set of two or more `SELECT` statements.
- Must have the same number of columns.
- The columns must have similar data types and same order.
- In pandas, you can achieve the same result by using the `concat()` function.

22. `CASE` statement for conditional logic (SQL)

```
SELECT name,
       CASE
           WHEN grade >= 90 THEN 'A'
           WHEN grade >= 80 THEN 'B'
           ELSE 'C'
```

```
        END as grade_letter
FROM students;
```

22. Use `apply()` and `lambda` for conditional logic (Pandas)

```
def get_grade_letter(grade):
    if grade >= 90:
        return 'A'
    elif grade >= 80:
        return 'B'
    else:
        return 'C'

df['grade_letter'] = df['grade'].apply(get_grade_letter)
```

- The `CASE` statement for conditional logic.
- In pandas, you can use the `apply()` function with a lambda function or a standalone function for conditional logic.

23. "`IN`" operator to specify multiple values in a `WHERE` clause (SQL)

```
SELECT * FROM students WHERE name IN ('John', 'Jane', 'Doe');
```

23. Use `isin()` for filtering on multiple values (Pandas)

```
df[df['name'].isin(['John', 'Jane', 'Doe'])]
```

- In SQL, you use the `IN` operator in a `WHERE` clause to specify multiple values.
- In pandas, you can use the `isin()` function to filter the DataFrame based on multiple values.

24. `BETWEEN` operator to filter values within a specific range (SQL)

```
SELECT * FROM students WHERE grade BETWEEN 80 AND 90;
```

24. Use boolean indexing for filtering values within a specific range (Pandas)

```
df[(df['grade'] >= 80) & (df['grade'] <= 90)]
```

- In SQL, you use the `BETWEEN` operator in a `WHERE` clause to filter values within a specific range.
- In pandas, you can use boolean indexing for the same purpose.

25. Using `COALESCE` to handle NULL values (SQL)

```
SELECT name, COALESCE(grade, 'No grade') as grade FROM students;
```

25. Use `fillna()` to handle missing values (Pandas)

```
df['grade'] = df['grade'].fillna('No grade')
```

- In SQL, `COALESCE()` function returns the first non-null value in a list.
- In pandas, you can use the `fillna()` function to fill NA/NaN values using the specified method.

26. Create a view (SQL)

```
CREATE VIEW top_students AS
SELECT name, grade FROM students WHERE grade > 90;
```

26. Use query or boolean indexing to create a new DataFrame (Pandas)

```
top_students = df[df['grade'] > 90][['name', 'grade']]
```

- A view is a virtual table based on a SQL query.
- In pandas, you can create a new DataFrame based on a subset of another DataFrame.

27. `JOIN` two tables on a common column (SQL)

```
SELECT students.name, grades.grade
FROM students
JOIN grades ON students.id = grades.student_id;
```

27. Merge two DataFrames on a common column (Pandas)

```
pd.merge(students, grades, left_on='id', right_on='student_id')[['name', 'grade
```

- In SQL, use the `JOIN` keyword to merge rows from two or more tables, based on a common column in both the tables.
- In pandas, you use the `merge()` function to merge two DataFrames based on a common column.

28. Perform a `LEFT JOIN` (SQL)

```
SELECT students.name, grades.grade
FROM students
LEFT JOIN grades ON students.id = grades.student_id;
```

28. Perform a `LEFT JOIN` (Pandas)

```
pd.merge(students, grades, how='left', left_on='id', right_on='student_id')[['r
```

- In SQL, a `LEFT JOIN` returns all records from the left table and only the matched records from the right table.
- In pandas, you use the `merge()` function with `how='left'` to perform a `LEFT JOIN`.

29. Perform a `RIGHT JOIN` (SQL)

```
SELECT students.name, grades.grade
FROM students
RIGHT JOIN grades ON students.id = grades.student_id;
```

29. Perform a `RIGHT JOIN` (Pandas)

```
pd.merge(students, grades, how='right', left_on='id', right_on='student_id')[['
```

- In SQL, a `RIGHT JOIN` returns all the records from the right table, and only the matched records from the left table.
- In pandas, you use the `merge()` function with `how='right'` to perform a `RIGHT JOIN`.

30. Perform a `FULL JOIN` (SQL)

```
SELECT students.name, grades.grade
FROM students
FULL JOIN grades ON students.id = grades.student_id;
```

30. Perform a `FULL JOIN` (Pandas)

```
pd.merge(students, grades, how='outer', left_on='id', right_on='student_id')[['
```

- In SQL, a `FULL JOIN` returns all records when there is a match in left or right.
- In pandas, you use the `merge()` function with `how='outer'` to perform a `FULL JOIN`.

31. `GROUP BY` multiple columns (SQL)

```
SELECT class, gender, AVG(grade)
FROM students
GROUP BY class, gender;
```

31. Group by multiple columns (Pandas)

```
df.groupby(['class', 'gender'])['grade'].mean().reset_index()
```

- In SQL, you use the `GROUP BY` statement to group rows that have the same values in specified columns into aggregated data. Here, it groups by `class` and `gender` and calculates the average grade for each group.
- In pandas, you use the `groupby()` function to achieve the same result.

32. Use `LIMIT` (SQL)

```
SELECT * FROM students ORDER BY grade DESC LIMIT 5;
```

32. Use `head()` (Pandas)

```
df.sort_values(by='grade', ascending=False).head(5)
```

- In SQL, you can use the `LIMIT` keyword to return the number of records. Here, it's used to return the top 5 students by grade.
- In pandas, you can achieve the same result by using the `head()` function after sorting the DataFrame.

