MapEmbed: Perfect Hashing with High Load Factor and Fast Update

Yuhan Wu*
Peking University

Zirui Liu*
Peking University

Xiang Yu*
Peking University

Jie Gui*
Peking University

Haochen Gan†
Peking University

Yuhao Han‡
National University of Defense Technology

Tao Li‡
National University of Defense Technology

Ori Rottenstreich§
Technion

Tong Yang*¶
Peking University

ABSTRACT

[1] Perfect hashing is a hash function that maps a set of distinct keys to a set of continuous integers without collision. However, most existing perfect hash schemes are static, which means that they cannot support incremental updates, while most datasets in practice are dynamic. To address this issue, we propose a novel hashing scheme, namely **MapEmbed Hashing**. Inspired by divide-and-conquer and map-and-reduce, our key idea is named **map-and-embed** and includes two phases: 1) Map all keys into many small virtual tables; 2) Embed all small tables into a large table by circular move. Our experimental results show that under the same experimental setting, the state-of-the-art perfect hashing (dynamic perfect hashing) can achieve around 15% load factor, around 0.3 Mops update speed, while our MapEmbed achieves around 90% ∼ 95% load factor, and around 8.0 Mops update speed per thread. All codes of ours and other algorithms are open-sourced at GitHub.

CCS CONCEPTS

• **Information systems** → **Data structures**.

KEYWORDS

Perfect Hashing, Hash Tables, KV Stores

*Department of Computer Science and Technology, Peking University, China
†School of Mathematical Sciences, Peking University, China
‡School of Computer, National University of Defense Technology, Changsha, China
§The Department of Computer Science and the Department of Electrical Engineering, Technion – Israel Institute of Technology, Haifa, Israel
¶National Engineering Laboratory for Big Data Analysis Technology and Application, Peking University, China
[1]Yuhan Wu, Zirui Liu, and Xiang Yu contribute equally to this paper. Tong Yang (yangtongemail@gmail.com) is the corresponding author.

KDD '21, August 14–18, 2021, Virtual Event, Singapore.

ACM ISBN 978-1-4503-8332-5/21/08…$15.00
https://doi.org/10.1145/3447548.3467240

ACM Reference Format:
Yuhan Wu, Zirui Liu, Xiang Yu, Jie Gui, Haochen Gan, Yuhao Han, Tao Li, Ori Rottenstreich, and Tong Yang. 2021. MapEmbed: Perfect Hashing with High Load Factor and Fast Update. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '21), August 14–18, 2021, Virtual Event, Singapore.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3447548.3467240

1 INTRODUCTION

1.1 Background and Motivation

Hashing is one of the most fundamental techniques and has a wide range of applications in various fields, such as data mining, databases, machine learning, storage, networking [1–6]. Perfect hashing is a special hash function that maps a set of different keys to a set of continuous integers without collision. It often consists of two parts: a small index and a large Key-Value (KV) table. The index is often small enough to be held in fast memory (*e.g.*, CPU/GPU Caches, SRAM in FPGA/ASIC), while the KV table is typically larger and is stored in slow memory (*e.g.*, DDR [7], DRAM [8]). For each insertion/lookup, perfect hashing first updates/looks-up the index then the KV table. Perfect Hashing has two advantages: 1) its lookup time is always constant because hash collision is eliminated, and each lookup needs only one hash probe in slow memory; 2) its index is often very small (*e.g.*, 1.56 bits per key [9]), and can be ignored compared with the large KV table (*e.g.*, each KV pair needs hundreds of Bytes or more); Thanks to these two advantages, the perfect hash is widely used in many fields, such as databases [10], networks [11], and bioinformatics [12].

Most existing perfect hash schemes are static [13–17], which means that they cannot support incremental updates, while most datasets in practice are dynamic. In order to insert new keys, the perfect hash can only reconstruct the whole function from the beginning. Among all existing works, only the Dynamic Perfect Hash (DPH) [18] supports incremental updates, drastically extending the application range of perfect hashing. Its key idea is using the divide-and-conquer methodology: it first divides the datasets into many small ones; and for each small dataset, it builds a perfect hash table using existing hash schemes. Therefore, each incremental update is indeed to reconstruct a small perfect hash table. However, to control the reconstruction time, it needs to use many hash buckets

for each small perfect hash table. Therefore, its load factor[2] can only achieve 10% ∼ 20% (see Figure 11(a)), *i.e.*, it is memory inefficient. The design goal of this paper is to keep all advantages of existing works (constant lookup time, and small index structure) and, at the same time, to support fast incremental updates and achieve a high load factor.

1.2 Our Proposed Solution

Towards the design goal, we propose a novel perfect hashing scheme, namely **MapEmbed Hashing**. It has the following advantages: 1) it has no hash collision; 2) its lookup time is always constant; 3) its index is quite small (*e.g.*, 0.5 ∼ 1.6 bits per key); 4) it supports fast incremental updates (20× faster than Dynamic Perfect Hashing); 5) its load factor is high (90% ∼ 95%).

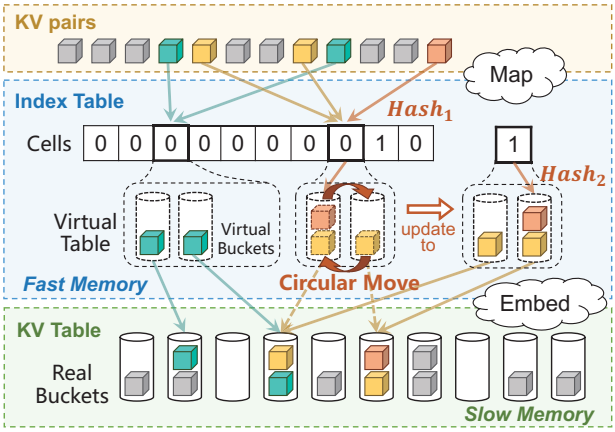


Figure 1: An example of MapEmbed Hashing.

Inspired by divide-and-conquer and map-and-reduce, our key idea is named **map-and-embed**, including two phases. **1) Map:** map all keys into many small virtual tables. **2) Embed:** merge and embed all virtual tables into a large table by circular moving.

We use an example to present the two phases. As shown in Figure 1, the data structure of MapEmbed Hashing consists of two parts: an *index table* in fast memory and a large *KV table* in slow memory. The index table is an array of index cells, each cell corresponds to a small virtual table consisting of virtual buckets, and each virtual bucket can store multiple KV pairs. First, we **map** each key into a **virtual bucket**[3] of an index cell using two hash functions: $\text{Hash}_1(\cdot)$ and $\text{Hash}_2(\cdot)$. Second, we **embed** each virtual table into the KV table, which consists of **real buckets**. Specifically, given a virtual table, each virtual bucket is associated with a real bucket in the KV table, and we push all KV pairs in each virtual bucket into its associated real bucket. If no bucket in the large table overflows, the embed phase succeeds. Otherwise, we circularly move the keys in the virtual table and try to embed them again. Specifically, for all keys in the virtual table, we will move them to the next adjacent virtual bucket. In other words, in one circular move, we will move the key in the i-th virtual bucket to the $(i+1)$-th virtual bucket and move the key in the last virtual bucket to the first virtual bucket.

[2]Load factor is the ratio of the number of inserted keys to the number of slots, where each slot can store one key.

[3]"Virtual bucket" is only a logical concept, not a bucket that actually stores KV pairs.

We will try embedding after one circular move – pushing all KV pairs in each virtual bucket into its associated real bucket. If the embedding succeeds, the insertion is complete. Otherwise, repeat the circular movement and embedding. The number of circular moves (the offset number of the keys in the same virtual table) is recorded in the corresponding index cell. The source codes of MapEmbed and other related algorithms are available at GitHub [4].

