Speeding up Data Manipulation Tasks with Alternative Implementations: An Exploratory Study

YIDA TAO, Shenzhen University, China
SHAN TANG, Shenzhen University, China
YEPANG LIU, Southern University of Science and Technology, China
ZHIWU XU, Shenzhen University, China
SHENGCHAO QIN, Teesside University, UK and Shenzhen University, China

As data volume and complexity grow at an unprecedented rate, the performance of data manipulation programs is becoming a major concern for developers. In this paper, we study how alternative API choices could improve data manipulation performance while preserving task-specific input/output equivalence. We propose a lightweight approach that leverages the comparative structures in Q&A sites to extracting alternative implementations. On a large dataset of Stack Overflow posts, our approach extracts 5,080 pairs of alternative implementations that invoke different data manipulation APIs to solve the same tasks, with an accuracy of 86%. Experiments show that for 15% of the extracted pairs, the faster implementation achieved >10x speedup over its slower alternative. We also characterize 68 recurring alternative API pairs from the extraction results to understand the type of APIs that can be used alternatively. To put these findings into practice, we implement a tool, AlterApi, to automatically optimize real-world data manipulation programs. In the 1,267 optimization attempts on the Kaggle dataset, 76% achieved desirable performance improvements with up to orders-of-magnitude speedup. Finally, we discuss notable challenges of using alternative APIs for optimizing data manipulation programs. We hope that our study offers a new perspective on API recommendation and automatic performance optimization.

CCS Concepts: • **Software and its engineering → Software libraries and repositories**.

Additional Key Words and Phrases: API selection, data manipulation, performance optimization, mining software repository, empirical study

ACM Reference Format:
Yida Tao, Shan Tang, Yepang Liu, Zhiwu Xu, and Shengchao Qin. 2021. Speeding up Data Manipulation Tasks with Alternative Implementations: An Exploratory Study. *ACM Trans. Softw. Eng. Methodol.* 1, 1, Article 1 (January 2021), 28 pages. https://doi.org/10.1145/3456873

This work is partially supported by the National Key Research and Development Program of China under Grant No. 2019YFE0198100, the National Natural Science Foundation of China under Grants No. 61772347, 61972260, 61836005, 61932021, 61802164, and the Guangdong Basic and Applied Basic Research Foundation under Grant No. 2019A1515011577. Shengchao Qin is the corresponding author.
Authors' addresses: Yida Tao, Shenzhen University, 3688 Nanhai Avenue, Shenzhen, China, yidatao@szu.edu.cn; Shan Tang, Shenzhen University, 3688 Nanhai Avenue, Shenzhen, China, tangshan2018@email.szu.edu.cn; Yepang Liu, Southern University of Science and Technology, 1088 Xueyuan Avenue, Shenzhen, China, liuyp1@sustech.edu.cn; Zhiwu Xu, Shenzhen University, 3688 Nanhai Avenue, Shenzhen, China, xuzhiwu@szu.edu.cn; Shengchao Qin, Teesside University, Middlesbrough, Tees Valley, UK, Shenzhen University, 3688 Nanhai Avenue, Shenzhen, China, s.qin@tees.ac.uk.

1049-331X/2021/1-ART1 $15.00
https://doi.org/10.1145/3456873

```
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.

1 INTRODUCTION

Data manipulation is the process of extracting, filtering, and transforming unorganized raw data for better readability and usablity. As data manipulation is becoming increasingly important in this fast-developing era of AI and Big Data [42, 54], various tools and libraries have emerged to support this type of task. Yet, the volume and complexity of data also grow at an unprecedented rate [42]. As a result, developers are often challenged by performance problems in the development of data manipulation programs, especially when the data is dynamic, subject to concept drift, and expected to be processed within specified time constraints [42, 54]. Although upgrading hardware or using more computing power could directly speed up data manipulations, such solutions are often expensive or impractical.

We observed that a more feasible and economic approach to optimizing data manipulation programs is to exploit *software redundancy*: for reliability and usability concerns, modern software often offers multiple ways to complete the same tasks [23]. 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, which is a common data manipulation task. Yet, different APIs of *NumPy*, which is a popular Python library for array manipulations [39], were used in these two solutions and the runtime difference 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 (line 16) for the same input (line 4). Developers could leverage alternative APIs as a cost-effective way to speed up their data manipulation programs. Previous studies have also suggested that different API usages could affect the performance in Java [31, 41], Androids [33–35], JavaScript [47], and Rails applications [51]. However, few studied this problem in the domain of data manipulation. Furthermore, the aforementioned studies mostly relied on predefined templates or manual efforts to identify inefficient API usages and potential alternatives, which is hardly scalable or extensible. To bridge this gap, we propose an approach to automatically identifying alternative implementations and we focus on data manipulation implementations in this work.

Identifying alternative implementations for practical data manipulation tasks presents a unique challenge, as developers typically do not keep alternative implementations in their code if one 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 whether they can indeed be used to solve the same tasks. In fact, identifying alternative implementations in terms of input/output equivalence is a special case of determining program equivalence, which is undecidable in general [30]. Prior research has leveraged techniques such as data flow analysis [48] and random testing [30] to detect functionally similar or equivalent programs.

In this paper, we tackle this challenge 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. Nonconstructive or irrelevant answers might be downvoted or even removed. For this reason, if two implementations are being compared in the same SO answer post, they are likely to be alternatives solutions for the same task proposed in the corresponding SO question post.

Based on the above observation, we propose an automatic approach to extracting alternative implementations from the *comparative structures* in SO answer posts. Specifically, we consider *consecutive profiling statements* and *comparative natural-language sentences* to be interesting comparative structures that are worth exploring. We aim to answer the following research questions:

- **RQ1 (Effectiveness)**: How effective are comparative structures from SO posts at revealing alternative data manipulation implementations?
- **RQ2 (Performance)**: What is the runtime performance difference of alternative implementations?
- **RQ3 (Characteristics)**: What are the characteristics of alternative data manipulation APIs?
- **RQ4 (Usefulness)**: How can we leverage the knowledge of alternative APIs to optimize real-world data manipulation programs?

We applied our approach on a dataset of 181,500 SO threads tagged with *NumPy*, *Pandas* and *SciPy*, which are popular data manipulation libraries in the Python data science ecosystem [16]. Our approach extracted 5,933 candidate implementation pairs from the dataset. We validated the input/output equivalence of these candidates via random testing and manual inspection, and found that 5,080 (86%) are true alternative implementations. By executing these pairs experimentally, we found that in 15% of the alternative implementations, the faster alternatives achieve a >10x speedup over the slower implementations, which illustrates the potential benefits of leveraging alternative implementations for performance optimization.

We abstracted API call sequences from the validated alternative implementations and identified 68 alternative API pairs that are mentioned in at least three SO posts. We manually characterized these recurring alternative API pairs and made the following observations. First, 60% pairs (41/68) involve APIs that are designed for similar types of tasks, whereas 40% pairs (27/68) involve conceptually different APIs that can only be used alternatively for certain tasks. Second, 18% (12/68) pairs use a specific API as an alternative to a generic API, while 13% pairs (9/68) use a concrete API as an alternative to a higher order API that takes a task-related function as an argument. Third, we found that for 46% (31/68) pairs, the documentation of one API contains explicit reference (e.g., hyperlinks) to its alternative API. These findings shed light on the type of alternative APIs and the reasons they can be used alternatively, which may help developers make better-informed decisions on selecting data manipulation APIs.

To put our findings into practice, we implemented a tool, named AlterApi, which leverages the knowledge of recurring alternative APIs to automatically optimize real-world data manipulation programs written in Python. In our experiments, AlterApi generated 1,267 valid alternative

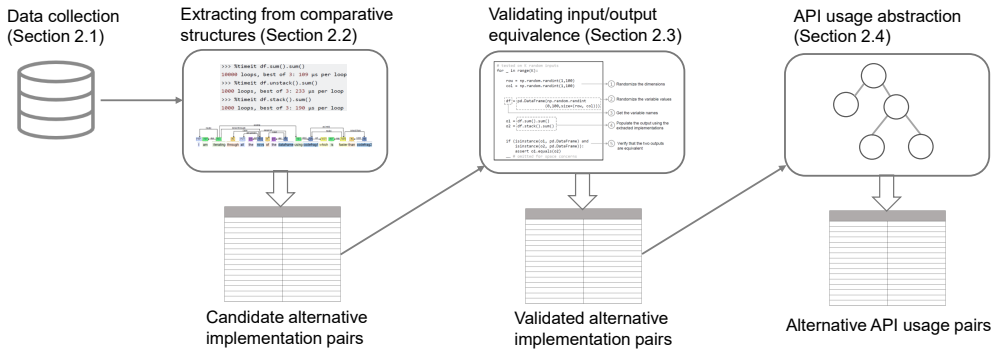


Fig. 2. Overview of the methodology.

implementations for 398 data manipulation programs collected from the *Kaggle* platform [6]. Our profiling shows that 76% of the generated alternatives speed up the original implementations, with a median speedup of 1.7x and a max speedup over 2000x. Finally, we discuss the generality of our approach as well as the challenges of optimizing data manipulation programs with alternative APIs.

This work extends our previously published new idea paper [49]. We have made significant improvements in this extension, including: (1) refining the approach and updating the dataset; (2) conducting a more robust validation of alternative implementations; (3) experimentally reproducing the profiling results reported on SO; (4) qualitatively characterizing alternative API pairs; (5) applying the mined knowledge to optimize real-world programs; and (6) discussing the challenges and generality of our approach with additional experiments. To summarize, this paper makes the following contributions:

- To the best of our knowledge, this is the first work to extract, characterize, and profile alternative data manipulation implementations. We provide empirical evidence that using alternative APIs to speed up data manipulation tasks is a common practice, and that the performance improvements can be substantial.
- We propose and implement a lightweight approach to automatically and effectively extracting alternative implementations by leveraging the comparative structures in SO posts.
- We qualitatively analyze the characteristics of alternative data manipulation APIs and leverage this knowledge to build a tool for automatic performance optimization, which achieves promising results on realistic data manipulation programs.
- We released a replication package for this project to facilitate future research.[1]

2 METHODOLOGY

In this section, we introduce our research methodology of identifying alternative implementations and APIs. As illustrated by Figure 2, from the collected SO posts (Section 2.1), we first extract candidate alternative implementations from the comparative structures (Section 2.2). We then leverage random testing to validate whether the extracted candidates are true alternatives (Section 2.3). Finally, we abstract API usages from the validated alternative implementations to identify alternative APIs (Section 2.4).

[1]Replication package: https://sites.google.com/view/alterapi-artifacts/

Table 1. The SO dataset used in our experiments. The last two columns show the number of extracted alternative implementation pairs and the number of validated pairs for each library.