33. Use `AVG` (SQL)

```
SELECT AVG(grade) FROM students;
```

33. Use `mean()` (Pandas)

```
df['grade'].mean()
```

- In SQL, you use the `AVG` function .
- In pandas, you use the `mean()` function.

34. Use `MIN` and `MAX` (SQL)

```
SELECT MIN(grade), MAX(grade) FROM students;
```

34. Use `min()` and `max()` (Pandas)

```
df['grade'].min(), df['grade'].max()
```

- In SQL, you use the `MIN` and `MAX` functions to get the minimum and maximum value of a numeric column, respectively.
- In pandas, you use the `min()` and `max()` functions to get the minimum and maximum value of a numeric column, respectively.

35. `TRUNCATE TABLE` (SQL)

```
TRUNCATE TABLE students;
```

35. Dataframe.drop (Pandas)

```
df = df.drop(df.index)
```

- The SQL `TRUNCATE TABLE` statement quickly removes all data from a table.
- In pandas, you can delete all data from a DataFrame using the `drop` method.

36. `EXCEPT` (SQL)

```
SELECT column_name(s) FROM table1
EXCEPT
SELECT column_name(s) FROM table2;
```

36. Use the `isin()` function with the `~` operator for a similar operation (Pandas)

```
df1[~df1['column_name(s)'].isin(df2['column_name(s)'])]
```

- In SQL, the `EXCEPT` command is used to return all rows in the first query that doesn't have a match in the second query.
- In pandas, you can use the `isin()` function with the `~` (NOT) operator to get all rows in the first DataFrame (`df1`) that have no match in the second DataFrame (`df2`).

37. `INTERSECT` operation to return all rows that are in both the first and the second query (SQL)

```
SELECT column_name(s) FROM table1
INTERSECT
```

```
SELECT column_name(s) FROM table2;
```

37. Use `merge()` for a similar operation (Pandas)

```
pd.merge(df1, df2, on='column_name(s)')
```

- In SQL, the `INTERSECT` operator returns all rows common in table 1 and table 2.
- In pandas, you can use the `merge()` function to get all rows that are in both DataFrames (`df1` and `df2`).

38. Use `NULLIF` to replace certain values with NULL (SQL)

```
SELECT name, NULLIF(grade, 0) as grade
FROM students;
```

38. Use `replace()` to replace certain values with NULL in pandas

```
df['grade'] = df['grade'].replace(0, np.nan)
```

- The `NULLIF` function in SQL returns NULL if the two given values are equal. Here, it's used to replace grades of 0 with NULL.
- In pandas, you can use the `replace()` function to replace certain values with NULL (or `np.nan`).

39. Use `ROW_NUMBER()` , `RANK()` , `DENSE_RANK()` window functions to assign a unique rank to each row (SQL)

```
SELECT name, grade,
ROW_NUMBER() OVER (ORDER BY grade DESC) as row_number,
RANK() OVER (ORDER BY grade DESC) as rank,
```

```
DENSE_RANK() OVER (ORDER BY grade DESC) as dense_rank
FROM students;
```

39. Use `rank()` and `argsort()` to assign a unique rank to each row in pandas

```
df['row_number'] = df['grade'].argsort().argsort() + 1
df['rank'] = df['grade'].rank(method='min', ascending=False)
df['dense_rank'] = df['grade'].rank(method='dense', ascending=False)
```

- `ROW_NUMBER()` assigns a unique row number to each row disregarding any duplicate values.
- `RANK()` gives the same rank to the rows with the same values and leaves gaps for the subsequent ranks.
- `DENSE_RANK()` also gives the same rank to the rows with the same values but does not leave gaps for the subsequent ranks.
- In pandas, you can use `argsort()` to get the `row_number` and `rank()` to get the `rank` and `dense_rank`.

40. Use `GROUP_CONCAT` to concatenate the values from a group (SQL)

```
SELECT class, GROUP_CONCAT(name SEPARATOR ', ') as students
FROM students
GROUP BY class;
```

40. Use `groupby()` and `join()` to concatenate the values from a group in pandas

```
df.groupby('class')['name'].apply(', '.join).reset_index()
```

- The `GROUP_CONCAT` concatenates values from a group into a single string with a separator between values. Here, it's used to create a list of students in each class.

- In pandas, you can use `groupby()` and `join()` to concatenate the values from a group.

41. Use `PIVOT` from long to wide format (SQL)

```
SELECT *
FROM (
    SELECT name, class, grade
    FROM students
) as s
PIVOT (
    AVG(grade)
    FOR class IN ('Class A', 'Class B', 'Class C')
);
```

41. Use `pivot_table()` -from long to wide format in pandas

```
df.pivot_table(index='name', columns='class', values='grade')
```

- The `PIVOT` operator in SQL allows you to rotate rows into columns.
- In pandas, you can use `pivot_table()` to achieve a similar result.

42. Use `UNPIVOT` to transform your data from wide to long format (SQL)

```
SELECT *
FROM students
UNPIVOT (
    grade
    FOR class IN (ClassA_grade, ClassB_grade, ClassC_grade)
) as u;
```

42. Use `melt()` to transform your data from wide to long format in pandas

```
df.melt(id_vars='name', var_name='class', value_name='grade')
```

- The `UNPIVOT` operator in SQL allows you to rotate columns into rows.
- In pandas, you can use `melt()` to achieve a similar result.

