Bloom Filter Implementation

Aaron Gorenstein

June 30, 2024

Contents

1 Introduction

Formal computer science—including things like algorithms and data structures, computational complexity theory, as well as the more "formal" side of practical

things like compilers, databases, and operating systems—is all just so fascinating. In an academic settings, i.e., textbooks and college courses, treatments of these topics typically focus on the underlying ideas to the exclusion of practical implementations. This is understandable: the insights of computer science are independent of the programming language, operating system, or hardware (even as some of those insights greatly influence these things!), and there are only so many pages in a textbook, or lectures in a course.

I think, however, such treatments would benefit from practical implementations of their ideas. This paper is the first in a hoped-for series that provides a stand-alone[1], practical[2], and approachable[3] implementation of a "textbook" algorithm.

We will discuss Bloom filters, introduced[2] over 50 years ago and still going strong. From the perspective that this document is to supplement existing textbook discussions (usually in specialized algorithms textbooks[3]), we will jump as quickly as possible into a concrete implementation. The rest of the document is as follows:

- In section 2 we will provide a complete implementation of a Bloom filter in standard C++17.
- In section 3 we will provide a stand-alone "driver" program that can take command-line parameters and actually execute code using the Bloom filter, so that we might examine the behavior of our implementation in a real environment.
- Finally, in section 4, we will present some findings and thoughts based on using that driver.

Along the way we will have brief interludes discussing how our practical work and findings complement academic treatments, for the learner.

2 Bloom Filter: Theory to Code

This section "reads along" with a textbook description of Bloom filters[3], and derives code from the prose.

2.1 Bloom Filter Implementation

The Bloom filter is a persistent object that summarizes a collection of data, so we expect it to have some state. In particular, we have a sequence A of n bits, and a collection of k different hash functions. This can be realized with the following code:

[1] In the sense that only a standard compiler and *minimal* external dependencies are needed.
[2] In the sense that the code runs and, at least sometimes, exhibits its purported behavior.
[3] In the sense that, with the accompanying text, readers will find the code completely comprehensible.

```
template <class V, size_t (*h)(uint32_t i, const V &)>
class bloom_filter {
private:
  simple_bit_set A;

public:
  size_t n = 0;
  size_t k = 0;
  bloom_filter() {}
  bloom_filter(const size_t n, const size_t k) : A(n), n(n), k(k) {}
  int count() const { return A.count(); }
  size_t size() const { return A.size(); }

  // insert fig. 2 (Bloom filter set)
  // insert fig. 3 (Bloom filter test)
};
```

Figure 1: Bloom Filter Implementation

To restate: The field `A` is the core sequence of bits. It's type, `simple_bit_set`, is specified later in section 2.2. When we talk about "setting" or "checking" bits, we are referencing to operations on `A`. The parameter `n` is the number of bits we'll be using in our sequence.

The hash function `h` is provided at compile-time as a template-parameter. Note that it doesn't just take a `V` (by reference), but also an index `i`. This `i` exactly corresponds to the index machinery our textbook relies on. The idea is that `h(0, v)` will give a *completely* different value than `h(1, v)`, even though `v` is the same between the two. While mechanically it is a single function, we can pretend it's an arbitrary-length array of hash functions, the first parameter being the "index" into that array.

Aside We made some design decisions for our Bloom filter: the size and density (`n` and `k`) are runtime parameters, but the type `V` and hash function `h` are compile-time. These considerations are (understandably) not discussed in theoretical treatments, but matter for practical implementations.

Part of these expositions is to consider *where* we might consider these questions. A Bloom filter, and other algorithms, have *theoretical* considerations (like the expected ϵ) as also practical considerations. These include design questions, like: What's the type-system interface for a Bloom filter? How much is the design constrained by the particular language features? These extend into practical considerations: For instance, alternative implementations could be more flexible with the type-system, but at the cost of an indirect call for each hash call.

In any case, with our parameters defined, we can turn to the key functionality: How do we add elements to our filter, and how do we test if an element was added to our filter? For both operations we will consider a quote from our reference, and then map that to code. Happily, at this point we will find the mapping very straightforward.

Adding Elements

> For each element $s \in S$, the bits $A[h_i(s)]$ are set to 1 for $1 \leq i \leq k$. A bit location can be set to 1 multiple times, but only the first change has an effect.[3, p. 109].

```
void set(const V &s) {
  for (size_t i = 0; i < k; i++) {
    A.set(h(i, s) % n);
  }
}
```

Figure 2: Bloom filter set

Testing an Element

> To check if an element x is in S, we check whether all array locations $A[h_i(x)]$ for $1 \leq i \leq k$ are set to 1. If not, then clearly x is not a member of S,.... If all $A[h_i(x)]$ are set to 1, we assume x is in S, although we could be wrong.[3, p. 109].

```
bool test(const V &s) const {
  for (size_t i = 0; i < k; i++) {
    if (!A.test(h(i, s) % n)) {
      return false;
    }
  }
  return true;
}
```

Figure 3: Bloom filter test

For our demonstrative implementation, that is all we need! This completes the core of our implementation, and we can wrap it in a header file.

What remains unimplemented as is the `simple_bit_set`, as well as our definitions of `h` and `V`.

2.2 Array of Bits

Some languages have this built in already. The standard C++ libraries provide a *fixed-length* (at compile time) bit-array, but combining it with the dynamically-resizable `std::vector` container leads to a natural implementation:

```cpp
#pragma once
#include <bitset>
#include <numeric>
#include <vector>
class simple_bit_set {
private:
  static const int page_size = 64;
  typedef std::bitset<page_size> page;
  std::vector<page> data;

public:
  simple_bit_set() {}
  simple_bit_set(const int size) : data((size / page_size) + 1, 0) {}
  bool test(unsigned int i) const {
    return data[(i / page_size)].test(i % page_size);
  }
  void set(unsigned int i) {
    data[(i / page_size)].set(i % page_size);
  }
  int size() const { return data.size() * page_size; }
  int count() const {
    int r = 0;
    for (const auto& p : data) { r += p.count(); }
    return r;
  }
};
```

Figure 4: simplebitset.h

The one "trick" that may be new to implementors is how we "translate" `i` into which 64-bit `page` to look up, and then look up the right bit within that page. We can also drill into another detail: the number of bits we set will always be some multiple of `page_size`. Consequently, we may sometimes have slightly more bits than our Bloom filter expects. This is fine, they're simply never referenced. This also isn't anticipated to appreciably change the performance or even the analysis, but another example of a "rough edge" between theory and practice.