Library	# SO threads	# SO answers	# extracted pairs	# validated pairs
NumPy	64,468	72,517	2968	2,413
Pandas	118,356	122,355	3426	3,134
SciPy	14,581	13,854	59	31

2.1 Dataset

In this study, we focus on three data manipulation libraries: *NumPy*, *Pandas*, and *SciPy*. We select these libraries mainly for three reasons. First, they are all core packages of the Python ecosystem for scientific computing [39]. Second, they aim at solving different classes of problems and together support various important tasks in data manipulations: Pandas provides high-level functions to work with tabular or structured data [9]; SciPy provides common numerical routines in science and engineering [15]; both libraries are built on top of NumPy, which is the fundamental package for efficient array computations [8]. In addition, these libraries are extremely popular and widely used. Stack Overflow recently reported that while Python is becoming the fastest-growing programming language, much of its popularity can be ascribed to libraries like Pandas and NumPy, which are among the top tags that are most often visited by Python users [45].

We used the official Stack Overflow data dump released on August 2019 as our data source and collected 181,500 threads that have numpy, pandas, or scipy tags [18]. Table 1 shows the number of SO threads and answer posts for each library.[2] Note that to ensure the extracted information is trustworthy, we consider only answer posts that are accepted or have positive scores (i.e., they received more upvotes than downvotes).

2.2 Extracting Candidate Alternative Implementations

As mentioned in Section 1, we consider *consecutive profiling statements* and *comparative natural-language sentences* as interesting structures that likely entail alternatives. We now describe how we extract such information from each structure.

2.2.1 Extracting from Consecutive Profiling Statements. If a data manipulation 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 (e.g., Fig. 1). Based on this observation, we propose to extract alternative implementations from consecutive profiling statements, which can be found in code blocks that are encompassed by the `<pre><code>` html tags in SO answer posts.

The profiling statements we search for are those that execute `timeit` [12], which is the standard Python profiling command that measures the execution time of small code snippets.[3] Specifically, the `timeit` command executes a given code snippet N times in a loop, repeats the loop R times, and reports the best average of the R repetitions. The number of loops and repetitions can be specified by developers or automatically determined by the `timeit` module during runtime [12]. We match patterns where `timeit` is used from the command line (`python -m timeit ''code fragment''`), from the Python interface (`timeit.timeit(''code fragment'')`), or from IPython (`%timeit`

[2]The sum of SO threads for all libraries is larger than 181,500 since a thread might have multiple tags.

[3]We do not consider consecutive commands that profile memory usage (e.g., `memit` [5]), since we found very few SO answer posts (0.0001%) containing consecutive memory profiling commands in our dataset, and most of them did not intend to compare alternative implementations.

```
1  a = np.arange(1000, dtype=np.double)
2  %timeit np.einsum('i->', a)
3  100000 loops, best of 3: 3.32 µs per loop
4  %timeit np.sum(a)
5  100000 loops, best of 3: 6.84 µs per loop
6
7  a = np.arange(10000, dtype=np.double)
8  %timeit np.einsum('i->', a)
9  100000 loops, best of 3: 12.6 µs per loop
10 %timeit np.sum(a)
11 100000 loops, best of 3: 16.5 µs per loop
```

Fig. 3. Code block from SO post 18365665. Line 4 and line 8 are not considered as consecutive since line 7 alters the input data.

```
1  >>> %timeit df.values.sum()
2  100000 loops, best of 3: 6.27 µs per loop
3  >>> %timeit df.sum().sum()
4  10000 loops, best of 3: 109 µs per loop
5  >>> %timeit df.unstack().sum()
6  1000 loops, best of 3: 233 µs per loop
7  >>> %timeit df.stack().sum()
8  1000 loops, best of 3: 190 µs per loop
```

Fig. 4. Code block from SO post 32340834. The 4 consecutive implementations can all be used to sum values in a dataframe. We extract 6 pairs of alternative implementations from this code block.

`code_fragment`). According to their respective usage syntax [5, 19], 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 respective execution time. This runtime information is used to analyze the performance differences between alternative implementations (RQ2).

We consider two profiling statements to be *consecutive* if there is no non-timeit code statement between them. Fig. 3 provides an example. In this code block, lines (2, 4) and lines (8, 10) are consecutive profiling statements. However, between line 4 and line 8, a non-timeit code at line 7 alters the input data, which violates the principle of input equivalence in our definition of alternative implementations. Therefore, we do not consider line 4 and line 8 to be consecutive. Although this method might lead to false negatives in finding alternative implementations, our approach in general is cost-effective and collected thousands of alternative implementations (Section 3.1), which are already sufficient for carrying out our empirical study.

Finally, if there are n consecutive profiling statements in an SO post, we extract all 2-combinations of them. Fig. 4 shows a code block containing 4 consecutive profiling statements, from which our approach will extract $C(4, 2) = 6$ pairs of alternative implementations.

Since we focus on Python-based data manipulation programs, this part of our approach leverages several language-specific features of Python. However, the idea of extracting programming alternatives from consecutive profiling statements is language-agnostic, and it should be easily adaptable to other programming languages once their profiling syntax is known. We further discuss this issue in Section 4.4.

2.2.2 Extracting from Comparative Sentences. In addition to consecutive profiling statements, we also leverage the comparative structures in natural language to detect alternative implementations. Note that natural language text in an SO answer post denotes all of its content except the code blocks, and natural language sentences that compare multiple code implementations would be the structure of interest.

We first use the Stanford CoreNLP toolkit [37] to split the text into sentences. We then identify sentences that contain efficiency-related comparative keywords (Table 2), which were derived based on common knowledge and the literature [35, 47]. Next, we identify code fragments in these sentences following the common practice on detecting code-like terms from informal natural language discussions [44, 50]. Basically, we develop a set of regular expressions based on the target libraries' usage syntax; contents matching these regular expressions or embedded in the

Table 2. The list of efficiency-related comparative keywords used in our study.

Type	Keywords
comparative adjectives	faster, slower, quicker, cheaper
comparative adverbs	more/less efficient/expensive/scalable
equative as	as fast/slow/quick/cheap/efficient/expensive/scalable as

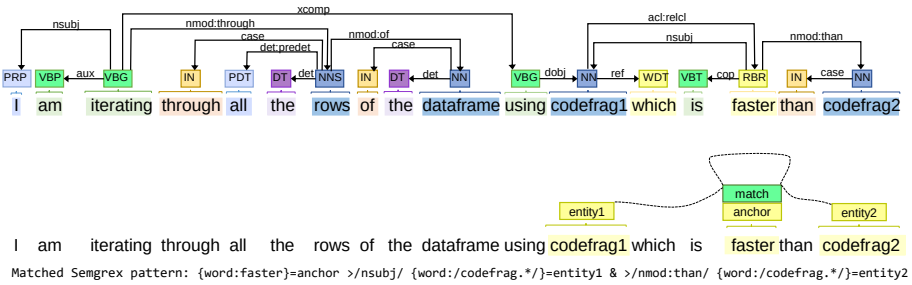


Fig. 5. An example of extracting alternative implementations using POS tagging, dependency parsing, and Semgrex matching at the sentence level.

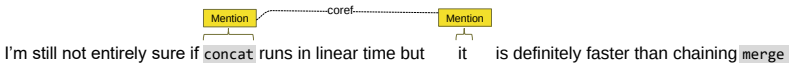


Fig. 6. Coreference resolution is also used if a pronoun is identified as the compared entity.

<code> tag are then considered as code fragments. We filtered out sentences that contain less than two code fragments (e.g., "*X is the fastest*"), since we cannot easily infer a pair of alternative implementations from such implicit comparisons. Finally, since the presence of code fragments might negatively affect subsequent NLP tasks [52, 53], we replace each detected code fragment with a unique identifier (e.g., `codefrag1`), which will be recovered once all NLP tasks are finished.

For candidate sentences that contain efficiency-related comparative keywords and multiple code fragments, we need to further determine which implementation is more efficient than the other(s). To this end, we leveraged *dependency patterns*, which describe specific natural language structures based on Part-of-Speech (POS) tags (e.g., noun, adjective, adverb) and dependency labels between words (e.g., nominal subject, object, conjunct). Fig. 5 shows an example of a dependency pattern, which has an *nsubj* edge from "faster" to *codefrag1* and an *nmod:than* edge from the same comparative adverb to *codefrag2*. To derive such eligible dependency patterns, we first reused seven general dependency patterns for comparative sentences, which were proposed in [27], to match candidate sentences. For candidates that were not matched in this first step, we randomly inspected 100 more sentences and further identified 42 new dependency patterns that were not covered in [27]. In total, 49 dependency patterns were derived. In the implementation, we represent these patterns using the *Semgrex* Stanford CoreNLP package [17], which allows users to specify dependency patterns in a regular-expression-like style.

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 keyword "faster". Contents inside the <code>

tag are replaced with `codefrag1` and `codefrag2`, respectively. We then annotate the sentence using POS tagging and dependency parsing and match it against our predefined Semgrex patterns. As shown in Fig. 5, this sentence matches a Semgrex pattern that has outgoing edges *nsubj* and *nmod:than* from "faster" to `codefrag`$*$ words, which are extracted as the compared entities. Finally, we recover the original code fragments and determine their performance ordering based on the meaning of the anchor word. As a result, `.itertuples()` is identified as a faster alternative to `.iterrows()`.

Note that the presence of negation in a sentence could alter the performance ordering of the extracted code fragments. For example, for sentences like "*A is not faster than B*" or "*I don't think A is faster than B*", it is more appropriate to consider `A` as a slower alternative to `B`. Hence, five of our predefined Semgrex patterns are used to identify negations in a sentence by explicitly matching the *neg* edge in its dependency tree. The performance ordering of code fragments extracted from the detected negative sentences are swapped accordingly.

We also observed cases where the word being compared is a *pronoun* that refers to a code fragment. Since direct Semgrex matching cannot detect such cases, we further perform *coreference resolution* [43] on the comparative sentences. Fig. 6 shows a sentence from SO post 40568957, where *it* is first identified as one of the compared entities by Semgrex matching. Since *it* is a pronoun, we use *CorefAnnotator* [43] in Stanford CoreNLP to check if there is any code fragment that is referred to by this pronoun. In this case, *it* refers to `concat`, which is detected as a faster alternative to `merge`.

2.3 Validating Task-Specific Input/Output Equivalence

After extracting candidate implementation pairs from comparative structures in SO posts, we need to validate whether the implementations in each pair are truly alternative. As introduced in Section 1, we define two implementations to be alternative in terms of task-specific input/output equivalence. We leverage the random testing approach proposed in the work of Jiang and Su [30] for such validations. Specifically, if two implementations always produce the same outputs on a selected number of random task inputs, then we are confident that they are truely alternative for this particular task [30]. As explained in [30], this assumption leveraged the Schwartz-Zippel lemma [46], which states that a few random tests are sufficient to determine whether two polynomials are equivalent with high probability.

