University of New South Wales

COMP4121 Project

An exploration of Bloom filters and its place amongst hash tables

Author: **Bao-Du Bui**

November 2020

Background

Bloom filters are a data structure which allows you to test whether an element exists in a set, with lower memory usage and better access times than other hash table implementations. It is probabilistic, and while it can guarantee negative matches, there is a slight chance it returns a false positive match. Through clever mathematical assumptions, we can produce constraints to minimise the chance of a false positive.

1.1 Description

Let there be a set of elements N, and we wish to store each element $e \in N$ in the set F. To do this, we introduce the set K which has k number of hash functions satisfying the following property:

$$\forall (K_i, K_j) \in K,\ K_i \neq K_j,\ K_i(e) \neq K_j(e).$$

That is to say, no two distinct hash functions in K will map an element e to the same value. Therefore, we can assume that if we pass e through K, we will get k different values (Figure 1.1).

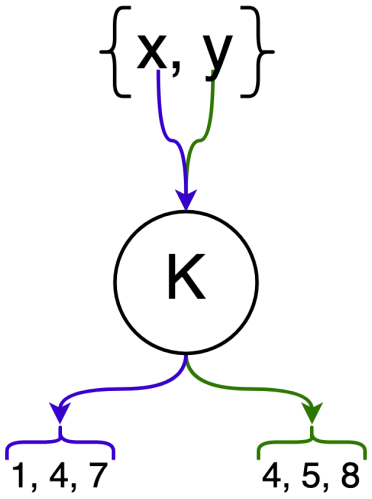

Figure 1.1: Elements x and y are hashed by k=3 hash functions.

Next, we introduce the bit array M which has m bits. This bit array is the underlying data structure that represents F, and we say:

$$e \in F \implies \forall K_i \in K,\ M[K_i(e)] = 1.$$

When we test set membership in F for every element e in N, there is a non-zero chance of a collision between two elements where $K_i(e_x) = K_j(e_y)$ (Figure 1.2).

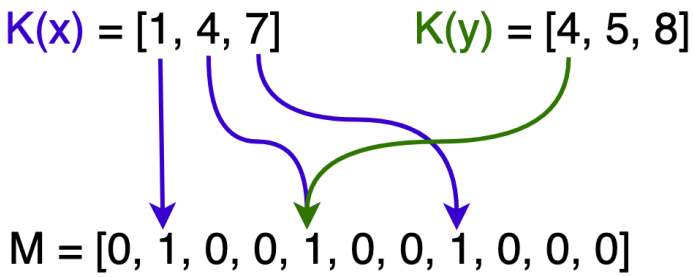

Figure 1.2: x is in F hence all of its hashed bits within M are set. Only one of y's hashed bits are set so it is not in F. x and y are also sharing a bit at $M[4]$.

As we hash more elements from N, more bits are set to 1 in M. Eventually, we find:

$$\text{while } e \notin F,\ M[K_i(e)] = 1\ \forall K_i \in K.$$

Furthermore, as we add more elements to F, the chance of a false positive occurring when we test set membership also increases (Figure 1.3). We call this chance ϵ.

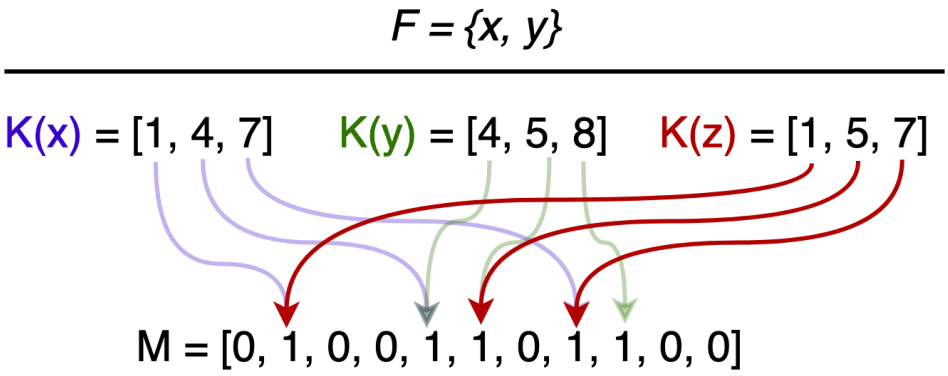

Figure 1.3: x and y are in F, z is not. However, z's hashed bits are all set, giving the (false) impression that z is in F.

Once an element is placed in F, it will remain there, as flipping bits to remove an element introduces the possibility of false negatives. We will show that there exists optimal parameters, k hash functions and m length bit array, to lower the false positive rate ϵ.

1.2 Proof

1.2.1 First attempt

Assume that a hash function in K maps to each array position with *equal probability*. The probability that a bit is not set by a hash function during the insertion of an element is:

$$1 - \frac{1}{m}.$$

The probability that every hash function in K leaves a certain bit at 0 will be

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

Thus, after inserting n elements, the probability that a bit is *still* 0 is

$$\left(1 - \frac{1}{m}\right)^{kn} \approx e^{-kn/m} = p,$$

and the probability that a bit is 1 after n insertions is

$$\left(1 - \left[1 - \frac{1}{m}\right]^{kn}\right) \approx (1 - p).$$

Next, we test set membership for an element NOT in the set. Following n insertions, each bit in the array has a chance of being set to 1 with the probability above. The probability that k bits are set to 1, which would lead to a false positive result for set membership, is often referred to as the error/false positive rate:

$$\epsilon = \left(1 - \left[1 - \frac{1}{m}\right]^{kn}\right)^k \approx (1 - p)^k.$$

There exists a major problem with this analysis, however. At the start we made an assumption that all bits would be set randomly and independently. This is not correct as we have established that *all* our hash functions in K will not hash an element to the same array position. For example, if we have hashed an element $k - 1$ times into $k - 1$ different positions, then the remaining $(m - (k - 1))$ bits in the array have a higher chance of being hashed to. Concretely, the k bit array positions for each element are in fact *dependent*.

