Polars is a high-performance DataFrame library in Python that is designed for speed, parallelism, and low memory usage. It is an alternative to Pandas, and it’s optimized for larger datasets and high-performance computing environments. Polars is written in Rust, which gives it an advantage in terms of speed and performance.

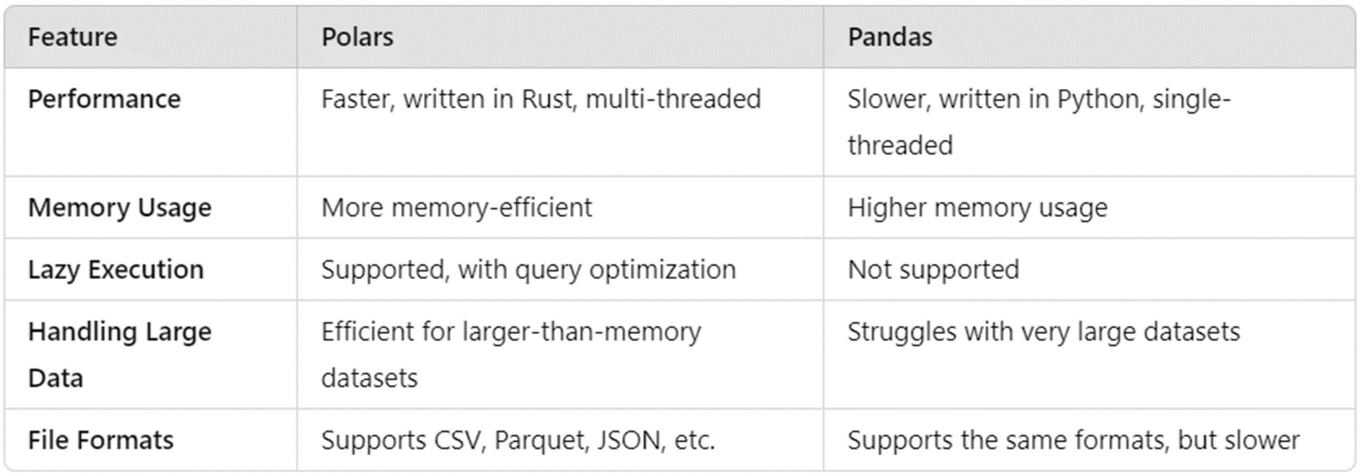
Key Features of Polars:

Fast: Polars uses a columnar memory layout and is written in Rust, making it significantly faster than Pandas for many operations.

Memory-efficient: Polars can handle larger datasets with lower memory usage due to its efficient data storage format.

Lazy execution: Polars supports lazy execution, allowing you to build query pipelines before actually executing the computations, optimizing them for speed.

Multi-threaded: Operations in Polars are parallelized to take advantage of modern CPUs with multiple cores.



Feature	Polars	Pandas
Performance	Faster, written in Rust, multi-threaded	Slower, written in Python, single-threaded
Memory Usage	More memory-efficient	Higher memory usage
Lazy Execution	Supported, with query optimization	Not supported
Handling Large Data	Efficient for larger-than-memory datasets	Struggles with very large datasets
File Formats	Supports CSV, Parquet, JSON, etc.	Supports the same formats, but slower

```
pip install openpyxl
```

```
Requirement already satisfied: openpyxl in /usr/local/lib/python3.10/dist-packages (3.1.5)
Requirement already satisfied: et-xmlfile in /usr/local/lib/python3.10/dist-packages (from openpyxl)
```

```
import pandas as pd

# Define the correct raw URL of the .xlsx file
url = "https://github.com/manishjainstorage/Advanced-Python/raw/main/StockMarketData-13krows.xlsx"

# Read the .xlsx file into a DataFrame, specifying the engine
df = pd.read_excel(url, engine='openpyxl')

# Display the first few rows of the DataFrame
df.head()
```

	date	open	high	low	close	volume	Name
0	2013-02-08	15.07	15.12	14.63	14.75	8407500	AAL
1	2013-02-11	14.89	15.01	14.26	14.46	8882000	AAL
2	2013-02-12	14.45	14.51	14.10	14.27	8126000	AAL
3	2013-02-13	14.30	14.94	14.25	14.66	10259500	AAL
4	2013-02-14	14.94	14.96	13.16	13.99	31879900	AAL

Next steps: Generate code with df   View recommended plots   New interactive sheet

```
import pandas as pd
import polars as pl
import time

# Load large dataset using Pandas
start_time = time.time()
df_pandas = pd.read_excel("https://github.com/manishjainstorage/Advanced-Python/raw/main/StockMarketData-
print(f'Pandas load time: {time.time() - start_time} seconds')

# Load large dataset using Polars
start_time = time.time()
df_polars = pl.read_excel("https://github.com/manishjainstorage/Advanced-Python/raw/main/StockMarketData-
print(f'Polars load time: {time.time() - start_time} seconds')
```

```
Pandas load time: 1.7088878154754639 seconds
Polars load time: 0.4479076862335205 seconds
```

1. Lazy Evaluation

Polars: Polars supports lazy evaluation, which allows the computation graph to be optimized before execution. This can lead to significant performance improvements when dealing with large datasets.

