Homework 2: MapReduce Concepts & Spark Fundamentals

Points: 20 | Due: See WebCampus for deadline

Author: Richard Young, Ph.D. | UNLV Lee Business School

Compute: CPU (free tier)

Learning Objectives

1. **Set up** Apache Spark on Google Colab
2. **Understand** how Spark partitions data for parallel processing
3. **Measure and compare** processing performance with different configurations
4. **Apply** K-Means clustering and interpret business results
5. **Explain** distributed computing through your own diagram

---

Why This Matters for Business

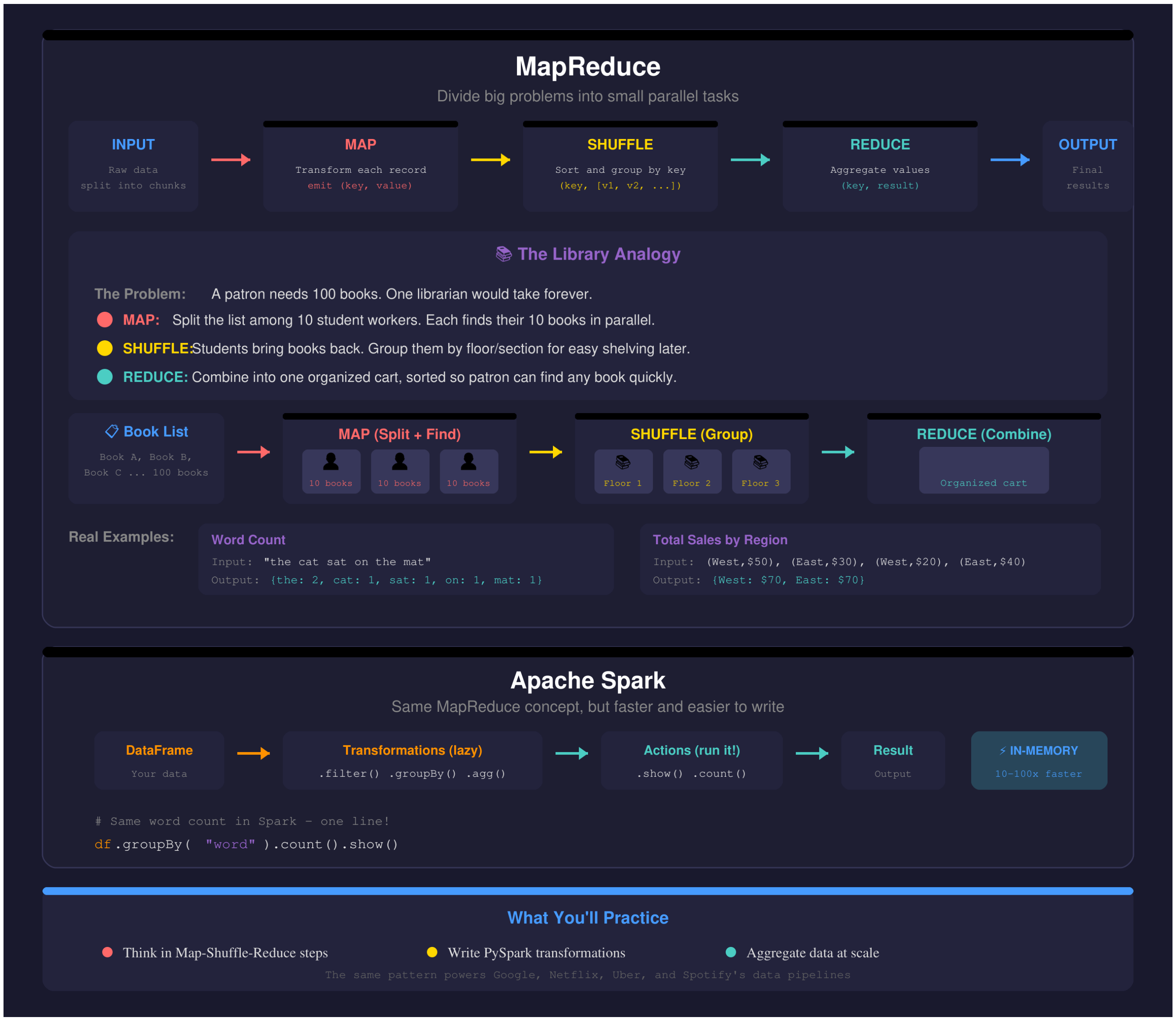
> **Scale or Fail:** Uber processes 100+ petabytes of data across their analytics platform. When your single-machine Pandas script takes 3 days to run, Spark can finish it in minutes by distributing work across hundreds of machines.
>
> **Real-Time Analytics:** Walmart processes over 1 million customer transactions per hour during Black Friday. Spark Streaming enables real-time inventory alerts—when a product sells faster than expected, stores get restocked before shelves empty.
>
> **Cost Optimization:** Airbnb reduced their data processing costs by 50% by understanding Spark partitioning. Poorly partitioned data means idle machines (waste) or overloaded machines (slow). This homework teaches you to diagnose both.
>
> **The MapReduce Revolution:** Google's MapReduce paper (2004) enabled them to index the entire web. The same pattern now powers fraud detection at Visa (processing 65,000 transactions/second), genomics research, and climate modeling.