We now look for a proof that makes no assumptions of independence.

1.2.2 Another try using Poisson approximations

Consider a "balls and bins" scenario where each throw of a ball into a bin is equivalent to hashing an element to an array position.

We have the following 2 cases:

- **Exact case**: n balls are thrown into m bins independently and uniformly at random
- **Poisson case**: number of balls in each bin are taken to be independent Poisson random variables with an expected value

We'll be using the following corollaries from Mitzenmacher and Upfal's book [6].

Corollary 4.6: Let $X_1, ..., X_n$ be independent Poisson trials such that $P(X_i = 1) = p_i$. Let $X = \sum_{i=1}^{n}$ and $\mu = E(X)$. For $0 < \delta < 1$. [p. 71]

$$P\left(|X - \mu| \geq \delta\mu\right) \leq 2\exp\left(-\frac{\mu\delta^2}{3}\right)$$

Corollary 5.9: any event that takes place with probability p in the Poisson case takes place with probability at most $pe\sqrt{n}$ in the exact case. [p. 109]

Each bin corresponds to an array position and thus a bit being set to 0 is equivalent to an empty bin in our scenario. The fraction of bits being set to 0 after n insertions is therefore equivalent to the fraction of empty bins after kn balls have been thrown into m bins.

We define X as the number of empty bins after the balls have been thrown into n bins, such that

$$X = \sum_{i=1}^{n} X_i,$$

$$\text{where } X_i = \begin{cases} 1 & \text{if bin is empty} \\ 0 & \text{otherwise} \end{cases}$$

then we can define

$$p' = \left(1 - \frac{1}{m}\right)^{kn},$$

$$E(X) = mp'.$$

In the Poisson case, each bin can be thought of as an independent Poisson random variable with expected value p'. Therefore, we can apply **corollary 4.6** and $E(X) = mp'$ to obtain the following:

$$P\left(\left|X - mp'\right| \geq \delta mp'\right) \leq 2\exp\left(-\frac{mp'\delta^2}{3}\right)$$

Let $\delta = \beta/p'$, choose small β

$$\therefore\ P\left(\left|X - mp'\right| \geq \beta m\right) \leq 2\exp\left(-\frac{m\beta^2}{3p'}\right)$$

We then apply **corollary 5.9** to obtain

$$\begin{aligned} P\left(\left|X - mp'\right| \geq \beta m\right) &\leq 2e\sqrt{kn}\exp\left(-\frac{m\beta^2}{3p'}\right) \\ &\leq 0.000001 \text{ when } m \text{ sufficiently large.} \end{aligned}$$

Essentially, taking the probability of an event using a Poisson approximation for all of the bins and multiplying it by $e\sqrt{kn}$ gives an upper bound for the probability of the event when kn balls are thrown into m bins (the exact case where events are independent).

This result tells us that when m is sufficiently large, the fraction of empty bins X/m is *very* close to p'. And since $p' \approx p$ we can use p to continue predicting actual performance.

1.2.3 Optimal k

The false positive rate is $\epsilon = (1-p)^k$ and we look for a k that minimizes ϵ. Rearranging ϵ gives us

$$\begin{aligned} \epsilon &= (1-p)^k \\ &= \exp\left(\ln\left([1-p]^k\right)\right) \\ &= \exp\left(k\ln\left([1-p]\right)\right) \\ &= \exp\left(k\ln\left(\left[1-e^{-kn/m}\right]\right)\right). \end{aligned}$$

If we let $g = k\ln\left(1-e^{-kn/m}\right)$ so that $\epsilon = e^g$, then minimizing the false positive ϵ is equivalent to minimizing g with respect to k. We have

$$\frac{dg}{dk} = \ln\left(1-e^{-\frac{nk}{m}}\right) + \frac{kn\cdot e^{-\frac{nk}{m}}}{m\left(1-e^{-\frac{nk}{m}}\right)}.$$

Solving this derivative when it is 0 and finding the global minimum gives us

$$k = \frac{m}{n}\ln\left(2\right).$$

1.2.4 Optimal m

To find an optimal length for our bit-array we substitute $k = \frac{m}{n} \ln(2)$ into our false positive equation and get

$$\begin{aligned} \epsilon &= \left(1 - e^{-\ln 2}\right)^{\frac{m}{n} \ln 2} \\ \ln \epsilon &= \frac{m}{n} \ln(2) \ln\left(1 - e^{-\ln 2}\right) \\ &= \frac{m}{n} \ln(2) \ln \frac{1}{2} \\ &= -\frac{m}{n} \ln(2)^2 \\ \therefore m &= \frac{-n \ln \epsilon}{\ln(2)^2} \end{aligned}$$

This effectively leaves ϵ as the only unknown variable left. However, when we consider the Bloom filter in a practical context, we will most likely have a false positive rate in mind, and can treat it as a constant.

Implementation

2.1 Overview

We will be comparing the Bloom filter against 4 popular and efficient implementations of hash tables:

- Google Dense Hash Set [7]
- Google Sparse Hash Set [7]
- TSL Robin Set [8]
- STD Unordered Set [1]

Implementations were compared based on time performance (insert, read) and memory performance (inserts). Currently, only small strings (15 characters) and medium strings (50 characters) are used for input, with up to $n = 3 \times 10^6$ elements for each test. Each test was performed 5 times for each implementation and an average-of-5 was used in the final table/graph. The false positive rate is set to 0.01.

Benchmarking was done using gcc's C++ compiler and the following command was run to compile: g++ -Iinclude -std=c++11 -O3. In addition, the tests were performed on a computer with the following specs:

- AMD Ryzen 5 2600 3.4GHz 6 core
- 8GB DDR4-2666 CL19

2.2 Small string (15 bytes)

For small strings, 15 alphanumeric characters composed our string (and 1 extra for null terminator).

2.2.1 Insert small string

Refer to table and figure A.1 for data and a visualisation.

