Hash Based Enhancement on Indonesia's National Election Voter Eligibility Check

I Putu Gede Wirasuta - 13517015[1]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]13517015@std.stei.itb.ac.id

***Abstract*—Large data access such as one in national election voter eligibility checks poses a challenge for service provider. Such system was introduced in 2019 in part of Indonesia's national election which perform quite poorly. This paper will discuss the possibility of applying cryptography concepts to enhance large data access. This paper uses xxHash, one of the fastest modern hash algorithms with bloom filter and hash table data structure to provide the enhancements.**

***Keywords*—Large data, Cryptography, Hash, xxHash, Bloom Filter, Hash Table**

I. Introduction

National election is the largest democratic event to be held in Indonesia. It is held every five years involving every adult citizen. In 2019, the number of eligible voters reached a sheer amount of 192.8 million individuals[1] or 71.94% of the population[2]. The number of eligible voters are expected to rise significantly in the next national election. This poses challenges for all stages of the election process, including the collection and verification of eligible voters.

In 2019, the government introduced a website to check eligibility status by the name lindungihakpilihmu[3]. In there, citizens can input their name, national identity number, and date of birth to check their eligibility status. Eligible voters are updated gradually over time, depending on when the province updates its data. The main drawback of this system is the time it took to query the status. It took about 3-5 seconds to complete the request, even if the voter is not eligible or the data inputted is not valid.

Cryptography concepts can and have been implemented to enhance everyday application. This paper will discuss the possibility of enhancement on Indonesia's election voter eligibility check using a hash-based system.

II. Hash Functions

Hash functions are functions that are able to take a message and produce a fixed length output[4]. The output is commonly referred to as digest, hash-code, hash-value, or simply hash[5]. Mathematically, a hash function (h) can be defined for domain (D) and range (R) as

$$h : D \rightarrow R \text{ and } |D| > |R|$$

Hash functions can be categorized into two categories based on how to generate the hash, keyed and unkeyed. A keyed hash function takes a message and secret to produce hash. This type of hash function is commonly used to authenticate messages. An unkeyed hash function only relies on the input message to produce hash. This type of hash function has a broader use case and will be the one to be used in this paper.

There are a number of properties of a hash function. In this paper, only uniformity and collision will be discussed further.

A. Uniformity

Uniformity is a measure of how evenly is a hash from range produced. There is no formal quantity for uniformity and it is usually not able to be inferred from the algorithm design. However, uniformity can be inferred from experiment and visualized in some way. A good hash function should be highly uniform regardless of its input.

B. Collision

Collision is an event where two different messages produce the same hash using the same algorithm[4]. Due to its nature of producing hash smaller in size than the arbitrary length message, collision is unavoidable. Collision is related to uniformity, the more uniform an algorithm is then the less likely it is to produce collision. A good hash function should also make computing the same hash from two different messages computationally infeasible.

C. Hash Function Algorithm Comparison

Several algorithms are considered to be used in this paper. Strong cryptographic hash algorithms such as SHA3 are not included due to its low bandwidth, unsuitable for this paper. Table 1 shows bandwidth and quality comparison between different algorithms. Quality is expressed as an arbitrary number ranging 1-10 based on uniformity and found collision.

Table 1. Comparison of Hash Algorithms [5]

Algorithm	Output Size (bit)	Bandwidth	Quality
MD5	128	0.6 GB/s	10

SHA1	160	0.8 GB/s	10
FNV64	64	1.2 GB/s	5
Murmur3	32	3.9 GB/s	10
xxHash64	64	19.4 GB/s	10

It should be noted that MD5 and SHA1 have been broken. FNV64 does not change as much when a section of the input is changed, leading to weaker uniformity.

For the purpose of this paper, speed is the most important aspect followed by bit length and quality. Which leads to xxHash64 as the chosen hash function algorithm to be used in the implementation.

III. Hash Table

Hash table is a data structure that builds on a regular array. But instead of using incrementing integers as its key, it uses hash of the content as its key. It is an effective data structure for key-value lookup operations. The expected complexity to search a value in a hash table is O(1) [6].

