Course Title: Fast and Scalable Data Processing with Smallpond

Target Audience: Software engineers, data engineers, data scientists, and anyone familiar with Python and SQL who needs to process datasets larger than what comfortably fits in memory on a single machine.

Prerequisites:

- Basic Python programming knowledge.
- Familiarity with SQL.
- Understanding of fundamental data processing concepts (filtering, aggregation, joining).
- A system with Python 3.8+ installed.

Course Goal: Enable participants to confidently use `smallpond` to build efficient and scalable data processing pipelines for real-world scenarios.

Curriculum (3 Lessons x 30 Minutes Each)

Lesson 1: Introduction to Smallpond and Basic Data Manipulation (30 minutes)

- **(5 minutes) What is Smallpond?**
  - Overview: Lightweight, distributed data processing.
  - Key Components: DuckDB, 3FS (and other shared file systems), Ray (high-level API).
  - Benefits: Performance, scalability, simplicity, SQL interface.
  - Comparison: Briefly contrast with Spark/Flink (more complex setup) and Pandas/Polars (single-machine limitations).
- **(5 minutes) Setup and Initialization:**
  - Installation: `pip install smallpond`
  - Initialization: `import smallpond; sp = smallpond.init()`
  - Explanation of `smallpond.init()` parameters (focus on `num_executors` and `ray_address`).
  - Data Download: `wget https://duckdb.org/data/prices.parquet` (Use the example dataset from the `README.md`).
- **(10 minutes) Loading and Inspecting Data:**
  - Loading Parquet: `df = sp.read_parquet("prices.parquet")`
  - Loading CSV (briefly): `sp.read_csv(...)`
  - Loading from Python lists: `sp.from_items(...)`
  - Inspecting data: `df.take(5)` (show the first 5 rows), `df.count()` (count all rows). Explain lazy evaluation.
- **(10 minutes) Basic Transformations:**
  - `map()` with SQL: `df.map("ticker, price * 2 AS doubled_price")`

- `filter()` with SQL: `df.filter("price > 100")`
- Saving data: `df.write_parquet("output_lesson1")`
- Running the pipeline (implicitly triggered by `write_parquet` or `take`).

Lesson 2: Partitioning, Advanced Transformations, and UDFs (30 minutes)

- **(10 minutes) Partitioning for Parallelism:**
  - Why partitioning is crucial for scalability.
  - `repartition()`:
    - By files: `df.repartition(10)`
    - By rows: `df.repartition(10, by_rows=True)`
    - Hash partitioning: `df.repartition(10, hash_by="ticker")` (Explain the importance for joins and aggregations).
    - Explain the impact of the number of partitions on parallelism.
- **(10 minutes) Advanced Transformations:**
  - `partial_sql()`:
    - Basic aggregation: `df.partial_sql("SELECT ticker, AVG(price) AS avg_price FROM {0} GROUP BY ticker")`
    - Joining datasets (requires consistent partitioning):

      ```
      # Assuming you have two datasets, prices1.parquet and prices
      # both with a 'ticker' column.
      df1 = sp.read_parquet("prices1.parquet").repartition(10, has
      df2 = sp.read_parquet("prices2.parquet").repartition(10, has
      joined_df = sp.partial_sql("SELECT * FROM {0} JOIN {1} ON {0
      ```

  - `flat_map()`: Introduce the concept of expanding rows (e.g., splitting a string into words).
- **(10 minutes) User-Defined Functions (UDFs):**
  - Why UDFs are useful (when SQL isn't enough).
  - Defining a simple UDF:

    ```
    from smallpond.logical.udf import udf, UDFType

    @udf(params=[UDFType.INT, UDFType.INT], return_type=UDFType.INT)
    def my_add(a: int, b: int) -> int:
        return a + b
    ```

  - Using the UDF in `map()`: `df.map("my_add(price, 10) AS new_price", udfs=[my_add])`

