PySpark - Hướng Dẫn Nâng Cao

Mục Lục

1. Window Functions
2. User Defined Functions (UDF)
3. Broadcast Variables và Accumulators
4. Partitioning và Coalescing
5. Caching và Persistence
6. Join Strategies
7. Structured Streaming
8. Spark MLlib Cơ Bản
9. Performance Tuning
10. Advanced DataFrame Operations
11. Handling Large Datasets
12. Error Handling và Debugging

Window Functions

Window Functions cho phép bạn thực hiện các phép tính trên một tập hợp các dòng liên quan đến dòng hiện tại.

Khái Niệm Cơ Bản

```
from pyspark.sql import SparkSession
from pyspark.sql.window import Window
from pyspark.sql.functions import row_number, rank, dense_rank, lag, lead, sum,
avg, max

spark = SparkSession.builder.appName("Window Functions").getOrCreate()

# Dữ liệu mẫu
data = [
    ("Alice", "Sales", 5000, "2023-01"),
    ("Bob", "Sales", 6000, "2023-01"),
    ("Charlie", "Sales", 5500, "2023-02"),
    ("David", "IT", 7000, "2023-01"),
    ("Eve", "IT", 8000, "2023-02"),
    ("Frank", "IT", 7500, "2023-03")
]
df = spark.createDataFrame(data, ["name", "department", "salary", "month"])
```

Ranking Functions

ROW_NUMBER()

```
from pyspark.sql.functions import col

# Đánh số thứ tự
windowSpec = Window.partitionBy("department").orderBy("salary")
df.withColumn("row_number", row_number().over(windowSpec)).show()

# Tìm top N trong mỗi nhóm
df.withColumn("row_number", row_number().over(windowSpec)) \
  .filter(col("row_number") <= 2) \
  .show()
```

RANK() và DENSE_RANK()

```
# RANK: Bỏ qua thứ hạng khi có giá trị bằng nhau
df.withColumn("rank", rank().over(windowSpec)).show()

# DENSE_RANK: Không bỏ qua thứ hạng
df.withColumn("dense_rank", dense_rank().over(windowSpec)).show()

# PERCENT_RANK: Thứ hạng phần trăm
from pyspark.sql.functions import percent_rank
df.withColumn("percent_rank", percent_rank().over(windowSpec)).show()
```

Aggregation Functions với Window

Running Total

```
# Tổng tích lũy
windowSpec =
Window.partitionBy("department").orderBy("month").rowsBetween(Window.unboundedPrec
eding, Window.currentRow)
df.withColumn("running_total", sum("salary").over(windowSpec)).show()
```

Moving Average

```
# Trung bình 3 tháng gần nhất
windowSpec = Window.partitionBy("department").orderBy("month").rowsBetween(-2, 0)
df.withColumn("moving_avg", avg("salary").over(windowSpec)).show()
```

Max/Min trong Window

```
# Giá trị max trong department
windowSpec = Window.partitionBy("department")
df.withColumn("max_salary", max("salary").over(windowSpec)).show()
```

LAG và LEAD

```
# LAG: Lấy giá trị của dòng trước
windowSpec = Window.partitionBy("department").orderBy("month")
df.withColumn("prev_salary", lag("salary", 1).over(windowSpec)).show()

# LEAD: Lấy giá trị của dòng sau
df.withColumn("next_salary", lead("salary", 1).over(windowSpec)).show()

# Tính thay đổi so với tháng trước
df.withColumn("prev_salary", lag("salary", 1).over(windowSpec)) \
  .withColumn("change", col("salary") - col("prev_salary")) \
  .show()
```

Window Spec Ranges

```
# ROWS BETWEEN: Dựa trên số dòng
# RANGE BETWEEN: Dựa trên giá trị

# Tổng 3 dòng trước đến dòng hiện tại
rows_window = Window.partitionBy("department").orderBy("month").rowsBetween(-2, 0)

# Tổng từ đầu đến dòng hiện tại
unbounded_window = Window.partitionBy("department").orderBy("month") \
    .rowsBetween(Window.unboundedPreceding, Window.currentRow)

# Tổng từ dòng hiện tại đến cuối
following_window = Window.partitionBy("department").orderBy("month") \
    .rowsBetween(Window.currentRow, Window.unboundedFollowing)
```

User Defined Functions (UDF)

UDF cho phép bạn định nghĩa hàm Python tùy chỉnh và sử dụng trong Spark SQL.

