Lab 1: HDFS Operations

To use labs listed in this document, you can use sandbox enviroment provided in the AWS Data Engineering Course in AWS Academy. To initiate the snadbox, we only need to set the roles for the EMR cluster and the core nodes. We can even use m4.xlarge machines for running the sandbox.

To run the lbas, we can use command line shell environment. We can connect to the cluster through command shell by connecting to the Primary EMR EC2 machine from the EC2 instances console. Before that, we need to make sure to add an inbound firewall rule to the Primary node in the firewall and security settings.

Learning Objectives

- Understand HDFS architecture
- Perform basic file operations
- Explore HDFS commands and directory structure

Dataset Preparation

Create sample datasets for practice:

```
# Create local sample files
echo -e "id,name,age,city\n1,John,25,NYC\n2,Jane,30,LA\n3,Bob,35,Chicago" >
employees.csv
echo -e
"product,price,category\niPhone,999,Electronics\nLaptop,1200,Electronics\nShirt,25
,Clothing" > products.csv

# Create a larger text file for testing
for i in {1..1000}; do echo "This is line $i with some sample data for testing
HDFS operations"; done > large_sample.txt
```

Basic HDFS Commands

1. Check HDFS Status

```
# Check HDFS health
hdfs dfsadmin -report

# List HDFS directories
hdfs dfs -ls /

# Check available space
hdfs dfs -df -h /
```

2. Create Directories

```
# Create user directory
hdfs dfs -mkdir -p /user/hadoop/datasets
hdfs dfs -mkdir -p /user/hadoop/output
hdfs dfs -mkdir -p /user/hadoop/temp

# Create project-specific directories
hdfs dfs -mkdir -p /user/hadoop/datasets/employees
hdfs dfs -mkdir -p /user/hadoop/datasets/products
hdfs dfs -mkdir -p /user/hadoop/datasets/logs
```

3. File Upload and Basic Operations

```
# Upload files to HDFS
hdfs dfs -put employees.csv /user/hadoop/datasets/employees/
hdfs dfs -put products.csv /user/hadoop/datasets/products/
hdfs dfs -put large_sample.txt /user/hadoop/datasets/

# List files
hdfs dfs -ls /user/hadoop/datasets/
hdfs dfs -ls -h /user/hadoop/datasets/  # with human-readable sizes

# View file contents
hdfs dfs -cat /user/hadoop/datasets/employees/employees.csv
hdfs dfs -head /user/hadoop/datasets/large_sample.txt  # first 1KB
hdfs dfs -tail /user/hadoop/datasets/large_sample.txt  # last 1KB
```

4. File Operations

```
# Copy files within HDFS
hdfs dfs -cp /user/hadoop/datasets/employees/employees.csv
/user/hadoop/datasets/employees/employees_backup.csv

# Move/rename files
hdfs dfs -mv /user/hadoop/datasets/employees/employees_backup.csv
/user/hadoop/datasets/employees_archive.csv

# Copy from HDFS to local
hdfs dfs -get /user/hadoop/datasets/employees/employees.csv
./downloaded_employees.csv

# Check file permissions
hdfs dfs -ls -la /user/hadoop/datasets/

# Change permissions
hdfs dfs -chmod 755 /user/hadoop/datasets/employees.csv
hdfs dfs -chown hadoop:hadoop /user/hadoop/datasets/products.csv
```

5. Advanced HDFS Operations

```bash
# Check file block information
hdfs fsck /user/hadoop/datasets/large_sample.txt -files -blocks -locations

# Merge files
hdfs dfs -getmerge /user/hadoop/datasets/employees/ ./merged_employees.txt

# Set replication factor
hdfs dfs -setrep 2 /user/hadoop/datasets/large_sample.txt

# Remove files
hdfs dfs -rm /user/hadoop/datasets/employees_archive.csv
hdfs dfs -rm -r /user/hadoop/temp/  # recursive delete
```

Exercise Questions

1. What's the default replication factor for your HDFS setup?
2. How many blocks does your large_sample.txt file use?
3. Try uploading a file larger than 128MB and observe the block distribution.

Lab 2: MapReduce Programming

Learning Objectives

- Write and execute MapReduce jobs
- Understand Map and Reduce phases
- Process real data using MapReduce

Sample Dataset Creation

```bash
# Create a word count dataset
cat << 'EOF' > sample_text.txt
Apache Hadoop is an open source framework
Hadoop provides distributed storage and processing
MapReduce is a programming model for Hadoop
HDFS is the storage layer of Hadoop
Hadoop ecosystem includes many tools
Big data processing with Hadoop is scalable
Hadoop clusters can process petabytes of data
EOF

# Upload to HDFS
hdfs dfs -put sample_text.txt /user/hadoop/datasets/
```

Example 1: Word Count (Python MapReduce using MRJob)

Python Implementation using MRJob

MRJob is a Python library that makes writing MapReduce jobs simple and intuitive. Instead of dealing with complex Java code, we can write clean Python that's easy to understand and test.

