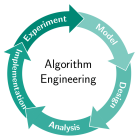
Karlsruhe Institute of Technology



Dynamic Retrievals

Bachelor's Thesis of

Loretta Jacobs

At the KIT Department of Informatics
Institute for Algorithm Engineering

First examiner:	Prof. Peter Sanders
Second examiner:	Dr. Thomas Bläsius
First advisor:	Dr. Stefan Walzer
Second advisor:	M.Sc. Stefan Hermann

01. July 2024 – 02. Dezember 2024

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

Dynamic Retrievals (Bachelor's Thesis)

I declare that I have developed and written the enclosed thesis completely by myself. I have not used any other than the aids that I have mentioned. I have marked all parts of the thesis that I have included from referenced literature, either in their original wording or paraphrasing their contents. I have followed the by-laws to implement scientific integrity at KIT.

Karlsruhe, 02. Dezember 2024

. . . . . .

(Loretta Jacobs)

Abstract

In computer science, different data structure types exist that store linked information, for example, dictionaries and retrieval data structures. Dictionaries store the key-value pairs, while retrieval data structures only store the values. Thus, retrieval data structures need less space. Most retrieval data structures are static, so they only have a construction operation and a retrieval (also known as lookup) operation, which returns the key's corresponding value.

Possible uses for retrieval data structures include filters like Bloom or XOR filters and perfect hash functions. Retrieval data structures are a growing area of research in computer science. This thesis focuses on designing a dynamic retrieval data structure that separates the space used by a lookup operation from the rest of the space. The main goals are to keep the space needed for a lookup compact and have insert, update, and delete operations with linear run times. VisionEmbedder is an existing dynamic retrieval data structure that separates the data structure into two. As VisionEmbedder had similar goals of having low space cost and fast lookup speeds, it is the main competition for our data structure. Our data structure needs less space for a lookup, while VisionEmbedder's lookup is faster.

Zusammenfassung

In der Informatik gibt es verschiedene Arten von Datenstrukturen, die dafür verantwortlich sind, zusammenhängende Informationen zu speichern, zum Beispiel die Wörterbuchdatenstruktur oder die Retrieval-Datenstruktur. Wörterbuchdatenstrukturen speichern die Schlüssel-Wert-Paare, während Retrieval-Datenstrukturen nur die Schlüsselwerte speichern und somit weniger Platz brauchen. Meistens sind Retrieval-Datenstrukturen statisch, also haben diese nur eine Konstruktionsoperation und Schlüsselwertabfragen (auch Lookup genannt).

Mögliche Anwendungen für Retrieval-Datenstrukturen sind Filter, wie Bloom-Filter oder XOR-Filter und perfekte Hashfunktionen. Retrieval-Datenstrukturen sind ein wachsendes Forschungsgebiet. In dieser Abschlussarbeit werden dynamische Retrieval-Datenstrukturen entwickelt, die den Platz für eine Schlüsselwertabfrage vom restlichen Platzbedarf trennen. Die Hauptziele sind es, den Platzbedarf kompakt zu halten und Einfüge-, Lösch- und Aktualisierungsoperationen mit linearer Laufzeit zu entwickeln. VisionEmbedder ist eine schon existierende Retrieval-Datenstruktur, die den Platzbedarf in zwei Teile trennt. Da VisionEmbedder ähnliche Ziele hat, von geringen Platzbedarfskosten und schnellen Lookup-Operationen, ist es der größte Wettbewerb für unsere Datenstruktur. Unsere Datenstruktur braucht weniger Platz, wohingegen VisionEmbedder eine schnellere Lookup-Operation hat.

Contents

Abstract i

Zusammenfassung iii

1 Introduction 1

2 Preliminaries 3
2.1 Retrieval Data Structures . . . 3
2.2 Sherman-Morrison Formula . . . 4

3 Related Works 5
3.1 Vision Embedder . . . 5
3.2 Bloom Filters and XOR Filters . . . 6

4 Data Structure 7
4.1 The Key in the Data Structure . . . 7

5 Algorithms 11
5.1 Lookup . . . 11
5.2 Update Key . . . 11
5.3 Insert Key . . . 12
5.4 Delete Key . . . 13

6 Implementation 17
6.1 Helper Functions . . . 17
6.1.1 Find Key Column . . . 17
6.1.2 Denominator Check . . . 18
6.1.3 Sherman-Morrison Formula . . . 18
6.1.4 Update Solution Rows . . . 19
6.2 Lookup . . . 19
6.3 Update . . . 19
6.4 Insert . . . 19
6.5 Delete . . . 20

7 Partitioning and Threshold Bumping 21
7.1 Algorithms . . . 21
7.2 Extract Keys from Dynamic Retrievals . . . 22

8 Evaluation **23**
8.1 Experiment Setup 23
8.2 Results 23
8.3 Vision Embedder 28

9 Conclusion **29**

Bibliography **31**

1 Introduction

The efficient storage and retrieval of key-value pairs $(x, f(x))$ has been a fundamental topic in computer science, with data structures ranging from maps and dictionaries to retrieval data structures. Static retrieval data structures store the values, typically requiring $(1+\epsilon)rn$, where $n \in \mathbb{N}$ is the number of keys, $r \in \mathbb{N}$ is the number of value bits, and $\epsilon > 0$ is the additional overhead. This thesis will create a dynamic retrieval data structure that supports insert, update, and delete operations, which requires extra steps to keep the data structure efficient and compact. We look at networking and telecommunications as inspiration for how to create the data structure efficiently.

In networking and telecommunications, separating the data plane and the control plane is a largely discussed topic, for example, when it comes to the development of 5G networks[5]. The data plane handles frequent operations, such as packet forwarding, so the separation would allow the data plane to have a simple and fast design to procure extreme efficiency. The control plane is responsible for the more complex operations, like routing updates. In the context of retrieval data structures, the data plane performs lookups, while the control plane manages the insert, update, and delete operations. As the main goal is efficiency, the data plane should be as compact as possible, and the insert, update, and delete algorithms should be linear.

Retrieval data structures are typically set up as linear systems, where the multiplication of a matrix with a key returns their corresponding values. This matrix forms the main part of the data plane. Constructing the matrix involves multiplying the inverse of a key matrix with the value matrix. To efficiently update the inverse we will use the Sherman-Morrison formula and to further improve the performance we will use partitioning and threshold bumping.

In Chapter 2 the preliminaries are discussed, including the Sherman-Morrison formula. Chapter 3 discussed related works such as VisionEmbedder and Bloom filters. Chapter 4 explores the relationship between the keys and the data structure, which is necessary for algorithm development. Chapters 5 and 6 detail the algorithms and their implementation. Chapter 7 introduces partitioning and threshold bumping to enhance the performance. Lastly, Chapter 8 evaluates the code through experiments and a comparison to VisionEmbedder.

2 Preliminaries

2.1 Retrieval Data Structures

In a universe $\mathcal{U}$ there is a set of keys $S \subseteq \mathcal{U}$ with $|S| = n$. In addition, a function $f : S \to \{0,1\}^r$ is given, which provides each key $x \in S$ with a corresponding value $f(x)$, where $r \in \mathbb{N}$ is the value size. Combining the key and its value creates a key-value pair $(x, f(x))$.

Retrieval data structures store the necessary information from key-value pairs so that a $lookup(x \in \mathcal{U})$ returns:

- $f(x)$ if $x \in S$.
- An arbitrary value $a \in \{0,1\}^r$ if $x \notin S$.