2.3 Hashing Scheme and Values to Hash

We need a so-called "good" hash for our Bloom filter. Hashing algorithms are a whole field into themselves, and one I hope to explore later. For now, we will defer to what I hear is established as "good practice" and use the public-domain MurmurHash3 algorithm[1]. This is understood to be an extremely fast and good hash, without the overhead of cryptographic guarantees. It provides a simple C-header interface, and most conveniently provides a "seed" parameter that exactly matches the use of i in h_i.

Here is our explicit hashing function:

```cpp
size_t string_hash(uint32_t i, const std::string &v) {
  uint32_t hash;
  MurmurHash3_x86_32(v.c_str(), v.size(), i, &hash);
  return static_cast<size_t>(hash);
}
```

Figure 5: Hashing Function

Note that implicit in our implementation is that we've also picked our type `V`: we'll be hashing a collection of `std::string`s. For our type system, generally, we'd want to have a hash function implemented "in tandem" with our value-type. This is so that it knows how to look at the whole value. In this case, a `std::string` value is actually a small object holding data like the length, and a *pointer* to the "real" content of the string.[4] We should hash the *contents* of the pointer rather than the *value* of the pointer. To hash the values would be a bug: In the case where we have two copies of the same string, their underlying bytes may exist in different memory locations so a naive hash would say they're *different*, but we need them to be reported the same! So our `string_hash` implementation takes care of telling our underlying hash algorithm which data to look at, and how, for hashing.

Personal Interlude More than once in my career a colleague has said "we need a hashing algorithm that provides good guarantees. Let's use SHA!" In a sense, I agree: the properties promised by *cryptographic* hashing functions are very nice. My background in computational complexity reminds me always that features (usually) have a price—in this case, CPU time. When we just want to get a nice hashing algorithm for some in-memory object, we typically don't need to worry about hackers trying to, given the hash result, figure out the actual value!

This completes all the components of our implementation. We have a Bloom filter library! We could say we're done here: "in principle" someone can go use this code and do Bloom-filter-y things with it. We are going to do just that: harness the code into a small command-line program that is purpose-fit to run experiments.

3 Harnessing the Code

We want to write code to run experiments on our Bloom filter. We'll go through the various components we need, chunk-by-chunk, and then at the end tie it all together into a single driver program.

3.1 Generating Random Strings

Bloom filters summarize sets of strings: so let's make lots of unique strings. We're not going to worry about efficiency at all, but just try to expediently generate guaranteed-unique strings. Our implementation follows a fairly standard pattern: generate stuff and deduplicate things via a set until we get to a big-enough set.

[4] In practice things are actually even a bit more complicated for `std::string`, where sometimes due to the so-called "short-string optimization" there isn't a pointer in use!

```cpp
const static std::string letter_set =
    "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789";

std::vector<std::string> create_strings(size_t count, size_t len) {
  std::random_device rd;
  std::default_random_engine e1(rd());
  std::uniform_int_distribution<int> uniform_dist(0, letter_set.length()-1);

  std::unordered_set<std::string> unique_strings;
  std::string output;
  output.resize(length_of_strings);
  while (unique_strings.size() < count) {
    for (int j = 0; j < length_of_strings; j++) {
      auto k = uniform_dist(e1);
      output[j] = letter_set[k];
    }
    unique_strings.insert(output);
  }
  std::vector<std::string> data(std::begin(unique_strings),
                                std::end(unique_strings));
  return data;
}
```

Figure 6: Create Strings

This is a pretty limited generator: There's lots of API limitations: we are fixed to create `count` strings all of the same length. There is no way to specify the "random seed" to enable repeatability. There are also probably lots of C++-ism improvements, like move-semantic stuff (including emplacing), as well as API changes like taking an output interator. Note also that a small-enough length, with a big-enough count request, means we'd never terminate (the set would simply never get big enough). But! For our purposes, despite all these limitations, it gets us enough to run our experiments.

3.2 Measuring Time

Bloom filters speed things up in theory. Some of our experiments can be to see if that happens in practice. There are myriad ways to measure time in C++, here we can have a timer-object that uses constructor and destructor semantics to time how long it takes for the computer to execute a certain scope-of-code.

```
class timer_object {
  std::chrono::time_point<std::chrono::steady_clock> start;
  double* elapsed_seconds;
public:
  timer_object(double* elapsed_seconds) {
    this->elapsed_seconds = elapsed_seconds;
    start = std::chrono::steady_clock::now();
  }
  ~timer_object() {
    auto end = std::chrono::steady_clock::now();
    std::chrono::duration<double> elapsed = end - start;
    *elapsed_seconds = elapsed.count();
  }
};
```

Figure 7: Timer Object

For those unfamiliar with C++, the idea is that we construct the timer object (via `timer_object(double*...)`), at which point it saves the the `start` time. At the end of the scope, the object is automatically destroyed, and that invokes the destructor, `~timer_object()`, which compares the current time with the saved time and writes the difference in the externally-provided `*elapsed_seconds`.

3.3 Receive Input Parameters

Our program is ultimately going to be called from the command line. We implement code to parse a (large!) number of flags that will enable us to run our program with various options. We can detail them here:

```
int c;
while ((c = getopt(argc, argv, "dfae:n:x:l:k:b:m:rhv:")) != -1) {
  switch (c) {
    // Which experiments to run:
    case 'd': run_density_experiment = true; break;
    case 'f': run_false_positive_experiment = true; break;
    case 'a': run_application_experiment = true; break;

    // Parameters to the experiment
    case 'e': desired_epsilon = std::strtod(optarg, nullptr); break;
    case 'n': num_strings_in_set = std::strtol(optarg, nullptr, 0); break;
    case 'x': num_strings_not_in_set = std::strtol(optarg, nullptr, 0); break;
    case 'l': length_of_strings = std::strtol(optarg, nullptr, 0); break;
    case 'k': num_hash_functions = std::strtol(optarg, nullptr, 0); break;
    case 'b': num_bits_per_value = std::strtod(optarg, nullptr); break;
    case 'm': num_bits_in_filter = std::strtol(optarg, nullptr, 0); break;
    case 'r': rounding_constant = 1; break;

    // Meta-parameters
    case 'h': print_help = true; break;
    case 'v': verbosity_level = std::strtol(optarg, nullptr, 0); break;
  }
}
```

Figure 8: Read Command-Line Parameters

The loop is a standard `getopt` interface. Some flags are just binary: the first three indicate which experiment(s) the user is requesting to run. The largest chunk of flags is how the user specifies, e.g., the number of strings to insert into the Bloom filter (`num_strings_in_set`), the number of strings to create guaranteed *not* to be in the Bloom filter (`num_strings_not_in_set`) and things like the desired ϵ or the raw size of the Bloom filter. Some of these flags are mutually exclusive, and most are ultimately used to construct the Bloom filter.

