How Do API Selections Affect the Runtime Performance of Data Analytics Tasks?

Yida Tao[1], Shan Tang[1], Yepang Liu[2], Zhiwu Xu[1,*], and Shengchao Qin[3]

[1]College of Computer Science and Software Engineering, Shenzhen University, Shenzhen, China
{yidatao,tangshan2018,xuzhiwu}@szu.edu.cn
[2]Department of Computer Science & Engineering, Southern University of Science and Technology, Shenzhen, China
liuyp1@sustech.edu.cn
[3]School of Computing & Digital Technologies, Teesside University, UK, s.qin@tees.ac.uk

***Abstract*—As data volume and complexity grow at an unprecedented rate, the performance of data analytics programs is becoming a major concern for developers. We observed that developers sometimes use alternative data analytics APIs to improve program runtime performance while preserving functional equivalence. However, little is known on the characteristics and performance attributes of alternative data analytics APIs. In this paper, we propose a novel approach to extracting alternative implementations that invoke different data analytics APIs to solve the same tasks. A key appeal of our approach is that it exploits the comparative structures in Stack Overflow discussions to discover programming alternatives. We show that our approach is promising, as 86% of the extracted code pairs were validated as true alternative implementations. In over 20% of these pairs, the faster implementation was reported to achieve a 10x or more speedup over its slower alternative. We hope that our study offers a new perspective of API recommendation and motivates future research on optimizing data analytics programs.**

***Index Terms*—API selection, data analytics, performance optimization, Stack Overflow**

I. Introduction

Data analytics is the process of transforming raw data to actionable intelligence, and it is becoming increasingly important in this fast-developing era of AI and Big Data [1]. As the volume and complexity of data grow at an unprecedented rate, developers are often challenged by performance problems in the development of data analytics programs [1]. Although upgrading hardware or using more computing power could directly speed up data analytics tasks, such solutions are often expensive or impractical.

We observed that a more feasible and economic approach to optimizing data analytics programs is to exploit *software redundancy*: for reliability and usability concerns, modern software often offers multiple ways to complete the same tasks [2]. For this reason, developers have the opportunity to boost program performance by replacing the usage of a library API (or API sequence) with a faster alternative. Fig. 1 shows a real example. The code fragments at line 7 and line 11 both compute the magnitude of a given list of vectors (line 4). Yet, they use different APIs of *numpy*, a popular Python library for data analytics [3]. The runtime

```
1  In [1]: import numpy as np
2
3  # Input data
4  In [2]: a = np.arange(1200.0).reshape((-1,3))
5
6  # Solution 1
7  In [3]: %timeit [np.linalg.norm(x) for x in a]
8  100 loops, best of 3: 4.23 ms per loop
9                        Execution time 1
10 # Solution 2
11 In [4]: %timeit np.sqrt((a*a).sum(axis=1))
12 100000 loops, best of 3: 18.9 µs per loop
13                          Execution time 2
14
15 # Verify that the two solutions have the same output
16 In [5]: np.allclose([np.linalg.norm(x) for x in a],np.sqrt((a*a).sum(axis=1)))
17 Out[5]: True
```

Fig. 1. Computing the magnitude of vectors using different *numpy* APIs (excerpt from Stack Overflow post 9184560). One implementation is significantly faster than the other.

difference of these two solutions is tremendous: the one that uses `numpy.ndarray.sum` and `numpy.sqrt` is nearly 224x faster than the one that uses `numpy.linalg.norm`.

We refer to code fragment pairs like line 7 and line 11 in Fig. 1 as *alternative implementations*, which invoke *alternative APIs* to solve the same task by producing the same output for the same task input (see line 16). Developers could leverage alternative APIs as a cost-effective way to speed up their data analytics programs. Previous studies have also suggested that different API usages could affect program performance. Selakovic and Pradel found inefficient API usage to be the most common root cause of performance issues in JavaScript programs [4]. Yang et al. reported that half of the performance issues in Rails applications can be improved by changing how the Rails APIs are used [5]. However, these studies focus on different domains, and their evaluations on inefficient API usages were carried out manually on a small scale. In this paper, we propose a novel approach to automatically identifying alternative data analytics implementations, which lays the foundation for further studies on the prevalence, characteristics, and runtime performance attributes of alternative data analytics APIs.