Before the test, we generate a vector of n small strings and then insert each string as an entry into the sets, measuring the performance of said insert operation. The Bloom filter's only overhead during insertion is setting bits to 1. We will consider the role the hash function plays later on. Every implementation performed as expected, including sparsehash, which prides itself on (relatively) low memory overhead (which we will explore later).

2.2.2 Read small string

Refer to table and figure A.2 for data and a visualisation.

Before the test, we generate a vector of n small strings and pre-load the strings into the hash tables. We then traverse the same vector of small strings, testing set membership and timing said read operation.

From our results, the sparsehash and std implementation perform almost equally, but not quite as fast as our other implementations. While the Bloom filter was the clear winner in our insertion battle, its victory is marginal when it comes to read operations.

2.2.3 Memory usage - small string

Refer to table and figure A.3 for data and a visualisation.

This test is conducted in the same manner as the insertion test, except we measure memory usage before and after the n insertions. Because the Bloom filter's memory overhead is determined before operations, we will consider them in our data for a fair comparison.

This is where the Bloom filter stands out, as a *space efficient* data structure. sparsehash, a memory efficient hash table in its own right, can only manage a 216MB overhead with 3 million elements, while our Bloom filter remains under 4MB, that's 1% of sparsehash' memory usage! Amazing!

Two implementations have rather funky lines on the graph, densehash and robin_set, as they grow in what's called a "power-of-2" growth policy, which we will explore later on. One of the main reasons why both of these implementations were able to compete with the Bloom filter on read operations is because of this growth policy.

2.3 String (50 bytes)

For strings, 50 alphanumeric characters composed our string (and 1 extra for null terminator).

2.3.1 Insert string

Refer to table and figure A.4 for data and a visualisation.

This test was conducted in the same manner as the *insert small string* test, with the only difference being the usage of 50 byte strings instead.

When handling bigger strings, the other implementations see a very obvious increase to the time taken to insert all n elements. The Bloom filter, however, takes nearly the exact same amount of time (increasing slightly due to the hash function having to hash a bigger string), and remains under the 500ms mark. The size of the element makes no difference to the Bloom filter bookkeeping overhead.

In the graph, the sparsehash time has been excluded due to the *exorbitant* time taken to insert all n elements, dwarfing all other implementations. The measured times are, however, left in table A.4.

2.3.2 Read string

Refer to table and figure A.5 for data and a visualisation.

This text was conducted in the same manner as the *read small string* test, with the only difference being the usage of 50 byte strings instead.

Read operations, again, prove to be the great equaliser, and we see a very tight cluster of times except for sparsehash, which separates itself from the pack. In our first comparison of read operations in 2.2.2, sparsehash and the std implementation were equal in performance. With bigger strings, however, the std implementation is the clear winner.

2.3.3 Memory usage - string

Refer to table and figure A.6 for data and a visualisation.

This test was conducted in the same manner as the *small string memory usage* test, with a bigger string (50 bytes).

Once again, the Bloom filter performs spectacularly. In fact, it uses the *same* memory overhead as it did in the 50 byte test at 2.2.3. Earlier we showed that m, the amount of bits in our bit array, is conditioned n. Meaning even if the elements we were hashing got larger, our memory overhead would remain the same. This is quite useful as it allows us to hash relatively long strings (e.g. URLs) with no extra space used compared to a smaller string.

It is clear that the reason sparsehash performs so *badly* in our insertions is because it takes extra steps to reduce space usage. We also see densehash and robin_set repeating the same behaviour in the earlier test, utilising the "power-of-2" growth policy which will be explored shortly.

2.4 False positives - is the math accurate?

We now test the false positive rate for the Bloom filter. The methodology is as follows: insert n strings into the Bloom filter, generate n new strings NOT currently in the set, and check set membership. We then count the number of times a false positive occurs. In this test we will set $\epsilon = 0.01$.

In tables A.7 and A.8, we see that the false-positive rates for both small and normal strings are very, very close to the value we set, 0.01. We can safely work with Bloom filters with the assumption that false-positives only occur at a rate of ϵ.

2.5 Improving time performance

There are 3 areas which can be optimised in the Bloom filter:

- modulo operations when hashing
- random array accesses when setting/checking bits
- hash function

2.5.1 power-of-2 trick

In 2.2.3 and 2.3.3, where we looked at the memory performance of these implementations, we saw in figures A.3 and A.6, the memory of densehash and robin_set *doubled* every couple increments of n. This is due to a power-of-2 trick used to improve the overall speed of the algorithm, at the expense of irregular memory increases.

First, we have to look at the modulo operator (%). The modulo operation is used in the hashing function for every hash table insertion and read. But this operator is, in fact, computationally expensive (relative anyways).

Take the operation (a % b) where a is an element and b is the size of our hash table. If we can make assurances that b is a power of 2, then we can use the faster equivalent
(a % b) = (a & (b - 1)). Using bit wise tricks, we can reduce a tiny bit of time per element, which quickly scales if n sufficiently large.

So why doesn't the Bloom filter use that? In some cases the table was *bigger* than it was supposed to be, which compromises the space-efficient ideal of the Bloom filter. Increasing the size of the bit array means increasing m, and we no longer have guarantees of the false-positive rate for our number of keys n.

Some hash table implementations like densehash and robin_set use this trick to sacrifice perfect memory usage in place of decreased insert and read times. It provides potential and a time-efficient Bloom filter variant can thus be explored.

2.5.2 Spatial locality (or lack of)

Due to the *random* nature we map elements to different bits in our bit array, an element might have bits spread out across the bit array. The Bloom filter is thus unable to make use of "spacial locality" which states "if a particular storage location is referenced at a particular time, then it is likely that nearby memory locations will be referenced in the near future"[3].

The Bloom filter has to perform many random accesses on the bit array, and will experience cache misses more often than other hash tables that are able to make use of spacial locality. If we were to utilise this powerful optimization, we would have to sacrifice the randomness of our hashing functions, which compromises the Bloom filter as a whole.

