W04 –
PYSPARK – CODING STANDARDS & BEST PRACTICES

2024

The Hoang

Coding Standards & Best Practices

Modularize Code

- Define reusable functions: Avoid duplicating code by abstracting commonly used logic into functions.
- Use classes and modules: If your project is large, break the code into classes or modules for better organization and reusability.

```
def clean_data(df):
    return df.filter(df["age"] > 18).dropna()

df_cleaned = clean_data(df)
```

Coding Standards & Best Practices

Avoid Hardcoding

Use configuration files or environment variables for paths, schema definitions, and other parameters.

```
import os
data_path = os.getenv("DATA_PATH", "/default/path")
df = spark.read.csv(data_path)
```

Coding Standards & Best Practices

Use Meaningful Variable Names

Make your variable names descriptive and adhere to ***PEP 8*** naming conventions.

```
# Bad
w = df.groupBy("age").count()

# Good
age_grouped_count = df.groupBy("age").count()
```

Coding Standards & Best Practices

Avoid collecting data to driver

Avoid excessive use of collect(): collect() brings all data to the driver, which can cause memory issues for large datasets. Instead, use take() or show() for debugging small samples

```
# Bad
data = df.collect()

# Good
data = df.take(10)  # For debugging
```

Coding Standards & Best Practices

Use schema definitions

Define schemas explicitly when reading data, especially from CSV or JSON, to avoid expensive inference.

```
from pyspark.sql.types import StructType, StructField, StringType, IntegerType

schema = StructType([
    StructField("name", StringType(), True),
    StructField("age", IntegerType(), True)
])

df = spark.read.schema(schema).csv("path/to/file")
```

Performance Optimization Best Practices

Partitioning and Parallelism

Tune parallelism: Set the number of partitions based on the size of your data and the cluster setup.

```
# Adjust parallelism
spark.conf.set("spark.sql.shuffle.partitions", "200")  # Default is often 200
```

Performance Optimization Best Practices

Repartition vs. Coalesce

Use repartition() to increase partitions (for parallelism) and coalesce() to reduce partitions (for actions like writing to disk).

```
large_df = df.repartition(100)  # Increase partitions
small_df = df.coalesce(5)       # Decrease partitions
```

Performance Optimization Best Practices

Avoid small file problems

When writing data to disk, avoid generating too many small partitions by using coalesce() before writing

```
df.coalesce(1).write.parquet("/path/to/output")
```

Performance Optimization Best Practices

Use Caching and Persistence

Cache reuseable DataFrames: If a DataFrame is used multiple times, cache or persist it in memory. Use unpersist() to free up the memory when not needed

```
df.cache()  # Cache in memory
df.unpersist()  # Remove from memory
```

Performance Optimization Best Practices

Choose appropriate persistence levels:

- MEMORY_ONLY
- MEMORY_AND_DISK
- MEMORY_ONLY_SER – “ser” means serialized => taking less memory
- MEMORY_AND_DISK_SER
- DISK_ONLY
- MEMORY_ONLY_2 => “_2” means replicate each partition to 2 cluster nodes
- MEMORY_AND_DISK_2

```
df.persist(StorageLevel.MEMORY_AND_DISK)  # Use disk if memory is insufficient
```

Performance Optimization Best Practices

Avoid Skew and Data Shuffling

Prevent data skew: If certain keys dominate a dataset, it can cause partition imbalance. Use techniques like salting to distribute the load

```
from pyspark.sql.functions import rand

df_salted = df.withColumn("salt", (rand() * 10).cast("int"))
df_salted_grouped = df_salted.groupBy("key", "salt").agg(...)
```

Performance Optimization Best Practices

- **Minimize shuffles**: Avoid wide transformations like groupBy, join, and distinct unless necessary. Use broadcast joins when one dataset is small

```
from pyspark.sql.functions import broadcast

df_joined = df1.join(broadcast(df2), "id")
```

Performance Optimization Best Practices

Filter Early and Reduce Data Size

Filter early: Apply filter() or select() as early as possible to reduce the size of the dataset.

```
# Bad: Large dataset processed before filtering
df_large = df.groupBy("age").count().filter("age > 30")

# Good: Filter first, then process smaller dataset
df_filtered = df.filter("age > 30").groupBy("age").count()
```

Performance Optimization Best Practices

Use mapPartitions() for Heavy Computations

Instead of applying a function to each element, it applies the function to an iterator over the elements in each partition. This gives more control over partition-level operations and can be more efficient for certain use cases.

