UNIVERSITY OF PALERMO
PHD JOINT PROGRAM:
UNIVERSITY OF CATANIA - UNIVERSITY OF MESSINA
XXXIV CYCLE

DOCTORAL THESIS

A Tour of Learned Static Sorted Sets Dictionaries: From Specific to Generic with an Experimental Performance Analysis

Author:
Domenico Amato

Supervisor:
Prof. Giosuè Lo Bosco

Co-Supervisor:
Prof. Raffaele Giancarlo

A thesis submitted in fulfillment of the requirements for the degree of Doctor of Philosophy

in

Mathematics and Computational Sciences

May 5, 2022

Signed:

Date:

"Predicting the future isn't magic, it's Artificial Intelligence"

Dave Waters

UNIVERSITY OF PALERMO

Abstract

Department of Mathematics and Computer Sciences

Doctor of Philosophy

A Tour of Learned Static Sorted Sets Dictionaries: From Specific to Generic with an Experimental Performance Analysis

by Domenico Amato

In recent years, in the era of Big Data, studying new methods to improve the performance of well-known procedures, such as searching in a Sorted Set, has become crucial in many fields. A new trend emerging in this scenario combines Machine Learning models with Data Structures, generating the so-called Learned Data Structures. In this thesis, we provide an in-depth experimental study of the use of these models, starting from some evidence known to experts in the field but not experimentally investigated concerning the use of very complex models such as Neural Networks. Then, we document a time/space trade-off scenario that is very important for practitioners and designers users. Furthermore, we investigate a comparison well known in the Literature, i.e., Branchy procedures versus Branch-free ones, and we place it in the context of Learned Data Structures. Finally, considering that the Learned Data Structures currently defined in the Literature only fit with specific Dictionaries and procedures, e.g., Binary Search, we have defined a new type of generic Learned Data Structure that can use a wide range of Dictionaries.

Acknowledgements

I would like to thank my Supervisor Prof. Giosuè Lo Bosco as well as my Co-Supervisor Prof. Raffaele Giancarlo for their support, patience and guidance. Part of this work has been funded by MIUR Project of National Relevance 2017WR7SHH "Multicriteria Data Structures and Algorithms: from compressed to learned indexes, and beyond". I would like to thank all the participants in this project for the productive exchange of ideas during meetings. I also acknowledge an NVIDIA Higher Education and Research Grant (donation of a Titan V GPU) essential for the experimental phase of this work.

Contents

List of Figures

List of Tables

List of Abbreviations

BBS	**B**inary Search **B**ranchy on **S**orted Layout
BFE	**B**inary Search Branch-**F**ree on **E**ytzeinger Layout
BFS	**B**inary Search Branch-**F**ree on **S**orted Layout
BFT	**B**inary Search Branch-**F**ree on B-**T**ree Layout
BFV	**B**inary Search Branch-**F**ree on **V**an emde Boas Layout
C	**C**ubic Model
CDF	**C**umulative **D**istribution **F**unction
IBS	**I**nterpolation Search **B**ranchy on **S**orted Layout
K-	**K**-ary Search
L	**L**inear Model
LR	**L**inear Regression Model
MSE	**M**ean **S**quare **E**rror
MLR	**M**achine **L**earning
MLR	**M**ultiple **L**inear **R**egression
NN	**N**eural **N**etwork
PGM	**P**icewise **G**eometric **M**odel
PR	**P**olynomial **Regression**
PSP	**P**redecessor **S**earch **P**roblem
Q	**Q**uadratic Model
RF	**R**eduction **F**actor
RMI	**R**ecursive **M**odel **I**ndex
RS	**R**adix **S**pline
SD	**S**tatic **D**ictionary
SOSD	**S**earch **O**n **S**orted **D**ata
SY-RMI	**SY**noptic **R**ecursive **M**odel **I**ndex
SLR	**S**imple **L**inear **R**egression
TIP	**T**hree **P**oints **I**nterpolation Search

To those who will always be here
even without being present...

Introduction

Searching for elements in a sorted set is a well-known problem in the Literature. It has been studied widely over the years, and it is considered one of the classic problems for the introduction to Data Structures. However, in recent years, the study of classic Data Structures with the aim of achieving time/space improvements has become increasingly important. For example, in the era of Big Data, a growing number of applications, such as Search Engines (Khuong and Morin, 2017), Decision Support System in Main Memory (Rao and Ross, 1999) or Bidding Systems for advertising (Khuong and Morin, 2017), require efficient search algorithms to be applied to a wide variety of data.

0.1 State of the Art

Starting from the work of Kraska et al., 2018, extending the one by Ao et al., 2011, a new trend has emerged that combines Machine Learning techniques with ones proper of Data Structures. This approach involves using the former to automatically exploit particular patterns in the data simulating an Index Structure that admits some errors in its prediction to improve the performance of classic search procedures. This new area goes under the name of Learned Data Structures and it has been growling very quickly (as illustrated in Ferragina and Vinciguerra, 2020a). this approach is applied to a variety of Data Structures and Algorithms, e.g. Learned Bloom Filter initially described by Mitzenmacher, 2018.

One motivation behind their rapid development may lie, as outlined by Kraska et al., 2018, in a perceived paradigm shift away from the so far used CPU and towards the use of GPUs and TPUs, pushing the use of advanced machine learning techniques, such as Neural Networks (Goodfellow, Bengio, and Courville, 2016a), and highly engineered platforms (Abadi et al., 2015). In fact, these architectures make it possible to parallelise the mathematical operations of Machine Learning models, reinforcing the paradigm's shift from if-then-else constructs, characteristic of classic algorithms, to one based on simple arithmetic operations.

0.2 Our Contributions

In this Thesis, we focus on searching in a Static Dictionary over Sorted Set using Learned Data Structure. In particular, in Chapter 1, we start by providing a definition of Static Dictionary Data Structures over Sorted Sets and describing the well-know Predecessor Search Problem. Then, we illustrate the main classic algorithms used in this Thesis to solve it. Finally, we reduce this problem to a Learning-Prediction one, introducing Learned Indexes. In particular, in this Thesis, our contributions are organized in the next Chapters as follow.

- In Chapter 2, starting from the consideration made by Kraska et al., 2018, we illustrate how a perceived change of paradigm in computer architecture is one

of the motivation for introducing Learned Indexes. Furthermore, we provide a first experimental evidence on the use of Neural Networks as Learned Indexes through the comparison with simpler models.

- In Chapter 3, we examine the advantage of using both simple model, e.g., Regression, and sophisticated ones, which organize different models in a hierarchy, e.g., **RMI** (Kraska et al., 2018), **PGM** (Ferragina and Vinciguerra, 2020b) and **RS** (Kipf et al., 2020), to improve classic search procedures. Then, considering the existence of time/space trade-off for the Predecessor Search Problem (as widely discussed by Pătraşcu and Thorup, 2006), we study if it is possible to improve those classic search algorithms using small or constant space with respect to the Table size, showing that, even imposing a space bound on the most complex model, we can achieve this improvement. In addition, we provide two new kinds of models that respect an imposed space bound.

- In Chapter 4, we discuss the difference between Standard and Uniform Search, observing how these can impact the performance of a Learned Index. These two procedures, introduced and discussed by Knuth, 1973, have been studied experimentally in the Literature on synthetic datasets by Khuong and Morin, 2017, who refer to them as Branchy and Branch-free, respectively. So, we extend this study by testing these two versions of Binary Search on real datasets. Then, we analyse their use as final search stage of Learned Indexes. In addition, we apply the same experimental procedure to the Standard and Uniform versions of k-ary Search.

- In Chapter 5, following an analysis of several features of Learned Indexes that operate on a specific type of Dictionaries, due to their natural fit with the sorted table layout, we define a new generic type of Learned Dictionary for Static Sorted Search, able to use as its final search stage various types of Dictionary, also different from sorted layout. These models base their idea on making explicit the partition that is normally implicit in Learned Indexes present so far in the Literature. In particular, we provide two instances of this Generic Learned Dictionary, mentioning some construction and query bounds. The first is characterised by a partition with fixed-length intervals derived from a generalisation of the method presented by Demaine, Jones, and Pătraşcu, 2004 for solving Interpolation Search for non-independent data. The other is a partition with variable-length intervals derived from the construction of a **PGM** Index. Then, we analysed the "boosting effect" for each of them on a variety of Data Structures and we compare these Generic Learned Dictionaries with the specific ones.

- In the Conclusions Chapter, we summarize the major contributions of this Thesis and underline the future direction for this work.

Chapter 1

Learned Data Structures

This Chapter introduces the Predecessor Search Problem (**PSP** for short), a well-known problem in the Literature, and presents which of the main classic algorithms are used to solve it. Then, it is shown how to reduce the **PSP** to a Learning-Prediction problem, introducing Learned Indexes. In particular, this Chapter is organized as follows.

- In Section 1.1, we present the definition of Static Dictionary Data Structures and we describe the classic **PSP** operation
- In Section 1.2, we describe classic algorithms on arrays and trees for searching in a sorted set.
- In Section 1.3, we preliminarily introduce a fundamental routine for Learning-Prediction problems, i.e. Linear Regression.
- In Section 1.4, we finally reduce the **PSP** to a Learning-Prediction problem, and we describe the most relevant Learned indexes in the Literature.
- In Section 1.5, we summarise the contents of the previous Sections.

The Bibliographic Notes relevant for this Chapter are in the corresponding Section.

1.1 Static Dictionary Data Structures over Sorted Sets

Given an universe U of integers, on which it is defined a total order relation, a *static sorted sets dictionary*) $\mathcal{SD}$ is a Data Structure that supports the following operations over a sorted set A:

1. $member(x) = TRUE$ if $x \in A$, otherwise $FALSE$.
2. $PSP(x) = \max\{y \in A | y < x\}$, i.e. Predecessor Search.
3 $range(x, y) = A \cap [x, y]$.

From now on, for brevity, we refer to $\mathcal{SD}$ simply as Dictionary.

In what follows, we focus on **PSP**, described more in-depth in the following Section, since all of the Dictionaries considered in this Thesis efficiently support range queries.

1.1.1 Predecessor Search Problem

Consider a $\mathcal{SD}$, as defined in the previous Section. The membership problem is defined as follows: for a given element $x \in U$, provide as answer "yes" if $x \in A$, and "no" otherwise.

Considering a Dictionary again, it is possible to extend the previous problem with the definition of the so-called Predecessor Search Problem, which, given a query element x, gives as answer an $A[j]$ such that $A[j] \leq x \leq A[j+1]$.

It is possible to solve both problems on an ordered set via the primitive operation $rank(x)$, which returns the number of elements in A less than or equal to x. In this way, the operation $membership(x)$ consists of evaluating the expression $A[rank(x)] == x$ and the operation $predecessor(x)$ simply returns $A[rank(x) - 1]$.

1.2 Standard Algorithmic Tools for PSP: Arrays and Trees

A description of the classic algorithms for searching in ordered tables is provided in this Section. We can distinguish among:

1. Search methods that act directly on the ordered table, i.e. Binary Search, Interpolation Search and k-ary Search (see Section 1.2.1).
2. Search methods that use a particular transformation of the table, other than sorted, i.e. Array Layouts (see Section 1.2.2).
3. Search methods for data stored in structures more complex than arrays, i.e. Self-Adjusting Binary Trees, B^+-Trees and CSS-Trees (see Section 1.2.3).

1.2.1 Sorted Table Search

Binary Search

Binary Search is the most well-known algorithm used to search in a sorted table. There are two main versions of it. The standard one, that makes use of if-then-else constructs within the while loop (Algorithm 1, rows 4 to 10), and the one that goes under the name of Uniform Binary Search (Algorithm 2), that executes the entire loop even if the query element is found. It is to be remarked that, in the Literature, these two routines are, respectively, also referred to as Branchy (denoted in what follow with **BBS**) and Branch-free (denoted with **BFS**) Binary Search. Throughout this Thesis, we adhere to this formalism.

The denomination Branch-free for Algorithm 2 comes from the fact that the compiler translated the if-then-else construct in line 8 in Assembler as a conditional instruction, which avoids altering the stream of Assembly code generated. In this way, better use of instruction pipelining is obtained. This aspect is discussed in more detail in Section 4.1. In instructions 6 and 7 of Algorithm 2, there is a prefetching procedure that loads data into the cache memory before it is used, to improve the performance of Binary Search over its Branchy counterpart.

Interpolation Search

The Standard Interpolation Search procedure (**IBS** for short), described in Algorithm 3, has proved to be highly effective in practice for searching in ordered tables, whose data are drawn from the universe via a Uniform distribution. The algorithm involves

Algorithm 1 C++ Implementation of Standard Binary Search.

```
1:  int StandardBinarySearch(int *A, int x, int left, int right){
2:      while (left < right) {
3:          int m = (left + right) / 2
4:          if(x < A[m]){
5:              rigth = m;
6:          }else if( x > A[m]){
7:              left = m+1;
8:          }else{
9:              return m
10:         }
11:       }
12:       return right;
13: }
```

Algorithm 2 C++ Implementation of Uniform Binary Search with prefetching.

```
1:  int UniformBinarySearch(int *A, int x, int left, int right){
2:      const int *base = A;
3:      int n = right;
4:      while (n > 1) {
5:          const int half = n / 2;
6:          __builtin_prefetch(base + half/2, 0, 0);
7:          __builtin_prefetch(base + half + half/2, 0, 0);
8:          base = (base[half] < x) ? &base[half] : base;
9:          n -= half;
10:     }
11:     return (*base < x) + base - A;
12: }
```

determining, by applying a straight line that approximates the distribution of the data, a pivot point (instruction 8) that divides the table into two complementary intervals, one of which is chosen to perform the next iteration. Unlike Binary Search, this division does not necessarily generate intervals of equal cardinality.

Algorithm 3 C++ Implementation of Standard Interpolation Search.

```
1:  int StandardInterpolationSearch(int *arr, int x, int start, int end){
2:      int lo = start, hi = (end - 1);
3:      while (lo ≤ hi && x ≥ arr[lo] && x ≤ arr[hi]) {
4:          if (lo == hi) {
5:              if (arr[lo] == x) return lo;
6:              return -1;
7:          }
8:          int pos = lo + (((double)(hi - lo) / (arr[hi] - arr[lo])) * (x - arr[lo]));
9:          if (arr[pos] == x) return pos;
10:         if (arr[pos] < x) lo = pos + 1;
11:         else hi = pos - 1;
12:     }
13:     return pos;
14: }
```

An additional implementation of Interpolation Search is the Three-Points Interpolation Search (**TIP** for short) described in Algorithm 4. Unlike Standard Interpolation Search, this version uses a 3-points Interpolation method to interpolate efficiently a variety of distributions that are not uniform.

K-ary Search

The k-ary Search is a generalisation of Binary Search. Once chosen a $k \geq 2$, it consists of two iterated steps, as follows.

(1) the sorted table is divided into k parts of the same size.

(2) at most $k-1$ checks are performed in order to find the correct range where to iterate next, until the element is found.

It is to be noted that $k = 2$ corresponds to a Standard Binary Search and that the time complexity of a k-ary Search is $log_k(n)$.

As for Binary Search described in the previous Section, k-ary Search can be implemented in two versions, i.e., the Standard one, as in Algorithm 5, and the Uniform one, as in Algorithm 6. In analogy with Uniform Binary Search, lines 9 and 10 of Algorithm 6 are translated by the compiler into "conditional moves", which avoids the generation of branches. For that reason, in the following, we refer to the Standard k-ary Search as Branchy (denoted with **K-BBS**) and to the Uniform one as Branch-free (denoted with **K-BFS**).

1.2.2 Array Layouts Other Than Sorted

Array Layouts organise Table elements as if they were stored in a virtual tree, applying the appropriate search algorithms. We now present the three most well-known Array Layouts.

