Crash Course on mrjob (Python MapReduce Framework)

Introduction

`mrjob` is an open-source Python library originally developed by Yelp for writing and running MapReduce jobs in pure Python. It provides a simple, high-level interface that abstracts the complexity of Hadoop Streaming. With `mrjob`, you can write MapReduce logic using Python classes and run the same code in multiple environments: locally for testing, on Hadoop clusters (via YARN or standalone mode), or on cloud-based systems like Amazon EMR.

This crash course covers everything you need to understand and use `mrjob` effectively: installation, architecture, basic and advanced usage, input/output formats, debugging, and practical examples.

---

1. Installation

You can install `mrjob` through pip:

```
pip install mrjob
```

It supports Python 3.x and works on Windows, Linux, and macOS. To use it with Hadoop, Hadoop must be installed and configured properly. In Docker environments such as the one described earlier, you can install `mrjob` inside the container using the same command.

Verify the installation:

```
python -m pip show mrjob
```

---

2. Conceptual Overview

`mrjob` acts as a wrapper around the Hadoop Streaming API. It allows you to express a MapReduce job as a Python class that defines the mapper and reducer logic. Instead of writing separate mapper and reducer scripts, you write them as methods in a single class.

A MapReduce job in `mrjob` typically involves the following methods:

- `mapper(self, _, line)`: Processes each line of input and emits key-value pairs.
- `reducer(self, key, values)`: Receives all values for a given key and emits the final output.
- `combiner(self, key, values)`: (Optional) Performs partial aggregation before the reducer stage, reducing data transfer.

Each method uses Python generators to yield output pairs rather than returning them directly.

---

3. Basic Job Structure

Here is the simplest `mrjob` example: a word count program.

```python
from mrjob.job import MRJob

class MRWordCount(MRJob):

    def mapper(self, _, line):
        for word in line.split():
            yield word, 1

    def reducer(self, word, counts):
        yield word, sum(counts)

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

Explanation:

- Each line of input is passed to `mapper()`, which splits it into words and yields `(word, 1)`.
- The framework groups all identical words together and passes them to the `reducer()`.
- The reducer sums the counts for each word and outputs the final frequency.

---

4. Running mrjob

You can run `mrjob` in several ways, depending on your environment.

4.1 Local Mode

Runs entirely on your local machine without Hadoop:

```
python wordcount_mrjob.py input.txt
```

This is the easiest mode for testing and debugging.

4.2 Hadoop Mode

Runs on a Hadoop cluster (like the Docker setup):

```
python wordcount_mrjob.py -r hadoop hdfs:///input/input.txt -o
hdfs:///output_mrjob
```

The `-r` option specifies the runner. Supported runners include:

- `local`: local system

- `hadoop`: Hadoop cluster
- `emr`: Amazon Elastic MapReduce
- `inline`: runs within the same process (for debugging)

---

5. File Input and Output

In local mode, input and output are handled through normal files. In Hadoop mode, paths are prefixed with `hdfs:///`.

Examples:

- Local input: `python job.py data.txt`
- HDFS input: `python job.py -r hadoop hdfs:///data/input.txt -o hdfs:///data/output`

`mrjob` automatically handles compression, splitting, and combining outputs into `part-00000` files when running on Hadoop.

---

6. Combiner Function

Combiners reduce network traffic between the map and reduce stages by performing partial aggregation. They are optional but improve efficiency.

Example with a combiner:

```python
from mrjob.job import MRJob

class MRWordCount(MRJob):

    def mapper(self, _, line):
        for word in line.split():
            yield word, 1

    def combiner(self, word, counts):
        yield word, sum(counts)

    def reducer(self, word, counts):
        yield word, sum(counts)

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

Here, the combiner aggregates counts locally before sending them to reducers.

---

7. Multiple Steps and Pipelines

`mrjob` supports multi-step workflows, allowing you to chain multiple mappers and reducers. This is useful for more complex data processing pipelines.

