University of Alberta

GEOMETRIC FILTER: A SPACE AND TIME EFFICIENT LOOKUP TABLE WITH BOUNDED ERROR

by

Yang Zhao

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

Fall 2009
Edmonton, Alberta

Examining Committee

This thesis is dedicated to my parents Dongping Zhao and Zhonghui Yang.

Abstract

Lookup tables are frequently used in many applications to store and retrieve key-value pairs. Designing efficient lookup tables can be challenging with constraints placed on storage, query response time and/or result accuracy.

This thesis proposes Geometric filter, a lookup table with a space requirement close to the theoretical lower bound, efficient construction, fast querying speed, and guaranteed accuracy. Geometric filter consists of a sequence of hash tables, the sizes of which form a descending geometric series. Compared with its predecessor, Bloomier filter, its encoding runs two times faster, uses less memory, and it allows updates after encoding.

We analyze the efficiency of the proposed lookup table in terms of its storage requirement and error bound, and run experiments on Web 1TB 5-gram dataset to evaluate its effectiveness.

Acknowledgements

First of all, I would like to thank my co-supervisors Dr. Davood Rafiei and Dr. Mario Nascimento, for their help and support of my research. On our research meetings, I learned not only how to do research, but also how to solve problems in a systematic and scientific way. I thank my co-supervisors for providing me the freedom in my research, which made the innovative research an enjoyable process to me. I also highly appreciate their great effort in reviewing my thesis and making it an excellent research work.

I would like to thank Dr. Marek Reformat, and Dr. Mohammad Salavatipour, for their valuable time, and constructive comments on my thesis. And, I also would also like to thank Dr. Paul Lu, and the Lab Instructor Stef Nychka, for their kindness and patience in helping me in my TA.

I would also like to thank my friends in the master program, especially Pirooz Chubak, Vahid Jazayeri, Reza Sherkat and Mojdeh Heravi, for their help in my daily research work. Thanks to my classmates, Richard Zhao, Ci Song, and Qin Dou, for providing me the valuable suggestions in the course work of the first year. And, thanks to Chonghai Wang, for providing me a lot of help when I first arrived in Edmonton.

Thanks to the Department of Computing Science at the University of Alberta, for providing me an excellent opportunity to take part in advanced research in Computer Science.

Table of Contents

List of Tables

List of Figures

List of Symbols

A	a hash table
M	number of locations in hash table A
k	number of hash functions in the hash table
h_j	the jth hash function of A, where $j \in \{1, 2, \ldots k\}$
N	number of hash tables in the Geometric filter
A_i	the ith hash table of the Geometric filter, where $i \in \{1, 2, \ldots N\}$
M_i	number of locations in hash table A_i, where $i \in \{1, 2, \ldots N\}$
$h_{i,j}$	the jth hash function of hash table A_i, where $i \in \{1, 2, \ldots N\}$, and $j \in \{1, 2, \ldots k\}$
x	a key
$v(x)$	the true value of key x
$g(x)$	the value of x decoded from the lookup table
$f(x)$	fingerprint of x
$\langle x, v(x) \rangle$	an item in the dataset, which consists of a key x and its true value $v(x)$
S	a dataset
$\|S\|$	number of tokens in a dataset S
S_i	if $i \in 1, 2, \ldots N$, it means the set of items tried on hash table A_i, $S_0 = S$; if $i = N + 1$, it means the set of items recorded in the overflow table
$\|S_i\|$	number of items in S_i
p	load factor, the fraction of occupied locations in a hash table
r	storage ratio, for hash table A_i, $r = M_i / \|S_i\|$
d	remainder rate of the Geometric filter
E	number of error bits in each hash location
V	number of value bits in each hash location
c	number of bits in each hash location, where $c = E + V$
e	mathematical constant used in natural exponential function $y = e^x$, the value of which is $2.71828\ldots$
e_o	overall false positive rate of the whole lookup table
e_h	false positive rate of each hash table in the Geometric filter
e_n	false positive rate of each location in the hash table

Chapter 1

Introduction

In this chapter, we introduce the problem addressed in this thesis and a few applications that can benefit from our work.

1.1 Problem definition

Lookup tables are widely used by various applications. Storing a small lookup table is straightforward. However, maintaining a relatively large lookup table in limited memory space is not trivial and has been the subject of past and current research. Our reference to lookup tables includes every binary relation in the form of $\langle key, value\rangle$. key is used for uniquely identify an item; it can be a gene sequence, an IP address, or generally a string of characters or bits. $value$ can be a boolean value, a numeric value, or any other data associated to the key.

There are numerous standard lossless compression algorithms that can be applied to lookup tables, compressing them to a small size; however, these algorithms are generally unsuitable for searching, because one needs to uncompress the whole data or part of it to access one particular item. Most applications need to have a fast direct access to data in the lookup tables, and some lossy compression algorithms can be introduced to achieve this goal.

Although requirements may vary from application to application, here are typical requirements for lookup tables:

- Compact Storage. Because of the high volume of data needed by applications, the lookup table has to be compact to fit into the limited storage space, typically the main memory.

- Fast access speed. The lookup table is usually a very frequently accessed component, therefore its query time should be short and ideally constant.

- Enough precision. The lossy compression will likely cause errors in the lookup table's querying result; however the probability of an error must be small enough so that the application's performance and accuracy are not seriously affected. Sometimes, the application also requires the lookup table's error to be one-sided. As an example, in a virus detecting program, erroneously reporting virus pattern to be normal would cause enormous consequence, but reporting a normal pattern to be a suspicious virus is sometimes acceptable. The requirement on the lookup table's precision is application dependent.

- Fast and easy build. Although building a lookup table can be done off-line, the building procedure should not take too much resources, e.g. memory; otherwise the building procedure may become a bottleneck.

- Updatability. Some applications need to change the lookup table's values after building the table, therefore the lookup table should have some flexibility. For example, many streaming data applications handle their data by a time window: the recent data is most important, while the older data is relatively less important and will finally be forgotten.

1.2 Lookup table variations and applications

Different applications have different requirements, hence the implementations of lookup tables would be different from one application to another. The simplest implementation is to record all the data in an array list and perform linear search. An array list with linear search can be used in a small dataset, but its performance on very large datasets would be too slow for real-time applications. In this section, we provide some applications with more sophisticated implementations.

1.2.1 Bloom filter for membership lookups

Bloom filter is a data structure designed to support set membership queries [2]. In a lookup table for membership queries, $value$ is a boolean value that represents the key's existence in the set, 1 for existing and 0 for non-existing. Let D be the universe of keys, and $S \subseteq D$. Given an element $x \in D$ as the key, we define its $value$ as a function $v(x)$:

$$v(x) = \begin{cases} 1 & x \in S \\ 0 & otherwise \end{cases}$$

The function $v(x)$ can be implemented via Bloom filter. The set S can be encoded into the Bloom filter, and after the encoding, the Bloom filter will be able to answer membership queries about set S, e.g. correctly evaluating $v(x)$. Bloom filter introduces a small false positive rate, but its memory space usage is lower than the theoretical lower bound for lossless data compressions, and its querying time is constant [4]. Bloom filter, discussed in more details in Section 2, is widely used in database and network applications where space is precious and speed is important.

Differential file

Differential file is a typical example that may use a Bloom filter to save running time. Most database systems need to handle interactive queries and modifications from multiple users. For example, an on-line ticketing system may query the database to get information about available seats, modify the database to reserve a seat, then let the buyer pay for the ticket and end the transaction. Although concurrency and on-line update are typical features of modern databases, implementing these two features on a very large database brings difficulties to ensuring the database accuracy, integrity, and recoverability [12] [20]. As a result, differential files are proposed as a way to ease these difficulties.

With differential files, typically the database system consists of two files, a main datafile and a differential file. The main datafile is typically large and contains the major part of the data. The main datafile is updated periodically, and during the intervals between two updates, the newly arrived data will be stored into the differential file, which is typically smaller than the main datafile [14]. At querying, the database system may check whether the data item being queried has changed or not. If it has changed, the system will search for the data in the differential file, if not, the search will be conducted in the main datafile. This two-file system can solve the difficulty of concurrent on-line updates [27] [23].

To efficiently detect whether a data item has changed, a Bloom filter may be used. When an update is recorded into the differential file, the key of the modified data item will be encoded into the Bloom filter. During the querying, the key is first queried in the Bloom filter. If the Bloom filter returned 1, we consider the item to be modified, and search for it in the differential file; otherwise we search for the item in the main datafile.

Bloom filter has a small false positive rate, which means an item unchanged could be erroneously reported as modified. In this case, we search for the item in the differential file and discover that it does not exist, then we search again in the main datafile. With a small probability of false positive, this kind of errors do not significantly increase the running time, and they do not affect the correctness of the queries.

Network-based applications

In applications where data is distributed in multiple nodes, Bloom filters are typically used as a compact storage for membership information. Instead of transmitting a complete member list, a Bloom filter may be sent, reducing the overhead of network communication.

One example is semi-join in distributed databases, which can be described as following: list the oil prices in towns all around the world where the yearly living expenditure is greater then 10000 dollars. A straight forward method to solve this problem contains two steps: 1) search the global income database for a set of all the towns with annual expenditure more than 10000 dollars; 2) search the oil price database for the oil prices of the towns appeared in the set of step 1). If the two databases are stored on different machines, then the semi-join needs to transmit the town set on the network. Given the large number of towns all over the world, transmission of the town set on the network would dramatically increase the cost of network transmission and compromise the running speed of the query [3] [21] [17].

A solution to this problem is to encode the town set into a Bloom filter, then transmit the Bloom filter on the network, instead of the original town set. And, on step 2), to query each town in the Bloom filter to determine whether it is in the set. Because Bloom filter uses much less space than the uncompressed member list, and its querying time is much shorter than database searching, the running time and network communication overhead can be greatly reduced [34] [18].

As another example, in peer-to-peer network, each node may maintain an array of Bloom filters corresponding to the nodes adjacent to it, and each Bloom filter may represent the resources owned by an adjacent node. Requests for resources can be processed by querying the node's own Bloom filters or forwarding the request to adjacent nodes iteratively [11] [7]. This resource locating method is called Gossip Protocol, and can be applied in some peer-to-peer applications, e.g. [25] [24] [28].

Whitaker and Wetherall proposed to use Bloom filter to avoid forwarding loops in Internet routing. With a small bloom filter in the packet, we can record the routers have been passed during the transmission; if the packet came to a router for the second time, the Bloom filter will return 1, hence the forwarding loop is detected and the packet is deleted instead of forwarding to the next hop [36].

Frequency lookup tables

Bloom filter can also be used in statistical language modeling to support a frequency lookup table, where key is a string consisting of a sequence of words, and $value$ is the string frequency.

In this application, $value$ is an integer greater or equal to zero; however, Bloom filter can only store membership information. Talbot et al. propose changes to a Bloom filter to answer queries about frequency [30]. The solution is based on the fact that if a key x has value $v(x) = n$, then we have $x \in \{x|v(x) \geq n\}$, $x \in \{x|v(x) \geq n-1\} \ldots$, and $x \in \{x|v(x) \geq 1\}$. With this change, a frequency query is equivalent to a series of membership queries.

Variations of Bloom filter

In the classic Bloom filter, the hash table is an array of bits, and the queries can only return a binary flag. But, there are variations of Bloom filters, such as Stable Bloom filter [8], and Spectral Bloom filter [6], etc. These variations enlarge the hash table from an array of bits to an array of integers, therefore the range of values is also enlarged. For example, the counting Bloom filter [10] can be used in counting the number of the keys or other value-based operations. However, these variations cannot directly encode the $value$ into the hash table. Even if a match is found, the returned $value$ approximates the true $value$, but is not necessarily equal to the true $value$.

1.2.2 Bloomier filter

Bloomier filter is another extension of the Bloom filter, where each cell can be a number other than 0 or 1. Unlike the Bloom filter, the Bloomier filter directly encodes and decodes the data. Despite of a small false positive rate, for keys stored in a Bloomier filter, the returned $value$ exactly matches the true $value$ [5]. Because of its direct encoding and decoding, Bloomier filter can concisely represent more complex data with non-binary values.

Frequency lookup with direct encoding and decoding

Talbot and Brants use the Bloomier filter in statistical language modeling to store the frequency of n-grams [29]. Despite a slow off-line encoding, the filter responds to a query almost instantly. As a frequency lookup table, the only significant drawback of Bloomier filter is that the data in hash table cannot be modified once it is encoded.

Chazelle et al. propose a method to allow updates into the Bloomier filter after its initial encoding [5]. In their proposal, a separate memory slot is allocated for each key encoded. Instead of directly encoding the $value$ of the key into the hash table, Bloomier filter encodes the address of the slot that holds the $value$. After encoding, although the hash table cannot be changed, the slot referenced by the address can still be changed. The drawback of this methods is that it almost doubles the memory usage of the Bloomier filter. Since Bloomier filter is usually used when memory is precious, this drawback makes the updatable Bloomier filter impractical for many scenarios.

FPGA-based Pattern-matching

In virus and spam detection, incoming data may be checked in real time against threats. The scan is actually a pattern matching, where the data within a time window may be compared to patterns in a lookup table, and if a match is found, then the data within this window is potentially harmful [15].

Because of the high data transmission rate, the scan must be very fast to avoid a system bottleneck [33]. This fast speed is implemented by a programmable semiconductor device called field-programmable gate array (FPGA). A FPGA is a chip that can be programmed by a customer after manufacturing, and it provides a fast and convenience platform to implement high performance logic functions on hardware [32].

Because the memory of FPGA is fast but small, the pattern lookup table has to be very compact, hence Bloom filter or Bloomier filter can be used for storage and lookup. And, because both Bloom and Bloomier filters are lossy compression algorithms, they have a certain false positive rate. After finding a potential match, an exact match may be needed to verify the potential matching and eliminate false positives.

Ho et al. suggests that Bloomier filter is better than Bloom filter in this application [15]. A Bloomier filter not only can detect a match, but it also return the ID number of the matched pattern; this pattern can be directly retrieved and the exact matching can be performed immediately. However, a Bloom filter can only find a match, without any further information about which pattern is matched. To determine the full pattern, one may do a pattern matching by a Finite State Machine (FSM), for example using Aho-Corasick algorithm [15]. The FSM exceeds the capacity of on-chip memory, therefore the exact matching runs on off-chip memory, which is much slower. Bloomier filter can provide the ID of the full pattern, and the FSM is not needed [16]. Generally speaking, Bloomier filter's greatest advantage over Bloom filter is its ability to represent complex data exactly.

1.3 Thesis overview

In Chapter 2, we will provide a detailed analysis of a few lookup table algorithms related to our research, including Bloom filter and Bloomier filter. In Chapter 3, we will introduce Geometric filter. In the same chapter, we analyze the Geometric filter's performance and describe its configuration procedures. In Chapter 4, we evaluate our Geometric filter on Google 5-gram dataset. Based on our experimental results, we also provide an analysis of the Geometric filter in the presence of the nonuniform hash functions.

The main contribution of this thesis is the Geometric filter. Geometric filter overcomes the drawbacks of the Bloomier filter while maintaining most of its advantages. The improvements include the following aspects: 1) the encoding speed is almost two times faster than Bloomier filter; 2) the encoding procedure does not use any additional memory beside its hash tables, while in Bloomier filter the encoding requires additional memory almost 7 times of its hash table; 3) the hash tables can be updated after encoding, while Bloomier filter does not allow modification to its hash table after encoding.

Chapter 2

Related Work

In this chapter, we present Bloom filter and Bloomier filter, the two lookup tables closely related to our Geometric filter. Background knowledge about applications will be included when necessary.

2.1 Bloom filter

Bloom filter is designed to support set membership queries [2]. But it can also be used to represent a general lookup table, if the lookups can be transformed to membership queries, as discussed in Section 2.1.2. In this section, we provide detailed introduction and analysis of the Bloom filter.

2.1.1 The algorithm

Bloom filter uses a bitmap to represent the membership of keys in a set. An empty bloom filter is an array of bits all initialized to zero. The Bloom filter has k hash functions. When encoding a set S, every element in S is mapped to k locations by the k hash functions, and the bits in those locations are set to one. When querying for an element x, x is also mapped to k locations by the same hash functions. According to the encoding rule, if x is encoded as an element of S, all the k locations should be one. If any one of the k locations is zero, x could not have been encoded before [2].

Bloom filter is a lossy compression algorithm. False positives can happen during the querying, because for an element x unencoded, its k hash locations may be set to one by other encoded elements and the querying algorithm will return $v(x) = 1$ although $x \notin S$. However, false negatives are impossible in Bloom filter, because locations changed to one can never be changed back to zero. In this case, Bloom filter error is said to be one-sided.

2.1.2 Bloom filter for general non-binary lookups

The classic Bloom filter only answers queries about set membership. Talbot and Osborne [30] propose a modified Bloom filter algorithm which transforms non-binary lookups into a series of set membership queries.

The authors apply their algorithm to statistical language modeling, where key is a string, and $value$ is the number of times the key is observed in the sample corpus, if the key does not exist in the sample corpus, its $value$ is zero. Consider a key x such that its value $v(x) = n$, then $v(x) \geq n$ is true, and for any integer $m \in [1, n]$, $v(x) \geq m$ is also true. We pair x with m to form a binary relation $\langle x, m\rangle$, then the encoding set $S = \{\langle x, m\rangle | v(x) \geq m \ is \ true\}$, and $\langle x, n\rangle \in S$ implies that for any integer $m \in [1, n]$, $\langle x, m\rangle \in S$. Based on this observation, Talbot and Osborne encode an item $\langle x, v(x)\rangle$ by pairing x with an integer m to form an encoding element $\langle x, m\rangle$, where m iterates from 1 to $v(x)$.

Equivalently, each query lookup is also an iterative procedure. In each iteration, the queried key x is paired with a number m to form a querying element $\langle x, m\rangle$, where m begins from 1 and increases by 1 after each iteration. If the Bloom filter returns 1 for $\langle x, m\rangle$ and 0 for $\langle x, m + 1\rangle$, then the iteration stops and m will be returned as the value of x.

The new algorithm has false positive errors happening in two ways: 1) for an unencoded key, if its hash locations are set to one while encoding other keys, the returned value will erroneously be non-zero; 2) for an encoded key x, the elements $\langle x, v(x)+1\rangle$, $\langle x, v(x)+2\rangle, \ldots$ may not be encoded, but the hash locations for these elements could be set to one while encoding other elements, and the return value for x can be larger than $v(x)$. However, the error can still be treated as one-sided, because for any key, the return value cannot be lower than its correct value.

2.1.3 Log-frequency Bloom filter

Talbot and Osborne soon discover a problem in their modification to the Bloom filter. In statistical language modeling, in particular, many keys in n-gram have very large frequencies (e.g. 10^5 or even more), and for these keys, Bloom filter is inefficient because there are too many iterations for encoding and querying. As an improvement, Talbot and Osborne propose Log-frequency Bloom filter [31], where the $value$s are quantized logarithmically. For a key x, the value is $v(x)$, and the quantized value is $q(v(x)) = 1 + \lfloor \log_b v(x) \rfloor$, where base $b > 1$. In encoding and querying, instead of iterating from $\langle x, 1 \rangle$ to $\langle x, v(x) \rangle$, the algorithm iterates from $\langle x, 1 \rangle$ to $\langle x, q(v(x)) \rangle$, then the running time and space usage is significantly reduced.