Algorithm 4 C++ Implementation of Three Points Interpolation Search.

```
1: int TipSearch(int *arr, int x, int start, int end){
2:    int mid = end/2;
3:    double y0 = A[left] - x;
4:    double y1 = A[mid] - x;
5:    double y2 = A[right] - x;
6:    double num = (y1*(mid-left)*(1+(y0-y1)/(y1-y2)));
7:    double den = y0-y2*((y0-y1)/(y1-y2));
8:    int expected = mid + num / den;
9:    while (left < right && left ≤ expected && right ≥ expected){
10:      if(abs((double)expected - (double)mid) < guard){
11:         return sequential(A, x, left, right);
12:      }
13:      if(A[mid] != A[expected]){
14:         left = mid;
15:      }else{
16:         right = mid;
17:      }
18:      if(expected+guard ≥ right || expected-guard ≤ left){
19:         return sequential(A, x, left, right);
20:      }
21:      mid = expected;
22:      y0 = A[left] - x;
23:      y1 = A[mid] - x;
24:      y2 = A[right] - x;
25:      num = (y1*(mid-right)*(mid-left)*(y2-y0));
26:      den = (y2*(mid-right)*(y0-y1)+y0*(mid-left)*(y1-y2));
27:      expected = mid + num/den;
28:   }
29: }
```

Algorithm 5 C++ Implementation of Standard k-ary Search.

```
1: int StandardKarySearch(int *arr, int x, int start, int end, int k){
2:    int left = start, right = end;
3:    while (left < right){
4:       int segLeft = left;
5:       int segRight = left + (right - left) / k;
6:       for (int i = 2; i ≤ k; ++i){
7:          if (x ≤ arr[segRight]) break;
8:          segLeft = segRight + 1;
9:          segRight = left + (i * (right - left)) / k;
10:      }
11:      left = segLeft;
12:      right = segRight;
13:   }
14:   return left;
15: }
```

Algorithm 6 C++ Implementation of Uniform k-ary Search.

```
1:  int UniformKarySearch(int *arr, int x, int start, int end, int k){
2:    int left = start, right = end;
3:    while (left ≤ right){
4:      int segLeft = left;
5:      int segRight = left + (1 * (right - left)) / k;
6:      for (int i = 2; i ≤ k; ++i){
7:        int nextSeparatorIndex = left + (i * (right - left)) / k;
8:        segLeft = x > arr[segRight] ? segRight + 1 : segLeft;
9:        segRight = x > arr[segRight] ? nextSeparatorIndex : segRight;
10:     }
11:     left = segLeft;
12:     right = segRight;
13:   }
14:   return left;
15: }
```

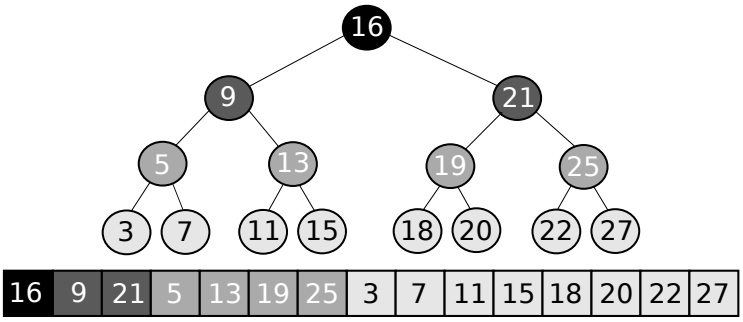


FIGURE 1.1: **An Example of Eytzinger Layouts** (see also Khuong and Morin, 2017). The sorted table is seen as stored a balanced Binary Search Tree. Then, such a tree is laid out in Breadth-First Search order in the array.

Eytzinger Layout

The sorted table is thought of as stored in a virtual complete balanced Binary Search Tree. Such a tree is laid out in Breadth-First Search order in an array. An example is provided in Fig. 1.1. We adopt a Uniform version with prefetching of the Binary Search procedure corresponding to this layout. It is reported in Algorithm 7 and denoted in what follows as **BFE**.

B-Tree Layout

The sorted table is thought of as stored in a B+1 Search Tree, which is laid out in Breadth-First Search order in an array. An example is provided in Fig. 1.2. We adopt a Uniform version with prefetching of the Binary Search procedure corresponding to this layout. It is reported in Algorithm 8 and denoted in what follows as **BFT**.

Van Emde Boas Layout

The table is organised starting from a complete Binary Search Tree and constructed recursively. In particular, we consider n the number of elements and h the height of

Algorithm 7 C++ Implementation of Uniform Binary Search with Eyzinger layout and prefetching.

```
1:  int UniformEytzingerSearch(int *A, int x, int left, int right){
2:      int i = 0;
3:      int n = right;
4:      while (i < n){
5:          __builtin_prefetch(A+(multiplier*i + offset));
6:          i = (x ≤ A[i]) ? (2*i + 1) : (2*i + 2);
7:      }
8:      int j = (i+1) >> __builtin_ffs(~(i+1));
9:      return (j == 0) ? n : j-1;
10: }
```

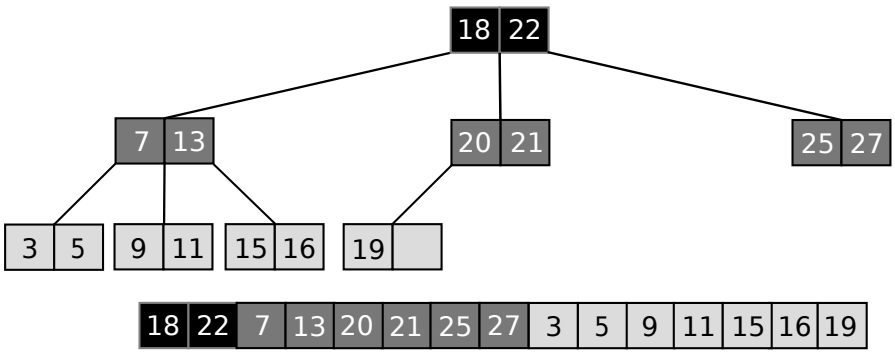


FIGURE 1.2: **An Example of B-Tree Layout** (see also Khuong and Morin, 2017). The sorted table is thought as stored in a B-Tree with $B = 2$. Then, such a Tree is laid out in Breath-First Search order in the array.

Algorithm 8 C++ Implementation of Uniform Binary Search with B-Tree layout and prefetching.

```
int UniformBTreeSearch(int* a, int x, int left, int right){
    int j = right;
    int i = left;
    int n = right - left +1;
    int B = cacheline/sizeof(T);
    while (i + B ≤ n) {
        __builtin_prefetch(a+child(i, B/2, B), 0, 0);
        const int *base = &a[i];
        const int *pred = uniform_inner_search(base, x, B);
        int nth = (*pred < x) + pred - base;
        {
            const int current = base[nth % B];
            int next = i + nth;
            j = (current ≥ x) ? next : j;
        }
        i = child(nth, i, B);
    }
    if (__builtin_expect(i < n, 0)) {
        const int *base = &a[i];
        int m = n - i;
        while (m > 1) {
            int half = m / 2;
            const int *current = &base[half];
            base = (*current < x) ? current : base;
            m -= half;
        }
        int ret = (*base < x) + base - a;
        return (ret == n) ? j : ret;
    }
    return j;
}

int* uniform_inner_search(int *base, int x, int C) {
    if (C ≤ 1) return base;
    const int half = C / 2;
    const T *current = &base[half];
    return uniform_inner_search((*current < x) ? current : base, x, C-half);
}
```

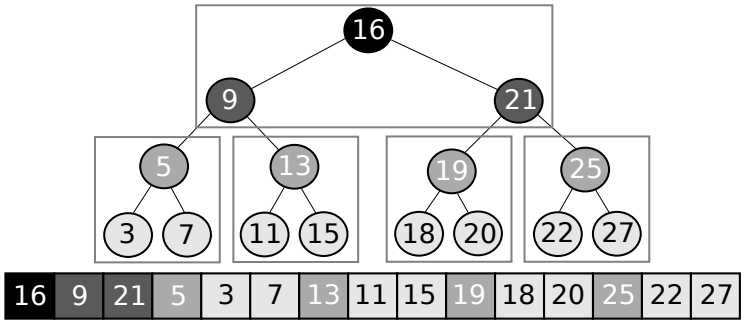


FIGURE 1.3: **An Example of Van emde Boas Layout** (see also Khuong and Morin, 2017). The table is seen as a complete Binary Search Tree. Then, starting from the root, the layout is constructed recursively.

the tree, if $n = 1$ the element is stored in $A[0]$, otherwise the top part of the tree of height $\lfloor h/2 \rfloor$ is stored recursively in the layout at location $A[0....,2^{1+\lfloor h/2 \rfloor} - 2]$. All leaves of this upper tree are at most $2^{1+\lfloor h/2 \rfloor}$ and the same procedure is applied on them recursively from left to right. An example is given in Fig. 1.3. The search algorithm is illustrated in Algorithm 9 and denoted in what follows as **BFV**. It should be noted that the extensive experiments conducted in the Literature indicate that such a layout may be superior to the ones mentioned earlier only occasionally and on large datasets. Due to such inconsistency in performance, it has been described here for the sake of completeness, but it is not included in the experimental part of this Thesis.

1.2.3 Search Trees

Self-adjusting Binary Trees

A Self-adjusting Binary Tree is a Binary Search Tree, with a standard search procedure, except for the Splay operations (see Algorithm 10) that move the accessed node to the root.

In more detail, if p and g denote, respectively, the parent and the grandparent of the accessed node x, the possible operation in a splay step consist of Zig, Zig-Zig and Zig-Zag steps, each of those convolves Rotation and Double Rotations around p and g. We do not give the details here.

Splay Step ensures that basic operations are performed in $O(logn)$ amortised time.

B^+-Trees

A B^+-Tree (as in Fig. 1.4) is a variant of a classic B-Tree, with the following features:

(1) All nodes have at most B children.

(2) All non-leaves nodes (except root) have at least $\lceil B/2 \rceil$ children.

(3) The root has at least two internal child nodes or at least one leaves child node.

(4) All non-leaves nodes with k children contain $k - 1$ keys and k pointers (as shown in Fig. 1.5).

(5) The leaves are connected as in a linked list called sequence set.

Algorithm 9 C++ Implementation of Standard Binary Search with Van emde Boas layout.

```
int StandardVanEmdeBoasSearch(int *a, int x, int n, dumdum *s) {
    int rtl[MAX_H+1];
    int j = n;
    int i = 0;
    int p = 0;
    for (int d = 0; i < n; d++) {
        rtl[d] = i;
        if (x < a[i]) {
            p <<= 1;
            j = i;
        } else if (x > a[i]) {
            p = (p << 1) + 1;
        } else {
            return i;
        }
        i = rtl[d-s[d].h0] + s[d].m0 + (p&s[d].m0)*(s[d].m1);
    }
    return j;
}


 struct dumdum {
     unsigned char h0, h1, dummy[2];
     int m0, m1;
 };
```

Algorithm 10 C++ Implementation of the Splay Step in a Self-adjusting Binary Tree

```
void splay(TreeNode *x) {
    while (x -> parent != NULL){
        TreeNode *p = x -> parent;
        TreeNode *g = p -> parent;
        if (g == NULL) zig(x);
        else if (g -> left == p && p -> left == x) zig_zig(x);
        else if (g -> right == p && p -> right == x) zig_zig(x);
        else zig_zag(x);
    }
    this -> root = x;
};
```

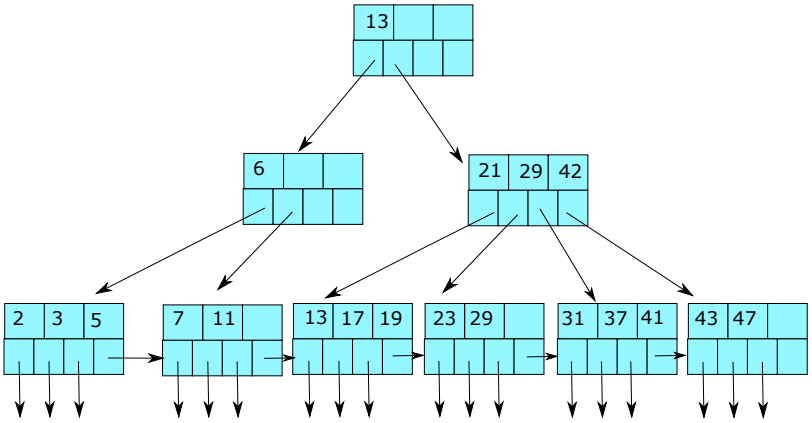


FIGURE 1.4: **An Example of B$^+$-Tree**. The table $A = [2, 3, 5, 7, 11, 15, 17, 19, 23, 29, 31, 37, 41, 43, 47]$ is stored in a B$^+$-Tree with $B = 4$. All table keys are stored in the leaves, connected in a linked list. Even, the internal node contains only the indexing structures

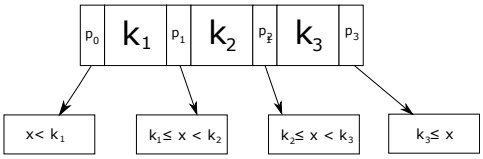


FIGURE 1.5: **An Example of an internal B$^+$-Tree Node wit B=4**. The internal node contains (a) keys k_i that represent different data sub-interval, and (b) pointers to the nodes of the lower level, corresponding to the selected sub-interval.

The search procedure of an element x consists of the following. Starting from the root and at each node, the pointer is chosen relative at the key k_i, such that $k_i \leq x < k_{i+1}$. Once a leaf is found, a Sequential Search on a linked list of at most B-1 keys is performed.

In particular, B$^+$-Trees perform better than the classic B-Tree counterpart because removing the pointer to data inside the internal node reduces context switching and cache misses. In addition, deletions and insertions can often be performed by changing only leaves and not the internal indexing structures of the tree.

CSS-Trees

A CSS-Tree is a complete k-ary Search Tree stored inside an array A that contains directory structures at the top and data at the end (as shown in Fig. 1.6). It is built so that it is possible to determine for each node its children just by calculating an interval offset with the following formulas: Given a node b, A store all children of b between the indexes $\lfloor \frac{b}{k} \rfloor * (k+1) + 1$ and $\lfloor \frac{b}{k} \rfloor * (k+1) + k + 1$.

A search procedure in a CSS-Tree starts from the root and performs a Binary Search to find the correct branch among all node's children. Then, it repeats this passage within the new branches interval until a leaf node, which maps a sorted table portion, is found. Finally, a Binary Search in the found portion is performed.

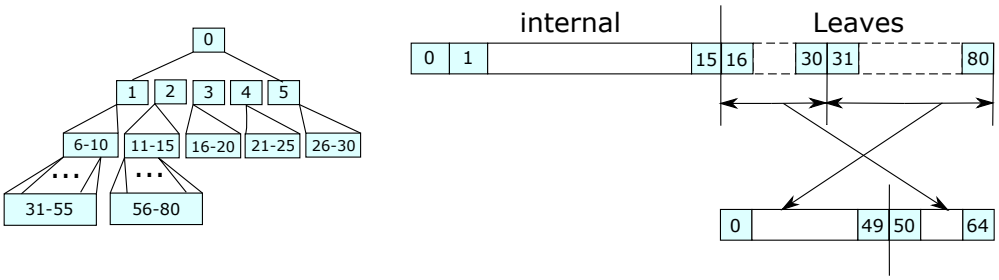


FIGURE 1.6: **An example of a CSS-Tree** (see also Rao and Ross, 1999). Starting from a Table in a k-ary Search Tree with $k = 4$, it is possible laid it out in an array with two section: (a) an initial part with only indexing structures,i.e. the internal nodes of the tree, and (b) an ending part with the leaves, containing the keys of the Table.

This procedure can provide better performance than simple Binary Search because, by choosing k smaller than the cache size, each search routine is performed without slowdown caused by context changes and cache misses.

1.3 Linear Regression

Regression Analysis is a methodology for approximating a function $F : R^m \to R$ into a function $\tilde{F}$. The independent variable $x \in R^m$ is referred to as predictor and the dependent variable y is referred to as outcome. The parameters of $\tilde{F}$ are determined through the minimization of an error function, the most commonly used of which is the Mean Square Error (**MSE** for short).

Linear regression (**LR** for short) is the case when a geometric linear form is assumed as model. In the case when $m = 1$, it is referred to as Simple Linear Regression (**SLR** for short) and as Multiple Linear Regression (**MLR** for short), otherwise.

For the general case of **LR**, given a sample set of n predictor-outcome couples $(\mathbf{x}_i, y_i)$, where $\mathbf{x}_i \in \mathbb{R}^m$ and $y_i \in \mathbb{R}$, the goal is to characterize the linear function model $\tilde{F}(\mathbf{x}) = \hat{\mathbf{w}}\mathbf{x}^T + \hat{b}$ by estimating the parameters $\hat{\mathbf{w}} \in \mathbb{R}^m$ and $\hat{b} \in \mathbb{R}$, using the sample set. We can define a matrix $\mathbf{Z}$ of size $n \times (m+1)$ (usually referred to as the design matrix), where $\mathbf{Z}_i$ is the i-th row of $\mathbf{Z}$ such that $\mathbf{Z}_i = [\mathbf{x}_i, 1]$. Moreover, $\mathbf{y}$ indicates the vector of size n such that the outcome y_j is its j-th component. The Mean Square Error minimization on the basis of the estimation is:

$$\mathbf{MSE}(\mathbf{w}, b) = \frac{1}{n} \left\| [\mathbf{w}, b]\mathbf{Z}^T - \mathbf{y} \right\|_2^2 \tag{1.1}$$

MSE is a convex quadratic function on $[\mathbf{w}, b]$, so that the unique values that minimize it can be obtained by setting its gradient $\nabla_{\mathbf{w},b}$ equal to zero. The closed form solution for the parameters $\mathbf{w}, b$ is

$$[\hat{\mathbf{w}}, \hat{b}] = \mathbf{y}\mathbf{Z}(\mathbf{Z}^T\mathbf{Z})^{-1} \tag{1.2}$$

It is to be noted that the **SLR** case is characterized by the choice of a polynomial of degree $g = 1$. The general case of Polynomial Regression (**PR** for short), using polynomials with degree $g > 1$, are special cases of **MLR**. Indeed, we can consider the model:

$$\tilde{F}(\mathbf{z}) = \sum_{i=1}^{g} w_i x^i + b = \mathbf{w}\mathbf{z}^T + b,$$

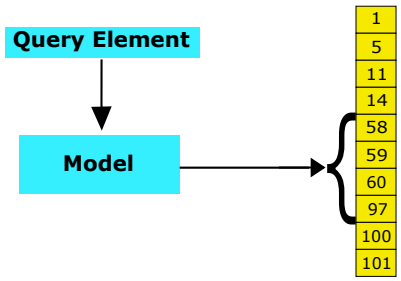


FIGURE 1.7: **A general paradigm of Learned Searching in a Sorted Set** (see also **Marcus20**). The model is trained on the data in the table. Then, given a query element, it is used to predict the interval in the table where to search (included in brackets in the figure).

where w is of size g, $\mathbf{z} = [x, .., x^{g-1}, x^g] \in \mathbb{R}^g$ is the predictor vector for **MLR**.

