# Problem Statement

Issue#499

- To allow a seamless integration of DAPHNE into the existing Python-based data science ecosystem, exchanging data between DaphneLib and established Python libraries for data science must be simple
- Data transfer should be efficient via shared memory, ideally in a zero-copy manner
- Initial infrastructure for a zero-copy data exchange between DAPHNE and numpy has been developed in the context of a bachelor thesis in DAPHNE

- This project is about extending the existing infrastructure by supporting the efficient data exchange with more data science libraries: Pandas, TensorFlow, and PyTorch.

# Pandas

Extend the Python Interface with the ability to transfer Pandas Data-Frames between Daphne and Python

# Pandas

## Solution Design

- Create a Daphne Matrix per Column with Numpy
- Use the “createFrame” Constructor with the Matrixes
- Use the Daphne TMP-Script Building Mechanism
- Use the “saveDaphneLibResult” Function for transfer back

# Pandas

## Implementation – Simplified

### Daphne Context

`from_pandas`

*df*: *DataFrame*
*shared_memory*: *Bool*
*verbose*: *Bool*
*keepIndex*: *Bool*

*:return:*
*A Frame DAGNode*

### Operation Node

`compute`

*type*: *'shared memory'*
*verbose*: *Bool*
*useIndexColumn*: *Bool*

*:return:*
*A Pandas Data-Frame*

### Script

`build_code`

*dag_root*: *DAGNode*
*type*: *'shared memory'*

`execute`

*Writes and Executes the DaphneDSL Script*

# Pandas

Implementation – Example

daphne > tmpdaphne.daphne

```
V0=receiveFromNumpy(0,797051168,3,1,2);
V1=receiveFromNumpy(0,797713280,3,1,2);
V2=receiveFromNumpy(0,828940112,3,1,7);
V3=createFrame(V0,V1,V2,"index","a","b");
V4=receiveFromNumpy(0,828442016,3,1,2);
V5=receiveFromNumpy(0,828292336,3,1,7);
V6=receiveFromNumpy(0,828442040,3,1,2);
V7=createFrame(V4,V5,V6,"c","d","e");
V8=cbind(V3,V7);
saveDaphneLibResult(V8);
```

```
dc = DaphneContext()

df = pd.DataFrame(
    {"a": [1, 2, 3],
     "b": [1.1, -2.2, 3.3]})
F = dc.from_pandas(df, keepIndex=True)

df2 = pd.DataFrame(
    {"c": [30, 25, 0],
     "d": [ -1, 2.4, 5.9],
     "e": [7, 5, 9]})
F2 = dc.from_pandas(df2)

F = F.cbind(F2)

df3 = F.compute(useIndexColumn=True)
```

# Pandas

## Implementation – Example

daphne > tmpdaphne.daphne

```
V0=receiveFromNumpy(0,797051168,3,1,2);
V1=receiveFromNumpy(0,797713280,3,1,2);
V2=receiveFromNumpy(0,828940112,3,1,7);
V3=createFrame(V0,V1,V2,"index","a","b");
V4=receiveFromNumpy(0,828442016,3,1,2);
V5=receiveFromNumpy(0,828292336,3,1,7);
V6=receiveFromNumpy(0,828442040,3,1,2);
V7=createFrame(V4,V5,V6,"c","d","e");
V8=cbind(V3,V7);
saveDaphneLibResult(V8);
```

```python
dc = DaphneContext()

df = pd.DataFrame(
    {"a": [1, 2, 3],
     "b": [1.1, -2.2, 3.3]})
F = dc.from_pandas(df, keepIndex=True)

df2 = pd.DataFrame(
    {"c": [30, 25, 0],
     "d": [ -1, 2.4, 5.9],
     "e": [7, 5, 9]})
F2 = dc.from_pandas(df2)

F = F.cbind(F2)

df3 = F.compute(useIndexColumn=True)
```