Log-frequency Bloom filter has the following advantages: 1) data can be compactly compressed and the space usage falls significantly below the lower bound of lossless compression algorithms; 2) the running time of querying is almost instantaneous; 3) the data can be efficiently encoded without any preprocessing.

Log-frequency Bloom filter also has a disadvantage: for an encoded key, false positives in log-frequency may lead to a large error in the recovered value. One usually needs to use an exponential function to recover the original value from the log-frequency, and a small error in the log-frequency would result an order of magnitude error in the recovered value. If $v(x)$ is large, the difference could be too large to be acceptable. For example, if $q(v(x)) = 1 + \lfloor \log_{10} v(x) \rfloor$, and $v(x) = 10^4$, then $q(v(x)) = 5$. However, if a false positive happened and the returned log-frequency is 6, then the error would be 9×10^4.

In order to overcome this drawback of the Bloom filter, Chazelle et al. propose Bloomier filter [5], as discussed next.

2.2 Bloomier filter

A Bloomier filter generalizes the Bloom filter, in that data can be encoded directly without any iteration [5]. The encoded data is a binary relation $\langle key, value \rangle$, where $value$ can be of any type even a string. Because no quantisation is needed, there is no restriction on the physical meaning of the $value$.

2.2.1 The algorithm

In the Bloomier filter, the hash table is expanded from a bit array to an array of integers. Each location of the hash table is a cell containing c bits, hence the unsigned integer stored in the cell is in the range of $[0, 2^c - 1]$. Bloomier filter has k hash functions ($h_1, h_2, \dots h_k$). With these hash functions, a key can be mapped to k locations in the hash table. Bloomier filter has another hash function called the fingerprint function $f()$, and this function maps each key x to a number in $[1, 2^c - 1]$. It is generally assumed that for any key x, $h_1(x)$, $h_2(x)$, $\dots h_k(x)$ and $f(x)$ are uniformly distributed and are independent from each other.

In the Bloomier filter, a key x is associated with its value $v(x)$ by the function in Eq. 2.1, where $\otimes$ means bitwise OR (or XOR); $\bigotimes$ means a series of $\otimes$ calculations; A is the hash table; and $g(x)$ is the return value. After encoding, we should have $v(x) = g(x)$, which means the return value equals to the real value.

$$\begin{aligned} g(x) &= f(x) \otimes (A[h_1(x)] \otimes A[h_2(x)] \dots \otimes A[h_k(x)]) \\ &= f(x) \otimes (\bigotimes_{i=1}^{k} A[h_i(x)]) \end{aligned} \tag{2.1}$$

Bloomier filter also allows false positives. In order to reduce false positive rate, Bloomier filter introduces error bits. From the c bits of each cell, E bits are used as error bits. When querying an unencoded key, the probability of false positive is $e_o = 1/2^E$, and this probability is also called the false positive rate [29].

The encoding of the Bloomier filter must satisfy two constraints: 1) After encoding a key, all the hash locations referenced by this key will be marked as occupied and cannot be changed. Otherwise, if a referenced hash location is changed after encoding, the return value of the keys that reference that hash location will also be changed and is no longer equal to the real value. 2) When encoding a key, at least one of the k hash locations must remain unoccupied. This unoccupied location is called the critical location. If $h_j(x)$ denotes the critical location, then:

$$A[h_j(x)] = v(x) \otimes f(x) \otimes (\bigotimes_{i=1 \cap i \neq j}^{k} A[h_i(x)]) \tag{2.2}$$

If all k locations are already occupied, then the key x cannot be correctly encoded, because encoding x needs to change at least one location. In Bloomier filter, this situation is referenced to as a “hash collision”.

2.2.2 Bipartite matching: preprocessing the dataset before encoding

In order to satisfy the two constraints of encoding, the algorithm needs to preprocess the dataset, and find a good encoding sequence.

The problem of finding an encoding sequence can be solved by finding an ordered matching in a bipartite graph [29], described as Algorithm 1. The bipartite matching algorithm selects the one-degree nodes on the right-hand-side and deletes all the edge that is adjacent to the node selected. The order of selection is a bipartite matching and can be used as a good sequence for Bloomier filter encoding. Algorithm 1 is an approximation, because its failure in finding a bipartite matching does not necessarily mean there is no bipartite matching exist. This approximation algorithm succeeds with high probability when $k = 3$ and the hash table size $M \geq 1.23|S|$. When $k \neq 3$, M needs to be larger than $1.23|S|$ for the bipartite matching to succeed [19] [13] [22]. Therefore, Bloomier filter's hash table size M is at least $1.23|S|$, regardless of k.

The running time of Algorithm 1 is $O((k+1) \times |S| + M)$. This is the number of edges and nodes that must be deleted before the algorithm returns a match, assuming that there is a match. Since there are $k \times |S|$ edges, $|S|$ LHS nodes and M RHS nodes, the total deletion time is $O(|S| + k \times |S| + M) = O((k+1) \times |S| + M)$. In a typical configuration, $k = 3$ and $M = 1.23|S|$, the bipartite running time is $O((3+1) \times |S| + 1.23|S|) = O(5.23|S|)$.

The memory needed for the bipartite matching is $O((2k+1) \times |S| + M)$. Suppose a vertex in the bipartite graph (either on LHS or RHS) takes 1 unit of space, and an edge takes 2 units. Then, the memory needed for storing the bipartite graph is $O(|S| + 2 \times k \times |S| + M) = O((2k+1) \times |S| + M)$. When $k = 3$ and

Algorithm 1: Ordered Matching

```
input  : $S$: n-gram dataset;
         $h_j, j \in \{1, 2, \ldots k\}$: the $k$ hash functions;
         $M$: number of locations in the hash table;
output: on success, return $matched$: the ordered matching;
         on failure, return FAIL;

         /* $degree\_one$ means a set containing one-degree vertices on RHS of
         the bipartite graph; $r2l_i$ means the $i$th vertex on RHS and $l2r_i$ means
         the $i$th vertex on LHS. */

$matched \leftarrow \emptyset$
forall $i \in \{1, 2, \ldots k\}$ do
    $r2l_i \leftarrow \emptyset$
forall $x_i \in S$ do
    $l2r_i \leftarrow \emptyset$
    forall $j \in \{1, 2, \ldots k\}$ do
        $l2r_i \leftarrow l2r_i \cup h_j(x_i)$
        $r2l_{h_j(x_i)} \leftarrow r2l_{h_j(x_i)} \cup x_i$
forall $i \in [0, M-1]$ do
    if $|r2l_i| = 1$ then
        $degree\_one \leftarrow i$
while $|degree\_one| \geq 1$ do
    $rhs \leftarrow$ POP $degree\_one$
    $lhs \leftarrow$ POP $r2l_{rhs}$
    PUSH($lhs, rhs$) onto $matched$
    forall $rhs` \in l2r_{lhs}$ do
        POP $r2l_{rhs`}$
        if $|r2l_{rhs`}| = 1$ then
            $degree\_one \leftarrow degree\_one \cap rhs`$
if $|matched| = |S|$ then
    return $matched$
else
    return FAIL
```

$M = 1.23|S|$, the memory requirement is $O((2 \times 3 + 1) \times |S| + 1.23 \times |S|)$ = $O(8.23|S|)$. When the bipartite matching is finished, the ordered sequence is stored in the LHS vertices. Majewski et al. provides a hypergraph data structure for the bipartite graph [9] [19], but considering the space of the stack to store the ordered sequence, the memory usage is the same.

2.2.3 Strengths and weaknesses of the Bloomier filter

The most obvious advantage of Bloomier filter for storing data with general $\langle key, value\rangle$ pairs is its direct encoding and decoding. Specifically, Bloomier filter is better than Bloom filter in the following aspects: 1) the running time of encoding depends only on the dataset size $|S|$; 2) for any key, the querying time is constant, and takes only a few CPU circles. Because the $value$ is decoded exactly the same as it is encoded, there is no restriction on the encoded $value$s, or their distribution, etc.

Bloomier filter also has the following drawbacks.

- Auxiliary memory. Running a bipartite matching has some space overhead. As discussed before, the memory needed by the bipartite matching is at least $8.23|S|$, which is almost 7 times the memory needed by the hash table. Although the memory space used to store the encoded Bloomier filter is relatively small, the auxiliary memory needed for the bipartite matching can easily exceed the capacity of main memory. The bottleneck for the size of encoding dataset is actually the bipartite matching, because the dataset has to be small enough so that the bipartite matching can run within the available memory space. In this case, large datasets need to be divided into smaller ones to be encoded separately and then organized by an index. The dataset division and indexing can increase both the complexity and the running time of the algorithm.

- Inflexibility. Bloomier filter does not allow any update to a location in the hash table after that location is occupied by a key. This is a big constrain in encoding more dynamic datasets. Although Bloomier filter can efficiently support static lookup tables, its support to dynamic tables comes with a high price of space usage. Instead of encoding the data into the hash table, the algorithm needs to allocate one extra slot of memory for each key, then encode the address of the slot into the hash table [5]. With this change, the memory usage (which is already high) doubles.

- Running time. The running time of bipartite matching is $O((k+1) \times |S| + M)$; given $k = 3$ and $M = 1.23|S|$, the running time is $O(5.23|S|)$. When encoding large datasets, the running time of bipartite matching will become significant, and the bipartite matching actually takes more time than the encoding procedure following it.

In the next chapter, we will introduce our Geometric filter. Geometric filter overcomes the limitations of the Bloomier filter while maintaining almost the same or better encoding and querying speed.

Chapter 3

Geometric filter

A major drawback of the Bloomier filter is its bipartite matching at encoding. In order to overcome this and other drawbacks discussed in the previous chapter, we propose a new data structure, called Geometric filter. Instead of only one hash table, Geometric filter has a sequence of hash tables, and the sizes of these hash tables form a descending geometric series. Geometric filter avoids the bipartite matching by resolving hash collisions when they happen. The data to be encoded is tried on the hash tables sequentially; if a collision happens and the data cannot be encoded into one hash table, it is tried on the subsequent hash tables until it is encoded.

Our presentation in this chapter is divided into three parts. In the first part, for the ease of presentation, we present a more *explicit* version of Geometric filter with some additional bitarrays. In this case, each hash table is accompanied by two bitarrays to indicate the status of every hash location. Introducing these two bitarrays makes it easy to explain the basic ideas of the algorithm. In the second part, we present a more *compact* version of Geometric filter, which is based on the same ideas, and offers the same functionality as before; however, the information represented by the two bitarrays is integrated into the hash table. In the third part, we provide a theoretical analysis of Geometric filter, and a discussion of its parameter settings.

3.1 Geometric filter with bitarrays

A Geometric filter consists of a sequence of N hash tables, denoted as A_i, $i = 1, 2, \ldots N$. Let M_i denote the size of hash table A_i. In our initial presentation, every hash table A_i is also accompanied by two bitarrays, a placement bitarray $placement_i$, and a validation bitarray $validation_i$. For every hash table A_i there are k hash functions, denoted as $h_{i,j}$, $j = 1, 2, \ldots k$. For simplicity of analysis, we assume these hash functions to be uniformly distributed and independent from each other.

There is also an overflow table to store the data items that cannot be encoded into any one of the N hash tables. For simplicity, we assume the overflow table is implemented as a sequential file with key-value pairs. This overflow table is not as compact as the hash tables, and searching the overflow table can be slower than the hash tables. However, with a proper parameter setting, as discussed in Section 3.4.2, only a small portion of data items will be recorded in the overflow table, and the probability of querying the overflow table is also very small; therefore, there is no significant influence on the overall performance.

3.1.1 Data encoding

When encoding an item $\langle x, v(x) \rangle$, the key x will be tried on the hash tables sequentially, starting from A_1 and continuing up to A_N (if needed). In each hash table, x is mapped to k locations and is tried on these hash locations sequentially. If there is one or more locations available, key x and its value $v(x)$ will be encoded into the first available location. In Geometric filter, "hash collision" means all the k locations corresponding to x are already occupied, hence the item cannot be encoded into this hash table. If a hash collision happens, x will be tried on the next hash table. Once x has been tried on all the N hash tables, and none of the hash tables contains any available location for x, then the item $\langle x, v(x) \rangle$ is added into the overflow table.

As introduced above, Geometric filter is different from Bloomier filter in its method to deal with hash collisions. Bloomier filter needs to prevent any hash collision from happening, but Geometric filter can solve the hash collisions once they happen. Bloomier filter has only one hash table, therefore if a hash collision happens, x would not be able to find any other location for encoding. Bloomier filter resolves the problem by running a computationally expensive bipartite matching to prevent any hash collision from happening. Geometric filter has more than one hash tables, and if one hash table does not contain any available location, x can still be tried on the subsequent hash tables in the series, hence it is likely that x will find a space in those hash tables.

The placement bitarray is used to find available locations in a hash table. Every hash table has a placement bitarray attached to it. When the hash table is empty, the placement bitarray is initialized to all zeros; when a location is occupied, its corresponding bit is set to one. By testing the corresponding bit in the placement bitarray, we can know whether a location is occupied or not.

Geometric filter is also different from classic Bloomier filter in its method to encode an item into a hash table. In order to encode or decode an item, Bloomier filter uses the contents of all the k hash locations, and the $value$ to be retrieved is calculated by a function shown in Eq. 2.1. However, Geometric filter uses only one location, which is the first unoccupied location among the k hash locations. The rest of the locations will not be touched and will remain in their original status, either occupied or empty.

Consider an element x and let $A[h(x)]$ be its first unoccupied location, after encoding, the value of the location is set to $f(x) \otimes v(x)$ as Eq. 3.1 shows, where $f(x)$ is the fingerprint x. At the same time, the corresponding bit in the placement bitarray is also set to one.

$$A[h(x)] = f(x) \otimes v(x) \tag{3.1}$$

3.1.2 Error bits, value bits and fingerprints

The bits in each location of the hash table can be divided into two parts: the error bits, and the value bits. Let E denote the number of error bits, V denote the number of value bits, and $c = E + V$. Given V value bits, the value $v(x) \in [0, 2^V - 1]$.

The key-value pair $\langle x, v(x)\rangle$ is not directly encoded into Geometric filter, because the length of x is often too long for the hash tables. Instead, we combine x's fingerprint $f(x)$ with x's value $v(x)$ to form a binary number that will be stored in the hash location. The fingerprint function maps x to a number $f(x) \in [0, 2^c - 1]$, which is typically more compact. Converting x into its fingerprint, however, is a lossy procedure, because multiple keys may be mapped to the same value, and this is the source of false positives.

3.1.3 Decoding data

When querying a key x, Geometric filter tries x on the N hash tables sequentially, from A_1 to A_N. In each hash table A_i, Geometric filter maps key x to k hash locations sequentially from $A_i[h_{i,1}(x)]$ to $A_i[h_{i,k}(x)]$, and tries to retrieve the return value $g(x)$ using Eq. 3.2.

$$g(x) = f(x) \otimes A[h_{i,j}(x)],\ i \in [1, N],\ j \in [1, k] \tag{3.2}$$

For any key x, its real value $v(x) < 2^V$, therefore the correct return value should be smaller than 2^V. In Geometric filter, $g(x) < 2^V$ is used as a condition to judge whether x and its value was encoded into the current location $A[h_{i,j}(x)]$.

Using error bits to identify a token's encoding location is a lossy procedure. The fingerprint takes a value in $[0, 2^c - 1]$ uniformly at random for every x not actually encoded into the current location; therefore the retrieved value $g(x)$ is also a random number uniformly distributed in $[0, 2^c - 1]$. Hence, on each location, the probability of a false positive $P(g(x) < 2^V) = \frac{2^V}{2^c} = \frac{1}{2^{c-V}} = \frac{1}{2^E}$.

If x is a key that was not previously encoded, Geometric filter still needs to try x on a sequence of hash locations, and if any one location in this sequence causes a false positive, x will erroneously get a non-zero value. Obviously, the probability of a false positive in the Geometric filter is higher than the probability of a false positive in one single hash location.

3.1.4 Possible errors in Geometric filter

Geometric filter is a lossy compression algorithm that introduces errors. The errors may happen in two ways: 1) for an unencoded key, the return value is valid such that $g(x) < 2^V$; 2) for an encoded key x the return value is valid but different from the true value, $g(x) \neq v(x)$.

The former error is caused by false positives that happen in the sequence of trials, as discussed in the previous section. With the same number of error bits in each location, Geometric filter's false positive rate would be higher than Bloomier filter; therefore, Geometric filter needs some extra error bits to overcome the additional false positive and achieve the same accuracy as Bloomier filter.

The latter error is also caused by false positive. Now, suppose x is an encoded key. During the querying, x should be tried on a sequence of hash locations until it retrieves a valid value. However, if a value within the valid range is retrieved before the search reaches the actual location where x is stored, then the returned value $g(x)$ would be different from $v(x)$. This kind of error is referred to as "encoded false positive".

3.1.5 Eliminating encoded false positives by re-scan

We cannot prevent false positives, because we cannot predict which keys will be tried on the Geometric filter during querying. However, encoded false positives are preventable, because they can only happen when querying a key within the encoding dataset S. If we scan the dataset S for a second time, we will be able to find out those items that can lead to an encoded false positive, and record them in the overflow table. The size of the overflow table is not significantly changed, because of the small probability of encoded false positives.

In the first scan, items in the dataset are encoded into the Geometric filter according to their sequential order in the dataset. However, when encoding an item $\langle x, v(x) \rangle$, we need to detect whether encoding this item would cause an encoded false positive. This is done by examining the return value $g(x)$ calculated on occupied locations.

For every occupied location tried for x, we calculate $g(x)$ via Eq. 3.2, and in the equation, $A[h_{i,j}(x)]$ is substituted with the value of the occupied location. If $g(x) < 2^V$, then the item previously encoded into this location will cause an encoded false positive, unless $g(x) = v(x)$. For any such location, a bit in the validation bitarray is set to zero, to indicate a potential source of encoded false positives.

With the validation bitarrays, the querying procedure needs to be modified. When querying a key x, x is tried in a sequence of hash locations, and for each hash location we calculate $g(x)$ through Eq. 3.2; however, if on one hash location $g(x) < 2^V$, this $g(x)$ cannot be directly returned as x's value. We need to check the validation bit for this location. If the bit is one, then there is no encoded false positive at this location, and the $g(x)$ can be returned as x's value. However, if the bit is zero, then there is an encoded false positive at this location; in this case, the $g(x)$ cannot be returned as x's value, and we need to search the overflow table for x.

All items that lead to an encoded false positive need to be recorded into the overflow table.