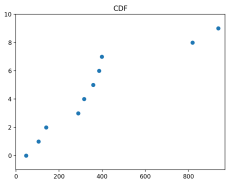
1.4 Learned Index Models

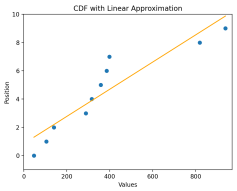
1.4.1 A simple view of Learned Search in Sorted Set

Consider a sorted table A of n keys, taken from a universe U. As shown in Section 1.1.1, Sorted Table Search reduces to Predecessor Search. In turn, such a problem can be transformed into a Learning-Prediction one, as follows. Regarding Fig. 1.7, the model learned from the data is a predictor of where a query element may be in the table. Once such a prediction is made, Binary Search is performed only on the interval returned by the model.

We now outline the basic technique that one can use to build a model for A. It relies on Linear Regression (more details are described in Section 1.3), with Mean Square Error Minimization. Consider the empirical *cumulative distribution function* (CDF for short) of table A, which maps elements to their relative position within the table. It is reminiscent of the CDF over the universe U. With reference to the example in Fig. 1.8, and assuming that one wants a linear model, i.e., $F(x) = ax + b$, it is possible to fit a straight line to the CDF and then use it to predict where a point x may fall in terms of rank and accounting also for approximation errors. More generally, to perform a query, the model is consulted and an interval in which to search for is returned. Then, to fix ideas, Binary Search on that interval is performed. Different models may use various schemes to determine the required range, as outlined in Section 1.4.2.

For this Thesis, it is essential to know how much of the table is discarded once the model makes a prediction on a query element. For instance, Binary Search, after the first test, discards 50% of the table. Because of the diversity across models to determine the search interval and to place all models on a par, we estimate the Reduction Factor (**RF** for short) of a model, i.e., the percentage of the table that is no longer considered for searching after a prediction, empirically. That is, with the use of the model and over a batch of queries, we determine the length of the interval to search into for each query. Based on it, it is immediate to compute the Reduction Factor for that query. Then, we take the average of those Reduction Factors over the entire set of queries as the Reduction Factor of the model for the given table.

(a) 

(b) 

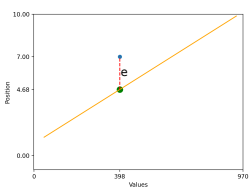
(c) 

FIGURE 1.8: **The Process of Learning a Simple Model via Linear Regression.** Let A be $[47, 105, 140, 289, 316, 358, 386, 398, 819, 939]$. (a) The CDF of A. In the diagram, the abscissa indicates the value of an element in the table, while the ordinate is its rank. (b) The straight line $F(x) = ax + b$ is obtained by determining a and b via Linear Regression, with Mean Square Error Minimization. (c) The maximum error ϵ one can incur is using F also important. In this case, it is $\epsilon = 3$, i.e., accounting for rounding it is the maximum distance between the rank of a point in the table and its rank as predicted by F. In this case, the interval to search into, for a given query element x, is given by $[F(x) - \epsilon, F(x) + \epsilon]$.

1.4.2 Model Classes Characterizing Model Space

Except for the ones operating on table layouts different from sorted and the Trees, all procedures mentioned in Section 1.2 have a natural Learned version. For each, its time and space performances depend critically on the model used to predict the interval to search into. Here we distinguish among four model classes characterising model space.

The first class consists of models that use constant space, while the other three consist of models that use space as a function of some model parameters. For each of those models, we determine the Reduction Factor as described in Section 1.4.1.

Atomic Models: One Level and no Branching Factor

- **Simple and Multiple Regression**. In this Thesis, we use linear, quadratic and cubic regression models. Each can be thought of as an atomic model in the sense that it cannot be divided into sub-models. Fig. 1.9(a) provides an example. In particular, the corresponding learned methods are prefixed by **L, Q,** or **C**, so that, for instance, **L-BFS** denotes the Branch-free version of Branch-free Binary Search with a linear model to restrict the search interval.

Two-Level RMIs with Parametric Branching Factor

- **Heuristically Optimized RMIs**. Informally, a Recursive Model Index (denoted with **RMI**) is a multi-level, directed graph, with Atomic Models at its nodes. When searching for a given key and starting with the first level, a prediction at each level identifies the model of the next level to use for the next prediction. This process continues until a final level model is reached. This latter is used to predict the interval to search into. An example is provided in Fig. 1.9(b). It is to be noted that Atomic Models are **RMI**s.

Multi-Level Models with Various Parameters

- **Neural Networks**. Another method to learn a function F is to use Neural Nets (**NN**s for short). In particular, we focus on feed-forward neural networks

where the general strategy consists of an iterative training phase during which an improvement of the $\tilde{F}$ approximation is made. Starting from an initial approximation $\tilde{F}_0$, at each step i, given a subset of data as input, an attempt is made to minimize an error function E so that $E(\tilde{F}_{i-1}) \geq E(\tilde{F}_i)$. A series of steps using the entire dataset is called epoch. The process can stop after a fixed number of epochs, or when, given a tolerance δ, $\mid E(\tilde{F}_{i-1}) - E(\tilde{F}_i) \mid \leq \delta$.

To perform a Regression using a **NN**, we need to represent the input integer as a string containing its 64-bit binary representation, and we call this vector $\vec{x}$. The characteristics of a **NN** are:

1. **ARCHITECTURE TOPOLOGY**:
   (a) The atomic element of **NN** is called Neuron. As described in Fig. 1.9(c).
   (b) The Number of Hidden Layers H.
   (c) for each hidden layer h_i, its neurons number n_{h_i}.
   (d) the connection between each layer. In our case a Fully connected Neural Net is used, i.e. each neuron of layer h_i is connected with each neuron of next layer h_{i+1}.
2. **THE LEARNING ALGORITHM**:
   (a) The error function E, that is used to measure how close is $\tilde{F}$ to F.
   (b) The gradient descent iterative process that starts from a $\tilde{F}_0$ and, at each step, better approximates F reducing E.

CDF Approximation-Controlled Models

- **Piecewise Geometric Model** (**PGM**). It is a multi-stage model, built bottom-up and queried top-down. It uses a user-defined approximation parameter ϵ that controls the prediction error at each stage. With reference to Fig. 1.9(d), the table is subdivided into three pieces. A prediction in each piece can be made via a linear model guaranteeing an error of ϵ. A new table is formed by selecting the minimum values in each of the three segments. This new table is possibly again partitioned into pieces, in which a linear model can make a prediction within the given error. The process is iterated until only one linear model suffices, as in the case in the Figure. A query is processed via a series of predictions, starting at the root of the tree.

- **Radix Spline** (**RS**). It is a two-stage model. It also uses a user-defined approximation parameter ϵ. With reference to Fig. 1.9(e), a spline curve approximating the CDF of the data is built through a greedy algorithm that, by scanning the points of the table, introduces a spline point in the model when the new point added to the range no longer respects the error parameters provided as input. Then, the radix table is used to identify spline points to use to refine the search interval.

1.5 Conclusions

In this Chapter, we have introduced a well-known problem in the Literature, and we have provided an overview of the main techniques to solve it. In particular, we have described both classic solutions from the Literature, as shown in Section 1.2,

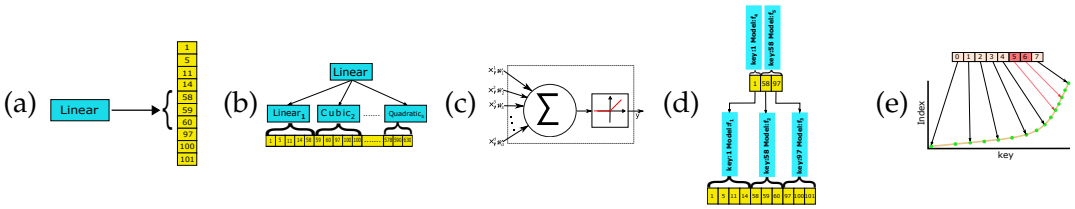


FIGURE 1.9: **Examples of various Learned Indexes**. (a) an Atomic Model, where the box linear means that the CDF of the entire dataset is estimated by a linear function via Regression, as exemplified in Figure 1.8. (b) An example of an **RMI** with two layers and branching factor equal to b. The top box indicates that the lower models are selected via a linear function. As for the leaf boxes, each indicates which Atomic Model is used for prediction on the relevant portion of the table. (c) An Example of the structure of a **NN**'s Neuron. The inputs are binary vector $(x^1, ..., x^d)$ and the weight parameter $(w^1, ..., w^d)$ are a vector of floating points. The final output is $y = max(0, \sum_{i=1}^{d} w^i x^i)$. (d) An example of a **PGM** Index. At the bottom, the table is divided into three parts. A new table is so constructed and the process is iterated. (e) An example of an **RS** Index. At the top, the buckets where elements fall, based on their three most significant digits. At the bottom, a linear spline approximating the CDF of the data, with suitably chosen spline points. Each bucket points to a spline point so that, if a query element falls in a bucket (say six), the search interval is limited by the spline points pointed to by that bucket and the one preceding it (five in our case).

and new methods based on a new Learning-Prediction approach, as in Section 1.4. For these latter, as described in Section 1.4.2, we have also distinguished four classes depending on the space that the model used.

Bibliographic Notes

The PSP is a well-known and well-studied problem in Computer Science at the origin of Algorithmic Design and Analysis (Knuth, 1973 and Aho, Hopcroft, and Ullman, 1974). Over the years, it has been extensively studied, e.g., Pătraşcu, 2008 and Mehlhorn and Tsakalidis, 1990. Uniform Binary Search was introduced by Knuth, 1973 and the terminology Branch-free to refer to it by Khuong and Morin, 2017. Array Layouts, other than sorted, have been extensively studied in the Literature, particularly by Khuong and Morin, 2017, from which we take all the implementations. The interested reader can find relevant references to Array Layouts in Khuong and Morin, 2017. The advantage of using prefetching is discussed for the first time by Prokop, 1999 within Cache Oblivious Algorithms. Such an advantage is confirmed by Khuong and Morin, 2017 within Binary Search procedures. Apparently, the straightforward generalisation of Binary Search to k-ary has been considered for the first time by Schlegel, Gemulla, and Lehner, 2009. We use the Branch-free implementation with prefetching provided by Schulz, Broneske, and Saake, 2018. Classic Interpolation Search was introduced by Peterson, 1957 and it has been extensively studied (see Mehlhorn and Tsakalidis, 1990). The **TIP** heuristic has been proposed by Van Sandt, Chronis, and Patel, 2019. Binary Search Trees are well-known Data Structures in Computer Science. Its generalisation with at most B children per nodes, i.e. B-Trees, has been introduced by Bayer and McCreight, 1970, and its variant, i.e. B^+-Trees, is described in Comer, 1979. Sleator and Tarjan, 1985 have introduced

the Self-adjusting Binary Trees and the CSS-Trees have been introduced by Rao and Ross, 1999.

Regression Analysis is a classic across many disciplines and it is widely discussed in Freedman, 2005, and in Goodfellow, Bengio, and Courville, 2016b. The characteristics of a Neural Network are illustrated in Goodfellow, Bengio, and Courville, 2016a. Concerning the architecture used for this Thesis, Kraska et al., 2018 describes it, but no experimentation has been reported in the Literature.

The fact that one can derive a CDF from the table and approximate that curve via regression to make a prediction in the table is not new, e.g., Ao et al., 2011. However it has been not much more than an isolated trick. The full generality of this trick comes out with the Learned Indexes introduced by Kraska et al., 2018. In that seminar paper, apart of the paradigm, an initial form of the **RMI** is provided, which is then perfected in Marcus, Kipf, et al., 2020. The **PGM** Index is introduced by Ferragina and Vinciguerra, 2020b and use, for its construction, the Piecewise Linear Approximation presented in Ferragina, Lillo, and Vinciguerra, 2020. The **RS** Index is introduced by Kipf et al., 2020 and its greedy construction algorithm is provided by Neumann and Michel, 2008.

Chapter 2

On the Suitability of Neural Network as a Learned Index Model

In this Chapter, we first highlight that one of the motivations for introducing Learned Indexes is a perceived change of paradigm in computer architectures that would favour the use of GPUs and TPUs over conventional CPUs. In turn, as an alternative to classic indexes, such paradigm shift favours the use of Neural Networks supported by highly engineered software platforms, such as Tensorflow. However, no evidence is given that this new approach leads to improvements, in particular regarding Neural Networks. Here we provide the first experimental evidence that these models are not competitive with the simple Atomic ones introduced in Section 1.4.2. However, as illustrated in the following Chapters, the use of those latter are inferior to the more structured ones, also described in Section 1.4.2. Therefore, regarding the ones introduced in Section 1.4.2, we show that Neural Networks are not competitive as Learned Index models. In particular, this Chapter is organized as follows.

- In Section 2.1, we discuss the role of Neural Networks in the sorted indexing context, with reference to modern hardware architectures.
- In Section 2.2, we illustrate the experimental methodology adopted in this Chapter.
- In Section 2.3, we illustrate and discuss the results obtained in our experiments.
- In Section 2.4, we provide conclusions on the experimental analysis conducted in the preceding Section.

The Bibliographic Notes relevant for this Chapter are in the corresponding Section.

The software used for the experimentation described in this Chapter is available on Github[1], while the relative datasets are available on a repository[2].

2.1 The Perceived Potential of the Neural Networks with the use of the Modern Computer Architectures

Indexes used for searching in a sorted set are usually engineering-optimised on well-known characteristics of the domain in which they are used and cannot represent the extreme variety present in real data. Therefore, to solve this problem, it seems possible to use Machine Learning models capable of learning the intrinsic characteristics

[1] https://github.com/globosco/A-Benchmarking-platform-for-atomic-learned-indexes
[2] http://math.unipa.it/lobosco/NNLI/

of the data. The ML models with most potential in this area are undoubtedly Neural Networks because of their power. In fact, they can learn any type of function with the right architecture. Unfortunately, they require a prohibitive computational power. In recent years, the introduction of GPU and TPU architectures in commercial computers, and the deployment of highly engineered development platforms such as Tensorflow, has improved the performance of all those operations involved in a Machine Learning process. Therefore, ML Models and Neural Networks have been increasingly used in many fields. The greatest strength of these new architectures is that they can parallelise math operations made by Neural Networks very well compared to a general-purpose set of instructions. In particular, recent studies even argue that the power of the GPU can be improved by 1000x in terms of time in the next few years, while, due to Moore's law constraints, those improvements are not seen for classic CPUs. Furthermore, a programming paradigm based on branches of the if-then-else type seems to have been overcome in favour of a paradigm that promotes mathematical operations carried out in pipelines; this aspect is discussed in more detail in Section 4.1 of Chapter 4.

For these reasons, using ML models such as Neural Networks, instead of the classic Data Structures, which make extensive use of branch instructions in their code, may lead to the deployment of substantial better Data Structures with benefit in several areas, such as Databases. In the following Section, we analyse an example of their use, i.e., Learned Bloom Filter.

2.1.1 From Motivation to Design and Implementation: The Case of Learned Bloom Filters

In order to be able to illustrate how a Learned Bloom Filter works, we start by defining its classic version. Then, we illustrate some examples, provided in the Literature, of the use of Neural Networks to improve Bloom Filters performance.

Classic Bloom Filter

Given a universe U of keys and $A \subset U$ such that $|A| = n$, we define a Bloom Filter as an array of bits v of size m with every element initialised to 0. Using k hash functions $h_j : U \to \{0, ..., m-1\}$ with $j \in \{1, ..., k\}$, each element x of A is encoded such that every bit of v at position $h_j(x)$ is equal to 1 for every $j \in \{1, ..., k\}$. To test whether an element $x \in U$ belongs to set A, we compute all values of the hash functions $h_j(x)$ with $j \in \{1, ..., k\}$ and reject the test if at least one of the values is equal to 0. This structure thus defined makes it possible to recognise with absolute certainty if a given query does not belong to set A but admits a false positive rate (**FPR**) ϵ, due to the hash collisions.

