POLITECNICO DI TORINO
INSTITUT EURECOM

Double Master's Degree in Computer Engineering

Master's Degree Thesis

March 2024

Search With Your Head!

A Study of Learned Indices for Hashing and Sequence Alignment

Supervisors
Prof. Raja APPUSWAMY
Prof. Paolo GARZA

Candidate
Ilaria PILO

EURECOM
Sophia Antipolis

Abstract

In the era of Machine Learning technology, its impact has extended across various domains, reaching countless applications. One result of this trend is the rise of *learned indices*, which employ Machine Learning models to enhance lookup. Among these structures, the Recursive Model Index (RMI) has proven to be particularly successful. In this work, we investigate the utilization of learned indices in two specific use cases.

Learned Functions: In Part I, we discuss the application of learned indices as hash functions, comparing them with many state-of-the-art options. The efficiency of these *learned functions* strongly correlates with the data distribution, and they can yield up to a 1.4x increase in probe throughput for simple datasets. However, the necessity of both a sort and a training phase makes them unsuitable to support non-partitioned join operations.

To hide cache stalls, we propose an implementation of the RMI-based bucket chaining hash table using Instruction Stream Interleaving. This structure increases the probe throughput up to 5x for high load factors, and eliminates the throughput variations between RMIs of different complexity.

Accel-Align-RMI: In Part II, we apply the RMI to the genomic aligner *Accel-Align*. On 16 cores, while the index introduces a 3.7x slowdown in search time, it provides a precise set of candidate locations, accelerating the embedding phase by 1.5x and the overall alignment by 1.2x. It also increases accuracy to 97.25%, an all-time maximum for Accel-Align.

However, the RMI is too precise at the expense of time, and alternative options with more balanced tradeoffs are available. Our proof-of-concept XXH32 hash table aligner, for instance, surpasses it in terms of time (but with slightly lower accuracy).

Source Code Availability: Both the learned function benchmarking repository[1] and the Accel-Align-RMI implementation[2] are available on GitHub.

[1] github.com/IlariaPilo/new-hashing-benchmark

[2] github.com/IlariaPilo/accel-align-rmi

Résumé

Les récentes avancées de l'apprentissage automatique lui ont permis de se propager dans de multiples domaines et de répondre à divers cas d'utilisation : parmi ceux-ci, les *indices appris* ("learned indices" en anglais). Cette technique consiste à utiliser des modèles d'apprentissage automatique pour améliorer la recherche d'index. Parmi ceux-ci, le Modèle Récursif d'Index (RMI) s'est avéré particulièrement efficace.

Dans ce projet, nous évaluons l'impact de l'apprentissage des index dans deux cas d'utilisation spécifiques.

Fonctions Apprises : Dans la Partie I, nous parlons de l'utilisation des indices appris en tant que fonctions de hachage et les comparons à l'état de l'art. Nous remarquons une forte corrélation entre la distribution des données et l'efficacité de ces *fonctions apprises* ; Sur des bases de données peu complexes, cette méthode permet d'obtenir une vitesse de recherche allant jusque 1,4 fois celle de la methode de référence. Cependant, la nécessité d'avoir à la fois une phase de triage et une phase d'entraînement rend cette méthode inadaptée pour les jointures non partitionnées.

Afin d'annuler les effets des blocages de la mémoire cache, nous utilisons une table de hachage à chaînage de seaux (basée sur RMI) qui permet l'entrelacement de flux d'instructions. Cette structure permet de multiplier par 5 la vitesse de recherche pour des facteurs de charge élevés, et élimine les variations de vitesse entre RMI de différentes complexités.

Accel-Align-RMI : Dans la Partie II, nous appliquons RMI à l'aligneur de génome *Accel-Align*. Avec 16 cœurs, nous observons un ralentissement de 3,7 fois de la vitesse de recherche. Cependant, on remarque une significative amélioration de la précision des positions candidates fournies, ce qui accélère la phase suivante considérablement et donne une vitesse de traitement globale 1,2 fois supérieure. Nous pouvons également noter que la précision atteint les 97,25% ce qui est un record pour Accel-Align.

Malgré tout, la précision de RMI se fait au détriment du temps de traitement et il existe des alternatives permettant un meilleur compromis. Par exemple, notre prototype d'aligneur de table de hachage XXH32 réalise des performances légèrement inférieurs en termes de précision mais avec des temps de traitements bien moindres.

Table of Contents

Acronyms

AES Advanced Encryption Standard.

CDF Cumulative Distribution Function.

CV coefficient of variation.

FIFO First In, First Out.

i.i.d. independently and identically distributed.

INDEL insertion/deletion.

ISI Instruction Stream Interleaving.

LLC last-level cache.

MAPQ mapping quality.

NPJ non-partitioned hash join.

NUMA non-uniform memory access.

OLTP Online Transaction Processing.

pdf probability density function.

PGM Piece-wise Geometric Model.

PLA-model Piece-wise Linear Approximation model.

RecSplit Recursive Splitting.

RMI Recursive Model Index.

SEE seed-embed-extend.

SFE seed-filter-extend.

SOSD Search on Sorted Data.

Part I

Learned Indices as Hash Functions

Chapter 1

Theoretical Background

1.1 Learned Indices and RMI

Let's begin with a simple thought experiment.

Suppose we have a list of 10 keys, ranging from 0 to 99. Keys have been sorted in increasing order. Now, we know that one of these keys is 42.

If we had to guess, where can we find this key on the list?

If you're wondering where the trick is, here it comes: we do not know! We first need to make some considerations on the key *distribution*.

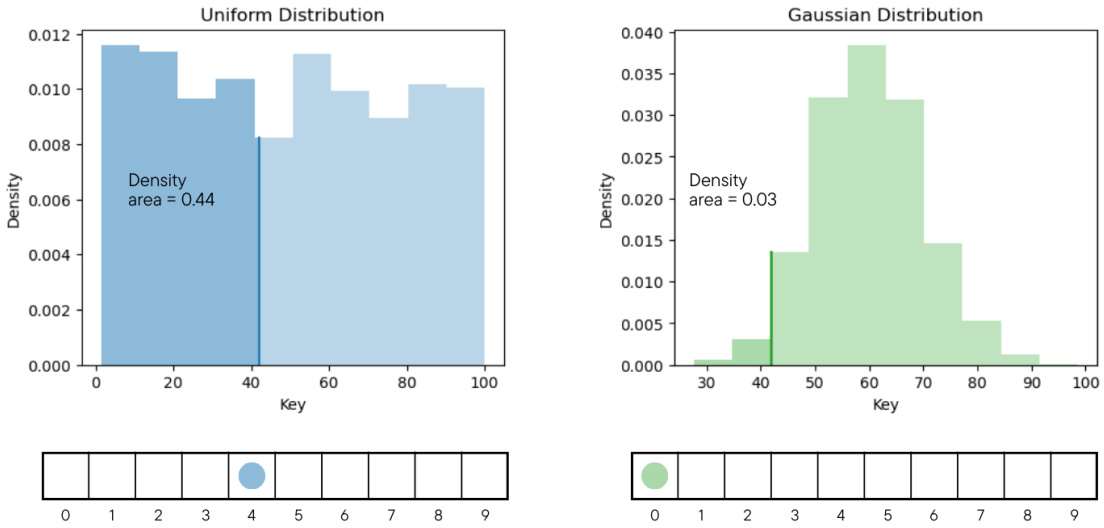


Figure 1.1: Comparison of guessed positions when the key distribution is Uniform (left) or Gaussian (right).

Figure 1.1 displays how our educated guess changes when we change the distribution. If we know that keys were drawn from a Uniform distribution, then key 42 will likely be in position 4. However, if they were drawn from a Gaussian distribution ($\mu = 60, \sigma^2 = 100$), then key 42 is expected to be in position 0.

More formally, what we have done up to now is use the Cumulative Distribution Function (CDF) of keys to predict a position in our list:

$$h(key) = \lfloor CDF(key) \cdot N \rfloor \tag{1.1}$$

where N is the number of keys.

Unfortunately, in many real-world scenarios, it's impossible to know how the keys were generated, and what their actual CDF is. However, we can still try to *learn* it!

1.1.1 Learned Indices – Formal Definition

A *learned index* [1] is an innovative approach to indexing that leverages the computation and learning of the empirical CDF of keys. This learned CDF is then utilized to predict the position of a key within a list, using Equation 1.1.

Such a model allows the building of a tailored index, that adapts to the unique distribution of keys. Given its flexibility, with Machine Learning as its foundation, the index became very popular in the scientific community, finding many different applications (e.g., [2, 3, 4]).

However, learned indices have an important limitation: whenever a new key is inserted, the underlying model must be retrained, and the entire index must be rebuilt from scratch. This makes learned indices unsuitable for applications where keys are frequently updated—e.g., in an Online Transaction Processing (OLTP) database.

Contrarily, more static use cases will possibly benefit from the flexibility of a learned index, offering efficient query performance without the need for frequent retraining or complete index rebuilding.

1.1.2 The Recursive Model Index

The *Recursive Model Index (RMI)* [1, 5] is a hierarchical learned index that utilizes multiple layers of staged models. It was the first learned index to be introduced in the literature, and it remains the most popular one to this day.

The RMI consists of a root model, some intermediate layers, and a leaf layer. Each layer is responsible for predicting the model to be used in the subsequent layer. The leaf model ultimately returns the estimated position in the index, along with the upper bound of the error. The hierarchical structure of the index is essential for accurately representing the empirical CDF of keys, even at a micro-level.

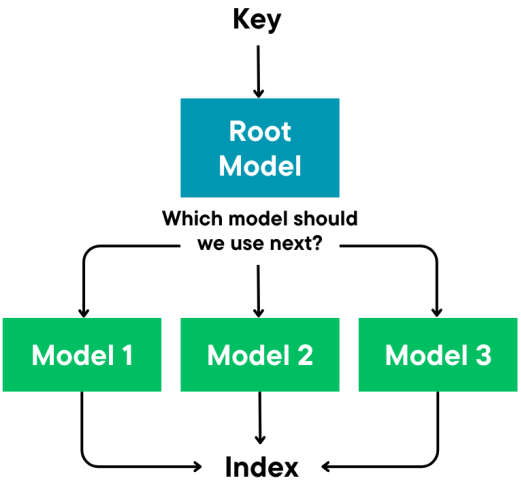


Figure 1.2: Example of two-stage RMI model.

When defining an RMI, several choices need to be made:

- The number of stages, which should be a minimum of two (root and leaf).
- The number of models for each stage, also known as the *width* of the stage.
- The type of model for each stage, such as linear, spline, or cubic.

Figure 1.2 illustrates a simple example of a two-stage RMI model. In practice, two layers are often sufficient to achieve good performance, as long as the second layer is wide enough [5, 6].

Width and performance. Intuitively, a wider leaf stage in an RMI corresponds to higher accuracy in predicting the position, but it also incurs a higher—and perhaps hidden—computational cost.

Querying a two-layer RMI has a fixed cost of $\mathcal{O}(1)$, as predicting the position only requires two mathematical operations, one for each layer. For example, given a root model r and a leaf model l_i, both linear, the final position h in the list for a key k is given by:

$$i \leftarrow b_r + m_r \cdot k \tag{1.2a}$$

$$h \leftarrow b_{li} + m_{li} \cdot k \tag{1.2b}$$

where (b_r, m_r) and (b_{li}, m_{li}) are the root and leaf model parameters, respectively.

However, the no-free-lunch fans out there will heave a sigh of relief as soon as they notice these model parameters must be stored somewhere. And the more they are, the more difficult is to keep them nearby...

The issue is, indeed, *cache* related. As the number of leaf models increases, so does the number of RMI parameters. If the parameter size surpasses the cache capacity, the overhead of continuously fetching parameters from memory grows exponentially.

For this reason, it's important to carefully select the width of the leaf stage, finding the soft spot that maximizes accuracy without adversely affecting the cache.

1.1.3 RadixSpline and PGM

While the RMI might be the most popular learned index, it's not the only one. We will now focus on two variants: RadixSpline and PGM.

RadixSpline Index. *RadixSpline* indices [7] approximate the CDF using a linear spline[1]. Spline points are then indexed using a compressed *radix* lookup table.

Similarly to RMI and width, RadixSpline also has a hyperparameter: the number of radix bits r, which will in turn set the size of the lookup table to 2^r.

Lookup on a RadixSpline index consists of three phases, as shown in Figure 1.3:

1. Use the r most significant bits of the key (let's call this value b) to query the lookup table. More specifically, we extract the spline points returned by entry b and entry $b + 1$ of the table.

2. Perform a bounded binary search between those points, to identify the two spline points directly surrounding the key.

3. Compute the linear interpolation between the two spline points to estimate the position.

The need to search across spline points makes RadixSpline slightly slower than RMI, which instead doesn't require any search in inner nodes.

Piece-wise Geometric Model Index. Similarly to RMI, *Piece-wise Geometric Model (PGM)* indices [8] are characterized by a multi-level structure.

More precisely, the leaf layer is a Piece-wise Linear Approximation model (PLA-model), which can predict the *approximate* position of a key up to a fixed, parametric error ϵ. This ϵ-approximate position can be turned into an *exact* position using a bounded binary search, costing $\mathcal{O}(\log_2(\epsilon))$.

The rest of the index is built using a bottom-up strategy: for each layer, we take the optimal PLA-model and select a subset of keys. Such keys are then used

[1] A *linear spline* is a function that is piece-wise linear.

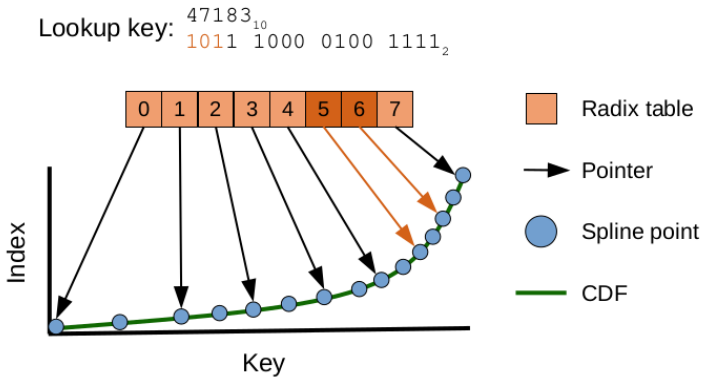


Figure 1.3: A radix spline index and example lookup process, with $r = 3$. Taken from [7].

to build a new PLA-model over them. This process recursively continues until one single key remains, which forms the root of the PGM index.

The inference phase of a PGM index resembles that of the RMI. However, it tends to incur slightly higher costs, as the expense for each layer is logarithmic (as opposed to the constant time in RMI), and PGM typically consists of more than two layers.

1.2 Hash Tables Building Blocks

Yes, learned indices are fascinating structures. But let's step back and focus on indices in general.

Indices are perfect for keeping data sorted and are efficient for locating specific elements and searching for a range of values. However, as datasets grow larger and more complex, the need for faster and more scalable methods becomes apparent. This is where hash tables come in.

Hash tables are composed of a fixed number of slots and use a hash function to determine the placement of each key into a specific slot. In an ideal scenario, keys can be retrieved from the hash table with a constant time complexity of $\mathcal{O}(1)$, making them a powerful alternative to indices.

As we consider these established data structures, we can't help but wonder... Can we do more? Can we explore the possibilities of *learned functions*?

1.2.1 Traditional Hash Functions

A traditional hash function maps keys to independently and identically distributed (i.i.d.) random outputs, belonging to a known range.

Due to the random nature of this assignment, it's common for different keys to be mapped to the same slot, a phenomenon known as a *collision*. The amount of collisions in a hash table determines its quality: the larger the collision number, the further the table is from the ideal $\mathcal{O}(1)$ complexity.

There is an almost unlimited selection of state-of-the-art hash functions available, which can effectively mimic true randomness in many scenarios.

Here, we briefly describe five of them.

Multiplicative hashing (MultiplyPrime). *Multiplicative* hash functions are a family of simple functions introduced in [9]. If the hash table has M slots, then the multiplicative hash function is:

$$h(k) = \left\lfloor M \left(\left(\frac{A}{2^w} k \right) \bmod 1 \right) \right\rfloor \tag{1.3}$$

where A is a constant and w is the machine word size.

Fibonacci hashing (FibonacciPrime). The *Fibonacci* hash function is a variant of MultiplyPrime where we set the constant A to $2^w/\Phi$, where Φ is the *Golden Ratio*.

We choose Φ for its distinctive ability to evenly subdivide any range, without ever looping back to the starting position [10].

Murmur hashing. *Murmur* hash functions emulate true randomness by employing repeated multiply and rotate instructions. The implementation involves three XOR operations, three shifts, and two multiplications [11].

xxHash. *xxHash* is a hash algorithm designed for rapid processing of large data blobs. It's very efficient, even reaching the limits of RAM speed [12].

AquaHash. *AquaHash* is a uniform hash function that harnesses Advanced Encryption Standard (AES) primitives, leveraging their hardware-level implementation on numerous CPUs [13].

1.2.2 Perfect Hash Functions

As we already noted, a drawback of traditional hash functions is their tendency to produce collisions, which can degrade the performance of hash tables.

Perfect hash functions address this issue by constructing *injective* functions. In simpler terms, when provided with an input set of keys, the function ensures that no collisions occur.

The implementation of perfect hash functions typically involves a one-time $\mathcal{O}(n)$ build algorithm, which must be executed before they can be used. This process necessitates prior knowledge of all possible keys, and attempting to query the function with a non-key results in arbitrary outcomes.

Given their additional complexity and more restrictive use cases, perfect hash functions are less used than traditional hash functions. Here, we will consider two of the most popular ones: MWHC and Recursive Splitting.

