Exploring Hashing Algorithms with Large Datasets (December 2023)

Maxwell A. Hunter and Christian Thede

***Abstract*—This report focuses on the theoretical, and some practical, applications of hashing large datasets (1,000,000+ lines) in order to better understand different hashing algorithms and their efficiencies. Using a Twitter dataset comprised of tweets containing the keyword, "Bitcoin", we compared and contrasted the SHA-256, SHA-1, and MD5 hashing algorithms. As we hashed through the Twitter data, we found a negligible difference between SHA-1, MD5, and SHA-256, looking at time and resource usage.**

I. Introduction

As we finished the multiple sections of course content in our class, we also concluded our project on Exploring Hashing Algorithms with Large Datasets. Through many different stages and even more failed attempts, our hashing code proved to work and even more so, was able to be measure not only with time, but also with memory usage. In the following passages, we will be talking about what a hashing algorithm is, the dataset and benchmark machine, different libraries used in our program, function definitions and the code within them, and the different iterations of each hashing algorithm tested throughout.

As the reader you may know, or may not know, that hashing isn't necessarily used to encrypt or encode a set of data with the intentions to get the data back. It is almost impossible, if not completely impossible. According to the Oxford Dictionary, hashing means to assign a numeric or alphanumeric string to (a piece of data) by applying a function whose output values are all the same number of bits in length. We will elaborate on this further in our report, but wanted to make the distinction that our research and program is limited in its practical application and that its purpose is rather to bring us to a better and deeper understanding on how efficient different hashing algorithms are. Do not be mistaken that hashing a large dataset, such as presented here today, is futile, but with further testing and research we would be able to show a more practical side of our program. For all intents and purposes, our program shows the performance benchmarking side of hashing. We do this by measuring the time it takes to hash a particular set of data and analyze the time and memory usage each takes up. By doing this we can study the efficiency and scalability of each implementation.

II. The Purpose of Hashing

There are many different purposes for hashing. This includes but is not limited to data integrity, data retrieval, password storage, digital signatures, cryptographic applications, efficient data retrieval, distributed systems, file and data deduplication, digital forensics, and performance benchmarking. All of these algorithms are created and designed to be fast and efficient. According to techtarget.com, hashing is the process of transforming any given key or a string of characters into another value. This is usually represented by a shorter, fixed-length value or key that represents and makes it easier to find or employ the original. For example, in password storage, a company does not actually hold the user's real password in storage. It holds the hashed version of it. This prevents hackers from being able to steal a user's password, let's say as a man-in-the-middle-attack, and using the password to gain access. Why? This is because the password they would be stealing is hashed, and if you used the hashed password as the password, it will itself be hashed into something different, thus access will be denied.

III. The Dataset and Machine

A. Too Small

To first understand what we are doing here today with hashing, we need to know what data we are going to be working with. For our program, we needed a dataset large enough to be able to quantifiably measure the TTH (time to hash). We noticed in the beginning stages of our research that a dataset that was too small was hashed in an infinitesimal amount of time. Our benchmark computer contained an Intel Core i9 13th generation processor that of which handled our laughable original dataset with extreme ease. When we attempted to measure the time it took to hash, we were met with a time that was not able to be practically managed or even measured consistently across the board. Since we faced this obstacle, it forced us to go looking for a dataset large enough that even the Core i9 would struggle.

B. *Finding a Larger Dataset*

As we were searching for a larger dataset, we were reminded of the enormous Twitter (now known as X) dataset we used last year in Python 101. This Twitter data contained a whopping 1,000,000+ lines of data. Within this data, was a plethora of tweets about everything related to bitcoin. For the purpose of our program, we were not particularly interested in the content of the dataset, but rather how we could manipulate the data with different hashing algorithms. Since we had then found a data set large enough for our computer, we were able to begin experimentation with it. But first, let us talk a little more in depth about our benchmark machine.

C. *The Benchmark Machine*

For our benchmark machine, we are using an Asus Vivobook laptop. The processor, as stated above, is an Intel Core i9-13900h. This processor is in the top of its class for laptop grade CPUs. Thus, requiring us to have to have a large amount of data so we could measure effectively. Paired with the 14 core CPU, the laptop features 16 GB of DDR5 memory running at 4800 megahertz. Although the PC does not need a discrete graphics card, the computer does have an RTX 3050 6GB to accelerate its performance. To maximize consistency, we kept the computer plugged into AC power to ensure it didn't go into a power saving mode that reduced performance. We also made it a point to ensure that the memory used outside of the Python program was kept to a minimum.