As previously mentioned, collision is unavoidable in hash results. For this reason, it is not possible to have an exact one key to one value mapping in a hash table. Rather, there are several resolutions to this problem. It will also be easier to address the content of a hash table as key and bucket rather than key and value because a single key can contain multiple values.

A. Chaining

Chaining is a method of collision resolution where a bucket entry is formed by chaining multiple values. Another possibility is to include the first value in the bucket and store the rest in an overflow container. Figure 1 and 2 shows both of these resolution strategies.

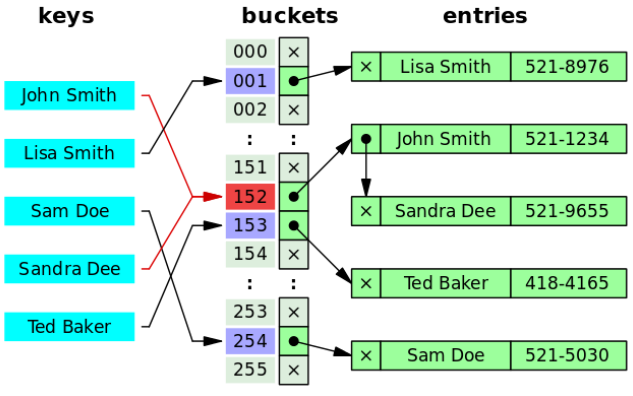

Figure 1. *Separate bucket and chain (wikimedia)*

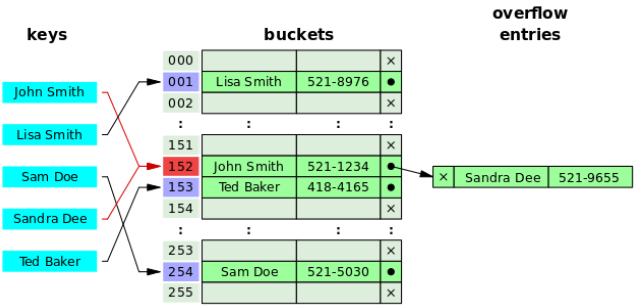

Figure 2. *First element in bucket with overflow container (wikimedia)*

A correct implementation of this resolution strategy with uniform hash would result in a relatively uniform length chain. While the expected complexity of searching a value is no longer O(1), it is still efficient at O(m + 1) with m as the expected length of chain.

B. Open Addressing

Open addressing collision resolution is formed by allocating a certain size to every bucket. Then, addition of value will start scanning from the intended hash index for an empty slot. Figure 3 shows this type of resolution strategy.

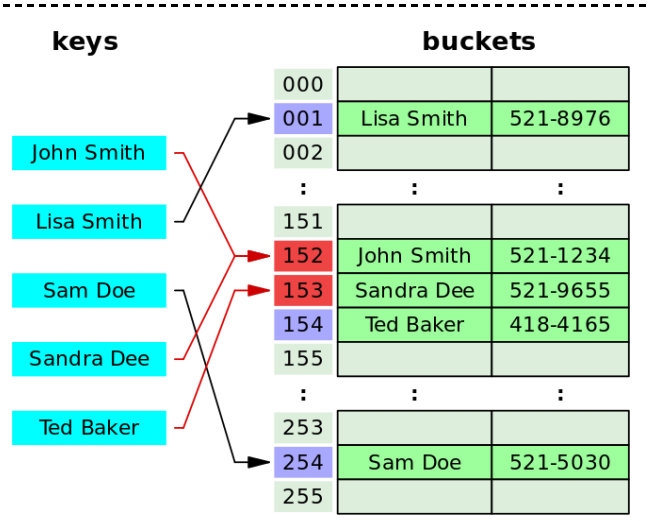

Figure 3. *Open addressing (wikimedia)*

In figure 3, note that John Smith and Sandra Dee have the same hash value but Sandra Dee is placed on the next index. This causes Ted Baker value to be placed not exactly where it's supposed to be. The main drawback of this resolution strategy is when a section of hash indexes are so populated, the expected complexity to search a value is increased significantly.

IV. Bloom Filter

Bloom filter is a data structure that gives probabilistic results if a value is in a set and a definitive result if a value is not in a set. It was introduced by Burton H Bloom in 1970[7] with the application which the great majority of values to be tested are

not present in the set.