3.4 Constructing the Bloom Filter

Some of the Bloom filter parameters can be specified from the previous options, and others can be computed from them. Here is that logic, intertwined with some more user-friendly options (like debug statements).

```
if (verbosity_level != 0) {
  print_driver_state();
}
if (num_bits_per_value == 0.0) {
  // Specify default to be "optimal".
  num_bits_per_value = -1.44 * std::log2(desired_epsilon);
}
if (num_bits_in_filter == 0) {
  num_bits_in_filter = num_strings_in_set * num_bits_per_value;
}
if (num_hash_functions == 0) {
  if (verbosity_level >= 1) {
    printf("Ideal number of hash-functions pre-rounding: %f\n", -std::log2(desired_epsilon));
  }
  num_hash_functions = size_t(-std::log2(desired_epsilon)) + rounding_constant;
}
if (verbosity_level >= 1) {
  printf("Calculated best bits per value: %f, total number of bits: %ld, number of hash functions: %ld\n",
          num_bits_per_value, num_bits_in_filter, num_hash_functions);
}
```

Figure 9: Prepare Experiments

See how things like `num_hash_functions`, if left uninitialized by the user, are automatically inferred to be the "optimal" settings given some other parameters. The deriviation of how things are optimal is the subject of the literature that this write-up is complementing. (That is to say, I'm not going to write how we determined the formulas for the optimal values.)

With these parameters prepared, we can finally compute our test-string-sets and Bloom filter:

```
auto total_strings = num_strings_in_set + num_strings_not_in_set;
strings = create_strings(total_strings, length_of_strings);
included_string_begin = std::begin(strings);
included_string_end = included_string_begin+num_strings_in_set;
excluded_string_begin = included_string_end;
excluded_string_end = excluded_string_begin+num_strings_not_in_set;
assert(excluded_string_end == std::end(strings));

if (verbosity_level > 1) {
  printf("will add %zu strings, test %zu strings",
    std::distance(included_string_begin, included_string_end),
    std::distance(excluded_string_begin, excluded_string_end));
}

bf = bloom_filter<std::string, string_hash>(num_bits_in_filter, num_hash_functions);
// Populate our Bloom filter set and a baseline std::set.
std::for_each(included_string_begin, included_string_end,
              [](auto &s) {
                bf.set(s);
                base_set.insert(s);
              });
```

Figure 10: Initialize Driver State

These sets are going to be shared between whatever experiments we run. So we can isolate out the core *state* of our program: we can move past our command-line parameters and think solely about the objects we'll use for our experiments.

That state is expressed here:

```
bloom_filter<std::string, string_hash> bf;
std::set<std::string> base_set;
std::vector<std::string> strings;
std::vector<std::string>::const_iterator
  included_string_begin,
  included_string_end,
  excluded_string_begin,
  excluded_string_end;
```

Figure 11: Driver State

Note that the four `const_iterator`s combine to define two distinct sequences: the first sequence being the strings *in* the Bloom filter, the second sequence being those strings *not* in the Bloom filter. These are both subsequences of the vector `strings`.

3.5 Defining our Experiments

As hinted in fig. 8 we are going to implement 3 experiments to run with our Bloom filters and sets. Their implementations are straightforward; only the application experiment needs helper functions.

```
if (run_density_experiment) {
  const double density =
    static_cast<double>(bf.count()) / static_cast<double>(num_bits_in_filter);
  printf("%f %zu\n", density, bf.size());
}
if (run_false_positive_experiment) {
  auto false_positives = std::count_if(excluded_string_begin, excluded_string_end,
                                       [](auto &v) { return bf.test(v); });
  double fp_rate = double(false_positives) / double(num_strings_not_in_set);
  printf("%f\n", fp_rate);
}
if (run_application_experiment) {
  double base_seconds = time_baseline();
  double bloom_seconds = time_bloom_filter();
  printf("%f %f\n", base_seconds, bloom_seconds);
}
```

Figure 12: Run Selected Experiments

Our first experiment simply reports some static properties of the already-initialized Bloom filter. The second experiment "just" validates the error rate (the ϵ, the false-positive rate) by seeing how many of the guaranteed-not-to-have-been-inserted-in-the-Bloom-filter strings nonetheless are reported as included. The third is the most involved: we use that same sequence of guaranteed-not-included strings, and see how much putting a Bloom filter "in front" of a normal set speeds things up. Those details follow:

```
double time_baseline() {
  double seconds = 0;
  size_t count = 0;
  {
    timer_object t(&seconds);
    for (auto it = excluded_string_begin; it != excluded_string_end; it++) {
      if (base_set.find(*it) != base_set.end()) {
        count++;
      }
    }
  }
  assert(!count);
  return seconds;
}
```

Figure 13: Time Baseline

```
double time_bloom_filter() {
  double seconds = 0;
  size_t count = 0;
  {
    timer_object t(&seconds);
    for (auto it = excluded_string_begin; it != excluded_string_end; it++) {
      if (bf.test(*it)) {
        if (base_set.find(*it) != base_set.end()) {
          count++;
        }
      }
    }
  }
  assert(!count);
  return seconds;
}
```

Figure 14: Time Bloom Filter

I call this "application" instead of, say, "benchmarking" because (even though of course it really is just a contrived benchmark) the intent is to express *how* Bloom filters are actually used. We might consider them alone as an object of study, but the intent is to save time by (sometimes) avoiding an expensive lookup.