Pandas: Does not support lazy evaluation. All operations in pandas are executed immediately

```
import polars as pl

# Create a lazy frame
df = pl.scan_csv('https://github.com/manishjainstorage/Advanced-Python/raw/main/StockMarketData-13krows_

# Define transformations
df_lazy = df.filter(pl.col('open') > 15).select(pl.col('close') * 2)

# Check the logical plan (computation graph)
print(df_lazy.explain())

# Execute the computation graph
df_eager = df_lazy.collect()
df_eager
```

```
SELECT [[(col("close")) * (2.0)]] FROM
 Csv SCAN [https://github.com/manishjainstorage/Advanced-Python/raw/main/StockMarketData-13krows_cs
 PROJECT 2/7 COLUMNS
 SELECTION: [(col("open")) > (15.0)]
shape: (13_862, 1)
close
  f64
 29.5
 31.0
31.82
 32.5
31.96
   …
82.06
82.784
80.76
 81.2
81.84
```

2. Efficient Parallel and Multi-threaded Execution

**Polars:* *Polars can automatically parallelize operations across multiple threads, making it more efficient for CPU-bound tasks and large datasets.

**Pandas:* *Pandas operations are single-threaded, which can lead to slower performance for large-scale computations.

```
import pandas as pd
import polars as pl
import numpy as np
import time

# Define the size of the dataset
N = 10**7  # 10 million rows

# Generate a large dataset
data = {
    'a': np.random.rand(N),
    'b': np.random.rand(N)
}

# Function to perform a computation in Pandas
def pandas_computation(data):
    df = pd.DataFrame(data)
    # Perform a computation (e.g., summing the product of two columns)
    result = (df['a'] * df['b']).sum()
    return result
```

```
# Function to perform a computation in Polars
def polars_computation(data):
    df = pl.DataFrame(data)
    # Perform a computation (e.g., summing the product of two columns)
    result = (df['a'] * df['b']).sum()
    return result

# Measure execution time for Pandas
start_time = time.time()
pandas_result = pandas_computation(data)
pandas_time = time.time() - start_time
print(f"Pandas result: {pandas_result}, Time taken: {pandas_time:.2f} seconds")

# Measure execution time for Polars
start_time = time.time()
polars_result = polars_computation(data)
polars_time = time.time() - start_time
print(f"Polars result: {polars_result}, Time taken: {polars_time:.2f} seconds")
```

```
Pandas result: 2500350.209496817, Time taken: 0.24 seconds
Polars result: 2500350.2094968148, Time taken: 0.16 seconds
```

Explanation

Data Generation:

We generate a large dataset with 10 million rows, containing two columns (a and b) filled with random numbers.

Pandas Computation:

The function pandas_computation(data) creates a Pandas DataFrame and computes the sum of the product of columns a and b. This operation runs in a single-threaded manner.

Polars Computation:

The function polars_computation(data) creates a Polars DataFrame and performs the same computation. Polars automatically parallelizes this operation, leveraging multiple threads for improved performance.

Execution Time Measurement:

We measure the execution time for both the Pandas and Polars computations using the time module.

Output:

The code prints the result of the computation along with the time taken for each library.

Expected Results

When you run this code, you should observe that:

Polars generally takes significantly less time compared to Pandas for large datasets due to its ability to utilize multiple threads for parallel execution.

The exact time may vary based on your machine's CPU capabilities and the current load, but Polars should consistently outperform Pandas in this example.

This demonstrates the efficiency of Polars in handling large-scale computations through parallel and multi-threaded execution compared to the single-threaded execution of Pandas.

3. Expressions API

Polars: Polars provides an advanced Expressions API that allows for more flexible and powerful transformations without intermediate DataFrame creation. It can chain multiple transformations efficiently.

Pandas: Operations in pandas are less optimized for chaining and often involve creating multiple intermediate DataFrames, which can lead to performance bottlenecks

```
df = pl.DataFrame({
    "a": [1, 2, 3],
    "b": [4, 5, 6],
    "c": [7, 8, 9]
})

# Apply a chain of expressions
df = df.with_columns([
    (pl.col("a") * pl.col("b")).alias("a_times_b"),
    (pl.col("c") + 5).alias("c_plus_5")
])
```

```
df
```

shape: (3, 5)

a	b	c	a_times_b	c_plus_5
i64	i64	i64	i64	i64
1	4	7	4	12
2	5	8	10	13
3	6	9	18	14

4. Dynamic Window Functions

Polars: Polars has dynamic window functions, which allow for powerful time-based aggregations and rolling window computations.

Pandas: Although pandas supports rolling windows, Polars offers more flexibility and performance, especially with dynamic windows.

```
df = pl.DataFrame({
    "time": [1, 2, 3, 4, 5],
    "value": [10, 20, 30, 40, 50]
})

# Dynamic window function with a window size of 2
df = df.with_columns([
    pl.col("value").rolling_sum(window_size=2).alias("rolling_sum")
])

df
```

shape: (5, 3)

time	value	rolling_sum
i64	i64	i64
1	10	null
2	20	30
3	30	50
4	40	70
5	50	90

High-Performance Computing in Python

High-Performance Computing (HPC) in Python allows for leveraging the power of parallelism and distributed computing to solve complex and large-scale computational problems efficiently. Below are key concepts, tools, and best practices to implement HPC in Python:

Key Concepts

Parallel Computing: Dividing a task into smaller sub-tasks that can be executed simultaneously across multiple processors or machines.