It is useful when you want to apply transformations over a partition of elements, rather than element-wise processing

```
rdd = sc.parallelize([1, 2, 3, 4], 2)  # 2 partitions
def process_partition(iterator):
    yield sum(iterator)

partitioned_rdd = rdd.mapPartitions(process_partition)
# Result: [3, 7]  (since each partition's elements are summed)
```

Performance Optimization Best Practices

Use reduceByKey() Instead of groupByKey()

Avoid **groupByKey**(): It shuffles all data, potentially causing memory issues. Use **reduceByKey**() for combining values by key without shuffling all data.

reduceByKey() applies the reduce function (e.g., sum, max, min) locally before shuffling the data, reducing the amount of data transferred across the network.

groupByKey() does not perform any aggregation before the shuffle; it transfers all records associated with a key, which can lead to higher network overhead and memory usage.

```
# Bad
rdd.groupByKey().mapValues(sum)

# Good
rdd.reduceByKey(lambda x, y: x + y)
```

Debugging and Monitoring Best Practices

Use Logs for Debugging

Enable logging: Use Spark's built-in logging to track the progress and errors in your application.

```
sc.setLogLevel("ERROR")  # Set log level to minimize verbosity
```

Debugging and Monitoring Best Practices

Use explain()

Call df.explain() to see the logical and physical execution plan of your DataFrame transformations.

```
df.groupBy("age").count().explain()
```

Debugging and Monitoring Best Practices

Use take() Instead of collect()

When debugging, use take() to view a small sample of the dataset instead of collect() to avoid memory overload.

```
# Bad
df.collect()  # Might crash for large data

# Good
df.take(10)  # View only the first 10 rows
```

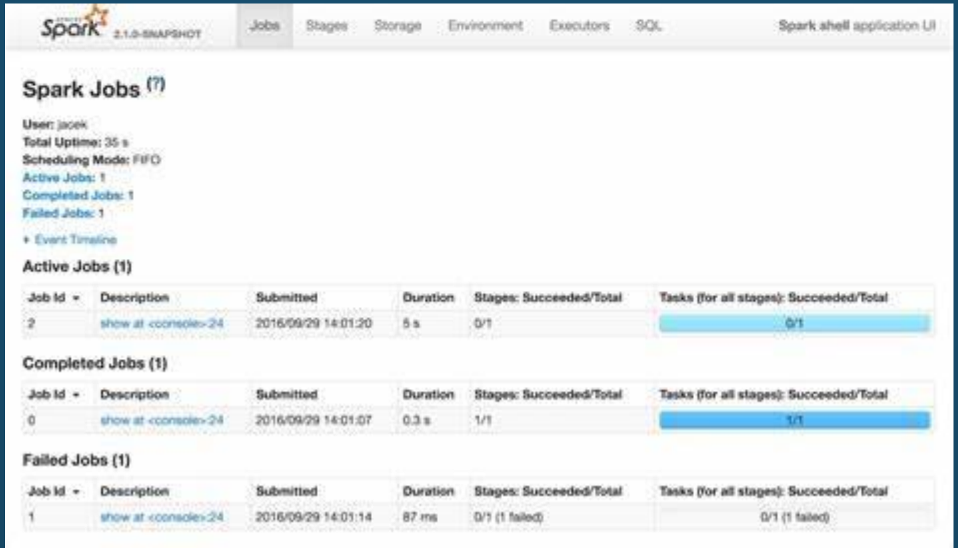
Debugging and Monitoring Best Practices

Monitor Spark UI

Use the Spark Web UI: This provides valuable insights into job execution, including task durations, shuffle behavior, and memory usage. The UI is available at http://<driver-node>:4040 when running locally or on a cluster.

Stages and tasks: Check if too many tasks are running out of memory or taking too long.

Storage tab: Monitor how much data is cached and persisted.