43. Use `ROLLUP` to create subtotals and grand totals in your result set (SQL)

```sql
SELECT class, gender, AVG(grade) as avg_grade
FROM students
GROUP BY class, gender WITH ROLLUP;
```

43. Use `groupby()`, `unstack()`, and `concat()` to create subtotals and grand totals in pandas

```python
grouped = df.groupby(['class', 'gender'])['grade'].mean().unstack()
totals = df.groupby('class')['grade'].mean()
grand_total = df['grade'].mean()
pd.concat([grouped, totals], axis=1).append(grand_total, ignore_index=True)
```

- The `ROLLUP` operator in SQL allows you to create subtotals and grand totals in your result set.
- In pandas, you can use `groupby()`, `unstack()`, and `concat()` to achieve a similar result.

44. Use `OUTER APPLY` to apply a subquery to each row in your query and include rows where the subquery returns no results (SQL)

```sql
SELECT students.name, grades.grade
FROM students
OUTER APPLY (
    SELECT grade
    FROM grades
    WHERE grades.student_id = students.student_id
) as grades;
```

44. In pandas, you can use a combination of `apply()`, a lambda function, and `fillna()` to achieve a similar result

```
df['grades'] = df.apply(lambda row: grades_df[grades_df['student_id'] == row['s
df['grades'] = df['grades'].fillna('No grades')
```

- The `OUTER APPLY` operator in SQL allows you to apply a subquery to each row in your query and includes rows where the subquery returns no results.
- In pandas, you can use `apply()` with a lambda function and `fillna()` to achieve a similar result.

45. Use `LIMIT` and `OFFSET` to paginate your query results (SQL)

```
SELECT * FROM students
ORDER BY name
LIMIT 10 OFFSET 20;
```

46. Use `iloc[]` to paginate your DataFrame in pandas

```
df.sort_values('name').iloc[20:30]
```

- The `LIMIT` and `OFFSET` keywords in SQL make it easy to paginate your query results. Here, the query returns 10 records, starting from the 21st record.
- In pandas, you can use `iloc[]` to achieve the same result.

47.Use `SUBSTRING` to extract a substring from a string (SQL)

```
SELECT SUBSTRING(name, 1, 5) as short_name
FROM students;
```

47. Use string slicing to extract a substring from a string in pandas

```
df['short_name'] = df['name'].str[:5]
```

- The `SUBSTRING` function in SQL is used to extract a substring from a string. Here, it's used to return the first 5 characters of each student's name.
- In pandas, you can use string slicing to extract a substring from a string.

48. Use `NTILE()` to divide rows into a specified number of groups (SQL)

```
SELECT name, grade,
       NTILE(4) OVER(ORDER BY grade DESC) as quartile
FROM students;
```

49. Use the `qcut()` function to divide rows into a specified number of groups in pandas

```
df['quartile'] = pd.qcut(df['grade'], 4, labels=False)
```

- The `NTILE()` function in SQL divides rows into a specified number of groups. It's used here to divide students into quartiles based on their grades.
- In pandas, you can use the `qcut()` function to achieve a similar result.

50. Use `TRY_CONVERT` to attempt a data type conversion and return NULL if it fails (SQL)

```
SELECT TRY_CONVERT(int, grade) AS int_grade
FROM students;
```

50. Use the `to_numeric()` function with errors='coerce' to attempt a data type conversion and replace non-convertible values with NaN in pandas

```
df_students['int_grade'] = pd.to_numeric(df_students['grade'], errors='coerce')
```

- The `TRY_CONVERT` function in SQL tries to convert a value to a specified data type and returns NULL if it fails. Here, it's used to convert the grade to an integer.
- In pandas, you can use the `to_numeric()` function with errors='coerce' to achieve a similar result.

51. Use `IIF` to return one of two values, depending on whether a Boolean expression is true or false (SQL)

```
SELECT IIF(grade >= 60, 'Pass', 'Fail') as result
FROM students;
```

51. Use the `np.where()` function to return one of two values, depending on whether a condition is true or false in pandas

```
df_students['result'] = np.where(df_students['grade'] >= 60, 'Pass', 'Fail')
```

- The `IIF` function in SQL returns one of two values, depending on whether a Boolean expression is true or false.
- In pandas, you can use the `np.where()` function to achieve a similar result.

51. Use `WAITFOR DELAY` (SQL)

```
WAITFOR DELAY '00:00:05';
```

51. Use the `sleep()`

```
import time
time.sleep(5)
```

- The `WAITFOR DELAY` command in SQL is used to pause the execution of a batch for a specified amount of time.
- In Python, you can use the `sleep()` function from the time module.