MWHC. *MWHC* [14] is a family of algorithms for generating perfect hash functions, characterized by $\mathcal{O}(n)$ build and $\mathcal{O}(1)$ query time.

Construction requires two steps. First, a special hash function is computed into an r-graph[2], which should be acyclic. This property is achieved probabilistically. Then, the function is refined deterministically to a perfect hash function.

If the number of keys is sufficiently large, choosing $r = 3$ as graph parameter yields optimal speed and minimizes space usage.

A simplified variant of MWHC is also available, requiring only 3 bits of storage per key. We will refer to this version as *BitMWHC* [6].

Recursive Splitting. As suggested by its name, *Recursive Splitting (RecSplit)* [15] works by partitioning the input set into smaller buckets, continuing this process until it becomes feasible to employ a brute-force search for a bijective function.

As MWHC, it accomplishes the expected complexities of $\mathcal{O}(n)$ for construction and $\mathcal{O}(1)$ for lookups, delivering state-of-the-art results.

1.2.3 Collision Resolution Strategies

Up to this point, we've discussed various state-of-the-art options for different hash functions. However, a hash table is more than its hash function: it also needs a collision management strategy (also known as hash *scheme*).

When collisions occur in a hash table, different resolutions are available, directly affecting the hash table's "physical" implementation.

We will now explore the most commonly used hash schemes, with a focus on how collisions are managed and their impact on the behavior and characteristics of the hash table.

[2]In mathematics, an r-graph is a generalization of a graph, in which each edge can join r vertices. In contrast, in an ordinary graph, an edge connects exactly two vertices.

Bucket chaining. In bucket chaining, the hash table is implemented as an array of linked lists. Each list element is called a *bucket*, and it can contain multiple key-value pairs.

In the insertion process, the key is hashed and then added to the appropriate list's bucket. When the bucket is full, a new one is created and appended to the list to accommodate additional entries. Lookup simply consists of searching for the key in the corresponding linked list.

Bucket chaining is a simple yet flexible hashing scheme, designed to handle an unlimited number of entries without requiring any reallocation. However, searching for a specific key in a—possibly lengthy—linked list can become an intensive operation, especially if we consider that different buckets might be allocated at scattered memory addresses. This leads to increased cache accesses and potential performance drawbacks.

Linear probing. In linear probing, the hash table simply consists of an array of slots. If a collision occurs when inserting a new entry, the algorithm searches for the next available slot in the hash table by linearly probing through the array until an empty slot is found. Similarly, lookup requires executing the linear probe algorithm starting from the predicted slot, and until either the key is found, an empty entry is encountered, or the entire array has been traversed.

Contrarily to bucket chaining, linear probing is a cache-friendly scheme, thanks to the linear scan operation. However, its performance quickly degrades when large continuous blocks of slots are occupied. This issue can be mitigated by keeping the table's load factor[3] always below 60% [16].

Cuckoo hashing. Cuckoo hashing uses multiple hash functions on the same hash table, which is again a simple array of slots. Insertion is based on a series of eviction rounds. For each function, the hash value is computed and the correspondent table slot is checked, until an empty slot is found. If all slots are already full, a victim entry is evicted and then re-inserted, until a valid configuration is found or a maximum number of iterations is reached.

The most common version of cuckoo hashing simply uses two hash functions. In this setup, querying the table involves checking only two slots, leading to a maximum of two cache misses regardless of the load factor.

This property makes cuckoo hashing an ideal choice as the load factor increases. However, in such scenarios, the risk of failure during the building phase becomes elevated. To mitigate this risk, a "bucketized" variant of cuckoo hashing can be adopted, allowing the storage of multiple entries per slot.

[3]The *load factor* of a hash table is the percentage of its occupied slots.

Figure 1.4: An example of insertion of key 2 in a hash table with $h(k) = k \bmod 6$. (a) In the bucket chaining table, the key is appended to the linked list in position $h = 2$. (b) In the linear probing table, the key is inserted in the first free slot, starting to scan from position $h = 2$.

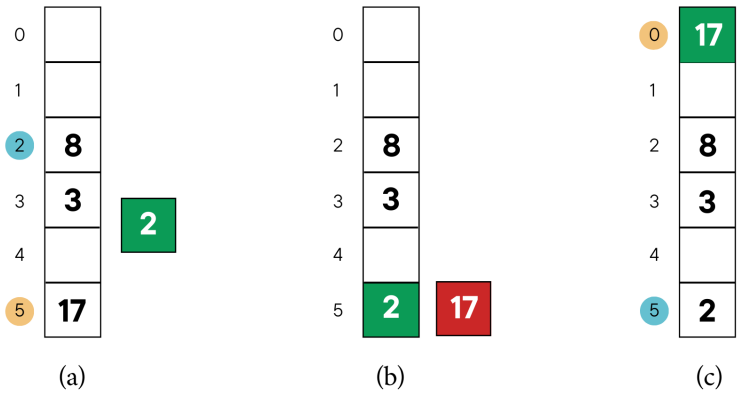


Figure 1.5: An example of insertion of key 2 in a cuckoo hash table using $h_1(k) = k \bmod 6$ and

$$h_2(k) = \begin{cases} (3+k) \bmod 6 & \text{if } (k \bmod 6) < 3 \\ (5-k) \bmod 6 & \text{otherwise.} \end{cases}$$

(a) The hash values for key 2 are $h_1 = 2$ and $h_2 = 5$, but both positions are already taken. (b) Key 17 is randomly selected as victim and ejected. (c) Key 17 is re-inserted at position $h_2 = 0$.

Chapter 2

The Birth of Learned Functions

2.1 "Can Learned Models Replace Hash Functions?"

Now that all puzzle pieces have found their place, it's finally time to talk about learned functions.

The first and most comprehensive study conducted on this subject is the article *Can Learned Models Replace Hash Functions?* [6], which benchmarks learned functions against their traditional and perfect counterparts.

The authors begin by conducting a mathematical analysis of the factors influencing collisions. This analysis aims to identify scenarios in which learned functions can exhibit fewer collisions than traditional hash functions.

Subsequently, they run and analyze a series of benchmarks, comparing all the hash functions outlined in Chapter 1 with a selection of learned functions. These functions are evaluated both individually and when inserted into a hash table using a given collision resolution strategy.

2.1.1 Our Contribution

While the relevance of the conducted experiments is undeniable, the study has several issues. It's occasionally vague and imprecise, but most importantly it lacks a usable codebase to reproduce the obtained results.

We aim to remedy that. First, we will provide a clearer explanation of the collision analysis and its implications for both traditional and learned functions.

Next, we will employ our tailor-made framework[1] to replicate all the experiments outlined in the paper. Our framework is designed to be extremely versatile, ensuring flexibility through parameter customization and consistency across different machines. It also includes utility scripts for downloading datasets, managing output files, and generating all the plots we will present in the following chapters of Part I.

Finally, expanding on the paper's results, we will consider parallel implementations of the tables, and conduct an in-depth analysis of the impact of cache size on these structures.

2.2 From Learned Indices to Learned Functions

Let's consider again how a generic learned index works.

Given a list of N sorted elements, the index predicts the position of a key k as:

$$h(k) = \lfloor CDF(k) \cdot N \rfloor \tag{2.1}$$

It also provides an upper bound ϵ for the error, ensuring that the actual position of k lies in the interval $[h(k) - \epsilon, h(k) + \epsilon]$. At this point, we can locate the actual position by running a bounded binary search, for an overall computational cost of $\mathcal{O}(\log_2(\epsilon))$.

Now, let's forget about errors and searches, and focus only on Equation 2.1. At the end of the day, this is a function mapping a key to a position in the range $[0, N)$. Sounds familiar, doesn't it? This is exactly what a hash function does!

2.2.1 Learned Functions – Formal Definition

Given a hash table with M slots, we define a *learned hash function* on top of it as:

$$h(k) = \lfloor CDF(k) \cdot M \rfloor \tag{2.2}$$

Similar to perfect functions, learned functions require knowing the set of keys in advance, and they must be trained on such a set before being used. In practice, this consists of building the learned index of our choice, as described in Section 1.1. This procedure is simpler than the construction algorithm of perfect functions, and more scalable in terms of memory occupation.

On the other hand, when compared to traditional hash functions, learned functions are more complex and slower to compute. However, their data-specific approach could reduce the amount of collisions in the table, and enhance the overall efficiency.

[1] github.com/IlariaPilo/new-hashing-benchmark

2.3 Understanding Collisions

2.3.1 Notation

For the sake of simplicity, we consider the task of mapping N keys to N locations (i.e., the load factor is equal to one). This task can be easily generalized, since if the number of locations increases, the amount of collisions will decrease.

We apply a given hash function f on some keys—we call $k_0, k_1, ..., k_{N-1}$ the list of sorted keys, and $h_0, h_1, ..., h_{N-1}$ the list of sorted hashes. For convenience, we let $h_{-1} = 0$. Notice that, since f is not necessarily monotonic (consider, for example, modulo-based hash functions), there is no guarantee that $h_i = f(k_i)$.

Starting from the sorted hashes, we can define a list of gaps $g_0, g_1, ..., g_{N-1}$ such that $g_i = h_i - h_{i-1}$. We assume that gaps are i.i.d., with probability density function (pdf) $f_G(z)$ and CDF $F_G(z)$.

2.3.2 Collisions – Causes

By definition, a collision occurs when two different keys are mapped to the same hash. Such a concept can be expressed by leveraging the definition of gaps: if the gap between two consecutive sorted hashes is ≥ 1, then the corresponding keys will be surely placed in different positions. Contrarily, if the gap is < 1, the corresponding keys might collide (if the hashes share the same closest integer). An example is shown in Figure 2.1.

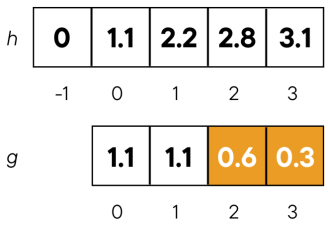


Figure 2.1: An example of sorted hashes h and their respective gaps g. Since g_0 and g_1 are both ≥ 1, we can immediately deduce the corresponding keys will be mapped to different positions, while further investigation is needed for g_2 and g_3. In this example, h_2 requires a minimum gap of 0.8 to be placed in a position different from h_1; since $g_2 = 0.6$, the requirement is not satisfied and a collision occurs. Contrarily, h_3 only needs a minimum gap of 0.2; since $g_3 = 0.3$, the corresponding keys don't collide. In general, the minimal gap when $g_i < 1$ can be computed as $1 - \{h_{i-1}\}$, where $\{h_{i-1}\}$ is the fractional part of h_{i-1}.

Since we want to avoid collisions, we would like all the gaps to be ≥ 1. Now, the sum of gaps is equal to:

$$\sum_{i=0}^{N-1} g_i = \sum_{i=0}^{N-1} (h_i - h_{i-1}) = h_{N-1} - h_0 \tag{2.3}$$

which is always lower than the number of positions N.[2] As a consequence, the ideal gap distribution to have no collisions would be the one where $g_i = 1 \, \forall i$.

Assuming that f is not a lattice distribution[3], the expected number of colliding keys $\mathbb{E}[c]$ can be estimated as follows.

Lemma. *As N grows large, $\mathbb{E}[c]$ converges to*

$$N \left(1 - \int_{u=0}^{1} \int_{t=1-u}^{\infty} f_G(t) \cdot \Big[1 - F_G(1 - \{t+u\})\Big] \, dt \, du\right) \tag{2.4}$$

where $\{x\}$ is the fractional part of x (i.e., $\{x\} = x - \lfloor x \rfloor$).

Proof. We begin by defining the probability that h_i is in a different location from h_{i-1}. As we pointed out above, this is given by two terms:

1. The probability that $g_i \geq 1$, OR
2. The probability that $g_i < 1$ AND the hashes are close to two different integers. More specifically, this term can be expressed as $(1 - \{h_{i-1}\}) \leq g_i < 1$, as explained in Figure 2.1.

Term 1 imposes the following lower bound on the probability:

$$\int_{t=1}^{\infty} f_G(t) \, dt = 1 - F_G(1) \tag{2.5}$$

For sufficiently large N, $\{h_{i-1}\}$ is known to converge to the Uniform distribution on $[0, 1]$ [17]. As a consequence, starting from the lower bound 2.5 and adding Term 2 conditions, we can express the probability that h_i is in a different location from h_{i-1} as:

$$\int_{u=0}^{1} \int_{t=1-u}^{\infty} f_G(t) \, dt \, du = 1 - \int_{u=0}^{1} F_G(1-u) \, du \tag{2.6}$$

where the external integral represents the sum over all possible values of $u = \{h_{i-1}\}$.

Now, the probability that h_i is alone in its slot is given by two terms:

---

[2] By definition, $h_i \in [0, N) \, \forall i$, meaning that $(h_{N-1} - h_0) < N$.

[3] A *lattice distribution* is a discrete probability distribution concentrated on a set of points of the form $a + nh$, where $h > 0$, a is a real number (called the *step*) and $n = 0, \pm 1, \pm 2, ...$

1. The probability that h_i is in a different location from h_{i-1} (which we just defined), AND

2. The probability that h_i is in a different location from h_{i+1}.

Again, Term 2 depends on $\{h_i\}$, which can be expressed as:

$$\begin{cases} u = \{h_{i-1}\} \\ t = g_i = (h_i - h_{i-1}) \end{cases} \quad \rightarrow \quad \{h_i\} = \{t + u\} \tag{2.7}$$

Taking this into consideration, the probability that h_i is alone in its slot becomes:

$$\int_{u=0}^{1} \int_{t=1-u}^{\infty} f_G(t) \cdot \Big[1 - F_G(1 - \{t + u\})\Big] \, dt \, du \tag{2.8}$$

Since we are interested in the expected number of colliding keys, we simply consider the complement probability and multiply by N. We finally obtain Lemma 2.4. □

2.3.3 Collisions – Effects

Collisions for traditional hash functions. As we already know, truly random hash functions output values that are uniformly distributed in the interval $[0, N)$. Therefore, the gap distribution can be approximated by the exponential distribution with mean 1, regardless of the input distribution.

In this case, the expected number of colliding keys will be:

$$\mathbb{E}[c] = (1 - e^{-1}) \cdot N \approx 0.632 \cdot N \tag{2.9}$$

Collisions for learned hash functions. For simplicity, we begin by considering a *single* linear model to approximate the CDF. Given the sorted input keys $k_0, k_1, ..., k_{N-1}$, the resulting hash function will be:

$$h(k) = m \cdot (k - k_0) \, , \qquad m = \frac{N - 1}{k_{N-1} - k_0} \tag{2.10}$$

Let's now consider the value of a generic gap g_i. This is going to be equal to:

$$g_i = h_i - h_{i-1} = m \cdot (k_i - k_{i-1}) \tag{2.11}$$

meaning that the hash gaps are essentially equal to the key gaps scaled by m.

As a consequence, if the input keys are evenly spaced, then the hashes will also be evenly spaced, thus resulting in few collisions. On the other hand, if the key gaps have a high variance, the amount of collisions will increase.

In piece-wise learned functions, this scaling occurs at the submodel level; however, the overall behavior remains the same.

Consequently, the main distinction between learned indices and learned functions lies in the impact of the submodel number. Increasing the number of submodels consistently enhances the CDF approximation, leading to improved accuracy in learned indices. This is not necessarily true in learned functions. Due to the significant influence the gap variance has on collisions, we anticipate the collision rate to stabilize after reaching a certain submodel number, below which we underfit.

2.3.4 To Each Their Own

This collision analysis highlights that the decision to use learned functions or not depends on the nature of the keys, and there's no one-size-fits-all answer.

For instance, in scenarios where keys are incremental, auto-generated values, potentially with deletions and noise, their gaps exhibit high regularity. In such cases, a learned function is likely to generate fewer collisions than a traditional one.

Conversely, in datasets characterized by irregular key gaps and complex patterns, learned functions may perform as—if not worse than—traditional hash functions.

Chapter 3

Classic Hash Experiments

In this chapter, we will revisit the experiments detailed in [6], which we implemented using our framework. We will present the results and offer our interpretations, drawing comparisons with the original article when needed.

3.1 General Setup

3.1.1 Datasets

The experiments are conducted on six different datasets, both synthetic and real, each with 100 million unique keys. Keys are unsigned integers, represented on 64 bits.

We will not consider string keys, as they rely on ad-hoc hash functions, which are typically more intricate and less efficient compared to integer hash functions, requiring a separate and focused study.

Synthetic datasets. To generate synthetic keys, we consider three different processes:

1. *gap_10* – We initially generate keys at regular, sequential intervals of 10. Then, 10% of these keys are uniformly randomly deleted. This dataset is interesting as it simulates the case of auto-generated database IDs, after the removal of some entries.
2. *uniform* – Contains keys which are uniformly randomly sampled in the interval $[0, 2^{50}]$.
3. *normal* – Contains keys which are first generated from a normal distribution having $\mu = 100$ and $\sigma = 20$, and then linearly scaled to the range $[0, 2^{50}]$.

Real datasets. For real keys, we consider three datasets from the Search on Sorted Data (SOSD) benchmark [18], which is used to compare learned index structures against classic indices:

1. *fb* – Contains randomly sampled Facebook user IDs.
2. *wiki* – Contains timestamps of edits from Wikipedia.
3. *osm* – Contains cell IDs from Open Street Map.

3.1.2 Hardware

The experiments are run on a machine with the following characteristics:

- CPU – Intel(R) Core(TM) i9-10920X CPU @ 3.50GHz
- Threads – 24 [1 socket, 12 cores, 2 threads per core]
- L3 size – 20.2 MB
- RAM size – 128 GB
- Architecture – x86_64

We use a Docker container to enforce maximum portability of the framework. The OS is Ubuntu 23.10. Binaries are compiled with clang++, using C++20 and optimization level 3.