# Pandas

## Implementation – Example

daphne > tmpdaphne.daphne

```
V0=receiveFromNumpy(0,797051168,3,1,2);
V1=receiveFromNumpy(0,797713280,3,1,2);
V2=receiveFromNumpy(0,828940112,3,1,7);
V3=createFrame(V0,V1,V2,"index","a","b");
V4=receiveFromNumpy(0,828442016,3,1,2);
V5=receiveFromNumpy(0,828292336,3,1,7);
V6=receiveFromNumpy(0,828442040,3,1,2);
V7=createFrame(V4,V5,V6,"c","d","e");
V8=cbind(V3,V7);
saveDaphneLibResult(V8);
```

```python
dc = DaphneContext()

df = pd.DataFrame(
    {"a": [1, 2, 3],
     "b": [1.1, -2.2, 3.3]})
F = dc.from_pandas(df, keepIndex=True)

df2 = pd.DataFrame(
    {"c": [30, 25, 0],
     "d": [ -1, 2.4, 5.9],
     "e": [7, 5, 9]})
F2 = dc.from_pandas(df2)

F = F.cbind(F2)

df3 = F.compute(useIndexColumn=True)
```

# Pandas

## Implementation – Example

daphne > tmpdaphne.daphne

```
V0=receiveFromNumpy(0,797051168,3,1,2);
V1=receiveFromNumpy(0,797713280,3,1,2);
V2=receiveFromNumpy(0,828940112,3,1,7);
V3=createFrame(V0,V1,V2,"index","a","b");
V4=receiveFromNumpy(0,828442016,3,1,2);
V5=receiveFromNumpy(0,828292336,3,1,7);
V6=receiveFromNumpy(0,828442040,3,1,2);
V7=createFrame(V4,V5,V6,"c","d","e");
V8=cbind(V3,V7);
saveDaphneLibResult(V8);
```

```python
dc = DaphneContext()

df = pd.DataFrame(
    {"a": [1, 2, 3],
     "b": [1.1, -2.2, 3.3]})
F = dc.from_pandas(df, keepIndex=True)

df2 = pd.DataFrame(
    {"c": [30, 25, 0],
     "d": [ -1, 2.4, 5.9],
     "e": [7, 5, 9]})
F2 = dc.from_pandas(df2)

F = F.cbind(F2)

df3 = F.compute(useIndexColumn=True)
```

# Pandas

## Implementation – Example

daphne > tmpdaphne.daphne

```
V0=receiveFromNumpy(0,797051168,3,1,2);
V1=receiveFromNumpy(0,797713280,3,1,2);
V2=receiveFromNumpy(0,828940112,3,1,7);
V3=createFrame(V0,V1,V2,"index","a","b");
V4=receiveFromNumpy(0,828442016,3,1,2);
V5=receiveFromNumpy(0,828292336,3,1,7);
V6=receiveFromNumpy(0,828442040,3,1,2);
V7=createFrame(V4,V5,V6,"c","d","e");
V8=cbind(V3,V7);
saveDaphneLibResult(V8);
```

```python
dc = DaphneContext()

df = pd.DataFrame(
    {"a": [1, 2, 3],
     "b": [1.1, -2.2, 3.3]})
F = dc.from_pandas(df, keepIndex=True)

df2 = pd.DataFrame(
    {"c": [30, 25, 0],
     "d": [ -1, 2.4, 5.9],
     "e": [7, 5, 9]})
F2 = dc.from_pandas(df2)

F = F.cbind(F2)

df3 = F.compute(useIndexColumn=True)
```

# Pandas

## Implementation – Example

daphne > tmpdaphne.daphne

```
V0=receiveFromNumpy(0,797051168,3,1,2);
V1=receiveFromNumpy(0,797713280,3,1,2);
V2=receiveFromNumpy(0,828940112,3,1,7);
V3=createFrame(V0,V1,V2,"index","a","b");
V4=receiveFromNumpy(0,828442016,3,1,2);
V5=receiveFromNumpy(0,828292336,3,1,7);
V6=receiveFromNumpy(0,828442040,3,1,2);
V7=createFrame(V4,V5,V6,"c","d","e");
V8=cbind(V3,V7);
saveDaphneLibResult(V8);
```

```
struct DaphneLibResult {
    void* address;
    int64_t rows;
    int64_t cols;
    int64_t vtc;
    //Added for Frame handling
    int64_t* vtcs;
    char** labels;
    void** columns;
};
```

```
df3 = F.compute(useIndexColumn=True)
```

# Tensorflow & PyTorch

Extend the Python Interface with the ability to transfer 2-d & n-d Tensors between Python and Daphne

