A3 Post Mortem
&
Networking API

Networking

```c
#include <unistd.h>
#include <stdio.h>
#include <sys/socket.h>
#include <stdlib.h>
#include <netinet/in.h>
#include <string.h>
```

https://www.geeksforgeeks.org/socket-programming-cc/

```c
int main()  {
    int server_fd, sock;
    struct sockaddr_in adr;
    int opt = 1;
    int addrlen = sizeof(adr);
    char buffer[1024];
    char *hello = "Hello from server";
    // Creating socket file descriptor
    assert((server_fd = socket(AF_INET, SOCK_STREAM, 0)) != 0)l
    // Forcefully attaching socket to the port 8080
    assert(setsockopt(server_fd, SOL_SOCKET,
      SO_REUSEADDR | SO_REUSEPORT, &opt, sizeof(opt)) != 0);
    adr.sin_family = AF_INET;
    adr.sin_addr.s_addr = INADDR_ANY;
    address.sin_port = htons( 8080 );
    // Attaching socket to the port 8080
    assert(bind(server_fd, (struct sockaddr *)&adr,
                                  sizeof(adr))>=0);
    assert(listen(server_fd, 3) >= 0);
    assert((sock = accept(server_fd, (struct sockaddr*)&adr,
                        (socklen_t*)&addrlen))>=0);
    int read = read(sock, buffer, 1024);
    printf("%s\n",buffer );
    send(sock, hello , strlen(hello) , 0 );
    printf("Hello message sent\n");
}
```

```c
int main() {
    int sock = 0;
    struct sockaddr_in serv;
    char *hello = "Hello from client";
    char buffer[1024] = {0};
    assert((sock = socket(AF_INET, SOCK_STREAM, 0)) >= 0);
    serv.sin_family = AF_INET;
    serv.sin_port = htons(PORT);
    // Convert IP addresses from text to binary form
    assert(inet_pton(AF_INET, "127.0.0.1",&serv.sin_addr)>0);
    assert(connect(sock,
                   (struct sockaddr *)&serv,
                    sizeof(serv)) >= 0);
    send(sock , hello , strlen(hello) , 0 );
    printf("Hello message sent\n");
    int valread = read( sock , buffer, 1024);
    printf("%s\n",buffer );
}
```

P1

- **What is the goal of the memo?**
    - Help pick a solution
    - Get the reader to believe you

This is your first report for SwDv and the first time you measured performance, grading took this into account. Expectations increase over time.

Writing

- **What to avoid?**
  - Typos & bad grammar
  - Getting lost in a sea of prose
  - First person writing
  - Humor

Argumentation

- **What to avoid?**
  - Unsupported claims
  - Non-sequiturs
  - Irrelevant observations

Description of the analysis

- Explain what you are measuring

- Explain how you measured it

Comparison of relative performance

- Give results
- Explain results
- Every graph is explained in text
- Text can be read without looking at graphs
- Graphs can be understood without reading text

Introduction

Introduction

In this report, we will be presenting our analysis of six different data adapter implementations for SoR files, with the goal of ultimately determining which would be best suited for use in our project. In doing this analysis, we will be looking into both performance (as measured by time and memory usage) and overall code quality.

The data adapters we have chosen to analyze were created by the following teams: rustaceans (Rust), Anonymous (C++), Prizes! (C++), the segfault in our stars (C++), bnullb (Scala), and Purgatory (Python).

Introduction:

The goal of this report is to assess the performance and quality of code provided by the six teams. We decided that performance for a data adapter is the measured output of time required to execute. Through multiple measurements from a set of data-adapters, we formulated a relative baseline to define excellent performance. To uphold impartiality, we decided to run all the tests on one machine. The machine we used was Jan 's mid-2014 MacBook Pro; it had 2.6 GHz Dual-Core Intel Core i5 processor, 16 GB 1600 MHz DDR3 memory, and Intel Iris 1536 MB graphics chip. The machine ran on macOS Catalina, version 10.15.2. To make sure that we were getting optimal results, we made sure that the machine's charger was plugged during the entire duration of testing. We used Microsoft Visual Studio Code as our source-code editor and ran the tests in the VS Code integrated terminal. While running our tests, we made sure that no other applications were running.