2.5.3 Picking a suitable hash function

The hash function is where we can find the most time to improve on. Uses a slow hash function kn times for large n will negatively affect the time efficiency of our insert and read operations. Picking a suitable hash function is therefore paramount to the efficiency of the Bloom filter, since it has little overhead in the first place.

Different hash functions were explored, including djb2[2], MurmurHash2[4], and finally XXH3[5]. djb2 was quite inefficient and was quickly disregarded. MurmurHash2 provided good performance, but its simplicity was unable to stand up to even the standard library hash function when compiler optimisation was enabled. We look for a hash function that can make full use of the processor, and arrive at XXH3.

XXH3 utilises SSE2, an extended instruction set for processors which allow for vector operations. The use of this feature eludes some at first, but when we consider the fact that all hash functions do is repeatedly apply bit operations on multiple integers before combining them, SSE2 can help parellize bit operations and *significantly* improve time efficiency.

XXH3, is however, not fully portable, since some CPUs may not support this extended instruction set. In such cases, std::hash may suffice.

2.6 Example usage scenario

Consider a web browser that has the feature to alert users about suspicious URLs. When it spots a suspicious URL, it runs a full diagnostic check on the website first before allowing the user to proceed. Using a Bloom filter, we can create a set of suspicious URLs that takes up minimal space on a server or in memory. Every time a user visits a URL, the browser can preempt the user and check to see if the URL is currently in the set of suspicious URLs. If the site is indeed suspicious, the browser runs a diagnostic check. If the site is not suspicious but there is a false positive, then the browser still runs the diagnostic check. Even if false positives occur, this is a worthy trade-off as we have saved space and reduced the time it takes to check a URL.

Conclusion

In conclusion, Bloom filters present themselves as a powerful alternative to hash tables, promising extreme space saving measures at the cost of a small false positive rate. Each element utilises k bits of space and the Bloom filter has overall $O(m)$ space complexity, m is size of bit array. Furthermore, it has some low insert and read times at $O(k) = O(1)$ since the number of hash functions we use, k, is extraordinarily low, and so we can assume *constant* operations. Space and time complexity can be be improved with trade-offs, and there exists some promising variants as discussed in the report.

Appendix A

Tables and figures

n	std unordered set	google dense hash set	google sparse hash set	tsl robin set	bloom filter
250000	83	64	152	70	24
500000	231	141	358	155	49
750000	334	297	586	308	55
1000000	671	343	1094	330	107
1250000	663	534	1027	547	115
1500000	709	563	1358	595	115
1750000	872	586	1942	877	130
2000000	1442	736	2317	861	546
2250000	1415	1182	2459	1366	381
2500000	1401	1030	2393	1172	273
2750000	1503	1145	2703	1254	342
3000000	1670	1207	2949	1269	364

