SDS, Uni Passau
Winter term 2020/21
miniHive, Milestone 4

Optimizing *miniHive*

In the fourth milestone, we optimize *miniHive*. The goal is to reduce the total amount of data stored in intermediary files within HDFS.

We have considered various options in class, and it is up to you which optimization(s) you implement. Chain folding is a good bet. You may also implement more than one idea.

1 Evaluation Data

We use data from the TPC-H benchmark, describing customers and their orders. This is a famous benchmark, where synthetic data can be generated based on a scale factor (SF).

The ER-like diagram in Figure 1 describes the schema, and has been taken from the official benchmark description. We assume that all data adheres to this schema.

The parentheses following each table name contain the prefix of the column names for that table. The arrows point in the direction of the one-to-many relationships between tables. The number/formula below each table name represents the cardinality (number of rows) of the table. Some are factored by SF, the scale factor.

2 Calling *miniHive*

For evaluating a single query, *miniHive* gets

- an optional flag telling miniHive to switch optimization on,
- the optional scale factor, allowing you to derive the *approximate* sizes of tables,
- the input files, already stored in HDFS or available as local files,
- the optional execution environment (LOCAL or HDFS), set to HDFS by default,
- the SQL query over TPC-H data to evaluate.

This is how we call the optimized *miniHive* for execution on local files:

```
miniHive.py --O --SF 1 --env LOCAL "select distinct N_NAME from NATION"
```

If called in `LOCAL` mode, this outputs an approximation of the HDFS storage costs (more on this later). **Make sure your submitted version does not write any other information to standard out, as this will confuse the test scripts.**

