Topic 6: MapReduce Framework and Model

1. MapReduce Overview

What is MapReduce?

- **Most important processing framework** in Hadoop
- **Platform and language-independent** programming model
- Core of most big data and NoSQL platforms
- **Pattern/format** for writing programs to process large datasets

Historical Context

- Introduced by **Google in 2004**
- Breakthrough in big data technology history
- **Hadoop v1** supported MapReduce only
- Basis for high-level languages (Hive, Pig)
- Heavily influenced Spark concepts

Programming Model Structure

- **Map phase**: Process input data
- **Reduce phase**: Aggregate results
- Additional phases: Partition, Shuffle and Sort, Combine

2. Problems MapReduce Solves

Limitations of Early Distributed Computing

1. **Complexity in parallel programming**
    - Manual data distribution
    - Complex synchronization
    - Error-prone coding
2. **Hardware failures**
    - Node failures common in large clusters
    - Difficult to handle programmatically
3. **Bottlenecks in data exchange**
    - Network congestion
    - Data movement overhead
4. **Scalability problems**
    - Difficulty adding more nodes
    - Performance degradation

MapReduce Design Goals (2004 Google Paper)

1. **Automatic parallelization and distribution**
    - Framework handles data distribution
    - Developer focuses on business logic
2. **Fault tolerance**
    - Automatic recovery from failures
    - Task re-execution on failure
3. **Input/Output (I/O) scheduling**
    - Optimized data access
    - Data locality awareness
4. **Status and monitoring**
    - Progress tracking
    - Job monitoring tools

3. MapReduce Data Model: Key-Value Pairs

Fundamental Concept

- **All data** represented as key-value pairs
- Input, output, and intermediate records
- Also called name-value or attribute-value pairs

Key-Value Pair Structure

```
Key: Identifier (e.g., attribute name)
Value: Data associated with key

Example:
Key: "City"          Value: "Sydney"
Key: "Employer"      Value: "Cloudera"
```

Important Properties

- Keys **not required to be unique** in MapReduce
- Values can be **simple** or **complex objects**
- Flexible data representation

4. MapReduce Processing Model

Complete Data Flow

```
Input Data (HDFS)
    ↓
InputFormat & RecordReader
    ↓
Map Phase (Multiple Mappers in parallel)
    ↓
Partition (Group by key)
    ↓
Shuffle and Sort (Transfer and organize)
    ↓
[Optional: Combine Phase]
    ↓
Reduce Phase (Multiple Reducers in parallel)
    ↓
Output (HDFS)
```

Processing Stages

1. InputFormat and RecordReader

- **InputFormat**: Defines how to split input data
- **RecordReader**: Converts input into key-value pairs
- Input split typically = HDFS block (128MB default)

2. Map Phase

- **Function signature**: `map(in_key, in_value) → list(intermediate_key, intermediate_value)`
- Processes one record at a time
- Emits zero or more key-value pairs
- **n blocks → at least n Map tasks**

3. Partition Phase

- **Partitioner** function determines which Reducer gets each key
- **Ensures**: Each key → exactly one Reducer
- **Number of partitions** = number of Reducers
- Can be customized for specific requirements

4. Shuffle and Sort

- **"Magic" of MapReduce** - where optimization happens
- **Shuffle**: Transfer data from Mappers to Reducers
- **Sort**: Group all values for each key together
- **Goal**: Minimize network data transmission

5. Combine Phase (Optional)

- **Local aggregation** on Mapper node

- Reduces data sent to Reducers
- **Requirements**: Function must be **commutative** and **associative**
- **Examples**: SUM, COUNT work; AVERAGE doesn't

6. Reduce Phase

- **Function signature**: `reduce(intermediate_key, list(intermediate_value)) → (out_key, out_value)`
- Processes all values for one key
- Emits zero or more output key-value pairs

5. WordCount Example (MapReduce "Hello World")

Problem

Count frequency of each word in a large text corpus

Map Function

```
function Map(Long lineNo, String line):
    // lineNo: line position in text
    // line: actual text of line
    for each word w in line:
        emit(w, 1)
```