Table A.1: Insertion times (ms) for 15 byte strings

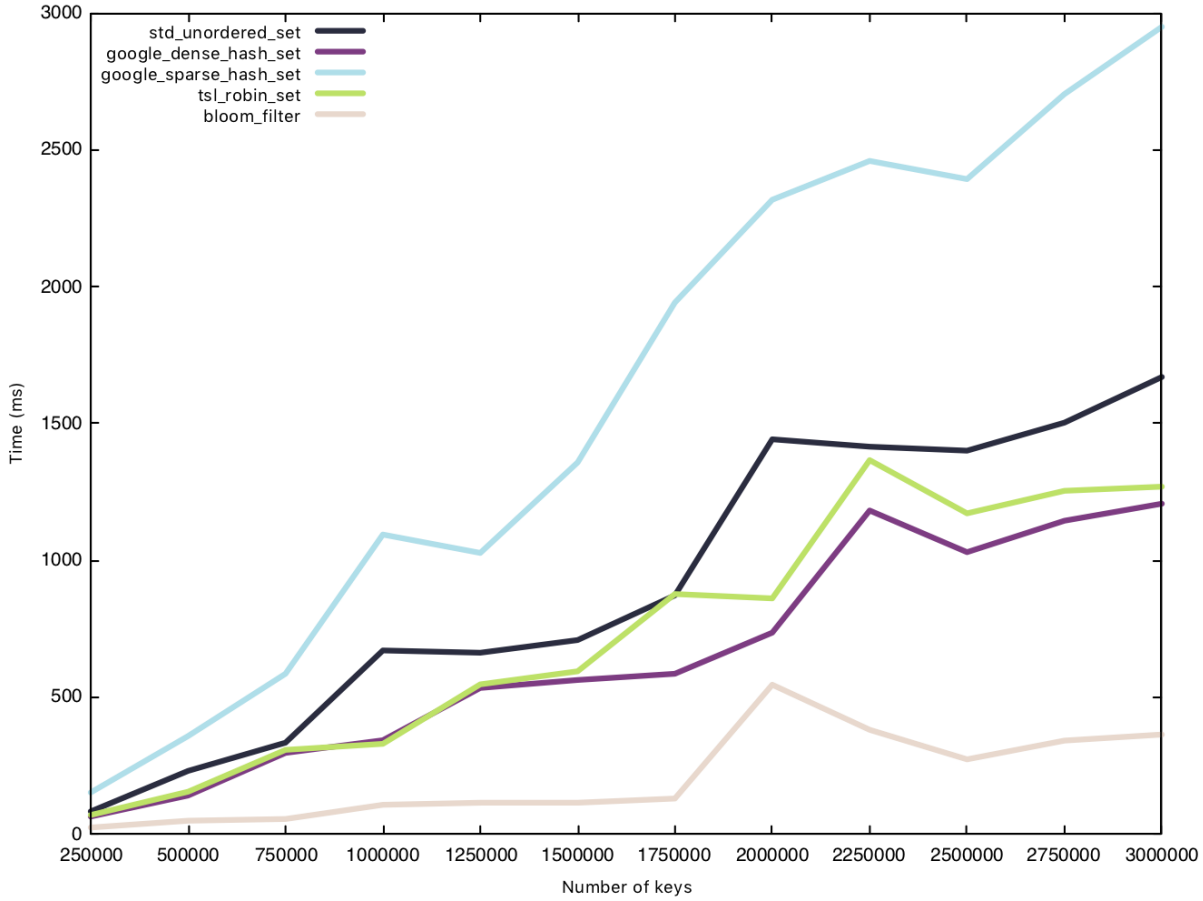

Figure A.1: Insertion times (ms) for 15 byte strings

n	std unordered set	google dense hash set	google sparse hash set	tsl robin set	bloom filter
250000	42	19	37	20	13
500000	86	42	80	43	31
750000	152	88	142	58	36
1000000	186	86	176	101	73
1250000	294	116	253	115	77
1500000	334	146	299	135	81
1750000	466	169	359	178	89
2000000	441	200	439	192	198
2250000	513	194	500	229	185
2500000	570	239	583	214	203
2750000	667	244	646	256	183
3000000	692	281	768	270	207

Table A.2: Read times (ms) for 15 byte strings

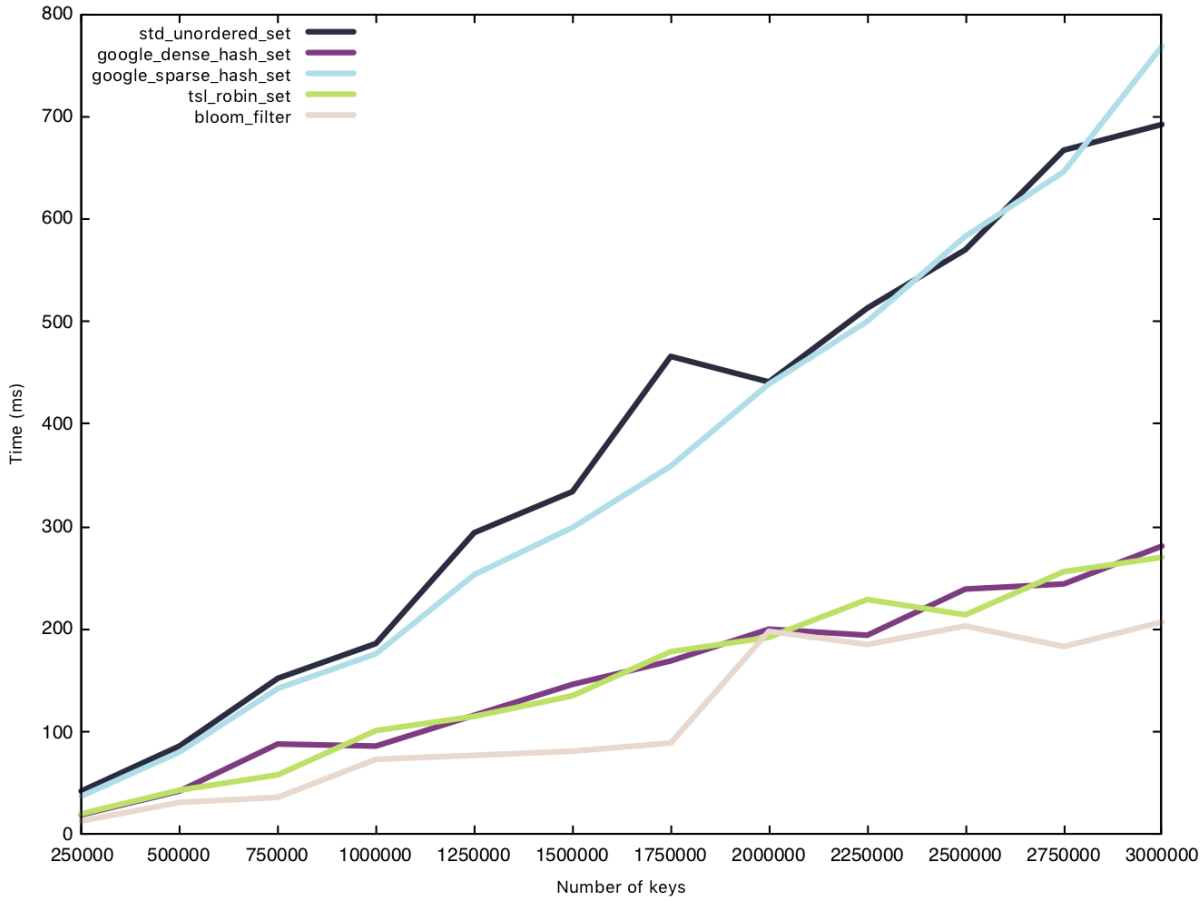

Figure A.2: Read times (ms) for 15 byte strings

n	std unordered set	google dense hash set	google sparse hash set	tsl robin set	bloom filter
250000	19112	24712	9341	30864	299
500000	38320	49288	18472	61580	599
750000	53940	98440	26472	123018	898
1000000	77788	98440	36696	123019	1198
1250000	92456	196744	44796	245901	1497
1500000	108080	196744	52684	245900	1797
1750000	123708	196744	62246	245897	2096
2000000	157944	196744	73098	245900	2396
2250000	169714	393352	81416	491658	2695
2500000	185340	393352	89193	491658	2995
2750000	200965	393352	97141	491661	3294
3000000	216592	393352	105004	491660	3594

