COMP 480 Final Project 1–32

Exploring Optimal Probabilistic Structures in Username-Data Storage System

Alina Wang JW146@RICE.EDU

Eric Jia EJJ3@RICE.EDU

Abstract

In lectures, we have studied a variety of hashing techniques and associated data structures, some of which are bloom filters and consistent hashing. Seeing a bloom filter's capability of managing large numbers of strings inspired the idea of managing names, and using consistent hashing to store objects across a distributed system brought the idea of a data storage service to fruition. With our project, we will explore a mock data storage service across a distributed system that allows users to make their own account, store data, and retrieve data.

1. Problem Setting

Traditional Bloom filters are effective for membership testing but suffer from drawbacks when it comes to deletions and high load factors, due to their use of randomized hash functions. These issues become more prominent in applications that require high insertion and deletion rates or handle very large datasets, such as database systems, network intrusion detection, or large-scale caching. In our study, we explore the impact of combining consistent hashing, implemented using list(ring), binary search trees (BSTs) and red-black trees, with different types of filters (standard Bloom filter, counting Bloom filter, and cuckoo filter). Consistent hashing will help distribute keys, improving the overall efficiency of lookups. Using BSTs and red-black trees can potentially allow better organization and even faster lookup times by efficiently managing the nodes where each hash bucket is stored, especially for filters with high update rates like counting and cuckoo filters.

2. Related Work

Some of the structures that we plan to use are outlined below.

2.1. Counting Bloom Filters

In a simple Bloom filter, insertion is straightforward; however, deletion poses a significant challenge. Since multiple keys may hash to the same bit position, setting a bit to 0 during deletion could inadvertently remove other elements that rely on that bit, falsely indicating their absence in the set. To address this issue, Fan et al. (2000) introduced the counting Bloom filter. Instead of storing single bits, each position in the filter holds a small counter.

On insertion, the associated counters are incremented, while deletion decrements them. This adjustment helps maintain the filter's integrity by ensuring that other elements are not incorrectly removed. However, the counting Bloom filter also has drawbacks. The additional counters increase the memory requirement compared to a standard Bloom filter, which can be a limitation for memory-constrained applications. Additionally, the risk of counter overflow must be managed, particularly if elements are inserted and deleted frequently.

2.2. Cuckoo Filters

While the counting Bloom filter allows for deletion, there is still room for improvement in terms of lookup efficiency. The cuckoo filter addresses this by optimizing lookup speed using only two hash functions and a series of small-sized buckets to store compact fingerprints of keys. By storing a fingerprint in two possible buckets, the cuckoo filter significantly reduces the lookup time since only two buckets need to be checked, which is faster than scanning multiple positions as with traditional Bloom filters, making it suitable for applications that demand high performance and flexibility. Cuckoo filters also have lower false positive rates than standard Bloom filters under high load (Fan et al. 2014). However, this lookup speed advantage comes with trade-offs in insertion efficiency. Since each bucket has limited capacity, insertion can fail if both target buckets are full. In such cases, the cuckoo filter uses an eviction process where an existing fingerprint is displaced, or removed, to make room for the new item. This eviction mechanism can become increasingly slow as more buckets reach capacity, especially in scenarios with high insertion rates or limited bucket space. As the filter nears full capacity, the probability of multiple evictions rises, leading to longer insertion times and even the possibility of insertion failures if no suitable location can be found.

2.3. Consistent Hashing with BSTs and Red-Black Trees

Consistent hashing is commonly used in distributed hash tables (DHTs) to balance data across nodes, minimizing data reallocation when nodes change. BSTs and red-black trees, due to their sorted nature, can make this reallocation efficient by maintaining ordered hash buckets and reducing rehashing. Red-black trees maintain a property of roughly balanced heights, improving search and update times over simple binary search trees in high-throughput applications. Compared to a list, where insertion and deletion are O(n) operations, binary search trees perform the same operations in O(log(n)) time (amortized), and red-black trees can do so in O(log(n)) time (CLRS). Storing each machine of a distributed system in a binary tree-like structure massively improves the asymptotic time complexities compared to lists, especially when concerned with the reality of system downtime requiring frequent remapping of data to different machines.

3. Hypothesis

We hypothesize that combining consistent hashing with a red-black tree in a cuckoo filter will improve lookup latency and efficiency, especially under high insertion and deletion workloads, in our use case of a distributed data storage system.

4. Experimental Settings

To evaluate the performance of each filter-tree combination, we'll conduct experiments on large datasets simulating realistic membership testing workloads. Our test data set will be an app user dataset from Kaggle. This dataset contains over 10,000 rows that will simulate a real-world usage. More data and different datasets may be added to simulate larger workloads. Our baseline is a simple and direct implementation of consistent hashing, using a list as a key ring, and a hash table instead of a filter. This structure acts as the control variable that our benchmarks can compare against. Our evaluation metrics are listed below:

- False Positive Rate: This is essential to ensure accuracy is not compromised.
- Lookup Latency: The time taken to perform membership checks will provide insights into performance improvements.
- Insertion Times: Each filter will be tested on the capability of inserting. We can extrapolate the data for deletion as well since these operations have symmetric implementations.
- Memory Usage: The memory footprint will be tracked to confirm that optimizations don't cause excessive memory overhead.

Our experimental process will have 3 phases:

- Tuning Parameters: We will test different sizes for the filters to find optimal configurations that minimize lookup time without increasing the false positive rate.
- Comparison of Configurations: For each dataset, we will compare the false positive rates, lookup latencies, insertion and deletion times, and memory usage across:
  - Standard Bloom filter
  - Counting Bloom filter
  - Cuckoo filter
- Filter-Tree Combinations: Each filter (standard, counting, cuckoo) will be implemented with consistent hashing backed by list, BST and red-black tree respectively. Five different false positive rates (0.01, 0.005, 0.001, 0.0005, 0.0001) will be considered for each applicable filter.

5. Filter Analysis

Cuckoo Filter Set Up Analysis

5.0.1. Experiment Design

Dataset: The dataset used for this experiment consists of unique names extracted from the dataset USA Name Data. The dataset is first loaded and processed into a list of unique names. These names serve as the input for the cuckoo filter's insertion and query operations.
Cuckoo Filter Control Variables:

- Load Factor: Set to 0.9, this parameter defines the filter's capacity and determines how many items can be inserted before resizing is required.
- Test Split: The dataset is split into training for insertion and test sets for query for false positive analysis. 80% of the names are used for insertion into the cuckoo filter, while the remaining 20% are used to calculate the false positive rate by querying items that were not inserted.

5.0.2. Fingerprint Size

The fingerprint size in a Cuckoo Filter determines the length of the unique identifier stored for each inserted item and significantly impacts the filter's performance and efficiency. Larger fingerprints reduce hash collisions and false positives, making them ideal for applications requiring high accuracy, but they increase memory consumption and computational overhead. Conversely, smaller fingerprints save memory but may lead to higher false positives and more frequent insertion failures due to collisions. The choice of fingerprint size involves a trade-off between memory efficiency and performance. Optimal configuration depends on the desired accuracy, load factor, and system resources.

Experiment Result

Fp Size	Failed Inserts	Insertion(s)	Query (s)	False Positive	Memory Usage (bytes)
2	8	0.7940	0.0978	0.4408	5,049,312
3	2	0.5790	0.0983	0.0029	5,526,704
4	0	0.5321	0.0984	0.0000	5,529,752
5	2	0.5605	0.0995	0.0000	5,529,592
6	0	0.5105	0.0997	0.0000	5,529,720
7	0	0.5498	0.1001	0.0000	5,529,944
8	0	0.5207	0.1007	0.0000	6,192,464

Table 1: Performance Metrics for Cuckoo Filter with Varying Fingerprint Size

The results in Table 1 and Figure 1 reveal the following observations:

- **Insertion Success Rate:** The insertion success rate remains consistently high across all configurations, achieving 100% for fingerprint sizes of 4, 6, 7, and 8 bytes. For sizes 2, 3, and 5 bytes, the rate slightly falls below 100%, though it is still excellent.
- **Insertion Time:** Insertion times generally decrease with larger fingerprint sizes, reaching the lowest time of 0.5105 seconds for 6 bytes. For sizes beyond 6 bytes, insertion times slightly increase, indicating a tradeoff between fingerprint size and insertion efficiency.
- **Query Time:** Query times show minimal variation across different fingerprint sizes, averaging approximately 0.1 seconds. This stability indicates that the fingerprint size has a negligible impact on query performance.