With one scan of the dataset, we are not able to record the previously encoded items, because the encoding function (Eq. 3.1) is a lossy encoding. In this case, we need to scan the dataset for the second time. The second scan is actually a querying test with an empty overflow table; this means every query that needs to search the overflow table will get an invalid return value. For each item $\langle x, v(x)\rangle$ in the dataset that is already encoded, there are only two situations of the query's return value $g(x)$: 1) $g(x) = v(x)$, which means $\langle x, v(x)\rangle$ was successfully encoded; 2) $g(x) \neq v(x)$ which means $\langle x, v(x)\rangle$ was not successfully encoded. The encoding failure could be caused by an encoded false positive or an overflow. But no matter what the reason is, once $g(x) \neq v(x)$, the item $\langle x, v(x)\rangle$ will be recorded into the overflow table. After the second scan, we have constructed the overflow table for actual querying.

3.1.6 Algorithms: Encoding with Re-scan, and Querying

The algorithms for the Geometric filter with bitarrays include: encoding (with re-scan), querying, and also a function that is used in both encoding and querying. In our presentation, we assume the bitarrays are part of the hash table, and if the hash table is an input parameter, then the accompanying bitarrays can also be used inside the procedure.

Algorithm 2 is a function that is invoked inside other algorithms to get the status information. It gives the following status information: 1) whether the location is occupied, 2) whether the validation bit is zero (whether there is a known encoded false positive at this location), and 3) for a given key x, would the retrieved value $g(x)$ be smaller than the possible maximum 2^V. Correspondingly, Algorithm 2 has four possible return values: *Open* , *Match* , *Recoverable-Collision* , and *Unrecoverable-Collision* .

- *Open* means the location is empty; and, the condition of $g(x) < 2^V$ does not apply to an empty location as it does not contain any data.

- *Match* means the location is occupied and $g(x)$ is within the range, i.e. $g(x) < 2^V$. In querying, a *Match* for x means x is encoded into this location, and $g(x)$ can be returned as x's value. However, in the encoding procedure, *Match* means a newly detected encoded false positive. There will be an encoded false positive at the location, if x is encoded. When an encoded false positive is detected, the location needs to be set invalid.

- *Recoverable-Collision* means the location is occupied, and $g(x) \geq 2^V$. In encoding, *Recoverable-Collision* means the location is occupied and x needs to find other locations to be encoded in. In querying, *Recoverable-Collision* means x is not encoded in this location, and we need to go on trying following locations. Validation bitarray is not checked here, because if we already know x is not encoded in this location, x cannot be related to any encoded false positive at this location.

- *Unrecoverable-Collision* means the location is occupied and invalid, and $g(x) <$

2^V. In encoding, *Unrecoverable-Collision* means the location is occupied, and false positives have already been detected at this location; if x is encoded, there will be another encoded false positive here. The location is already set invalid, therefore, there is no need to change the validation bit again. In querying, *Unrecoverable-Collision* means x is related to an encoded false positive and therefore we need to stop trying in the hash tables and begin to search in the overflow table.

```
Algorithm 2: getStatus($f(x), A_i, h$)
input : $f(x)$: the key's fingerprint;
        $A_i$: the hash table (including its $placement_i$ and $validation_i$);
        $h$ the hash location being tested;
output: Open , Match , Recoverable-Collision , or Unrecoverable-Collision
        (status of the hash location)
1 if $placement_i[h] = 0$ then
2     return Open
3 if $f(x) \otimes A_i[h] \leq V$ then
4     if $validation_i[h] = 1$ then
5         return Match
6     else
7         return Unrecoverable-Collision
8 else
9     return Recoverable-Collision
```

Algorithm 3 gives the steps for encoding data with a re-scan. This algorithm scans the dataset twice.

The first scan builds up the hash tables and their accompanying bitarrays. After the first time of scanning, most items are already successfully encoded into the hash tables, except a small portion of items. For an item $\langle x, v(x) \rangle$, there are three possible reasons for its encoding failure: 1) overflow, which means x did not find any available location in the N hash tables and needed to be stored in the overflow table; 2) x found an available location in the hash tables and the item $\langle x, v(x) \rangle$ was encoded into that location, however an encoded false positive was detected on this location when encoding another item later, therefore the location is set invalid; 3) an encoded false positive was detected when trying to encode x, therefore x could not be encoded into the hash tables and needed to be stored in the overflow table.

The second scan is actually a querying test (Algorithm 4) with an empty *overflow-table* as a parameter. In this case, only the items encoded in the hash tables can get correct return values; the items that fail to be encoded into the hash tables, including those in the overflow table and encoded false positives, will get a return value of zero, which is incorrect. For these items, we record them into the overflow table.

The querying procedure is described by Algorithm 4. Given a key x, this algorithm tries x on a sequence of hash locations. For every such location, the algorithm gets the status, and takes further actions based on the status. Possible actions include: 1) returning $g(x)$ as calculated on this location; 2) trying the next location; or 3) stopping the search and checking the overflow table.

The overflow table can be implemented as a sequential or sorted file, and the procedure $search$ *overflow-table* $for\ x$ can be either a linear search or a binary search. More efficient implementations of the overflow table (e.g. hashing) are also possible, but according to Section 3.4.2, the probability of accessing *overflow-table* is very small, and the size of *overflow-table* is also very small. Therefore, although linear search is slower than hash table lookup, the overall performance of the Geometric filter is not significantly affected.

In the next section, we introduce an improved version of Geometric filter that functions without the bitarrays. By manipulating the bitwise calculation applied to the hash location, the status information represented by the two bitarrays can be incorporated into the hash table. In this case, the two bitarrays are not necessary, and the space usage is also reduced.

3.2 Geometric filter without bitarrays

As we introduced in the previous section, in Geometric filter, every hash table is accompanied by two bitarrays, the placement bitarray and the validation bitarray. These bit vectors carry the information about whether a hash location is occupied and whether a hash location is valid. On the other hand, these two bitarrays also

Algorithm 3: EncodeRescan(S, $A_1, \dots A_N$)

```
input : $S$: encoding dataset;
        $A_i$, $i \in \{1, 2, \dots N\}$: the $N$ hash tables, each hash table includes its
        hash functions $h_{i,j}()$, $j \in \{1, 2, \dots k\}$, $placement_i$ and $validation_i$;
output: $A_i$, $i \in \{1, 2, \dots N\}$: hash tables with data filled in;
        overflow-table: the overflow table;

for $i \leftarrow 1$ to $N$ do
    $placement_i[] \leftarrow all\ 0s$
    $validation_i[] \leftarrow all\ 1s$
foreach item $\langle x, v(x) \rangle \in$ dataset $S$ do
    $stored \leftarrow false$
    $i \leftarrow 1$
    while $i \leq N$ AND $stored = false$ do
        $j \leftarrow 1$
        while $j \leq k$ AND $stored = false$ do
            $status \leftarrow getStatus(f(x), A_i, h_{i,j}(x))$
            switch $status$ do
                case Open
                    $A[h_{i,j}(x)] \leftarrow f(x) \otimes v(x)$
                    $placement_i[h_{i,j}(x)] \leftarrow 1$
                    $stored \leftarrow true$
                    break
                case Match
                case Unrecoverable-Collision
                    $validation_i[h_{i,j}(x)] \leftarrow 0$
                    $stored \leftarrow true$
                    break
                case Recoverable-Collision
                    break
foreach item $\langle x, v(x) \rangle \in$ dataset $S$ do
    $g(x) \leftarrow query(x, A_{1 \dots N}, \emptyset)$
    if $g(x) \neq v(x)$ then
        add $\langle x, v(x) \rangle$ into overflow-table
    else
        continue
```

Algorithm 4: query($x, A_1 \ldots A_N$, *overflow-table*)

```
input  : $x$: querying key;
         $A_i$, $i \in \{1, 2, \ldots N\}$: the $N$ hash tables, each hash table includes its
         hash functions $h_{i,j}$, $j \in \{1, 2, \ldots k\}$, $placement_i$ and $validation_i$;
output: $g(x)$: the key's value, 0 for unencoded key;

status ← NONE
for $i \leftarrow 1$ to $N$ do
    for $j \leftarrow 1$ to $k$ do
        $status \leftarrow getStatus(f(x), A_i, h_{i,j}(x), placement_i)$
        switch $status$ do
            case Open
                $g(x) \leftarrow 0$
                return $g(x)$
                break
            case Match
                $g(x) \leftarrow f(x) \otimes A_i[h_{i,j}(x)]$
                return $g(x)$
                break
            case Unrecoverable-Collision
                $g(x) \leftarrow$ search overflow-table for $x$
                return $g(x)$
                break
            case Recoverable-Collision
                break
if $status$ = Recoverable-Collision then
    $g(x) \leftarrow$ search overflow-table for $x$
    return $g(x)$
```

increase the Geometric filter's space usage, where each hash location actually take $c+2$ bits, including c bits in the hash table, and 2 bits in the placement bitarray and the validation bitarray.

In this section, we introduce a number of encoding rules to merge the two bitarrays into their corresponding hash table. With these encoding rules, the information represented by the two bitarrays can be incorporated into the hash table, hence resulting in smaller space usage and more elegant design of the Geometric filter.

However, we still need to keep the status of each hash location, and when testing a key on a hash location, we should be able to tell if the result is: 1) *Open* , 2) *Match* , 3) *Recoverable-Collision* , or 4) *Unrecoverable-Collision* . The status is retrieved by a different function called $getStatus()$; but most parts of the $EncodeRescan()$ procedure and the $query()$ procedure are the same as the previous version of Geometric filter.

3.2.1 Relational encoding: separating fingerprints and values

Our earlier version of the Geometric filter uses Eq. 3.1 to calculate the value of the hash location for encoding $\langle x, v(x)\rangle$. In that bitwise XOR calculation, the key's fingerprint $f(x)$ and the key's value $v(x)$, and the hash location all have c bits. This equation is introduced following the encoding function used by the Bloomier filter (Eq. 2.2). For convenience, the XOR calculation in Eq. 2.1, Eq. 2.2, Eq. 3.1 and Eq. 3.2 does a "uniform encoding", meaning that all the variables of the XOR calculations uniformly have c bits. Uniform encoding is especially useful in the Bloomier filter, because the value of an encoding is a combination of the values of several other locations. However, in Geometric filter, the value of an encoding hash location does not depend on any other location, and we can use a more effective method to encode the data.

In our new encoding, the hash location is divided into two parts: 1) the error bits of the location is called the fingerprint, denoted as $f(x)$; 2) the value bits of the location, still denoted as $v(x)$. At encoding, the binary relation $\langle f(x), v(x)\rangle$ is stored in the hash location. In this binary relation, $f(x)$ is the tuple identifier. We do not directly use the key x as the identifier, because in real applications, the key's length can easily exceed the capacity of a hash location.

The querying procedure remains similar to the querying procedure discussed before. When querying for a key x, we still need to test x on a sequence of hash locations until we find the location where x is encoded. However, the condition $g(x) < 2^V$ is not applicable any more.

Let $T()$ be the function that returns the number represented by the error bits of a hash location, and $G()$ be the function that returns the number represented by the value bits of a hash location.

Suppose we are testing x on a hash location $A[h(x)]$. First, we calculate x's fingerprint $f(x)$. Then, we compare $f(x)$ with $T(A[h(x)])$, if $f(x) = T(A[h(x)])$, then we return $G(A[h(x)])$ as x's $value$; if $f(x) \neq T(A[h(x)])$, then x does not match with this location, and we need to try x on the hash locations that follows up in the sequence.

In this encoding, for any key x, its fingerprint $f(x)$ consists of E error bits; if we assume the fingerprints are uniformly distributed in $[0, 2^E - 1]$, then the probability of two different keys having the same fingerprint is $\frac{1}{2^E}$, and false positive rate in each location $e_n = \frac{1}{2^E}$. Hence the false positive rate in each location remains the same as before.

3.2.2 Incorporating the placement bitarray

In Geometric filter, the placement bitarray carries the information of whether a hash location is occupied. In order to incorporate the placement bitarray into the hash table, we need to make an assumption on the fingerprint function.

Assumption 3.2.1 *For any key x, its fingerprint $f(x)$ is non-zero.*

Assumption 3.2.1 can be satisfied by changing the range of the fingerprint function such that it always returns non-zero values. However, if the fingerprint function is fixed, one can use a simple method where a particular bit (say the top-most bit) is set to one when the fingerprint is zero.

With this assumption, we can incorporate the placement bitarray into the hash table: Every occupied location must have at least one non-zero bit, because when storing the relation $\langle f(x), v(x)\rangle$, at least one bit of $f(x)$ is one. On the other hand, every unoccupied location must have all bits zero.

By incorporating the placement bitarray into the hash table, we can reduce the memory usage of Geometric filter. With this change, there are $2^E - 1$ possible values for $f(x)$, and the false positive rate in each location is $e_n = \frac{1}{2^E-1}$, instead of $\frac{1}{2^E}$. However, most applications require each hash location to have more than 12 error bits [31], and when $E \geq 12$, the difference in false positive rate is negligible.

3.2.3 Incorporating the validation bitarray

In order to incorporate the validation bitarray into the hash table, we need to make an assumption about the key's value $v(x)$.

Assumption 3.2.2 *For any key x that exists in encoding dataset S, its value $v(x)$ is non-zero.*

For simplicity, we assume the $value$ $v(x)$ is a positive integer, defined as:

$$v(x) = \begin{cases} integer\ in\ [1, 2^V - 1] & x\ exists\ in\ S \\ 0 & otherwise. \end{cases}$$

Assumption 3.2.2 can be satisfied by changing the range of the $value$s to be stored. In frequency-based querying applications, where $v(x)$ indicates the frequency of x, $v(x)$ for an existing key x should be always greater than zero, because $v(x) = 0$ means x does not exist in the dataset. If the value $v(x) \in [0, 2^V - 1]$, the state that $v(x) = 0$ is actually wasted because no existing key would have a frequency of zero.

When testing a key x on an occupied hash location, denoted as $A[h(x)]$, we use the following rules to retrieve the validation information: 1) if $G(A[h(x)]) = 0$, then the hash location $A[h(x)]$ is home to an encoded false positive and is invalid; 2) if $G(A[h(x)])) \neq 0$, then the location $A[h(x)]$ is not home to any encoded false positive and is valid.

In frequency-based querying applications, by incorporating the validation bitarray into the value part of each location, we can reduce the memory usage without introducing any additional error into the Geometric filter.

However, in case where $v(x)$ can take the value of zero, using the state $v(x) = 0$ to represent invalid status may affect the number of distinct values that can be represented, but this is negligible when the range of $v(x)$ is not small.

3.2.4 Revised Algorithms

With the two bitarrays incorporated into the hash table, the algorithms also need to be changed accordingly.

In $getStatus()$, instead of directly getting the occupancy and validation information from the two bitarrays, the new algorithm needs to retrieve the information from the fingerprint and the value part of the hash location. The new $getStatus()$ is described by Algorithm 5, in which functions $T()$ and $G()$ retrieve the error bits and the value bits of a hash location respectively.

```
Algorithm 5: getStatus($f(x), A_i, h$)
input  : $f(x)$: the key's fingerprint;
         $A_i$: the hash table;
         $h$ the hash location being tested;
output: Open , Match , Recoverable-Collision , or Unrecoverable-Collision :
         status of the hash location
1 if $A_i[h] = 0$ then
2     return Open
3 if $T(A_i[h]) = f(x)$ then
4     if $G(A_i[h]) \neq 0$ then
5         return Match
6     else
7         return Unrecoverable-Collision
8 else
9     return Recoverable-Collision
```

The procedures $EncodeRescan()$ and $query()$ remain almost the same with changes in a few lines as discussed next:

- According to the new encoding rules, the relation $\langle f(x), v(x)\rangle$ will be stored into the hash location. Therefore, in $EncodeRescan()$ (Algorithm 3), Line 13 needs to be changed from $A[h_{i,j}(x)] \leftarrow f(x) \otimes v(x))$ to $A[h_{i,j}(x)] \leftarrow$

$\langle f(x), v(x)\rangle$. Also note that the function $f(x)$ has E bits and $v(x)$ has V bits now; and $c = E + V$.

- When setting a hash location as invalid, we change its value bits to all zeros, therefore, in $EncodeRescan()$ (Algorithm 3), Line 19 needs to change from $validation_i[h_{i,j}(x)] \leftarrow 1$ to $A[h_{i,j}(x)] \leftarrow \langle f(x), 0\rangle$.
- The return value $g(x)$ can be directly retrieved from the hash location's value bits by function $G()$ now; therefore, in $query()$ (Algorithm 4), Line 11 needs to change from $g(x) \leftarrow f(x) \otimes A_i[h_{i,j}(x)]$ to $g(x) \leftarrow G(A_i[h_{i,j}(x)])$.

3.3 Quantitative basis of Hash tables

3.3.1 Analysis of uniformly distributed hashing

The analysis in this section is based on the assumption that all the hash functions have uniform distributions.

Definition Load factor of a hash table, denoted as p, is the fraction of locations in the hash table that are occupied.

Not every item in the dataset can be encoded into the hash table, some of them will be rejected because of hash collisions. According to Geometric filter's k-attempt encoding rule, a key would not be rejected unless all of its k hash locations are occupied. Assuming the hash functions to be uniformly distributed, and the load factor is p, the probability of a key being rejected is p^k. If the key is not rejected, then the load factor increases by $\frac{1}{M}$, where M is the hash table's size. The expectation of the load factor after trying the key is $p + (1 - p^k) \times \frac{1}{M}$. Let $p(i)$ be the expected load factor after encoding i items, where $i \geq 0$, then $p(i + 1) = p(i) + (1 - p^k(i)) \times \frac{1}{M}$.

If the distribution of the hash function is uniform and the hash tables are very large, given the same k, and the same storage ratio, hash tables should all have the same load factor. Specifically, for hash table A_1 and A_2, if the hash table size M_1 is d times M_2 and the trying set size $|S_1|$ is also d times $|S_2|$, then the load factor p_1 is the same as p_2 after the k-attempt encoding.

We prove this statement in two steps. In Lemma 3.3.1, we prove that the statement is true when d is an integer greater than one. In the following Theorem 3.3.2, we generalize the statement to the case where d is any positive real number, which means the correctness of the statement does not depend on d.

Lemma 3.3.1 *By the k-attempt encoding rule of Geometric filter, data items are tried on hash tables A_2 and A_1, where $M_2 \times d$ = M_1 (integer $d > 1$). Let M_1 and M_2 be sufficiently large, then*

$$p_2(i) = p_1(i \times d).$$

Proof Throughout this proof, we hold the following point as self-evident: when M_1 is sufficiently large, we consider M_1 as infinity, and $\frac{1}{M_1}$, $\frac{1}{M_1^2}$ and $o(\frac{1}{M_1^2})$ as infinitesimals on increasing orders; and the same to M_2.