3.2 Hash Function Experiments

In this section, we focus on hash function experiments, which analyze function-specific characteristics without considering any particular hash table structure.

For the sake of clarity, in the plots, we will represent traditional functions with diamond marks (◆), perfect functions with circle marks (●), and learned functions with square marks (■).

3.2.1 Build Time Analysis

Contrary to traditional hash functions, both perfect and learned functions require an initial building process. However, since learned functions are less complex than their perfect counterparts, we expect their construction to be faster.

Figure 3.1 illustrates the relationship between build time and dataset size. Notably, MWHC consistently exhibits a build time that is two orders of magnitude slower than the learned functions one.

It's worth mentioning an anomaly in the results with respect to the original paper, where the behavior of PGM is expected to align with RMI. This deviation

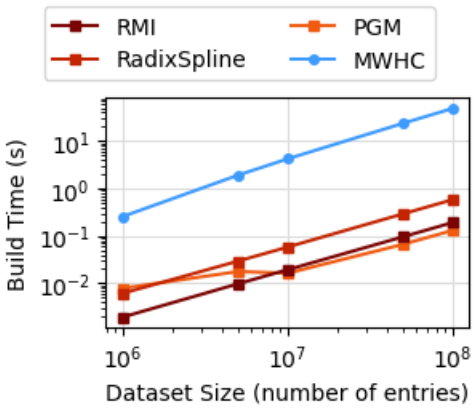


Figure 3.1: Effect of dataset size on function building time.

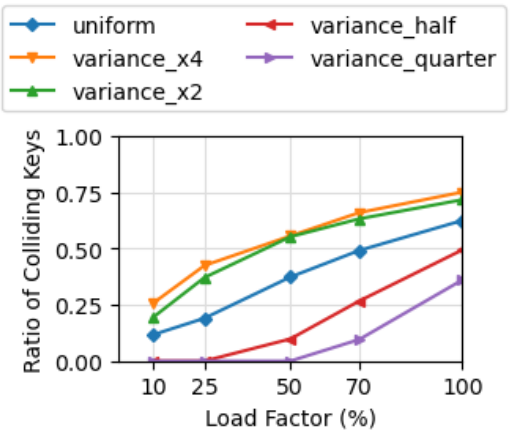


Figure 3.2: Effect of key gap distribution on learned hash function collisions.

may be attributed to the use of a multi-thread implementation of PGM, where the thread number depends on the dataset size.

3.2.2 Gap Distribution Analysis

As we proved in Section 2.3, collisions in learned hash functions are strongly correlated with the variance of the key gap distribution.

To experimentally validate this claim, we consider many variations of the *uniform* dataset. Each variation is characterized by a gap distribution with a different variance. We then train an RMI hash function with 100 submodels on each of these dataset variations.

Figure 3.2 displays how the behavior of these datasets aligns with our expectations. Variations with higher variances (*variance_x2* and *variance_x4*), exhibit higher collision rates, while those with lower variances (*variance_half* and *variance_quarter*) show fewer collisions. This empirical evidence supports the correlation between gap distribution variance and collision rates in learned hash functions.

3.2.3 Collisions Analysis

Here, we focus on what is perhaps the most critical property of hash functions: their throughput/collision ratio tradeoff.

For each dataset, we shuffle its 100M keys and hash each of them using various functions, to obtain the collision ratio. During this process, we keep track of the cumulative time to compute the hashes, which we use to get the throughput. For

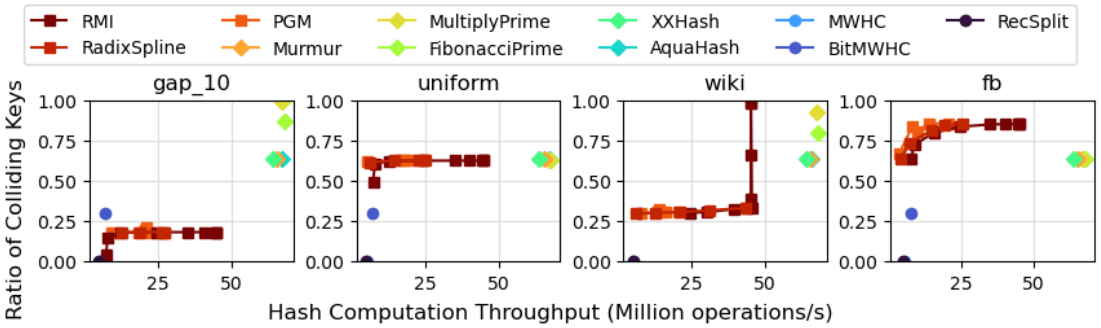


Figure 3.3: Computation throughput/collision ratio tradeoffs for various hash functions.

learned hash functions, we repeat the experiment varying the number of submodels. Figure 3.3 shows the obtained results[1].

Traditional functions exhibit collision ratios that closely match those of truly random hash functions (see Equation 2.9). As expected, they are also the fastest, primarily due to their cache-friendly nature and the absence of complex operations. However, it's worth noting that functions in the *Prime* group are less stable, displaying significantly higher collisions in *gap_10* and *wiki*.

Perfect functions perform as anticipated, with extremely low throughput and zero collisions, in line with their definition. BitMWHC—the compressed variant of MWHC—is slightly faster, in exchange for a consistent 30% collision rate.

As we previously noted, the effectiveness of learned functions heavily depends on the dataset. Specifically, we observe up to 3.5x lower collision ratios in datasets where gaps between entries are regular, such as *gap_10*. In contrast, they show high collision ratios in irregular datasets, where gaps between entries vary significantly (e.g., *fb*, with 4.7x greater collision ratios).

Notably, adding submodels to learned functions reduces the throughput, since the function no longer fits in cache. However, this addition has minimal effects on collision ratios. The most notable difference compared to the original article is that learned functions never manage to outperform traditional functions in terms of throughput. This divergence could potentially be attributed to hardware differences, particularly concerning cache size.

Proportion of collisions vs. RMI number of submodules. As discussed in Section 2.3, the complexity of the RMI model doesn't always correlate with

[1]Tradeoffs in *normal* are similar to *uniform*, and in *osm* they are similar to *fb*. For the sake of brevity, we will omit them.

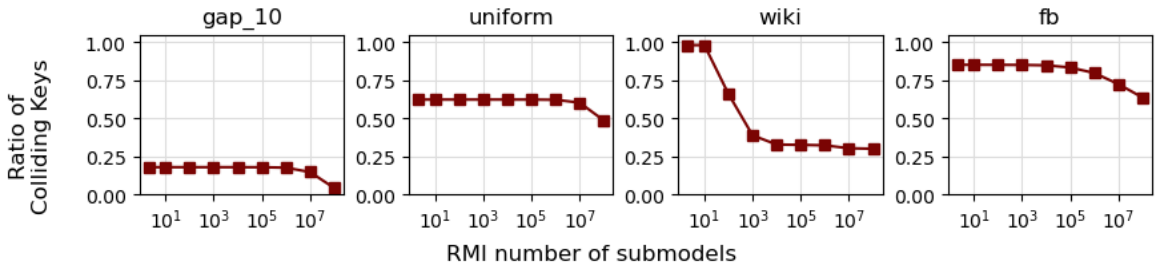


Figure 3.4: Proportion of collisions using RMI of increasing size.

the number of collisions. More specifically, we expect to observe an underfit and an overfit threshold for the submodel number, between which the collision ratio remains constant.

We consider the results of the previous experiment, and focus on the impact of the RMI submodel count. Figure 3.4 illustrates how both synthetic and real datasets align with our expectations.

Overall, to minimize collisions and simultaneously mitigate negative cache impact, the number of submodels should be chosen just above the underfit threshold.

3.3 Hash Table Experiments

In this section, we examine how hash functions behave in the context of different collision management architectures. For simplicity, we focus only on Murmur and MultiplyPrime as traditional functions, while MWHC represents the perfect function category.

Throughout the subsequent experiments, the number of submodels for learned functions is chosen to achieve the best tradeoff between throughput and collision ratio for each dataset.

3.3.1 Probe and Insert Throughput

In this experiment, we calculate the insert (Figure 3.6) and probe (Figure 3.5) throughput for different hash functions and collision management architectures at various load factors. For each combination, we start by inserting 100M entries into the hash table in random order. Subsequently, we probe 100M entries that are uniformly and randomly sampled.

Recall that a hash function's influence on the table throughput can be attributed to two key factors: the hash computation throughput and the number of collisions. The lower the number of collisions, the closer the table's performance is to the ideal time complexity of $\mathcal{O}(1)$.

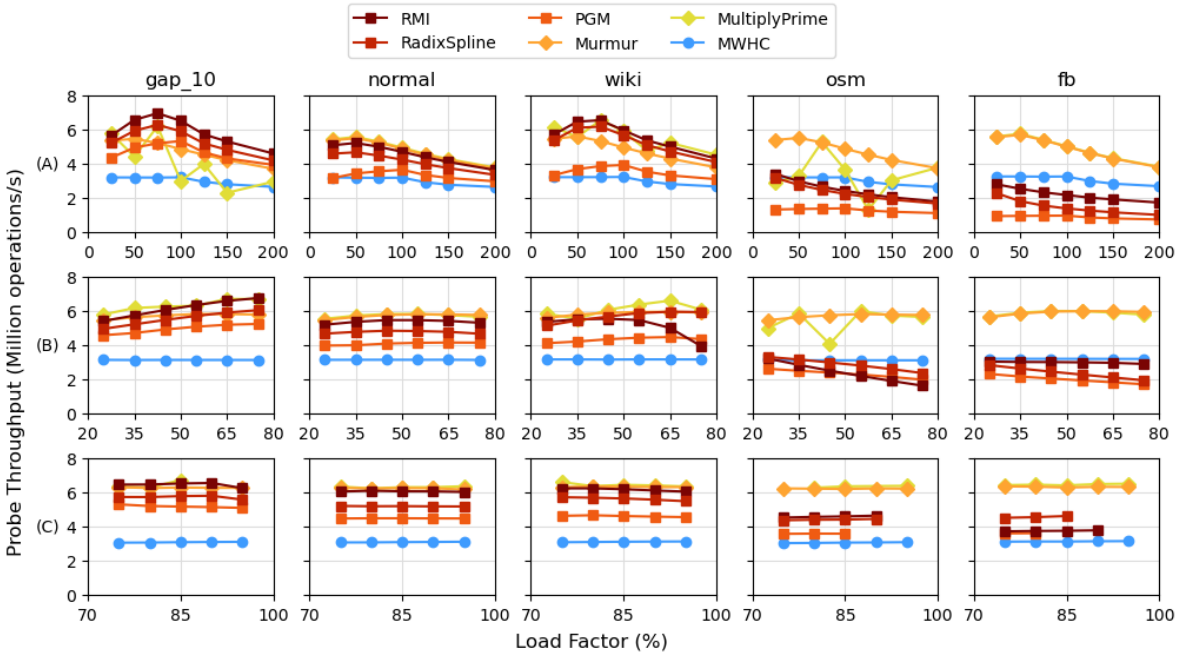


Figure 3.5: Effect of load factor on probe throughput for different hash functions and schemes: (A) bucket chaining, (B) linear probing, (C) cuckoo hashing.

Hash function comparison. In both insert and probe, given a resolution strategy, the behavior of traditional and perfect functions remains the same. This consistency aligns with the results of the collision experiment, as all datasets displayed similar throughput and collision percentages. For analogous reasons, perfect functions exhibit a constant behavior also with respect to the load factor. This is expected since, by definition, perfect functions have no collisions (as long as the load factor remains under 100%).

On the other hand, learned functions exhibit a more variable behavior. This variability is expected given their diverse collision ratios when the dataset changes. Specifically, they tend to outperform traditional hash functions in datasets where they experience significantly fewer collisions (*gap_10* and *wiki*), yielding up to 1.4x faster throughputs.

As observed in the build experiment, PGM sometimes exceeds expectations, possibly due to its multi-thread implementation. Finally, we notice again the unstable behavior of MultiplyPrime, whose performance varies greatly when changing both dataset and load factor.

Hash scheme comparison. Comparing different resolution strategies, we observe that there is no single scheme that is distinctly faster than the others. This

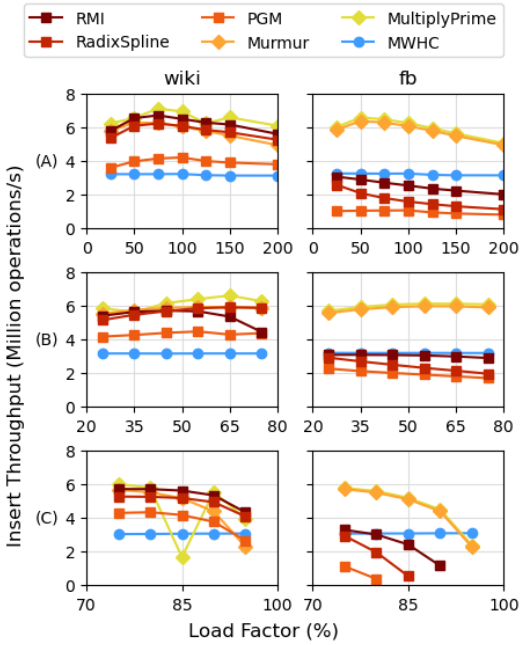


Figure 3.6: Effect of load factor on insert throughput for different hash functions and schemes: (A) bucket chaining, (B) linear probing, (C) cuckoo hashing.

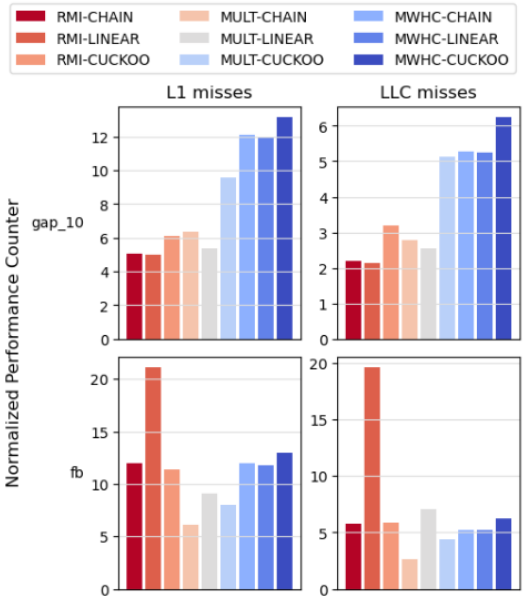


Figure 3.7: Normalized cache performance counters for the probe experiment.

contrasts with the results in the original article, where bucket chaining and linear probing were reported to be potentially twice as fast as cuckoo hashing.

Cuckoo hashing demonstrates a notably consistent probe throughput, in line with its definition (always involving two lookups, one in the main function slot and the other in the secondary one). On the other hand, its insert throughput can sometimes become quite low, as building the table may require numerous iterations of eviction and re-insertion.

Finally, when compared to the original paper, we observe that probe and insert throughputs have similar values, whereas we initially expected probe throughput to be nearly twice as fast. Again, this deviation could be attributed to differences in cache size.

3.3.2 Cache Performance Counters

In this experiment, we probe again the three kinds of tables using RMI, MultiplyPrime and MWHC, while keeping track of two cache performance counters: L1 cache misses and last-level cache (LLC) misses. The load factor is fixed at 80%. Results are displayed in Figure 3.7.

***gap_10* analysis.** For the *gap_10* dataset, we observed in the previous experiment that RMI-based tables achieve the highest throughput, followed by MultiplyPrime, and lastly, MWHC. These results are reflected by the performance counters.

As RMI utilizes only 10 submodels, they fit comfortably within the cache. Consequently, the majority of cache misses are attributed to collisions. Similarly, being a traditional modulo-based hash function, MultiplyPrime has cache misses predominantly stemming from collisions. The higher collision rate with respect to RMI correlates with an increase in cache misses.

On the other hand, MWHC variants are collision-free by definition, which means that all cache misses are a result of the r-graph traversal, rather than table searches.

When comparing different hash schemes, we note that bucket chaining and linear probing produce comparable results. On the other hand, the cuckoo hashing approach exhibits more cache misses. This observation aligns with the fact that, despite necessitating only two table lookups, the interested positions could be located far apart in memory, resulting in additional cache misses.

***fb* analysis.** For the *fb* dataset, we observed that MultiplyPrime tables are the best performers, followed by MWHC and RMI. Again, the cache miss findings are consistent with such results.

In this case, the RMI function employs 10M submodels—thus exceeding the cache capacity—and it has the highest number of collisions, resulting in significantly elevated cache miss counts. This effect is more pronounced in the linear probe table, where the need to scan very large clusters before locating the correct entry amplifies cache misses.

Surprisingly, in the case of MultiplyPrime, the linear probe variant underperforms, whereas the other two remain relatively constant. This phenomenon could be attributed to the differences in the distribution of colliding keys, even if the collision ratio remains the same. Again, scanning large clusters to find the correct entry in the table appeared to have a greater impact on performance.

Finally, MWHC variants exhibit behavior consistent with the observations from the *gap_10* dataset, largely owing to their collision-free property.

3.3.3 Range Queries

In these experiments, we focus on range queries. Notably, learned functions like RMI and RadixSpline are monotonic, in contrast to traditional hash functions like MultiplyPrime. This property allows us to use bucket chain tables to perform not only point but also range queries.