For our tests we looked at the runtime, the accuracy, the memory leakage, and the actual functionality of the tests. Additionally we looked at the quality of the code, and the ease in which it could be made and integrated with our testing environment to make our analysis. Many of the executables provided did not behave according to the agreed upon spec, so the group was unable to construct a scalable test infrastructure to reliably gauge the performance of all groups.

The Makefile the group used was able to build all targets successfully: the six executables from the six groups. However when running tests many of the executables had different command line behavior which resulted in needing to tweak our tests to fit each group's arbitrary requirements, such as the order of command line arguments.

The group initially had difficulty running tests of certain groups which made sorers in different languages. These did not have the same behavior that we expected as for the python files we were unable to run ./sorer from the command line, so had to abandon two groups and choose new ones to run our tests on.

Groups tests:
Group 1 = https://github.com/csstransky/cs4500-assignment-1
Group 2 = https://github.com/Nomenokes/cs4500asSor
Group 3 = https://github.com/az0977776/cs4500-a1p1
Group 4 = https://github.com/Xiaocheng-Zhang/A1P1
Group 5 = https://github.com/ZhichaoC/A3Prize/
Group 6 = https://github.com/yth/CS4500A1P1

Description of the analysis

Description of the analysis.

The first step is experimental design. As we have no information on how the adapters will be used, we create a *synthetic* benchmark suite. There are two scenarios, or use-cases, where performance of a data adapter may matter: (a) if it is invoked very frequently on small files, (b) if it is invoked on very large files. For any other use case, performance is mostly irrelevant. So our experiment will consist of reading small files multiple times and reading a single large file. As we don't know what data types will be most frequently used, we will assess if the choice of data type matters for performance.

For pragmatic reasons we limit our performance study to a single hardware and software configuration. The numbers we report were obtained on a desktop running macOS Catalina. The programs were compiled locally, and run outside of Docker.

We performed a *single-build* of each system, and measured multiple *runs* of that build. A limitation of this experiment is that each run experiences startup costs. For the large file scenario startup costs are likely amortized. But for the small file scenario, it would be preferable to measure multiple *iterations* of the systems.

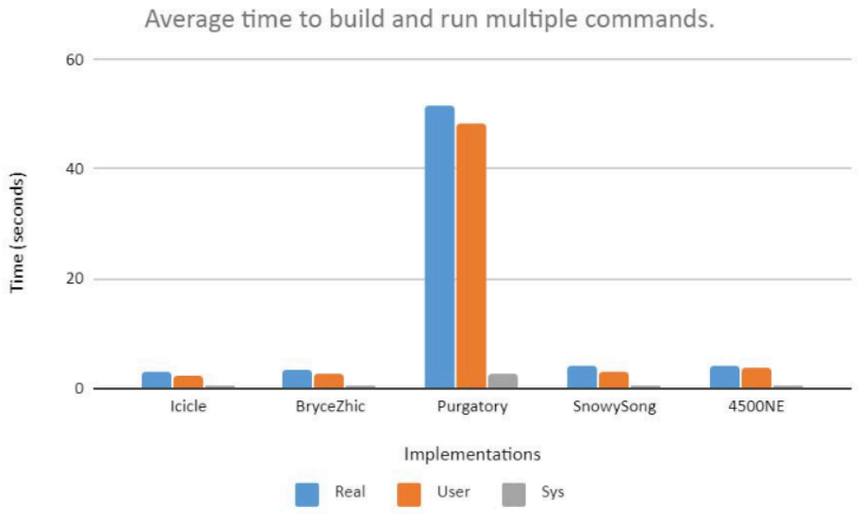
For small files, we measure the time to read and parse one file 100 times. We repeat the experiment five times. For large files, we measure the time to read and parse the large file once. We repeat the experiment three times.