UDF Cơ Bản

```
from pyspark.sql.functions import udf, col
from pyspark.sql.types import StringType, IntegerType

# Định nghĩa hàm Python
```

```
def capitalize_name(name):
    return name.upper() if name else None

# Đăng ký UDF
capitalize_udf = udf(capitalize_name, StringType())

# Sử dụng UDF
df.withColumn("upper_name", capitalize_udf(col("name"))).show()
```

UDF với Lambda Function

```
from pyspark.sql.functions import udf, col
from pyspark.sql.types import StringType

# UDF với lambda
add_prefix_udf = udf(lambda x: f"Mr. {x}" if x else None, StringType())
df.withColumn("title_name", add_prefix_udf(col("name"))).show()
```

UDF với Nhiều Tham Số

```
from pyspark.sql.functions import udf, col
from pyspark.sql.types import IntegerType

# UDF nhận nhiều tham số
def calculate_bonus(salary, performance_score):
    return salary * 0.1 * performance_score

bonus_udf = udf(calculate_bonus, IntegerType())
df.withColumn("bonus", bonus_udf(col("salary"), col("performance_score"))).show()
```

UDF Trả Về Struct

```
from pyspark.sql.functions import udf, col
from pyspark.sql.types import StructType, StructField, StringType

# UDF trả về struct
def split_name(name):
    parts = name.split()
    return {"first": parts[0] if len(parts) > 0 else "",
            "last": parts[1] if len(parts) > 1 else ""}

schema = StructType([
    StructField("first", StringType(), True),
    StructField("last", StringType(), True)
])
```

```
split_udf = udf(split_name, schema)
df.withColumn("name_parts", split_udf(col("name"))).show()
```

Pandas UDF (Vectorized UDF)

Pandas UDF nhanh hơn nhiều so với UDF thông thường vì chúng chạy trên toàn bộ cột.

```
from pyspark.sql.functions import pandas_udf, col, PandasUDFType
import pandas as pd

# Scalar Pandas UDF
@pandas_udf("double")
def pandas_multiply(a: pd.Series, b: pd.Series) -> pd.Series:
    return a * b

df.withColumn("result", pandas_multiply(col("salary"), col("age"))).show()

# Grouped Map Pandas UDF (Lưu ý: GROUPED_MAP đã deprecated trong Spark 3.0+)
# Trong Spark 3.0+, sử dụng applyInPandas thay thế
def subtract_mean(pdf):
    pdf['salary'] = pdf['salary'] - pdf['salary'].mean()
    return pdf

# applyInPandas với schema
df.groupBy("department").applyInPandas(
    subtract_mean,
    schema=df.schema
).show()
```

Đăng Ký UDF cho Spark SQL

```
# Đăng ký UDF để dùng trong Spark SQL
spark.udf.register("capitalize_sql", capitalize_name, StringType())

# Sử dụng trong SQL
df.createOrReplaceTempView("employees")
spark.sql("SELECT name, capitalize_sql(name) as upper_name FROM employees").show()
```

Lưu Ý về UDF

- **Performance**: UDF thường chậm hơn built-in functions vì cần serialize/deserialize dữ liệu
- **Nên dùng**: Khi logic phức tạp không thể thực hiện với built-in functions
- **Tránh dùng**: Khi có thể dùng built-in functions hoặc SQL expressions
- **Pandas UDF**: Nhanh hơn UDF thông thường, nên dùng khi có thể

Broadcast Variables và Accumulators

Broadcast Variables

Broadcast Variables cho phép bạn lưu trữ dữ liệu read-only trên mỗi node để tránh gửi lại nhiều lần.

```
from pyspark.sql.functions import broadcast, udf, col
from pyspark.sql.types import StringType

# Tạo broadcast variable
lookup_dict = {"NY": "New York", "CA": "California", "TX": "Texas"}
broadcast_dict = spark.sparkContext.broadcast(lookup_dict)

# Sử dụng trong UDF
def lookup_state(code):
    return broadcast_dict.value.get(code, "Unknown")

lookup_udf = udf(lookup_state, StringType())
df.withColumn("state_name", lookup_udf(col("state_code"))).show()

# Broadcast Join (tự động khi bảng < 10MB, hoặc thủ công)
small_df = spark.createDataFrame([("NY", "New York")], ["code", "name"])
large_df.join(broadcast(small_df), large_df.code == small_df.code).show()
```