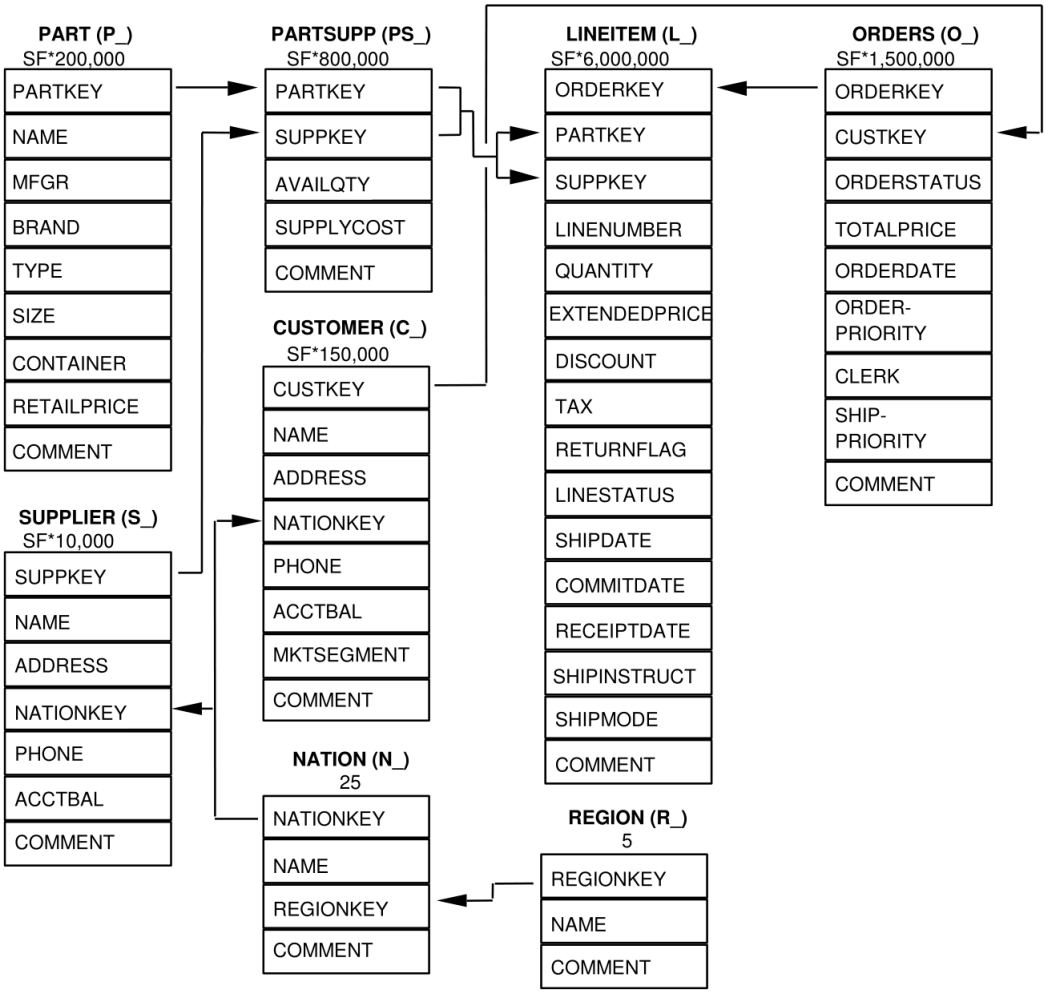

Figure 1: The TCP-H benchmark data [Source: `www.tpc.org`.]

3 Material in StudIP

- A sample dataset that was generated with scale factor 0.01 (`data_0_01.zip`).
- The file `miniHive.py`. You may alter this file.
- The file `costcouter.py`. You must not alter this file.
- A list of SQL test queries in `miniHive.q`.

4 What to Submit

- All files to run miniHive: `sql2ra.py`, `raopt.py`, `ra2mr.py` and `miniHive.py`.
- A README file (`README.md`) with a 1-paragraph description of the optimization(s) that you have implemented. The README must have this format:

```
# Author: <your lastname>, <your firstname>

# My approach:
<1-paragraph description of your optimizations>
```

5 Earning Points

Your submission will be tested by a fully automated Python script. **Make sure you do not break the functionality.**

The script parses the local temporary files, assuming that they have the format familiar from Milestone 3, namely "`some-key json-encoded-value`". **Do not change this format. Do not fiddle with the JSON-encoding of the value**. If you deviate from this format, your submission wont be considered for the bonus points ranking. The script then determines the total number of characters encoding MapReduce values in all output and intermediary files (the input files are the same for everybody, so we ignore them). **Do not fiddle with the intermediary files. If you do so, you will earn zero points for this milestone and your submission wont be considered for the bonus points ranking.**

The README file must be comprehensible. Your implementation must run in the unoptimized and the optimized mode.

You must implement at least one genuine optimization. Optimizing the format of temporary files (e.g., using compression) will earn zero points and you will be excluded from the bonus points ranking.

The automated scripts must be able to run all test queries from `miniHive.q` with your implementation on HDFS and on LOCAL files with a secret scale factor, and measure the storage costs for temporary files.

In order to receive the points, your implementation has to fulfil the following criteria:

- For all of the queries in `miniHive.q`, your costs must be lower or equal than those of my (unoptimized) implementation of Milestone 3 (your implementation of Milestone 3 should have the same costs; to assure that your optimization does not actually make things worse).
- For at least half of the queries in `miniHive.q`, your costs must be lower than those of the (unoptimized) implementation of Milestone 3. (You can't just resubmit your code from Milestone 3).
- For at least three queries in `miniHive.q`, your optimization must be able to reduce the costs by one third.

In addition, the best 10 implementations, measured based on the optimization and the produced costs, are earning bonus points on top. See the kickoff-slides in Stud.IP for all details. The queries for the ranking will be selected by us and may differ from the queries given in `miniHive.q`.

Remarks on grading: For Milestone 4, all you are required to provide is a correct and clean implementation that

- passes all Praktomat unit tests (the tests of milestone 3), and
- can execute the queries from the same unit tests on the Cloudera Quickstart VM, running on MapReduce proper, and
- meets all criteria specified in Section 5.

Upload your submission as one submission to Praktomat (`https://praktomat.sdbs.fim.uni-passau.de/SDS_20_WS/`), in time for the deadline.

Praktomat runs the unit tests of `test_ra2mr.py` and `test_e2e.py` (from Milestone 3) to check your solution. Your solution should also work for other, similar queries as in `test_ra2mr.py` and `test_e2e.py`.

After the Praktomat deadline has passed, your implementation will also be tested by us on the Cloudera Quickstart VM. Your implementation will be executed on Hadoop (`--exec-environment HDFS`) and data from HDFS will be processed. We recommend that you try this out beforehand (especially since the Cloudera Quickstart VM uses Python 3.6 instead of Python 3.8); also, just because the unit test pass locally does not mean that the code runs on MapReduce (e.g., when your code relies on global variables).