Hướng Dẫn Đồ Án: Xử Lý Dữ Liệu Lớn Thời Gian Thực với PySpark

Thông Tin Tổng Quan

Mục Tiêu Đồ Án

Thiết kế và triển khai một pipeline xử lý dữ liệu lớn có khả năng mở rộng và chịu lỗi, sử dụng Apache PySpark với kiến trúc Medallion để phân tích dữ liệu chuyến đi taxi NYC.

Kiến Trúc và Công Nghệ

- **Framework**: Apache PySpark 3.4+
- **Kiến trúc**: Medallion (Bronze → Silver → Gold)
- **Xử lý**: Batch Processing + Streaming Processing
- **Định dạng dữ liệu**: Parquet, Delta Lake (tuỳ chọn)
- **Visualization**: Matplotlib, Seaborn, Plotly
- **Environment**: Jupyter Notebook, Python 3.8+

Cấu Trúc Nhóm và Phân Công

- **Team Leader/Data Engineer**: Quản lý dự án, thiết kế kiến trúc tổng thể
- **Data Pipeline Engineer**: Phát triển batch processing pipeline
- **Streaming Engineer**: Phát triển real-time streaming pipeline
- **Analytics Engineer**: Data quality, visualization và insight

---

Phần 1: Chuẩn Bị Dữ Liệu và Môi Trường

1.1 Nguồn Dữ Liệu Chính

NYC Yellow Taxi Trip Records

- **Nguồn chính**: NYC Taxi & Limousine Commission (TLC) - https://www.nyc.gov/site/tlc/about/tlc-trip-record-data.page
- **Định dạng**: Parquet files, được cập nhật hàng tháng với độ trễ 2 tháng
- **Kích thước**: Khoảng 10-15 triệu trips/tháng, ~1-2GB/file
- **Thời gian**: Dữ liệu từ 2009 đến hiện tại

Nguồn Dữ Liệu Bổ Sung

1. **AWS Open Data**: Registry of Open Data on AWS - nyc-tlc-trip-records-pds
2. **Kaggle**: NYC Yellow Taxi Trip Data datasets
3. **Azure Open Datasets**: Microsoft Azure Open Datasets - NYC Taxi

1.2 Schema Dữ Liệu (Data Dictionary)

Theo official TLC data dictionary, dataset chứa các trường sau:

Field Name	Data Type	Description
VendorID	Integer	Mã nhà cung cấp TPEP (1=Creative Mobile, 2=Curb Mobility, 6=Myle, 7=Helix)
tpep_pickup_datetime	Timestamp	Thời gian bắt đầu chuyến đi
tpep_dropoff_datetime	Timestamp	Thời gian kết thúc chuyến đi
passenger_count	Integer	Số hành khách
trip_distance	Double	Khoảng cách chuyến đi (miles)
RatecodeID	Integer	Mã loại giá (1=Standard, 2=JFK, 3=Newark, 4=Nassau/Westchester, 5=Negotiated, 6=Group)
store_and_fwd_flag	String	Cờ lưu trữ và chuyển tiếp (Y/N)
PULocationID	Integer	ID vùng đón khách
DOLocationID	Integer	ID vùng trả khách
payment_type	Integer	Loại thanh toán (1=Credit, 2=Cash, 3=No charge, 4=Dispute, 5=Unknown, 6=Voided)
fare_amount	Double	Cước phí cơ bản
extra	Double	Phụ phí khác
mta_tax	Double	Thuế MTA
tip_amount	Double	Tiền tip
tolls_amount	Double	Phí cầu đường
improvement_surcharge	Double	Phụ phí cải thiện
total_amount	Double	Tổng tiền
congestion_surcharge	Double	Phụ phí ùn tắc
airport_fee	Double	Phí sân bay
cbd_congestion_fee	Double	Phí ùn tắc CBD (từ 2025)

1.3 Cài Đặt Môi Trường

```
# requirements.txt
pyspark==3.4.1
pandas==2.0.3
matplotlib==3.7.2
seaborn==0.12.2
plotly==5.15.0
jupyter==1.0.0
```

```
delta-spark==2.4.0
pyarrow==12.0.1
```

```
# Cài đặt Spark
pip install pyspark[sql]
pip install -r requirements.txt

# Download sample data
wget https://d37ci6vzurychx.cloudfront.net/trip-data/yellow_tripdata_2024-
01.parquet
```