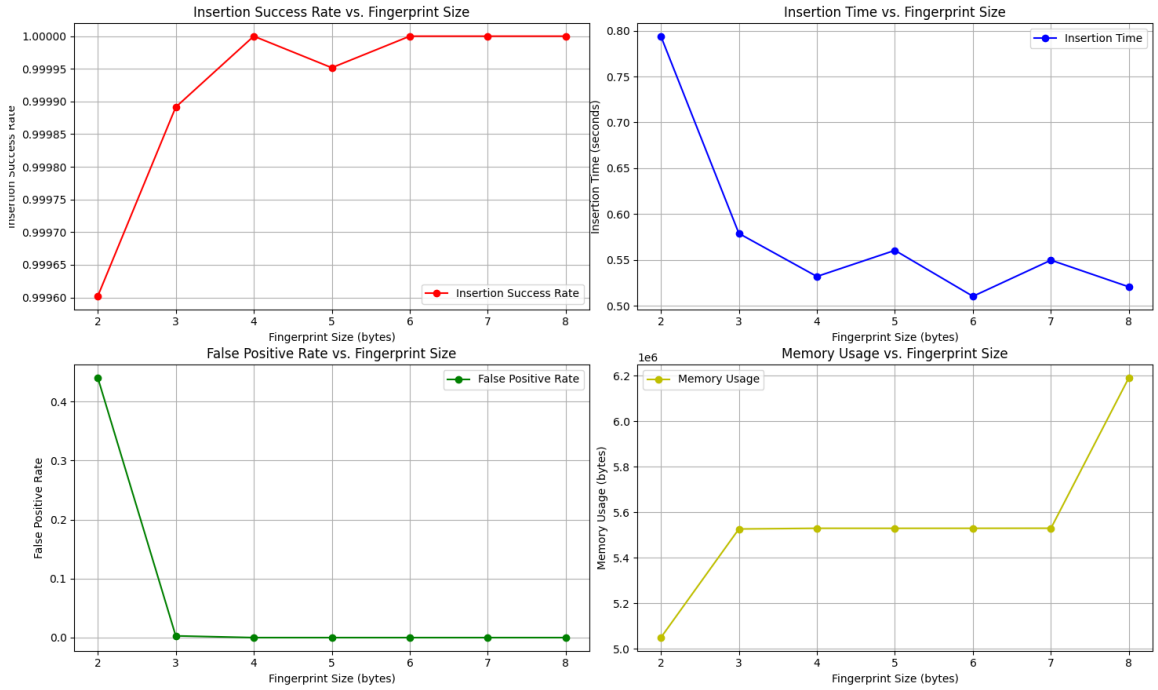


Figure 1: Fingerprint Size Analysis

- **False Positive Rate:** The false positive rate is highest for a 2-byte fingerprint (44.08%) and drops to near-zero for sizes of 3 bytes and above. This demonstrates the importance of increasing fingerprint size to mitigate false positives effectively.
- **Memory Usage:** Memory usage steadily increases with fingerprint size. The memory requirements for 3 to 7 bytes are similar, hovering around 5,529,000 bytes. However, an 8-byte fingerprint requires significantly more memory (6,192,464 bytes), highlighting a potential tradeoff between memory efficiency and fingerprint precision.

Conclusion: A fingerprint size of 6 bytes continues to provide an excellent balance, offering 100% insertion success, the lowest insertion time, no false positives, and stable memory usage. While sizes 7 and 8 bytes maintain similar performance, the increased memory usage for 8 bytes may make it less desirable under memory constraints.

5.0.3. Bucket Size

The bucket size in a Cuckoo Filter determines how many items can be stored in each bucket and plays a crucial role in balancing insertion success and memory efficiency. Larger bucket sizes reduce the likelihood of eviction during insertion, increasing the filter's tolerance for higher load factors and improving insertion success rates. However, this comes at the cost of slightly increased memory usage per bucket and potentially slower query times due to additional checks within larger buckets. Smaller bucket sizes are more memory-efficient but may lead to more frequent insertion failures and evictions under high load, limiting the filter's overall capacity. Selecting an appropriate bucket size requires considering the

expected load, insertion patterns, and system constraints to optimize performance.

Experiment Result

Fp Size	Insert Success	Insertion(s)	Query (s)	False Positive	Memory Usage (bytes)
2	0.9998	0.746	0.096	0.0	7,699,536
3	1.0000	0.517	0.094	0.0	6,245,280
4	0.9999	0.537	0.096	0.0	5,529,144
5	0.9999	0.496	0.099	0.0	5,517,888
6	1.0000	0.465	0.096	0.0	5,205,496

Table 2: Performance Metrics for Cuckoo Filter with Varying Bucket Sizes

The results in Table 2 and Figure 2 reveal the following observations:

- **Insertion Success Rate:** The insertion success rate remains exceptionally high (99.98%) for all configurations, demonstrating the effectiveness of the Cuckoo filter even at high loads.
- **Insertion Time:** The insertion time decreases as the bucket size increases, suggesting better handling of potential collisions due to fewer necessary evictions.
- **Query Time:** Query time remains consistently low across all bucket sizes (approximately 0.094–0.099 seconds), indicating stable performance regardless of configuration.
- **False Positive Rate:** The false positive rate is 0 for all configurations, confirming the high reliability of the setup.
- **Memory Usage:** Larger bucket sizes reduce memory usage, as fewer buckets are required. The memory usage decreases from 7,699,536 bytes for bucket size 2 to 5,205,496 bytes for bucket size 6.

Conclusion: A bucket sizes of 6 provide an optimal balance between high insertion success rates, minimal insertion time, and efficient memory usage.

5.0.4. Number of Buckets

The formula for calculating the number of buckets in a Cuckoo Filter,

$$N_{\text{buckets}} = \frac{S}{L \times B},$$

is derived from the need to store a dataset of size S within the constraints of bucket size B and load factor L. The dataset size S represents the total number of elements to be inserted into the filter, while the bucket size B defines how many slots each bucket has to hold elements. The load factor L determines how full a bucket can be before resizing is triggered, thus affecting the number of elements each bucket can effectively hold. By

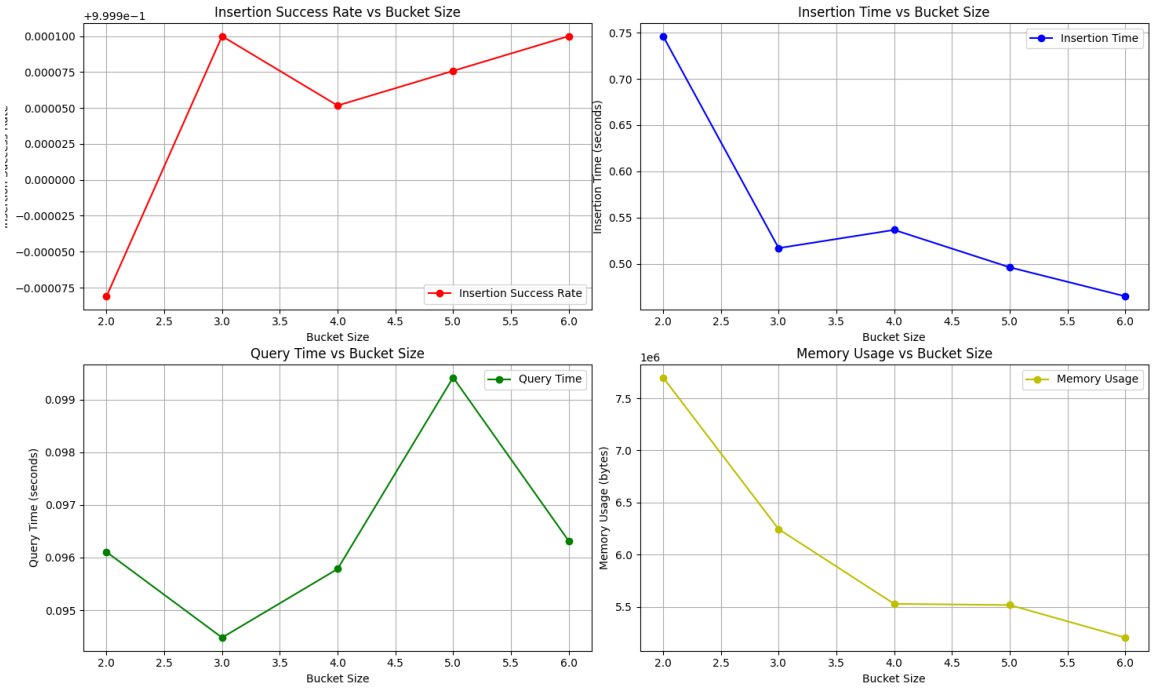


Figure 2: Bucket Size Analysis

dividing the dataset size S by $L \times B$, the formula accounts for the total available slots after considering the load factor, providing the necessary number of buckets to store all elements without exceeding the capacity of the filter. This approach ensures that the filter remains efficient and avoids overflow by distributing elements evenly across the available buckets.