Create `wordcount_mrjob.py`:

```python
#!/usr/bin/env python3
"""
Word Count MapReduce using MRJob
Simple Python implementation of the classic word count problem
"""

import re
from mrjob.job import MRJob

class MRWordCount(MRJob):

    def mapper(self, _, line):
        # Extract words from each line
        words = re.findall(r'[a-z]{2,}', line.lower())
        for word in words:
            yield word, 1

    def reducer(self, word, counts):
        # Sum up the counts for each word
        yield word, sum(counts)

if __name__ == '__main__':
    MRWordCount.run()
```

Installation and Running

```bash
# Install MRJob
pip3 install mrjob

# Test locally first
python3 wordcount_mrjob.py sample_text.txt

# For Hadoop, use the simple approach - let MRJob auto-detect
python3 wordcount_mrjob.py -r hadoop hdfs:///user/hadoop/datasets/sample_text.txt
> wordcount_results.txt

# Or even simpler - run locally on HDFS data
hdfs dfs -get /user/hadoop/datasets/sample_text.txt ./
python3 wordcount_mrjob.py sample_text.txt

# View results
cat wordcount_results.txt
```

Example 2: Sales Analysis MapReduce

Complete Sales Analysis Implementation

Now let's build a more complex example that analyzes sales data to calculate revenue by category. This example demonstrates real-world MapReduce usage with structured data.

Business Problem: We have sales transaction data and need to:

1. Calculate total revenue per product category
2. Count number of transactions per category
3. Find average transaction value per category

Create `sales_analysis_complete.py`:

```
#!/usr/bin/env python3
"""
Sales Analysis MapReduce Implementation
======================================

This script analyzes sales transaction data to generate business insights.
It calculates revenue, transaction counts, and averages by product category.

Input Data Format (CSV):
date,category,product,price,quantity
2023-01-01,Electronics,iPhone,999.99,2
2023-01-01,Electronics,Laptop,1299.99,1
2023-01-01,Clothing,Shirt,29.99,3

MapReduce Logic:
1. MAP: Extract (category, revenue) pairs from each transaction
2. REDUCE: Aggregate all metrics for each category

Output Format:
category  total_revenue  transaction_count  avg_revenue
Electronics  5999.96  3  1999.99
"""

import sys
import json
from decimal import Decimal

def mapper():
    """
    SALES MAPPER FUNCTION
    ====================
    Processes each sales transaction line and emits category-based metrics.

    Processing Steps:
    1. Parse CSV line to extract fields
    2. Calculate transaction revenue (price * quantity)
    3. Emit category with revenue data
    4. Handle data validation and error cases

    Input: "2023-01-01,Electronics,iPhone,999.99,2"
    Output: "Electronics  {'revenue': 1999.98, 'count': 1}"
    """
    print("=== SALES MAPPER STARTED ===", file=sys.stderr)

    processed_lines = 0
    valid_transactions = 0
    total_revenue = Decimal('0')

    for line_num, line in enumerate(sys.stdin, 1):
        line = line.strip()

        # Skip empty lines and header
        if not line or line.startswith('date,') or line.startswith('#'):
            continue

        try:
            # Parse CSV fields (assuming comma-separated)
            fields = line.split(',')

            if len(fields) >= 5:  # Ensure we have all required fields
                date = fields[0].strip()
                category = fields[1].strip()
                product = fields[2].strip()
                price = Decimal(fields[3].strip())
                quantity = int(fields[4].strip())

                # Calculate transaction revenue
                transaction_revenue = price * quantity

                # Validate data
                if category and price >= 0 and quantity > 0:
                    # Create metrics object for this transaction
                    metrics = {
                        'revenue': float(transaction_revenue),
                        'count': 1,
                        'product': product,
                        'date': date
                    }

                    # Emit category with metrics (JSON format for complex data)
                    print(f"{category}\t{json.dumps(metrics)}")

                    valid_transactions += 1
                    total_revenue += transaction_revenue

                    # Debug info for large datasets
                    if valid_transactions % 1000 == 0:
                        print(f"Processed {valid_transactions} valid
transactions", file=sys.stderr)
                else:
                    print(f"Invalid data in line {line_num}: {line}",
file=sys.stderr)
                else:
                    print(f"Insufficient fields in line {line_num}: {line}",
file=sys.stderr)

        except (ValueError, IndexError) as e:
            print(f"Error parsing line {line_num}: {line} - {e}", file=sys.stderr)
            continue

        processed_lines += 1

    print(f"=== MAPPER SUMMARY ===", file=sys.stderr)
    print(f"Lines processed: {processed_lines}", file=sys.stderr)
    print(f"Valid transactions: {valid_transactions}", file=sys.stderr)
    print(f"Total revenue processed: ${total_revenue}", file=sys.stderr)

def reducer():
    """
    SALES REDUCER FUNCTION
    =====================
    Aggregates sales metrics for each product category.

    Aggregation Logic:
    1. Group all transactions by category
    2. Sum total revenue for each category
    3. Count total transactions per category
    4. Calculate average revenue per transaction
    5. Track additional metrics (top products, date ranges)

    Input (grouped by category):
    Electronics  {"revenue": 1999.98, "count": 1, "product": "iPhone"}
    Electronics  {"revenue": 1299.99, "count": 1, "product": "Laptop"}