Distributed Computing: Involves multiple computers working together to solve a problem, often over a network.

Concurrency: Managing multiple computations at the same time, which may involve parallel execution but does not necessarily mean they execute simultaneously.

Vectorization: Using libraries that leverage low-level optimizations to perform operations on entire arrays or data structures at once, which is often more efficient than looping.

Vectorization techniques and **Parallel processing with multiprocessing and concurrent.futures**

```
""" NumPy Array Operations

NumPy is the fundamental package for numerical computing in Python.
It provides powerful array objects and functions that support vectorization """

import numpy as np

# Create two large NumPy arrays
a = np.random.rand(1000000)
b = np.random.rand(1000000)

# Element-wise addition (vectorized operation)
c = a + b  # This operation is executed in C, making it faster than a Python loop
print(c)
```

```
[1.0881665  1.38354092 0.69107068 ... 0.72146779 1.61068822 1.43944195]
```

```
"""Broadcasting

Broadcasting allows NumPy to perform arithmetic operations on arrays of
different shapes without the need for explicit loops."""

import numpy as np

# Create a 1D array and a 2D array
a = np.array([1, 2, 3])
b = np.array([[10], [20], [30]])

# Broadcasting the 1D array to match the shape of the 2D array
result = a + b
print(result)
```

```
[[11 12 13]
 [21 22 23]
 [31 32 33]]
```

```
"""Using Numba
```

```
Numba is a just-in-time (JIT) compiler that translates a subset of Python and NumPy
code into fast machine code.

It can significantly speed up array operations."""

import numpy as np
from numba import jit

@jit(nopython=True)
def add_arrays(a, b):
    return a + b

# Create large arrays
a = np.random.rand(1000000)
b = np.random.rand(1000000)

# Call the JIT-compiled function
c = add_arrays(a, b)
print(c)
```

```
[1.81199835 0.39919907 1.25690862 ... 0.34071594 1.17085443 1.05272053]
```

```
"""SciPy for Scientific Computing

SciPy builds on NumPy and provides additional functionality for optimization,
 integration, interpolation,
and other scientific computations, often with vectorized implementations."""

from scipy import integrate
import numpy as np

# Define a function
def f(x):
    return x**2

# Vectorized integration
result, error = integrate.quad(f, 0, 1)  # Integrates f(x) from 0 to 1
print(result, error)
```

```
0.33333333333333337 3.700743415417189e-15
```

```
"""Dask for Out-of-Core Computing

Dask is a parallel computing library that integrates with NumPy and pandas to
handle larger-than-memory datasets efficiently,
 with support for lazy evaluations and parallelism."""

import dask.array as da

# Create a large Dask array
x = da.random.random((10000, 10000), chunks=(1000, 1000))

# Perform a vectorized operation
y = x + x.T  # Transpose and add
result = y.compute()  # Compute the result
print(result)
```

```
[[0.40631271 1.59814771 0.86065558 ... 0.81201027 1.09129456 1.29148079]
 [1.59814771 0.14809533 0.77080944 ... 1.58567307 0.90198611 1.13361025]
 [0.86065558 0.77080944 0.65898193 ... 0.89693504 1.07957005 1.92099078]
 ...
 [0.81201027 1.58567307 0.89693504 ... 0.74616648 1.16125423 0.77108196]
 [1.09129456 0.90198611 1.07957005 ... 1.16125423 1.86125274 0.59751993]
 [1.29148079 1.13361025 1.92099078 ... 0.77108196 0.59751993 0.47497537]]
```