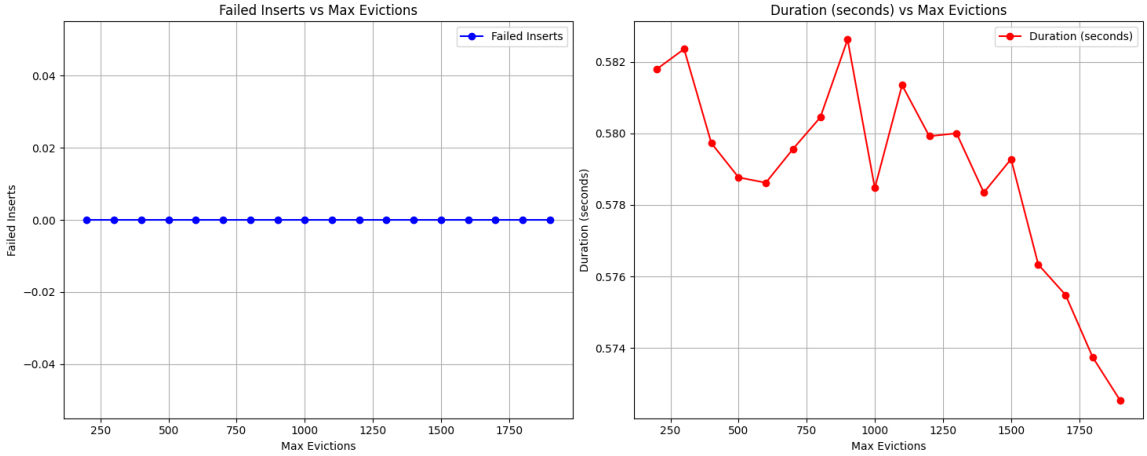
Given the dataset size $S = 103564 \times 0.8$, load factor $L = 0.9$, and bucket size $B = 6$, the formula for the number of buckets is:

$$N_{\text{buckets}} = \frac{S}{L \times B} = \frac{103564 \times 0.8}{0.9 \times 6} \approx 15343$$

Conclusion: A number of buckets of 15343 provides an optimal configuration for the given dataset size, load factor, and bucket size. This choice ensures efficient performance, balancing memory usage and query speed, while maintaining a low rate of insertion failures.

5.0.5. Max Eviction

The maximum eviction limit in a Cuckoo Filter controls how many relocation attempts are made when inserting an item encounters a conflict. A higher eviction limit increases the likelihood of successful insertions, especially at higher load factors, by allowing more attempts to resolve conflicts. However, it also increases the time complexity of insertions, as prolonged relocation chains can significantly delay the process. On the other hand, a lower eviction limit results in faster insertion times but a higher probability of failed insertions, particularly when the filter is near capacity. Therefore, the max eviction parameter is a critical tuning factor for balancing insertion success rate and time efficiency, depending on

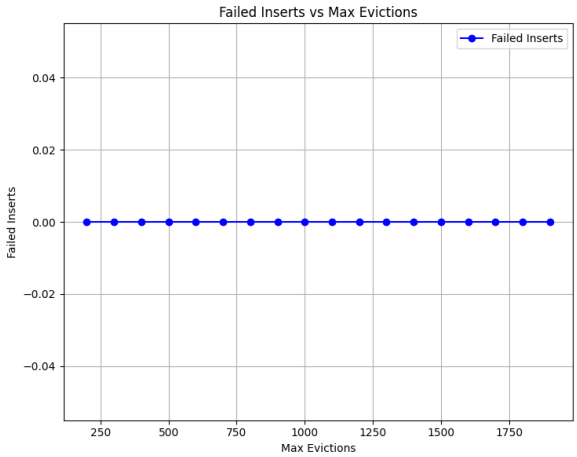


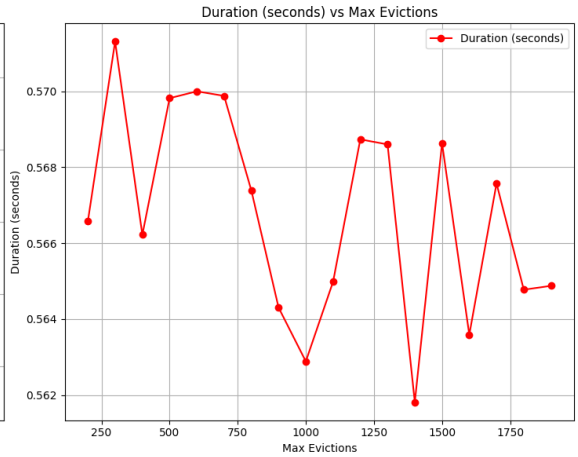
the application's tolerance for delays and failed inserts.

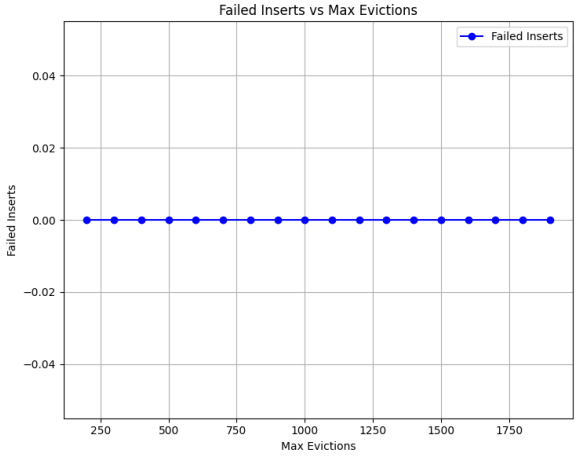
Experiment Result

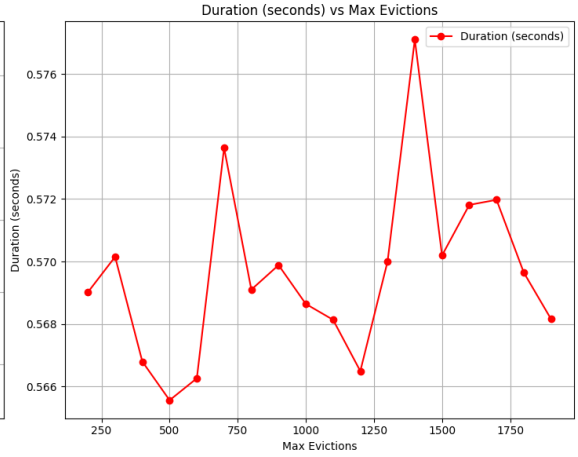
Max Evictions	Failed Inserts	Duration (seconds)
200	0	0.5697
300	0	0.5689
400	0	0.5684
500	0	0.5743
600	0	0.5691
700	0	0.5744
800	0	0.5706
900	0	0.5768
1000	0	0.5706
1100	0	0.5718
1200	0	0.5704
1300	0	0.5707
1400	0	0.5669
1500	0	0.5698
1600	0	0.5702
1700	0	0.5734
1800	0	0.6012
1900	0	0.5720

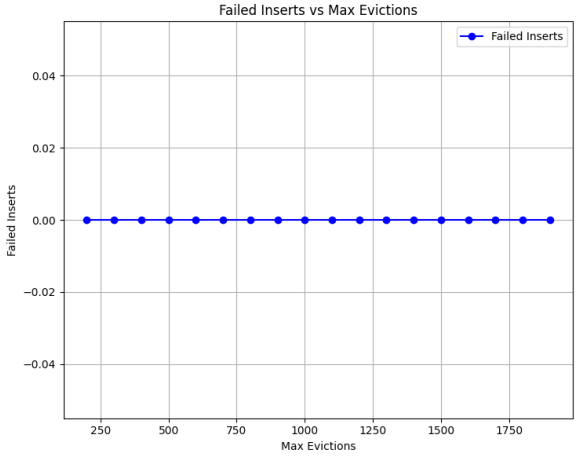
Conclusion: In multiple runs of the experiment 5.0.5, 5.0.5, 5.0.5, 5.0.5, using a fingerprint size of 4, a bucket size of 6, and the entire dataset as the insertion set, the failure rate for insertions consistently remains at 0. Additionally, the insertion time does not display a consistent pattern, indicating that variations in insertion durations are likely influenced by other factors such as system performance or random hash collisions. Based on these

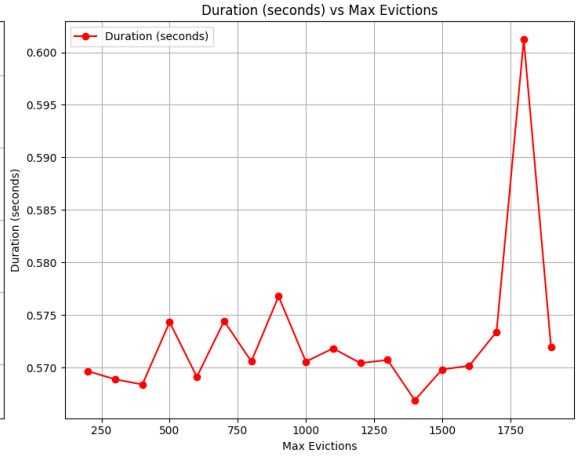

Failed Inserts vs Max Evictions
Failed Inserts
Failed Inserts
0.04
0.02
0.00
−0.02
−0.04
250
500
750
1000
1250
1500
1750
Max Evictions


Duration (seconds) vs Max Evictions
Duration (seconds)
Duration (seconds)
0.570
0.568
0.566
0.564
0.562
250
500
750
1000
1250
1500
1750
Max Evictions


Failed Inserts vs Max Evictions
Failed Inserts
Failed Inserts
0.04
0.02
0.00
−0.02
−0.04
250
500
750
1000
1250
1500
1750
Max Evictions


Duration (seconds) vs Max Evictions
Duration (seconds)
Duration (seconds)
0.576
0.574
0.572
0.570
0.568
0.566
250
500
750
1000
1250
1500
1750
Max Evictions


Failed Inserts vs Max Evictions
Failed Inserts
Failed Inserts
0.04
0.02
0.00
−0.02
−0.04
250
500
750
1000
1250
1500
1750
Max Evictions


Duration (seconds) vs Max Evictions
Duration (seconds)
Duration (seconds)
0.600
0.595
0.590
0.585
0.580
0.575
0.570
250
500
750
1000
1250
1500
1750
Max Evictions

findings, a reasonable bound for the maximum number of evictions is 500, as it ensures efficient insertions without any failures while avoiding excessive relocation attempts that could increase insertion time.

