User-centric Systems for Data Science
Fall Semester 2021

Prof. John Liagouris
liagos@bu.edu

Assignment 4

Submission deadline: December 10 2021, 11:59 pm

This is the last course assignment and focuses on performance analysis for distributed data processing applications. For this assignment, you will use:

- *Your operator library from Assignments 1 and 2*
- *The Ray framework to distribute your Python code*
- *A tracing system (Jaeger) to instrument, collect, and visualize end-to-end application traces*
- *The Mass Open Cloud (MOC) to deploy your application in a cluster of machines*

The first three tasks of the assignment can be done locally (on your computer) but the fourth task must be done on MOC. Setting up the cloud environment for the fourth task requires some time and we highly recommend that you start early (see detailed instructions below).

In contrast to previous assignments, this assignment requires less coding and more independent exploration of Jaeger, Ray, and MOC. Make sure you read the provided documentation carefully.

The assignment must be completed individually, however, discussion with other students is encouraged.

0. Create an account on MOC **3**

1. TASK I: OpenTracing tutorial (credits: 20/100) **3**

2. TASK II: Restructure your operator library (credits: 20/100) **3**

3. TASK III: Trace your operator library on Ray (credits: 40/100) **4**

4. TASK IV: Distributed tracing on MOC (credits: 20/100) **4**

6. Testing **5**

7. Logging **5**

8. Git **5**

9. Deliverables **6**

10. Resources **6**

0. Create an account on MOC

If you plan to do Task IV, you must create a project on the Mass Open Cloud (MOC) using your BU credentials. Visit this link to request an account and use the information below:

Project name: CS599L1_FALL2021_FIRST_LAST_NAME

Project description: Project for CS 599 L1 "User-centric Systems for Data Science" (Instructor: I. Liagouris)

When your project is approved (expect a couple of days), log in and invite me (liagos@bu.edu) as a "Project Administrator". See here for instructions on inviting users to your project.

Students who have problems accessing MOC (e.g. cannot access even via VPN or connection is too slow) should contact me ASAP.

1. TASK I: OpenTracing tutorial (credits: 20/100)

The first task of this assignment is to familiarize yourself with Jaeger. OpenTracing has an excellent tutorial for beginners.

Hint #1: The tutorial suggests that you use Jaeger within a Docker container. To do so, you will first need to install Docker Desktop.

Hint #2: You may also want to have a look at this tutorial, which discusses some additional Jaeger features, such as RPC tracing and baggages (see also Lecture 18).

2. TASK II: Restructure your operator library (credits: 20/100)

The second task is to restructure your operator library from Assignments #1 and #2 so that it supports push-based data-parallel execution. To do so, you need to work on the Ray branch from Assignment #1 and implement the following changes:

A. Replace `get_next()` with the method `execute(tuples: List[ATuple]) -> bool` that applies the logic of the operator to a list of `tuples` (provided by the previous operator in the plan) and pushes the output batch of tuples to the next operator.

B. Modify the constructor of each relational operator so that it accepts one or more (data-parallel) instances of the same input logical operator.

C. Implement a `Sink` operator that simply storse the output of the query in memory. `Sink` should have a method `get_result() -> List[ATuple]` that returns the query output.

In the end, you should be able to execute the recommendation query from Assignment #1 by calling the `execute()` method on *each* data-parallel instance of the Scan operator to get the recommended movie id. Assuming two instances, this will look as follows:

```
# Your implementation of the recommendation query
scan_1 = Scan.remote(...)
scan_2 = Scan.remote(...)
select_1 = Select.remote(...)
...
sink = Sink(...)
...
# Start query execution
scan_1.execute.remote()
scan_2.execute.remote()
movie_id = ray.get(sink.get_result()) # Blocking call
```

Hint: The `execute()` method of the Scan operator should read lines from the input file within a loop and push the corresponding batches to the next operator in the plan until the file is exhausted.

3. TASK III: Trace your operator library on Ray (credits: 40/100)

The third task is to instrument your operator library using Jaeger and generate traces for an execution of the recommendation query using one instance per operator. The result should be a typical span tree, like those you generated in Task I. Each span in the tree should represent a call of the `execute()` method on a relational operator. The span hierarchy (cf. Lecture 17) must reflect the sequence of `execute()` calls during query evaluation.