    Clothing     {"revenue": 89.97, "count": 1, "product": "Shirt"}

    Output:
    Electronics  3299.97  2  1649.985  iPhone,Laptop
    Clothing     89.97    1  89.97     Shirt
    """
    print("=== SALES REDUCER STARTED ===", file=sys.stderr)

    current_category = None
    category_revenue = Decimal('0')
    category_count = 0
    category_products = set()
    categories_processed = 0

    for line in sys.stdin:
        line = line.strip()

        if line:
            try:
                # Parse mapper output
                category, metrics_json = line.split('\t', 1)
                metrics = json.loads(metrics_json)

                # Extract transaction data
                transaction_revenue = Decimal(str(metrics['revenue']))
                transaction_count = metrics['count']
                product = metrics.get('product', 'Unknown')

                if current_category == category:
                    # Same category - accumulate metrics
                    category_revenue += transaction_revenue
                    category_count += transaction_count
                    category_products.add(product)
                else:
                    # New category - output previous category's results
                    if current_category is not None:
                        avg_revenue = category_revenue / category_count if category_count > 0 else 0
                        products_list = ','.join(sorted(category_products)[:5])  # Top 5 products

                        # Output: category, total_revenue, count, avg_revenue, top_products
                        print(f"{current_category}\t{category_revenue}\t{category_count}\t{avg_revenue:.2f}\t{products_list}")
                        categories_processed += 1

                    # Start new category
                    current_category = category
                    category_revenue = transaction_revenue
                    category_count = transaction_count
                    category_products = {product}

            except (ValueError, json.JSONDecodeError) as e:
                print(f"Error parsing reducer input: {line} - {e}", file=sys.stderr)
                continue

    # Output the last category
    if current_category is not None:
        avg_revenue = category_revenue / category_count if category_count > 0 else 0
        products_list = ','.join(sorted(category_products)[:5])
        print(f"{current_category}\t{category_revenue}\t{category_count}\t{avg_revenue:.2f}\t{products_list}")
        categories_processed += 1

    print(f"=== REDUCER SUMMARY ===", file=sys.stderr)
    print(f"Categories processed: {categories_processed}", file=sys.stderr)

def local_test():
    """
    LOCAL TESTING FOR SALES ANALYSIS
    ================================
    Test the sales analysis with sample data to verify correctness.
    """
    print("=== TESTING SALES ANALYSIS ===")

    # Sample sales data
    test_sales_data = [
        "2023-01-01,Electronics,iPhone,999.99,2",      # $1999.98
        "2023-01-01,Electronics,Laptop,1299.99,1",     # $1299.99
        "2023-01-01,Clothing,Shirt,29.99,3",           # $89.97
        "2023-01-02,Electronics,Mouse,25.99,5",        # $129.95
        "2023-01-02,Books,Python Book,49.99,2",        # $99.98
        "2023-01-02,Clothing,Jeans,79.99,2",           # $159.98
        "2023-01-03,Electronics,Keyboard,89.99,3",     # $269.97
        "2023-01-03,Books,Java Book,59.99,1"           # $59.99
    ]

    print("\nSample sales data:")
    for i, line in enumerate(test_sales_data, 1):
        print(f"  {i}. {line}")

    # Simulate mapper
    print("\n=== MAPPER SIMULATION ===")
    mapped_results = []

    for line in test_sales_data:
        fields = line.split(',')
        category = fields[1].strip()
        price = float(fields[3].strip())
        quantity = int(fields[4].strip())
        revenue = price * quantity

        metrics = {'revenue': revenue, 'count': 1}
        mapped_results.append((category, metrics))
        print(f"  {category} -> Revenue: ${revenue:.2f}")

    # Sort by category (Hadoop does this automatically)
    mapped_results.sort(key=lambda x: x[0])

    # Simulate reducer
    print("\n=== REDUCER SIMULATION ===")
    from collections import defaultdict

    category_stats = defaultdict(lambda: {'revenue': 0, 'count': 0})

    for category, metrics in mapped_results:
        category_stats[category]['revenue'] += metrics['revenue']
        category_stats[category]['count'] += metrics['count']

    print("\nFinal Results:")
    print("Category\t\tTotal Revenue\tTransactions\tAvg Revenue")
    print("-" * 60)

    for category in sorted(category_stats.keys()):
        stats = category_stats[category]
        avg_revenue = stats['revenue'] / stats['count']
        print(f"
{category:15s}\t${stats['revenue']:8.2f}\t{stats['count']:8d}\t${avg_revenue:8.2f}
")

    print("=== TESTING COMPLETED ===")

def main():
    """Main entry point for sales analysis"""
    if len(sys.argv) != 2:
        print("Sales Analysis MapReduce")
        print("Usage: python3 sales_analysis_complete.py [mapper|reducer|test]")
        print("\nModes:")
        print("  mapper  - Process sales transactions and emit category metrics")
        print("  reducer - Aggregate metrics by category")
        print("  test    - Run local test with sample data")
        sys.exit(1)

    mode = sys.argv[1].lower()

    if mode == "mapper":
        mapper()
    elif mode == "reducer":
        reducer()
    elif mode == "test":
        local_test()
    else:
        print(f"Error: Unknown mode '{mode}'")
        sys.exit(1)

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

Running the Sales Analysis

Step 1: Test Locally