ASUS - 15.6" OLED Laptop - Intel Core i9-13900H - NVIDIA RTX3050 6GB with 16GB Memory – 1TB SSD - Black

Model: Q540VJ-I93050 SKU: 6534578

IV. Libraries

Now that we have established our dataset and benchmarking machine, it is time to focus on our actual program. We have implemented a total of four different libraries that have assisted us in our endeavors.

```
import pandas as pd
import hashlib
import time
import psutil
```

A. *Pandas*

First up, we have the Pandas library. We import it as pd to save the hassle of typing pandas every time. According to w3schools.com, Pandas is a Python library used for working with data sets. It has functions for analyzing, cleaning, exploring, and manipulating data. The name "Pandas" has a reference to both "Panel Data", and "Python Data Analysis" and was created by Wes McKinney in 2008. We can either use a series or data frame when operating with pandas. For our program, we need to utilize the data frame ability of this in order to take advantage of the two-dimensional table with labeled axes. Our Twitter data is a CSV (comma separated value) file. This means that we have to use the data frame in order to properly manipulate and parse through our data.

```
selected_columns = ["id", "possibly_sensitive", "source",
                    "text", "user_screen_name"]
```

In the above graphic, we have chosen only five columns to hash out of the thirty available. We chose this because these columns contained the most sensitive content. By using the built in functions of pandas, we were able to retrieve only the columns that we wanted from the Twitter data. Once these columns are separated, they are entered into the df (DataFrame) and can be gone through line by line. At this point we can hash this data at the line level and output it into the output file.

B. *Hashlib*

Next, we have the hashlib library. According to ioflood.com, Python's hashlib module provides a variety of hash functions, including md5, sha1, and sha256. All of which were tested in our program. These functions are used to create hash objects, which are then used to generate hashes of data. Another explanation from python.org explains that the hashlib module provides a helper function for efficient hashing of a file or file-like object.

```
def hash_data(data):
    h = hashlib.new("md5")
    h.update(data)
    return h.digest()
```

In the above graphic, the data we pass into the function is hashed using the md5 algorithm and then returned to the digest. This is where we changed each algorithm from md5, to sha1, and finally to sha256. To sum this library up, it essentially takes the burden of implementing our own hash algorithm and allows us to freely change which one we want to use.

C. *Time*

Now, we have the time library. This is a simple and short library that allows us to start a clock before we run a certain part of our code.

```
# Calculate and print benchmarking information
print("\nBenchmarking Information:")
print(f"Total time: {end_time - start_time} seconds")
print(f"Time for reading CSV: {read_end_time - read_start_time} seconds")
print(f"Time for hashing and writing: {hash_end_time - hash_start_time} seconds")
```

We start the time clock before we read the CSV file and then stop it after the file is read. We also start another time clock before the hashing and writing and then stop it after the hashing and writing is completed. This allows us to have a quantifiable measurement to see which hashing algorithm is more efficient. These results will be discussed in a further section.

D. Psutil

Finally, we have the Psutil library. According to readthedocs.io, psutil (python system and process utilities) is a cross-platform library for retrieving information on running processes and system utilization (CPU, memory, disks, network, sensors) in Python. It is useful mainly for system monitoring, profiling, limiting process resources and the management of running processes. In our program we use the psutil library to monitor both memory and CPU usage. Below is an example of the memory usage.

```
Memory Usage:
Memory used: 446.53 MB
Memory peak: 484.15 MB

Hashing and writing complete!
```

The code lets us compare and contrast the differences between the md5, sha1, and sha256 programs. In the above example, we see that the md5 model used an average of 446.53 MB of data while peaking at 484.15 MB. This does not include the memory usage from other programs running on the benchmarking computer but instead only records the resources used by the python program. Another way we can use the psutil library is by measuring the CPU usage while running the program.

```
CPU Usage: 3.4%

Benchmarking Information:
Total time: 302.9454784393
Time for reading CSV: 4.31
Time for hashing and writi

Memory Usage:
Memory used: 438.43 MB
Memory peak: 484.07 MB
```

Above is an example of the CPU usage after the running of the md5 hashing algorithm. Using the psutil library has helped us gain a better understanding of how efficient each hashing algorithm is.

V. Functions and Their Definitions

Now that we have coved all of the libraries used for our program, it is time to dive into the meat of our code. For our first function, we are using, “def_hash” to be our call name.

```
def hash_data(data):
    h = hashlib.new("md5")
    h.update(data)
    return h.digest()
```

As briefly described previously in this report, the hash_data function’s entire purpose is to take the data that is passed into the parameter and perform a hash on said data.