Phần 2: Thiết Kế Pipeline theo Kiến Trúc Medallion

2.1 Giai Đoạn 1: Bronze Layer (Raw Data Ingestion)

Mục tiêu: Nhập dữ liệu thô từ nhiều nguồn, không biến đổi

```
# bronze_layer.py
from pyspark.sql import SparkSession
from pyspark.sql.types import *

def create_bronze_pipeline():
    spark = SparkSession.builder \
        .appName("NYC_Taxi_Bronze_Layer") \
        .config("spark.sql.adaptive.enabled", "true") \
        .config("spark.sql.adaptive.coalescePartitions.enabled", "true") \
        .getOrCreate()

    # Schema định nghĩa cho Yellow Taxi
    yellow_taxi_schema = StructType([
        StructField("VendorID", IntegerType(), True),
        StructField("tpep_pickup_datetime", TimestampType(), True),
        StructField("tpep_dropoff_datetime", TimestampType(), True),
        StructField("passenger_count", IntegerType(), True),
        StructField("trip_distance", DoubleType(), True),
        StructField("RatecodeID", IntegerType(), True),
        StructField("store_and_fwd_flag", StringType(), True),
        StructField("PULocationID", IntegerType(), True),
        StructField("DOLocationID", IntegerType(), True),
        StructField("payment_type", IntegerType(), True),
        StructField("fare_amount", DoubleType(), True),
        StructField("extra", DoubleType(), True),
        StructField("mta_tax", DoubleType(), True),
        StructField("tip_amount", DoubleType(), True),
        StructField("tolls_amount", DoubleType(), True),
        StructField("improvement_surcharge", DoubleType(), True),
        StructField("total_amount", DoubleType(), True),
        StructField("congestion_surcharge", DoubleType(), True),
```

```
        StructField("airport_fee", DoubleType(), True),
        StructField("cbd_congestion_fee", DoubleType(), True)
    ])

    # Đọc dữ liệu từ nhiều nguồn
    df_yellow = spark.read \
        .schema(yellow_taxi_schema) \
        .option("multiline", "true") \
        .parquet("data/raw/yellow_tripdata_*.parquet")

    # Thêm metadata cho tracking
    from pyspark.sql.functions import current_timestamp, lit

    df_bronze = df_yellow \
        .withColumn("ingestion_timestamp", current_timestamp()) \
        .withColumn("source_file", lit("yellow_taxi")) \
        .withColumn("data_layer", lit("bronze"))

    # Lưu vào Bronze layer với partitioning
    df_bronze.write \
        .mode("overwrite") \
        .partitionBy("year", "month") \
        .parquet("data/bronze/yellow_taxi/")

    return df_bronze
```

Yêu cầu Bronze Layer:

- Giữ nguyên dữ liệu gốc, không làm sạch
- Thêm metadata: ingestion timestamp, source information
- Partition theo năm/tháng để tối ưu performance
- Fault-tolerance: checkpointing và error handling

2.2 Giai Đoạn 2: Silver Layer (Data Cleaning & Validation)

Mục tiêu: Làm sạch dữ liệu, validate và standardize

```
# silver_layer.py
from pyspark.sql.functions import *
from pyspark.sql.types import *

def create_silver_pipeline(spark, bronze_df):

    # Data Quality Rules
    df_silver = bronze_df \
        .filter(col("tpep_pickup_datetime").isNotNull()) \
        .filter(col("tpep_dropoff_datetime").isNotNull()) \
        .filter(col("trip_distance") > 0) \
        .filter(col("fare_amount") > 0) \
        .filter(col("total_amount") > 0) \
```

```
        .filter(col("passenger_count").between(1, 8)) \
        .filter(col("PULocationID").isNotNull()) \
        .filter(col("DOLocationID").isNotNull())

    # Feature Engineering
    df_silver = df_silver \
        .withColumn("trip_duration_minutes",
                    (unix_timestamp("tpep_dropoff_datetime") -
                     unix_timestamp("tpep_pickup_datetime")) / 60) \
        .withColumn("pickup_hour", hour("tpep_pickup_datetime")) \
        .withColumn("pickup_day_of_week", dayofweek("tpep_pickup_datetime")) \
        .withColumn("pickup_month", month("tpep_pickup_datetime")) \
        .withColumn("pickup_year", year("tpep_pickup_datetime")) \
        .withColumn("speed_mph",
                    when(col("trip_duration_minutes") > 0,
                         col("trip_distance") / (col("trip_duration_minutes") /
60))
                    .otherwise(0)) \
        .withColumn("tip_percentage",
                    when(col("fare_amount") > 0,
                         col("tip_amount") / col("fare_amount") * 100)
                    .otherwise(0))

    # Data Quality Filters
    df_silver = df_silver \
        .filter(col("trip_duration_minutes").between(1, 180)) \
        .filter(col("speed_mph") < 100) \
        .filter(col("tip_percentage") <= 50)

    # Add data quality scores
    df_silver = df_silver \
        .withColumn("quality_score",
                    when((col("trip_distance") > 0) &
                         (col("trip_duration_minutes") > 0) &
                         (col("fare_amount") > 0), 1.0)
                    .otherwise(0.5)) \
        .withColumn("processing_timestamp", current_timestamp())

    return df_silver
```

