Bachelor Thesis

Dynamically Generating Language Bindings for HPC Libraries

Niklas Dießner

niklas.diessner@st.ovgu.de

January 22, 2023

First Reviewer:
Jun.-Prof. Dr. Michael Kuhn

Second Reviewer:
Michael Blesel

Supervisor:
Jun.-Prof. Dr. Michael Kuhn

Abstract

As Python is a fairly easy to use language, it is suited for beginners and prototyping, as it offers many features making it easy to get fast results. Performancewise however Python often plays the second fiddle compared to compiled languages, like C or Rust. The gap between convenience and performance can be bridged by generating language bindings for code written in a performance focused compiled language like C, so it can be used in Python.

JULEA is a storage framework fitted for High Performance Computing (HPC) applications. It is merely usable for applications written in C. By generating a wrapper automatically, this thesis aims to extend the compatibility of the JULEA clients to Python while adding as little overhead as possible in order to maintain Python support.

The result is a Python module that offers support for the key value, object and item client. With the drawback of a slight, however manageable decrease in performance, said support allows the implementation of newer clients, which might be added to the C library in the future, in the module by simply modifying one script and compiling the module.

Contents

1. Introduction **1**
1.1. Motivation 1
1.2. Thesis Goals 1
1.3. Thesis Outline 2

2. Background **3**
2.1. Python 3
2.2. Example Library 3
2.3. CFFI 4
2.4. SWIG 6
2.5. Pybind11 7
2.6. Ctypes 9
2.7. Cython 9
2.8. Shroud 10
2.9. JULEA 12

3. Design and Implementation **13**
3.1. Preprocess using Python 14
3.2. Preprocess using GCC 18
3.3. Improvements for easier usage 23

4. Related Work **27**
4.1. HPC and Python 27
4.2. Python performance 27
4.3. JULEA Python module 28
4.4. Language bindings for other languages 28

5. Evaluation **29**
5.1. Usage of the module 29
5.2. Performance benchmarks 30
5.2.1. Key Value 31
5.2.2. Object 33
5.2.3. Item 36
5.2.4. Database 39
5.2.5. Result 39

6. Conclusion **41**
6.1. Summary 41
6.2. Conclusion 41
6.3. Future Work 42

Bibliography **43**

Acronyms **47**

A. Listings **49**

B. Benchmark measurements **61**

Chapter 1.

Introduction

In this chapter, a small introduction will be given, including a brief discussion of the motivation and the goals of this thesis. This will be followed up by an outline on what the remainder of the thesis will contain.

1.1. Motivation

When writing code for HPC solutions C is a good language choice because it is a language that is fairly low level. This comes at the drawback of not being as user friendly as other high level languages. When prototyping and researching, this low level approach can lead to some overhead, which can divert the focus from the main topic.

For this case languages like Python offer many features that are more comfortable to use than C. Programming in Python yields results that are not as performant as the equivalent in C, but there is the possibilty of calling performant functions written in C from Python. An example for this is NumPy. This Python module is the Go-To module for scientific computing when working with Python. A large part of its functions are written in C and accessible from Python [Harris et al., 2008].

For research in the field of HPC this approach offers a good compromise between the performace of C and the comfort of Python. This philosophy of fast prototyping is in the spirit of the JULEA framework [Kuhn, 2017]. This storage framework is written in C and aims to enable users to tweak and play around in the user space. This framework would greatly benefit if it could be used in more languages than just C.

1.2. Thesis Goals

For this thesis the goal is to create a toolchain which enables developers of HPC libraries/frameworks written in C to add support for other languages e.g. Python. The main focus with regard to languages is Python, but if more languages can be supported with little overhead these options will also be explored.

The framework which will be used for examining the developed toolchain is JULEA [Kuhn, 2017]. It is a storage framework which is well documented and enables the user to write data to a variety of storage solutions in their HPC applications.

1.3. Thesis Outline

Chapter 2 will give some background on the languages and libraries used in the development of the toolchain. This will be followed by the concept for designing the solution and the implementation details for it in Chapter 3. In Chapter 4 the result will be compared to related work.

Chapter 5 will analyse the developed solution with regard to performance. Chapter 6 is going to conclude this thesis.

Chapter 2.

Background

In this chapter, some background on the used programming language and libraries will be provided. In addition there will be some background on the storage framework JULEA.

2.1. Python

Python is a programming language which relies on an interpreter to execute code. With scripts written in Python there is no need for compilation. The interpreter interprets the code at runtime. Memory is managed by a garbage collector. This makes prototyping and playing around with the language very quick and easy. But it comes at the drawback of overhead, meaning the same functionality written in Python and a native language like C will run faster in the native language.

Nevertheless Python is a very popular language surpassing C in popularity over the recent years [TIOBE Software B.V., 2022]. This is also thanks to the wide variety of libraries available for the language. As a result of this, there are many applications that are built upon Python [Python Software Foundation, 2001a].

To compensate the slower runtime, it is possible to call compiled code written in languages like C or C++ from Python. Some of these options will be explored in the following sections.

2.2. Example Library

For the sake of testing the capabilities of the different solutions, as well as checking how easy it is to get them up and running an example library is created. It contains a function for computing the power of a floating point number and a function for computing the n-th Fibonacci number (see Listing A.1 and A.2 in Listings). It will be referenced in the following sections as *example library*.

2.3. CFFI

The C Foreign Function Interface (CFFI) enables its users to run C functions from within Python. It can be used in two diffent modes [Rigo and Fijalkowski, 2012].

ABI (binary level interface), also called *out-of-line*, is the way of operation in which a shared libary is loaded into memory at runtime. The call to the library is then performed on the object created by the load instruction. This mode lets the user call functions of a shared libary without any previous compilation steps necessary.

When using ABI mode, the declaration of the functions provided by the library need to be set first. It is possible to use the header file of the library, as it contains all the relevant information and no further code needs to be written. To start, the declaration of the functions, contained in the example library, is imported into CFFI using the command `cdef` (see line 7 of Listing 2.1). One limitation of the cdef command is the missing support for imports and compiler directives. This means a header that includes another header can not be processed by CFFI in this way. It must be preprocessed and the result of the preprocessing must then be used for the `cdef` call. Macros which are available in GNU compiler collection (GCC) are also not supported. Afterwards the shared library `libexample.so` is then loaded into memory (see line 8 of Listing 2.1). All functions declared through the `cdef` command can then be called as methods of the object created with the `dlopen` command (see lines 12 and 15 of Listing 2.1).

```
1  import cffi, os
2
3  if __name__=="__main__":
4      ffi = cffi.FFI()
5      headerfile = os.getcwd()+"/example.h"
6      with open(headerfile) as h_file:
7          ffi.cdef(h_file.read())
8      lib = ffi.dlopen("./libexample.so")
9      bases = [ 1.2, 58.2, 2.0 ]
10     exponents = [ 2, 0, -2 ]
11     for i in range(3):
12         res = lib.power_verbose(bases[i], exponents[i])
13         print("in python:\t{:.2f}^{:d} is
              ↪ {:.2f}".format(bases[i], exponents[i], res))
14         print(30*"-")
15     print("{:d}th fibonacci number is {:d}".format(6, lib.fib(6)))
```

Listing 2.1: Test script for CFFI *ABI mode*

Because function calls are going through the *libffi* library, the ABI mode is much slower than the API mode. For the purpose of this thesis this circumstance is not optimal as it brings quite some overhead.

API (source level interface), also called *in-line*, compiles a CPython wrapper that calls the functions of the shared library directly. This wrapper can be imported as a Python module. Getting this mode to work takes a bit longer because of the initial compilation time, this time is, however, saved when running the code multiple times because the overhead is much smaller compared to the ABI mode.

Before compiling the wrapper, it is necessary to declare the functions provided by the library first. The method is the same as the one used in the ABI mode. The same limitations apply here with regard to includes, directives and macros. Again, the declarations of the header can be used, as they contain all information CFFI needs (see line 7 of Listing 2.2). Next some properties for the wrapper module need to be set with the `set_source` command. The first string will set the name of the module created when compiling (see line 9 of Listing 2.2). Secondly we declare which source will be included. Here the header of the example library is included again (see line 10 of Listing 2.2). The last three parameters are passed to GCC when compiling the wrapper (see lines 11-13 of Listing 2.2). Finally the compiler is invoked with all configuration set in the previous lines by calling the `compile` command.

```
1  import cffi
2  import os
3
4  ffi = cffi.FFI()
5  headerfile = os.getcwd()+"/example.h"
6  with open(headerfile) as h_file:
7      ffi.cdef(h_file.read())
8  ffi.set_source(
9          "cffi_libexample",
10         '#include "example.h"',
11         libraries=["example"],
12         library_dirs=[os.getcwd()],
13         extra_link_args=["-Wl,-rpath,."]
14 )
15 ffi.compile()
```

Listing 2.2: Build script for CFFI *API mode*

```
1  import cffi_libexample
2
3  if __name__ == "__main__":
4      bases = [ 1.2, 58.2, 2.0 ]
5      exponents = [ 2, 0, -2 ]
6      for i in range(3):
7          res = cffi_libexample.lib.power_verbose(bases[i],
              ↪ exponents[i])
8          print("in python:\t{:.2f}^{:d} is
              ↪ {:.2f}".format(bases[i], exponents[i], res))
9          print(30*"-")
10     print("{:d}th fibonacci number is {:d}".format(6,
          ↪ cffi_libexample.lib.fib(6)))
```

Listing 2.3: Test script for CFFI *API mode*

After compiling the wrapper and placing it in a folder which is part of the PYTHONHOME environment variable, it can just be imported as a module (see line 1 of Listing 2.3). The imported module contains the two properties `ffi` and `lib`. `ffi` offers some functions when working with C data types like pointers. Whereas the property we are interested in is the `lib`.

It contains all the functions we declared in our build script. Now we can just call the functions (see lines 7 and 10 of Listing 2.3).

2.4. SWIG

The Simplefied Wrapper and Interface Generator (SWIG) is a tool offering the ability to generate wrappers for C and C++ assemblies which can target a variety of high-level programming langugages, for example Python, Go and C# [Fulton et al., 1999].

To generate a Python module from a C library using SWIG an intermediate file needs to be generated. For the example library such a file can be seen in Listing 2.4.

In line 1 the name of the module is defined. It is followed by the declarations used for interfacing with the C/C++ library (see lines 2-7 of Listing 2.4). In lines 9-12 the functions which shall be interfaced by the module are declared.

```
%module example
%{
float   power(float base, int exponent);
float   power_verbose(float base, int exponent);
void    loop (float* result, float factor, int counter);
int     fib(int number);
%}

float   power(float base, int exponent);
float   power_verbose(float base, int exponent);
void    loop (float* result, float factor, int counter);
int     fib(int number);
```

Listing 2.4: Intermediate file for SWIG (`example.i`)

If the Python module generated by SWIG should interface all functions of a header, the intermediate file can be simplified, as shown in Listing 2.5.

```
%module example
%{
#include "example.h"
%}
%include "example.h"
```

Listing 2.5: Simplified version of intermediate file for SWIG (`example.i`)

Before building the module, SWIG needs to generate the code for the wrapper first. The invocation for the example library can be seen in Listing 2.6.

```
$ swig -python example.i
```

Listing 2.6: Generate SWIG wrapper code

With the code generated by this command, a shared library containing the Python module can be compiled. To accomplish this, the object files of both the library and the wrapper need to be compiled first (see Listing 2.7). The option -fPIC is set because we want to generate a shared library. Generating position independent code (PIC) is helpful when creating a shared library as it makes constant addresses accessable through a global offset table [Free Software Foundation, Inc., 1988]. We also need to provide the headers for Python with the -I flag (in the example the headers for Python 3.8 are used).

```
$ gcc -c -fPIC example.c example_wrap.c -I/usr/include/python3.8
```

Listing 2.7: Compile SWIG module

With the object files compiled, the final build step is linking the shared library. To achieve this the flag -shared is provided. We also provide the object files created in the compilation step. Lastly we specify the name of the output file (see Listing 2.8).

```
$ gcc -shared example.o example_wrap.o -o example.so
```

Listing 2.8: Link SWIG module

Now the module can be imported and the functions provided by the C Library can simply be called (see Listing 2.9).

```
import example

if __name__ == "__main__":
    bases = [ 1.2, 58.2, 2.0 ]
    exponents = [ 2, 0, -2 ]
    for i in range(3):
        res = example.power_verbose(bases[i], exponents[i])
        print("in python:\t{:.2f}^{:d} is
           ↪ {:.2f}".format(bases[i], exponents[i], res))
        print(30*"-")
    print("{:d}th fibonacci number is {:d}".format(6,
       ↪ example.fib(6)))
```

Listing 2.9: Test script for SWIG

2.5. Pybind11

Pybind11 is framework that provides interoperability between C++11 and Python. It is based on Boost.Python [Abrahams and Seefeld, 2002], which also aims to offer interoperability between C++ and Python. The framework generates a shared library which is imported as a Python module, similar to the other solutions [Jakob, 2017].

To create a module with Pybind11 the library must be compiled and linked with a C++ compiler. Although C and C++ share many similarities C is no subset of C++ [Stroustrup, 2021]. This also means a shared library written in C is not compatible with Pybind11. The library must be

compiled using a C++ compiler and one can not be sure their valid C code produces the same results when compiled with a C++ compiler or wheter it can be compiled at all.

The functions that shall be exposed through Pybind11 must be defined in a C++ source file. For the example library such a source file can be seen in Listing 2.10. When compiling the module this file needs to be passed to the compiler. Pybind11 offers some headers which also need to be included. This is achieved by calling the module with the includes flag `python3 -m pybind11 -includes`. The headers for Python itself need to be included as well via the `-I` flag (here Python 3.8 will be used again). The shared library which shall be called by the module also needs to be provided. For this the flag `-lexample` is provided (see Listing 2.11).

```cpp
#include <pybind11/pybind11.h>
#include "example.h"

PYBIND11_MODULE(pybind_libexample, module)
{
    module.doc() = "pybind11 example plugin";
    module.def("c_power_verbose", &power_verbose, "A function
       ↪ that raises the first parameter to the power of the
       ↪ second");
}
```

Listing 2.10: Source file for Pybind11 defining which functions to expose in Python module (`pybind_wrapper.cpp`)

```
$ g++ -shared -std=c++11 -fPIC `python3 -m pybind11 --includes`
   ↪ -I/usr/include/python3.8 -I. pybind_wrapper.cpp -o
   ↪ pybind_libexample`python3-config --extension-suffix` -L.
   ↪ -lexample -Wl,-rpath,.
```

Listing 2.11: Compile and link Pybind11 Module

The module can simply be imported with the name provided in the compile step. Functions can then be called with the name provided in the C++ source file (`pybind_wrapper.cpp`). The usage of the example library with a Python module created by Pybind11 can be seen in Listing 2.12.

```python
import pybind_libexample

if __name__ == "__main__":
    bases = [ 1.2, 58.2, 2.0 ]
    exponents = [ 2, 0, -2 ]
    for i in range(3):
        res = pybind_libexample.c_power_verbose(bases[i],
           ↪ exponents[i])
        print("in python:\t{:.2f}^{:d} is
           ↪ {:.2f}".format(bases[i], exponents[i], res))
        print(30*"-")
```

Listing 2.12: Test script for Pybind11

Because this framework only offers support for C++ libraries it is not suitable for the scope of this thesis, as the HPC library, for which the interoperability will be evaluated, is written in C.

2.6. Ctypes

Ctypes is a library included with the Python runtime to call functions provided by shared libraries from within Python. Using this library a Python wrapper can be created which invokes the functions of said shared library [Python Software Foundation, 2001b].

With the use of this library no further compilation step is needed. The shared library is just loaded at runtime and the functions can be called (see Listing 2.13).

```
1  import ctypes
2  import os
3
4  if __name__ == "__main__":
5      c_lib = ctypes.CDLL(os.getcwd()+"/libexample.so")
6      c_lib.power_verbose.restype = ctypes.c_float
7      bases = [ 1.2, 58.2, 2.0 ]
8      exponents = [ 2, 0, -2 ]
9      for i in range(3):
10         res = c_lib.power_verbose(ctypes.c_float(bases[i]),
              ↪ exponents[i])
11         print("in python:\t{:.2f}^{:d} is
              ↪ {:.2f}".format(bases[i], exponents[i], res))
12         print(30*"-")
```

Listing 2.13: Test script for Ctypes

This approach has already been evaluated as a solution for providing Python interoperability with JULEA [Coym, 2019]. A limitation of this solution is that it is quite stiff and needs to be modified manually everytime the function signature in C changes. This high amount of maintenance makes it impractical to provide a Python API in this way as it means too much overhead.

2.7. Cython

Cython is a programming language aiming to be a superset of Python. Its goal is to achieve C-like performance with the code being mostly written in Python. Cython source code gets translated into optimized C/C++ code which is then compiled into a Python module [Behnel et al., 2009]. It is also possible to wrap a library written in C into a Python module using Cython. Infact CFFI is build upon Cython.

To wrap a C library using Cython the functions of the library need to be imported from the headers in the Cython code first (see Listing 2.14). The imported function is then called in a Python function which will be exposed in the Python module later on.

```
cdef extern from "example.h":
    float power_verbose(float base, int exponent)

def py_power_verbose(int_param, float_param):
    return power_verbose(int_param, float_param)
```

Listing 2.14: Cython source code for wrapper module (cython_libexample.pyx)

With all functions declared, the next step is to invoke the Cython translation tool. This will convert the Cython source code into C/C++ code. The call to the Cython tool can be seen in Listing 2.15. Here the parameter `-3` declares that the Cython code should be translated based on Python 3 syntax.

```
$ cython -3 cython_libexample.pyx -o cython_wrapper.c
```

Listing 2.15: Translate Cython code into C code

After generating the C source code using cython, the module needs to be compiled using GCC (see Listing 2.16). The parameters for the compiler call are pretty similar to the GCC calls in the previous sections. The flags `-shared` and `-fPIC` are used because we want to generate a shared library using position independent code. The header files for Python 3.8 need to be included via the `-I` flag. Finally the example library needs to be linked with `-lexample`.

```
$ gcc -shared -fPIC -I/usr/include/python3.8 -I. cython_wrapper.c
  ↪ -o cython_libexample`python3-config --extension-suffix` -L.
  ↪ -lexample -Wl,-rpath,.
```

Listing 2.16: Compile Cython module

With the module compiled it can be used in a Python script (see Listing 2.17).

```
import cython_libexample

if __name__ == "__main__":
    bases = [ 1.2, 58.2, 2.0 ]
    exponents = [ 2, 0, -2 ]
    for i in range(3):
        res = cython_libexample.py_power_verbose(bases[i],
          ↪ exponents[i])
        print("in python:\t{:.2f}^{:d} is
          ↪ {:.2f}".format(bases[i], exponents[i], res))
        print(30*"-")
```

Listing 2.17: Test script for Cython

2.8. Shroud

Shroud is a tool providing the user with the ability to generate Fortran and Python interfaces for libraries written in C or C++. Declarations for functions that should be wrapped are written in a YAML file processed by the Shroud executable [Lawrence Livermore National Laboratory, 2017].

Listing 2.18 shows the YAML file for our example library. It declares the name of the resulting Python module. Besides Python and Fortran, Shroud can also generate a C API for a library written in C++. In the `options` section of the file, the user can declare which code to generate. In our example the only option enabled is Python, as it is the language we are focused on. Lastly in the `declarations` section the functions which shall be provided through the wrapper are declared.

```yaml
library: shroud_libexample

cxx_header: example.h

options:
  wrap_python: True
  wrap_fortran: False
  wrap_c: False

declarations:
  - decl: float power(float base, int exponent)
  - decl: float power_verbose(float base, int exponent)
  - decl: void loop (float* result, float factor, int counter)
  - decl: int fib(int number)
```

Listing 2.18: Shroud input file for example library (`exmple.yaml`)

With the Shroud declarations in place, the Shroud binary can be invoked. The only two parameters necessary are the YAML file and the output directory for Python (see Listing 2.19).

```
$ shroud example.yaml --outdir-python .
```

Listing 2.19: Generating Python module using Shroud

After generating the module, it needs to be installed by calling the `setup.py` script (see Listing 2.20).

```
$ python setup.py install
```

Listing 2.20: Installing Python module generated by Shroud

Afterwards the module should be simple to import and use (see Listing 2.21). When testing, however, the module build by shroud could not be imported properly due to some undefined symbol (see Listing 2.22). This error can surely be fixed but this would take some time. Because of the other approaches already yielding results there was no more effort put into debugging this approach.

```python
import shroud_libexample

if __name__ == "__main__":
    bases = [ 1.2, 58.2, 2.0 ]
    exponents = [ 2, 0, -2 ]
    for i in range(3):
```

```
7         res = shroud_libexample.power_verbose(bases[i],
              ↪ exponents[i])
8         print("in python:\t{:.2f}^{:d} is
              ↪ {:.2f}".format(bases[i], exponents[i], res))
9         print(30*"-")
10    print("{:d}th fibonacci number is {:d}".format(6,
          ↪ shroud_libexample.fib(6)))
```

Listing 2.21: Test script for Shroud (shroud_test.py)

```
1 ImportError:
      ↪ /.../shroud_libexample.cpython-310-x86_64-linux-gnu.so:
      ↪ undefined symbol: _Z4loopPffi
```

Listing 2.22: Error when trying to import Python module created by Shroud

2.9. JULEA

JULEA is a storage framework tailored for HPC. It offers an I/O interface for HPC applications. The usage is pretty flexible, providing several clients for storing data and metadata [Kuhn, 2017]. It also supports several backends including POSIX, LevelDB and MongoDB.

The object client provides the user with the ability to store data and accessing it through arbitrary namespaces. Objects can be stored on a single server or distributed over multiple servers. Both of these use cases can be abstracted through JULEA.

The kv client enables the user to store metadata accessible through arbitrary namespaces like the object client. It offers the storage of key-value pairs in a simple manor.

The item client offers an interface supporting collections and items. The collections represent an entity that can contain items. They offer somewhat of a hierarchy, even though it is a flat one. Items can be distributed over multiple data servers using JULEA's distributions similiar to the object client.

Chapter 3.

Design and Implementation

In this chapter, the implementation process of Python language bindings for JULEA will be described. The framework used will be CFFI and the design process, different approaches and their possible pitfalls will be explored.

The framework of choice to be evaluated in this chapter will solely be CFFI, since it seems the most promising for the use case and yielded fast results. Other frameworks that could be viable solutions like SWIG or Shroud described in Chapter 2 are not investigated further as doing so would exceed this chapters purpose.