NumPy optimization strategies

Optimizing performance with NumPy is essential for efficient numerical computing in Python, especially when working with large datasets or complex calculations. Here are several strategies and best practices for optimizing NumPy code:

```
""" Utilize Vectorization

Description: Replace explicit loops with NumPy's built-in array operations,
which are optimized and run in compiled C code"""

import numpy as np

# Instead of this:
a = np.random.rand(1000000)
b = np.random.rand(1000000)
c = np.empty_like(a)
for i in range(len(a)):
    c[i] = a[i] + b[i]

# Use vectorized operations:
c = a + b  # Faster and more concise



"""Use Broadcasting

Description: Broadcasting allows NumPy to perform operations on arrays of different
 shapes without creating large temporary arrays."""

import numpy as np

a = np.array([1, 2, 3])
b = np.array([[10], [20], [30]])

# Broadcasting
result = a + b  # Automatically expands a to match the shape of b



"""Pre-allocate Arrays

Description: When creating arrays, pre-allocate them with the desired shape
and size to avoid resizing during operations. """
```

```
# Instead of dynamically growing an array:
result = []
for i in range(1000):
    result.append(i**2)
result = np.array(result)  # Conversion to NumPy array afterward

# Pre-allocate:
result = np.empty(1000)
for i in range(1000):
    result[i] = i**2



"""Avoid Unnecessary Copies

Description: Be mindful of operations that create copies of arrays
(e.g., slicing or reshaping). Use views when possible."""

# Slicing creates a view, while some operations create copies
a = np.array([1, 2, 3, 4, 5])
b = a[1:3]  # b is a view, changes in a reflect in b

# Use np.copy() if you need an independent copy
c = np.copy(a[1:3])  # Creates a new array



""" Use In-place Operations

Description: Modify existing arrays in place to save memory and improve performance."""

a = np.array([1, 2, 3])
a += 1  # In-place addition, modifies a directly




"""Choose the Right Data Type

Description: Use the most efficient data types for your data to save memory and improve performance.
For instance, use float32 instead of float64 when high precision is not needed."""

a = np.array([1, 2, 3], dtype=np.float32)  # Using float32 instead of float64




"""Leverage NumPy Functions

Description: Use NumPy's built-in functions for mathematical operations,
which are optimized for performance."""

a = np.random.rand(1000000)
mean_value = np.mean(a)  # Use NumPy's mean function



"""Profile Your Code

Description: Use profiling tools like cProfile or line_profiler to identify bottlenecks
```

```
 in your code and optimize those specific areas."""

import cProfile

def compute():
    a = np.random.rand(1000000)
    b = np.random.rand(1000000)
    return a + b

cProfile.run('compute()')
```

```
         6 function calls in 0.029 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.005    0.005    0.029    0.029 <ipython-input-76-a0af4d21ffa1>:8(compute)
        1    0.000    0.000    0.029    0.029 <string>:1(<module>)
        1    0.000    0.000    0.029    0.029 {built-in method builtins.exec}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
        2    0.025    0.012    0.025    0.012 {method 'rand' of 'numpy.random.mtrand.RandomState' ob
```

```
""" Avoid Global Variables

Description: Using global variables can slow down performance in NumPy.
Prefer passing arrays as function arguments."""

# Slow with global variable
global_array = np.array([1, 2, 3])

def process():
    return global_array * 2  # Slower access

# Better approach
def process(arr):
    return arr * 2

result = process(np.array([1, 2, 3]))
```

Profiling and Optimization

Profiling and optimization are crucial steps in improving the performance of your Python code, particularly when dealing with numerical computations or data processing tasks. Below is a guide on how to effectively profile your Python code to identify bottlenecks and subsequently optimize those areas for better performance.

```
"""1. Profiling Techniques

a. Using cProfile

cProfile is a built-in Python module for profiling your code.
 It provides a way to see where the time is being spent in your program."""

import cProfile
import pstats

def compute():
    # Simulate a heavy computation
    total = 0
    for i in range(1000000):
        total += i ** 2
    return total

# Profile the compute function
cProfile.run('compute()', 'output.stats')

# Load stats and print in a readable format
with open('output.txt', 'w') as f:
    p = pstats.Stats('output.stats', stream=f)
    p.sort_stats('cumulative').print_stats()



"""
b. Using Line Profiler

line_profiler is a third-party module that provides line-by-line profiling of functions.

"""

# Step 1: Install Line Profiler
!pip install line_profiler

# Step 2: Load the Line Profiler Extension
%load_ext line_profiler

# Step 3: Define the function to profile
def my_function():
    total = 0
    for i in range(10000):
        total += i ** 2
    return total

# Run the function first (optional)
my_function()

# Step 4: Use lprun to profile the function
%lprun -f my_function my_function()
```

```
Requirement already satisfied: line_profiler in /usr/local/lib/python3.10/dist-packages (4.1.3)
The line_profiler extension is already loaded. To reload it, use:
  %reload_ext line_profiler
```

```
# Step 1: Install required libraries
!pip install line_profiler

# Step 2: Load the Line Profiler Extension
%load_ext line_profiler

# Step 3: Define the function to profile
def my_function():
    total = 0
    for i in range(10000):
        total += i ** 2
    return total

# Step 4: Create a Line Profiler instance
from line_profiler import LineProfiler

# Create a profiler
profiler = LineProfiler()
profiler.add_function(my_function)

# Run the function under the profiler
profiler.run('my_function()')

# Step 5: Get profiling results as a string
profiling_results = io.StringIO()
profiler.print_stats(stream=profiling_results)
profiling_results = profiling_results.getvalue()

# Step 6: Save the profiling results to a text file
txt_file_path = "/content/profiling_results.txt"
with open(txt_file_path, "w") as txt_file:
    txt_file.write(profiling_results)

# Step 7: Download the text file
from google.colab import files
files.download(txt_file_path)
```

```
"""
c. Memory Profiling

To profile memory usage, you can use memory_profiler. This is useful to identify memory bottlenecks.

"""
```

```
# Step 1: Install the memory_profiler package
!pip install memory_profiler

# Step 2: Load the memory_profiler extension
%load_ext memory_profiler
```

```
# Step 3: Define a function to profile
def my_function():
    """Function that consumes memory by creating a large list."""
    large_list = [i for i in range(10**6)]  # Create a large list
    return sum(large_list)

# Step 4: Use memory_profiler's memory usage monitoring manually
from memory_profiler import memory_usage

# Function to monitor memory usage while executing my_function
def profile_memory():
    mem_usage = memory_usage((my_function,), interval=0.1)  # Capture memory usage
    return mem_usage

# Step 5: Call the profiling function and print memory usage
mem_usage = profile_memory()
print(f"Memory usage (in MiB): {mem_usage}")







"""
2. Analyzing Profiling Results

After profiling, you'll get insights into:

Function Call Counts: How many times each function was called.
Cumulative Time: Total time spent in a function and all its sub-functions.
Per-Call Time: Average time spent in each call.
By analyzing these metrics, you can identify functions that take the most time or are
called frequently and may need optimization.


"""

import cProfile
import pstats
import io
import numpy as np

# Step 1: Define a function to profile
def my_function():
    """Function that performs some computations."""
    total = 0
    for i in range(1, 10000):
        total += np.sqrt(i)  # Some heavy computation
    return total

# Step 2: Profile the function using cProfile
def profile_my_function():
    pr = cProfile.Profile()
    pr.enable()  # Start profiling

    my_function()  # Call the function to be profiled
    pr.disable()  # Stop profiling

    # Step 3: Create a stream to capture the output
    s = io.StringIO()
    sortby = pstats.SortKey.CUMULATIVE
    ps = pstats.Stats(pr, stream=s).sort_stats(sortby)

    # Step 4: Print profiling results to the stream
    ps.print_stats()

    # Step 5: Get the profiling results as a string
    profiling_results = s.getvalue()

    # Return profiling results for further analysis
    return profiling_results

# Step 6: Analyze the profiling results
profiling_results = profile_my_function()

# Print the profiling results
print("Profiling Results:")
print(profiling_results)

# Optionally, you can save the results to a text file
with open("profiling_results.txt", "w") as file:
    file.write(profiling_results)



""" 3. Code Optimization techniques

Once you identify bottlenecks, consider the following optimization strategies:

a. Optimize Algorithms

Choose Efficient Algorithms: Evaluate whether a more efficient algorithm can be used.

Algorithm Complexity: Analyze the time complexity of your algorithms and look for improvements.
b. Utilize Vectorization

Replace explicit loops with vectorized operations using libraries like NumPy."""

import numpy as np
import time

# Original algorithm: O(n^2) complexity
def sum_of_squares_nested(n):
    total = 0
    for i in range(n):
        for j in range(n):
            total += i**2
    return total

# Optimized algorithm: O(n) complexity
def sum_of_squares_optimized(n):
    return n * (n - 1) * (2 * n - 1) // 6  # Formula for sum of squares
```

```
# Vectorized operation using NumPy
def sum_of_squares_vectorized(n):
    arr = np.arange(n)
    return np.sum(arr**2)

# Testing the algorithms
n = 10000

# Measure time for original algorithm
start_time = time.time()
result_nested = sum_of_squares_nested(n)
end_time = time.time()
print(f"Nested Sum of Squares Result: {result_nested}, Time: {end_time - start_time:.5f} seconds")

# Measure time for optimized algorithm
start_time = time.time()
result_optimized = sum_of_squares_optimized(n)
end_time = time.time()
print(f"Optimized Sum of Squares Result: {result_optimized}, Time: {end_time - start_time:.5f} seconds")

# Measure time for vectorized algorithm
start_time = time.time()
result_vectorized = sum_of_squares_vectorized(n)
end_time = time.time()
print(f"Vectorized Sum of Squares Result: {result_vectorized}, Time: {end_time - start_time:.5f} seconds'
```