Yêu cầu Silver Layer:

- Data validation và outlier detection
- Feature engineering cho analysis
- Data quality scoring
- Standardization của data types

2.3 Giai Đoạn 3: Gold Layer (Business Analytics)

Mục tiêu: Tạo ra business-ready datasets cho phân tích

```
# gold_layer.py
def create_gold_aggregations(spark, silver_df):

    # 1. Hourly Trip Analytics
    hourly_stats = silver_df \
        .groupBy("pickup_year", "pickup_month", "pickup_hour") \
        .agg(
            count("*").alias("total_trips"),
            avg("fare_amount").alias("avg_fare"),
            avg("tip_amount").alias("avg_tip"),
            avg("trip_distance").alias("avg_distance"),
            avg("trip_duration_minutes").alias("avg_duration"),
            avg("passenger_count").alias("avg_passengers")
        )

    # 2. Location-based Analytics (Hotspots)
    location_stats = silver_df \
        .groupBy("PULocationID", "pickup_hour") \
        .agg(
            count("*").alias("pickup_count"),
            avg("fare_amount").alias("avg_fare_from_location"),
            avg("trip_distance").alias("avg_trip_distance")
        ) \
        .withColumn("pickup_density_rank",
                    dense_rank().over(Window.partitionBy("pickup_hour")
                                      .orderBy(desc("pickup_count"))))

    # 3. Payment Analytics
    payment_stats = silver_df \
        .groupBy("payment_type", "pickup_year", "pickup_month") \
        .agg(
            count("*").alias("payment_count"),
            avg("total_amount").alias("avg_total_amount"),
            avg("tip_amount").alias("avg_tip_amount"),
            sum("total_amount").alias("total_revenue")
        )

    # 4. Driver Performance Analytics
    vendor_stats = silver_df \
        .groupBy("VendorID", "pickup_year", "pickup_month") \
        .agg(
            count("*").alias("vendor_trip_count"),
            avg("trip_distance").alias("avg_trip_distance"),
            avg("speed_mph").alias("avg_speed"),
            avg("quality_score").alias("avg_quality_score")
        )

    # Save Gold Layer Tables
    hourly_stats.write.mode("overwrite").partitionBy("pickup_year",
"pickup_month") \
        .parquet("data/gold/hourly_trip_analytics/")

    location_stats.write.mode("overwrite").partitionBy("pickup_hour") \
```

```
        .parquet("data/gold/location_hotspots/")

    payment_stats.write.mode("overwrite").partitionBy("pickup_year",
"pickup_month") \
        .parquet("data/gold/payment_analytics/")

    vendor_stats.write.mode("overwrite").partitionBy("pickup_year",
"pickup_month") \
        .parquet("data/gold/vendor_performance/")

    return {
        "hourly_stats": hourly_stats,
        "location_stats": location_stats,
        "payment_stats": payment_stats,
        "vendor_stats": vendor_stats
    }
```