Grading

Component	Points	Effort	What We're Looking For
Spark Setup	3	*	Create a Spark session and explain configuration
Data Partitioning	5	**	Demonstrate understanding of data distribution
Performance Experiment	5	**	Analyze why speedup isn't linear
K-Means Clustering	5	**	Apply clustering and interpret results
Diagram	2	*	Clear diagram explaining distributed processing
Total	**20**		

The Big Picture: MapReduce & Spark

Figure 1: MapReduce paradigm and Spark's in-memory processing model



The Problem: A patron needs 100 books. One librarian would take forever.

● **MAP:** Split the list among 10 student workers. Each finds their 10 books in parallel.

● **SHUFFLE:** Students bring books back. Group them by floor/section for easy shelving later.

● **REDUCE:** Combine into one organized cart, sorted so patron can find any book quickly.

Real Examples:

Word Count

```
Input:  "the cat sat on the mat"
Output: {the: 2, cat: 1, sat: 1, on: 1, mat: 1}
```

Total Sales by Region

```
Input:  (West,$50), (East,$30), (West,$20), (East,$40)
Output: {West: $70, East: $70}
```

```
# Same word count in Spark - one line!
df.groupBy( "word" ).count().show()
```

What You'll Practice

● Think in Map-Shuffle-Reduce steps
● Write PySpark transformations
● Aggregate data at scale

The same pattern powers Google, Netflix, Uber, and Spotify's data pipelines

Component	Points	Effort	What We're Looking For

Effort Key: * Straightforward | ** Requires thinking | *** Challenge

---

The Big Picture

MapReduce breaks big problems into small pieces that can be solved in parallel:

1. **Map:** Transform each piece of data independently
2. **Shuffle:** Group related data together
3. **Reduce:** Combine results into final answer

Example: Counting words in 1 billion documents - Map: Each machine counts words in its assigned documents - Shuffle: Group all counts for "apple" together, all counts for "banana" together - Reduce: Sum the counts for each word

---

Instructions

1. Open `MIS769_HW2_MapReduce_Spark.ipynb` in Google Colab
2. Complete Part 1: Install Spark and create a session
3. Complete Part 2: Explore data partitioning
4. Complete Part 3: Run performance experiment with 1, 2, and 4 cores
5. Complete Part 4: Apply K-Means to Netflix data and name your clusters
6. Complete Part 5: Draw your own diagram of how Spark processes data

---

What Your Output Should Look Like

Spark Session:

```
□  SPARK SESSION CREATED
============================================================
App Name: MIS769_HW2
Master: local[2]
Spark Version: 3.5.0
Python Version: 3.10.12
```

Partitioning Analysis:

```
□  PARTITION ANALYSIS
------------------------------------------------------------
Total Records: 100,000
Number of Partitions: 4
Records per Partition:
  Partition 0: 25,234 (25.2%)
  Partition 1: 24,892 (24.9%)
  Partition 2: 25,012 (25.0%)
  Partition 3: 24,862 (24.9%)
```

```
✓ Data is well-balanced across partitions
```

Performance Results:

```
□  PERFORMANCE EXPERIMENT
============================================================
Configuration     Time (sec)     Speedup
------------------------------------------------------------
local[1]          12.4           1.0x (baseline)
local[2]           7.2           1.7x
local[4]           5.1           2.4x

Note: Speedup is sub-linear due to:
- Overhead of coordination between workers
- Data shuffling between partitions
- Colab's CPU limitations
```