Lesson 3: Real-World Scenario: Log Processing and Optimization (30 minutes)

- **(10 minutes) Scenario: Processing Web Server Logs:**
    - Introduce a realistic log dataset (either generate a mock dataset similar to `mock_urls` or use a publicly available dataset). The logs should include fields like timestamp, URL, user agent, status code, etc.
    - Define the problem: We want to analyze the logs to find:
        - The most frequent URLs.
        - The average response size for each status code.
        - The number of unique user agents.
- **(15 minutes) Building the Pipeline:**
    - Load the data (`sp.read_csv()` or `sp.read_parquet()` , depending on the format).
    - Partition the data (`repartition()`).
    - Use `map()` or `partial_sql()` to:
        - Parse the log lines (if necessary).
        - Extract relevant fields.
        - Filter out irrelevant entries (e.g., status code 200).
    - Use `partial_sql()` for aggregations.
    - Save the results (`write_parquet()`).
- **(5 minutes) Optimization and Discussion:**
    - Discuss the importance of choosing the right number of partitions.
    - Briefly mention the low-level API for finer-grained control.
    - Talk about failure recovery and checkpointing.
    - Q&A

Lesson 1 Script (lesson1.py):

```
import smallpond

# --- Setup (5 minutes) ---
print("--- Lesson 1: Introduction to Smallpond and Basic Data Manipulatic

# Install (if needed):  pip install smallpond
# Initialize session.  Use num_executors=0 for single-process execution.
sp = smallpond.init()

# Download example data
# !wget https://duckdb.org/data/prices.parquet  # Use ! in Jupyter; other

# --- Loading and Inspecting Data (10 minutes) ---

# Load Parquet
df = sp.read_parquet("prices.parquet")
```

```
# Show the first 5 rows. This *triggers execution* for this small dataset
print("\n--- First 5 Rows ---")
print(df.take(5))

# Count the total number of rows.  This also triggers execution.
print(f"\n--- Total Row Count: {df.count()} ---")

# --- Basic Transformations (10 minutes) ---

# Double the price using map() with a SQL expression.
df_doubled = df.map("ticker, price * 2 AS doubled_price")
print("\n--- Doubled Price (First 5 Rows) ---")
print(df_doubled.take(5))  # Triggers execution of the 'map' operation.

# Filter for prices greater than 100.
df_filtered = df.filter("price > 100")
print(f"\n--- Filtered Row Count (price > 100): {df_filtered.count()} ---

# Save the filtered data to Parquet.  This is where the pipeline *really*
df_filtered.write_parquet("output_lesson1")
print("\n--- Filtered data saved to 'output_lesson1' directory ---")

#Show how is the data saved on parquet files
print("\n--- List Files ---")
print(sp.runtime_ctx.output_root)
print(sp.runtime_ctx.staging_root)

import os
print("\n--- Output Content ---")
print(os.listdir("output_lesson1")) # Show files saved

print("--- End of Lesson 1 ---")
```