2.4 Giai Đoạn 4: Streaming Layer (Real-time Processing)

Mục tiêu: Xử lý dữ liệu real-time với Structured Streaming

```
# streaming_pipeline.py
from pyspark.sql import SparkSession
from pyspark.sql.functions import *
from pyspark.sql.types import *

def create_streaming_pipeline():
    spark = SparkSession.builder \
        .appName("NYC_Taxi_Streaming") \
        .config("spark.sql.streaming.checkpointLocation", "checkpoint/") \
        .getOrCreate()

    # Đọc streaming data từ file source (simulation)
    streaming_df = spark.readStream \
        .format("parquet") \
        .schema(yellow_taxi_schema) \
        .option("path", "data/streaming_input/") \
        .option("maxFilesPerTrigger", 1) \
        .load()

    # Real-time transformations
    streaming_processed = streaming_df \
        .withColumn("pickup_hour", hour("tpep_pickup_datetime")) \
        .withColumn("trip_duration_minutes",
                    (unix_timestamp("tpep_dropoff_datetime") -
                     unix_timestamp("tpep_pickup_datetime")) / 60) \
        .filter(col("trip_duration_minutes") > 0)

    # Real-time aggregations (sliding windows)
    windowed_counts = streaming_processed \
        .withWatermark("tpep_pickup_datetime", "10 minutes") \
```

```
        .groupBy(
            window("tpep_pickup_datetime", "5 minutes", "1 minute"),
            "PULocationID"
        ) \
        .agg(
            count("*").alias("trip_count"),
            avg("fare_amount").alias("avg_fare"),
            avg("trip_duration_minutes").alias("avg_duration")
        ) \
        .withColumn("processing_time", current_timestamp())

    # Output to console for monitoring
    query_console = windowed_counts.writeStream \
        .outputMode("update") \
        .format("console") \
        .option("truncate", False) \
        .trigger(processingTime="30 seconds") \
        .start()

    # Output to files for persistence
    query_files = windowed_counts.writeStream \
        .outputMode("append") \
        .format("parquet") \
        .option("path", "data/streaming_output/realtime_aggregations/") \
        .option("checkpointLocation", "checkpoint/streaming_agg/") \
        .trigger(processingTime="1 minute") \
        .start()

    return query_console, query_files
```

Phần 3: Implementation Guide

3.1 Cấu Trúc Thư Mục Dự Án

```
nyc_taxi_pipeline/
├── data/
│   ├── raw/                    # Raw parquet files
│   ├── bronze/                 # Bronze layer
│   ├── silver/                 # Silver layer
│   ├── gold/                   # Gold layer
│   └── streaming_input/        # Streaming simulation
├── src/
│   ├── bronze_layer.py
│   ├── silver_layer.py
│   ├── gold_layer.py
│   ├── streaming_pipeline.py
│   ├── data_quality.py
│   └── utils.py
├── notebooks/
│   ├── 01_data_exploration.ipynb
```

```
│   ├── 02_pipeline_development.ipynb
│   ├── 03_analytics_dashboard.ipynb
│   └── 04_streaming_demo.ipynb
├── config/
│   ├── spark_config.py
│   └── pipeline_config.yaml
├── tests/
│   ├── test_bronze_layer.py
│   ├── test_silver_layer.py
│   └── test_streaming.py
├── checkpoint/                 # Streaming checkpoints
├── logs/                      # Application logs
├── requirements.txt
├── README.md
└── main_pipeline.py           # Main execution script
```

3.2 Main Pipeline Execution

```python
# main_pipeline.py
from src.bronze_layer import create_bronze_pipeline
from src.silver_layer import create_silver_pipeline
from src.gold_layer import create_gold_aggregations
from src.streaming_pipeline import create_streaming_pipeline
from pyspark.sql import SparkSession
import logging

def main():
    # Setup logging
    logging.basicConfig(level=logging.INFO)
    logger = logging.getLogger(__name__)

    # Initialize Spark
    spark = SparkSession.builder \
        .appName("NYC_Taxi_Pipeline_Main") \
        .config("spark.sql.adaptive.enabled", "true") \
        .config("spark.sql.adaptive.coalescePartitions.enabled", "true") \
        .config("spark.sql.adaptive.advisoryPartitionSizeInBytes", "128MB") \
        .getOrCreate()

    try:
        # Step 1: Bronze Layer
        logger.info("Starting Bronze Layer Processing...")
        bronze_df = create_bronze_pipeline()
        logger.info(f"Bronze Layer: {bronze_df.count()} records processed")

        # Step 2: Silver Layer
        logger.info("Starting Silver Layer Processing...")
        silver_df = create_silver_pipeline(spark, bronze_df)
        silver_df.cache()  # Cache for multiple uses
        logger.info(f"Silver Layer: {silver_df.count()} records processed")
```

```
        # Step 3: Gold Layer
        logger.info("Starting Gold Layer Processing...")
        gold_tables = create_gold_aggregations(spark, silver_df)
        logger.info("Gold Layer aggregations completed")

        # Step 4: Streaming (optional)
        if "--streaming" in sys.argv:
            logger.info("Starting Streaming Pipeline...")
            query_console, query_files = create_streaming_pipeline()
            query_console.awaitTermination()

    except Exception as e:
        logger.error(f"Pipeline execution failed: {str(e)}")
        raise
    finally:
        spark.stop()

if __name__ == "__main__":
    main()
```