# Tensorflow & PyTorch

## Solution Design

- Optional: Save the original shape of the n-d Tensor
- Flatten the Tensor and transform it to Numpy Array
- Reuse existing Numpy Functions for the Matrix Transfer
- Extend Compute Function to transform Matrix to Tensor

# Tensorflow & PyTorch

Implementation – Simplified

**Daphne Context**

`from_pytorch`

*tensor: Tensor*
*shared_memory: Bool*
*verbose: Bool*
*return_shape: Bool*

*:return:*
*A Matrix DAGNode*
*(original shape: shape)*

**Operation Node**

`compute`

*type: 'shared memory'*
*verbose: Bool*
*isTensorflow: Bool*
*isPytorch: Bool*
*shape: shape*

*:return:*
*A Pandas Data-Frame*

**Script**

`build_code`

*dag_root: DAGNode*
*type: 'shared memory'*

`execute`

*Writes and Executes the DaphneDSL Script*

# Challenges

During Implementation

# Challenges

Memory Management – Prevent Memory Overflow

Zero Copy Approach – Prevent Copying within all functions

Preserve Python object integrity during Daphne transfer

Memory Management – Prevent Memory Overflow

# Challenges

**Added a delete Function for manual Memory free up**

- Delete all the C++ pointers related to the object
- Reset the RefCounter for the object in Daphne
- Mark the Python Object as deleted

# Challenges

Zero Copy Approach – Prevent Copying within all functions

**Critically assessed & benchmarked each function**

- Checked the Memory usage with “memory-profiler”
- Usage of only Zero-Copy Functions within Python
- Benchmarked each transformation step

# Challenges

Preserve Python object integrity during Daphne transfer

**Python Input Objects can be reproduced from the Daphne Output**

- Compared Python Input to Output
- Implemented Functionality to preserve Tensor Shapes
- Implemented Functionality to preserve DF Index Column

# Further Adjustments

Extend the Python Interface with Daphne SQL and Join Operations for Frames

# Further Adjustments

Daphne SQL

```python
# Create Daphne Frames
customers_frame = dc.from_pandas(customers_df)

customers_table = customers_frame.registerView("Customers")

# SQL Example 1: Simple SELECT query
query1 = "SELECT c.CustomerID, c.CompanyName, c.ContactName FROM Customers as c"
result_frame1 = dc.sql(query1)
output_result1 = result_frame1.compute_sql([customers_table])

print(f"\nResult of running the SQL Query:\n\n{query1}\n")
print(output_result1)
```

# Further Adjustments

Daphne Joins

```python
# Customers DataFrame
customers_df = pd.DataFrame({
    'CustomerID': [101, 102, 103],
    'CompanyName': [1, 2, 3],
    'ContactName': [1, 2, 3]
})

# Orders DataFrame
orders_df = pd.DataFrame({
    'OrderID': [10643, 10692, 10702],
    'CustomerID': [101, 102, 103],
    'OrderDate': [20230715, 20230722, 20230725]
})

# Create Daphne Frames
customers_frame = dc.from_pandas(customers_df)
orders_frame = dc.from_pandas(orders_df)

join_result1 = customers_frame.innerJoin(orders_frame, "CustomerID", "CustomerID")
print(join_result1.compute())
```

# Experiments & Results

Conduct tests & experiments and benchmarks for our solutions

# Experiments & Results

## Pandas

pandas performancetest 1

pandas Performancetest 2

pandas Performancetest 3

pandas performancetest 1

# Experiments & Results

## Pandas

**Showcase handling of different DF types**

- importing DFs in various types from python into daphne with the from_pandas() function
- Series, Sparse DF and Categorical DF are converted to regular DFs to be supported
- rbind() and cartesian() are performed as a computations
- Minimal time lost for DF conversion step

# Experiments & Results

## Pandas

pandas performancetest 2

**Benchmark importing DFs with from_pandas()**

- The type of DF is checked and converted if necessary
- Execution times for each column, all columns, overall and type check are saved

