Comparative Analysis of Hashing, Bloom Filters, and Cuckoo Filters for Membership Testing

AARAV GOSALIA, University of British Columbia Okanagan, Canada

Membership testing is a fundamental problem in computer science and data processing, where the task is to decide whether a given element (e.g., a username, a word, or a network address) is part of a stored set.

This report presents a comparison of different techniques: linear and binary search (baseline exact methods), hashing (exact membership), Bloom filters (probabilistic with false positives), and Cuckoo filters (probabilistic with deletions and potential false negatives). I generate large-scale synthetic datasets, run benchmarks across dataset sizes ($10K$, $100K$, $1M$, $10M$, $100M$), and analyze runtime and accuracy trade-offs. Due to computational and storage constraints, only smaller datasets are stored in the GitHub repository, while large-scale tests are reported but excluded from version control.

Additional Key Words and Phrases: Hashing, Bloom Filter, Cuckoo Filter, Membership Testing, Probabilistic Data Structures

1 Introduction

Membership testing is the task of determining whether a particular element exists within a set. It is a core operation in computer science, with applications ranging from login username checking and spell checking to intrusion detection and large-scale web systems. As datasets grow, the efficiency and scalability of membership tests become critical.

This assignment focuses on comparing several techniques for membership testing. I begin with two baseline approaches: **linear search**, which checks elements one by one, and **binary search**, which operates on sorted data. Both are exact but can become slow at scale. I then study more advanced techniques: **hashing**, which provides expected constant-time lookups, and two probabilistic data structures, the **Bloom filter** and the **Cuckoo filter**.

The goal of this work is to benchmark these methods in terms of runtime, accuracy, and error rates, using synthetic datasets of increasing sizes. By comparing exact and probabilistic approaches, I highlight the trade-offs between speed, memory usage, and correctness.

2 Background

This section describes the membership testing techniques considered in this study, their underlying data structures, and their theoretical performance.

2.1 Linear Search

Linear search is the simplest approach: given a query, each element of the dataset is checked sequentially. This requires no preprocessing or additional data structures.

- Time complexity: $O(n)$ per query.
- Space complexity: $O(n)$ to store the list of elements.

Although easy to implement, linear search is impractical for large datasets, which is why in this study I restricted it to datasets of up to $1M$ elements.

2.2 Binary Search

Binary search improves efficiency by operating on sorted data. It repeatedly halves the search interval until the query is found or the interval is empty.

- Time complexity: $O(\log n)$ per query (after an $O(n \log n)$ sort).

Author's Contact Information: Aarav Gosalia, University of British Columbia Okanagan, Kelowna, Canada.

- Space complexity: $O(n)$ storing the sorted list.

Binary search is much faster than linear search, but requires sorting in advance and is still slower than hashing or probabilistic methods for large-scale queries.

2.3 Hashing

Hashing uses a hash table or set data structure. Each element is mapped to a bucket using a hash function, allowing for near-constant time lookups. In our implementation, Python's built-in `set()` was used, which internally maintains a hash table with dynamic resizing and open addressing.

- Time complexity: $O(1)$ average case, $O(n)$ worst case.
- Space complexity: $O(n)$, but with overhead due to hash buckets and pointers (larger than sorted arrays).

Hashing provides exact results, unlike probabilistic methods, but can consume more memory. Compared to binary search, it is usually faster but less cache-friendly.

2.4 Bloom Filter

A Bloom filter is a probabilistic structure that uses a fixed-size bit array and multiple independent hash functions. When inserting an element, each hash function computes an index and marks the corresponding bit in the array. To test membership, all of those bits are checked:

(1) If any of the bits is 0, the element is *definitely not present*.
(2) If all are 1, the element is *possibly present*.

- Time complexity: $O(k)$ per query/insert, where k is the number of hash functions (usually small, e.g., 3–7).
- Space complexity: $O(n \log(1/\epsilon))$ bits, where ϵ is the target false positive probability. This is significantly smaller than storing all elements explicitly.
- Error: False positives are possible; false negatives are impossible.

In this project I used the `pybloom-live` Python package [1], which provides a standard implementation of Bloom filters. Compared to hashing, Bloom filters save memory by not storing the elements themselves. They are especially useful when the dataset is very large and only approximate membership is acceptable.

