MDS
Winter Term 2024/25
miniHive, Milestone 3

The goal of the coding project is to build a mini-version of Apache Hive, called *miniHive*. In the third milestone, we translate relational algebra queries into a physical query plan of MapReduce jobs. The MapReduce jobs can then be executed directly on Hadoop MapReduce.

1. Read the chapter on "Workflow Systems" for MapReduce engines in chapter 2.4.1 of the book "Mining Massive Datasets". The Python module `luigi` is such a workflow engine that can execute MapReduce jobs. `luigi` is already installed in the miniHive Docker container.

   A good starting point are the `luigi` examples on GitHub: `https://github.com/spotify/luigi/tree/master/examples`, e.g. the `hello_world.py` file. If you are interested in details, our setup is inspired by `luigi/examples/wordcount_hadoop.py`.

   However, **you are not expected to dig deep into** `luigi`. Understanding (and appreciating) *what* it does for you should be enough.

   If you want to develop your code on our personal computing device, outside of the miniHive Docker container, make sure you have `luigi` installed.

2. Add code to the provided Python module `ra2mr` which compiles a relational algebra query into a MapReduce workflow. We make the following simplifying assumptions:

   - We assume that the queries are the output of Milestone 2 and therefore use the operators σ, π, ρ, and $\bowtie_p$ (Equi-join) only. (We do not implement the cross product as a MapReduce job, it doesn't make much sense for *big* data.)
   - We assume that our input consists of "flat" JSON documents (no arrays or nested objects). Here is the data for relation `Person` from the *pizza* example. Each line is a key-value pair, separated by a tab. The key is the relation name, the value is a flat JSON document, encoding one tuple.

     ```
     Person        {"Person.name": "Amy", "Person.age": 16, "Person.gender": "female"}
     Person        {"Person.name": "Ben", "Person.age": 21, "Person.gender": "male"}
     ...
     ```

   You find the input files (`*.json`) and the skeleton code for `ra2mr.py` on GitHub: `https://github.com/miniHive/assignment/tree/master/milestone3`

   You only need to flesh out the code for Mapper and Reducer functions. The boilerplate code for building workflows with `luigi` is already provided!

   A task parameter of type `ra2mr.ExecEnv` controls the execution environment:

   - Set the task parameter `exec_environment` to `HDFS` to run tasks on Hadoop. This assumes that all input files reside in HDFS.
     This is our *production mode*. Unless you are willing to endure long waits, this mode is unsuitable for development. Instead, use `LOCAL` for development, described next.

- Set the task parameter `exec_environment` to `LOCAL` to run tasks without HDFS or Hadoop involved (or even installed). This assumes that all input files reside in the same directory as the `.py`-files.
  This is intended as the *development mode*: You will have quick turnarounds and can easily inspect any intermediate data written to temporary files.
- The unit tests set the task parameter `exec_environment` to `MOCK`. All files are then kept in main memory only. This mode is intended for *unit testing*.

This is how you would interact with `ra2mr` from within Python code:

```
import luigi
import radb
import ra2mr

# Take a relational algebra query...
raquery = radb.parse.one_statement_from_string("\project_{name} Person;")

# ... translate it into a luigi task encoding a MapReduce workflow...
task = ra2mr.task_factory(raquery, env=ra2mr.ExecEnv.HDFS)

# ... and run the task on Hadoop, using HDFS for input and output:
# (for now, we are happy working with luigi's local scheduler).
luigi.build([task], local_scheduler=True)
```

You can also execute `luigi` tasks from the command-line, as described here `https://luigi.readthedocs.io/en/stable/running_luigi.html`. This is useful for development and manual testing.

For instance, to evaluate a selection query locally on the container, you can write

```
PYTHONPATH=. luigi --module ra2mr SelectTask \
--querystring "\select_{gender='female'} Person;" \
--exec-environment LOCAL --local-scheduler
```

Inspect the `*.tmp`-files for intermediate results and the final output. Remember to clear any output files before starting the next task, since `luigi` will refuse to recompute them.

Similarly, to evaluate a projection query while writing to the local file system, write

```
PYTHONPATH=. luigi --module ra2mr ProjectTask \
--querystring "\project_{name} Person;" \
--exec-environment LOCAL --local-scheduler
```

To run the queries on Hadoop, simply switch to the `HDFS` environment. Make sure that all required input files have been loaded ("put") into HDFS, and that any previous output has been cleared.

3. Combine your code of all three milestones, to execute SQL queries in *miniHive*. The unit tests in `test_e2e.py` check this for you.

Remarks: For Milestone 3, you are not required to make your implementation particularly efficient; instead, focus on correctness.

Do not implement a hard-coded solution, i.e., a solution that works only for the test cases provided. We will check all solutions for plagiarism using an automated tool.

Praktomat will run the suites of unit tests of `test_ra2mr.py` and `test_e2e.py` (both availabe on GitHub) when you upload your solution. Upload `sql2ra.py`, `raopt.py` (from the previous milestones) and `ra2mr.py` as *one submission* to Praktomat, in time for the deadline. As `ra2mr.py` relies on the output of `sql2ra.py` and `raopt.py`, you may encounter some problems introduced by your previous milestones. In this case, you may have to revise your previous milestone implementations accordingly.

The deadline for submitting Milestone 3 is December 2, 2024, 12 noon. A successful submission that passes all public tests and passes the plagiarism check earns 10 points.