```
# Test the sales analysis logic
python3 sales_analysis_complete.py test
```

Step 2: Create Sample Sales Data

```
# Create comprehensive sales dataset
cat << 'EOF' > sales_data_extended.csv
date,category,product,price,quantity
2023-01-01,Electronics,iPhone,999.99,2
2023-01-01,Electronics,Laptop,1299.99,1
2023-01-01,Clothing,Shirt,29.99,3
```

```
2023-01-02,Electronics,Mouse,25.99,5
2023-01-02,Books,Python Book,49.99,2
2023-01-02,Clothing,Jeans,79.99,2
2023-01-03,Electronics,Keyboard,89.99,3
2023-01-03,Books,Java Book,59.99,1
2023-01-04,Electronics,Monitor,299.99,1
2023-01-04,Clothing,Sweater,59.99,2
2023-01-05,Sports,Basketball,29.99,1
2023-01-05,Sports,Tennis Racket,89.99,1
2023-01-06,Home,Coffee Maker,79.99,1
2023-01-06,Home,Blender,49.99,2
2023-01-07,Electronics,Headphones,199.99,1
2023-01-07,Books,Data Science,69.99,1
EOF

# Upload to HDFS
hdfs dfs -put sales_data_extended.csv /user/hadoop/datasets/
hdfs dfs -put sales_analysis_complete.py /user/hadoop/scripts/
```

Step 3: Run MapReduce Job

```
# Remove previous output
hdfs dfs -rm -r /user/hadoop/output/sales_analysis_python

# Run the sales analysis job
hadoop jar $HADOOP_HOME/share/hadoop/tools/lib/hadoop-streaming-*.jar \
  -files sales_analysis_complete.py \
  -mapper "python3 sales_analysis_complete.py mapper" \
  -reducer "python3 sales_analysis_complete.py reducer" \
  -input /user/hadoop/datasets/sales_data_extended.csv \
  -output /user/hadoop/output/sales_analysis_python

# View results with headers
echo "=== SALES ANALYSIS RESULTS ==="
echo "Category        Total_Revenue  Transactions  Avg_Revenue  Top_Products"
echo "==================================================================="
hdfs dfs -cat /user/hadoop/output/sales_analysis_python/part-00000

# Generate summary report
echo -e "\n=== BUSINESS INSIGHTS ==="
echo "Top revenue categories:"
hdfs dfs -cat /user/hadoop/output/sales_analysis_python/part-00000 | sort -k2 -nr
| head -3

echo -e "\nMost active categories (by transaction count):"
hdfs dfs -cat /user/hadoop/output/sales_analysis_python/part-00000 | sort -k3 -nr
| head -3

echo -e "\nHighest average transaction value:"
hdfs dfs -cat /user/hadoop/output/sales_analysis_python/part-00000 | sort -k4 -nr
| head -3
```

Expected Output:

```
Books         159.97    3     53.32    Data Science,Java Book,Python Book
Clothing      249.95    4     62.49    Jeans,Shirt,Sweater
Electronics  4099.89    7     585.70
Headphones,iPhone,Keyboard,Laptop,Monitor,Mouse
Home          179.97    2     89.99    Blender,Coffee Maker
Sports        119.98    2     59.99    Basketball,Tennis Racket
```

Business Insights from the Output:

1. **Electronics** generates the highest revenue ($4,099.89) with 7 transactions
2. **Electronics** also has the highest average transaction value ($585.70)
3. **Clothing** has the most transactions (4) but lower revenue per transaction
4. **Home** category has good average transaction value despite fewer transactions

This demonstrates how MapReduce can process business data to generate actionable insights!py /user/hadoop/

Run streaming job

hadoop jar $HADOOP_HOME/share/hadoop/tools/lib/hadoop-streaming-*.jar
-files sales_mapper.py,sales_reducer.py
-mapper sales_mapper.py
-reducer sales_reducer.py
-input /user/hadoop/datasets/sales_data.txt
-output /user/hadoop/output/sales_analysis

View results

```
hdfs dfs -cat /user/hadoop/output/sales_analysis/part-00000
```

Lab 3: Apache Hive Data Warehousing

Learning Objectives

- Create and manage Hive tables
- Perform SQL-like queries on big data
- Understand partitioning and bucketing

Dataset Preparation

```
# Create customer data
cat << 'EOF' > customers.csv
customer_id,name,age,city,state,signup_date
1,John Smith,28,New York,NY,2023-01-15
2,Jane Doe,32,Los Angeles,CA,2023-01-20
3,Bob Johnson,45,Chicago,IL,2023-02-01
4,Alice Brown,29,Houston,TX,2023-02-15
5,Charlie Wilson,38,Phoenix,AZ,2023-03-01
6,Diana Miller,25,Philadelphia,PA,2023-03-10
7,Eve Davis,35,San Antonio,TX,2023-03-15
8,Frank Garcia,42,San Diego,CA,2023-04-01
9,Grace Rodriguez,31,Dallas,TX,2023-04-10
10,Henry Martinez,27,San Jose,CA,2023-04-15
EOF

# Create orders data
cat << 'EOF' > orders.csv
order_id,customer_id,product_category,product_name,quantity,price,order_date
1001,1,Electronics,Laptop,1,1299.99,2023-02-01
1002,2,Clothing,Jeans,2,79.99,2023-02-02
1003,3,Books,Python Guide,1,49.99,2023-02-03
1004,1,Electronics,Mouse,2,25.99,2023-02-05
1005,4,Clothing,Shirt,3,29.99,2023-02-10
1006,5,Electronics,Keyboard,1,89.99,2023-02-15
1007,2,Books,Java Handbook,1,59.99,2023-02-20
1008,6,Electronics,Monitor,1,299.99,2023-03-01
1009,7,Clothing,Dress,1,99.99,2023-03-05
1010,3,Electronics,Tablet,1,499.99,2023-03-10
EOF

# Upload to HDFS
hdfs dfs -mkdir -p /user/hadoop/hive/data
hdfs dfs -put customers.csv /user/hadoop/hive/data/
hdfs dfs -put orders.csv /user/hadoop/hive/data/
```

Start Hive

```
# Start Hive CLI
hive

# Or use Beeline (recommended)
beeline -u jdbc:hive2://localhost:10000
```

Hive Operations

1. Create Database and Tables

```sql
-- Create database
CREATE DATABASE IF NOT EXISTS ecommerce;
USE ecommerce;