Main Experimental Results: 1) MapEmbed's index structure is quite small (*e.g.*, 0.5 ∼ 1.6 bits per key); **2)** MapEmbed supports fast incremental updates (20× faster than Dynamic Perfect Hashing); **3)** MapEmbed's load factor is high (90% ∼ 95%); **4)** We implemented MapEmbed on a FPGA platform, achieving fast lookup speed (367 Million operations per second).

2 RELATED WORK

Existing hashing schemes can be divided into two categories: perfect hashing without collision and imperfect hashing with collision. Perfect hashing can be further divided into two categories: static perfect hashing and dynamic perfect hashing. Perfect hashing schemes only need one probe of a bucket for each query, but cannot support incremental updates. In practice, one bucket can store multiple KV pairs. Thanks to cache line and SIMD (Single Instruction and Multiple Data [19]), probing a bucket with multiple KV pairs achieves similar speed with probing a bucket with only one KV pair. Therefore, in our perfect hashing, we store multiple KV pairs in each bucket to fully utilize the characteristics of commodity memory. For more hashing schemes, please refer to the literature [20–24], and the survey [25].

2.1 Perfect Hashing

Perfect hashing searches hash functions that map n distinct elements in a set S to m ($m \geq n$) buckets without collision. When $m = n$, it is called minimal perfect hashing.

Static Perfect Hashing expects to find a perfect hash function for each key in a given set. Typical schemes of static perfect hashing include FKS, CHD, BDZ, BMZ, BRZ, CHM, and FCH [13–17, 26–29]. The FKS hashing [14] consists of many buckets, and each bucket has many slots. To insert an item, FKS first hashes it into a subtable and uses another hash function to hash it into a bucket in that subtable. When a hash collision occurs in a bucket, the FKS hashing changes the hash function of the subtable repeatedly until there is no collision. To reduce hash collision, the load factor of each subtable is kept very low, *i.e.*, below $1/\sqrt{k}$ where k is the size of each subtable.

Dynamic Perfect Hashing (DPH) [18] aims to support incremental update based on FKS [14]. Its key idea is divide-and-conquer: it divides the KV pairs into many groups and builds one small perfect hash table for each group. When a hash collision happens in a small perfect hash table, it just reconstructs the table by changing the hash function by brute force. To make reconstructions infrequent, the load factor of each subtable is kept very low (often less than 15%). It is the most well-known work to enable Perfect hashing to support increment update, but it is inefficient in both update speed and memory usage. Compared to dynamic perfect hashing,

[4]https://github.com/MapEmbed/MapEmbed

we aim to improve both the load factor and update speed significantly.

2.2 Imperfect Hashing

Most existing hash schemes [30–35] are imperfect hashings, which cannot achieve one hash probe per lookup. In these schemes, hash collision is inevitable and addressed using different strategies. Typical solutions include hash chaining, open addressing, using multiple subtables, *etc.*

Cuckoo hashing [30] is an illuminating hashing scheme using the kick operation. It has two hash functions corresponding to two subtables, and each table consists of multiple buckets. To insert a key, it selects two candidate buckets from the two tables by hashing and checks if there is an available (not full) bucket among the two buckets. If so, it inserts the key into the available bucket, and the insertion succeeds. Otherwise, it kicks an old key key_{old} from one of the two candidate buckets and inserts the new key into it. Then it probes the other candidate bucket of key_{old} and tries to insert the old key to the bucket. If the bucket is unavailable, the kick operation performs again. This procedure repeats until there is no collision. When allowing at most 500 kicks and each bucket stores four items, the cuckoo hashing can achieve a load factor of 95%. Cuckoo hashing needs two hash probes in the worst case for each lookup, and 1.5 on average. Many variants and applications of cuckoo hashing have achieved great success, including BCHT [31], cuckoo filter [32, 36], and more [33–35, 37–43].

Table 1: Main Symbols Used in This Paper.

Symbol	Meaning
S	Number of cells in fast memory
M	Number of buckets in slow memory
D	Number of buckets that a cell is associated with
N	Number of items a bucket can store
L	Number of layers of cell arrays in fast memory
$C[j]$	The offset number stored in the j^{th} cell (cell j)
$\mathcal{H}_c(\cdot)$	The hash function mapping a key to a cell
$\mathcal{H}_D(\cdot)$	The hash function mapping a key to an index, which is an integer in $[0, D-1]$
$h_b^{(i)}(\cdot)$	The hash function (`bucket function`) that is used for selecting the i^{th} ($i \in [0, D-1]$) associated bucket given a cell
$h^i(\cdot)$	The hash function that is used for constructing $h_b^{(i)}(\cdot)$
$\mathcal{B}[h_b^{(i)}(j)]$	The i^{th} ($i \in [0, D-1]$) bucket that is associated with cell j

3 THE MAPEMBED HASH TABLE

3.1 Data Structure

As shown in Figure 2, the data structure of MapEmbed consists of two parts: **1) A large KV table** used to store KV pairs (items). This table is stored in slow memory. It is organized as an array of M buckets, $\mathcal{B}[0], \mathcal{B}[1], \ldots, \mathcal{B}[M-1]$, where each bucket can store N items. **2) A small index table** used to track the index of the current items. This table is stored in fast memory. It is organized as an array of S cells and is associated with a hash function $\mathcal{H}_c(\cdot)$. Each cell j stores an offset number $C[j]$. Each cell j is associated with D buckets by `bucket functions` $h_b^{(0)}(j), h_b^{(1)}(j), \ldots, h_b^{(D-1)}(j)$, which are called `the` D `associated buckets of cell` j. Specifically, the D associated buckets of cell j are $\mathcal{B}[h_b^{(0)}(j)], \mathcal{B}[h_b^{(1)}(j)], \ldots, \mathcal{B}[h_b^{(D-1)}(j)]$. The main symbols used in this paper are shown in Table 1.

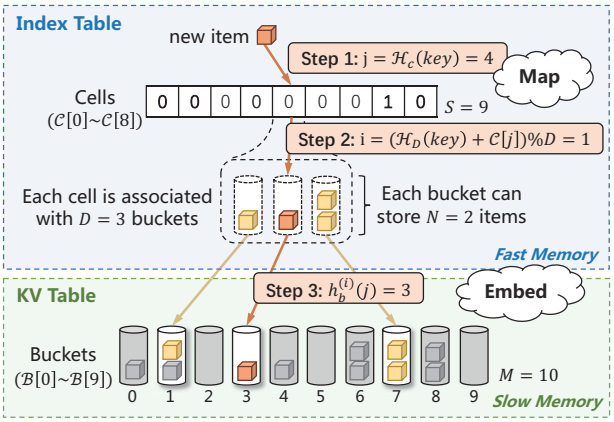


Figure 2: Insertion Operation of MapEmbed.

3.2 Basic Operations

Insertion: The process to insert a new item $\langle key, value \rangle$ is illustrated in Figure 2 and Figure 3. There are four steps. The Map Phase includes the first two steps, and the Embed phase includes the rest two steps. The details are as follows:

- Step 1: select a cell. We select cell j by calculating $j = \mathcal{H}_c(key)$.
- Step 2: select a bucket. The bucket is selected from the D associated buckets of cell j, *i.e.*, from $\mathcal{B}[h_b^{(0)}(j)]$, $\mathcal{B}[h_b^{(1)}(j)]$, $\ldots$, $\mathcal{B}[h_b^{(D-1)}(j)]$. First, we calculate an index value i by $i = (\mathcal{H}_D(key) + C[j]) \bmod D$, where $C[j]$ is the offset number stored in cell j. Second, we select the i^{th} bucket to place the item, *i.e.*, select $\mathcal{B}[h_b^{(i)}(j)]$.
- Step 3: insert the item. We insert the item into bucket $\mathcal{B}[h_b^{(i)}(j)]$. Note this may cause $\mathcal{B}[h_b^{(i)}(j)]$ overflow. If overflow does not happen, the insertion completes. Otherwise, we go to step 4.
- Step 4: update offset and move items. We first check whether $C[j] = D-1$. If so, which means the items of cell j cannot be moved, the insertion fails. Otherwise, we increment $C[j]$ by one. Then we perform a **circular move operation** as follows. For each of the D associated buckets of cell j, for each item in it, we execute step 1 to check whether the item is mapped into cell j. Take the i^{th} associated bucket $\mathcal{B}[h_b^{(i)}(j)]$ as an example, for each item $\langle key_0, value_0 \rangle$ in it, we check whether $\mathcal{H}_c(key_0) = j$. If so, we move the item to the next associated bucket $\mathcal{B}[h_b^{((i+1) \bmod D)}(j)]$. Finally, we check if there is any bucket overflow under the current arrangement. If not, the insertion operation completes. Otherwise, we repeat step 4.