Example:

```
Input:  "Hello World Hello"
Output: (Hello, 1), (World, 1), (Hello, 1)
```

Shuffle and Sort (Automatic)

```
Groups:
(Hello, [1, 1])
(World, [1])
```

Reduce Function

```
function Reduce(String word, List<int> counts):
    // word: the word
    // counts: list of all counts for this word
    sum = 0
    for each c in counts:
```

```
        sum += c
    emit(word, sum)
```

Example:

```
Input:  (Hello, [1, 1])
Output: (Hello, 2)

Input:  (World, [1])
Output: (World, 1)
```

6. Common Map and Reduce Patterns

Common Map Functions

1. Filtering

```
// Filter log messages for errors only
Map(k, v) =
    if (ERROR in v) then
        emit(k, v)
```

2. Value Transformation

```
// Convert text to lowercase
Map(k, v) =
    emit(k, v.toLowerCase())
```

3. Projection

```
// Extract specific fields
Map(k, v) =
    emit(v.field1, v.field2)
```

Common Reduce Functions

1. Sum Reducer (Most Common)

```
reduce(k, list) =
    sum = 0
    for int i in list:
```

```
        sum += i
    emit(k, sum)
```

2. Count Reducer

```
reduce(k, list) =
    emit(k, list.length)
```

3. Max/Min Reducer

```
reduce(k, list) =
    max_val = MIN_VALUE
    for val in list:
        if val > max_val:
            max_val = val
    emit(k, max_val)
```

4. Average Reducer

```
reduce(k, list) =
    sum = 0
    count = 0
    for val in list:
        sum += val
        count += 1
    emit(k, sum/count)
```

7. Real-World Example: Shopping Cart Analysis

Problem

Analyze abandoned shopping carts to understand purchasing behavior

Input Data

```
User logs containing:
- User ID
- Items added to cart
- Items purchased
- Session information
```

Map Function

```
Map(logEntry):
    userId = logEntry.userId
    abandoned = logEntry.cartItems - logEntry.purchased

    for item in abandoned:
        emit(item, 1)  // Count abandoned items
```

Reduce Function

```
Reduce(item, counts):
    totalAbandoned = sum(counts)
    emit(item, totalAbandoned)
```

Result

Identify most commonly abandoned items for business insights

8. Average Social Contacts Example

Problem

For 1 billion people, compute average number of social contacts by age

SQL Equivalent

```
SELECT age, AVG(contacts)
FROM social.person
GROUP BY age
```

Map Function

```
function Map:
    input: integer K (batch of 1M records)

    for each social.person record in K-th batch:
        let Y = person.age
        let N = person.contacts
        produce output (Y, (N, 1))
    end for
end function
```

Example:

```
Input records:
{age: 25, contacts: 150}
{age: 25, contacts: 200}
{age: 30, contacts: 300}

Map output:
(25, (150, 1))
(25, (200, 1))
(30, (300, 1))
```

Shuffle and Sort

```
Groups by age:
(25, [(150, 1), (200, 1)])
(30, [(300, 1)])
```

Reduce Function

```
function Reduce:
    input: age Y

    S = 0      // Sum of contacts
    C_new = 0  // Count of people

    for each (N, C) in values:
        S += N * C
        C_new += C
    end for

    A = S / C_new  // Average
    produce output (Y, (A, C_new))
end function
```

Example:

```
Input:  (25, [(150, 1), (200, 1)])
Output: (25, (175, 2))  // Average = 350/2

Input:  (30, [(300, 1)])
Output: (30, (300, 1))  // Average = 300/1
```

9. Map-Only Jobs

When to Use

- No aggregation or reduction needed
- ETL (Extract, Transform, Load) operations
- File format conversions
- Image processing
- Simple filtering or transformation

Characteristics

- **Zero Reducers** configured
- **Only Map phase** executes
- **Direct output** to HDFS
- **Faster** than full MapReduce job