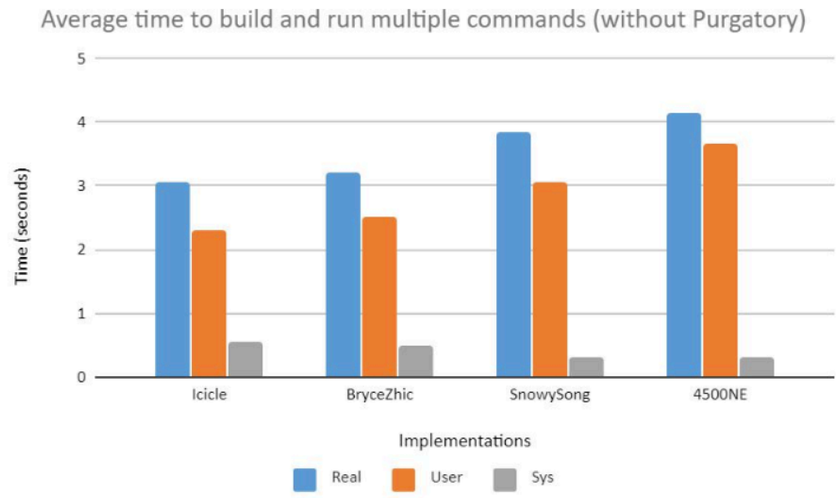
The global ranking is determined by adding the ranking on small files with the ranking on large files.

Performance Comparison

Runtime

First benchmark:





In order to generate these graphs, we timed the execution of our Makefile five times and then took the average of the resulting real, user, and sys times for each adapter. We excluded Prizes! because it did not complete our test suite in under thirty minutes. The adapters' performance on this benchmark ranked from best to worst are:

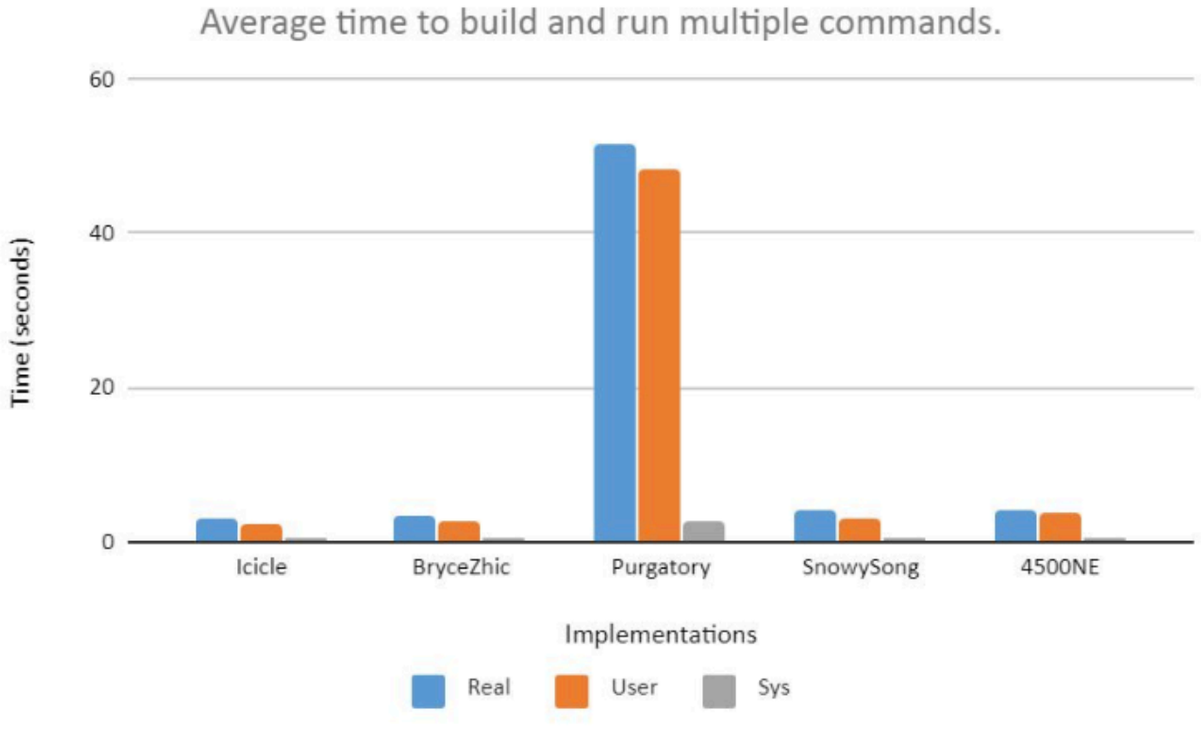
1. Icicle
2. BryceZhic
3. SnowySong
4. 4500NE
5. Purgatory
6. Prizes!