Lookup: The process to look up a key is illustrated in Figure 4 and proceeds as follows:

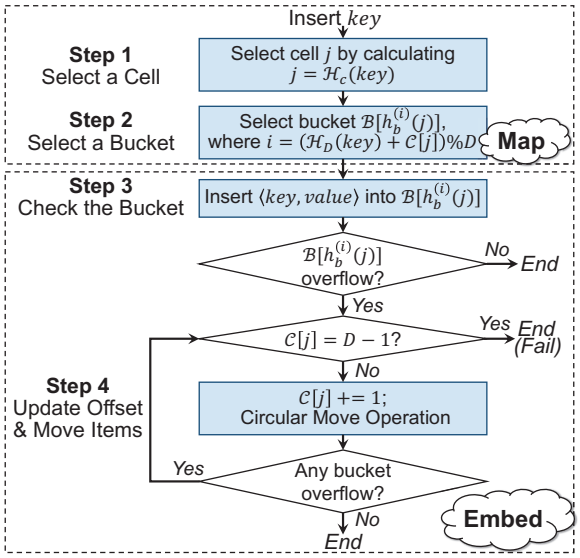


Figure 3: Insertion Operation.

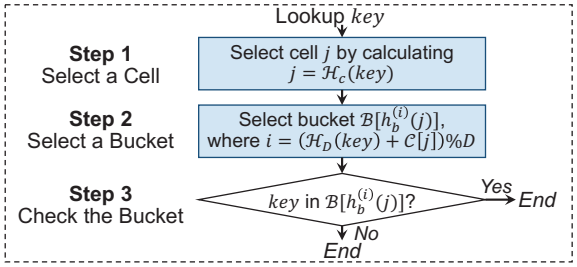


Figure 4: Lookup Operation.

- Step 1: select a cell. We select cell j for the key by calculating $j = \mathcal{H}_c(key)$.
- Step 2: select a bucket. We select bucket $\mathcal{B}[h_b^{(i)}(j)]$ where $i = (\mathcal{H}_D(key) + C[j]) \bmod D$.
- Step 3: check the selected bucket. We search bucket $\mathcal{B}[h_b^{(i)}(j)]$ for key and return the results.

Deletion: To delete an item, we first look up it and then delete it.
Examples: We illustrate the *circular move operation* in Figure 5. The three examples show a continuous insertion process of three different KV pairs. Each column is an example.
Example 1: The blue item $\langle key_1, value_1\rangle$ is mapped to cell 5 by $\mathcal{H}_c(key_1) = 5$ and is dispatched to the 3^{th} associated bucket ($\mathcal{B}[4]$) of cell 5, where $\mathcal{H}_D(key_1) = 3$ and $h_b^{(i)}(j) = 4$. As bucket $\mathcal{B}[4]$ is full, we increment $C[5]$ by one and start the circular move operation. For each of *the D associated buckets of cell* 5 (the buckets with white background), we check each item in it for whether the item is mapped into cell 5 in step 1. Then we move all the items mapped to cell 5 (the colored items) to the next associated bucket. After this operation, no bucket overflows, so the insertion completes.
Example 2: The green item $\langle key_2, value_2\rangle$ happens to be mapped to cell 5 again. Next, it is dispatched to a full bucket $\mathcal{B}[7]$. Then we increment $C[5]$ and perform the circular move operation twice to arrange all items mapped into cell 5 successfully.
Example 3: The red item $\langle key_3, value_3\rangle$ is mapped into cell 3 and bucket $\mathcal{B}[4]$, and overflow happens at $\mathcal{B}[4]$. We increment $C[3]$ by one and perform the circular move operation for cell 3.
Analysis: Due to space limitation, we present the theoretical analysis about how the parameter of MapEmbed influences its performance in Appendix A.

3.3 Dynamic Expansion

In practical applications, the dataset often changes dynamically, so we cannot always set the optimal size of the KV table beforehand. To address this problem, we propose a *dynamic expansion* algorithm, which doubles the number of buckets in slow memory, while keeping the cells in fast memory unchanged.

As mentioned before, the D associated buckets of cell j are selected by `bucket functions` $h_b^{(0)}(j), h_b^{(1)}(j), \ldots, h_b^{(D-1)}(j)$. We first elaborate on how the D `bucket functions` are constructed. This is done by D hash functions $h^0(\cdot), h^1(\cdot), \ldots, h^{D-1}(\cdot) : cell \mapsto \{0, ..., INT\_MAX\}$, which hash a cell (or a cell's unique ID) to an integer. The i^{th} `bucket function` is constructed by $h_b^{(i)}(\cdot) = h^i(\cdot) \bmod M$, where M is the number of buckets in slow memory.

Our *dynamic expansion* algorithm proceeds as follows. First, we perform a **copy operation** to copy all the M buckets in slow memory, and then append the copied M buckets to the original M buckets. After this, there are $2M$ buckets (numbered $0, 1, \ldots, 2M-1$) in slow memory, where $\mathcal{B}[0]$ is the same as $\mathcal{B}[M]$, $\mathcal{B}[1]$ is the same as $\mathcal{B}[M+1]$, *etc.* Second, we mark all buckets in slow memory as *redundant*, meaning there could be redundant items in the bucket.

Afterward, when a *redundant* bucket is probed during insertion/lookup operation, we perform a **redundancy-clean operation**: For each item in this bucket, we check whether it is a redundancy. Specifically, given a *redundant* bucket $\mathcal{B}[b]$, for each item $\langle key, value\rangle$ in it, we check whether $b \neq h_b^{(i)}(j)$, where $j = \mathcal{H}_c(key)$, and $i = (\mathcal{H}_D(key) + C[j]) \bmod D$. If so, which means the item is a redundancy, we delete it from bucket b. After checking all items in a *redundant* bucket, we mark the bucket as "*clean*".
Analysis: We briefly explain why the expansion operation is reasonable. For an item $\langle key, value\rangle$, suppose before expansion, it is mapped to cell j and bucket $\mathcal{B}[b]$, where $j = \mathcal{H}_c(key)$, $b = h_b^{(i)}(j) = h^i(j) \bmod M$, and $i = (\mathcal{H}_D(key) + C[j]) \bmod D$. After expansion, the item exists in both $\mathcal{B}[b]$ and $\mathcal{B}[b+M]$. According to our rules for selecting the bucket in step 1, the item should be placed in $\mathcal{B}[b']$, where $b' = h_b^{(i)}(j) = (h^i(j) \bmod 2M)$. And we have:

$$(h^i(j) \bmod 2M) = \begin{cases} (h^i(j) \bmod M) & = & b \\ (h^i(j) \bmod M) + M & = & b + M \end{cases}$$

For example, suppose $h^i(j) = 10$ and $M = 7$, then $b = h^i(j) \bmod M = 10 \bmod 7 = 3$. And $b' = h^i(j) \bmod (2M) = 10 \bmod 14 = 3 + 7 = b + M$. This property guarantees that each item can still be retrieved after the expansion operation.