Accumulators

Accumulators cho phép bạn tích lũy giá trị từ các tasks khác nhau.

```
# Tạo accumulator
counter = spark.sparkContext.accumulator(0)

# Sử dụng trong RDD operations
def increment_counter(x):
    global counter
    if x > 100:
        counter += 1
    return x

rdd = spark.sparkContext.parallelize(range(1, 1000))
rdd.map(increment_counter).collect()

print(f"Total values > 100: {counter.value}")

# Named Accumulator
named_counter = spark.sparkContext.accumulator(0, "My Counter")
```

Custom Accumulators

```
from pyspark import AccumulatorParam
```

```
class VectorAccumulator(AccumulatorParam):
    def zero(self, initial_value):
        return [0.0] * len(initial_value)

    def addInPlace(self, v1, v2):
        return [x + y for x, y in zip(v1, v2)]

vector_acc = spark.sparkContext.accumulator([0.0, 0.0, 0.0], VectorAccumulator())
```

Partitioning và Coalescing

Repartition

```
# Repartition thành N partitions
df_repartitioned = df.repartition(10)

# Repartition theo cột (tạo partition cho mỗi giá trị cột)
df_repartitioned = df.repartition("department")

# Repartition theo nhiều cột
df_repartitioned = df.repartition("department", "year")
```

Coalesce

```
# Coalesce: Giảm số partitions (hiệu quả hơn repartition vì không shuffle)
df_coalesced = df.coalesce(5)

# Lưu ý: Coalesce chỉ giảm số partitions, không tăng
```

Số Partitions Tối Ưu

```
# Kiểm tra số partitions hiện tại
print(f"Number of partitions: {df.rdd.getNumPartitions()}")

# Quy tắc chung:
# - Số partitions = 2-3x số cores
# - Mỗi partition nên có 100-200MB dữ liệu
# - Tránh quá nhiều partitions (overhead)
# - Tránh quá ít partitions (không tận dụng parallelism)
```

PartitionBy khi Ghi

```

```

```
# Ghi dữ liệu với partition
df.write.partitionBy("department", "year") \
    .parquet("output/partitioned_data")

# Điều này tạo cấu trúc:
# output/partitioned_data/
#   department=Sales/
#     year=2023/
#       part-xxxxx.parquet
```

Caching và Persistence

Caching giúp lưu DataFrame/RDD trong memory để truy cập nhanh hơn khi dùng lại.

Cache và Persist

```
# Cache với storage level mặc định (MEMORY_AND_DISK)
df.cache()

# Persist với storage level tùy chỉnh
from pyspark import StorageLevel

# MEMORY_ONLY: Chỉ lưu trong memory
df.persist(StorageLevel.MEMORY_ONLY)

# MEMORY_AND_DISK: Lưu trong memory, nếu không đủ thì lưu disk
df.persist(StorageLevel.MEMORY_AND_DISK)

# MEMORY_ONLY_SER: Serialized trong memory
df.persist(StorageLevel.MEMORY_ONLY_SER)

# DISK_ONLY: Chỉ lưu trên disk
df.persist(StorageLevel.DISK_ONLY)

# MEMORY_AND_DISK_SER_2: Serialized, replicate 2 lần
df.persist(StorageLevel.MEMORY_AND_DISK_SER_2)
```

Unpersist

```
# Bỏ cache
df.unpersist()

# Bỏ cache và xóa ngay lập tức
df.unpersist(blocking=True)
```

Khi Nào Nên Cache

- DataFrame được sử dụng nhiều lần
- Kết quả của expensive operations (join, aggregation)
- Khi làm việc với iterative algorithms (ML)

Khi Nào KHÔNG Nên Cache

- DataFrame chỉ dùng 1 lần
- DataFrame quá lớn, không fit vào memory
- Khi có thể tính toán lại nhanh hơn

Join Strategies

Spark có nhiều chiến lược join khác nhau, mỗi loại phù hợp với trường hợp cụ thể.