Identifying alternative implementations for practical data analytics tasks is challenging, since developers typically do not keep alternative implementations in their code if one

*Zhiwu Xu is the corresponding author.

implementation is already sufficient for the task. Even if alternative solutions are implemented by different developers or in different contexts, it is hard to determine that they indeed solve the same tasks. In fact, identifying programming alternatives is a particular case of determining program equivalence, which is undecidable in general [6]. Previous work leverages techniques such as data flow analysis [7] and random testing [6] to detect functionally similar or equivalent programs.

In this paper, we tackle this problem from a novel perspective. We observed that alternative implementations are often discussed on Q&A sites such as Stack Overflow (SO for short), and such discussions usually involve some sort of comparison between the alternatives (e.g., Fig. 1 compares the execution time of two implementations). According to SO guidelines, users should post answers that directly address the question. Unconstructive or irrelevant answers might be downvoted or removed. For this reason, if two implementations are compared in an SO answer post, it is likely that they are alternative solutions to the task described in the corresponding question post. This observation motivates us to exploit the comparative structures in SO posts to discover alternative implementations.

II. Extracting Alternative Implementations

A. Extracting from Consecutive Profiling Statements

If a data analytics task has multiple solutions with different runtime performance, developers may profile these solutions in the same context in order to select the most efficient one. Based on this observation, we propose to extract alternative implementations from consecutive profiling statements, which can be found in SO code blocks encompassed by the `<pre><code>` tag. The type of profiling statements we search for is the one that executes *timeit*, which is the standard Python profiling command that measures execution time of small code fragments [8]. We match patterns where timeit is used from the command line, the Python interface, and from IPython [3]. Based on the respective timeit usage syntax, we extract the code fragment that is being profiled and the corresponding execution time, which is typically reported right after the timeit statement in SO code blocks. For example, from the code block shown in Fig. 1, we extract code fragments at line 7 and line 11 since they are being profiled by timeit consecutively. We also extract 4.23 ms and 18.9 μs as the corresponding execution time.

Note that two profiling statements are consecutive if there is no non-timeit code statements (i.e., regular code) between them, since non-timeit code may alter the input data, which violates the principle of input equivalence in our definition of alternative implementations. In addition, if there are n consecutive profiling statements in an SO code block, we extract all 2-combinations of them.

B. Extracting from Comparative Sentences

In addition to consecutive profiling statements, we also leverage the comparative structures in natural language to detect alternative implementations. First, we extract natural language text from SO answer posts and use the Stanford CoreNLP toolkit [9] to split the text into sentences. We then identify comparative sentences that contain efficiency-related comparative keywords such as "faster", "slower", and "more efficient". We proceed to identify the code fragments that are being compared in the comparative sentences. Following the common practice on detecting code-like terms from informal natural language discussions [10], we develop a set of regular expressions based on the target libraries' usage syntax. Contents matching these regular expressions or embedded in the `<code>` tag are considered code fragments. Since the presence of code fragments might negatively affect subsequent NLP tasks, we replace each detected code fragment with a unique identifier (e.g., `codefrag1`), which will be recovered once all NLP tasks are finished.

Fig. 2. An example of extracting alternative implementations using POS tagging, dependency parsing, and Semgrex matching at the sentence level.

Next, we perform Part-of-Speech (POS) tagging and dependency parsing to annotate each sentence with POS tags of each word (e.g., noun, adjective) and relations among words (e.g., nominal subject, conjunct) [9]. We then use *Semgrex* [11], a Stanford CoreNLP package that allows users to specify regular-expression-like patterns based on lemmas, POS tags, and dependency labels, to extract code fragments that are being compared. We have developed 49 Semgrex patterns for this purpose. These patterns use comparative keywords as anchors and follow certain dependency labels to search for words that start with `codefrag`.