Example:

```
from mrjob.job import MRJob
from mrjob.step import MRStep

class MRWordStats(MRJob):

    def steps(self):
        return [
            MRStep(mapper=self.mapper_get_words,
                   reducer=self.reducer_count_words),
            MRStep(reducer=self.reducer_find_max)
        ]

    def mapper_get_words(self, _, line):
        for word in line.split():
            yield word, 1

    def reducer_count_words(self, word, counts):
        yield None, (sum(counts), word)

    def reducer_find_max(self, _, word_count_pairs):
        yield max(word_count_pairs)

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

Explanation:

- Step 1 counts word occurrences.
- Step 2 finds the word with the maximum count.

---

8. Working with JSON and Other Formats

By default, `mrjob` reads and writes text data. However, you can use JSON, CSV, or custom formats.

Example using JSON:

```
from mrjob.job import MRJob
from mrjob.protocol import JSONValueProtocol

class MRJSONExample(MRJob):
    INPUT_PROTOCOL = JSONValueProtocol

    def mapper(self, _, record):
        yield record['category'], 1

    def reducer(self, category, counts):
        yield category, sum(counts)
```

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

This reads JSON input where each line is a JSON object.

---

9. Custom Configuration

You can configure your job through command-line options or configuration files (`mrjob.conf`).

Example configuration file (`mrjob.conf`):

```
runners:
  hadoop:
    hadoop_streaming_jar: /usr/local/hadoop/share/hadoop/tools/lib/hadoop-streaming.jar
    jobconf:
      mapreduce.job.reduces: 2
```

Then run:

```
python job.py -r hadoop --conf-path mrjob.conf hdfs:///input
```

This allows flexible tuning of Hadoop parameters like reducers, memory limits, and input formats.

---

10. Debugging mrjob Jobs

`mrjob` provides several built-in debugging aids.

- **Verbose mode:** Add `--verbose` to show detailed progress.
- **Local testing:** Run without Hadoop to isolate logic errors.
- **View logs:** When running on Hadoop, logs are stored in the Hadoop job history and can be viewed via the YARN ResourceManager UI or by checking the local `logs/` directory that `mrjob` creates.

---

11. Performance Tips

1. Use combiners when possible to reduce network I/O.
2. Avoid large numbers of small files; combine input into larger chunks.
3. Use JSON or simple delimited formats for lightweight serialization.
4. Use appropriate data types; avoid heavy objects or unnecessary parsing in mappers.
5. Always test locally before running large jobs on Hadoop.

---

12. Advanced Usage

12.1 Chaining Jobs Programmatically

You can programmatically invoke multiple `mrjob` jobs in a single Python script. This is helpful when combining different data processing tasks.

12.2 Using mrjob with Amazon EMR

If you have AWS credentials configured, you can submit jobs directly to Amazon EMR using:

```
python job.py -r emr s3://bucket/input/ -o s3://bucket/output/
```

`mrjob` automatically packages your job, uploads it to S3, and runs it on an EMR cluster.

12.3 Custom Protocols

Protocols define how data is serialized and deserialized between stages. Common options include `RawValueProtocol`, `JSONProtocol`, and `PickleProtocol`.

Example:

```python
from mrjob.protocol import RawValueProtocol
OUTPUT_PROTOCOL = RawValueProtocol
```

---

13. Testing and Unit Testing

Because `mrjob` jobs are plain Python classes, you can test them directly.

Example with `unittest`:

```python
import unittest
from wordcount_mrjob import MRWordCount
from mrjob.protocol import RawValueProtocol

class TestWordCount(unittest.TestCase):

    def test_mapper(self):
        job = MRWordCount()
        mapper_output = list(job.mapper(None, "hello world hello"))
        self.assertIn(("hello", 1), mapper_output)
        self.assertIn(("world", 1), mapper_output)

if __name__ == '__main__':
    unittest.main()
```

This allows you to verify logic before deploying on Hadoop.

---

14. Real-World Example: Average Calculation

This example demonstrates computing the average value for each key.

```python
from mrjob.job import MRJob

class MRAverage(MRJob):

    def mapper(self, _, line):
        name, value = line.split(',')
        yield name, float(value)

    def reducer(self, name, values):
        total = count = 0
        for v in values:
            total += v
            count += 1
        yield name, total / count

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

Input:

```
Alice,10
Bob,20
Alice,30
```

Output:

```
"Alice" 20.0
"Bob" 20.0
```

---

15. Summary

`mrjob` simplifies the process of writing MapReduce programs in Python by hiding the complexity of Hadoop Streaming. It enables you to:

- Write mapper, combiner, and reducer logic in Python.
- Test and debug jobs locally.
- Run the same code seamlessly on Hadoop or Amazon EMR.
- Use multiple steps, combiners, and custom data formats.
- Integrate with existing Hadoop and HDFS infrastructures.

Its advantages are simplicity, portability, and the ability to test code quickly before scaling up to large clusters. For teaching and learning purposes, `mrjob` is an excellent bridge between Python programming and distributed data processing concepts.

By mastering `mrjob`, students gain a practical understanding of how MapReduce works under the hood, while using familiar Python syntax instead of low-level Java or shell scripts.