Next, we have our main and big definition that does most of the heaving lifting and parsing through data.

```
def hash_and_write_rows(file_path,
                        selected_columns,
                        output_file_path):
```

It is called, hash_and_write_rows. What this function does is it takes the parameters of file_path, selected_columns, and output_file_path and performs different operations of each. To explain further in depth, we will be providing a deeper explanation of each. So, in the beginning of this function, we are utilizing the time library. To start, we begin taking down time to measure the entire length of how long the program is running.

```
# Record the start time
start_time = time.time()
```

Next, we need to open our output file as file so that we can write our newly hashed code to it, along with the header at the top.

```
with open(output_file_path, 'wb') as file:
    # Write header to the output file
    file.write(','.join(selected_columns).encode('utf-8') + b'\n')
```

Once this has been completed, it is necessary for us to start another time clock. This time clock is going to be the start time for the CSV part of our time keeping.

```
# Record the end time for reading the CSV
read_end_time = time.time()
```

Next, we need to write all of the Twitter data into our Pandas DataFrame system. As covered before, we were able to parse through most of the Twitter data to only input the certain columns of data that we wanted. So, in the below code, we can see this in action and how the selected columns are represented as file_path, and conversly moved into the DataFrame.

```
# Read only the specified columns
# from the CSV file into a DataFrame
df = pd.read_csv(file_path,
                 usecols=selected_columns)
```

Now the we have our data in the DataFrame, we can't forget to stop our read CSV time.

```
# Record the end time for reading the CSV
read_end_time = time.time()
```

After we stop the read CSV time, it is now time for us to start the hashing time, and then explain into further detail how our program takes the data and sends it to our hashing function.

```
# Record the start time for
# hashing and writing
hash_start_time = time.time()
```

In the next code, the **for index, row in df.iterrows():** part, iterates over the DataFrame df row by row. The index represents the index of the present row, and the "row" part is from the Pandas Series funcction that represents the data in that row. As for the "**row.to_csv(index=False)**", part of the code, this converts the current row to a comma-separted-value (CSV) string without including the index. This is then encoded using UTF-8.

According to twilio.com, UTF-8 is a "variable-width" encoding standard. This means that it encodes each code point with a different number of bytes, between one and four. As a space-saving measure, commonly used code points are represented with fewer bytes than infrequently appearing code points.

Next, we use our **hash_data():** function. We call this using the encoded row data as the argument. This sends our data through the fuction as we have already talked about, and returns it to the digest. Second to last, we have the **file.write(hashed_result** + b'\n'**)** part. This writes the hashed result to our output file.

Lastly, we print our result to the console using, **print(f"Hashed result for row {index}: {hashed_result.hex()}")**. This prints the index of the current row along with the hexadecimal representation of the hashed result.

```
# Iterate through each row, hash it, and write to the output file
for index, row in df.iterrows():
    hashed_result = hash_data(row.to_csv(index=False).encode('utf-8'))
    file.write(hashed_result + b'\n')
    print(f"Hashed result for row {index}: {hashed_result.hex()}")
```

After this, we can end the hash time clock, and then end the whole program time clock.

VI. The Algorithms

A. MD5

The MD5 hash function produces a 128-bit (16-byte) hash value. It is similar to SHA-1 in that they are both considered insecure for cryptographic applications due to certain vulnerabilities such as collision attacks. A collision attack to when two different inputs produce the same hash value. MD5 belongs to the Merkle-Damgård construction, which is a design used by a plethora of hash functions. The downfall, like we mentioned, of this construction, is that it has poor collision resistance. MD5 used to be used years ago for applications such as digital signatures, checksums, and even data integrity verification.

B. SHA-1

The SHA-1 hash function produces a 160-bit(20-byte) hash value. Again, it is similar to MD5 in that they both are very poor at ensuring that two inputs won't yield the same output. This can be very dangerous in certain cases because this increases the chances of a hacker guessing a correct password if SHA-1 is being used as the hashing algorithm for such. If the password hash is stored in the database, and the hacker is brute force guessing the password, his or her chances go up if more than one guess can result in the same password hash. SHA-1 also belongs to the Merkle-Damgård construction. SHA-1 was also used years ago for applications mentioned above in MD5.

C. SHA-256

SHA-256 produces a 256-bit (32-byte) hash value. It is presently considered to be secure and is widespread in its use for cryptographic applications. As for its collision resistance, SHA-256 is designed to be collision resistant, this makes it very hard to find different inputs that will result in the same hash value. Unlike the previous two algorithms, this algorithm is part of the SHA-2 family, which uses a different structure. This structure is way more resistant to known attacks. It is currently recommended for secure applications like password storage, digital signatures and cryptographic protocols.