Broadcast Hash Join

```
from pyspark.sql.functions import broadcast

# Tự động khi bảng nhỏ < 10MB (có thể cấu hình)
# Hoặc thủ công với broadcast()
large_df.join(broadcast(small_df), "id").show()

# Lợi ích: Tránh shuffle, nhanh
# Nhược điểm: Chỉ dùng được với bảng nhỏ
```

Sort Merge Join

```
# Mặc định cho các bảng lớn
# Yêu cầu: Cả 2 bảng phải được sort
df1.join(df2, df1.id == df2.id).show()

# Lợi ích: Hoạt động tốt với dữ liệu lớn
# Nhược điểm: Cần shuffle và sort
```

Shuffle Hash Join

```
# Khi một bảng vừa (không đủ nhỏ để broadcast)
# Spark tự động chọn
df1.join(df2, "id").show()

# Có thể bật/tắt bằng cấu hình
spark.conf.set("spark.sql.join.preferSortMergeJoin", "false")
```

Các Loại Join

```
# Inner Join (mặc định)
df1.join(df2, "id", "inner").show()

# Left Join
df1.join(df2, "id", "left").show()

# Right Join
df1.join(df2, "id", "right").show()

# Outer Join
df1.join(df2, "id", "outer").show()

# Left Semi Join (chỉ giữ dòng của df1 có match)
df1.join(df2, "id", "left_semi").show()

# Left Anti Join (chỉ giữ dòng của df1 KHÔNG có match)
df1.join(df2, "id", "left_anti").show()
```

Tối Ưu Join

```
# 1. Broadcast small tables
df1.join(broadcast(df2), "id")

# 2. Sử dụng cột partition cho join
df1.join(df2, ["partition_col", "id"])

# 3. Đảm bảo data skew không quá lớn
# 4. Tăng spark.sql.shuffle.partitions nếu cần
spark.conf.set("spark.sql.shuffle.partitions", "200")
```

Structured Streaming

Structured Streaming cho phép xử lý dữ liệu stream real-time.

Khái Niệm Cơ Bản

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import window, count

spark = SparkSession.builder \
    .appName("StructuredStreaming") \
    .master("local[*]") \
    .getOrCreate()
```

Đọc Stream từ File

```
# Đọc CSV files từ thư mục (monitor mode)
stream_df = spark.readStream \
    .schema(schema) \
    .option("maxFilesPerTrigger", 1) \
    .csv("input/streaming/")

# Xử lý dữ liệu
result = stream_df.groupBy("category").agg(count("*").alias("count"))
```

Ghi Stream

```
# Ghi vào console (để debug)
query = result.writeStream \
    .outputMode("complete") \
    .format("console") \
    .start()

query.awaitTermination()

# Ghi vào memory
query = result.writeStream \
    .outputMode("complete") \
    .format("memory") \
    .queryName("streaming_results") \
    .start()

# Ghi vào file
query = result.writeStream \
    .outputMode("append") \
    .format("parquet") \
    .option("checkpointLocation", "checkpoint/") \
    .option("path", "output/streaming/") \
    .start()
```

Output Modes

```
# Complete: Toàn bộ kết quả (cho aggregations)
.writeStream.outputMode("complete")

# Append: Chỉ thêm dòng mới (không có aggregations)
.writeStream.outputMode("append")

# Update: Chỉ cập nhật dòng thay đổi
.writeStream.outputMode("update")
```

Window Operations

```
from pyspark.sql.functions import window, col

# Window 5 phút, slide 1 phút
windowed_counts = stream_df \
    .withWatermark("timestamp", "10 minutes") \
    .groupBy(
        window(col("timestamp"), "5 minutes", "1 minute"),
        col("category")
    ) \
    .agg(count("*").alias("count"))
```

Watermark

```
# Watermark để xử lý late data
stream_df \
    .withWatermark("timestamp", "10 minutes") \
    .groupBy(window(col("timestamp"), "5 minutes"), "category") \
    .agg(count("*"))
```

Kafka Integration

```
# Đọc từ Kafka
df = spark.readStream \
    .format("kafka") \
    .option("kafka.bootstrap.servers", "localhost:9092") \
    .option("subscribe", "topic-name") \
    .load()

# Parse JSON
from pyspark.sql.functions import from_json, col
from pyspark.sql.types import StructType, StringType

schema = StructType().add("name", StringType())
df_parsed = df.select(
    from_json(col("value").cast("string"), schema).alias("data")
)

# Ghi vào Kafka
df.writeStream \
    .format("kafka") \
    .option("kafka.bootstrap.servers", "localhost:9092") \
    .option("topic", "output-topic") \
    .start()
```

Spark MLlib Cơ Bản

MLlib là thư viện machine learning của Spark.

Pipeline