An important result concerning Bloom Filter is that the **FPR** depends only on *apriori* defined values k and m. In fact, if x is an element of the Universe U such that $x \notin A$ and ρ is the fraction of bits set to 1, then

$$Pr(y \text{ is a False Positive}) = \rho^k \tag{2.1}$$

and

$$E[\rho] = 1 - (1 - \frac{1}{m})^{kn} \approx 1 - e^{-kn/m} \tag{2.2}$$

So, the **FPR** value for a given Bloom Filter is

$$FPR \approx (1 - e^{-kn/m})^k \tag{2.3}$$

Learned Bloom Filter

Due to the motivations anticipated in Section 2.1, ML Models have been extensively used within the design and implementation of Learned Bloom Filters, since the very start of the area of Learned Data Structures. In particular, the construction of a Basic Learned Bloom Filter is performed by adding a Binary Classifier f able to reduce the number of keys filtered by the Bloom Filter. The Classifier provides as output an estimated probability that a given query x belongs to the set A so that, choosing a threshold τ, due to the possible presence of false negatives, a backup Bloom Filter is constructed only for each element $x \in A$ such that $f(x) < \tau$. Therefore, it is possible to build a smaller Bloom Filter thanks to the ability of the Classifier to filter out the elements that with high probability belong to A. The main criticality of this Data Structure lies in the necessity to establish the model parameters empirically by searching experimentally for those that minimise the space/**FPR** trade-off. However, this is the simplest version of Learned Bloom Filters described in the Literature, which has been extensively experimentally analysed with different types of models from the simplest Bayesian Classifiers to the most complex Convolutional and Recurrent Neural Networks.

In addition, as evidence of the increasing importance of this new paradigm, more complex models have been proposed in the Literature to improve the performance of Learned Bloom Filters. Some examples are the following.

- **Sandwich Learned Bloom Filter**. In order to reduce the **FPR** of a Learned Bloom Filter, this model uses a small preliminary Bloom Filter to filter out a proportion of negative items. The positive results are then successively filtered out by the Learned Bloom Filter constructed as described previously.
- **Partitioned Learned Bloom Filter**. The goal of this model is to use classifier information to identify a partition of the estimated probabilities such that it is possible to construct in each region a Backup Bloom Filter that optimises the space/**FPR** trade-off.
- **Adaptive Learned Bloom Filter**. Starting from the same considerations as in the previous model, it uses each region to define different groups of independent hash functions that are applied to a single backup Bloom Filter.

Although there is growing interest in using Neural Networks Models to improve Data Structures such as Bloom Filters, there is no experimental evidence in the Literature of their use for Learned Indexes, as well as Learned Hash Function and Rank/Select Data Structures. Therefore, in the following Sections, we propose an analysis of their use in the Learned Index field.

2.2 Experimental Methodology

2.2.1 Datasets

For this Chapter, we generate two synthetic datasets using random sampling in $[1, 2^{r-1} - 1]$ with r the integer precision used, i.e., 64 bits. In detail, we have:

1. **uni** that contains data sampled from a Uniform distribution defined via the *PDF* as

$$U(x,a,b) = \begin{cases} \frac{1}{b-a} & \text{if } x \in [a,b] \\ 0 & otherwise \end{cases} \tag{2.4}$$

   where $a = 1$ e $b = 2^{r-1} - 1$.

2. **logn** that contains data sampled from a Log-normal distribution defined via the *PDF* as

$$L(x,\mu,\sigma) = \frac{e^{-\frac{(lnx-\mu)^2}{2\sigma^2}}}{x\sqrt{2\pi\sigma}} \tag{2.5}$$

   where $\mu = 0$ e $\sigma = 1$ are respectively mean and variance of the distribution.

We also use two real datasets available in the Literature, namely:

1. **real-wl** that contains timestamps of about 715M requests performed by a Web server during 2016.

2. **real-iot** that consists of timestamps of about 26M events recorded during 2017 by IoT sensors deployed in academic buildings.

All datasets are ordered and without duplicates. Their details are summarized in Table 2.1. We anticipate that, as evident from the analysis in Section 2.2.3, the use of GPU training for the **NN**s severely limits the size of the datasets that we can use.

Each query dataset is generated from each dataset described above. It has a size equal to 50% of the reference dataset and contains, in equal parts, both elements present and not present in the dataset. For all the experiments, the queries are not sorted.

Uniform Distribution			
Name	**Size (KB)**	**Items**	**Type**
uni	1.10e+04	1.05e+06	integer
Log-normal Distribution			
Name	**Size (KB)**	**Items**	**Type**
logn	1.05e+04	1.05e+06	integer
Real Distribution			
Name	**Size (KB)**	**Items**	**Type**
real-wl	3.48e+05	3.16e+07	integer
real-iot	1.67e+05	1.52e+07	integer

TABLE 2.1: **A summary of the Datasets**. For each dataset in the collection, it is shown: the name used (column **Name**), its size in Kilobyte (column **Size (KB)**), the number of elements in it (column **Items**), and the type of its elements (final column **Type**).

2.2.2 Hardware

Experiments have been performed using a workstation equipped with an Intel Core i7-8700 3.2GHz CPU and an Nvidia Titan V GPU. The total amount of system memory is 32 Gbyte of DDR4. The GPU is also supplied with its own 12 Gbyte of DDR5

memory and adopts a CUDA parallel computing platform. CPU and GPU are connected with a PCIe 3 bus with a bandwidth of 32Gbyte/s. The operating system is Ubuntu LTS 20.04.

2.2.3 Models, Training and Query

We consider the Atomic Models introduced in Section 1.4.2. They have been training exclusively on CPU, following the algorithm described in Section 1.3.

As far as Neural Networks are concerned, with reference to the architecture presented in Section 1.4.2, we consider three types of **NNs** with different hidden layers, as specified next.

- **NN0** for zero hidden layers.
- **NN1** for one hidden layer with 256 units.
- **NN2** for two hidden layers with 256 units each.

We point out that we must transform both training and query datasets to use them as input of the **NN**s. Specifically, each item is converted into its binary representation with a fixed number of bits, i.e., 64 bits.

NNs have been trained using the highly-engineered Tensorflow platform with GPU memory support. As far as queries are concerned, following the Literature, we do not use Tensorflow for NNs queries within the GPU. Moreover, we also anticipate that, even using our own NVDIA CUDA implementation for queries, the use of GPU for queries compared with the analogous task on Atomics Models on CPU does not lead to any advantages, as we show in Section 2.3.2. For this reason, the full set of experiments, whose results are reported in Section 2.3.3, are performed only with the use of the CPUs.

2.3 Experiments, Results and Discussion

2.3.1 Training: GPU vs CPU

In Tables 2.2 and 2.3, we report the training times per element for each method described in the preceding Section, and we also indicate the respective **RF**, computed as indicated in Section 1.4.1. For what concerns Atomic Models L, Q and C, the training time is the time needed to solve Eq. 2 in Section 1.3. Regarding the NN models, the used learning algorithm is stochastic gradient descent with momentum parameter equal to 0.9 and a learning rate equal to 0.1. The Batch size is 64, and the number of epochs is 2000.

As is evident from the Tables here presented, even with GPU support and the use of a highly-engineered platform, **NN**s are not competitive to simple Atomic Models. Indeed, for each dataset, despite having similar **RF**s, the **NN**s training time per item is four orders of magnitude higher than the one obtained with Atomic Models.

2.3.2 Query: GPU only for NNs

Before comparing **NN**s with Atomic Models, we perform a preliminary experiment, only on **NN0** and **uni**, to see if there could be a real advantage from using the GPU for queries. In Table 2.4, we report the query time per element resulting from this experiment. As evident from that Table, on GPUs, the copy operations from CPU

NN0		
Dataset	**Training Time (s)**	**% Reduction Factor**
uni	2.55e-04	94.08
logn	1.39e-04	54.40
real-wl	2.50e-04	99.99
real-iot	1.28e-04	89.90
NN1		
Dataset	**Training Time (s)**	**% Reduction Factor**
uni	4.18e-04	99.89
logn	3.79e-04	94.21
real-wl	2.31e-04	99.88
real-iot	4.20e-04	98.54
NN2		
Dataset	**Training Time (s)**	**% Reduction Factor**
uni	4.49e-04	99.87
logn	8.60e-04	97.14
real-wl	2.33e-04	99.84
real-iot	3.57e-04	97.31

TABLE 2.2: **NN training with the use of Tensorflow on GPU**. For each dataset and each model, it is shown: the training time per element expressed in seconds (column **Training Time (s)**) and the percentage of the table reduction (column **% Reduction Factor**), as described in Section 1.4.1.

L		
Dataset	**Training Time (s)**	**% Reduction Factor**
uni	8.20e-08	99.94
logn	5.61e-08	77.10
real-wl	5.82e-08	99.99
real-iot	7.70e-08	96.48
Q		
Dataset	**Training Time (s)**	**% Reduction Factor**
uni	1.27e-07	99.98
logn	1.02e-07	90.69
real-wl	1.14e-07	99.99
real-iot	1.25e-07	99.10
C		
Dataset	**Training Time (s)**	**% Reduction Factor**
uni	1.84e-07	99.97
logn	1.74e-07	95.76
real-wl	1.24e-07	99.45
real-iot	1.63e-07	98.87

TABLE 2.3: **Linear (L), Quadratic (Q) and Cubic (C) Models Training**. The Legend is as in Table 2.2.

Methods	Copy (s)	Op. (s)	Search (s)	Query (s)
NN0-BBS	3.27e-08	4.20e-09	1.84e-09	3.27e-08
BBS	2.55e-09	-	1.89e-09	4.44e-09

TABLE 2.4: **Query Time on GPUs**. **NN0-BBS** refers to Binary Search with **NN0** as the prediction step, while **BBS** is the Binary Search executed on GPU without a previous prediction. For each of these methods executed on GPU, we report: the time for CPU-GPU, and vice versa, copy operations (column **Copy (s)**), the time for maths operation (column **Op. (s)**), the time to search into the interval (column **Search (s)**) and the total time to complete the query process (column **Query (s)**). Every time in the Table is per element and is expressed in seconds.

Dataset	BFS	NN0-BFS	NN1	NN2
uni	2.81e-07	1.31e-07	1.56e-06	5.16e-06
logn	2.08e-07	1.92e-07	1.69e-06	5.24e-06
real-wl	3.38e-07	4.59e-07	Space Error	Space Error
real-iot	3.07e-07	4.76e-07	1.90e-06	1.94e-05

TABLE 2.5: **CPU Prediction Effectiveness-Neural Networks Models**. **NN0-BFS** refers to Binary Search with **NN0** as the prediction step, while the other two columns refer to the time taken by **NN1** and **NN2** to predict the search interval only. The time is reported as time per query in second. When the model and the queries are too big to fit in the main memory, a space error is reported.

to GPU, and vice versa, cancel the one order of magnitude speed-up of the maths operations. In addition, a classic Binary Search on the GPU is by itself faster than its Learned counterparts, making the use of **NN**s on this architecture unnecessary.

2.3.3 Query: CPU only

The query experiments results are summarized in Tables 2.5 and 2.6. As we can see, **NN**s are also not competitive for the query phase.

The query time on **NN1** and **NN2** is even two orders of magnitude greater than with Linear Regression. In addition, in some cases, the transformed dataset cause a space allocation error when the Intel Math Kernel Library attempts to perform NNs computations. On the other hand, the simplest model, i.e. **L**, shows very efficient query times despite obtaining smaller Reduction Factors in the training phase.

Dataset	BFS	L-BFS	Q-BFS	C-BFS
uni	2.81e-07	9.42e-08	8.11e-08	9.39e-08
logn	2.08e-07	1.60e-07	1.59e-07	1.54e-07
real-wl	3.38e-07	5e05e-08	2.12e-7	1.80e-7
real-iot	3.07e-07	8.32e-08	1.99e-7	2.57e-7

TABLE 2.6: **CPU Prediction Effectiveness-Atomic Models**. The Table reports results with Linear, Quadratic and Cubic models. The Legend is as in Table 2.5.

2.4 Conclusions

As noted in Section 2.1, a perceived paradigm shift is one of the motivations for the introduction of the Learned Indexes. Despite that, as described in Section 2.3.2, the use of a GPU architecture for Learned Indexes is still premature, due to a data transfer bottleneck in the communication between CPU and GPU memory. However, we point out that the identification of this problem is new in the Literature.

In addition, as evident from the analysis in Section 2.3, Atomic Models can achieve significant table reductions and very competitive training and querying times compared to those performed by Neural Network Models. Although this may not be surprising to the specialist, those experimental findings, reported here, have not been quantified before.

From now on, we no longer consider **NN**s in the following Chapters.

Bibliographic Notes

The potential power of GPUs and TPUs compared to the CPUs is mentioned in Kraska et al., 2018. In addition, the reader can find an extensive description of the GPU and TPU architectures potential in Wang, 2017 and Sato, Young, and Patterson, 2017. Moore's law, fundamental in the semiconductor industry, has been introduced in Moore et al., 1965. The Neural Networks training has been performed using the highly-engineered platform Tensorflow, described in Abadi et al., 2015. However, following the indication in Kraska et al., 2018, we have performed queries using our native NVIDIA CUDA implementation, due to a significant Tensorflow invocation overhead.

The universal ability of Neural Networks to learn functions, including the simple networks described in Section 1.4.2, is widely studied in the Literature. The reader can refer to Ohn and Kim, 2019.

Bloom Filter is a well-known Data Structure to solve the Approximate Membership Problem and it is introduced by Bloom, 1970. In Kraska et al., 2018, a Learned version of Bloom Filters that use Recurrent Neural Networks was proposed for the first time. Successively, the versions called Sandwich and Partitioned Learned Bloom Filters were described by Mitzenmacher, 2018 and Vaidya et al., 2020, respectively. An other version is the Adaptive Learned Bloom Filters proposed by Dai and Shrivastava, 2020. In addition, an extensive study on the choice of models and parameters of a Learned Bloom Filters was presented by Fumagalli et al., 2022.

Real datasets used for experiments in this Chapter, are the same in Kraska et al., 2018 and Galakatos et al., 2019, while the synthetic ones are introduced in this Thesis to provide an exhaustive assessment of Neural Networks. It should be noted that, since we only present negative results, it has not been considered necessary additional benchmark datasets available in the Literature.

Chapter 3

Learned Sorted Table Search and Static Indexes in Small Space: A Comprehensive Experimental Analysis

This Chapter examines the advantages of using Learned Indexes in the field of Sorted Table Search procedures. It is first considered a simple scenario consisting of "textbook" algorithms and models. Next, more sophisticated Learned Indexes are analysed through the Search on Sorted Data (**SOSD** for short) benchmark platform. Finally, it is shown that even imposing a space-bound on the most complex models, it is possible to improve query time. In particular, this Chapter is organized as follows.

- In Section 3.1, we introduce, referring to Sections 1.2 and 1.4, the procedures used for the experiments described in this Chapter.
- In Section 3.2, we illustrate the experimental methodology adopted in this Chapter.
- In Section 3.3, we introduce new models capable of using constant or small space with respect to the table size.
- In Section 3.4, we illustrate and discuss the results obtained in our experiments.
- In Section 3.5, we provide conclusions on the experimental analysis of the preceding Section.

The Bibliographic Notes relevant for this Chapter are in the corresponding Section.

The software used for the experimentation described in this Chapter is available on Github[1], while the relative datasets are available on a repository[2].

3.1 Classic and Learned Sorted Table Search

In this Chapter, we have considered the main classic algorithms for Sorted Table Search among those described in Chapter 1. In particular, we choose the following.

[1] https://github.com/globosco/A-learned-sorted-table-search-library
[2] http://math.unipa.it/lobosco/LSTS/

- The two Binary Search versions, i.e. **BBS** and **BFS**, used as a baseline to compare against and described in detail in Section 1.2.1.
- *k*-ary Search in its two versions, i.e., **K-BBS** and **K-BFS** (detailed in Section 1.2.1).
- All the array layouts described in Section 1.2.2. For the sake of readability and clarity, **BFV** and **BFT** have been omitted from discussion of this Chapter because they are not competitive compared to **BFE**.
- Interpolation Search (**IBS** for short) and its variant **TIP** as shown in Section 1.2.1.
- The classic index B$^+$-Tree as described in Section 1.2.3.

As illustrated in Section 1.4.2, except for the ones operating on table layouts different than sorted and B$^+$-Trees, all the procedures mentioned above have a natural Learned version.

Concerning model classes described in Section 1.4.2, we have considered each model described there except for **NN**s, whose exclusion has been justified in the previous Chapter.

Most importantly, in Section 3.3, we introduce two new models, which use constant or small space in addition to the table size. The first, which we refer to as **KO-**, uses constant space in addition to the table and represents a new category in the hierarchy introduced in Section 1.4, i.e., Two-Level Hybrid Models. The second, which we refer to as **SY-RMI**, uses little space in addition to the table, about 0.05%, and is a member of the Two-Level **RMI**s with Parametric Branching Factor class.