If the data structure is dynamic the following modifying operations must be possible:

- $insert(x, a \in \{0,1\}^r)$: A key $x \notin S$ is inserted into the data structure, which inserts the key into the set $S' = S \bigcup \{x\}$ with the value a.
- $update(x, a \in \{0,1\}^r)$: A key's $x \in S$ value updates to be a.
- $delete(x)$: A key $x \in S$ is deleted, which deletes the key from the set $S' = S \backslash \{x\}$.

A possible strategy to implement this, is to store variables in a *solution matrix* $Z \in \{0,1\}^{m \times r}$, where the j'th row of the matrix is a *solution row* $Z_j \in \{0,1\}^r$ and m is the capacity. Using a hash function $k : S \to \{0,1\}^m$ on the key, the solution matrix is chosen so that multiplying it with the solution matrix gives us the value $f(x)$:

$$h : f(x) = k(x) \cdot Z$$

These equations h are called *key equations* and can be used for the lookup operation. To better understand these equations regard the following example: There are two keys x_1 and x_2 and their hash values are $k(x_1) = 11001000$ and $k(x_2) = 00011010$. The first key x_1 has ones at the indexes 1, 2 and 5, meaning that when the hashed key $k(x)$ is multiplied with the matrix Z only the solution rows Z_1, Z_2, and Z_5 are multiplied with 1. This means that the multiplication consists of each column adding the values located in the solution rows 1, 2, and 5. We are working in $\mathbb{F}_2$ therefore the key equation can also be shown as the addition of the solution rows $h_1 : f(x_1) = Z_1 \oplus Z_2 \oplus Z_5$. Following the same logic, the second key x_2 creates the following simplified key equation $h_2 : f(x_2) = Z_4 \oplus Z_5 \oplus Z_7$. The solution matrix Z needs to solve both key equations simultaneously. Since Z_1, Z_2, Z_4, and Z_7 only appear once in their key-equations, per each key-equation, one solution row is selected

to be assigned the key's value $f(x)$. In this case, we will assign the values to $Z_2 = f(x_1)$ and $Z_7 = f(x_2)$, while the rest of the solution rows need values that do not contradict the equations. The most obvious value to choose for all the solution rows is $0 = Z_1 = Z_5 = Z_4$.

From $|S| = n$, it follows that n key equations are acquired from the n keys. Solving the key equations is equivalent to solving a system of equations. Hence, we fill a *key matrix* $K \in \{0,1\}^{m \times m}$ with the hashed keys $k(x)$ and the value matrix $F \in \{0,1\}^{m \times r}$ with the associated values. Meaning that if the hashed key $k(x)$ is in the i'th row of K, then the value $f(x)$ is also in the i'th row of F. These matrices create a more generalized form of the key equations:

$$F = K \cdot Z \;\Leftrightarrow\; Z = K^{-1} \cdot F$$

Using the inverse of the key matrix K^{-1} to determine the matrix Z, requires the key matrix K to be invertible. Therefore, the matrix has to be square and more importantly the key matrix needs to have a full rank. In other words, a hashed key can only be added to the key matrix K if it is not linearly dependent on the existing hashed keys. In addition, any rows that do not have a key cannot have a zero vector, so they default to the row's unit vector. From now on, when speaking about the key it could be either x or the hashed version $k(x)$. For example, "inserting a key $k(x)$" means inserting the hashed version of the key.

Inserting keys alters the key matrix K and computing a new inverse each time is extremely time-consuming. Therefore, a shortcut would come in handy and this would be where the Sherman-Morrison formula comes in.

2.2 Sherman-Morrison Formula

The Sherman-Morrison formula[7] explores the connection between the inverse A^{-1} before and after the matrix $A \in \mathbb{R}^{m \times m}$ is altered. The alteration consists of the addition of the product of a column vector $u \in \{0,1\}^m$ and a row vector $v^T \in \{0,1\}^m$ to the matrix A: $A' = (A + uv^T)$.

$$A'^{-1} = A^{-1} - \frac{A^{-1}uv^TA^{-1}}{1 + v^TA^{-1}u}$$

Altering a matrix can lead to it not being invertible because certain rows or columns could develop a linear dependency with each other. In addition, the formula can not always be solved, because we cannot divide by 0. A' is not invertible if the Sherman-Morrison formula can not be solved, due to the denominator being $0 = 1 + v^TA^{-1}u$.

The Sherman-Morrison formula is a specialization of the Woodbury formula, where u and v^T are not vectors but matrices with $U \in \mathbb{R}^{m \times a}$ and $V \in \mathbb{R}^{a \times m}$, where $a \in \mathbb{N}$ is the rank.

$$(A - UV)^{-1} = A^{-1} + A^{-1}U(\mathbb{1} - VA^{-1}U)^{-1}VA^{-1}$$

There are two instances where the Sherman-Morrison formula can be used in our algorithms. These instances include when a new key is inserted to or removed from the key matrix K.

3 Related Works

Perfect hash functions and retrieval data structures are closely linked. A minimal perfect hash function can be used to create a retrieval data structure[1] with lookups in constant time but with an overhead of $n \log \epsilon$. Also in combination with a cuckoo hash table a retrieval data structure can be used to make a perfect hash function[10].

A lot of retrieval data structures are built from linear equations. In the paper from Genuzio, Ottoviano and Vigna[6], they try to solve the linear equation through the use of hypergraphs and peeling. Their technique was based on a combination of new results and techniques from other papers. The data structure ended up with an overhead of $1.23rn$, where r is the value bit amount and n is the number of keys. A paper by Martin Dietzfelbinger and Stefan Walzer [3] reduces construction time and in theory achieves a space of $nr + O(\log n)r$. Belazzougui and Venturini[1] discuss the idea of compressing static functions resulting for an overhead of only $o(n)$ when $\log(1/\epsilon) = o(\log n / \log \log n)$.

3.1 Vision Embedder

VisionEmbedder is a dynamic retrieval data structure that our implementation will be compared to later on. Just like our data structure Vision Embedder[11] consists of two parts: the data plane which is the value table and an assistant table, which is used to update the value table.

The value table is made up of three arrays A_j, with $j \in \{1, 2, 3\}$, that each have their own unique hash function h_j. A lookup operation retrieves a value from each array and adds them together to calculate the keys corresponding value:

$$A_1[h_1(k_i)] \oplus A_2[h_1(k_i)] \oplus A_3[h_1(k_i)] = v$$

The calculation consists of a constant amount of XOR operations, therefore a lookup has a constant runtime.

To calculate the value table the assistant table is used. It uses a counter $C_j[t]$ to record the number of keys mapped to $A_j[t]$ and it uses buckets $S_j[t] = \{\langle k_i, v_i \rangle | h_j(k_i) = t.\}$ to keep track of the set of keys mapped to each integer.

In theory their proposed update algorithm compares the cost of update $A_j[t]$, with $t \in \mathbb{N}$, for the key's three integers. Then, the integer with the lowest cost is chosen and updated. After that, the other affected key-value pairs are updated too, while making sure that the

previously updated integer is not modified again. This algorithm is not perfected, since their analysis determined the probability for an endless loop after n insertions and t updates to be $O(t/n^2)$. Their solution is to reconstruct the data structure if this occurs.

Their delete operation does not alter the value table, it only deletes any trace of the key-value pair from the assistant table.

3.2 Bloom Filters and XOR Filters

Retrieval data structures can be used to answer *isMember* operations through filters. Static retrieval data structures are used in XOR Filters[4], which has a construction operation and a $isMember(x \in \mathcal{U})$ query with a false positive rate of 2^{-r}. Dynamic retrieval data structures are used in Bloom filters[2] which has 2 main operations: insert and a $isMember(x \in \mathcal{U})$ query. Before our dynamic retrieval data structures can be used for Bloom filters deleting a key will need an extra step, to minimize the possibility of a false positive. Bloom filters can be used in web caching, address lookup, network measurement and more[8].