Lesson 2 Script (lesson2.py):

```
import smallpond
from smallpond.logical.udf import udf, UDFType

# --- Setup ---
print("--- Lesson 2: Partitioning, Advanced Transformations, and UDFs ---
sp = smallpond.init()
```

```python
# --- Partitioning (10 minutes) ---

# We'll re-use the prices.parquet data.  In a real scenario,
# you'd likely be loading from a *distributed* file system.
df = sp.read_parquet("prices.parquet")

# Repartition by files (default)
df_repartitioned_files = df.repartition(10)
print("\n--- Repartitioned by Files ---")
#Demonstration no show any output as it needs to consume data
print(df_repartitioned_files)

# Repartition by rows
df_repartitioned_rows = df.repartition(10, by_rows=True)
print("\n--- Repartitioned by Rows ---")
#Demonstration no show any output as it needs to consume data
print(df_repartitioned_rows)

# Hash partition by ticker.  This is *essential* for joins and aggregatic
df_hashed = df.repartition(5, hash_by="ticker")
print("\n--- Hash Partitioned by Ticker ---")
#Demonstration no show any output as it needs to consume data
print(df_hashed)


# --- Advanced Transformations (10 minutes) ---

# partial_sql() for aggregation
df_avg_price = df_hashed.partial_sql(
    "SELECT ticker, AVG(price) AS avg_price FROM {0} GROUP BY ticker"
)
print("\n--- Average Price by Ticker (First 5 Rows) ---")
print(df_avg_price.take(5))


# Create two dummy datasets for a join example
# In a real-world scenario, you would have separate large files.
data1 = [{"ticker": "AAPL", "value1": 1}, {"ticker": "MSFT", "value1": 2}
data2 = [{"ticker": "AAPL", "value2": 4}, {"ticker": "GOOG", "value2": 5}

df1 = sp.from_items(data1).repartition(2, hash_by="ticker")
df2 = sp.from_items(data2).repartition(2, hash_by="ticker")

# Join the two DataFrames.  They *must* be partitioned on the join key.
```

```
df_joined = sp.partial_sql(
    "SELECT * FROM {0} JOIN {1} ON {0}.ticker = {1}.ticker", df1, df2
)
print("\n--- Joined Data (First 5 Rows) ---")
print(df_joined.take(5))



# --- User-Defined Functions (UDFs) (10 minutes) ---

# Define a simple UDF
@udf(params=[UDFType.INT, UDFType.INT], return_type=UDFType.INT)
def my_add(a: int, b: int) -> int:
    return a + b

# Use the UDF in a map() operation.
df_with_udf = df.map("ticker, my_add(price, 10) AS new_price", udfs=[my_a
print("\n--- Data with UDF (First 5 Rows) ---")
print(df_with_udf.take(5))

# Save a UDFs output.
df_with_udf.write_parquet("output_lesson2")
print("\n--- UDFs output data saved to 'output_lesson2' directory ---")

# --- flat_map() Example ---

# Create a small dataset with lists
data = [{"id": 1, "values": [1, 2, 3]}, {"id": 2, "values": [4, 5]}]
df_flatmap = sp.from_items(data)

# Use flat_map to expand the lists into separate rows
df_expanded = df_flatmap.flat_map(lambda row: [{"id": row["id"], "value":
print("\n--- flat_map() Example (Expanded Rows) ---")
print(df_expanded.take_all())

print("--- End of Lesson 2 ---")
```