We developed an algorithm to automatically generate executable random-testing code for each candidate implementation pair. The random-testing workflow is defined in the code template shown in Fig. 7, which instantiates input variables with random values, executes the given implementation pair, compares the equivalence of the two outputs, and runs the entire process for K times.

One major challenge here is that the automatically generated code should instantiate random task inputs with proper data, otherwise the generated code will not execute properly. For example, to validate whether `np.concatenate([va,vb],0)` and `np.vstack([va,vb])` are true alternatives for the data merging task (SO post 27138503), we need to initialize the input variables `va` and `vb` to be of `numpy.ndarray` type with matching dimensions. To tackle this problem, we design our testing template to have five parts as shown in Fig. 7 — each part is automatically constructed with different strategies. Below we elucidate the code generation strategy for each part of the random-testing code.

① **Randomizing the dimensions**: First, we randomize *dimensionality*, which is often required to create typical data manipulation inputs including arrays, matrices and tables. In particular, we create three random integers, which can be used to generate 1D, 2D and 3D data that are commonly used in SO tasks.

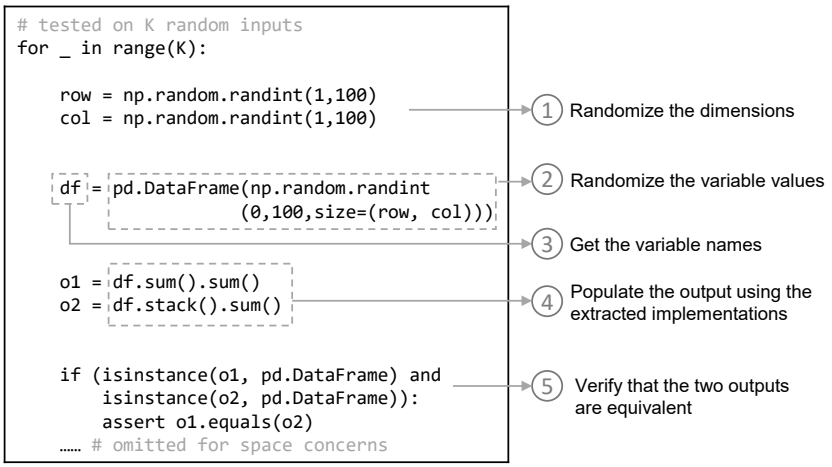


Fig. 7. The code template for random testing. ① and ② are for randomizing different types and values of the input; ③ is the name of the input variable extracted from the candidate implementation pair, which is used to instantiate output variables in ④. ⑤ verifies the output equivalence.

② **Randomizing the variable values**: Next, we generate the input instantiation code using common *container types* and *element types* used in data manipulation tasks. Specifically, we consider three container types including `pandas.DataFrame`, `pandas.Series`, and `numpy.ndarray`, and four element types including `int`, `float`, `bool`, and `string`. Accordingly, we create input instantiation code for the 20 combinations[4] of the container and element types, such as 2D `numpy.ndarray` of floats, `pandas.Series` of integers, and `pandas.DataFrame` of strings. Dimensions are randomized as in ① and values are randomized using the corresponding random-generator APIs such as `numpy.random.rand`.

③ **Extracting the variable names**: To execute the generated code properly, the names of the input variables should be consistent with those used in the given implementation pair. We determine the input variable names by extracting all the `Name` nodes from the abstract syntax tree of the given implementation. Note that there could be multiple input variables that need to be instantiated. For efficiency, our algorithm only considers cases where the number of input variables is 1 or 2. The values generated in ② are assigned to these variables, respectively. As one variable has 20 possible types and two variables have $20 \times 20 = 400$ possible type combinations, our algorithm will automatically generate $20+400 = 420$ different pieces of input instantiation code for each candidate implementation pair.

④ **Output instantiation**: We assign the two implementations in the given candidate pair to `o1` and `o2`, which are used as the output variables.

⑤ **Verifying output equivalence**: The last part of the template involves code that `asserts` the equivalence of `o1` and `o2` based on their types. This part of the template code is predefined and it checks the equivalence of all the container and element types described in ②.

Fig. 8 shows the random-testing code generated for the task described in Fig. 1, for which the input variable is a 2D array of integers. The code randomizes the integer values as well as the array

[4]The number of combinations is computed as $4 + 4 + 4 \times 3 = 20$, since we consider 1D, 2D and 3D dimensions for `numpy.ndarray`.

Task: Computing the magnitude of a list of vectors (SO post 9184560)

Input: a 2D numpy.ndarray of integers

```
1   # tested on 10 random inputs
2   for k in range(10):
3       # randomizing the dimension
4       row = np.random.randint(1,100)
5       col = np.random.randint(1,100)
6       # randomizing the input variable
7       a = np.random.randint(0,100, size=(row,col))
8       # output
9       o1 = np.sqrt((a*a).sum(axis=1))
10      o2 = [np.linalg.norm(x) for x in a]
11      # verify that the two outputs are equivalent
12      if (isinstance(o1, np.ndarray)
13                  and isinstance(o2, np.ndarray)):
14          np.testing.assert_equal(o1,o2)
```

Fig. 8. The random testing code generated for validating the implementation pair from Fig. 1.

dimensions when instantiating the input (line 4, 5, and 7), whose variable name `a` is extracted from the given implementation pair (line 9 and 10).

Given a candidate implementation pair, our algorithm automatically generates and executes the random testing code. Once there exists a test code that runs successfully without `Assertion Error` for all K executions, the process stops and we consider the given implementation pair to be valid alternatives. We followed [30] to set K to be 10, which effectively limits the execution time while preserving sufficient confidence. For implementation pairs that fail on all 420 random testing attempts, however, we could not directly consider them as invalid for two reasons. First, random testing may fail due to inappropriate input instantiations rather than `Assertion Error`. Specifically, inputs that are not taken into account in our initial algorithm include arrays with higher dimensions, sparse matrices, and variables that are particularly task-specific. For example, SO post 57299067 requires the second dimension of the input array to be 2; SO post 18876279 requires one of the two input arrays to be monotonically increasing or decreasing; SO post 39132838 requires the input DataFrame to have one categorical data column and two integer columns indicating the groups. In such cases, the random testing code failed since our algorithm did not generate the proper task inputs in the first place. Second, implementations extracted from comparative sentences could be incomplete (e.g., `.itertuples` in the example of Fig. 5). As a result, the generated random testing code would fail with `SyntaxError` or `AttributeError` rather than `Assertion Error`.

To handle the abovementioned issues, we manually inspected corresponding SO threads to determine proper input instantiations and recover incomplete implementations. Note that such information is in general easily accessible from SO threads, especially from those with accepted or upvoted answers, since SO explicitly enforces users to provide minimal and reproduciable examples when asking and answering questions [4].

In our experiment, 774 SO threads were manually inspected by the first author, who had three years of industrial experience in data science. The workload of this task is ~65 man-hours, approximately five minutes for each SO thread. It is worth mentioning that task inputs from SO threads were often presented in a highly flexible and unstructured manner. For instance, task inputs could be described as code, table, image, and natural-language text, and the descriptions could be located in question posts, answer posts where candidate implementation pairs are extracted from, or other answer posts of the same thread. This observation further illustrated the necessity of manual inspection for faithfully recovering proper task inputs.

Whenever a new type of input was discovered, we added its instantiation method to our algorithm and reran the algorithm on the remaining implementation pairs. In other words, the algorithm was

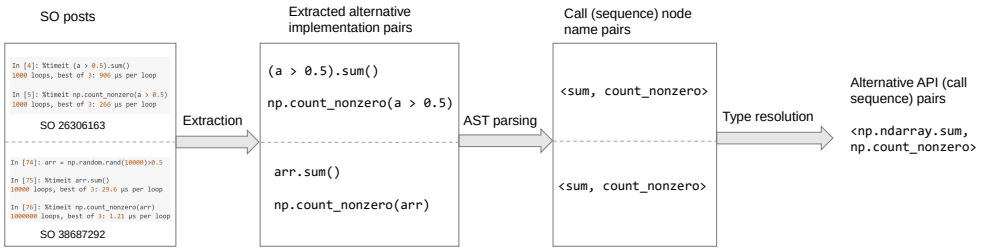


Fig. 9. An example of API usage abstraction, which identifies a same alternative API usage pattern from two different implementation pairs.

incrementally improved and executed until all candidate pairs are considered as either valid (i.e., there exists a test that runs successfully without `Assertion Error` for all K inputs) or invalid (i.e., there exists a test that fails with `Assertion Error` on one or more inputs).

2.4 Characterizing Alternative API Pairs

Given the validated alternative implementations, we now identify emerging patterns from these implementation pairs in order to study the characteristics of alternative APIs (RQ3). Fig. 9 shows an illustrative example of this process, in which two alternative implementation pairs were extracted from two different SO posts. Although these two pairs use different variable names and syntax in argument passing, they essentially represent the same alternative usage pattern, in which `numpy.ndarray.sum` and `numpy.count_nonzero` can both be used to count the occurrence of certain elements in an array.

To better capture such patterns, we developed an algorithm to model each concrete implementation in the extracted pairs as an API call sequence, which abstracts away syntactic details but preserves the essence of API usage [55]. Our algorithm first uses the Python *ast* module [10] to parse a code fragment into an abstract syntax tree, on which it performs a post-order traversal on the `ast.Call` and `ast.Subscript` nodes to get the API sequence [56]. Note that we can only acquire the simple names of APIs in this step, whereas their specific types (or fully qualified names [11]) are still ambiguous. For example, the name `sum` in the previous example (Fig. 9) may refer to either `numpy.ndarray.sum`, `pandas.DataFrame.sum`, or `scipy.ndimage.sum`.

For this reason, we further resolve the type of each simple name in the extracted sequence. Our algorithm first uses SO tags to determine which library a simple name may refer to. The algorithm then matches the simple name against all APIs of that library to ensure that APIs with the same simple name do exist. To further reduce ambiguity, our algorithm applies regular expressions on the name of the receiver object to determine its type information based on naming conventions. For example, `df` and its variations (e.g., `df1` and `df_raw`) are conventional names for the `pandas.DataFrame` type, while `arr` and its variations are typically used for the `numpy.ndarray` type. With these heuristics, concrete implementations can be uniquely resolved (e.g., `arr.sum()` in Fig. 9 is resolved to the `numpy.ndarray.sum` type).

