Universität Göttingen
Department of Computer Science

Julian Kunkel

Exercise Week 10
HPDA / Autumn 2025
240 Minutes Total
Discussion in Week 12: 2026-01-19

---

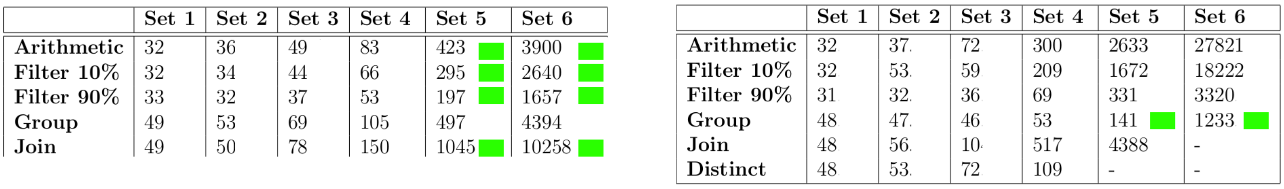
1. The tasks described in this worksheet are part of the formative assessment. They serve the purpose to prepare you for the examination. We will discuss the solutions during the next **interactive session** after they are handed out – while they fit to the lecture of the week they are handed out, they might be discussed in two weeks time due to the bi-weekly exercise schedule.
2. Make sure to plan your time for the whole sheet carefully. The complete exercise should represent approximately three hours of independent study. The time limit indicates how much time you should spend on each task, and not how much time you may actually need; it is important that you engage with the material and not that you complete all tasks perfectly. Feel free to collaborate and team up.
3. The exercises are designed to challenge you and train you further as guided self-study. The time limit might be too ambitious for you; you may team up with colleagues. It is not an issue as long as you manage to at least partially resolve each task within the time budget. If you (and your team) are struggling, reach out for help in Teams! You may also share your thoughts via the Studip Forum.
4. We recommend that you create a (private) Git repository (see `https://gitlab.gwdg.de`) where you store your findings and outcomes while processing the exercises. This portfolio of work can be useful in the future.

Contents

Task 1: Performance Analysis (60 min)

See the following benchmarks that were performed by IBM [1] and discussed on slide 52 of the lecture "Designing Distributed Systems and Performance Modelling":

- Test several operations, data set increases 10x in size
  - Set 1: 772 KB
  - Set 2: 6.4 MB
  - Set 3: 63 MB
  - Set 4: 628 MB
  - Set 5: 6.2 GB
  - Set 6: 62 GB
- Five data/compute nodes, configured to run 8 reduce and 11 map tasks:



	Set 1	Set 2	Set 3	Set 4	Set 5	Set 6
Arithmetic	32	36	49	83	423 🟩	3900 🟩
Filter 10%	32	34	44	66	295 🟩	2640 🟩
Filter 90%	33	32	37	53	197 🟩	1657 🟩
Group	49	53	69	105	497	4394
Join	49	50	78	150	1045 🟩	10258 🟩

	Set 1	Set 2	Set 3	Set 4	Set 5	Set 6
Arithmetic	32	37	72	300	2633	27821
Filter 10%	32	53	59	209	1672	18222
Filter 90%	31	32	36	69	331	3320
Group	48	47	46	53	141 🟩	1233 🟩
Join	48	56	10	517	4388	-
Distinct	48	53	72	109	-	-

Figure 1: Time for **Pig (left) and Hive (right)**.
Source: B. Jakobus (modified), Averaged performance[1]

Apply our performance assessment method for the results obtained with Pig and Hive. First, think how the different workloads may be implemented on the given hardware. Discuss the efficiency and potential causes of inefficiency.

Portfolio (directory: `10/performance`)

`10/performance/discussion.txt`	Your discussion of the performance analysis.

Further Reading

1. Web Archive IBM Benchmark: https://web.archive.org/web/20170829050635/http://www.ibm.com/developerworks/library/ba-pigvhive/pighivebenchmarking.pdf

Task 2: Building an API with OpenAPI (180 min)

With OpenAPI it is possible to define an API via a specification and to generate documentation, server stubs and client code from it. For this exercise you should use OpenAPI 3.X to define an API for a service that collects data about the hardware of servers. Define a list of hardware attributes for a sever, such as CPU model, CPU cores, RAM capacity and so on. Also include an ID field.
It should be possible to GET a list of all entries, GET a specific entry, to add an entry via POST, update an entry via PUT and remove an entry via DELETE. Define the path as you see fit, for example, using the id in the path for GET on a specific entry.

You can use the swagger.io online editor to work on your specification (https://editor.swagger.io/).
You don't need to implement security mechanisms for this exercise and it is enough to support JSON content types.

From the editor generate Python-Flask server code.
Create a folder for the server and extract the server code into it.
Create a Python virtual environment, install the dependencies and start the server.
Have a look at the README.md file that was generated.
You should then be able to find the generated API documentation under http://127.0.0.1:8080/ui/.

Try out sending a request to your defined endpoints. It should return "do some magic!".

To implement the backend find the files `__main__.py` and `default_controller.py`. The requests are handled in the controller while any initialization code should be called from main. Furthermore, under models you should find a generated class for your data type, e.g., server hardware.

Instantiate a dictionary or list that will hold your current set of objects at the bottom of the file that holds the definition for your model. You do not need to implement persistence to disk for now.
Implement the backend such that API requests add, read, modify and remove instances of objects in the list or dictionary that you created.

Check that your API works via the web interface. You may write tests to ensure that you get the expected results.

Consider that you want to make a change to your API such as adding a new API endpoint and a new data field for your object.
How difficult is such a change? What parts of your application do you need to change?

Hints

- When working on your backend implementation, you can add `debug=True` to the `app.run()` parameters for flask in main, to enable debug mode, which includes an auto re-loader that automatically detects changes and restarts the server.

Portfolio (directory: `10/OpenAPI`)

`10/OpenAPI/openapi-spec-server.yaml`	OpenAPI specification for your API.
`10/OpenAPI/openapi-server-code/`	Folder containing the source code for your OpenAPI Flask app