2.5 Cuckoo Filter

Cuckoo filters are based on the idea of *Cuckoo hashing*. Instead of storing full elements, they store short *fingerprints* (a few bits) of each element inside hash table "buckets." Each element can be placed in one of two possible buckets determined by its hash. If both are full, an existing fingerprint is "kicked out" (cuckoo-style) and reinserted into its alternate location. This process may cascade, but typically completes quickly.

To check membership, the query's fingerprint is computed and checked against both possible buckets.

- Time complexity: $O(1)$ expected per query/insert, similar to Bloom filters.
- Space complexity: $O(n \cdot f)$ bits, where f is the fingerprint size (commonly 4–8). In practice this is lower than hashing (since only fingerprints are stored), but typically higher than Bloom filters (due to bucket structures and relocation).
- Error: False positives are possible; false negatives are rare under very high load or after repeated insertions and relocations if items are dropped.

Compared to Bloom filters, Cuckoo filters have two major advantages:

(1) They support **deletion** of elements, since fingerprints can be removed from buckets (something Bloom filters cannot do without extensions).
(2) They avoid the fixed k hash lookups of Bloom filters — instead, only two buckets are checked.

In this project, I used the `cuckoopy` Python package [2], which provides a practical implementation of Cuckoo filters with tunable parameters (bucket size, fingerprint length).

3 Methodology

I implemented each membership method in Python and tested them using synthetic datasets.

3.1 Dataset Generation

Two types of files were used:

- `logins_N.csv`: containing randomly generated usernames representing the stored dataset, where $N \in \{10K, 100K, 1M, 10M, 100M\}$.
- `queries_Q.csv`: containing query usernames along with a label (`is_present`) indicating whether they truly belong to the dataset. Q was set to about 10% of N.

The usernames were created using synthesized functions: sequential (`user0`, `user1`, ...), adjective-noun combinations (`brave_otter_15`), and random-looking strings with indices. A mixed scheme randomly selected among these, producing a varied dataset. Queries were generated with a controlled *duplication rate* (50% present, 50% absent).

Large files (`logins_10M.csv`, `logins_100M.csv`, and corresponding queries) were too large to upload to GitHub (exceeding size limits) and were excluded via `.gitignore`. Smaller datasets were retained for reproducibility.

3.2 Evaluation Procedure

Each method was implemented as a function that:

(1) Loaded the dataset of logins.
(2) Iterated over the query set.
(3) Checked membership using the corresponding data structure.
(4) Recorded true positives, true negatives, false positives, and false negatives.

For hashing, Python's built-in `set()` was used. For Bloom filters, the `pybloom-live` package was used with error rate 0.001. For Cuckoo filters, the `cuckoopy` package was used with bucket size 2 and fingerprint length 4 bits.

3.3 Benchmarking

For each dataset size ($10K$, $100K$, $1M$, $10M$, $100M$):

- I measured **runtime** for all queries.
- I calculated **accuracy** as $(TP + TN)/(TP + TN + FP + FN)$.
- I reported **false positive rates** for Bloom and Cuckoo filters.

Linear search was only applied to $N \leq 1M$, since beyond that the runtime would be prohibitively long on the available hardware (Apple MacBook Pro M2). The Cuckoo filter was also skipped at $N = 100M$, as it consumed excessive memory and took too long to run on this machine. Similarly, I did not attempt to generate a dataset of size $N = 1B$, as the time and memory required would exceed practical limits.

4 Results

This section presents the experimental outcomes of benchmarking the five membership testing methods (Linear, Binary, Hash, Bloom, Cuckoo). I report runtimes, accuracies, and error counts across dataset sizes ranging from 10K to 100M.

4.1 Runtime Plots

Figure 1 shows the runtime of linear search on smaller datasets only (10K, 100K, 1M). Figure 2 compares Binary, Hash, Bloom, and Cuckoo across all dataset sizes. As discussed, Cuckoo at 100M was skipped due to excessive memory use and runtime overhead.

