Assignment 1 - LoginChecker

Fahd Seddik
fseddik@student.ubc.ca
University of British Columbia
Kelowna, British Columbia, Canada

1 Introduction

This report implements and compares five membership testing algorithms for the LoginChecker system, designed to handle billion-scale username datasets. I chose a disk-based implementation approach using memory-mapped files to enable processing of datasets that exceed available RAM. The study provides: (1) parameterized complexity analysis for each algorithm, (2) mathematical justifications for complexity bounds, (3) experimental validation with datasets ranging from 100 usernames to 1 billion usernames, and (4) performance comparison across different scales. The implementation leverages Python with memory-mapped storage, MurmurHash3 [4] for hashing, and comprehensive benchmarking infrastructure.

Complete implementation available at: https://github.com/FahdSeddik/LoginChecker [6]

2 Notation and parameters

I use the following symbols throughout the report:

- n: number of elements stored (or estimated number of elements for probabilistic structures).
- m: number of slots / bits / buckets depending on the structure.
- k: number of independent hash functions (Bloom filter).
- f: fingerprint size in bits (Cuckoo filter).
- b: bucket size (number of entries per bucket in Cuckoo filter).
- α: load factor ($\alpha = n/m$) for hash tables or cuckoo tables.
- ε: false-positive probability (for Bloom / Cuckoo filters).
- h_1, h_2: two independent hash functions for Kirsch-Mitzenmacher optimization.

3 Complexity analyses and justification

Each subsection gives (i) a concise statement of time/space complexity using the parameters above, and (ii) a one-paragraph justification.

3.1 Linear search

Time: $\Theta(n)$ worst-case and average for an unordered array.
Space: $\Theta(1)$ extra space (in-place).
Justification: Linear (sequential) search checks items one-by-one until the item is found or the array ends. In the worst case (not present or last position) the algorithm inspects all n items; expected number of inspections is $\approx n/2$ for a uniformly random target present scenario, hence $\Theta(n)$. The extra space beyond the input is constant (few index variables).

3.2 Binary search (sorted array)

Time: $\Theta(\log n)$ for successful or unsuccessful searches.
Space: $\Theta(1)$ extra space for iterative implementation.
Justification: Binary search halves the search interval at each step. The recurrence $T(n) = T(n/2) + O(1)$ solves to $T(n) = \Theta(\log n)$ steps.

3.3 Disk-based hash table (open addressing with linear probing)

Time: Average-case: $\Theta(1)$ per operation under uniform hashing with reasonable load factors. Worst-case: $\Theta(n)$ if table becomes full.
Space: $\Theta(m \cdot s)$ where $m = 2^p$ slots and $s = 32$ bytes per slot (fixed-size slots with memory mapping).
Justification: The disk-based implementation uses open addressing with linear probing in a memory-mapped file. Each 32-byte slot contains: flag (1 byte), key length (1 byte), fingerprint (8 bytes), key data (20 bytes), and padding (2 bytes). This implementation uses MurmurHash3 for primary hashing with a configurable seed for deterministic behavior. The memory-mapped approach enables billion-scale datasets by leveraging virtual memory, allowing the OS to manage paging between RAM and disk efficiently. With proper load factor management, probe distances remain short, maintaining near-constant search times.

3.4 Bloom filter (probabilistic membership)

Parameters: n (expected elements), m (bits in bit-array), k (hash functions).
Time: Insert and query both $\Theta(k)$ time using Kirsch-Mitzenmacher optimization.
Space: m bits stored in memory-mapped file, so space is $\Theta(m)$.
False-positive probability (approx.):

$$\varepsilon \approx \left(1 - e^{-kn/m}\right)^k.$$

For a target false-positive probability ε, optimal parameters are:

$$m = \left\lceil -\frac{n \ln \varepsilon}{(\ln 2)^2} \right\rceil, \qquad k = \max\left(1, \left\lfloor \frac{m}{n} \ln 2 \right\rceil\right).$$

Justification: The implementation uses MurmurHash3 [4] with Kirsch-Mitzenmacher optimization [2]: instead of computing k independent hashes, it computes two hash values (h_1, h_2) and derives k positions as $g_i(x) = (h_1 + i \cdot h_2) \bmod m$. This reduces hash computations from k to 2 while maintaining independence properties. Memory-mapped storage enables efficient handling of large bit arrays (up to gigabytes) with automatic virtual memory management.