We applied this algorithm to all the validated alternative implementations and aggregated the resulting alternative API (call sequence) pairs in descending order of occurrence. We then selected pairs that recurred at least three times to study their characteristics, as these pairs appeared in at least three different SO posts and are therefore more likely to represent common usages of alternative APIs. Since there is no existing characterization on this type of data, we applied the *inductive coding* method [24] during this process. Basically, two of the authors browsed through the recurring API pairs and proposed an initial list of categories to characterize the data. Next, they

Table 3. The number of extracted/validated alternative implementation pairs with respect to different comparative structures.

Structure	# extracted	# validated
Consecutive profilings	5,757	4,935 (85.7%)
Comparative sentences	176	145 (82.4%)
Total	5,933	5,080 (85.6%)

independently labeled each of the recurring pairs using the established categories. Finally, these labels were compared, discussed and adjusted in an iterative fashion until an agreement has been reached. We present the characterization results in Section 3.3.

3 RESULTS

In this section, we present the experiment results with respect to each research question.

3.1 RQ1: Effectiveness

We applied the approach described in Section 2 on the SO dataset and extracted 5,933 candidate implementation pairs, among which 5,080 pairs (85.6%) from 1,867 answer posts were validated as true alternative implementations. This indicates that comparative structures in SO posts can indeed be exploited to effectively reveal programming alternatives.

Table 3 presents the number of alternative implementation pairs extracted from different comparative structures. From consecutive profiling statements, 5,757 candidate pairs were extracted and 4,935 (85.7%) of them were validated as true alternatives. The primary reason for false positives is that consecutive profiling statements are sometimes used to break down the execution cost of an implementation by profiling every single step of it, rather than to compare alternative implementations like we assumed.

From comparative sentences, 176 candidate pairs were extracted and 145 (82.4%) were validated as true alternatives. Part of the inaccuracy was due to the parsing error of complicated and ungrammatical/colloquial sentences, which is a well-known issue in the NLP domain [28]. In addition, natural language sentences sometimes elide important context information, which also leads to incorrect extractions. For example, the sentence "*flatnonzero is faster than where*" in SO post 47068979 intends to propose that using `numpy.flatnonzero` together with `numpy.unravel_index` is faster than `numpy.where`. However, since `numpy.unravel_index` is omitted from the sentence, the implementation pair extracted from it (i.e., `flatnonzero` and `where`) is therefore incomplete and should not be considered as valid alternatives. Overall, the top three types of comparative sentences are in the forms of "*X is faster than Y*"(58.6%), "*X ..., and is faster than Y*"(9.0%), and "*X is ... times faster than Y*"(6.2%). In particular, the first pattern already accounts for over half of the instances, while the top three patterns together account for nearly 74% instances. The remaining patterns in the long tail of the distribution are mostly resulted from the flexibility of natural language sentences. For example, two Semgrex patterns are needed to match "*X is faster than Y*" and "*X runs faster than Y*", which have practically the same meaning but different dependency relations. A complete list of Semgrex patterns for extracting comparative sentences is available on our project website.

The alternative implementations extracted from comparative sentences are far fewer than those extracted from consecutive profiling statements. A primary reason is that the requirement of explicitly comparing multiple code-like terms limits the number of eligible sentences. The sentence "*... it returns exactly the same thing as @Justin Peel's code above and runs ~100x faster*" from SO post 3443640 is such an example. The pronoun *it* refers to an implementation mentioned in the prior

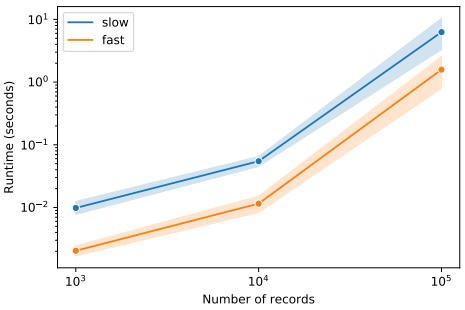


Fig. 10. Avg. execution time of validated alternative implementation pairs under varied workload.

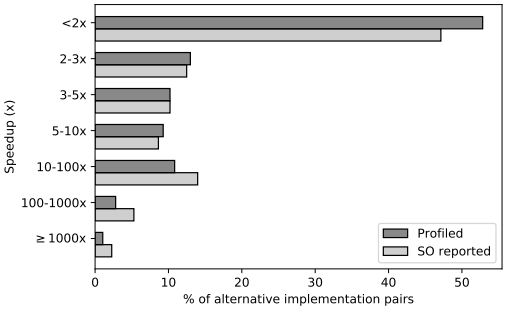


Fig. 11. Runtime speedups of alternative implementations.

paragraph of the sentence, and this implementation is being compared with a solution proposed by another user. Our approach currently works at the sentence level and hence could not handle such cases. Nonetheless, comparative sentences do appear to have rich information on alternative implementations, and extending our approach to SO thread level might help extract more of such pairs.

Table 1 shows the number of validated alternative implementations for each target library. The majority of these 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 that 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 common 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. Finally, among the 5,080 alternative implementations, 1,306 use APIs from different libraries to solve the same tasks. For example, `numpy.cumprod` and `pandas.DataFrame.cumprod` is an alternative pair as both can be used to compute the cumulative product of data. For this reason, the numbers of extracted and validated alternative implementation pairs (i.e., 5,933 and 5,080, respectively) are less than the sums of respective numbers for each library (Table 1), since multiple libraries could be counted for one single implementation pair.

Consecutive profiling statements and comparative sentences can both be leveraged to effectively reveal alternative implementations. Nearly 86% implementation pairs extracted from these structures are validated as true alternatives.

3.2 RQ2: Performance

To address RQ2, we experimentally collected the runtime performance of the validated alternative implementations. Specifically, for each alternative implementation pair, we automatically generated a *profiling program* by modifying its generated random-testing code (see Section 2.3) as follows. First, a code snippet that records the execution time of each implementation is added to the end of the random-testing code. Second, instead of randomly generating the size of input (e.g., line 4 in Fig. 8), we now set the number of rows to be 1,000, 10,000 and 100,000, which replaces the original loop of K random inputs (e.g., line 2 in Fig. 8). By executing the profiling programs, we were able to collect the execution time of alternative implementation pairs under a controlled workload.

Table 4. Occurrence of alternative API pairs. Pairs with ≥ 3 occurrences are used to study the characteristics.

Occurrence	# of alternative API pairs.
[1, 2]	2901
[3, 5]	56
[6, 10]	10
[11, 15]	2

We conducted the experiment on an Intel Core i9-8950 CPU (2.9GHz) machine with 32GB of memory running 64-bit Windows 10 and Python 3.6.8. Fig. 10 shows the average execution time of validated alternative implementations under varied workload. Fig. 11 further shows the runtime speedups. We observed that in 52.8% pairs, the faster implementations improve the task runtime performance by less than 2x. The faster implementations in 10.8% and 2.8% pairs achieve 10–100x and 100–1000x speedup, respectively. The faster implementations in 1.1% pairs even achieve more than 1000x speedup over their slower alternatives.

As described in Section 2.2.1, SO users tend to report an implementation's execution time right after its profiling statement in SO code blocks. We also extracted this information and computed the corresponding speedups for comparison. As shown in Fig. 11, the performance improvements reported from SO are relatively consistent with the profiling results collected in our experiments.

Alternative implementations using different data manipulation APIs do improve task runtime performance, and in many cases the improvement is quite significant.

3.3 RQ3: Characteristics

To characterize alternative API pairs, we abstracted each implementation to its API call sequence as described in Section 2.4. There are 5,123 unique implementations in the 5,080 validated alternative implementation pairs. By manual evaluation, we found that 4,661 (90.1%) implementations were resolved correctly. The primary reason for inaccurate resolutions is that our approach could incorrectly resolve a call to an API with the same simple name but belonging to another module or library (i.e., with different fully qualified names) due to the lack of context.

In total, we identified 2,969 distinct alternative API (call sequence) pairs from the 5,080 validated alternative implementations. Table 4 aggregates these alternative API pairs by their occurrences in the extracted data. As described in Section 2.4, two of the authors manually characterized the 68 recurring pairs with ≥3 occurrences. The Cohen's kappa coefficient [38] for this task is 0.79, which means that the two annotators had substantial agreement (kappa ∈ [0.61–0.8]) for characterizing alternative API pairs. Below we present our observations.

3.3.1 APIs with Similar Purposes. First, APIs that are designed to solve similar types of tasks can often be used alternatively. Table 5 presents two examples. The first example shows an alternative implementation pair using `pandas.DataFrame.iterrows` and `pandas.DataFrame.itertuples` to iterate over rows of tabular data (#1). The main difference between these two APIs is that `iterrows` returns each row as a Pandas *Series* while `itertuples` returns each row as a *namedtuple*. In the second example, `pandas.DataFrame.pivot_table` and `pandas.crosstab` can both be used to count the frequency of groups (#2). In fact, the terms *pivot table* and *cross tabulation* are often used interchangeably as both denote the method that analyzes summary statistics on groups of categorical data. The main difference between these two APIs is that `pandas.DataFrame.pivot_table` takes a *DataFrame* as input with *mean* being the default aggregation function, while `pandas.crosstab` takes array-like objects as input with *frequency* being the default aggregation function.

Table 5. Examples of alternative implementations, task descriptions, and the corresponding alternative APIs.

Task	Alternative Implementations	Alternative APIs
#1 SO post 32680162: iterate rows of a DataFrame	`> [row.a * 2 for idx, row in df.iterrows()]` `> [row.a * 2 for row in df.itertuples()]`	`pd.DataFrame.iterrows` `pd.DataFrame.itertuples`
#2 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)`	`pd.DataFrame.pivot_table` `pd.crosstab`
#3 SO post 34032455: create an array from a list	`> np.array([0,1,2,3,4,5,6])` `> np.fromiter([0,1,2,3,4,5,6.],dtype=int)`	`numpy.array` `numpy.fromiter`
#4 SO post 52145257: count the occurrence of an element in an array	`> np.sum(rr == 'A')` `> np.count_nonzero(rr == 'A')`	`numpy.sum` `numpy.count_nonzero`
#5 SO post 45648761: convert an array to 0 or 1 based on a given threshold	`> np.where(A > 0.5, 1, 0)` `> (A > 0.5).astype(int)`	`numpy.where` `numpy.ndarray.astype`
#6 SO post 47418496: transform time-like strings to the datetime type	`> df['time'].apply(lambda x:parser.parse(x))` `> pd.to_datetime(df.time)`	`pandas.Series.apply` `pandas.to_datetime`

A special type of such similar APIs uses a *specific* API as an alternative to a *generic* API. Table 5 shows an example of using `numpy.array` or `numpy.fromiter` to create an array (#3). While `numpy.array` is a generic API for array creation, `numpy.fromiter` specializes in creating a one-dimensional array from an iterable object. As another example, Fig. 3 shows that `numpy.einsum` and `numpy.sum` are both used to sum the array elements. While `numpy.sum` is designed specifically for this purpose, `numpy.einsum` is a generic API for common linear algebraic array operations.

