Part 1 Answers

1.Optimize Python Script for Multicore Execution

When a Python script is running in a single thread, I can leverage multiple cores by using libraries such as `multiprocessing` or frameworks like `concurrent.futures`. Here's a solution using `multiprocessing.Pool` to distribute work across multiple cores:

```python
import multiprocessing as mp
import time

# Function to check if a number is prime
def is_prime(n):
    if n <= 1:
        return False
    for i in range(2, int(n**0.5) + 1):
        if n % i == 0:
            return False
    return True

# Function to process a chunk of data (find prime numbers)
def process_chunk(data_chunk):
    return [n for n in data_chunk if is_prime(n)]

# Multi-threaded approach using multiprocessing
def parallel_find_primes(start, end, num_cores):
    data = list(range(start, end))
    chunk_size = len(data) // num_cores
    chunks = [data[i : i + chunk_size] for i in range(0, len(data), chunk_size)]

    with mp.Pool(num_cores) as pool:
        results = pool.map(process_chunk, chunks)

    primes = [item for sublist in results for item in sublist]
    return primes

# Single-threaded approach to find prime numbers
def single_thread_find_primes(start, end):
    primes = [n for n in range(start, end) if is_prime(n)]
    return primes

# Main program to test the performance of both approaches
if __name__ == "__main__":
    start_num = 2
    end_num = 1000000  # 1 million

    # Measure time for single-threaded approach
    start_time = time.time()
    single_thread_primes = single_thread_find_primes(start_num, end_num)
    single_thread_duration = time.time() - start_time
    # Measure time for multi-threaded approach
    num_cores = mp.cpu_count()
    start_time = time.time()
    multi_thread_primes = parallel_find_primes(start_num, end_num, num_cores)
    multi_thread_duration = time.time() - start_time

    assert set(single_thread_primes) == set(
        multi_thread_primes
    ), "Single-threaded and multi-threaded results do not match"

    print(f"Single-threaded duration: {single_thread_duration:.2f} seconds")
    print(f"Multi-threaded duration: {multi_thread_duration:.2f} seconds")
```

This program shows how to use single thread and multithread to calculate prime numbers from 2 to 1000000, and save the results in a list. Multithreading is more advantageous than single thread when a single task is more complex, otherwise a lot of time will be spent on dividing work and data transfer. Therefore, the `is_prime` function here uses a less efficient version to show the performance advantage of multithreading.

On my computer, the execution results of the above program are as follows:

```
Single-threaded duration: 3.54 seconds
Multi-threaded duration: 1.32 seconds
```

---

2. Debugging and Preventing Invalid Data Format Issues in Azure Data Factory

During a pipeline run in Azure Data Factory, a step failed due to an invalid data format. Here's how I would debug this issue and prevent it from happening in the future:

Debugging the Issue

1. **Check the Pipeline Run Logs** : I would start by navigating to the Monitor tab in Azure Data Factory. In the **Monitor** section, I can locate the failed pipeline run and inspect the specific activity that failed. The error message usually provides details about the failure, such as:

    - **Schema mismatch**: Columns in the data don't match the expected schema.
    - **Data type mismatch**: The data types do not match (e.g., trying to load a string into a numeric field).
    - **Invalid file format**: The file format doesn't align with the expected structure (e.g., an incorrectly formatted CSV or JSON file).

    By carefully reviewing the logs, I can determine exactly where the data format issue occurred.

2. **Examine the Data Format** : Next, I would examine the **source data** to ensure it matches the expected format. This could involve downloading the source file (e.g., a CSV, JSON, or Parquet file) and manually inspecting it to look for issues such as:

- Missing columns
- Incorrect data types
- Formatting errors (e.g., incorrect delimiters in CSV files)

I would also compare the source data format with the format expected by Azure Data Factory. This includes reviewing the dataset configuration to ensure that the schema, data types, and file format match what's being provided.

3. **Use Data Preview in ADF** : If I were working with a **Mapping Data Flow** or transformation activity, I would use the **Data Preview** feature in ADF. This allows me to preview the data as it moves through the pipeline and helps identify any formatting issues before they cause a failure. This visual inspection can be useful in spotting errors such as missing or misaligned columns, unexpected values, or incorrect data types.

4. **Review the Linked Services and Datasets** : I would then review the **Linked Services** and **Datasets** to ensure that the pipeline is correctly connected to the data source and destination. For example, I would check that the dataset schema aligns with the source data and that the Linked Services credentials and connections are correctly configured.

5. **Test with Sample Data** : To isolate the issue, I would run the pipeline in **debug mode** with sample data. This helps me trace the steps where the pipeline breaks and allows me to inspect the data as it flows between different pipeline activities. Debug mode also provides immediate feedback, so I can verify that the issue has been resolved before running the full pipeline again.

Preventing Future Invalid Data Format Issues

1. **Use Data Validation** : To prevent such issues in the future, I would implement **data validation checks** within the pipeline. This could involve adding **validation activities** such as:

   - **Filter Activity**: To filter out invalid rows based on data types or schema.
   - **Lookup Activity**: To check data integrity before processing.

2. **Use Schema Mapping** : I would configure **Schema Mapping** in ADF to ensure that the source columns map correctly to the destination schema. This way, even if the column order changes in the source data, the pipeline will still work correctly. I can set explicit mappings between columns during activities like the **Copy Activity** or **Mapping Data Flow** to enforce the expected data format.