5.0.6. Final Cuckoo Filter Set Up Configuration

Given the names data, we configured the cuckoo filter for the optimal performance using the following parameters:

- **Bucket Size**: 6
- **Load Factor**: 0.9
- **Number of Buckets**:
$$\frac{\text{Dataset Size}}{\text{Load Factor} \times \text{Bucket Size}}$$
- **Fingerprint Size**: 6
- **Max Eviction**: 500

Dataset Size =	8284	41425	74565	82851
Load Factor = 0.1	0.4881	2.4529	4.4024	4.8496
Load Factor = 0.5	0.4897	2.4877	4.3980	4.8597
Load Factor = 0.9	0.7123	2.8725	7.5901	6.7192

Table 4: Insertion Time Table for Cuckoo Filter

Dataset Size =	8284	41425	74565	82851
Load Factor = 0.1	0.1251	0.6220	1.1218	1.2319
Load Factor = 0.5	0.1261	0.6213	1.1241	1.2366
Load Factor = 0.9	0.1256	0.6226	1.1277	1.2379

Table 5: Query Time Table for Cuckoo Filter

5.1. Filter Scalability and Performance Analysis

In this study, we conducted a comprehensive performance and scalability analysis, focusing on key metrics such as insertion time, query time, false positive rate, and memory consumption. The analysis was performed across various load factors and data set sizes, providing insights into the behavior and efficiency of the different filter types under different operational conditions.

5.1.1. Cuckoo Filter

Conclusion:

- **Insertion Time:**
  - As the dataset size increases, the insertion time for all load factors increases. For example, at a load factor of 0.1, insertion time increases from 0.4881 seconds for 8284 elements to 4.8496 seconds for 82851 elements.
  - The insertion time grows more significantly at higher load factors, with load factor 0.9 having the highest insertion times across all dataset sizes.
- **Query Time:**
  - Query times remain relatively consistent across varying dataset sizes and load factors. The query time is approximately 0.125 seconds for small datasets and slightly increases as dataset size grows, but it remains quite stable.

Dataset Size =	8284	41425	74565	82851
Load Factor = 0.1	0	0	0	0
Load Factor = 0.5	0	0	0	0
Load Factor = 0.9	0	0	0	0

Table 6: False Positive Rate Table for Cuckoo Filter

Dataset Size =	8284	41425	74565	82851
Load Factor = 0.1	1560000	7749600	13945080	15511544
Load Factor = 0.5	744392	3721600	6700112	7448720
Load Factor = 0.9	652368	3265072	5875408	6530696

Table 7: Memory Usage Table for Cuckoo Filter

Dataset Size =	8284	41425	74565	82851
Load Factor = 0.1	0.0633	0.3147	0.7677	0.6271
Load Factor = 0.5	0.0640	0.3121	0.5601	0.6179
Load Factor = 0.9	0.0630	0.3146	0.5627	0.6199

Table 8: Insertion Time Table for Bloom Filter

- **False Positive Rate:**
  - The false positive rate remains at 0 for all load factors and dataset sizes, indicating the cuckoo filter is highly effective in minimizing false positives under the tested conditions.
- **Memory Usage:**
  - Memory usage increases with both dataset size and load factor. For instance, at a load factor of 0.1, memory usage rises from 1,560,000 bytes for 8284 elements to 15,511,544 bytes for 82851 elements.
  - Higher load factors such as 0.9 result in higher memory consumption, but the memory usage is still manageable even at the largest dataset size.

5.1.2. Bloom Filter

Conclusion:

- **Insertion Time:**
  - Insertion times are relatively low and stable across different dataset sizes and load factors, with insertion times ranging from 0.0139 seconds for small datasets to 0.1412 seconds for large datasets at a load factor of 0.9.
- **Query Time:**

Dataset Size =	8284	41425	74565	82851
Load Factor = 0.1	0.0139	0.0688	0.1215	0.1340
Load Factor = 0.5	0.0141	0.0701	0.1244	0.1359
Load Factor = 0.9	0.0144	0.0708	0.1261	0.1412

Table 9: Query Time Table for Bloom Filter

Dataset Size =	8284	41425	74565	82851
Load Factor = 0.1	0	0	0	0
Load Factor = 0.5	0.0053	0.0060	0.0061	0.0057
Load Factor = 0.9	0.0352	0.0394	0.0381	0.0381

Table 10: False Positive Rate Table for Bloom Filter

Dataset Size =	8284	41425	74565	82851
Load Factor = 0.1	65176	323040	580896	645368
Load Factor = 0.5	13608	65184	116752	129648
Load Factor = 0.9	7880	36528	65184	72344

Table 11: Memory Usage Table for Bloom Filter

  - Query times are consistently low and show minimal variation with increasing dataset size and load factor, with values around 0.125 to 0.1412 seconds.

- **False Positive Rate:**
  - At a load factor of 0.1, the false positive rate is 0, demonstrating the filter's high accuracy at lower load factors.
  - At higher load factors (0.9), the false positive rate increases, peaking at 0.0381, indicating a tradeoff between efficiency and accuracy at higher load factors.
- **Memory Usage:**
  - Memory usage is relatively low for Bloom filters. For instance, at a load factor of 0.1, memory usage increases from 65,176 bytes for 8284 elements to 645,368 bytes for 82851 elements.
  - Memory usage decreases with increasing load factors, reflecting the efficiency of Bloom filters in managing memory under higher load factors.

5.1.3. Counting Bloom Filter

Conclusion:

- **Insertion Time:**

Dataset Size =	8284	41425	74565	82851
Load Factor = 0.1	0.0701	0.3537	0.6432	0.7147
Load Factor = 0.5	0.0678	0.3347	0.8535	0.6619
Load Factor = 0.9	0.0675	0.3327	0.5982	0.6610

Table 12: Insertion Time Table for Counting Bloom Filter

Dataset Size =	8284	41425	74565	82851
Load Factor = 0.1	0.0144	0.0721	0.1271	0.1391
Load Factor = 0.5	0.0146	0.0736	0.1285	0.1439
Load Factor = 0.9	0.0147	0.0748	0.1324	0.1451

Table 13: Query Time Table for Counting Bloom Filter

Dataset Size =	8284	41425	74565	82851
Load Factor = 0.1	0	0	0	0
Load Factor = 0.5	0.0053	0.0060	0.0061	0.0057
Load Factor = 0.9	0.0352	0.0394	0.0381	0.0381

Table 14: False Positive Rate Table for Counting Bloom Filter

  - Insertion times for Counting Bloom Filters are similar to those of the standard Bloom Filter. They remain consistent at around 0.067 to 0.145 seconds across varying dataset sizes and load factors.

- **Query Time:**
  - Query times are stable and show minimal variation across different dataset sizes, ranging from 0.0144 to 0.1451 seconds. The query time slightly increases with the load factor and dataset size.
- **False Positive Rate:**
  - The false positive rate for Counting Bloom Filters is zero for low load factors (0.1) but increases as the load factor grows, similar to the standard Bloom Filter.
- **Memory Usage:**
  - Memory usage for Counting Bloom Filters is higher than the standard Bloom Filter. For example, at a load factor of 0.1, memory usage increases from 4,126,000 bytes for 8284 elements to 41,258,464 bytes for 82851 elements.
  - Higher load factors, such as 0.9, lead to lower memory usage, with the filter effectively managing memory across large datasets.

Dataset Size =	8284	41425	74565	82851
Load Factor = 0.1	4126000	20629368	37132248	41258464
Load Factor = 0.5	825824	4126496	7427072	8252320
Load Factor = 0.9	459112	2292808	4126496	4584936

Table 15: Memory Usage Table for Counting Bloom Filter

5.2. Filter Performance Comparison Analysis

To compare the performance of different filters under the same load factor and dataset size, we can analyze the results for each filter in terms of insertion time, query time, false positive rate, and memory usage.

5.2.1. Experiment Result

Filter Type	Insertion Time	Query Time	False Positive	Memory Usage (bytes)
Counting Bloom Filter	0.0698	0.0143	0.0	4126000
Cuckoo Filter	0.4941	0.1251	0.0	1557280
Bloom Filter	0.0639	0.0141	0.0	65176

Table 16: Performance metrics for Load Factor = 0.1, Dataset Count = 8284.

Filter Type	Insertion Time	Query Time	False Positive	Memory Usage (bytes)
Counting Bloom Filter	0.3544	0.0724	0.0	20629368
Cuckoo Filter	2.4676	0.6275	0.0	7747584
Bloom Filter	0.3172	0.0690	0.0	323040

Table 17: Performance metrics for Load Factor = 0.1, Dataset Count = 41425.

Filter Type	Insertion Time	Query Time	False Positive	Memory Usage (bytes)
Counting Bloom Filter	0.6515	0.1290	0.0	37132248
Cuckoo Filter	4.4955	1.1282	0.0	13946680
Bloom Filter	0.5666	0.1237	0.0	580896

Table 18: Performance metrics for Load Factor = 0.1, Dataset Count = 74565.