A first naïve approch to generating a JULEA Python module using CFFI is reading a JULEA client header into CFFI and compiling the module. An example of the code one might write for this can be seen in Listing 3.1

```
1  import cffi
2
3  ffi = cffi.FFI()
4  with open("julea-kv.h") as file:
5      ffi.cdef(file.read())
6  ffi.set_source(
7          "julea_kv",
8          '#include "include/julea-kv.h"',
9          libraries=["julea-kv", "julea"],
10         library_dirs=["bld"],
11         extra_link_args=["-Wl,-rpath,."]
12 )
13 ffi.compile()
```

Listing 3.1: Generating Python module for Julea by simply reading in `julea-kv.h`

Unfortunatly this does not work, as CFFI does not support the usage of compiler directives [Rigo and Fijalkowski, 2012]. The call in line 5 will fail with an error, as `julea-kv.h` contains only compiler directives like `#include` or `#define` which can not be processed by CFFI directly (see Listing 3.2). This is the reason some preprocessing of the header files is needed to generate the module.

```
1  #ifndef JULEA_KV_H
2  #define JULEA_KV_H
3
4  #include <kv/jkv.h>
```

```
5  #include <kv/jkv-iterator.h>
6  #include <kv/jkv-uri.h>
7
8  #undef JULEA_KV_H
9
10 #endif
```

Listing 3.2: Excerpt of JULEA Key Value client header (`julea-kv.h`)

3.1. Preprocess using Python

Just ignoring all the directives leaves one with an empty library because the root headers for the clients, as mentioned ealier, include other header files which contain the actual declaration of functions and structs. To stick with our example of the `julea-kv.h` header file the declarations are located in the `kv` subdirectory.

Preprocessing these headers using Python boils down to performing some string matching and generating an aggregated header file containing only declarations and no compiler directives. In Listing 3.3 an example for such an aggregation script can be observed.

```
1  import cffi
2  import header_preprocessor as hp
3
4  ffi = cffi.FFI()
5
6  def collect_julea(filename):
7      includes = hp.get_additional_compiler_flags(["julea",
          ↪ "julea-object", "julea-kv", "julea-db"])
8      dirs = hp.get_include_dirs(includes)
9      output = hp.read_header_file("~/julea/include/julea-kv.h",
          ↪ list(filter(lambda entry: not "dependencies" in
          ↪ entry,dirs)))
10     content = ""
11     for line in output.split('\n'):
12         if "G_DEFINE_AUTOPTR_CLEANUP_FUNC" in line:
13             continue
14         content += line
15     with open(filename, "w") as file:
16         file.write(content)
17
18 def build(filename):
19     with open(filename, "r") as file:
20         header_content = file.read()
21     includes = hp.get_additional_compiler_flags(["glib-2.0",
          ↪ "julea", "julea-object", "julea-kv", "julea-db"])
22     include_dirs = hp.get_include_dirs(includes)
23     constant_defs = """
24                     typedef char... gchar;
```

```
25                     """
26     ffi.cdef(constant_defs+header_content, override=True)
27     ffi.set_source(
28             "julea_kv",
29             """
30                 #include "julea-kv.h"
31                 #include "julea.h"
32             """,
33             libraries=["julea-kv", "julea"],
34             include_dirs=include_dirs,
35             library_dirs=["bld"],
36             extra_compile_args=includes,
37             extra_link_args=["-Wl,-rpath,."]
38             )
39     ffi.compile(verbose=True)
40
41 if __name__ == "__main__":
42     filename = "test-header.h"
43     collect_julea(filename)
44     build(filename)
```

Listing 3.3: Preprocessing the header files using Python

After importing modules the first step in this script, is generating an instance of the `FFI` class. It is part of the CFFI module and will be used to set the source for the JULEA module and also to compile the module as described in Chapter 2.3.

The function `collect_julea` is the one responsible for generating the aggregated header file. In line 7 of Listing 3.3 the helper function `get_additional_compiler_flags` of the module `header_preprocessor` is called. This method gets the compiler flags needed for compiling and linking a C library or executable which uses the JULEA shared library. To achieve this `pkg-config` is called and the relevant libraries are passed as parameters (see lines 1-2 of Listing 3.4). The output is then stripped of any duplicate entries and the flag `-fsanitize` is removed (see lines 4-8 of Listing 3.4). The reason for this will be explained later in this chapter.

```
1 def get_additional_compiler_flags(libraries,
    ↪ remove_sanitize=True):
2     flags_buffer = os.popen("pkg-config --cflags
        ↪ {libs}".format(libs=' '.join(libraries)))
3     flags = flags_buffer.read().strip().split(' ')
4     flags = [*set(flags)]
5     if remove_sanitize:
6         for s in flags:
7             if "-fsanitize" in s:
8                 flags.remove(s)
9     return flags
```

Listing 3.4: Definition of `get_additional_compiler_flags` (excerpt of `header_preprocessor.py`)

The output of get_additional_compiler_flags is then passed to another function of the preprocessor module called get_include_dirs (see line 8 of Listing 3.3) which extracts the include directories from the compiler flags by removing the -I in front of the directory names (see Listing 3.5).

```
1 def get_include_dirs(flags):
2     return [ str.strip("-I") for str in flags if "-I" in str ]
```

Listing 3.5: Definition of get_include_dirs (excerpt of header_preprocessor.py)

The function read_header_file is then used to read in the key-value client header file and replace all the includes with the respective file content (see line 9 of Listing 3.3). This function does some setup and then calls an internal function which does the preprocessing (see Listing 3.6).

```
1 def read_header_file(path, include_dirs=[]):
2     include_dirs.insert(0, ".") # search first in current
       ↪ directory
3     filename = get_filename_from_path(path)
4     included_files = [ filename ]
5     return read_header_file_internal(path, include_dirs,
       ↪ included_files)
```

Listing 3.6: Definition of read_header_file (excerpt of header_preprocessor.py)

The function read_header_file_internal composes the header content line by line. In order to achieve this, it opens the file which path is passed through the parameter path. It then iterates over all lines of the file (see lines 5-6 of Listing 3.9). Each line is checked whether it is an include directive or not using the function is_include (see line 9 of Listing 3.9). It returns a tuple of a boolean and a string. The boolean indicates whether the line contains an include directive and the string contains the filename of the file to be included. If the line contains no include directive, the string is equal to None. Include directives are identified using a regular expression which also extracts the filename (see Listing 3.7).

```
1 def is_include(line):
2     ex = r"#include\s+(<(([\w-]+\/)*[\w-]+\.h)>" + r"|" +
       ↪ r"\"(([\w-]+\/)*[\w-]+\.h)\")"
3     match_result = re.search(ex, line)
4     if match_result == None:
5         return False, ""
6     filename = match_result.group(2)
7     if filename != None:
8         return True, filename
9     filename = match_result.group(4)
10    if filename != None:
11        return True, filename
12    # this should never need to be called
13    return False, ""
```

Listing 3.7: Definition of is_include (excerpt of header_preprocessor.py)

If the line contains an include directive the next step is to check if the file was already included. Therefore the array `included_files` stores all filenames which were already processed (see lines 7-8 of Listing 3.9). This needs to be done to prevent an infinite loop due to circular references between header files as we can not rely on include guards.

In the case the file has not been included yet, the function iterates over the array of include directories and searches for the file in each of the directories (see lines 9-15 of Listing 3.9). As soon as the file is found the filename is added to the array of included files (see line 12 of Listing 3.9). Then `read_header_file_internal` is called recursively with the path of the newly found file and both the `include_dirs` array and the `included_files` array. The result of this function call is then appended to the result of the current function call (see line 13 ofListing 3.9). After appending the string result, the processing of this line is finisched and the loop continues to the next line.

If the line does not contain an include directive, the next step is to check whether it is another kind of precompiler directive, like the definition of a macro or an include guard (see line 16 of Listing 3.9). This check is achieved by calling the function `is_precompiler_directive`. It simply checks if the line starts with a number sign (#) as these are at the beginning of every precompiler directive (see Listing 3.8). Lines like these are just skipped.

```
1 def is_precompiler_directive(line):
2     return line.startswith('#')
```

Listing 3.8: Definition of `is_precompiler_directive` (excerpt of `header_preprocessor.py`)

```
1  def read_header_file_internal(path, include_dirs=[],
     ↪ included_files=[]):
2      content = ""
3      with open(path) as header:
4          for line in header:
5              is_include_line, filename = is_include(line)
6              if is_include_line:
7                  if filename in included_files:
8                      continue
9                  for directory in include_dirs:
10                     path = os.path.join(directory, filename)
11                     if os.path.exists(path):
12                         included_files.append(filename)
13                         content +=
                               ↪ read_header_file_internal(path,
                               ↪ include_dirs, included_files)
14                         break;
15                 continue
16             if is_precompiler_directive(line):
17                 continue
18             content += line
19         return content
```

Listing 3.9: Definition of `read_header_file_internal` (excerpt of `header_preprocessor.py`)

If none of the both previous checks are successful, the line is appended to the function result (see line 18 of Listing 3.9). After iterating over all lines and files the aggregated content is returned as a string (see line 19 of Listing 3.9).

After this call to `read_header_file` the program flow continues in `collect_julea`. In lines 11 to 14 of Listing 3.3 the content is stripped of any lines containing the `G_DEFINE_AUTOPTR_CLEANUP_FUNC` macro, because it can not be handled by CFFI. This means the call to `unref` needs to be done manually for all JULEA structs. This will be explained in more detail in Chapter 5. At last the aggregated header content is written to a file specified by the `filename` parameter of the function (see line 16 of Listing 3.3).

With the aggregated header file created, the build process with CFFI can begin with the call of `build` (see line 44 of Listing 3.3). The header file is read into memory (see lines 19-20 of Listing 3.3). In the lines 21 to 25 the parameters for CFFI are gathered in order for it to build the module. In lines 26 to 38 the parameters are set, similar to the way described in Chapter 2.3. Finally the compile step is called, which generates the C code to interact with the JULEA library and then compiling it into a shared object file to be used with Python.

Although this approch theorectically works, it has proven to be very tedious as there are many special cases which need to be dealt with. Macros need to be ignored and as this routine relies on string matching new macros must be added manually. This might also cause new problems leading to more maintenance over head which should be avoided by relying on CFFI. To conclude, with the aim of developing a more resilient build process it is best to use a tool already built to preprocess source files for C.

3.2. Preprocess using GCC

GCC offers functionality to output the result of preprocessing. For this the parameter `-E` needs to be passed. Regarding this option the GCC manual says:

> Stop after the preprocessing stage; do not run the compiler proper. The output is in the form of preprocessed source code, which is sent to the standard output.
>
> Input files that don't require preprocessing are ignored.
> [Free Software Foundation, Inc., 1988]

This option looks like a good solution to the current problem. Listing 3.10 shows the first iteration of a build script using the GCC preprocessor. The header to be processed is the key-value client header, the same one used in the previous section.

```
1 import cffi
2 import os
3
4 ffi = cffi.FFI()
5 os.system("gcc -E -P ~/julea/include/julea-kv.h $(pkg-config
    ↪ --cflags glib-2.0 julea julea-object julea-kv julea-db) -o
    ↪ header.h")
6 with open("header.h") as header:
7     header_content = header.read()
8 ffi.cdef(header_content)
```

```
ffi.set_source(
        "julea_kv",
        '#include "include/julea-kv.h"',
        libraries=["julea-kv", "julea"],
        library_dirs=["bld"],
        extra_link_args=["-Wl,-rpath,."]
)
ffi.compile(verbose=True)
```

Listing 3.10: Simple build script for CFFI module using GCC preprocessor

Using the `system` function of the `os` Python module, a shell command can be invoked. In the build script GCC is invoked with the flag for preprocessing. In addition `pkg-config` is used to get the include flags for C needed for JULEA. The name of the output file is set to `header.h` (see line 5 of Listing 3.10). To prevent GCC from bloating the file with linemarkers, the flag `-P` is passed [Free Software Foundation, Inc., 1988]. These line markers indicate from which file a line of the preprocessor result originated. As this information is not relevant for CFFI, GCC is inhibited from generating the linemarkers.

The file generated by GCC is then read into memory and passed to CFFI (see lines 6-8 of Listing 3.10). After setting the definitions, the remaining build parameters for CFFI are set and the `compile` method is invoked (see lines 9-16 of Listing 3.10).

This simple invokation of GCC and CFFI does not produce a functional Python module. The main problem are macros and static inline functions which are not supported by CFFI. These could be processed using regular expression and then removed from the header file, but there is a simpler way to get rid of these definitions.

To begin with, all of the inline functions are defined in dependencies of JULEA. Of those dependencies only a small amount of type definitions are acutally needed. The rest of the dependency library functions and types can be ignored, as it is not necessary to interface with JULEA. To specify, the dependency libraries causing problems are glib and bson. To prevent these headers from being included without modifying the JULEA files itself, a stub file is created in the current directory for every file which is otherwise included by the JULEA headers . This means an empty file is generated which has the same name as the one of the provided by the dependency (see lines 38-46 of Listing 3.11). If we add the current directory as an include directory (see line 47 of Listing 3.11), GCC includes the empty files and not the dependency library headers. The required types, which are now missing, are defined manually in the build script (see lines 1-36 of Listing 3.11).

```
content = """typedef int gint;
typedef unsigned int guint;
typedef gint gboolean;
typedef char gchar;
typedef unsigned short guint16;
typedef signed int gint32;
typedef unsigned int guint32;
typedef signed long gint64;
typedef unsigned long guint64;
typedef void* gpointer;
typedef const void *gconstpointer;
```

```
12 typedef unsigned long gsize;
13 typedef guint32 GQuark;
14 typedef struct _GError GError;
15 struct _GError
16 {
17     GQuark   domain;
18     gint     code;
19     gchar   *message;
20 };
21 typedef struct _GModule GModule;
22 typedef struct _GInputStream GInputStream;
23 typedef struct _GOutputStream GOutputStream;
24 typedef struct _GKeyFile GKeyFile;
25 typedef struct _GSocketConnection GSocketConnection;
26 typedef void (*GDestroyNotify) (gpointer data);
27 typedef struct _bson_t
28 {
29     uint32_t    flags;
30     uint32_t    len;
31     uint8_t     padding[120];
32 } bson_t;
33 #include <julea-kv.h>
34 """
35 with open("temp.h", "w") as file:
36     file.write(content)
37
38 with open("glib.h", "w") as file:
39         file.write("")
40 with open("gmodule.h", "w") as file:
41     file.write("")
42 with open("bson.h", "w") as file:
43     file.write("")
44 system("mkdir -p gio")
45 with open("gio/gio.h", "w") as file:
46     file.write("")
47 os.system("gcc -E -P temp.h -I. $(pkg-config --cflags glib-2.0
   ↪ julea julea-object julea-kv julea-db) -o header.h")
48 system("rm -rf glib.h gmodule.h bson.h gio temp.h")
```

Listing 3.11: Ignoring glib and bson headers (excerpt of the build script for CFFI module using GCC preprocessor)

After generating the preprocessed header, all temporary files are removed (see line 48 of Listing 3.11). The rest of the procedure is the same as before.

The focus will now be shifted to the macros defined by glib. For the purpose of interfacing with the JULEA shared library through a Python module, it is enough to just ignore these macros. This can be achieved by just defining them through the command line and letting them result in an empty string (see Listing 3.12).

```
macros = [
    "-D'G_DEFINE_AUTOPTR_CLEANUP_FUNC(x, y)=''",
    "-D'G_END_DECLS=''",
    "-D'G_BEGIN_DECLS=''",
    "-D'G_GNUC_WARN_UNUSED_RESULT=''",
    "-D'G_GNUC_PRINTF(x, y)=''"
]
system(f"gcc -E -P {' '.join(macros)} temp.h -I. $(pkg-config
  ↪ --cflags glib-2.0 julea julea-object julea-kv julea-db) -o
  ↪ header.h")
```

Listing 3.12: Ignoring glib macros (excerpt of the build script for CFFI module using GCC preprocessor)

The modified build script for the CFFI module now builds the module without errors. But when one tries to import the module an error occurs, stating the symbol `backend_info` is undefined (see Listing 3.13). This error occurs because the function `backend_info` is declared in `core/jbackend.h`, which is included by `julea.h`. But in the linking process no library which contains a definition for this function is provided. This can be solved by adding `kv-null` as another library referenced in the linking process (see line 4 of Listing 3.14). This function will never be called when working with the clients, but a definition needs to be provided, so that the module can be imported.

```
ImportError: /.../julea_wrapper.cpython-38-x86_64-linux-gnu.so:
  ↪ undefined symbol: backend_info
```

Listing 3.13: Import error of Python module (undefined `backen_info`)

```
ffi.set_source(
        "julea_kv",
        '#include "include/julea-kv.h"',
        libraries=["julea-kv", "julea", "kv-null"],
        library_dirs=["bld"],
        extra_link_args=["-Wl,-rpath,."]
)
```

Listing 3.14: Linking against `kv-null` (excerpt of the build script for CFFI module using GCC preprocessor)

This modification to the build script fixes the import error regarding JULEA's `backend_info`. Now, however, another error regarding loading a library for address sanitization arises (see Listing 3.15). The Address Sanitizer helps to detect memory errors in C and C++. It can detect memory leaks, buffer overflows, use after free and more [Serebryany et al., 2011]. To use this feature, the flag `-fsanitize=address` needs to be added to the compiler parameters. When compiling JULEA, the Meson option `-Db_sanitize=address,undefined` enables this feature. If the shared library of JULEA, which the Python module interacts with, is linked against the Address Sanitizer library, the Python module itself also needs to be linked against said Address Sanitizer library.

Even when passing the option through the set_source the import error still persists. Luckily it can be circumvented by compiling JULEA without adress sanitization like shown in Listing 3.16. When modifying C code, the address sanitizer helps to identify memory errors. As the Python module will only be interfacing with the library and no modifications of the C source code are required, disabling the sanitizer is no problem for this use case.

```
ASan runtime does not come first in initial library list; you
  ↪ should either link runtime to your application or manually
  ↪ preload it with LD\_PRELOAD.
```

Listing 3.15: Import error of Python module (ASan runtime not loaded)

```
$ meson setup --prefix="${HOME}/julea-install"
$ ninja -C bld
```

Listing 3.16: Compiling JULEA without address sanitization

With these modifications to the build script, CFFI can compile a module that is able to in interact with the JULEA shared library. After creating a working module for the key-value client, the next step is to build a module for the other clients.

The process does not deviate much from the creation of the kv module, because only the name of libraries which shall be included need to be changed. In the process of validating the results using a hello-world script which can be seen in Listing A.4, a problem with the struct JBatch surfaces (see Listing 3.17). When building separate modules for each client, this struct is included in every module. But one module can not use the JBatch of another, hence making it impossible to, for example, bundle operations of a key-value client and an object client into the same batch. To avoid this problem all clients can be compiled into one single Python module.

```
TypeError: initializer for ctype 'struct JBatch *' appears indeed
  ↪ to be 'struct JBatch *', but the types are different (check
  ↪ that you are not e.g. mixing up different ffi instances)
```

Listing 3.17: Error using Python code with JBatch defined in different modules

With all these changes and tidying up the final code for building JULEA's Python module is displayed in Listing 3.18. Here all JULEA clients and the name of the JULEA library itself are passed to a function contained in a helper script, which abstracts the parameter preperation and building calls (see Listing A.6). The second function call copies a Python script in JULEA's build directory, containing some functionality which helps in interacting with the JULEA Python module. The contents of this script are discussed in the next section.

```python
from build_helper import build, copy_main_module

if __name__ == "__main__":
    build(["julea", "julea-object", "julea-kv", "julea-db",
      ↪ "julea-item"])
    copy_main_module()
```

Listing 3.18: Final build script for JULEA module built with CFFI (build.py)

3.3. Improvements for easier usage

As mentioned in the previous section there is a module containing some helper functionality copied to the build directory as part of the compiling process. This module is the one to be imported when using JULEA with Python and is simply called `julea`.

It provides the objects `lib` and `ffi`, which are the most important ones when interacting with JULEA's C Library (see line 1 of Listing 3.19). The `lib` object contains all client functions and constants defined within the C code. They have the same names and signatures as their C counterparts. The `ffi` object is used to initialize and cast C data types. Every pointer created using `ffi.new` is freed by CFFI automatically as soon as there are no more references to it. In line 6 of Listing 3.19 it is used to initialize an array of characters. The result of this function is a pointer which can later be passed to a function within `lib` that is expecting a pointer to a character as argument.

```
1  from julea_wrapper import lib, ffi
2
3  encoding = 'utf-8'
4
5  def encode(string):
6      result = ffi.new('char[]', string.encode(encoding))
7      return result
8
9  def read_from_buffer(buffer):
10     char = ffi.cast('char*', buffer)
11     bytearr = b''
12     i = 0
13     byte = char[i]
14     while byte != b'\x00':
15         bytearr += byte
16         i += 1
17         byte = char[i]
18     return bytearr.decode()
```

Listing 3.19: Functions and objects provided by JULEA module (excerpt of `julea.py`)

The two functions provided by this module convert between strings and string buffers. The encode function takes a string as argument and encodes it using the UTF-8 encoding (see lines 5-7 of Listing 3.19). This encoded string is saved into a character array and a pointer to this array is returned. When using this function the result **must** be saved into a variable. If this is not the case, the buffer will be freed as there are no more references to it, causing undefined behavior. The documentation of `ffi.new` reads:

> When the returned <cdata> object goes out of scope, the memory is freed. In other words the returned <cdata> object has ownership of the value of type `cdecl` that it points to. This means that the raw data can be used as long as this object is kept alive, but must not be used for a longer time. [Rigo and Fijalkowski, 2012]

The function `read_from_buffer` expects a pointer to a buffer as argument. This pointer is then cast into a character pointer and read into a Python byte array until a null byte is encountered.

This byte array is then decoded and returned. This result is a string (see lines 9-18 of Listing 3.19).

Besides the objects and functions `julea.py` also contains two classes. Their purpose is to simplify the use of batches with JULEA. The idea behind batches in JULEA is to aggregate calls to client functions that interact with the backend into a collection of instructions which can be executed in one run. Every function call that creates, reads, updates or deletes data takes a pointer to a `JBatch` struct as an argument. The operation is performed when that batch is executed. As it is necessary to pass a pointer to this struct every time an operation shall be executed, it makes sense to create a wrapper which simplifies the creation, execution and deletion of this struct.