Because $M_2 \times d = M_1$:

$$\begin{aligned} p_2\left(i+1\right) &= p_2\left(i\right) + \left(1 - p_2^k\left(i\right)\right) \times \frac{1}{M_2} \\ &= p_2\left(i\right) + \left(1 - p_2^k\left(i\right)\right) \times \frac{d}{d \times M_2} \\ &= p_2\left(i\right) + d \times \left(1 - p_2^k\left(i\right)\right) \times \frac{1}{M_1} \end{aligned} \tag{3.3}$$

And, for any i, $p_1(i+1) = p_1(i) + (1 - p_1^k(i)) \times \frac{1}{M_1}$, therefore (for simplicity, we denote $p_1(i \times d)$ as p):

$$
\begin{aligned}
&p_1\left(i\times d+1\right)\\
&=p_1\left(i\times d\right)+\left(1-p_1^k\left(i\times d\right)\right)\times\frac{1}{M_1}\\
&=p+\left(1-p^k\right)\frac{1}{M_1}
\end{aligned}
$$

$$
\begin{aligned}
&p_1\left(i\times d+2\right)\\
&=p_1\left(i\times d+1\right)+\left(1-p_1^k\left(i\times d+1\right)\right)\frac{1}{M_1}\\
&=\left[p+\left(1-p^k\right)\frac{1}{M_1}\right]+\left\{1-\left[p+\left(1-p^k\right)\frac{1}{M_1}\right]^k\right\}\frac{1}{M_1}\\
&=\left[p+\left(1-p^k\right)\frac{1}{M_1}\right]\\
&+\left\{1-\left[p^k+\binom{k}{1}p^{k-1}\left(1-p^k\right)\frac{1}{M_1}+\binom{k}{2}p^{k-2}\left(1-p^k\right)^2\frac{1}{M_1^2}+\dots\right]\right\}\frac{1}{M_1}\\
&=\left[p+\left(1-p^k\right)\frac{1}{M_1}\right]+\left\{1-p^k-\frac{kp^{k-1}\left(1-p^k\right)}{M_1}+o\left(\frac{1}{M_1}\right)\right\}\frac{1}{M_1}\\
&=p+\left(1-p^k\right)\frac{1}{M_1}+\left(1-p^k\right)\frac{1}{M_1}-\frac{kp^{k-1}\left(1-p^k\right)}{M_1^2}+o\left(\frac{1}{M_1^2}\right)\\
&=p+2\left(1-p^k\right)\frac{1}{M_1}-\frac{kp^{k-1}\left(1-p^k\right)}{M_1^2}+o\left(\frac{1}{M_1^2}\right)
\end{aligned}
$$

$$
\begin{aligned}
&p_1\left(i\times d+3\right)\\
&=p_1\left(i\times d+2\right)+\left(1-p_1^k\left(i\times d+2\right)\right)\frac{1}{M_1}\\
&=\left[p+2\left(1-p^k\right)\frac{1}{M_1}-\frac{kp^{k-1}\left(1-p^k\right)}{M_1^2}\right]\\
&+\left\{1-\left[p+2\left(1-p^k\right)\frac{1}{M_1}-\frac{kp^{k-1}\left(1-p^k\right)}{M_1^2}\right]^k\right\}\frac{1}{M_1}\\
&=\left[p+2\left(1-p^k\right)\frac{1}{M_1}-\frac{kp^{k-1}\left(1-p^k\right)}{M_1^2}\right]\\
&+\left\{1-p^k-\frac{2kp^{k-1}\left(1-p^k\right)}{M_1}+o\left(\frac{1}{M_1}\right)\right\}\frac{1}{M_1}\\
&=p+3\left(1-p^k\right)\frac{1}{M_1}-\left(1+2\right)\frac{kp^{k-1}\left(1-p^k\right)}{M_1^2}+o\left(\frac{1}{M_1^2}\right)
\end{aligned}
$$

$\ldots$

$$\begin{aligned}
&p_1\left((i+1)\times d\right) = p_1\left(i\times d+d\right)\\
&= p_1\left(i\times d+(d-1)\right) + \left(1-p_1^k\left(i\times d+(d-1)\right)\right)\frac{1}{M_1}\\
&= p + d\left(1-p^k\right)\frac{1}{M_1} - \left(1+2+\ldots+(d-1)\right)\times\frac{kp^{k-1}\left(1-p^k\right)}{M_1^2} + o\left(\frac{1}{M_1^2}\right)\\
&= p + d\left(1-p^k\right)\frac{1}{M_1} - \frac{d\left(d-1\right)}{2}\times\frac{kp^{k-1}\left(1-p^k\right)}{M_1^2} + o\left(\frac{1}{M_1^2}\right)
\end{aligned} \tag{3.4}$$

Let $b=\frac{d^2k}{2}$. Because $0\le p\le 1$ and $1-p^k\le 1$, $\frac{d(d-1)}{2}\times\frac{kp^{k-1}(1-p^k)}{M_1^2}\le\frac{b}{M_1^2}$. And because $0<\frac{d(d-1)}{2}\times\frac{kp^{k-1}\left(1-p^k\right)}{M_1^2}<\frac{b}{M_1^2}$, then

$$p+d\left(1-p^k\right)\frac{1}{M_1}-\frac{b}{M_1^2}\le p_1\left((i+1)\times d\right)\le p+d\left(1-p^k\right)\frac{1}{M_1} \tag{3.5}$$

We prove the following statement by induction on i, the number of items encoded:

$$0\le p_2(i)-p_1(i\times d)\le i\frac{b}{M_1^2} \tag{3.6}$$

1) When $i=0$, $p_2(0)=p_1(0)=0$, therefore $0\le p_2(i)-p_1(i\times d)\le i\frac{b}{M_1^2}$. Hence Eq. 3.6 is true when $i=0$.

2) Assuming $0\le p_2(i)-p_1(i\times d)\le i\frac{b}{M_1^2}$, we will prove $0\le p_2(i+1)-p_1((i+1)\times d)\le(i+1)\frac{b}{M_1^2}$ as follows:

We denote $p_1(i\times d)=p$, then

$$p\le p_2(i)\le p+i\frac{b}{M_1^2} \tag{3.7}$$

According, to Eq. 3.3, $p_2(i+1)=p_2(i)+d(1-p_2^k(i))\frac{1}{M_1}$, therefore the derivative of $p_2(i+1)$ with regard to $p_2(i)$ is

$$\frac{\delta p_2(i+1)}{\delta p_2(i)}=1-p_2^{k-1}(i)\times\frac{dk}{M_1}. \tag{3.8}$$

Because M_1 is sufficiently large, the value of $\frac{dk}{M_1}$ is close to zero, then

$$\frac{\delta p_2(i+1)}{\delta p_2(i)} > 0. \tag{3.9}$$

As Eq. 3.9 shows, the value of $p_2(i+1)$ increases with the value of $p_2(i)$. Because of Eq. 3.7 and the fact that $p_2(i+1)$ is monotonically non-decreasing with regard to $p_2(i)$, given a fixed p, the value of $p_2(i+1) - p_1((i+1)\times d))$ reaches its minimum when $p_2(i) = p$. Because of Eq. 3.3, $p_2(i+1) \geq p + d(1-p^k)\frac{1}{M_1}$, and using Eq. 3.4 and Eq. 3.5,

$$\begin{aligned}
&p_2(i+1) - p_1((i+1)d) \\
&\geq \left[p + d\left(1-p^k\right)\frac{1}{M_1}\right] - \left[p + d\left(1-p^k\right)\frac{1}{M_1}\right] \\
&= 0
\end{aligned}$$

Also because of Eq. 3.9, with a fixed p, the value of $p_2(i+1) - p_1((i+1)\times d))$ reaches its maximum when $p_2(i) = p + i\frac{b}{M_1^2}$. Then,

$$\begin{aligned}
&p_2(i+1) - p_1((i+1)d) \\
&\leq \left(p + i\frac{b}{M_1^2}\right) + \left[1 - \left(p + i\frac{b}{M_1^2}\right)^k\right]\frac{d}{M_1} \\
&- \left[p + d\left(1-p^k\right)\frac{1}{M_1} - \frac{b}{M_1^2}\right] \\
&= p + i\frac{b}{M_1^2} + \left[1 - p^k - kp^{k-1}\times i\frac{b}{M_1^2} - o\left(\frac{1}{M_1^2}\right)\right]\frac{d}{M_1} \\
&- \left[p + \left(1-p^k\right)\frac{d}{M_1} - \frac{b}{M_1^2}\right] \\
&= (i+1)\times\frac{b}{M_1^2} + o\left(\frac{1}{M_1^2}\right)
\end{aligned}$$

Hence, $0 \leq p_2(i+1) - p_1((i+1)d) \leq (i+1)\frac{b}{M_1^2}$.

3) Because of 1) and 2), Eq. 3.6 is proved: for any integer i, $0 \leq p_2(i) - p_1(i\times d) \leq i\frac{b}{M_1^2}$. Because the value of i is not infinity, then $i\frac{b}{M_1^2} \leq \frac{ib}{dM_1} = o(1)$ is negligible. Hence $p_2(i) = p_1(i\times d)$. ∎

Definition Storage ratio, denoted as r, is the ratio of the hash table size M to the encoding dataset size $|S|$, $r = \frac{M}{|S|}$.

Theorem 3.3.2 *By the k-attempt encoding rule, hash table A_1 has dataset S_1 with $r_1 = \frac{M_1}{|S_1|}$; hash table A_2 has dataset S_2 with $r_2 = \frac{M_2}{|S_2|}$. Provided that M_1 and M_2 are sufficiently large, if $r_1 = r_2$, then after finishing the encoding of S_1 and S_2, load factors*

$$p_1 = p_2.$$

Proof Let A be an additional hash table, which has dataset S with $r = \frac{M}{|S|}$. Assume that $r = r_1 = r_2$, and the size of A is $M = M_1 \times t_1 = M_2 \times t_2$, where t_1, t_2 are positive integers.

Let p denote the load factor of A after encoding S. Let $p(i)$, $p_1(i)$, and $p_2(i)$ denote the load factor of A, A_1, and A_2 respectively, after trying the ith item in the dataset.

Because M_1 and M_2 are sufficiently large, and $M = M_1 \times t_1 = M_2 \times t_2$, therefore M is also sufficiently large. Because $r = r_1 = r_2$, therefore,

$$|S| = \frac{M}{r} = \frac{M_1 \times t_1}{r} = |S_1| \times t_1. \tag{3.10}$$

When the trials are finished, every item in the datasets has been tried on the hash table, therefore $p_1 = p_1(|S_1|) = p_1(M_1 \times \frac{1}{r_1}) = p_1(M_1 \times \frac{1}{r})$, and $p = p(|S|) = p(M \times \frac{1}{r}) = p(t_1 M_1 \times \frac{1}{r})$. Because $M = t_1 M_1$, according to Lemma 3.3.1, $p_1 = p$.

Similarly, $p_2 = p$. Hence $p_1 = p_2$. ∎

Theorem 3.3.2 actually generalizes the Lemma 3.3.1 to the case that d can be any positive number, and the load factor p depends only on storage ratio r. In the following part, we analyze the functional relationship between p and r.

Theorem 3.3.3 *By the k-attempt encoding rule, dataset S is encoded into hash table A with $r = \frac{M}{|S|}$. Given k, and provided that M is sufficiently large, then*

1) load factor p is a function of r (which can be written as $p = p_k(r)$), 2) $p_k(r)$ is continuous, monotonically decreasing, and 3) its inverse function $r = p_k^{-1}(p)$ exists and $p_k^{-1}(p)$ is also continuous and monotonically decreasing.

Proof 1) When M is sufficiently large, we consider it as infinity. Given k, according to Theorem 3.3.2, p is determined by r, hence p is a function of r, denoted as $p = p_k(r)$, and because $r = \frac{M}{|S|}$, $M \neq 0$ and $|S| \neq 0$, then $r \in (0, +\infty)$.

2) For the convenience of calculus analysis, we use Δr to denote a small change in the value of r, and $\Delta|S|$ to denote a small change in the value of $|S|$. For any $r_0 \in (0, +\infty)$, let $\Delta r = -\frac{r^2}{M}$, then $\lim_{M\to\infty} \Delta r = 0$, and $r = \frac{M}{|S|}$, then

$$
\begin{aligned}
&|S| = \frac{M}{r} \\
&\Rightarrow \Delta|S| = M \times \left(\frac{1}{r} - \frac{1}{r - \frac{r^2}{M}}\right) = M \times \frac{\frac{r^2}{M}}{r\left(r - \frac{r^2}{M}\right)} \\
&\Rightarrow \lim_{M\to\infty} \Delta|S| = M \times \frac{\frac{r^2}{M}}{r\left(r - \frac{r^2}{M}\right)} = 1
\end{aligned}
$$

Provided the load factor $p_k(r)$, when $\Delta|S| = 1$, then $\Delta p = p_k(r + \Delta r) = p_k(r) + (1 - p_k^k(r))\frac{1}{M}$, therefore

$$
\begin{aligned}
&\lim_{\Delta r\to 0} \frac{p_k(r + \Delta r) - p_k(r)}{\Delta r} \\
&= \lim_{M\to\infty} \frac{p_k(r) + (1 - p_k^k(r))\frac{1}{M} - p_k(r)}{-\frac{r^2}{M}} \\
&= -\frac{1 - p_k^k(r)}{r^2} < 0
\end{aligned}
$$

Therefore, according to the definition of *function derivative*, the derivative of $p_k(r)$ exists for $r \in (0, +\infty)$ and $p_k'(r) < 0$. Hence, $p_k(r)$ is continuous and monotonically decreasing.

3) Because $p_k(r)$ is continuous and monotonically decreasing, its inverse function $p_k^{-1}(p)$ exists, and the inverse function is also continuous and monotonically decreasing. ∎

Theorem 3.3.3 specifies the functional relationship between p and r, which means when the hash table is large enough, given k, p is determined by r, and vice versa.

3.3.2 Load factor calculations

Let S be a dataset that is encoded into hash table A with a very large size M (treated as infinity). As Theorem 3.3.2 shows, given k, the load factor p after the k-attempt encoding is a function of storage ratio r, and here we denote this function as $p_k(r)$. When $k = 1$, $p_1(r)$ can be calculated by a mathematical formula; when $k \neq 1$, $p_k(r)$ can be determined by a Monte Carlo experiment.

When $k = 1$, each key is tried once; therefore the number of trials is $|S|$. After trying the dataset, a location being empty means there was no key hashed to this location, and its probability $P(empty) = (1 - \frac{1}{M})^{|S|} = (1 - \frac{1}{M})^{-M \times \frac{|S|}{-M}}$. When $M \to \infty$, $\lim_{M\to\infty} P(empty) = e^{-\frac{|S|}{M}} = e^{-\frac{1}{r}}$. Then, the probability of a location being occupied $P(occupy) = 1 - e^{-\frac{1}{r}}$. Hence:

$$p_1(r) = 1 - e^{-\frac{1}{r}} \tag{3.11}$$

When $k > 1$, every item have k trials before being rejected. For each item, the number of trials may vary from 1 to k, therefore it is difficult to provide a formula of $p_k(r)$. In this case, the value of $p_k(r)$ can be determined by a Monte Carlo experiment, as shown in Algorithm 6.

Because the hash functions are assumed to be uniformly random, we use a random function instead of real hash functions. We run $|S|$ iterations simulating encoding $|S|$ items in the dataset. In each iteration, we use the random function to select k bits in the bitarray of size $r|S|$, and set the first zero bit to one. Finally, by counting up the bits set to one, we can find a very close estimation of $p_k(r)$. And, because the time of generating a random number is much less than using real hash function to process a key, this Monte Carlo experiment finishes in a very short time.

According to Theorem 3.3.2, $p_k(r)$ does not depend on the dataset's size $|S|$; however, this conclusion is based on the assumption that the hash table size $M \to \infty$. In reality, M cannot be infinite, therefore the results closely approximates Theorem 3.3.2, but with some difference. $|S|$ needs to be provided in Algorithm 6 to prevent the difference getting too large for real applications, and the reason will be discussed in Section 3.3.3.

Algorithm 6: loadFactor($r, k, |S|$)

```
input  : $r$: storage ratio;
         $k$: number of hash functions
         $|S|$: size of dataset
output: $p$: hash table's expected load factor after encoding
1  $M \leftarrow |S| \times r$
2  allocate bitarray $bitarray[M]$
3  $bitarray[M] \leftarrow all\ 0s$
4  $c \leftarrow 0$
5  for $i \leftarrow 1\ to\ |S|$ do
6      for $j \leftarrow 1\ to\ k$ do
7          $h \leftarrow random() \mod M$
8          if $bitarray[h] = 0$ then
9              $bitarray[h] \leftarrow 1$
10             $c \leftarrow c + 1$
11             break
12 $p \leftarrow c \ / \ M$
13 return $p$
```

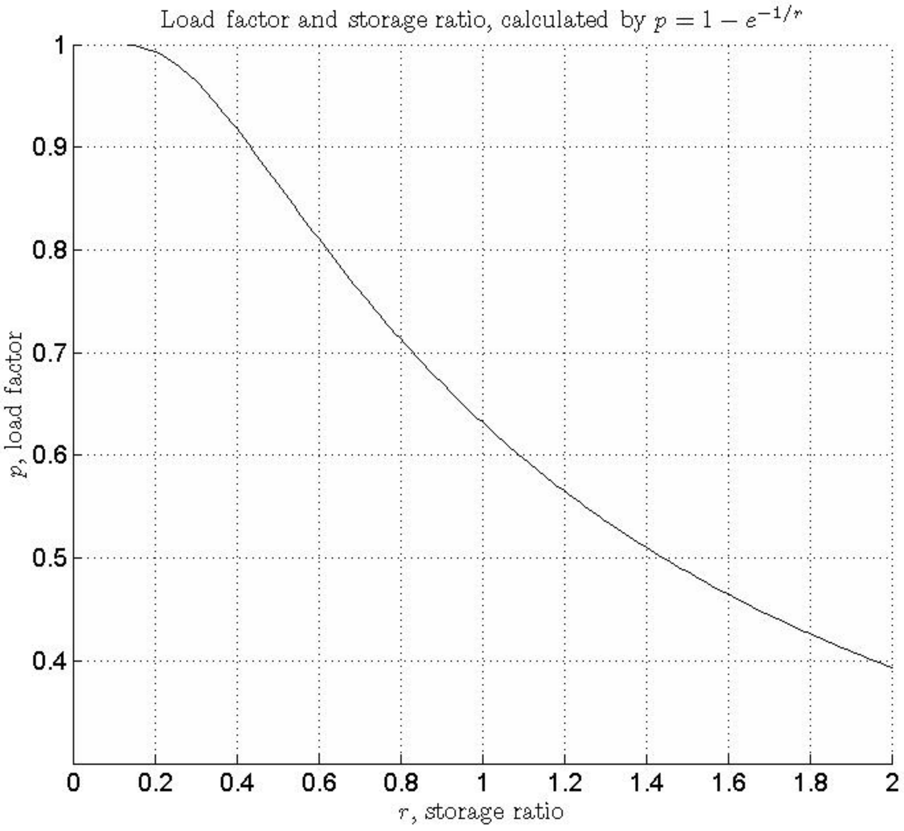
When $k = 1$, we plot the curves of $p_k(r)$ by both Eq. 3.11 and Algorithm 6. As Figure 3.1 shows, the two figures are almost identical.

3.3.3 Finite sizes and non-uniform distributions

Discussions about load factor function $p_k(r)$ are based on two assumptions: all the hash functions are uniformly random, and hash table size $M \to \infty$. However, these two assumptions cannot be satisfied in real applications. The hash table size is not infinite, and the hash functions' distribution is not uniform.

