Introduction:

Computer Configuration : Intel i5-6300U CPU @ 2.40 GHz, 64 bits
Kernel: 5.5.1-arch1-1
RAM: 8 GB

Our goal of the report is to pick a best data adapter among the 6 groups we chose. The selecting process is based on the benchmark file we created, and the performance is the factor we gave the highest consideration. After picking the adapter with better performance, we will choose the best one with better code quality.

Here are the groups that we chose:

chunky_boys	https://github.com/danielgaooooo/a1p1	chunkyboys
RebAng	https://github.com/angelawang24/A3_a1_p1	rebang
danyth	https://github.com/yth/CS4500A1P1	danyth
OHE	https://github.ccs.neu.edu/bdewitt/CS4500_Assn1_Part1	ohe
rotwang.ai	https://github.ccs.neu.edu/seannolly/swd_assignment_3.git	rotwangai
rustaceans	https://github.com/SamedhG/sorer.git	rustaceans

Description of the analysis performed:

First, our group used Python created a benchmark file with 20 columns and 1 million rows of data, resulting in a 170 Mb file. We then used a bash script (available on GitHub) to automatically run the same commands on all programs 10 times. Here are the benchmark command that we used to measure:

print col type:
These 4 commands measures if the code could run print_col_type well, with different starting bit, and it outputs the selected col correctly. It also tests the speed of searching the whole file, as well as a segment at the beginning and a segment in the middle. We wanted to see if only using a segment of the file affected performance.

```
"-from 0 -len 10000 -print_col_type 10"
"-from 100000000 -len 10000 -print_col_type 15"
"-from 0 -len 200000000 -print_col_type 9"
```

print col idx:
The first three essentially tested the speed it parsed the schema, this command mainly compares the performance of actually returning the data. We compared the index that contains more and more bits to test the performance. Starts with 0, then 5k, 10k, 100k….until 999999. As well as a test starting at a large bit size and find the index in the middle. Again we also wanted to see if segmenting the file had an effect on performance

as well, using a segment in the middle of the file.

```
"-from 0 -len 10000 -print_col_idx 0 0"
"-from 0 -len 200000000 -print_col_idx 0 0"
"-from 0 -len 200000000 -print_col_idx 10 5000"
"-from 0 -len 200000000 -print_col_idx 10 10000"
"-from 0 -len 200000000 -print_col_idx 10 100000"
"-from 0 -len 200000000 -print_col_idx 10 500000"
"-from 0 -len 200000000 -print_col_idx 10 700000"
"-from 0 -len 200000000 -print_col_idx 16 999999"
"-from 100000000 -len 10000 -print_col_idx 7 10"
```

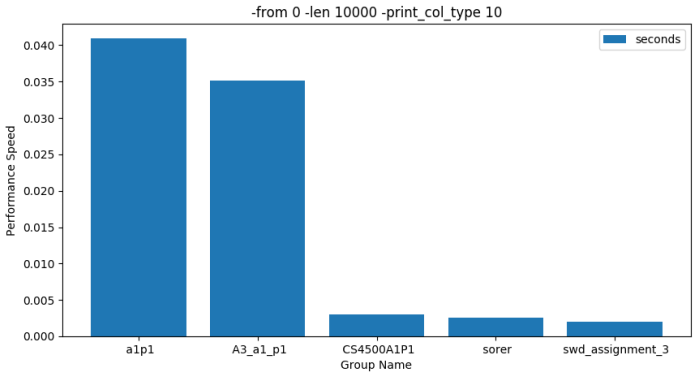
Is_missing_idx:

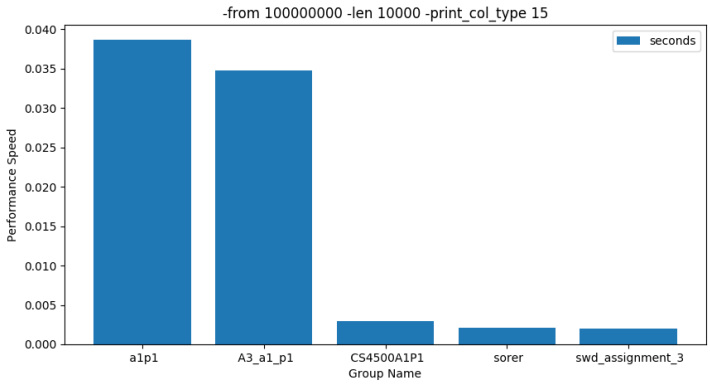
Same as printing col index, to test the performance if the missing index in different place in the benchmark file, as well as checking if it prints the correct output when the col is missing