`JBatchResult` encapsulates a boolean, so it can convey the return value of a call to `j_batch_execute` (see lines 1-11 of Listing 3.20). The second class, `JBatch`, abstracts the initialization, execution and unrefferencing of a `JBatch` struct.

When an instance of this class is created, a `JBatchResult` instance needs to be passed as argument, so the result of the execute call can later be examined. In addition to saving the reference to the result, a `JBatch` struct is initialized using the default semantics template (see lines 14-17 Listing 3.20). The other two methods correspond to the use of this class within a scope created using the `with` keyword in Python (see lines 19-31 of Listing 3.20). This ensures that at the end of this scope, the batch is executed and unreferenced afterwards. An example usage can be seen in the hello world example for JULEA written in Python (see lines 21-26 of Listing A.4).

```
1  class JBatchResult:
2      IsSuccess = False
3
4      def __init__(self):
5          pass
6
7      def Succeed(self):
8          self.IsSuccess = True
9
10     def Fail(self):
11         self.IsSuccess = False;
12
13 class JBatch:
14     def __init__(self, result):
15         template = lib.J_SEMANTICS_TEMPLATE_DEFAULT
16         self.batch = lib.j_batch_new_for_template(template)
17         self.result = result
18
19     def __enter__(self):
20         return self.batch
21
22     def __exit__(self, exc_type, exc_value, tb):
23         if lib.j_batch_execute(self.batch):
24             self.result.Succeed()
25         else:
```

```python
                self.result.Fail()
            lib.j_batch_unref(self.batch)
            if exc_type is not None:
                return False
            else:
                return True
```

Listing 3.20: Classes provided by JULEA module (excerpt of `julea.py`)

The explicit call to `j_batch_unref` is required because the auto pointer functionalty of glib is not present in the JULEA Python module. When unreferencing the structs manually, it is ensured that the memory will be freed correctly.

Such classes encapsulating the structs can also be written for other data types such as `JKV` or `JItem`. This makes the effort of maintaining the code higher, because these classes need to be kept up to date when the underlying structs change. In general, one must balance between usability and maintainability. A wrapper might be simpler to use and the interface behaves more like an actual Python class, but for every change in the underlying C Library these wrapper classes might need to be adjusted to function correctly. The approach with helper methods is less maintenance-intensive, as the methods they interface with are less prone to changes because they are less specific for the library.

In this case a wrapper class for the batches is chosen because the interface is not likely to be subject of large changes in the future. The other methods are helpful for passing string arguments between Python and C so they do not depend on future changes of the JULEA library. These choices are made to keep the effort of maintaining the Python module as low as possible, while also offering a little bit of easier usage.

Chapter 4.

Related Work

In this chapter, related work in the area of Python performance, HPC in Python and language bindings for other languages will be discussed.

4.1. HPC and Python

In HPC applications, like data science or machine learning, Python is quite popular due to its simplicity [Data Science Council of America, 2021][Hillier, 2022]. Because HPC applications often involve simulations studied by scientists, whose main field of studies is not necessarily computer science, it is good to have tools that are easy to understand and maintain [Whitfield, 2022].

The adoption of Python in the context of HPC is also conveyed by the existence of training lectures for HPC with Python, one example being held by Monte Lunacek of the University of Colorado Boulder [Lunacek, 2013]. In this lecture he focuses on multiprocessing taking a look at SCOOP and mpi4py.

In addition to this, there are multiple resources provided by universities explaining how to use Python on their cluster infrastructure for HPC, encouraging students to work with Python [The Trustees of Princeton University, 2023] [Riga Technical University, 2023] [Fullard and Wang, 2022].

4.2. Python performance

The impact on performance when using Python instead of C can be immense. In an HPC environment the difference in runtime might be more than 100 times when compared to pure C code. In a paper by Langtangen and Cai, they examined the efficiency of Python for HPC [Langtangen and Cai, 2008]. Their test problems involve computations used for solving partial differential equations. The Python code is compared with implementations in C and Fortran. On top of that, they explore optimizations with the use of vectorization and Python modules developed for scientific computing, like `numpy`, `numarray` and `Numeric`.

The advantage of using NumPy for intense computational work over pure Python is also highlighted in an article by Candido, where the performance of pure Python is compared to NumPy and TensorFlow [Candido, 2018]. The application used for testing is a gradient descent. In the results NumPy and TensorFlow were in the same range of runtime whereas pure Python ran one order of magnitude slower. A thing to note about the TensorFlow implementaion is

the fact that it was executed on the Central Processing Unit (CPU) and the runtime could be faster when leveraging execution power of a dedicated Graphics Processing Unit (GPU).

Besides using libraries written in compiled programming languages, another way of improving Python performance can be the usage of a different Python implementation. In most cases when talking about Python, what is referenced is the CPython implementation of the programming language, written in C. PyPy is an alternative implementation using a just-in-time compiler (JIT) [The PyPy Project, 2023]. In his master's thesis Roghult compared the performance of CPython, Cython, Jython and PyPy against each other [Roghult, 2016]. The problems used for testing the performance were dictionary usage, list iteration, tuple iteration, generators and the creation of a large collection of objects. The tests were run on two different machines, a Mac Pro with four cores and a Dell with two cores. On the Mac PyPy was the fastest in 10 out of 26 tests. It is notable that CPython was the fastest when performing dictionary operations on this machine. On the Dell machine PyPy was the fastest in 12 out of 26 tests, with PyPy being the fastest in one of the dictionary tests. Depending on the problem and hardware PyPy can be faster than the standard CPython implementation.

4.3. JULEA Python module

There has already been an attempt to generate a Python module which offers an interface for the JULEA library [Coym, 2019]. The module was generated using ctypes (see Chapter 2.6 for details on the usage of ctypes) which offers a more object oriented interface. As a drawback, the effort for maintaining the code and keeping it in line with JULEA's changes is quite high, as the interfacing classes are not generated automatically from the header files of JULEA.

The runtime increases of the Python version vary depending on the client and the operation. The increases lie between 7% and 223% while the impact on the object client seems to be greater than on the key-value client.

4.4. Language bindings for other languages

Python is not the only language with the need for language bindings. Calling functions of libraries written in another language enables the reuse of code, because functionality does not have to be rewritten in another language. It is important to note, however, that this reimplementation process has the potential to cause bugs, if not tested correctly.

Solutions like SWIG offer a wide variety of languages to be targeted [Fulton et al., 1999]. This has the potential to offer functionality written in C/C++ to many languages at once. A toolset similar to SWIG but limited to the .NET ecosystem is CppSharp [Matos et al., 2010].

Diplomat works quite alike [Goregaokar et al., 2021]. It is a tool set created to allow other languages to call code written in Rust. It has support for C/C++, C# and JavaScript. For Rust itself there is a tool called Bindgen which can generate interfaces from C and C++ headers to be used in Rust [Álvarez et al., 2012].

Chapter 5.

Evaluation

In this chapter, the usability of the Python module created as part of this thesis will be examined. Additionally the performance of the Python code will be compared to the pure C code.

5.1. Usage of the module

To use the JULEA module with Python it needs to be imported using the command `import julea` (see line 1 of Listing A.4). As described in Chapter 3.3 it exposes the classes for easier use of the `JBatch` struct, the library functions, the ffi interface and two helper functions to deal with the conversion of strings between C and Python.

The first step after importing the module is to set up the arguments for the function calls. As an example we will look at the usage of the key value client within Listing A.4. The parameters needed are the `JBatchResult`, namespace, key and the value to be written (see Listing 5.1).

```
1 value_kv = encode("Hello Key-Value!")
2 ...
3 result = JBatchResult()
4 hello = encode("hello")
5 world = encode("world")
6 ...
7 kv = lib.j_kv_new(hello, world)
```

Listing 5.1: Setting up arguments for key value client (excerpt of Listing A.4)

```
1 with JBatch(result) as batch:
2     ...
3     lib.j_kv_put(kv, value_kv, len(value_kv), ffi.NULL, batch)
```

Listing 5.2: Adding put operation to the batch and executing it (excerpt of Listing A.4)

As a next step the put operation can be added to the batch and executed. To do this a scope is created using the `with` statement. In this scope all operations are added to the batch and then executed when the scope is left (see Listing 5.2). The arguments passed into the function are `kv` as identifier, `value_kv` as the encoded value to be written, the length of this value and the batch. The argument passed for `value_destroy` is `ffi.NULL`, because the value will be freed by CFFI, so no function needs to be passed which frees the memory to which `value_kv` points. To

check whether the batch was executed successfully, `result.IsSuccess` needs to be evaluated (see line 28 of Listing A.4).

After storing the value, we are able to read it. In order to be able to do this, the arguments for the JULEA client function need to be prepared first. Two pointers are required, a pointer that references a `void` pointer and one that references an `unsigned int`. As we can not get the addresses of a Python variable using the ampersand symbol, we create the pointers using `ffi.new`. The identifier is the same as created earlier for the put operation. A new `JBatchResult` is also created for the new batch (see Listing 5.3).

```
1 result = JBatchResult()
2 with JBatch(result) as batch:
3     buffer_ptr = ffi.new("void**")
4     length = ffi.new("unsigned int *")
5     lib.j_kv_get(kv, buffer_ptr, length, batch)
```

Listing 5.3: Getting value with key value client (excerpt of Listing A.4)

If the batch execution is successful, the value is saved inside the buffer and `length` points to the length of the read data. To dereference the pointers, they need to be indexed like the first value of an array. When reading from the buffer, it is cast into a `char**` using the function `ffi.cast`. The `char*`, `char_buff_ptr` references, is passed into the function `read_from_buffer` to get the string. The length is read by dereferencing the pointer (see Listing 5.4).

```
1 char_buff_ptr = ffi.cast("char**", buffer_ptr)
2 print(f"KV contains: '{read_from_buffer(char_buff_ptr[0])}'
   ↪ ({length[0]} bytes)")
```

Listing 5.4: Reading the get result from a buffer (excerpt of Listing A.4)

The value can be deleted by calling the `lib.j_kv_delete` function and passing `kv` and `batch` as arguments (see Listing 5.5).

```
1 with JBatch(result) as batch:
2     lib.j_kv_delete(kv, batch)
```

Listing 5.5: Deleting the stored value (excerpt of Listing A.4)

At the end of the script `kv` needs to be unreferenced to avoid memory leaks. This is achieved by calling the function `lib.j_kv_unref` and passing `kv` as argument (see line 69 of Listing A.4). For the batches this call is not necessary because, they are unreferenced at the end of the scope as described in Chapter 3.3.

5.2. Performance benchmarks

The benchmarks were performed on a node of the HPC cluster of the Otto-von-Guericke University in Magdeburg. The hardware specification of the node can be seen in Table 5.1. The runs were performed 10 times. The first five times the Python benchmark was executed before the C benchmark. The other five times the order of execution was inverted, so the C code was run before the Python code. All following diagrams display the average operations per second

Processor	AMD EPYC^{TM} 7443 @ 2.85 GHz
Cores	24
Threads	48
L1 Cache	64 KiB (per core)
L2 Cache	512 KiB (per core)
L3 Cache	128 MiB
Memory	128 GiB

Table 5.1.: Hardware specifications of computing node used for benchmarking

and the standard deviation over these 10 runs. The measurements for each run can be found in Chapter B.

Benchmark jobs were scheduled using the Slurm Workload Manager used to manage workloads between the nodes of the cluster [SchedMD, 2022]. The job scripts can be seen in Listing A.7 and Listing A.8.

The file operations were perfomed on Tmpfs, a file system which keeps the files in virtual memory, to focus on the execution time of the operations and avoid being heavily influenced by delays caused by the file system [Rohland et al., 2001].

The version of GCC used to build JULEA and the Python module was 12.2.0. The version of Python used to build the module and run the benchmarks was 3.8.12. As for Python modules used in the build process, CFFI was used in version 1.15.1 and pycparser was used on version 2.21. The benchmarks implemented in Python were programmed to closely resemble the benchmarks written in C. Most of the operations have a *normal* and *batch* implementation for benchmarking. The used operation is the same, but when executing the batch variant all operations are collected into one batch which is executed, opposed to executing every operation in a single batch with the normal version. In tables and figures the batch verion is indicated by the suffix *-batch*. The metric used for measuring performance is operations per seconds.

5.2.1. Key Value

The first client to be examined is the key value client. The backend used for benchmarking this client was Lightning Memory-Mapped Database (LMDB). Results of the key value client benchmarks are displayed in Figure 5.1.

The Python runs achieve less operations per second than the C runs. Generally the Python performance of the batch versions seem to be performing closer to the C version than their normal counter parts. The decrease in performance is about 5 percentage points less for the batch operations.

An exception is the *put* operation which, according to the averages, has the smallest performance decrease with just 12%. But when looking at the results for the C runs of the *put* operation, the large standard deviation of the values stands out. When looking at the *put-batch* performance, the actual performance decrease for *put* is probably closer to 20%.

The other operations have a performance decrease ranging from 22% to 26% for the normal version and from 15% to 22% for the batch version. The *get-batch* operation has the smallest

performance decrease, with 15% (ignoring *put*), and the *delete* operation has the largest decrease in performance, with 26%, when comparing Python and C.

When comparing the results of these benchmarks with the ones of the previous Python wrapper module for JULEA, they seem to perform similarly [Coym, 2019]. The performance impact for the normal variants of the CFFI implemantaion seems to be less significant than the impact for the ctypes implementation. However, the ctypes implementation benefits more from the usage of batches, and hence the performance decrease for the batch variants of the operations is smaller with the ctypes implementation. These performance differences might originate from the different backends used for testing, with LMDB used for the CFFI implementation and LevelDB used for the ctypes implementation. The ctypes implementation was also benchmarked on the standard Linux file system and the system had significantly less system memory.

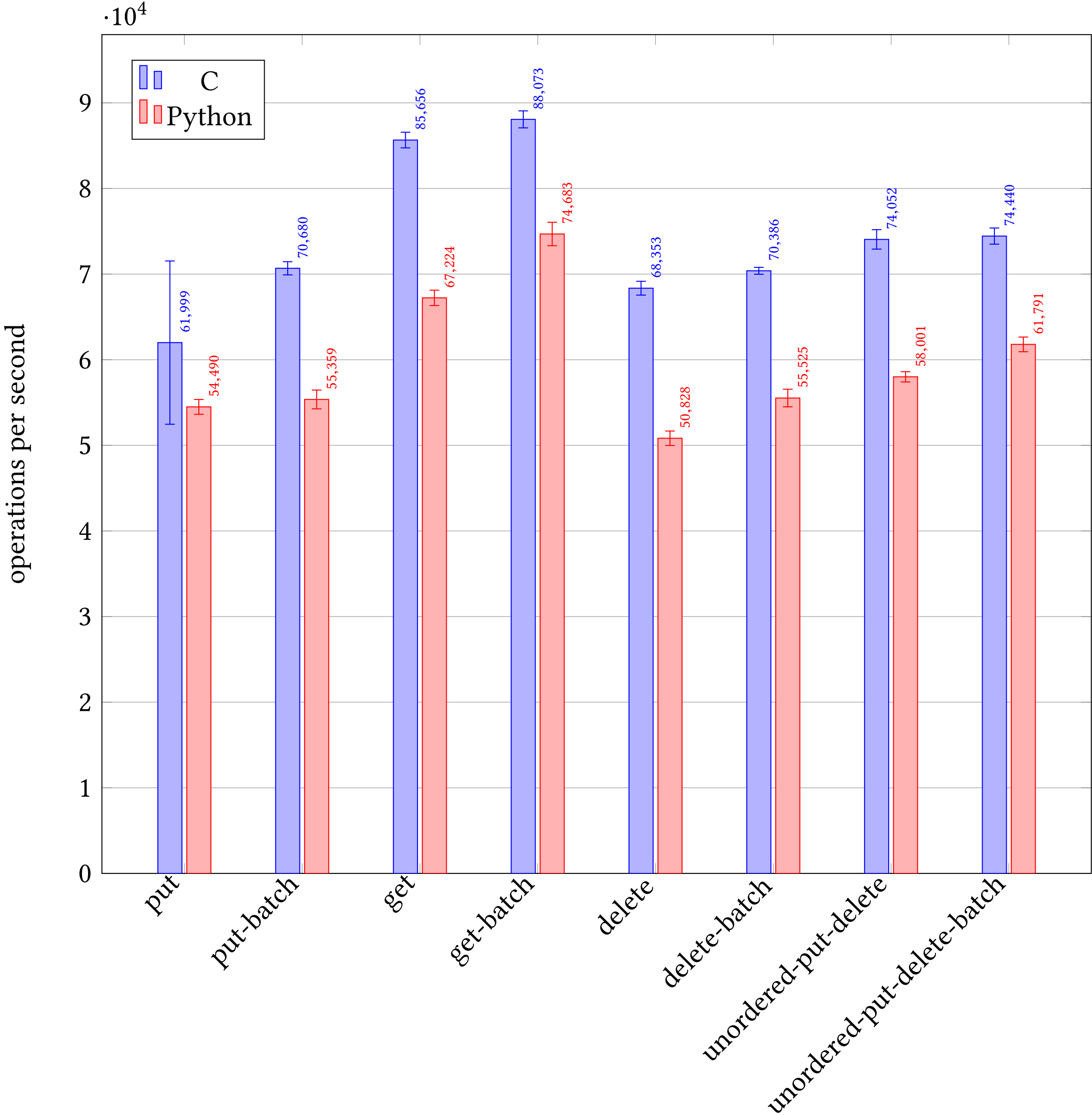

Figure 5.1.: Key-Value Client Performance

5.2.2. Object

The next two clients, distributed object and object client, both use the Portable Operating System Interface (POSIX) as backend. The data is stored as files in the file system.

The distributed object client distributes the data among multiple servers, as the name suggests. The results of the distributed object client benchmarks are displayed in Figure 5.2.

When comparing Python against C, Python achieves less operations per seconds than C but the decrease is smaller than the performance decrease of the key value client, ranging from 4% to 24%. Similar to the key value benchmarks, the benchmarks for the distributed object client perform better in their batch variant compared to their normal variant, with the batch variant yielding a performance decrease which is about 4 percent points smaller than the decrease of the normal variant.

Outliers to this are the *read* and the *write* operation. For both the *read* and *read-batch* benchmark the Python implementation has a performance decrease of 7%. The *write* operation, however, is another story. For the *write* benchmark runs the decrease in performance is 11%, whereas the decrease for the *write-batch* runs lies at 24%. Although the standard deviation of the *write-batch* runs is larger than the standard deviation of the other operations both for the C and Python implementation, this can not be the only reason for this huge gap. As the code for C and Python has the same control flow and the normal write runs lie in the range of the other operations, there seems to be no obvious explanation for this anomaly.

The *status* operation performes better in the normal variant compared to the batch variant as well, with the normal one having a performance decrease 3 percentage points lower than the batch variant.

When the results of these benchmarks are compared to the ctypes implementation of the previous attempt of providing a Python module for JULEA, the outcome looks alike [Coym, 2019]. Both the *status* and *write* methods have a larger performance decrease when executed in bulk. Like the results of the key value client, the ctypes implementation benefits more from the batch execution, but this time the CFFI implementation outperformes it in every benchmark.

The object client stores the data on only one server. The results of the object client benchmarks are displayed in Figure 5.3.

The performance of the object client is pretty similar to the distributed object client. The decrease in performance for the Python implementation ranges from 4% to 26%. Like the other two clients, the perfomance of the object client in Python is mostly closer to the C implementation for the batch variants of the benchmarks. The performance decrease is about 4 percentage points smaller for the batch variants.

For the *write* and *status* operations the object client behaves like the distributed object client. The performance decrease for the batch variant of the *status* operation is also 3 percentage points larger than for the normal variant. The difference between *write* and *write-batch* is 14 percentage points, which is 1 percentage point more than with the distributed object client.

Compared to the ctypes implementation, it is out performed by the CFFI implementation, with the ctypes implementation yielding a performance decrease of up to 68% [Coym, 2019]. Other than the distributed object client, the object client of the ctypes implementation does not share the same behavior for *status* and *write* with regard to the batch variant. Overall the ctypes implementation still benefits more from the operations being executed in one batch.