A bloom filter is composed of n individually addressable bits and k unique hash algorithms. To add a value v into the set, get the hash of v for every hash algorithm. For every hash, mark the corresponding bit to one.

To check whether a value x might exist in the set, get the hash of x for every hash algorithm. If bits corresponding to every hash are all one, then x might be in the set. If any of the bits are zero, then x is definitely not in the set. Figure 4 shows an example of a bloom filter with x,y,z in the set and w test results in not in the set.

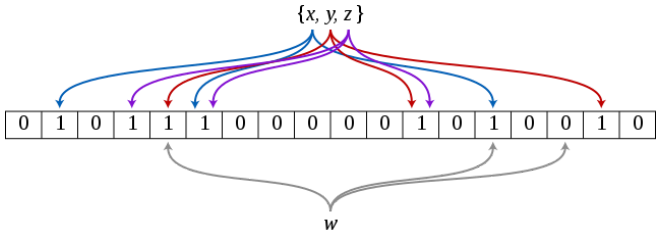

Figure 4. *Bloom filter example (wikimedia)*

The probability of false positives in a bloom filter increases as more elements are added into the set. It is possible to calculate the optimal number of required bits and unique hash algorithm under the assumption of hash independence as

$$m = -\frac{n\,ln(p)}{(ln\,2)^2}$$

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

where m is required bits, n is number of expected elements, p is the false positive probability, and k is the number of unique hash algorithms[7].

An important property of bloom filter for the purpose of this paper is that it is possible to union two bloom filters into one by simply OR-ing every bit. This will be useful when updating the value of eligible voters in batch without recalculating the filter.

There have been numerous publications on bloom filter improvement to accommodate more use cases, such as enabling element deletion and counting element instead of bits. In this paper, the original design of bloom filter is used as it is considered enough for the requirement.

V. Design and Implementation

Indonesia's national election voter eligibility check is modeled as a simple create and read service. This service receives requests and performs a simple check followed by a query to the database. It is implemented in Typescript, running in Node.js version 12. The database used is PostgreSQL with one table. Figure 5 shows the table's schema.

Column	Type	Comment
id	integer *Auto Increment* [**nextval('registered_voter_id_seq')**]	
national_id_number	character varying(20)	
name	character varying(255)	
dob	date	
gender	character(1)	
voting_location	character varying(255)	

Indexes

PRIMARY	*id*

Figure 5. *Eligible voter table schema*

Enhancements to the service are built into two modules. Both modules use xxHash as the hash function. The first module utilizes a hash table. Separate bucket and chain model is used. Sharded database act as the chain storage while value to bucket mapping is implemented in the application layer. This module aims to lower the time required to create, read, and update data.

The second module utilizes a bloom filter. The bloom filter is used to test whether a voter exists in the database. This filter is made to facilitate 200 million values with false positive probability of 0.001. The optimal number of required bits and unique hash algorithms are

$$m = -\frac{n\,ln(p)}{(ln\,2)^2} = 2.87 \times 10^9 bits = 342.7\ MiB$$

$$k = \frac{m}{n} ln(2) = 9.94 \approx 10$$

This module aims to eliminate lookup operations for non-existing eligible voters so that the request time and service load is lower.

The service and two modules are combined into four distinct endpoint variants. Every endpoint receives and returns values with the same format but different inner workings. Figure 5,6,7, and 8 illustrates the different inner workings of every endpoint.

Figure 6. */variant1 endpoint inner workings*

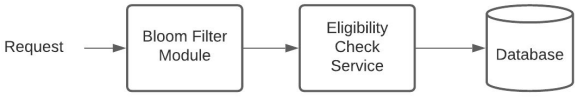

Figure 7. */variant2 endpoint inner workings*

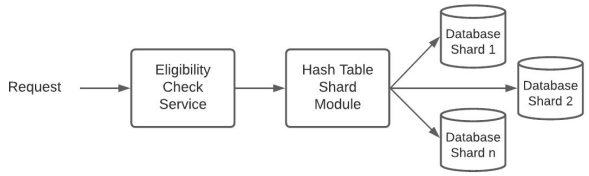

Figure 8. */variant3 endpoint inner workings*

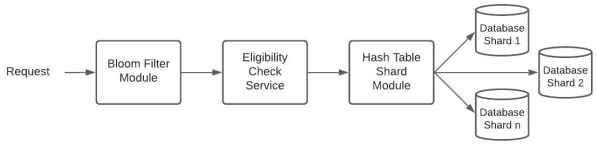

Figure 9. */variant4 endpoint inner workings*

Design process on this paper focuses on building testable prototype with hash based enhancement, so other components are left at its default configuration to minimize variability. No index is created at the database level. No cache layer is present at the application or database level.