The implementation for the *isMember* query is identical for both filters. A fingerprint $f_b : \mathcal{U} \to \{0,1\}^r$ and a retrieval data structure D_R are compared to each other to determine what the *isMember* function returns:

$$isMember(x \in \mathcal{U}) = \begin{cases} 1 & , f_b(x) = lookup(D_R, x) \\ 0 & , f_b(x) \neq lookup(D_R, x) \end{cases}$$

4 Data Structure

Building the data structure is quite simple. Given is a set of keys $S = \{x_1, x_2, x_3, x_4\}$, a value matrix $F \in \{0,1\}^{m \times r}$ and the key matrix $K \in \{0,1\}^{m \times m}$, which start off as a unit matrices. One after the other, each key $k(x)$ is added to the key matrix K replacing a unit vector. When choosing the row the key $k(x)$ is added to we need to assure that the matrix K can still be inverted. This might cause gaps between inserted keys. Whichever row of the key matrix the key $k(x)$ is inserted into, is the same row of the value matrix F that the corresponding value $f(x)$ is inserted into.

$$K = \begin{pmatrix} k(x_1) \\ k(x_2) \\ e_3 \\ k(x_3) \\ k(x_4) \\ e_6 \\ \dots \\ e_m \end{pmatrix}, \quad F = \begin{pmatrix} f(x_1) \\ f(x_2) \\ 0 \\ f(x_3) \\ f(x_4) \\ 0 \\ \dots \\ 0 \end{pmatrix}$$

Now, the inverse of the key matrix K^{-1} can be calculated and be used in the generalized key equation $Z = K^{-1} \cdot F$ to determine the solution matrix.

$$Z = \begin{pmatrix} Z_1 \\ Z_2 \\ Z_3 \\ Z_4 \\ Z_5 \\ Z_6 \\ \dots \\ Z_m \end{pmatrix}$$

The next section explores the connection between the key and the data structure, which will further explain the connection of the solution rows Z_j and the inverse key matrix.

4.1 The Key in the Data Structure

Understanding how the key interacts with the data structure is imperative to understand the algorithms later on. Regard this example of a key matrix K and its inverse K^{-1}:

$$K = \begin{pmatrix} 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \text{ and } K^{-1} = \begin{pmatrix} 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

The first four rows of the key matrix K contain keys, while the last row is a unit vector. Each row i in the key matrix is connected to the $i'th$ column of the inverse matrix, also known as the corresponding key's *key column*. In regard to the second key $k(x_2) = 01100$, which is located in the second row of the key matrix K, the key column is the second column in K^{-1}. It only has a 1 in the first and second row, therefore the key equation h_2 is only involved with these two rows. To understand what being involved in a row means, we look at the second row of the inverse K^{-1}. The first three key columns have a 1, meaning that the combination of the first three key equations create an equation for the second row.

$$\begin{aligned} h_1 \oplus h_2 \oplus h_3 : f(x_1) \oplus f(x_2) \oplus f(x_3) &= (k(x_1) \oplus k(x_2) \oplus k(x_3)) \times Z \\ &= (11001 \oplus 01100 \oplus 11101) \times Z \\ &= 01000 \times Z \\ &= Z_2 \end{aligned}$$

As the example shows us, the keys added together always equal the row's unit vector. This is due to the fact that $K \times K^{-1} = \mathbb{1}$. To determine what the solution rows are made up of we need to add the values of the corresponding keys.

$$\begin{aligned} Z_1 &= f(x_2) \oplus f(x_3) \oplus Z_5 \\ Z_2 &= f(x_1) \oplus f(x_2) \oplus f(x_3) \\ Z_3 &= f(x_1) \oplus f(x_3) \\ Z_4 &= f(x_1) \oplus f(x_4) \end{aligned}$$

Z_5 is undefined and can be assigned any value, like 0. A lookup for a key would give the following result:

$$\begin{aligned} \text{lookup}(x_2) &= k(x_2) \times Z \\ &= 01100 \times Z \\ &= Z_2 \oplus Z_3 \\ &= f(x_1) \oplus f(x_2) \oplus f(x_3) \oplus f(x_1) \oplus f(x_3) \\ &= f(x_2) \end{aligned}$$

Next, multiplying a key $k(x)$ with the inverse key matrix K^{-1} gives us a unit vector e_i. The index i of the unit vector e_i is the index for the key's key column, i.e., the i'th column in K^{-1} is the key column. This is possible due to the condition that $K \times K^{-1} = \mathbb{1}$ and $e_i \times K = k(x)$.

$$
\begin{aligned}
& K \times K^{-1} = \mathbb{1} \\
\Leftrightarrow \quad & e_i \times K \times K^{-1} = e_i \times \mathbb{1} \\
\Leftrightarrow \quad & k(x) \times K^{-1} = e_i
\end{aligned}
$$

Key columns contain a lot of useful information about how the key equations interact with each other and with the solution matrix. The usefulness of key columns and therefore the inverse key matrix is important for the next chapter.

5 Algorithms

Dynamic retrieval data structures need to be able to insert a key, update a key and delete a key. Before developing the algorithms, we need to understand what information the data structure needs to store. For the lookup of a key the solution matrix Z is needed. As we explained the importance of the key columns in section 4.1, we also want to store the inverse matrix. It is not necessary to save the key matrix since it can be calculated from the inverse. Lastly, to prevent a key from being added to a row where a key already is, we use a key position vector $p \in \{0,1\}^m$ to mark occupied rows.

5.1 Lookup

A lookup receives a key $k(x)$ and returns the corresponding value $f(x)$. To find the value the key equation $h : f(x) = k(x) \times Z$ can be used. Multiplying the key with the solution matrix has r steps, one for every column of the solution matrix. Therefore, the runtime is in $O(r)$, which could be constant if $r = 1$.

Algorithm 1: Lookup

```
1 Function Lookup(k(x))                                    // key
2     product = k(x) × Z
3     return product
```

5.2 Update Key

The update function receives a key $k(x)$ and an update value $new \in \{0,1\}^r$. Currently the lookup function for the key $k(x)$ returns the old value $old \in \{0,1\}^r$. The goal of the update function is to alter the data structure so that the lookup function returns new for $k(x)$.

Considering that the only change to the key equation is that the value is now new instead of old, the next step is to determine where the key's value pops up in the data structure. Neither the key position array p nor the inverse key matrix K^{-1} store the values, that leaves the solution matrix Z as the element that needs to be altered. As was shown in 4.1 each solution row Z_j is made up of various values $f(x)$. Which values they are made up of can be deduced from the inverse K^{-1}. Therefore, the update algorithm needs to use the key column to determine the solution rows that need updating.

Algorithm 2: Update Key

```
1 Function updateKey(k(x), new)                                  // key, value
2     e_i ← k(x) × K^-1
3     i ← index of e_i
4     valueDifference ← lookup(k(x)) ⊕ new
5     for j ← 1 to m do
6         if K^-1_{j,i} = 1 then                  // m is the number of rows in K^-1
7             Z_j ← Z_j⊕ valueDifference
```

First, the algorithm has to calculate the old value of the key old. Second, the algorithm determines the key column i. Third, for every row $j \in \mathbb{N}$ of the inverse key matrix K^{-1}, where the key column i has a 1 in it the corresponding solution row Z_j needs to be updated. Section 4.1 showed that having a 1 in the key column means that the key's value $f(x)$ appears in the solution row Z_j. Since the current dataset is still working with the old value, every solution row looks like $Z_j = old \oplus a$, where $a \in \{0,1\}^r$ is a combination of other solution rows and values from other keys. To update a solution row the difference of the old and new key value ($old \oplus new$) is added onto it:

$$\begin{aligned} & Z_j = old \oplus a \\ \Leftrightarrow\ & Z_j = old \oplus a \oplus (old \oplus new) \\ \Leftrightarrow\ & Z_j = new \oplus a \end{aligned}$$

The update algorithm has a linear runtime based on the capacity: $O(m)$. This is because finding the key column is in $O(m)$ and the m columns in K^{-1} are traversed.