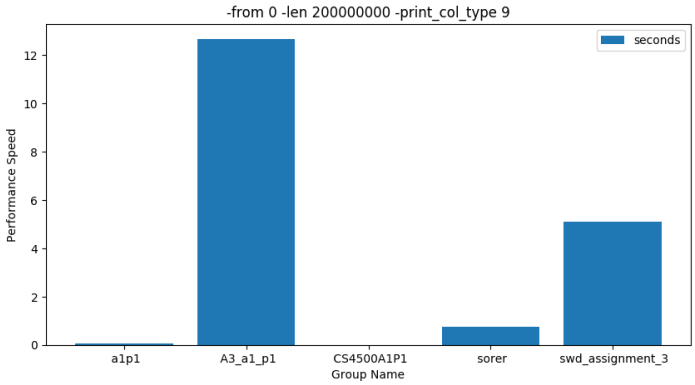
```
"-from 0 -len 10000 -is_missing_idx 14 15"
"-from 0 -len 200000000 -is_missing_idx 14 15"
"-from 0 -len 200000000 -is_missing_idx 4 250000"
"-from 0 -len 200000000 -is_missing_idx 4 750020"
"-from 0 -len 200000000 -is_missing_idx 19 999999"
"-from 500000 -len 50000 -is_missing_idx 9 100"
```

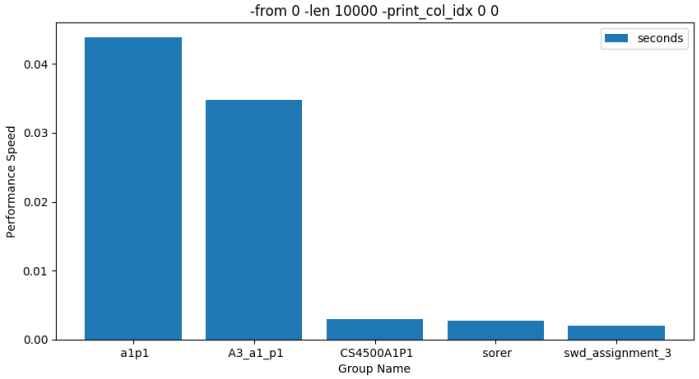
Comparison of the products' relative performance:

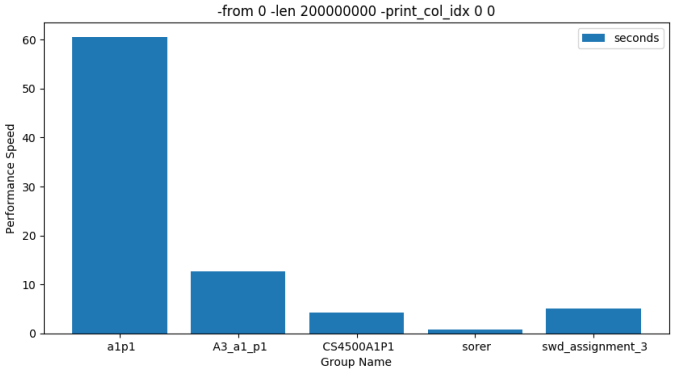
Here is the graph we used to compare performance, each graph is the average performance over all 10 times the command was run:

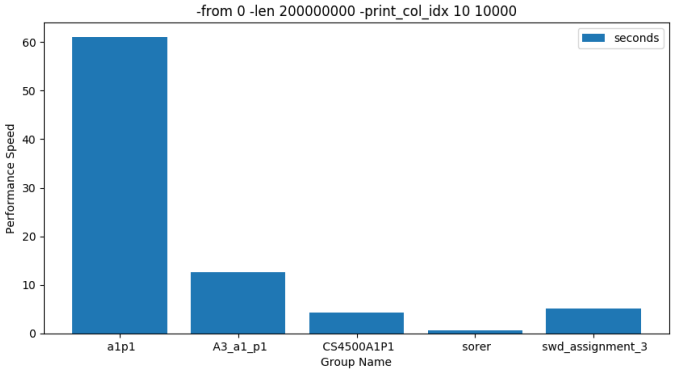


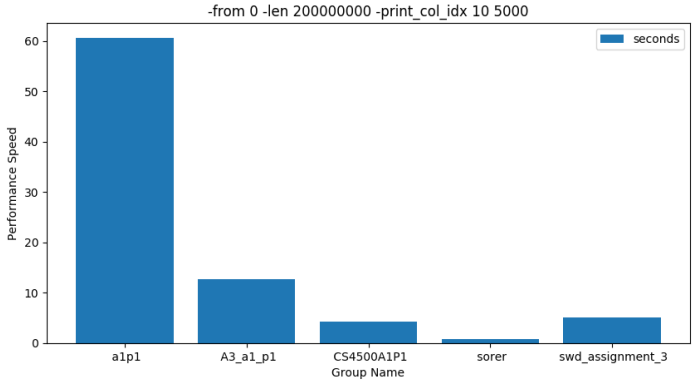


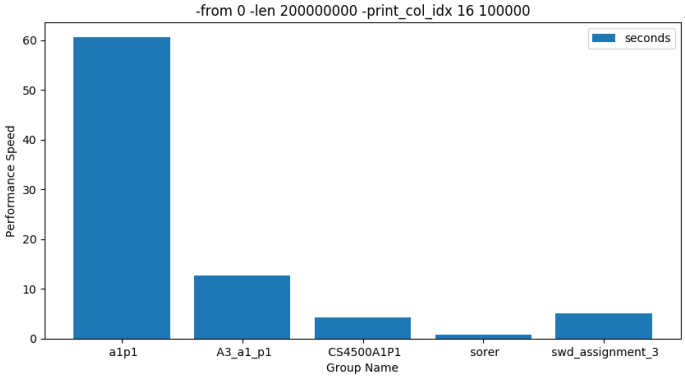


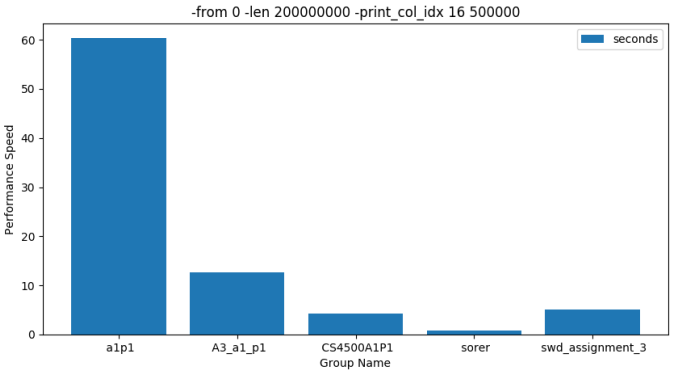

-from 0 -len 10000 -print_col_idx 0 0
seconds
Performance Speed
0.00
0.01
0.02
0.03
0.04
a1p1
A3_a1_p1
CS4500A1P1
sorer
swd_assignment_3
Group Name

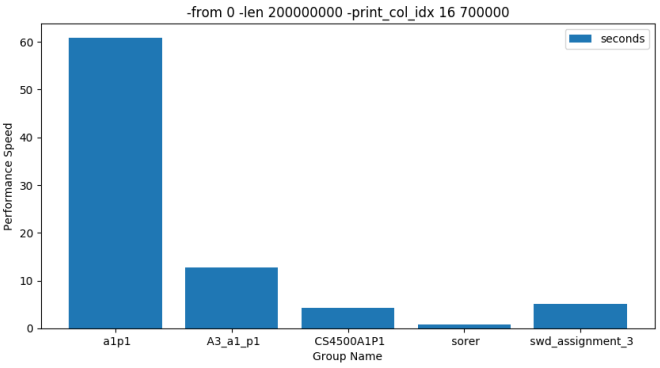

-from 0 -len 200000000 -print_col_idx 0 0
seconds
Performance Speed
0
10
20
30
40
50
60
a1p1
A3_a1_p1
CS4500A1P1
sorer
swd_assignment_3
Group Name