52. Complex Query in SQL-I

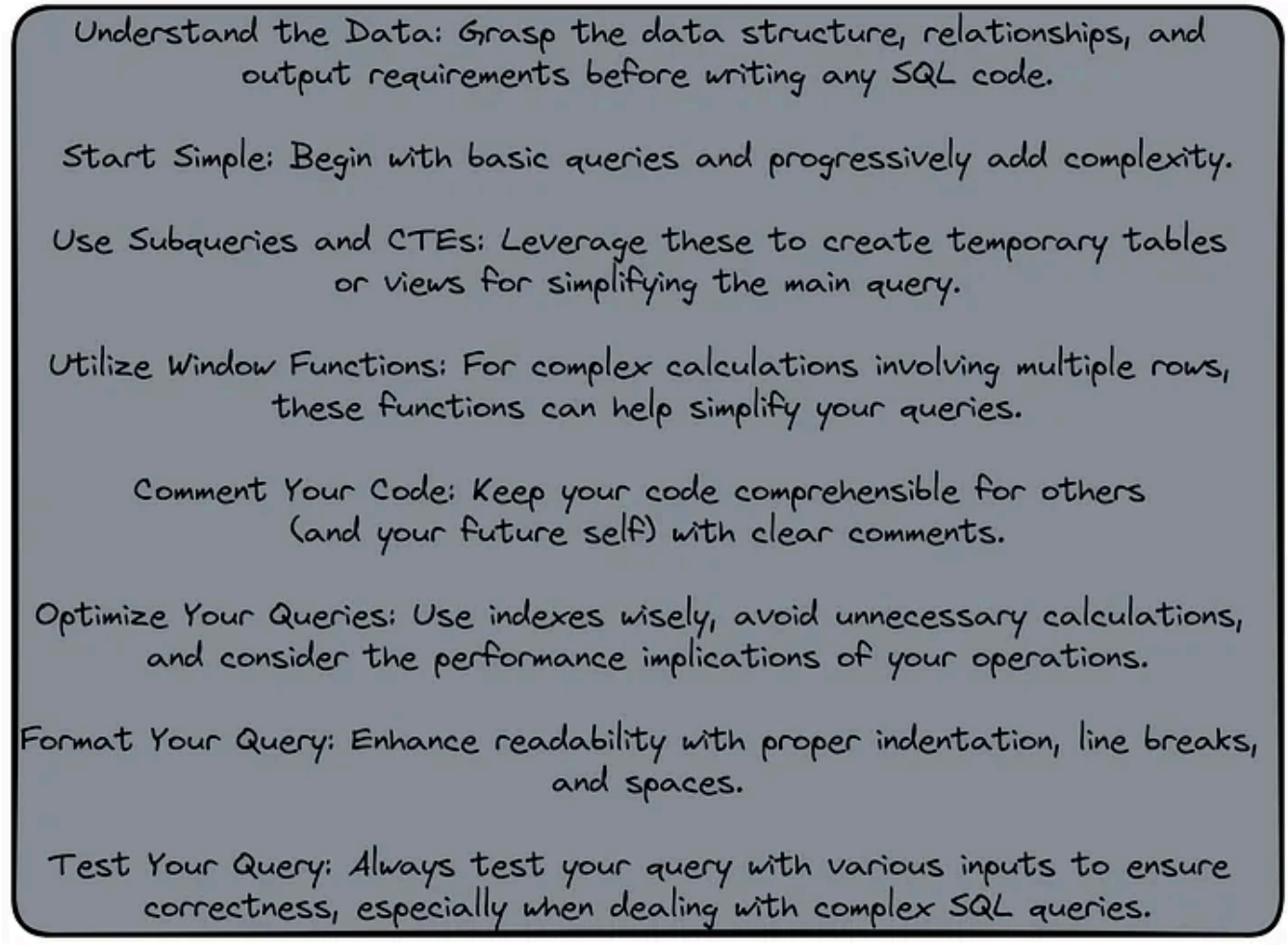


Image by the Author

```
SELECT
    orders.order_id,
    orders.customer_id,
    customers.customer_name,
    RANK() OVER (PARTITION BY customers.customer_id ORDER BY orders.order_amour
FROM
    orders
INNER JOIN
    customers ON orders.customer_id = customers.customer_id
ORDER BY
    customers.customer_id, rank;
```

- We select the `order_id` and `customer_id` from the `orders` table, and `customer_name` from the `customers` table.
- We create a new column called `rank` using the `RANK()` function. This ranks the order amounts within each customer (defined by `PARTITION BY customers.customer_id`), in descending order (`ORDER BY orders.order_amount DESC`).
- We use `INNER JOIN` to combine rows from `orders` and `customers` where the `customer_id` matches.
- We order the result by `customer_id` and `rank`.