3.3 Data Quality Framework

```
# data_quality.py
from pyspark.sql.functions import *
from pyspark.sql.types import *

class DataQualityChecker:
    def __init__(self, spark_session):
        self.spark = spark_session

    def check_data_quality(self, df, layer_name):
        """Comprehensive data quality assessment"""

        quality_metrics = {}

        # 1. Completeness Check
        total_rows = df.count()
        for col_name in df.columns:
            null_count = df.filter(col(col_name).isNull()).count()
            quality_metrics[f"{col_name}_completeness"] = (total_rows -
null_count) / total_rows

        # 2. Validity Check
        if layer_name == "silver":
            # Business rule validations
            valid_trips = df.filter(
                (col("trip_distance") > 0) &
                (col("fare_amount") > 0) &
                (col("trip_duration_minutes") > 0) &
                (col("passenger_count").between(1, 8))
            ).count()
```

```
            quality_metrics["business_rule_validity"] = valid_trips / total_rows

        # 3. Consistency Check
        amount_consistency = df.filter(
            col("total_amount") >= (col("fare_amount") + col("extra") +
col("mta_tax"))
        ).count()
        quality_metrics["amount_consistency"] = amount_consistency / total_rows

        # 4. Uniqueness Check (if applicable)
        distinct_rows = df.distinct().count()
        quality_metrics["uniqueness"] = distinct_rows / total_rows

        return quality_metrics

    def generate_quality_report(self, quality_metrics):
        """Generate quality report"""
        report = []
        for metric, score in quality_metrics.items():
            status = "PASS" if score >= 0.95 else "WARN" if score >= 0.8 else
"FAIL"
            report.append(f"{metric}: {score:.2%} - {status}")

        return "\n".join(report)
```

Phần 4: Analytics và Visualization

4.1 Key Performance Indicators (KPIs)

1. **Operational KPIs**:
    - Total trips per hour/day
    - Average trip duration
    - Average trip distance
    - Peak hour identification
2. **Financial KPIs**:
    - Total revenue per period
    - Average fare per trip
    - Tip percentage by payment type
    - Revenue by location zones
3. **Service Quality KPIs**:
    - Data completeness rates
    - Processing latency
    - Error rates in pipeline

4.2 Visualization Examples

```python
# analytics_dashboard.py
import matplotlib.pyplot as plt
import seaborn as sns
import plotly.express as px
import plotly.graph_objects as go

def create_analytics_dashboard(gold_data):

    # 1. Trip Volume Heatmap
    hourly_df = gold_data["hourly_stats"].toPandas()

    fig, axes = plt.subplots(2, 2, figsize=(16, 12))

    # Hourly trip distribution
    axes[0,0].plot(hourly_df.groupby('pickup_hour')['total_trips'].mean())
    axes[0,0].set_title('Average Trips by Hour of Day')
    axes[0,0].set_xlabel('Hour')
    axes[0,0].set_ylabel('Number of Trips')

    # Fare distribution
    axes[0,1].hist(hourly_df['avg_fare'], bins=50, alpha=0.7)
    axes[0,1].set_title('Distribution of Average Fares')
    axes[0,1].set_xlabel('Average Fare ($)')
    axes[0,1].set_ylabel('Frequency')

    # Location hotspots
    location_df = gold_data["location_stats"].toPandas()
    top_locations = location_df.nlargest(20, 'pickup_count')
    axes[1,0].barh(range(len(top_locations)), top_locations['pickup_count'])
    axes[1,0].set_title('Top 20 Pickup Locations')
    axes[1,0].set_xlabel('Number of Pickups')

    # Payment method analysis
    payment_df = gold_data["payment_stats"].toPandas()
    payment_summary = payment_df.groupby('payment_type')['payment_count'].sum()
    axes[1,1].pie(payment_summary.values, labels=payment_summary.index,
autopct='%1.1f%%')
    axes[1,1].set_title('Payment Methods Distribution')

    plt.tight_layout()
    plt.savefig('analytics_dashboard.png', dpi=300, bbox_inches='tight')
    plt.show()

def create_interactive_dashboard(gold_data):
    """Create interactive Plotly dashboard"""

    hourly_df = gold_data["hourly_stats"].toPandas()

    # Interactive time series
    fig = go.Figure()
    fig.add_trace(go.Scatter(
        x=hourly_df['pickup_hour'],
        y=hourly_df['total_trips'],
```

```
        mode='lines+markers',
        name='Total Trips',
        line=dict(color='blue', width=2)
    ))

    fig.update_layout(
        title='NYC Taxi Trips by Hour',
        xaxis_title='Hour of Day',
        yaxis_title='Number of Trips',
        hovermode='x unified'
    )

    fig.show()
```