We identified 41 recurring pairs that use similar-purpose APIs, 12 of which use generic vs. specific APIs. We also observed that for 31 of these 41 pairs, the relations between alternative APIs can be established directly from the API documentation. In other words, the documentation of one of the APIs in these 31 pairs explicitly refers to the alternative APIs. For example, `pandas.DataFrame.itertuples` is listed in the "See also" section of the documentation for `pandas.DataFrame.iterrows`. This finding suggests that when developers are seeking for alternative APIs to use, the API documentation can be a good place to start. We further discuss this possibility in Section 4.3.

3.3.2 APIs with Different Purposes. The remaining 27 pairs use APIs that are designed essentially for different purposes and can be used alternatively only under certain circumstances. Table 5 shows two examples. First, `numpy.sum` is used to sum array elements whereas `numpy.count_nonzero` is used to count the number of non-zero values in an array. Although these two APIs are designed for different purposes, they yield the same result when the input is a boolean array (#4). As another example, `numpy.where` returns values depending on a given condition while `numpy.ndarray.astype` casts an array to a specified type. Although these two APIs provide different functionalities, both can be used to create a boolean array indicating whether each element of the input array satisfies a given condition (#5).

Table 6. Top 10 input types used in validated alternative implementations.

Type of input	%
`pandas.DataFrame` of strings	27.9
`pandas.DataFrame` of integers	15.2
`pandas.Series` of integers	6.2
`pandas.Series` of strings	4.3
2D `numpy.ndarray` of integers	3.8
1D `numpy.ndarray` of integers	3.0
`pandas.DataFrame` of datetimes	2.1
`pandas.Series` of booleans	1.8
3D `numpy.ndarray` of integers	1.3
monotonic `numpy.ndarray` of integers	1.1

We consider pairs with API sequences of length greater than one as APIs with different purposes. The API pair shown in Fig. 1 is such an example: `numpy.ndarray.sum` and `numpy.sqrt` are alternative to `numpy.linalg.norm` only when they are invoked sequentially. However, when used individually, these two APIs solve tasks different from that of `numpy.linalg.norm`.

A special type of APIs with different purposes uses a concrete API as an alternative to a *higher order* API that takes a function as an argument. For example, Table 5 shows two implementations that convert time-like strings to the *datetime* type (#6). One implementation calls `pandas.Series.apply` to apply a parser function on the input data, whereas its alternative implementation directly calls the `pandas.to_datetime` API that is designed specifically for such a task. We identified 9 pairs that use concrete and higher order APIs as alternatives.

This observation also indicates that if alternative implementations invoked APIs with different purposes, then they can only be used alternatively on specific task inputs. To put this into perspective, we further analyzed the types of inputs used in the random-testing code of validated alternative implementations. Table 6 presents the top ten dominating input types, which are all basic data structures in data manipulation tasks.

For developers of data manipulation programs, identifying APIs that are designed for different purposes but alternative for certain tasks can be challenging. Unlike APIs with similar purposes, alternative APIs with different purposes are typically not linked in the documentation and their usages are also rarely documented. Therefore, developers sometimes need to be flexible or even creative at coming up with alternative solutions for certain tasks, which also requires extensive familiarity with the libraries.

We identified two distinctive characteristics of alternative APIs: generic vs. specific APIs and concrete vs. higher order APIs. We also observed that API documentation could be an informative data source for identifying alternative APIs with similar purposes.

3.4 RQ4: Usefulness

The profiling results reported in Section 3.2 reveal that the performance improvements of alternative implementations could be quite substantial. This motivates us to further explore the possibility of applying our mined knowledge to optimize real-world data manipulation programs. To this end, we conducted an experiment on optimizing *Kaggle* programs using the knowledge of 68 recurring alternative API pairs. Kaggle is an online community for data science practitioners [6]. It is most famous for the *Kaggle competitions* [7], which are hosted by companies or organizations looking

Algorithm 1: AlterApi workflow.

```
   Input  : src ← a Python source file.
   Output : alternatives ← ∅
1  template ← code template for instrumentation ;
2  for stmt in get_candidates(src) do
3      if match_syntax(stmt) then
4          new_src ← generation(stmt, template)
5          result ← run(new_src)
6          alternatives ← alternatives ∩ result
7      end
8  end
```

for the best solutions to their real-life problems. Therefore, the datasets used in the competitions are in general close to real data. For our experiment, we collected the top 5,000 Python programs published by competition participants on the Kaggle platform,[5] which tackle a broad range of practical data science problems in realistic settings.

By inspecting the 68 recurring alternative API pairs, we derived the syntactic patterns for detecting API usages that could be replaced by faster alternatives and rules for the corresponding code transformations. This knowledge is encoded in our tool, AlterApi, which automatically identifies potentially inefficient API usages in an input program, generates alternative implementations, verifies the output equivalence, and reports the runtime speedups after optimization. The workflow of this tool is shown in Algorithm 1 and detailed as below.

- **Candidate Identification**: Given an input Python program, we traverse its abstract syntax tree to get all the `Call` and `Subscript` nodes that represent API usages. The names of these nodes are matched against the target APIs in our recurring patterns to identify candidate statements (`get_candidates` at line 2). We then check the number of parameters and the AST node types of these parameters to further narrow down eligible statements that might match the target API usages (`match_syntax` at line 3). For example, if AlterApi identified a `Call` node from the AST of a Kaggle program that has the name `where` and three parameters, with the first parameter of the `Compare` type (e.g., `A > 0.5`) and the second and third parameters being booleans (e.g., 0 and 1), it selects the corresponding code as a candidate for optimization since it matches the recurring pattern like `np.where(A > 0.5,1,0)` shown in Table 5 (#5).
- **Code Generation**: We define a code template similar to Fig. 7 (but without the randomization parts), which will be used to instrument the input program. Specifically, we populate the template with two types of information. First, certain dynamic information is required to construct the alternative implementations. For example, to generate the `np.einsum` alternative for the `np.sum(a)` type of usage, the dimension of the input variable `a` is required to determine the correct format of the first parameter of `np.einsum` (Fig. 3). To access such type information, we populate the template with an auxiliary code that checks the type of the receiver object, the parameters and certain properties of the parameters (e.g., dimension) based on the specific recurring patterns being targeted. In this manner, we are able to dynamically acquire such information when executing the program. Second, we construct the alternative implementation based on the transformation rules derived from the recurring

[5] https://www.kaggle.com/kernels?sortBy=hotness&group=everyone&pageSize=20&language=Python&kernelType=Script.

Table 7. Top 10 recurring alternative API pairs that contribute the most in the Kaggle experiment.

API	Alternative API
`pandas.DataFrame.ix`	`pandas.DataFrame.loc`
`numpy.hstack`	`numpy.concatenate`
`numpy.hstack`	`numpy.append`
`pandas.DataFrame.ix`	`pandas.DataFrame.iloc`
`numpy.where`	`numpy.nonzero`
`numpy.ones`	`numpy.empty`
`pandas.DataFrame.replace`	`pandas.Series.map`
`numpy.ndarray.dot`	`numpy.tensordot`
`numpy.sum`	`numpy.ndarray.sum`
`pandas.DataFrame.apply`	`numpy.where`

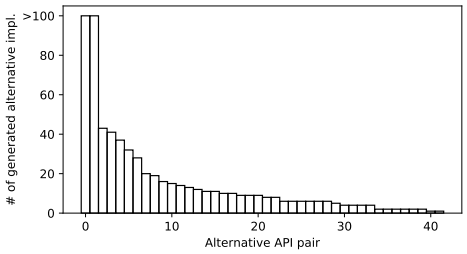


Fig. 12. # of alternative implementations generated from each recurring alternative API pair.

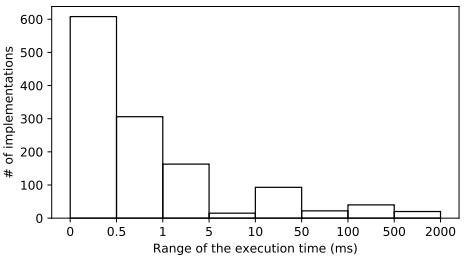


Fig. 13. Distribution of the execution time of the original implementations.

API pairs and populate the template accordingly. The populated template is then inserted back to the AST of the input program to form the instrumented code (`generation` at line 4).

- **Profiling**: Finally, AlterApi runs the instrumented code, which executes the original implementation and its generated alternative in the same context, verifies the output equivalence, and reports the runtime difference (`run` at line 5).

The experimental environment is the same as the one reported in Section 3.2. As Kaggle programs typically involve the training of machine learning/deep learning models, which often requires a huge amount of time, we skipped a candidate program if its execution exceeds 20 minutes for the sake of efficiency.

Our tool generated 1,267 valid alternative implementations for code in 398 programs, which use datasets from 49 Kaggle competitions with sizes ranging from 6.7MB to 14.3GB. The average alternative implementations generated for each program is 3.2 (±6.6). These alternative implementations are generated using the patterns of 42 recurring alternative API pairs, and Fig. 12 shows this distribution. Table 7 further lists the top 10 recurring alternative API pairs that contribute the most in this experiment. For the remaining API pairs, we were unable to identify corresponding usages in the Kaggle dataset. For example, to find the first index of elements that meet certain conditions in an array, `np.argmax` is a faster alternative to `np.where` as suggested in several SO posts. However, the usage pattern of `np.where` to solve such a task requires one condition parameter followed by two indexing operations (e.g., `np.where(arr>5)[0][0]`). We did not identify any instance that matches such specific usage pattern in the dataset.

Table 8. Performance impact of the generated alternative implementations.

	Speedup	Slowdown
≤2x	570	191
2–5x	332	92
5–10x	27	22
10–100x	13	2
>100x	15	3
Total	957	310

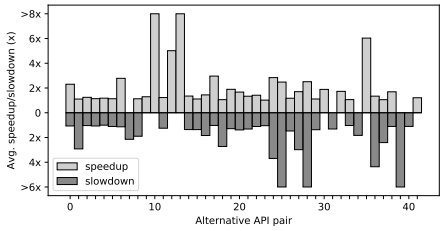


Fig. 14. Avg. speedup/slowdown of alternative impl. generated from recurring alternative API pairs.