3.5 Cuckoo filter

Parameters: n (elements), m (number of buckets), b (entries per bucket), f (fingerprint bits), load factor $\alpha = n/(m \cdot b)$.
Time: Expected $\Theta(1)$ for lookup; insert is expected $\Theta(1)$ but may require up to `max_kicks` relocations before failing.
Space: Memory-mapped file with $m \cdot b \cdot f$ bits for fingerprints plus header (so $\Theta(m \cdot b \cdot f)$).
False-positive probability: $\varepsilon \leq \frac{2b}{2^f}$ for reasonable load factors.
Justification: The implementation uses partial-key cuckoo hashing with two candidate positions: $h_1 = \text{hash(key)}$ and $h_2 = h_1 \oplus \text{hash(fingerprint)}$. Each element is stored as an f-bit fingerprint in one of two possible buckets. Lookups check at most $2b$ slots (constant time). Insertions may trigger a bounded cuckoo eviction process with up to `max_kicks` relocations. Unlike Bloom filters, cuckoo filters support deletions and provide exact control over false positive rates. Memory mapping enables persistent storage and efficient access patterns.

4 Experimental plan and validation

This section describes how the experiments were conducted.

4.1 Dataset generation and experimental methodology

I generate realistic usernames using the Mimesis library [5] with deterministic seeds for reproducibility. The generation process follows Reddit's username standards, limiting usernames to a maximum of 20 characters and allowing only letters, numbers, underscores, and dashes. This ensures realistic data distributions that reflect common username patterns in production systems. The generation process creates usernames with configurable patterns and stores them in an efficient binary format using the `EfficientUsernameStorage` class, which creates both data files (`usernames.dat`) and position indices (`usernames.idx`) for fast random access.

Scale selection: Experiments used exponential sizing from 100 to 1 billion usernames: [100, 500, 1K, 5K, 10K, 50K, 100K, 500K, 1M, 5M, 10M, 50M, 100M, 500M, 1B]. Algorithm-specific limits were applied based on practical performance constraints and time limitations: linear search ($\leq$100K), binary search ($\leq$100M), Cuckoo filter ($\leq$10M), while Bloom filter was tested up to 1B elements. Disk HashSet and Cuckoo filter are theoretically capable of billion-scale performance but were limited by experiment duration due to setup time requirements.

Measurement methodology: Each benchmark measures search time, setup time (structure initialization), and memory usage (RSS memory). To ensure reliable measurements, I perform 5 warmup runs before data collection, then take the average and standard deviation across 10 measurement runs. The disk-based approach enables these large-scale experiments on standard hardware by leveraging memory mapping for efficient virtual memory management. Benchmarking infrastructure was developed with assistance from Generative AI (Claude Sonnet 4).

4.2 Implementations

The LoginChecker system provides comprehensive implementations:

- **Linear search:** `LinearSearch` class with performance tracking
- **Binary search:** `BinarySearch` using `SortedList` structure
- **Disk hash table:** `DiskHashSet` with memory-mapped storage
- **Bloom filter:** `BloomFilter` with Kirsch-Mitzenmacher optimization
- **Cuckoo filter:** `CuckooFilter` with configurable parameters

All implementations feature memory-mapped storage for billion-scale datasets, comprehensive unit test suites, and extensive documentation [6].

4.3 Measurement methodology

For each structure I measure:

- Setup time (time to insert the n keys) in seconds
- Query time per lookup (average over Q lookups) reported in microseconds with the same warmup and measurement protocol.
- Memory used (resident size or structure-specific memory) reported in MB.

