A3

Xiaocheng Zhang, Tian Xia

2/7/2020

Introduction

Throughout this report, we have evaluated 6 sets of data adaptors that read sor files and store information in a columnar representation. Due to the reason that the management team has to deal with large sor files, our team hence tested each data adaptor's performance in terms of running time and memory usage. Different hardware and software infrastructures have been utilized for performance assessment so that the accurancy and stability are ensured. In addition to performance, our team also takes a closer look at the code quality and documentation in order to ensure that this data adaptor can be used in the long run. In the end of this report, we will give our recommendation that consists of a list of the projects in order of preference, and a rationale for our top choice.

Description of the Analysis Performed

1) Our team will first check if the adaptor transform data into columnar representation since many adaptors on the market did not meet this requirement. If an adaptor does not do so, we will pass it to the next one.
2) Our team will examine codes correctness by running 25 customized tests to check functionality of '-print_col_type', '-print_col_idx', '-is_missing_idx' and '-f', '-from', '-len'. Some of the adaptors may fail to follow schema to print value, while some may appear to have segmentation faults. The correctness will be more emphasized than other requirements below during evaluation.
3) Third, our team will take a closer look into its run time and memory usage. To document the run time, we will use the system built-in command 'time' to record. The result will be categorized into three types: real, user, and sys.
4) According to standard format, main function should be short and clean, and the task should be well distributed into helper function to be completed. If there are too much code in one function, we think the format can be improved by having more sub functions.
5) Documentation for each class, constructor and method should be done in a professional format. We expect one user can quickly adopt using their code by looking at their documentation.

Comparison of the Products' Relative Performance

Team 1:

Name: danyth

This team used c++ to implement this Assignment. They did save data as column and call methods to get from the dest column.

They process the data by parsing the file character by character, using switch statements on the ASCII value of each character, and using local variables to hold the state of the SoR.

- Run time:

```
Fast:
real    0m40.883s
user    0m1.169s
sys     0m0.700s

Slow:
real    0m44.975s
user    0m1.343s
sys     0m0.936s
```

- Memory Usage:
  These are data came from running *sorer* by *Valgrind*:

```
HEAP SUMMARY:
==6==     in use at exit: 0 bytes in 0 blocks
==6==   total heap usage: 1,006 allocs, 1,006 frees, 121,812 bytes allocated
- Test failed:
    None
```

Team 2:

Name: nullptr

This team used c++ to implement this Assignment.
They did save data as column and call methods to get from the dest column.

They used a file source iterate over the lines, parsing to figure out the number of elements Then they used a fresh source iterator, parse the lines (assuming missing value to reach the needed length) in order to infer the schema.

Run time:

```
Fast:
real    2m31.587s
user    1m8.424s
sys     1m20.749s

Slow:
real    2m44.017s
user    1m11.018s
sys     1m25.063
```

- Memory Usage:
  These are data came from running *sorer* by *Valgrind*:

```
HEAP SUMMARY:
==76653==     in use at exit: 83,890 bytes in 164 blocks
==76653==   total heap usage: 205 allocs, 41 frees, 301,610 bytes allocated

    LEAK SUMMARY:
==76173==    definitely lost: 0 bytes in 0 blocks
==76173==    indirectly lost: 0 bytes in 0 blocks
==76173==      possibly lost: 72 bytes in 3 blocks
==76173==    still reachable: 65,736 bytes in 7 blocks
==76173==         suppressed: 18,082 bytes in 154 blocks
```

- Test failed:

```
./sorer -f 1.sor -print_col_idx 0 3
"+1" | +1
./sorer -f 2.sor -print_col_idx 3 0
"hi" | hi
./sorer -f 1.sor -from 1 -len 74 -print_col_idx 0 6
"+2.2" | +2.2
```

- For failed tests, it seems like they failed to follow the schema to print values. All the failed cases are related to String output.

Team 3:

Name: BryceZhic

This team used c++ to implement this Assignment. They did not save data as column and call methods to get from the dest column.

They did not have new classes. Therefore all the code for this project can be found in main.cpp. This violates our rule on code style.