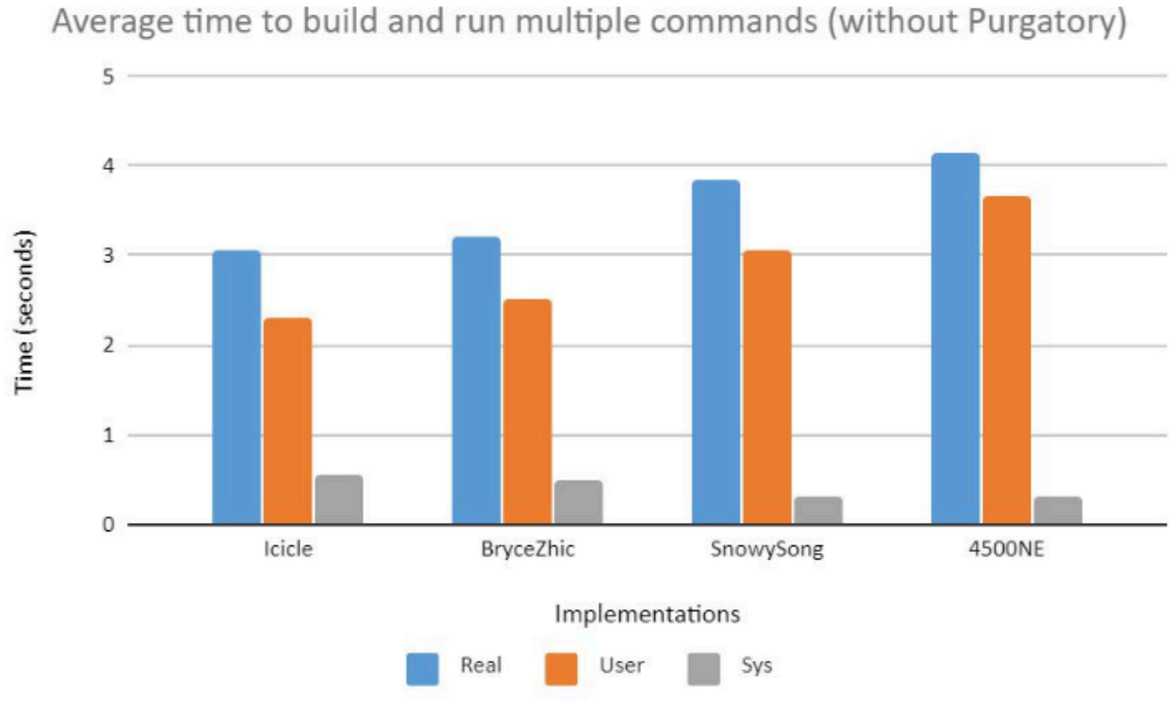
Our test suite for this benchmark consisted of the following commands:

-./sorer -f test/3.sor -from 100 -len 10000 -print_col_type 0
-./sorer -f test/3.sor -from 200 -len 100000 -is_missing_idx 9 240
-./sorer -f test/3.sor -from 300 -len 10000 -print_col_type 15
-./sorer -f test/3.sor -from 400 -len 100000 -is_missing_idx 0 1000
-./sorer -f test/3.sor -from 500 -len 10000 -print_col_idx 0 54
-./sorer -f test/3.sor -from 600 -len 10000 -print_col_type 10
-./sorer -f test/3.sor -from 700 -len 10000 -print_col_type 1
-./sorer -f test/3.sor -from 800 -len 10000 -print_col_type 2
-./sorer -f test/3.sor -from 900 -len 10000 -print_col_type 3
-./sorer -f test/3.sor -from 1000 -len 100000 -is_missing_idx 1 450
-./sorer -f test/3.sor -from 1100 -len 100000 -is_missing_idx 1 761
-./sorer -f test/3.sor -from 1200 -len 10000 -print_col_idx 12 0
-./sorer -f test/3.sor -from 1300 -len 10000 -print_col_idx 3 45
-./sorer -f test/3.sor -from 1400 -len 10000 -print_col_idx 2 76