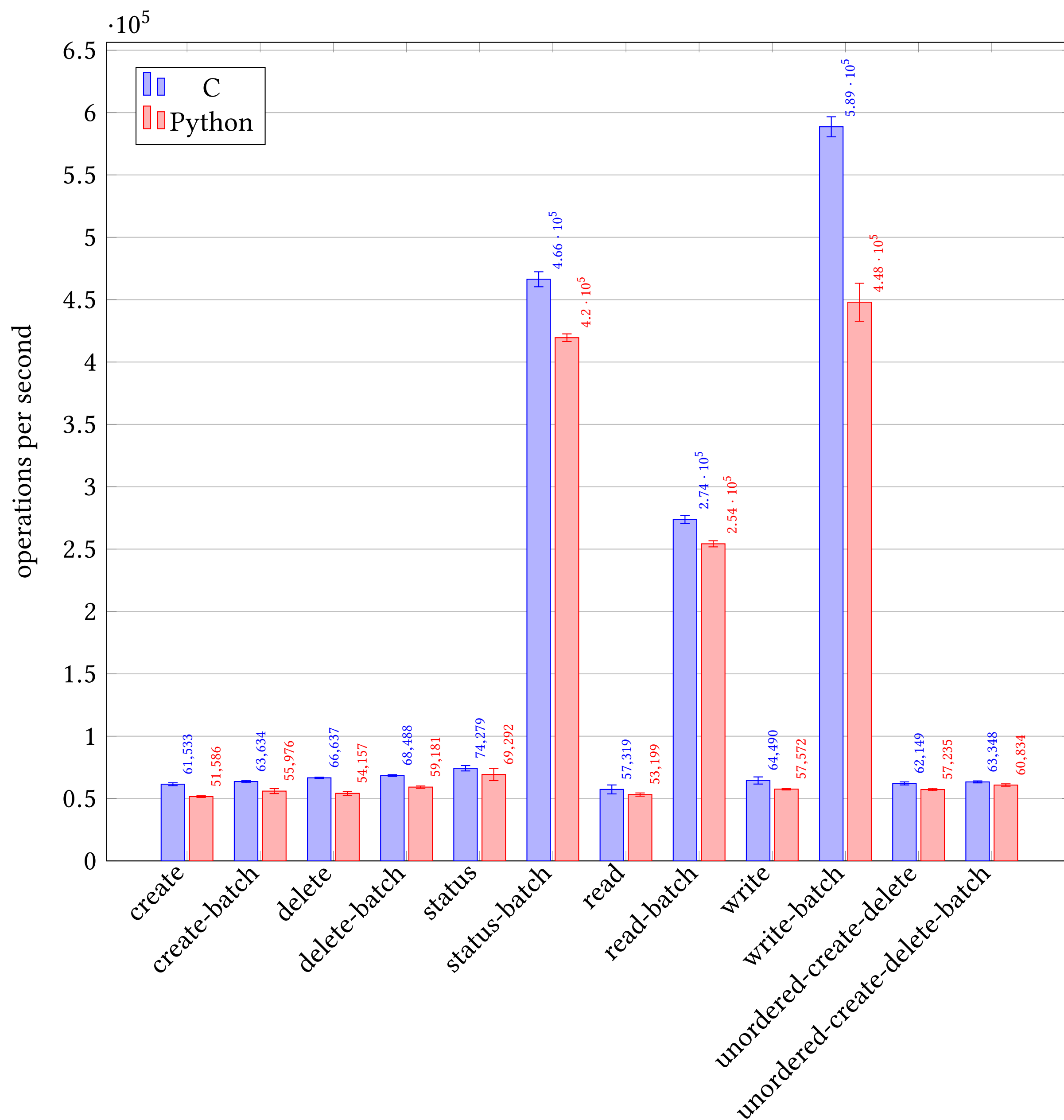

Figure 5.2.: Distributed Object Client Performance

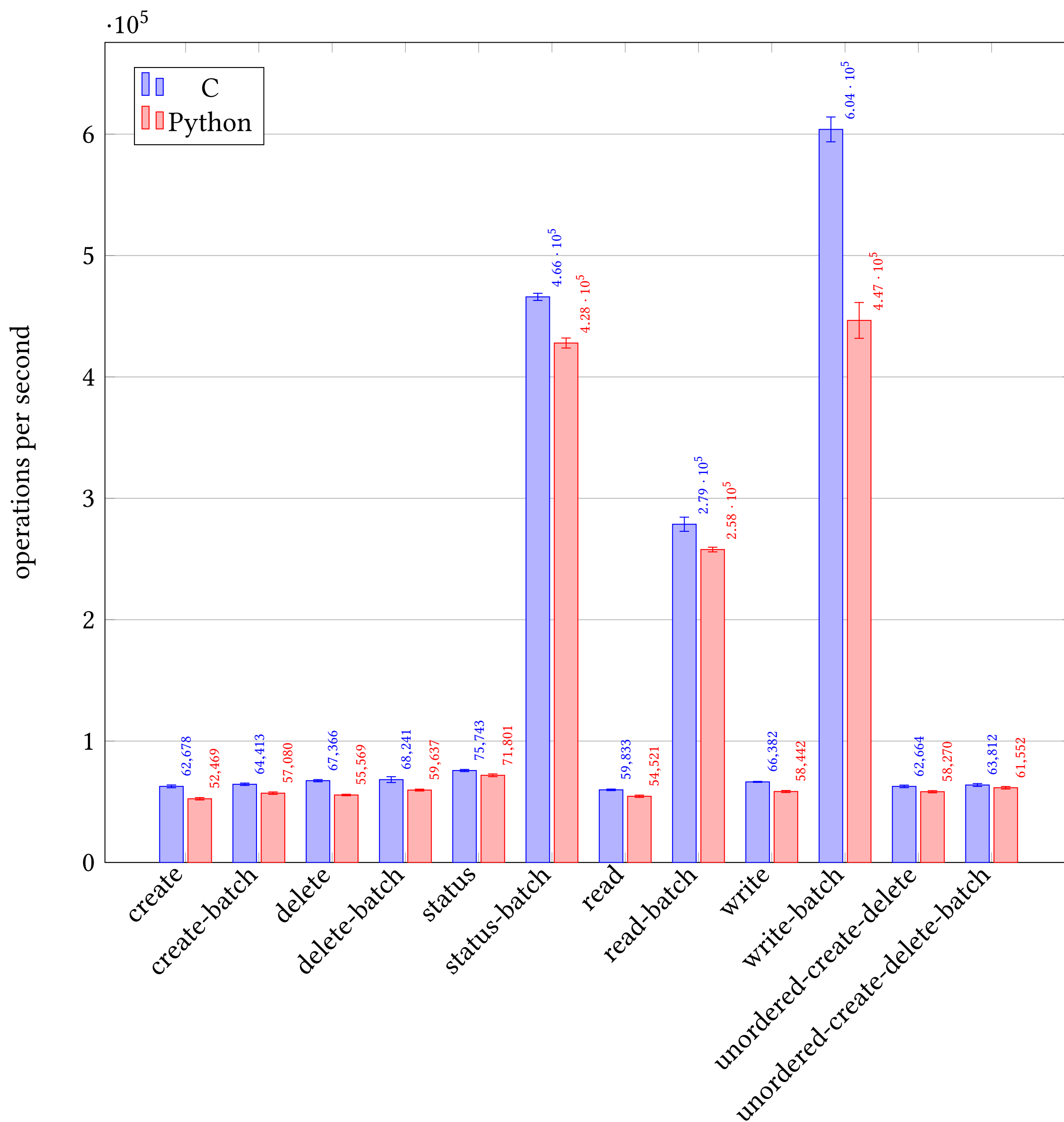

Figure 5.3.: Object Client Performance

5.2.3. Item

The item client uses the object and key value client internally, so the backends used for benchmarking are the same with LMDB used for key value and POSIX used for object. Its benchmark results are displayed in Figure 5.4.

The first thing to point out is the missing data for *delete-batch-without-get* for the Python implementation. This benchmark did not finish for any of the 10 benchmark runs. This is most likely due to an error in the implementation of the Python benchmarks.

Next, when we compare the performance of the Python implementation against the C implementation, the decrease in performance ranges from 5% to 27%. It is similar to the performance of the other clients.

The performance decrease for *get-status*, *read* and *write* is smaller than their batch counterparts, with a decrease ranging from 5 to 15 percentage points less than the batch versions. As these functions internally rely on the object client it makes sense that the performance of this client is also reflected in the performance of the item client. The *create*, *delete* and *unordered-create-delete* operations benefit from the batch execution with a performance decrease being some percentage points smaller for the batch variant.

As there is not item client implementation in the ctypes implementation of the Python module it can not be compared against the CFFI implementation.

The item collection internally relies on the key value client. The backend used for benchmarking is also the LMDB. Results of the benchmark runs are displayed in Figure 5.5.

The first observation to point out is the fact that the item collection in Python has the largest decrease in performance over all benchmarks, ranging from a 15% to 23% decrease when comparing to the C benchmark. The Python tests also have a fairly large standard deviation when compared to the benchmark runs of the other operations. The theme of operations benefiting from the aggregation of tasks into a batch continues with the item collection. As it relies on the key value client internally, this makes sense.

Similar to the item client, there is not item collection implementation within the ctypes Python wrapper. So the CFFI implementation of item collection can not be compared to ctypes either.

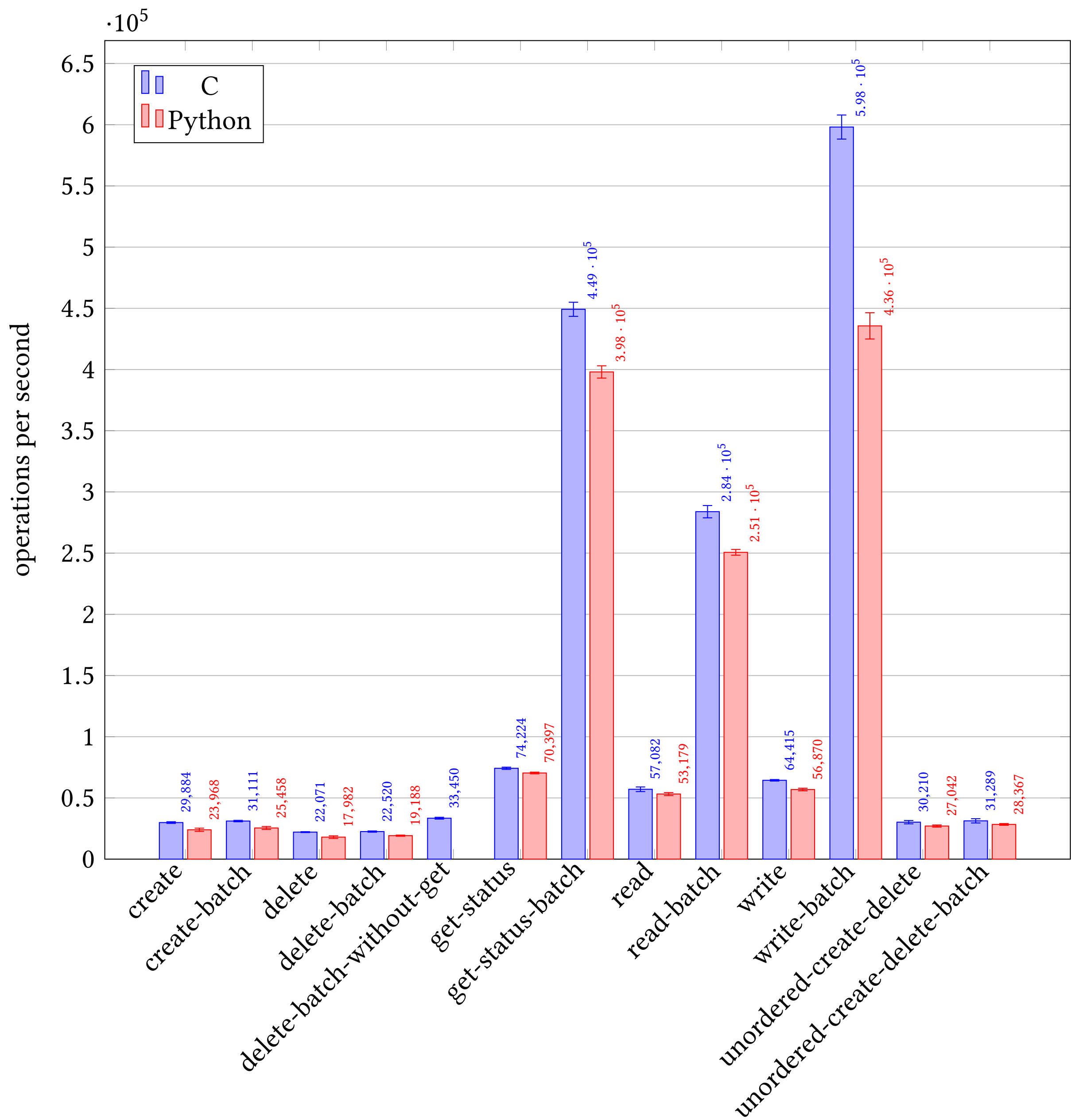

Figure 5.4.: Item Client Performance

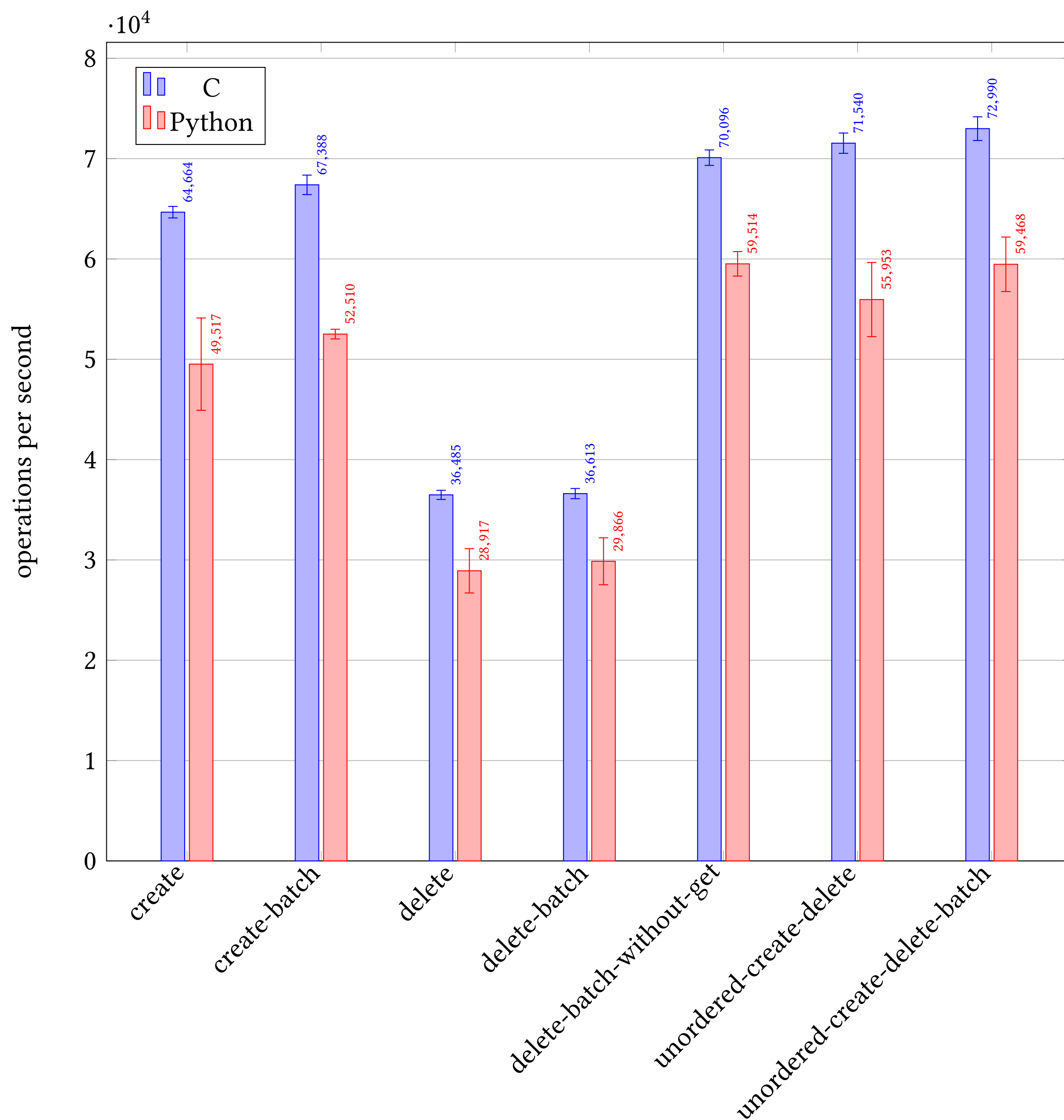

Figure 5.5.: Item Collection Performance

5.2.4. Database

The benchmarks for the database client, as well as the iterator and schema, did not produce results that can be evaluated. For every group of benchmarks about half of the Python benchmarks failed. The other half of the operations run in the benchmarks proposed a performance increase for the Python implementation over the C library. This is impossible because the Python implementation just wraps the C functions.

As the database client was not tested for correctness farther than the hello world example, it might be a problem with the way the functions are invoked or a problem with the wrapper of the database functions in general. This error most likely propagated to the benchmarks, resulting in the measurements seen in Chapter B. There was not enough time to fix these issues in the scope of this thesis, so it will be necessary to look into these issues as part of future work.

5.2.5. Result

The Python code achieves less operations per second than the C code, but due to CFFI wrapping the C code this was expected. The *read* operations perform 7% to 22% worse than the C implementation, depending on the client. The *write* operations perform 11% to 27% worse than the C implementation, also depending on the client and whether the operations are executed in bulk. The *delete* operations perform 13% to 26% worse than the C implementation, depending on the client. The operations benefit from the batch execution, but the *write* and the *status* operation of the Python object client can not benefit from the batch execution as much as the C operations, leading to a larger drop in performance.

Chapter 6.

Conclusion

In this chapter, the work of this thesis will be summarized and a conclusion will be drawn. Afterwards there will be a look into future work to be done, building on this thesis.

6.1. Summary

This thesis presented an approach to automatically generate a module for Python given a library written in C. As described in Chapter 1, Python as a language in the context of HPC is high in popularity due to its simplicity. Some possible approaches to generating a Python module, which interfaces with a given library written in C, were shown in Chapter 2. CFFI was chosen and in Chapter 3 the implementation process was depicted. The methods for processing a large amount of header files were explained. First using custom logic, written in Python, and finally setteling on available toolchains to simplify the process.

Chapter 4 took an excursion into Python in HPC and its performance in general, the previous version of a JULEA Python module and the generation of language bindings for other languages. In Chapter 5 the resulting module was put to the test using a script utilizing three of the clients for reading and writing data. The performance of the working clients was compared to the C clients and the affirmentioned other Python module for JULEA.

6.2. Conclusion

As shown in Chapter 5 the module created with the help of CFFI works well for the key value, (distributed) object and item client. It demonstrates that CFFI can be used to create Python modules for larger libraries with nested includes, given some preprocessing. The decrease in performance compared to C is less than 30% which is not negligible but manageable. The module generated using CFFI, in some cases performes, better than the Python module previously implemented for JULEA using ctypes. The database client could not be utilized beyond the example Python script.

6.3. Future Work

As described in Chapter 5.2.4, most of the database client's features do not work properly yet. A first point to start is verifying the hypothesis that all datatypes can be written and read correctly, using the database client. Afterwards the issues with the database benchmarks need to be fixed so, that the performance of the Python database client can be compared to the C database client.

The benchmarks for the item client, item collection, distributed object client and object client can be expanded by a run measuring the performance of their respective get method. Fixing the broken delete-batch-without-get benchmark of the item client might also be possible in the process. Additionally some profiling of the Python code might be done to evaluate whether the impact on performance of the Python wrapper can be mitigated.

Other solutions explored in Chapter 2 can also be investigated further. SWIG and Shroud look like promising solutions and can potentially offer support for more languages than just Python and C. Bindgen for Rust, which is mentioned in Chapter 4, also bears the potential for offering JULEA support for another language with manageable effort.

Bibliography

[Abrahams and Seefeld, 2002] Abrahams, D. and Seefeld, S. (2002). Boost.python. https://www.boost.org/doc/libs/1_79_0/libs/python/doc/html/index.html. Accessed: 2022-12-05. (Cited on page 7)

[Behnel et al., 2009] Behnel, S., Bradshaw, R., Seljebotn, D. S., Ewing, G., Stein, W., Gellner, G., and et al. (2009). Cython's Documentation. https://cython.readthedocs.io/en/latest/. Accessed: 2022-07-19. (Cited on page 9)

[Candido, 2018] Candido, R. (2018). Pure Python vs NumPy vs TensorFlow Performance Comparison. https://realpython.com/numpy-tensorflow-performance/. Accessed: 2023-01-17. (Cited on page 27)

[Coym, 2019] Coym, J. (2019). Python for High Performance I/O. Bachelor's thesis, Universität Hamburg, Hamburg. (Cited on pages 9, 28, 32, and 33)

[Data Science Council of America, 2021] Data Science Council of America (2021). Top 6 Programming Languages for Data Science in 2021. https://www.dasca.org/world-of-big-data/article/top-6-programming-languages-for-data-science-in-2021. Accessed: 2023-01-17. (Cited on page 27)

[Free Software Foundation, Inc., 1988] Free Software Foundation, Inc. (1988). GCC man page. https://manpages.org/gcc. Accessed: 2022-11-14. (Cited on pages 7, 18, and 19)

[Fullard and Wang, 2022] Fullard, A. and Wang, X. (2022). Python on HPC. https://docs.icer.msu.edu/Python_on_HPC/. Accessed: 2023-01-17. (Cited on page 27)

[Fulton et al., 1999] Fulton, W. S., Betts, O., Köppe, M., Wette, K., Wang, J., Kabak, D., Spytz, Z., Mitchell, D. J., and et al. (1999). SWIG-4.0 Documentation. https://swig.org/Doc4.0/SWIGDocumentation.html. Accessed: 2022-07-19. (Cited on pages 6 and 28)

[Goregaokar et al., 2021] Goregaokar, M., Okabayashi, Q., Laddad, S., Bastian, R., Cortier, B., Carr, S. F., Tatum, G., Braniecki, Z., and et al. (2021). Diplomat. https://github.com/rust-diplomat/diplomat. Accessed: 2022-10-11. (Cited on page 28)

[Harris et al., 2008] Harris, C., Oliphant, T. E., Cournapeau, D., Berg, S., Peterson, P., Wieser, E., Picus, M., Virtanen, P., and et al. (2008). NumPy. https://numpy.org. Accessed: 2022-11-14. (Cited on page 1)

[Hillier, 2022] Hillier, W. (2022). What's the Best Language for Machine Learning? https://careerfoundry.com/en/blog/data-analytics/best-machine-learning-languages/. Accessed: 2023-01-17. (Cited on page 27)

[Jakob, 2017] Jakob, W. (2017). pybind11 Documentation. https://pybind11.readthedocs.io/en/stable/. Accessed: 2022-07-19. (Cited on page 7)

[Kuhn, 2017] Kuhn, M. (2017). JULEA: A Flexible Storage Framework for HPC. In Kunkel, J. M., Yokota, R., Taufer, M., and Shalf, J., editors, *High Performance Computing*, pages 712–723, Cham. Springer International Publishing. (Cited on pages 1 and 12)

[Langtangen and Cai, 2008] Langtangen, H. P. and Cai, X. (2008). On the Efficiency of Python for High-Performance Computing: A Case Study Involving Stencil Updates for Partial Differential Equations. In Bock, H. G., Kostina, E., Phu, H. X., and Rannacher, R., editors, *Modeling, Simulation and Optimization of Complex Processes*, pages 337–357, Berlin, Heidelberg. Springer Berlin Heidelberg. (Cited on page 27)

[Lawrence Livermore National Laboratory, 2017] Lawrence Livermore National Laboratory (2017). Shroud. `https://shroud.readthedocs.io/en/develop/`. Accessed: 2022-07-19. (Cited on page 10)

[Lunacek, 2013] Lunacek, M. (2013). Python for High Performance Computing. `https://sea.ucar.edu/sites/default/files/python_hpc.pdf`. Accessed: 2023-01-17. (Cited on page 27)

[Matos et al., 2010] Matos, J., Dobrev, D., Henrich, M., Corrado, A., Holzer, E., Valkeinen, T., Kupstys, R., Spilman, T., and et al. (2010). CppSharp. `https://github.com/mono/CppSharp`. Accessed: 2023-01-17. (Cited on page 28)

[Python Software Foundation, 2001a] Python Software Foundation (2001a). Applications for Python. `https://www.python.org/about/apps/`. Accessed: 2022-09-12. (Cited on page 3)

[Python Software Foundation, 2001b] Python Software Foundation (2001b). ctypes – A foreign function library for Python. `https://docs.python.org/3.10/library/ctypes.html`. Accessed: 2022-07-19. (Cited on page 9)

[Riga Technical University, 2023] Riga Technical University (2023). HPC Software – Python. `https://hpc.rtu.lv/hpc/hpc-software/python-hpc/`. Accessed: 2023-01-17. (Cited on page 27)

[Rigo and Fijalkowski, 2012] Rigo, A. and Fijalkowski, M. (2012). CFFI documentation. `https://cffi.readthedocs.io/en/latest/`. Accessed: 2022-07-19. (Cited on pages 4, 13, and 23)

[Roghult, 2016] Roghult, A. (2016). Benchmarking Python Interpreters : Measuring Performance of CPython, Cython, Jython and PyPy. Master's thesis, KTH, School of Computer Science and Communication (CSC). (Cited on page 28)

[Rohland et al., 2001] Rohland, C., Dickins, H., KOSAKI, M., and Down, C. (2001). Tmpfs. `https://www.kernel.org/doc/html/latest/filesystems/tmpfs.html`. Accessed: 2023-01-18. (Cited on page 31)

[SchedMD, 2022] SchedMD (2022). Slurm Workload Manager Documentation. `https://slurm.schedmd.com`. Accessed: 2023-01-18. (Cited on page 31)

[Serebryany et al., 2011] Serebryany, K., Stepanov, E., Buka, V., Potapenko, A., Iskhodzhanov, T., Morehouse, M., Konovalov, A., Aizatsky, M., and et al. (2011). sanitizers. `https://github.com/google/sanitizers`. Accessed: 2023-01-13. (Cited on page 21)

[Stroustrup, 2021] Stroustrup, B. (2021). Bjarne Stroustrup's FAQ – Is C a subset of C++? `https://www.stroustrup.com/bs_faq.html#C-is-subset`. Accessed: 2022-12-05. (Cited on page 7)

[The PyPy Project, 2023] The PyPy Project (2023). PyPy documentation. https://doc.pypy.org/en/latest/. Accessed: 2023-01-18. (Cited on page 28)

[The Trustees of Princeton University, 2023] The Trustees of Princeton University (2023). Python on the HPC Clusters. https://researchcomputing.princeton.edu/support/knowledge-base/python. Accessed: 2023-01-17. (Cited on page 27)

[TIOBE Software B.V., 2022] TIOBE Software B.V. (2022). TIOBE Index for September 2022. https://www.tiobe.com/tiobe-index/. Accessed: 2022-09-12. (Cited on page 3)

[Whitfield, 2022] Whitfield, B. (2022). 17 High-Performance Computing Applications and Examples. https://builtin.com/hardware/high-performance-computing-applications. Accessed: 2023-01-17. (Cited on page 27)

[Álvarez et al., 2012] Álvarez, E. C., Fitzgerald, N., You, J.-Y., Kulp, D., Barnard, E., Shulepov, S., Quan, X., Finkenauer, T., and et al. (2012). bindgen. https://github.com/rust-lang/rust-bindgen. Accessed: 2022-09-15. (Cited on page 28)

Acronyms

CFFI C Foreign Function Interface

CPU Central Processing Unit

GCC GNU compiler collection

GPU Graphics Processing Unit

HPC High Performance Computing

JIT just-in-time compiler

LMDB Lightning Memory-Mapped Database

ops operations

POSIX Portable Operating System Interface

rt runtime

SWIG Simplefied Wrapper and Interface Generator

Appendix A.