If M is not infinite, then when $k = 1$, $p_1(r)$ is not equal to $1 - e^{-\frac{1}{r}}$ (Eq. 3.11), although the values are close to each other.

The random function in Monte Carlo experiment and the hash functions actually do not have a uniform distribution. Hash functions with non-uniform distributions increase the number of hash collisions and reduce the hash table's load factor. Non-uniform distributions have different effects on hash tables with different sizes. The distribution on a small hash table is usually more uniform than the distribution on a large hash table, and the reason will be discussed later in this section. In this case,



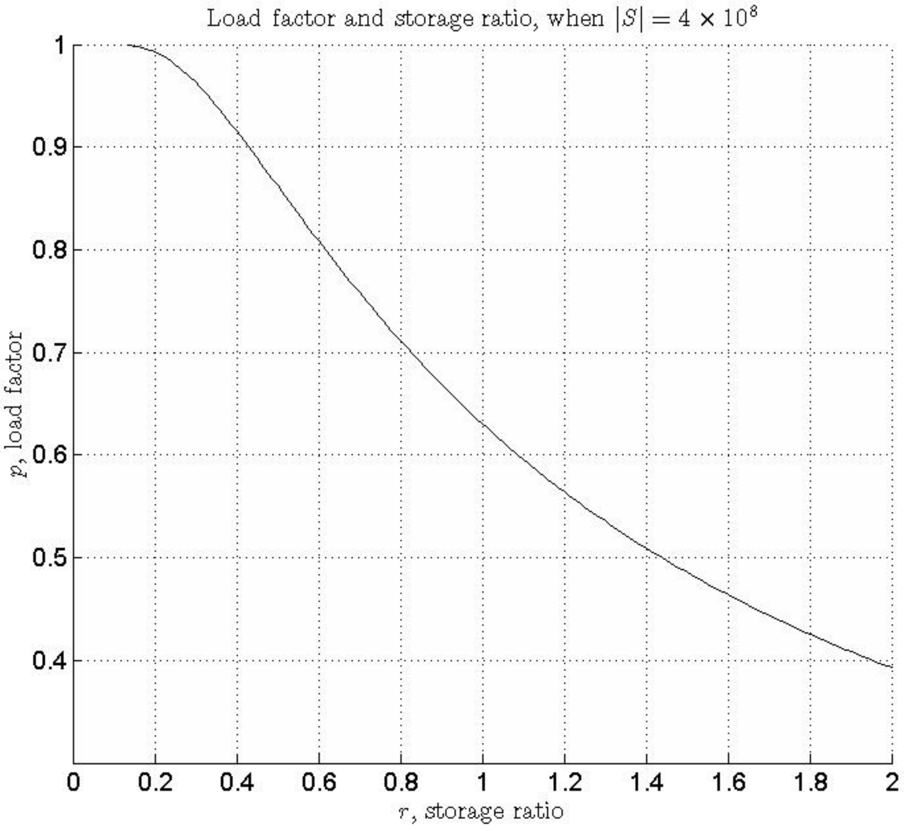


Figure 3.1: Load factor calculated by Eq. 3.11 and Algorithm 6 when $|S| = 4 \times 10^8$ and $k = 1$

with the same storage ratio r the load factor p is higher in a small hash table than in a large hash table.

Hash functions are derived from a basis function $H(x)$ with range $[0, M_H - 1]$, and M_H is larger than the hash table's size M. For a hash table of size M, the hash function $h(x) = H(x) \bmod M$, and $h(x)$'s range is $[0, M - 1]$.

Here, we denote $P_H(v)$ as the probability that $H(x) = v$, and $P_h(v)$ as the probability that $h(x) = v$. Because $H(x)$ does not have a uniform distribution, $P_H(v)$ is not a constant when $v \in [0, M_H - 1]$. For simplicity, we suppose $M_H = M \times a$ where a is an integer. Then, for the hash function $h(x)$, $P_h(v) = P_H(v) + P_H(v + M) + \ldots + P_H(v + M \times (a - 1))$. By summing up a items from $P_H(v)$ to $P_H(v + (a - 1)M)$, $P_h(v)$ is more even than $P_H(v)$. Similarly, when derived from the same basis function, hash functions with a small range have more uniform distributions than hash tables with a large range.

Non-uniform hash functions will cause additional collisions in the hash table, and reduce the hash table's load factor. In most cases, the impact is unnoticeable [35] [26]. However, the sizes of Geometric filter's hash tables form a descending geometric series. If a large hash table does not achieve the designed load factor, and does not absorb the designed amount of items, then there will be additional items going to the consequent hash tables. For the largest hash table in the front, the amount of the additional items is insignificant; but for the smaller hash tables following, these extra items will be a heavy burden. The smaller hash tables won't be able to absorb the extra items either, then the extra amount will be passed on to the next one, and so forth.

As a result, the hash tables at the end of the sequence have a higher load factor than expected, and some of them can be filled up to 100%. And, the number of items going to the overflow table would also exceed the designed amount, which means the size of the overflow table is not negligible. In this case, the Geometric filter's performance is harmed in three aspects: 1) the false positive rate e will be higher than designed; and 2) the querying speed will be slower than expected, and 3) the storage space will be increased.

Being aware of the difference in hash table sizes, and non-uniform hash functions, the following principles need to be applied:

- if the load factor is calculated by $p(r) = 1 - e^{-\frac{1}{r}}$ (Eq. 3.11), then the value needs to be reduced by a small amount (the change is usually made on the 3rd or the 4th digit after the decimal), because actually achieved load factor is lower than theoretical calculation;
- if the load factor is calculated by Monte Carlo experiment (Algorithm 6), then the size of the bitarray in the experiment needs to be greater or equal to the size of the hash tables, or the calculated load factor needs to be deducted by a small amount.

3.4 Quantitative basis of Geometric filter

In this section, we provide some mathematical formulas of Geometric filter, which will be used by the parameter setting and the theoretical analysis in later sections.

3.4.1 Hash tables in the sequence

Geometric filter consists of a sequence of hash tables, the sizes of which form a descending geometric sequence. And, the numbers of items tried on these hash tables also form a descending geometric sequence.

Definition Trying set is denoted as S_i, where $i = 1, 2, \ldots N, N+1$. When $1 \leq i \leq N$, S_i means the set of data items tried on the hash table A_i. S_{N+1} means the set of items rejected by the last hash table (A_N) and stored in the overflow table.

Obviously, $S_1 = S$, because hash table A_1 is the first hash table in the sequence, and every item in dataset S will be tried on A_1.

Definition Reject rate, denoted as d, is the proportion of items that are tried on a hash table, and rejected because of hash collisions.

For hash table A_i, where $1 \leq i \leq N$, the trying set is S_i, and the items rejected by A_i form the next trying set S_{i+1}. In this case, the reject rate:

$$d_i = \frac{|S_{i+1}|}{|S_i|} \tag{3.12}$$

In hash table A_i, we denote the load factor as p and the storage ratio as r. The number of occupied locations is $|S_i - S_{i+1}|$. Since, $S_{i+1} \subseteq S_i$, $|S_i - S_{i+1}| = |S_i| - |S_{i+1}|$, hence the load factor of this hash table $p = \frac{|S_i|-|S_{i+1}|}{M_i}$. Since $d_i = \frac{|S_{i+1}|}{|S_i|}$, then $|S_i| - |S_{i+1}| = (1 - d_i) \times |S_i|$. And, $\frac{1}{r} = \frac{|S_i|}{M_i}$, therefore $p = \frac{(1-d_i)\times|S_i|}{M_i} = (1 - d_i) \times \frac{|S_i|}{M_i} = (1 - d_i) \times \frac{1}{r}$. Then, $p \times r = 1 - d_i$, hence:

$$d_i = 1 - p \times r \tag{3.13}$$

Assumption 3.4.1 *In Geometric filter, all the hash tables, from A_1 to A_N, have the same storage ratio r.*

Assumption 3.4.1 is satisfied by adjusting the sizes of hash tables in the sequence as described following.

Given dataset S and storage ratio r, because $S_1 = S$, then $M_1 = r|S_1| = r|S|$. Given k, load factor p is a function of r, denoted as $p_k(r)$, then according to Eq. 3.13, $d_i = 1 - r \times p_k(r)$. Provided that r is a constant in the hash tables, then the value of d_i are also a constant, which we denote as d. $|S_2| = |S_1| \times d = |S| \times d$, then $M_2 = r|S_2| = r|S| \times d = M_1 \times d, \ldots$, hence the sequence of the sizes of trying sets is:

$$|S_i| = |S| \times d^{i-1} \tag{3.14}$$

And the sequence of the sizes of the hash tables is:

$$M_i = r|S| \times d^{i-1} = M_1 \times d^{i-1} \tag{3.15}$$

Because $d = 1 - p \times r < 1$, the two sequences above are descending geometric series.

3.4.2 Length of the hash sequence

In Geometric filter, we assume that the overflow table's size is negligible, and that most items in the dataset are absorbed by the hash tables. Therefore, the hash table sequence needs to be long enough such that the overflow table size gets to the smallest size $|S_{N+1}| = 1$. According to Eq. 3.14, $|S_{N+1}| = |S| \times d^{(N+1)-1}$, hence

$$\begin{aligned} N &= \left\lceil \log_d \left(\frac{1}{|S|} \right) \right\rceil \\ &= \left\lceil \log_{1-r\times p} \left(\frac{1}{|S|} \right) \right\rceil = \left\lceil \log_{1-r\times p_k(r)} \left(\frac{1}{|S|} \right) \right\rceil \end{aligned} \tag{3.16}$$

Because of the non-uniformity of hash functions, there are more hash collisions than theoretically expected in the hash tables, therefore $|S_{N+1}|$ can be larger than one. However, compared with the large size of the whole dataset, the overflow table is still negligible.

3.4.3 Overall space usage

Ignoring the overflow table, we only consider the memory space used by the hash tables. For convenience, we use bit as the smallest unit for space measurement.

The space of the Geometric filter $space_G$ is the sum of the spaces of all the hash tables; using Eq. 3.15, the space can be calculated via Eq. 3.17, where c is the number of bits per location.

$$\begin{aligned} space_G &= \sum_{i=1}^{N} M_i \times c \\ &= \sum_{i=1}^{N} r|S|d^{i-1} \times c \\ &\leq \lim_{N\to\infty} \sum_{i=1}^{N} r|S|d^{i-1} \times c \\ &= r|S|\frac{1}{1-d} \times c \\ &= |S|\frac{r}{1-(1-rp)} \times c \\ &= \frac{|S|}{p} \times c = \frac{|S|}{p_k(r)} \times c. \end{aligned} \tag{3.17}$$

3.4.4 False positive rates

On each hash location, the false positive rate is determined by the number of error bits in the location. Given E error bits, the false positive rate on each hash location (also called "natural false positive rate") is $e_n = \frac{1}{2^E}$.

During querying, a key will be tried on a sequence of locations, and false positive could happen on any one of them, hence the Geometric filter's overall false positive rate $e_o \geq e_n$.

Lemma 3.4.2 *If all the hash tables have the load factor p, and the false positive rate on each location is e_n, then the Geometric filter's overall false positive rate*

$$e_o \leq \frac{p}{1-p} e_n$$

Proof On each hash table, a key is tried in at most k locations, and false positive could happen in any one of these locations. According to Algorithm 4, false positive happens when the following two conditions are satisfied: 1) x is mapped to a occupied location, and 2) a false positive happens for x on this location. False positive rate on a hash table, denoted as e_h, can be calculated by Eq. 3.18. For convenience, we use the following notations:

"$FP.j$" means false positive happens on the jth hash location, where $1 \leq j \leq k$; and "$\neg FP.\ 1 \ldots j$" means false positive does not happen on the first j hash locations. The false positive rate of the hash table equals to the sum of the probabilities of false positives of all the hash locations. For each hash location, the probability of false positive is the probability that one false positive happens on the location and no false positive happens on the locations before it.

$$\begin{aligned} e_h &= P(FP.\ 1) + P(FP.\ 2|\neg FP.\ 1) + \ldots \\ &\quad + P(FP.\ k|\neg FP.\ 1 \ldots k-1) \\ &= p \times e_n + p^2(1-e_n) \times e_n + \ldots + p^k(1-e_n)^{k-1} \times e_n \\ &\leq e_n \times (p + p^2 + \ldots + p^k) \\ &= e_n \times \sum_{i=1}^{k} p^i = \frac{p(1-p^k)}{1-p} e_n \end{aligned} \tag{3.18}$$

In the Geometric filter, a key also needs to be tried on a sequence of hash tables, and if false positive happen on any one of these hash tables, the returned value will be erroneous, therefore the overall false positive rate $e_o \geq e_h \geq e_n$, and its value is calculated by Eq. 3.19. For convenience, in the following probability functions, notation "A_i" means the key is tried on hash table A_i.

$$
\begin{aligned}
e_o &= \sum_{i=1}^{N} P(A_i) \times P(FP.|A_i) \\
&= \sum_{i=1}^{N} P(A_i) \times e_h \\
&\leq \lim_{N\to\infty} \sum_{i=1}^{N} p^{k\times(i-1)} \times e_h \\
&= e_h \times \frac{1}{1-p^k} = e_n \times \frac{\frac{p(1-p^k)}{1-p}}{1-p^k} \\
&= e_n \times \frac{p}{1-p}
\end{aligned}
\tag{3.19}
$$

∎

For a key not encoded, the algorithm will search the overflow table for the key if false positive did not happen in the hash tables; however, we do not consider false positives in the overflow table, because it is implemented as a sequential file with key-$value$ pairs that does not have any false positive.

3.5 Parameter Settings

In order to configure the Geometric filter, we need to know the values of following variables: 1) $|S|$, the size of the encoding set; 2) e_o, the allowed false positive rate; 3) V, the number of value bits; and 4) k, the number of hash functions.

We use an example to illustrate the parameter setting, for which $|S| = 4 \times 10^8$, $e_o = 1/2^{16}$, $V = 16$, and $k = 1$.

The obtained parameters will minimize the memory space usage for encoding the dataset, and the parameters are: 1) N, the length of hash table sequence; 2) r, the storage ratio in each hash table; 3) E, the number of error bits in each location.

3.5.1 Minimizing the memory usage

In order to save space, we let the actual overall false positive rate reach its maximum allowed value e_o. According to Lemma 3.4.2, $\frac{p}{1-p}e_n = \frac{p}{1-p} \times \frac{1}{2^E} = e_o$, therefore the number of error bits $E = \lceil \log_2 \frac{p}{(1-p)e_o} \rceil$, then the number of bits per location

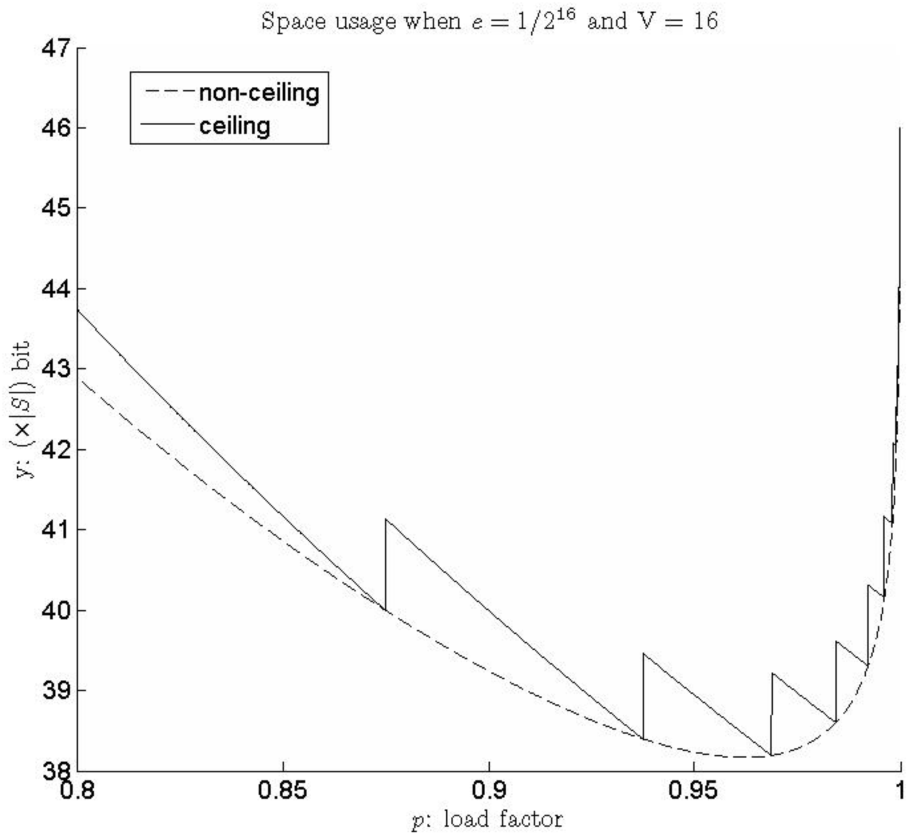


Figure 3.2: Geometric filter's space usage (with or without ceiling), when $e_o = 1/2^{16}$ and $V = 16$

$c = V + E = V + \lceil \log_2 \frac{p}{(1-p)e_o} \rceil$. According to Eq. 3.17, space usage of the Geometric filter (measured in bits) is:

$$space_G = \frac{|S|}{p} \times \left(V + \left\lceil \log_2 \frac{p}{(1-p)e_o} \right\rceil \right) \tag{3.20}$$

Provided that $|S|$ is a given constant, the minimization of $space_G$ is equivalent to the following minimization:

$$\arg \min_{p \in (0,1)} \frac{1}{p} \times \left(V + \left\lceil \log_2 \frac{p}{(1-p)e_o} \right\rceil \right) \tag{3.21}$$

Given $e_o = \frac{1}{2^{16}}$ and $V = 16$, we plot the curves of $y = \frac{1}{p} \times (V + \lceil \log_2 \frac{p}{(1-p)e_o} \rceil)$ and $y = \frac{1}{p} \times (V + \log_2 \frac{p}{(1-p)e_o})$. As Figure 3.2 shows, the minimum y values of the ceiling and non-ceiling curves are close to each other. This is because the non-ceiling curve's derivative is close to zero near its minimum point, therefore a small shift in p would not change the value of y dramatically.