3.4 Optimization using Multi-Layer Index

To improve the load factor, we propose the multi-layer version of MapEmbed by dividing the cell array into L (*e.g.*, 3) layers (see Figure 6). Each layer has an independent hash function $\mathcal{H}_c(\cdot)$ to map a key into a cell. The hash function of the i^{th} layer is denoted

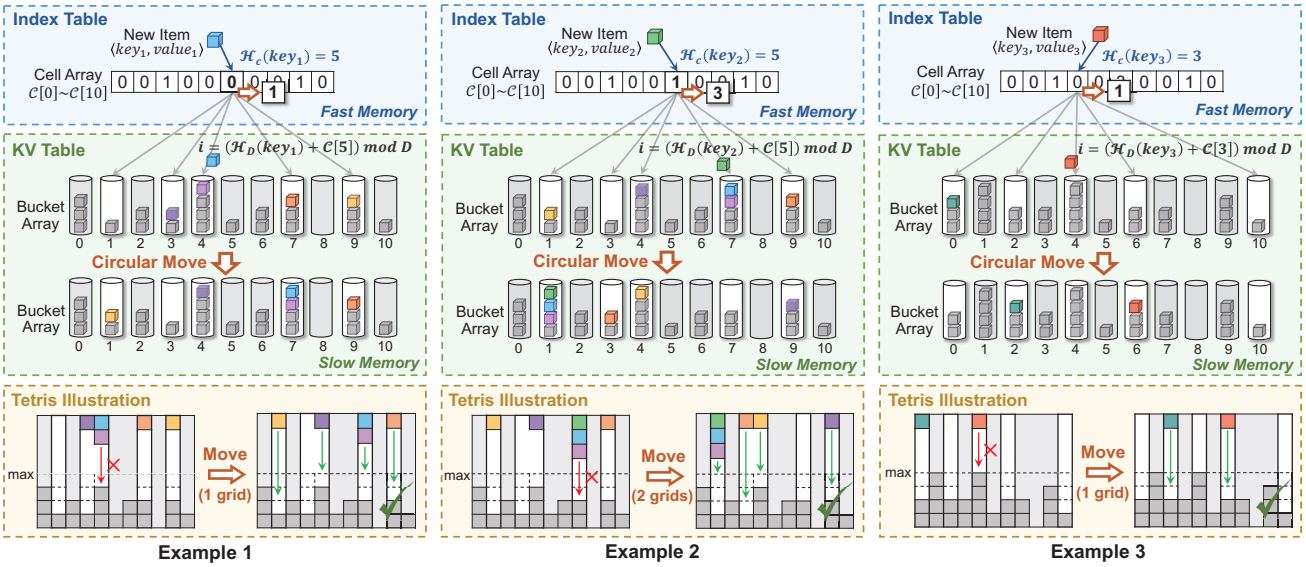


Figure 5: Three examples of MapEmbed illustrating the circular move operation ($S = 11$, $D = 5$, $N = 4$, and $M = 11$).

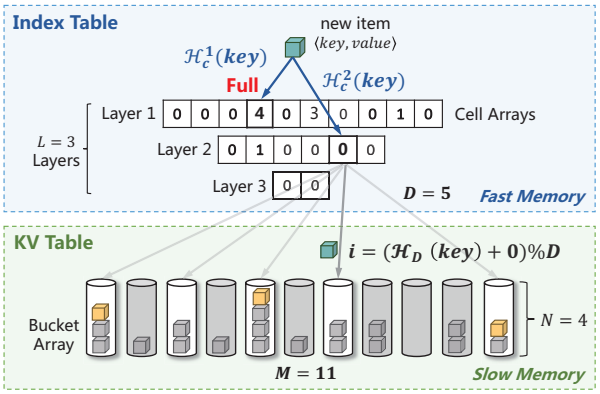


Figure 6: Multi-Layer MapEmbed.

as $\mathcal{H}_c^i(\cdot)$. The number of cells is smaller and smaller layer by layer. For each cell j ($j \in [0, S^{(l)}]$ where $S^{(l)}$ denotes the number of cells in the l^{th} layer), we use $C[j] = D - 1$ to indicate that cell j "*full*". When an item is mapped into a "*full*" cell, we should insert it to the next layer. The available range of the offset number decreases from $[0, D-1]$ to $[0, D-2]$.

Insertion: To insert an item $\langle key, value \rangle$, we first try to insert it into the first layer (suppose it is mapped to cell j at the first layer where $j = \mathcal{H}_c^1(key)$). If the insertion fails, *i.e.*, overflow happens and $C[j] = D-2$ (maximum available offset), we delete all the items that were mapped into cell j from its D mapped buckets, and insert these items into the second layer. Then we mark cell j as "*full*", *i.e.*, set $C[j]$ to $D-1$, and insert $\langle key, value \rangle$ into the second layer. This process repeats layer by layer until all items are successfully inserted, or insertion fails at the last layer. Note that when inserting an item, if we find its current mapped cell "*full*", we try to insert the item into the next layer. For example, as shown in Figure 6, when inserting $\langle key, value \rangle$, we first select its mapped cell $j = \mathcal{H}_c^1(key)$ at the first layer. Then we find cell j is "*full*", so we insert the item into the second layer.

Lookup: Given a key, to look up its value, we first look up it at the first layer. If its mapped cell at the first layer is marked as "*full*", we look up it at the second layer. This process repeats until the mapped cell is not "*full*".

4 EXPERIMENTAL RESULTS

4.1 Experimental Setup

Platform: All the CPU simulation experiments were performed on a server with 18-core CPU (36 threads, Intel CPU i9-10980XE @3.00 GHz), which have 128GB memory.

Implementation: We implement all the codes with C++ and build them with g++ (Ubuntu 7.5.0-6ubuntu2) 7.5.0 and -O3 option. The hash functions we use are 32-bit Murmur Hash [44] with random initial seeds. We use SIMD [19] to accelerate the process of probing a bucket.

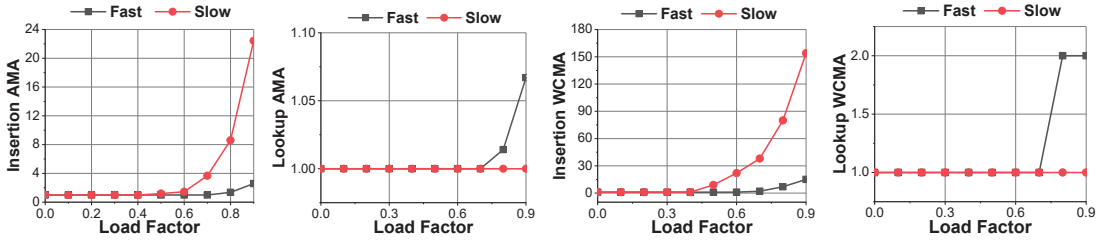
Datasets:

1) CAIDA Dataset: CAIDA Anonymized Internet Trace [45] is a set of anonymized IP packets collected from high-speed monitors on backbone links. It contains approximately 30M packets belonging to 1.1M different flows. Each flow is identified by a 4 Byte source IP address and a 4 Byte destination IP address. We combine the two IP addresses to form the 8B key of each KV pair and use the 8B timestamp of each packet as the value field of each KV pair.

2) WebDocs Dataset: WebDocs [46] is a collection of web HTML documents built by crawling web pages. It contains about 10M packets, of which 2.7M packets are distinct. We use the 13 Byte packet in this dataset as the key of each KV pair, and randomly generate the 13 Byte value field of each KV pair.

3) Synthetic Dataset: We generate ten different synthetic datasets. Each dataset contains 10M KV pairs. The key and value fields of each KV pair are both 4 Byte strings generated randomly from a uniform distribution. We guarantee all the keys in each dataset are distinct.

Default Parameters: We set the default parameters of MapEmbed as follows. We let each bucket stores $N = 16$ elements. We divide



(a) Insertion AMA. (b) Lookup AMA. (c) Insertion WCMA. (d) Lookup WCMA.