- Run time:

```
Fast:
real    0m6.514s
user    0m4.841s
sys     0m1.557s

Slow:
real    0m6.813s
user    0m5.022s
sys     0m1.616s
```

- Memory Usage: These are data came from running *sorer* by *Valgrind*:

```
HEAP SUMMARY:
==78070==     in use at exit: 83,866 bytes in 163 blocks
==78070==   total heap usage: 193 allocs, 30 frees, 162,098 bytes allocated
```

```
    LEAK SUMMARY:
==78070==    definitely lost: 0 bytes in 0 blocks
==78070==    indirectly lost: 0 bytes in 0 blocks
==78070==      possibly lost: 72 bytes in 3 blocks
==78070==    still reachable: 65,736 bytes in 7 blocks
==78070==         suppressed: 18,058 bytes in 153 blocks
```

- Test failed:

```
None
```

Team 4:

Name: SnowySong

This team used Python to implement this Assignment.
They did save data as column and call methods to get from the dest column.
No more redundant code, the structure is compressed and useful.
The whole program was implemented by one class with enums.

Their SoR types are defined by enums called “SoRTypes”
Class “SorInterpreter” is the main Object function that provided most methods to read and save data.

Their code runs fast and go through these steps:
1. Set the initial status for reading at first: Method: “**init**” in “class SorInterpreter” in 31 line.
2. Read the data by lines, get the schema during reading. Method: "___get_column_schema" in 101 line in sorer. "___itemize_sor_row" is the main method read the data in line 136.
3. Call parser to parse the required data follow the schema.

- Run time:

```
Fast:
real    0m2.885s
user    0m1.313s
sys     0m1.281s

Slow:
real    0m3.097s
user    0m1.297s
sys     0m1.500s
```

```
- Test failed:
./sorer -f 2.sor -print_col_idx 2 1
-.2 | -0.2
./sorer -f 4.sor -print_col_idx 1 1
-0.000000002 | -2e-09
./sorer -f 4.sor -print_col_idx 1 2
10000000000 | 1e+10
```

- For failed tests, they are subtle problems which are not problems. It is easy to handle these cases by update their “args.print_col_type” method with more type checker in 229 in sorer.

- In contrasts to other groups’ work, this program has many advantages:

  - It has simplified methods, which reduced redundant code.

  - Has many comments that help readers understand the whole program.

- However, they used python. They don't need to handle most problems.

Team 5:

Name: rotwang.ai

This team used C++ to implement this Assignment.
They designed six different header files, which form a dataframe that helps to write and read data.
Their code is clear in main.cpp file. Names of methods are readable. Total processes of this program are logical and readable. Each method has a comment that helped readers to understand.

SoR types are saved as *enum* in *token.h* file

Code speed is median (to fast) and followes these steps:
1. Create a Configuration class that takes argc and argv, automatically parse the command, and save configuration.
2. Create a dataframe class within saved configuration, generate the schema and save all data. If it cannot find a valid schema, the program will quit with error message.
1. They saved all data by "vector<vector>", a 2D Array that saved data into inner array as one column.
2. Their method "parse" will be able to push parsed data into their dataframe.
3. They will call *run* method in dataframe to print the data that follows the configuration which saved commands in previous steps.

- Run time:

```
Fast:
real    0m12.731s
user    0m10.344s
sys     0m7.484s

Slow:
real    0m13.486s
user    0m10.563s
sys     0m7.844s

- Test failed:
./sorer -f 1.sor -print_col_idx 0 3
+1
./sorer -f 2.sor -print_col_idx 3 0
hi
./sorer -f 3.sor -print_col_idx 2 10
+1.2
./sorer -f 4.sor -print_col_idx 0 1
+2147483647
./sorer -f 4.sor -print_col_idx 1 1
-0.000000002
./sorer -f 1.sor -from 1 -len 74 -print_col_idx 0 6
+2.2
```

- Those failed tests are caused by *run* method in dataframe, it dose not print the data follow the schema.

- For different data types, they need to print chosen data as the format that matches them.

- Memory Usage:
  These are data came from running *sorer* by *Valgrind*:

```
==2066== HEAP SUMMARY:
==2066==     in use at exit: 48 bytes in 2 blocks
==2066==   total heap usage: 39 allocs, 37 frees, 1,082,374 bytes allocated
    ==2066== 24 bytes in 1 blocks are definitely lost in loss record 1 of 2

    ==2066== LEAK SUMMARY:
==2066==    definitely lost: 48 bytes in 2 blocks
==2066==    indirectly lost: 0 bytes in 0 blocks
==2066==      possibly lost: 0 bytes in 0 blocks
==2066==    still reachable: 0 bytes in 0 blocks
==2066==         suppressed: 0 bytes in 0 blocks

    ==2066== 24 bytes in 1 blocks are definitely lost in loss record 1 of 2
==2066==    at 0x4835DEF: operator new(unsigned long) (vg_replace_malloc.c:334)
==2066==    by 0x10C23D: analyze_tokens_for_schema (schema.h:54)
==2066==    by 0x10C23D: analyze_for_schema (dataframe.h:229)
==2066==    by 0x10C23D: DataFrame::parse() (dataframe.h:122)
==2066==    by 0x10A38E: main (main.cpp:27)

    ==2066== 24 bytes in 1 blocks are definitely lost in loss record 2 of 2
==2066==    at 0x4835DEF: operator new(unsigned long) (vg_replace_malloc.c:334)
==2066==    by 0x10AF7A: Token::tokenize_line(std::__cxx11::basic_string<char, std::char_traits<char>, s
==2066==    by 0x10C966: process (dataframe.h:94)
==2066==    by 0x10C966: DataFrame::parse() (dataframe.h:158)
==2066==    by 0x10A38E: main (main.cpp:27)
```