For the ease of our analysis, we simplify the minimization of the ceiling expression to minimization of the following non-ceiling expression:

$$\arg\min_{p\in(0,1)} \frac{1}{p} \times \left(V + \log_2 \frac{p}{(1-p)e_o}\right) \tag{3.22}$$

Lemma 3.5.1 *Given the number of value bits V, allowed false positive rate e_o, and provided that $V + \log_2 \frac{1}{e_o} \geq 3$ then the minimum value of*

$$f(p) = \frac{|S|}{p} \times \left(V + \log_2 \frac{p}{(1-p)e_o}\right)$$

is reached when the load factor is [1]*:*

$$p = 1 - \frac{1}{1 - W_{-1}\left(-e^{1-\ln 2\times(V+\log_2 \frac{1}{e_o})}\right)}$$

Proof If $f(p)$ reaches the minimum value, then its derivative $f'(p) = 0$, therefore:

$$f'(p) = \frac{-1}{p^2}\left[V + \log_2 \frac{1}{e_o} + \log_2 \frac{p}{1-p}\right] + \frac{1}{p^2} \times \frac{1}{\ln 2 \times (1-p)} = 0$$

Therefore,

$$\begin{aligned}
&\frac{1}{p^2}\left[V + \log_2 \frac{1}{e_o} + \log_2 \frac{p}{1-p}\right] = \frac{1}{p^2} \times \frac{1}{\ln 2 \times (1-p)}\\
&\Rightarrow \left(V + \log_2 \frac{1}{e_o}\right) + \log_2 \frac{p}{1-p} = \frac{1}{\ln 2 \times (1-p)}\\
&\Rightarrow \ln 2 \times \left(V + \log_2 \frac{1}{e_o}\right) + \ln p - \ln(1-p) = \frac{1}{1-p}
\end{aligned}$$

[1] $W_0(x)$ is the Lambert-W function on the branch where $x \in [-1/e, +\infty)$ and $w \geq -1$; $W_{-1}(x)$ is the Lambert-W function on the branch where $x \in [-1/e, 0)$ and $w \leq -1$. Please see [1] for details.

For convenience, we let $a = \ln 2(V + \log_2 \frac{1}{e_o})$, and $x = 1 - p$, then,

$$\begin{aligned}
& a + \ln(1-x) - \ln x = \frac{1}{x} \\
& \Rightarrow ax + x\ln(1-x) - x\ln x - 1 = 0 \\
& \Rightarrow e^{ax + x\ln(1-x) - x\ln x - 1} = \frac{e^{ax + x\ln(1-x)}}{e^{x\ln x + 1}} = e^0 = 1 \\
& \Rightarrow \frac{e^{ax}(1-x)^x}{ex^x} = 1 \\
& \Rightarrow e^{ax}(1-x)^x = ex^x \\
& \Rightarrow \frac{(1-x)^x}{x^x} = \frac{e}{e^{ax}} \\
& \Rightarrow \left(\frac{1-x}{x}\right)^x = \frac{1}{e^{ax-1}} \\
& \Rightarrow \left(\frac{1}{x} - 1\right)^x = \frac{1}{e^{ax-1}}
\end{aligned}$$

For convenience, we let $t = \frac{1}{x} - 1$, therefore $x = \frac{1}{t+1}$, and,

$$\begin{aligned}
& t^{\frac{1}{t+1}} = \frac{1}{e^{\frac{a}{t+1} - 1}} \\
& \Rightarrow \ln t^{\frac{1}{t+1}} = \ln \frac{1}{e^{\frac{a}{t+1} - 1}} \\
& \Rightarrow \frac{\ln t}{t+1} = 1 - \frac{a}{t+1} \\
& \Rightarrow \ln t = t + 1 - a \\
& \Rightarrow e^{\ln t} = e^{t+1-a} \\
& \Rightarrow t = e^{1-a}e^t \\
& \Rightarrow te^{-t} = e^{1-a} \\
& \Rightarrow (-t)e^{(-t)} = -e^{1-a}
\end{aligned}$$

It is provided that $V + \log_2 \frac{1}{e_o} \geq 3$, therefore $a = \ln 2 \times (V + \log_2 \frac{1}{e_o}) \geq 2$, then $-e^{1-a} \in (-1/e, 0)$. According to the *properties of Lambert-W function*, the transcendental equation has two real roots:

$$-t = W_0(-e^{1-a}) \ or \ -t = W_{-1}(-e^{1-a})$$

We consider these two roots respectively.

If $-t = W_0(-e^{1-a})$, then according to the *properties of Lambert-W function*, $-t \in (-1, 0)$, therefore $t \in (0, 1)$. And, $t = \frac{1}{x} - 1$ and $x = 1 - p$, therefore

$p = 1 - \frac{1}{1+t} \in (0, 1/2)$. If $-t = W_{-1}(-e^{1-a}$, we can similarly get that $t \in (1, +\infty)$ and $p = 1 - \frac{1}{1+t} \in (1/2, 1)$.

For both roots, $f'(p) = 0$, therefore:

$$\begin{aligned} f'(p) &= \frac{-1}{p^2}\left[V + \log_2 \frac{1}{e_o} + \log_2 \frac{p}{1-p}\right] + \frac{1}{p^2} \times \frac{1}{\ln 2 \times (1-p)} = 0 \\ \Rightarrow &-\left[V + \log_2 \frac{1}{e_o} + \log_2 \frac{p}{1-p}\right] + \frac{1}{\ln 2 \times (1-p)} = 0 \end{aligned}$$

We take the second derivative of $f(p)$:

$$\begin{aligned} f''(p) &= (f'(p))' \\ &= \frac{2}{p^3}\left[\left(V + \log_2 \frac{1}{e_o}\right) + \log_2 \frac{p}{1-p}\right] - \frac{1}{p^3}\frac{1}{\ln 2 \times (1-p)} \\ &+ \frac{-2}{p^3}\frac{1}{\ln 2 \times (1-p)} + \frac{1}{p^2}\frac{1}{\ln 2 \times (1-p)^2} \\ &= \frac{-2}{p^3}\left[-\left(V + \log_2 \frac{1}{e_o} + \log_2 \frac{p}{1-p}\right) + \frac{1}{\ln 2 \times (1-p)}\right] \\ &+ \frac{1}{\ln 2 p^2 (1-p)}\left(\frac{1}{1-p} - \frac{1}{p}\right) \\ &= \frac{1}{\ln 2 p^2 (1-p)}\left(\frac{1}{1-p} - \frac{1}{p}\right) \end{aligned}$$

If $-t = W_0(-e^{1-a})$, then $p \in (0, 1/2)$, therefore $\frac{1}{1-p} < \frac{1}{p}$ and $f''(p) < 0$, hence this point is the maximum of $f(p)$, which contradicts with the condition of minimum space.

If $-t = W_{-1}(-e^{1-a})$, then $p \in (1/2, 1)$, therefore $\frac{1}{1-p} > \frac{1}{p}$ and $f''(p) > 0$, hence this point is the minimum of $f(p)$. Therefore, the minimum value of $f(p)$ space is reached when the load factor is

$$p = 1 - \frac{1}{1 - W_{-1}(a)} = 1 - \frac{1}{1 - W_{-1}(\ln 2(V + \log_2 \frac{1}{e_o}))} \qquad \blacksquare$$

Lemma 3.5.1 is based on the condition that $V + \log_2 \frac{1}{e_o} \geq 3$, but this condition is satisfied by most applications. In Geometric filter, $V \geq 1$, otherwise the Geometric filter cannot represent any data; and in most applications, the false positive rate $e_o \leq \frac{1}{2^{12}}$ [29], otherwise the data stored in Geometric filter would not have enough accuracy; therefore $V + \log_2 \frac{1}{e_o} \geq 15 > 3$.

3.5.2 Setting values for the parameters

Lemma 3.5.1 provides a close estimation of the load factor p. However, because of the ceiling in Eq. 3.21, the value of p provided by the lemma may be not equal to the actual setting. As Figure 3.2 shows, the curve with ceiling is sectionally continuous, and on each continuous section, the minimal point intersects with the curve without ceiling. The overall minimum point is one of these intersections. If the minimum point of the non-ceiling curve is between two intersections, then one of these two intersections is the actual minimum point.

Lemma 3.5.2 *Provided the false positive rate e_o and the number of error bits E, then for each intersection point of curve $f(p) = \frac{1}{p}(V + \lceil \log_2 \frac{p}{(1-p)e_o} \rceil)$ and curve $f(p) = \frac{1}{p}(V + \log_2 \frac{p}{(1-p)e_o})$, the corresponding load factor:*

$$p = 1 - \frac{1}{e_o 2^E + 1}$$

Proof On an intersection, the ceiling and non-ceiling curves have the same value, therefore the number of error bits $E = \log_2 \lceil \frac{p}{(1-p)e_o} \rceil = \log_2 \frac{p}{e_o(1-p)}$, then

$$\begin{aligned} & E = \log_2 \frac{p}{e_o(1-p)} \\ & \Rightarrow \frac{p}{e_o(1-p)} = 2^E \\ & \Rightarrow -1 + \frac{1}{1-p} = e_o 2^E \\ & \Rightarrow \frac{1}{1-p} = e_o 2^E + 1 \\ & \Rightarrow 1 - p = \frac{1}{e_o 2^E + 1} \\ & \Rightarrow p = 1 - \frac{1}{e_o 2^E + 1} \end{aligned}$$

∎

Setting the load factor p and error bits E. Suppose the load factor calculated by Lemma 3.5.1 is $\hat{p}$, let E_0 be an integer such that $E_o \leq \log_2 \frac{\hat{p}}{(1-\hat{p})e_o} \leq E_0 + 1$, then in the actual setting, $E = E_0$ or $E_0 + 1$. By Lemma 3.5.2, $p = 1 - \frac{1}{e_o 2^{E_0} + 1}$ or

$1 - \frac{1}{e_o 2^{E_0+1}+1}$. With these two candidate values of p, we calculate the value of Eq. 3.22, the one with the smaller value will be chosen. With p determined, E is also determined correspondingly.

In our example, if $V = 16$ and the allowed error rate $e_o = 1/2^{16}$. By Lemma 3.5.1, estimated load factor $\hat{p} = 1 - 1/\left(1 - W_{-1}\left(-e^{1-\ln 2\times\left(V+\log_2 \frac{1}{e_o}\right)}\right)\right) = 0.9606$, therefore $\log_2 \frac{p}{e_o(1-p)} = 20.6073$. Because $20 \le 20.6073 \le 21$, $E = 20$ or 21. Correspondingly, $p = 1 - \frac{1}{e_o 2^E+1} = 0.9412$ or 0.9697. Because $space_G|_{p=0.9606} > space_G|_{p=0.9697}$ (Eq. 3.22), we choose $p = 0.9697$ and $E = 21$.

Setting of the storage ratio r. Given k, by Lemma 3.3.3, $p = p_k(r)$, and the inverse function $r = p_k^{-1}(p)$ exists, therefore r is determined when p is known. For clarity, function $p_k^{-1}(p)$ is also denoted as $r_k(p)$.

When $k = 1$, by Eq. 3.11, $p_1(r) = 1 - e^{-1/r}$, therefore the inverse function:

$$r_1(p) = p_1^{-1}(p) = -\frac{1}{\ln(1-p)} \tag{3.23}$$

In our example, given $k = 1$ and $p = 0.9697$, by Eq. 3.23, $r = 0.2860$.

When $k > 1$, function $r_k(p)$ can also be calculated by a Monte Carlo experiment, as Algorithm 7. The return value of r does not depend on M; however, for the reason discussed in Section 3.3.3, M needs to be greater or equal to the largest hash table size in the Geometric filter, so that the hash function's non-uniform distribution does not harm the overall performance.

Setting of the length of hash table sequence N. With p, r and $|S|$ known, according to Eq. 3.16, the length of hash table sequence $N = \lceil \log_{1-r\times p}(1/|S|) \rceil$. In our example, $|S| = 4 \times 10^8$, therefore $N = 61$.

Constructing the hash table sequence With all the parameters set, we can construct the Geometric filter, in which the hash table sizes $M_i = r|S| \times d^{i-1}$, $1 \le i \le N$, where $d = 1 - r \times p$. In our example, the sizes of the constructed Geometric filter is shown in Figure 3.3.

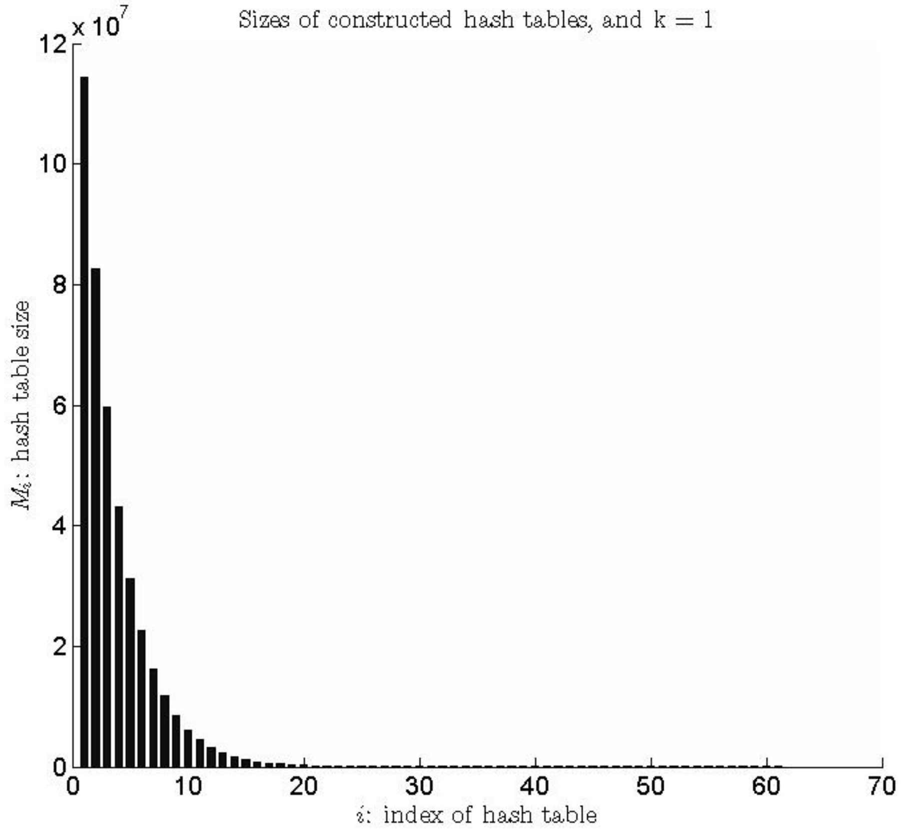


Figure 3.3: Hash tables in the Geometric filter. The configuration is: $k = 1$, $r = 0.2860$, $p = 0.9697$, $N = 61$, and $|S| = 4 \times 10^8$. The size of hash table is measured by its location number.

Algorithm 7: storageRatio(p, k, M)

```
input  : $p$: load factor
         $k$: number of hash functions
         $M$: size of hash table
output: $r$: storage ratio
$|S| \leftarrow M \times p$
allocate bitarray $bitarray[M]$
$bitarray[M] \leftarrow all\ 0s$
$c \leftarrow 0$
while $c < |S|$ do
    for $j \leftarrow 1\ to\ k$ do
        $h \leftarrow random() \mod M$
        if $bitarray[h] = 0$ then
            $bitarray[h] \leftarrow 1$
            $c \leftarrow c + 1$
            break
$r \leftarrow M/|S|$
return $r$
```

3.6 Algorithm analysis

In this section, we provide a theoretical analysis of Geometric filter, including the running time of the algorithms, and a comparison of space usage between Geometric filter and Bloomier filter.

3.6.1 Running time of encoding

Because we need to encode the whole dataset before using the Geometric filter as a lookup table of that dataset, we investigate the running time for encoding the whole dataset S, not for encoding an individual item. According to the encoding algorithm (Algorithm 3), each key will be tried k times on each hash table, until it finds an available location. For simplicity, we assume all the k trials are finished, despite that the key may find an available location before finishing all the k trials, therefore we get an upper-bound of the encoding time, and this simplification will also be applied to the analysis in this section.

According to the definition of trying set, the number of keys tried on hash table

A_i is $|S_i|$, where $1 \leq i \leq N$. For any two adjacent trying set S_i and S_{i+1}, the reject rate $d = \frac{|S_{i+1}|}{|S_i|}$, and $d = 1 - r \times p$ (Eq. 3.13). Then an upper-bound of the running time of encoding S:

$$\begin{aligned}
time(encoding\ S) &\leq \sum_{i=1}^{N}(|S_i| \times k) \\
&= k \times |S| \times \sum_{i=1}^{N} d^{i-1} \\
&\leq k \times |S| \times \lim_{N\to\infty} \sum_{i=1}^{N} d^{i-1} \\
&= \frac{k}{1-d} \times |S| = \frac{k}{1-(1-r \times p)} \times |S| \\
&= \frac{k}{r \times p}|S|
\end{aligned} \tag{3.24}$$

Given k, p, and r, the encoding time of dataset S is linear on the dataset size $|S|$. In our example, the Geometric filter has $k = 1$, $p = 0.9697$, and $r = 0.2860$, hence the encoding time $time(encoding) = |S| \times k/(r \times p) = 3.6058|S|$. Considering the preprocessing of bipartite matching, the encoding of Geometric filter is almost two times faster than the encoding of Bloomier filter.

3.6.2 Running time of querying

The running time of querying a key is discussed in two situations: the key is found, and the key is not found.

If the key is found, ignoring the false positives, the match happens on the location where the key is stored. Moreover, we also neglect the overflow table, because the overflow table size $|S_{N+1}|$ is negligible. If the query result is a match, then the search tries the same number of locations as the encoding of the item, therefore the average of running time of a matched querying is the same with the average encoding time of each item in the dataset. According Eq. 3.24, the expectation of query time is

$$time(matching\ x) \leq \frac{k}{p \times r} \tag{3.25}$$

With our illustration example, the Geometric filter has $k = 1$, $p = 0.9697$, and $r = 0.2860$, then the running time $time(match\ x) = k/(r \times p) = 3.6058$.

If the key is not matched, according to Algorithm 4, we need to consider two situations: 1) the key is tested on an empty location; or 2) the key is tried on all the hash tables but did not match any, and there is also no match for the key in the overflow table. In the following equation, $Ept.A_i$ means the key hits an empty location in hash table A_i, and $time(overflow)$ means the running time of linearly searching the key in the overflow table. Then, the expectation of running time is:

$$\begin{aligned}
&time(\neg matching\ x) \\
&\leq k \times (1 + P(\neg Ept.A_1) + P(\neg Ept.A_2|\neg Ept.A_1) + \dots \\
&+ P(\neg Ept.A_N|\neg Ept.A_1, A_2, \dots A_{N-1})) \\
&+ time(overflow) \times P(\neg Ept.A_1, \dots A_N) \\
&= k \times (1 + p^k + p^{2k} + \dots p^{(N-1)k}) \\
&+ time(overflow) \times p^{N \times k} \\
&= k \times \sum_{i=0}^{N} p^{i \times k} + time(overflow) \times p^{N \times k} \\
&= \frac{k(1 - p^{k \times N})}{1 - p^k} + time(overflow) \times p^{N \times k}
\end{aligned}$$

Given a correct configuration, as discussed in Section 3.3.3 and Section 4.2, the load factors of the hash tables at the end of the sequence are lower than the hash tables in the front, because the real hash functions do not have uniform distribution. If we ignore the running time of accessing the overflow table, the querying time can be simplified as Eq. 3.26, which is an upper bound of the actual running time.

$$\frac{k(1 - p^{k \times N})}{1 - p^k} \tag{3.26}$$

In our example, the Geometric filter has $k = 1$, $p = 0.9697$, and $r = 0.2860$. For the query of an item not matched, the running time $time(\neg matching\ x) = k/(1 - p^k) + p^{N \times k} time(overflow) = 27.9516$. This is the upper-bound, and the running time in reality is lower than it.