To assess the performance of such structures, we compare them with *RMI-Sort*, a simple RMI index built on top of the sorted, complete dataset. Such a structure

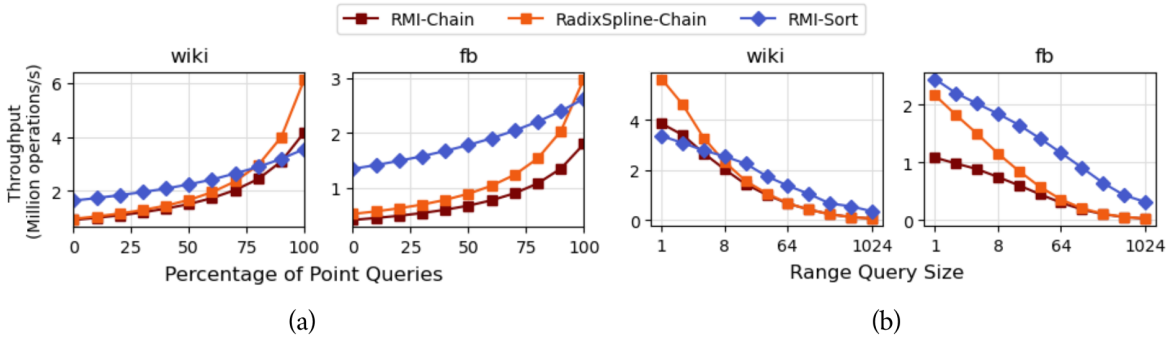


Figure 3.8: Effect of point query percentage (a) and range query size (b) on the probe throughput.

keeps track of the maximum negative and positive errors made by the RMI function and utilizes them to perform a bounded binary search.

Mixed workload throughput. We subject our three structures to a mixed workload, comprising both point and range queries, for a total of 100M queries. Results are displayed in Figure 3.8a. RMI-Sort manages to outperform both bucket chain structures until nearly all queries are point queries (specifically, 90% in the *wiki* dataset and 100% in the *fb* dataset).

This outcome aligns with expectations, as RMI-Sort only requires two bounded binary searches to find keys within the specified range. In contrast, the bucket chain table first determines the initial bucket using the function, and then linearly scans all buckets until a key greater than the range maximum is found. It's clear that this operation quickly becomes less efficient than two bounded binary searches, especially when there are many collisions and considering that different buckets can be widely separated in memory—thus leading to many cache misses.

Full range throughput. Now, we focus on a workload entirely composed of range queries. We vary the query size from 1 to 1024 and then perform 100M queries. Results are presented in Figure 3.8b. Once again, RMI-Sort outperforms the bucket chain hash tables in most scenarios, for the same reasons discussed previously.

Notably, it's intriguing to observe that the monotonic version of RMI performs less efficiently than RadixSpline, whereas, with the non-monotonic version, the trend was the opposite. This observation suggests that the additional checks required to make the function monotonic significantly slow it down.

Chapter 4

Multi-thread Table Access

As an original contribution to the article [6], we want to investigate the impact of enabling parallel table access for both insertion and probing on throughput values. We will conduct a comparative analysis with single-thread results, providing scalability assessments for hash tables and functions.

4.1 Parallel Table Implementation

For our implementation, we make the following assumptions:

1. The insertion and probe phases are distinctly separated in time: once we begin probing the table, new entries cannot be added concurrently.
2. The number of threads used will never exceed the number of physical threads on the machine.

From Assumption 1, we deduce that there is no need to manage concurrent access during the probing phase, since threads access the structure in a read-only mode, and insertions cannot occur simultaneously. However, this is not the case during the insertion phase, where we must protect the table from concurrent modifications.

To maximize parallelism, our approach consists of protecting each slot of the table with a lock, allowing multiple threads to access different slots simultaneously. To achieve this, we employ a spinlock structure, as it's smaller with respect to a mutex[1], and entails lower management costs. Furthermore, based on Assumption 2, there is no need to yield the CPU, making the mutex excessive.

After such modifications, when inserting into a table, a thread must acquire the following locks:

[1] A spinlock is approximately 10x smaller than a classic mutex.

- In bucket chaining, only one lock is acquired, as inserting in the table involves simply appending a new element to the slot's linked list.
- In linear probing, a lock is required for each iteration of the probing function, as each attempt involves reading and potentially writing to a different slot.
- In cuckoo hashing, two locks must be acquired simultaneously—one for the primary hash function slot and the other for the secondary one—for each iteration of the insertion algorithm. To prevent deadlocks, we enforce a specific acquisition order, giving priority to lower indices.

4.2 Parallel Table Experiments

4.2.1 Probe and Insert Throughput

We execute again the probe experiment, as described in Section 3.3.1, while allowing parallel insertion and probing. The experiment is run on 24 threads.

Per-function throughput analysis. We compare the throughputs per function with the sequential ones displayed in Figures 3.5 and 3.6. As anticipated, probe throughputs remain unchanged, as there is no lock acquisition phase, and elements continue to be probed in random order.

On the contrary, intriguing alterations were observed in the insertion behavior, as illustrated in Figure 4.1. Comparing these curves to the sequential insertion data in Figure 3.6, it becomes evident how the lock acquisition has a degrading effect on the per-thread performance.

As expected, cuckoo hashing proves to be the most affected scheme, given its necessity for simultaneous ownership of two locks over multiple iterations to insert a single entry.

Notably, linear probing appears to be less impacted than bucket chaining, despite the requirement for multiple lock acquisitions for a single insertion. This behavior is logically explained by the greater parallelism allowed by linear probing, in contrast to bucket chaining where linked list access is strictly sequential.

Overall throughput analysis. We compare the time required to complete 100M insertions and probes in a table. As we are employing 24 parallel threads, in the best-case scenario, we would expect the parallel execution time to be 24 times faster than the serial one.

The average ratios of parallel execution time to serial execution time can be found in Table 4.1. Notably, the probe ratios closely approach the optimal 24x improvement, given that no locking is involved. It's worth mentioning that bucket chaining with MultiplyPrime exhibits a ratio higher than 24 (24.4), which can be

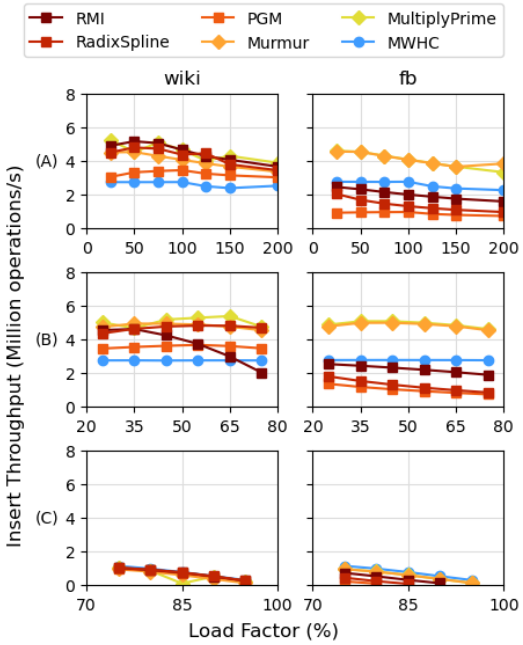


Figure 4.1: Effect of load factor on *parallel* insert throughput for different hash functions and schemes: (A) bucket chaining, (B) linear probing, (C) cuckoo hashing.

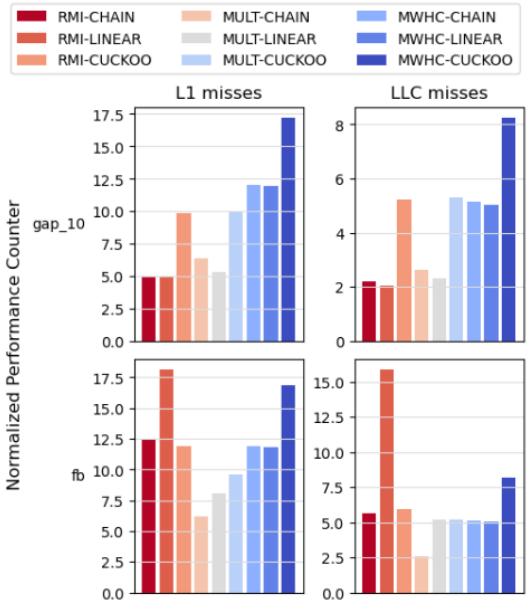


Figure 4.2: Normalized cache performance counters for the *parallel* probe experiment.

		RMI	RadixSpline	PGM	Murmur	MultPrime	MWHC
Probe	**(A)**	22.9	23.1	22.7	22.9	24.4	22.4
	(B)	21.8	21.6	21.0	23.1	23.6	22.5
	(C)	21.4	20.9	20.7	21.2	21.1	22.4
Insert	**(A)**	18.2	18.4	19.0	17.3	17.2	19.2
	(B)	17.3	16.1	16.0	19.0	19.2	20.1
	(C)	4.4	4.2	4.8	3.8	3.7	7.0

Table 4.1: Average serial/parallel ratios of probe and insert times, for different functions and schemes: (A) bucket chaining, (B) linear probing, (C) cuckoo hashing.

attributed to the previously observed unstable behavior of the function. On the other hand, the insert ratios are naturally lower than the optimal 24x due to the performance decrement introduced by the lock acquisition process. This decrement is particularly pronounced in the case of cuckoo hashing, where 24 threads only yield a 4x improvement.

4.2.2 Cache Performance Counters

We conclude the parallel table access analysis by repeating the performance counter analysis of Section 3.3.2. Results are presented in Figure 4.2.

***gap_10* analysis.** For the *gap_10* dataset, parallel counters closely resemble the sequential ones. The most important difference occurs in the values for the RMI and MWHC variants of the cuckoo scheme, which have increased.

As we already saw in the serial version of this experiment, cuckoo variants perform less effectively for this dataset—they may result in two misses per lookup, surpassing the other schemes[2]. When many lookups are executed in parallel, and especially when the cache is partially used to store frequently accessed function parameters (as in the RMI and MWHC cases), the cache pollution effect is intensified, leading to more cache misses.

***fb* analysis.** In the *fb* dataset, parallel counters are again similar to the serial ones, with one notable difference: the higher values for the RMI table with linear probing.

As in the *gap_10* case, the RMI linear probing table already had the most misses due to the large complexity of the function (10M submodels) and its lower performance, with significant clusters to be scanned before finding the correct entry. Once again, with multiple simultaneous lookups and large structures frequently accessed in different positions concurrently, the cache pollution effect amplifies, resulting in higher counters.

[2]Recall that *gap_10* has relatively few collisions, when compared to the other datasets in our study.

Chapter 5

Join – An Application

In this chapter, we will analyze a practical application for hash tables: non-partitioned hash join (NPJ) operations over two input relations.

5.1 Join Implementation

The flow of a classic NPJ routine is illustrated in Algorithm 1. To begin with, a hash table is constructed on the smallest input table. This hash table uses elements in the join column as keys, and elements in the projection columns as values. Subsequently, we iterate through the rows of the largest table, querying the hash structure to obtain join correspondences (when available).

Since we are dealing with a *non-partitioned join*, the hash table stores all the

Algorithm 1 Non-partitioned hash join

```
Input: small_table, large_table, join_col, proj_cols
Returns: result
hash_table ← build_hash_table(
    keys = small_table[join_col],
    values = small_table[proj_cols]
)
for each row in large_table do
    key ← row[join_col]
    small_values ← hash_table(key)
    if small_values.size() != 0 then
        result += (key, small_values, row[proj_cols])
    end if
end for
```

entries of the smallest input table, which makes it vulnerable to performance degradation due to its inability to fit entirely in cache[1]. Our investigation aims to determine whether learned functions can provide benefits in such a scenario.

Notably, the *online* construction of the hash structure adds a layer of complexity, as the speed of this process directly influences the join performance. This aspect raises concerns for learned functions, which require both a sorted dataset and a training phase.

5.1.1 Assumptions

Joins are complex operations, and writing an efficient implementation for a generic use case is notably challenging. Therefore, we opt for a slightly simplified approach, making the following assumptions:

1. *No duplicates in the smallest table* – We want the table on which we are building the hash structure not to have duplicates in the join column. This is necessary as the hash table implementation is not able to handle duplicate insertions.
2. *Always a "hit" in the largest table* – We expect the table querying the hash structure to consistently "hit" during probing. Although misses (join entries in a table with no match in the other one) are normal, allowing them introduces variability in timings and makes the benchmarks less reliable.

ID	CourseName
1	OS
2	Data Science
3	Computer Architectures

ID	StudentCode	CourseID
1	S123456	1
2	S424242	1
3	S594675	3
4	S314159	3

Figure 5.1: Illustration of a standard use case satisfying our assumptions: the smallest table is on the left and the largest on the right. The highlighted column represents the join column.

[1]On the other hand, the hash table for a *partitioned join* will store only a portion of the entries, making it more cache-friendly.

While these assumptions may seem stringent at first, they align with one of the most common use cases for joins, illustrated in Figure 5.1. In this scenario, the smallest table (left) uses its primary key column as the join column. This makes the join column duplicate-free by definition, satisfying Assumption 1. On the other hand, the largest table (right) features the join column as a foreign key. This way, we ensure that every entry in this column is always present in the smallest table due to database constraints, fulfilling Assumption 2.

5.2 Join Experiments

5.2.1 Database Generation

We want to run the join experiments starting from the same datasets we used up to now (more specifically, *wiki* and *fb*), to ensure fair comparisons across experiments.

Two pairs of tables are constructed from these datasets: (10Mx25M) and (25Mx25M)[2], where the numbers indicate the size of the small and large table, respectively. Each table contains only two columns: *join* and *value*. Each entry of *value* is set to a random 64-bit value. On the other hand, *join* contains real entries from the dataset, randomly drawn without (smallest) or with (largest) repetition.

The hash table load factor is fixed at 80%.

5.2.2 Single-thread Running Times

Figure 5.2 presents the running times for different hash functions and collision resolution strategies, emphasizing distinct phases of the NPJ. It's evident that both learned (i.e., RMI) and perfect (i.e., MWHC) hash functions are consistently outperformed by traditional functions (i.e., MultiplyPrime) in both the build and probe phases. As anticipated, RMI is significantly penalized by the sort phase, which alone consumes more time than the entire join operation when using MultiplyPrime.

These results appear to contradict findings in [6], where RMI-based hash tables were reported to be as performant as MultiplyPrime-based ones, if not better. However, upon closer examination, a critical factor was not considered in the authors' analysis: the sorting time, which proved to be highly significant.

On the other hand, our results align with the subsequent analysis presented in [19], demonstrating that the utilization of basic learned models in join operations such as NPJ does not yield performance benefits. Ad-hoc modifications to the models are necessary to obtain interesting improvements, as discussed in [19].

[2]We are interested in studying the corner case where both tables have the same size.

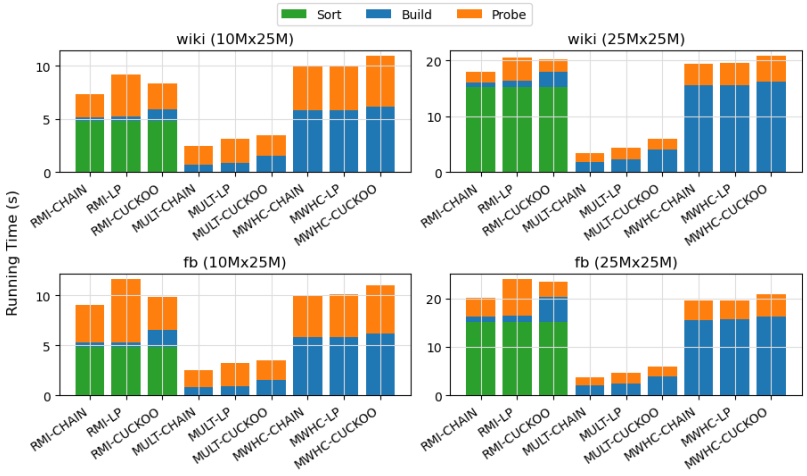


Figure 5.2: *Serial* running times for the three NPJ phases, using different hash functions and collision resolution strategies.

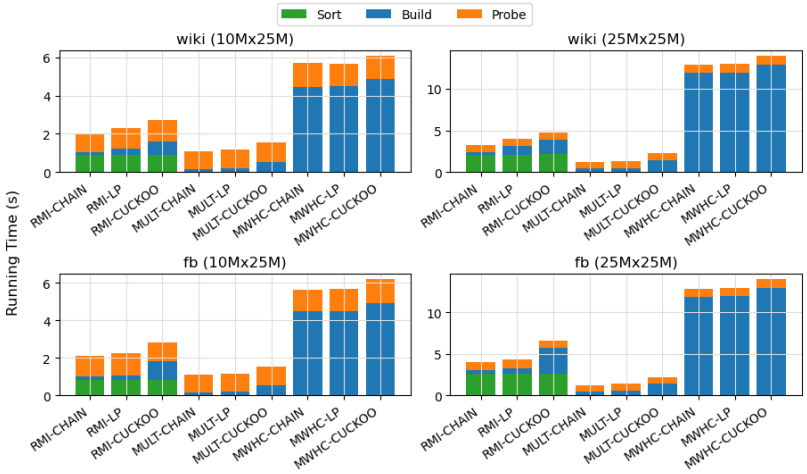


Figure 5.3: *Parallel* running times for the three NPJ phases. Experiments run on 24 threads.

5.2.3 Multi-thread Running Times

We leverage our implementation of multi-thread hash table structures to execute the NPJ experiment in parallel. More specifically, we parallelize both the build and the probe phases across 24 threads. To improve the sort phase, we use an implementation of the QuickSort algorithm based on OpenMP tasks [20].

As illustrated in Figure 5.3, the time gains achieved with 24 threads are especially important in the probe phase, followed by the sort, and finally by the build phase.