Clustering Results:

```
□  K-MEANS CLUSTERING RESULTS
============================================================
Cluster 0 (n=2,341): "Weekend Binge Watchers"
  - Avg movies/month: 28.3
  - Peak viewing: Saturday 8pm
  - Top genre: Drama

Cluster 1 (n=1,892): "Casual Viewers"
  - Avg movies/month: 4.2
  - Peak viewing: Sunday 7pm
  - Top genre: Comedy
```

Common Mistakes (and How to Avoid Them)

Mistake	Symptom	Fix
Spark not installed	`ModuleNotFoundError: pyspark`	Run `!pip install pyspark` first
Java not found	`JAVA_HOME is not set`	Run `!apt-get install openjdk-11-jdk-headless`
Too many partitions	Slow performance	Repartition: `df.coalesce(4)`
Too few partitions	Not utilizing cores	Repartition: `df.repartition(8)`
Collecting too much data	Memory crash	Use `.limit(1000).collect()` or `.take(100)`

Forgetting Spark is lazy	Nothing happens	Add an action: `.count()`, `.show()`, `.collect()`

If you see this error:

```
Py4JJavaError: An error occurred while calling o42.count
```

Check: Usually a data type mismatch or null values. Use `.printSchema()` to debug.

If Spark seems slow: - Check partitions: `df.rdd.getNumPartitions()` - Avoid `collect()` on large datasets - Use `cache()` for repeated operations

---

Questions to Answer

- **Q1:** What does `local[2]` mean? What would `local[4]` do differently?
- **Q2:** Why doesn't 4 cores give exactly 4x speedup?
- **Q3:** What characterizes each cluster? Give them descriptive business names.
- **Q4:** How does the Map-Reduce pattern apply to your clustering task?

Going Deeper (Optional Challenges)

Challenge A: Partition Optimization

Experiment with different numbers of partitions (2, 4, 8, 16, 32). Find the optimal number for your dataset size and document the performance curve. At what point do more partitions hurt performance?

Challenge B: Custom MapReduce

Implement a word count from scratch using PySpark's RDD API (not DataFrames). Use `flatMap`, `map`, and `reduceByKey`. Compare performance to the DataFrame approach.

Challenge C: Real-Time Simulation

Use Spark Structured Streaming to process data from a simulated stream. Create a simple producer that generates fake transactions and a consumer that computes running averages.

Quick Reference

```
# Install Spark in Colab
!pip install pyspark
!apt-get install openjdk-11-jdk-headless -qq

# Create Spark Session
from pyspark.sql import SparkSession

spark = SparkSession.builder \
    .appName("MIS769_HW2") \
    .master("local[2]") \
    .getOrCreate()

# Load data
df = spark.read.csv("data.csv", header=True, inferSchema=True)

# Basic operations
df.show(5)                    # Display first 5 rows
df.printSchema()              # Show column types
df.count()                    # Count rows (triggers computation)

# Partitioning
df.rdd.getNumPartitions()     # Check current partitions
df = df.repartition(4)        # Increase partitions
df = df.coalesce(2)           # Decrease partitions (no shuffle)

# MapReduce pattern with DataFrames
from pyspark.sql.functions import col, avg, count

# Map (transform)
df_mapped = df.select(col("category"), col("value") * 2)

# Reduce (aggregate)
df_reduced = df.groupBy("category").agg(
    count("*").alias("count"),
    avg("value").alias("avg_value")
)

# K-Means Clustering
from pyspark.ml.clustering import KMeans
from pyspark.ml.feature import VectorAssembler

# Prepare features
assembler = VectorAssembler(inputCols=["col1", "col2"], outputCol="features")
df_features = assembler.transform(df)

# Fit model
kmeans = KMeans(k=3, seed=42)
model = kmeans.fit(df_features)
predictions = model.transform(df_features)

# Performance timing
import time
start = time.time()
df.count()  # Force computation
elapsed = time.time() - start
print(f"Time: {elapsed:.2f} seconds")
```

Key Spark Concepts: | Concept | Description | Example | |———|————-|———| | Transformation | Lazy operation, builds plan | `filter()`, `select()`, `groupBy()` | | Action | Triggers computation | `count()`, `show()`, `collect()` | | Partition | Chunk of data on one node | `repartition(4)` | | Shuffle | Data movement between nodes | Happens during `groupBy()` | | Cache | Store in memory for reuse | `df.cache()` |

Submission

Upload to Canvas: - Your completed `.ipynb` notebook with all cells executed

Richard Young

Richard Young, Ph.D.