Figure 7: Performance of MapEmbed Hashing under Different Load Factor (Synthetic).

MapEmbed's index cells into $L = 3$ layers. The first layer has 3 times as many cells as the second layer, and the second layer has 3 times as many cells as the third layer. Each cell is associated with $D = 16$ buckets. Furthermore, we set the size of the slow memory to the same as the size of the insertion workload, *i.e.*, the distinct KV pairs in the dataset, and set the fast memory size to 0.015 times of the slow memory. All experiments are repeated 10 times and the average result is reported.

Evaluation Metrics:

1) Load Factor: We insert KV pairs into the hash table sequentially until we encounter the 8^{th} insertion failure. The ratio of the number of the inserted KV pairs to the number of KV pair slots in slow memory is defined as the Load Factor, reflecting memory efficiency.

2) Throughput (Mops): Million operations per second (Mops). We use *Throughput* to evaluate the average insertion/lookup speed.

3) Average Memory Access (AMA): $\frac{Q}{n}$, where n refers to the number of the KV pairs we insert or look up, and Q refers to the number of memory accesses (fast memory or slow memory) during the whole insertion or lookup process.

4) Worst-Case Memory Access (WCMA): $\max\{q_i\}_{i=1}^{n}$, where q_i refers to the number of memory accesses when inserting or looking-up the i^{th} KV pair.

4.2 The Impact of Algorithm Parameters

AMA *vs.* Load Factor (Figure 7(a)-7(b)): We find that when the load factor is below 60%, the insertion AMA for both fast and slow memory is 1. The lookup AMA for slow memory is always 1, and the lookup AMA for fast memory is 1 when the load factor is below 80%.

WCMA *vs.* Load Factor (Figure 7(c)-7(d)): We find that when the load factor is below 40%, the insertion WCMA for both fast memory and slow memory is 1. When the load factor is 90%, the insertion WCMA for fast memory is about 150, and the insertion WCMA for slow memory is about 15. The lookup WCMA for slow memory is always 1, and the lookup WCMA for fast memory is 1 in most cases.

Load Factor *vs.* Cell Size (Figure 8(a)): We find that letting each cell occupy 4 bits (associated with $2^4 = 16$ buckets) is the best choice. It is noted that there is a trade-off between the size and the number of index cells. We explore how many bits a cell should use, or how many buckets a cell should be associated with (*i.e.*, how to choose D). Here, we let D take any positive integer value, rather than just the power of two. For each value D_i, we define the equivalent cell size as $t = \log_2 D_i$ bits. We set the fast memory to a total size of 84, 000 bits and change the number of buckets M. From Figure 8(a), we can observe the impact of D on load factor, which indicates that allocating 4 bits to each cell is the best choice.

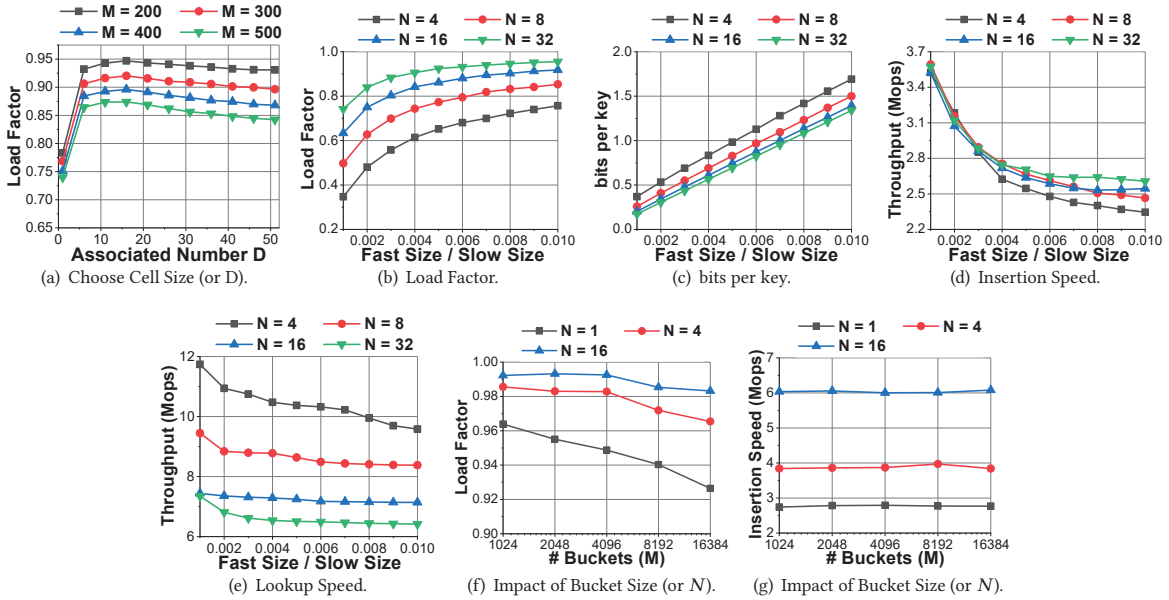
Performance *vs.* Fast-Slow Memory Ratio (Figure 8(b)-8(e)): We find that as the Fast-Slow Memory Ratio becomes larger, MapEmbed's load factor becomes higher. We fix the total memory size (fast memory and slow memory) and change the ratio of fast memory to slow memory (*fast-slow ratio*). We also change the number of KV pairs that a bucket can accommodate (N). The experiments are conducted on the CAIDA and WebDocs datasets. Since the results are similar on the two datasets, we only present the results on CAIDA here. From Figure 8(b), we find that the load factor is higher when the *fast-slow ratio* is larger or N is larger. From Figure 8(c), we find that a KV pair consumes more space in fast memory when the *fast-slow ratio* is larger or N is smaller. In particular, we find that when the *fast-slow ratio* is 0.005 and $N = 32$, *MapEmbed can achieve a load factor of **93%**, and each KV pair in slow memory only consumes **0.5 bits** on average in fast memory*. Figure 8(d) and Figure 8(e) show the average processing speed of MapEmbed with the change of *fast-slow ratio* and N. We can see that the larger the *fast-slow ratio* is, the slower the insertion speed will be. Moreover, when N becomes larger, both the insertion speed and the lookup speed decrease.

Impact of Bucket Size (N) (Figure 8(f)-Figure 8(g)): We find that the performance of MapEmbed is better when using larger bucket size. When using the bucket size of $N = 16$, MapEmbed achieves a load factor of 98% and the insertion speed of 6 Mops. This is because smaller N leads to more frequent collisions, which compromises the load factor and the insertion speed. But larger N will slow down the lookup speed (see Figure 8(e)).

Single-thread Speed *vs.* Load Factor (Figure 9(a)): We find that when the load factor is below 50%, the single-thread insertion throughput is higher than 10 Mops. When the load factor exceeds 50%, the insertion speed drops because of more frequent memory access. And we find that the single-thread lookup throughput can reach 30 Mops, which does not decrease with the increase of load factor. The experimental results are consistent with that of the AMA and WCMA.

Multi-thread Speed (Figure 9(b)): We find that after using multi-threaded acceleration, MapEmbed can achieve up to 250 Mops query throughput and 20 Mops insertion throughput. Here, we conduct experiments using 32-threads on the synthetic dataset, and we vary the workload from 1 million KV pairs to 16 million KV pairs.

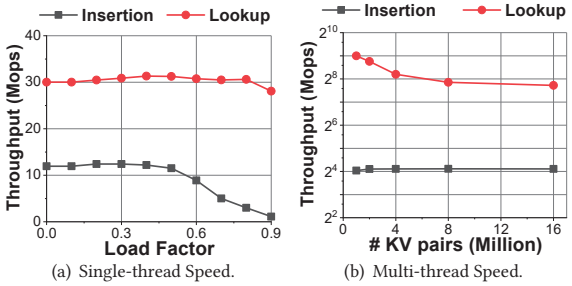
Load Factor *vs.* Number of Cell Layers (Figure 10): We find that the multi-layer optimization can significantly improve the load



(a) Choose Cell Size (or D). (b) Load Factor. (c) bits per key. (d) Insertion Speed.