Fig. 13 shows the execution time of the original implementations, while the performance impact of the generated alternative implementations is shown in Table 8. For the 1,267 generated alternative implementations, 957 (76%) do achieve performance improvements over the original implementations. Among these successful optimizations, approximately 60% (570) offer slight performance improvements (≤2x), while 3% (28) achieve significant speedup (>10x). As an example of an effective optimization, we generated `np.where(df['time']<3,1,0)` as an alternative to `df.apply(lambda row: 1 if row['time']<3 else 0,axis=1)`, which achieves a ~1300x speedup. On the other hand, 310 (24%) generated alternative implementations turned out to be counterproductive — they slowed down the target task rather than saving execution time. Although 62% (191) of such slowdown is relatively trivial (≤2x), there are a few cases where the performance degradation is significant (>10x). As an example, while several SO posts suggest that `np.fromiter` could be a faster alternative to `np.array` for initializing one-dimensional arrays, alternative implementations generated using this pattern instead incurred dramatic slowdown on certain Kaggle programs.

The experiment results indicate that the alternative API pairs mined from SO can indeed be leveraged to optimize practical data manipulation tasks. However, the optimization outcomes may not always be consistent. Fig. 14 illustrates that implementations generated from most of the recurring alternative API pairs exhibit both performance speedup and slowdown in different optimatization instances. We further discuss this consistency issue in Section 4.1.

Alternative API pairs are useful for identifying optimization opportunities in realistic data manipulation programs and improving runtime performance. However, the outcomes of optimization attempts are not always consistent with those reported from SO.

4 DISCUSSIONS

In this section, we discuss the challenges of optimizing data manipulation programs with alternative APIs (Section 4.1 and 4.2). We also explore the applicability of our approach in other context (Section 4.3 and 4.4).

4.1 Consistency of Optimization Across Data Sizes

Various factors can affect the execution time of a program (e.g., the input data, the library version, the OS and hardware, etc.). In particular, we observed that 7.8% of the SO posts containing consecutive profiling statements have compared the same alternative implementation pairs over different sizes of input data. For example, in Fig. 3, the same alternative implementations `np.einsum('i->',a)` and `np.sum(a)` are profiled for input data of size 1,000 (line 1) and 10,000 (line 7). Given that such

comparisons are not mandatory, this practice indicates that developers consider input data size to be an important factor for evaluating the performance difference between alternative implementations.

To further understand this issue, we executed 68 pairs of alternative implementations, which use the 68 recurring alternative API pairs described in Section 3.3, under different input sizes. Specifically, for each pair, we synthesized five input data with sizes (i.e., # of rows) ranging from 100 to 1,000,000. We quantified the performance difference of alternative implementations under each input condition by calculating the speedup (x) of their execution time. In total, we collected 340 speedup values, five for each of the 68 tasks.

For 45 (66%) tasks, the ordering of the alternative implementations in terms of execution time is consistent over different data sizes. This means that an implementation always remains faster (or slower) compared to its alternative regardless of the input data size. Yet, the extent of performance difference varies dramatically in some cases. Fig. 15a shows two such examples. In the first example, `np.where` and `pd.Series.map` are both used to alter data given a threshold, and `np.where` has a trivial speedup (~1.5x) over `pd.Series.map` when the input data is small. However, the speedup becomes significant (>100x) for input sizes larger than 100,000. In the second example, `np.ndarray.dot` and `np.tensordot` are both used to compute the dot product of matrices. When the input size is small, `np.ndarray.dot` exhibits a non-trivial speedup (~12x) over `np.tensordot`. However, their performance gap narrows as the input size grows. When the input size reaches one million, the two implementations have nearly comparable performance.

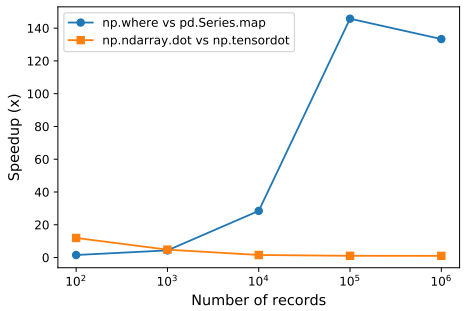
To quantify the consistency of speedups across different input sizes, we compute the *Coefficient of Quartile Variation (cqv)*, which is a measure of relative dispersion [13], of the five speedup values for each pair of alternative implementations. We found that only 28 pairs exhibited a relatively consistent speedup (i.e., $cqv<0.3$) that is not affected much by the input data size. On the other hand, 12 pairs showed moderate variation ($0.3 \leq cqv < 0.7$) and 5 pairs showed severe variation ($cqv \geq 0.7$) across different input sizes.

For 23 (34%) tasks, the same alternative implementations exhibited contradictory runtime behaviors on different input sizes (i.e., both performance improvement and degradation are observed). For example, as shown in Fig. 15b, `pandas.read_csv` is faster than `pandas.read_hdf` for input sizes less than 1,000, whereas `pandas.read_hdf` takes the lead for larger input data; `numpy.column_stack` is slightly slower than `numpy.transpose` for input sizes less than 1,000, but becomes faster than the latter as the input size grows.

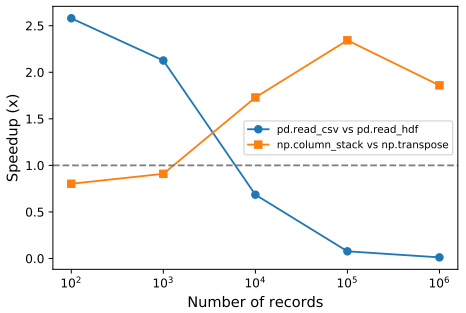
These results indicate that input data size can be a critical factor that affects the outcome of a performance optimization attempt. In particular, an alternative solution that is allegedly much faster than the original implementation may turn out to be alike or even slower when the input data size varies. Yet, for this observation to be transformed to actionable rules and reliably guide practical optimizations, an in-depth code analysis of the target APIs and a more controlled experiment considering confounding environment variables would be required. In the future, we plan to apply these techniques to reduce the risk of unintended optimization consequences.

4.2 Side Effects of Optimizing With Alternative APIs

In addition to performance degradation, we also observed non-performance related concerns when developers evaluate alternative implementations. The first emerging concern is *code readability*. We found in several SO posts where developers consider an alternative implementation to be faster but harder to understand. For example, `numpy.einsum` leverages the Einstein summation convention and is often faster and more flexible for linear algebraic array operations [3]. However, compared to specific APIs with expressive names such as `numpy.sum`, `numpy.einsum` is generally less straightforward, especially for novice developers (Fig. 3). As another example, in SO thread 38550190, a user proposed to use `df.apply(" ".join, axis=1)` for merging dataframe columns with string values.



(a) Cases where an implementation remains faster than its alternative, but the extent of performance improvements varies dramatically.



(b) Cases where an implementation exhibits both performance improvment and degradation over its alternative. Note that a speedup value below 1.0 means a slowdown.

Fig. 15. Examples of alternative implementations whose performance differences are sensitive to the input data size.

Another user proposed a faster alternative `df[0].str.cat(df.ix[:, 1:].T.values, sep=' ')`, but described it as "uglier" compared to the previous solution.

Apart from readability, developers also care about the *deprecation* and *availability* of alternative APIs. In SO post 13842286, `pandas.DataFrame.set_value` was proposed as a faster alternative to `pandas.DataFrame.at` for setting cell values. However, the user also explicitly emphasized that `set_value` has already been deprecated. In SO post 30141358, `pandas.rolling_mean` was proposed as a faster alternative to `numpy.convolve` for computing the running mean. However, the user later updated the post to mention that `rolling_mean` was already removed from the recent version of Pandas. For cases like these, simply choosing the faster alternatives might instead introduce maintainability risks or even cause program crashes.

4.3 Mining Alternatives from API Documentation

In addition to SO, we also explored how API documentation can be leveraged for mining alternative implementations. We considered both the *API docstrings* and the official *user guide* when collecting API documentation data. In particular, a Python docstring is a string literal, surrounded by triple double quotes, that appears as the first statement of an API definition. We traversed the abstract syntax tree of our target libraries to find all API nodes and used Python's introspection capability to obtain the docstring content of each API. A user guide provides high-level descriptions on the design, purposes, and usages of library APIs. We collected the user guide data of the target libraries by recursively searching their `doc/source` folders for files of the reStructuredText format (i.e., files with the *.rst* extension), which is a markup language widely used for technical documentation.

The collected API documentation data has over 208K sentences. Yet, applying our proposed approach on this data revealed only 13 pairs of alternative implementations, five of which were already detected from the SO data. This result indicates that API documentation has a limited occurrence of comparative structures to be exploited. Stack Overflow, on the other hand, encourages exchange of ideas among users, who are therefore more likely to discuss alternative answers and solutions. Hence, compared to API documentation that is inherently descriptive and authored

Table 9. Extracted alternative implementations from other domains.

Library	From consecutive profiling statements	From comparative sentences	# SO threads
R	33	43	5,806
BeautifulSoup	11	7	10,560
Django	1	10	12,921

mainly by a few library developers, our approach that exploits comparative structures is more cost-effective when applied to data like SO that is discussion-based and contributed by the crowd.

Section 3.3 reports that some of the extracted recurring patterns can be found in the "see also" section of API documentation. Inspired by this finding, we further explored whether the "see also" information from API documentation can be leveraged to detect alternative implementations. To this end, we developed a pattern-based algorithm to automatically extract API pairs that are mentioned in the "see also" section of one another's API docstring. This algorithm extracted 3570 API pairs for the three subject libraries, 62 of which were also observed from the SO data.

Despite the large volume of new pairs being discovered, we identified several issues of the extracted data that limit the usability of this approach. First, APIs appeared in "see also" are not necessarily alternatives. For example, `pandas.DataFrame.select_dtypes` is in the "see also" of the `pandas.DataFrame.describe` API's docstring. However, `select_dtypes` is used to select a DataFrame's columns based on their data types, while `describe` simply provides descriptive statistics without offering any selection functionality. Second, API pairs identified from "see also" may not be practical alternatives even though they could produce equivalent outputs in specific cases. For example, `pandas.DataFrame.tail` is in the "see also" of `pandas.DataFrame.head`, with the former returns the last n rows of the input DataFrame and the latter returns the first n rows. Although the two APIs will produce the same output by both setting n to be the size of the input DataFrame, such a usage could be too unorthodox to be observed in practice. Third, the data extracted from "see also" are all one-to-one API pairs, with no pairs involving API call sequences that are instead commonly observed from the SO data. Furthermore, the "see also" documentation contains no direct description on the specific tasks, concrete implementations, and profiling details of the API pairs. The lack of such information makes it difficult to judge the validity, the usefulness, and most important of all, the performance implications of the extracted API pairs.

Taken together, we believe that the SO data, which provides information on the tasks, concrete code, and profiling results of alternative implementations, is more preferable to be exploited for the purpose of performance optimization. API documentation, with its relatively scarce information on API performance comparisons, might not be as straightforward as SO for extracting alternative implementations and quantifying their performance impact.