Caching and Memoization

Caching and memoization are optimization techniques used to enhance performance by storing the results of expensive function calls and reusing them when the same inputs occur again.

Caching refers to storing the results of expensive computations, typically in a way that is easily accessible. This is useful when the same computation might be needed multiple times. In Python, caching can be implemented using decorators like functools.lru_cache.

Key Differences

Scope:

Caching is generally used for any function that may need to store results, regardless of whether it's recursive.

Memoization specifically targets recursive functions to avoid redundant calculations.

Implementation:

Caching is often implemented using decorators like lru_cache in Python.

Memoization typically involves a custom wrapper function to handle the cache.

Use Cases

Caching:

Web applications that fetch data from a database or API frequently. Image processing applications that perform repeated transformations.

Memoization:

Recursive algorithms like Fibonacci series, factorial calculations, or other dynamic programming problems where overlapping subproblems occur

```
import time
from functools import lru_cache

@lru_cache(maxsize=None)  # Cache results without a size limit
def expensive_function(n):
    """Simulates an expensive computation."""
    time.sleep(2)  # Simulate a delay
    return n * n

# Using the cached function
start_time = time.time()
print(expensive_function(4))  # First call, computes the value
print(f"Time taken: {time.time() - start_time} seconds")

start_time = time.time()
print(expensive_function(4))  # Second call, retrieves from cache
print(f"Time taken: {time.time() - start_time} seconds")
```

```
16
Time taken: 2.003897190093994 seconds
16
Time taken: 0.00017333030700683594 seconds
```

Memoization

*Memoization is a specific form of caching that is used primarily to optimize recursive functions by storing the results of expensive function calls and returning the cached result when the same inputs occur again. This is especially useful in algorithms like Fibonacci series computation *

```
def memoize(func):
    cache = {}

    def wrapper(n):
        if n not in cache:
            cache[n] = func(n)
        return cache[n]

    return wrapper

@memoize
def fibonacci(n):
    """Calculates the nth Fibonacci number."""
    if n <= 1:
        return n
    return fibonacci(n - 1) + fibonacci(n - 2)

# Using the memoized Fibonacci function
print(fibonacci(10))  # Computes the value
print(fibonacci(10))  # Retrieves from cache
```

```
55
55
```

joblib is a library in Python that provides tools for lightweight pipelining in Python and is particularly useful for parallel computing and caching. One of its key features is the ability to easily cache results of function calls using the Memory class. This is especially useful for functions that perform expensive computations or data processing, as it allows you to save and reuse the results without recalculating them.