(e) Lookup Speed. (f) Impact of Bucket Size (or N). (g) Impact of Bucket Size (or N).

Figure 8: The Impact of MapEmbed Hashing Parameters (CAIDA).



(a) Single-thread Speed. (b) Multi-thread Speed.

Figure 9: Speed Performance of MapEmbed (Synthetic).

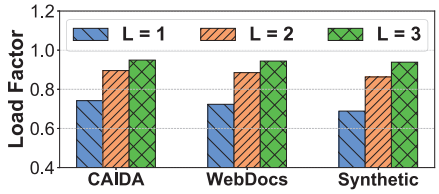


Figure 10: The impact of Multi-Layer Index.

factor of MapEmbed. We keep the size of fast memory unchanged and vary the number of layers of index cells in fast memory. When using $L = 3$ layers of index cells, MapEmbed can achieve a load factor of more than 97%. In contrast, when just using $L = 1$ layer of index cells, the load factor is below 80%.

4.3 Evaluation of Dynamic Expansion

We evaluate the expansion performance of MapEmbed in terms of bits per key and load factor (see Figure 12). In the beginning, we set the *fast-slow ratio* to 0.29 and set the number of buckets M to 4000. Then we insert the KV pairs from the Synthetic dataset into MapEmbed sequentially. Every time the 8^{th} insertion failure happens, we perform the dynamic expansion operation to double the buckets and then proceed to insert the items. From Figure 12(a), we find that when the number of expansion increases, bits per key decreases significantly, because more KV pairs are inserted. From Figure 12(b), we find that when the number of expansion increases, the load factor drops, because the *fast-slow ratio* decreases due to the expansion operation.

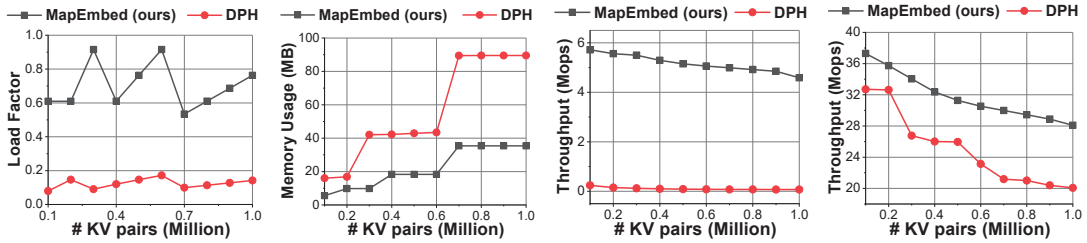
4.4 Comparison with Prior Art

4.4.1 *Dynamic Perfect Hashing (DPH).*

We evaluate the memory efficiency and insertion/lookup speed of MapEmbed and DPH. We change the insertion workload from 0.1 Million to 1 Million. For Figure 11(a) and 11(b), we initialize the number of buckets $M = 50$, and insert the KV pairs into the two schemes. For MapEmbed, each time the insertion operation fails, we perform the dynamic expansion operation to double the buckets. For Figure 11(c), and 11(d), the number of buckets M in MapEmbed is dynamically adjusted according to the insertion workload. For each dataset, we try to insert all KV pairs sequentially into the two schemes until the 8^{th} insertion failure. Next, we look up the inserted keys in the two schemes. Since the experimental results on the three datasets are similar, we only show some representative results. The other parameters of MapEmbed are set as the default parameters described in Section 4.1.

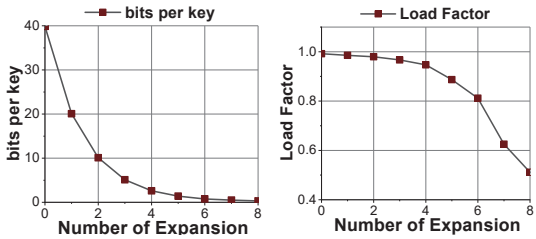
Memory Efficiency (Figure 11(a)-11(b)): We find that the load factor of MapEmbed is about 5.0× higher than DPH. We also find that when the insertion workload is fixed, DPH needs 5.0× space than MapEmbed. As DPH adopts a very low load factor to ensure the rebuilding operation infrequent, its memory efficiency is poorer.

Insertion/Lookup Speed (Figure 11(c)-11(d)): We find that the insertion speed of MapEmbed is about 20× faster than DPH. When the load factor is below 20%, the insertion speed of MapEmbed achieves 21.69× faster than DPH. The superiority of MapEmbed



(a) Load Factor (WebDocs). (b) Memory Usage (CAIDA). (c) Insertion Speed (Synthetic). (d) Lookup Speed (Synthetic).

Figure 11: MapEmbed Hashing *vs.* DPH.



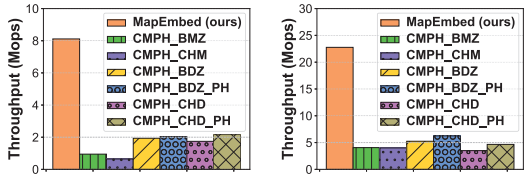
(a) bits per key. (b) Load Factor.

Figure 12: Evaluation of Expansion Operation (Synthetic).

is due to 2 reasons. First, in most cases, MapEmbed only needs to access memory twice per insertion, while DPH might need to rebuild the full hash table, which results in many memory accesses. Second, MapEmbed does not need to change hash function by brute force during insertion, which is time-consuming. And we find that the lookup speed of MapEmbed is about 1.41× faster than DPH.

4.4.2 *Static Perfect Hashing (SPH).*

We compare MapEmbed with four minimal perfect hashing (MPH) schemes (*i.e.*, BMZ, CMH, BDZ, and CHD) and two perfect hashing (PH) schemes (*i.e.*, BDZ_PH and CHD_PH) in CMPH library [13]. The CMPH library encapsulates the newest and most efficient MPH algorithms in a production-quality API. As the algorithms in the CMPH library can only be constructed statically, we define their average insertion throughput as the number of inserted keys divided by the construction time. The experiments are performed on the CAIDA dataset and the Synthetic dataset. Since the experimental results on the two datasets are similar, we only show results on the CAIDA dataset.



(a) Insertion Speed. (b) Lookup Speed.

Figure 13: MapEmbed Hashing *vs.* SPH (CAIDA).

Insertion/Lookup Speed (Figure 13): We find that the insertion/lookup speed of MapEmbed is significantly better than the six static perfect hashing schemes. The fastest SPH scheme CMPH_CHD can only achieve a insertion throughput of 2.07 Mops, whereas that of MapEmbed can reach 8.0 Mops. And the lookup throughput of MapEmbed is also at least 4× faster than SPH.

Memory Efficiency (Table 2): We find that MapEmbed significantly outperforms the candidate MPH/PH schemes in terms of memory efficiency. The load factor of the two candidate PH schemes depends on the size of the index structure. As shown in Table 2, BDZ_PH has a load factor of 81.3% at the cost of 1.95 bits per key, and CHD_PH has a load factor of 50.0% at the cost of 0.67 bits per key [13]. Compared with the two PH schemes, MPH needs more space for the index ($\geqslant$ 2.07 bits per key). We can see that MapEmbed performs better (93% load factor at the cost of 0.5 bits per key).

Table 2: Memory Efficiency of MapEmbed and SPH.

