1. Write a well readable python script to handle that task. Try to find a ​good tradeoff​ between runtime and RAM needs.

pseudocode:

* 1. Read line of JSON file;
  2. Fine item matches title, city, country in the created list;
  3. Compare category score, update it or add it;
  4. Write merged list to file.

Implementation breakdown:

1. JSON loads eachline, and normalize nested dictionary:

return column = ['title', 'category', 'city', 'country', 'score']

1. Breaks into Sorting algorithm and searching algorithm:

First sort list by title (Bubble or Insertion or Selection or Timsort)

Second search title (LinearSearch or BinarySearch or TernarySearch)

1. Check title, city, country matches

replace the score, discard or add as new

Repeat until end

1. Generator holding the result, when dump request writes each line into file

In the file, I only include merged result from Timsort and BinarySearch.

Python 3.6

Script dependency [json, sys, time, psutil, functools]

Usage of Command Line: -i input -o output (optional: -s second read file)

(optional: -sorting -searching)

1. Give a short estimation of the expected runtime of your algorithm, the needed RAM and I/O costs. How do you expect it to scale on far bigger datasets?

Timsort is the fastest one, in the log event I only trial Timsort with (BinarySearch).

Note: LinearSearch was implemented within BinarySearch and TernarySearch with precision 10. Trying to trade off accuracy and performance.

====================================

Running Start at time: 2018-02-12 17:52

Sorting Algorithm: Timsort; Searching Algorithm:BinarySearch

File Name: merging\_challenge\_data.json, Total Lines: 247497, Merging Time: 19239.354105472565

Total Dumping Lines:135228

Total time running dump: 0.7526736259460449 seconds

Memory Usage: 92.23828125MB

CPU Start Usage: scputimes(user=769716.5, system=52454.0, idle=4527452.0, interrupt=4263.890625, dpc=3228.828125)

CPU End Usage: scputimes(user=815407.375, system=57143.0, idle=4631014.0, interrupt=4813.046875, dpc=3425.4375002384186)

====================================

For scale up, I would seek a solution from database. Or map reduce application to manipulate file on HFDS. PySpark is one of the option.

1. Think of an ​algorithm​ that is ​very RAM efficient​. How would such a solution look like and how would it affect runtime and I/O.

Also, can modify algorithm as generator-based. Generators introduce the yield statement to Python. It works a bit like return because it returns a value. The difference is that it saves the state of the function. The next time the function is called, execution continues from where it left off, with the same variable values it had before yielding.

For instance the code below, is generator-based linear search

for x in self.seen:

if(tccCheck(d, x)):

if int(d['score']) > int(x['score']):

yield json.loads(d)

else:

pass

else:

yield json.loads(d)

Also multi-threading can be concerned. But currently I have no clear thinking over it, and willing to learn more from industry practice. For example, a list can be partition into parallel processes seems workable but we need to adjust the partitioned list in order to put same title together. I would like to explore more.

1. What could further be done to optimize I/O operations in this case?

Design level

Current design is file in and file out within pure Python environment. Maybe could consider different platform and programming language.

Algorithms and data structures

Current data structure uses list of dictionary in Python. List of string can be considered, for example extracting information from text using regular expression.

Current sorting and searching algorithms are the basic algorithms. Timsort is the fastest sorting which is as built-in function in python. Searching algorithms should consider using logarithmic O(log n), for example binary search.

Source code level

For example writing string using format is better than using +. In Python str is immutable, so the left and right string have to be copied into the new string for every pair of concatenation. If you concatenate four strings of length 10, you will be copying (10+10) + ((10+10)+10) + (((10+10)+10)+10) = 90 characters, instead of just 40 characters. And things gets quadratically worse as the number and size of the string increases. Java optimizes this case some of the times by transforming the series of concatenation to use StringBuilder.

Also some looping can be improved, for example if only apply linear search, no sorting is required.

Memory profiler can be used to profiling functions, but currently it reduces performance significantly, so didn’t use it, but tried on smaller data example. Though further study can be done based on profiling if it is required.

Filename: C:/Users/ROLLE/Google Drive/Data Merging/DataMergeTest(Binary Search).py

Line # Mem usage Increment Line Contents

================================================

82 94.2 MiB 94.2 MiB @profile(stream=fp)

83 def readJSON(self, filename):

84 94.2 MiB 0.0 MiB ln = 0

85 94.2 MiB 0.0 MiB start = time.time()

86 94.2 MiB 0.0 MiB print(filename)

87 94.2 MiB 0.0 MiB with open(filename, 'r') as f:

88 94.2 MiB 0.0 MiB while True:

89 94.2 MiB 0.0 MiB try:

90 95.1 MiB -0.0 MiB for line in f:

91 95.1 MiB 0.1 MiB title, category, city, country, score = self.extract\_inf\_from\_string(line)

92 95.1 MiB 0.0 MiB d = {"title":title, "category":category, "city":city, "country":country, "score":score}

93 95.1 MiB 0.1 MiB whole\_pattern = '\"{}\": {{\"category\": {{\"\w\*\": \d\*}},\"country\": \"{}\", \"city\": \"{}\"'.format(title,country,city)

94 95.1 MiB 0.0 MiB add = True

95 95.1 MiB -0.0 MiB ln = ln+1

96 """

97 self.update\_progress(round(ln/self.length,3))

98 print("Mergeing Successed Lines:{}".format(ln))

99 """

100 95.1 MiB -0.0 MiB if self.seen:

101 95.1 MiB 0.7 MiB add = self.linearSearch(d)

102 """

103 add2 = self.binarySearch(list(filtered\_list), 0, len(sorted\_list)-1, d)

104 """

105 95.1 MiB -0.0 MiB if add:

106 95.1 MiB -0.0 MiB self.seen.append(d)

107 else:

108 continue

109 95.1 MiB -0.0 MiB if not self.seen:

110 94.2 MiB 0.0 MiB self.seen.append(d)

111 95.1 MiB 0.0 MiB end = time.time()

112 95.1 MiB 0.0 MiB print("Merging Time: {}".format(end-start))

113 95.1 MiB 0.0 MiB with open("Logging.txt", "a") as file:

114 95.1 MiB 0.0 MiB file.write("File Name: {}, Total Lines: {}, Merging Time: {}".format(filename, ln, end-start)+"\n")

115

116 95.1 MiB 0.0 MiB return self.seen

117 except json.JSONDecodeError as e:

118 print(e)

119 continue

Finally, I wish I can learn more from your industrial practice to improve my understanding of system. Thank you indeed for this opportunity!