```
from pyspark.ml import Pipeline
from pyspark.ml.feature import VectorAssembler, StringIndexer
from pyspark.ml.regression import LinearRegression

# Tạo features
assembler = VectorAssembler(
    inputCols=["feature1", "feature2", "feature3"],
    outputCol="features"
)

# Indexer cho categorical variables
indexer = StringIndexer(inputCol="category", outputCol="categoryIndex")

# Model
lr = LinearRegression(featuresCol="features", labelCol="label")

# Pipeline
pipeline = Pipeline(stages=[indexer, assembler, lr])
model = pipeline.fit(training_df)
```

Feature Engineering

```
from pyspark.ml.feature import (
    VectorAssembler, StringIndexer, OneHotEncoder,
    StandardScaler, MinMaxScaler, Tokenizer, CountVectorizer
)

# Vector Assembler
assembler = VectorAssembler(
    inputCols=["col1", "col2"],
    outputCol="features"
)

# String Indexer
indexer = StringIndexer(inputCol="category", outputCol="categoryIndex")

# One Hot Encoder
encoder = OneHotEncoder(inputCol="categoryIndex", outputCol="categoryVec")

# Standard Scaler
scaler = StandardScaler(
    inputCol="features",
    outputCol="scaledFeatures",
    withStd=True,
    withMean=True
)
```

```
# Tokenizer
tokenizer = Tokenizer(inputCol="text", outputCol="words")

# Count Vectorizer
cv = CountVectorizer(inputCol="words", outputCol="features")
```

Model Training

```
from pyspark.ml.regression import LinearRegression
from pyspark.ml.classification import LogisticRegression, RandomForestClassifier
from pyspark.ml.clustering import KMeans

# Linear Regression
lr = LinearRegression(featuresCol="features", labelCol="label")
lr_model = lr.fit(training_df)

# Logistic Regression
logreg = LogisticRegression(featuresCol="features", labelCol="label")
logreg_model = logreg.fit(training_df)

# Random Forest
rf = RandomForestClassifier(featuresCol="features", labelCol="label")
rf_model = rf.fit(training_df)

# K-Means Clustering
kmeans = KMeans(featuresCol="features", k=3)
kmeans_model = kmeans.fit(training_df)
```

Model Evaluation

```
from pyspark.ml.evaluation import RegressionEvaluator,
BinaryClassificationEvaluator

# Regression
evaluator = RegressionEvaluator(
    labelCol="label",
    predictionCol="prediction",
    metricName="rmse"
)
rmse = evaluator.evaluate(predictions)

# Classification
evaluator = BinaryClassificationEvaluator(
    labelCol="label",
    rawPredictionCol="rawPrediction",
    metricName="areaUnderROC"
)
auc = evaluator.evaluate(predictions)
```

Cross Validation

```
from pyspark.ml.tuning import CrossValidator, ParamGridBuilder

# Param grid
paramGrid = ParamGridBuilder() \
    .addGrid(lr.regParam, [0.1, 0.01]) \
    .addGrid(lr.maxIter, [10, 20]) \
    .build()

# Cross validator
cv = CrossValidator(
    estimator=pipeline,
    estimatorParamMaps=paramGrid,
    evaluator=evaluator,
    numFolds=3
)

cv_model = cv.fit(training_df)
```

Performance Tuning

Cấu Hình Spark

```
from pyspark.sql import SparkSession

spark = SparkSession.builder \
    .appName("Tuned App") \
    .config("spark.sql.shuffle.partitions", "200") \
    .config("spark.executor.memory", "4g") \
    .config("spark.executor.cores", "4") \
    .config("spark.driver.memory", "2g") \
    .config("spark.sql.adaptive.enabled", "true") \
    .config("spark.sql.adaptive.coalescePartitions.enabled", "true") \
    .getOrCreate()
```

Các Tham Số Quan Trọng

```
# Shuffle partitions
spark.conf.set("spark.sql.shuffle.partitions", "200")

# Broadcast join threshold (10MB mặc định)
spark.conf.set("spark.sql.autoBroadcastJoinThreshold", "50MB")

# Adaptive Query Execution (AQE)
```

```
spark.conf.set("spark.sql.adaptive.enabled", "true")
spark.conf.set("spark.sql.adaptive.coalescePartitions.enabled", "true")