Listings

```
#pragma once

float power(float base, int exponent);
float power_verbose(float base, int exponent);
void loop (float* result, float factor, int counter);
int fib(int number);
```

Listing A.1: Header Example Libary (example.h)

```
#include <stdio.h>
#include "example.h"

float power_verbose(float base, int exponent) {
    float res = power(base, exponent);
    printf("in c:\t\t%.2f^%d is %.2f\n", base, exponent, res);
    return res;
}

float power(float base, int exponent) {
    float result = 1.f;
    if (!exponent) { // check if exponent is 0
        return result;
    }
    if (exponent > 0) {
        loop(&result, base, exponent);
        return result;
    }
    if (exponent < 0) {
        loop(&result, base, -exponent);
        return 1/result;
    }
    return 0.f;
}

void loop (float* result, float factor, int counter) {
    for (int i = 0; i < counter; ++i) {
        *result *= factor;
```

```
29      }
30  }
31
32  int fib(int number) {
33      if(number < 2) {
34          return 1;
35      }
36      return fib(number-2) + fib(number-1);
37  }
```

Listing A.2: Implementation Example Libary (example.c)

```
1   #include <julea.h>
2   #include <julea-object.h>
3   #include <julea-kv.h>
4   #include <julea-db.h>
5
6   #include <locale.h>
7   #include <stdio.h>
8
9   int
10  main(int argc, char** argv)
11  {
12      g_autoptr(JBatch) batch = NULL;
13      g_autoptr(JDBEntry) entry = NULL;
14      g_autoptr(JDBSchema) schema = NULL;
15      g_autoptr(JKV) kv = NULL;
16      g_autoptr(JObject) object = NULL;
17
18      gchar const* hello_world = "Hello World!";
19      guint64 nbytes;
20
21      (void)argc;
22      (void)argv;
23
24      // Explicitly enable UTF-8 since functions such as
           ↪ g_format_size might return UTF-8 characters.
25      setlocale(LC_ALL, "C.UTF-8");
26
27      /// \todo: this example does not clean up after itself and
           ↪ will only work if the object, kv pair and db entry do
           ↪ not exist already
28      /// \todo: add more error checking
29
30      batch =
           ↪ j_batch_new_for_template(J_SEMANTICS_TEMPLATE_DEFAULT);
31      object = j_object_new("hello", "world");
32      kv = j_kv_new("hello", "world");
33      schema = j_db_schema_new("hello", "world", NULL);
```

```
34	j_db_schema_add_field(schema, "hello", J_DB_TYPE_STRING,
	    ↪ NULL);
35	entry = j_db_entry_new(schema, NULL);
36	j_db_entry_set_field(entry, "hello", hello_world,
	    ↪ strlen(hello_world) + 1, NULL);
37
38	j_object_create(object, batch);
39	j_object_write(object, hello_world, strlen(hello_world) + 1,
	    ↪ 0, &nbytes, batch);
40	j_kv_put(kv, g_strdup(hello_world), strlen(hello_world) + 1,
	    ↪ g_free, batch);
41	j_db_schema_create(schema, batch, NULL);
42	j_db_entry_insert(entry, batch, NULL);
43
44	if (j_batch_execute(batch))
45	{
46	    g_autoptr(JDBIterator) iterator = NULL;
47	    g_autoptr(JDBSelector) selector = NULL;
48
49	    gchar buffer[128] = { '\0' };
50	    g_autofree gpointer value = NULL;
51	    guint32 length = 0;
52	    g_autofree gchar* db_field = NULL;
53	    guint64 db_length = 0;
54
55	    j_object_read(object, buffer, 128, 0, &nbytes, batch);
56
57	    if (j_batch_execute(batch))
58	    {
59	        printf("Object contains: %s (%" G_GUINT64_FORMAT "
	            ↪ bytes)\n", buffer, nbytes);
60	    }
61
62	    j_kv_get(kv, &value, &length, batch);
63
64	    if (j_batch_execute(batch))
65	    {
66	        printf("KV contains: %s (%" G_GUINT32_FORMAT "
	            ↪ bytes)\n", (gchar*)value, length);
67	    }
68
69	    selector = j_db_selector_new(schema,
	        ↪ J_DB_SELECTOR_MODE_AND, NULL);
70	    j_db_selector_add_field(selector, "hello",
	        ↪ J_DB_SELECTOR_OPERATOR_EQ, hello_world,
	        ↪ strlen(hello_world) + 1, NULL);
71	    iterator = j_db_iterator_new(schema, selector, NULL);
72
```

```
73          while (j_db_iterator_next(iterator, NULL))
74          {
75              JDBType type;
76
77              j_db_iterator_get_field(iterator, "hello", &type,
                   ↪ (gpointer*)&db_field, &db_length, NULL);
78              printf("DB contains: %s (%" G_GUINT64_FORMAT "
                   ↪ bytes)\n", db_field, db_length);
79          }
80      }
81
82      return 0;
83  }
```

Listing A.3: JULEA example written in C (hello-world.c)

```
1   from julea import JBatchResult, JBatch, ffi, lib, encode,
       ↪ read_from_buffer
2
3   if __name__ == "__main__":
4       try:
5           value_obj = encode("Hello Object!")
6           value_kv = encode("Hello Key-Value!")
7           value_db = encode("Hello Database!")
8
9           result = JBatchResult()
10          hello = encode("hello")
11          world = encode("world")
12          nbytes_ptr = ffi.new("unsigned long*")
13          _object = lib.j_object_new(hello, world)
14          kv = lib.j_kv_new(hello, world)
15          schema = lib.j_db_schema_new(hello, world, ffi.NULL)
16          lib.j_db_schema_add_field(schema, hello,
               ↪ lib.J_DB_TYPE_STRING, ffi.NULL)
17          entry = lib.j_db_entry_new(schema, ffi.NULL)
18
19          lib.j_db_entry_set_field(entry, hello, value_db,
               ↪ len(value_db), ffi.NULL)
20
21          with JBatch(result) as batch:
22              lib.j_object_create(_object, batch)
23              lib.j_object_write(_object, value_obj,
                   ↪ len(value_obj), 0, nbytes_ptr, batch)
24              lib.j_kv_put(kv, value_kv, len(value_kv), ffi.NULL,
                   ↪ batch)
25              lib.j_db_schema_create(schema, batch, ffi.NULL)
26              lib.j_db_entry_insert(entry, batch, ffi.NULL)
27
28          if result.IsSuccess:
```

```
29            result = JBatchResult()
30            buffer = ffi.new("gchar[]", 128)
31            with JBatch(result) as batch:
32                lib.j_object_read(_object, buffer, 128, 0,
                    ↪ nbytes_ptr, batch)
33            if result.IsSuccess:
34                print(f"Object contains:
                    ↪ '{read_from_buffer(buffer)}'
                    ↪ ({nbytes_ptr[0]} bytes)")
35            with JBatch(result) as batch:
36                lib.j_object_delete(_object, batch)
37
38            result = JBatchResult()
39            with JBatch(result) as batch:
40                buffer_ptr = ffi.new("void**")
41                length = ffi.new("unsigned int *")
42                lib.j_kv_get(kv, buffer_ptr, length, batch)
43            if result.IsSuccess:
44                char_buff_ptr = ffi.cast("char**", buffer_ptr)
45                print(f"KV contains:
                    ↪ '{read_from_buffer(char_buff_ptr[0])}'
                    ↪ ({length[0]} bytes)")
46            with JBatch(result) as batch:
47                lib.j_kv_delete(kv, batch)
48
49            try:
50                selector = lib.j_db_selector_new(schema,
                    ↪ lib.J_DB_SELECTOR_MODE_AND, ffi.NULL)
51                lib.j_db_selector_add_field(selector, hello,
                    ↪ lib.J_DB_SELECTOR_OPERATOR_EQ, value_db,
                    ↪ len(value_db), ffi.NULL)
52                iterator = lib.j_db_iterator_new(schema,
                    ↪ selector, ffi.NULL)
53
54                while lib.j_db_iterator_next(iterator, ffi.NULL):
55                    _type = ffi.new("JDBType *")
56                    db_field_ptr = ffi.new("void**")
57                    db_length_ptr = ffi.new("unsigned long*")
58                    lib.j_db_iterator_get_field(iterator, hello,
                        ↪ _type, db_field_ptr, db_length_ptr,
                        ↪ ffi.NULL)
59                    print(f"DB contains:
                        ↪ '{read_from_buffer(db_field_ptr[0])}'
                        ↪ ({db_length_ptr[0]} bytes)")
60
61            finally:
62                with JBatch(result) as batch:
63                    lib.j_db_entry_delete(entry, selector, batch,
```

```
                        ↪ ffi.NULL)
64                   lib.j_db_schema_delete(schema, batch,
                        ↪ ffi.NULL)
65               lib.j_db_selector_unref(selector)
66               lib.j_db_iterator_unref(iterator)
67
68       finally:
69           lib.j_kv_unref(kv)
70           lib.j_object_unref(_object)
71           lib.j_db_schema_unref(schema)
72           lib.j_db_entry_unref(entry)
```

Listing A.4: JULEA example written in Python (hello-world.py)

```
1  import os
2  import re
3
4  def get_additional_compiler_flags(libraries,
     ↪ remove_sanitize=True):
5      flags_buffer = os.popen("pkg-config --cflags
         ↪ {libs}".format(libs=' '.join(libraries)))
6      flags = flags_buffer.read().strip().split(' ')
7      flags = [*set(flags)]
8      if remove_sanitize:
9          for s in flags:
10             if "-fsanitize" in s:
11                 flags.remove(s)
12     return flags
13
14 def get_include_dirs(flags):
15     return [ str.strip("-I") for str in flags if "-I" in str ]
16
17 def read_header_file(path, include_dirs=[]):
18     include_dirs.insert(0, ".") # search first in current
         ↪ directory
19     filename = get_filename_from_path(path)
20     included_files = [ filename ]
21     return read_header_file_internal(path, include_dirs,
         ↪ included_files)
22
23 def get_filename_from_path(path):
24     match_result = re.search(r"\/([\w-]+\/)*([\w-]+\.h)", path)
25     if match_result != None:
26         return match_result.group(2)
27     return ""
28
29 def read_header_file_internal(path, include_dirs=[],
     ↪ included_files=[]):
30     content = ""
```

```
    with open(path) as header:
        for line in header:
            is_include_line, filename = is_include(line)
            if is_include_line:
                if filename in included_files:
                    continue
                for directory in include_dirs:
                    path = os.path.join(directory, filename)
                    if os.path.exists(path):
                        included_files.append(filename)
                        content += read_header_file_internal(path, include_dirs, included_files)
                        break;
                continue
            if is_precompiler_directive(line):
                continue
            content += line
        return content

def is_include(line):
    ex = r"#include\s+(<(([\w-]+\/)*[\w-]+\.h)>" + r"|" + r"\"(([\w-]+\/)*[\w-]+\.h)\")"
    match_result = re.search(ex, line)
    if match_result == None:
        return False, ""
    filename = match_result.group(2)
    if filename != None:
        return True, filename
    filename = match_result.group(4)
    if filename != None:
        return True, filename
    # this should never need to be called
    return False, ""

def is_precompiler_directive(line):
    return line.startswith('#')
```

Listing A.5: Helper script for preprocessing header files with Python (header_preprocessor.py)

```
from os import popen, system
from os.path import dirname
import cffi

def create_header(filename, libraries):
    content = """typedef int gint;
typedef unsigned int guint;
typedef gint gboolean;
```

```
 9 typedef char gchar;
10
11 typedef unsigned short guint16;
12 typedef signed int gint32;
13 typedef unsigned int guint32;
14 typedef signed long gint64;
15 typedef unsigned long guint64;
16
17 typedef void* gpointer;
18 typedef const void *gconstpointer;
19
20 typedef unsigned long gsize;
21
22 typedef guint32 GQuark;
23
24 typedef struct _GError GError;
25 struct _GError
26 {
27     GQuark  domain;
28     gint    code;
29     gchar  *message;
30 };
31
32 typedef    struct _GModule GModule;
33
34 typedef struct _GInputStream GInputStream;
35 typedef struct _GOutputStream GOutputStream;
36
37 typedef struct _GKeyFile GKeyFile;
38
39 typedef struct _GSocketConnection GSocketConnection;
40
41 typedef void (*GDestroyNotify) (gpointer data);
42
43 typedef struct _bson_t
44 {
45     uint32_t    flags;
46     uint32_t    len;
47     uint8_t     padding[120];
48 } bson_t;
49
50 typedef struct JBatch JBatch;
51 extern "Python" void cffi_j_kv_get_function(gpointer, guint32,
      ↪ gpointer);
52 extern "Python" void cffi_j_batch_async_callback(JBatch*,
      ↪ gboolean, gpointer);
53
54 """
```

```
55     for library in libraries:
56         content+=f"#include <{library}.h>\n"
57     with open(filename, "w") as file:
58         file.write(content)
59
60 def get_additional_compiler_flags(libraries,
   ↪ remove_sanitize=True):
61     flags_buffer = popen(f"pkg-config --cflags {'
       ↪ '.join(libraries)}")
62     flags = flags_buffer.read().strip().split(' ')
63     # remove duplicate parameters
64     flags = [*set(flags)]
65     if remove_sanitize:
66         for s in flags:
67             if "-fsanitize" in s:
68                 flags.remove(s)
69     return flags
70
71 def get_include_dirs(flags):
72     return [ str.strip("-I") for str in flags if "-I" in str ]
73
74 def collect_julea(filename, libraries, debug = False):
75     temp_filename = "temp.h"
76     create_header(temp_filename, libraries)
77     includes = get_additional_compiler_flags(libraries)
78     flags = list(filter(lambda entry: not "dependencies" in
       ↪ entry, includes))
79     # create dummy headers for files intentionally not included
80     with open("glib.h", "w") as file:
81         file.write("")
82     with open("gmodule.h", "w") as file:
83         file.write("")
84     with open("bson.h", "w") as file:
85         file.write("")
86     system("mkdir -p gio")
87     with open("gio/gio.h", "w") as file:
88         file.write("")
89     # list of macros to be ignored
90     macros = [
91             "-D'G_DEFINE_AUTOPTR_CLEANUP_FUNC(x, y)='",
92             "-D'G_END_DECLS='",
93             "-D'G_BEGIN_DECLS='",
94             "-D'G_GNUC_WARN_UNUSED_RESULT='",
95             "-D'G_GNUC_PRINTF(x, y)='"
96             ]
97     # let preprocessor collect all declarations
98     system(f"gcc -E -P {' '.join(macros)} {temp_filename} -I. {'
       ↪ '.join(flags)} -o {filename}")
```

```python
99      # remove temporary files needed to please the preprocessor
100     system(f"rm -rf glib.h gmodule.h bson.h gio {temp_filename}")
101 
102 def process(libs, tempheader, debug=False):
103     ffi = cffi.FFI()
104     libraryname = "julea_wrapper"
105     with open(tempheader, "r") as file:
106         header_content = file.read()
107     includes = get_additional_compiler_flags(libs+["glib-2.0"],
           ↪ remove_sanitize=True)
108     include_dirs = get_include_dirs(includes)
109     ffi.cdef(header_content, override=True)
110     outdir = f"{dirname(__file__)}/../bld/"
111     headerincludes = ""
112     for lib in libs:
113         headerincludes += f'#include "{lib}.h"\n'
114     ffi.set_source(
115             libraryname,
116             headerincludes,
117             libraries=libs+["kv-null"],
118             include_dirs=include_dirs,
119             library_dirs=[outdir],
120             extra_compile_args=includes,
121             extra_link_args=["-Wl,-rpath,."]
122             )
123     ffi.compile(tmpdir=outdir, verbose=debug)
124     if not debug:
125         system(f"rm -f {tempheader} {outdir+libraryname}.o
               ↪ {outdir+libraryname}.c")
126 
127 def copy_main_module():
128     system(f"cp {dirname(__file__)}/julea.py
           ↪ {dirname(__file__)}/../bld/julea.py")
129 
130 def build(include_libs, debug=False):
131     header_name = "header_julea.h"
132     collect_julea(header_name, include_libs, debug)
133     process(include_libs, header_name, debug)
```

Listing A.6: Final helper script for CFFI build process (build_helper.py)

```bash
1 #!/bin/bash
2 
3 #SBATCH --nodes=1
4 #SBATCH --ntasks=1
5 #SBATCH --cpus-per-task=48
6 #SBATCH --nodelist=ant13
7 #SBATCH --output=benchmarks/julea-%j-tmpfs.txt
8 
```

```
9  hostname
10 . julea/scripts/environment.sh
11 julea-config --user \
12   --object-servers="$(hostname)" --kv-servers="$(hostname)"
       ↪ --db-servers="$(hostname)" \
13   --object-backend=posix --object-component=server
       ↪ --object-path="/dev/shm/julea-$(id -u)/posix" \
14   --kv-backend=lmdb --kv-component=server
       ↪ --kv-path="/dev/shm/julea-$(id -u)/lmdb" \
15   --db-backend=sqlite --db-component=server
       ↪ --db-path="/dev/shm/julea-$(id -u)/sqlite"
16 . julea/python/testenv/bin/activate
17 ./julea/scripts/setup.sh clean-local
18 ./julea/scripts/setup.sh start-local
19 python print_python_header.py
20 python julea/benchmark/python/benchmark.py -m
21 ./julea/scripts/setup.sh stop-local
22 ./julea/scripts/setup.sh clean-local
23 ./julea/scripts/setup.sh start-local
24 python print_c_header.py
25 julea-benchmark -m --machine-separator=,
26 ./julea/scripts/setup.sh stop-local
27 ./julea/scripts/setup.sh clean-local
```

Listing A.7: Job script for running benchmarks using Slurm, execute Python first (run-benchmark1.sh)

```
1  #!/bin/bash
2
3  #SBATCH --nodes=1
4  #SBATCH --ntasks=1
5  #SBATCH --cpus-per-task=48
6  #SBATCH --nodelist=ant13
7  #SBATCH --output=benchmarks/julea-%j-tmpfs-c-first.txt
8
9  hostname
10 . julea/scripts/environment.sh
11 julea-config --user \
12   --object-servers="$(hostname)" --kv-servers="$(hostname)"
       ↪ --db-servers="$(hostname)" \
13   --object-backend=posix --object-component=server
       ↪ --object-path="/dev/shm/julea-$(id -u)/posix" \
14   --kv-backend=lmdb --kv-component=server
       ↪ --kv-path="/dev/shm/julea-$(id -u)/lmdb" \
15   --db-backend=sqlite --db-component=server
       ↪ --db-path="/dev/shm/julea-$(id -u)/sqlite"
16 . julea/python/testenv/bin/activate
17 ./julea/scripts/setup.sh clean-local
18 ./julea/scripts/setup.sh start-local
```

```
19 python print_c_header.py
20 julea-benchmark -m --machine-separator=,
21 ./julea/scripts/setup.sh stop-local
22 ./julea/scripts/setup.sh clean-local
23 ./julea/scripts/setup.sh start-local
24 python print_python_header.py
25 python julea/benchmark/python/benchmark.py -m
26 ./julea/scripts/setup.sh stop-local
27 ./julea/scripts/setup.sh clean-local
```

Listing A.8: Job script for running benchmarks using Slurm, execute C first (run-benchmark2.sh)

Appendix B.