Filter Type	Insertion Time	Query Time	False Positive	Memory Usage (bytes)
Counting Bloom Filter	0.7355	0.1455	9.656×10^{-5}	41258464
Cuckoo Filter	4.9525	1.2840	0.0	15507128
Bloom Filter	0.6962	0.1413	9.656×10^{-5}	645368

Table 19: Performance metrics for Load Factor = 0.1, Dataset Count = 82851.

5.2.2. Analysis

- **Insertion Time:** All filter types exhibit an increase in insertion time as the dataset size grows. Among them, cuckoo filters are the slowest, especially at higher load factors.

Filter Type	Insertion Time	Query Time	False Positive	Memory Usage (bytes)
Counting Bloom Filter	0.0756	0.0157	0.0	8270000
Cuckoo Filter	0.5163	0.1325	0.0	3114560
Bloom Filter	0.0685	0.0153	0.0	130352

Table 20: Performance metrics for Load Factor = 0.5, Dataset Count = 8284.

Filter Type	Insertion Time	Query Time	False Positive	Memory Usage (bytes)
Counting Bloom Filter	0.3652	0.0760	0.0	41358736
Cuckoo Filter	2.5704	0.6532	0.0	15495168
Bloom Filter	0.3260	0.0712	0.0	646080

Table 21: Performance metrics for Load Factor = 0.5, Dataset Count = 41425.

Filter Type	Insertion Time	Query Time	False Positive	Memory Usage (bytes)
Counting Bloom Filter	0.6725	0.1331	0.0	74343264
Cuckoo Filter	4.6782	1.1742	0.0	27893360
Bloom Filter	0.5880	0.1264	0.0	1161792

Table 22: Performance metrics for Load Factor = 0.5, Dataset Count = 74565.

Filter Type	Insertion Time	Query Time	False Positive	Memory Usage (bytes)
Counting Bloom Filter	0.7560	0.1472	9.656×10^{-5}	82516928
Cuckoo Filter	5.0528	1.3247	0.0	31014256
Bloom Filter	0.7205	0.1448	9.656×10^{-5}	1288960

Table 23: Performance metrics for Load Factor = 0.5, Dataset Count = 82851.

Filter Type	Insertion Time	Query Time	False Positive	Memory Usage (bytes)
Counting Bloom Filter	0.0831	0.0168	0.0	14886000
Cuckoo Filter	0.5468	0.1399	0.0	5599104
Bloom Filter	0.0752	0.0162	0.0	233664

Table 24: Performance metrics for Load Factor = 0.9, Dataset Count = 8284.

Filter Type	Insertion Time	Query Time	False Positive	Memory Usage (bytes)
Counting Bloom Filter	0.4027	0.0837	0.0	74323440
Cuckoo Filter	2.7830	0.7071	0.0	27921584
Bloom Filter	0.3565	0.0763	0.0	1161792

Table 25: Performance metrics for Load Factor = 0.9, Dataset Count = 41425.

Filter Type	Insertion Time	Query Time	False Positive	Memory Usage (bytes)
Counting Bloom Filter	0.7403	0.1467	0.0	148686528
Cuckoo Filter	5.1379	1.2884	0.0	55817472
Bloom Filter	0.6544	0.1408	0.0	2326656

Table 26: Performance metrics for Load Factor = 0.9, Dataset Count = 74565.

Filter Type	Insertion Time	Query Time	False Positive	Memory Usage (bytes)
Counting Bloom Filter	0.8316	0.1614	9.656×10^{-5}	164723744
Cuckoo Filter	5.7683	1.4442	0.0	61752944
Bloom Filter	0.7450	0.1580	9.656×10^{-5}	2577920

Table 27: Performance metrics for Load Factor = 0.9, Dataset Count = 82851.

- **Query Time:** Query times remain relatively stable for Bloom and Counting Bloom Filters. However, cuckoo filters show slight variations in query time depending on the load factor.
- **False Positive Rate:** For lower load factors (0.1), all filters exhibit zero false positives. As the load factor increases, the false positive rate also rises.
- **Memory Usage:** Memory usage increases with both dataset size and load factor. The cuckoo filter consumes the most memory across varying load factors.

5.2.3. Conclusion

Based on the tables above, the choice of filter depends on the application requirements:

- **Bloom Filter**: Ideal for applications prioritizing speed and minimal memory usage. It performs best for all dataset sizes and load factors in terms of insertion and query times. However, it could not support deletion operations.
- **Counting Bloom Filter**: Suitable for scenarios requiring deletion capabilities, despite its higher memory usage and slower speeds.
- **Cuckoo Filter**: Best suited for scenarios where moderate memory usage is acceptable, and speed is not critical. Its performance is consistent and better than Counting Bloom Filter, but the query time takes longer than Counting Bloom Filter when it comes to a bigger dataset inserted regardless of the load factor.

6. Filter-Tree Combinations

To continue our investigation of optimizing the system as a whole, we must consider various settings of each portion of the system. First, the username storage is passed through a filter, which can be one of the standard Bloom Filter, Counting Bloom Filter, and Cuckoo Filter. Second, the data storage should simulate a distributed system, implemented with

Consistent Hashing. The Consistent Hashing algorithm will be backed by one of three structures: a list for the ring, a binary search tree for server storage, and a red-black tree for server storage. As a baseline metric, we will use a set to store usernames and another set to store the data. The dataset for usernames will be the USA names data used prior, and the dataset to simulate server data storage will be a list of over 100 thousand Medium articles.

6.1. Evaluation Metrics

Standards for operation come from two distinct stages of a system's lifecycle. One is the overhead cost, and the other is the cost of operation. We will evaluate the overhead cost with the time necessary to construct the mock system, using a constant 10000 servers, and the cost of operation with the time and memory necessary to run through a series of insertion operations. For the control group, however, we note that a single set representing data storage fails to share the upsides of a distributed system. Much like the idea of putting all your eggs in one basket, storing everything in one large server leaves the system susceptible to complete failure. That said, we will explore the best Consistent Hashing structure by simulating server outages.

6.2. Experiment Results

The exact experiment results can be found in the code repository. We will examine the overhead costs and running costs in turn. The results are in a file named *insertion_log.txt*.

6.2.1. Overhead Costs

We first explore the time needed for different false positive rates given by each permutation of simple/counting bloom filter and false positive rate.

