SE446: Big Data Engineering

Week 3B: Implementing MapReduce in Python

Prof. Anis Koubaa

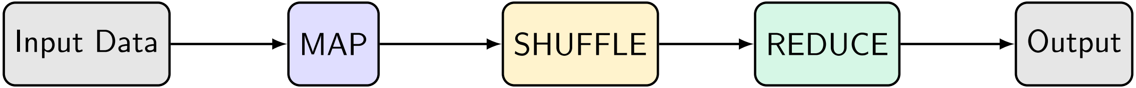
SE 446
Alfaisal University

`https://github.com/aniskoubaa/big_data_course`

Spring 2026

Today's Agenda

Quick Recap: The MapReduce Pipeline



Map Phase

$(k_1, v_1) \rightarrow [(k_2, v_2)]$

➔ Process each record independently and emit key-value pairs.

Reduce Phase

$(k_2, [v_2]) \rightarrow [(k_3, v_3)]$

➔ Aggregate all values associated with the same key.

Two Ways to Practice MapReduce

Local Emulator (Today)

- Pure Python script
- Runs entirely on your laptop
- Uses loops instead of distributed nodes
- **Purpose**: Learn the algorithm logic

✔ Fast iteration, easy debugging

Real Hadoop Cluster (Lab 01)

- Hadoop Streaming utility
- Runs on multi-node HDFS cluster
- Parallel processing across machines
- **Purpose**: Learn production deployment

✔ Real-world scalability

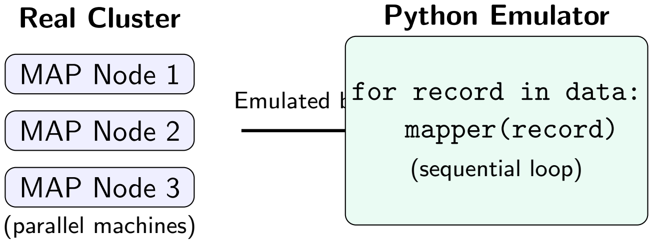
Key Insight

The **mapper** and **reducer** logic you write is **identical** in both approaches. Only the execution environment differs!

Why Use an Emulator?

Problem: Setting up a Hadoop cluster takes time and infrastructure.

Solution: Emulate the MapReduce behavior locally.



Bottom line: Once your algorithm works locally, it will work on a 1000-node cluster!

How the Emulator Works

The Python emulator simulates all three MapReduce phases:

Phase	Real Cluster	Python Emulator
Map	Parallel nodes process chunks	`for` loop iterates records
Shuffle	Network transfers data by key	`defaultdict` groups by key
Reduce	Parallel reducers aggregate	`for` loop iterates keys

The Trade-Off

- **Lost**: True parallelism, fault tolerance, distributed storage
- **Gained**: Simplicity, debuggability, instant feedback

Building the Emulator: Overview

We will build a simple Python function called `map_reduce()` that:

1. **Takes inputs**:
   - `data`: A list of records (e.g., lines, dictionaries)
   - `mapper`: A function you write
   - `reducer`: A function you write
2. **Executes** the Map → Shuffle → Reduce pipeline
3. **Returns** the final aggregated results

Your job as a developer: Write the `mapper` and `reducer` functions.
The framework's job: Handle the shuffle and coordination.

The Emulator Code

```
from collections import defaultdict

def map_reduce(data, mapper, reducer):
    """Python MapReduce Emulator - Simulates distributed processing locally."""

    # ===== MAP PHASE (simulates parallel map nodes) =====
    mapped = []
    for record in data:
        result = mapper(record)
        if result: mapped.append(result)

    # ===== SHUFFLE PHASE (simulates network grouping) =====
    shuffled = defaultdict(list)
    for key, value in mapped:
        shuffled[key].append(value)

    # ===== REDUCE PHASE (simulates parallel reducers) =====
    results = []
    for key, values in shuffled.items():
        results.append(reducer(key, values))

    return results
```

Word Count: The "Hello World" of Big Data

Goal: Count how many times each word appears in a dataset.

Think in Key-Value pairs:

- **Key**: The word itself
- **Value**: The number 1 (one occurrence)

Mapper logic:
For each word, emit `(word, 1)`

Reducer logic:
Sum all the 1s for each word

Example trace:

1. Input: `"hello world hello"`
2. Map: `[("hello",1), ("world",1), ("hello",1)]`
3. Shuffle: `{"hello":[1,1], "world":[1]}`
4. Reduce: `[("hello",2), ("world",1)]`

Word Count: The Code

```python
# 1. Prepare Data
documents = [
    "Hello World Hello",
    "World of Big Data",
    "Hello Big Data World"
]

# 2. Input Splitting
words = [word for doc in documents
          for word in doc.split()]

# 3. Mapper: Emit (word, 1)
def word_mapper(word):
    return (word.lower(), 1)
```

```python
# 4. Reducer: Sum values
def count_reducer(word, counts):
    return (word, sum(counts))

# 5. Run MapReduce
results = map_reduce(
    words,
    word_mapper,
    count_reducer
)

print(results)
# [('hello', 3),
#  ('world', 3), ...]
```

Loading Chicago Crime Data

Dataset: Chicago Crime Records

Key Columns:

- `Primary Type`: THEFT, ASSAULT, BATTERY, etc.
- `District`: Police district number (1-25)
- `Arrest`: True/False - was someone arrested?
- `Location Description`: STREET, RESIDENCE, etc.