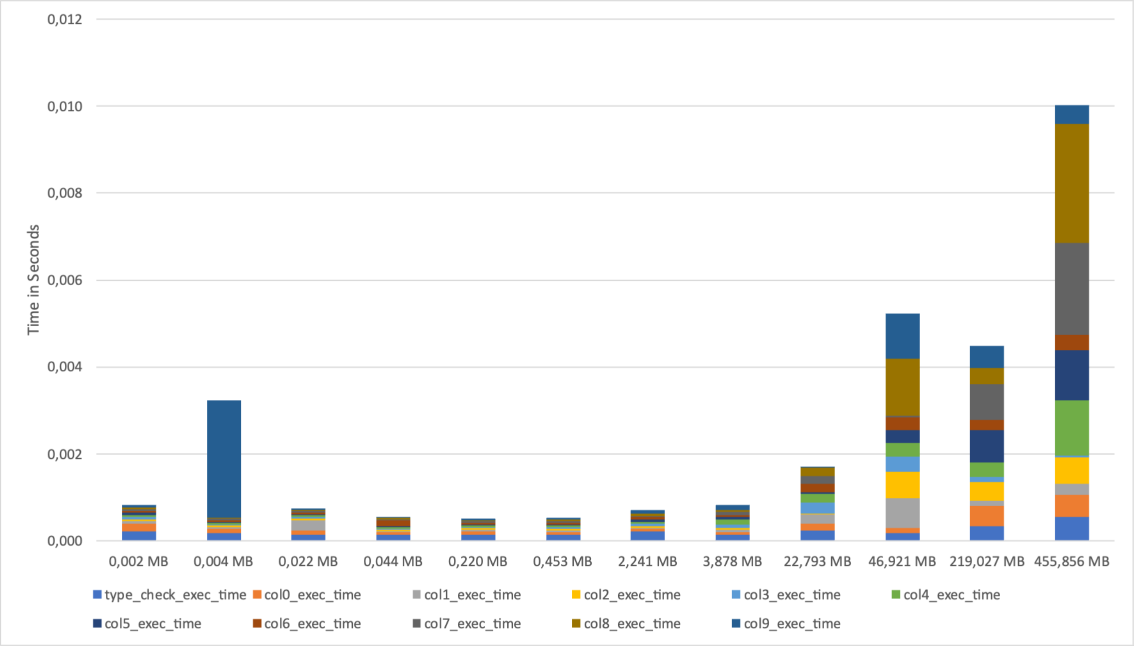
**Still Present Issue**

- With larger dataframe sizes, execution times per column show significant variance
- “Cold Start Effects”

# Experiments & Results

## Pandas

pandas performancetest 2



# Experiments & Results

Pandas

pandas performancetest 3

**Benchmark Daphne compute() for pandas DFs**

- Execution Times for the execute() function, the computing operation of building the dataframe and overall compute() function are saved

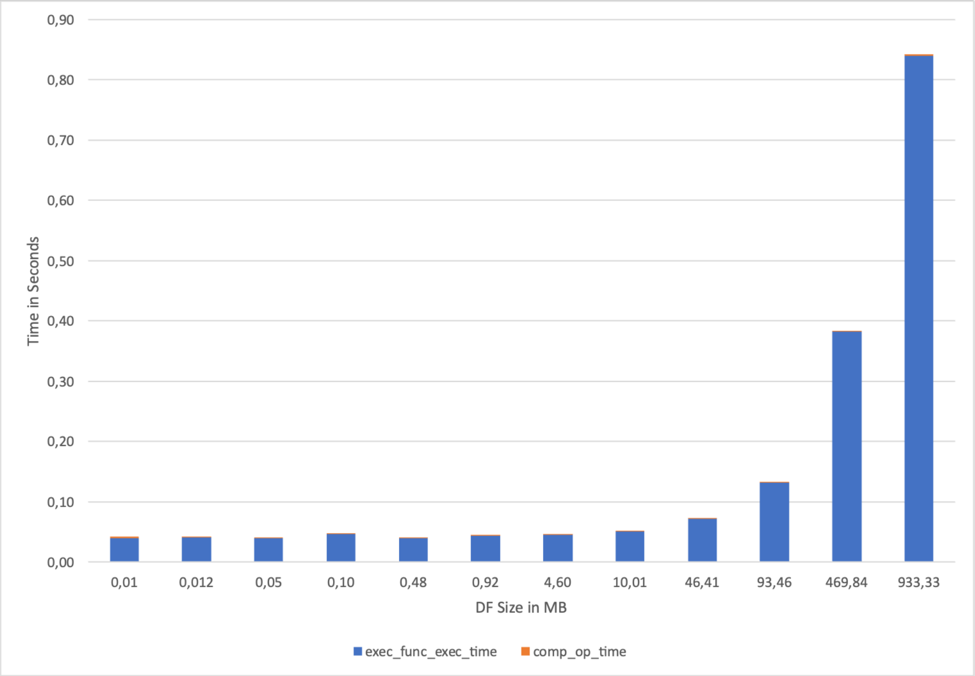
**Still Present Issue**

- “Execute”-Function is still a significant bottleneck
- “Cold Start Effects”