-- Create customers table
CREATE TABLE customers (
    customer_id INT,
    name STRING,
    age INT,
    city STRING,
    state STRING,
    signup_date DATE
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
STORED AS TEXTFILE
TBLPROPERTIES ("skip.header.line.count"="1");

-- Load data into customers table
LOAD DATA INPATH '/user/hadoop/hive/data/customers.csv' INTO TABLE customers;

-- Create orders table
CREATE TABLE orders (
    order_id INT,
    customer_id INT,
    product_category STRING,
    product_name STRING,
    quantity INT,
    price DECIMAL(10,2),
    order_date DATE
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
STORED AS TEXTFILE
TBLPROPERTIES ("skip.header.line.count"="1");

-- Load data into orders table
LOAD DATA INPATH '/user/hadoop/hive/data/orders.csv' INTO TABLE orders;
```

2. Basic Queries

```sql
-- Show tables
SHOW TABLES;

-- Describe table structure
DESCRIBE customers;
DESCRIBE FORMATTED orders;

-- Simple SELECT queries
SELECT * FROM customers LIMIT 5;
SELECT * FROM orders LIMIT 5;

-- Filtering and aggregation
SELECT state, COUNT(*) as customer_count
FROM customers
GROUP BY state
ORDER BY customer_count DESC;

-- Calculate total sales by category
SELECT
    product_category,
    COUNT(*) as total_orders,
    SUM(quantity * price) as total_revenue,
    AVG(price) as avg_price
FROM orders
GROUP BY product_category
ORDER BY total_revenue DESC;
```

3. Advanced Queries and Joins

```sql
-- Join customers and orders
SELECT
    c.name,
    c.city,
    c.state,
    o.product_category,
    o.product_name,
    o.quantity * o.price as order_value
FROM customers c
JOIN orders o ON c.customer_id = o.customer_id
ORDER BY order_value DESC;

-- Customer analysis
SELECT
    c.name,
    c.age,
    c.state,
    COUNT(o.order_id) as total_orders,
    SUM(o.quantity * o.price) as total_spent,
    AVG(o.price) as avg_order_value
FROM customers c
LEFT JOIN orders o ON c.customer_id = o.customer_id
GROUP BY c.customer_id, c.name, c.age, c.state
HAVING total_orders > 0
ORDER BY total_spent DESC;

-- Monthly sales trend
SELECT
    YEAR(order_date) as year,
    MONTH(order_date) as month,
    COUNT(*) as orders_count,
    SUM(quantity * price) as monthly_revenue
FROM orders
GROUP BY YEAR(order_date), MONTH(order_date)
ORDER BY year, month;
```

4. Partitioned Tables

```
-- Create partitioned table
CREATE TABLE orders_partitioned (
    order_id INT,
    customer_id INT,
    product_name STRING,
    quantity INT,
    price DECIMAL(10,2),
    order_date DATE
)
PARTITIONED BY (product_category STRING)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
STORED AS TEXTFILE;

-- Enable dynamic partitioning
SET hive.exec.dynamic.partition = true;
SET hive.exec.dynamic.partition.mode = nonstrict;

-- Insert data with partitioning
INSERT INTO TABLE orders_partitioned PARTITION(product_category)
SELECT order_id, customer_id, product_name, quantity, price, order_date,
product_category
FROM orders;

-- Query partitioned table
SELECT * FROM orders_partitioned WHERE product_category = 'Electronics';

-- Show partitions
SHOW PARTITIONS orders_partitioned;
```

Lab 4: Apache HBase NoSQL Database

Learning Objectives

- Create and manage HBase tables
- Perform CRUD operations
- Understand column families and row keys

Start HBase

```
# Start HBase shell
hbase shell

# Check HBase status
status
version
```

HBase Operations

1. Create Tables and Column Families

```
# Create user profiles table
create 'user_profiles', {NAME => 'personal'}, {NAME => 'preferences'}, {NAME =>
'activity'}

# Create product catalog table
create 'product_catalog', {NAME => 'details'}, {NAME => 'pricing'}, {NAME =>
'inventory'}

# Create user activity table
create 'user_activity', {NAME => 'actions'}, {NAME => 'metadata'}

# List tables
list

# Describe table structure
describe 'user_profiles'
```

2. Insert Data (PUT operations)

```
# Insert user profile data
put 'user_profiles', 'user001', 'personal:name', 'John Smith'
put 'user_profiles', 'user001', 'personal:email', 'john.smith@email.com'
put 'user_profiles', 'user001', 'personal:age', '28'
put 'user_profiles', 'user001', 'personal:city', 'New York'
put 'user_profiles', 'user001', 'preferences:category', 'Electronics'
put 'user_profiles', 'user001', 'preferences:notifications', 'true'
put 'user_profiles', 'user001', 'activity:last_login', '2023-06-01'
put 'user_profiles', 'user001', 'activity:total_orders', '5'

put 'user_profiles', 'user002', 'personal:name', 'Jane Doe'
put 'user_profiles', 'user002', 'personal:email', 'jane.doe@email.com'
put 'user_profiles', 'user002', 'personal:age', '32'
put 'user_profiles', 'user002', 'personal:city', 'Los Angeles'
put 'user_profiles', 'user002', 'preferences:category', 'Books'
put 'user_profiles', 'user002', 'preferences:notifications', 'false'
put 'user_profiles', 'user002', 'activity:last_login', '2023-06-02'
put 'user_profiles', 'user002', 'activity:total_orders', '3'
```