52. Equivalent in pandas:I

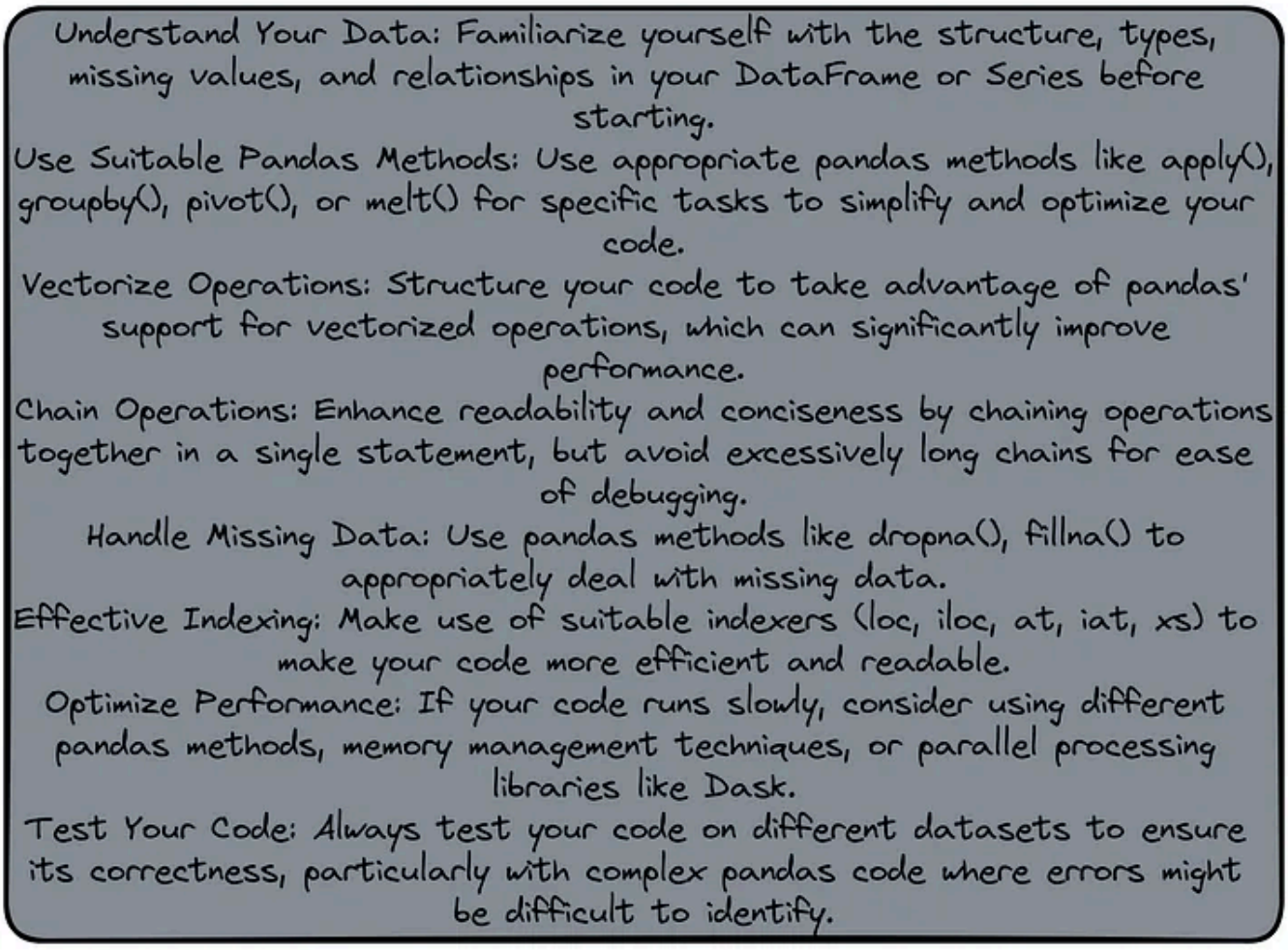


Image by the Author

```
import pandas as pd

# perform an inner join on the dataframes
```

```
merged_df = pd.merge(orders_df, customers_df, on='customer_id', how='inner')

# Then, we create a new column 'rank' using the `rank()` method
merged_df['rank'] = merged_df.sort_values('order_amount', ascending=False)\
                             .groupby('customer_id')['order_amount']\
                             .rank(method='min', ascending=False)

# Finally, we sort the dataframe by 'customer_id' and 'rank'
merged_df = merged_df.sort_values(['customer_id', 'rank'])
```

- We use `pd.merge()` to perform an inner join on `orders_df` and `customers_df` where the `customer_id` matches.
- We create a new column called `rank` using the `rank()` method. This ranks the order amounts within each customer (defined by `groupby('customer_id')`), in descending order (`sort_values('order_amount', ascending=False)`).
- We sort the result by `customer_id` and `rank` using `sort_values(['customer_id', 'rank'])`.

53.Complex Query in SQL-II

```
SELECT t1.id, t1.name, t2.total_purchases
FROM Customers AS t1
LEFT JOIN (
    SELECT CustomerID, COUNT(*) as total_purchases
    FROM Orders
    GROUP BY CustomerID
) AS t2
ON t1.id = t2.CustomerID
WHERE t2.total_purchases > 5
ORDER BY t2.total_purchases DESC;
```

- This SQL query is retrieving customers and their total purchases from two tables: `Customers` and `Orders`.
- It first creates a subquery (the part inside the parentheses) that groups orders by customer ID and counts them.
- It then performs a LEFT JOIN to associate each customer with their order count.

- The WHERE clause filters out customers who have made more than 5 purchases.
- Finally, it orders the remaining customers in descending order of their total purchases.

53.Equivalent in pandas:

```
# assuming we have two dataframes customers and orders
total_purchases = orders.groupby('CustomerID').size()
customers_with_purchases = customers.merge(total_purchases, how='left', left_or
customers_with_purchases.columns = ['id', 'name', 'total_purchases']
result = customers_with_purchases[customers_with_purchases.total_purchases > 5]
```

54.Window Functions in SQL:

```
SELECT id, name, total_purchases, RANK() OVER(ORDER BY total_purchases DESC) as
FROM Customers;
```

- Window functions are a type of function where the result for each row depends not only on the current row but also on other rows in the same result set.
- In this case, the RANK() function is used to rank customers based on their total purchases, in descending order.
- The OVER clause specifies the order of the rows in each partition.