# Experiments & Results

## Pandas

pandas performancetest 3



# Experiments & Results

PyTorch

PyTorch Performancetest 1

PyTorch Performancetest 2

# Experiments & Results

## PyTorch

PyTorch Performancetest 1

**Benchmark importing PyTorch tensors with from_pytorch()**

- Execution Times for the PyTorch Tensor Reshape Execution, Numpy Execution time and overall execution are saved

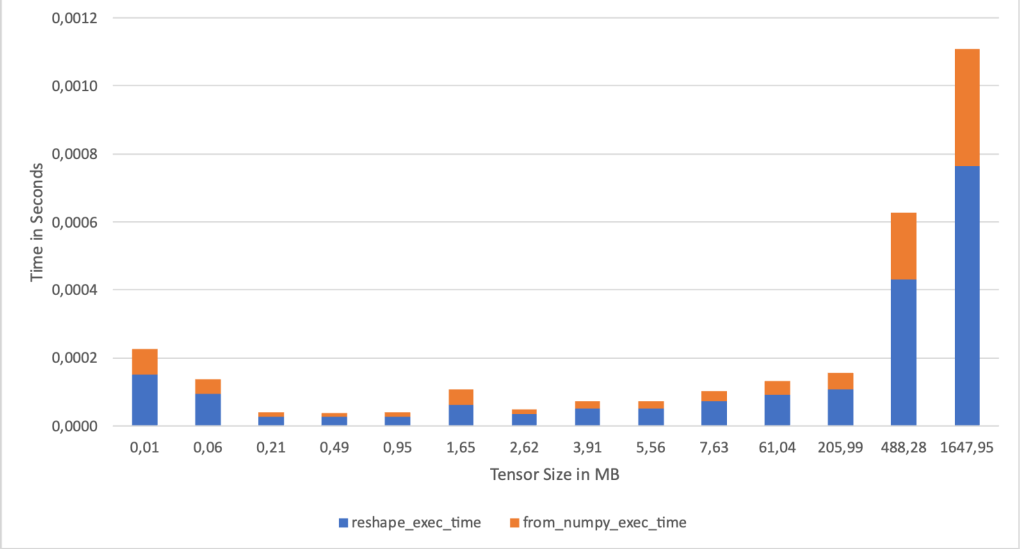
**Still Present Issue**

- “Cold Start Effects”

# Experiments & Results

## PyTorch

### PyTorch Performancetest 1



# Experiments & Results

## PyTorch

### PyTorch Performancetest 2

**Benchmark Daphne compute() for pytorch tensors**

- Execution Times for the execute() function execution, the PyTorch Tensor Transformation execution and overall execution are saved