Take the sentence "*I am iterating through all the rows of the dataframe using <code>.itertuples()</code> which is faster than <code>.iterrows()</code>*" from SO post 35108263 as an example. This sentence is first identified as comparative for containing the word "faster". Contents inside the `<code>` tag are replaced with `codefrag1` and `codefrag2`, respectively. After annotating the sentence using POS tagging and dependency parsing, we find a matching Semgrex pattern that has outgoing edges *nsubj* and *nmod:than* from "faster" to `codefrag*` words, which are extracted as the compared entities (Fig. 2). Finally, we recover the original code fragments and determine their performance ordering based on the meaning of the anchor word. In this example, `.itertuples()` is identified as a faster alternative to `.iterrows()`.

III. Emerging Results

A. Dataset

Our dataset includes 143,452 SO threads tagged with *numpy*, *pandas* and *scipy*, which are popular data analytics libraries in the Python data science ecosystem [3]. We used the official SO data dump released on December 2018 for data collection. To ensure that the extracted information is trustworthy, we consider only answer posts that are accepted

TABLE I
DATASET STATISTICS AND THE # OF VALIDATED ALTERNATIVE IMPLEMENTATION PAIRS FOR EACH LIBRARY.

Library	# SO threads	# SO answers	# validated alternative implementations
numpy	54,862	64,161	2,359
pandas	89,461	96,669	3,052
scipy	12,762	12,803	32

or have positive scores (i.e., they received more upvotes than downvotes). Table I shows the number of SO threads and trustworthy answer posts for each library.[1]

B. Alternative Implementations

We applied the approach described in Section II on the SO dataset and extracted 5,575 candidate implementation pairs. We programmatically validated whether each pair is truly alternative based on the input and output equivalence. Specifically, for each candidate implementation pair, we constructed a validation program that executes the pair on the same input to solve the task described in its SO thread. For most pairs extracted from consecutive profiling statements, their input variable definitions were directly extracted from the same code blocks (e.g., line 4 in Fig. 1). We manually restored the input definitions for the remaining pairs since those definitions typically locate in separate code blocks or other posts of the same SO thread. To determine output equivalence, our validation program checks the type of execution result and calls the corresponding object comparison method. For example, if two outputs of type `numpy.ndarray` are equivalent, then calling `numpy.allclose()` on them will yield to `True` (line 16 in Fig. 1).

Among the 5,575 extracted candidate pairs, 4,786 pairs (85.8%) were validated as true alternative implementations. [2] Regarding the extraction methodology, 5,412 candidate pairs were extracted from consecutive profiling statements and 4,652 (86%) of them were validated as true alternatives. On the other hand, 163 candidate pairs were extracted from comparative sentences and 134 (82%) were validated as true alternatives. This indicates that comparative structures in SO posts can indeed be leveraged to effectively reveal programming alternatives. Table II shows two examples of the extracted alternative implementations.

Table I shows the number of validated alternative implementations for each target library. The majority of alternative implementations use APIs of *pandas* and *numpy*, whereas only a few use *scipy*. Apart from the fact that *pandas* and *numpy* have a larger amount of SO posts, another reason for this phenomenon is probably because these two libraries are designed to work with low-level data structures. Specifically, *numpy* is used to work with arrays and *pandas* is used to work with tabular and time series data. Developers might have more flexibility when using these two libraries since they would be able to directly manipulate the data. *Scipy*, on the other hand, provides high-level algorithmic APIs for commonly used tasks in scientific computing. Therefore, it may not offer as much flexibility of using different APIs for the same tasks as *pandas* and *numpy* do.

[1]The sum of SO threads for all libraries is larger than 143,452 since a thread might have multiple tags.

[2]The sum of validated alternative implementation pairs in Table I is larger than 4,786 since a pair might invoke APIs of multiple libraries.