3.6 Tying it all together

We have all of our utilities and interfaces. Now we just need to put it all in an executable program. At a high-level, we describe our driver like so:

```
// insert ?? (Driver Headers)
// insert ?? (Bloom Filter Settings)
// insert ?? (Driver Utility Functions)
// insert fig. 11 (Driver State)
// insert fig. 13 (Time Baseline)
// insert fig. 14 (Time Bloom Filter)
int main(int argc, char* argv[]) {
  // insert fig. 8 (Read Command-Line Parameters)
  // insert ?? (Error-Check Parameters)
  // insert fig. 9 (Prepare Experiments)
  // insert fig. 10 (Initialize Driver State)
  // insert fig. 12 (Run Selected Experiments)
}
```

Figure 15: driver.cpp

The complete listing can be found in appendix B. Some of the chunks (like the headers, error-checking code, printing out the "help" instructions, and so on) are not included in the text and just appear in the listing. The exposition here is hopefully enough to describe how this driver works at a high level, and the complete code is included to keep things unambiguous.

Now let's write scripts to exercise our CLI program in various ways to finally learn more about Bloom filters in practice.

We have our complete program: we're now ready to get some empirical results!

4 Experiments

We now explore whether our driver successfully confirms theoretical results.

4.1 Optimal Bit-Density

As mentioned in our motivating textbook[3, p. 111], an interesting property of an "optimal" Bloom filter is that its density (the proportion of bits set to 1) should approach 0.5. This touches on the complicated question of what is "optimal", but for our purposes: Given a size N collection of strings to hash and desired false-positive rate ϵ, we should be able to determine a "best" number-of-bits m and number-of-hashes k.

Can we validate this manifesting in our code, empirically? Yes. We might want a chart to show that, no matter the specified ϵ, number of input strings, or length of strings, our default hash-function-count and Bloom-filter-size will yield the desired ϵ and have density very close to 0.5. (We're not going to be exhaustive here: this document is to illustrate that theoretical properties can be empirically observed, not a full validation of our Bloom filter implementation.)

So let's make an interesting graph: let's vary ϵ, and chart how density varies. Here we have a terse script that runs this experiments and plots the result:

```
#!/bin/bash
e=0.8
output="$(basename "$0" .sh).fig.tex"
for i in {1..10}; do
  measured=$(./driver -d -e "$e" -n 100000 -l 24 -r)
  echo "$e $measured"
  e=$(echo "$e / 2" | bc -l)
done | gnuplot -e "
  set terminal tikz size 3.5in,2.4in;
  set xlabel 'Specified $\epsilon$';
  set xrange [*:1];
  set logscale x;
  set ylabel 'Bit Density';
  set output '$output';
  plot '-' using 1:2 with linespoints notitle"
```

Figure 16: plot-density.sh

The desired epsilon (`e`) starts at 0.8, and on each iteration (`e=$(echo "$e / 2" | bc -l)`) halves it: so we're running our experiment with $\epsilon \in \{0.8, 0.4, 0.2, 0.1, 0.05 \ldots\}$. (Note that the X-axis may seem sort of "backwards": the bottom-left is showing that with very-low ϵ, i.e., a "better" Bloom filter, whereas ones intuition may suggest our chart starts with a "worse" Bloom filter on the left.) The re-

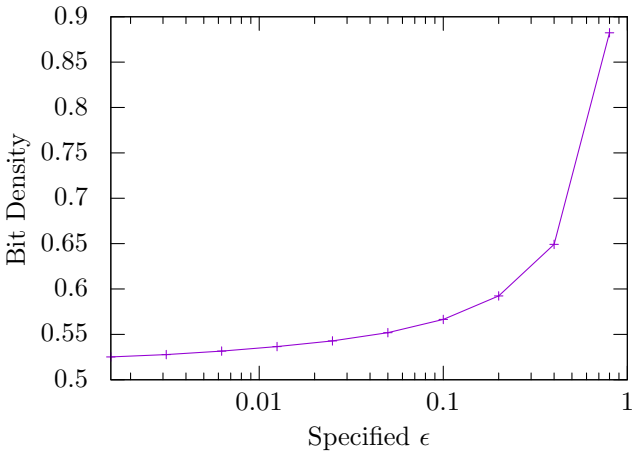

Figure 17: Density by Desired Epsilon

sult in fig. 17 confirms that, as we request smaller and smaller ϵ, the resulting bit-density approaches 0.5. It doesn't quite approach 0.5 due to rounding.

Intuition—and our code in fig. 9—suggests that closer our density gets to 0.5, the more bits we need to dedicate to each value in the Bloom filter. (Note that, however, the number of bits dedicated to each value is wholly a function of ϵ and the total number of strings—*not* the size of the strings. That's neat!) We can confirm that with a very similar experiment: we have the exact same script as before, but instead read the next column in the output (that says the size of the Bloom filter).

```
#!/bin/bash
e=0.8
output="$(basename "$0" .sh).fig.tex"
for i in {1..10}; do
  measured=$(./driver -d -e "$e" -n 100000 -l 24 -r)
  echo "$e $measured"
  e=$(echo "$e / 2" | bc -l)
done | gnuplot -e "
  set terminal tikz size 3.5in,2.4in;
  set xlabel 'Specified $\epsilon$';
  set xrange [*:1];
  set logscale x;
  set ylabel 'Filter Size';
  set output '$output';
  plot '-' using 1:3 with linespoints notitle"
```

Figure 18: plot-size.sh

The (log-scale, again) result in fig. 19 clearly confirms that our Bloom filter gets bigger the closer ϵ gets to 0.

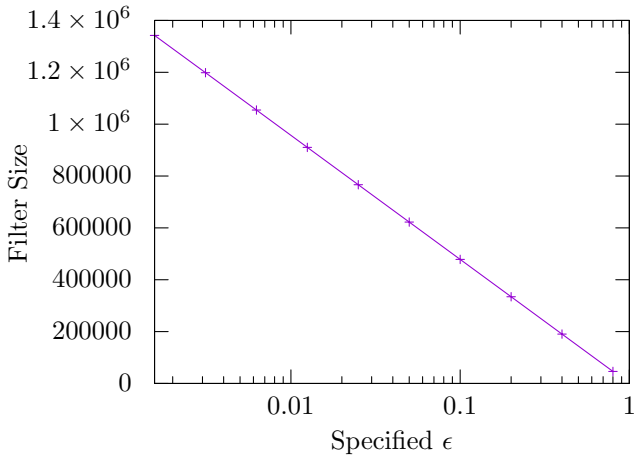

Figure 19: Bloom Filter Size by Desired Epsilon

In our prior experiments we've let the driver compute, given the ϵ, the "optimal" number of hash functions. This invites an empirical question: how bad is it to get that wrong? In other words, how far off does our density vary from 0.5 if we have a typo somewhere leading to the wrong hash-function count? We compute this as follows:

```
#!/bin/bash
output="$(basename "$0" .sh).fig.tex"
for hash_count in {1..40}; do
  result=$(./driver -d -e 0.0009 -n 100000 -l 24 -k "$hash_count")
  echo "$hash_count $result"
done | gnuplot -e "
  set terminal tikz size 3.5in,2.4in;
  set yrange [0:1];
  set xlabel 'Number of Hash Functions';
  set ylabel 'Bit Density';
  set output '$output';
  plot '-' using 1:2 with linespoints notitle"
```