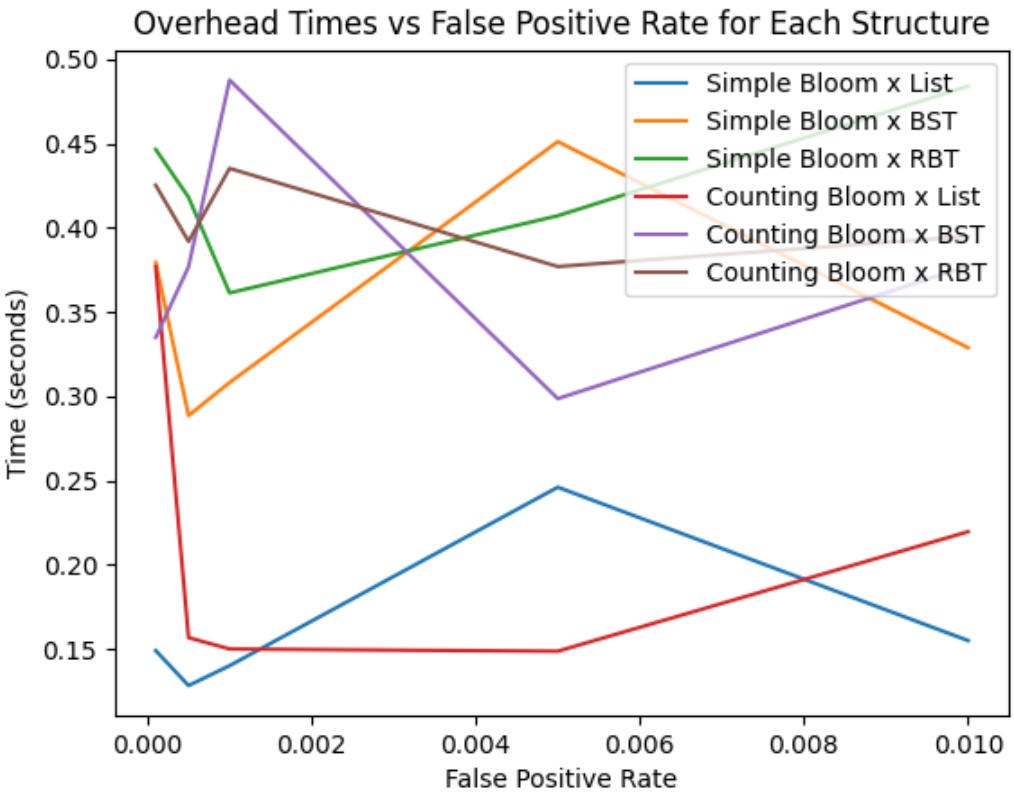


Figure 3: Time for Varying False Positive Rates

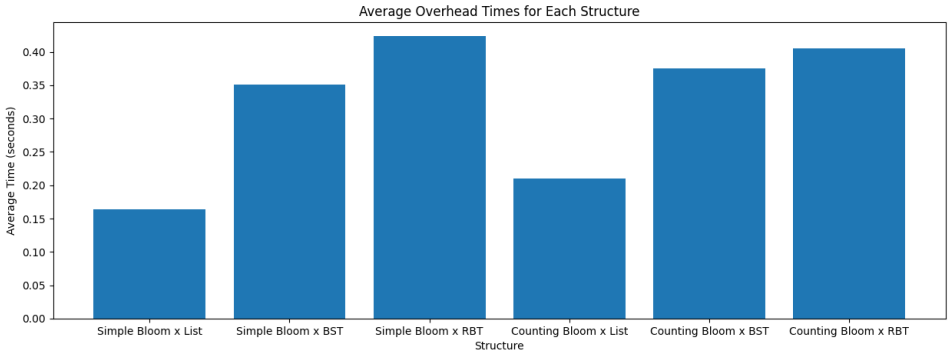


Figure 4: Average Time for Different Structures

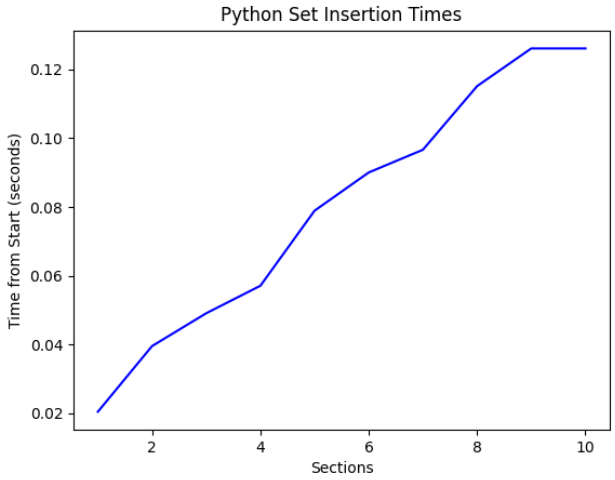


Figure 5: Insertion Time for Python Set

While the lines seem scattered, there is a very clear trend that we can distinguish. The red and blue lines are both list-backed consistent hashing algorithms, and these generally take the lowest amount of time across all the false positive rates. The trend for BST and RBT backed consistent hashing algorithms is much less clear, but it can be seen that consistent hashing algorithms backed by RBTs have a much more stable time overhead than those backed by BSTs.
These observations can be attributed to the data structures themselves. BSTs are much more sensitive to the order of insertions. For instance, in-order insertion will lead to the worst-case insertion and lookup times as the BST is essentially rendered to a glorified list. For BSTs to be as efficient as possible, we require an extremely robust hashing function that exponentially bisects the sample space as evenly as possible. This is the shortcoming of BSTs that RBTs attempt to address. RBTs employ the ability to rotate, and the rotations occur in an attempt to keep the tree height roughly balanced. This is why RBTs have worst-case insertion and lookup of $O(log(n))$ while BSTs have worst-case insertion and lookup of $O(n)$. Due to the overhead incurred by rotations, however, we see that RBTs have an overall higher average time needed.

6.2.2. Insertion Times

When running the benchmark, we partition the data into 10 different sections to see how the time and memory usage accumulates.

The first thing we present is the baseline metric (figure 5), namely the time needed to insert all the usernames and article names in Python sets. The total time required for insertion is around 0.125 seconds, and there appears to be three significant "humps" in the line. This can be attributed to hashtable resizing, which will become more apparent with the memory analysis.

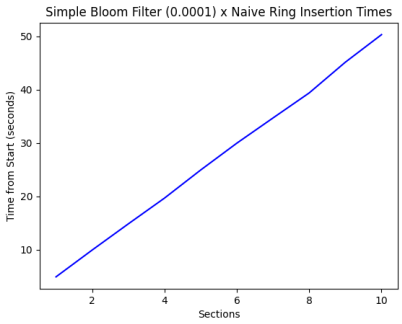


Figure 6: Insertion Time with Simple Bloom Filter (FPR=0.0001) and List Backing

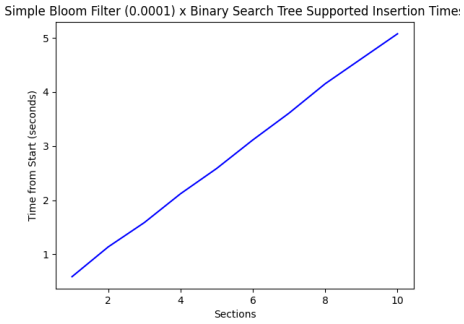


Figure 7: Insertion Time with Simple Bloom Filter (FPR=0.0001) and BST Backing

Let us then analyze the effect of the structure backing the consistent hashing algorithm (figures 6 through 8).

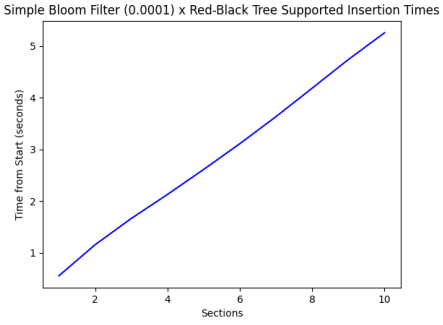


Figure 8: Insertion Time with Simple Bloom Filter (FPR=0.0001) and RBT Backing

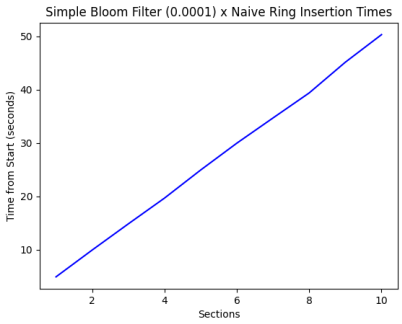


Figure 9: Insertion Time with Simple Bloom Filter (FPR=0.0001) and List Backing

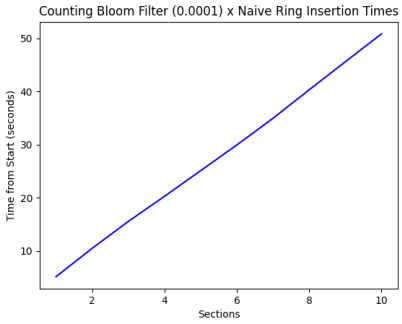


Figure 10: Insertion Time with Counting Bloom Filter (FPR=0.0001) and List Backing

There is a very consistent, linear growth rate in time, meaning the insertions occur at nearly the same rate throughout. However, while both BST and RBT backed consistent hashing algorithms finish at around 5 seconds, the list-backed algorithm falls at 50 seconds. The combination of the prior observations leave us to conclude that the server lookup time is the factor that makes the difference, and that despite the lower overhead observed prior, the insertion time for the list-backed algorithm takes a massive toll, taking ten times the amount of time seen with its peers.

Let us proceed to investigate the effect that the filter has on the insertion times (figures 9 through 11).

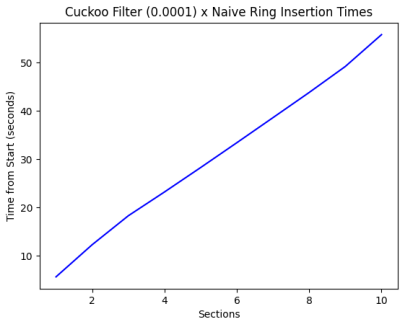


Figure 11: Insertion Time with Cuckoo Filter and List Backing

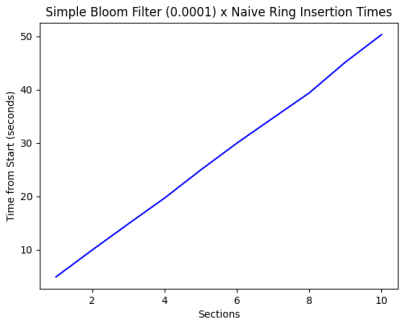


Figure 12: Insertion Time with Simple Bloom Filter (FPR=0.0001) and List Backing

Surprisingly, the simple and counting bloom filter results are nearly identical. The Cuckoo filter, meanwhile, has a slightly higher but still comparable insertion time than the others. Perhaps the false positive rate can illustrate a greater difference. *Note:* the "0.0001" in the title of the plot for the simple and counting bloom filters represent the false positive rate but, in the cuckoo filter, is meaningless and simply a placeholder for the code. Cuckoo filter parameters do not include a false positive rate, and we instead use the optimal parameters from the cuckoo filter analysis above.