Algorithms	Load Factor	bits per key
MapEmbed	93%	0.5
BDZ	100%	2.6
BDZ_PH	81.2%	1.95
CHD	100%	2.07
CHD_PH	50%	0.67

4.5 Evaluation on FPGA Platform

We implement the lookup module of MapEmbed on an FPGA network experimental platform (Virtex-7 VC709). The FPGA integrated with the platform is xc7vx690tffg1761-2 with 433200 Slice LUTs, 866400 Slice Register, and 850 Block RAM Tile (*i.e.*, 30.6Mb on-chip memory). The implementation mainly consists of three hardware modules: calculating hash values (calculating), searching the table (searching), and writing the matched results (writing). FPGA-based MapEmbed is fully pipelined, which can input one key in every clock, and output one result after four clocks. According to the synthesis report (see Table 3), the clock frequency of our implementation in FPGA is 367 MHz, meaning the throughput of the system can be 367 Mops. Moreover, the logic resource usage is 0.98%, and memory resource usage is 0.3%.

Table 3: Performance on FPGA Platform.

Module	Resource Overhead			Frequency (MHz)
	LUTs	Register	Block Tile	
Calculating	1911	47	0	367
Searching	2312	2268	0	367
Writing	5	5	0	367
Total	4228	2320	0	367

5 CONCLUSION

Perfect hashing is well known for its constant worst-case lookup time: only one hash probe for each lookup. Most perfect hashing scheme cannot support incremental updates. The state-of-the-art hashing can support incremental updates at the cost of memory

inefficiency. To address this issue, we propose MapEmbed Hashing. Extensive experiments on CPU and FPGA platforms show that compared to the state-of-the-art hashing (dynamic perfect hashing): 1) MapEmbed improves the update speed 20 times; 2) MapEmbed improves the load factor from around 15% to 90% ∼ 95%; 3) MapEmbed's index structure is quite small (*e.g.*, 0.5 ∼ 1.6 bits per key).

ACKNOWLEDGMENT

We thank Yikai Zhao for his help on experiments. This work is supported by Primary Research & Development Plan of China (2016YFB1000304), and National Natural Science Foundation of China (NSFC) (No. U20A20179).

REFERENCES

[1] Haoyu Zhang and Qin Zhang. Minjoin: Efficient edit similarity joins via local hash minima. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1093–1103, 2019.
[2] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.
[3] Helen HW Chan, Chieh-Jan Mike Liang, Yongkun Li, Wenjia He, Patrick PC Lee, Lianjie Zhu, Yaozu Dong, Yinlong Xu, Yu Xu, Jin Jiang, et al. Hashkv: Enabling efficient updates in {KV} storage via hashing. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 1007–1019, 2018.
[4] Haoyu Song, Sarang Dharmapurikar, Jonathan Turner, and John Lockwood. Fast hash table lookup using extended bloom filter: an aid to network processing. *ACM SIGCOMM Computer Communication Review*, 35(4):181–192, 2005.
[5] Hao-Jun Michael Shi, Dheevatsa Mudigere, Maxim Naumov, and Jiyan Yang. Compositional embeddings using complementary partitions for memory-efficient recommendation systems. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 165–175, 2020.
[6] Wei Wu, Bin Li, Ling Chen, and Chengqi Zhang. Efficient attributed network embedding via recursive randomized hashing. In *IJCAI International Joint Conference on Artificial Intelligence*, 2018.
[7] The benefits of altera's high-speed ddr sdram memory interface solution. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/wp/wp_stratix_ddr.pdf.
[8] Wang Feng and Hamdi Mounir. Matching the speed gap between sram and dram. In *Proc. IEEE HSPR*, pages 104–109, 2008.
[9] Emmanuel Esposito, Thomas Mueller Graf, and Sebastiano Vigna. Recsplit: Minimal perfect hashing via recursive splitting. In *2020 Proceedings of the Twenty-Second Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 175–185. SIAM, 2020.
[10] Chin-Chen Chang and Chih-Yang Lin. Perfect hashing schemes for mining association rules. *The Computer Journal*, 48(2):168–179, 2005.
[11] Yi Lu, Balaji Prabhakar, and Flavio Bonomi. Perfect hashing for network applications. In *2006 IEEE International Symposium on Information Theory*, pages 2774–2778. IEEE, 2006.
[12] Rayan Chikhi, Antoine Limasset, and Paul Medvedev. Compacting de bruijn graphs from sequencing data quickly and in low memory. *Bioinformatics*, 32(12):i201–i208, 2016.
[13] C minimal perfect hashing library. http://cmph.sourceforge.net/.
[14] Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with o(1) worst case access time. 1984.
[15] Djamal Belazzougui, Fabiano C Botelho, and Martin Dietzfelbinger. Hash, displace, and compress. In *European Symposium on Algorithms*, pages 682–693. Springer, 2009.
[16] Fabiano C Botelho, Rasmus Pagh, and Nivio Ziviani. Simple and space-efficient minimal perfect hash functions. In *Workshop on Algorithms and Data Structures*, pages 139–150. Springer, 2007.
[17] Fabiano C Botelho, David M Gomes, and Nivio Ziviani. A new algorithm for constructing minimal perfect hash functions. *differences*, 100(2):09, 2004.
[18] Martin Dietzfelbinger, Anna Karlin, Kurt Mehlhorn, Friedhelm Meyer Auf Der Heide, Hans Rohnert, and Robert E Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM Journal on Computing*, 23(4):738–761, 1994.
[19] Intel instructions. Available: https://software.intel.com/sites/landingpage/IntrinsicsGuide/.
[20] Michael A Bender, Martin Farach-Colton, Rob Johnson, Russell Kraner, Bradley C Kuszmaul, Dzejla Medjedovic, Pablo Montes, Pradeep Shetty, Richard P Spillane, and Erez Zadok. Don't thrash: How to cache your hash on flash. *Proc. VLDB Endow.*, 5(11):1627–1637, 2012.
[21] Lailong Luo, Deke Guo, Richard TB Ma, Ori Rottenstreich, and Xueshan Luo. Optimizing bloom filter: Challenges, solutions, and comparisons. *IEEE Communications Surveys & Tutorials*, 21(2):1912–1949, 2018.
[22] Prashant Pandey, Michael A Bender, Rob Johnson, and Rob Patro. A general-purpose counting filter: Making every bit count. In *Proceedings of the 2017 ACM international conference on Management of Data*, pages 775–787, 2017.
[23] Kaan Kara and Gustavo Alonso. Fast and robust hashing for database operators. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4. IEEE, 2016.
[24] Stefan Richter, Victor Alvarez, and Jens Dittrich. A seven-dimensional analysis of hashing methods and its implications on query processing. *PVLDB*, 9(3):96–107, 2015.
[25] A. Kirsch, M. Mitzenmacher, and G. Varghese. Hash-based techniques for high-speed packet processing. In *In Algorithms for Next Generation Networks*, page 181, 2010.
[26] Gnu gperf. https://www.gnu.org/software/gperf/.
[27] Fabiano C Botelho, Yoshiharu Kohayakawa, and Nivio Ziviani. An approach for minimal perfect hash functions for very large databases. *Universidade Federal de Minas Gerais, Belo Horizonte, Brazil, Tech. Rep*, 95, 2006.
[28] Zbigniew J Czech, George Havas, and Bohdan S Majewski. An optimal algorithm for generating minimal perfect hash functions. *Information processing letters*, 43(5):257–264, 1992.
[29] Edward A Fox, Qi Fan Chen, and Lenwood S Heath. A faster algorithm for constructing minimal perfect hash functions. In *Proceedings of the 15th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 266–273, 1992.
[30] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.
[31] Orestis Polychroniou, Arun Raghavan, and Kenneth A Ross. Rethinking simd vectorization for in-memory databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1493–1508, 2015.
[32] Bin Fan, Dave G Andersen, Michael Kaminsky, and Michael D Mitzenmacher. Cuckoo filter: Practically better than bloom. In *ACM CoNEXT*, 2014.
[33] Salvatore Pontarelli, Pedro Reviriego, and Michael Mitzenmacher. Emoma: exact match in one memory access. *IEEE Transactions on Knowledge and Data Engineering*, 30(11):2120–2133, 2018.
[34] Bin Fan, David G Andersen, and Michael Kaminsky. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In *NSDI*, volume 13, pages 385–398, 2013.
[35] Kai Zhang, Kaibo Wang, Yuan Yuan, Lei Guo, Rubao Lee, and Xiaodong Zhang. Mega-kv: a case for gpus to maximize the throughput of in-memory key-value stores. *Proceedings of the VLDB Endowment*, 8(11):1226–1237, 2015.
[36] Lailong Luo, Deke Guo, Ori Rottenstreich, Richard T. B. Ma, Xueshan Luo, and Bangbang Ren. The consistent cuckoo filter. In *IEEE INFOCOM*, 2019.
[37] Dagang Li, Rong Du, Ziheng Liu, Tong Yang, and Bin Cui. Multi-copy cuckoo hashing. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 1226–1237. IEEE, 2019.
[38] Flaviene de Cristo, Eduardo de Almeida, and Marco Alves. Vivid cuckoo hash: Fast cuckoo table building in simd. In *Anais Principais do XX Simpósio em Sistemas Computacionais de Alto Desempenho*, pages 288–299. SBC, 2019.
[39] Alex D Breslow and Nuwan S Jayasena. Morton filters: faster, space-efficient cuckoo filters via biasing, compression, and decoupled logical sparsity. *Proceedings of the VLDB Endowment*, 11(9):1041–1055, 2018.
[40] Harald Lang, Thomas Neumann, Alfons Kemper, and Peter Boncz. Performance-optimal filtering: Bloom overtakes cuckoo at high throughput. *Proceedings of the VLDB Endowment*, 12(5):502–515, 2019.
[41] Minmei Wang, Mingxun Zhou, Shouqian Shi, and Chen Qian. Vacuum filters: more space-efficient and faster replacement for bloom and cuckoo filters. *Proceedings of the VLDB Endowment*, 13(2):197–210, 2019.
[42] David Eppstein, Michael T Goodrich, Michael Mitzenmacher, and Manuel R Torres. 2-3 cuckoo filters for faster triangle listing and set intersection. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 247–260, 2017.
[43] Sourav Dutta, Ankur Narang, and Suman K Bera. Streaming quotient filter: a near optimal approximate duplicate detection approach for data streams. *Proceedings of the VLDB Endowment*, 6(8):589–600, 2013.
[44] Murmur hashing source codes. https://github.com/aappleby/smhasher/blob/master/src/MurmurHash3.cpp.
[45] CAIDA [on line]. Available: http://www.caida.org/home.
[46] The web page data set. http://fimi.ua.ac.be/.