Figure 20: plot-density-per-hash.sh

The resulting plot shows that you can be slightly off with k before it's "obviously" wrong. That's interesting, and an empirical result that seems hard (to me, at least) to derive analytically.

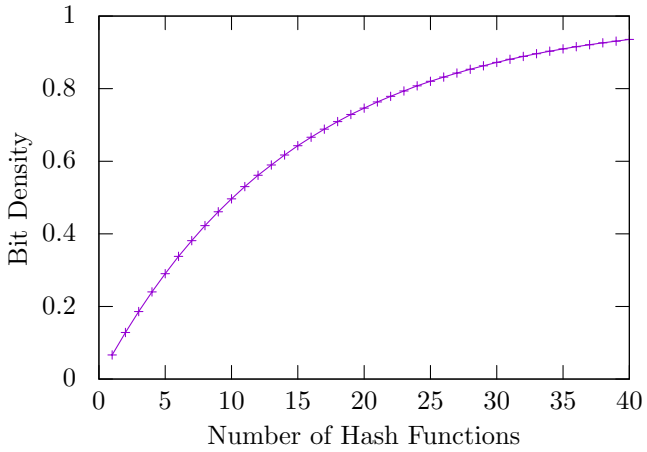

Figure 21: Density by Hash Function Count

4.2 False Positive Rate

We've confirmed density. Now, quickly: to what degree does our code empirically match the expected false-positive rate? Basically, when we specify an epsilon, do we get that epsilon? We can validate this with our other experiment run in our driver:

```
#!/bin/bash
output="$(basename "$0" .sh).fig.tex"
tmpfile=$(mktemp)
e=0.8
for i in {1..10}; do
  measured=$(./driver -f -e "$e" -n 100000 -x 100000 -l 24 -r)
  echo "$e $e $measured"
  e=$(echo "$e / 2" | bc -l)
done > $tmpfile

gnuplot -e "
  set terminal tikz size 3.5in,2.4in;
  set xlabel 'Specified Epsilon';
  set ylabel 'Actual Epsilon';
  set xrange [*:1];
  set logscale x;
  set yrange [*:1];
  set logscale y;
  set key at graph 0.60,0.95;
  set output '$output';
  plot '$tmpfile' using 1:2 with linespoints title 'Expected eps',
       '$tmpfile' using 1:3 with linespoints title 'Measured eps'
  "

rm $tmpfile
```

Figure 22: validate-epsilon-empirically.sh

We can see that it seems to track very precisely in fig. 23, so that's reassuring.

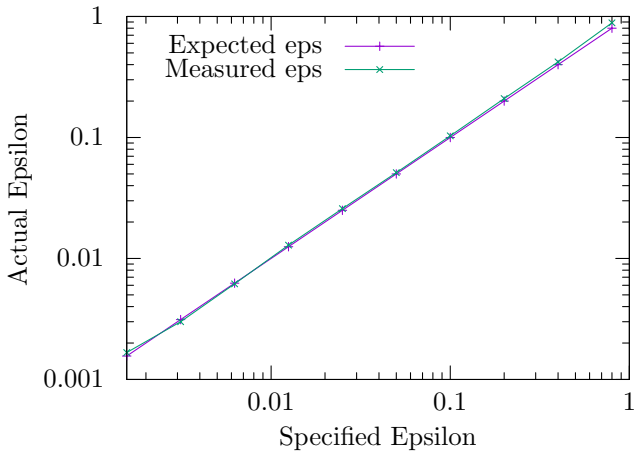

Figure 23: Comparing Empirical to Theoretical ϵ

Interlude: one thing I realized is that, curiously, with $\epsilon > 0.5$, the false positive rate was always (at first) exactly 1.0. Even for $\epsilon = 0.51$ or similar. In debugging I found that this is because we'd always round down.

The static properties of our Bloom filter seem to match our theoretical expectations. Now: do we get the runtime improvements we expect?

4.3 Measuring Set Speedup

We've explored and confirmed some of the static properties of our Bloom filter. Now let's explore the whole motivation: how much does putting a Bloom filter in front of a (slow) `std::set` help? In the following experiment we fix our Bloom filter to have an $\epsilon = 0.01$, request one million strings be in the set, and vary the excluded strings from $100,000$ to one million. We run our timing ("application") experiment, see fig. 12 on these parameters and plot how the speedup scales.

```
#!/bin/bash
output="$(basename "$0" .sh).fig.tex"
tmpfile=$(mktemp)
for miss_count in {1..10}; do
  count="$miss_count"00000
  result=$(./driver -a -e 0.01 -n 1000000 -x $count -l 24)
  echo "$count $result"
done > $tmpfile

gnuplot -e "
  set terminal tikz size 5in,2.4in;
  set xlabel 'Number of Excluded Values';
  set format x '$%.0t \cdot 10^{%T}$';
  set ylabel 'Time (Seconds)';
  set output '$output';
  plot '$tmpfile' using 1:2 with linespoints title 'Without BF',
       '$tmpfile' using 1:3 with linespoints title 'With BF'"

rm $tmpfile
```

Figure 24: acceleration-by-count.sh

As we can see in the results in fig. 25 we get a *real* win with Bloom filters. In reality we should also profile the resulting programs and confirm that the execution really changes in the way we expect. I've seen silly things like one-half of the experiment accidentally allocate everything twice. Profiling serves the purpose of getting the real "root cause" of the performance change. For this exposition, however, let's just take it on faith. Sequel work can give instrumented assembly.