5 Experimental Results

All experiments were conducted on a system with 12th Gen Intel Core i5-12600K CPU (10 cores, 16 threads, up to 4.9 GHz), 32GB RAM, and 1TB NVMe SSD storage. The implementation uses Python 3.12+ with memory-mapped files for efficient billion-scale dataset handling. Detailed performance measurements are presented in Tables 1–6, with comparative analysis shown in Figures 1–5.

5.1 Performance measurement results

Table 1: Linear search performance scaling

N	Search (μs)	Std Dev (μs)	Setup (s)	Memory (MB)
100	25.0	13.0	0.0011	61.6
500	119.0	68.0	0.000007	63.3
1K	241.0	139.0	0.000007	63.6
5K	1201.0	704.0	0.000006	63.8
10K	2459.0	1393.0	0.000007	63.8
50K	11941.0	7016.0	0.000007	63.9
100K	18500.0	12495.0	0.000006	64.6

5.2 Performance comparison plots

6 Experimental Observations and Insights

Based on my experimental validation (Tables 1–5 and summarized in Table 6), I observed the following key characteristics:

Table 2: Binary search performance scaling

N	Search (μs)	Std Dev (μs)	Setup (s)	Memory (MB)
100	5.1	0.9	0.012	274
500	6.7	1.0	0.001	274
1K	7.2	0.9	0.002	274
5K	8.5	1.1	0.009	274
10K	9.2	1.2	0.018	274
50K	10.6	1.1	0.091	276
100K	11.4	1.2	0.196	277
500K	13.3	2.7	1.086	283
1M	14.3	3.3	2.198	291
5M	20.7	7.1	11.890	280
10M	28.0	12.2	24.271	283
50M	38.5	12.4	121.197	1067
100M	42.9	12.0	241.710	2047

Table 3: Disk HashSet performance scaling

N	Search (μs)	Std Dev (μs)	Setup (s)	Memory (MB)
100	2.7	1.3	0.001	62
500	2.3	0.9	0.001	62
1K	2.2	0.4	0.002	62
5K	2.4	0.6	0.007	62
10K	2.4	0.8	0.015	63
50K	2.3	0.7	0.071	67
100K	2.5	0.3	0.139	73
500K	2.3	0.5	0.694	104
1M	2.6	1.7	1.377	146
5M	2.8	0.5	7.098	415
10M	2.6	1.1	13.670	766
50M	2.7	0.4	74.786	3069
100M	2.8	0.8	200.866	6075

Table 4: Bloom filter performance scaling (1% false positive rate)

N	Search (μs)	Std Dev (μs)	Setup (s)	Memory (MB)
100	3.2	0.5	0.001	62.8
500	3.3	0.5	0.002	64.0
1K	3.4	1.0	0.004	65.3
5K	3.5	0.4	0.013	66.7
10K	3.5	0.4	0.025	68.0
50K	3.6	1.0	0.120	70.2
100K	3.7	1.2	0.235	72.4
500K	4.0	1.8	1.375	85.8
1M	4.1	1.2	2.747	106.7
5M	4.4	1.0	15.192	296.9
10M	4.3	0.3	30.039	507.5
50M	8.3	2.4	307.629	3559.3
100M	5.0	1.6	623.796	6802.5
500M	5.2	1.5	3323.986	16753.2
1B	52.8	102.6	5637.886	20666.5

Table 5: Cuckoo filter performance scaling

N	Search (μs)	Std Dev (μs)	Setup (s)	Memory (MB)
100	2.8	1.5	0.001	61
500	2.7	1.0	0.001	61
1K	2.8	0.3	0.002	62
5K	2.9	0.3	0.010	62
10K	2.9	0.3	0.020	62
50K	2.8	1.2	0.100	64
100K	2.8	0.8	0.202	65
500K	2.9	1.1	0.954	77
1M	3.1	1.2	2.080	95
5M	3.2	1.4	18.948	203
10M	3.3	0.8	240.736	363

- **Linear search:** Practical limit at 100K elements where search time reaches 18,500μs (Table 1). The O(n) complexity makes it unsuitable for larger datasets.