- From *Valgrind* output, program's total Heap memory took 1,082,374 bytes, with 39 allocs. However, their code did not free all allocs after ending program.
  - Two memory lost happen in *parse* method in file datafram.h:94.
  - Vector in Token class wasn't freed.

- The Memory Usage of this program is *7 level* out of 10.

Team 6:

Name: segfault

This team used c++ to implement this Assignment.
They did save data as column and call methods to get from the dest column.
The program determines the schema and datatypes by reading the first 500 lines in the file.
Datatypes are stored as int constants with 0 representing BOOL, 1 representing INT, 2 representing FLOAT, and 3 representing STRING

- Run time:

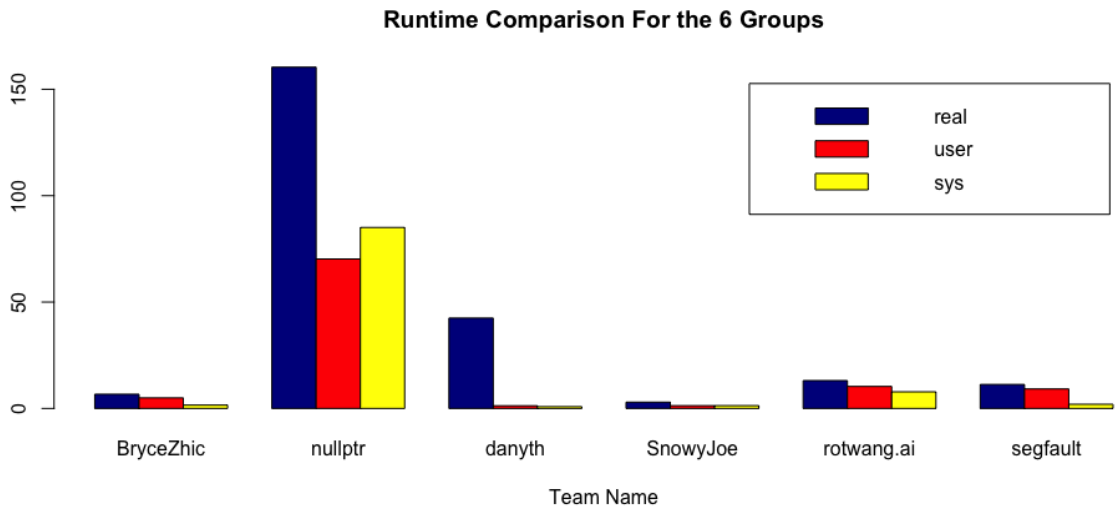
```
Fast:
real    0m11.319s
user    0m9.156s
sys     0m1.969s

Slow:
real    0m11.281s
user    0m9.031s
sys     0m2.156s
```