5.3 Insert Key

The insert function inserts a new key $k(x)$ and its value $f(x)$ into the data structure. The intended result is that a lookup for the key $k(x)$ returns $f(x)$, while still working properly for the other keys in the data structure. To insert the key $k(x)$ into the data structure the key also needs to be inserted into the key matrix. Since the data structure only stores the inverse key matrix the Sharon-Morrison formula can be used to alter the inverse, thereby also altering the hypothetical key matrix.

Begin with the inverse K^{-1} and a key $k(x)$ which is to be inserted into the key matrix. This means that the key $k(x)$ needs to be inserted into a row i, which doesn't contain a key. It should be noted, that these rows are not empty as any rows without a key default to their unit vector. Therefore, in one move the key $k(x)$ has to be inserted and the unit vector e_i has to be deleted from the row i. To simplify this we create the row vector $v^T \in \{0,1\}^m$ which needs to be inserted into the key matrix K.

$$v^T = k(x) - e_i = k(x) + e_i$$

Subtracting e_i is the same as adding e_i, because the vector is in the space $\mathbb{F}_2$. The unit vector e_i also gives us the column vector $u = e_i$ for the Sherman-Morrison formula. Determining which row of the key matrix K the key $k(x)$ can be added to can be done with the help of the denominator in the Sherman-Morrison Formula. If a key is currently not occupying the i'th row in K and the denominator $1 + v^T K^{-1} e_i$ of the Sherman-Morrison formula is unequal to 0, then the key $k(x)$ can be added. Since the space is in $\mathbb{F}_2$, the denominator check can be simplified to the following:

$$\begin{aligned} & 0 \neq 1 + v^T K^{-1} e_i \\ \Leftrightarrow \quad & 1 = 1 + v^T K^{-1} e_i \\ \Leftrightarrow \quad & 0 = v^T K^{-1} e_i \end{aligned}$$

This also means that if the key can be added, then the denominator equals 1. Consequently, the Sherman-Morrison formula can be simplified:

$$K'^{-1} = (K + e_i v^T)^{-1} = K^{-1} - \frac{K^{-1} e_i v^T K^{-1}}{1 + v^T A^{-1} e_i} = K^{-1} - K^{-1} e_i v^T K^{-1}$$

This simplified version is also applicable when deleting a key from the key matrix.

Next, since the i'th row of the key matrix now has a key, this needs to be indicated in the key positions vector by changing the entry at the i'th index to a 1. Lastly, the solution matrix Z needs to be updated. Currently, a lookup for the key returns a random variable a but it should actually return $f(x)$. This task is identical to updating a key to have the value $f(x)$.

Algorithm 3: Insert Key

```
1 Function insertKey(k(x), f(x))                          // key, value
2     for i ← 1 to m do                  // m is the number of columns in K^-1
3         if p_i = 0 then                          // Checks if row i has a key
4             if 1 = v^T K^-1 e_i then                          // v^T = k(x) + e_i
5                 K'^-1 = K^-1 + K^-1 e_i v^T K^-1
6                 p_i ← 1
7                 updateKey(k(x), f(x))
8                 return true
9     return false
```

Due to the use of the Sherman-Morrison formula and update key having a linear runtime, the insert algorithm has a linear runtime $O(m)$, where m is the capacity.

5.4 Delete Key

The delete function deletes a key $k(x)$ from the data structure. More specifically, by deleting the key from the key matrix K. In addition, since the key is no longer in the key matrix, the

index in the key positions vector p needs to be set to 0. Due to the ambiguity of lookup for keys not in the data structure, the solution matrix does not have to be altered.

Just like in section 5.3 the Sherman-Morrison formula is used to delete a key from the key matrix. Begin with an inverse key matrix K^{-1} and the key $k(x)$ which is to be deleted from the key matrix. An element needs to be iterated through to find a case where the denominator check $0 = v^T K^{-1} e_i$ passes. The row i in the key matrix K is fixed, but the vector that replaces the row is not fixed. Therefore, instead of using e_i in the row vector, the row vector $v^T = k(x) + e_j$ iterates through various possible unit vectors, where $j \in [1...m]$. Once a unit vector e_j is found where the denominator check passes, then the row vector v^T, and the column vector e_i are used in the simplified Sherman-Morrison formula to compute the new inverse matrix $K'^{-1} = K^{-1} - K^{-1} e_i v^T K^{-1}$. K' is not an acceptable key matrix yet though, because if $i \neq j$ then the i'th row of the key matrix has e_j in it's row instead of the intended e_i. Before we discuss how to convert the key matrix back into an acceptable state a condition needs to be proven.

Condition: The denominator check can only pass for a row j that has a key in it. This is because if the row j has a unit vector e_j and the Sherman-Morrison formula tried to add the unit vector to e_j too, then two rows i and j would have e_j in it's row, since $i \neq j$. Therefore the two rows would be linearly dependent on one another and an inverse wouldn't exist. Since the Sherman-Morrison formula is explicitly used to calculate the inverse the inverse shouldn't be calculable, therefore the denominator check shouldn't have passed.

Using the condition, that row j needs to have a key, the key matrix can be returned to an acceptable state by switching the rows i and j in the key matrix K. For example, if a key was removed from the 3rd row of a key matrix while adding the unit vector e_1, then the switch will create an acceptable state for the key matrix.

$$K' = \begin{pmatrix} k(x_1) \\ k(x_2) \\ e_1 \\ k(x_4) \\ e_5 \end{pmatrix} \Rightarrow K'' = \begin{pmatrix} e_1 \\ k(x_2) \\ k(x_1) \\ k(x_4) \\ e_5 \end{pmatrix}$$

In section 4.1 the connection between the key in K and the inverse matrix K^{-1} showed that the rows in K and the columns in K^{-1} correspond to each other. Therefore, switching the rows i and j in the key matrix is equivalent to switching the columns i and j of the inverse. Now, the key matrix is in an acceptable state and any traces of the key $k(x)$ have been removed, that part of the algorithm is complete. Lastly, the deletion of the key needs to be marked in the key position vector p by setting the j'th position to $p_j = 0$.

Algorithm 4: Delete Key

```
1 Function deleteKey(k(x))                                   // key
2     e_i ← k(x) × K^-1
3     i ← index of e_i
4     j ← 1
5     while 0 = v^T K^-1 e_i do                              // v^T = k(x) + e_j
6         j++
7     K'^-1 = K^-1 + K^-1 e_i v^T K^-1
8     Swap columns i and j in K^-1
9     j index removed from p
```

First, the key column i of the key $k(x)$ is determined, this is also the key's row in the key matrix. Next, the denominator check will find a unit vector e_j, where the check passes. Then, the Sherman-Morrison formula is used to alter the inverse key matrix. After that, the columns i and j in the inverse key matrix K^{-1} are switched, if $i \neq j$. Lastly, the index j is removed from the key positions vector p, since that is where the key's position was technically switched to.