Note: We generate 1,000 synthetic records in the notebook to ensure consistency across all students.

Sample Record (as dict):

```
{
  'ID': 1,
  'Primary Type': 'THEFT',
  'District': 3,
  'Arrest': False,
  'Location Description':
      'STREET'
}
```

Exercise 1: Count Crimes by Type

Goal: How many crimes of each type occurred?

Design Decision:

- **Key**: Crime type (e.g., "THEFT")
- **Value**: 1 (for counting)

Why this works:
Shuffle groups all records with the same crime type. Reducer sums the 1s to get the total count.

Expected Output:

```
THEFT             300
BATTERY           250
ASSAULT           150
CRIMINAL DAMAGE   100
BURGLARY          100
OTHER             100
```

Exercise 1: The Code

```python
# 1. Prepare Data
crime_records = crimes_df.to_dict('records')

# 2. Mapper: Emit (type, 1)
def crime_type_mapper(record):
    return (record['Primary Type'], 1)

# 3. Reducer: Sum counts (from previous example)
def count_reducer(crime_type, counts):
    return (crime_type, sum(counts))

# 4. Run MapReduce
crime_counts = map_reduce(crime_records, crime_type_mapper, count_reducer)

print(crime_counts[:3])
# [('THEFT', 300), ('BATTERY', 250), ...]
```

Exercise 2: Count Crimes per District

Goal: Which police districts have the most crime?

Design Decision:

- **Key**: District number
- **Value**: 1 (for counting)

Observation:
The only change from Exercise 1 is what we use as the key! The reducer stays the same.

This is the power of MapReduce: Change the key, get different insights.

Code Change:

```
# OLD (count by type)
crime_type = record['Primary Type']
return (crime_type, 1)

# NEW (count by district)
district = record['District']
return (district, 1)
```

Exercise 2: The Code

```python
# Mapper: extract district as key
def district_mapper(record):
    district = record['District']
    return (district, 1)

# Same reducer (count pattern)
results = map_reduce(crime_records, district_mapper, count_reducer)

# Sort by count (descending)
sorted_districts = sorted(results, key=lambda x: x[1], reverse=True)

print("Top 5 Districts by Crime Count:")
for district, count in sorted_districts[:5]:
    print(f"  District {district}: {count} crimes")
```

Exercise 3: Filter - Crimes with Arrests

Goal: Count only crimes where an arrest was made.

The Filter Pattern:

- Mapper returns `None` for unwanted records
- Framework skips `None` values
- Only matching records reach the reducer

When to use:
Any time you want to analyze a **subset** of your data.

Logic Flow:

```
IF record['Arrest'] == True:
    EMIT (crime_type, 1)
ELSE:
    EMIT None  # Skip this record
```

Key Pattern

Return `None` from mapper to filter records. The framework handles the rest.

Exercise 3: The Code

```
def arrest_mapper(record):
    """
    Only emit crimes where arrest was made
    """
    if record['Arrest'] == True:
        return (record['Primary Type'], 1)
    return None  # Filter out - no arrest

# Run MapReduce
arrest_counts = map_reduce(crime_records, arrest_mapper,
    count_reducer)

print("Crimes with Arrests:")
for crime_type, count in sorted(arrest_counts, key=lambda x: x[1],
    reverse=True)[:5]:
    print(f"  {crime_type}: {count} arrests")
```

Exercise 4: Aggregation - Arrest Rate by Type

Goal: What percentage of each crime type results in an arrest?

Challenge:
We need two pieces of information:

1. Total number of crimes
2. Number of arrests

Solution: Emit a **tuple** as the value!

```
(crime_type, (arrested, 1))
```

- `arrested`: 1 if arrest, else 0
- 1: always 1 (for total count)

Reducer Logic:

```
total_arrests = sum(v[0] for v in values)
total_crimes = sum(v[1] for v in values)
rate = total_arrests / total_crimes * 100
```

Example:
THEFT receives values:
`[(1,1), (0,1), (0,1), (1,1)]`
$\rightarrow$ 2 arrests / 4 total = 50%

Exercise 4: The Code

```
def arrest_stats_mapper(record):
    """Emit (crime_type, (arrested, total))"""
    crime_type = record['Primary Type']
    arrested = 1 if record['Arrest'] else 0
    return (crime_type, (arrested, 1))

def arrest_rate_reducer(crime_type, values):
    """Calculate arrest rate"""
    total_arrests = sum(v[0] for v in values)
    total_crimes = sum(v[1] for v in values)
    rate = (total_arrests / total_crimes) * 100
    return (crime_type, round(rate, 1))

results = map_reduce(crime_records, arrest_stats_mapper, arrest_rate_reducer)

print("Arrest Rates by Crime Type:")
for crime_type, rate in sorted(results, key=lambda x: x[1], reverse=True)[:5]:
    print(f"  {crime_type}: {rate}%")
```

Chaining MapReduce Jobs

Problem: Find the top 5 crime types.