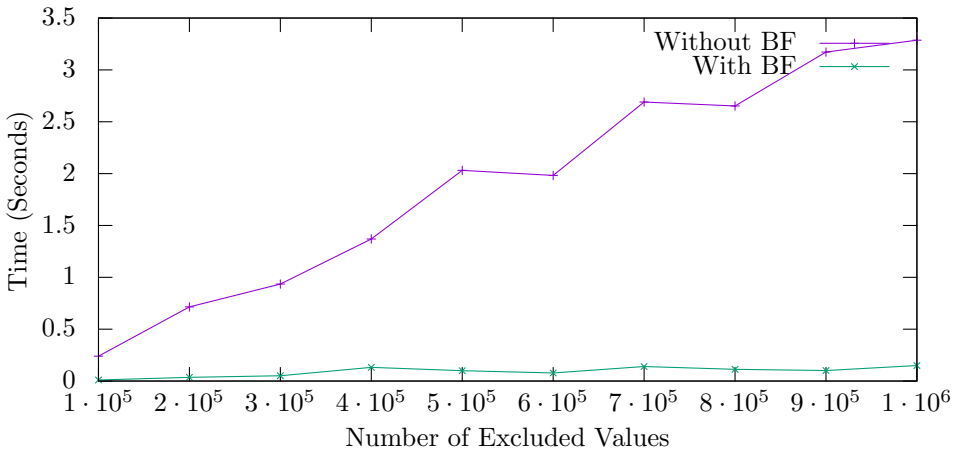

Figure 25: Comparing Speedup With Bloom Filters

We may still be curious in one more variation: recall the Bloom filters interestingly *don't* take up more room based on the size of each *element* (only the count of elements). Would this be reflected in much-larger strings, so that the `std::set` has to do even more, worse, string comparisons? The following is the same experiment as before, but note `-l 240`: our input strings are getting pretty big.

```
#!/bin/bash
output="$(basename "$0" .sh).fig.tex"
tmpfile=$(mktemp)
for miss_count in {1..10}; do
  count="$miss_count"00000
  result=$(./driver -a -e 0.01 -n 1000000 -x $count -l 240)
  echo "$count $result"
done > $tmpfile

gnuplot -e "
  set terminal tikz size 5in,2.4in;
  set xlabel 'Number of Excluded Values';
  set format x '$%.0t \cdot 10^{%T}$';
  set ylabel 'Time (Seconds)';
  set output '$output';
  plot '$tmpfile' using 1:2 with linespoints title 'Without BF',
       '$tmpfile' using 1:3 with linespoints title 'With BF'"

rm $tmpfile
```

Figure 26: acceleration-by-count-big.sh

Surprisingly, *the Bloom filter* slows down in fig. 27! Or at least, the wins are smaller. While contrary (at least superficially) to our theoretical understanding, some ad-hoc profiling I did suggests we're spending a lot of time hashing the strings to compute the Bloom filter bits. I think there's a lot more insight to mine here, but again for brevity we'll leave more analysis for another exposition.

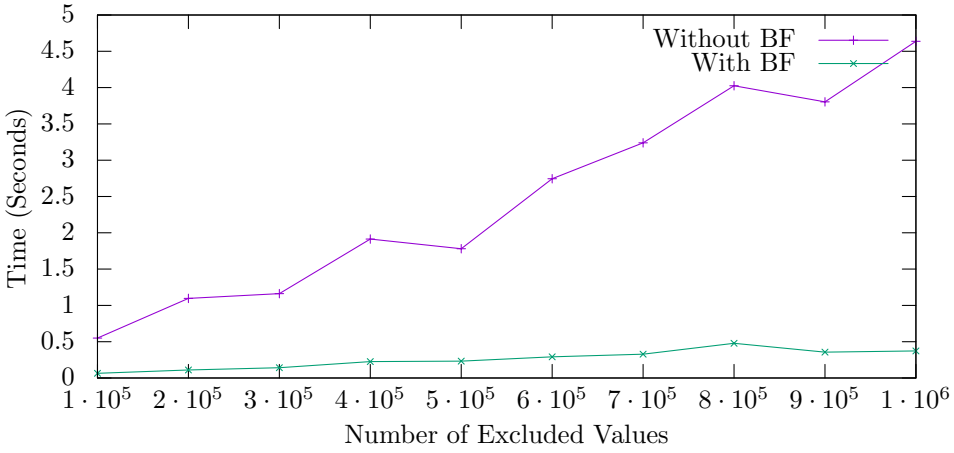

Figure 27: Comparing Speedup with Bloom Filters on Big Strings

We've previously committed to $\epsilon = 0.01$. Maybe that's too much. Focusing just on the Bloom filter case now, how much speedup do we get from $\epsilon = 0.01$ versus $\epsilon = 0.1$? We'll not look at those exact numbers, but here is such an experiment:

```
#!/bin/bash
output="$(basename "$0" .sh).fig.tex"
e=0.8
for i in {1..8}; do
  result=$(./driver -a -e $e -n 1000000 -x 1000000 -l 24)
  e=$(echo "$e / 2" | bc -l)
  echo "$e $result"
done | gnuplot -e "
  set terminal tikz size 5in,2.4in;
  set xlabel 'Specified $\epsilon$';
  set xrange [*:1];
  set logscale x;
  set ylabel 'Time (Seconds)';
  set output '$output';
  plot '-' using 1:3 with linespoints title 'With BF'"
```

Figure 28: acceleration-by-precision.sh

The results in fig. 29 suggests TODO.