3.2 Experimental Methodologies

3.2.1 Hardware

All the experiments have been performed on the same workstation described in Section 2.2.2. For the sake of this Chapter, it is also helpful to know that its CPU has three levels of cache memory: (a) 64kb of L1 cache; (b) 256kb of L2 cache; (c) 12Mb of shared L3 cache. In Addition, the *cls* and *size*, defined in Section 3.3.3, are respectively 64 and 8 bytes.

3.2.2 Datasets

We use five kinds of benchmark datasets presented in the Literature and we generate four versions for each to obtain data fitting different memory levels. A summary of those datasets is provided in Table 3.1. However, for a more in-depth description, refer to Appendix A.

3.2.3 Software Systems for Learned Indexes Training

The software systems used to train Learned Indexes highly depends on the their model class. More details for each class are provided in the following.

Atomic Models: Linear Regression

We have implemented a simple closed-form regression in Python to train linear, quadratic and cubic models, as discussed in Section 1.3.

L1 memory level			
Name	**Size (KB)**	**Items**	**Type**
amzn32	3e+1	7.5e+3	integer
amzn64	3e+1	3.75e+3	integer
face	3e+1	3.75e+3	integer
osm	3e+1	3.75e+3	integer
wiki	3e+1	3.75e+3	integer
L2 memory level			
Name	**Size (KB)**	**Items**	**Type**
amzn32	2.52e+2	6.3e+4	integer
amzn64	2.52e+2	3.15e+4	integer
face	2.52e+2	3.15e+4	integer
osm	2.52e+2	3.15e+4	integer
wiki	2.52e+2	3.15e+4	integer
L3 memory level			
Name	**Size (KB)**	**Items**	**Type**
amzn32	6e+3	1.5e+6	integer
amzn64	6e+3	7.5e+5	integer
face	6e+3	7.5e+5	integer
osm	6e+3	7.5e+5	integer
wiki	6e+3	7.5e+5	integer
L4 memory level			
Name	**Size (KB)**	**Items**	**Type**
amzn32	8e+5	2e+8	integer
amzn64	1.6e+6	2e+8	integer
face	1.6e+6	2e+8	integer
osm	1.6e+6	2e+8	integer
wiki	1.6e+6	2e+8	integer

TABLE 3.1: **A summary of the Datasets**. For each dataset in the collection, it is shown: the name used (column **Name**), its size in Kilobyte (column **Size (KB)**), the number of elements in it (column **Items**), and the type of its elements (final column **Type**).

Two-Level RMIs with Parametric Branching Factor: CDFShop

To train two-level **RMI**s, we use the software platform available online called **CDFShop**. Through this, it is possible to train an **RMI** by specifying the models for each level and the Branching Factor value. It is also possible to use an optimisation tool to extract the best models for a given table using Pareto optimisation. In particular, we use the optimization software to obtain up to ten versions of the generic model for a given table. For each model, the optimization software picks an appropriate branching factor and the type of regression to use within each part of the model. Those latter quantities are the parameters that control the precision of the prediction and its space occupancy. As pointed out in the Literature, it is also to be remarked that this optimization process provides only approximations to the real optimum. It is heuristic in nature, with no theoretic approximation performance guarantees. The problem of finding an optimal model in polynomial time is open.

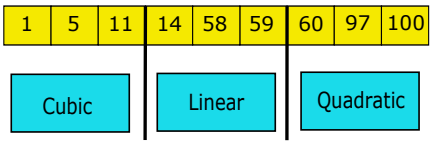


FIGURE 3.1: **An example of a KO-BFS, with k = 3**. The top part divides the table into three segments, and it is used to determine the model to pick at the second stage. Each box indicates which Atomic Model is used for prediction on the relevant portion of the table.

CDF Approximation-Controlled Models: SOSD Platform

For this class of models, we have chosen to use the construction parameters contained in the highly-engineered benchmark software **SOSD**. In particular, we decided to use models provided in this platform so that our study could be consistent with those already present in the Literature. Therefore, for a given table, we have built ten models of both those described in Section 1.4.2, provided by **SOSD**.

3.3 Constant and Small Space Indexes

A fundamental methodological question has been overlooked so far in the Literature, i.e., if it is possible to achieve speed-ups analogous to Learned Indexes using constant or small space with respect to the table size. For this purpose, as illustrated in the following, we have introduced two new models. We have also modified the bi-criteria version of **PGM**, which can return the best query time index within a given amount of space the model is supposed to use.

3.3.1 A Two-Level Hybrid Model, with Constant Branching Factor

As shown in Fig. 3.1, this model partitions the table into a fixed number of segments. For each, Atomic Models are computed to approximate the CDF of the table elements in that segment. Finally, we assign to each segment the model that guarantees the best Reduction Factor. Then, we make queries performing a Sequential Search in the set containing only each starting interval element. We pick up the selected segment and use the corresponding model to get the final search interval. We denote such a model as **KO-BFS** or **KO-BBS**, depending on the base Binary Search routine used. The number of segments is independent of the input and bounded by a small constant, i.e., at most 20 in this Thesis.

3.3.2 Synoptic RMIs

For a given set of tables of approximately the same size, we use **CDFShop**, as in Section 3.2.3, to obtain a set of models (at most 10 for each table). We compute the ratio (branching factor)/(model space) for the entire set of obtained models. We take the median of those ratios to measure the branching factor per unit of model space, denoted *UB*. Among the **RMI**s returned by **CDFShop**, we pick the relative majority winner, i.e., the one that provides the best query time over a set of simulations. When using such a model on tables of approximately the same size as the ones used as input to **CDFShop**, we set the branching factor to be a multiple of *UB*, which depends on how much space the model is expected to use relative to the input table size. Since this model can be intuitively considered as the one that best summarizes

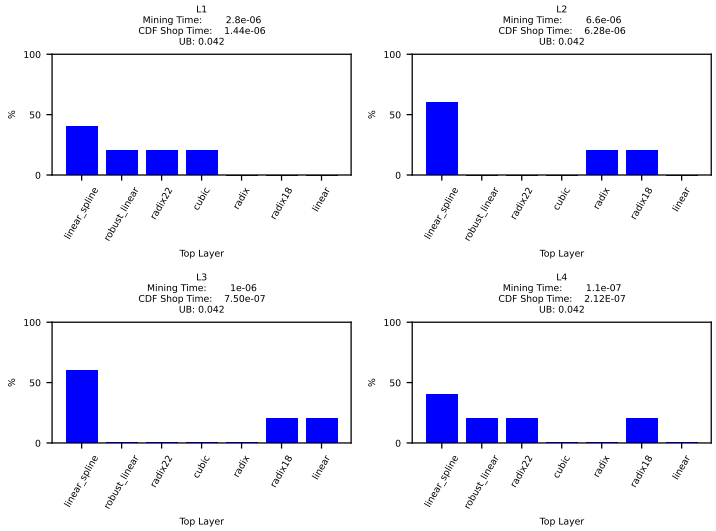


FIGURE 3.2: **Time and UB for the identification of SY-RMIs**. For each memory level, only the top layer of the various models is indicated in the abscissa, while the ordinate indicates the number of times, in percentage, the given model is the best in terms of query performance on a table. The branching factor per unit of space as well as the time it took to identify the proper **SY-RMI** (average time per element, over all **RMI**s returned by **CDFShop**) are reported on top of each figure. For comparison, we also report the same time for the output of **CDFShop**.

the output of **CDFShop** in terms of query time, for the given set of tables, we refer to it as synoptic and denote it as **SY-RMI**.

Mining SODS Output for the Synoptic RMI

For each memory level, we need to process the output of **SOSD** on all tables in this study, as described in Section 3.3.2, to obtain the corresponding **SY-RMI** with its UB. We anticipate that the simulation to identify the relative majority of **RMI**s is performed on query datasets extracted as described in Appendix A, but using only 1% of the number of query elements specified there.

The fact that the datasets represent different challenges for the learning of the CDF is well represented by the variety of models that perform best (see histograms in Figure 3.2). Therefore, given such a variety, it is far from obvious that the median UB is the same for each memory level. Moreover, the relative majority model is also the same across memory levels, i.e., linear spline, with linear models for each second layer segment.

3.3.3 Bi-Criteria PGM

PGM has a bi-criteria version that returns the best query time index, given the amount of space the model is supposed to use. It is to be noted that bi-criteria **PGM** needs a maximum and a minimum approximation denoted with ϵ_m and ϵ_M. In particular, ϵ_m is set to $2 * cls/size$ where cls is the cache line size and $size$ is the number of bytes of long integers. Preliminary experiments we have conducted indicate that such a setting, as far as identifying the best query time ϵ when a space-bound is specified, leads to cases in which the space resource is not fully used. Therefore, we

	L	Q	C	15O-BFS	SY-RMI 2%	SOSD RMI	SOSD RS	SOSD PGM
Datasets	Training Time							
amzn32	7.9e-09	1.4e-08	1.4e-08	3.7e-08	1.2e-06	1.2e-07	9.5e-09	2.4e-08
amzn64	7.9e-09	1.4e-08	1.4e-08	3.7e-08	1.1e-06	2.2e-07	2.1e-08	5.0e-08
face	8.0e-09	1.4e-08	1.4e-08	3.6e-08	1.3e-06	2.5e-07	2.1e-08	6.5e-08
osm	8.0e-09	1.4e-08	1.4e-08	3.6e-08	1.2e-06	2.5e-07	2.2e-08	7.4e-08
wiki	7.9e-09	1.4e-08	1.4e-08	3.6e-08	1.1e-06	2.2e-07	1.9e-08	4.1e-08

TABLE 3.2: **Training time for L4 tables, in seconds and per element**. The first column indicated the datasets. The remaining columns indicate the model used for the learning phase. The **SOSD** columns refer to the entire output of that library, averaged over a number of models and elements in each table.

modified the ϵ_m formulas to be parametric, i.e. the multiplicative constant is now a parameter a. So, we have used $a = 0.5, 1, 1.5$ and refer to this **PGM** as **PGM_M_a** for our purpose.

3.4 Experiments, Results and Discussion

3.4.1 Learning the CDF of a Sorted Table

Models need to learn the CDF function of the table in which search. For the uses intended in this Thesis, three indicators are essential: the time required for learning, the Reduction Factor that one obtains and the time to perform the prediction. We have trained the models described in the preceding Sections for each of the table datasets presented in Appendix A.

We report here only the Table 3.2 for L4 memory level. As for the training time for the entire set of experiments, it is reported in Tables B.1-B.4 in Appendix B, in terms of time per element. For **KO-BFS**, we report the model's time that has achieved top query times consistently across experiments. However, it is to be noted that the same training time applies to **KO-BBS**. Finally, for **PGM**, **RMI** and **RS**, the timing results have been obtained by summing the time to train all models returned by **SOSD** and taking the average. In those tables, we also include the time that the **SY-RMI**s, at 2% of space occupancy, take to learn the CDF of all tables at each memory level.

Atomic and Hybrid Models

Here we consider the learning phase of the Atomic Models **L**, **Q**, and **C** and of the **15O-BFS** model. As expected, regarding the first class, as the degree of the regression grows, so does the training time. Moreover, on both **L3** and **L4** memory levels, the training times are "stable" for datasets, while there is some variation on the other two memory levels. This is justified because the performance of the matrix products involved in the computation depends on the magnitude of the operands involved. As dataset size grows, such an effect is amortized on a larger and larger number of elements. In general, those methods perform very well compared to the others, in particular on the last two memory levels. Unfortunately, as we will see when we consider Reduction Factor and query time, their use is limited to datasets that are "very easy" to learn, i.e., with a regular CDF easily summarized by a polynomial function. As for **15O-BFS**, its training time is in line with the one of the previous models, although we anticipate that it provides higher Reduction Factors and better query times.

Two Level RMIs

We consider the learning phase of the two kinds of two Level **RMI**s in the following.

- **SY-RMI**. Since the simulation step can be seen as a post-processor of **CDFShop** output, it is an important finding that its execution time is comparable with the one of **CDFShop**, except for the **L3** memory level. That is, the simulation is not too imposing in terms of time if used as a post-processor of **CDFShop**. Also, since this is a "one time only" process, the output can then be reused repeatedly, in analogy with the output of **CDFShop**.
- **Heuriustically Optimized RMIs**. As for **RMIs**, they lag behind the other methods in training time. However, it is quite remarkable that the heuristic optimizations result in training times that are not "too far" from the other ones. This becomes evident when we compare the training time of a single model, the **SY-RMI**, with the average training time obtained with the use of **CDFShop**, which returns several models. Indeed, training one **RMI** model is costly, while such a cost can be amortized over the training of several models. This aspect does not seem evident in the current State of the Art.

CDF Approximation-Controlled Models

CDF Aproximation-Controlled models can be built in one pass over the input, with important implications. In this Thesis as well as in the Literature, the **RS** is reported as superior to the **PGM** in terms of training. However, this seems to hold on large datasets, while the **PGM** is "more effective" in training time across the memory hierarchy. The reason may be that those those two indexes use streaming procedures to approximate the CDF within a parameter ϵ, the main difference between the two is that the first uses an approximate algorithm while the second uses an exact one, apparently settling for an "approximation" pays off with large workloads.

3.4.2 Constant Space Models: Query Experiments

Constant Space Models represent an elementary scenario, in which we use nearly standard textbook code. In particular, the models considered in this Section use constant space.

The experiments regarding the classic algorithm, described in Section 3.1, and the constant space models, illustrated in Section 3.3.1, have been performed on all tables considered for this research. The query datasets have been generated as described in Appendix A and the corresponding models have been learned as described in Section 3.4.1. With the exception of **TIP** and of its learned version, the full set of results are reported in Figures B.1-B.5 of Appendix B. In all Figures, we report **K-BFS** and **K-BBS** with $k = 6$, while **KO-BFS** and **KO-BBS** with $k = 15$, since these are the best performing across experiments. Here we provide two representative cases, i.e., Figures 3.3-3.4.

As for **TIP** and its learned versions, their query performance are not reported since they turned out not to be competitive on the benchmarking datasets used in this study. For completeness, in Figure 3.5, we provide the query time performance on only one dataset to show that **TIP** can be sped up with the simple models considered here. All query times are averages over one million queries.

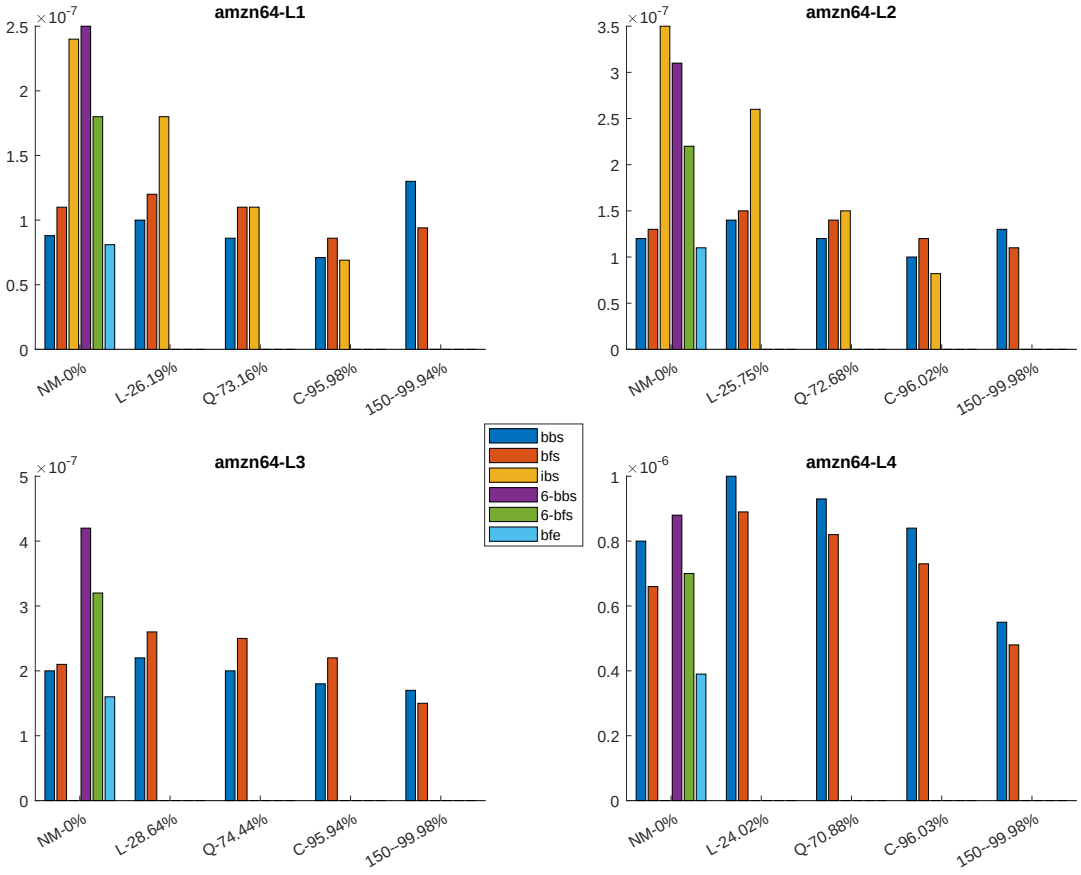


FIGURE 3.3: **Query times for the amzn64 dataset on Sorted Table Search Procedures.** The methods are the ones in the legend (middle of the four panels, the notation is as in the main text and each method has a distinct colour). For each memory level, the abscissa reports methods are grouped by model. From left to right, no model, linear, quadratic, cubic and **KO-**, with $k = 15$, and with **BFS** and **BBS** as search methods. **K-BFS** is reported with $k = 6$. For each model, the Reduction Factor corresponding to the table is also reported on the abscissa. On the ordinate, it is reported the average query time, in seconds. For memory level **L4**, **IBS**, **L-IBS** and **Q-IBS** have been excluded, since inclusion of their query time values ($3.1e-06$, $2.1e-06$, $1.2e-06$, respectively) would make the histograms poorly legible.