The following is implementation of bloom filter module as service for the prototype.

```
import * as XXH from 'xxhash';
import * as uuid from 'uuid';

export class BloomFilterService {
 private bloomFilter;
 private seed = [];

 constructor(private k: number, private m: number) {
   this.m = Math.round(m);
   const requiredBytes = Math.ceil(m / 8);
   this.bloomFilter = new Uint8Array(requiredBytes);
   for (let i = 0; i < k; i++) {
     this.seed[i] = uuid.v4().slice(0, 8);
   }
 }

 add(value: string) {
   for (let i = 0; i < this.k; i++) {
     const digest = XXH.hash64(
       Buffer.from(value, 'utf8'),
       Buffer.from(this.seed[i], 'utf8'),
       'hex',
     );
     let index = parseInt(digest, 16);
     index %= this.m;
     const bloomFilterIdx = Math.floor(index / 8);
     const bitIdx = index - bloomFilterIdx * 8;

     const newBloomFilterValue =
       this.bloomFilter[bloomFilterIdx] | (1 << bitIdx);
     this.bloomFilter[bloomFilterIdx] = newBloomFilterValue;
   }
 }

 test(value: string): boolean {
   let probablyPresent = true;
   for (let i = 0; i < this.k; i++) {
     const digest = XXH.hash64(
       Buffer.from(value, 'utf8'),
       Buffer.from(this.seed[i], 'utf8'),
       'hex',
     );
     let index = parseInt(digest, 16);
     index %= this.m;
     const bloomFilterIdx = Math.floor(index / 8);
     const bitIdx = index - bloomFilterIdx * 8;

     const bitValue = (this.bloomFilter[bloomFilterIdx] >>
bitIdx) & 1;
     probablyPresent &&= bitValue == 1 ? true : false;
   }

   return probablyPresent;
 }
}
```

The following is implementation of hash table based filter as service for the prototype.

```
import * as XXH from 'xxhash';
import * as uuid from 'uuid';
import { Connection } from 'typeorm';

export class HashBasedDBService {
 private connections = [];
 private seed;

 constructor(private connections: Connection[]) {
   this.seed = uuid.v4();
 }

 getInstance(value: string) {
   const digest = XXH.hash64(
     Buffer.from(value, 'utf8'),
     Buffer.from(this.seed, 'utf8'),
     'hex',
   );
   let index = parseInt(digest, 16);
   index %= this.connections.length;
   return this.connections[index];
 }
}
```

V. Experiment

Experiment on the service is divided into four operations with two initial states. The operations are create one eligible voter, create bulk eligible voter, read one eligible voter, and read bulk eligible voter. The initial states are eligible voters and eligible voters don't exist. Combination of these operations and initial states expected results are presented in table 2.

	Eligible Voter(s) Exist	Eligible Voter(s) don't Exist
Create One	Operation succeed.	Operation fails.

Eligible Voter	Returns newly added voter.	Status code 403
Create Bulk Eligible Voter	Operation succeed. Returns all newly added voter.	Operation fails. Status code 403
Read One Eligible Voter	Operation fails. Status code 404	Operation succeed. Returns requested voter.
Read Bulk Eligible Voter	Operation fails. Status code 404	Operation succeed. Returns all requested voter.

Data used for experiment are 10 million rows generated randomly. This number is chosen due to the limitation of the experiment machine while still quite large to show a significant difference. For sharded database, 10 shards are used.

All tests are run on Ubuntu Server 20.04 with 2 vCPU and 8GB of RAM inside Docker container.