# Dynamic partition pruning
spark.conf.set("spark.sql.optimizer.dynamicPartitionPruning.enabled", "true")

# Compression
spark.conf.set("spark.sql.parquet.compression.codec", "snappy")
```

Tối Ưu Query

```
# 1. Sử dụng column pruning
df.select("col1", "col2")  # Chỉ chọn cột cần thiết

# 2. Push down predicates
df.filter("age > 25")  # Filter sớm

# 3. Broadcast small tables
df1.join(broadcast(df2), "id")

# 4. Repartition hợp lý
df.repartition(200, "key")

# 5. Cache khi cần
df.cache()

# 6. Sử dụng Parquet format
df.write.parquet("output.parquet")
```

Monitoring Performance

```
# Xem execution plan
df.explain(True)

# Xem với formatted output
df.explain("formatted")

# Spark UI
# Truy cập: http://localhost:4040
# Xem jobs, stages, tasks, memory usage
```

Advanced DataFrame Operations

Pivot và Unpivot

```
# Pivot
df.groupBy("year").pivot("month").sum("sales").show()

# Unpivot (stack)
from pyspark.sql.functions import expr
df.select("year", expr("stack(12, " +
    ", ".join([f"'{i}', month_{i}" for i in range(1, 13)]) +
    ") as (month, sales)")).show()
```

Array Operations

```
from pyspark.sql.functions import array, array_contains, size, explode

# Tạo array column
df.withColumn("array_col", array("col1", "col2"))

# Kiểm tra array contains
df.filter(array_contains("array_col", "value"))

# Kích thước array
df.withColumn("array_size", size("array_col"))

# Explode array thành nhiều dòng
df.select("id", explode("array_col").alias("value"))
```

Map Operations

```
from pyspark.sql.functions import map, map_keys, map_values, lit, col

# Tạo map column
df.withColumn("map_col", map(lit("key1"), col("value1")))

# Lấy keys
df.select(map_keys("map_col"))

# Lấy values
df.select(map_values("map_col"))
```

JSON Operations

```
from pyspark.sql.functions import get_json_object, json_tuple, from_json, to_json,
col, struct
from pyspark.sql.types import StructType, StringType

# Parse JSON string
df.select(get_json_object(col("json_col"), "$.field"))
```

```
# Multiple fields
df.select(json_tuple(col("json_col"), "field1", "field2"))

# Parse to struct
schema = StructType().add("field1", StringType())
df.select(from_json(col("json_col"), schema))

# Convert to JSON
df.select(to_json(struct("col1", "col2")))
```

Complex Types

```
from pyspark.sql.functions import struct, when, col

# Struct
df.withColumn("struct_col", struct("col1", "col2"))

# Access struct fields
df.select("struct_col.col1")

# Nested operations
df.withColumn("complex", struct(
    struct("nested1", "nested2").alias("inner"),
    "outer"
))
```

Handling Large Datasets

Chunk Processing

```
from pyspark.sql.functions import col

# Cách 1: Sử dụng filter với điều kiện (khuyến nghị - hiệu quả nhất)
# Yêu cầu: DataFrame phải có cột ID hoặc key tuần tự
from pyspark.sql.functions import max as max_func
batch_size = 10000
max_id = df.agg(max_func("id")).collect()[0][0]  # Giả sử có cột "id"

for start_id in range(0, int(max_id) + 1, batch_size):
    end_id = min(start_id + batch_size, int(max_id) + 1)
    batch = df.filter((col("id") >= start_id) & (col("id") < end_id))
    process_batch(batch)

# Cách 2: Sử dụng window function với row_number (không hiệu quả cho dữ liệu lớn)
# from pyspark.sql.functions import row_number
# from pyspark.sql.window import Window
#
```

```
# window_spec = Window.orderBy("id")  # Phải có cột để order
# df_with_row = df.withColumn("row_num", row_number().over(window_spec))
# for i in range(0, total_rows, batch_size):
#     batch = df_with_row.filter(
#         (col("row_num") > i) & (col("row_num") <= i + batch_size)
#     ).drop("row_num")
#     process_batch(batch)
```

Incremental Processing

```
from pyspark.sql.functions import col, max

# Xử lý incremental
last_processed_id = get_last_processed_id()
new_data = df.filter(col("id") > last_processed_id)

# Xử lý new_data
process(new_data)

# Update last processed
update_last_processed_id(new_data.agg(max("id")).collect()[0][0])
```

Sampling

```
# Random sampling
sample_df = df.sample(fraction=0.1, seed=42)