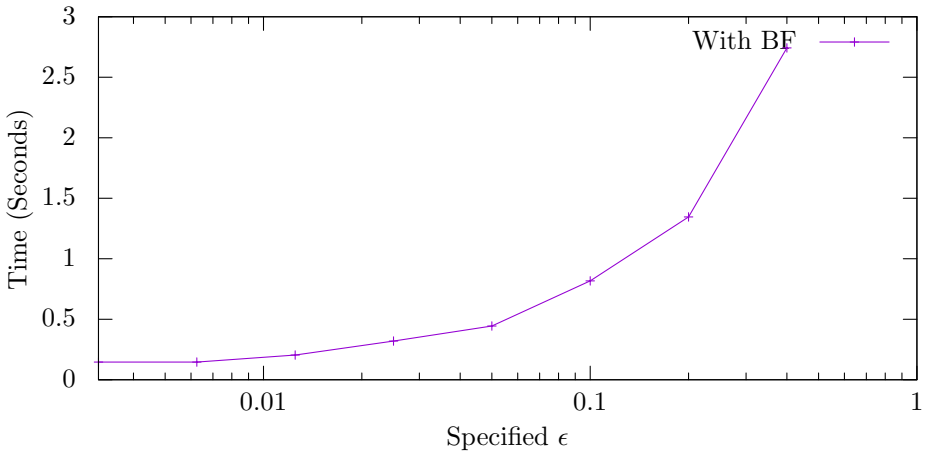

Figure 29: Runtime Variation by Epsilon

5 Parting Thoughts

Each experiment I ran lead to more thoughts on *other* interesting experiments to run. That's fun and exciting! Bloom filters are really amenable to implementation and experimenation. I wonder if we can get to such a point for other, more sophisticated theoretical objects like those used in SAT solvers. The "wins" in Bloom filters partly come from using uniform random numbers (hashing), and so to a sense it's workload-agnostic: we only have a few dimensions of the input to consider (and the moment we consider another dimension, e.g., input-string-length, we do start getting unintuitive results). SAT solvers, compilation algorithms, and other "big" things seem much more sensitive to workloads. Just something I'm ruminating on.

I think literate code, at best, comes at the very, very end. Trying to write this code "notebook-style" just didn't work for me.

While the empiricism in this write-up was fun and I think a valuable (uncommon) contribution, it was not as obviously-educational as I would have guessed. I think for a student this hopefully demonstrates some new ways of thinking about programming in-the-small (a discrete boundary between our library and our driver-program, things like `getopt`), and that it's *possible* to place theoretical results in practice. I would be curious if a students walks away feeling empowered to play with other theoretical objects in code. Maybe C++ was the wrong choice; I thought it'd help with, e.g., profiling.

A Library Code Listings

A.1 Bloom Filter (bloomfilter.h)

```cpp
#pragma once
#include "simplebitset.h"
#include <cstddef>
#include <cstdint>

template <class V, size_t (*h)(uint32_t i, const V &)>
class bloom_filter {
private:
  simple_bit_set A;

public:
  size_t n = 0;
  size_t k = 0;
  bloom_filter() {}
  bloom_filter(const size_t n, const size_t k) : A(n), n(n), k(k) {}
  int count() const { return A.count(); }
  size_t size() const { return A.size(); }

  void set(const V &s) {
    for (size_t i = 0; i < k; i++) {
      A.set(h(i, s) % n);
    }
  }
  bool test(const V &s) const {
    for (size_t i = 0; i < k; i++) {
      if (!A.test(h(i, s) % n)) {
        return false;
      }
    }
    return true;
  }
};
```

A.2 Simple Bit Set (simplebitset.h)

```cpp
#pragma once
#include <bitset>
#include <numeric>
#include <vector>
class simple_bit_set {
private:
  static const int page_size = 64;
  typedef std::bitset<page_size> page;
  std::vector<page> data;

public:
```

```
  simple_bit_set() {}
  simple_bit_set(const int size) : data((size / page_size) + 1, 0) {}
  bool test(unsigned int i) const {
    return data[(i / page_size)].test(i % page_size);
  }
  void set(unsigned int i) {
    data[(i / page_size)].set(i % page_size);
  }
  int size() const { return data.size() * page_size; }
  int count() const {
    int r = 0;
    for (const auto& p : data) { r += p.count(); }
    return r;
  }
};
```