Explore the span tree using Jaeger's web-based UI and report the top-2 most time-consuming operators in your implementation. Push the related screenshot(s) to your Gitlab repository.

4. TASK IV: Distributed tracing on MOC (credits: 20/100)

The fourth and final task is to deploy your instrumented library on MOC and perform Task III in a distributed setting. For this task, you have additional flexibility to decide the particular cluster configuration, actor placement, type of VMs, etc.

A. As a first step, you need to create a small cluster of VMs on MOC. You are free to choose between Windows and Linux machines (we recommend the latter). Make sure you activate a Floating IP ("Networks→Floating IPs") for at least one VM so that you can access the cluster from the outside world. To do so, you will also need to allow SSH connections ("Networks→ Security Groups") and upload your public SSH key ("Key Pairs"). In the end, you should be able to access your cluster via SSH using the Floating IP.

B. The next step is to install and deploy Ray on your cluster. See here for more information. The easiest way to check if Ray is deployed successfully is to search for Ray processes running on your cluster VMs. Before moving to the next step, make sure you can run a simple Ray application (e.g. calling a method on a remote actor).

C. The third step is to deploy your implementation for Task III on the Ray cluster and trace an execution of the recommendation query with the following parallelism: **8 Scan instances (4 per input file), 4 Select instances (one for each Scan instance reading from a partition of Friends.txt), 2 Join instances, 2 Group-by instances, 1 Order-by, 1 Project, and 1 Sink instance**. Each instance should be a Ray actor and each actor can be placed on *any* VM. Make sure parts of your recommendation pipeline run on different VMs but do *not* spend time optimizing the actor placement. The goal of this task is to profile *a* placement, understand where time is spent, and identify bottlenecks.

D. Push the related Jaeger screenshot(s) to your Gitlab repositoty.

Hint: Make sure each VM has more vCPUs than the number of Ray actors you run on it. For example, if you plan to run three operators/actors on a VM, make sure that it has at least 4 vCPUs.

6. Testing

You must have a simple test for each operator you implement and we strongly recommend using Pytest for this purpose. In case you are not familiar with Pytest, you might want to first spend some time reading the code snippets provided in the documentation.

All test functions must be added to the separate `tests.py` file provided with the code skeleton. Before submitting your solution, make sure the command `pytest tests.py` runs all your test functions successfully.

7. Logging

Logging can save you hours when debugging your program, and you will find it extremely helpful as your codebase grows in size and complexity. Always use Python's logger to generate messages and avoid using `print()` for that purpose. Keep in mind that logging has some performance overhead even if set to INFO level.

8. Git

You will use git to maintain your codebase and submit your solutions. If you are not familiar with git, you might want to have a look here. Note that well-documented code is always easier to understand and grade, so please make sure your code is clean, readable, and has comments.

Make sure your git commits have meaningful messages. Messages like "*Commit*" or "*Fix*", etc. are pretty

vague and must be avoided. Always try to briefly describe what each commit is about, e.g. "*Add Join operator*", "*Fix provenance tracking in AVG operator*", etc., so that you can easily search your git log if necessary.

Each time you finish a task (including the related tests), we strongly recommend that you use a commit message "Complete *Task X*". You will find these commits very helpful in case you want to rollback and create new branches in your repository.

9. Deliverables

Each submission must be marked with a commit message "*Submit Assignment X*" in git. If there are multiple submissions with the same message, we will only consider the last one before the deadline. **We will not accept late submissions for Assignment #4**. All solutions must be submitted by December 10th at 11.59pm.

Your submission must contain:

1. The code you wrote to solve the assignment tasks (create a new file `assignment_4.py`)
2. The code you wrote for testing (in `tests.py`)
3. The screenshots you generated as images (in a new folder `traces`). Use the following naming convention for screenshots: `task_x_screenshot_y.png`

Before submitting your solution, always make sure your code passes all tests successfully.

10. Resources

- Jaeger: https://www.jaegertracing.io
- MOC documentation: https://docs.massopen.cloud/en/latest/
- OpenTracing: https://opentracing.io
- Python tutorial: https://docs.python.org/3/tutorial/
- Python logger: https://docs.python.org/3/library/logging.html
- Pytest: https://docs.pytest.org/en/stable/
- Ray documentation: https://docs.ray.io/en/latest/
- Git Handbook: https://guides.github.com/introduction/git-handbook/