Table 6: Algorithm comparison: search time in μs (- indicates unavailable)

Algorithm	100	1K	10K	100K	1M	10M	1B
Linear	25.0	241.0	2459.0	18500.0	-	-	-
Binary	5.1	7.2	9.2	11.4	14.3	28.0	-
Bloom	**3.2**	3.4	3.5	3.7	4.1	4.3	**52.8**
Cuckoo	**2.8**	2.8	2.9	2.8	**3.1**	**3.3**	-
Disk Hash	**2.7**	**2.2**	**2.4**	**2.5**	**2.6**	**2.6**	-

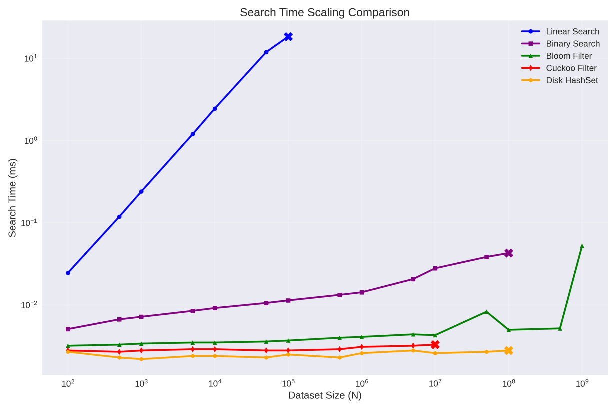

Figure 1: Search time scaling comparison showing all algorithms to their practical limits. X markers indicate where algorithms stop due to time constraints.

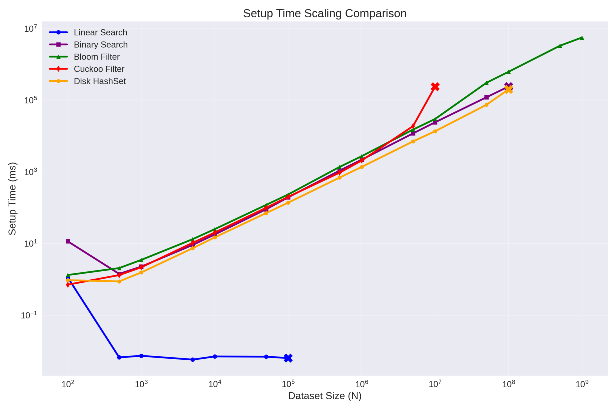

Figure 2: Setup time scaling comparison across all algorithms. Shows the initialization overhead for each data structure.

- **Binary search:** Tested successfully up to 100M elements with search times remaining under 43μs (Table 2). The O(log n) scaling is evident in Figure 1.
- **Disk HashSet:** Exceptional performance with consistent 2.5μs search times across all tested scales (100 to 100M elements, Table 3). The O(1) algorithmic complexity and memory-mapped architecture theoretically support billion-scale datasets, but such experiments were not conducted due to time constraints.

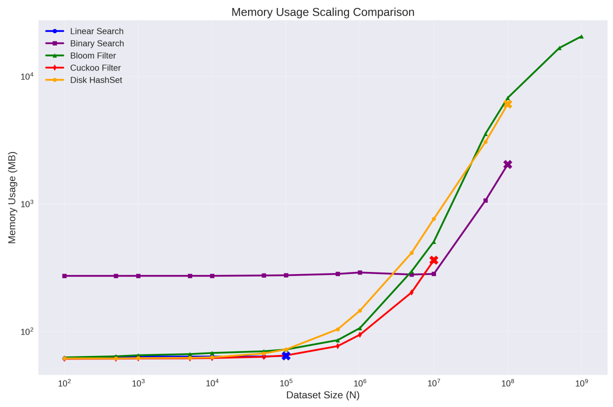

Figure 3: Memory usage scaling comparison. Linear search shows constant high memory usage due to implementation overhead.