D. Speed

Since we are here today to discuss the efficiency of these applications, let us differentiate which algorithm should be faster. Generally, MD5 is faster than the latter two because it involves fewer rounds of computation. That being that it is 128 bit. By this logic, the next fastest should be SHA-1 because it is 160 bit. And finally, SHA-256 should be the third fastest because it entails the largest bit size being 256. Our job in the next section is to test whether or not this is true.

VII. Final Testing

A. Let us take a look at the test results for MD5.

```
CPU Usage: 3.4%

Benchmarking Information:
Total time: 302.94547843933105 seconds
Time for reading CSV: 4.311934947967529 seconds
Time for hashing and writing: 298.62654852867126 seconds

Memory Usage:
Memory used: 438.43 MB
Memory peak: 484.07 MB

Hashing and writing complete!
```

We can see that the MD5 algorithm has yielded a time of 302.95 seconds, 298.63 seconds of which came from the time for hashing and writing and 4.31 seconds of which came from the time for reading the CSV file. As for the CPU usage, we can see that we used a total of 3.4%. For the average memory usage, we see 438.43 MB, and a peak of 484.07 MB.

B. Let us take a look at the test results for SHA-1

```
Benchmarking Information:
Total time: 300.88542675971985 seconds
Time for reading CSV: 4.380641937255859 seconds
Time for hashing and writing: 296.48717045783997 seconds

Memory Usage:
Memory used: 438.23 MB
Memory peak: 480.03 MB

CPU Usage:
CPU usage: 3.4%

Hashing and writing complete!
```

We can see that the SHA-1 algorithm has yielded a time of 300.89 seconds, 296.49 seconds of which came from the time for hashing and writing and 4.38 seconds of which came from the time for reading the CSV file. As for the CPU usage, we can see that we used a total of 3.4%. For the average memory usage, we see 438.23 MB, and a peak of 480.03 MB.

C. Let us take a look at test results for SHA-256

```
Benchmarking Information:
Total time: 302.0512192249298 seconds
Time for reading CSV: 4.392325162887573 seconds
Time for hashing and writing: 297.6414029598236 seconds

Memory Usage:
Memory used: 438.20 MB
Memory peak: 483.74 MB

CPU Usage:
CPU usage: 3.3%

Hashing and writing complete!
```

We can see that the SHA-256 algorithm has yielded a time of 302.05 seconds, 297.64 seconds of which came from the time for hashing and writing and 4.39 seconds of which came from the time for reading the CSV file. As for the CPU usage, we can see that we used a total of 3.3%. For the average memory usage, we see 438.20 MB, and a peak of 483.74 MB.

VIII. Conclusion of Results

We expected to see the fastest run time by the MD5 hashing algorithm. Our results show that it actually came in last place at 298.63 seconds. SHA-1 was expected to be the second fastest, but it actually came in as the fastest at 296.49 seconds. And lastly, we expected SHA-256 to be the third fastest, but actually showed to be the second fastest at 297.64 seconds. As for memory usage, we see MD5 taking up the most memory, SHA-1 as the second most, and SHA-256 as the least memory intensive. As for CPU usage, both MD5 and SHA-1 tied at 3.4%. SHA-256 yielded CPU usage of 3.3%.

Given all of these results, we are able to see that certain functions are faster and more efficient than each other, but the difference is almost negligible. These results show that although one may be slightly faster and more efficient than another, our computers are at a point in history where datasets this size are not necessarily negatively affected by a logically faster hashing algorithm. Given this, it is obvious that the user of these algorithms should choose the most secure algorithm. That of which is SHA-256 in this study.

REFERENCES

[1] "Hash Algorithm Comparison: MD5, SHA-1, SHA-2 & SHA-3," *Code Signing Store*. Available: https://codesigningstore.com/hash-algorithm-comparison

[2] "hashlib — Secure hashes and message digests — Python 3.8.4rc1 documentation," *docs.python.org*. Available: https://docs.python.org/3/library/hashlib.html

[3] "hashlib: Secure hash and message digest algorithm library," *PyPI*. Available: https://pypi.org/project/hashlib/. [Accessed: Dec. 06, 2023]

[4] Pandas, "Python Data Analysis Library — pandas: Python Data Analysis Library," *Pydata.org*, 2018. Available: https://pandas.pydata.org/

[5] "pandas," *PyPI*, Jan. 30, 2020. Available: https://pypi.org/project/pandas/

[6] R. Python, "A Beginner's Guide to the Python time Module – Real Python," *realpython.com*. Available: https://realpython.com/python-time-module/

[7] "Psutil module in Python," *GeeksforGeeks*, Mar. 10, 2020. Available: https://www.geeksforgeeks.org/psutil-module-in-python/

[8] G. Rodola, "psutil: Cross-platform lib for process and system monitoring in Python.," *PyPI*. Available: https://pypi.org/project/psutil/