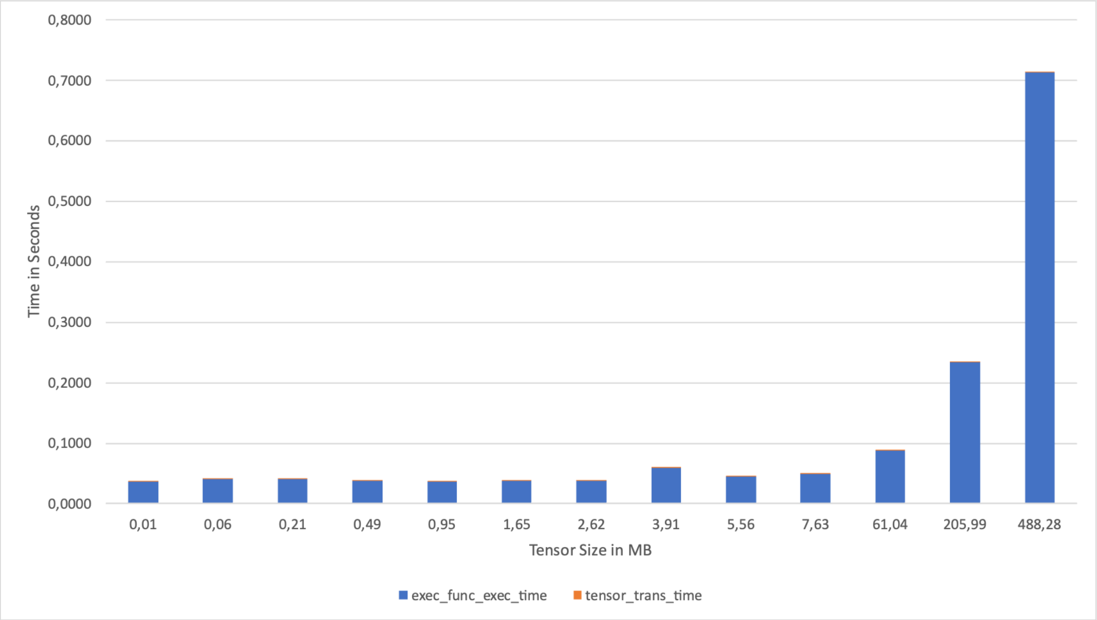
**Still Present Issue**

- “Execute”-Function is still a significant bottleneck
- “Cold Start Effects”

# PyTorch Performancetest 2

# Experiments & Results

## PyTorch



# Experiments & Results

## Tensorflow vs PyTorch

from_pytorch() is 75x faster than from_tensorflow()

Torch has 750x faster Tensor Transformation in compute()

Tensorflow has no native Numpy Transformation Function

# Conclusion

## Summary & Forward Directions

# Conclusion

Summary of PR#585

- Added Pandas Shared Memory Support for Frames
- Added Pytorch & Tensorflow Shared Memory Support for 2d & nd Tensors
- Added Updates to the Numpy functions & to the Frame and Matrix Operators in Python
- Added Support for Daphne SQL and Joins for Frames
- Added a delete function for Daphne Objects in Python to prevent Memory Overflow
- Designed all functions in a zero-copy manner with strong focus on performance
- Implemented Examples for all the added functions
- Implemented Benchmarks & Listed Benchmark Results
- Implemented Functionality Tests into the automated Test Script

# Conclusion

Areas to Improve & Forward Direction

## Data Frame handling

- String Support
- Enhance the current column wise approach (e.g., Vectorization)
- Add further Join Types (currently only Inner)
- Enhance the SQL capabilities
- Add further Frame Operators

## Tensor handling

- Add Native Tensor Support in Daphne
- Add Tensor specific Operators in Daphne
- Review the Tensor Functions in GPU Accelerated Environments
- Zero-Copy Tensor Transformation is only supported for CPU Operation!

# Conclusion

Areas to Improve & Forward Direction

## Overall Improvements

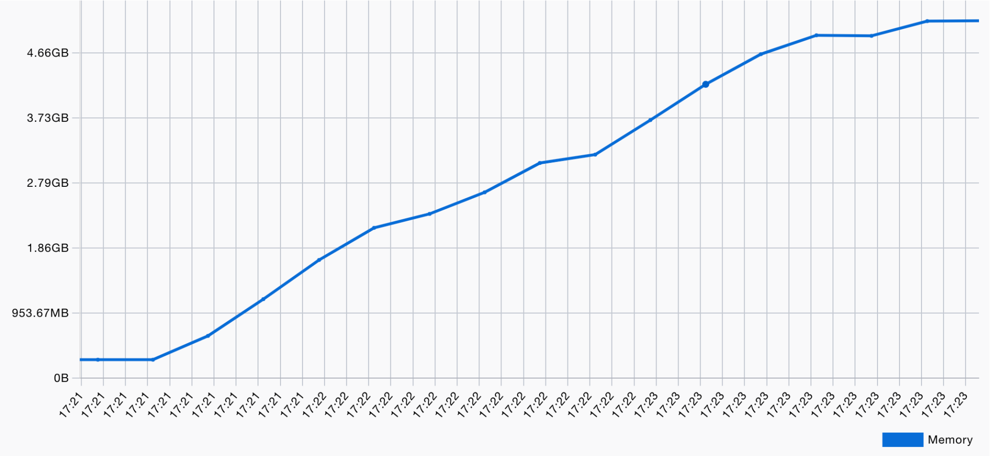
- Multi Return DAGNode Type for Multi Return Handling within Daphne (Partially Done)
- Enhance the current approach with temporary daphne files
- Enhance the Delete Function for DAGNode Objects (integrate into "__del__()" Function )