Table A.3: Memory usage (Kilobytes) for 15 byte strings

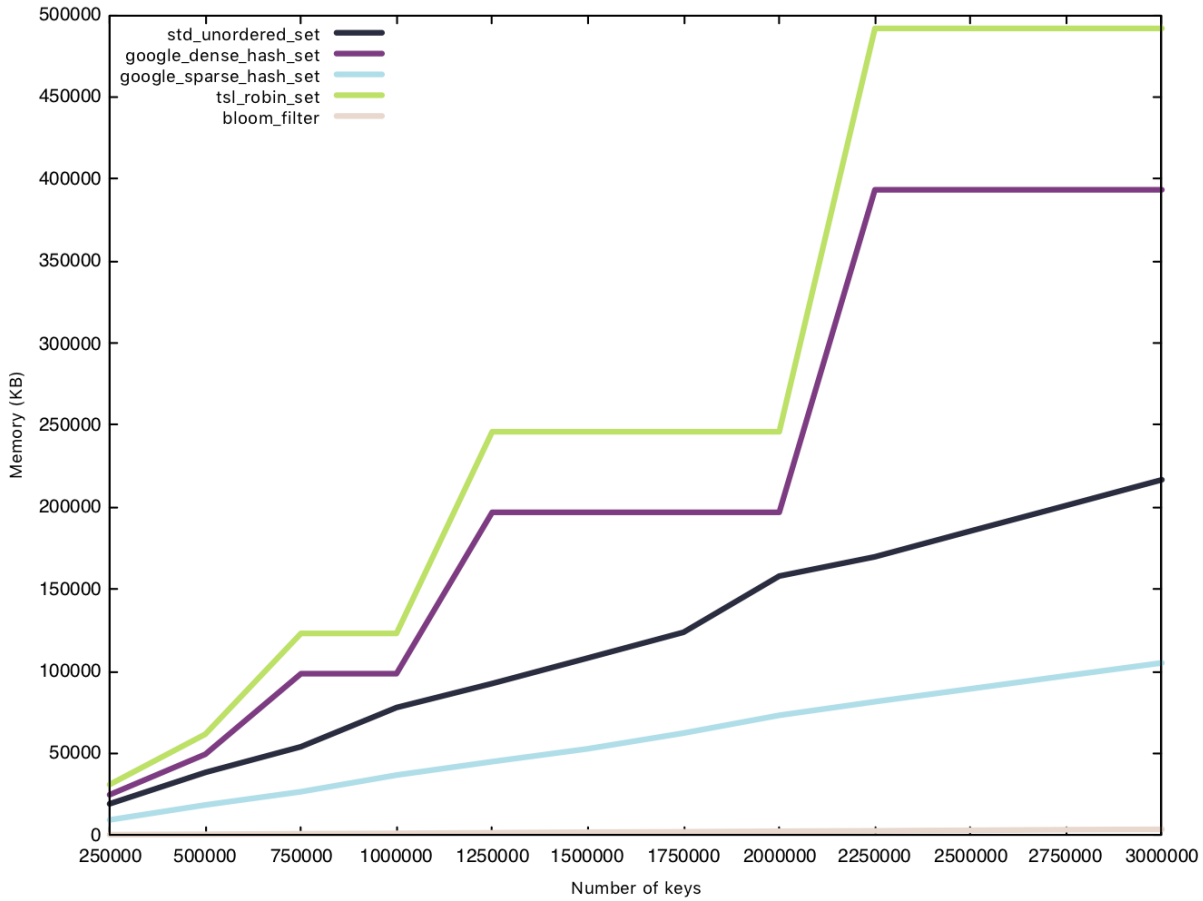

Figure A.3: Memory usage (Kilobytes) for 15 byte strings

n	std unordered set	google dense hash set	google sparse hash set	tsl robin set	bloom filter
250000	137	130	881	112	35
500000	333	319	2428	240	82
750000	439	549	2747	419	79
1000000	669	608	4006	448	149
1250000	785	1017	4806	715	165
1500000	914	1078	5874	791	166
1750000	1049	1169	7792	851	176
2000000	1425	1250	8529	928	389
2250000	1551	2130	9544	1416	399
2500000	1655	2121	10174	1635	425
2750000	1855	2206	11951	1573	341
3000000	1793	2116	12171	1669	487

Table A.4: Insertion times (ms) for strings (50 bytes)