54. Equivalent in pandas:

```
customers['rank'] = customers['total_purchases'].rank(method='min', ascending=F
```

55.Complex Aggregations in SQL:

```
SELECT department,
       AVG(salary) as average_salary,
```

```
        MAX(salary) as max_salary,
        MIN(salary) as min_salary
FROM Employees
GROUP BY department;
```

- This SQL query calculates the average, maximum, and minimum salary for each department in the Employees table.
- It uses the GROUP BY clause to group rows that have the identical values in the department column.

Equivalent in pandas:

```
df = employees.groupby('department')['salary'].agg(['mean', 'max', 'min']).rena
```

In pandas, we use the `groupby()` function to group the data, and then use the `agg()` function to perform multiple aggregations at once.

56. Complex Query III:SQL

```
SELECT c.country,
       COUNT(o.order_number) as total_orders,
       SUM(o.total_amount) as total_sales
FROM Customers c
JOIN Orders o ON c.customer_id = o.customer_id
WHERE o.order_date BETWEEN '2023-01-01' AND '2023-12-31'
GROUP BY c.country;
```

- The query is taking data from a table `Customers` (alias as `c`) and joining it with a table `Orders` (alias as `o`) on `customer_id`.
- It's filtering records with `order_date` within the year 2023.
- Then, it's grouping the records by `country` in `Customers`.
- For each country, it calculates the total number of orders and the total sales amount.

Equivalent in pandas:

```
# Ensure that 'order_date' is in datetime format
orders['order_date'] = pd.to_datetime(orders['order_date'])

# Filter records for the year 2023
orders_2023 = orders[(orders['order_date'].dt.year == 2023)]

# Merge Customers with Orders
merged_df = pd.merge(customers, orders_2023, on='customer_id', how='inner')

# Perform the necessary aggregations
result = merged_df.groupby('country').agg(
    total_orders=('order_number', 'count'),
    total_sales=('total_amount', 'sum')
)
```

- The `pd.to_datetime` function is used to ensure that `order_date` is a datetime object. This allows us to filter and extract year information.
- We filter for records from the year 2023 using boolean indexing(Where clause in SQL)
- We then merge the `customers` DataFrame with the `orders_2023` DataFrame using `pd. merge`. This is equivalent to the JOIN operation in SQL.
- We then use the `groupby` method in combination with the `agg` method to perform the required aggregations. This is equivalent to the GROUP BY and aggregation functions in the SQL query.

57. Complex Query IV:SQL

```
SELECT c.customer_id,
       CASE
           WHEN MAX(o.order_date) IS NOT NULL THEN 'Existing Customer'
           ELSE 'New Customer'
       END as customer_status
FROM Customers c
LEFT JOIN Orders o ON c.customer_id = o.customer_id
GROUP BY c.customer_id;
```

- The query is taking data from a `Customers` table (alias as `c`) and left joining it with an `Orders` table (alias as `o`) on `customer_id`.
- It's grouping the records by `customer_id` in `Customers`.
- For each customer, it checks if they have a maximum `order_date` (i.e., if they have placed any orders). If they have (`MAX(o.order_date) IS NOT NULL`), they're labeled as 'Existing Customer'. If not, they're labeled as 'New Customer'. This is done using a `CASE` statement.

Equivalent in pandas:

```
# Merge Customers with Orders
merged_df = pd.merge(customers, orders, on='customer_id', how='left')

# Perform necessary aggregations and transformations
result = merged_df.groupby('customer_id').agg(
    max_order_date=('order_date', 'max')
)

result['customer_status'] = np.where(result['max_order_date'].notnull(), 'Exist
```

- We merge the `customers` DataFrame with the `orders` DataFrame using `pd.merge`. This is equivalent to the LEFT JOIN operation in SQL.
- We then use the `groupby` method in combination with the `agg` method to find the maximum order date for each customer. This is equivalent to the `GROUP BY` and `MAX` functions in the SQL query.
- We then create a new column, `customer_status`, where if `max_order_date` is not null, we assign 'Existing Customer', otherwise 'New Customer'. This is done using the `np.where` function, which is similar to the `CASE` statement in SQL.

58.Recursive CTEs in SQL for Hierarchy:

```
WITH RECURSIVE employee_hierarchy AS (
    SELECT id, name, manager_id
    FROM Employees
    WHERE name = 'John Doe'
    UNION ALL
```

```
    SELECT e.id, e.name, e.manager_id
    FROM Employees e
    INNER JOIN employee_hierarchy eh ON eh.id = e.manager_id
)
SELECT * FROM employee_hierarchy;
```

- This query uses a recursive CTE to fetch the hierarchy of employees under 'John Doe'.
- It starts with 'John Doe' as the base case and then recursively joins the Employees table on manager_id to fetch all direct and indirect subordinates.

Equivalent in pandas:

```
def fetch_hierarchy(df, employee_name):
    hierarchy = df[df['name'] == employee_name]
    for _, employee in hierarchy.iterrows():
        hierarchy = hierarchy.append(fetch_hierarchy(df, employee['id']), ignor
    return hierarchy

hierarchy = fetch_hierarchy(employees, 'John Doe')
```

59. Ranking within groups in SQL

```
SELECT name, department, salary,
       RANK() OVER(PARTITION BY department ORDER BY salary DESC) as salary_rank
FROM Employees;
```

- This SQL query ranks employees within their respective departments based on their salary in descending order. The highest salary gets a rank of 1.
- This uses the window function RANK() in combination with the OVER clause to partition the data by department.

