Hadoop & MapReduce Cheat Sheet

G.B. - v 0.0.1

---

1. Key Definitions

- **Big Data (3Vs):** Characterized by **Volume** (scale of data), **Variety** (different forms of data), and **Velocity**.
- **Apache Hadoop:** A scalable, fault-tolerant distributed system for Big Data. It moves the code/computation to the data (data locality) rather than moving data across the network.
- **HDFS (Hadoop Distributed File System):** Distributed storage system that provides global file namespace and stores data redundantly (typically 3 replicas) to ensure persistence and availability.
- **MapReduce Paradigm:** A programming model that separates the *what* (Map and Reduce functions defined by the developer) from the *how* (scheduling, synchronization, and fault tolerance handled by the framework).
- **Shuffle & Sort:** Phase between Map and Reduce where:
  - Data is **partitioned** across reducers
  - Keys are **sorted**
  - Values are **grouped by key**
- **Combiner:** A "mini-reducer" executed LOCALLY on the Mapper's output to pre-aggregate data and reduce network traffic during the shuffle and sort phase. It is not guaranteed to execute and it must be **associative** and **commutative**.

2. Architecture: Blocks, Mappers, and Reducers

- **Block Size & Splits:** Files in HDFS are split into fixed-size **blocks/chunks** (typically 64-128MB). An **Input Split** is a logical representation of data to be processed. Usually, one split corresponds to one HDFS block.
- **Number of Mappers:** Automatically determined by the Hadoop framework. **One Mapper task is instantiated for each Input Split**. You cannot arbitrarily set the exact number of Mappers. You have **at least one mapper** per input file. `Map` is called once per line in input file.
- **Number of Reducers:** This is **settable by the user** in the Driver class (e.g., `job.setNumReduceTasks(n)`).
- **Reducer Input:** The framework automatically groups all Mapper outputs by key (Shuffle and Sort phase). The Reducer receives a single key and an **Iterable list of all values** associated with that key: `key : [<value1>, <value2>, ...]`.

3. File Input Formats: Standard vs KeyValue

- **TextInputFormat (Standard):** Used for plain text files. Files are broken into lines.
  - **Key:** Position (offset) of the line in the file (`LongWritable`).
  - **Value:** The content of the entire line (`Text`).
- **KeyValueTextInputFormat:** Used for plain text files where each line follows a `key<separator>value` structure (default separator is the `TAB` character `\t`).
  - **Key:** The text preceding the separator (`Text`).
  - **Value:** The text following the separator (`Text`).

4. Counters

Counters are a useful channel for gathering statistics about the job (e.g., counting invalid records, specific events, etc.).

- **How they work:** They are global variables managed by the Hadoop framework. Every Mapper/Reducer can increment a counter, and the framework safely aggregates these increments.
- **Usage:**
  1. Define them using a Java `enum`.
  2. Increment them inside the Mapper or Reducer using `context.getCounter(EnumName).increment(1)`.
  3. Retrieve the final value in the Driver (you cannot see its content otherwise) **after** the job has completed successfully.

5. Complete Java Implementation

DRIVER CLASS (Universal)

```java
import org.apache.hadoop.conf.*;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.*;
import org.apache.hadoop.mapreduce.lib.input.*;
import org.apache.hadoop.mapreduce.lib.output.*;
import org.apache.hadoop.util.*;



public class MyDriver extends Configured implements Tool {
    // [MODIFIABLE] Define Global Counters
    public static enum COUNTERS {
    INVALID_RECORDS, SPECIAL_EVENTS }
    public int run(String[] args) throws Exception {
        Configuration conf = this.getConf();

        // --- PASSING PARAMETERS TO MAP/REDUCE ---
        // [MODIFIABLE] Set custom parameters here BEFORE Job.getInstance()
        conf.set("my.custom.threshold", args[3]);

        Job job = Job.getInstance(conf, "My Job Name");
        // JOIN MAP-SIDE Add hdfs file businessrules.txt in the distributed cache
        //job.addCacheFile(new Path("businessrules.txt").toUri());
        job.setJarByClass(MyDriver.class);

        // [MODIFIABLE] Input / Output paths
        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        // [MODIFIABLE] Input / Output formats
        job.setInputFormatClass(TextInputFormat.class);
        job.setOutputFormatClass(TextOutputFormat.class);

        // --- MAPPER SETUP ---
        job.setMapperClass(MyMapper.class);
        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(IntWritable.class);

        // --- REDUCER SETUP ---
        // [MODIFIABLE] Comment this line out if you want a MAP-ONLY job
        job.setReducerClass(MyReducer.class);

        // --- COMBINER SETUP ---
        // [MODIFIABLE] Uncomment to USE a Combiner
        // job.setCombinerClass(MyReducer.class);

        // --- NUM REDUCE TASKS (MAP-ONLY SWITCH) ---
        // [MODIFIABLE] Set to 0 for MAP-ONLY
        job.setNumReduceTasks(Integer.parseInt(args[2]));

        // --- FINAL OUTPUT TYPES ---
        // NOTE: If Map-only, these MUST match the MAPPER'S output classes!
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);

        // [FIXED] Run job and wait for completion
        boolean success = job.waitForCompletion(true);

        // --- READ COUNTERS AFTER JOB COMPLETION ---
        if (success) {
            long invalidCount = job.getCounters()
                                   .findCounter(COUNTERS.INVALID_RECORDS)
                                   .getValue();
            System.out.println("Total invalid records: " + invalidCount);
        }

        return success ? 0 : 1;
    }

    public static void main(String[] args) throws Exception {
        int res = ToolRunner.run(new Configuration(), new MyDriver(), args);
        System.exit(res);
    }
}
```