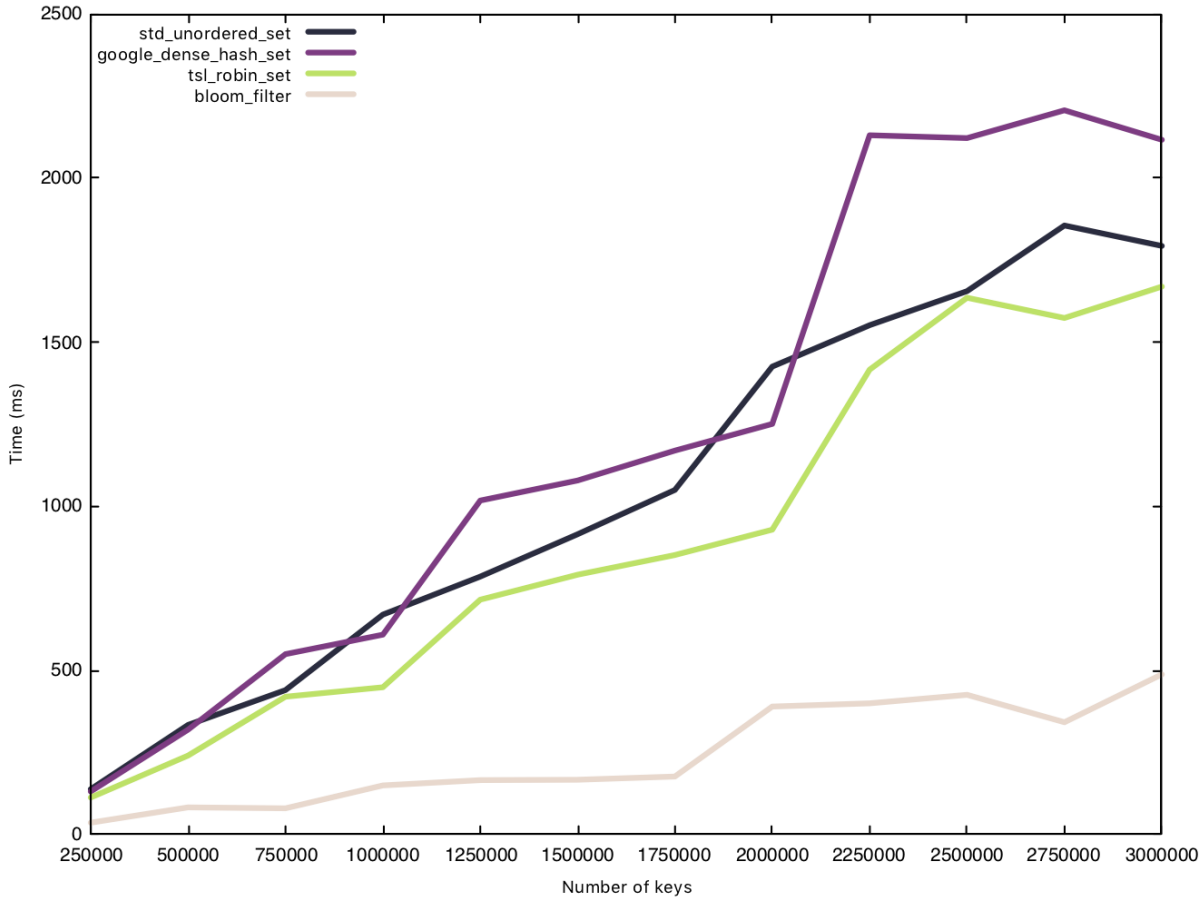

Figure A.4: Insertion times (ms) for 50 byte strings

n	std unordered set	google dense hash set	google sparse hash set	tsl robin set	bloom filter
250000	73	62	95	60	53
500000	151	137	207	127	114
750000	247	193	351	172	152
1000000	355	315	487	275	247
1250000	453	344	688	326	284
1500000	522	405	772	364	311
1750000	617	481	823	446	350
2000000	706	634	1028	549	557
2250000	746	616	1124	539	611
2500000	849	711	1283	621	666
2750000	964	788	1474	678	706
3000000	1076	867	1784	778	915

Table A.5: Read times (ms) for strings (50 bytes)

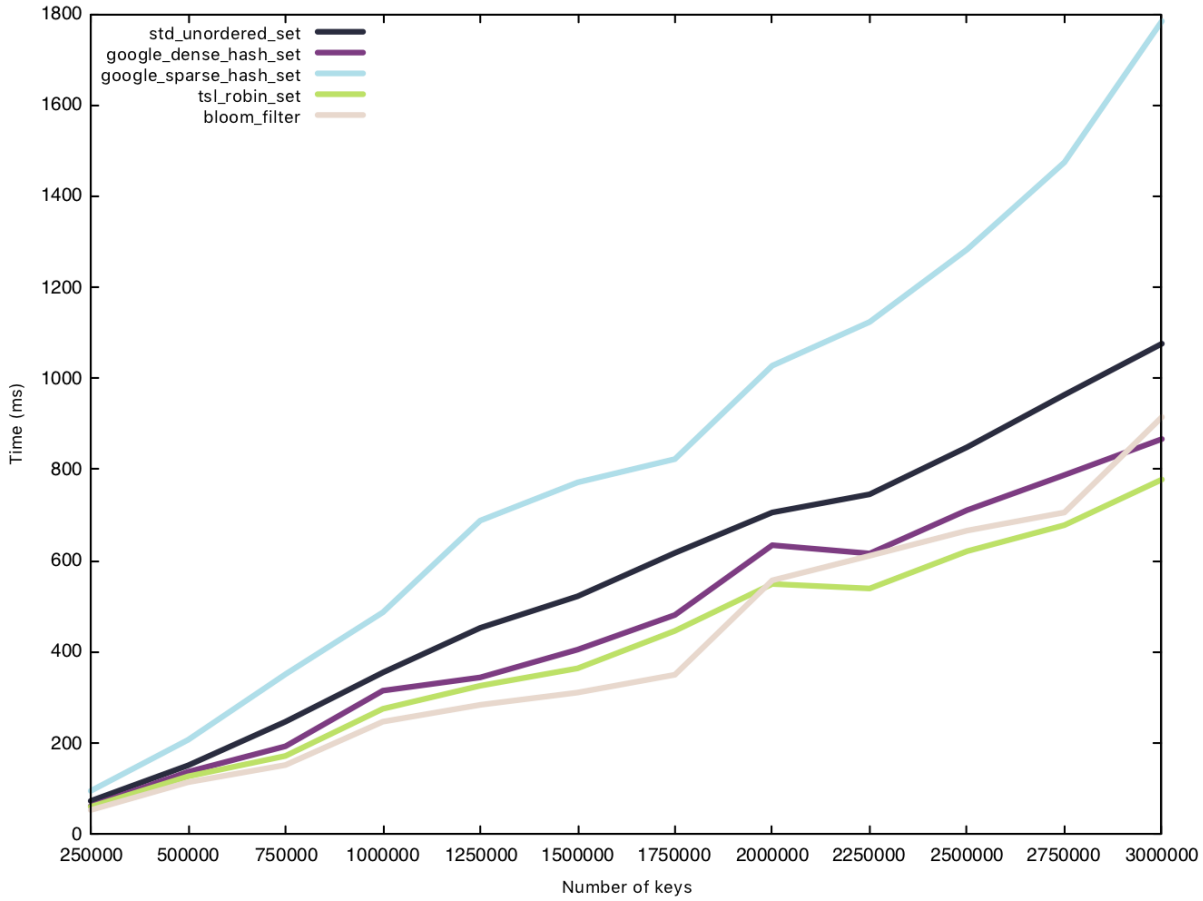

Figure A.5: Read times (ms) for 50 byte strings

n	std unordered set	google dense hash set	google sparse hash set	tsl robin set	bloom filter
250000	34736	40972	24447	38922	299
500000	69560	81932	48704	77828	599
750000	100812	163856	72675	155657	898
1000000	139336	163852	97212	155661	1198
1250000	170584	327697	121242	311306	1497
1500000	201840	327700	145080	311308	1797
1750000	233092	327700	173552	311309	2096
2000000	279101	327700	194220	311308	2396
2250000	310352	655389	218290	622608	2695
2500000	341604	655388	242256	622605	2995
2750000	372856	655388	266108	622611	3294
3000000	404108	655388	289833	622610	3594