3. **Add Error Handling** : I would also add error-handling mechanisms to the pipeline. This could include:

   - **Retry Policies**: To automatically retry transient errors.
   - **Error Pathways**: If an error occurs, the pipeline could log the error and continue processing other data, or send an alert for manual intervention using tools like **Web Activity** or **Logic Apps**.

   This ensures that small errors don’t stop the entire pipeline, and I can respond to issues proactively.

4. **Monitor Data Quality with Alerts** : To stay on top of potential format issues, I would set up **alerts** in the ADF **Monitor** tab. These alerts would notify me if a pipeline step fails, allowing me to address the issue immediately. This kind of proactive monitoring ensures that I’m aware of issues as they happen, rather than discovering them after the fact.

Example: Handling an Invalid CSV Format

If, for example, the issue is with a CSV file, I would:

1. Download the CSV file and manually inspect it for issues like missing columns or incorrect delimiters.
2. Adjust the **dataset schema** in ADF to match the actual structure of the file.
3. Add a **validation activity** to check the column count or data types before processing the file.
4. Use **schema mapping** in the **Copy Activity** to map the source columns correctly to the destination columns.

By following these steps, I can identify, debug, and prevent invalid data format issues in Azure Data Factory pipelines, ensuring smoother data integration and processing.

---

3. Upload File to Azure Blob Storage using Python

Using the Azure SDK (`azure-storage-blob`), I can write a Python script to upload a file to Azure Blob Storage. The script will check if the container exists and create it if it doesn't.

To ensure security, I firstly put some key variables to the `.env` files. The values would be replaced by real ones when actually running.

```
AZURE_CONNECTION_STRING=my_connection_string
AZURE_CONTAINER_NAME=my_container_name
AZURE_FILE_PATH=path_to_my_file
```

The `connection_string` can be found in **Storage accounts** - **Security + networking** - **Access keys**.

Before running Python script, I would install the Azure SDK by running:

```
pip install azure-storage-blob
```

Then run this script to upload my own file to the Azure Blob Storage container.

```python
from azure.storage.blob import BlobServiceClient
import os
from dotenv import load_dotenv

load_dotenv()

# Retrieve values from environment variables
connection_string = os.getenv("AZURE_CONNECTION_STRING")
container_name = os.getenv("AZURE_CONTAINER_NAME")
file_path = os.getenv("AZURE_FILE_PATH")
blob_name = os.path.basename(file_path)

# Initialize the BlobServiceClient
blob_service_client = BlobServiceClient.from_connection_string(connection_string)
```

```
# Check if the container exists, create if it doesn't
container_client = blob_service_client.get_container_client(container_name)
if not container_client.exists():
    container_client.create_container()
    print(f'New container "{container_name}" created.')

# Upload the file
blob_client = blob_service_client.get_blob_client(
    container=container_name, blob=blob_name
)

with open(file_path, "rb") as data:
    blob_client.upload_blob(data, overwrite=True)

print(f"File {file_path} uploaded to container {container_name} as blob
{blob_name}.")
```

---

4. Download Logs from Azure

To download logs from Azure, I can use the **Azure Monitor** SDK or query logs via **Azure Activity log** . Below is a Python script to retrieve Azure activity logs.

To ensure security, I firstly put some key variables to the `.env` files. The values would be replaced by real ones when actually running.

```
SUBSCRIPTION_ID=my_subscription_id
RESOURCE_ID=my_resource_id
```

Before running Python script, I would install the Azure SDK by running:

```
pip install azure-identity azure-mgmt-monitor
```

Then run this script to download activity logs from the specific resource.

```
from azure.identity import DefaultAzureCredential
from azure.mgmt.monitor import MonitorManagementClient
from datetime import datetime, timedelta
import csv
from dotenv import load_dotenv
import os

load_dotenv()

# Initialize credentials and MonitorManagementClient
credential = DefaultAzureCredential()
subscription_id = os.getenv(
    "SUBSCRIPTION_ID"
)  # Replace with your Azure subscription ID
monitor_client = MonitorManagementClient(credential, subscription_id)

# Define the time range for the Activity Logs (e.g., last 1 day)
start_time = datetime.now() - timedelta(days=1)
end_time = datetime.now()

# Define the resource you want to filter (e.g., specific resource name or resource
ID)
resource_id = os.getenv("RESOURCE_ID")

# Query Activity Logs for the specific resource
activity_logs = monitor_client.activity_logs.list(
    filter=f"eventTimestamp ge '{start_time}' and eventTimestamp le '{end_time}'
and resourceId eq '{resource_id}')"
)

# Define a CSV file to store the logs
csv_file = "activity_logs.csv"

# Write the logs to a CSV file
with open(csv_file, mode="w", newline="") as file:
    writer = csv.writer(file)
    # Write header row to match the format you showed
    writer.writerow(
        [
            "Correlation id",
            "Operation name",
            "Status",
            "Event category",
            "Level",
            "Time",
            "Subscription",
            "Event initiated by",
            "Resource type",
            "Resource group",
            "Resource",
        ]
    )

    # Write log data
    for log in activity_logs:
        writer.writerow(
            [
                log.correlation_id,
                log.operation_name.localized_value if log.operation_name else
None,
                log.status.localized_value if log.status else None,
                log.category.value if log.category else None,
                log.level if log.level else None,
                log.event_timestamp.isoformat() if log.event_timestamp else None,
                log.subscription_id,
                log.caller,
                log.resource_type.value if log.resource_type else None,
                log.resource_group_name,
                log.resource_id,
            ]
        )

print(f"Activity logs saved to {csv_file}")
```