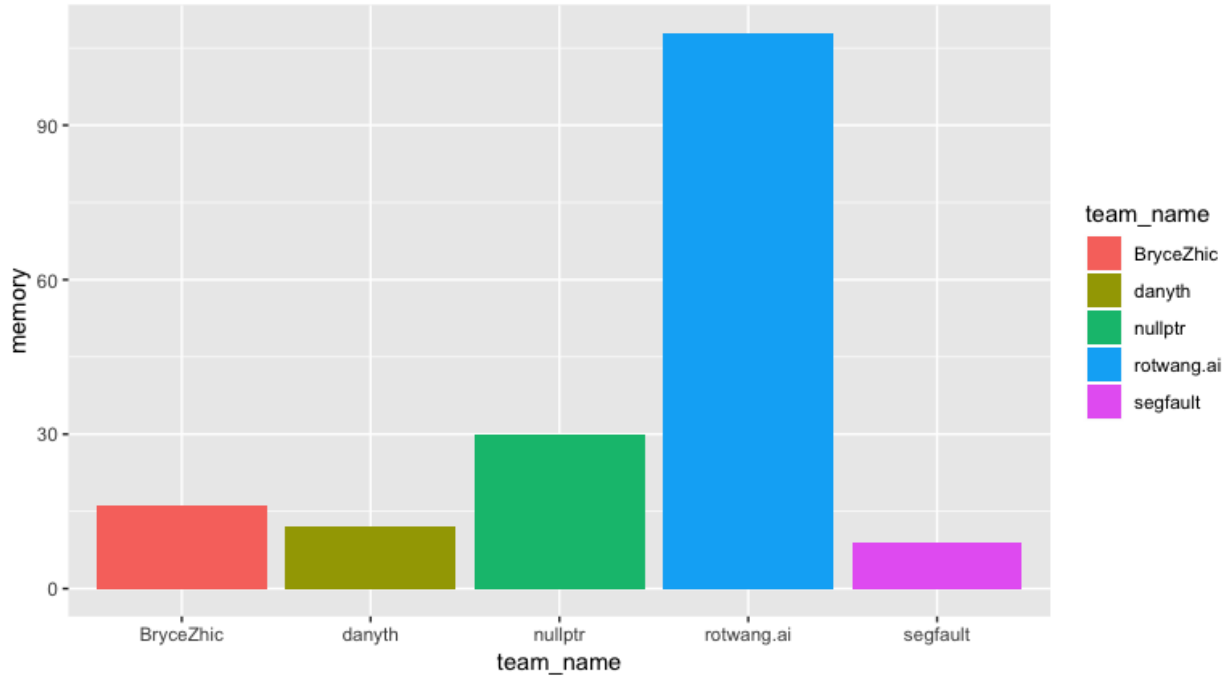
- Memory Usage:
  These are data came from running *sorer* by *Valgrind*:

```
==2163== HEAP SUMMARY:
==2163==     in use at exit: 73 bytes in 5 blocks
==2163==   total heap usage: 17 allocs, 12 frees, 94,437 bytes allocated
```

- Test failed:

```
./sorer -f 1.sor -print_col_idx 0 3
+1
./sorer -f 2.sor -print_col_idx 3 0
Hi
./sorer -f 2.sor -print_col_idx 3 1
ho ho ho
./sorer -f 1.sor -from 1 -len 74 -print_col_idx 0 6
+2.2
```





Threats to validity

In C++, there are a few issues that could cause non-deterministic results. A C++ compiler, given the same input, might render different outputs when run multiple times on the same machine, or on different machines. The causes might be that iteration order and sorting order can be non-deterministic in a C++ compiler. In addition to the random results generated by the same compiler, when developers run programs on different compilers, for instance, GCC and Clang, invalid results might be expected. Similarly, if the same compiler is hosted on different operating systems, it does not always guarantee the same output. Besides the internals of a compiler, implementation of program code can be a potential risk for validity. If the program code depends on the time it is run at, no two runs will generate the same output. Common cases are when we initialize or update a variable that uses the current time, or use the time for numerical operations. Another similar but trivial scenario is when we use random number generation, two programs will not render the same results. Allocators that use dynamic memory allocators can cause unpredictable results. This is specifically noticeable in the case of a real-time system. If the program has a varying pattern of allocation, dellocation, and reallocation, the timing and sequence can make the program unpredictable. Multiple-threaded programs are also more likely to see unpredictable results if the order of instructions cannot be controlled. Last but not least, a common and easily made mistable is an uninitialized variable. In other programming languages, for example, python, if developers run programs on different versions, python 2 and python 3, different results can also be expected.

Recommendation

Our ranking for 6 teams is rotwang.ai -> danyth -> BryceZhic -> nullptr -> SnowySong -> segfault Team rotwang.ai is our top choice because the program is easily extendable and it embraces Single Responsibility Principle. The program has a defined architecture, and each file serves a single purpose. For example, files are named as buffer, dataframe, and dataframe. It is easy to understand for other developers to develop and maintain.
Another reason that makes this program easy to adapt and extend is that it creates an ‘action’ enum type

and was used in multiple files. The use of enum makes the program less error prone, because of less human typo errors. In addition, if in the future, there is a new type being added to the program, we can simply add a if statement to check for the added type.
Lastly, this program shows clear error message when given invalid input. In the program, when it attempts to create a dataframe class with saved configuration, if it is unable to find a valid schema, the program will quit and gives an error message. The error message allows developers to debug faster and quitting the program when an error occurs prevents further errors.