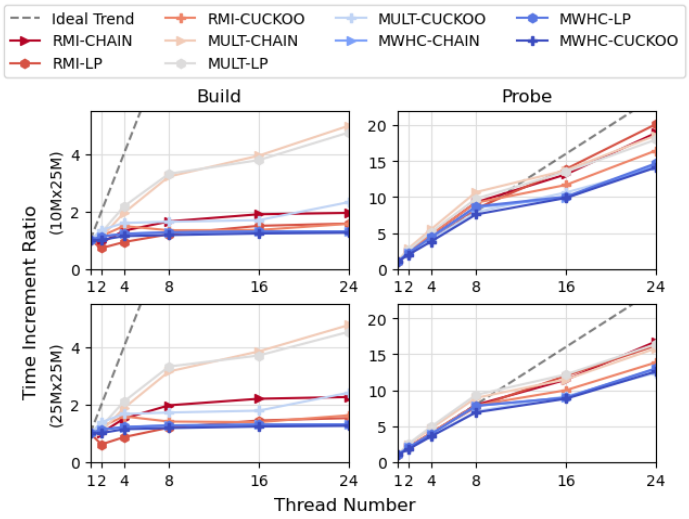


Figure 5.4: Time increment ratio trends when changing the number of threads, for both build and probe phases, averaged on different datasets, and over the (10Mx25M) and (25Mx25M) variants.

Time increment ratios. We analyze the time increment ratio trends when changing the number of threads, as displayed in Figure 5.4. It's important to note that we exclude the sort phase from this analysis since OpenMP tasks dictate thread usage, resulting in a consistent increment of around 7x.

Probe curves closely follow the ideal trend—sometimes even surpassing it. However, with an increase in the thread count beyond 8, we observe a slight decrement compared to the results obtained in the parallel insert experiment. This could be attributed to the fact that we are now working with 10-25% of the data, potentially worsening the cache pollution as the number of threads increases.

Conversely, build curves display more pronounced degradations attributable to locks. The reduction in table size—from over 200M to 13M/32M—elevates the likelihood of lock contention, leading to a more substantial performance decline.

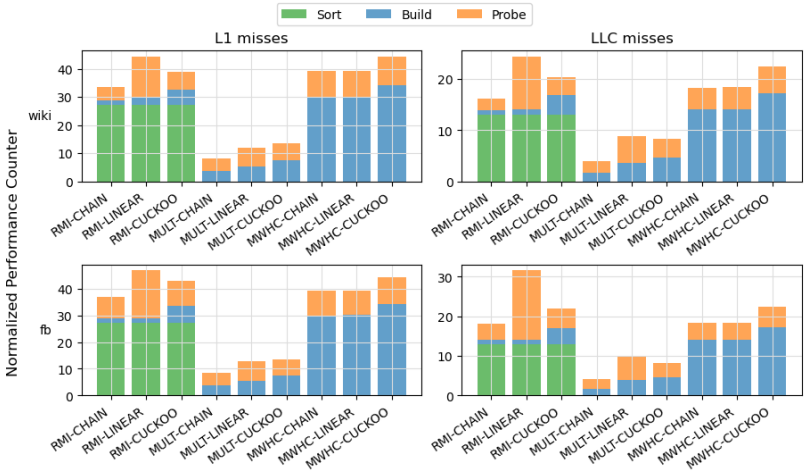
5.2.4 Cache Performance Counters

We conduct the same experiments while monitoring L1 cache misses and LLC misses. The normalized values for these counters in the serial and parallel experiments are shown in Figures 5.5a and 5.5b, respectively. In both cases, the majority of misses occur during the build phase or, in the case of RMI-based tables, in the sort phase.

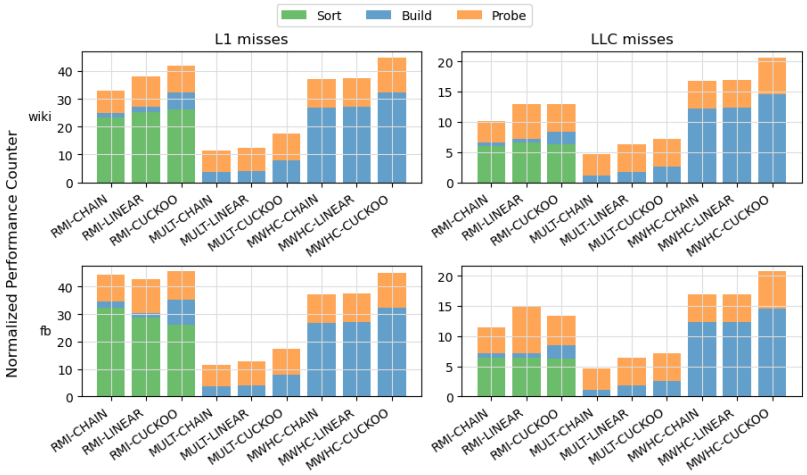
In the single-thread run, there is a structure with significantly more misses than the others during the probing phase: the RMI table with linear probing (in both datasets). This observation aligns with the performance counter values from the

probe experiment of Section 3.3.2.

In contrast, in the parallel run, the misses during the probing phase are more evenly distributed and slightly higher than in the serial one. This is attributed to the increased cache pollution, caused by multiple threads executing simultaneously. Interestingly, the RMI table with linear probing experiences fewer misses in the parallel execution than in the serial one. This is possibly due to the higher likelihood of threads accessing the same slot while linearly searching for the correct entry, reducing the number of misses.



(a)



(b)

Figure 5.5: Normalized cache performance counters for the serial (a) and parallel (b) join experiments, respectively.

Chapter 6

Cache Influence on Performance

As we observed numerous times, both hash tables and non-traditional hash functions are significantly influenced by cache dynamics. In this chapter, we will explore the extent of this influence and investigate strategies to mitigate cache-related limitations.

6.1 Pareto Probing Distribution

To delve deeper into the role of cache on the functions' performance, we introduce a new probe distribution based on the *Pareto principle*[1], where 20% of the data is probed 80% of the time.

Figure 6.1 illustrates an example of probe distribution from a real program run. Each slice represents the percentage of entries that were queried n times (e.g., 62.7% of entries were never probed, 17.0% were probed once, and so on). As expected, nearly 20% of the entries are probed twice or more (with a maximum of 19), while the remaining 80% are probed once or not at all.

Evaluation. We re-conduct all the probe-related experiments described in Chapter 3, to observe if and how performance changes when using such a distribution.

Surprisingly, the introduction of the Pareto probing distribution doesn't yield any significant differences in the observed trends. Both throughput values and

[1]The *Pareto principle* (or 80/20 rule) refers to the phenomenon where, in many situations, approximately 80% of the effects come from 20% of the causes. This principle is often used to describe uneven distributions, where a small portion of items or factors is responsible for the majority of the observed outcomes.

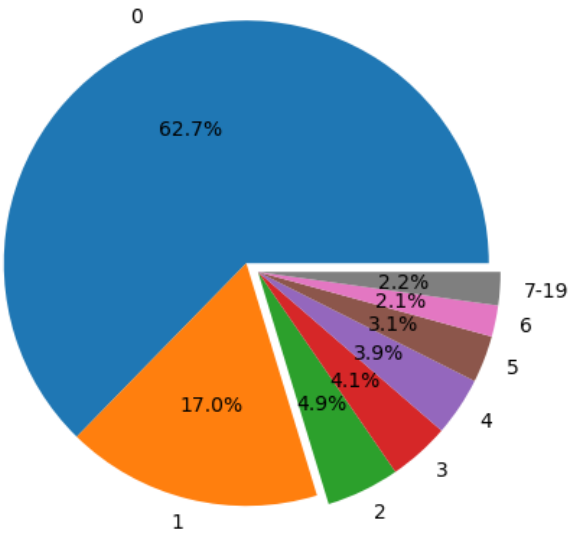


Figure 6.1: Illustrative example of the Pareto probe distribution, where each slice represents the percentage of entries that were queried n times.

performance counters are almost identical to the ones obtained with a uniform probe distribution.

Although we are mostly accessing 20% of the dataset, this portion remains too large to entirely fit in cache (160 MB). As a consequence, cache misses are still inevitable, as long as duplicated accesses occur sufficiently apart in time.

Overall, we are not able to draw any conclusion regarding the impact of cache size on the performance of different hash functions and tables, and different experiments must be designed to investigate such an impact.

6.2 Instruction Stream Interleaving

Let's recap what we have learned so far about the relationship between hash structures and cache.

Tables with high collision ratios tend to result in an increased number of cache misses, penalizing the overall execution performance. This effect is notably pronounced when employing a bucket chaining architecture, where buckets associated with the same linked list might be scattered far apart in memory.

Similarly, the performance of learned functions is linked to cache dynamics. Functions with a substantial number of submodels may exceed the available cache capacity, leading to significant performance slowdowns compared to functions with fewer submodels.

As an original contribution, we examine whether and to what extent these structures—hash tables and learned functions—can benefit from the utilization of Instruction Stream Interleaving (ISI).

6.2.1 Coroutines and ISI Background

In the context of computer systems, *Instruction Stream Interleaving* is a versatile technique employed to manage and multiplex multiple independent instruction streams. ISI can be leveraged to hide cache stalls through the following steps:

1. When we believe our program is about to access a variable that is not in cache, issue a prefetch request to the memory.
2. Switch to a different instruction stream.
3. Go back to step 1 until all instruction streams are terminated.

The core idea underlying this approach is that, upon returning to the stream that initiated the prefetch request, such a request will be completed, and the variable will be cached and ready for use.

However, as changing streams involves performing a context switch operation, it's crucial to halt a stream only when a variable access will result in a cache miss. Otherwise, the overhead of context switching will surpass the cost of simply fetching the resource from cache.

In C++20, the implementation of ISI can be naturally achieved through *coroutines*, which offer an abstraction for managing multiple instruction streams and their contexts.

This implementation, specifically in the context of mitigating cache stalls, has been extensively studied in [21].

Coroutine basic blocks. For the implementation of coroutine-based structures, we leverage some existing basic blocks available on GitHub[2]:

- *Scheduler* and *Throttler* – To manage and collect different streams in a First In, First Out (FIFO) fashion.
- *Prefetcher* – To request fetch operations from memory to cache, given an address and a scheduler.

These blocks provide an intuitive way to develop coroutine-based code , effectively abstracting the complexity of the C++ standard.

[2]github.com/turingcompl33t/coroutines

6.2.2 Bucket Chaining ISI Hash Table

As we mentioned earlier, our objective is to utilize ISI and coroutines to mitigate the impact of cache stalls caused by the spatial dispersion of buckets within the same linked list.

To achieve this goal, we enhance the existing bucket chaining implementation to accommodate two new types of lookup operations:

1. *Sequential multilookup* – Conducts multiple lookups in a traditional, sequential manner.
2. *Interleaved multilookup* – Conducts multiple lookups, employing coroutines to interleave different requests.

In the interleaved multilookup operation, the prefetch operator is invoked each time we access the next bucket in a linked list, under the assumption that such a bucket will probably not be in cache. Then, we context switch between the current stream and the following one in the scheduler, until all streams have run to completion and the queue is empty.

Evaluation. To evaluate this approach, we first insert 100M entries in the hash table, in a random order. Subsequently, we probe 100M uniformly sampled entries, employing both sequential and interleaved lookup strategies. The time gains achieved by using interleaved lookup as opposed to sequential—as well as the relative throughputs—are illustrated in Figure 6.2.

In most scenarios, a load factor of 200%—the maximum we used in the basic probe experiments—is not sufficient to yield any gain from the interleaved lookup. This behavior might be attributed to the linked lists not being sufficiently long to justify prefetching, with the majority of misses likely originating from accessing the bucket array.

However, as we increment the load factor, we observe a reduction in the bucket array size and an increase in linked list lengths. Such structures benefit from the additional streams, with time gains as large as 4x. Functions that consistently achieve stable collision ratios, such as Murmur and MWHC, also exhibit consistent trends across various datasets. This aligns with the expectation that the average linked list length is influenced by the collision ratio. MultiplyPrime follows a similar pattern, except for the *osm* dataset, where such a function displayed an unstable behavior in many other experiments.

Conversely, functions with more variable collision ratios—specifically, all the learned functions—benefit from interleaved lookup starting from lower load factors, and in datasets where they are not particularly efficient.

Surprisingly, the time gain of PGM consistently outperforms that of all the other functions. In the collision experiment of Section 3.2.3, PGM ranked as the slowest

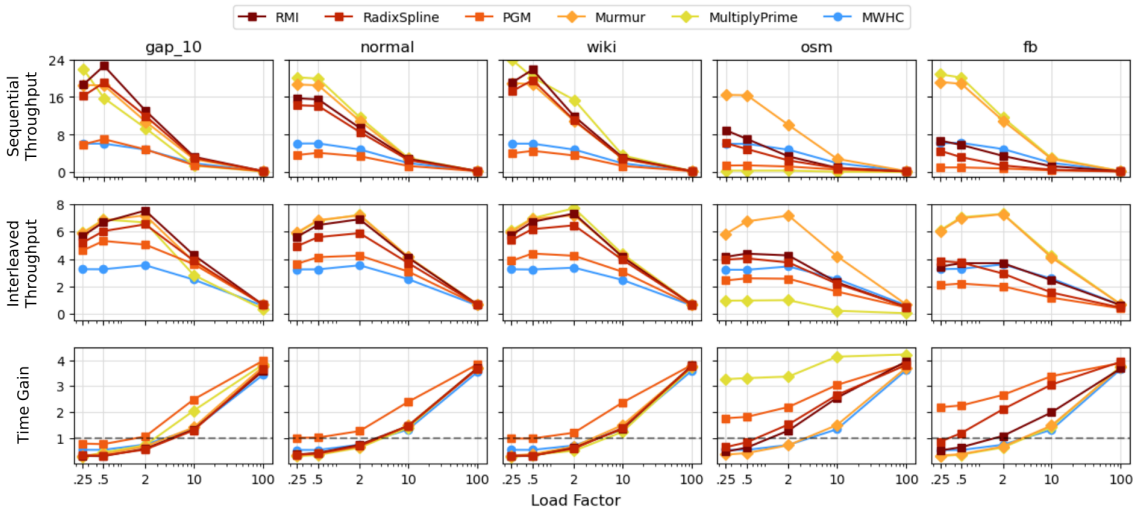


Figure 6.2: Sequential throughput, interleaved throughput and sequential/interleaved lookup time gains on different hash functions when varying the load factor. Throughput is expressed as million operations per second. Run on 10 streams, using the custom bucket chaining ISI hash table.

among all learned functions despite having the same collision ratios. This suggests that PGM, being among the less efficient hash functions in terms of throughput vs number of collisions, likely benefits from interleaved lookup due to a need for many prefetches (indicative of a high collision ratio), occurring at long intervals (aligning with its higher throughput).

6.2.3 RMI ISI Hash Function

Similarly to the previous use case, we employ ISI and coroutines to prefetch entries from the RMI submodels array. With this structure, we aim to address the challenges faced by functions with a high number of submodels, which experience notable slowdowns due to cache limitations.

Once again, we introduce two new types of hash operations: sequential and interleaved multihash. In this scenario, the prefetch operator is triggered each time we access the submodels array.

Evaluation. We evaluate this implementation by hashing 100M uniformly sampled entries. Results are displayed in Figure 6.3. The outcomes are not particularly promising, as the time gains never exceed 1 (indicating no actual gains).

Trends correlate with the cache size, with improvements becoming noticeable from 100k submodels onwards. This implies that smaller functions fit within the

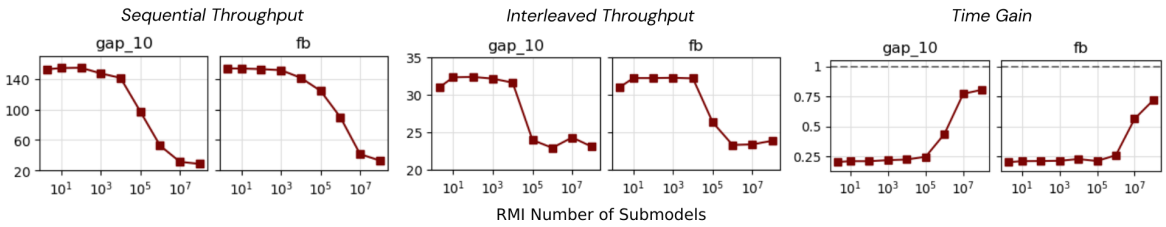


Figure 6.3: Sequential throughput, interleaved throughput and sequential/interleaved hashing time gains when varying the number of submodels in the RMI. Throughput is expressed as million operations per second. Run on 10 streams, using the custom RMI ISI hash function.

cache and are thus significantly penalized by the context switch between streams (as, in this case, prefetching from memory is completely useless).

However, with an increasing number of submodels, the time gain curves also rise, plateauing for 100M elements.

While this experiment cannot be considered successful, it still makes room for an interesting follow-up: can we integrate the single prefetching hash task into the ISI hash table? What potential improvements might this integration bring, and under what specific scenarios could these enhancements be observed?

6.2.4 RMI ISI Hash Table

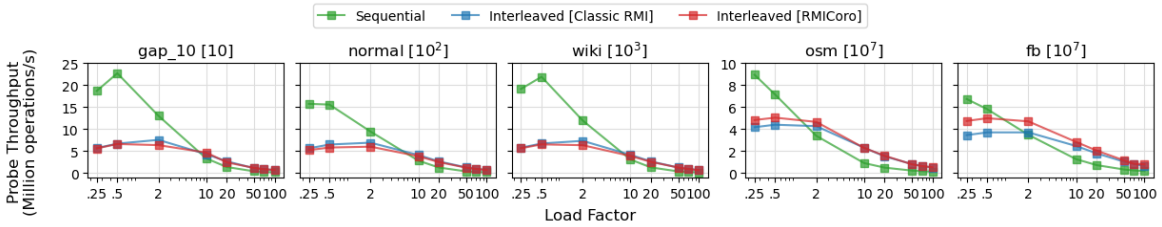
To continue this investigation, we develop a variant of the bucket chaining hash table that incorporates prefetching for both buckets and RMI submodels.