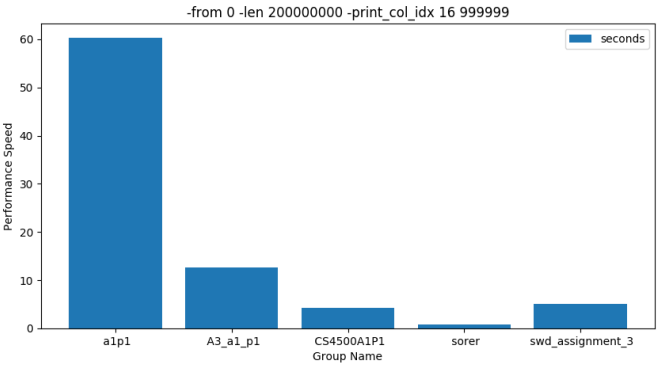

-from 0 -len 200000000 -print_col_idx 10 10000
seconds
Performance Speed
0
10
20
30
40
50
60
a1p1
A3_a1_p1
CS4500A1P1
sorer
swd_assignment_3
Group Name

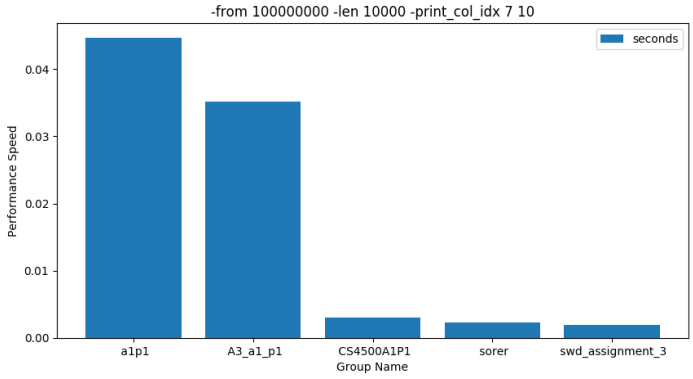

-from 0 -len 200000000 -print_col_idx 10 5000
seconds
Performance Speed
0
10
20
30
40
50
60
a1p1
A3_a1_p1
CS4500A1P1
sorer
swd_assignment_3
Group Name

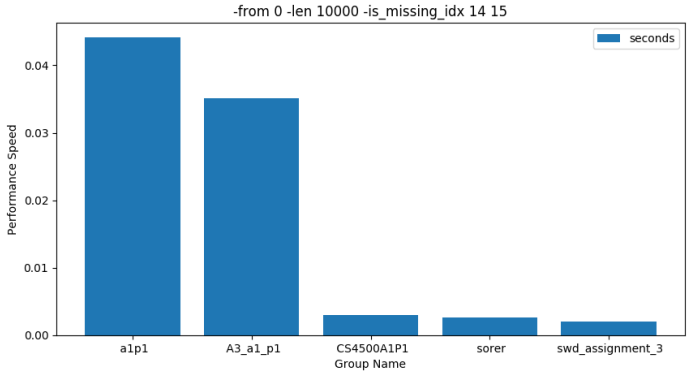

-from 0 -len 200000000 -print_col_idx 16 100000
seconds
Performance Speed
0
10
20
30
40
50
60
a1p1
A3_a1_p1
CS4500A1P1
sorer
swd_assignment_3
Group Name