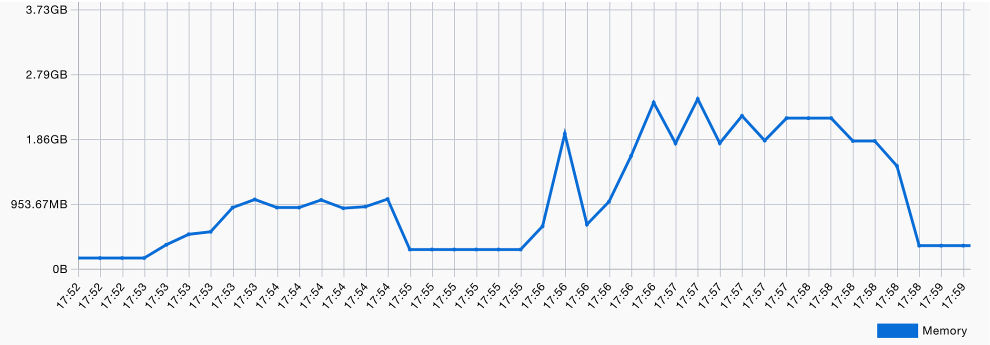
## Environment based

- Continue with further testing within different Environment Setups
  (Currently only Docker Containers within ARM MacBooks could be tested)
- Integrate Libraries into Daphne Containers via pip:
  - tensorflow (newest Version)
  - pandas (newest Version)
  - pillow (had to be installed manually for PyTorch)
  - torch, torchvision, torchaudio (newest Version)

# Backup

# Memory Management – Prevent Memory Overflow

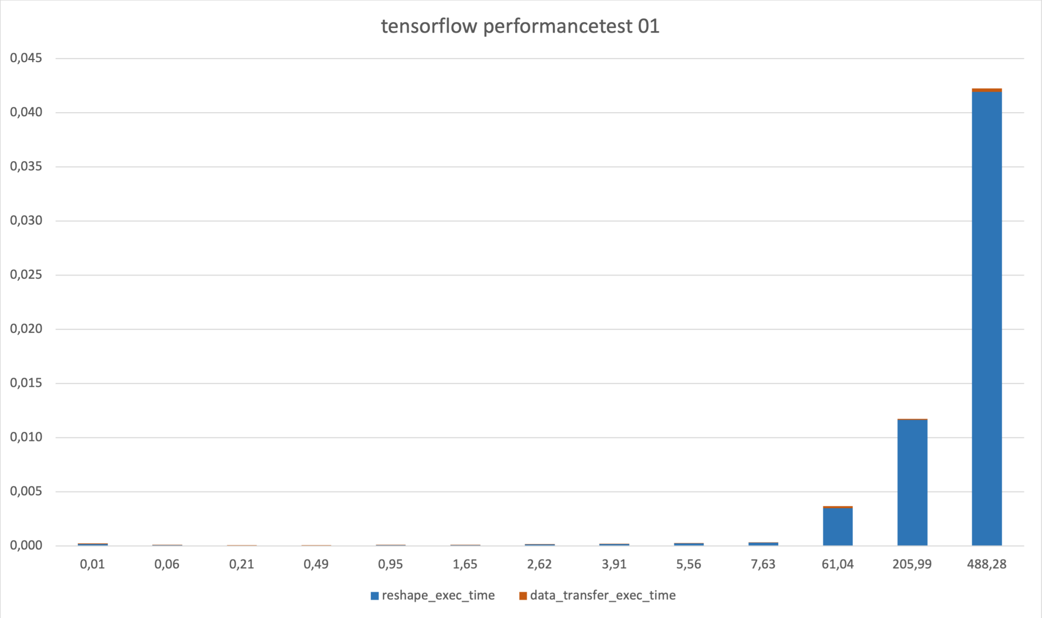


# Experiments & Results

TensorFlow

## Tensorflow Performancetest 1



# Experiments & Results

## TensorFlow

Tensorflow Performancetest 2