Again, we initially insert 100M entries into the hash table in a random order. Subsequently, we query 100M uniformly randomly sampled entries using three different strategies: sequential, interleaved with only bucket prefetching ("Classic RMI"), and interleaved with full prefetching ("RMICoro"). Results are presented in Figure 6.4.

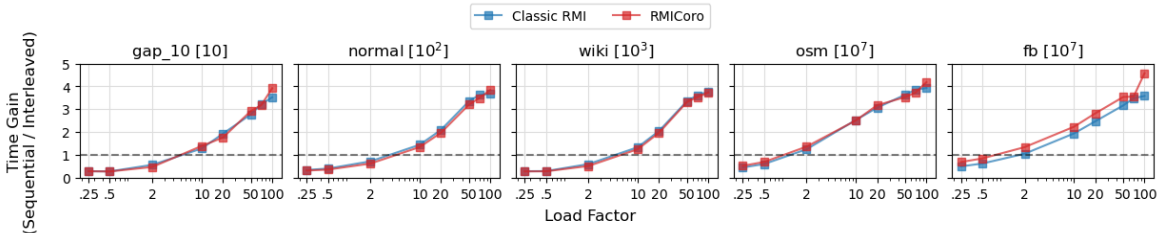
gap_10, *normal*, and *wiki* benefit from interleaved lookup starting at higher load factors (1,000%) compared to *osm* and *fb*, where the breakeven line is surpassed at a load factor of 200%. This can be attributed to the higher complexity of the *osm* and *fb* datasets, characterized by higher collision ratios and, consequently, longer linked lists.

Examining the time gain trends presented in Figure 6.4b, we observe they are either comparable or better for RMICoro. This implies that submodel prefetching has no drawback, even with small RMI sizes.

These results are intriguing, especially given that submodel prefetching alone was causing a 4x slowdown when the submodel array fits in cache. This suggests



(a)



(b)

Figure 6.4: Probe throughputs (a) and sequential/interleaving lookup time gains (b) on the RMI hash function with and without submodel prefetching. Run on 10 streams. Numbers in square brackets indicate the number of submodels for the underlying RMI model.

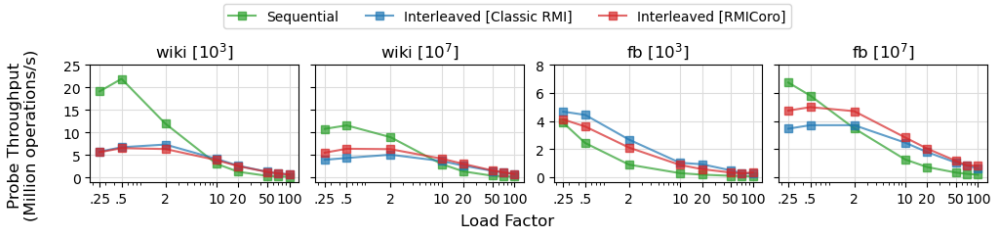
that the RMICoro structure fills the performance gap between bigger and smaller models, especially for high load factors.

To further investigate this theory, we rerun the experiment on *wiki* and *fb*, utilizing model sizes of 10^3 and 10^7. Results are displayed in Figure 6.5.

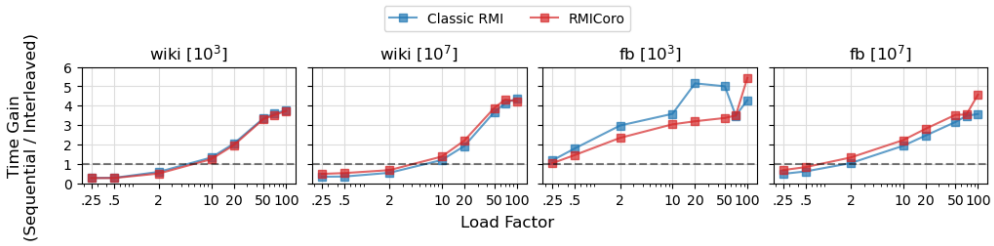
Based on our previous experiments (see Figure 3.4), we know that:

- *wiki* is a regular dataset, and 10^3 submodels are enough to achieve low collision rates. Using 10^7 submodels doesn't bring any additional advantage in terms of collisions. Moreover, this variant exceeds cache capacity, negatively impacting the performance.
- *fb* is a highly irregular dataset. In this case, 10^3 submodels are insufficient, leading to underfitting and high collision rates. Conversely, an RMI of size 10^7 is complex enough to reduce the collision rate without overfitting.

These characteristics are reflected by the throughput trends of Figure 6.5a. At low load factors, a comparison between *wiki* 10^3 and *wiki* 10^7 reveals similar curves for interleaved lookups, but significantly lower throughput for sequential 10^7. In contrast, for *fb* 10^3, sequential lookup consistently performs worse than the interleaved variants. This is expected, given the long linked lists resulting from underfitting. *fb* 10^7 exhibits a behavior similar to *wiki*.



(a)



(b)

Figure 6.5: Probe throughputs (a) and sequential/interleaving lookup time gains (b) on the RMI hash function with and without submodel prefetching. Run on 10 streams. Numbers in square brackets indicate the number of submodels for the underlying RMI model. Different submodel sizes are used for the same dataset to assess the benefits of prefetching.

These results validate our hypothesis that, for sufficiently high load factors[3]:

- Interleaved lookup outperforms sequential lookup.
- Models with larger sizes outperform smaller ones.
- The addition of submodel prefetching comes with no additional cost, and it can enhance performance.

Therefore, the RMICoro structure emerges as the optimal choice for complex datasets and/or high load factors, enabling the selection of bigger-size models over simpler ones, without incurring in runtime penalizations.

[3]Where the *sufficiently* threshold depends on the complexity of the dataset.

Chapter 7

Conclusions

In the first part of this thesis, we discussed the utilization of learned indices as hash functions. We compared these *learned functions* with both traditional functions—fast but prone to collisions—and perfect functions—collision-free but considerably slow.

We proved, both analytically and experimentally, that collisions in piece-wise linear learned functions strongly depend on the distribution of the gaps between consecutive keys. If the variance of such distribution is low, and gaps are regular, linear functions can achieve fewer collisions than traditional hash functions.

We also demonstrated the impact of the number of submodels on performance. Specifically, increasing this number beyond a threshold results in decreased throughput due to cache misses. Conversely, reducing it below a threshold leads to underfitting of the model, thereby increasing collisions. For these reasons, this is a parameter that should be carefully tuned, according to the dataset's complexity.

We analyzed these functions in action in hash tables implementing various schemes. As expected, we found that the learned function effectiveness depends on the dataset and it can even surpass traditional hash functions, especially in bucket chaining tables.

When examining the multi-thread versions of these structures, we found that they scale effectively without significant drawbacks. Learned functions maintain performance comparable to the sequential case, unaffected by the increased cache pollution. Notably, cuckoo hashing experiences the most substantial impact during the insertion phase, as it requires acquiring two locks simultaneously to write to the table.

We explored non-partitioned joins as a practical application of hash tables. Unlike "classic" hash structures, NPJ requires the table to be constructed online, deeply penalizing learned functions as they require both a sort and a train phase. Achieving meaningful improvements demands ad-hoc modifications of the models, a topic beyond the scope of this research.

Finally, we investigated the relationship between these structures and cache performance. We crafted specialized implementations of RMI and bucket chaining hash tables using coroutines to mitigate cache stalls. These structures demonstrated that memory prefetching can significantly accelerate multiple lookups in the table, especially under high load factors, provided the keys are known in advance.

Furthermore, this ISI variant eliminates throughput variations among RMI models of different complexities, allowing the selection of larger models over simpler ones and mitigating the need for fine-tuning the submodel number.

Part II

RMI and Genomic Aligners

Chapter 8

Genomic Aligners and Accel-Align

8.1 Genomic Aligners

A genomic aligner is a software performing an operation known as *sequence alignment*: given a reference genome[1] and a set of sequenced reads[2], we want to find the origin location of each read in the reference.

These tools play a significant role in genomic research. For example, they can help recognize the function of various DNA regions in a biological process. Moreover, they facilitate the identification of markers to reveal the genetic origins of diseases.

In essence, sequence alignment consists of searching for substrings using an approximate fashion: since reads are generated by error-prone machines, and discrepancies may exist between the reference genome and the sequenced organism, these strings are inclined to substitution and insertion/deletion (INDEL) errors.

What makes this problem especially challenging is the size of the reference, which is normally around 3 billion characters long[3].

---

[1] A *reference genome* is a representative DNA sequence. It serves as a standard for comparing and analyzing the genetic information of individuals within a species.

[2] A *sequenced read* is a short segment of DNA produced through sequencing technologies.

[3] To put it in perspective, this is equivalent to approximately 1,000 copies of Tolstoy's "War and Peace"!

8.1.1 SFE Aligners

Numerous implementations have been presented in the literature. State-of-the-art aligners, such as Bowtie2 [22], BWA-MEM [23], and Minimap2 [24], use a *seed-filter-extend (SFE)* strategy for performing alignment. This consists of four distinct phases:

1. [Offline] *Indexing* – The reference genome is initially indexed using popular structures, such as hash tables or FM-indices.
2. *Seeding* – The read is broken down into smaller segments, known as *seeds*. Each seed is associated with its candidate locations in the reference, using the pre-built index.
3. *Filtering* – Candidate locations are filtered using specific techniques. This phase involves a performance-accuracy tradeoff: the more locations filtered, the faster the alignment, but with a higher risk of eliminating the true match.
4. *Extending* – The entire read is aligned at each of the remaining locations, and the edit distance[4] between read and candidate is computed. The candidate with the lowest distance is the predicted position.

These tools can map thousands of reads per second; nevertheless, they encounter scalability issues, attributable to the edit distance. While such a metric is ideal for our use case, accounting for both substitution and INDEL errors, it comes at a significant computational cost: for a read of length R, computing the edit distance has a complexity of approximately $\mathcal{O}(R^2)$. This emphasizes the importance of the filtering phase in modern aligners.

Filtering techniques. Filtering techniques vary among aligners and can be classified as either elimination-based or selection-based.

Elimination-based techniques prioritize maintaining a high accuracy by cautiously eliminating candidate locations, to minimize the risk of erroneously discarding the true match. These techniques are computationally challenging, and require specialized hardware for efficiency. Some examples are adjacency filtering [25] and shifted hamming distance [26].

Selection-based techniques employ less intensive selection criteria to choose only a few candidate locations, sacrificing accuracy for improved performance. An example is the voting system in Subread [27].

[4]Informally, the *edit distance* between two words is the minimum number of single-character edits (insertions, deletions or substitutions) required to change one word into the other. It's also known as the Levenshtein distance.

8.2 Accel-Align

Accel-Align [28] is a genomic aligner that introduces a novel strategy, called *seed-embed-extend (SEE)*, as an alternative to the traditional SFE approach. Instead of filtering candidate locations and then computing the various edit distances, Accel-Align employs a randomized algorithm [29] to reformulate the alignment optimization problem.

The algorithm generates a one-to-one mapping e, called *embedding*, with the following property. For any two strings x and y, and their corresponding embedded versions $e(x)$ and $e(y)$, the ratio of the edit distance between x and y to the Hamming distance [5] between $e(x)$ and $e(y)$ is very low. This ratio is commonly referred to as the *distortion*.

We can use such a procedure to transform a read r and a candidate string $\hat{r}$: this way, their edit distance can be approximated using the Hamming distance between their embeddings.

Consequently, the alignment optimization problem is translated as follows:

$$\min_{\hat{r}}\Big\{d_E(r,\hat{r})\Big\} \longmapsto \min_{\hat{r}}\Big\{d_H(e(r),e(\hat{r}))\Big\} \tag{8.1}$$

where $d_E(\cdot)$ and $d_H(\cdot)$ are the edit and Hamming distances, respectively.

This conversion is beneficial because the Hamming distance incurs a linear computational cost ($\mathcal{O}(R)$), enabling Accel-Align to achieve speed-ups of up to 9x for BWA-MEM, 12x for Bowtie2, and 3x for Minimap2, although with slightly lower (yet still comparable) accuracy.

8.2.1 Building the Index

Similarly to the aligners we discussed in Section 8.1.1, the first (offline) phase of Accel-Align consists in indexing the reference genome.

The index is implemented as a hash table of key-value pairs, which are 32-bit unsigned integers. We set the number of slots to $M = (2^{29} - 1)$.

To populate the table, we first consider all the overlapping seeds of length k—the so-called *k-mers*—in the reference string. It's important to note that there are only four distinct nitrogenous bases—A, C, G, and T—which allows us to encode each base using just 2 bits. Consequently, k-mer will be represented by a $2k$-bit value. k-mers where the any-base symbol "N" appears are ignored.

Each integer k-mer is hashed using a simple modulo-based hash function, and paired to its position (that is, the origin of its first element within the reference

[5] The *Hamming distance* between two words measures the minimum number of substitutions required to change one word into the other. Contrary to edit distance, it does not detect insertion nor deletion.

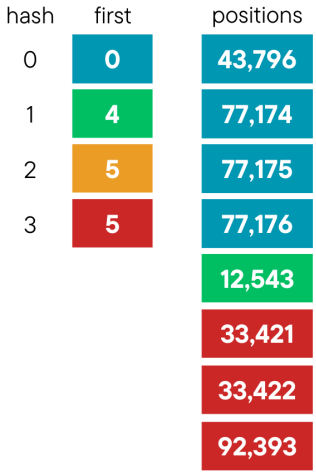


Figure 8.1: Example of the content of the *positions* and *first* array.

genome). With this implementation, there will be at most M distinct hashes and $N - k - 1$ positions, where N is the length of the reference genome.

The table is efficiently stored by using two 32-bit arrays:

1. *positions* – Contains the positions, sorted by increasing hash value.
2. *first* – Contains, for each possible hash value, its cumulative position[6]. This coincides with the index of the first position relative to that hash value.

Notice that the *first* array has always $M + 1$ entries[7], and a fixed size of 2.14 GB.

Figure 8.1 displays how the index can be used to recover the position list for a hash. Given a generic hash value h, the relative locations will be found as following:

$$b = first[h]$$
$$e = first[h + 1]$$
$$pos\_h = positions[b : e]$$

Choosing k. The choice of k significantly influences the aligner's performance. A smaller k reduces the probability that a k-mer in the read includes errors, correlating with improved accuracy. However, it also results in longer candidate location lists, decelerating the embedding phase —a larger number of locations must be embedded. The default k value for Accel-Align is set to 32, which is significantly higher compared to other aligners (where $15 \leq k \leq 22$).

[6]The cumulative position is the sum of all the positions of hashes lower than the current one.

[7]We add an extra element, whose content is equal to *positions*.size(), to avoid illegal accesses when $h = (M - 1)$.

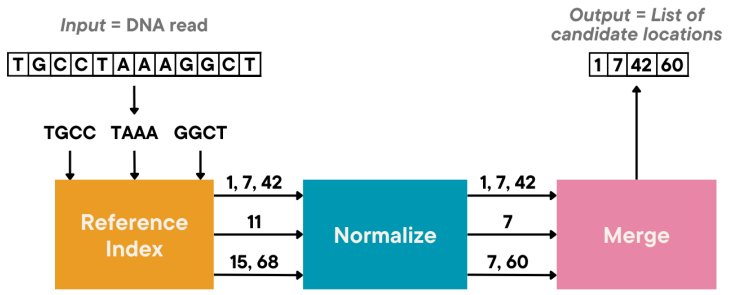


Figure 8.2: Flow of the seeding phase in Accel-Align, illustrated with an example featuring a read of length 12 and seeds of length 4 (4-mers).

8.2.2 Seeding

As previously discussed, the most critical phase in SFE-based aligners is typically the filtering phase, given its impact on both accuracy and runtime. In Accel-Align, however, the phase with these characteristics is the seeding phase.

Figure 8.2 illustrates the flow of the seeding phase in Accel-Align. We begin by splitting the DNA read into non-overlapping k-mers. Each k-mer is then fed to the reference genome index, producing a list of locations where the k-mer appears. The list is then normalized, by removing from each location the offset of the k-mer in the read. Finally, all lists are merged to remove duplicates.

8.2.3 Embedding

In the embedding phase, we first extract strings of length R from the reference genome at each candidate location. Subsequently, we calculate the embeddings of both the read and the extracted strings, and compute the Hamming distance between each embedded reference and the embedded read.

During this process, we maintain a record of the top two candidate locations with the least Hamming distance. Finally, we pass these top two candidates to the extending phase.

8.2.4 Extending

Finally, in the extending phase, the read is aligned at the top two candidate locations, and a mapping quality (MAPQ) is reported, indicating the degree of confidence in the alignment for the read.

While state-of-the-art SFE aligners face computational challenges in this phase due to the edit distance, Accel-Align has the advantage of handling only the top two candidate locations, making this the most lightweight stage in the Accel-Align process.

8.3 Accel-Align Index Precision

The current implementation of Accel-Align[8] faces a significant issue.

To understand the problem, let's recall the simple index used by the aligner. This index includes a list of positions and a *first* array for constant-time navigation. However, the original values of the k-mers are not saved.