In general the delete algorithm has a linear runtime based on the capacity: $O(m)$. This is because finding the key column is in $O(m)$ and so is the Sherman-Morrison formula.

6 Implementation

As we discussed in the previous chapter the dynamic retrieval data structure has three variables. These are the solution matrix Z, the inverse key matrix K^{-1} and the key positions vector p. Beginning with the inverse key matrix K^{-1}, as we already discussed the key matrix is a square matrix, so the inverse has to be square too. In addition, the rows of the key matrix are filled with the keys $k(x)$, meaning that the number of columns has to equal the number of bits in $k(x)$. For this implementation a key is a 64 bit integer `uint64_t` and the inverse key matrix is an array of 64 keys. Therefore, the inverse key matrix has 64 columns and 64 rows. As the Sherman-Morrison formula works with columns of the inverse key matrix and update works with key columns, it is wise to save the inverse matrix column-wise.

Next, the solution matrix Z has the same number of rows as the inverse key matrix, but the number of columns equals to the value type size r. The strongest criteria for deciding if the solution matrix should be stored column-wise or row-wise is the lookup function, because the lookup function should be as efficient as possible. During the lookup the key is multiplied with the solution matrix columns. An iteration through the matrix is needed, so whichever type size is smaller would be the more efficient option. Typically, the value types are small for example 1 bit or 8 bits, so a column-wise implementation would be better. This means that the solution matrix Z will be implemented as an array made out of r keys. If values only have one bit, then the runtime is constant.

Lastly, the key positions vector p, will be implemented as a key `uint64_t`, where each bit marks if that corresponding row in the key matrix has a key.

To prevent the need to code the same behavior twice helper functions are needed. These helper functions are: find key column, the denominator check for the Sherman-Morrison formula, and the Sherman-Morrison formula. In addition, the implementation of updating solution rows is separated into an extra function because both update key and insert key update the solution matrix.

6.1 Helper Functions

6.1.1 Find Key Column

Both the update and delete key algorithm require the keys respective column in the inverse. In section 4.1 the formula $k(x) \times K^{-1} = e_i$ was introduced to convey the connection between the key and the inverse key matrix. From the formula the steps to find the key column can

be deduced. First, multiply the key with the columns of the inverse matrix. Then, check if the product only has one 1. If it does the index for the 1 is returned, because this is also the index for the key column. If there is not exactly one 1, the key is not in the data structure and the function returns −1.

6.1.2 Denominator Check

The three general input variables are the key $k(x)$, the index i for e_i and the index j for e_j. As a reminder the denominator check is $0 = v^T K^{-1} e_i$, where $v^T = k(x) \oplus e_j$ for delete and $v^T = k(x) \oplus e_i$ for insert. To be able to implement both at the same time insert can have j default to i, so that $e_j = e_i$.

First, the row vector v^T is calculated by adding the key $k(x)$ and a 1 at the j'th index. $K^{-1}e_i$ is equal to the i'th column of the inverse matrix. Multiply these two variables together and return true if the product is 0: `0 == parity((key ⊕ e_i) & inverseKeyMatrix[i])`. Adding and multiplying the variables can all be done in constant runtime

6.1.3 Sherman-Morrison Formula

Just like with the denominator check the insert and delete algorithm both use the Sherman-Morrison formula. In addition, the input variables are exactly the same: key $k(x)$, the index i for e_i and the index j for e_j, where $e_j = e_i$ for insert. The simplified Sherman-Morrison formula is used to calculate the new inverse key matrix $K'^{-1} = K^{-1} + K^{-1}e_i v^T K^{-1}$.

Each bit of the column vector $v^T K^{-1}$ needs to be calculated separately by multiplying the key with the m inverse matrix columns. Additionally, $K^{-1}e_i$ (the i'th inverse column) needs to be multiplied with $v^T K^{-1}$, this creates a matrix. Each column a of the matrix is either a 0 column or $K^{-1}e_i = u_i$ based on if the a'th bit of $v^T K^{-1}$ is a 1 or 0 bit.

$$
\begin{aligned}
v^T K^{-1} &= \begin{pmatrix} 1 & 0 & 1 & 1 & \dots & 0 \end{pmatrix} \\
\Leftrightarrow u_i v^T K^{-1} &= \begin{pmatrix} u_i & 0 & u_i & u_i & \dots & 0 \end{pmatrix}
\end{aligned}
$$

This shows that calculating the matrix $K^{-1}e_i v^T K^{-1}$ and adding it to the original inverse key matrix can be done separately for each bit in $v^T K^{-1}$. When a bit for $v^T K^{-1}$ is calculated, then if the bit is 1 the $K^{-1}e_i$ can be added to the corresponding column of the inverse key matrix.

6.1.4 Update Solution Rows

The update solution rows function receives two input variables, the key column index and the value difference, which needs to be added to certain solution rows. In section 5.2 we explained that the ones in the key column are the columns that need the value difference added to it. Since values in the solution matrix are stored column-wise, the logic of how the value difference is added needs to be altered. Instead of adding the value difference, the key column is added for every 1 bit of the value difference. In total r operations are needed.

Now, these helper functions can be used to implement the general functions: lookup, update, insert and delete.

6.2 Lookup

A lookup for a key involves retrieving the corresponding value. To determine the value $f(x)$ that needs to be returned the key needs to be multiplied with the solution matrix $k(x) \times Z$. Each value bit is calculated separately by multiplying the key with a solution matrix column. The runtime of the lookup function is in $O(r)$. In particular the runtime is constant in $r = 1$

6.3 Update

First, the find key column function is used to determine the key column. Next, the value difference between the old value of the key and the new value is calculated, this requires the lookup function to be called to determine the old value. Lastly, the update solution rows function uses the value difference and the key column to update the solution matrix.

6.4 Insert

First, different values of i are iterated through until the denominator check passes for the given key. Once an iteration where the denominator check passes is found, the Sherman-Morrison formula is used to calculate the new inverse key matrix. Next, the newly calculated inverse key matrix is used to update the value for the key. Just like with the update key function the value difference of the new value and the old value is calculated and then the update solution rows function is called.

6.5 Delete

The delete function has 3 steps. First, finding the key column i. Next, different possibilities for the index j are iterated through until the denominator check passes. Once a denominator check passes for j, the Sherman-Morrison formula recalculates the inverse key matrix. Lastly, the columns i and j of the inverse key matrix K^{-1} are swapped.

7 Partitioning and Threshold Bumping

Partitioning is the idea of splitting up a data set into smaller data sets, thereby making them easier to manage and enhancing the performance. Each smaller data set can be called a bucket and consists of its own retrieval data structure. A hash function $l : \mathcal{U} \to \{1, ..., b\}$, where $b \in \mathbb{N}$ is the amount of buckets, is used to separate the key-value pairs to their corresponding bucket.

Threshold bumping allows for scalability and adaptability in regards to inserting and deleting keys. Instead of having one level of partitioning, multiple levels of partitioning are used to catch keys that are bumped. Thresholds are used to indicate if a key is in that level or was bumped to the next level. Each bucket has a retrieval data structure and a bucket threshold b_t. In addition, each partitioning level uses a hash function $t : \mathcal{U} \to \{1, ..., w\}$, where $w \in \mathbb{N}$ is the maximum threshold, to determine a key's threshold. The key's threshold is then compared to the bucket's threshold to determine if the key is in the level or in one underneath it, e.g. the key is in the level if $t(x) < b_t$. In case a key insertion fails for a retrieval, that buckets threshold is lowered. Lowering the retrieval threshold means that the key, where the insertion failed, could be bumped to the next level and all of the keys in the retrieval where $t(x) \geq b_t$ need to be bumped to the next level too.