# Stratified sampling
sample_df = df.sampleBy("category", fractions={0: 0.1, 1: 0.2}, seed=42)
```

Handling Skewed Data

```
from pyspark.sql.functions import col, rand, concat, lit

# Phát hiện skew
df.groupBy("key").count().orderBy("count", ascending=False).show()

# Xử lý skew: Salt key
df_salted = df.withColumn("salted_key",
    concat(col("key"), lit("_"), (rand() * 10).cast("int"))
)

# Join với salted key
result = df_salted.join(df2_salted, "salted_key")
result = result.drop("salted_key")
```

Memory Management

```
# Giảm số partitions nếu quá nhiều
df.coalesce(100)

# Tăng partitions nếu quá ít
df.repartition(200)

# Cache selective
df.cache()
# ... operations
df.unpersist()

# Sử dụng disk storage level nếu memory không đủ
from pyspark import StorageLevel
df.persist(StorageLevel.MEMORY_AND_DISK_SER)
```

Error Handling và Debugging

Exception Handling

```
try:
    result = df.collect()
except Exception as e:
    print(f"Error: {e}")
    # Log error
    # Handle gracefully
```

Debugging với Explain

```
# Xem execution plan
df.explain()

# Xem chi tiết
df.explain(True)

# Xem formatted plan
df.explain("formatted")

# Xem codegen plan
df.explain("codegen")
```

Logging

```

```

```
import logging

# Cấu hình logging
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

# Sử dụng trong code
logger.info("Processing started")
logger.error("Error occurred")
```

Checkpoint

```
# Checkpoint để break lineage
df.checkpoint()

# Hoặc với eager checkpoint
df.localCheckpoint(True)
```

Testing

```
# Unit test với small data
test_data = [("test", 1)]
test_df = spark.createDataFrame(test_data, ["name", "value"])
result = process_function(test_df)
assert result.count() > 0

# Integration test với sample data
sample_df = df.sample(0.01)
result = complex_operation(sample_df)
result.show()
```

Best Practices Nâng Cao

1. **Luôn sử dụng AQE** (Adaptive Query Execution)
2. **Monitor Spark UI** để hiểu performance bottlenecks
3. **Sử dụng Parquet** cho storage format
4. **Partition hợp lý** dựa trên query patterns
5. **Broadcast small tables** khi join
6. **Cache selectively** - chỉ cache khi thực sự cần
7. **Tránh data skew** - sử dụng salting nếu cần
8. **Sử dụng checkpoint** để break long lineages
9. **Tối ưu shuffle operations** - giảm số lần shuffle
10. **Sử dụng column pruning** và predicate pushdown
11. **Tránh collect()** - chỉ dùng khi thực sự cần

12. **Sử dụng Pandas UDF** thay vì UDF thông thường khi có thể

Ví Dụ Hoàn Chỉnh: ETL Pipeline

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, when, sum, avg, window
from pyspark.sql.window import Window

# Khởi tạo Spark với cấu hình tối ưu
spark = SparkSession.builder \
    .appName("Advanced ETL Pipeline") \
    .config("spark.sql.adaptive.enabled", "true") \
    .config("spark.sql.shuffle.partitions", "200") \
    .getOrCreate()

# Đọc dữ liệu
raw_df = spark.read.parquet("input/data.parquet")

# Clean và transform
cleaned_df = raw_df \
    .filter(col("value").isNotNull()) \
    .withColumn("category",
        when(col("type") == "A", "Category_A")
        .when(col("type") == "B", "Category_B")
        .otherwise("Other")
    )

# Window function cho running totals
from pyspark.sql.functions import lag
windowSpec = Window.partitionBy("category").orderBy("date")
result_df = cleaned_df \
    .withColumn("running_total",
        sum("amount").over(windowSpec)
    ) \
    .withColumn("prev_amount",
        lag("amount", 1).over(windowSpec)
    )

# Aggregation
aggregated = result_df.groupBy("category").agg(
    sum("amount").alias("total"),
    avg("amount").alias("average")
)

# Cache kết quả
aggregated.cache()

# Ghi kết quả
aggregated.write \
    .mode("overwrite") \
    .parquet("output/aggregated.parquet")
```

```
# Unpersist
aggregated.unpersist()

spark.stop()
```

Tài Liệu Tham Khảo

- Spark SQL Performance Tuning
- Structured Streaming Guide
- MLlib Guide
- Spark Configuration
- Best Practices