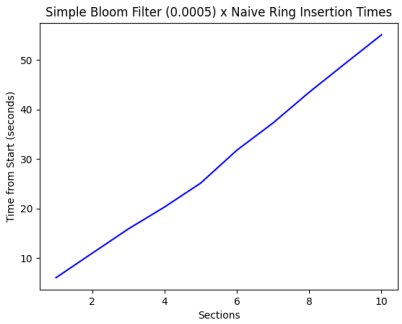


Figure 13: Insertion Time with Simple Bloom Filter (FPR=0.0005) and List Backing

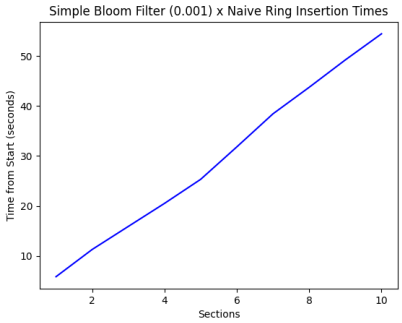


Figure 14: Insertion Time with Simple Bloom Filter (FPR=0.001) and List Backing

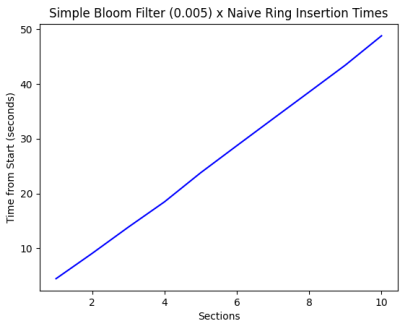


Figure 15: Insertion Time with Simple Bloom Filter (FPR=0.005) and List Backing

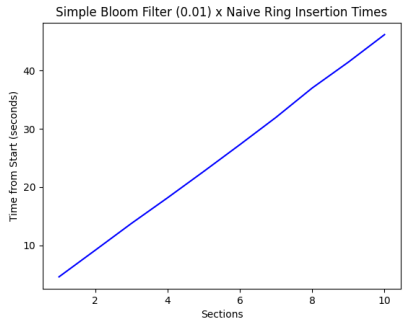


Figure 16: Insertion Time with Simple Bloom Filter (FPR=0.01) and List Backing

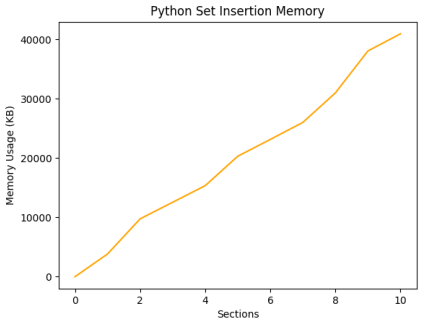


Figure 17: Insertion Memory for Python Set

As seen in figures 12 through 16, the false positive rate appears to have no impact at all on the insertion times, though higher false positive rates demonstrate slightly shorter insertion times. This can be attributed to a combination of a fast hash function and the logarithmic relationship between the number of hash functions, the size of the filter, and the false positive rate. These factors make hashing very efficient and thus keep the insertion times relative constant despite stricter false positive rates.
Conclusion: insertion times take a drastic dip when the consistent hashing algorithm is list-backed. Meanwhile, the filter type and false positive rate bear little to no impact on. Regardless, Python sets have significantly lower times than our filters.

6.2.3. Insertion Memory

Now, we will evaluate the memory tradeoffs, using the same batches of experiments as those in the insertion time evaluation.

Figure 17 is the baseline memory using a Python set.

Figures 18 through 20 are for different consistent hashing algorithm backings.

Figures 21 through 23 are for different filter types.

Figures 24 through 28 are for different false positive rates.

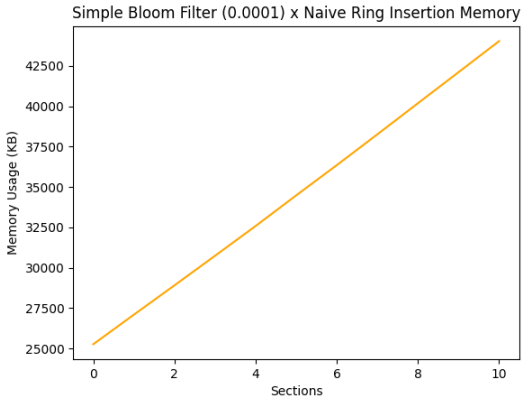


Figure 18: Insertion Memory with Simple Bloom Filter (FPR=0.0001) and List Backing

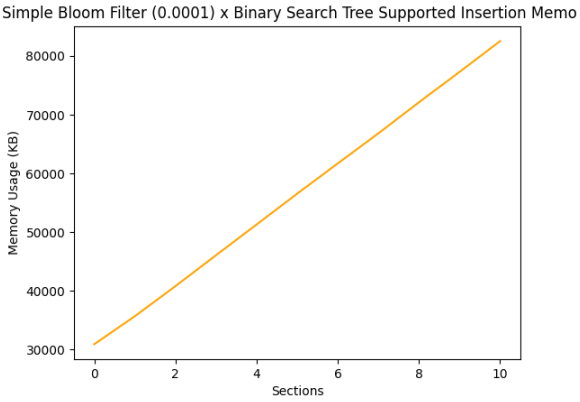


Figure 19: Insertion Memory with Simple Bloom Filter (FPR=0.0001) and BST Backing

Beginning with the baseline memory usage, we can see the same "humps" as in the insertion times graph. Between the list-backed consistent hashing algorithm and the two tree-backed ones, there is a significant difference. The trees take up significantly more memory, growing two times faster than the list-backed algorithm. From the filter perspective, all the memory usages are relatively the same, with the counting bloom filter using the most memory and the simple bloom filter using the least memory. Furthermore, the changes in false positive rate brought no difference in memory usage when seeing the data on the current scale.

Conclusion: There is very little tradeoff for using a lower false positive rate with a large dataset. The consistent hashing algorithm, however, can take a massive toll on the memory due to storing servers as nodes. Of course, having less servers will help us reduce the memory by cutting down the tree height for tree-backed implementations. Unlike with insertion times, the insertion memory for the baseline Python set is slightly less, but in the same relative range, as the other implementations.

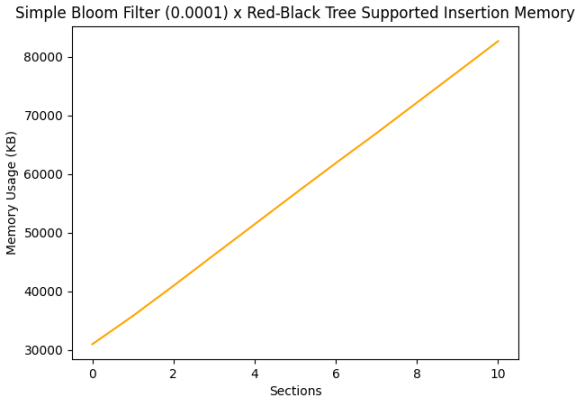


Figure 20: Insertion Memory with Simple Bloom Filter (FPR=0.0001) and RBT Backing

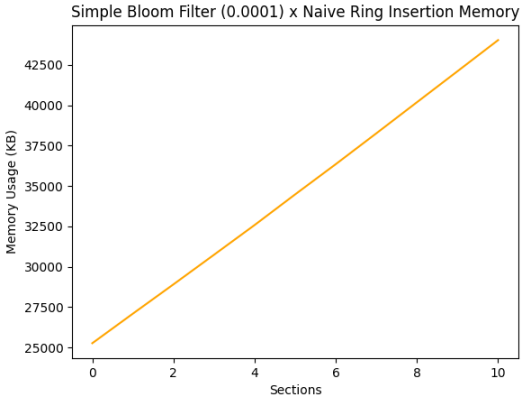


Figure 21: Insertion Memory with Simple Bloom Filter (FPR=0.0001) and List Backing

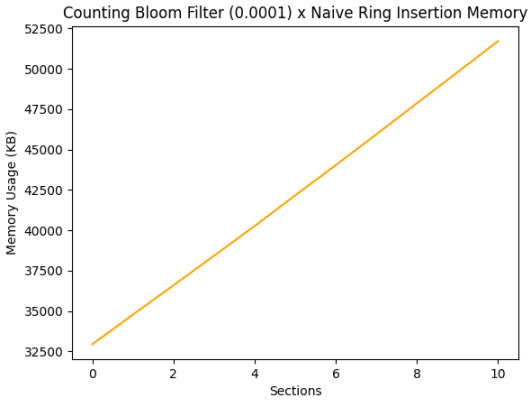


Figure 22: Insertion Memory with Counting Bloom Filter (FPR=0.0001) and List Backing

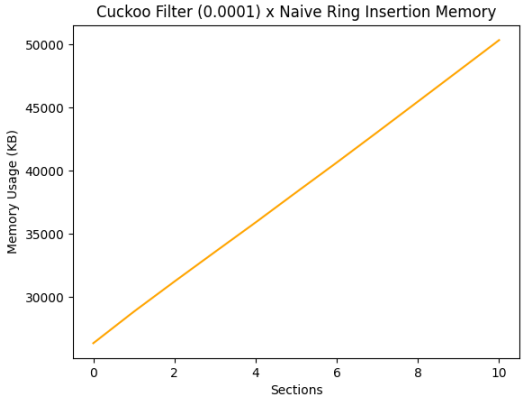


Figure 23: Insertion Memory with Cuckoo Filter and List Backing

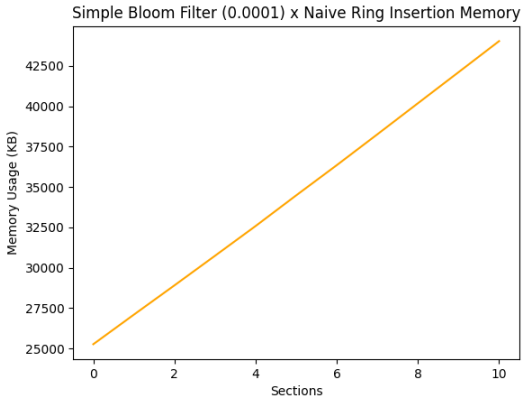


Figure 24: Insertion Memory with Simple Bloom Filter (FPR=0.0001) and List Backing