MAPPER CLASS

```java
import java.io.IOException;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.Mapper;

// [MODIFIABLE] Generics: <InputKey, InputVal, OutputKey, OutputVal>
class MyMapper extends Mapper<LongWritable, Text, Text, IntWritable> {
    private int threshold; // Variable to store the shared parameter
    private ArrayList<String> rules;
    // [OPTIONAL] setup() is executed ONLY ONCE per Mapper -instance-, not per line (HashMap!)
    protected void setup(Context context) {
        // [MODIFIABLE] Retrieve the shared parameter. Default value is 0.
        threshold = context.getConfiguration().getInt("my.custom.threshold", 0);
        String nextLine;
        rules = new ArrayList<String>();
        // Open the business rules file (that is shared by means of the distributed cache mechanism)
        URI[] CachedFiles = context.getCacheFiles();
        // This application has one single single cached file.
        // Its path is URIsCachedFiles[0]
        BufferedReader rulesFile = new BufferedReader(new FileReader(new File(CachedFiles[0].getPath())));
        // Each line of the file contains one rule
        while ((nextLine = rulesFile.readLine()) != null) {
          rules.add(nextLine);
        }
        rulesFile.close();
    }

    protected void map(LongWritable key, Text value, Context context) throws IOException,
     InterruptedException {
        String line = value.toString();
        // [MODIFIABLE] Counter usage: check for invalid/empty records
        if (line.trim().isEmpty()) {
            context.getCounter(COUNTERS.INVALID_RECORDS).increment(1);
            return; // Skip further processing for this record
        }
        // [MODIFIABLE] Standard mapping logic
        String[] words = line.split("\\s+");
        for (String word : words) {
            context.write(new Text(word), new IntWritable(1));
        }
    }
    // [OPTIONAL] cleanup() is executed ONLY ONCE per Mapper after all records are processed
    protected void cleanup(Context context) throws IOException, InterruptedException {
        // [MODIFIABLE] Emit results based on the in-mapper local variables
        // for (Map.Entry<String, Integer> entry : localMap.entrySet()) {
        //     context.write(new Text(entry.getKey()), new IntWritable(entry.getValue()));
        // }
    }

}
```

REDUCER CLASS (Can act as Combiner if associative and commutative)

```java
import java.io.IOException;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.Reducer;
// [MODIFIABLE] Generics: <InputKey, InputVal, OutputKey, OutputVal>
class MyReducer extends Reducer<Text, IntWritable, Text, IntWritable> {
    // [MODIFIABLE] Local variable to maintain state across MULTIPLE keys processed by this Reducer
    // private int totalSumAcrossAllKeys = 0;
    // [OPTIONAL] setup() is executed ONLY ONCE per Reducer instance
    protected void setup(Context context) {
        // You can read parameters here just like in the Mapper
        // int someParam = context.getConfiguration().getInt("my.param", 0);
    }
    // [FIXED] reduce() signature. Notice the Iterable for the values!
    protected void reduce(Text key, Iterable<IntWritable> values, Context context) throws IOException,
     InterruptedException {
        int sum = 0;

        // [MODIFIABLE] Custom logic: iterate over the list of values
        for (IntWritable val : values) {
            sum += val.get();
        }

        // [MODIFIABLE] Emit the final aggregated (Key, Value) pair
        context.write(key, new IntWritable(sum));
    }

    // [OPTIONAL] cleanup() is executed ONLY ONCE per Reducer after ALL keys have been processed
    protected void cleanup(Context context) throws IOException, InterruptedException {
        // [MODIFIABLE] Emit final global results based on Reducer's local variables
        // context.write(new Text("TOTAL_PROCESSED_BY_THIS_REDUCER"), new IntWritable(totalSumAcrossAllKeys)
     );
    }
}
```