As illustrated in the following Sections, our experiments bring to light non-obvious scenarios, regarding speed-ups with constant space models. First, our experiments complete a methodologically important analogy between Learned Searching in Sorted Sets and the classic Data Structures approach, by considering the "entry-level" routines in this area. Another important indication concerns Binary Search with array layout other than Sorted. Indeed, the fact that **BFE** performs so well brings to light the need to consider models that are able to speed up such a procedure.

Atomic Models

(a) **Binary Search procedures**. The Learned versions of **BFS** and **BBS** are better than the respective simple one apparently only when the Reduction Factor is

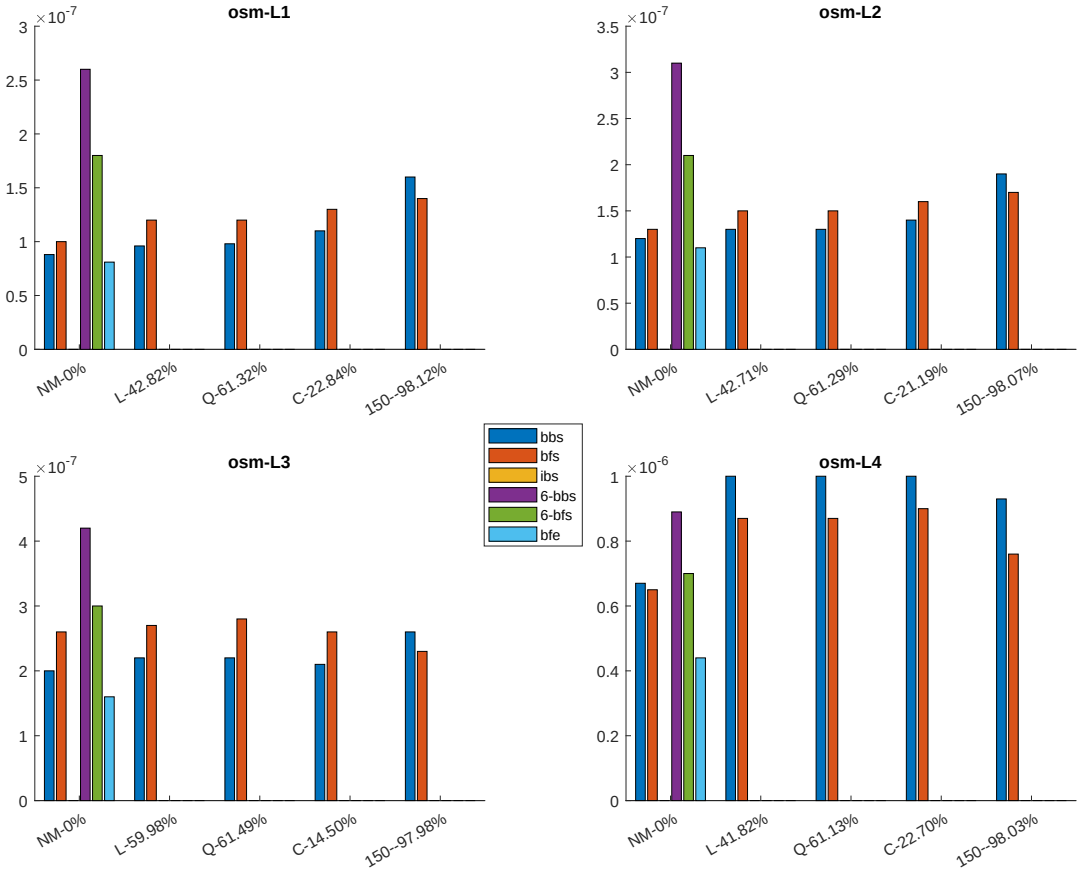


FIGURE 3.4: **Query times for the osm dataset on Sorted Table Search Procedures.** The figure legend is as in Figure 3.3. For the last three memory levels, **IBS** has been excluded, since inclusion of its query time values ($2.2e-06$, $6.5e-06$, $6.4e-05$, respectively) would make the histograms poorly legible. Its regression-based Learned versions have been excluded for the same reason (data not shown). However, they have better query time performance with respect to **IBS**, in particular **Q** and **C**.

quite high, e.g., 95%. Indeed, using a poor or even fairly good Reduction Factor may result in a Learned version slower than **BFS** or **BBS**, respectively. An example is provided in Figure B.4 of Appendix B. The cost of prediction, which is a constant time process for the simple models used here, requires only arithmetic operations, resulting in a loss compared to the speed of **BFS** or **BBS**, respectively. In other words, even though reducing the size of the table to search into brings theoretical benefits, the trade-off between the cost of the prediction vs reduction in search plays a key role in practice. Quite surprisingly, a good layout such as Eytzinger can be very competitive, even when the Learned version of **BFS** can speed up its non-learned version (see again Figures B.1-B.5 of Appendix B). Indeed, the regularity in data access and compactness of the **BFE** code makes it the one of choice across the memory hierarchy, reinforcing results already present in the Literature.

(b) **Interpolation Search procedures**. **IBS** and **TIP** seem to take consistent advantage of the prediction phase, with substantial speed-ups, unless the CDF of the dataset is close to uniform. This is the case for **face-L1-face-L2**. However, as the Reduction Factor increases for those procedures, from the one given by **L**

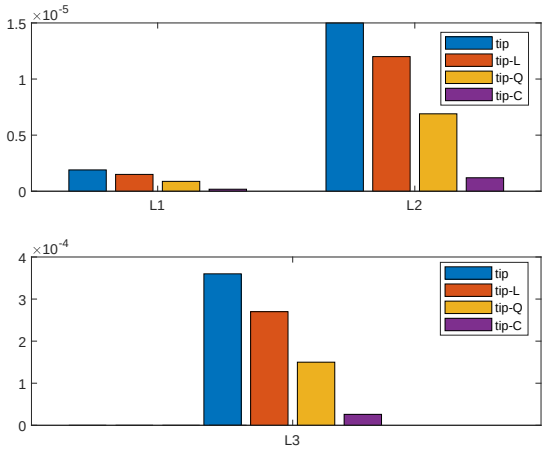


FIGURE 3.5: **Query times for TIP and its Learned Variants on the amzn32 dataset**. For each memory level, the abscissa reports models with **TIP** as search methods. From left to right, no model, linear, quadratic and cubic. On the ordinate, it is reported the average query time, in seconds. On the fourth memory level the procedures were stopped due to their poor execution times.

to the one given by **C**, the speed-up tends to increase (e.,g., see Figure B.1 in Appendix B). It is interesting to note that, although the CDF of **face-L4** looks uniform, there are many "rough spots" that make the prediction particularly challenging for simple regression methods (see the abscissa of the query times histograms for **face-L4** in Figure B.3 of the B).

In comparison with the other procedures, the Learned version of **IBS** is quite competitive on small datasets, with a somewhat simple CDF to learn (see Figures B.1-B.3 in Appendix B). However, it is to be appreciated that it can even be better than the very fast **BFE** in those circumstances.

Two Level Hybrid Model

KO-BBS and **KO-BFS** are consistently better than **K-BBS** and **K-BFS**, respectively, except for **L4-osm**, for any given $k \in [3, 20]$.

As for **L4-osm**, we observe the same results but with $k > 12$. **KO-BFS** is consistently better than **BFS** for any given $k \in [3, 20]$ with the exception of the **osm** datasets. As for **BBS**, there is no single $k \in [3, 20]$ for which **KO-BBS** is better across memory levels.

For the sake of readability, the results mentioned above are not shown and are available upon request.

3.4.3 Parametric Space Models: Query Experiments

As widely discussed in Section 1.4, the models considered here have a space occupancy that depends on parameters specific to the models. Moreover, all the experiments are supported by a highly effective software environment like **SOSD**.

We have repeated the same experiments reported in Section 3.4.2 using Parametric Space Models as illustrated in Section 3.1. We have considered three space-bound: 0.05%, 0.7%, 2% for the bi-criteria **PGM** and for **SY-RMI**. For each percentage, this is the amount of additional space the model can use with respect to the table size. However, we do not consider models that use a percentage of space higher

than 10% of each table size. Therefore, we report the one with the best query time for the remaining models. Moreover, we take as a baseline the **SOSD** version of **BBS**, which is implemented via vectors rather than arrays (as in the elementary case). For completeness, we also include our own vector implementation of **BFS** as a different baseline, executed within the **SOSD** software.

The full set of results are reported in Figures B.6-B.10 of Appendix B. For completeness, we report in Figures 3.6-3.7 the same representative datasets, as for the constant space case. Query times are again averages over one million queries. Moreover, in order to gain a synoptic quantitative evaluation of the relationships among space, query time and prediction accuracy, Table B.5 in Appendix B reports the average space, query time and Reduction Factor computed on all experiments performed in this study, normalized with respect to the best query time model coming out of **SOSD**.

Our experiments show that both **SY-RMI** and the bi-criteria **PGM** are able to perform better than **BBS** and **BFS** across datasets and memory levels, with very little additional space. As far those two Binary Search routines are concerned, and within the **SOSD** software environment, one can enjoy the speed of Learned Indexes with very little of a space penalty. Our study also provides additional useful insights into the relation time-space in Learned Indexes.

SOSD Models with at Most 10% of Additional Space

Both the **RS** and the **B$^+$-tree** are not competitive to the other Learned Indexes. Those latter consistently use less space and time, across datasets and memory levels. As for the **RMI**s coming out of **SOSD**, they cannot operate in small space at the **L1** memory level. On the other memory levels, they are competitive with **PGM_M** and **SY-RMI**, but seem to require more space than them.

Small Space Models

As concerns the **PGM_M** and the **SY-RMI**, except for the **L1** memory level, it is possible to obtain models that take space very close to a user-defined bound. The **L1** memory level is an exception since the table size is really small. As for query time, the **PGM_M** performs better on the **L1** and **L4** memory levels, while the **SY-RMI** on the remaining two. This complementarity and good control of space make those two models quite useful in practice.

3.5 Conclusions

In this Chapter, we have provided a systematic experimental analysis regarding the ability of Learned Model Indexes to perform better than Binary and Interpolation Search in small space. However, our results also indicate that, although a small model with good accuracy may not provide the best query time, prediction power is somewhat marginal to assess performance. Indeed, across memory levels, we see a space hierarchy of model configurations. The most striking feature of this hierarchy is that the gain in query time between the best model and the others is within small constant factors, while the difference in space occupancy may be several orders of magnitude. This brings to light the acute need to investigate the existence of "small space" models that should close the time gap between those and the best performing methods.

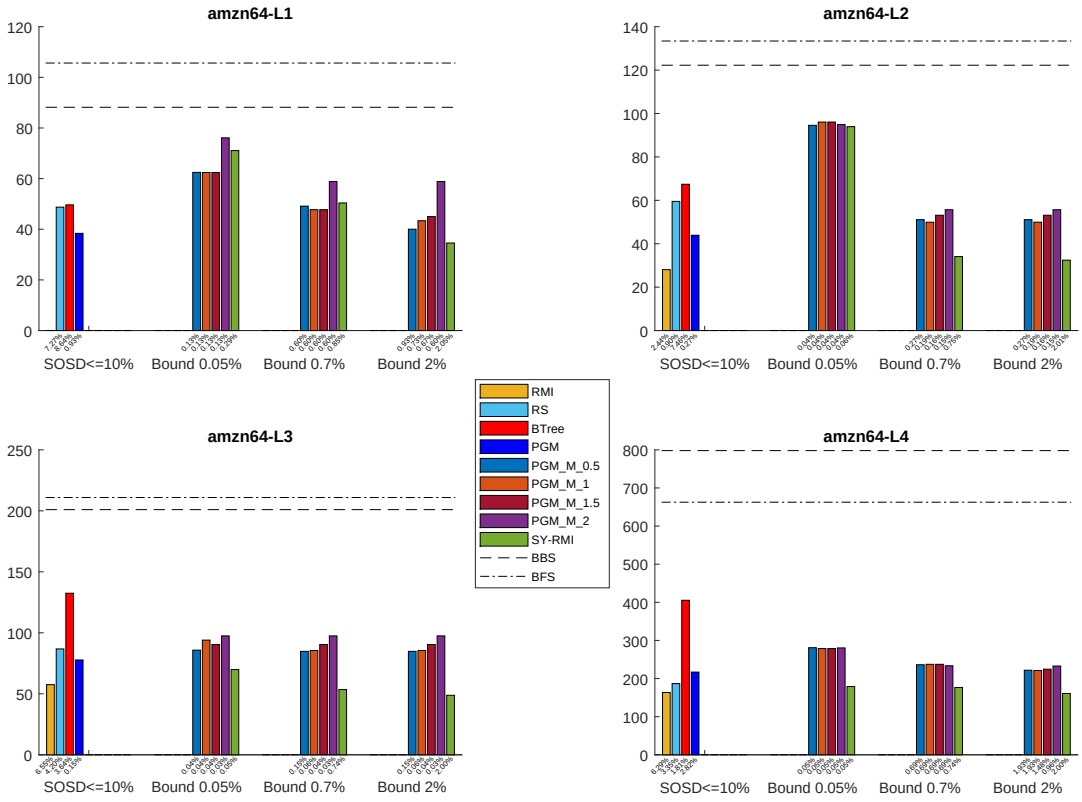


FIGURE 3.6: **Query times for the amzn64 dataset on Learned Indexes in Small Space.** The methods are the ones in the legend (middle of the four panels, the notation is as in the main text and each method has a distinct colour). For each memory level, the abscissa reports methods grouped by space occupancy, as specified in the main text. When no model in a class output by **SOSD** takes at most 10% of additional space, that class is absent. The ordinate reports the average query time, with **BBS** and **BFS** executed in **SOSD** as baseline (horizontal lines).

Bibliographic Notes

Most of the references for this Chapter are illustrated in the Bibliographic Notes Section of Chapter 1.

The benchmarking platform **SOSD** has been proposed in Marcus, Kipf, et al., 2020, while the optimization platform **CDFShop** has been described in Marcus, Zhang, and Kraska, 2020. For consistency with the benchmarking studies in the Literature, we use the same datasets as in Marcus, Kipf, et al., 2020 and all models in Section 3.4.3 use the Branchy version of Binary Search contained in **SOSD**.

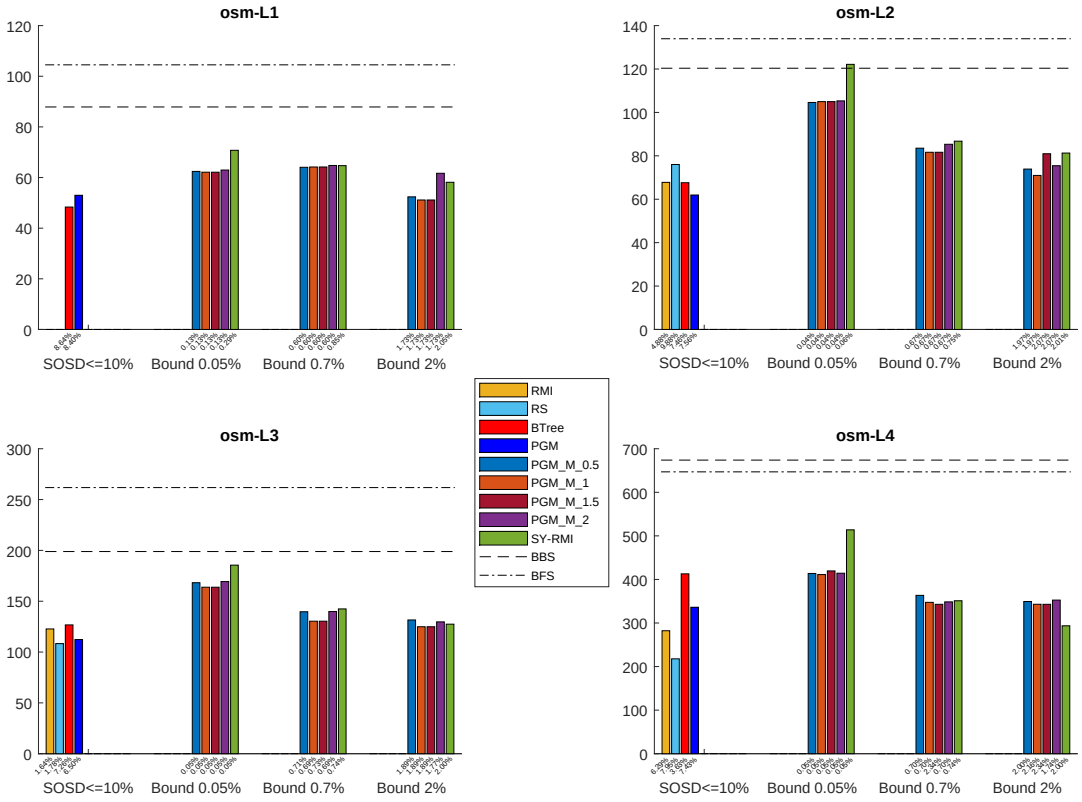


FIGURE 3.7: **Query times for the osm dataset on Learned Indexes in Small Space.** The figure legend is as in Figure 3.6.