Table A.6: Memory usage (KB) for 50 byte strings

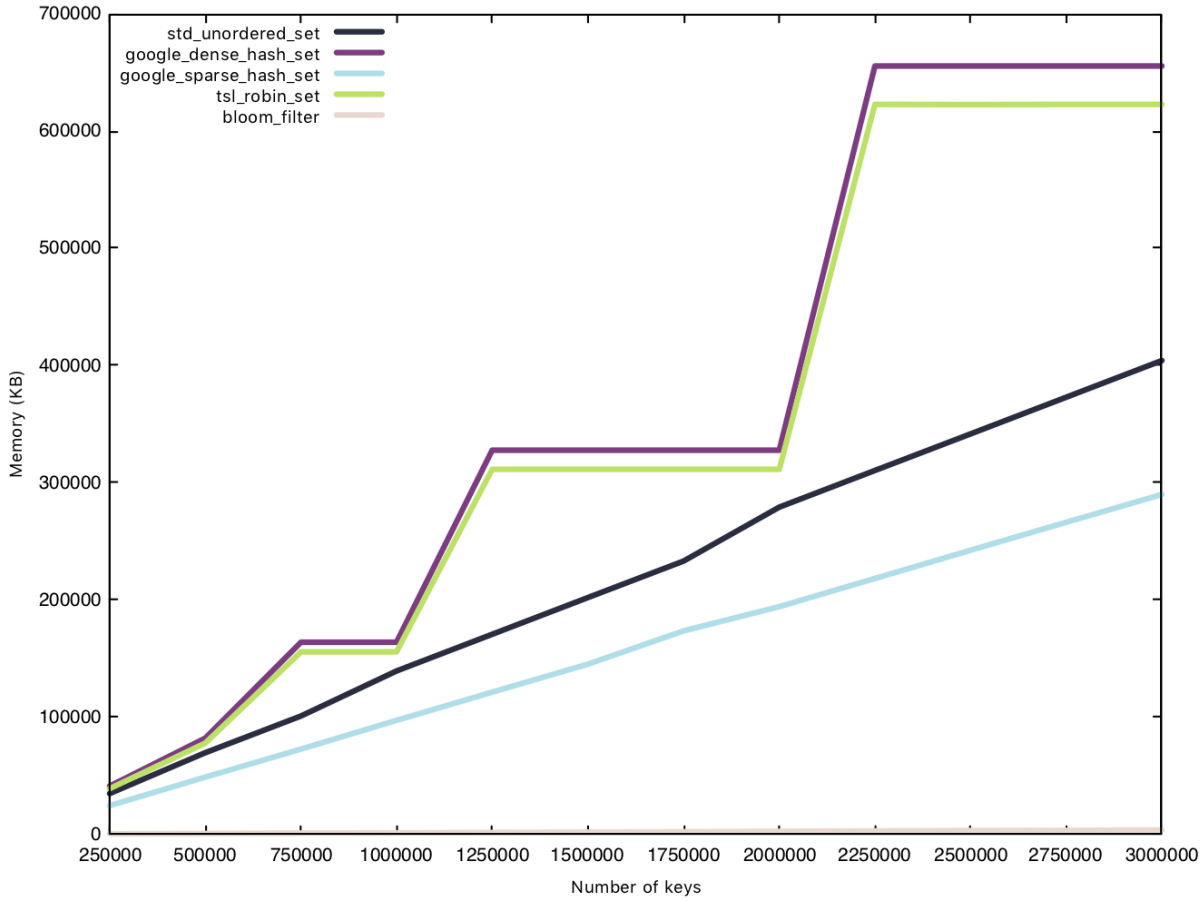

Figure A.6: Memory usage (KB) for 50 byte strings

n	false-positives	false-positive rate
250000	2526	0.010104
500000	4974	0.009949
750000	7527	0.010037
1000000	10078	0.010078
1250000	12633	0.010107
1500000	14982	0.009988
1750000	17639	0.010080
2000000	20165	0.010082
2250000	22709	0.010093
2500000	25027	0.010011
2750000	27600	0.010037
3000000	30153	0.010051

Table A.7: Small string (15 bytes) false positive rates for Bloom filter

n	false-positives	false-positive rate
250000	2544	0.010178
500000	5061	0.010124
750000	7462	0.009950
1000000	9967	0.009967
1250000	12526	0.010021
1500000	15093	0.010062
1750000	17569	0.010040
2000000	20035	0.010018
2250000	22621	0.010054
2500000	25053	0.010022
2750000	27531	0.010012
3000000	30062	0.010021

Table A.8: String (50 bytes) false positive rates for Bloom filter

Bibliography

[1] std::unordered_set. URL https://en.cppreference.com/w/cpp/container/unordered_set.

[2] URL http://www.cse.yorku.ca/~oz/hash.html.

[3] Locality of reference, Nov 2020. URL https://en.wikipedia.org/wiki/Locality_of_reference.

[4] Aappleby. aappleby/smhasher. URL https://github.com/aappleby/smhasher.

[5] Cyan4973. Cyan4973/xxhash. URL https://github.com/Cyan4973/xxHash.

[6] M. Mitzenmacher and E. Upfal. *Probability and computing: randomization and probabilistic techniques in algorithms and data analysis*. Cambridge University Press, 2018.

[7] Sparsehash. sparsehash/sparsehash. URL https://github.com/sparsehash/sparsehash.

[8] Tessil. Tessil/robin-map. URL https://github.com/Tessil/robin-map.