7.1 Algorithms

Partitioning needs two hash functions; one to assign key-value pairs to their buckets and the other one to calculate the key's threshold. In addition, it needs it's own insert, update, delete and lookup functions to be able to pass on the commands to the corresponding retrieval data structure.

The implementation for the update, delete and lookup functions follows the same setup as before. First, the key's corresponding retrieval data structure is determined. Second, the key is hashed. Third, the key threshold is calculated and used to determine if the key is in that partitioning level or not. If it is in that level then the function (e.g. update key) is called on the retrieval data structure for the key. Otherwise, the function (e.g. update) is called on the next partitioning level.

The insert function uses the same foundation as the other functions but with an extra step. If the key is in the partitioning level and the insertion into the retrieval data structure fails, then bump is called. After that, the key is either bumped to the next level or inserted into the retrieval data structure. There is a possibility that the insertion will fail again, so bump is

repeatedly called until the key is either successfully inserted into the retrieval data structure or bumped to the next level.

The bump function retrieves the keys from the retrieval data structure, lowers the retrieval threshold, and bumps the keys that no longer belong in the current level. The most costly part of this function is retrieving the keys from the retrieval data structure, making the runtime quadratic.

7.2 Extract Keys from Dynamic Retrievals

If the threshold of a retrieval data structure is reduced, then the key-value pairs stored in the retrieval data structure need to be checked, because the keys that are no longer smaller than the retrieval threshold need to be bumped to the next level and deleted from the retrieval data structure. If a key needs to be bumped can only be determined by the key's threshold, so all the keys in the retrieval data structure need to be retrieved and the keys that need to bumped will be bumped.

The hashed keys $k(x)$ can be found in the rows of the key matrix $K \in \{0,1\}^{m \times m}$, which can be calculated by inverting the inverse key matrix K^{-1}. Since the matrix K^{-1} is stored column-wise and the keys are stored row-wise in K, the way it is stored needs to be flipped. Once K is calculated the rows i with keys are indicated by the key positions vector p, where $p_i = 1$. Lastly, the values are gathered and the key-value pairs are returned. Switching the matrix from column-wise to row-wise and calculating the inverse of K^{-1} make the runtime $O(m^2)$.

Since the key matrix stores the hashed version of the keys, if a key is bumped the hashed version of the key will be sent to the next level. This means that any key that is sent to the next level needs to be hashed first, regardless of whether they were ever inserted into that level's corresponding retrieval data structure or not. To reduce the likelihood of hash collisions in later partitioning levels, each partitioning level uses one hash function to hash the keys instead of one per bucket.

Repeatedly hashing a key for every partitioning level is not the most elegant way. The only alternative would require either using a reversible hash function or extra storage. Using a reversible hash function would defeat the purpose of hashing the key in the first place. The extra storage would amount to $m \times m \times b$ bits of space, which stores the keys for every retrieval data structure $b \in \mathbb{N}$. Although, the extra information stored would be able to speed up bumping from $O(m^2)$ to $O(m)$.

8 Evaluation

This chapter will focus on using experiments to evaluate the code. For example, we identify which configuration has the smallest data plane. In addition, the implementation will be compared to VisionEmbedder.

8.1 Experiment Setup

The CPU used is a AMD Ryzen 7 PRO 4750U with Radeon Graphics, it has 8 cores, 16 logical processors and a clock speed of 1700Mhz. The implementation is compiled using G++ 13.2.0 in Visual Studio Code with the optimizations from Vision Embedder "-o3", "Ofast" and "inline".

The experiments use test data obtained from VisionEmbedder containing $n = 100,000$ key-value pairs. Each pair has a 32 bit key and 8 bit value. Although since the system operates on a 64-bit system the keys are converted to 64-bit keys, these are then hashed using randomly generated hash seeds. The threshold maximum for the implementation is 8. We use an implementation with 2 partitioning levels and a hash map at the base. The amount of buckets c in the first partitioning level is used to calculate how many buckets are in the next level $c' = c \times a$, with $a \in (0, 1]$.

8.2 Results

For the first experiment the amount of buckets in the first level is set to $c = n/96$ and the next capacity is $c' = c \times 0.6$. With $c = n/96$ buckets the first level can store a maximum of 66624 key-value pairs, which is equal to about 66.6%. The n keys are added in 10% increments, which means 10 000 keys are added. Each increment counts the amount of insert fails and measures the insert throughput for the first partitioning level. An insert fail describes when the insertion of a key into a retrieval data structure fails for the first time.

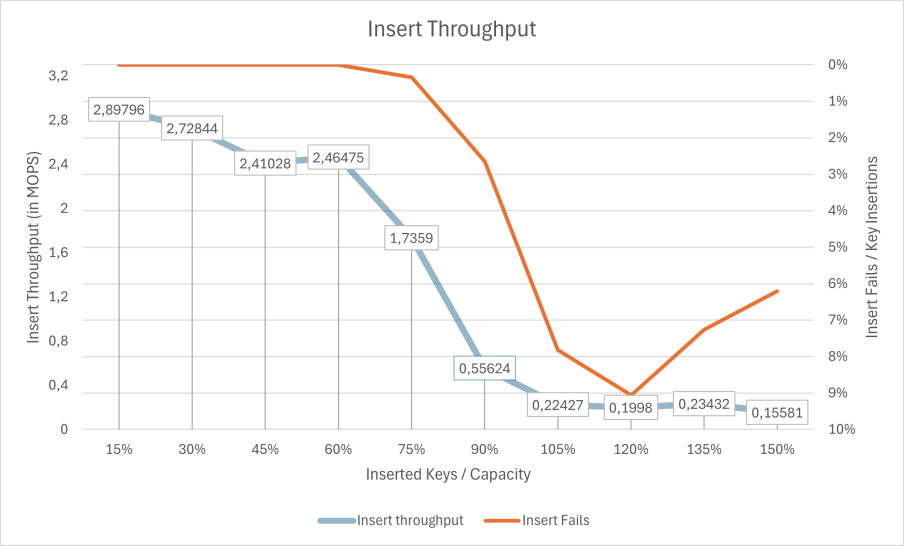


Figure 8.1: Insert Throughput

The horizontal axis shows the amount of inserted keys divided by the key-value pair capacity 66624. The left vertical axis displays the insert throughput in MOPS (million operations per second), which the blue line uses. The right axis shows the insert fail divided by the total amount of insertions in that increment, which the orange line uses.

Claim 1: The throughput is mostly affected by bumping.
Until about 60% of the capacity is reached the throughput ranges between 3 and 2.5 MOPS. No insertions have failed and therefore bumping hasn't been called yet, which means that all of the keys are being inserted into the retrieval data structures. As the dynamic retrieval data structures fill up the throughput lessens a little, most likely due to the fact, that the retrieval data structures take longer to find an empty row to insert the key in. Then, once 70% of the capacity has been reached in the first level the first insertions start to fail and bumping is called for less than 1% of the insertions. Still this causes the throughput to sink to 1.7 MOPS. After that the amount of insertions that fails rises, causing the throughput to worsen even more. Towards the end the insertion fails start to lessen, but the throughput doesn't improve because the other levels are starting to fill up.

In the next experiment the values for the variables stay the same ($c = n/96$ and $c' = c \times 0.6$) but different information is gathered. Each increment measures the number of keys, as well as the threshold average for the first partitioning level.