3.6.3 Comparison of memory space usage of Geometric filter and Bloomier filter

Now we compare the memory space needed by Geometric filter and Bloomier filter for encoding the same dataset S, with V value bits, and false positive rate e_o. As discussed in Section 2.2.2, Bloomier filter needs additional memory space for preprocessing the dataset; however, in this section, we do not consider the preprocessing.

For Geometric filter, according to Lemma 3.5.1, the memory usage is

$$space_G = \sum M \times c_G = \frac{|S|}{p} \times \left[\left(V + \log_2 \frac{1}{e_o} \right) + \log_2 \frac{p}{1-p} \right].$$

where

$$p = 1 - \frac{1}{1 - W_{-1}\left(-e^{1-\ln 2 \times \left(V + \log_2 \frac{1}{e_o}\right)}\right)}.$$

For Bloomier filter, the minimum memory space is reached when $k = 3$ [5] [19], and the minimum memory usage

$$space_B = M \times c_B = 1.23 \left(V + \left\lceil \log_2 \frac{1}{e_o} \right\rceil \right).$$

Encoding the same dataset S, whether Geometric filter use more or less memory than Bloomier filter, depends on $V + \log_2 \frac{1}{e_o}$, and for convenience, we let $c = V + \log_2 \frac{1}{e_o}$, and we plot the curves of $\frac{space_G}{space_B}$ for c from 15 to 40.

As Figure 3.4 shows, when $c \geq 26$, $space_G/space_B < 1$ which means Geometric filter uses less space than Bloomier filter. $c = V + \log_2 \frac{1}{e_o}$ is close to $V + \lceil \log_2 \frac{1}{e_o} \rceil$, which is the number of bits per location in Bloomier filter, therefore the figure above actually shows that when the Bloomier filter needs more than 26 bits per location, then the Geometric filter uses less space than it.

Talbot et al. [29] presents that in order to guarantee the accuracy, $E \geq 12$ in the Bloomier filter. Assuming the number of value bits $V \geq 3$, most applications of Bloomier filter need more than 15 bits per location, which means $space_G \leq 1.1 space_B$. Therefore, the Geometric filter's space usage is close or less

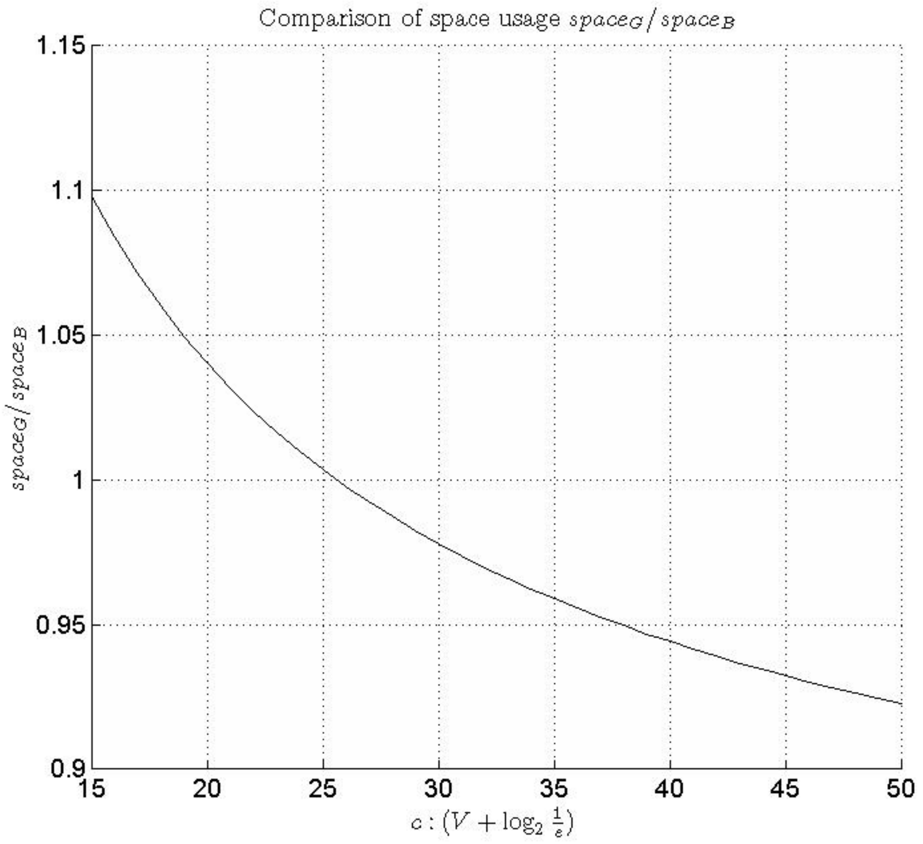


Figure 3.4: Comparison of memory space usage ($space_G / space_B$), for $c = V + \log_2 \frac{1}{e_o}$ from 15 to 40

than the Bloomier filter in most cases. And, Bloomier filter's minimum space usage $1.23|S| \times c$ is the theoretical lower bound of the lookup table [5], hence Geometric filter's space usage is close to the theoretical lower bound.

Chapter 4

Experimental Study

In this section, we use experiments to verify the results of theoretical analysis of Geometric filter, including the space usage, running time, and false positive rate.

4.1 Experiment settings

The encoding datasets are chosen from Google 1-TB n-gram corpus, and the files are sequentially named from 3gm-0000 to 3gm-0039, each containing 10^7 items.

The allowed false positive rate $e_o = 1/2^{16}$, and the value bit number $V = 16$. In different experiments, the dataset size $|S|$ varies from 10^7 to 4×10^8, and number of hash functions k varies from 1 to 6.

4.2 Adjustment of load factor and storage ratio

As discussed in Section 3.3.3, because of hash function's non-uniform distribution, with the same storage ratio, the actual load factor in a hash table is slightly lower than the theoretical estimate, and the load factor on a large hash table is also lower than the load factor on a small hash table.

For example, if $e_o = 1/2^{16}$ and $V = 16$, then the minimum memory space is reached when $p = 0.9697$. When $k = 1$, we can use a formula to calculate $r = -1/\ln(1-p) = 0.2860$. For $|S| = 4 \times 10^8$, the theoretically calculated parameter settings are $r = 0.2860$, $N = 61$, $E = 21$. However, according to Monte Carlo experiment, $r = 0.2822$, which is lower than the theoretically calculated value. This means the actually achieved load factor in the hash table is smaller than

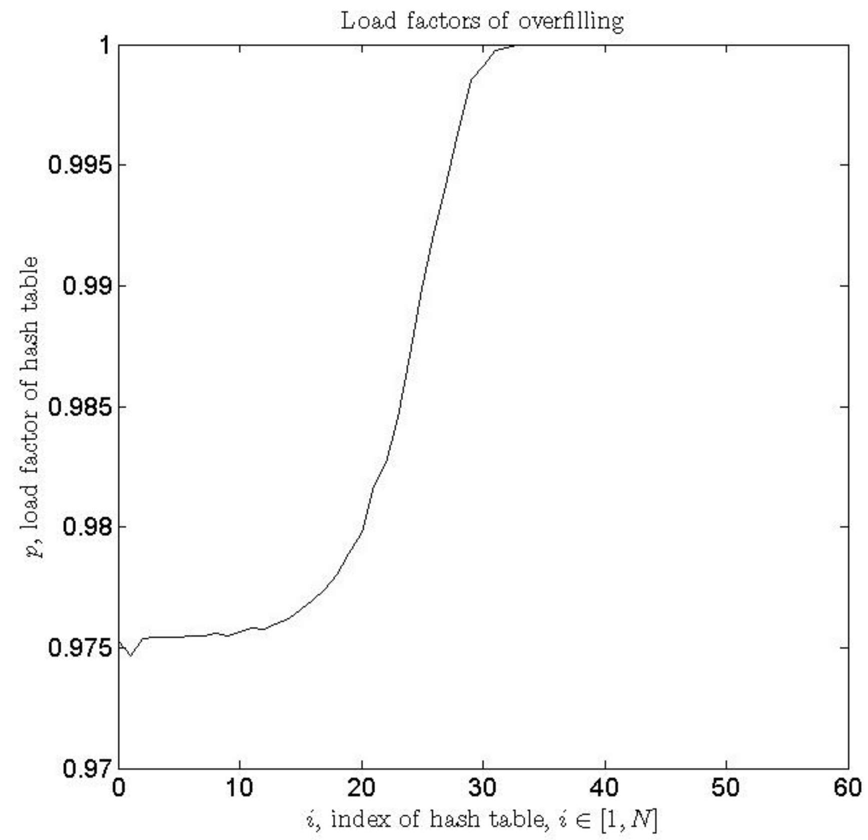


Figure 4.1: If $|S| = 40 \times 10^8$, over filling happens for $p = 0.9697$ and $k = 1$ when $r = 0.2860$ but its correct value should be 0.2822

the theoretical value, hence it does not absorb the designed amount of items, and the extra rejected items will be pushed to the subsequent hash tables in the descending geometric sequence.

If the Geometric filter is constructed with $r = 0.2860$, as Figure 4.1 shows, the load factors of hash tables raise up to 100%. And, as a result, the overflow table's size is 41260, and the probability of searching overflow table when querying an un-encoded key is almost 100%. The overflow table is too large, and also accessed too often. The change in the value of p in Figure 3.1 seems insignificant, but Geometric filter's performance is significantly affected.

In real applications, if we use the theoretical formula to calculate r or p, then the values need to be adjusted. If r remain unchanged then p needs to be deducted by a small amount (on the 3rd or 4th digit after the decimal); if p remains the same then

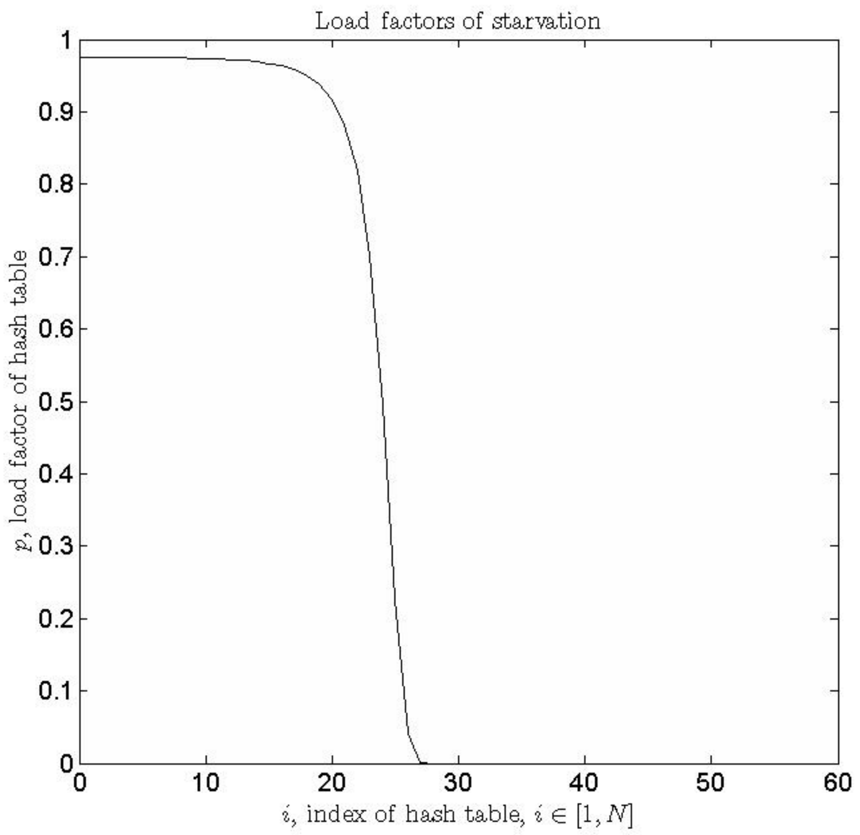


Figure 4.2: Starvation happens in the last a few hash tables when the configuration is correct (For $|S| = 40 \times 10^8$, $p = 0.9697$, $k = 1$ and $r = 0.2822$)

r needs to be deducted by a small amount. For simplicity, the values of r used in the experiments are all obtained by Monte Carlo experiments on a bitarray of size 4×10^8, which is larger or equal to the $|S|$ of the experiments.

If the value of r and p are correct for the largest hash table, then in the following smaller hash tables, the value of p will be higher than the largest hash table provided the same r. At first, the hash tables following the largest one absorbs more items than designed, thus the number of items going to the subsequent hash table drops correspondingly. If $k = 1$, and $r = 0.2822$, as Figure 4.2 shows, the hash tables at the end of the sequence do not get any item, because all items are absorbed by hash tables before them, hence their load factors drop to zero. The lower load factors do not harm the Geometric filter's performance; instead, it reduces the false positive rate and makes the querying speed faster.

Configurations					e_o ($\times 10^{-5}$)		$over$
k	p	E	r	N	allowed	actual	$flows$
1	0.9697	21	0.2822	70	1.526	0.712	1006
2	0.9697	21	0.4738	36	1.526	0.758	1070
3	0.9697	21	0.6039	26	1.526	0.703	1004
4	0.9697	21	0.6946	20	1.526	0.866	930
5	0.9697	21	0.7598	17	1.526	0.840	988
6	0.9697	21	0.8081	15	1.526	0.927	914

Table 4.1: Details of Geometric filter experiments, with parameter settings for $e_o = 1/2^{16}$, $V = 16$, and $|S| = 4 \times 10^8$. The configurations in this table can also be used for encoding smaller datasets

4.3 False positive rate and overflow table size

Because $e_o = 1/2^{16}$ and $V = 16$, the minimum space is reached when $p = 0.9697$ and $E = 21$ (the detailed steps are provided in Section 3.5.2). Then, we calculate the value of r with regard to k from 1 to 6 by Monte Carlo experiments (Algorithm 7) on a bitarray of size 4×10^8, therefore these values can be used on datasets of size 4×10^8 or smaller.

After parameter setting, we construct the Geometric filter, encode the dataset of size $S = 4 \times 10^8$ into it, and test the false positive rate e_o, as Table 4.1 shows. In order to test e_o, we generate $|S| = 4 \times 10^8$ different keys not belonging to the encoded dataset S, and measure how many of these keys can get positive return value from querying the Geometric filter. This kind of test is referred to as "negative test" ($\neg test$).

As the Table 4.1 shows, Geometric filter false positive rate $e_o \leq 1/2^{16}$, and the overflow table size is negligible compared with the dataset size. The actual false positive rate is smaller than the allowed false positive rate $e_o = 1/2^{16}$, because of the following two reasons: 1) the quantitative formulas of false positive rate is an upper bound, and the simplifications, such as letting $N \to \infty$, could enlarge the value; 2) the hash tables in the end of the sequence actually have lower load factors than the designed p, therefore the overall false positive rate drops correspondingly.

4.4 Comparison of Geometric filter and Bloomier filter

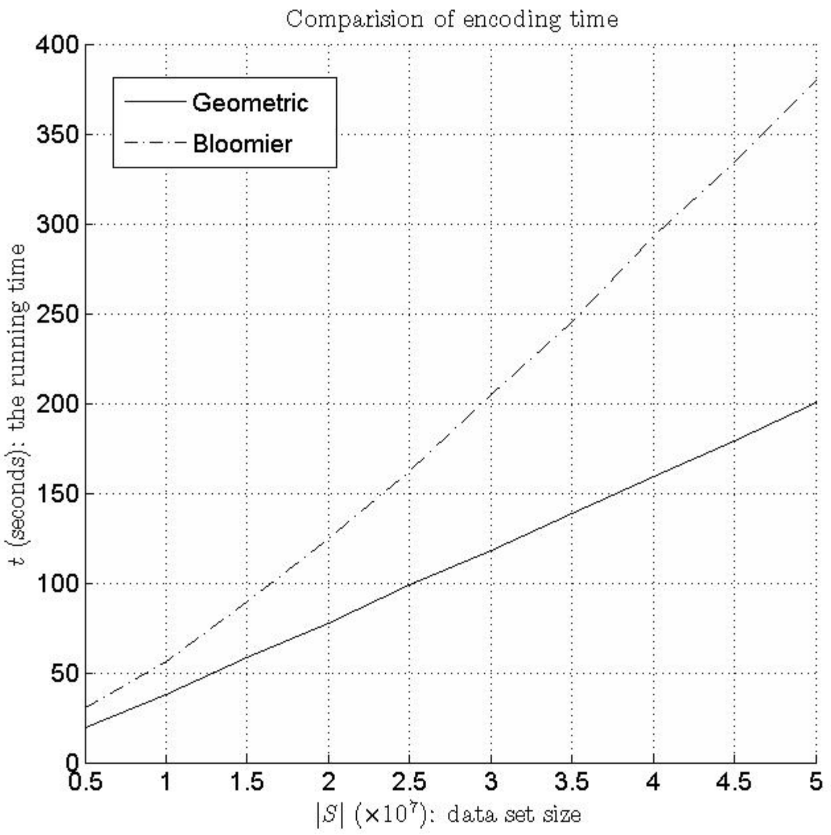
Under the condition of $e_o = 1/2^{16}$ and $V = 16$, for 10 datasets sized from 0.5×10^7 to 5×10^7, we encode them with both Geometric filter and Bloomier filter, and compare the performance of the two data structures, including the space, the running time for encoding, and the running time for querying. The experiments are running on the same dual CPU system: Intel(R) Core(TM)2 Duo CPU E8400 @ 3.00GHz $\times$ 2, and we ensure that the program can always get its required amount of memory, and no data is stored in swap.

The configuration for Bloomier filter is $k = 3$ and $r = 1.23$, which is the minimum space of Bloomier filter. For Geometric filter, k may have different values, and the configuration also changes with the value of k. In our experiment, we encode the datasets with k from 1 to 6, and compare their results with the Bloomier filter's. However, as Table 4.2 shows the performance of encoding with different values of k are close to each other, and a small figure cannot show the 6 curves clearly; therefore the value used for comparison is an average of the 6 experiments.

The results are shown in Figures 4.3 and 4.4. In the figures, *positive test* means querying all the keys in S and checking whether the filter can return the correct value. *negative test ($\neg$ test)* means generating $|S|$ keys that do not belong to S, and testing whether the filter return 0 value for these keys.

We also measure Geometric filter's running time on a large dataset with size $|S| = 4 \times 10^8$. The detailed configurations are the same as in Table 4.1, and the results are listed in Table 4.2. We cannot run Bloomier filter on this large dataset, because the bipartite matching for this dataset would require about 12.3 GB memory, which is not feasible for most current machines. If the bipartite matching could be processed, the encoded filter would be 1.88 GB, which is still larger than Geometric filter's space usage listed in Table 4.2.