Benchmark measurements

In the following tables runtime (rt) and operations (ops) will be abbreviated.

Name	**rt in s (C)**	**# ops (C)**	**rt in s (Python)**	**# ops (Python)**
put	1.007277	69494.290051	0.927475717	50000
put-batch	1.004885	71649.989800	0.909710972	50000
get	1.001949	83836.602462	0.752080605	50000
get-batch	1.004899	88566.114605	0.680750232	50000
delete	1.011152	69227.969682	1.020115638	50000
delete-batch	1.012832	71087.801333	0.889014649	50000
unordered-put-delete	1.004645	73657.859244	1.718879165	100000
unordered-put-delete-batch	1.010076	75241.862989	1.597762384	100000

Table B.1.: Key Value client benchmark run 01 of 10 (first Python, then C)

Name	rt in s (C)	# ops (C)	rt in s (Python)	# ops (Python)
put	1.006400	43720.190779	0.939028998	50000
put-batch	1.010439	71256.156977	0.906936823	50000
get	1.009777	87147.954449	0.764073995	50000
get-batch	1.008082	87294.485964	0.680670752	50000
delete	1.007277	68501.514479	0.997374046	50000
delete-batch	1.000541	70961.609769	0.89961961	50000
unordered-put-delete	1.007820	75410.291520	1.73706685	100000
unordered-put-delete-batch	1.011067	75168.114477	1.674743908	100000

Table B.2.: Key Value client benchmark run 02 of 10 (first Python, then C)

Name	rt in s (C)	# ops (C)	rt in s (Python)	# ops (Python)
put	1.004029	68723.114571	0.898732675	50000
put-batch	1.006089	70570.297459	0.887333499	50000
get	1.003542	85696.463128	0.746234532	50000
get-batch	1.006054	89458.418733	0.679709197	50000
delete	1.008643	69400.174294	0.976243774	50000
delete-batch	1.008449	70405.146914	0.890508467	50000
unordered-put-delete	1.004252	73686.684219	1.720790705	100000
unordered-put-delete-batch	1.004764	75639.652695	1.606906527	100000

Table B.3.: Key Value client benchmark run 03 of 10 (first Python, then C)

Name	rt in s (C)	# ops (C)	rt in s (Python)	# ops (Python)
put	1.000180	50990.821652	0.942543228	50000
put-batch	1.002288	70837.922833	0.893026812	50000
get	1.006743	87410.590389	0.734800685	50000
get-batch	1.000930	86919.165176	0.652381778	50000
delete	1.014219	68032.643837	0.980369498	50000
delete-batch	1.004711	70667.087351	0.937421769	50000
unordered-put-delete	1.006972	75473.796689	1.729610557	100000
unordered-put-delete-batch	1.021958	76324.075940	1.596570546	100000

Table B.4.: Key Value client benchmark run 04 of 10 (first Python, then C)

Name	rt in s (C)	# ops (C)	rt in s (Python)	# ops (Python)
put	1.009537	50518.207852	0.906547152	50000
put-batch	1.005150	70636.223449	0.888991327	50000
get	1.000365	85968.621453	0.73317825	50000
get-batch	1.003583	89678.681285	0.669839325	50000
delete	1.005996	69582.781641	0.975673628	50000
delete-batch	1.009151	70356.170682	0.912112003	50000
unordered-put-delete	1.013633	73004.726563	1.708219696	100000
unordered-put-delete-batch	1.002035	73849.715828	1.59886111	100000

Table B.5.: Key Value client benchmark run 05 of 10 (first Python, then C)

Name	rt in s (C)	# ops (C)	rt in s (Python)	# ops (Python)
put	1.020472	62716.076482	0.904755376	50000
put-batch	1.013502	71040.807024	0.934767293	50000
get	1.009775	86157.807432	0.734906451	50000
get-batch	1.006389	88434.988856	0.665519925	50000
delete	1.000467	68967.792041	0.984568691	50000
delete-batch	1.012096	71139.496649	0.907492592	50000
unordered-put-delete	1.001733	75868.519855	1.749891521	100000
unordered-put-delete-batch	1.023229	76229.270281	1.62935025	100000

Table B.6.: Key Value client benchmark run 06 of 10 (first C, then Python)

Name	rt in s (C)	# ops (C)	rt in s (Python)	# ops (Python)
put	1.001793	70872.924846	0.907459851	50000
put-batch	1.003165	72769.683950	0.895681606	50000
get	1.009026	86221.762373	0.753153092	50000
get-batch	1.006902	88389.932685	0.672522844	50000
delete	1.009158	68373.832442	1.00074823	50000
delete-batch	1.001720	70878.089686	0.892863119	50000
unordered-put-delete	1.024735	76117.240067	1.756640616	100000
unordered-put-delete-batch	1.014536	76882.436897	1.620561578	100000

Table B.7.: Key Value client benchmark run 07 of 10 (first C, then Python)

Name	rt in s (C)	# ops (C)	rt in s (Python)	# ops (Python)
put	1.011000	69238.377844	0.904729866	50000
put-batch	1.005257	70628.704898	0.883674618	50000
get	1.005466	85532.479467	0.731994994	50000
get-batch	1.010787	89039.530584	0.65229936	50000
delete	1.012172	69158.206313	0.963409491	50000
delete-batch	1.005131	70637.558686	0.873801283	50000
unordered-put-delete	1.014944	74880.978655	1.716587633	100000
unordered-put-delete-batch	1.023747	76190.699460	1.637950646	100000

Table B.8.: Key Value client benchmark run 08 of 10 (first C, then Python)

Name	rt in s (C)	# ops (C)	rt in s (Python)	# ops (Python)
put	1.008864	68393.757731	0.924020907	50000
put-batch	1.008681	70388.953495	0.938404475	50000
get	1.005435	87524.305400	0.74665578	50000
get-batch	1.006547	90408.098181	0.655687779	50000
delete	1.001790	69874.923886	0.965674036	50000
delete-batch	1.007500	71464.019851	0.914272064	50000
unordered-put-delete	1.003847	75708.748445	1.703004655	100000
unordered-put-delete-batch	1.026482	74039.291483	1.615356952	100000

Table B.9.: Key Value client benchmark run 09 of 10 (first C, then Python)

Name	rt in s (C)	# ops (C)	rt in s (Python)	# ops (Python)
put	1.002836	69802.041411	0.922974701	50000
put-batch	1.012498	72098.907850	0.897027528	50000
get	1.008531	86264.081124	0.741982264	50000
get-batch	1.009580	88155.470592	0.687791507	50000
delete	1.012926	68119.487505	0.97560565	50000
delete-batch	1.009643	71312.335152	0.890808384	50000
unordered-put-delete	1.021625	74391.288389	1.7019827	100000
unordered-put-delete-batch	1.019303	76522.878869	1.608465504	100000

Table B.10.: Key Value client benchmark run 10 of 10 (first C, then Python)

Name	rt in s (C)	# ops (C)	rt in s (Python)	# ops (Python)
create	1.015685	62027.104860	0.974380073	50000
create-batch	1.006443	63590.287776	0.891548433	50000
delete	1.003839	66743.770664	1.004198885	50000
delete-batch	1.002979	68795.059518	0.837687292	50000
status	1.010583	75204.114853	0.69752027	50000
status-batch	1.011921	474345.329329	1.197525965	500000
read	1.016741	59012.078789	0.970661792	50000
read-batch	1.024868	282963.269416	1.94260268	500000
write	1.004904	63687.675639	0.875692322	50000
write-batch	1.015477	610550.509760	1.185479975	500000
unordered-create-delete	1.015764	63006.761413	1.749027089	100000
unordered-create-delete-batch	1.025314	64370.524542	1.630882755	100000

Table B.11.: Distributed Object client benchmark run 01 of 10 (first Python, then C)

Name	rt in s (C)	# ops (C)	rt in s (Python)	# ops (Python)
create	1.000627	62960.523752	0.989791362	50000
create-batch	1.004772	64691.293149	0.907983608	50000
delete	1.001767	66881.819824	0.924481374	50000
delete-batch	1.002592	69819.029077	0.856803232	50000
status	1.006116	76531.930712	0.711566461	50000
status-batch	1.012983	473848.031013	1.192051833	500000
read	1.002775	58836.728080	0.954618619	50000
read-batch	1.025765	282715.826724	1.96163515	500000
write	1.004018	66731.871341	0.883286845	50000
write-batch	1.003386	597975.255784	1.098942444	500000
unordered-create-delete	1.012293	63222.802094	1.791505065	100000
unordered-create-delete-batch	1.016132	64952.191251	1.690314589	100000

Table B.12.: Distributed Object client benchmark run 02 of 10 (first Python, then C)

Name	rt in s (C)	# ops (C)	rt in s (Python)	# ops (Python)
create	1.000728	60955.624306	0.955711245	50000
create-batch	1.014433	63089.430253	0.895070584	50000
delete	1.003641	66756.937989	0.907592566	50000
delete-batch	1.008717	68403.724732	0.848151852	50000
status	1.009658	74282.578853	0.696375093	50000
status-batch	1.010766	464993.875932	1.193778043	500000
read	1.008155	58522.746998	0.916033745	50000
read-batch	1.030011	281550.391209	2.008703473	500000
write	1.001864	64879.065422	0.867847686	50000
write-batch	1.011131	603284.836485	1.108795251	500000
unordered-create-delete	1.001166	63925.462910	1.741763377	100000
unordered-create-delete-batch	1.003067	63804.312175	1.646345685	100000

Table B.13.: Distributed Object client benchmark run 03 of 10 (first Python, then C)

Name	rt in s (C)	# ops (C)	rt in s (Python)	# ops (Python)
create	1.012722	63196.020231	0.961400348	50000
create-batch	1.014678	64059.731264	0.894557201	50000
delete	1.003306	66779.227873	0.920430271	50000
delete-batch	1.005441	69621.191099	0.841787605	50000
status	1.008149	75385.682077	0.70966837	50000
status-batch	1.015638	472609.335216	1.207386242	500000
read	1.001176	59929.522881	0.925044097	50000
read-batch	1.018808	274830.978948	1.94949675	500000
write	1.013874	66083.162208	0.856886413	50000
write-batch	1.003431	587982.631591	1.094357828	500000
unordered-create-delete	1.031588	63979.030388	1.74926817	100000
unordered-create-delete-batch	1.018909	64775.166379	1.631232716	100000

Table B.14.: Distributed Object client benchmark run 04 of 10 (first Python, then C)

Name	rt in s (C)	# ops (C)	rt in s (Python)	# ops (Python)
create	1.009365	64396.922818	0.978426079	50000
create-batch	1.007831	65487.169972	0.985228195	50000
delete	1.012726	68132.940203	0.924178868	50000
delete-batch	1.006878	69521.828861	0.853094404	50000
status	1.007187	75457.685613	0.723746805	50000
status-batch	1.019901	470633.914468	1.182873568	500000
read	1.004482	59732.279921	0.923567806	50000
read-batch	1.002757	279230.162442	1.970084427	500000
write	1.010600	66297.249159	0.866947396	50000
write-batch	1.004157	577598.921284	1.085974618	500000
unordered-create-delete	1.011955	65220.291416	1.757609789	100000
unordered-create-delete-batch	1.028938	66087.558240	1.641477508	100000

Table B.15.: Distributed Object client benchmark run 05 of 10 (first Python, then C)

Name	rt in s (C)	# ops (C)	rt in s (Python)	# ops (Python)
create	1.006366	59620.456176	0.976539616	50000
create-batch	1.005386	62662.499776	0.885310628	50000
delete	1.003942	66736.923049	0.928626059	50000
delete-batch	1.004775	67676.843074	0.871576141	50000
status	1.001409	74894.473687	0.71518589	50000
status-batch	1.017894	471561.871865	1.178075718	500000
read	1.010522	57396.078462	0.952318191	50000
read-batch	1.023372	283376.914748	1.964744737	500000
write	1.006165	56650.748138	0.872725238	50000
write-batch	1.009868	584234.771277	1.092463291	500000
unordered-create-delete	1.018419	62842.503920	1.794061856	100000
unordered-create-delete-batch	1.006423	63591.551465	1.671032601	100000

Table B.16.: Distributed Object client benchmark run 06 of 10 (first C, then Python)

Name	rt in s (C)	# ops (C)	rt in s (Python)	# ops (Python)
create	1.013765	62144.579858	0.971966919	50000
create-batch	1.005537	64642.076821	0.854597526	50000
delete	1.010355	66313.325514	0.898683582	50000
delete-batch	1.005583	68616.911782	0.822539388	50000
status	1.006277	68569.588692	0.694021992	50000
status-batch	1.008168	486030.106093	1.180228638	500000
read	1.011869	47436.970596	0.985442544	50000
read-batch	1.019005	274777.847017	1.98557585	500000
write	1.001429	65905.820582	0.854302194	50000
write-batch	1.012262	592731.921182	1.2021389	500000
unordered-create-delete	1.030910	62081.074003	1.694610811	100000
unordered-create-delete-batch	1.022358	62600.380689	1.586781071	100000

Table B.17.: Distributed Object client benchmark run 07 of 10 (first C, then Python)

Name	rt in s (C)	# ops (C)	rt in s (Python)	# ops (Python)
create	1.004054	61749.666851	0.954030119	50000
create-batch	1.007505	64515.808855	0.87144813	50000
delete	1.002660	67819.599864	0.90562635	50000
delete-batch	1.007035	69510.990184	0.829383476	50000
status	1.008615	76342.310991	0.690206401	50000
status-batch	1.013306	473696.987879	1.193505202	500000
read	1.010576	58382.546191	0.91822897	50000
read-batch	1.001490	269598.298535	1.948717425	500000
write	1.003480	65771.116515	0.853847012	50000
write-batch	1.011933	602806.707559	1.125201092	500000
unordered-create-delete	1.010841	63313.617077	1.709748076	100000
unordered-create-delete-batch	1.021039	64640.038236	1.651715702	100000

Table B.18.: Distributed Object client benchmark run 08 of 10 (first C, then Python)

Name	rt in s (C)	# ops (C)	rt in s (Python)	# ops (Python)
create	1.007256	61553.368756	0.951009705	50000
create-batch	1.007323	64527.465371	0.888795733	50000
delete	1.007152	67517.117575	0.916775677	50000
delete-batch	1.008252	68435.272134	0.846760751	50000
status	1.004749	75640.781927	0.710647902	50000
status-batch	1.020571	470324.945545	1.193715932	500000
read	1.002622	59843.091414	0.940708708	50000
read-batch	1.014654	275956.138743	1.955788357	500000
write	1.002949	66802.997959	0.882332577	50000
write-batch	1.012203	592766.470757	1.090723965	500000
unordered-create-delete	1.020193	62733.227928	1.744219682	100000
unordered-create-delete-batch	1.000097	63993.792602	1.652370796	100000

Table B.19.: Distributed Object client benchmark run 09 of 10 (first C, then Python)

Name	rt in s (C)	# ops (C)	rt in s (Python)	# ops (Python)
create	1.015655	62028.936991	0.98086034	50000
create-batch	1.001845	63882.137456	0.869671428	50000
delete	1.000124	65991.817015	0.909773798	50000
delete-batch	1.014173	69021.754671	0.842867	50000
status	1.005740	75566.249727	0.911158976	50000
status-batch	1.014294	473235.570752	1.200099456	500000
read	1.000287	57983.358776	0.917592481	50000
read-batch	1.006665	278146.155871	1.979778949	500000
write	1.013239	66124.576729	0.872049531	50000
write-batch	1.003505	587939.272849	1.091388598	500000
unordered-create-delete	1.027304	62298.988420	1.744619914	100000
unordered-create-delete-batch	1.002709	63827.092407	1.639887497	100000

Table B.20.: Distributed Object client benchmark run 10 of 10 (first C, then Python)

Name	rt in s (C)	# ops (C)	rt in s (Python)	# ops (Python)
create	1.002586	62837.502219	0.948290622	50000
create-batch	1.004810	64688.846648	0.882712882	50000
delete	1.012245	67177.412583	0.900313424	50000
delete-batch	1.005505	68622.234599	0.830185938	50000
status	1.005284	76595.270590	0.68198496	50000
status-batch	1.015786	472540.476045	1.154856789	500000
read	1.008902	60461.769329	0.939847503	50000
read-batch	1.000822	289761.815787	1.915365243	500000
write	1.013195	67114.425160	0.837053367	50000
write-batch	1.014642	630764.348411	1.138641364	500000
unordered-create-delete	1.005299	63662.651609	1.735817072	100000
unordered-create-delete-batch	1.018909	62812.282549	1.625052742	100000

Table B.21.: Object client benchmark run 01 of 10 (first Python, then C)

Name	rt in s (C)	# ops (C)	rt in s (Python)	# ops (Python)
create	1.005183	63669.998398	0.992082405	50000
create-batch	1.003741	65754.014233	0.887972213	50000
delete	1.011279	68230.428991	0.900214724	50000
delete-batch	1.012385	62229.290240	0.842381199	50000
status	1.009277	76292.236918	0.707855229	50000
status-batch	1.010391	475063.613987	1.156081105	500000
read	1.004102	60750.800218	0.921462227	50000
read-batch	1.014267	285920.768397	1.932394991	500000
write	1.005176	67649.844405	0.869996839	50000
write-batch	1.014859	610922.305463	1.079676472	500000
unordered-create-delete	1.019280	64751.589357	1.756756684	100000
unordered-create-delete-batch	1.002181	65856.367263	1.644400963	100000

Table B.22.: Object client benchmark run 02 of 10 (first Python, then C)

Name	rt in s (C)	# ops (C)	rt in s (Python)	# ops (Python)
create	1.003614	62773.137880	0.949111916	50000
create-batch	1.008633	64443.657901	0.856174158	50000
delete	1.005277	67643.047638	0.899884862	50000
delete-batch	1.010994	70227.914310	0.833005105	50000
status	1.010852	76173.366625	0.697420599	50000
status-batch	1.008614	465985.996625	1.183999477	500000
read	1.006243	59627.743994	0.899960172	50000
read-batch	1.025518	282783.919931	1.94788203	500000
write	1.004448	66703.303705	0.860958106	50000
write-batch	1.010052	623730.263392	1.092600178	500000
unordered-create-delete	1.029049	64136.887553	1.691405496	100000
unordered-create-delete-batch	1.014327	65067.774002	1.601982038	100000

Table B.23.: Object client benchmark run 03 of 10 (first Python, then C)

Name	rt in s (C)	# ops (C)	rt in s (Python)	# ops (Python)
create	1.005578	63644.988256	0.940741588	50000
create-batch	1.000232	65984.691552	0.87027095	50000
delete	1.002365	68837.200022	0.898788885	50000
delete-batch	1.005208	68642.509809	0.855408784	50000
status	1.006115	76532.006779	0.69581168	50000
status-batch	1.000451	469788.125555	1.161064374	500000
read	1.009328	59445.492446	0.900723486	50000
read-batch	1.020122	284279.723406	1.95276901	500000
write	1.005836	66611.256706	0.852160505	50000
write-batch	1.013452	621637.729266	1.162387401	500000
unordered-create-delete	1.003289	63790.194052	1.733619131	100000
unordered-create-delete-batch	1.015317	65004.328697	1.624563008	100000

Table B.24.: Object client benchmark run 04 of 10 (first Python, then C)

Name	rt in s (C)	# ops (C)	rt in s (Python)	# ops (Python)
create	1.001737	64887.290776	0.963123302	50000
create-batch	1.014524	66040.823086	0.89518509	50000
delete	1.000579	68960.072118	0.919144999	50000
delete-batch	1.011515	71180.358176	0.839742904	50000
status	1.005261	77591.789595	0.702675086	50000
status-batch	1.004786	467761.294445	1.155555256	500000
read	1.010577	61351.089526	0.908071988	50000
read-batch	1.027868	282137.395074	1.958843975	500000
write	1.003489	66767.049763	0.863933724	50000
write-batch	1.008166	595140.086057	1.067585009	500000
unordered-create-delete	1.017673	64853.838119	1.721015294	100000
unordered-create-delete-batch	1.028494	66116.088183	1.618560683	100000

Table B.25.: Object client benchmark run 05 of 10 (first Python, then C)

Name	rt in s (C)	# ops (C)	rt in s (Python)	# ops (Python)
create	1.011578	61290.379981	0.959640471	50000
create-batch	1.010925	63308.356208	0.891898855	50000
delete	1.008013	66467.396750	0.911373913	50000
delete-batch	1.002577	69820.073670	0.845174505	50000
status	1.004615	76646.277430	0.710152353	50000
status-batch	1.014399	473186.586343	1.173936605	500000
read	1.004341	60736.343533	0.912472749	50000
read-batch	1.006545	288114.291959	1.925580459	500000
write	1.013769	67076.424708	0.872869109	50000
write-batch	1.001991	598807.773723	1.133461999	500000
unordered-create-delete	1.010069	61381.945194	1.736318096	100000
unordered-create-delete-batch	1.029293	64121.683525	1.672781912	100000

Table B.26.: Object client benchmark run 06 of 10 (first C, then Python)

Name	rt in s (C)	# ops (C)	rt in s (Python)	# ops (Python)
create	1.013662	63137.416614	0.918293181	50000
create-batch	1.010861	64301.620104	0.86061991	50000
delete	1.011166	67249.096588	0.883293736	50000
delete-batch	1.004915	68662.523696	0.816892066	50000
status	1.004175	74688.176862	0.682114346	50000
status-batch	1.014990	472911.063163	1.179100092	500000
read	1.015230	60084.906868	0.95231991	50000
read-batch	1.032215	280949.220850	1.954579507	500000
write	1.005884	66608.078069	0.845453545	50000
write-batch	1.010216	613730.132962	1.165479916	500000
unordered-create-delete	1.029836	64087.874186	1.668513526	100000
unordered-create-delete-batch	1.004716	65690.204993	1.57039813	100000

Table B.27.: Object client benchmark run 07 of 10 (first C, then Python)

Name	rt in s (C)	# ops (C)	rt in s (Python)	# ops (Python)
create	1.004381	63720.839004	0.957196627	50000
create-batch	1.007227	65526.440415	0.850852445	50000
delete	1.002547	67827.244010	0.889796882	50000
delete-batch	1.009851	70307.401785	0.833357098	50000
status	1.010596	77182.177646	0.68111925	50000
status-batch	1.021425	469931.713048	1.166371351	500000
read	1.004338	60736.524955	0.908058364	50000
read-batch	1.033068	280717.242234	1.928178031	500000
write	1.010816	67272.381917	0.835763051	50000
write-batch	1.007995	605161.731953	1.176163385	500000
unordered-create-delete	1.001228	63921.504393	1.680765344	100000
unordered-create-delete-batch	1.004593	63707.391949	1.612045831	100000