Example Use Cases

1. **ETL Routines**: Extract and transform data without aggregation
2. **File Conversion**: Convert CSV to JSON
3. **Image Processing**: Resize or filter images
4. **Data Validation**: Check data quality, emit errors

10. Combiner Function

Purpose

- **Local aggregation** on Mapper node **before** Shuffle and Sort
- **Reduces network traffic** between Map and Reduce
- **Mini-Reducer** running on Mapper output

Requirements for Combiner

Function must be:

1. **Commutative**: Order doesn't matter
    - Example: a + b = b + a ✓
2. **Associative**: Grouping doesn't matter
    - Example: (a + b) + c = a + (b + c) ✓

Examples

Can Use Combiner:

- SUM: Commutative and associative ✓
- COUNT: Commutative and associative ✓
- MAX/MIN: Commutative and associative ✓

Cannot Use Combiner:

- AVERAGE: Not associative ✗
    - (1 + 2)/2 + 3 ≠ 1 + (2 + 3)/2

- MEDIAN: Not commutative or associative ✗

Performance Impact

```
Without Combiner:
Mapper output: 1M records → Network → Reducer
Network traffic: HIGH

With Combiner:
Mapper output: 1M records → Combiner → 1K records → Network → Reducer
Network traffic: LOW (99.9% reduction)
```

11. Election Analogy for MapReduce

Voting Process

Map Phase (Polling Stations):

- Each polling station (Mapper) collects votes
- Outputs: (Candidate, Vote) pairs
- Multiple stations work in parallel

Partition (Courier Assignment):

- Group votes by district/region
- Assign to counting centers

Shuffle and Sort (Vote Transport):

- Transport votes to counting centers
- Sort by candidate at each center

Combine (Preliminary Count):

- Each station pre-counts votes
- Reduces data sent to central count

Reduce Phase (Central Counting):

- Each counting center (Reducer) totals votes for candidates
- Final output: (Candidate, Total Votes)

Key Points for Exam

Core Concepts

1. **MapReduce = Map + Shuffle/Sort + Reduce**
2. **Key-Value Pairs**: Fundamental data model
3. **Automatic Parallelization**: Framework handles distribution

4. **Fault Tolerance**: Automatic recovery from failures
5. **Data Locality**: Process data where it's stored

Function Signatures

Map: `(K1, V1) → list(K2, V2)` **Reduce**: `(K2, list(V2)) → list(K3, V3)` **Combiner**: Same as Reduce (must be commutative and associative)

Processing Flow

1. InputFormat → RecordReader
2. Map Phase (parallel)
3. Partition
4. Shuffle and Sort
5. [Optional: Combine]
6. Reduce Phase (parallel)
7. OutputFormat

Key Characteristics

- **n input blocks → at least n Mappers**
- **Each Mapper**: Processes one block
- **Each Reducer**: Processes one or more keys
- **Partitioner**: Determines Reducer for each key
- **Shuffle and Sort**: Groups values by key

Combiner Requirements

- Must be **commutative**: a ⊕ b = b ⊕ a
- Must be **associative**: (a ⊕ b) ⊕ c = a ⊕ (b ⊕ c)
- **Examples**: SUM ✓, COUNT ✓, AVG ✗

Common Patterns

Map Functions:

- Filtering
- Transformation
- Projection
- Parsing

Reduce Functions:

- Sum
- Count
- Max/Min
- Average
- Join

Design Goals (2004)

1. Automatic parallelization
2. Fault tolerance
3. I/O scheduling
4. Status monitoring

Map-Only Jobs

- No Reducers (0 Reducers)
- Direct output to HDFS
- Use for: ETL, conversions, filtering
- Faster than full MapReduce

WordCount Pattern

Map: `(lineNo, line) → list(word, 1)` **Reduce**: `(word, list(1,1,1,...)) → (word, count)`

Important to Remember

- Keys don't need to be unique
- Values can be complex objects
- Hadoop handles distribution automatically
- Developer focuses on Map and Reduce logic
- Shuffle and Sort is automatic ("magic")
- > 90% of MapReduce time is HDFS I/O