Debugging and Monitoring Best Practices

Use assert() for Data Validations

To ensure data integrity during development, use assertions to validate assumptions about your data.

```
# Assert that no null values exist in the 'age' column
assert df.filter(df.age.isNull()).count() == 0, "Null values found in 'age' column"
```

Debugging and Monitoring Best Practices

Handle Exceptions Properly

Use try-except blocks to gracefully handle errors, especially when dealing with external data sources.

```
try:
    df = spark.read.csv("/path/to/file")
except Exception as e:
    print(f"Error reading file: {e}")
```

Handling and Managing Data

Avoid count() for Large Datasets

Avoid using count() on large datasets as it triggers a full computation.

```
# Bad: Triggers full computation of the dataset
df.count()

# Good: Sample data for a quick check
df.sample(withReplacement=False, fraction=0.01).count()
```

Handling and Managing Data

Use broadcast() for Small Datasets

Use broadcasting for small lookup tables to avoid shuffling large datasets during joins

```
small_df_broadcast = broadcast(small_df)
df = large_df.join(small_df_broadcast, "id")
```

Handling and Managing Data

Use Columnar Formats

Prefer Parquet or ORC over CSV or JSON for storing structured data, as they are more efficient for both storage and query performance..

```
df.write.parquet("/path/to/output")
```

Miscellaneous Tips

Use the Latest PySpark Version

Keep your PySpark version up-to-date to take advantage of performance improvements and new features.

Miscellaneous Tips

Avoid Using UDFs Unless Necessary

Avoid using User Defined Functions (UDFs) unless absolutely required, as they can be slower than built-in Spark SQL functions. Instead, leverage PySpark SQL functions

```
from pyspark.sql.functions import col

# Prefer this:
df.select(col("name").alias("username"))

# Over this:
from pyspark.sql.functions import udf
my_udf = udf(lambda x: x.upper())
df.select(my_udf(col("name")))
```

Miscellaneous Tips

Use Vectorized UDFs (Pandas UDFs) Where Necessary

If UDFs are necessary, prefer Pandas UDFs (vectorized UDFs) which are much faster than standard UDFs.

```
from pyspark.sql.functions import pandas_udf

@pandas_udf("string")
def to_upper(s: pd.Series) -> pd.Series:
    return s.str.upper()

df.withColumn("name_upper", to_upper(df["name"])).show()
```

Miscellaneous Tips

Use Vectorized UDFs (Pandas UDFs) Where Necessary

If UDFs are necessary, prefer Pandas UDFs (vectorized UDFs) which are much faster than standard UDFs.

```
from pyspark.sql.functions import pandas_udf

@pandas_udf("string")
def to_upper(s: pd.Series) -> pd.Series:
    return s.str.upper()

df.withColumn("name_upper", to_upper(df["name"])).show()
```

Prefer PySpark Pandas API over Pandas

When working with big data in PySpark, it's often better to prefer PySpark's Pandas API (pyspark.pandas) over the original Pandas API for several reasons related to scalability, performance, and memory management.

Feature	**Pandas**	**PySpark Pandas API** (pyspark.pandas)
Memory Management	Entire data in memory	Distributed, lazy evaluation, spilling to disk if needed
Dataset Size Limitations	Limited by system memory	Can handle datasets larger than memory
Execution	Single-threaded	Distributed across a Spark cluster
Performance	Slower for large datasets	Faster for large datasets due to parallelization
Interoperability with PySpark	No direct interoperability	Seamless conversion to/from PySpark DataFrame
Cluster Utilization	Not cluster-aware	Leverages Spark cluster resources

Summary of Best Practices

Category	Best Practices
Coding Standards	Modularize code, use meaningful variable names, avoid hardcoding, avoid excessive collect(), define schemas explicitly.
Performance Optimization	Tune partitioning, use cache()/persist(), avoid data shuffling, filter early, use reduceByKey(), broadcast small datasets, use mapPartitions() properly.
Debugging and Monitoring	Use Spark UI, explain(), take() for debugging, handle exceptions, use assertions for validations, monitor logs.
Data Management	Avoid count(), use broadcast() for small datasets, prefer columnar formats (Parquet, ORC), partition data when writing.
Miscellaneous	Avoid UDFs unless necessary, use Pandas UDFs if required, keep PySpark up-to-date. Prefer PySpark Pandas APIs over the Original Pandas APIs