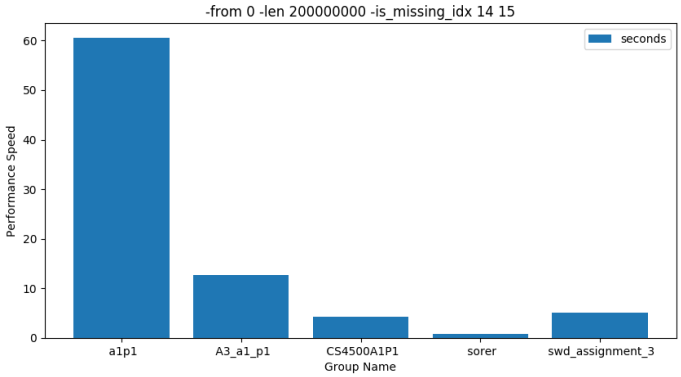

-from 0 -len 200000000 -print_col_idx 16 500000
seconds
Performance Speed
0
10
20
30
40
50
60
a1p1
A3_a1_p1
CS4500A1P1
sorer
swd_assignment_3
Group Name

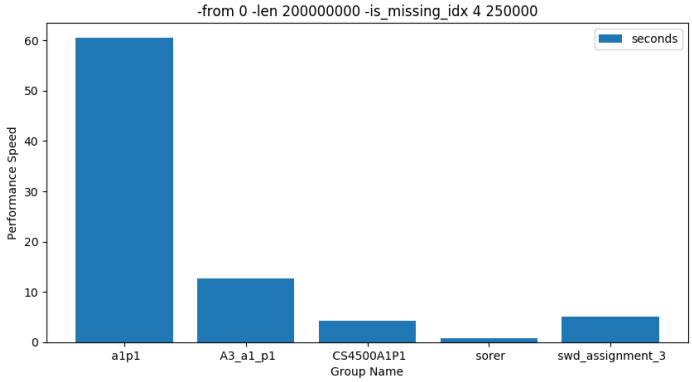

-from 0 -len 200000000 -print_col_idx 16 700000
seconds
Performance Speed
0
10
20
30
40
50
60
a1p1
A3_a1_p1
CS4500A1P1
sorer
swd_assignment_3
Group Name

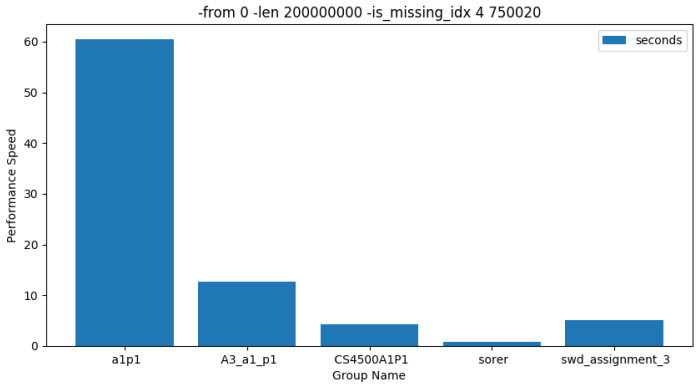

-from 0 -len 200000000 -print_col_idx 16 999999
seconds
Performance Speed
0
10
20
30
40
50
60
a1p1
A3_a1_p1
CS4500A1P1
sorer
swd_assignment_3
Group Name

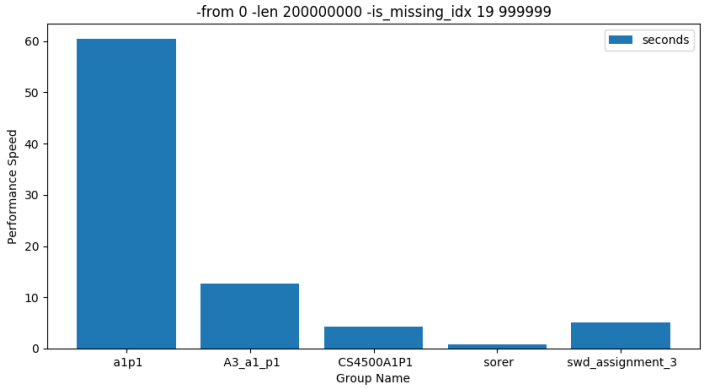

-from 100000000 -len 10000 -print_col_idx 7 10
seconds
Performance Speed
0.00
0.01
0.02
0.03
0.04
a1p1
A3_a1_p1
CS4500A1P1
sorer
swd_assignment_3
Group Name