B Driver Code Listings

```
#include <algorithm>
#include <cassert>
#include <chrono>
#include <cmath>
#include <cstdio>
#include <cstdlib>
#include <unordered_set>
#include <set>
#include <string>
#include <unistd.h>

#include "bloomfilter.h"
#include "datagen.h"
#include "murmur/MurmurHash3.h"
double desired_epsilon = 0.0;
int num_strings_in_set = 0;
int num_strings_not_in_set = 0;
int length_of_strings = 0;
int rounding_constant = 0;

bool print_help = false;
int verbosity_level = 0;

size_t num_hash_functions = 0;
double num_bits_per_value = 0.0;
size_t num_bits_in_filter = 0;

bool run_density_experiment = false;
bool run_false_positive_experiment = false;
bool run_application_experiment = false;
const static std::string letter_set =
```

```
32      "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789";
33
34  std::vector<std::string> create_strings(size_t count, size_t len) {
35    std::random_device rd;
36    std::default_random_engine e1(rd());
37    std::uniform_int_distribution<int> uniform_dist(0, letter_set.length()-1);
38
39    std::unordered_set<std::string> unique_strings;
40    std::string output;
41    output.resize(length_of_strings);
42    while (unique_strings.size() < count) {
43      for (int j = 0; j < length_of_strings; j++) {
44        auto k = uniform_dist(e1);
45        output[j] = letter_set[k];
46      }
47      unique_strings.insert(output);
48    }
49    std::vector<std::string> data(std::begin(unique_strings),
50                                  std::end(unique_strings));
51    return data;
52  }
53  size_t string_hash(uint32_t i, const std::string &v) {
54    uint32_t hash;
55    MurmurHash3_x86_32(v.c_str(), v.size(), i, &hash);
56    return static_cast<size_t>(hash);
57  }
58  void print_driver_state() {
59    printf("parameters:\n"
60      "\tprint_help=%s\n"
61      "\tverbosity_level=%d\n"
62      "\trun_density_experiment=%s\n"
63      "\trun_false_positive_experiment=%s\n"
64      "\trun_application_experiment=%s\n"
65      "\tdesired_epsilon=%f\n"
66      "\tnum_strings_in_set=%d\n"
67      "\tnum_strings_not_in_set=%d\n"
68      "\tlength_of_strings=%d\n",
69      print_help ? "true" : "false", verbosity_level,
70      run_density_experiment ? "true" : "false",
71      run_false_positive_experiment ? "true" : "false",
72      run_application_experiment ? "true" : "false",
73      desired_epsilon, num_strings_in_set, num_strings_not_in_set,
74      length_of_strings);
75  }
76  class timer_object {
77    std::chrono::time_point<std::chrono::steady_clock> start;
78    double* elapsed_seconds;
79  public:
80    timer_object(double* elapsed_seconds) {
81      this->elapsed_seconds = elapsed_seconds;
```

```
82      start = std::chrono::steady_clock::now();
83    }
84    ~timer_object() {
85      auto end = std::chrono::steady_clock::now();
86      std::chrono::duration<double> elapsed = end - start;
87      *elapsed_seconds = elapsed.count();
88    }
89  };
90  bloom_filter<std::string, string_hash> bf;
91  std::set<std::string> base_set;
92  std::vector<std::string> strings;
93  std::vector<std::string>::const_iterator
94    included_string_begin,
95    included_string_end,
96    excluded_string_begin,
97    excluded_string_end;
98  double time_baseline() {
99    double seconds = 0;
100   size_t count = 0;
101   {
102     timer_object t(&seconds);
103     for (auto it = excluded_string_begin; it != excluded_string_end; it++) {
104       if (base_set.find(*it) != base_set.end()) {
105         count++;
106       }
107     }
108   }
109   assert(!count);
110   return seconds;
111 }
112 double time_bloom_filter() {
113   double seconds = 0;
114   size_t count = 0;
115   {
116     timer_object t(&seconds);
117     for (auto it = excluded_string_begin; it != excluded_string_end; it++) {
118       if (bf.test(*it)) {
119         if (base_set.find(*it) != base_set.end()) {
120           count++;
121         }
122       }
123     }
124   }
125   assert(!count);
126   return seconds;
127 }
128 int main(int argc, char* argv[]) {
129   int c;
130   while ((c = getopt(argc, argv, "dfae:n:x:l:k:b:m:rhv:")) != -1) {
131     switch (c) {
```

```
132         // Which experiments to run:
133         case 'd': run_density_experiment = true; break;
134         case 'f': run_false_positive_experiment = true; break;
135         case 'a': run_application_experiment = true; break;
136 
137         // Parameters to the experiment
138         case 'e': desired_epsilon = std::strtod(optarg, nullptr); break;
139         case 'n': num_strings_in_set = std::strtol(optarg, nullptr, 0); break;
140         case 'x': num_strings_not_in_set = std::strtol(optarg, nullptr, 0); break;
141         case 'l': length_of_strings = std::strtol(optarg, nullptr, 0); break;
142         case 'k': num_hash_functions = std::strtol(optarg, nullptr, 0); break;
143         case 'b': num_bits_per_value = std::strtod(optarg, nullptr); break;
144         case 'm': num_bits_in_filter = std::strtol(optarg, nullptr, 0); break;
145         case 'r': rounding_constant = 1; break;
146 
147         // Meta-parameters
148         case 'h': print_help = true; break;
149         case 'v': verbosity_level = std::strtol(optarg, nullptr, 0); break;
150       }
151     }
152     if (!run_density_experiment && !run_false_positive_experiment && !run_application_experiment) {
153       printf("Error: no requested experiment!\n");
154       return 1;
155     }
156     if (desired_epsilon < 0 || desired_epsilon >= 1) {
157       printf("Error: desired_epsilon must be between 0 and 1, got %f\n", desired_epsilon);
158       return 1;
159     }
160     if (num_strings_in_set < 0) {
161       printf("Error: num_strings_in_set must be non-negative, got %d\n", num_strings_in_set);
162       return 1;
163     }
164     if (length_of_strings <= 0) {
165       printf("Error: length_of_strings must be positive, got %d\n", length_of_strings);
166       return 1;
167     }
168     if (num_bits_per_value < 0) {
169       printf("Error: num_bits_per_value must be positive, got %f\n", num_bits_per_value);
170       return 1;
171     }
172     if (run_false_positive_experiment || run_application_experiment) {
173       if (num_strings_not_in_set <= 0) {
174         printf("Error, for these experiments num_strings_not_in_set must be positive, got %d\n",
175                num_strings_not_in_set);
176         return 1;
177       }
178     }
179     if (verbosity_level != 0) {
180       print_driver_state();
181     }
```

```
182     if (num_bits_per_value == 0.0) {
183       // Specify default to be "optimal".
184       num_bits_per_value = -1.44 * std::log2(desired_epsilon);
185     }
186     if (num_bits_in_filter == 0) {
187       num_bits_in_filter = num_strings_in_set * num_bits_per_value;
188     }
189     if (num_hash_functions == 0) {
190       if (verbosity_level >= 1) {
191         printf("Ideal number of hash-functions pre-rounding: %f\n", -std::log2(desired_epsilon));
192       }
193       num_hash_functions = size_t(-std::log2(desired_epsilon)) + rounding_constant;
194     }
195     if (verbosity_level >= 1) {
196       printf("Calculated best bits per value: %f, total number of bits: %ld, number of hash functions: %
197               num_bits_per_value, num_bits_in_filter, num_hash_functions);
198     }
199     auto total_strings = num_strings_in_set + num_strings_not_in_set;
200     strings = create_strings(total_strings, length_of_strings);
201     included_string_begin = std::begin(strings);
202     included_string_end = included_string_begin+num_strings_in_set;
203     excluded_string_begin = included_string_end;
204     excluded_string_end = excluded_string_begin+num_strings_not_in_set;
205     assert(excluded_string_end == std::end(strings));
206 
207     if (verbosity_level > 1) {
208       printf("will add %zu strings, test %zu strings",
209         std::distance(included_string_begin, included_string_end),
210         std::distance(excluded_string_begin, excluded_string_end));
211     }
212 
213     bf = bloom_filter<std::string, string_hash>(num_bits_in_filter, num_hash_functions);
214     // Populate our Bloom filter set and a baseline std::set.
215     std::for_each(included_string_begin, included_string_end,
216                   [](auto &s) {
217                     bf.set(s);
218                     base_set.insert(s);
219                   });
220     if (run_density_experiment) {
221       const double density =
222         static_cast<double>(bf.count()) / static_cast<double>(num_bits_in_filter);
223       printf("%f %zu\n", density, bf.size());
224     }
225     if (run_false_positive_experiment) {
226       auto false_positives = std::count_if(excluded_string_begin, excluded_string_end,
227                                            [](auto &v) { return bf.test(v); });
228       double fp_rate = double(false_positives) / double(num_strings_not_in_set);
229       printf("%f\n", fp_rate);
230     }
231     if (run_application_experiment) {
```

```
232       double base_seconds = time_baseline();
233       double bloom_seconds = time_bloom_filter();
234       printf("%f %f\n", base_seconds, bloom_seconds);
235     }
236   }
```