Runtime

First benchmark:





In order to generate these graphs, we timed the execution of our Makefile five times and then took the average of the resulting real, user, and sys times for each adapter. We excluded Prizes! because it did not complete our test suite in under thirty minutes. The adapters' performance on this benchmark ranked from best to worst are:

1. Icicle
2. BryceZhic
3. SnowySong
4. 4500NE
5. Purgatory
6. Prizes!

Our test suite for this benchmark consisted of the following commands:

-./sorer -f test/3.sor -from 100 -len 10000 -print_col_type 0

-./sorer -f test/3.sor -from 200 -len 100000 -is_missing_idx 9 240

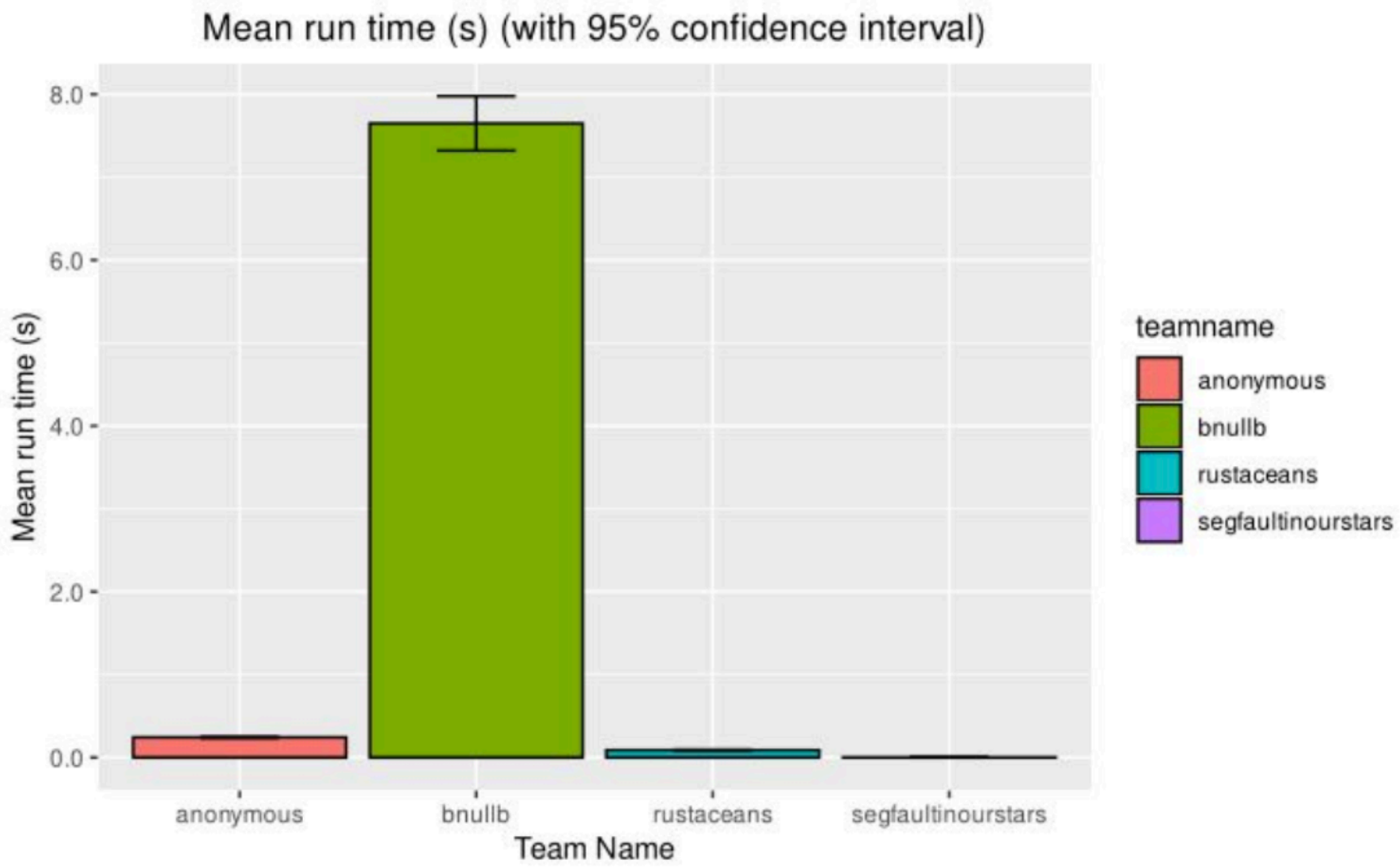


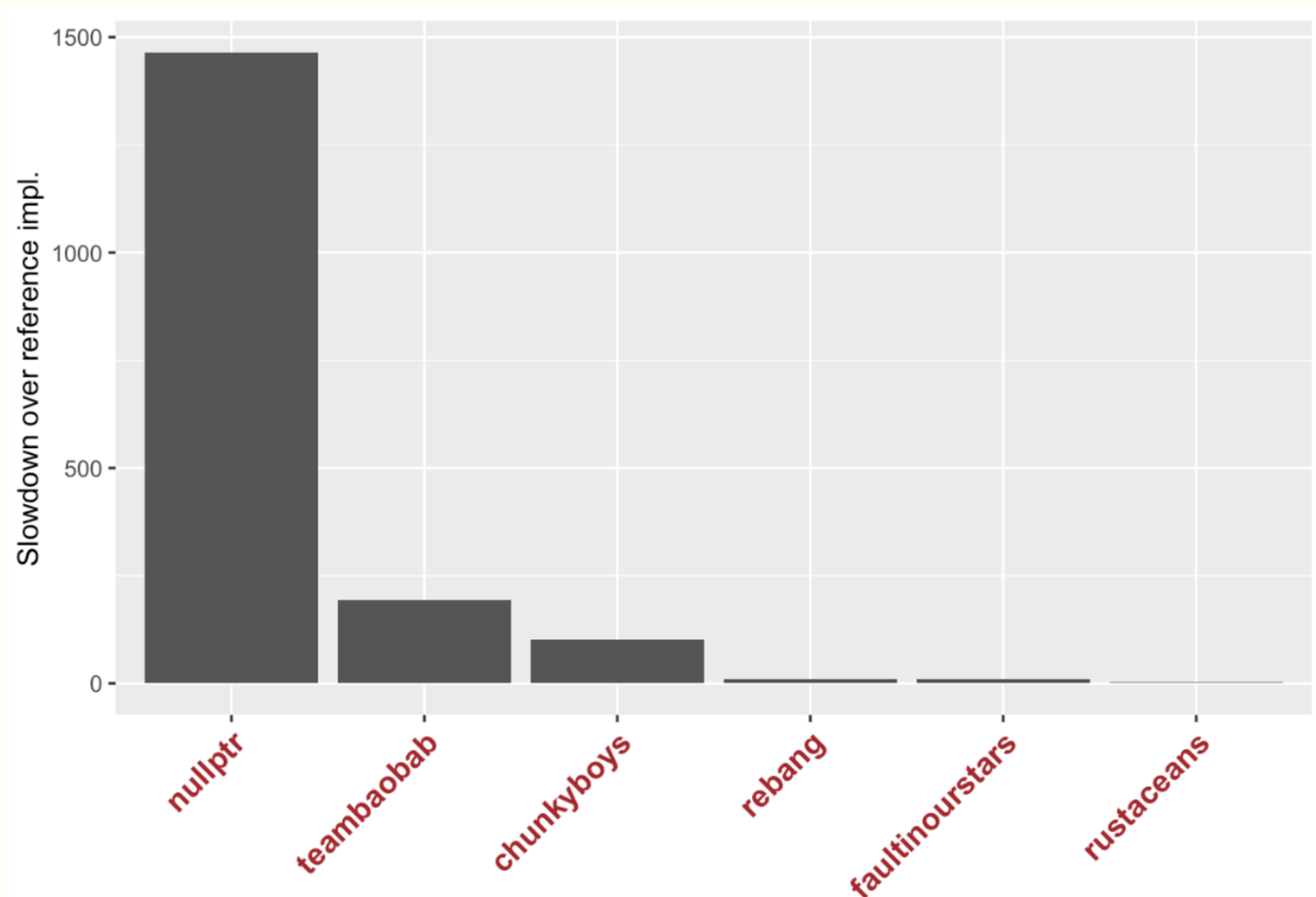
Figure 6: Mean run time and maximum memory usage for data adapters for 1000x1000 benchmark.