Chapter 4

Standard Vs Uniform Binary Search and their Variants in Learned Static Indexing: The Case of the SOSD Benchmarking Software

This Chapter provides a study of how different kinds of searches can affect the performance of Learned Indexes. First, we discuss the difference between Uniform and Standard Binary Search, with particular attention for how these procedures can take advantage from modern computer architectures. Then, we describe the experimental methodology and, finally, we analyse the obtained results. In particular, this Chapter is organized as follows.

- In Section 4.1, we discuss the difference between Uniform and Standard Binary Search with the reference to the use on modern computer architectures.
- In Section 4.2, we illustrate the experimental methodology adopted in this Chapter.
- In Section 4.3, we illustrate and discuss the results obtained in our experiments.
- In Section 4.4, we provide conclusions on the experimental analysis of the preceding Section.

The Bibliographic Notes relevant for this Chapter are in the corresponding Section.

The software used for the experimentation described in this Chapter is available on Github[1], while the relative datasets are available on a repository[2].

4.1 Uniform and Standard Binary Search on Modern Computer Architectures

Several elements of modern computer architectures can influence classic search algorithms in a sorted set. In particular, the communication latency between the CPU

[1] https://github.com/globosco/A-modified-SOSD-platform-for-benchmarking-Branchy-vs-Branch-free-search-algorithms

[2] http://math.unipa.it/lobosco/LSTS/

and Main Memory can be considered the biggest problem in the practical performance of an algorithm. In order to avoid that continuous reads of the Main Memory slow down the execution, different hierarchical memory levels have been included in the CPUs. In general, cache memory is faster but smaller than Main Memory. Therefore, if the cache memory contains the data needed for execution, latency is drastically reduced, and performance improves. Therefore, it is then essential to avoid frequent context changes to minimise reads and loads from Main Memory. Furthermore, executing an instruction in the CPU takes several clock cycles to perform different operations, such as fetch, decode, read and execute. In modern computer architectures, a pipeline mechanism is used to simultaneously handle several consecutive instructions in different phases of the cycle to speed up the processing of these steps. However, "conditional jumps" in the code prevent the CPU from knowing which instruction to fetch next. Modern CPUs have introduced branch prediction systems that try to guess the result of a "conditional jump" by anticipating the caching of the necessary instructions and data. Obviously, if the prediction fails, a context change must be made.

A procedure such as the Standard Binary Search, which has if-then-else type constructs within the loop, during its execution makes many "conditional jumps" that are difficult to predict. This results in generating several context changes, with consequent slow down of the procedures. In contrast, Uniform Binary Search replaces "conditional jumps" with "conditional moves" that do not modify and interfere with the execution flow in the processor pipeline. An example of the Assembly code generated by the compiler for the Uniform Binary Search is provided in Listing 4.2.4. In particular, the while loop is implemented in lines 8-15. As anticipated in Section 1.2.1, the line 8 of Algorithm 2 is translated into the "conditional move" at line 12 of Listing 4.2.4, that sets the variable *base* without the use of jump. In general, this instruction is predictable by the Branch Predictor with an accuracy of the 95%, with a consequent performance increase. It should be noted that analogous changes can be made in k-ary Search routines in order to achieve equal improvements.

For the reasons discussed above, used in a stand-alone context, Uniform Search can be better than the Standard one in the first two memory levels. However, this advantage within Learned Data Structures has not been covered in the Literature and is the subject of this Chapter.

Following the nomenclature used in the Literature, we refer to Standard and Uniform Search as Branchy and Branch-free, respectively.

4.2 Experimental Methodology

4.2.1 Hardware

We use the same hardware of Chapter 3, described in Section 3.2.1.

4.2.2 Datasets

We use, with the exception of the query tables, the same datasets of experiments in Chapter 3 and more in-depth described in Section 3.2.2 and Appendix A. As for query dataset generation, for each of the tables built as described in the mentioned Sections, we extract uniformly and at random (with replacement) from the Universe U a total of two million elements, 50% of which present, 50% absent in each table.

4.2.3 Binary Search and Its Variant

In this Chapter, we use our versions of Branchy and Branch-free Binary Search and k-ary Search, following Algorithms described in Section 1.2.1. In particular, we use $k = 3$ as recommended in the Literature. Finally, although not directly usable within the Learned Indexing framework, we also include the Branch-free Eytzinger Layout (described in Section 1.2.2), since it is an useful baseline to compare the performance of those routines within **SOSD** with analogous performances obtained in the Literature.

4.2.4 Index Model Classes in SOSD

From the models available in **SOSD**, we choose the ones that have been the most successful among ones in benchmark studies in the Literature, i.e. **RMI**, **RS** and **PGM**, described in Section 1.4. For the convenience of the reader, the description of the training phase for these models is provided in Section 3.2.3. We point out that we have modified the **SOSD** library so that an implementation of a Learned Index can use both Branchy and Branch-free versions of Binary and k-ary Searches. In what follows, we refer to a Learned Index that uses **BFS** as Branch-free and to one that uses **BBS** as Branchy. An analogous terminology holds for k-ary Search.

```
1  .cfi_startproc
2  movq 8(%rdi), %rdx ;    move n into rdx
3  movq (%rdi), %r8 ; move a into r8
4  cmpq $1, %rdx ; compare n and 1
5  movq %r8, %rax ; move base into rax
6  jbe .L2 ; quit if n <= 1
7  .L3:
8  movq %rdx, %rcx ; put n into rcx
9  shrq %rcx ; rcx = half = n/2
10 leaq (%rax,%rcx,4), %rdi ; load &base[half] into rdi
11 cmpl %esi, (%rdi) ; compare x and base[half]
12 cmovb %rdi, %rax ; set base = &base[half] if x > base[half]
13 subq %rcx, %rdx ; n = n - half
14 cmpq $1, %rdx ; compare n and 1
15 ja .L3 ; keep going if n > 1
16 .L2:
17 cmpl %esi, (%rax) ; compare x to *base
18 sbbq %rdx, %rdx ; set dx to 00..00 or 11...11
19 andl $4, %edx ; set dx to 0 or 4
20 addq %rdx, %rax ; add dx to base
21 subq %r8, %rax ; compute base - a (* 4)
22 sarq $2, %rax ; (divide by 4)
23 ret
24 .cfi_endproc
```

LISTING 4.1: **Assembly code for Uniform Binary Search** (see also Khuong and Morin, 2017). The while loop is implemented in lines 8-15, while the line 12 is the "conditional move".

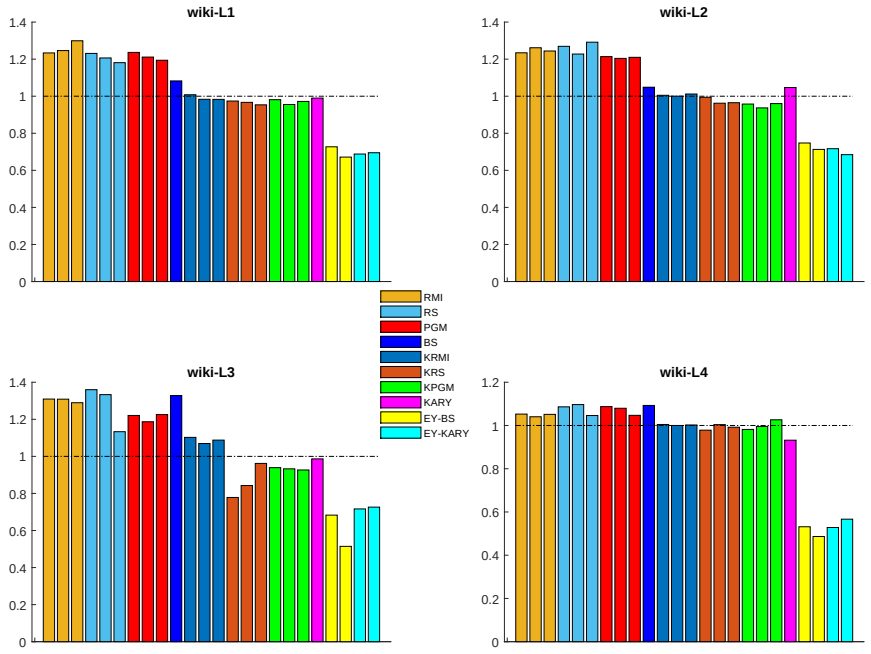


FIGURE 4.1: **Branch-free vs Branchy on the wiki dataset**. From left to right, **RMI**, **RS** and **PGM**, highest rank first. For each, and according to rank, the bar height indicates the ratio of Branch-free/Branchy Binary Search average query times for the three best models, reported by memory level. The following blue bar shows the same ratio for the two versions of Binary Search (indicated as **BS**). Next, we have analogous bar heights for k-ary instead of Binary Search (**KRMI**, **KRS** and **KPGM**). In magenta, the ratio of the two versions of k-ary alone (indicated as **KARY**). The last two bars are the average query times ratios of **BFE/BBS** and **BFE/BFS** respectively. y. The last two bars report the homologous ratios for k-ary Search. A bar height below one indicates that Branch-free indexing is better than its Branchy counterpart.

4.3 Experiments, Results and Discussion

4.3.1 Computational Experiments

For all the Learned Indexes returned by the training step, we execute query experiments using both their Branchy and Branch-free versions. Then, for each category, we take up to the three best average query times, when **SODS** returns more than three models. Finally, proceeding by ascending rank, we take the ratio query time Branch-free/Branchy. A ratio with value below one indicates that a Branch-free version of a model is superior to its corresponding Branchy version of equal rank. As anticipated in Section 4.2.3, as a baseline to confirm the findings in the Literature regarding Binary Search are also valid within **SOSD**, we report the average query time ratio of **BFS**/**BBS** and of **BBS** and **BFS** with **BFE** respectively. Moreover, we also consider those ratios for k-ary Search instead of Binary Search. All these results are reported on Figures C.1-C.5 in Appendix C and, for conciseness, we report here one case in Figure 4.1.

4.3.2 Analysis

Coherence of Literature Results within SOSD

In the Literature has been observed that **BFS** is able to improve the query times with respect to **BBS**, on synthetically generated data, only for the case of **L1** and **L2** memory levels. It is also to be remarked that *k*-ary Search is not considered. Our experiments, performed within the highly engineered and optimized **SOSD** platform and on real datasets provide a somewhat different picture. With reference to Figure 4.1 for the **wiki** dataset and Figures C.1-C.5 of the Appendix C for all datasets, we do not find that **BFS** is superior to **BBS** on **L1** and **L2** memory levels (see blu bar in the mentioned Figures). On the other hand, we find that **K-BFS** is slightly superior to **K-BBS** on all memory levels. However, it is quite comforting that, as in the Literature, **BFE** is better than Sorted Layout Binary Search (see yellow bars in the mentioned Figures).

The Relevance of Branch-free vs Branchy in Learned Indexing in SOSD: Search Time

The results concerning this part of the experiments are reported again in Figure 4.1 for the **wiki** dataset and Figures C.1-C.5 of Appendix C for all datasets. Those results are in agreement with the preceding point. However, they offer some insights that we now outline. When Learned Indexes use **BBS** as terminal for the search phase, their gain with respect to the use of **BFS** is in percentage superior to the gain obtained by using **BBS** as opposed to **BFS** on the entire tables. As for the Learned versions that use *k*-ary Search, there is a slight advantage of the the Branch-free version with respect to the Branchy one.

In view of the fact that we use ratios among the three best Branch-free models with their corresponding Branchy part and that, for models with the same rank can actually be different, we consider the ability of each of those models to reduce the search interval. Those results are reported in Table 4.1 for the case of wiki dataset, and Tables C.1-C.5 on Appendix C. It is easy to see that homologous models have basically the same ability to reduce the search interval.

The Relevance of Branch-free vs Branchy in Learned Indexing in SOSD: Space.

It is natural to ask whether the best performing Branch-free models, in terms of time, offer some gain with respect to their Branchy versions, in terms of model space. The relative results are presented in Figure 4.2. Although not uniform across datasets, the Branch-free versions offer some advantage with respect to their Branchy part.

4.4 Conclusions

In this Chapter, following the recommendations provided in the Literature, we have analysed how a programming paradigm making use of conditional moves can benefit over the use of if-then-else constructs in modern computer architectures. However, using a highly engineered platform and real benchmark datasets, we obtained results in contrast with the Literature regarding two different types of Binary Search. In particular, we have shown, via an extensive set of experiments on benchmark datasets that, for Learned Indexes, the choice of Branchy Binary or k-ary Search routine is the most appropriate for the final search stage in the **SOSD** software platform. Finally, we discussed how, although not uniform across datasets, Branch-free

	RMI		RS		PGM	
	BBS	**BFS**	**BBS**	**BFS**	**BBS**	**BFS**
L1	99.84%-5	99.86%-5	97.53%-91	97.53%-91	99.73%-9	45.48%-2033
L2	99.97%-9	99.95%-16	99.71%-91	99.71%-91	99.89%-33	99.89%-33
L3	100.00%-5	100.00%-9	99.99%-91	99.99%-91	100.00%-33	100.00%-9
L4	100.00%-30	100.00%-39	100.00%-91	100.00%-91	100.00%-33	100.00%-33
	K-BBS	**K-BFS**	**K-BBS**	**K-BFS**	**K-BBS**	**K-BFS**
L1	99.84%-5	99.86%-5	97.53%-91	97.53%-91	99.73%-9	99.73%-9
L2	99.97%-9	99.95%-16	99.71%-91	99.71%-91	99.94%-17	99.89%-33
L3	100.00%-5	100.00%-9	99.99%-91	99.99%-91	100.00%-9	100.00%-9
L4	100.00%-30	100.00%-39	100.00%-91	100.00%-91	100.00%-33	100.00%-65

TABLE 4.1: **Search Range for the wiki dataset.** The columns of table report the model classes. Each model class is divided into Branchy (**BBS**) and Branch-free (**BFS**) versions of Binary and k-ary Searches (in this latter case **K-BBS** and **K-BFS**). In each class, we consider the best performing models.The rows report the memory levels. Each memory level corresponds to a row in the table. For those rows, each entry contains the pair Reduction Factors in percentage - number of elements to search after a prediction is made.

Learned Indexes seems to use less space than their Branchy counterparts. It is also to be remarked that the Eytzinger Layout confirms to be competitive in time with respect to the sorted layout procedures, within **SOSD**. Unfortunately, none of the Learned Indexes implementations that we know can use that layout for the final search stage. This asks for Learned Indexes actually able to support that layout. In turn, those Indexes do not seem to be readily obtainable from the current ones, since their prediction stage is based on the CDF of the table, whose arrangement is closely related to a sorted layout. In the next Chapter, we show how to define a new Learned Generic Index that is able to use also layouts other than sorted in its finale stage.

Bibliographic Notes

Most of the references in this Chapter are illustrated in the Bibliographic Notes Section of Chapter 1. Concerning the platform used to train Learned Indexes, the references are provided in the Bibliographic Notes Section of Chapter 3. Listing 4.2.4 in Section 4.1 is provided by Khuong and Morin, 2017, who discuss the difference between Branchy and Branch-free Binary Search more in-depth via an experimental analysis on synthetic datasets.