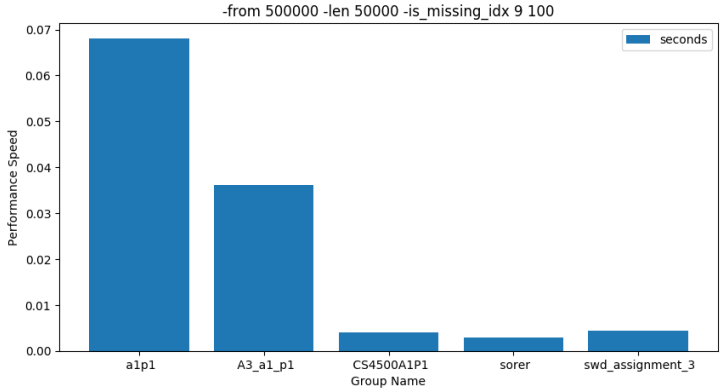

-from 0 -len 10000 -is_missing_idx 14 15
seconds
Performance Speed
0.00
0.01
0.02
0.03
0.04
a1p1
A3_a1_p1
CS4500A1P1
sorer
swd_assignment_3
Group Name


-from 0 -len 200000000 -is_missing_idx 14 15
seconds
Performance Speed
0
10
20
30
40
50
60
a1p1
A3_a1_p1
CS4500A1P1
sorer
swd_assignment_3
Group Name


-from 0 -len 200000000 -is_missing_idx 4 250000
seconds
Performance Speed
0
10
20
30
40
50
60
a1p1
A3_a1_p1
CS4500A1P1
sorer
swd_assignment_3
Group Name







Our Ranking is as follows:

#1. sorer (rustaceans)

This is the group with the best performance among the 6 groups; this group performed the best on almost every command we tested. Their approach was multi-threaded parsing of the schema to improve performance. They utilize the abilities of Rust well, and since Rust can expose C bindings and call C code, they should be able to integrate with the adapter

easily. Furthermore Rust helps prevent common memory bugs while maintaining low level control and performance.

#2. CS4500A1P1 (danyth)
The second place group's performance is a slightly worse than the first group, but it also has a consistent look-up time that is very quick. It uses memory mapping to avoid the pitfalls of managing the file in memory themselves, and is well written. As it is in C++, it is easy to integrate with.

#3. swd_assignment_3 (rotwang.ai)
This group's performance is very similar to the second place group, it didn't perform as good as the second group on the tests using a segment in the middle of the file, as well as when it is returning an item at a large row index. The code is written in C++, and shouldn't be difficult to integrate with, but the performance is lacking.

#4. A3_a1_p1 (RebAng)
Compare to the first three groups, this adapter's performance has a poor performance. The data size is getting larger, the worse performance we can see from this adapter. For all the groups that we tested this group consistently ranked number 4 for all tests, but all the code works fine. The code is written in python, and while easy to read as a result performance suffers.

#5. a1p1 (chunky_boys)
With all the data we see above, this is the group has the worst performance on mostly every command we run. It took over a minute on commands that returned a value with a large row index. This is likely due to it being implemented in python, which is easy to understand but has abysmal performance characteristics.

#6. CS4500_Assn1_Part1 (OHE)
This group is not included on the graphs comparing performance, because it returned only 0 for any given index, and sometimes crashed in our 12^{th} and 18^{th} tests where we used a segment in the middle of the file. The rest of the groups have at least working code, as this is non-functional we ranked it last. It is written in C++, which likely contributes to the bugs. Because this was not functional, we did not include them in our graphs.

Threats to validity:

We have not tried to give any error commands to handle, there should be nothing that is confusing or should cause an error. The only thing that might break is potentially caused by the large amount of data, which given the directive for the assignment, we felt the program was required to handle.

Recommendation to management:

We recommend the sorer by **rustaceans** as the best choice among these 6 groups. This group used Rust, which may cause readability issues for people who don't know the language, but it is the most efficient among the 6 groups. Furthermore due to the design of Rust, common memory errors can be avoided and less buggy code provided. If one knows Rust, it is pretty easy to understand this implementation of sorer.