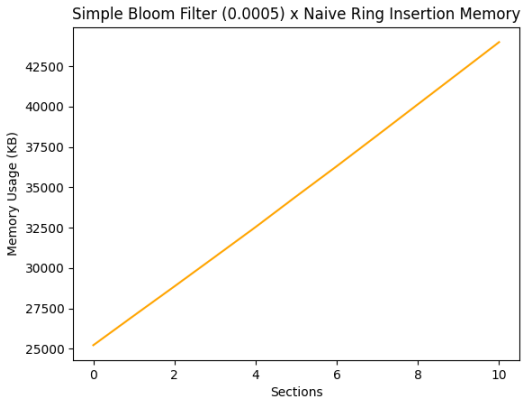


Figure 25: Insertion Memory with Simple Bloom Filter (FPR=0.0005) and List Backing

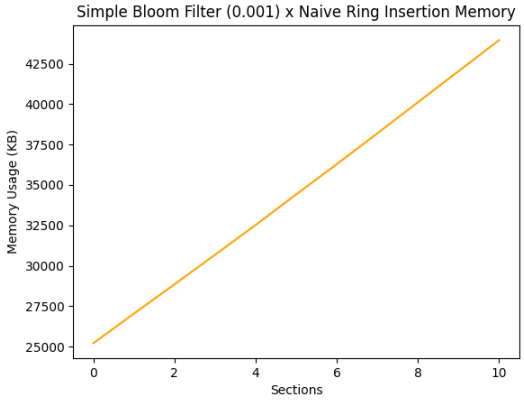


Figure 26: Insertion Memory with Simple Bloom Filter (FPR=0.001) and List Backing

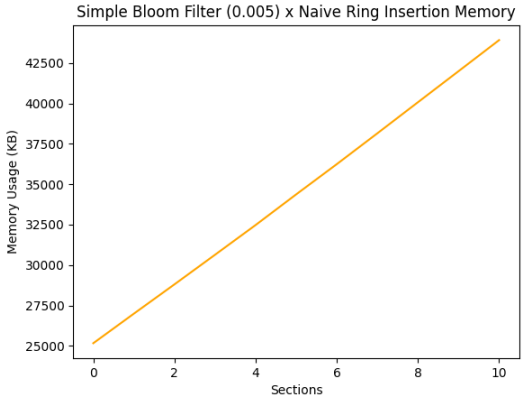


Figure 27: Insertion Memory with Simple Bloom Filter (FPR=0.005) and List Backing

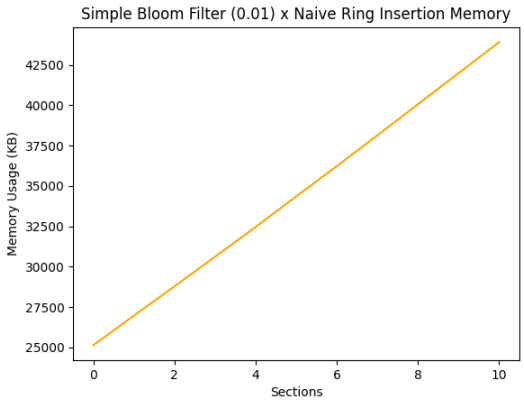


Figure 28: Insertion Memory with Simple Bloom Filter (FPR=0.01) and List Backing

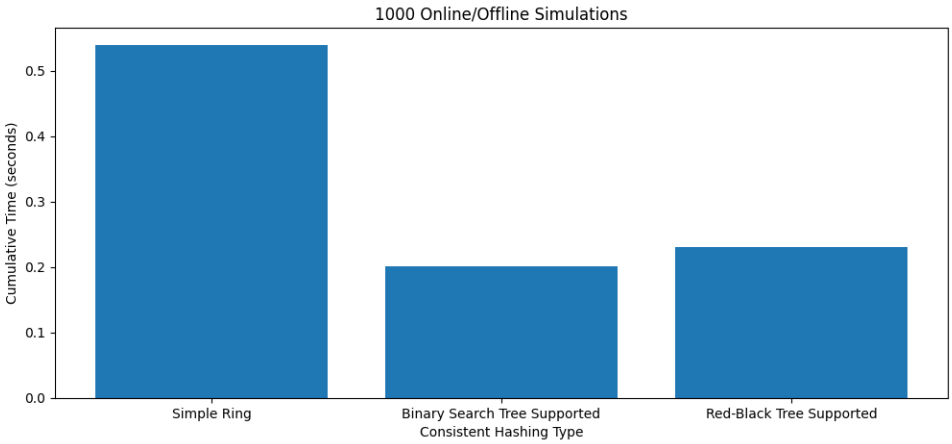


Figure 29: Simulation of 1000 Online/Offline Events

6.2.4. Insertion Conclusions

Of all of the hybrid systems tested, disregarding the baseline, we conclude that tree-backed consistent hashing leads to the best insertion times at the cost of significant memory. The type of filter to use truly depends on the use case, as does the false positive rate. A simple bloom filter can process insertion just as well as the other filters but likes a critical feature of removing items. Meanwhile, lowering the false positive rate leads to negligible insertion time and memory differences when processing systems at scale in our current context.

Having seen that Python sets perform better both in time and memory compared to our systems, the benefits of using these structures comes under scrutiny, but we must duly note the importance of trading off certain resources in favor of developing a better and more robust overall system. We can take a glimpse at what that means with an analysis of the benefits of distributed systems, specifically consistent hashing.

6.2.5. Power of Consistent Hashing

As mentioned before, having redundancy in a system is very important, especially with the reliance that we have on computers and data in the current world. Distributed systems must then have efficient way of handling the massive amounts of data flowing through. We mocked 1000 simulations of taking a server offline and putting it back online, shown in figure 29.

Even with a list-backed consistent hashing implementation, the total cost of time for redundancy is less than a second for the dataset of more than 100,000 rows. The tree-backed implementations can do the same much more efficiently, totaling under around a quarter of a second.

Figure 30 further demonstrates this point with a simulation of random events. We placed a probability of 20% server online/offline event for each time step, for a total of 1000 time steps.

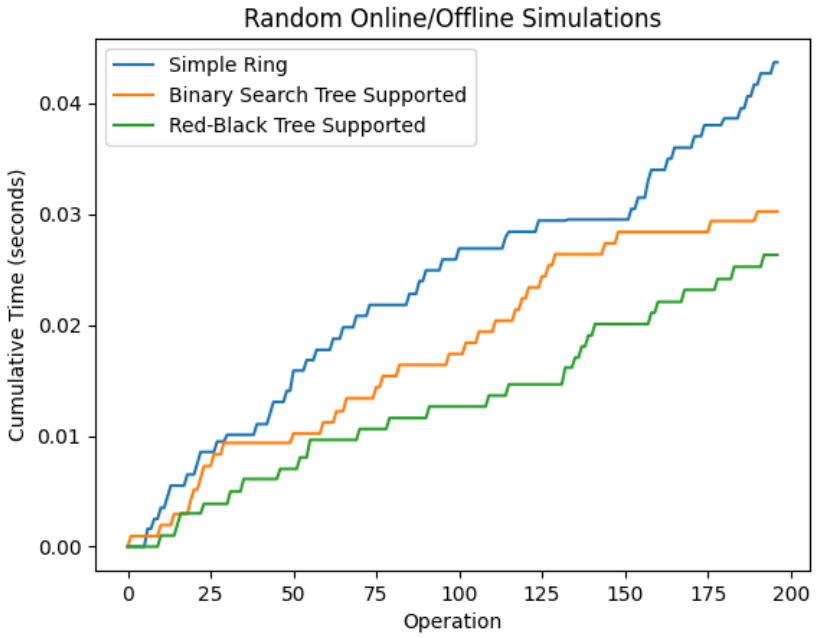


Figure 30: Simulation of Online/Offline Events for 1000 Time Steps

Here, we see that the two tree-based implementation once again outperforms the list-based implementation, but even so, around 200 operations took at most 0.045 seconds. The No Free Lunch Theorem is at full display here as we clearly see that efficiently supporting redundancy comes at severe memory costs, but the capabilities of a distributed computing system, which can not be mocked on a hashtable, far exceeds those of one that keeps all data on a single server.

7. Discussion of Results

In summary, we have explored each of the filters, systems composed of filters for username storage and consistent hashing for data storage, and the importance of consistent hashing and distributed systems. Of all filters, bloom filters have the best insertion and lookup times, but due to its lack of removal capabilities, a bloom filter is not ideal for the username-data storage system if we hope to have a feature of deleting usernames via deleting user accounts. When it comes to large datasets, counting bloom filters outperform cuckoo filters. From insertion time and memory analysis, we conclude that a lower false positive rate is not very costly. On the other hand, consistent hashing implemented with BSTs and RBTs are the best choices and comparable in insertion times but consume large amounts of memory. In modern computing contexts, time is much more valuable than memory, so we can state that BSTs and RBTs are more suitable in consistent hashing algorithms, with RBTs being a safer choice due to its height balance invariant. Now, we return to our original hypothesis, where we believed that the combination of a cuckoo filter and a consistent hashing using a red-black tree will be optimal. We now believe that after thorough testing of the components and

the system as whole, a counting bloom filter will work better and should still be combined with a consistent hashing with red-black tree.