This means that, for a given seed $\mathcal{S}$, the index returns the location of ANY seed a that satisfies the following condition:

$$a \bmod (2^{29} - 1) = \mathcal{S} \bmod (2^{29} - 1) \tag{8.2}$$

As a consequence, the aligner ends up processing seeds that share the same hash but have nothing in common with $\mathcal{S}$. We call these *false seeds* (Figure 8.3).

Now, as the number of false seeds (and their respective positions) increases, the aligner's performance is adversely affected. This is because the pipeline spends time processing candidate locations related to false seeds, which are inherently incorrect.

Figure 8.3: Three cases of false seeds, compared with the actual seed (on top).

8.3.1 Definition

To quantify this phenomenon, we study the index *precision* over a set of reads. Algorithm 2 describes how this metric is computed. Firstly, we segment the read into non-overlapping seeds. For each seed, we determine the number of actual positions compared to those returned by the index. We compute the same values for the reverse complement[9] of the seed. Summing these partial results, we obtain the final precision as the ratio of actual to indexed positions.

[8] github.com/yanlolo/accel-align-release

[9] The *reverse complement* of a DNA string is a new string that is formed by reversing the original DNA sequence and replacing each nucleotide with its complementary base: A-T and C-G.

Algorithm 2 Index precision over a set of reads

```
Input: reference, read, kmer_len
Returns: precision
actual ← 0
indexed ← 0
for each seed in read do
    // Get positions from the reference index
    positions ← reference.index(seed)
    indexed ← indexed + positions.size()
    for each p in positions do
        // Check if the position is correct
        if reference.get_kmer(p, kmer_len) == seed then
            actual ← actual + 1
        end if
    end for
    // Repeat for the reverse complement of seed
end for
precision ← actual / indexed
```

It's important to note that the index precision is heavily dependent on the seed size k: as k increases, the compression effect of the hash function also rises, leading to a higher number of false seeds.

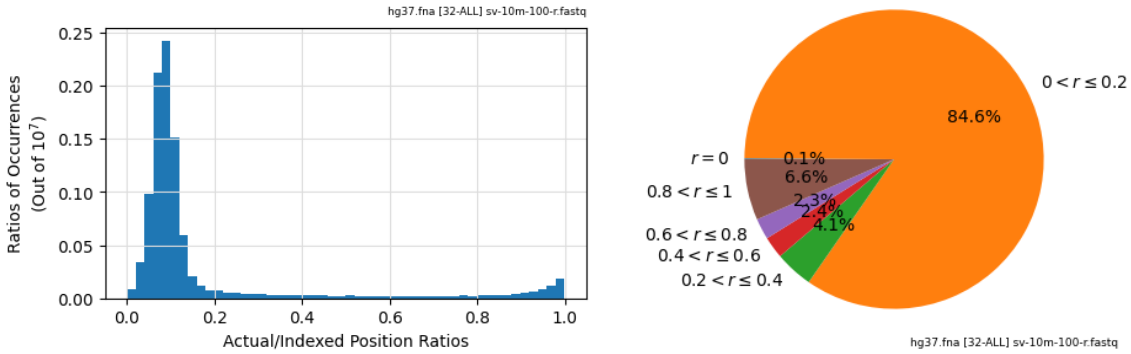
8.3.2 Evaluation

As our reference, we select the *hg37* human genome, and we build on top of it an index with seed size $k = 32$. We then consider a set of 10^7 synthetic reads of length 100, and compute the index precision over each of them. Figure 8.4 displays the obtained results.

The precision values highlight the relevance of this issue: in the majority of cases (84.6% of reads), over 80% of candidate locations are false, with the most prominent peak in the distribution occurring at $r \approx 0.1$.

This suggests that the application of an index with "perfect" precision will eliminate this large portion of false positions, thereby reducing the workload on subsequent stages of the aligner. However, such a perfect index introduces challenges, as it involves increased memory requirements (for storing the original k-mers) and time costs (checking k-mers deviates from the theoretical $\mathcal{O}(1)$ cost).

Will the benefits of fewer candidate locations justify the elevated indexing cost?



(a) **(b)**

Figure 8.4: Distribution of the index precision values for 10^7 reads, with $k = 32$.

Chapter 9

Accel-Align-RMI

9.1 Motivations

With the advent of RMI in the literature, many leading state-of-the-art aligners have introduced RMI-based versions, which enhanced their performance. Examples include BWA-MEM [30] and Minimap2 [31].

Given that we are looking for an indexing structure with perfect precision, the RMI emerges as a natural candidate in this investigation.

Our aim is to explore whether Accel-Align can, like the aforementioned aligners, derive advantages from such an index. Moreover, we want to assess whether substituting the fast yet imprecise hash table with a slower but perfect index can enhance the alignment pipeline, leading to an overall performance improvement.

9.2 Accel-Align-RMI Implementation

We will now present the techniques employed in the integration of the RMI index into Accel-Align. The following sections detail the specific methodologies utilized throughout the implementation process.

9.2.1 Key Generation

Before constructing the index, the first step involves generating the necessary key-value pairs from the reference genome.

To accomplish this, we divide the genome into its overlapping k-mers. Similarly to the classic Accel-Align index, the key of a k-mer is its $2k$-bit integer representation. To address hardware and language constraints, along with compiler optimizations, we extend $2k$ to the closest larger power of 2 `XX` (which will be either 32 or 64).

Once all keys have been derived, we proceed with the generation of the files `keys_uintXX` and `pos_uint32`. These files serve as storage for keys and positions within the reference genome, respectively.

Both files begin with an unsigned 64-bit counter indicating the total number of entries. In `keys_uintXX`, each entry consists of a pair of (`XX`, 32)-bit unsigned integer values representing the key and its cumulative position *first*. `pos_uint32` contains a simple list of 32-bit unsigned integer positions. The detailed generation process can be found in Algorithm 3. An example of the final structure is displayed in Figure 9.1.

Again, as in the classic index, the content of the generated files remains constant given a specific reference genome. As a result, it's sufficient to train the RMI once—offline—without impacting the alignment performance.

Algorithm 3 Binary files generation

Input: *keys*, *positions*, *kmer_len*
`XX` $\leftarrow$ 32 **if** 2**kmer_len* $\leq$ 32 **else** 64
Sort *keys* in ascending order
`========== keys_uintXX ==========`
write(*number_different_keys*, 8)
first $\leftarrow$ 0
prev_key $\leftarrow -1$
for each k **in** *keys* **do**
    if k != *prev_key* **then**
        write(k, sizeof(k))
        write(*first*, 4)
    end if
    first $\leftarrow$ *first* + 1
end for
// Add synthetic key to avoid useless runtime checks
write(0, sizeof(k))
write(*number_positions*, 4)
`========== pos_uint32 ==========`
write(*number_positions*, 8)
for each p **in** *positions* **do**
    write(p, 4)
end for

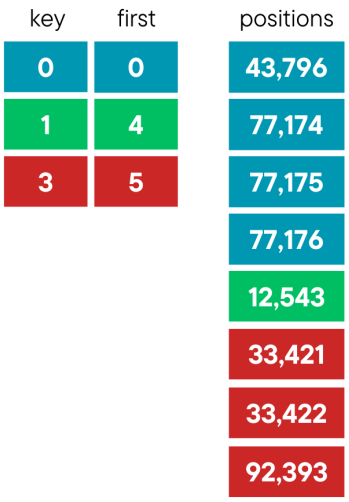


Figure 9.1: Example of the content of `keys_uintXX` (right) and `pos_uint32` (left) files. For the sake of brevity, the number of entries in both files is omitted.

9.2.2 Dynamic Library Implementation

`RMI`[1] [32] is the most popular implementation of the Recursive Model Index, written in Rust and fully parallelizable.

The tool can work in two different modes:

- *train* – In the default mode, the tool trains the parameters of a specified RMI architecture. Moreover, it generates C++ code that must be utilized to incorporate the trained architecture in larger projects.
- *optimize* – The tool also offers an "optimize" mode, where it automatically fine-tunes the hyperparameters, returning a concise set of architectures that cover the Pareto front[2].

The main drawback of the original repository implementation lies in its *code generation* approach. Because each index is closely tied to specific C++ code, utilizing a new index[3] would necessitate recompiling the entire aligner.

To address this challenge, a novel approach is employed, treating the index as a library. To achieve so, we encapsulate the generated code within a shared object.

[1] github.com/learnedsystems/RMI

[2] In multi-objective optimization, the *Pareto front* represents the set of optimal solutions that cannot be improved in one objective without sacrificing performance in another objective.

[3] "Utilizing a new index" means more than just employing a different index structure: it also includes changes in the k-mer length, or adopting a different reference genome!

This shared object will be dynamically loaded during runtime, eliminating the need for full project recompilation when switching between different indices.

Furthermore, to simplify the management of the shared object, a generic C++ class wrapper is utilized. This wrapper provides convenient control and interaction with the dynamically loaded index. These modifications are incorporated into our personal fork of the repository[4].

The fork also offers several new features, which further enhance the functionality and usability of the `RMI` tool:

1. *Removal of* `<filesystem>` *library requirement* – The generated code eliminates the dependency on the `<filesystem>` library. This adjustment ensures compatibility with older versions of C++, thereby expanding the range of supported environments.
2. *Support for key-value structured files* – We introduce support for structured files containing key-value pairs, where the value is disregarded during the index training process. This is an important improvement, as these are the binary files we generate from the reference genome.
3. *Enhanced optimization output* – The optimization process provides now improved output, including detailed information such as building time. This enhancement allows users to gain insights into the performance and efficiency of the index-building process.

These additional features contribute to the overall usability and versatility of the tool, making it a valuable and enhanced version of the original repository implementation.

9.2.3 Accel-Align Integration

With all the necessary preparations in place, we can finally integrate the RMI into Accel-Align.

Firstly, we utilize the aforementioned library wrapper class to load the index and establish communication with it. This step enables us to leverage the functionalities of the index within Accel-Align seamlessly. Next, we modify the existing file loading procedure in Accel-Align to accommodate the new format of the `keys_uintXX` and `pos_uint32` files.

We proceed by implementing a helper function that performs a bounded binary search using the output of the index lookup function, which includes predicted position and maximum allowable error. The binary search function aids in efficiently

[4] github.com/IlariaPilo/RMI

locating the desired seed within the index structure, and returns the correspondent positions interval. Finally, we replace every direct index access in Accel-Align with a call to the binary search function.

This implementation successfully abstracts the index lookup from the aligner's logic, allowing to effectively swap between different indices, as long as all of them implement the same interface (i.e., a `lookup` function returning the positions interval).

9.3 Proof-of-Concept Variants

The classic Accel-Align index and the RMI are based on opposite approaches: the former prioritizes speed and compression, while the latter sacrifices speed for perfect precision. To facilitate a fair comparison and enable a thoughtful analysis, we introduce two new proof-of-concept Accel-Align variants, each based on indices with varying performance-precision tradeoffs (Figure 9.2).

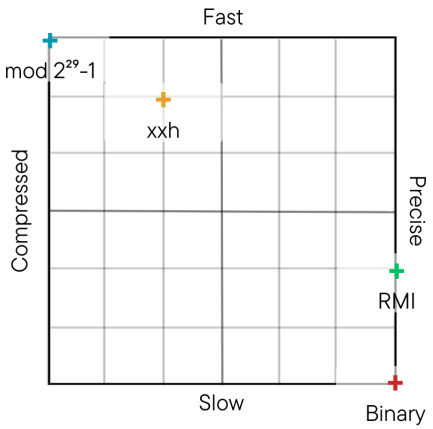
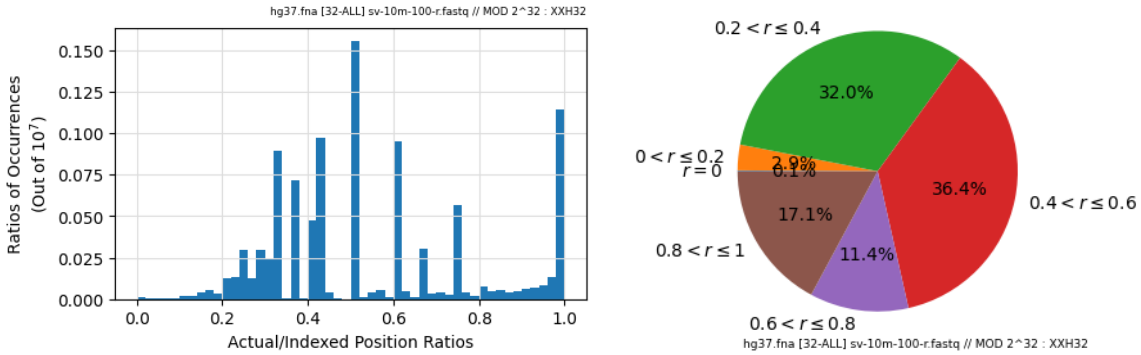


Figure 9.2: Visual representation for the performance-precision tradeoffs of the Accel-Align variants.

9.3.1 Binary

The *binary* variant of Accel-Align serves as a baseline for evaluating the RMI. It employs the same underlying structures (`keys_uintXX` and `pos_uint32`), but instead of leveraging the learned index to predict position and maximum error, it performs a full-length binary search. Therefore, it also achieves perfect precision.

We anticipate this variant to be the slowest among the four. Its inclusion allows us to assess the performance gains offered by the RMI bounded binary search compared to the basic approach.



(a) **(b)**

Figure 9.3: Distribution of the *XXH* index precision values for 10^7 reads, with $k = 32$.

9.3.2 XXH

The *XXH* variant of Accel-Align provides a less compressed hash-based alternative. It employs the same arrays as the classic index (*first* and *positions*), but with number of slots $M = 2^{32}$ and hash function xxHash32. This version is expected to be slightly slower than the classic one (due to more slots leading to increased cache misses), but it's significantly more precise, as illustrated in Figure 9.3.

XXH poses a real challenge for RMI: is a perfect index going to be superior to a less compressed, faster hash table?

9.4 Evaluation

We now assess the performance of the RMI aligner with a seed length of 32, comparing it to the other three versions (classic, binary and XXH).

As a reminder, we expect the RMI version to exhibit longer seeding times (attributed to a bounded binary search vs. direct access) but shorter embedding times (due to fewer candidate locations) with respect to the original implementation.

9.4.1 Setup

Our reference is the *hg37* human genome, and we attempt to align 10^7 synthetic reads of length 100 to it. Experiments are conducted using 16 threads, a configuration commonly used for sequence alignment [24].

We use a server having the following specifications:

- CPU – Intel(R) Xeon(R) Gold 5317 CPU @ 3.00GHz

- Threads – 48 [2 sockets, 12 cores per socket, 2 threads per core]
- L3 size – 37.7 MB
- RAM size – 128 GB
- Architecture – x86_64

This machine has multiple sockets, prompting the need to make the aligner *NUMA-aware*[5] for enhanced benchmark reliability. To achieve this, the chosen thread number is dynamically used to compute the minimum number of sockets required to run the program. We then utilize the `numactl` utility to activate only the relevant sockets for both computation and memory allocation.

9.4.2 Hyperparameter Optimization

To determine the optimal architecture for the index, we employ the *optimize* mode of the `RMI` tool, which provides us with a selection of the top 10 architectural choices. Results are displayed in Figure 9.4.

Since we want to prioritize speed, and we do not have strict space requirements, we select the fastest index configuration (`radix,linear` with branching factor 16777216).

Models	Branch	AvgLg2	MaxLg2	Size (B)	Time (s)
radix,linear	16777216	6.48855	19.96305	402653200	267.82
radix,linear	8388608	7.43584	20.05112	201326608	264.02
radix,linear	4194304	8.07029	20.24968	100663312	263.58
radix22,linear	1048576	9.43219	31.24944	41943040	262.44
radix,linear	524288	10.86551	20.60409	12582928	235.44
radix,linear	262144	11.58189	20.89362	6291472	225.68
radix18,linear	32768	14.01674	31.24944	1835008	232.74
radix,linear	32768	14.49187	21.38813	786448	222.41
radix,linear	1024	18.95660	23.72112	24592	266.70
radix,linear	128	21.93567	25.47009	3088	267.03

Figure 9.4: Output of the `RMI` optimizer for $k = 32$, displaying the 10 most promising architectures of decreasing complexity.

[5] *Non-uniform memory access (NUMA)* is a memory design where the access time depends on the memory location with respect to the processor. In this context, a *NUMA-aware* software is designed to make informed decisions about where to allocate and access memory based on the proximity to the processor or processors it is running on.

9.4.3 Results

Size comparison. The four index variants share identical *positions* values, occupying 11.46 GB in total. This consistency is expected, since these values are determined solely by the reference genome and the k-mer size. Therefore, we focus on the other index components, outlined in Table 9.1.

In hash-based variants, the size of the *first* array is determined by the slot number M, and remains constant regardless of reference genome or k value. Specifically, it will always be equal to $4(M+1)$, resulting in 2.14 GB for the classic aligner and 17.18 GB for XXH.

On the other hand, perfect-precision indices must retain all distinct overlapping 32-mers from the reference. Each 32-mer requires 12 bytes of storage (8 for the seed and 4 for the cumulative position), totaling 30.60 GB. The RMI requires extra space to store its parameters, amounting to 402 MB.

Overall, the RMI and binary variants are significantly more costly in terms of storage occupation, exceeding the size of the classic implementation by over 14 times. This increase is attributed to the absence of compression, coupled with the requirement to track the reference seeds for achieving perfect precision.

While XXH presents a more balanced option, it remains 8x larger than the original index.

Accuracy comparison. The original aligner achieves an accuracy of 97.08% over the 10^7 reads. XXH slightly improves these results, with an accuracy of 97.15%. Perfect-precision indices reach an accuracy of 97.25%, an all-time maximum for Accel-Align with $k = 32$.