Benefits of Using Joblib for Caching

Persistence: Cached results are stored on disk, which means you can reuse them across different sessions of your script.

Parallel Computing: joblib also supports parallel processing, allowing you to distribute computations across multiple CPU cores efficiently.

Ease of Use: Adding caching to functions is straightforward with decorators.

```
"""
Below is an example demonstrating how to cache results using joblib.Memory.
This will save the results of an expensive function call to disk,
 so it can be reused in subsequent calls with the same arguments.
"""

from joblib import Memory
import time

# Create a Memory object to store cache results
memory = Memory('./cachedir', verbose=0)

@memory.cache
def expensive_function(n):
    """Simulates an expensive computation."""
    time.sleep(2)  # Simulate a delay
    return n * n

# Using the cached function
start_time = time.time()
print(expensive_function(4))  # First call, computes the value
print(f"Time taken: {time.time() - start_time} seconds")

start_time = time.time()
print(expensive_function(4))  # Second call, retrieves from cache
print(f"Time taken: {time.time() - start_time} seconds")
```

```
start_time = time.time()
print(expensive_function(5))  # Computes a different value
print(f"Time taken: {time.time() - start_time} seconds")
```

```
16
Time taken: 2.0046210289001465 seconds
16
Time taken: 0.0019807815551757812 seconds
25
Time taken: 2.004910707473755 seconds
```

Distributed Computing

Distributed computing is a field of computer science that involves multiple computers working together to solve complex problems or process large datasets. In a distributed computing system, tasks are distributed across multiple machines, which collaborate to achieve a common goal. This approach can significantly enhance performance, fault tolerance, and scalability.

Key Concepts of Distributed Computing

Distributed Systems:

A collection of independent computers that appears to its users as a single coherent system. These computers can communicate and coordinate their actions through a network.

Parallel Processing:

This involves dividing a task into smaller sub-tasks that can be processed simultaneously across multiple processors or nodes, reducing the overall computation time.

Scalability:

Distributed systems can be scaled by adding more machines to the network, allowing them to handle increased workloads without a complete redesign of the system.

Fault Tolerance:

Distributed systems can continue to operate even if one or more nodes fail. This is achieved through redundancy and replication of data and services.

Load Balancing:

Distributing workloads evenly across nodes to prevent any single node from becoming a bottleneck.

Communication:

Nodes in a distributed system communicate with each other using various protocols (e.g., HTTP, TCP/IP) and mechanisms (e.g., message queues, RPC).

Common Distributed Computing Frameworks

Apache Hadoop:

A framework for processing and storing large datasets across clusters of computers using the MapReduce programming model.

Apache Spark:

A unified analytics engine for big data processing, with built-in modules for streaming, SQL, machine learning, and graph processing.

Dask:

A flexible library for parallel computing in Python that scales from a single machine to large clusters. It integrates well with NumPy, Pandas, and other Python libraries.

Ray:

A distributed execution framework that makes it easy to build and scale applications in Python. It's particularly useful for machine learning and reinforcement learning tasks.

TensorFlow and PyTorch:

Both frameworks support distributed training of machine learning models, allowing computations to be performed across multiple GPUs or nodes.

```
!pip install dask[complete]



import dask.array as da
import dask

# Create a large Dask array
x = da.random.random((10000, 10000), chunks=(1000, 1000))

# Perform a computation (e.g., mean)
mean_result = x.mean()

# Trigger the computation
result = mean_result.compute()

print(f"The mean of the array is: {result}")

"""
Dask Array:

A Dask array is created, simulating a large 10,000 x 10,000 array with random values.
The array is divided into smaller chunks (1,000 x 1,000) for distributed processing.

Computation:

The mean of the Dask array is computed. The actual computation is not performed until the
compute() method is called, allowing Dask to optimize the execution plan and run the calculation
 in parallel.
```

```
"""
```

```
The mean of the array is: 0.49996479465525767
```