4.4 Generality of Our Approach

While our approach was evaluated on Python-based data manipulation libraries in this study, we believe that the idea of exploiting comparative structures to reveal programming alternatives is also applicable to other programming languages and applications in other domains. To demonstrate this, we first applied our approach to SO threads tagged with *R*, which is another popular programming language widely used for data manipulations [14]. We detected alternative implementations on R from both comparative natural-language sentences and consecutive profiling statements (Table 9). Fig. 16a shows a comparative sentence, from which our approach extracts `bind_rows()` as a faster alternative to `rbind()`. Fig. 16b shows two code snippets (lines 1 and 4) that are being consecutively

For a large number of repetions `bind_rows()` is also much faster than `rbind()`

(a) A comparative sentence that discusses R solutions (SO post 45638814).

```
1 > system.time({ df <- do.call(''rbind'', listOfDataFrames) })
2    user  system elapsed
3    0.25    0.00    0.25
4 > system.time({ df2 <- ldply(listOfDataFrames, data.frame) })
5    user  system elapsed
6    0.30    0.00    0.29
7 > identical(df, df2)
8 [1] TRUE
```

(b) Except from SO post 2851434. The two code snippets that are being consecutively profiled by R's built-in function `system.time()` are alternative implementations.

```
1 m = matrix(rnorm(1200000), ncol=600)
2 v = rep(c(1.5, 3.5, 4.5, 5.5, 6.5, 7.5), length = ncol(m))
3 library(microbenchmark)
4
5 microbenchmark(t(t(m)*v),
6   m %*% diag(v),
7   m * rep(v, rep.int(nrow(m),length(v))),
8   m * rep(v, rep(nrow(m),length(v))),
9   m * rep(v, each = nrow(m)))
```

(c) Except from SO post 32364355. The five code snippets that are being profiled by R's *microbenchmark* package are alternative implementations.

Fig. 16. Comparative structures reveal alternative implementations in R.

profiled by `system.time()`, which is R's built-in function for measuring execution time. Line 7 validates that these two implementations indeed produce identical results.

Next, we applied our approach to SO threads tagged with *Django* and *BeautifulSoup*, respectively. Django is a Python framework for web development [2] while Beautiful Soup is a Python library for processing HTML and XML files [1] — both of which focus on applications that are quite different from data manipulation. Our approach also detects valid alternative implementations on these datasets of other domains (Table 9). For example, we detected alternatives for counting the objects in Django's `QuerySet` from consecutive profiling statements in SO post 58661001. From SO post 58265903, we extracted a comparative sentence "*Use Select() which is faster than findAll()*" that reveals a valid pair of alternatives in Beautiful Soup.

The above observations indicate that our approach of exploiting comparative structures to detect alternative implementations is also applicable to other programming paradigms and applications domains. In particular, due to the universality of natural language structures, the idea of extracting alternatives from comparative sentences should be applicable to general textual contents regardless of which programming language or application domain is being targeted. As for leveraging consecutive profiling statements, our approach can be easily adapted to other programming languages by replacing `timeit` to the specific profiling syntax of the subject programming language. For example, `system.time()` (Fig. 16b) and `microbenchmark` are both commonly used for profiling code snippets in R. Fig. 16c shows a performance comparison between five code snippets using microbenchmark (lines 5–9). We executed these five code snippets on 10 random inputs of proper types and validated that they are indeed alternatives for producing equivalent outputs.

Nonetheless, our proposed approach is still the most effective when applied to data manipulation tasks, possibly because of the inherent flexibility of such tasks. Even for simple data manipulation tasks like summing all values in a table, there could be various alternative implementations such as summing by columns first then by rows or the other way around (see Fig. 4 for four different implementations of this task). In other words, a data manipulation task is more likely to be broken down into different combinations of smaller and common computation operators. Other application domains, however, may not have such degree of redundancy as their tasks (e.g., starting servers in web development) are often atomic.

5 THREATS TO VALIDITY

Our experiments were conducted on NumPy, Pandas, and SciPy. The results may not generalize to other libraries or programming languages. However, as discussed in Section 4.4, our approach should also be applicable to other subjects and programming paradigms with only minor implementation adjustments. We plan to conduct experiments on a more varied set of data manipulation libraries in the future.

Our approach can only detect alternative implementations from consecutive profiling statements and comparative sentences with predefined keywords, explicit code mentions and specific dependency patterns. Our approach is also limited to comparative structures, whereas other manifestations of alternative implementations, such as plain descriptions with no comparative semantics (e.g., "`A` *and* `B` *are both recommended for this task*"), cannot be detected. In addition, as our experiments were conducted exclusively on the SO data, we cannot detect alternative implementations that are discussed on other websites or through other channels of communication. However, the aim of this study is not uncovering all possible alternative implementations for data manipulation libraries, but rather exploring the feasibility of using comparative structures to effectively reveal programming alternatives. Nevertheless, one way to mitigate this threat is to extend our approach from code-block level and sentence level to SO thread level, which remains as future work.

The random testing approach we adopted to validate input/output equivalence only explored limited number of combinations (i.e., 420) of common container types, element types, and dimensions. Consequently, as described in Section 2.3, complex or task-specific inputs could not be automatically generated and had to be manually constructed. We envision that automated test generation techniques, such as genetic algorithms that evolve random inputs with mutation operators until certain criteria are fulfilled [36], could also be adapted here to guide the automatic generation of sophisticated and proper input data.

The characteristics of alternative API pairs are derived by manual inspection, which could be subjective and error-prone. We mitigate this threat by having two authors inspect the data independently and reconcile the differences between their labelings (Section 2.4). Nonetheless, there could be other ways to characterize alternative API pairs.

In our experiments on optimizing Kaggle programs, we measured the performance impact of alternative implementations at the statement level, whereas the program-level performance impact may differ. Also, the profiling results reported in Sections 3.4 and 4.1 may not generalize to other execution environments. In addition, AlterApi is built based on the 68 recurring API pairs, which limits it generality. However, the primary focus of this paper is to automatically and effectively discover alternative implementation pairs, and our proposed approach is to leverage comparative structures from SO discussions for this purpose. AlterApi is easily extensible once alternative implementation pairs are discovered.

6 RELATED WORK

In this section, we discuss representative pieces of related work from recent years and compare our work with them.

6.1 API Misuse

Prior research has studied *API misuse* issues, which refer to the usages that violate an API's contract or usage constraints, thus lead to bugs, program crashes, and vulnerabilities. Campos et al. proposed a search-based approach to finding fixes for API-usage-related bugs of Java and JavaScript from Stack Overflow [22]. Zhang et al. used Java API usage patterns mined from GitHub to detect potential API misuse issues in code snippets on Stack Overflow [55]. Amann et al. created an API

misuse dataset [20] and systematically evaluated static API-misuse detectors on this dataset [21]. The API misuse patterns studied in this line of work are typically combinations of different violation types on various API usage elements, such as redundant iterations, missing preconditions, and incorrect order of API calls [20, 55]. Another type of API misuse that has drawn much attention is the violation of domain-specific constraints, such as cryptography protocols [32]. Our study, however, does not aim to address the misuse of individual APIs. The alternative implementations we extracted from Stack Overflow are all viable solutions to the target tasks, only that some of them are more efficient than the others.

6.2 Performance Implications of API Usages

Studies have shown that different API usages could affect program performance. Kawrykow and Robillard studied Java applications and observed several cases where a sequence of API calls can be replaced by a more efficient one [31]. Oliveira et al. proposed an approach that uses energy profiles and static analysis to recommend energy-efficient alternatives for Java collection implementations [41]. Linares-Vásquez et al. found energy bugs caused by suboptimal API choices in Android apps [33]. Liu et al. identified performance issues in Android apps due to the misuse of the list scrolling API [34] and wakelock APIs [35]. Selakovic and Pradel studied JavaScript programs and found inefficient API usage to be the most common root cause of performance issues [47]. Yang et al. reported that half of the performance issues in Rails applications can be improved by changing how the Rails APIs are used [51]. To the best of our knowledge, we are the first to systematically study how API selections affect the runtime performance of data manipulation programs.

6.3 Software Similarity and Redundancy

Software can be similar in many ways. Huang et al. mined Stack Overflow to discover similar technologies and the aspects that are being compared [29]. The scope of their target technologies ranges from libraries and tools to concepts like algorithms and protocols. Chen et al. mined Stack Overflow tags to identify analogical libraries for different programming languages (e.g., Java's OpenNLP vs. Python's NLTK) [25] and different mobile platforms (e.g., iOS's AFNetworking vs. Android's Volley) [26]. Nguyen et al. used API embeddings created by word2vec to identify APIs with similar usage contexts and relations, which were used to migrate equivalent API usages from Java to C# [40]. Even within the same software system, redundant code elements are prevalent [23]. Various techniques such as data flow analysis [48] and random testing [30] have been proposed to detect functional clones. Different from these studies, our work exploits consecutive profiling statements and comparative sentences from Stack Overflow posts to unveil the functional similarity between different API usages.

7 CONCLUSIONS

We have presented the first empirical study of how alternative data manipulation APIs affect the runtime performance of programs. A key appeal of our approach is that it 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 5,080 alternative implementations obtained in this manner appear to have substantial runtime difference according to our profiling experiments. This further indicates that a technique for detecting faster API alternatives is desirable.

We distilled 68 recurring alternative API pairs from the extraction results. By studying their characteristics, we found that specific and concrete APIs are often used as alternatives to generic and higher order APIs, respectively. We leveraged this knowledge of alternative API pairs to automatically generate alternative implementations for realistic data manipulation programs, and

observed nontrivial speedups in 76% of the optimization attempts. We hope that our study can help developers to use data manipulation APIs more efficiently and researchers to develop techniques that effectively optimize data manipulation programs. A replication package of this work is made available at: https://sites.google.com/view/alterapi-artifacts/.

ACKNOWLEDGMENT

We sincerely thank the editor and anonymous reviewers for their constructive suggestions and insightful comments on improving this manuscript.

REFERENCES

[1] [n.d.]. Beautiful Soup Documentation. https://www.crummy.com/software/BeautifulSoup/bs4/doc/.
[2] [n.d.]. Django: The web framework for perfectionists with deadlines. https://www.djangoproject.com/.
[3] [n.d.]. Einstein summation convention. https://en.wikipedia.org/wiki/Einstein_notation.
[4] [n.d.]. How to create a Minimal, Reproducible Example. https://stackoverflow.com/help/minimal-reproducible-example
[5] [n.d.]. IPython built-in magic commands. https://ipython.readthedocs.io/en/stable/interactive/magics.html.
[6] [n.d.]. Kaggle. https://www.kaggle.com/.