Equivalent in pandas:

```
employees['salary_rank'] = employees.groupby('department')['salary'].rank(metho
```

In pandas, we use the `groupby()` function to partition the data by department and then use the `rank()` function to rank the salaries.

60.Calculate Percentage of Total in SQL

```
SELECT department, COUNT(*) as num_employees,
       (COUNT(*) * 100.0 / (SELECT COUNT(*) FROM Employees)) as percentage
FROM Employees
GROUP BY department;
```

- This SQL query calculates the number of employees in each department and their percentage of the total number of employees.

Equivalent in pandas

```
df = employees['department'].value_counts(normalize=True) * 100
df = df.reset_index().rename(columns={'index': 'department', 'department': 'per
```

In pandas, we use the `value_counts()` function with `normalize=True` .

Conclusion:

- SQL is suited for -large datasets, complex queries and data manipulations. Perfectly suited for relational databases.
- Pandas is generally more suitable for smaller datasets, data exploration, and preprocessing. If your data fits in memory and comfortable with Python, using pandas can be a great choice.

References:

Books:

1. “Learning SQL: Generate, Manipulate, and Retrieve Data” by Alan Beaulieu. Amazon Link

2. “Python for Data Analysis: Data Wrangling with Pandas, NumPy, and ” by Wes McKinney. Amazon Link

YouTube Videos:

1. “SQL Full Course” by freeCodeCamp.org. YouTube Link
2. “Pandas Tutorial” by Corey Schafer. YouTube Link

Online Courses:

1. “Introduction to SQL” by Khan Academy. Link
2. “Data Wrangling with Pandas” by DataCamp. Link

Official Documentation:

1. SQL syntax from W3Schools. Link
2. pandas Documentation. Link

Level Up Coding

Thanks for being a part of our community! Before you go:

- 👏 Clap for the story and follow the author 👉
- 📰 View more content in the Level Up Coding publication
- 💰 Free coding interview course ⇒ View Course
- 🔔 Follow us: Twitter | LinkedIn | Newsletter

🚀 👉 **Join the Level Up talent collective and find an amazing job**

Python | Sql | Data Science | Machine Learning | Programming

Following

Written by Senthil E

2.7K Followers · Writer for Level Up Coding

ML/DS - Certified GCP Professional Machine Learning Engineer, Certified AWS Professional Machine learning Speciality,Certified GCP Professional Data Engineer .

More from Senthil E and Level Up Coding

Senthil E in Level Up Coding

Navigating the World of LLMs: A Beginner's Guide to Prompt Engineering-Part 2

From Basics To Advanced Techniques

32 min read · Mar 17, 2024

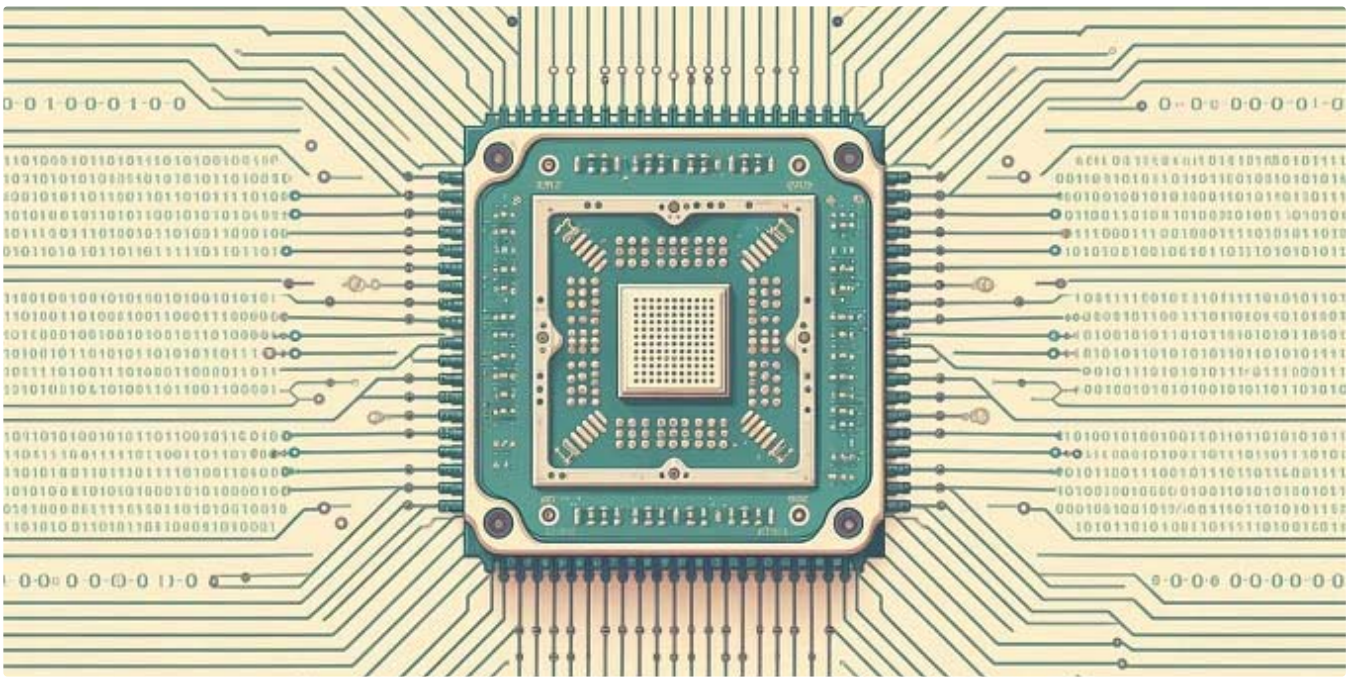
Gencay I. in Level Up Coding

Meet with DevinAI: Is This the Actual End of Coding?