CUSTOM DATA CLASS (Personalized Data Type)

```java
import java.io.*;
import org.apache.hadoop.io.Writable;

// [FIXED] Complex Values must implement Writable.
// If used as a Key, it must implement WritableComparable<MyCustomData>
public class MyCustomData implements Writable {

    // [MODIFIABLE] Custom internal state
    private float sum = 0;
    private int count = 0;

    // [FIXED] A default empty constructor is strictly required by Hadoop
    public MyCustomData() {}

    public MyCustomData(float sum, int count) {
        this.sum = sum;
        this.count = count;
    }

    // [FIXED] Serialization method (order matters)
    @Override
    public void write(DataOutput out) throws IOException {
        out.writeFloat(sum);
        out.writeInt(count);
    }

    // [FIXED] Deserialization method (order matters)
    @Override
    public void readFields(DataInput in) throws IOException {
        sum = in.readFloat();
        count = in.readInt();
    }

    public float getSum() { return sum; }
    public int getCount() { return count; }

    @Override
    public String toString() { return "Sum: " + sum + ", Count: " + count; }
}
```

6. MapReduce Design Patterns

Pattern	Examples	Tasks	Implementation & Management
Numerical Summarizations	Word count, Min/Max/Avg	N Mappers M Reducers	**Map:** Emits `(key, summary_field)`. **Reduce:** Computes the final aggregate statistics. **Comb.:** usable if the task is commutative and associative.
Inverted Index	Web search engines (Word → URLs)	N Mappers M Reducers	**Map:** Emits `(keyword, record_ID)`. **Reduce:** Receives IDs and concatenates them into a list. **Combiner:** Usually not useful (no numerical aggregation).
Counting with Counters	Count records, track specific events	N Mappers **0** Reducers	**Map-Only Job:** Mappers evaluate records and simply increment global `Counters` (no output emitted). Only driver can read and prints results after completion.
Filtering	Distributed grep, Data cleaning	N Mappers **0** Reducers	**Map-Only Job:** Mappers evaluate a condition/rule. Emits the record `(key, record)` only if the condition is met
Top K	Outliers analysis,s	N Mappers **1** Reducer	**Map:** Uses `setup()` to init a local list, updates it in `map()`, and emits local Top K in `cleanup()` with a shared "null" key. **Reduce:** Merges all local lists to find the global Top K.
Distinct	Duplicate removal, unique values	N Mappers M Reducers	**Map:** Emits the record as key `(record, null)`. **Reduce:** Framework groups identical keys; reducer emits `(key, null)` exactly once for each group.
Shuffling	Data anonymization, shuffling	N Mappers M Reducers	**Map:** Emits `(random_key, input_record)` . **Reduce:** Emits `(input_record, null)` to randomize order
Job Chaining	Complex workflows	Multiple Jobs	**Driver:** Manages workflow by executing jobs in order . Output of phase/Job i becomes the input for phase/Job $i+1$.
Reduce-Side Join	Natural/Theta/ Join on 2 large tables	N Mappers M Reducers	**Map:** One Mapper class per table. Emits `(common_attr, table_name + record)` . **Reduce:** Gathers records for a key, computes the "local join". Careful, **you must use a list usually (not the iterable)** .
Map-Side Join	Join a LARGE table with a SMALL table	N Mappers **0** Reducers	**Map-Only Job:** Small table is loaded into main memory via **Distributed Cache** in the `setup()` method . Mapper performs local join on the fly for each record of the large table

7. MapReduce SQL Operators

Operator	Description	Tasks	Implementation & Management
Selection (σ)	Applies a predicate $\forall$ record.	N Mappers **0** Reducers	**Map-Only Job:** Uses the *Filtering pattern*. If record r satisfies condition C, emit `(r, null)`. Otherwise, discard .
Projection (Π)	Keeps only a subset of attributes and **removes duplicates**.	N Mappers M Reducers	**Map:** Selects subset of attributes to form a new record r', emits `(r', null)` . **Reduce:** Emits `(r', null)` exactly once for each group (removes duplicates) .
Union ($R \cup S$)	Combines two relations with the same schema.	N Mappers M Reducers	**Map:** For each input record t in R or S, emits `(t, null)`. **Reduce:** Emits `(t, null)` exactly once for each group to eliminate duplicated records .
Intersection ($R \cap S$)	Returns records appearing in BOTH relations.	N Mappers M Reducers	**Map:** For $t \in R$, emits `(t, "R")`. For $t \in S$, emits `(t, "S")`. **Reduce:** Emits `(t, null)` ONLY if the list of values contains BOTH "R" and "S" (size is 2)
Difference ($R - S$)	Returns records appearing in R but NOT in S.	N Mappers M Reducers	**Map:** Needs 2 Mapper classes. For $t \in R$, emits `(t, "R")`. For $t \in S$, emits `(t, "S")` . **Reduce:** Emits `(t, null)` ONLY if the list of values contains ONLY the value "R" .
Join & Group By	Table joins and aggregations.	Varies	Implemented using **Join Patterns** (Map/Reduce side) or **Summarization Pattern** respectively .