```
# Insert product data
put 'product_catalog', 'prod001', 'details:name', 'iPhone 14'
put 'product_catalog', 'prod001', 'details:category', 'Electronics'
put 'product_catalog', 'prod001', 'details:brand', 'Apple'
put 'product_catalog', 'prod001', 'pricing:price', '999.99'
put 'product_catalog', 'prod001', 'pricing:discount', '10'
put 'product_catalog', 'prod001', 'inventory:stock', '50'
put 'product_catalog', 'prod001', 'inventory:warehouse', 'WH001'

put 'product_catalog', 'prod002', 'details:name', 'MacBook Pro'
put 'product_catalog', 'prod002', 'details:category', 'Electronics'
put 'product_catalog', 'prod002', 'details:brand', 'Apple'
put 'product_catalog', 'prod002', 'pricing:price', '1999.99'
put 'product_catalog', 'prod002', 'pricing:discount', '5'
put 'product_catalog', 'prod002', 'inventory:stock', '25'
put 'product_catalog', 'prod002', 'inventory:warehouse', 'WH002'
```

3. Read Data (GET and SCAN operations)

```
# Get specific row
get 'user_profiles', 'user001'

# Get specific column family
get 'user_profiles', 'user001', 'personal'

# Get specific column
get 'user_profiles', 'user001', 'personal:name'

# Scan entire table
scan 'user_profiles'

# Scan with column family filter
scan 'user_profiles', {COLUMNS => 'personal'}

# Scan with column filter
scan 'user_profiles', {COLUMNS => 'personal:name'}

# Scan with row key range
scan 'user_profiles', {STARTROW => 'user001', ENDROW => 'user003'}

# Scan with limit
scan 'user_profiles', {LIMIT => 2}

# Count rows
count 'user_profiles'
```

4. Update and Delete Operations

```
# Update existing data
put 'user_profiles', 'user001', 'personal:age', '29'
put 'user_profiles', 'user001', 'activity:total_orders', '6'

# Delete specific column
delete 'user_profiles', 'user001', 'preferences:notifications'

# Delete entire row
deleteall 'user_profiles', 'user002'

# Verify changes
get 'user_profiles', 'user001'
scan 'user_profiles'
```

5. Advanced Operations

```
# Create table with versioning
create 'user_sessions', {NAME => 'session_data', VERSIONS => 3}

# Insert multiple versions
put 'user_sessions', 'session001', 'session_data:status', 'active'
put 'user_sessions', 'session001', 'session_data:status', 'inactive'
put 'user_sessions', 'session001', 'session_data:status', 'expired'

# Get all versions
get 'user_sessions', 'session001', {COLUMN => 'session_data:status', VERSIONS =>
3}

# Filter operations
scan 'product_catalog', {FILTER => "ValueFilter(=,'substring:Apple')"}
scan 'user_profiles', {FILTER => "ColumnPrefixFilter('personal:')"}

# Atomic increment
incr 'user_profiles', 'user001', 'activity:total_orders', 1
get 'user_profiles', 'user001', 'activity:total_orders'
```

6. Bulk Load Example

Create bulk data file and load:

```
# Exit HBase shell first
exit

# Create bulk data file
cat << 'EOF' > bulk_users.txt
user003 personal:name   Alice Johnson
user003 personal:email  alice.j@email.com
user003 personal:age    25
```

```
user003 preferences:category    Clothing
user004 personal:name   Bob Wilson
user004 personal:email  bob.w@email.com
user004 personal:age    35
user004 preferences:category    Sports
EOF

# Use importtsv tool
hbase org.apache.hadoop.hbase.mapreduce.ImportTsv \
  -Dimporttsv.separator='\t' \
  -
Dimporttsv.columns=HBASE_ROW_KEY,personal:name,personal:email,personal:age,prefere
nces:category \
  user_profiles \
  /path/to/bulk_users.txt
```

Lab 5: Apache Spark Data Processing

Learning Objectives

- Process data using Spark RDDs and DataFrames
- Perform transformations and actions
- Compare Spark with MapReduce performance

Start Spark

```
# Start Spark shell (Scala)
spark-shell

# Or PySpark (Python)
pyspark

# For SQL operations
spark-sql
```

Dataset Preparation

```
# Create comprehensive sales dataset
cat << 'EOF' > sales_large.csv
order_id,customer_id,product_category,product_name,quantity,price,order_date,regio
n
1001,101,Electronics,Laptop,1,1299.99,2023-01-15,North
1002,102,Clothing,Jeans,2,79.99,2023-01-16,South
1003,103,Books,Python Guide,1,49.99,2023-01-17,East
1004,101,Electronics,Mouse,2,25.99,2023-01-18,North
1005,104,Clothing,Shirt,3,29.99,2023-01-19,West
1006,105,Electronics,Keyboard,1,89.99,2023-01-20,North
```

```
1007,102,Books,Java Handbook,1,59.99,2023-01-21,South
1008,106,Electronics,Monitor,1,299.99,2023-01-22,East
1009,107,Clothing,Dress,1,99.99,2023-01-23,West
1010,103,Electronics,Tablet,1,499.99,2023-01-24,East
1011,108,Sports,Basketball,1,29.99,2023-01-25,North
1012,109,Sports,Tennis Racket,1,89.99,2023-01-26,South
1013,110,Home,Coffee Maker,1,79.99,2023-01-27,West
1014,101,Electronics,Headphones,1,199.99,2023-01-28,North
1015,111,Books,Data Science Book,1,69.99,2023-01-29,East
EOF