A MATHEMATICAL ANALYSIS

We provide the theoretical analysis of MapEmbed Hashing. We assume the *circular move* operation does not change the distribution of the number of items in a bucket. Based on this assumption, we first derive the probability of *circular move* operation during the t^{th} insertion in Section A.1. We then estimate of the number of memory accesses of MapEmbed's insertion/lookup operation in Section A.2. For the symbols we use in this section, please refer to Table 1.

A.1 Probability of Circular Move

In this section, we derive the probability of the *circular move* operation during the t^{th} insertion operation in the basic version of MapEmbed.

First, it is noticed that the cells in fast memory are randomly mapped to the buckets in slow memory, which means that a bucket can correspond to arbitrary cells. Thus, the possibility of bucket i corresponding to cell j is:

$$p_{i,j} = 1 - \frac{C_{M-1}^{D}}{C_{M}^{D}} \tag{1}$$

where C_a^b represents the combination number, which is defined as:

$$C_a^b := \frac{a!}{b!(a-b)!}$$

Note that all the buckets and cells are symmetrical, thus we suppose:

$$p = p_{i,j} = 1 - \frac{C_{M-1}^{D}}{C_{M}^{D}} = 1 - \frac{M-D}{M} = \frac{D}{M}$$

where $M \geqslant D$ according to the definition.

Then we define the random variable X as the number of cells corresponding to a given bucket. Note that X obeys binomial distribution, so we have

$$P(X = k) = C_S^k \cdot p^k (1-p)^{S-k} \tag{2}$$

where $E(X) = pS$, and $D(X) = p(1-p)S$.

We assume that the *circular move* operation has not happened, and define the random variable Y as the number of items in bucket i after the t^{th} insertion. Then we have

$$P(Y = y | X = k) = C_t^y \cdot q^y (1-q)^{t-y} \tag{3}$$

where $q := \frac{k}{S}$ is the possibility of an item being inserted into the cell corresponding to bucket i.

According to *Total Probability Theorem*, we have

$$\begin{aligned} P(Y = y) &= \sum_{k=1}^{S} P(Y = y | X = k) \cdot P(X = k) \\ &= \sum_{k=1}^{S} [C_t^y \cdot q^y (1-q)^{t-y}] \cdot C_S^k \cdot p^k (1-p)^{S-k} \end{aligned}$$

where q and p are defined as described above.

The *circular move* operation occurs at bucket i when the bucket overflows. Since bucket i is randomly selected, the probability of *circular move* is:

$$\begin{aligned} P(circular) &= \sum_{y=N}^{+\infty} P(Y = y) \\ &= \sum_{y=N}^{+\infty} \sum_{k=1}^{S} [C_t^y \cdot q^y (1-q)^{t-y}] \cdot C_S^k \cdot p^k (1-p)^{S-k} \end{aligned}$$

A.2 Estimation of Memory Accesses

In this section, we consider the multi-layer version of MapEmbed. We first give an estimation of the number of fast memory access during the t^{th} insertion operation. Then we provide the estimated number of fast memory access during a lookup operation after t items have been inserted.

We use $p_c^{(l)}$ to denote the probability of *circular move* occurring in the l^{th} layer during the t^{th} insertion , *i.e.*, $p_c^{(l)} := P^{(l)}(circular)$. Recall that when inserting an item, we first map it into a cell j in the first layer. The insertion operation in the first layer fails if cell j is "*full*", *i.e.*, $C[j] = D-1$, which means the *circular move* operation has happened in cell j for $D-2$ times. So the possibility of the selected cell in the l^{th} layer to be "*full*" is $p_f^{(l)} = (p_c^{(l)})^{D-2}$.

Suppose during the t^{th} insertion, there are $\mathcal{H}[j]$ items in slow memory mapping into cell j. During the t^{th} insertion, if we need to mark cell j as '*full*", we should delete the $\mathcal{H}[j]$ items from their buckets and insert these items into the next layer. This process incurs another $\mathcal{H}[j]$ memory accesses. As $\mathcal{H}[j]$ is independent from the possibility of successful insertion, we have $E(\mathcal{H}[j]) = \frac{t}{S^{(l)}}$, where $S^{(l)}$ denotes the number of cells in the l^{th} layer.

Let the random variable I denote the number of fast memory accesses during the t^{th} insertion, we have

$$E(I) = E(I_1) + E(I_2) \tag{4}$$

where the first term represents the cost of probing fast memory for the t^{th} item, and the second term represents the cost of deleting the $\mathcal{H}[j]$ items and inserting them into the next layer.

And we have

$$E(I_1) = \sum_{k=1}^{L} k \cdot (p_f^{(l)})^{k-1} (1 - p_f^{(l)})$$

and

$$\begin{aligned} E(I_2) &= \sum_{k=1}^{L} (k-1) \cdot (p_f^{(l)})^{k-1} (1 - p_f^{(l)}) \cdot E(\mathcal{H}[j]) \\ &= \sum_{k=1}^{L} (k-1) \cdot (p_f^{(l)})^{k-1} (1 - p_f^{(l)}) \cdot \frac{t}{S^{(l)}} \end{aligned}$$

Let p_b denote the possibility that a given bucket is full after t items have been inserted. We use the random variable U to denote the number of fast memory access during a lookup operation. We have

$$E(U) = \sum_{k=1}^{L} k \cdot p_b^{k-1} (1 - p_b) \tag{5}$$