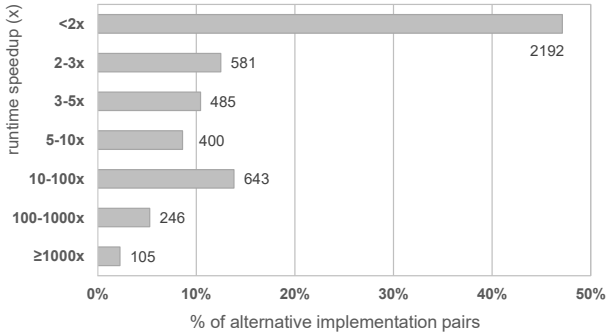

Fig. 3. Performance speedup of alternative implementation pairs.

C. Performance Impact of Alternative Implementations

As described in Section II-A, we extracted an implementation's execution time as well when we processed consecutive profiling statements in SO code blocks. These profiling results allowed us to analyze the performance difference of the 4,652 alternative implementation pairs extracted from consecutive profiling statements. Fig. 3 presents the results. We observed that in 2,192 pairs (47.1%), the faster implementations improve the task runtime performance by less than 2x. For the remaining 2460 pairs (52.9%), the faster implementations achieve 2x or more speedup over their alternatives. In particular, the faster implementations in 13.8% and 5.3% pairs achieve 10–100x and 100–1000x speedup, respectively. The faster implementations in 2.3% pairs even achieve more than 1000x speedup over their slower alternatives.

The results show that alternative implementations using different data analytics APIs do improve task runtime performance, and sometimes the improvement is quite significant. Hence, we believe that leveraging alternative APIs to optimize data analytics programs is a promising future direction.

D. Consistency Across Input Data Sizes

Various factors can affect the performance of alternative implementations. In our experiments, we observed that 10.3% of the SO posts containing consecutive profiling statements have compared the same alternative implementation pairs over different sizes of input data. This indicates that developers consider input data size to be important when evaluating the performance difference of alternative implementations. To further understand this issue, we quantify the performance difference of the extracted alternative implementations under five synthesized input data with sizes (i.e., # of rows) ranging from 100 to 1,000,000. Our experiments were performed on an Intel Core i9-8950 CPU (2.9GHz) machine with 32GB memory running 64-bit Windows 10 and Python 3.6.8.

Our preliminary results show that input data size can dramatically affect the extent of performance improvements. Fig. 4 shows two examples. In the first example,

TABLE II
EXAMPLES OF ALTERNATIVE IMPLEMENTATIONS, THEIR TASK DESCRIPTIONS, AND THE CORRESPONDING ALTERNATIVE API PAIRS.

	Task	Alternative Implementations	Alternative APIs
1	SO post 39132838: count the frequency of groups	`>>> df.pivot_table(index=['id','group'],columns='term',aggfunc='size',fill_value=0)` `>>> pd.crosstab([df.id, df.group], df.term)`	`pandas.DataFrame.pivot_table` `pandas.crosstab`
2	SO post 52145257: count the occurrence of a character in an array	`>>> np.sum(rr == 'A')` `>>> np.count_nonzero(rr == 'A')`	`numpy.sum` `numpy.count_nonzero`

`numpy.where` and `pandas.Series.map` are both used to alter data given a threshold, and `numpy.where` has a trivial speedup (∼1.5x) over `pandas.Series.map` when the input data is small. However, the speedup becomes significant (>100x) for input sizes larger than 100K. In the second example, `numpy.ndarray.dot` and `numpy.tensordot` are both used to compute the dot product of matrices, and `numpy.ndarray.dot` exhibits a non-trivial speedup (∼12x) over `numpy.tensordot` when the input size is small. However, their performance gap narrows as the input size grows. When the input size reaches 1 million, the two implementations have nearly comparable performance.

We also found cases where an alternative implementation exhibited both performance improvement and degradation on different input data sizes. For example, we found that `pandas.read_csv` is faster than `pandas.read_hdf` for input sizes less than 1K, whereas `pandas.read_hdf` takes the lead for larger inputs.