# Upload to HDFS
hdfs dfs -put sales_large.csv /user/hadoop/spark/data/
```

Spark Operations

1. RDD Operations (Scala)

```
// Start spark-shell
// Read data as RDD
val salesRDD = sc.textFile("/user/hadoop/spark/data/sales_large.csv")

// Remove header
val header = salesRDD.first()
val dataRDD = salesRDD.filter(row => row != header)

// Parse CSV data
case class Sale(order_id: Int, customer_id: Int, category: String,
               product: String, quantity: Int, price: Double,
               order_date: String, region: String)

val parsedRDD = dataRDD.map(line => {
  val fields = line.split(",")
  Sale(fields(0).toInt, fields(1).toInt, fields(2), fields(3),
       fields(4).toInt, fields(5).toDouble, fields(6), fields(7))
})

// Basic RDD operations
parsedRDD.count()
parsedRDD.take(5).foreach(println)

// Transformations
val electronicsRDD = parsedRDD.filter(_.category == "Electronics")
val revenueRDD = parsedRDD.map(sale => (sale.category, sale.quantity *
sale.price))

// Actions
val categoryRevenue = revenueRDD.reduceByKey(_ + _).collect()
categoryRevenue.foreach(println)

// Find top customers by spending
```

```
val customerSpending = parsedRDD.map(sale => (sale.customer_id, sale.quantity *
sale.price))
                                .reduceByKey(_ + _)
                                .sortBy(_._2, false)
                                .take(5)
customerSpending.foreach(println)
```

2. DataFrame Operations (Python/PySpark)

```
# Start pyspark
from pyspark.sql import SparkSession
from pyspark.sql.functions import *

# Create SparkSession (if not already created)
spark = SparkSession.builder.appName("SalesAnalysis").getOrCreate()

# Read CSV as DataFrame
df = spark.read.option("header", "true").option("inferSchema",
"true").csv("/user/hadoop/spark/data/sales_large.csv")

# Show DataFrame info
df.show()
df.printSchema()
df.describe().show()

# Basic operations
df.count()
df.select("product_category", "price", "quantity").show()

# Aggregations
category_stats = df.groupBy("product_category").agg(
    count("order_id").alias("total_orders"),
    sum(col("quantity") * col("price")).alias("total_revenue"),
    avg("price").alias("avg_price"),
    max("price").alias("max_price")
)
category_stats.show()

# Filter and sort
electronics_df = df.filter(df.product_category ==
"Electronics").orderBy(desc("price"))
electronics_df.show()

# Customer analysis
customer_analysis = df.groupBy("customer_id").agg(
    count("order_id").alias("total_orders"),
    sum(col("quantity") * col("price")).alias("total_spent")
).orderBy(desc("total_spent"))
customer_analysis.show()

# Regional analysis
```

```
regional_analysis = df.groupBy("region", "product_category").agg(
    sum(col("quantity") * col("price")).alias("revenue")
).orderBy("region", desc("revenue"))
regional_analysis.show()
```

3. Spark SQL Operations

```
# Register DataFrame as temporary table
df.createOrReplaceTempView("sales")

# SQL queries
spark.sql("SELECT COUNT(*) as total_orders FROM sales").show()

spark.sql("""
    SELECT product_category,
           COUNT(*) as orders,
           SUM(quantity * price) as revenue,
           AVG(price) as avg_price
    FROM sales
    GROUP BY product_category
    ORDER BY revenue DESC
""").show()

spark.sql("""
    SELECT region,
           product_category,
           SUM(quantity * price) as revenue
    FROM sales
    GROUP BY region, product_category
    ORDER BY region, revenue DESC
""").show()

# Complex query with window functions
spark.sql("""
    SELECT customer_id,
           order_id,
           quantity * price as order_value,
           SUM(quantity * price) OVER (PARTITION BY customer_id) as
customer_total,
           ROW_NUMBER() OVER (PARTITION BY customer_id ORDER BY quantity * price
DESC) as order_rank
    FROM sales
    ORDER BY customer_id, order_rank
""").show()
```

4. Performance Comparison

```
# Word Count comparison between MapReduce and Spark
# Create large text file first
text_data = []
for i in range(10000):
    text_data.append(f"Apache Spark is faster than MapReduce for iterative
algorithms line {i}")

# Save to file
with open("large_text.txt", "w") as f:
    for line in text_data:
        f.write(line + "\n")

# Upload to HDFS
# hdfs dfs -put large_text.txt /user/hadoop/spark/data/

# Spark Word Count
import time

start_time = time.time()

text_rdd = sc.textFile("/user/hadoop/spark/data/large_text.txt")
word_counts = text_rdd.flatMap(lambda line: line.split()) \
                     .map(lambda word: (word.lower(), 1)) \
                     .reduceByKey(lambda a, b: a + b)

# Trigger action
result = word_counts.collect()

end_time = time.time()
print(f"Spark Word Count took: {end_time - start_time} seconds")
print(f"Total unique words: {len(result)}")

# Show top 10 words
sorted_words = sorted(result, key=lambda x: x[1], reverse=True)[:10]
for word, count in sorted_words:
    print(f"{word}: {count}")
```

5. Machine Learning with Spark MLlib

```
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.regression import LinearRegression
from pyspark.ml.evaluation import RegressionEvaluator

# Prepare data for ML (predict price based on quantity)
# Add some features
ml_df = df.withColumn("revenue", col("quantity") * col("price")) \
          .withColumn("category_encoded",
                     when(col("product_category") == "Electronics", 1)
                     .when(col("product_category") == "Clothing", 2)
                     .when(col("product_category") == "Books", 3)
```

```
                         .when(col("product_category") == "Sports", 4)
                         .otherwise(5))