Table B.28.: Object client benchmark run 08 of 10 (first C, then Python)

Name	rt in s (C)	# ops (C)	rt in s (Python)	# ops (Python)
create	1.000570	62964.110457	0.953154097	50000
create-batch	1.012052	64225.948864	0.874598205	50000
delete	1.014159	68036.668806	0.890902844	50000
delete-batch	1.010659	68272.285707	0.854242293	50000
status	1.000953	75927.640958	0.711849078	50000
status-batch	1.013236	473729.713512	1.186618473	500000
read	1.014497	60128.319749	0.919689942	50000
read-batch	1.013705	286079.283421	1.928000743	500000
write	1.002843	65812.893943	0.86107464	50000
write-batch	1.007681	595426.528832	1.094159064	500000
unordered-create-delete	1.023798	62512.331534	1.721348156	100000
unordered-create-delete-batch	1.005412	63655.496453	1.657927918	100000

Table B.29.: Object client benchmark run 09 of 10 (first C, then Python)

Name	rt in s (C)	# ops (C)	rt in s (Python)	# ops (Python)
create	1.008961	61449.352354	0.951007595	50000
create-batch	1.003553	64769.872643	0.89191616	50000
delete	1.001774	67879.581622	0.905151513	50000
delete-batch	1.004032	69718.893422	0.834993046	50000
status	1.005765	74570.103354	0.694455703	50000
status-batch	1.002937	468623.652333	1.167382956	500000
read	1.011299	60318.461701	0.910911834	50000
read-batch	1.029261	281755.550827	1.951950066	500000
write	1.010808	67272.914342	0.857923363	50000
write-batch	1.009835	604059.078958	1.098314487	500000
unordered-create-delete	1.029642	64099.949303	1.719741347	100000
unordered-create-delete-batch	1.016290	64942.093300	1.623312866	100000

Table B.30.: Object client benchmark run 10 of 10 (first C, then Python)

Name	rt in s (C)	# ops (C)	rt in s (Python)	# ops (Python)
insert	1.147187	14281.891270	0.259827185	50000
insert-batch	1.115009	29388.103594	0.142342724	50000
insert-index-single	1.164842	10549.070174	0.317058326	50000
insert-batch-index-single	1.136254	18024.138969	-	-
insert-index-all	1.234914	13267.320639	0.274157707	50000
insert-batch-index-all	1.022416	24037.182517	0.162902554	50000
insert-index-mixed	1.242954	9886.126116	0.330830996	50000
insert-batch-index-mixed	1.051089	15587.642911	0.222067006	50000
delete	1.247860	1641.209751	0.049383306	50000
delete-batch	1.164147	1759.228001	0.040457264	50000
delete-index-single	1.197648	13680.146420	0.324737661	50000
delete-batch-index-single	1.171145	20984.592002	0.208255596	50000
delete-index-all	1.042421	15717.258190	0.553215924	50000
delete-batch-index-all	1.103476	25983.347168	0.427385035	50000
delete-index-mixed	1.256204	13042.467625	-	-
delete-batch-index-mixed	1.039905	19694.106673	0.218451995	50000
update	1.033376	6193.292664	-	-
update-batch	1.013320	7073.777287	-	-
update-index-single	1.080837	15158.622438	-	-
update-batch-index-single	1.004329	24470.069071	-	-
update-index-all	1.087672	15063.364691	-	-
update-batch-index-all	1.115427	25704.954246	-	-
update-index-mixed	1.149065	14258.549342	-	-
update-batch-index-mixed	1.076370	22832.297444	-	-

Table B.31.: Database client benchmark run 01 of 10 (first Python, then C)

Name	rt in s (C)	# ops (C)	rt in s (Python)	# ops (Python)
insert	1.163939	14076.339052	0.269220309	50000
insert-batch	1.042161	27512.063875	0.15020249	50000
insert-index-single	1.161327	10580.999150	0.322294736	50000
insert-batch-index-single	1.146229	17867.284810	-	-
insert-index-all	1.244532	13164.788049	0.273775205	50000
insert-batch-index-all	1.032480	23802.882380	0.17612479	50000
insert-index-mixed	1.232987	9966.041816	0.344161243	50000
insert-batch-index-mixed	1.053833	15547.055368	0.244531976	50000
delete	1.227448	1668.502454	0.049246005	50000
delete-batch	1.182722	1731.598803	0.042283214	50000
delete-index-single	1.199331	13660.949313	0.322595328	50000
delete-batch-index-single	1.035713	19773.817650	0.221742694	50000
delete-index-all	1.031440	15884.588536	0.541861588	50000
delete-batch-index-all	1.082523	26486.273271	0.434674308	50000
delete-index-mixed	1.263218	12970.049508	-	-
delete-batch-index-mixed	1.034566	19795.740436	0.241689219	50000
update	1.035872	6178.369528	-	-
update-batch	1.000650	7163.343827	-	-
update-index-single	1.075219	15237.825968	-	-
update-batch-index-single	1.121591	25563.685871	-	-
update-index-all	1.085274	15096.648404	-	-
update-batch-index-all	1.124294	25502.226286	-	-
update-index-mixed	1.137270	14406.429432	-	-
update-batch-index-mixed	1.063708	23104.084956	-	-

Table B.32.: Database client benchmark run 02 of 10 (first Python, then C)

Name	rt in s (C)	# ops (C)	rt in s (Python)	# ops (Python)
insert	1.123853	14578.419064	0.260170857	50000
insert-batch	1.120610	29241.216837	0.144389467	50000
insert-index-single	1.167092	10528.732953	0.317355558	50000
insert-batch-index-single	1.183839	17299.649699	-	-
insert-index-all	1.247051	13138.195631	0.275852387	50000
insert-batch-index-all	1.019771	24099.528227	0.16412699	50000
insert-index-mixed	1.241468	9897.959512	0.336980862	50000
insert-batch-index-mixed	1.146776	14287.009843	0.221363191	50000
delete	1.229349	1665.922370	0.049005864	50000
delete-batch	1.168711	1752.357940	0.041173899	50000
delete-index-single	1.205789	13587.783601	0.317234667	50000
delete-batch-index-single	1.179315	20839.215986	0.208680738	50000
delete-index-all	1.036601	15805.502792	0.549001139	50000
delete-batch-index-all	1.096509	26148.440186	0.430995196	50000
delete-index-mixed	1.253080	13074.983241	-	-
delete-batch-index-mixed	1.066458	19203.756735	0.225374815	50000
update	1.009360	6087.025442	-	-
update-batch	1.009325	7101.775939	-	-
update-index-single	1.098137	14919.814194	-	-
update-batch-index-single	1.017610	24150.706066	-	-
update-index-all	1.078320	15194.005490	-	-
update-batch-index-all	1.138511	25183.770732	-	-
update-index-mixed	1.147188	14281.878820	-	-
update-batch-index-mixed	1.080600	22742.920600	-	-

Table B.33.: Database client benchmark run 03 of 10 (first Python, then C)

Name	rt in s (C)	# ops (C)	rt in s (Python)	# ops (Python)
insert	1.122505	14595.926076	0.260312057	50000
insert-batch	1.035099	27699.765916	0.150647092	50000
insert-index-single	1.161039	10583.623806	0.323823675	50000
insert-batch-index-single	1.002579	16341.854358	-	-
insert-index-all	1.237790	13236.494074	0.276526381	50000
insert-batch-index-all	1.068123	23008.586090	0.164774615	50000
insert-index-mixed	1.239828	9911.052178	0.347261841	50000
insert-batch-index-mixed	1.163264	14084.507042	0.229952232	50000
delete	1.226703	1669.515767	0.04833578	50000
delete-batch	1.166396	1755.835925	0.040613166	50000
delete-index-single	1.205079	13595.789156	0.322380427	50000
delete-batch-index-single	1.014616	20184.976385	0.208546477	50000
delete-index-all	1.044087	15692.178908	0.547841422	50000
delete-batch-index-all	1.163425	24644.476438	0.427457686	50000
delete-index-mixed	1.270892	12891.732736	-	-
delete-batch-index-mixed	1.156801	17703.995761	0.221798034	50000
update	1.033371	6193.322630	-	-
update-batch	1.010636	7092.563495	-	-
update-index-single	1.109530	14766.612890	-	-
update-batch-index-single	1.005861	24432.799363	-	-
update-index-all	1.090394	15025.761330	-	-
update-batch-index-all	1.156004	24802.682344	-	-
update-index-mixed	1.146317	14292.730545	-	-
update-batch-index-mixed	1.122764	21888.838616	-	-

Table B.34.: Database client benchmark run 04 of 10 (first Python, then C)

Name	rt in s (C)	# ops (C)	rt in s (Python)	# ops (Python)
insert	1.121766	14605.541619	0.264681911	50000
insert-batch	1.009989	28388.427993	0.149774086	50000
insert-index-single	1.181345	10401.703143	0.337360001	50000
insert-batch-index-single	1.164508	17586.826368	-	-
insert-index-all	1.221642	13411.457694	0.288938051	50000
insert-batch-index-all	1.030302	23853.200324	0.167006019	50000
insert-index-mixed	1.243516	9881.658137	0.362449904	50000
insert-batch-index-mixed	1.039954	15754.542989	0.225302636	50000
delete	1.210589	1691.738484	0.04860583	50000
delete-batch	1.162345	1761.955357	0.040844631	50000
delete-index-single	1.196766	13690.228499	0.325957039	50000
delete-batch-index-single	1.177266	20875.486084	0.225297016	50000
delete-index-all	1.029000	15922.254616	0.563747831	50000
delete-batch-index-all	1.091987	26256.722836	0.427058944	50000
delete-index-mixed	1.252418	13081.894384	-	-
delete-batch-index-mixed	1.044456	19608.293695	0.223458589	50000
update	1.001227	6136.470551	-	-
update-batch	1.008957	6850.638828	-	-
update-index-single	1.072395	15277.952620	-	-
update-batch-index-single	1.135304	25254.909698	-	-
update-index-all	1.058914	15472.455742	-	-
update-batch-index-all	1.121058	25575.839965	-	-
update-index-mixed	1.139276	14381.063061	-	-
update-batch-index-mixed	1.135518	21642.985844	-	-

Table B.35.: Database client benchmark run 05 of 10 (first Python, then C)

Name	rt in s (C)	# ops (C)	rt in s (Python)	# ops (Python)
insert	1.135507	14428.797004	0.265019756	50000
insert-batch	1.016089	28218.000589	0.151142111	50000
insert-index-single	1.162915	10566.550436	0.315117254	50000
insert-batch-index-single	1.137780	17999.964844	-	-
insert-index-all	1.227974	13342.302036	0.274070795	50000
insert-batch-index-all	1.044764	23523.015724	0.163116418	50000
insert-index-mixed	1.252487	9810.880273	0.326947349	50000
insert-batch-index-mixed	1.065543	15376.197863	0.219130839	50000
delete	1.214640	1686.096292	0.047779516	50000
delete-batch	1.163484	1760.230480	0.040656976	50000
delete-index-single	1.205079	13595.789156	0.325823174	50000
delete-batch-index-single	1.032322	19838.771236	0.204577899	50000
delete-index-all	1.055552	15521.736494	0.543239344	50000
delete-batch-index-all	1.113827	25741.879125	0.423332976	50000
delete-index-mixed	1.261920	12983.390389	-	-
delete-batch-index-mixed	1.041397	19665.891106	0.2191113	50000
update	1.027230	6230.347634	-	-
update-batch	1.014064	7068.587387	-	-
update-index-single	1.109832	14762.594699	-	-
update-batch-index-single	1.006035	24428.573559	-	-
update-index-all	1.061449	15435.503731	-	-
update-batch-index-all	1.140801	25133.217800	-	-
update-index-mixed	1.176376	13927.519773	-	-
update-batch-index-mixed	1.132343	21703.671061	-	-

Table B.36.: Database client benchmark run 06 of 10 (first C, then Python)

Name	rt in s (C)	# ops (C)	rt in s (Python)	# ops (Python)
insert	1.137477	14403.807725	0.358898098	50000
insert-batch	1.122148	29201.139244	0.148844339	50000
insert-index-single	1.200573	10235.112734	0.318527803	50000
insert-batch-index-single	1.133011	18075.729185	-	-
insert-index-all	1.234273	13274.210811	0.272427086	50000
insert-batch-index-all	1.029303	23876.351278	0.162738816	50000
insert-index-mixed	1.237894	9926.536521	0.331516445	50000
insert-batch-index-mixed	1.045401	15672.454876	0.223694245	50000
delete	1.215453	1684.968485	0.048331399	50000
delete-batch	1.189619	1721.559592	0.040789697	50000
delete-index-single	1.211272	13526.276509	0.322047032	50000
delete-batch-index-single	1.189816	20655.294600	0.20656594	50000
delete-index-all	1.038215	15780.931695	0.541394023	50000
delete-batch-index-all	1.108104	25874.827634	0.440125899	50000
delete-index-mixed	1.281471	12785.306886	-	-
delete-batch-index-mixed	1.060489	19311.845762	0.236640247	50000
update	1.023635	6252.228578	-	-
update-batch	1.035183	7171.678824	-	-
update-index-single	1.082361	15137.278597	-	-
update-batch-index-single	1.156691	24787.951147	-	-
update-index-all	1.065944	15370.413455	-	-
update-batch-index-all	1.157315	24774.586003	-	-
update-index-mixed	1.152275	14218.827971	-	-
update-batch-index-mixed	1.088653	22574.686333	-	-

Table B.37.: Database client benchmark run 07 of 10 (first C, then Python)

Name	rt in s (C)	# ops (C)	rt in s (Python)	# ops (Python)
insert	1.110912	14748.242885	0.254458586	50000
insert-batch	1.108448	29562.054332	0.147823982	50000
insert-index-single	1.146421	10718.575462	0.317693208	50000
insert-batch-index-single	1.007331	16264.763022	-	-
insert-index-all	1.219228	13438.011594	0.271649492	50000
insert-batch-index-all	1.036066	23720.496571	0.165788523	50000
insert-index-mixed	1.232319	9971.444082	0.341741122	50000
insert-batch-index-mixed	1.072781	15272.455422	0.227809318	50000
delete	1.213737	1687.350719	0.048243599	50000
delete-batch	1.160546	1764.686622	0.042072864	50000
delete-index-single	1.185623	13818.895214	0.318846964	50000
delete-batch-index-single	1.142378	21513.019333	0.205081912	50000
delete-index-all	1.032729	15864.762198	0.541654565	50000
delete-batch-index-all	1.122321	25547.058284	0.428183433	50000
delete-index-mixed	1.245148	13158.275161	-	-
delete-batch-index-mixed	1.033427	19817.558473	0.221928845	50000
update	1.013607	6314.084256	-	-
update-batch	1.005974	7125.432665	-	-
update-index-single	1.073168	15266.947952	-	-
update-batch-index-single	1.116499	25680.273784	-	-
update-index-all	1.048422	15627.295116	-	-
update-batch-index-all	1.099725	26071.972539	-	-
update-index-mixed	1.133422	14455.339670	-	-
update-batch-index-mixed	1.052672	23346.303502	-	-

Table B.38.: Database client benchmark run 08 of 10 (first C, then Python)

Name	rt in s (C)	# ops (C)	rt in s (Python)	# ops (Python)
insert	1.177018	13919.923060	0.257456997	50000
insert-batch	1.101201	29756.602110	0.142997787	50000
insert-index-single	1.270034	9675.331527	0.322777227	50000
insert-batch-index-single	1.141933	17934.502287	-	-
insert-index-all	1.003877	12240.543413	0.273704516	50000
insert-batch-index-all	1.051520	23371.880706	0.160156002	50000
insert-index-mixed	1.307726	9396.463785	0.34148111	50000
insert-batch-index-mixed	1.059564	15462.964012	0.21984403	50000
delete	1.215197	1685.323450	0.050411057	50000
delete-batch	1.167254	1754.545283	0.041590029	50000
delete-index-single	1.253517	13070.425052	0.32873277	50000
delete-batch-index-single	1.027879	19924.524190	0.209446603	50000
delete-index-all	1.083885	15115.994778	0.55040861	50000
delete-batch-index-all	1.118119	25643.066615	0.430088988	50000
delete-index-mixed	1.281858	12781.446931	-	-
delete-batch-index-mixed	1.061170	19299.452491	0.227929725	50000
update	1.030922	6208.035137	-	-
update-batch	1.006333	6359.723869	-	-
update-index-single	1.092752	14993.337921	-	-
update-batch-index-single	1.123702	25515.661626	-	-
update-index-all	1.073531	15261.785640	-	-
update-batch-index-all	1.121451	25566.877197	-	-
update-index-mixed	1.154749	14188.364744	-	-
update-batch-index-mixed	1.094721	22449.555640	-	-

Table B.39.: Database client benchmark run 09 of 10 (first C, then Python)

Name	rt in s (C)	# ops (C)	rt in s (Python)	# ops (Python)
insert	1.208393	13558.502904	0.262484515	50000
insert-batch	1.111760	29473.987191	0.145877413	50000
insert-index-single	1.194451	10287.571445	0.323699232	50000
insert-batch-index-single	1.132095	18090.354608	-	-
insert-index-all	1.226328	13360.210319	0.276691223	50000
insert-batch-index-all	1.024129	23996.976943	0.162914147	50000
insert-index-mixed	1.236191	9940.211505	0.332128208	50000
insert-batch-index-mixed	1.041822	15726.294895	0.220795436	50000
delete	1.218171	1681.208960	0.049993045	50000
delete-batch	1.190189	1720.735110	0.040663874	50000
delete-index-single	1.207475	13568.810948	0.329878106	50000
delete-batch-index-single	1.183463	20766.175199	0.207560553	50000
delete-index-all	1.025534	15976.067103	0.564161475	50000
delete-batch-index-all	1.085201	26420.911886	0.442808588	50000
delete-index-mixed	1.275857	12841.564533	-	-
delete-batch-index-mixed	1.044143	19614.171622	0.221933822	50000
update	1.022474	6259.327866	-	-
update-batch	1.001885	7154.513742	-	-
update-index-single	1.084148	15112.327837	-	-
update-batch-index-single	1.001729	24533.581438	-	-
update-index-all	1.060571	15448.282105	-	-
update-batch-index-all	1.145899	25021.402410	-	-
update-index-mixed	1.140723	14362.820772	-	-
update-batch-index-mixed	1.098151	22379.435979	-	-

Table B.40.: Database client benchmark run 10 of 10 (first C, then Python)

Name	rt in s (C)	# ops (C)	rt in s (Python)	# ops (Python)
get-simple	1.015076	6304.946625	-	-
get-simple-index-single	1.122640	32836.884487	-	-
get-simple-index-all	1.002673	32680.644637	-	-
get-simple-index-mixed	1.026928	31908.760887	-	-
get-range	1.005576	2784.473774	0.005793543	50000
get-range-index-single	1.002074	3672.383477	-	-
get-range-index-all	1.000422	2734.845895	0.004785138	50000
get-range-index-mixed	1.003775	3745.859381	0.004403856	50000

Table B.41.: Database iterator benchmark run 01 of 10 (first Python, then C)

Name	rt in s (C)	# ops (C)	rt in s (Python)	# ops (Python)
get-simple	1.003012	6380.781087	-	-
get-simple-index-single	1.110084	33208.297750	-	-
get-simple-index-all	1.107720	33279.168021	-	-
get-simple-index-mixed	1.104680	33370.749900	-	-
get-range	1.003158	2823.084699	0.005774294	50000
get-range-index-single	1.001652	3578.088997	-	-
get-range-index-all	1.001758	2827.030081	0.004697657	50000
get-range-index-mixed	1.001225	3771.380059	0.004886128	50000

Table B.42.: Database iterator benchmark run 02 of 10 (first Python, then C)

Name	rt in s (C)	# ops (C)	rt in s (Python)	# ops (Python)
get-simple	1.029219	6218.307280	-	-
get-simple-index-single	1.110380	33199.445235	-	-
get-simple-index-all	1.114000	33091.561939	-	-
get-simple-index-mixed	1.117098	32999.790529	-	-
get-range	1.004987	2802.026295	0.005925994	50000
get-range-index-single	1.003467	3619.451362	-	-
get-range-index-all	1.001456	2779.952389	0.004473906	50000
get-range-index-mixed	1.001468	3674.605679	0.005010539	50000

Table B.43.: Database iterator benchmark run 03 of 10 (first Python, then C)

Name	rt in s (C)	# ops (C)	rt in s (Python)	# ops (Python)
get-simple	1.003253	6379.248305	-	-
get-simple-index-single	1.108359	33259.981649	-	-
get-simple-index-all	1.104291	33382.505155	-	-
get-simple-index-mixed	1.110030	33209.913246	-	-
get-range	1.002275	2745.753411	0.005935644	50000
get-range-index-single	1.000284	3694.950634	-	-
get-range-index-all	1.003064	2839.300384	0.004710328	50000
get-range-index-mixed	1.000988	3724.320371	0.004843178	50000

Table B.44.: Database iterator benchmark run 04 of 10 (first Python, then C)

Name	rt in s (C)	# ops (C)	rt in s (Python)	# ops (Python)
get-simple	1.008773	6344.341096	-	-
get-simple-index-single	1.119915	29259.363434	-	-
get-simple-index-all	1.073851	22885.856604	-	-
get-simple-index-mixed	1.076139	22837.198540	-	-
get-range	1.004969	2722.472036	0.005812962	50000
get-range-index-single	1.004195	3489.362126	-	-
get-range-index-all	1.003496	2710.524008	0.004564847	50000
get-range-index-mixed	1.002255	3512.080259	0.004903648	50000

Table B.45.: Database iterator benchmark run 05 of 10 (first Python, then C)

Name	rt in s (C)	# ops (C)	rt in s (Python)	# ops (Python)
get-simple	1.012461	6321.231139	-	-
get-simple-index-single	1.002904	32673.117267	-	-
get-simple-index-all	1.019041	32155.722881	-	-
get-simple-index-mixed	1.112650	33131.712578	-	-
get-range	1.002232	2857.621788	0.006218685	50000
get-range-index-single	1.001167	3707.673145	-	-
get-range-index-all	1.003521	2822.063514	0.004686006	50000
get-range-index-mixed	1.000407	3774.463793	0.004875017	50000

Table B.46.: Database iterator benchmark run 06 of 10 (first C, then Python)