Scaling computations across multiple cores or machines

Scaling computations across multiple cores or machines is essential for maximizing the performance of computationally intensive tasks, especially in fields like data analysis, machine learning, and scientific computing. Here's an overview of how to achieve this scaling using various techniques and frameworks.

```
"""

1. Parallel Computing

Parallel computing involves breaking a task into smaller subtasks that can be executed simultaneously
on multiple processors or cores. This can be done either on a single machine with multiple cores or
across multiple machines in a cluster.

a. Multiprocessing in Python

The multiprocessing module in Python allows you to create processes that can run concurrently,
 enabling parallel execution of tasks. Here's a basic example:
"""

import multiprocessing

def square(n):
    return n * n

if __name__ == "__main__":
    # Define a list of numbers
    numbers = [1, 2, 3, 4, 5]

    # Create a pool of worker processes
    with multiprocessing.Pool(processes=4) as pool:
        # Map the function to the list of numbers
        results = pool.map(square, numbers)

    print(results)

"""
Process Pool: A pool of worker processes is created, allowing multiple tasks to run concurrently.

Mapping: The map function applies the square function to each element in the numbers list,
distributing the tasks across the worker processes.
"""
```

```
[1, 4, 9, 16, 25]
```

```
"""
2. Distributed Computing Frameworks

For larger-scale computations, especially when working with big data,
distributed computing frameworks like Apache Spark, Dask, and Ray can be utilized.

a. Dask
Dask is a flexible parallel computing library for analytics in Python that integrates
well with NumPy and Pandas.

"""

import dask.array as da

# Create a large Dask array
x = da.random.random((10000, 10000), chunks=(1000, 1000))

# Perform a computation (e.g., mean)
mean_result = x.mean()

# Trigger the computation
result = mean_result.compute()

print(f"The mean of the array is: {result}")
```

```
The mean of the array is: 0.4999540542630303
```

```
"""
b. Apache Spark

Apache Spark is a distributed computing framework designed for fast computation,
ideal for big data processing.
"""

!pip install pyspark

from pyspark.sql import SparkSession

# Initialize Spark session
spark = SparkSession.builder.appName("example").getOrCreate()

# Create a DataFrame
data = [("Alice", 1), ("Bob", 2), ("Cathy", 3)]
df = spark.createDataFrame(data, ["Name", "Value"])

# Perform operations
result = df.groupBy("Name").sum("Value").show()
```

```
Collecting pyspark
  Downloading pyspark-3.5.3.tar.gz (317.3 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 317.3/317.3 MB 1.9 MB/s eta 0:00:00
  Preparing metadata (setup.py) ... done
Requirement already satisfied: py4j==0.10.9.7 in /usr/local/lib/python3.10/dist-packages (from pyspa
Building wheels for collected packages: pyspark
  Building wheel for pyspark (setup.py) ... done
  Created wheel for pyspark: filename=pyspark-3.5.3-py2.py3-none-any.whl size=317840625 sha256=4a565
  Stored in directory: /root/.cache/pip/wheels/1b/3a/92/28b93e2fbfdbb07509ca4d6f50c5e407f48dce4ddbda
Successfully built pyspark
Installing collected packages: pyspark
Successfully installed pyspark-3.5.3
+-----+----------+
     | Name|sum(Value)|
     +-----+----------+
     |Alice|         1|
     |  Bob|         2|
     |Cathy|         3|
     +-----+----------+




"""
3. Using GPU Acceleration

For tasks that can benefit from parallel computation on the GPU,
libraries like CuPy and TensorFlow can be used to offload computations to GPUs.

a. CuPy

CuPy is a GPU-accelerated library that is similar to NumP
"""

!pip install cupy

import cupy as cp

# Create a large CuPy array
x = cp.random.random((10000, 10000))

# Perform a computation (e.g., mean)
mean_result = cp.mean(x)

print(f"The mean of the array is: {mean_result}")
```

```
Collecting cupy
  Using cached cupy-13.3.0.tar.gz (3.4 MB)
  Preparing metadata (setup.py) ... done
Requirement already satisfied: numpy<2.3,>=1.22 in /usr/local/lib/python3.10/dist-packages (from
Requirement already satisfied: fastrlock>=0.5 in /usr/local/lib/python3.10/dist-packages (from cu
Building wheels for collected packages: cupy
  error: subprocess-exited-with-error

  × python setup.py bdist_wheel did not run successfully.
  │ exit code: 1
  ╰─> See above for output.

  note: This error originates from a subprocess, and is likely not a problem with pip.
  Building wheel for cupy (setup.py) ... error
  ERROR: Failed building wheel for cupy
  Running setup.py clean for cupy
Failed to build cupy
ERROR: ERROR: Failed to build installable wheels for some pyproject.toml based projects (cupy)
---------------------------------------------------------------------------
ImportError                               Traceback (most recent call last)
/usr/local/lib/python3.10/dist-packages/cupy/__init__.py in <module>
     16 try:
---> 17     from cupy import _core  # NOQA
     18 except ImportError as exc:

                              2 frames
ImportError: libcuda.so.1: cannot open shared object file: No such file or directory

The above exception was the direct cause of the following exception:

ImportError                               Traceback (most recent call last)
/usr/local/lib/python3.10/dist-packages/cupy/__init__.py in <module>
     17     from cupy import _core  # NOQA
     18 except ImportError as exc:
---> 19     raise ImportError(f'''
     20 ================================================================
     21 {_environment._diagnose_import_error()}

ImportError: 
================================================================
Failed to import CuPy.

If you installed CuPy via wheels (cupy-cudaXXX or cupy-rocm-X-X), make sure that the package
matches with the version of CUDA or ROCm installed.

On Linux, you may need to set LD_LIBRARY_PATH environment variable depending on how you
installed CUDA/ROCm.
On Windows, try setting CUDA_PATH environment variable.

Check the Installation Guide for details:
  https://docs.cupy.dev/en/latest/install.html

Original error:
  ImportError: libcuda.so.1: cannot open shared object file: No such file or directory
================================================================


---------------------------------------------------------------------------
NOTE: If your import is failing due to a missing package, you can
manually install dependencies using either !pip or !apt.

To view examples of installing some common dependencies, click the
"Open Examples" button below.
```

OPEN EXAMPLES

Next steps: Explain error

5. Optimization Strategies for Scaling

When scaling computations, consider the following strategies:

Chunking: Divide large datasets into smaller chunks that can be processed in parallel.