# Feature engineering
assembler = VectorAssembler(
    inputCols=["quantity", "category_encoded"],
    outputCol="features"
)

ml_data = assembler.transform(ml_df).select("features", "price")
ml_data.show()

# Split data
train_data, test_data = ml_data.randomSplit([0.8, 0.2], seed=42)

# Train model
lr = LinearRegression(featuresCol="features", labelCol="price")
model = lr.fit(train_data)

# Make predictions
predictions = model.transform(test_data)
predictions.select("features", "price", "prediction").show()

# Evaluate model
evaluator = RegressionEvaluator(labelCol="price", predictionCol="prediction",
metricName="rmse")
rmse = evaluator.evaluate(predictions)
print(f"Root Mean Squared Error: {rmse}")
```

Lab 6: Integration Exercise - Complete Data Pipeline

Objective

Create an end-to-end data pipeline using all components.

Pipeline Architecture

1. **Data Ingestion**: Upload raw data to HDFS
2. **Data Processing**: Use MapReduce/Spark for data transformation
3. **Data Warehousing**: Store processed data in Hive
4. **Real-time Access**: Load data into HBase for fast queries
5. **Analytics**: Perform analysis using Spark

Implementation

```
# 1. Create comprehensive dataset
cat << 'EOF' > customer_transactions.csv
transaction_id,customer_id,product_id,category,amount,timestamp,location
T001,C001,P001,Electronics,299.99,2023-06-01 10:30:00,New York
```

```
T002,C002,P002,Clothing,79.99,2023-06-01 11:15:00,Los Angeles
T003,C001,P003,Books,49.99,2023-06-01 14:20:00,New York
T004,C003,P001,Electronics,299.99,2023-06-01 16:45:00,Chicago
T005,C002,P004,Electronics,199.99,2023-06-02 09:30:00,Los Angeles
EOF

# Upload to HDFS
hdfs dfs -put customer_transactions.csv /user/hadoop/pipeline/raw/
```

Complete Pipeline Script (Python/Spark)

```python
# Complete data pipeline
from pyspark.sql import SparkSession
from pyspark.sql.functions import *

# Initialize Spark
spark = SparkSession.builder \
    .appName("CompleteDataPipeline") \
    .enableHiveSupport() \
    .getOrCreate()

# 1. Data Ingestion
raw_df = spark.read.option("header", "true") \
    .option("inferSchema", "true") \
    .csv("/user/hadoop/pipeline/raw/customer_transactions.csv")

# 2. Data Processing and Transformation
processed_df = raw_df.withColumn("date", to_date(col("timestamp"))) \
                     .withColumn("hour", hour(col("timestamp"))) \
                     .withColumn("amount_category",
                                 when(col("amount") < 50, "Low")
                                 .when(col("amount") < 200, "Medium")
                                 .otherwise("High"))

# 3. Store in Hive
processed_df.write.mode("overwrite").saveAsTable("pipeline.processed_transactions"
)

# 4. Aggregate data
daily_summary = processed_df.groupBy("date", "category", "location") \
    .agg(count("transaction_id").alias("transaction_count"),
         sum("amount").alias("total_amount"),
         avg("amount").alias("avg_amount"))

daily_summary.write.mode("overwrite").saveAsTable("pipeline.daily_summary")

# 5. Real-time metrics
real_time_metrics = processed_df.groupBy("customer_id") \
    .agg(count("transaction_id").alias("total_transactions"),
         sum("amount").alias("total_spent"),
         max("timestamp").alias("last_transaction"))
```

```
real_time_metrics.show()

# Save results
real_time_metrics.write.mode("overwrite") \
    .option("header", "true") \
    .csv("/user/hadoop/pipeline/output/customer_metrics")

print("Pipeline completed successfully!")
```

Troubleshooting Common Issues

1. EMR Cluster Issues

```
# Check cluster status
aws emr describe-cluster --cluster-id <cluster-id>

# Check step status
aws emr list-steps --cluster-id <cluster-id>

# SSH connection issues
ssh -i ~/.ssh/id_rsa -o ServerAliveInterval=60 hadoop@<master-dns>
```

2. HDFS Issues

```
# Check HDFS health
hdfs dfsadmin -report
hdfs fsck / -files -blocks

# Safe mode issues
hdfs dfsadmin -safemode leave

# Permission issues
hdfs dfs -chmod -R 755 /user/hadoop/
```

3. Memory Issues

```
# Check memory usage
free -h
htop

# Adjust Spark memory
spark-shell --driver-memory 2g --executor-memory 2g
```

4. Common Error Solutions

- **Out of Memory**: Reduce dataset size or increase cluster resources
- **Permission Denied**: Check file permissions and ownership
- **Connection Refused**: Ensure services are running
- **File Not Found**: Verify file paths and existence

Assessment Questions

Practical Exercises

1. **HDFS**: Upload a 100MB file and analyze its block distribution
2. **MapReduce**: Implement a job to find the maximum sale amount per region
3. **Hive**: Create a partitioned table by date and analyze performance
4. **HBase**: Design a schema for storing user session data
5. **Spark**: Compare performance of the same query in Hive vs Spark

Mini-Project

Create a complete e-commerce analytics pipeline that:

1. Processes daily transaction logs
2. Stores data in both Hive (for analytics) and HBase (for real-time queries)
3. Generates daily reports using Spark
4. Handles data quality issues and missing values

This comprehensive lab guide provides hands-on experience with the entire Hadoop ecosystem while working within AWS Academy's constraints.