Here's Why It Could Be the Final Curtain for Coding

✦ · 6 min read · Mar 17, 2024

18



Dr. Ashish Bamania in Level Up Coding

Ditch JSON! Here Are 5 (Better) Data Serialization Formats To Use In Your Next Project

Have you heard about “Cap’n Proto”, the Infinity times faster protocol?

✨ · 5 min read · Apr 1, 2024

23

Senthil E in Level Up Coding

Unleashing the Potential of LLMs: How Enterprises are Leveraging AI for Enhanced Services

From Chatbots to Automation: Exploring the Versatile Use Cases of LLMs in Enterprises

58 min read · Mar 31, 2024

See all from Senthil E

See all from Level Up Coding

Recommended from Medium

Egor Howell in Towards Data Science

How I Self-Study Data Science

My techniques and methods for learning data science and technical fields

· 8 min read · 6 days ago

3

```
*__, a, b, *_ = [1, 2, 3, 4, 5, 6]
print(__, _)
```

What does this print?

- A) Syntax error
- B) `[1] [4, 5, 6]`
- C) `[1, 2] [5, 6]`
- D) `[1, 2, 3] [6]`
- E) `<generator object <genexpr> at 0x1003847c0>`

Liu Zuo Lin

You’re Decent At Python If You Can Answer These 7 Questions Correctly

No cheating pls!!

✦ · 6 min read · Mar 6, 2024

 19 

Lists

Predictive Modeling w/ Python

20 stories · 1111 saves

Practical Guides to Machine Learning

10 stories · 1324 saves

Coding & Development

11 stories · 567 saves

General Coding Knowledge

20 stories · 1124 saves

Sze Zhong LIM in Data And Beyond

Mastering Exploratory Data Analysis (EDA): Everything You Need To Know

A systematic approach to EDA your data and prep it for machine learning.

18 min read · Apr 6, 2024

3

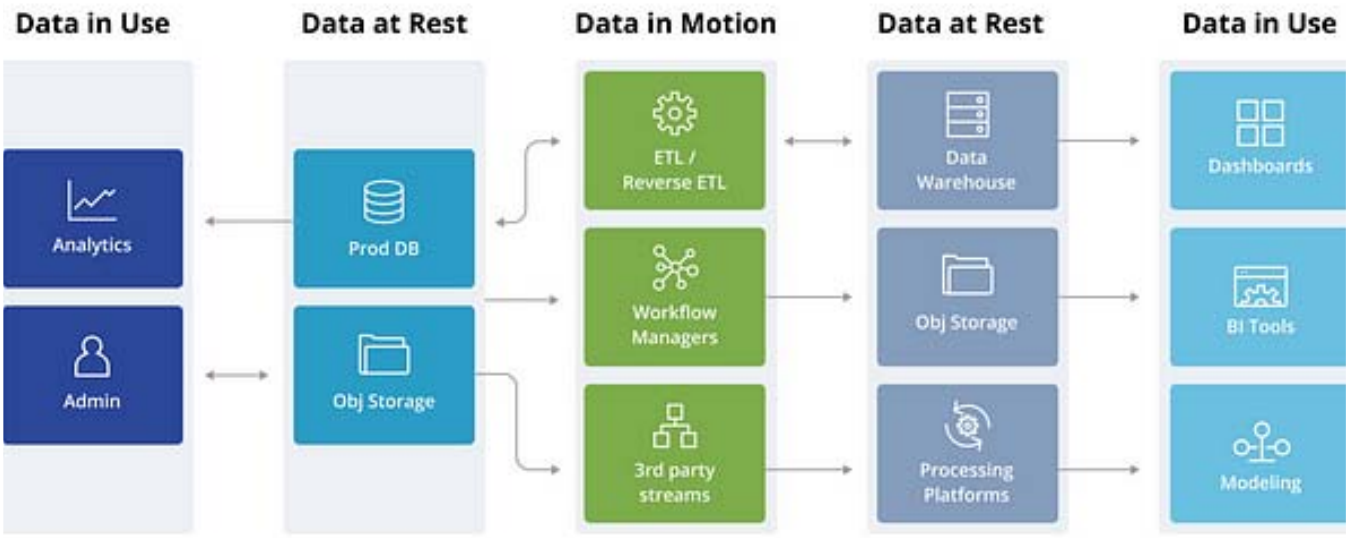
Youssef Hosni in Level Up Coding

13 SQL Statements for 90% of Your Data Science Tasks

Structured Query Language (SQL) is a programming language designed for managing and manipulating relational databases. It is widely used by...

✦ · 15 min read · Feb 26, 2023

3.8K 40



Mudra Patel

Data Engineering concepts: Part 9, Data Security

This is Part 9 of my 10 part series of Data Engineering concepts. And in this part, we will discuss about Data Security.

7 min read · Apr 8, 2024

52 1

Neelam Yadav

Ultimate Python Cheat Sheet: Practical Python For Everyday Tasks

This Cheat Sheet is a product of necessity. Tasked with re-engaging with a Python project after an extended stint in Node/Typescript...

33 min read · 6 days ago


415

See more recommendations