Notice how a reduction in compression correlates with an increase in accuracy. As the index type does not influence the alignment strategy, but merely prunes incorrect candidate locations, reducing the compression eliminates instances where the read was mistakenly aligned to a false candidate location.

	Classic	**XXH**	**RMI**	**Binary**
seeds	—	—	10.20	10.20
first	2.14	17.18	20.40	20.40
Extra	—	—	0.40	—
Overall	2.14	17.18	31.00	30.60

Table 9.1: Storage occupation for the four variants of Accel-Align, in GB.

Time comparison. Table 9.2 presents the running times for the four variants, detailing some key phases of the alignment process.

These phases include the index lookup procedure (time needed to obtain candidate locations from a seed), hit count (merging candidate locations from different seeds), embedding, and extending. The combination of index lookup and hit count constitutes the seeding phase.

Analyzing the lookup time results, we find that the RMI brings a significant 3.7x improvement over the full-length binary search, proving that the predicted position and error bounds have a meaningful effect on the search speed. This effect is particularly important considering that, in the binary variant, the lookup phase is so long that it surpasses the embedding as the most time-consuming aspect of the alignment process.

Contrary to expectations, XXH exhibits the fastest lookup time among the variants, while we anticipated it to be slightly slower due to increased cache misses. Further investigation into this behavior will be conducted in Section 9.5.

When examining the embedding times, the classic index yields the poorest results due to its low precision. Surprisingly, the perfect-precision variants fail to surpass XXH in performance. This suggests that XXH's lower compression, coupled with the effective randomization of xxHash and the Accel-Align embedding heuristics, are enough to enhance the speed of this phase.

In summary, compared to the original implementation, Accel-Align-RMI demonstrates superior performance despite the lookup phase being nearly 3.7x slower. The reduced number of candidate locations accelerates the embedding by 1.5x, resulting in an overall 1.2x improvement.

However, the XXH variant emerges as the most balanced among the four: it surpasses all in terms of time efficiency, offers heightened accuracy compared to the original Accel-Align, and demands only 8x additional storage.

	Classic	**XXH**	**RMI**	**Binary**
Index lookup	0.63	0.47	2.31	8.56
Hit count	4.77	4.23	4.46	5.11
Embedding	10.32	6.66	6.72	6.92
Extending	1.44	1.32	1.36	1.45
Overall	23.84	18.37	20.34	25.97

Table 9.2: Comparison of running times for 10^7 reads, in seconds. Benchmarks run on 16 threads, averaged over 7 runs.

9.5 Scalability Analysis

We examine how altering the thread number influences alignment time, with a specific emphasis on the consistency of time across multiple runs. Results are illustrated in Figure 9.5.

Time and CV trends. The time curves for the four variants consistently decrease until 24 threads, where they begin to increase and then decrease again, reaching a minimum for 44 threads. This pattern is easily explained by the hardware architecture of our machine: as each socket supports 24 threads, exceeding this number causes Accel-Align to run on two different sockets, leading to slowdowns due to NUMA.

For a similar reason, the CV[6] curves exhibit noticeable instability when the thread number exceeds 24. Interestingly, we also observe a peak around 16. This is likely due to hyperthreading, which activates after 12 threads and reaches its peak instability around 16—when almost half of the cores are utilizing it—before stabilizing again.

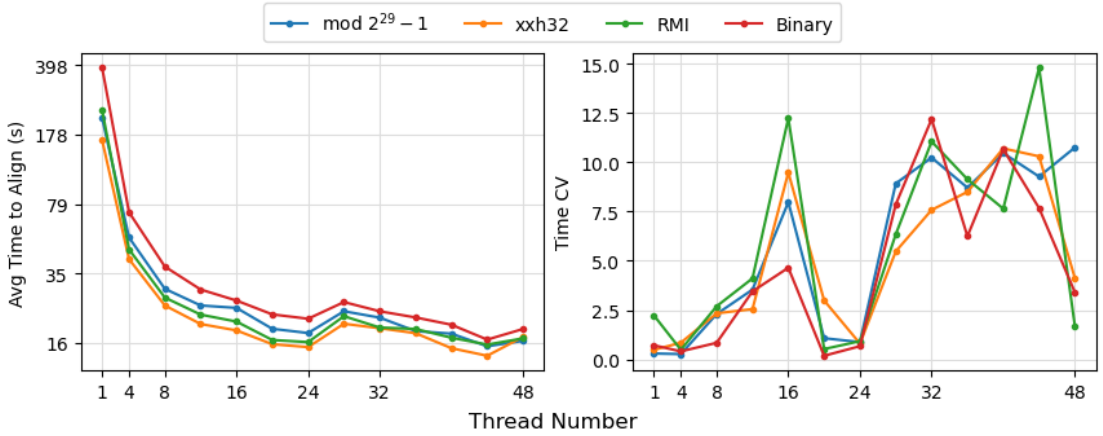


Figure 9.5: Average time to align and CV for $k = 32$ over 7 runs, varying the thread number.

Scalability of variants. Upon analyzing the four variants, XXH consistently emerges as the fastest option, followed by RMI, classic Accel-Align, and finally binary. An interesting exception to this behavior is observed in the single-thread

[6]The *coefficient of variation (CV)* is a statistical measure of dispersion, defined as the ratio of the standard deviation to the mean, and often expressed as a percentage.

	Classic	**XXH**	**RMI**	**Binary**
Index lookup	0.31	0.60	40.58	173.90
Hit count	57.14	51.59	61.52	73.00
Embedding	109.21	71.38	82.87	88.18
Extending	17.75	17.71	17.75	17.77
Overall	215.43	166.64	235.54	385.43

Table 9.3: Comparison of running times for 10^7 reads, in seconds. Benchmarks run *sequentially*, averaged over 7 runs.

execution, where the original implementation outperforms RMI. Detailed timing information is provided in Table 9.3.

While the lookup time for the classic index is now lower than the one of XXH—as anticipated—the contrast becomes impressing when comparing it to RMI and binary variants: the classic index's lookup time is over 130x lower than that of RMI, and more than 560x lower than that of the binary variant!

These results are particularly noteworthy if we recall the 16-thread data from Table 9.2. In that context, the RMI lookup time was only 3.7x slower, and the binary lookup time was, similarly, 14x slower. Even more intriguingly, the lookup time for the classic index with 16 threads is twice that of the single-thread execution.

Overall, we can infer that the three new variants—but especially RMI and binary—exhibit superior scalability compared to the original version, demonstrating significant performance gains with an increased number of threads.

Chapter 10

Conclusions

In the second part of this thesis, we investigated the integration of the RMI to enhance the performance of the genomic aligner Accel-Align.

We discovered that the main issue of Accel-Align lays in its index. This hash-based structure returns imprecise sets of candidate locations, thus slowing down the following stages of alignment. Our analysis revealed that, in 84% of the cases, over 80% of the candidate locations turned out to be false positives.

Motivated by this finding, we presented *Accel-Align-RMI*, a version of Accel-Align leveraging the RMI to achieve perfect lookup precision. Additionally, we developed two proof-of-concept variants to explore diverse index tradeoffs.

Accel-Align-RMI followed the predicted pattern—slower lookup times but a notable acceleration in the embedding phase—ultimately surpassing the performance of the classic implementation. However, the XXH variant emerged as the top-performing Accel-Align implementation, coupling a faster lookup time with a greater precision.

In terms of accuracy, all three new versions demonstrated enhancements, with Accel-Align-RMI (and binary) achieving an all-time maximum.

Finally, we investigated the scalability of the four variants. Classic Accel-Align was found to be negatively affected by parallel execution, whereas the other three variants exhibited much greater scalability.

Overall, the RMI did not bring to Accel-Align all the performance benefits claimed by the other aligners. However, it did contribute to enhancing accuracy and paved the way for additional experiments with the aligner's index—the first being XXH.

Bibliography

[1] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. *The Case for Learned Index Structures*. 2018. arXiv: 1712.01208 [cs.DB] (cit. on p. 4).

[2] Yao Tian, Tingyun Yan, Xi Zhao, Kai Huang, and Xiaofang Zhou. *A Learned Index for Exact Similarity Search in Metric Spaces*. 2022. arXiv: 2204.10028 [cs.DB] (cit. on p. 4).

[3] Hussam Abu-Libdeh et al. *Learned Indexes for a Google-scale Disk-based Database*. 2020. arXiv: 2012.12501 [cs.DB] (cit. on p. 4).

[4] Chuzhe Tang, Youyun Wang, Zhiyuan Dong, Gansen Hu, Zhaoguo Wang, Minjie Wang, and Haibo Chen. «XIndex: a scalable learned index for multicore data storage». In: *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '20. San Diego, California: Association for Computing Machinery, 2020, pp. 308–320. ISBN: 9781450368186. DOI: 10.1145/3332466.3374547. URL: https://doi.org/10.1145/3332466.3374547 (cit. on p. 4).

[5] Marcel Maltry and Jens Dittrich. *A Critical Analysis of Recursive Model Indexes*. 2021. arXiv: 2106.16166 [cs.DB] (cit. on pp. 4, 5).

[6] Ibrahim Sabek, Kapil Vaidya, Dominik Horn, Andreas Kipf, Michael Mitzenmacher, and Tim Kraska. «Can Learned Models Replace Hash Functions?» In: *Proc. VLDB Endow.* 16.3 (Nov. 2022), pp. 532–545. ISSN: 2150-8097. DOI: 10.14778/3570690.3570702. URL: https://doi.org/10.14778/3570690.3570702 (cit. on pp. 5, 9, 13, 19, 29, 35).

[7] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. *RadixSpline: A Single-Pass Learned Index*. 2020. arXiv: 2004.14541 [cs.DB] (cit. on pp. 6, 7).

[8] Paolo Ferragina and Giorgio Vinciguerra. «The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds». In: *PVLDB* 13.8 (2020), pp. 1162–1175. ISSN: 2150-8097. DOI: 10.14778/3389133.3389135. URL: https://pgm.di.unipi.it (cit. on p. 6).

[9] Donald Knuth. *The Art Of Computer Programming, vol. 3: Sorting And Searching*. Addison-Wesley, 1973, pp. 391–392 (cit. on p. 8).

[10] Malte Skarupke. *Fibonacci Hashing: The Optimization That the World Forgot*. 2018. URL: https://probablydance.com/2018/06/16/fibonacci-hashing-the-optimization-that-the-world-forgot-or-a-better-alternative-to-integer-modulo/ (cit. on p. 8).

[11] Austin Appleby. *MurmurHash3.cpp*. https://github.com/aappleby/smhasher/blob/master/src/MurmurHash3.cpp. Accessed: February 29, 2024 (cit. on p. 8).

[12] Yann Collet. *xxHash - Extremely fast hash algorithm*. URL: https://xxhash.com/ (cit. on p. 8).

[13] Andrew Rogers. *AquaHash*. https://github.com/jandrewrogers/AquaHash (cit. on p. 8).

[14] B. S. Majewski, N. C. Wormald, G. Havas, and Z. J. Czech. «A Family of Perfect Hashing Methods». In: *The Computer Journal* 39.6 (Jan. 1996), pp. 547–554. ISSN: 0010-4620. DOI: 10.1093/comjnl/39.6.547. eprint: https://academic.oup.com/comjnl/article-pdf/39/6/547/1103380/390547.pdf. URL: https://doi.org/10.1093/comjnl/39.6.547 (cit. on p. 9).

[15] Emmanuel Esposito, Thomas Mueller Graf, and Sebastiano Vigna. *RecSplit: Minimal Perfect Hashing via Recursive Splitting*. 2019. arXiv: 1910.06416 [cs.DS] (cit. on p. 9).

[16] Stefan Richter, Victor Alvarez, and Jens Dittrich. «A seven-dimensional analysis of hashing methods and its implications on query processing». In: *Proc. VLDB Endow.* 9.3 (Nov. 2015), pp. 96–107. ISSN: 2150-8097. DOI: 10.14778/2850583.2850585. URL: https://doi.org/10.14778/2850583.2850585 (cit. on p. 10).

[17] Oliver Knill. *Probability and Stochastic Processes with Applications*. Harvard Mathematics Department, 1994, p. 332 (cit. on p. 16).

[18] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. *SOSD: A Benchmark for Learned Indexes*. 2019. arXiv: 1911.13014 [cs.DB] (cit. on p. 20).

[19] Ibrahim Sabek and Tim Kraska. *The Case for Learned In-Memory Joins*. 2022. arXiv: 2111.08824 [cs.DB] (cit. on p. 35).

[20] Michael Beukman. *Parallel Quicksort using OpenMP*. Medium. 2023. URL: https://mcbeukman.medium.com/parallel-quicksort-using-openmp-9d18d7468cac (cit. on p. 36).

[21] Christopher Jonathan, Umar Farooq Minhas, James Hunter, Justin Levandoski, and Gor Nishanov. «Exploiting Coroutines to Attack the "Killer Nanoseconds"'». In: *Proc. VLDB Endow.* 11.11 (July 2018), pp. 1702–1714. ISSN: 2150-8097. DOI: 10.14778/3236187.3236216. URL: https://doi.org/10.14778/3236187.3236216 (cit. on p. 41).

[22] Ben Langmead and Steven L Salzberg. «Fast gapped-read alignment with Bowtie 2». In: *Nature Methods* 9.4 (Apr. 2012), pp. 357–359. ISSN: 1548-7105. DOI: 10.1038/nmeth.1923. URL: https://doi.org/10.1038/nmeth.1923 (cit. on p. 52).

[23] Heng Li. «Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM». In: (2013). arXiv: 1303.3997 [q-bio.GN] (cit. on p. 52).

[24] Heng Li. «Minimap2: pairwise alignment for nucleotide sequences». In: *Bioinformatics* 34.18 (May 2018), pp. 3094–3100. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/bty191. eprint: https://academic.oup.com/bioinformatics/article-pdf/34/18/3094/48919122/bioinformatics_34_18_3094.pdf. URL: https://doi.org/10.1093/bioinformatics/bty191 (cit. on pp. 52, 64).

[25] Hongyi Xin, Donghyuk Lee, Farhad Hormozdiari, Samihan Yedkar, Onur Mutlu, and Can Alkan. «Accelerating read mapping with FastHASH». In: *BMC Genomics* 14.1 (2013), S13. ISSN: 1471-2164. DOI: 10.1186/1471-2164-14-S1-S13. URL: https://doi.org/10.1186/1471-2164-14-S1-S13 (cit. on p. 52).

[26] Hongyi Xin, John Greth, John Emmons, Gennady Pekhimenko, Carl Kingsford, Can Alkan, and Onur Mutlu. «Shifted Hamming distance: a fast and accurate SIMD-friendly filter to accelerate alignment verification in read mapping». In: *Bioinformatics* 31.10 (Jan. 2015), pp. 1553–1560. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/btu856. eprint: https://academic.oup.com/bioinformatics/article-pdf/31/10/1553/49013086/bioinformatics_31_10_1553.pdf. URL: https://doi.org/10.1093/bioinformatics/btu856 (cit. on p. 52).

[27] Yang Liao, Gordon K Smyth, and Wei Shi. «The Subread aligner: fast, accurate and scalable read mapping by seed-and-vote». In: *Nucleic Acids Res* 41.10 (2013), e108. ISSN: 1362-4962 (Electronic), 0305-1048 (Print). DOI: 10.1093/nar/gkt214. URL: https://doi.org/10.1093/nar/gkt214 (cit. on p. 52).

[28] Yiqing Yan, Nimisha Chaturvedi, and Raja Appuswamy. «Accel-Align: a fast sequence mapper and aligner based on the seed–embed–extend method». In: *BMC Bioinformatics* 22.1 (2021), p. 257. ISSN: 1471-2105. DOI: 10.1186/

s12859-021-04162-z. URL: https://doi.org/10.1186/s12859-021-04162-z (cit. on p. 53).

[29] Diptarka Chakraborty, Elazar Goldenberg, and Michal Koucký. «Streaming algorithms for embedding and computing edit distance in the low distance regime». In: *Proceedings of the Forty-Eighth Annual ACM Symposium on Theory of Computing*. STOC '16. Cambridge, MA, USA: Association for Computing Machinery, 2016, pp. 712–725. ISBN: 9781450341325. DOI: 10.1145/2897518.2897577. URL: https://doi.org/10.1145/2897518.2897577 (cit. on p. 53).

[30] Darryl Ho, Saurabh Kalikar, Sanchit Misra, Jialin Ding, Vasimuddin Md, Nesime Tatbul, Heng Li, and Tim Kraska. «LISA: A Case For Learned Index based Acceleration of Biological Sequence Analysis». In: *bioRxiv* (2023). DOI: 10.1101/2020.12.22.423964. eprint: https://www.biorxiv.org/content/early/2023/11/24/2020.12.22.423964.full.pdf. URL: https://www.biorxiv.org/content/early/2023/11/24/2020.12.22.423964 (cit. on p. 59).

[31] Saurabh Kalikar, Chirag Jain, Md Vasimuddin, and Sanchit Misra. «Accelerating minimap2 for long-read sequencing applications on modern CPUs». In: *Nature Computational Science* 2 (2 2022), pp. 78–83. ISSN: 2662-8457. DOI: 10.1038/s43588-022-00201-8. URL: https://doi.org/10.1038/s43588-022-00201-8 (cit. on p. 59).

[32] Ryan Marcus, Emily Zhang, and Tim Kraska. «CDFShop: Exploring and Optimizing Learned Index Structures». In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. SIGMOD '20. Portland, OR, USA: Association for Computing Machinery, 2020, pp. 2789–2792. ISBN: 9781450367356. DOI: 10.1145/3318464.3384706. URL: https://doi.org/10.1145/3318464.3384706 (cit. on p. 61).