The experiments illustrate two advantages of Geometric filter over Bloomier filter: 1) when the dataset is small, Geometric filter takes less time for encoding the dataset; 2) Bloomier filter is not able to encode very large datasets because the size



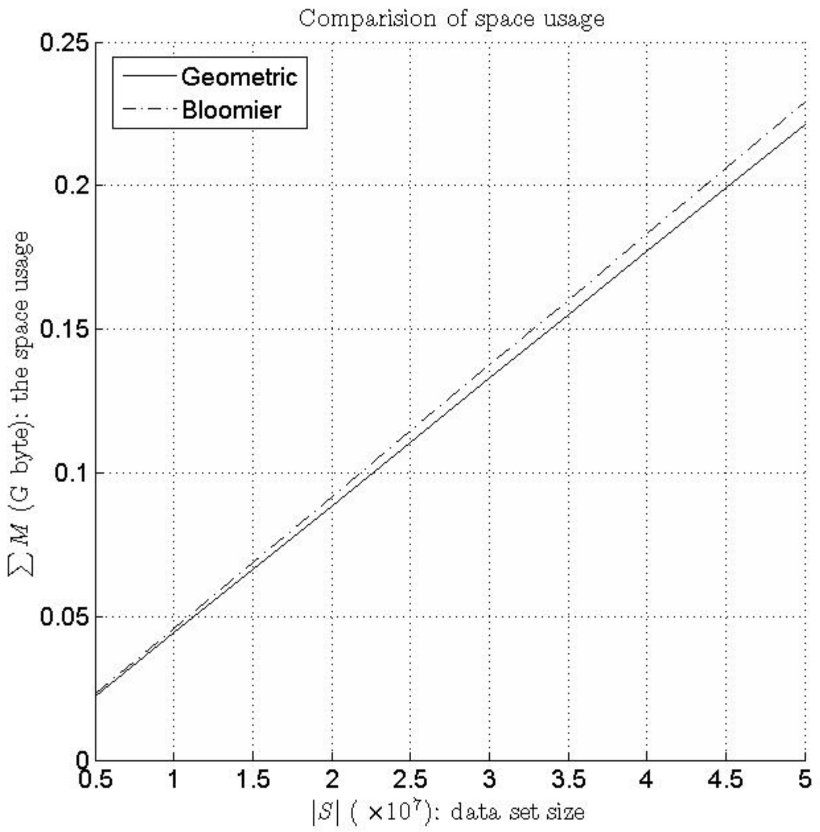
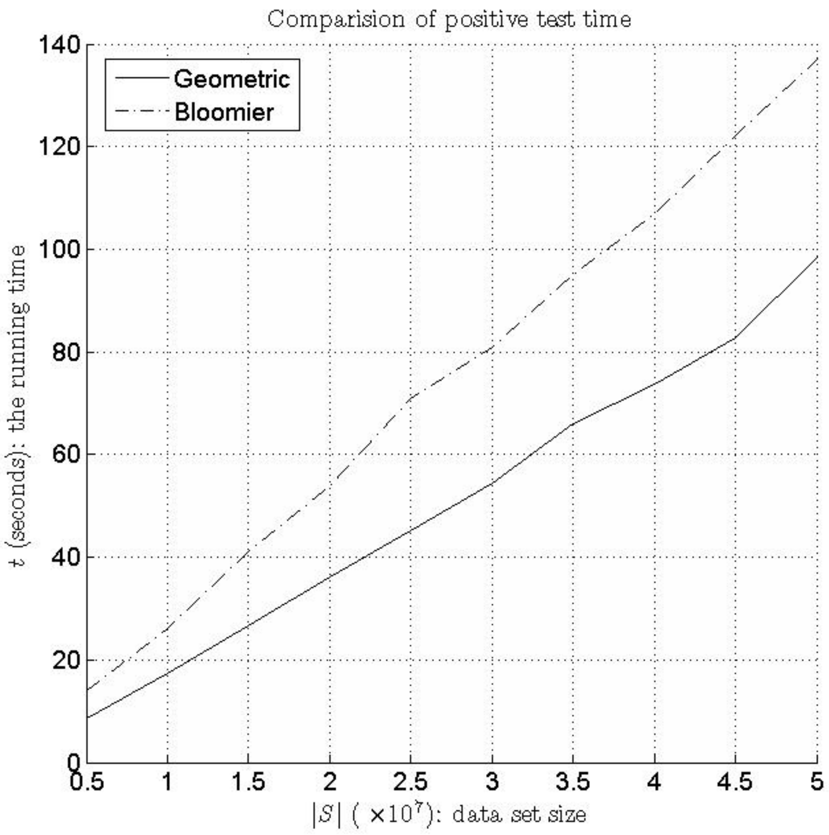


Figure 4.3: Comparison of Geometric filter and Bloomier filter (encoding time and space usage), when $|S| \in [0.5, 5.0] \times 10^7$, $E = 16$, and $V = 16$



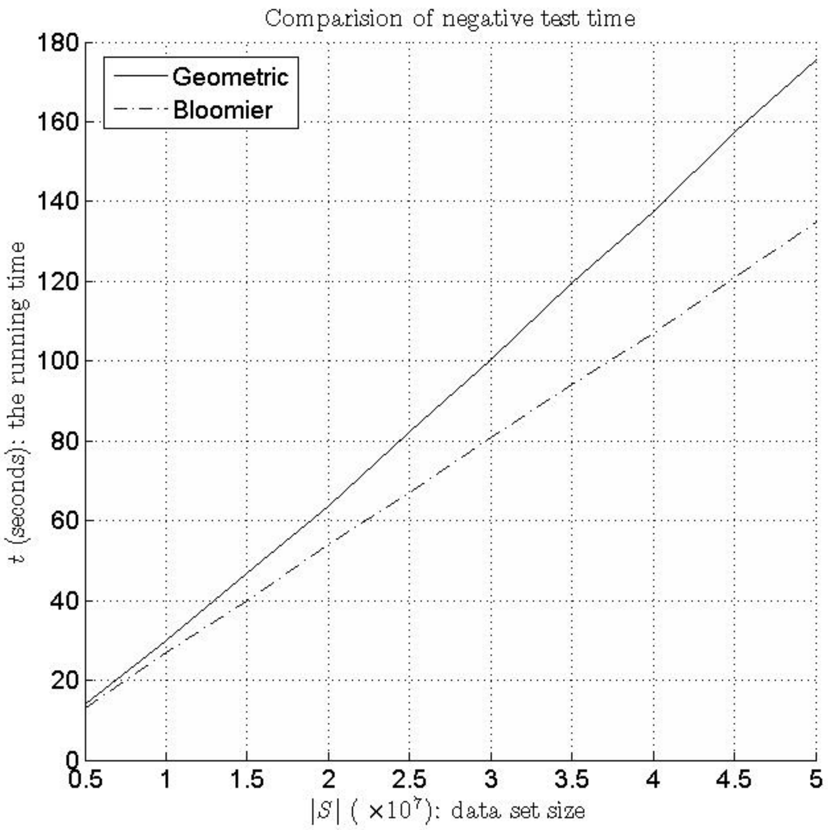


Figure 4.4: Comparison of Geometric filter and Bloomier filter (running time of testings), when $|S| \in [0.5, 5.0] \times 10^7$, $E = 16$, and $V = 16$

k	1st scan	2nd scan	test	$\neg$ test
1	1026	1267	1149	1802
2	1049	1233	1142	1722
3	975	1195	1084	1738
4	1018	1207	1106	1784
5	989	1197	1105	1797
6	993	1206	1105	1873

Table 4.2: Running time of Geometric filter, when $|S| = 4 \times 10^8$, $E = 16$, and $V = 16$

of bipartite graph would exceed the memory's capacity; however, Geometric filter does not have this problem.

4.5 Tradeoff between hash table number N and hash function number k

As Table 4.3 shows, given the same dataset size $|S|$, when the hash function number k becomes larger, the hash table number N becomes smaller. As Table 4.2 shows, this trade off affects the Geometric filter's performance.

On one hand, a small N benefits the Geometric filter's running time. When switching from one hash table to the next, the algorithm needs to reset seeds of the hash functions, and to change other parameters, like the hash table's address, etc. These operations would add to the program's running time. The smaller the N is, the less times of switching there would be. As Table 4.3 shows, when k increases, N would drop and the running time would drop with N.

On the other hand, a small N may also harm the performance. When N decreases, $k \times N$ has a slow but steady trend of increasing, which means the average times of trial for querying a key is actually increasing, slowly but steadily. A small N corresponds to a large k, which means r would become large. When r is large, the size of the first hash table $|S| \times r$ is also large. As discussed before, hash function's nonuniform distribution is more significant on large hash tables, therefore a large hash tables has more hash collisions. Hash collisions will reduce the efficiency of storage, and the increase of $k \times N$ means the program needs to perform

k	N	p	$e(\times 10^{-5})$	$N \times k$
1	70	0.9697	0.712	55
2	36	0.9697	0.758	72
3	26	0.9697	0.703	78
4	20	0.9697	0.866	80
5	17	0.9697	0.840	85
6	15	0.9697	0.927	90

Table 4.3: Tradeoff between N and k, when $|S| = 4 \times 10^8$, $E = 16$, and $V = 16$

more operations to store the same amount of data. As k increases, false positive rate e increases while space usage increases, which means we pay more space but get an even higher error rate. As a result, N cannot be too small.

Configuration with large k and small N is feasible when the hash function perfectness is improved. For example, given the same range of hash function, we can improve its distribution by increasing the range of the basis hash function. However, an increased range of basis hash function also means an increase of running time. If N decreases to 1, Geometric filter would reduce to a common hash table with rehashing. However, this extreme situation is not realistic, because when the hash table is too large, its efficiency is low.

Chapter 5

Conclusions and future work

This thesis presents algorithms and theoretical analysis of a Geometric filter, an efficient in-memory lookup tables for binary relations.

5.1 Geometric filter's advantages and drawbacks

Geometric filter is an extension of the Bloomier filter, a state-of-the-art data storage algorithm, for more efficient lookups. It offers some nice properties of the Bloomier filter, but also has the following advantages: 1) the encoding procedure is much faster; 2) the encoding procedure does not need any auxiliary memory except the filter's storage space; 3) the space usage is also relatively smaller than Bloomier filter, when the data precision is high.

Compared with other generic compression algorithms, Geometric filter also has the following advantages: 1) Beside the dataset size, Geometric filter's encoding does not require any knowledge of the dataset being encoded, therefore it can be used to handle unpredictable incoming data, such as on-line streaming data; 2) Geometric filter's storage format consists of two separate parts, the key x, and its value $v(x)$, the two parts do not affect each other. For the same reason, Geometric filter can be updated after encoding, because changing the encoding of one item does not affect the encoding of other items.

However, as discussed in Section 3.6.2, Geometric filter's running time is slow when a query results in a "mismatch". According to our experimental results, for a mismatch query, Geometric filter can take twice the time of Bloomier filter. Because of this drawback, Geometric filter would be most efficient for lookup tables that get “match” results for most queries; when most queries get unmatch results, the Geometric filter will be less effective.

5.2 Future applications

One application of our Geometric filter is n-gram frequency lookup table in statistical language model. Because Geometric filter does not need any auxiliary memory when encoding, the size of the dataset to be encoded linearly depends on the amount of available memory. Because of its fast querying speed, Geometric filter can also be used to store a large amount of data that needs fast accessing speed. An example is the pattern matching lookup table of an on-line virus scan program [16]. In this application, key x is the data stream being scanned in a window; a return value of 0 means the data does not contain any virus characteristic pattern; and non-zero return values means the data potentially contains virus, and the return value is a reference number or address of the next step of operation.

Because Geometric filter does not have any restriction on the value $v(x)$, it can be used to store any kind of value, including those with very complex meanings. If we consider x as the key attribute and $v(x)$ as all the rest attributes of a relation, Geometric filter can also be used to implement a relational database. $v(x)$ can be an action, and x can be a trigger event; in this case, Geometric filter can be used as an action driven table that has an extremely fast response.

Geometric filter allows updates, and is also flexible for adding in new data, therefore it can be used as an incremental counting data structure, for example in network traffic monitoring. The encoding flexibility enables it to record visits from new sources, while its updatability makes it possible to increase the visit counter if there are repeated visits from the same source.

5.3 Future improvements

Geometric filter’s querying procedure can be improved by introducing pipelining. As Algorithm 4 shows, for any hash table A_i where $i > 1$, the querying procedure accesses A_i if and only if it cannot find a match in the hash tables before A_i. Moreover, when A_i is accessed, all the hash tables before it are no longer needed for the same query. Therefore, without finishing the current query, we can allow the next query begin testing on the hash tables before the A_i. In this way, we can test multiple keys concurrently and reduce the average running time of each key. If implemented, this pipelining would be very useful on multi-processor machines.

Geometric filter's algorithm may also need to be modified according to different applications. For example, when using Geometric filter to store a relational database, it may be necessary to store x itself instead of its fingerprint $f(x)$, therefore there is no need to increase the error bits to overcome the false positives.

Bibliography

[1] D. A. Barry, P. J. Culligan-Hensley, and S. J. Barry. Real values of the w-function. *ACM Trans. Math. Softw.*, 21(2):161–171, 1995.

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

[3] K. Bratbergsengen. Hashing methods and relational algebra operations. In *VLDB '84: Proceedings of the 10th International Conference on Very Large Data Bases*, pages 323–333, San Francisco, CA, USA, 1984. Morgan Kaufmann Publishers Inc.

[4] A. Broder and M. Mitzenmacher. Network applications of bloom filters: a survey. *Internet Mathematics*, 1(4):485–509, May 2004.

[5] B. Chazelle, J. Kilian, R. Rubinfeld, and A. Tal. The bloomier filter: An efficient data structure for static support lookup tables. In *In Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 30–39, 2004.

[6] S. Cohen and Y. Matias. Spectral bloom filters. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 241–252, New York, NY, USA, 2003. ACM.

[7] S.E. Czerwinski, B.Y. Zhao, T.D. Hodes, A.D. Joseph, and R.H. Katz. An architecture for a secure service discovery service. In *MobiCom '99: Proceedings of the 5th annual ACM/IEEE international conference on Mobile computing and networking*, pages 24–35, New York, NY, USA, 1999. ACM.

[8] F. Deng and D. Rafiei. Approximately detecting duplicates for streaming data using stable bloom filters. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 25–36, New York, NY, USA, 2006. ACM.

[9] J. Ebert. A versatile data structure for edge-oriented graph algorithms. *Commun. ACM*, 30(6):513–519, 1987.

[10] L. Fan, P. Cao, J. Almeida, and A.Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Trans. Netw.*, 8(3):281–293, 2000.

[11] M.C. Francisco, C. Peery, R.P. Martin, and T.D. Nguyen. Planetp: Using gossiping to build content addressable peer-to-peer information sharing communities. Technical Report DCS-TR-487, Department of Computer Science, Rutgers University, May 2002.

[12] L.L. Gremillion. Designing a bloom filter for differential file access. *Commun. ACM*, 25(9):600–604, 1982.

[13] G. Havas, B.S. Majewski, N.C. Wormald, and Z.J. Czech. Graphs, hypergraphs and hashing. In *WG '93: Proceedings of the 19th International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 153–165, London, UK, 1994. Springer-Verlag.

[14] T.R. Hill and A. Srinivasan. Performance analysis in a differential file environment. In *WSC '85: Proceedings of the 17th conference on Winter simulation*, pages 278–283, New York, NY, USA, 1985. ACM.

[15] J. Ho and G. Lemieux. Perg: A scalable fpga-based pattern-matching engine with consolidated bloomier filters. In *International Conference on Field-Programmable Technology, 2008*, pages 73–80, Dec. 2008.

[16] J.T.L. Ho and G.F. Lemieux. Perg-rx: a hardware pattern-matching engine supporting limited regular expressions. In *FPGA '09: Proceeding of the ACM/SIGDA international symposium on Field programmable gate arrays*, pages 257–260, New York, NY, USA, 2009. ACM.

[17] Z. Li and K.A. Ross. Perf join: an alternative to twoway semijoin and bloomjoin. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*, pages 137–144. ACM Press, 1995.

[18] L.F. Mackert and G.M. Lohman. R* optimizer validation and performance evaluation for local queries. *SIGMOD Rec.*, 15(2):84–95, 1986.

[19] B.S. Majewski, N.C. Wordmald, G. Havas, and Z.J. Czech. A family of perfect hashing methods. *British Computer Journal*, 39(6):547–554, 1996.

[20] J.K. Mullin. A second look at bloom filters. *Commun. ACM*, 26(8):570–571, 1983.

[21] J.K. Mullin. Optimal semijoins for distributed database systems. *Software Engineering, IEEE Transactions on*, 16(5):558–560, May 1990.

[22] B. Pittel, J. Spencer, and N. Wormald. Sudden emergence of a giant k-core in a random graph. *J. Comb. Theory Ser. B*, 67(1):111–151, 1996.

[23] R.L. Rappaport. File structure design to facilitate on-line instantaneous updating. In *SIGMOD '75: Proceedings of the 1975 ACM SIGMOD international conference on Management of data*, pages 1–14, New York, NY, USA, 1975. ACM.

[24] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A scalable content-addressable network. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 161–172, New York, NY, USA, 2001. ACM.

[25] A. Rowstron and P. Druschel. Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility. *SIGOPS Oper. Syst. Rev.*, 35(5):188–201, 2001.

[26] J. P. Schmidt and A. Siegel. The analysis of closed hashing under limited randomness. In *STOC '90: Proceedings of the twenty-second annual ACM symposium on Theory of computing*, pages 224–234, New York, NY, USA, 1990. ACM.

[27] D.G. Severance and G.M. Lohman. Differential files: their application to the maintenance of large databases. *ACM Trans. Database Syst.*, 1(3):256–267, 1976.

[28] I. Stoica, R. Morris, D. Karger, M. Frans Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 149–160, New York, NY, USA, 2001. ACM.

[29] D. Talbot and T. Brants. Randomized language models via perfect hash functions. In *Proceedings of ACL-08: HLT*, pages 505–513, Columbus, Ohio, June 2008. Association for Computational Linguistics.

[30] D. Talbot and M. Osborne. Randomised language modelling for statistical machine translation. *In 45th Annual Meeting of the Association of Computational Linguists (To appear)*, pages 512–519, June 2007.

[31] D. Talbot and M. Osborne. Smoothed bloom filter language models: Terascale lms on the cheap. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, pages 468–476, Prague, Czech Republic, June 2007. Association for Computational Linguistics.

[32] S. Trimberger, D. Carberry, A. Johnson, and J. Wong. A time-multiplexed fpga. In *FPGAs for Custom Computing Machines, 1997. Proceedings., The 5th Annual IEEE Symposium on*, pages 22–28, Apr 1997.

[33] D. Uluski, M. Moffie, and D. Kaeli. Characterizing antivirus workload execution. *SIGARCH Comput. Archit. News*, 33(1):90–98, 2005.

[34] P. Valduriez and G. Gardarin. Join and semijoin algorithms for a multiprocessor database machine. *ACM Trans. Database Syst.*, 9(1):133–161, 1984.

[35] Mark N. Wegman and J. Lawrence Carter. New classes and applications of hash functions. In *Foundations of Computer Science, 1979., 20th Annual Symposium on*, pages 175–182, Oct. 1979.

[36] A. Whitaker and D. Wetherall. Forwarding without loops in icarus. In *Open Architectures and Network Programming Proceedings, 2002 IEEE*, pages 63–75, July 2002.