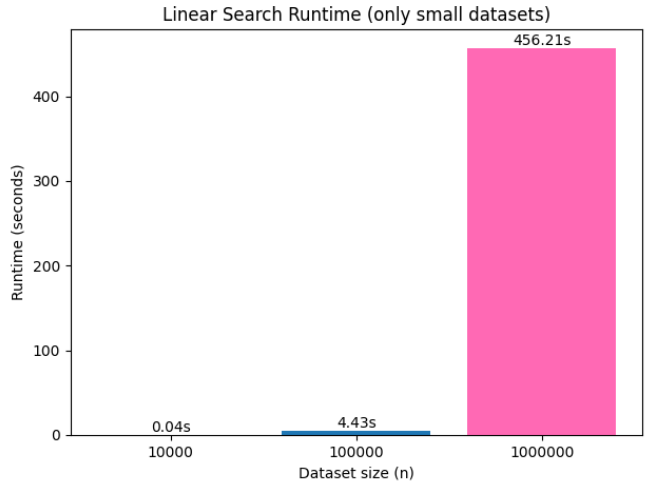

Fig. 1. Runtime of Linear Search on smaller datasets (10K, 100K, 1M).

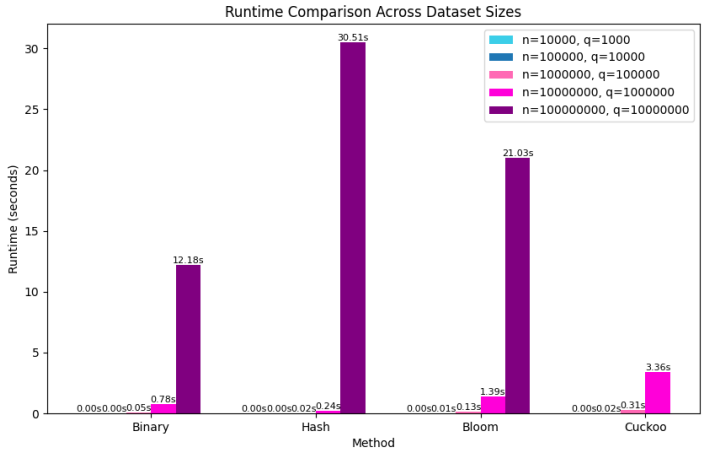

Fig. 2. Runtime comparison of Binary, Hash, Bloom, and Cuckoo filters across all dataset sizes.

4.2 Detailed Results Tables

Tables 1 and 3 summarize runtimes, accuracies, false positives (FP), and false negatives (FN). For compactness, the first three dataset sizes (10K, 100K, 1M) are shown side by side, and the last two (10M, 100M) are shown below.

Table 1. Results for smaller datasets (10K, 100K, 1M).

(a) 10K (q=1K)

Method	Time	Acc.	FP	FN
Linear	0.0450	1.0000	0	0
Binary	0.0002	1.0000	0	0
Hash	0.0001	1.0000	0	0
Bloom	0.0011	1.0000	0	0
Cuckoo	0.0021	0.9930	0	7

(b) 100K (q=10K)

Method	Time	Acc.	FP	FN
Linear	4.4328	1.0000	0	0
Binary	0.0028	1.0000	0	0
Hash	0.0009	1.0000	0	0
Bloom	0.0127	0.9999	1	0
Cuckoo	0.0243	0.9960	0	40

Table 2. Results for 1M dataset.

Method	Time	Acc.	FP	FN
Linear	456.2058	1.0000	0	0
Binary	0.0457	1.0000	0	0
Hash	0.0172	1.0000	0	0
Bloom	0.1292	0.9995	49	0
Cuckoo	0.3078	0.9955	0	449

Table 3. Results for larger datasets (10M, 100M).

(a) 10M (q=1M)

Method	Time	Acc.	FP	FN
Binary	0.7806	1.0000	0	0
Hash	0.2404	1.0000	0	0
Bloom	1.3908	0.9995	534	0
Cuckoo	3.3643	0.9955	0	4502

(b) 100M (q=10M)

Method	Time	Acc.	FP	FN
Binary	12.1779	1.0000	0	0
Hash	30.5110	1.0000	0	0
Bloom	21.0310	0.9995	5120	0
Cuckoo	Skipped (too large)			

5 Discussion

The experimental results clearly illustrate how different membership testing methods behave as dataset sizes increase from 10K to 100M.