Name	rt in s (C)	# ops (C)	rt in s (Python)	# ops (Python)
get-simple	1.037898	6412.961582	-	-
get-simple-index-single	1.000299	32758.205297	-	-
get-simple-index-all	1.121708	32864.167858	-	-
get-simple-index-mixed	1.092310	29998.809862	-	-
get-range	1.002723	2712.613553	0.006049294	50000
get-range-index-single	1.003211	3444.938303	-	-
get-range-index-all	1.004548	2643.975201	0.004706977	50000
get-range-index-mixed	1.002084	3528.646301	0.004822336	50000

Table B.47.: Database iterator benchmark run 07 of 10 (first C, then Python)

Name	rt in s (C)	# ops (C)	rt in s (Python)	# ops (Python)
get-simple	1.009160	6341.908122	-	-
get-simple-index-single	1.109673	33220.597419	-	-
get-simple-index-all	1.121225	32878.325046	-	-
get-simple-index-mixed	1.017546	32202.966745	-	-
get-range	1.001279	2684.566440	0.005852052	50000
get-range-index-single	1.003812	3522.571956	-	-
get-range-index-all	1.001553	2747.732771	0.004446955	50000
get-range-index-mixed	1.002948	3573.465424	0.004687616	50000

Table B.48.: Database iterator benchmark run 08 of 10 (first C, then Python)

Name	rt in s (C)	# ops (C)	rt in s (Python)	# ops (Python)
get-simple	1.012991	5559.772989	-	-
get-simple-index-single	1.033473	31706.682226	-	-
get-simple-index-all	1.015069	32281.549333	-	-
get-simple-index-mixed	1.011169	32406.056752	-	-
get-range	1.002989	2775.703422	0.006262855	50000
get-range-index-single	1.002578	3542.866490	-	-
get-range-index-all	1.005180	2753.735649	0.004768326	50000
get-range-index-mixed	1.000594	3629.843873	0.004714636	50000

Table B.49.: Database iterator benchmark run 09 of 10 (first C, then Python)

Name	rt in s (C)	# ops (C)	rt in s (Python)	# ops (Python)
get-simple	1.003522	6377.538310	-	-
get-simple-index-single	1.000019	32767.377420	-	-
get-simple-index-all	1.100975	33483.049116	-	-
get-simple-index-mixed	1.107313	33291.399993	-	-
get-range	1.001609	2811.476335	0.006010933	50000
get-range-index-single	1.003069	3588.985404	-	-
get-range-index-all	1.001665	2747.425536	0.004748406	50000
get-range-index-mixed	1.002832	3557.923959	0.004914027	50000

Table B.50.: Database iterator benchmark run 10 of 10 (first C, then Python)

Name	rt in s (C)	# ops (C)	rt in s (Python)	# ops (Python)
create	1.029087	2623.684878	-	-
create-batch	1.001578	2595.903664	-	-
delete	1.013506	6512.048276	9.152204514	5000
delete-batch	1.006372	6657.577914	9.203082379	5000

Table B.51.: Database schema benchmark run 01 of 10 (first Python, then C)

Name	rt in s (C)	# ops (C)	rt in s (Python)	# ops (Python)
create	1.032954	2613.862766	-	-
create-batch	1.025201	2633.629893	-	-
delete	1.002267	6585.071643	9.265281469	5000
delete-batch	1.009600	6636.291601	9.240100433	5000

Table B.52.: Database schema benchmark run 02 of 10 (first Python, then C)

Name	rt in s (C)	# ops (C)	rt in s (Python)	# ops (Python)
create	1.036487	2604.953077	-	-
create-batch	1.022293	2641.121479	-	-
delete	1.007250	6552.494415	9.425406606	5000
delete-batch	1.008318	6545.554081	9.287570188	5000

Table B.53.: Database schema benchmark run 03 of 10 (first Python, then C)

Name	rt in s (C)	# ops (C)	rt in s (Python)	# ops (Python)
create	1.036413	2605.139071	-	-
create-batch	1.026246	2630.948135	-	-
delete	1.001446	6390.758963	9.22054927	5000
delete-batch	1.002811	6581.499405	9.304047822	5000

Table B.54.: Database schema benchmark run 04 of 10 (first Python, then C)

Name	rt in s (C)	# ops (C)	rt in s (Python)	# ops (Python)
create	1.009233	2477.128671	-	-
create-batch	1.021254	2545.889661	-	-
delete	1.007613	6450.889379	9.380357572	5000
delete-batch	1.007841	6647.874020	9.497660757	5000

Table B.55.: Database schema benchmark run 05 of 10 (first Python, then C)

Name	rt in s (C)	# ops (C)	rt in s (Python)	# ops (Python)
create	1.028520	2625.131257	-	-
create-batch	1.028139	2626.104058	-	-
delete	1.011324	6526.098461	9.315243824	5000
delete-batch	1.002176	6585.669583	9.244126538	5000

Table B.56.: Database schema benchmark run 06 of 10 (first C, then Python)

Name	rt in s (C)	# ops (C)	rt in s (Python)	# ops (Python)
create	1.009331	2476.888157	-	-
create-batch	1.021973	2641.948466	-	-
delete	1.001401	6590.766336	9.316446728	5000
delete-batch	1.006484	6557.481291	9.405501374	5000

Table B.57.: Database schema benchmark run 07 of 10 (first C, then Python)

Name	rt in s (C)	# ops (C)	rt in s (Python)	# ops (Python)
create	1.012898	2665.618848	-	-
create-batch	1.007734	2679.278460	-	-
delete	1.004422	6570.943289	9.070532319	5000
delete-batch	1.008463	6643.773743	9.007394258	5000

Table B.58.: Database schema benchmark run 08 of 10 (first C, then Python)

Name	rt in s (C)	# ops (C)	rt in s (Python)	# ops (Python)
create	1.022851	2639.680657	-	-
create-batch	1.005639	2585.420812	-	-
delete	1.003572	6576.508711	9.272569606	5000
delete-batch	1.014671	6603.125545	9.229454808	5000

Table B.59.: Database schema benchmark run 09 of 10 (first C, then Python)

Name	rt in s (C)	# ops (C)	rt in s (Python)	# ops (Python)
create	1.032827	2614.184176	-	-
create-batch	1.024136	2636.368607	-	-
delete	1.010318	6631.575405	9.251996872	5000
delete-batch	1.012122	6619.755326	9.173609651	5000

Table B.60.: Database schema benchmark run 10 of 10 (first C, then Python)

Name	rt in s (C)	# ops (C)	rt in s (Python)	# ops (Python)
create	1.013095	30599.302138	2.169573875	50000
create-batch	1.006188	31803.201787	1.855203734	50000
delete	1.038200	22153.727605	2.908732426	50000
delete-batch	1.037788	23126.110535	2.625526986	50000
delete-batch-without-get	1.010656	34630.972359	-	-
get-status	1.011153	75161.721322	0.720908495	50000
get-status-batch	1.001852	459149.654839	1.243675882	500000
read	1.002047	58879.473717	0.985105125	50000
read-batch	1.004780	288620.394514	1.971762559	500000
write	1.001947	63875.634140	0.850016613	50000
write-batch	1.012876	602245.487108	1.118814329	500000
unordered-create-delete	1.011127	31647.854325	3.674608321	100000
unordered-create-delete-batch	1.007023	33762.883271	3.648923842	100000

Table B.61.: Item client benchmark run 01 of 10 (first Python, then C)

Name	rt in s (C)	# ops (C)	rt in s (Python)	# ops (Python)
create	1.002471	30923.587815	2.341092679	50000
create-batch	1.003438	31890.360939	2.089084829	50000
delete	1.016328	22630.489370	3.317527503	50000
delete-batch	1.013343	22697.151902	2.770494585	50000
delete-batch-without-get	1.007211	33756.581292	-	-
get-status	1.001529	74885.500070	0.71302294	50000
get-status-batch	1.008954	455917.712800	1.265658749	500000
read	1.009895	58421.915150	0.935719209	50000
read-batch	1.003124	289096.861405	1.984334282	500000
write	1.014239	65073.419579	0.906096337	50000
write-batch	1.020220	597910.254651	1.187646898	500000
unordered-create-delete	1.003883	31876.224620	3.779005186	100000
unordered-create-delete-batch	1.028257	33065.663545	3.630305734	100000

Table B.62.: Item client benchmark run 02 of 10 (first Python, then C)

Name	rt in s (C)	# ops (C)	rt in s (Python)	# ops (Python)
create	1.026705	29219.688226	2.002421007	50000
create-batch	1.017180	31459.525354	2.175747181	50000
delete	1.017689	22600.224627	2.700416599	50000
delete-batch	1.025541	23402.282308	2.551490116	50000
delete-batch-without-get	1.031699	32955.348411	-	-
get-status	1.011009	73194.204997	0.704178698	50000
get-status-batch	1.008894	446032.982652	1.258468168	500000
read	1.015025	58126.647127	0.919880488	50000
read-batch	1.003272	289054.214610	2.032550831	500000
write	1.004264	64724.016792	0.887778751	50000
write-batch	1.014565	591386.456265	1.178246594	500000
unordered-create-delete	1.051415	32337.373920	3.868873036	100000
unordered-create-delete-batch	1.015929	33466.905660	3.552927567	100000

Table B.63.: Item client benchmark run 03 of 10 (first Python, then C)

Name	rt in s (C)	# ops (C)	rt in s (Python)	# ops (Python)
create	1.012724	29623.075981	2.074989337	50000
create-batch	1.014905	30544.730788	1.887624362	50000
delete	1.034344	22236.315965	2.776367839	50000
delete-batch	1.034363	23202.686098	2.620235625	50000
delete-batch-without-get	1.005737	33806.054664	-	-
get-status	1.009630	75275.100779	0.707792559	50000
get-status-batch	1.012424	454355.092333	1.291114108	500000
read	1.003203	52830.783002	0.916498488	50000
read-batch	1.004501	288700.558785	1.967215983	500000
write	1.013206	65139.764273	0.869595466	50000
write-batch	1.010303	593881.241568	1.12055885	500000
unordered-create-delete	1.048727	28606.110074	3.781982923	100000
unordered-create-delete-batch	1.013037	33562.446386	3.585668866	100000

Table B.64.: Item client benchmark run 04 of 10 (first Python, then C)

Name	rt in s (C)	# ops (C)	rt in s (Python)	# ops (Python)
create	1.002894	29913.430532	1.958333944	50000
create-batch	1.007038	31776.357992	1.92217356	50000
delete	1.020012	22548.754328	2.716320335	50000
delete-batch	1.023818	23441.666390	2.60445936	50000
delete-batch-without-get	1.016010	33464.237557	-	-
get-status	1.006051	74548.904578	0.709766173	50000
get-status-batch	1.003631	448371.961408	1.240500341	500000
read	1.007662	59543.775591	0.936881865	50000
read-batch	1.005083	288533.384805	1.995314691	500000
write	1.011185	66258.894268	0.898351382	50000
write-batch	1.004385	617293.169452	1.122448504	500000
unordered-create-delete	1.033398	32901.166830	3.655714909	100000
unordered-create-delete-batch	1.062661	30113.084041	3.472859161	100000

Table B.65.: Item client benchmark run 05 of 10 (first Python, then C)

Name	rt in s (C)	# ops (C)	rt in s (Python)	# ops (Python)
create	1.003538	29894.234199	2.185745597	50000
create-batch	1.004009	30876.217245	1.87190537	50000
delete	1.020921	22528.677537	2.716934534	50000
delete-batch	1.030219	23296.017643	2.650656465	50000
delete-batch-without-get	1.022533	34228.724159	-	-
get-status	1.010228	74240.666463	0.713436432	50000
get-status-batch	1.011062	454967.153350	1.242386546	500000
read	1.009845	57434.556788	0.947043552	50000
read-batch	1.027945	282116.261084	1.997234397	500000
write	1.012659	65174.950304	0.887020627	50000
write-batch	1.000950	609421.050002	1.19819833	500000
unordered-create-delete	1.017558	31447.838846	3.762740656	100000
unordered-create-delete-batch	1.038019	32754.699095	3.524058167	100000

Table B.66.: Item client benchmark run 06 of 10 (first C, then Python)

Name	rt in s (C)	# ops (C)	rt in s (Python)	# ops (Python)
create	1.023321	30293.524710	2.030159478	50000
create-batch	1.006486	31793.785507	1.987866032	50000
delete	1.013296	22698.204671	2.68463184	50000
delete-batch	1.026393	22408.570596	2.512571994	50000
delete-batch-without-get	1.019157	33360.905140	-	-
get-status	1.008969	75324.415319	0.69904755	50000
get-status-batch	1.003268	458501.616717	1.248053006	500000
read	1.012056	55332.906479	0.964658869	50000
read-batch	1.010940	286861.732645	1.988732427	500000
write	1.008765	64435.225251	0.858985151	50000
write-batch	1.002722	608344.087394	1.149797619	500000
unordered-create-delete	1.042916	30683.199797	3.516689141	100000
unordered-create-delete-batch	1.059780	32082.130253	3.405623064	100000

Table B.67.: Item client benchmark run 07 of 10 (first C, then Python)

Name	rt in s (C)	# ops (C)	rt in s (Python)	# ops (Python)
create	1.001755	30945.690313	1.971541692	50000
create-batch	1.023246	31273.027209	2.029231443	50000
delete	1.019572	22558.485325	2.650029058	50000
delete-batch	1.010282	23755.743446	2.549024066	50000
delete-batch-without-get	1.003716	34870.421514	-	-
get-status	1.002729	75793.160465	0.70074421	50000
get-status-batch	1.014986	453208.221591	1.240265372	500000
read	1.015343	59093.331022	0.920461112	50000
read-batch	1.026368	282549.728752	2.001872511	500000
write	1.001488	63904.909495	0.871902532	50000
write-batch	1.008060	605122.710950	1.117922301	500000
unordered-create-delete	1.013040	29613.835584	3.541206088	100000
unordered-create-delete-batch	1.054883	30335.117733	3.397092145	100000

Table B.68.: Item client benchmark run 08 of 10 (first C, then Python)

Name	rt in s (C)	# ops (C)	rt in s (Python)	# ops (Python)
create	1.003337	29900.222956	1.99456108	50000
create-batch	1.003412	31891.187269	1.934868211	50000
delete	1.020474	22538.545813	2.781740234	50000
delete-batch	1.028594	23332.821308	2.619882632	50000
delete-batch-without-get	1.019960	34315.071179	-	-
get-status	1.012156	74099.249523	0.718139823	50000
get-status-batch	1.012719	444348.333546	1.274382476	500000
read	1.005880	57660.953593	0.960711469	50000
read-batch	1.016797	285209.338737	2.018607742	500000
write	1.003206	65789.080209	0.880081265	50000
write-batch	1.007819	605267.414089	1.142245308	500000
unordered-create-delete	1.056467	32182.737369	3.821740379	100000
unordered-create-delete-batch	1.029603	33022.436803	3.610422556	100000

Table B.69.: Item client benchmark run 09 of 10 (first C, then Python)

Name	rt in s (C)	# ops (C)	rt in s (Python)	# ops (Python)
create	1.013638	30582.910270	2.197780659	50000
create-batch	1.008831	30728.635421	1.932458919	50000
delete	1.008370	22809.087934	2.664320757	50000
delete-batch	1.006217	21864.071070	2.570346369	50000
delete-batch-without-get	1.005356	33818.866153	-	-
get-status	1.005681	75570.682950	0.716205042	50000
get-status-batch	1.008858	455961.096606	1.259520613	500000
read	1.007481	58561.898438	0.920332757	50000
read-batch	1.005823	288321.106199	1.993610495	500000
write	1.012747	65169.287097	0.885068263	50000
write-batch	1.009434	604299.042830	1.148150751	500000
unordered-create-delete	1.047633	30545.047741	3.610953648	100000
unordered-create-delete-batch	1.030331	31057.980397	3.44591441	100000

Table B.70.: Item client benchmark run 10 of 10 (first C, then Python)

Name	rt in s (C)	# ops (C)	rt in s (Python)	# ops (Python)
create	1.015829	64971.565096	0.964078983	50000
create-batch	1.002810	67809.455430	0.944888412	50000
delete	1.009679	36645.310044	1.690278758	50000
delete-batch	1.026561	37016.796859	1.641598457	50000
delete-batch-without-get	1.005135	70637.277580	0.832977544	50000
unordered-create-delete	1.012418	71116.870700	1.73715345	100000
unordered-create-delete-batch	1.011243	73177.267976	1.642818385	100000

Table B.71.: Item collection benchmark run 01 of 10 (first Python, then C)

Name	rt in s (C)	# ops (C)	rt in s (Python)	# ops (Python)
create	1.014765	66025.138825	1.03072687	50000
create-batch	1.007165	68509.132069	0.946753073	50000
delete	1.009567	37639.899085	1.775482132	50000
delete-batch	1.027216	36993.193252	1.697733542	50000
delete-batch-without-get	1.000424	69970.332579	0.82884914	50000
unordered-create-delete	1.017887	72699.621864	2.216467586	100000
unordered-create-delete-batch	1.024825	74159.002757	1.936511345	100000

Table B.72.: Item collection benchmark run 02 of 10 (first Python, then C)

Name	rt in s (C)	# ops (C)	rt in s (Python)	# ops (Python)
create	1.000393	63974.857881	1.381978263	50000
create-batch	1.005948	68592.014697	0.964619326	50000
delete	1.000212	36992.157663	2.222544552	50000
delete-batch	1.011510	37567.596959	2.174367392	50000
delete-batch-without-get	1.000521	70963.028262	0.864549797	50000
unordered-create-delete	1.002399	71827.685383	1.777054692	100000
unordered-create-delete-batch	1.019547	72581.254224	1.678449421	100000

Table B.73.: Item collection benchmark run 03 of 10 (first Python, then C)

Name	rt in s (C)	# ops (C)	rt in s (Python)	# ops (Python)
create	1.012431	65189.627738	0.954272116	50000
create-batch	1.010724	69257.284877	0.945026812	50000
delete	1.018899	37295.158794	1.701326073	50000
delete-batch	1.011021	36596.668121	1.630410262	50000
delete-batch-without-get	1.009167	71345.971479	0.828976441	50000
unordered-create-delete	1.018561	72651.515226	1.728576732	100000
unordered-create-delete-batch	1.009954	73270.663812	1.639031082	100000

Table B.74.: Item collection benchmark run 04 of 10 (first Python, then C)

Name	rt in s (C)	# ops (C)	rt in s (Python)	# ops (Python)
create	1.010383	65321.764123	0.985475485	50000
create-batch	1.013683	68068.617112	0.945686327	50000
delete	1.020994	37218.632039	1.672618364	50000
delete-batch	1.012836	37518.413642	1.633147747	50000
delete-batch-without-get	1.013532	69065.406914	0.881613509	50000
unordered-create-delete	1.004977	73633.525941	1.731332401	100000
unordered-create-delete-batch	1.025771	74090.610867	1.66630413	100000

Table B.75.: Item collection benchmark run 05 of 10 (first Python, then C)

Name	rt in s (C)	# ops (C)	rt in s (Python)	# ops (Python)
create	1.002468	64839.974942	0.983727046	50000
create-batch	1.014531	67026.044547	0.954407943	50000
delete	1.019433	36294.685379	1.694750294	50000
delete-batch	1.011254	36588.235992	1.63642152	50000
delete-batch-without-get	1.002977	70789.260372	0.833845512	50000
unordered-create-delete	1.006778	71515.269503	1.740923131	100000
unordered-create-delete-batch	1.021406	74407.238649	1.660687804	100000

Table B.76.: Item collection benchmark run 06 of 10 (first C, then Python)

Name	rt in s (C)	# ops (C)	rt in s (Python)	# ops (Python)
create	1.000307	65979.744219	0.985902386	50000
create-batch	1.011198	68235.894454	0.95947687	50000
delete	1.003221	36881.205637	1.662815064	50000
delete-batch	1.024487	38067.832974	1.62255865	50000
delete-batch-without-get	1.000939	69934.331663	0.834659395	50000
unordered-create-delete	1.008432	71397.972298	1.774117913	100000
unordered-create-delete-batch	1.006055	73554.626735	1.659742266	100000

Table B.77.: Item collection benchmark run 07 of 10 (first C, then Python)

Name	rt in s (C)	# ops (C)	rt in s (Python)	# ops (Python)
create	1.001752	64886.319169	0.974410003	50000
create-batch	1.002039	67861.630136	0.95225892	50000
delete	1.000480	35982.728290	1.655338643	50000
delete-batch	1.003556	36868.894212	1.636078765	50000
delete-batch-without-get	1.013702	71026.790911	0.844815812	50000
unordered-create-delete	1.024901	72202.095617	1.73564352	100000
unordered-create-delete-batch	1.005590	75577.521654	1.631170637	100000

Table B.78.: Item collection benchmark run 08 of 10 (first C, then Python)

Name	rt in s (C)	# ops (C)	rt in s (Python)	# ops (Python)
create	1.006320	64591.779951	0.946360019	50000
create-batch	1.013732	66092.418904	0.941110861	50000
delete	1.008149	36700.924169	1.677333544	50000
delete-batch	1.024503	37091.155419	1.606081061	50000
delete-batch-without-get	1.001210	70914.193825	0.820727578	50000
unordered-create-delete	1.001203	71913.488074	1.723516028	100000
unordered-create-delete-batch	1.020312	74487.019657	1.636964461	100000

Table B.79.: Item collection benchmark run 09 of 10 (first C, then Python)

Name	rt in s (C)	# ops (C)	rt in s (Python)	# ops (Python)
create	1.007411	65514.472246	1.004034167	50000
create-batch	1.007934	68456.863247	0.968520641	50000
delete	1.020359	37241.794310	1.662891744	50000
delete-batch	1.008036	37697.066375	1.591768822	50000
delete-batch-without-get	1.002607	69817.984514	0.833920141	50000
unordered-create-delete	1.004227	73688.518632	1.800786904	100000
unordered-create-delete-batch	1.006008	75546.118918	1.703256096	100000

Table B.80.: Item collection benchmark run 10 of 10 (first C, then Python)

Statement of Authorship

I herewith assure that I wrote the present thesis independently, that the thesis has not been partially or fully submitted as graded academic work and that I have used no other means than the ones indicated. I have indicated all parts of the work in which sources are used according to their wording or to their meaning.

I am aware of the fact that violations of copyright can lead to injunctive relief and claims for damages of the author as well as a penalty by the law enforcement agency.

Halle (Saale), January 22, 2023

____

Signature