For this test, we set a three minute cutoff time (as we felt any project that took more than this much time on a large file was ineligible to be chosen). With this restriction, Purgatory and Prizes! were excluded from the trial. We found that the results from the other four projects fell largely in line with our previous tests. While segfault in our stars did attain low run time and memory usage, it was not able to correctly print a result from the provided test file. Bnullb again returned an out of range error. Rustaceans and Anonymous both returned the expected result and performed well.

Large files

For large files, we use a 91MB file with rows such as `<1><><+1.2><"bye"><1><><+1.2><"bye"><1><><+1.2><"bye"><1><><+1.2><"bye">`. Each measurement is one run of the data adapter reading the file once. The `nullptr` implementation is an outlier with a slowdown of 1465.2x. The first three implementations have slowdowns of 1.6x for `rustaceans`, 9.2x for `faultinourstars`, 10.7x for `rebang`.

The throughput for large files is 47.16MB/sec (`rustaceans`) and 8.22MB/sec (`faultinourstars`).



Threats to validity

Threats to validity

There are several threats to validity:

- We have not tested for correctness of the solution. An incorrect solution, once fixed to be specification compliant, may run slower.
- We are aware that some of the solutions take shortcuts, we have tried to craft our queries so as to force evaluation, but there may still be cases where some solution speed up performance by not doing the work that they are expected to do.
- The small file experiment over-represents start up costs, this is likely to hurt languages with more heavy runtime systems such as Python and Java.

Recommendations

Recommendation to Management

Based on the above analysis, we would rank the projects that passed our benchmarks without errors in the following order based on **performance alone**:

1. **Rustaceans**
2. **Anonymous**
3. **Purgatory**
4. **Prizes!**

We have excluded segfault in our stars and bnullb from this ranking as we discovered bugs in these implementations during our testing which we feel disqualifies them from use. Additionally, we would note that the gap between the first two adapters and the last two is significant. Purgatory and Prizes! were both omitted from our final trial as their runtime on a large input file (1000x1000) exceeded the three minute cutoff time we set. The top two adapters boast strong performance metrics with little variance, and have the added benefit of being not limited by platform. For these reasons, we feel that the best choice for our purposes is either the rustaceans or Anonymous adapters.

Given that the performance of these two implementations is so similar, we looked more closely into the other benefits and disadvantages of each:

	Benefits	Disadvantages
Rustaceans	• Slightly better performance in data processing tests • Multithreaded • Relatively well documented • Good argument processing (never segfaults, provides readable errors)	• Longer build time • Written in Rust (more challenging to maintain) • Harder to interface with CwC
Anonymous	• Easier to interface with CwC as it is written in C++ • Shorter build time and less memory usage while building	• Worse documentation • Slightly worse total run time performance with large input files • Single-threaded • Can segfault if arguments are not passed in correctly

Figure 7: Direct comparison of rustaceans and Anonymous data adapters.

Based on these considerations, we would recommend that management select the **rustaceans** project. While we do recognize that it will be slightly more challenging to integrate with CwC, we find that its strong performance, use of multithreading, and more robust documentation make it the superior overall choice.

P2

- **What should be in the README?**
    - A description of the classes with a design rationale
        - Avoid the trivial
        - Explain choices
- **Use-cases**

Use case

Statement:

Imagine we want to compute the taxes of all employees as their tax rate multiplied by their monthly salary minus the deductions if any.

```
// Creating a data frame
Schema scm("IFBII");          // the schema
DataFrame df(scm);            // the data frame

// TODO: show how to populate the table

// Computing the taxes
class Taxes : public Rower {
public:
 size_t salary=0, rate=1, isded=2, ded=3, taxes=4;
 void accept(Row& r) {
   int tx = (int) r.get_int(salary) * get_float(rate);
   tx -= r.get_bool(isded) ? r.get_int(ded) : 0;
   r.set(taxes, tx);
 }
};

Taxes tx;       // create our rower
df.map(tx);     // apply it to every row
```