Why is this hard?

- MapReduce processes each key **independently**
- Finding "top 5" requires comparing **all** keys together

Solution: Two-stage MapReduce

1. **Job 1: Count crimes by type**
   - Map: (record) → (type, 1)
   - Reduce: (type, [1,1,1...]) → (type, count)
2. **Job 2: Find top 5**
   - Map: (type, count) → ("all", (type, count)) // *same key!*
   - Reduce: ("all", [(type, count)...]) → sorted top 5

Trick

Using a **constant key** (like "all") sends all data to a single reducer!

Multi-Stage Example

```python
# Stage 1: Count by type
crime_counts = map_reduce(crime_records, crime_type_mapper, count_reducer)

# Stage 2: Find top 5
def top_mapper(item):
    """Send all to same reducer with dummy key"""
    crime_type, count = item
    return ("all", (crime_type, count))  # Key="all" sends all to one reducer

def top_reducer(key, values):
    """Sort and take top 5"""
    sorted_values = sorted(values, key=lambda x: x[1], reverse=True)
    return (key, sorted_values[:5])

top_5 = map_reduce(crime_counts, top_mapper, top_reducer)

print("Top 5 Crime Types:")
for crime_type, count in top_5[0][1]:
    print(f"  {crime_type}: {count}")
```

MapReduce Design Patterns

Pattern	Map Output	Reduce Operation
Counting	(key, 1)	sum(values)
Sum	(key, value)	sum(values)
Average	(key, (value, 1))	sum(v)/sum(c)
Max/Min	(key, value)	max/min(values)
Filter	(key, value) or None	pass through
Distinct	(value, None)	emit key only
Top N	(constant, (key, value))	sort and slice

Golden Rule: Always ask yourself:

1. **What is my key?** $\rightarrow$ What do I want to group by?
2. **What is my value?** $\rightarrow$ What do I want to aggregate?

Common Mistakes

1. **Forgetting to handle None**
   - Mapper can return None to filter
   - Framework must check for None
2. **Wrong key choice**
   - Key determines grouping
   - Choose key based on what you want to aggregate
3. **Non-hashable keys**
   - Keys must be hashable (strings, numbers, tuples)
   - Lists, dicts cannot be keys
4. **Type mismatches**
   - Ensure values are consistent types
   - `sum([1, 1, '1'])` will fail!

Debugging Tips

```python
# Add debugging to mapper
def debug_mapper(record):
    result = crime_type_mapper(record)
    print(f"Mapper: {record['ID']} -> {result}")
    return result

# Test with small sample first
sample = crime_records[:5]
test_results = map_reduce(sample, debug_mapper, count_reducer)

# Verify shuffle output
def debug_map_reduce(data, mapper, reducer):
    # ... map phase ...
    print(f"After map: {len(mapped)} pairs")
    # ... shuffle phase ...
    print(f"After shuffle: {len(shuffled)} keys")
    for key in list(shuffled.keys())[:3]:
        print(f"  {key}: {len(shuffled[key])} values")
```

From Emulator to Real Cluster (Lab 01)

What changes when moving to Hadoop Streaming?

Aspect	**Python Emulator**	**Hadoop Streaming**
Input	Python list	HDFS files
Mapper I/O	Function call	stdin $\rightarrow$ stdout
Shuffle	`defaultdict`	Hadoop framework
Reducer I/O	Function call	stdin $\rightarrow$ stdout
Output	Python list	HDFS files

Key insight: The **logic** stays the same! Only the **I/O mechanism** changes.

In Lab 01, you will:

- Write `mapper.py` that reads from `sys.stdin`
- Write `reducer.py` that reads sorted input from `sys.stdin`
- Submit to the real Hadoop cluster using `mapred streaming`

Today's Lab Tasks

Lab 01: MapReduce with Hadoop Streaming:

1. Implement word count mapper and reducer
2. Test locally using Linux pipes: `cat test.txt | python3 mapper.py | sort | python3 reducer.py`
3. Run on real Hadoop cluster

Notebook Practice:

1. Count crimes by type
2. Find district with most crimes
3. Calculate arrest rate by crime type
4. (Bonus) Find top 5 crime locations

Deliverables:

- Complete lab notebook
- Commit to team GitHub repo

Summary

What You Learned Today

- **Emulator concept**: Practice MapReduce locally before cluster deployment
- **Framework**: Built a reusable `map_reduce()` function
- **Patterns**: Count, filter, average, multi-stage
- **Debugging**: Test with small samples, add print statements

Key Takeaway:

Think: **What is my key? What is my value?**

Next Week: Milestone 2 work session + MapReduce on Hadoop cluster