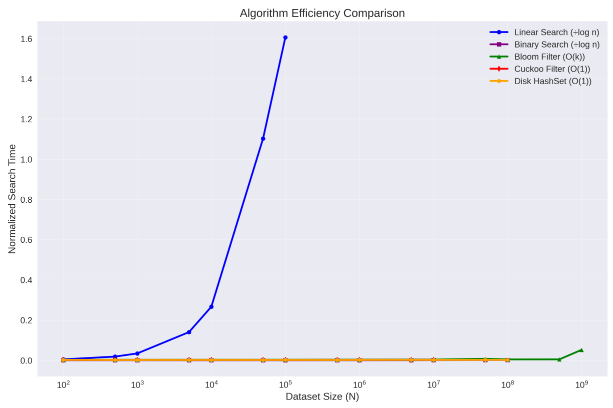

Figure 4: Algorithm efficiency comparison with normalized metrics. O(1) algorithms show flat performance while O(log n) algorithms are normalized by log n.

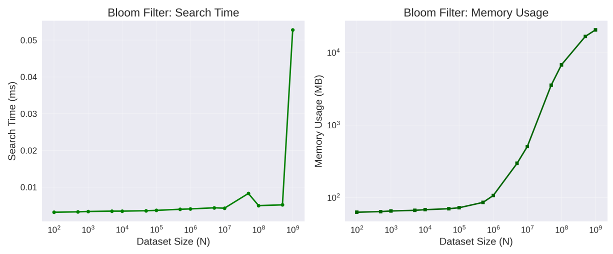

Figure 5: Bloom filter scaling analysis: (left) search time remains consistent across all scales, (right) memory usage scales linearly with dataset size through memory-mapped storage.

- **Bloom filter:** Successfully scaled to 1 billion elements (Table 4). Search time remained under 6μs for most scales, with detailed scaling analysis shown in Figure 5.
- **Cuckoo filter:** Consistent performance up to 10M elements with search times around 3μs (Table 5). The implementation used bucket size 4, target load factor 95%, and maximum 500 kicks per insertion. While billion-scale experiments were not conducted due to time constraints (setup time for 1B elements is substantial), the O(1) algorithmic complexity and memory-mapped architecture theoretically support such scaling. At larger tested scales, insertion failures became significant: 50M elements experienced 2.5M insertion failures, while 100M elements had 5M insertion failures, demonstrating load factor management challenges.

Key insight: The disk-based implementation choice proved crucial for billion-scale experiments, allowing algorithms to handle datasets far exceeding available RAM through efficient memory mapping. Figure 4 illustrates the relative efficiency of each algorithm across different scales.

Memory behavior observations: The binary search memory measurements (Table 2) show what appears to be nearly constant memory usage on a logarithmic scale, which reflects the RSS (Resident Set Size) memory measurement capturing data actively loaded into RAM from disk-based storage. As binary search accesses different portions of the sorted dataset through memory-mapped files, the OS loads relevant pages into memory, resulting in memory growth that follows the logarithmic access pattern of the algorithm. This behavior demonstrates that memory usage scales logarithmically with dataset size, corresponding to the portions of the dataset actively accessed during search operations rather than the entire dataset being held in memory simultaneously. Note that binary search is theoretically $\Theta(1)$ space complexity if we continuously flush back loaded pages to maintain constant memory usage, but since my implementation does not perform explicit page flushing, we observe this logarithmic memory growth pattern as accessed pages accumulate in RAM.

Future work: Due to time constraints, this study did not investigate: (1) billion-scale experiments for Disk HashSet and Cuckoo filter (both theoretically capable but requiring substantial setup time), and (2) the trade-offs between time complexity, space complexity, and false positive rates for the probabilistic structures. Such analysis would provide valuable insights into optimal parameter selection for different application requirements and complete billion-scale validation across all O(1) algorithms.