Our results show that input data size can be a critical factor that affects the outcome of performance optimization using alternative implementations. Specifically, an alternative solution that is allegedly much faster than the original implementation according to SO posts may turn out to be alike or even slower when the input data size varies. We believe that this issue requires further investigation in order to reduce potential risk of unintended optimization consequences.

IV. CONCLUSIONS

We have presented a novel idea of leveraging comparative structures in Stack Overflow discussions to discover alternative data analytics implementations. Our approach exploits crowd knowledge and the very nature of comparison to reveal programming alternatives, which is essentially different from conventional approaches that harness program analysis and testing. The alternative implementations extracted in this manner appear to have substantial performance difference according to the profiling results reported from Stack Overflow. This further indicates that a technique for detecting faster API alternatives is desirable. We also found that input data sizes often affect the performance comparison outcomes of alternative implementations, which, however, has been rarely studied in previous research. In the future, we plan to dive into the characteristics, root causes, and caveats of alternative data analytics APIs. We hope that our results establish a basis for developing techniques that automatically and reliably optimize data analytics programs.

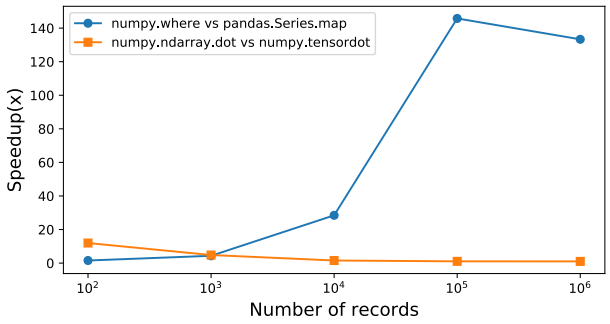

Fig. 4. Cases where an implementation remains faster than its alternative, but the extent of performance improvements varies dramatically.

ACKNOWLEDGMENTS

This work was partially supported by the National Natural Science Foundation of China under Grant No. 61772347, 61972260, 61932021 and 61802164.

REFERENCES

[1] C. E. Otero and A. Peter, "Research directions for engineering big data analytics software," *IEEE Intelligent Systems*, vol. 30, no. 1, pp. 13–19, Jan 2015.

[2] A. Carzaniga, A. Mattavelli, and M. Pezzè, "Measuring software redundancy," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, May 2015, pp. 156–166.

[3] W. McKinney, *Python for data analysis: Data wrangling with Pandas, NumPy, and IPython*. O'Reilly Media, Inc., 2012.

[4] M. Selakovic and M. Pradel, "Performance issues and optimizations in javascript: An empirical study," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, May 2016, pp. 61–72.

[5] J. Yang, P. Subramaniam, S. Lu, C. Yan, and A. Cheung, "How not to structure your database-backed web applications: A study of performance bugs in the wild," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: ACM, 2018, pp. 800–810.

[6] L. Jiang and Z. Su, "Automatic mining of functionally equivalent code fragments via random testing," in *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ser. ISSTA '09. New York, NY, USA: ACM, 2009, pp. 81–92.

[7] F.-H. Su, J. Bell, G. Kaiser, and S. Sethumadhavan, "Identifying functionally similar code in complex codebases," in *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, May 2016, pp. 1–10.

[8] "The Python standard library: Debugging and profiling," https://docs.python.org/3/library/debug.html.

[9] C. D. Manning, M. Surdeanu, J. Bauer, J. Finkel, S. J. Bethard, and D. McClosky, "The Stanford CoreNLP natural language processing toolkit," in *Association for Computational Linguistics (ACL) System Demonstrations*, 2014, pp. 55–60.

[10] P. C. Rigby and M. P. Robillard, "Discovering essential code elements in informal documentation," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 832–841.

[11] "Semgrex," https://nlp.stanford.edu/nlp/javadoc/javanlp/edu/stanford/nlp/semgraph/semgrex/SemgrexPattern.html.