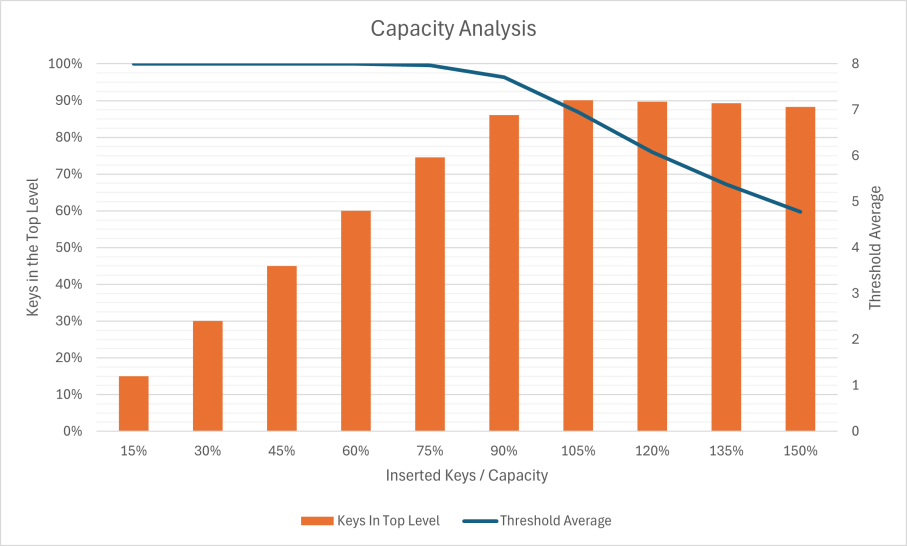


Figure 8.2: Capacity Analysis

Just like with the last graph, the horizontal axis shows the amount of inserted keys in the first level divided by the key-value pair capacity 66624. On the left vertical axis the amount of key-value pairs in the first level is shown using the orange bars. The blue line uses the right axis to show the threshold average in the first level.

Claim 2.1: The 90% of the first-level capacity is reached, after adding keys equivalent to 105% of that capacity. Based on the graph, that is also when the most key-value pairs are in the first level. As was previously stated, the first-level capacity is at 66.6%. This value could be helpful when choosing the next capacity multiplier $c' = c \times 0.6$.

Claim 2.2: The partitioning level is never at max capacity.
The maximum capacity that is reached is 90% and after that the keys lessen. A possible reason why the maximum capacity is never reached is because the implementation for threshold bumping is not optimal. In addition, during these experiments the same threshold maximum is always used, so the optimal threshold maximum can be explored in the future.

Towards the end of the graph in 8.2 it looks like the capacity is decreasing. The next experiment will measure the same things as the previous experiment but for less buckets $c = n/128$ and a next capacity of $c' = c \times 0.6$.

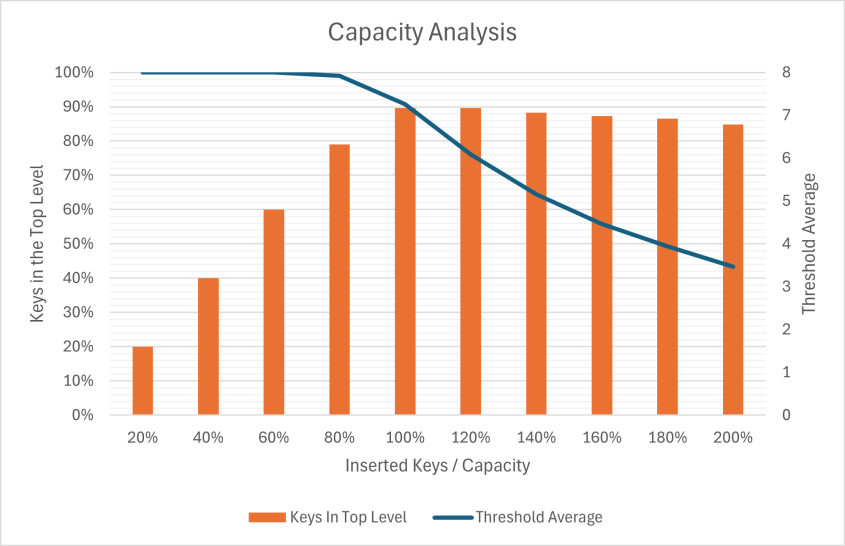


Figure 8.3: Capacity Analysis for a Smaller Capacity

This graph is just like the graph from 8.2 just for a smaller capacity of 49984 key-value pairs.

Claim 3: After adding key-value pairs equivalent to 120% of the first-level capacity, the amount of key-value pairs continues to lessen.
The average key threshold is steadily decreasing, and along with it the number of key-value pairs in the first level is also declining, though at a slower pace. When a threshold reaches 0, then all of the keys in that retrieval data structure are bumped. Eventually, the threshold average will reach 0 causing all the keys to bumped and leaving the current level empty.

The same values for the variables as the last experiment will be used in the next experiment $c = n/128$ and a next capacity of $c' = c \times 0.6$. During this experiment the average amount of keys bumped is calculated based on how often the bump function is called.

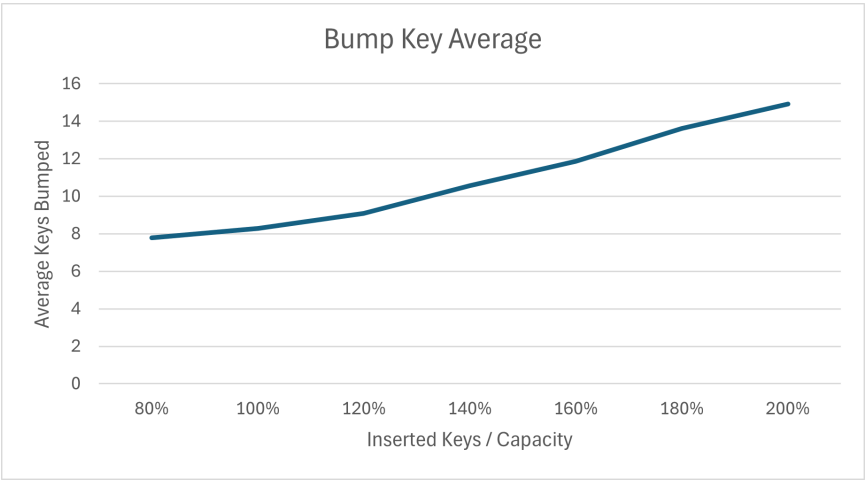


Figure 8.4: Average Keys Bumped

Just like the previous graphs the horizontal axis shows how many keys were added in relation to the first-level capacity. The vertical axis shows the average amount of keys being bumped per increment. The average is the number of keys bumped divided by how often bump is called.

Claim 4: The amount of keys that are bumped increases over time.
This happens due to how thresholds work. When the retrieval threshold is at it's maximum $b_t = 8$, then the key-value pairs stored in the retrieval data structure can have eight different possible thresholds $\{0, ..., 7\}$. Therefore, when keys are bumped about an eight of the keys are bumped, which is shown in the graph. When the threshold sinks the possible key thresholds lessen and more keys will have the same threshold. Therefore, when the threshold is reduced again more keys will be bumped. For example, if the threshold is reduced from 6 to 5, then about a sixth of the key-value pairs are bumped.

Lastly, one of the goals for our dynamic retrieval data structure is to have a compact data plane, therefore how certain variables influence the data plane needs to be studied. The dependent variables are the capacity $c = \{n/64, n/80, n/96, n/112, n/128\}$ and factor $a = \{0.3, ..., 0.4\}$ for the next capacity $c' = c \times a$.