[7] [n.d.]. Kaggle Competitions. https://www.kaggle.com/competitions/.
[8] [n.d.]. NumPy. http://www.numpy.org/.
[9] [n.d.]. pandas. https://pandas.pydata.org/.
[10] [n.d.]. The Python ast module. https://docs.python.org/3/library/ast.html.
[11] [n.d.]. Python Qualified Name. https://docs.python.org/3/glossary.html#term-qualified-name.
[12] [n.d.]. The Python Standard Library: Debugging and Profiling. https://docs.python.org/3/library/debug.html.
[13] [n.d.]. Quartile coefficient of dispersion. https://en.wikipedia.org/wiki/Quartile_coefficient_of_dispersion.
[14] [n.d.]. The R Project for Statistical Computing. https://www.r-project.org/.
[15] [n.d.]. The SciPy library. https://www.scipy.org/scipylib/index.html.
[16] [n.d.]. SciPy.org. https://www.scipy.org/.
[17] [n.d.]. Semgrex. https://nlp.stanford.edu/nlp/javadoc/javanlp/edu/stanford/nlp/semgraph/semgrex/SemgrexPattern.html.
[18] [n.d.]. Stack Exchange archive. https://archive.org/details/stackexchange.
[19] [n.d.]. timeit: Measure execution time of small code snippets. https://docs.python.org/3.6/library/timeit.html.
[20] Sven Amann, Sarah Nadi, Hoan Anh Nguyen, Tien N. Nguyen, and Mira Mezini. 2016. MUBench: A Benchmark for API-Misuse Detectors. In *Proceedings of the 13th International Conference on Mining Software Repositories (MSR 2016)*. https://doi.org/10.1145/2901739.2903506
[21] Sven Amann, Hoan Anh Nguyen, Sarah Nadi, Tien N. Nguyen, and Mira Mezini. 2018. A Systematic Evaluation of Static API-Misuse Detectors. *IEEE Transactions on Software Engineering* (2018). https://doi.org/10.1109/TSE.2018.2827384
[22] Eduardo C. Campos, Martin Monperrus, and Marcelo A. Maia. 2016. Searching Stack Overflow for API-usage-related Bug Fixes Using Snippet-based Queries. In *Proceedings of the 26th Annual International Conference on Computer Science and Software Engineering* (Toronto, Ontario, Canada) *(CASCON '16)*. IBM Corp., Riverton, NJ, USA, 232–242.
[23] A. Carzaniga, A. Mattavelli, and M. Pezzè. 2015. Measuring Software Redundancy. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. 156–166. https://doi.org/10.1109/ICSE.2015.37
[24] Yanto Chandra and Liang Shang. 2019. *Qualitative research using R: A systematic approach*. Springer.
[25] Chunyang Chen and Zhenchang Xing. 2016. SimilarTech: Automatically Recommend Analogical Libraries Across Different Programming Languages. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (Singapore, Singapore) *(ASE 2016)*. ACM, New York, NY, USA, 834–839. https://doi.org/10.1145/2970276.2970290
[26] Chunyang Chen, Zhenchang Xing, and Yang Liu. 2018. What's Spain's Paris? Mining analogical libraries from Q&A discussions. *Empirical Software Engineering* (2018), 1–40.
[27] Samir Gupta, ASM Ashique Mahmood, Karen Ross, Cathy Wu, and K Vijay-Shanker. 2017. Identifying comparative structures in biomedical text. In *BioNLP 2017*. 206–215.
[28] Homa B Hashemi and Rebecca Hwa. 2016. An evaluation of parser robustness for ungrammatical sentences. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*. 1765–1774.
[29] Yi Huang, Chunyang Chen, Zhenchang Xing, Tian Lin, and Yang Liu. 2018. Tell Them Apart: Distilling Technology Differences from Crowd-scale Comparison Discussions. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE 2018)*. ACM, New York, NY, USA, 214–224. https://doi.org/10.1145/3238147.3238208

[30] Lingxiao Jiang and Zhendong Su. 2009. Automatic Mining of Functionally Equivalent Code Fragments via Random Testing. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis* (Chicago, IL, USA) *(ISSTA '09)*. ACM, New York, NY, USA, 81–92. https://doi.org/10.1145/1572272.1572283

[31] D. Kawrykow and M. P. Robillard. 2009. Detecting inefficient API usage. In *2009 31st International Conference on Software Engineering - Companion Volume*. 183–186. https://doi.org/10.1109/ICSE-COMPANION.2009.5070977

[32] Stefan Krüger, Johannes Späth, Karim Ali, Eric Bodden, and Mira Mezini. 2018. CrySL: An Extensible Approach to Validating the Correct Usage of Cryptographic APIs. In *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[33] Mario Linares-Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Rocco Oliveto, Massimiliano Di Penta, and Denys Poshyvanyk. 2014. Mining Energy-greedy API Usage Patterns in Android Apps: An Empirical Study. In *Proceedings of the 11th Working Conference on Mining Software Repositories* (Hyderabad, India) *(MSR 2014)*. ACM, New York, NY, USA, 2–11. https://doi.org/10.1145/2597073.2597085

[34] Yepang Liu, Chang Xu, and Shing-Chi Cheung. 2014. Characterizing and Detecting Performance Bugs for Smartphone Applications. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. ACM, New York, NY, USA, 1013–1024. https://doi.org/10.1145/2568225.2568229

[35] Y. Liu, C. Xu, S. C. Cheung, and J. Lü. 2014. GreenDroid: Automated Diagnosis of Energy Inefficiency for Smartphone Applications. *IEEE Transactions on Software Engineering* 40, 9 (Sep. 2014), 911–940. https://doi.org/10.1109/TSE.2014.2323982

[36] Stephan Lukasczyk, Florian Kroiß, and Gordon Fraser. 2020. Automated Unit Test Generation for Python. In *International Symposium on Search Based Software Engineering*. Springer, 9–24.

[37] Christopher D. Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven J. Bethard, and David McClosky. 2014. The Stanford CoreNLP Natural Language Processing Toolkit. In *Association for Computational Linguistics (ACL) System Demonstrations*. 55–60.

[38] Mary McHugh. 2012. Interrater reliability: The kappa statistic. *Biochemia medica : časopis Hrvatskoga društva medicinskih biokemičara / HDMB* 22 (10 2012), 276–82. https://doi.org/10.11613/BM.2012.031

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

[40] Trong Duc Nguyen, Anh Tuan Nguyen, Hung Dang Phan, and Tien N. Nguyen. 2017. Exploring API Embedding for API Usages and Applications. In *Proceedings of the 39th International Conference on Software Engineering* (Buenos Aires, Argentina) *(ICSE '17)*. IEEE Press, Piscataway, NJ, USA, 438–449. https://doi.org/10.1109/ICSE.2017.47

[41] Wellington Oliveira, Renato Oliveira, Fernando Castor, Benito Fernandes, and Gustavo Pinto. 2019. Recommending Energy-efficient Java Collections. In *Proceedings of the 16th International Conference on Mining Software Repositories* (Montreal, Quebec, Canada) *(MSR '19)*. IEEE Press, Piscataway, NJ, USA, 160–170. https://doi.org/10.1109/MSR.2019.00033

[42] C. E. Otero and A. Peter. 2015. Research Directions for Engineering Big Data Analytics Software. *IEEE Intelligent Systems* 30, 1 (Jan 2015), 13–19. https://doi.org/10.1109/MIS.2014.76

[43] Marta Recasens, Marie-Catherine de Marneffe, and Christopher Potts. 2013. The Life and Death of Discourse Entities: Identifying Singleton Mentions. In *North American Association for Computational Linguistics (NAACL)*.

[44] Peter C. Rigby and Martin P. Robillard. 2013. Discovering Essential Code Elements in Informal Documentation. In *Proceedings of the 2013 International Conference on Software Engineering* (San Francisco, CA, USA) *(ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 832–841.

[45] David Robinson. 2017. Why is Python Growing So Quickly? https://stackoverflow.blog/2017/09/14/python-growing-quickly/.

[46] Jacob T Schwartz. 1980. Fast probabilistic algorithms for verification of polynomial identities. *Journal of the ACM (JACM)* 27, 4 (1980), 701–717.

[47] M. Selakovic and M. Pradel. 2016. Performance Issues and Optimizations in JavaScript: An Empirical Study. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. 61–72. https://doi.org/10.1145/2884781.2884829

[48] Fang-Hsiang Su, J. Bell, G. Kaiser, and S. Sethumadhavan. 2016. Identifying functionally similar code in complex codebases. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*. 1–10. https://doi.org/10.1109/ICPC.2016.7503720

[49] Yida Tao, Shan Tang, Yepang Liu, Zhiwu Xu, and Shengchao Qin. 2019. How Do API Selections Affect the Runtime Performance of Data Analytics Tasks?. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering* (San Diego, CA, USA).

[50] Christoph Treude and Martin P. Robillard. 2016. Augmenting API Documentation with Insights from Stack Overflow. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. ACM, New York, NY, USA, 392–403. https://doi.org/10.1145/2884781.2884800

[51] Junwen Yang, Pranav Subramaniam, Shan Lu, Cong Yan, and Alvin Cheung. 2018. 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 (Gothenburg, Sweden) *(ICSE '18)*. ACM, New York, NY, USA, 800–810. https://doi.org/10.1145/3180155.3180194

[52] D. Ye, Z. Xing, C. Y. Foo, J. Li, and N. Kapre. 2016. Learning to Extract API Mentions from Informal Natural Language Discussions. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 389–399. https://doi.org/10.1109/ICSME.2016.11

[53] Deheng Ye, Zhenchang Xing, Jing Li, and Nachiket Kapre. 2016. Software-specific Part-of-speech Tagging: An Experimental Study on Stack Overflow. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing* (Pisa, Italy) *(SAC '16)*. ACM, New York, NY, USA, 1378–1385. https://doi.org/10.1145/2851613.2851772

[54] Jasmine Zakir, Tom Seymour, and Kristi Berg. 2015. BIG DATA ANALYTICS. *Issues in Information Systems* 16, 2 (2015).

[55] Tianyi Zhang, Ganesha Upadhyaya, Anastasia Reinhardt, Hridesh Rajan, and Miryung Kim. 2018. Are Code Examples on an Online Q&A Forum Reliable?: A Study of API Misuse on Stack Overflow. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. ACM, New York, NY, USA, 886–896. https://doi.org/10.1145/3180155.3180260

[56] Hao Zhong, Tao Xie, Lu Zhang, Jian Pei, and Hong Mei. 2009. MAPO: Mining and Recommending API Usage Patterns. In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming* (Italy) *(Genoa)*. Springer-Verlag, Berlin, Heidelberg, 318–343. https://doi.org/10.1007/978-3-642-03013-0_15