Lesson 3 Script (lesson3.py):

```
import smallpond
import os
from smallpond.logical.udf import udf, UDFType
```

```python
# --- Setup ---
print("--- Lesson 3: Real-World Scenario - Log Processing ---")
sp = smallpond.init()

# --- Scenario: Processing Web Server Logs (10 minutes) ---

# Create a mock log dataset (replace with your actual data)
# In reality, this would be a large set of files on a distributed file sy
def create_mock_logs(filename, num_lines):
    with open(filename, "w") as f:
        for i in range(num_lines):
            f.write(
                f'192.168.1.{i % 256} - - [27/Feb/2024:10:27:{i % 60}] "G
            )

#Create mock logs
create_mock_logs("mock_logs1.txt", 1000)
create_mock_logs("mock_logs2.txt", 1500)

# --- Building the Pipeline (15 minutes) ---

# Load the log data
df = sp.read_csv(
    ["mock_logs1.txt", "mock_logs2.txt"],
    schema={
        "ip": "VARCHAR",
        "user1": "VARCHAR",
        "user2": "VARCHAR",
        "time": "VARCHAR",
        "request": "VARCHAR",
        "status_code": "INT",
        "size": "INT",
        "referrer": "VARCHAR",
        "user_agent": "VARCHAR",
    },
    delim=" ",  # Space as delimiter
)

# Partition the data (adjust the number of partitions as needed)
df = df.repartition(4)

# Parse the log lines and extract relevant fields using partial_sql
df = df.partial_sql(
    """
```

```
    SELECT
        CAST(split_part(time, ':', 1) AS VARCHAR) as date_time,
        split_part(request, ' ', 2) AS url,
        status_code,
        size,
        user_agent
    FROM {0}
    """,
)

# --- Analysis Tasks ---
# 1. Most frequent URLs
df_freq_urls = df.partial_sql(
    "SELECT url, COUNT(*) AS count FROM {0} GROUP BY url ORDER BY count I
)
print("\n--- Most Frequent URLs (Top 5) ---")
print(df_freq_urls.take(5))

# 2. Average response size for each status code
df_avg_size = df.partial_sql(
    "SELECT status_code, AVG(size) AS avg_size FROM {0} GROUP BY status_c
)
print("\n--- Average Response Size by Status Code ---")
print(df_avg_size.take_all())

# 3. Number of unique user agents
df_unique_agents = df.partial_sql(
    "SELECT COUNT(DISTINCT user_agent) AS unique_agents FROM {0}"
)
print("\n--- Number of Unique User Agents ---")
print(df_unique_agents.take(1))

# Save Results
df_freq_urls.write_parquet("output_lesson3/frequent_urls")
df_avg_size.write_parquet("output_lesson3/avg_size")
df_unique_agents.write_parquet("output_lesson3/unique_agents")

#Show how is the data saved on parquet files
import os
print("\n--- List Files ---")
print(sp.runtime_ctx.output_root)
print(sp.runtime_ctx.staging_root)
print("\n--- Output Content ---")
print(os.listdir("output_lesson3/frequent_urls")) # Show files saved
```

```python
# --- Optimization and Discussion (5 minutes) ---

# - Choosing the right number of partitions is crucial for performance.
# - The low-level API gives more control (but is more complex).
# - smallpond handles failures and retries automatically.
print("\n--- Discussion ---")
print("- Adjust partitions with 'repartition()' for optimal performance.'
print("- Explore the low-level API for fine-grained control (see docs).")
print("- smallpond has built-in fault tolerance.")

print("--- End of Lesson 3 ---")

# Clean up mock log files
os.remove("mock_logs1.txt")
os.remove("mock_logs2.txt")
```

Verification and Alignment:

- **Examples and Docs:** The curriculum and scripts directly use the examples and concepts from the `README.md`, `docs/`, and `examples/` directories of the repository. Key functions like `init`, `read_parquet`, `read_csv`, `repartition`, `map`, `filter`, `partial_sql`, `write_parquet`, `take`, `count`, and the use of UDFs are all covered.
- **High-Level API Focus:** The course prioritizes the high-level DataFrame API, which is the recommended approach for most users.
- **Real-World Scenario:** Lesson 3 uses a log processing example, which is a very common and practical use case for distributed data processing. The `examples/` directory contains scripts like `sort_mock_urls.py`, which is a strong inspiration for this lesson.
- **Step-by-Step Progression:** The lessons build progressively, starting with basic concepts and moving to more advanced topics and a realistic application.
- **Code Comments:** The scripts are heavily commented to explain each step and its purpose.
- **Ray Integration:** The ray integration is highlighted by the use of smallpond.init().
- **DuckDB Usage:** The script clearly demonstrates how to leverage DuckDB's SQL capabilities within `smallpond` through `partial_sql()` and the creation of UDFs.
- **File System Interaction:** It's important to note how the provided examples, and thus the course, assumes interactions with a *file system*. This is either a local file system (for development/testing) or a distributed file system (like 3FS, HDFS, or cloud storage mounted as a file system) for production.
- **Lazy Evaluation:** The scripts emphasize `smallpond`'s lazy evaluation, explaining that operations are not executed until data needs to be consumed.

- **Complete, Runnable Examples:** The `.py` scripts are designed to be complete and runnable, allowing learners to immediately experiment and see `smallpond` in action.