Phần 5: Deliverables và Assessment

5.1 Yêu Cầu Nộp Bài

Code Deliverables:

1. **Bronze Layer Script** (`bronze_layer.py`) - 15 điểm
2. **Silver Layer Script** (`silver_layer.py`) - 20 điểm
3. **Gold Layer Script** (`gold_layer.py`) - 20 điểm
4. **Streaming Pipeline** (`streaming_pipeline.py`) - 20 điểm
5. **Main Pipeline** (`main_pipeline.py`) - 10 điểm

Documentation Deliverables:

1. **Technical Documentation** (PDF) - 10 điểm
2. **Analytics Notebook** (Jupyter) - 15 điểm
3. **Data Quality Report** - 10 điểm

Presentation:

1. **Live Demo** - 20 điểm
2. **Architecture Explanation** - 10 điểm
3. **Q&A Session** - 10 điểm

5.2 Evaluation Criteria

Tiêu chí	Trọng số	Mô tả
Code Quality	25%	Clean code, documentation, error handling
Architecture Design	20%	Medallion implementation, scalability
Data Processing	20%	ETL correctness, performance optimization
Streaming Implementation	15%	Real-time processing, fault tolerance

Tiêu chí	Trọng số	Mô tả
Analytics & Insights	10%	Business value, visualization quality
Documentation	10%	Technical docs, user guides

5.3 Bonus Points

- **Delta Lake Integration** (+5 điểm)
- **Advanced Streaming** (Kafka integration) (+5 điểm)
- **Machine Learning Pipeline** (+10 điểm)
- **Cloud Deployment** (AWS/Azure) (+10 điểm)
- **Advanced Monitoring** (Spark UI analysis) (+5 điểm)

Phần 6: Project Timeline

Week 1-2: Setup & Data Exploration

- Environment setup
- Data download và exploration
- Team role assignment
- Architecture design

Week 3-4: Bronze & Silver Layer Development

- Implement data ingestion
- Data cleaning và validation
- Quality framework setup
- Unit testing

Week 5-6: Gold Layer & Analytics

- Business logic implementation
- Aggregation pipelines
- Analytics development
- Performance tuning

Week 7-8: Streaming & Integration

- Streaming pipeline development
- End-to-end testing
- Documentation
- Final presentation prep

Phần 7: Troubleshooting và Best Practices

7.1 Common Issues

1. **Memory Issues**:

```
# Solution: Optimize partitioning and caching
df.repartition(200).cache()
```

2. **Slow Performance**:

```
# Solution: Broadcast smaller tables
spark.conf.set("spark.sql.adaptive.enabled", "true")
```

3. **Streaming Lag**:

```
# Solution: Adjust trigger intervals
.trigger(processingTime="10 seconds")
```

7.2 Performance Optimization

1. **Partitioning Strategy**: Partition by date columns
2. **Caching**: Cache frequently accessed DataFrames
3. **Broadcast Joins**: For small lookup tables
4. **Adaptive Query Execution**: Enable AQE features
5. **Resource Allocation**: Tune executor memory and cores

7.3 Data Quality Best Practices

1. **Schema Evolution**: Handle schema changes gracefully
2. **Data Validation**: Implement comprehensive checks
3. **Monitoring**: Set up alerts for data quality issues
4. **Recovery**: Implement data recovery mechanisms

Tài Liệu Tham Khảo

1. **Official Documentation**:
   - Apache Spark Documentation
   - PySpark API Reference
2. **NYC TLC Data Sources**:
   - NYC TLC Trip Record Data
   - AWS Open Data Registry
3. **Learning Resources**:
   - Spark: The Definitive Guide
   - Learning Spark (2nd Edition)

- Databricks Academy Training

4. **Community Resources**:
   - Stack Overflow Spark Tags
   - Spark User Mailing List
   - GitHub Spark Examples

Lưu ý: Đây là hướng dẫn chi tiết cho đồ án Big Data. Sinh viên cần customize theo yêu cầu cụ thể của trường và có thể mở rộng thêm các tính năng advanced như Machine Learning integration, Cloud deployment, hoặc Real-time Dashboard.