Load Balancing: Ensure that the workload is evenly distributed among available cores or machines.

Data Locality: Minimize data transfer times by processing data where it resides.

Optimized Algorithms: Use algorithms with lower time complexity and memory usage

```
import dask.array as da
import dask

# Step 1: Create a large Dask array with chunking
# Create an array of random numbers (e.g., 10 million elements) and chunk it into smaller arrays
large_array = da.random.random(size=(10000000,), chunks=(1000000,))

# Step 2: Define an optimized algorithm function
def optimized_computation(x):
    """A simple optimized computation: Calculate the square and then the mean."""
    return da.mean(x ** 2)

# Step 3: Perform the computation
# Use Dask to apply the optimized function to the large array
mean_result = optimized_computation(large_array)

# Step 4: Trigger computation (this uses load balancing internally)
result = mean_result.compute()  # This will execute the task in parallel, leveraging multiple cores

# Step 5: Print the result
print(f"The mean of the squares of the array is: {result}")
```

```
The mean of the squares of the array is: 0.33338164035171375
```

Start coding or generate with AI.

Start coding or generate with AI.

Start coding or generate with AI.

Start coding or generate with AI.

Start coding or generate with AI.

Start coding or generate with AI.

Start coding or generate with AI.

Start coding or generate with AI.

Start coding or generate with AI.

Start coding or generate with AI.

Start coding or generate with AI.

Start coding or generate with AI.

Start coding or generate with AI.

Efficient Data Visualization

Fast plotting with Matplotlib and Seaborn

Reduce Data Size: When plotting large datasets, consider sampling or aggregating your data to reduce the number of points being plotted.

Use Faster Backends: Matplotlib supports various backends for rendering. You can switch to a faster backend for interactive plotting, like Agg or Qt5Agg

import matplotlib

matplotlib.use('Agg') # Use a non-interactive backend for speed

Batch Plotting: If you need to create multiple plots, consider creating them in a loop and showing them at once to reduce rendering time.

Avoid Redundant Settings: Set properties like labels and titles only once for multiple plots if they share common attributes.

Caching: If generating plots takes time, consider saving them to disk and loading them when needed instead of regenerating them every time.

```
!pip install matplotlib seaborn pandas pillow
```

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib
import seaborn as sns

# Step 1: Use a faster backend for rendering
matplotlib.use('Agg')  # Use a non-interactive backend for speed

# Step 2: Generate sample stock market data
dates = pd.date_range(start='2023-01-01', periods=15)
data = {
    'Date': dates,
    'Open': np.random.uniform(100, 200, size=15),
    'Close': np.random.uniform(100, 200, size=15),
    'Volume': np.random.randint(1000, 5000, size=15)
}

# Create a DataFrame
stock_data = pd.DataFrame(data)

# Step 3: Sample or aggregate data if necessary (here we take the first 15 rows)
# Since we already have only 15 rows, we'll plot them directly.

# Step 4: Batch plotting
# Create multiple plots in a loop
fig, axes = plt.subplots(nrows=2, ncols=1, figsize=(10, 10))
fig.subplots_adjust(hspace=0.4)  # Adjust space between plots

# Plot Open vs Date
axes[0].plot(stock_data['Date'], stock_data['Open'], marker='o', label='Open Price', color='blue', linew
axes[0].set_title('Stock Market Open Prices')
axes[0].set_xlabel('Date')
axes[0].set_ylabel('Open Price')
axes[0].legend()
axes[0].grid(True)

# Plot Close vs Date
axes[1].plot(stock_data['Date'], stock_data['Close'], marker='o', label='Close Price', color='green', li
axes[1].set_title('Stock Market Close Prices')
axes[1].set_xlabel('Date')
axes[1].set_ylabel('Close Price')
axes[1].legend()
axes[1].grid(True)

# Step 5: Save the plots to disk instead of displaying them
plt.savefig('stock_market_plots.png')
plt.close()  # Close the figure to free up memory

# Step 6: Loading saved plots (as an example of caching)
from PIL import Image
saved_plot = Image.open('stock_market_plots.png')
saved_plot.show()  # This will display the saved plot in an image viewer
```

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

# Step 1: Set up inline plotting for Google Colab
```

```
%matplotlib inline

# Step 2: Generate sample stock market data
dates = pd.date_range(start='2023-01-01', periods=15)
data = {
    'Date': dates,
    'Open': np.random.uniform(100, 200, size=15),
    'Close': np.random.uniform(100, 200, size=15),
    'Volume': np.random.randint(1000, 5000, size=15)
}

# Create a DataFrame
stock_data = pd.DataFrame(data)

# Step 3: Sample or aggregate data if necessary (here we take the first 15 rows)
# Since we already have only 15 rows, we'll plot them directly.

# Step 4: Batch plotting
# Create multiple plots in a loop
fig, axes = plt.subplots(nrows=2, ncols=1, figsize=(10, 10))
fig.subplots_adjust(hspace=0.4)  # Adjust space between plots

# Plot Open vs Date
axes[0].plot(stock_data['Date'], stock_data['Open'], marker='o', label='Open Price', color='blue', linewid
axes[0].set_title('Stock Market Open Prices')
axes[0].set_xlabel('Date')
axes[0].set_ylabel('Open Price')
axes[0].legend()
axes[0].grid(True)
```