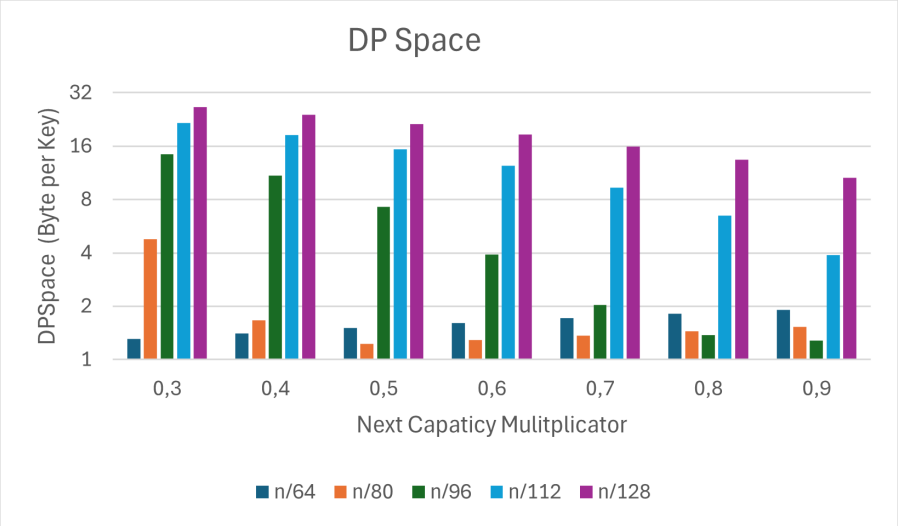


Figure 8.5: Data Plane Comparison

This graph shows the space the data plane needs in bytes per key, shown by the vertical axis. The horizontal axis shows the values for the next capacity factor a. Each next capacity factor has a group of various capacities c, as seen in the legend.

Claim 5.1: The implementation with the smallest data plane has a capacity of $c = n/80$ and a next capacity of $c' = c \times 0.5$. It needs 1.228 bytes per key.
1.228 bytes per key is very efficient, since a value has 8 bits (1 byte).

Claim 5.2: The higher the factor a is the fewer buckets the first level needs for optimal space efficiency.
If there are less buckets in the first level, the second level needs to compensate by having a

bigger capacity, otherwise most of the key-value pairs end up in the base level. For a factor $a \leq 0.4$ the capacity with the most optimal space is $c = n/64$. Next, for $a = \{0.5, 0.6, 0.7\}$ the best capacity is $c = n/80$. After that, $a = \{0.8, 0.9\}$ has the most optimal space for a capacity of $c = n/96$.

8.3 Vision Embedder

To be able to compare our implementation with VisionEmbedder we used their demo that is available on GitHub[9]. Unfortunately, their implementation does not include their update and delete algorithms. For our algorithm a bucket size of $c = n/80$ and a next capacity of $c' = c \times 0.6$ is chosen.

	Our Algorithm	VisionEmbedder
CP (Byte per key)	11.69	47.6
DP (Byte per key)	1.29	1.7
Insert (in MOPS)	0.61	1.27
Update (in MOPS)	5.47	NA
Delete (in MOPS)	14.66	NA
Lookup (in MOPS)	28.27	142.3

One of our main goals was to have a compact data plane. If we look at the table, we can see that our implementation not only has a more compact data plane but also a more compact control plane. However, their implementation has a faster lookup. This is understandable when it comes to lookup because they only need to add a constant amount of values together, while we need to add $r = 8$ values together. If the test data was using values with 1 bit, the lookup throughput might look different. While the overall insert throughput is faster for VisionEmbedder, in 8.1 we saw the throughput is most impacted by bumping and insertions without bumping are much faster than 1.27 MOPS.

9 Conclusion

The development of our dynamic retrieval data structure successfully achieved the wanted separation of the data plane from the control plane, as the lookup only needs the solution matrix. Our main goals were to design a compact data plane and to develop insert, update and delete algorithms with a linear runtime in $O(m)$, these were achieved.

In the evaluation the best results were observed for a capacity of $c = n/80$ and a next capacity of $c' = c \times 0.6$. Compared to VisionEmbedder, our implementation is more compact, though their lookup is faster. Future work could focus on optimizing the threshold function. Additionally, bumping could be improved. For instance, the extract keys function could be modified to return only the keys from the retrieval data structure and bumping would use lookup to get the values for the bumped keys. Finally, the impact of having different number of levels is another area for future research.

Bibliography

[1] Djamal Belazzougui and Rossano Venturini. “Compressed Static Functions with Applications”. In: *Proceedings of the 2013 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pp. 229–240. DOI: 10.1137/1.9781611973105.17. URL: https://epubs.siam.org/doi/abs/10.1137/1.9781611973105.17.

[2] James Blustein and Amal El-Maazawi. “Bloom Filters | A Tutorial, Analysis, and Survey”. In: (Jan. 2002), pp. 1–13. URL: https://cdn.dal.ca/content/dam/dalhousie/pdf/faculty/computerscience/technical-reports/CS-2002-10.pdf.

[3] Martin Dietzfelbinger and Stefan Walzer. “Constant-Time Retrieval with O(log m) Extra Bits”. In: *36th International Symposium on Theoretical Aspects of Computer Science (STACS 2019)*. Ed. by Rolf Niedermeier and Christophe Paul. Vol. 126. Leibniz International Proceedings in Informatics (LIPIcs). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019, 24:1–24:16. ISBN: 978-3-95977-100-9. DOI: 10.4230/LIPIcs.STACS.2019.24. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.STACS.2019.24.

[4] Peter C. Dillinger and Stefan Walzer. *Ribbon filter: practically smaller than Bloom and Xor*. 2021. URL: https://arxiv.org/abs/2103.02515.

[5] Truong-Xuan Do and Younghan Kim. “Control and data plane separation architecture for supporting multicast listeners over distributed mobility management”. In: *ICT Express* 3.2 (2017). Special Issue on Patents, Standardization and Open Problems in ICT Practices, pp. 90–95. ISSN: 2405-9595. DOI: https://doi.org/10.1016/j.icte.2017.06.001. URL: https://www.sciencedirect.com/science/article/pii/S2405959517300577.

[6] Marco Genuzio, Giuseppe Ottaviano, and Sebastiano Vigna. *Fast Scalable Construction of (Minimal Perfect Hash) Functions*. 2016. arXiv: 1603.04330 [cs.DS]. URL: https://arxiv.org/abs/1603.04330.

[7] William W. Hager. “Updating the Inverse of a Matrix”. In: *SIAM Review* 31.2 (1989), pp. 221–239. ISSN: 00361445, 10957200. URL: http://www.jstor.org/stable/2030425 (visited on 11/25/2024).

[8] Yi Lu, Balaji Prabhakar, and Flavio Bonomi. “Bloom filters: Design innovations and novel applications”. In: 2005. URL: https://api.semanticscholar.org/CorpusID:8364104.

[9] *VisionEmbedder*. https://github.com/VisonEmbedder/VisionEmbedder. 2024.

[10] Stefan Walzer. "Peeling close to the orientability threshold: spatial coupling in hashing-based data structures". In: *Proceedings of the Thirty-Second Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA '21. Virtual Event, Virginia: Society for Industrial and Applied Mathematics, 2021, pp. 2194–2211. ISBN: 9781611976465.

[11] Yuhan Wu et al. "VisionEmbedder: Bit-Level-Compact Key-Value Storage with Constant Lookup, Rapid Updates, and Rare Failure". In: *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. 2024, pp. 42248–4261. DOI: 10.1109/ICDE60146.2024.00324. URL: https://yangtonghome.github.io/uploads/VisionEmbedder_final.pdf.