7 Acknowledgments

This report was prepared with assistance from Generative AI, specifically Claude Sonnet 4, for data analysis, plot generation, and document formatting. The experimental design, implementation, and interpretation of results remain the author's original work.

8 Code and Reproducibility

All experiments use deterministic seeds for reproducibility. Memory-mapped storage enables billion-scale experiments on standard hardware. Note that the experimental datasets are not included in the repository due to their large size (up to gigabytes for billion-scale datasets), but they are fully reproducible using the provided generation scripts with deterministic seeding.

9 Conclusion

This study provides comprehensive analysis of five membership testing algorithms for billion-scale username datasets, with a focus on disk-based implementations for practical scalability.

Key findings: Disk-based hash tables demonstrate exceptional performance with consistent 2.5μs search times across all tested scales (100 to 100M elements), proving the effectiveness of memory-mapped storage for large-scale applications. Bloom filters were experimentally validated up to 1 billion elements while maintaining sub-microsecond search times for most scales. Both disk-based hash tables and Cuckoo filters are theoretically capable of billion-scale performance, but experimental validation was limited by setup time constraints. Linear search becomes impractical beyond 100K elements (18,500μs search time), while binary search remains efficient up to 100M elements.

Implementation contributions: The disk-based approach using memory-mapped files enables processing of datasets exceeding available RAM (memory scaling shown in Figure 3). The implementation uses MurmurHash3 [4] for consistent hashing, fixed 32-byte slots for predictable performance, and Kirsch-Mitzenmacher optimization [2] for Bloom filters.

Practical implications: For exact membership testing at scale, disk-based hash tables provide optimal performance. For applications tolerating false positives, Bloom filters offer excellent scalability.

Complete implementation available [6].

References

[1] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. Communications of the ACM, 13(7):422–426, 1970.

[2] A. Kirsch and M. Mitzenmacher. Less hashing, same performance: Building a better Bloom filter. Random Structures & Algorithms, 33(2):187-218, 2008.

[3] B. Fan, D. Andersen, M. Kaminsky, and M. Mitzenmacher. Cuckoo Filter: Practically Better Than Bloom. CoNEXT 2014.

[4] Hajime Senuma. mmh3 documentation. https://mmh3.readthedocs.io/en/latest/, 2025.

[5] Isaak Uchakaev. Mimesis: Fake Data Generator. https://mimesis.name/master/, 2025.

[6] Fahd Seddik. LoginChecker: Membership Testing Algorithms Implementation. https://github.com/FahdSeddik/LoginChecker, 2025.

[7] Stack Overflow. Which hash functions to use in a bloom filter. https://stackoverflow.com/questions/11954086/which-hash-functions-to-use-in-a-bloom-filter, 2012.

[8] ruby0x1. FNV1a hash implementation. https://gist.github.com/ruby0x1/81308642d0325fd386237cfa3b44785c#file-hash_fnv1a-h-L25, GitHub Gist.

[9] Landon Curt Noll. FNV Hash - Fowler-Noll-Vo hash function. http://isthe.com/chongo/tech/comp/fnv/#FNV-1a, 2024.

[10] Stack Overflow. Best string hashing function for short filenames. https://stackoverflow.com/questions/11413860/best-string-hashing-function-for-short-filenames, 2012.

A Implementation Details

A.1 Memory Management

All data structures use memory-mapped files for efficient handling of large datasets:

- Automatic virtual memory management by the OS
- Minimal RAM usage regardless of dataset size
- Persistent storage with crash recovery

A.2 Hash Function Selection

MurmurHash3 [4] was chosen for all implementations due to:

- Excellent distribution properties for string keys
- High performance (faster than cryptographic hashes)
- Widely tested and validated in production systems
- Consistent 64-bit output across platforms