As expected, **linear search** performed the worst. Its runtime grew rapidly with n, becoming completely impractical at 1M (over 450 seconds). The runtime plot in Figure 1 highlights this steep scaling, which is why I restricted linear search to datasets of size 1M or smaller.

Binary search produced a more surprising outcome. While its logarithmic time complexity is well understood, the benchmarks show that it scaled extremely well in practice and even outperformed all methods except hashing at 10M. This likely stems from the efficiency of Python's

highly optimized sorting and array access, which make repeated binary lookups extremely fast and cache-friendly.

Hashing (using Python's `set()`) delivered excellent performance up to 10M, consistently outperforming binary search. However, at 100M it scaled poorly, taking longer than both binary search and Bloom filters. This may be due to memory management overheads, cache inefficiency at very large sizes, or RAM limitations on the test hardware (MacBook Pro M2). Since Python sets allocate memory dynamically and maintain internal hash buckets, their efficiency can degrade when memory pressure grows.

The **Bloom filter** maintained solid performance across all tested dataset sizes. At 100M, it scaled better than hashing, completing in about 21 seconds compared to 30 seconds for hash sets. However, Bloom filters inherently allow false positives. In our tests, this meant that some absent usernames were incorrectly reported as present. This behavior is consistent with Bloom filter theory, where false positives occur due to overlapping bit positions, while false negatives remain impossible.

The **Cuckoo filter** also performed well up to 10M, with runtimes in the same ballpark as Bloom. However, it produced false negatives, meaning that some queries for truly present usernames were reported as absent. This issue arises from fingerprint collisions and bucket evictions under high load. For the 100M dataset, the Cuckoo filter was skipped entirely, since it consumed excessive memory and took too long to run on the available hardware.

Overall, the runtime plot in Figure 2 shows a clear story: binary search, hashing, Bloom, and Cuckoo all vastly outperform linear search, but each comes with trade-offs in scaling and correctness. Surprisingly, binary search often outpaced the probabilistic methods, which were expected to dominate at larger scales.

6 Conclusion

This study compared linear search, binary search, hashing, Bloom filters, and Cuckoo filters for the task of membership testing. The results highlight several key points:

- Linear search quickly became impractical, confirming its $O(n)$ scaling.
- Binary search performed far better than expected, scaling gracefully and even outperforming Bloom and Cuckoo filters despite being an exact method.
- Hashing showed outstanding performance until 10M, but then scaled poorly at 100M, likely due to memory and caching issues.
- Bloom filters scaled reliably to 100M, but produced false positives as expected.
- Cuckoo filters worked well up to 10M but produced false negatives and could not be tested at 100M due to hardware constraints.

The most surprising result was the strength of binary search, which consistently rivaled or outperformed probabilistic methods that sacrifice accuracy for speed. However, it is important to note the limitations of this study: only moderate dataset sizes (up to 100M) were tested, constrained by the performance of the MacBook Pro M2. For even larger datasets, probabilistic methods such as Bloom and Cuckoo filters may demonstrate their true advantages.

In conclusion, while binary search emerged as a surprisingly strong competitor, further testing at larger scales and on more powerful hardware is necessary to fully evaluate the benefits of Bloom and Cuckoo filters in real-world applications.

GitHub Repository

The full code (dataset generation, benchmark scripts, and unit tests) is available at: https://github.com/aaravg31/login_checker

Acknowledgements

This report was partly supported by interactive assistance from ChatGPT [3], which was used for code troubleshooting, clarifications, and grammar review.

References

[1] Jay Baird and contributors. 2016. pybloom-live: A Python implementation of Bloom filters. https://pypi.org/project/pybloom-live/. Accessed: 2025-10-02.

[2] Parsa Ghaffari and contributors. 2019. cuckoopy: A Python implementation of Cuckoo filters. https://pypi.org/project/cuckoopy/. Accessed: 2025-10-02.

[3] OpenAI. 2025. ChatGPT: Large Language Model Assistance. https://openai.com/chatgpt. Used for code troubleshooting, clarifications, and grammar review in report preparation..