A. Create One Eligible Voter Results

Table 3 shows the experiment result of creating one eligible voter in different variant of endpoints.

Endpoint	Eligible Voter(s) Exist	Eligible Voter(s) don't Exist
/variant1	0.051s	0.038s
/variant2	0.088s	0.082s
/variant3	0.063s	0.066s
/variant4	0.10s	0.070s

B. Create Bulk Eligible Voter Results

Table 4 shows the experiment result of creating 100 eligible voters in different variant of endpoints.

Endpoint	Eligible Voter(s) Exist	Eligible Voter(s) don't Exist
/variant1	0.25s	0.24s
/variant2	0.32s	0.28s
/variant3	0.48s	0.41s
/variant4	0.50s	0.41s

C. Read One Eligible Voter Results

Table 5 shows the experiment result of reading one eligible voter in different variant of endpoints.

Endpoint	Eligible Voter(s) Exist	Eligible Voter(s) don't Exist
/variant1	1.48s	1.43s
/variant2	1.51s	0.012s
/variant3	0.85s	0.70s
/variant4	0.88s	0.010s

D. Read Bulk Eligible Voter Results

Table 6 shows the experiment result of reading 100 eligible voter in different variant of endpoints.

Endpoint	Eligible Voter(s) Exist	Eligible Voter(s) don't Exist
/variant1	27.65s	27.71s
/variant2	28.01s	0.057s
/variant3	18.43s	18.31s
/variant4	19.11s	0.062s

VI. Analysis

From the create eligible voter result, it can be seen that modern databases such as PostgreSQL already have a high bandwidth for data creation. Either creating one or on hundred data, the service is able to respond quickly due to this high bandwidth. It also shows that variant 3 is slower in all creation experiments, which suggests that sharding the data to multiple databases only hurts the performance in such situation. Variant 2 are also slower but by a large amount. This is because bloom filter can't give a definitive answer whether the data is already present in the database, which means every creation query must be performed. Variant 4 is significantly slower than the others due to its lengthy process that increases the time needed (combination of variant 2 and 3).

The results from read eligible voter experiments shed some light on the power of hash based enhancement. In both single and bulk experiments, variant 3 is faster significantly than variant 1. This is caused by the fact that the operation is parallelized to multiple databases, which resulted in a larger bandwidth. Variant 2 is significantly faster than all other variants in the case that the eligible voter(s) don't exist. This is caused by the fast rejection by bloom filter. Variant 4 produces the best results compared to other variant in both presence and absence of the eligible voter(s).

This makes a very good argument in using both bloom filter and hash table module for reading data part on a national election. For a long amount of time, the data in the national election eligibility database won't be complete. Using both of these module can save user a significant amount of time and the server a significant amount of processing resources.

VII. Conclusion

Hash based enhancement has a high chance of improving service performance in case of reading data. This paper has tested bloom filter based module and hash table based sharding module which performs significantly better than simply querying the database for reading operation. This makes the modules suitable to enhance Indonesia's national election voter eligibility check website. But this enhancement is not suitable for create operation due to its poor performance.

VIII. Acknowledgment

The author would like to thank God the Almighty for grace and blessings. The author would also thank Dr. Ir. Rinaldi, M.T. as the lecturer of Cryptography (IF4020). The author also express gratitude to family and friends for their support in the making of this paper.

References

[1] Syafii, "Jumlah Pemilih Pemilu 2019 Bertambah Jadi 192.866.254". Kompas.com.
[2] BPS, "Penduduk, Laju Pertumbuhan Penduduk, Distribusi Persentase Penduduk, Kepadatan Penduduk, dan Rasio Jenis Kelamin Penduduk Menurut Provinsi, 2019".
[3] Lingungihakpilihmu, "Lindungi Hak Pilihmu".
[4] Munir, Rinaldi. 2020. Slide Kuliah IF4020 Kriptografi: Fungsi Hash
[5] xxHash, "Comparison of Hash Algorithm".
[6] Cormen, Thomas H. 2001. Introduction to algorithms.
[7] Bloom, Burton H. 1970. Space/Time Trade-offs in Hash Coding with Allowable Errors.
[8]

Pernyataan

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 3 Desember 2020

I Putu Gede Wirasuta 13517015