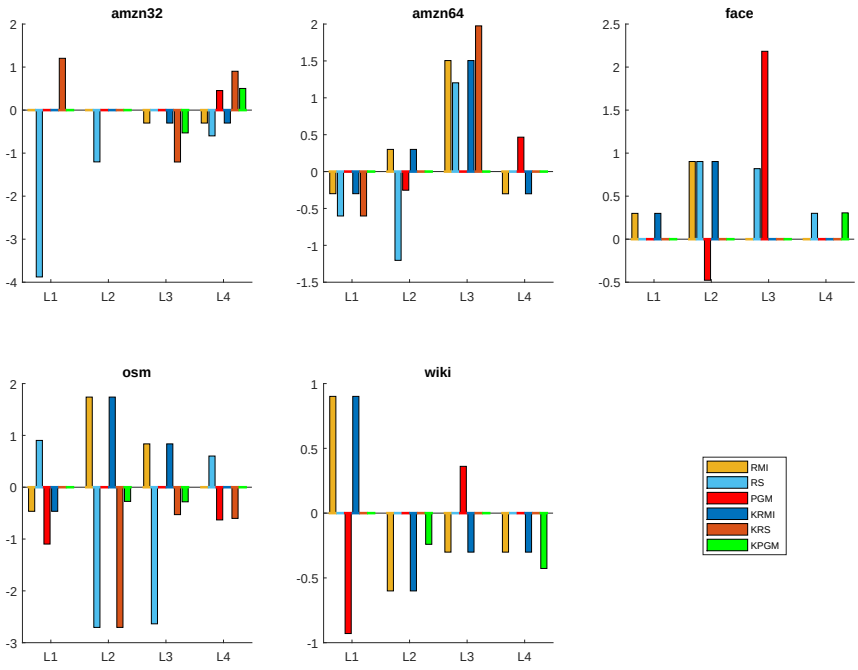


FIGURE 4.2: **Model size ratios of all the models:** The model size ratios Branch-free/Branchy of the fastest (query times) **RMI**, **RS**, **PGM**, **KRMI**,**KRS**, **KPGM** computed on each dataset, for each memory level. The y scale is logarithmic (base 10). Negative values indicate the case when, for each Learned Index, the fastest one using Branch-free Search routines has a model size less than the corresponding fastest one using Branchy routines, with zero values indicating equal model sizes.

Chapter 5

Generic Learned Static Sorted Sets Dictionaries

This Chapter provides a new generic type of Learned Dictionary for Static Sorted Search, which differs from those mentioned so far in that it is able to use, as its final stage for searching, various types of Dictionaries instead of only a specific one, e.g. Binary Search. In particular, starting from the definition of a Static Dictionary Data Structure given in Section 1.1, we define two paradigmatic types of Generic Learned Static Sorted Sets Dictionaries. Then, we provide instances for each of them, mentioning some construction and query time bounds, when they are not available in the Literature. Since the Specific Learned Dictionaries can be seen as boosters of elementary search procedures, we investigate the boosting property of those Generic Learned Static Sorted Sets Dictionaries, in addition to a comparison with the specific ones, provided in the Literature. In particular, this Chapter is organized as follows.

- In Section 5.1, we define a paradigm for Generic Learned Dictionaries.
- In Section 5.2, we describe a first family of Generic Learned Dictionaries, which is based on the division of the Universe U into fixed length intervals.
- In Section 5.3, we describe another family for this paradigm, which is based on the division of the Universe U into variable length intervals.
- In Section 5.4, we illustrate the experimental methodology adopted in this Chapter.
- In Section 5.5, we illustrate and discuss the results obtained in our experiments.
- In Section 5.6, we provide conclusions on the experimental analysis of the preceding Section.

The Bibliographic Notes relevant for this Chapter are in the corresponding Section.

The software used for the experimentation described in this Chapter is available on Github[1] while the relative datasets are available on a repository[2].

5.1 From Specific to Generic Learned Dictionaries

The models that have been proposed so far for Learned Indexes and described in Section 1.4, when queried, all return an index i into the table and an approximation

[1] https://github.com/globosco/An-implementation-of-Generic-Learned-Static-Sorted-Sets-Dictionaries

[2] http://math.unipa.it/lobosco/LSTS/

ϵ accounting for the error in predicting the position of the query element within the table. Then, the search is finalized via Binary Search in the intervals $[i - \epsilon; i + \epsilon]$ of A. However, other search routines such as Interpolation Search can be used for the final search stage. In theoretic terms, those Learned Models yield search procedures that, in the worst case scenario, are no worse than the basic routines we have mentioned earlier, provided that the prediction can be made in $O(log\ n)$ time. However, in practice, the extensive experiments reported in the Literature show that they can be thought of as very impressive boosters of the mentioned routines. So, in the following Sections of this Chapter, we investigate whether it is possible to reproduce this "boosting effect" by modifying these Learned Models so that they can be used on Data Structures that are more complex than the simple search routines on Sorted Sets.

5.1.1 Models Specific for Binary and Interpolation Search

Binary Search and analogous procedures have the flexibility that they need only pointers in the table to perform the search. Therefore, they are very well suited to be used in the final search stage of the models that have been proposed so far for Learned Indexing. On the other hand, the predictions of those Learned Models naturally fit the sorted table layout required by standard routines. Unfortunately, for the search stage, if one wants to use a data structure that needs more than the two pointers or an array layout different than the sorted one, all models we know of cannot be used, at least in their current operational definition. For a more in-depth description of these models, the reader is refereed to the division into a class hierarchy characterising model space in Section 1.4.2.

5.1.2 Models for Generic Dictionaries

First, we start by recalling the definition of a *static sorted sets dictionary* $\mathcal{SD}$ as a Data Structure that supports the operations $member(x)$, $PSP(x)$ and $range(x, y)$, as described in Section 1.1. Therefore, we are interested in a Generic Model that can be applied on a wide variety of Dictionaries instead of just simple search routines in Sorted Sets, i.e. Binary Search and Interpolation Search. For this reason, we introduce the definition of a new kind of model.

Definition 5.1.1. *A Generic Model of type D for table A is a "black box" that returns an explicit partition of the sorted universe U into intervals, with the elements of A assigned to intervals and kept sorted. A visit of the partition from left to right provides A. Moreover, in $O(\log n)$ time, given an element $x \in U$, it provides as output the unique interval in the partition where x must be searched for, in order to assess whether it is in A or not.*

The main difference between the models characterized by Definition 5.1.1 and the ones used so far for Learned Indexing is that, apart from the partition of U being explicit, no two elements can have an intersecting prediction interval. It is to be noted, however, that the **PGM** Index can be easily transformed into a Generic Model, as outlined in Section 5.3. In principle, multi-layer **RMI**s, with a tree structure, can also be adapted to be Generic Models. However, due to the way they are implemented and "learned" right now, such a transformation would require a major reorganization of their implementation and "learning" code. The same considerations apply to the **RS** Model. For those reasons, among the models proposed so far, we consider only the **PGM** as Generic Model in the following Sections.

Definition 5.1.2. *Let $\hat{D}$ be an instance of a Generic Model of type D for A, consisting of k intervals and let $\mathcal{SD}$ be a Dictionary. A Learned Dictionary $\mathcal{D}$ is obtained by building separately an $\mathcal{SD}$ for each sorted set within each interval in the partition. In order to answer a query, $\hat{D}$ returns a pointer to the $\mathcal{SD}$ built on the appropriate interval, which is then queried.*

Generic Models can be subdivided into two families: The ones in which the intervals of the partition are of fixed length and the ones in which their lengths is variable. We discuss an example of the first type in Section 5.2. The method, overlooked so far within the development of Learned Indexes, is related to the estimation of probability density function via histograms, as well as Data Structures introduced for Dynamic Interpolation Search. As for the second type and as already anticipated, we discuss the **PGM** in Section 5.3.

5.2 Learned Dictionaries: The Case of Equal Length Intervals - Binning

5.2.1 Construction

Fix an integer $k = O(n)$. The universe U is divided into k bins $B_1, \cdots, B_k$, each representing a range of integers of size $\frac{A[n]-A[1]}{k}$. Each bin has associated the interval of elements of A falling into its range. Let $\mathcal{SD}$ be a Dictionary. Its Learned version D_k is built as follows. For each of the mentioned intervals, we build a Data Structure $\mathcal{SD}$ containing the elements in that interval. Moreover, there is an auxiliary array such that its j-th entry provides a pointer to the data structure assigned to bin j, which may be empty (no elements in the bin). As for query, given an element $x \in U$, the bin in which we should search is identified via the formula $i = \lceil \frac{(x-A[1])k}{A[n]-A[1]} \rceil$.

When $\mathcal{SD}$ is one of the Dictionaries described in Section 1.2.1, the corresponding Learned version is particularly simple. Indeed, the k pointers to the chosen $\mathcal{SD}$ can be the start and the end of each interval. Then, the elements of A in that interval are rearranged according to the selected layout. That is, we obtain a new array, which is a succession of the same layout type applied to each interval.

Theorem 5.2.1. *D_k can be built in worst case (a) in linear time, if $\mathcal{SD}$ can be build in $O(c)$ time, for c elements; (b) $O(n \log n)$ time, if $\mathcal{SD}$ can be build in $O(c \log c)$ time, for c elements; (c) $O(n \log \log n)$ time, if $\mathcal{SD}$ can be build in $O(c \log \log c)$ time, for c elements.*

Proof. Point (a) is straightforward. As for (b), the total time to build the k data structures is given by $O(\sum_{i=1}^{k} c_i \log c_i)$ which, by the convexity of the log funtion, is bounded by $O(n \log n)$. Point (c) follows along the same lines as (b), for the convexity of the log log function.

□

5.2.2 Worst Case Search Time

We now bound the time to seach for a given element $x \in U$ in A, using D_k. Let the gap ratio be $\Delta = \frac{G_{max}}{G_{min}}$, where G denotes the distance between two consecutive elements in A. Notice that $G_{min} > 0$, since A is a set, implying that Δ is finite. The following theorem is adapted from results already known in the Literature.

Theorem 5.2.2. *Given an element $x \in U$ in A, the time to search for it in the model D_k, built for $\mathcal{SD}$, is $O(\log \min(n, \frac{n\Delta}{k}))$, assuming that searching in $\mathcal{SD}$ can be done in logarithmic time. Analogous bounds hold for log-log searching time.*

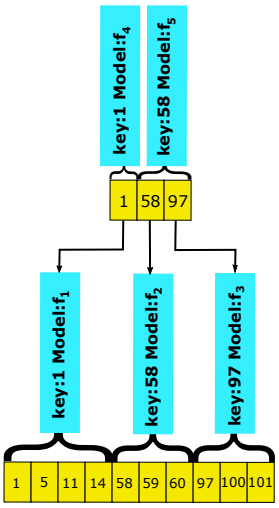


FIGURE 5.1: **Examples of a PGM Index**. At the bottom, the table is divided into three parts, according the maximum error ϵ. A new table is so constructed and the process is iterated. The keys in the last level give a partition of the Universe U. In this example, the partition P is $\{[1,57],[58,96],[97,101]\}$

Proof. Since the first part of the bound comes from the fact that each bin cannot have more than n elements, we show next that the number of elements of A falling into each bin of D_k is bounded by $\frac{n\Delta}{k}$.

G_{max} is at least $\frac{A[n]-A[1]}{n}$. Indeed, the bound is satisfied with equality when all elements are evenly spaced in U. Consider now the variation, with respect to evenly spaced, for an arbitrary placement of the elements in the bins. If the distance between two elements decreases, then other distances either stay as in the even spaced case increase. Now, G_{min} can be rewritten as $\frac{G_{max}}{\Delta}$, which is lower bounded by $\frac{A[n]-A[1]}{n\Delta}$. Since each bin has width $\frac{A[n]-A[1]}{k}$, no more than $\frac{A[n]-A[1]}{kG_{min}}$ elements can be in each bin. Using the lower bound for G_{min}, the result follows. □

5.3 Learned Dictionaries: The Case of Variable Length Intervals - The PGM

5.3.1 Construction

Fix an integer $\epsilon \leq \frac{n}{2}$. As already described in Section 1.4.2, a **PGM** Index can be built using a Piecewise Linear Approximation (**PLA** for short), so that a prediction at each node can be made by a linear model guaranteeing a maximum error ϵ. As anticipated in Section 5.1.2, the **PGM**, thanks to its built, can be easily transformed into a Generic Model. In fact, as illustrated in Figure 5.1, keys in the nodes of the last level provide a partitions of the Universe U with k variable length intervals. As shown in the previous Section, for each of the mentioned intervals, we build a Data Structure $\mathcal{SD}$ containing the elements in that interval.

Theorem 5.3.1. *D_k can be built in worst case (a) linear time, if $\mathcal{SD}$ can be build in $O(c)$ time, for c elements; (b) $O(n \log\ n)$ time, if $\mathcal{SD}$ can be build in $O(c \log\ c)$ time, for c elements; (c) $O(n \log\log\ n)$ time, if $\mathcal{SD}$ can be build in $O(c \log\log\ c)$ time, for c elements.*

Proof. As already known in the Literature, the **PGM** can be built in $O(n)$ time. So, D_k can be built in a time equal to the sum of the time to build the **PGM** and the time to

build the Dictionaries in each interval. As for point (a), the time is $O(n) + O(\sum_{i=1}^{k} c_i)$. As $\sum_{i=1}^{k} c_i = n$, we can conclude that the time is $O(n)$. Due to the convexity of the *log* and *loglog* functions, points (b) and (c) follows along the same line as (a). □

5.3.2 Worst Case Search Time

We now bound the time to search for a given element $x \in U$ in A, using D_k. The following lemma is derived from one presents in the Literature.

Lemma 5.3.2. *Given an element $x \in U$ in A, the time to search for it in the model D_k, built for $\mathcal{SD}$, is $O(\log n)$, assuming that the search in $\mathcal{SD}$ can be done in logarithmic time.*

Proof. It has been proven in the Literature that the optimal number of segments determined by the **PLA** algorithm is $m_{opt} \leq \frac{n}{2\epsilon}$. Having fixed $\epsilon \leq \frac{n}{2}$, in the worst case, that is the case with the maximum error, we have exactly one segment which contains all the n elements of A. From this comes a search time equal to $O(\log n)$ □

5.4 Experimental Methodologies

5.4.1 Hardware

We use the same hardware described in Section 3.2.1 of Chapter 4.

5.4.2 Datasets

We use the same datasets described in Section 4.2.2 of Chapter 4.

5.4.3 Dictionaries

In this Chapter, with reference to Dictionaries described in Section 1.2, we use the following methods to test the "boosting property" of the Generic Model introduced in Section 5.1.2.

- **Binary Search and Its Variants**. We use both Standard and Uniform Binary Search (**BBS** and **BFS** respectively), described in Section 1.2.1.
- **Interpolation Search**. We use only the Standard Interpolation (**IBS**) described in Section 1.2.1, since the **TIP** implementation is not competitive as illustrated in Chapter 3.
- **Array Layout other than Sorted**. We use Eytzinger and B-Tree Layouts described in Sections 1.2.2. In particular, for the latter, we use two different values of B. We denoted those two layouts as **BFT512** and **BFT32k**, to indicate that they have nodes occupying 512 bytes and 32 kilobytes of memory, respectively.
- **Search Trees**. We use the three kinds of Search Trees described in Section 1.2.3, i.e. Self-Adjusting Binary Tree, B^+-Tree and CSS Tree.

5.5 Experiments, Results and Discussion

Two types of experiments have been conducted to study the efficiency of Generic Learned Dictionaries, as described in the following.

(a) **Boosting**. As anticipated in Section 5.1, we investigate whether Generic Learned Dictionaries provided the "boosting effect" already known in the Literature for Specific Learned Indexes. So, as anticipated in Section 5.1.2, we analyse the following two cases.

(1) **Binning**. For each dataset, we increase the space occupied by the Generic Learned Dictionaries, growing the number of bins in percentage with respect to the number of elements in the given Tables, i.e. from 0% to 100%, and we calculate the ratios between the query time of a Generic Learned Dictionary and the respective none learned method applied on the whole Sorted Table.

(2) **PGM**. For each dataset, we choose ϵ as a power of two in the interval $[1, \frac{n}{2}]$. That is, we built models with an increasing error that partitions the Universe U from very small intervals to the one that contains the whole dataset. Then, for each of these models, we calculate the ratios between the query time of the **PGM** as a Generic Learned Dictionary and the respective none learned method applied on the whole Sorted Table.

For the sake of clarity, for each of the two cases, a ratio under one indicates that the Generic Learned Dictionary performs better than the corresponding none learned method.

(b) **Competitiveness of Generic Learned Dictionaries with respect to Specific ones**. These experiments provide a comparison with the models that represent the State of the Art and that have already been extensively discussed in the previous Chapters. This experiment is only performed in the case of the Binning Learned Dictionaries, since the **PGM** is already considered in Chapter 3. In particular, in the following, we discuss two possible scenarios.

(1) **No Bounds on Space**. In this case, the use of space is not relevant. For each dataset and for each memory level, we have built a Binning Learned Dictionary for each $\mathcal{SD}$ in Section 5.4.3. Then, among all these Learned Dictionaries used, we have chosen the one with the smallest average query time. So, we have compared it with the best **RMI**, **PGM** and **RS** selected from the **SOSD** output, as previously illustrated in Section 3.4.3 of Chapter 3.

(2) **Bounds on Space**. This scenario introduce a condition of limited available space. Therefore, as in the experiments of Chapter 3, we have imposed three space bounds for each dataset and memory level. Then, we have selected the best Binning Learned Dictionary that satisfy these bounds. Finally, we have compared it with the best **RMI**, **PGM** and **RS**, which also satisfy the bounds, selected from the **SOSD** output. In addition, for the sake of completeness, for each bound, we report also the **SY-RMI** analysed in Chapter 3.

5.5.1 Boosting

Binning

Figure 5.2 reports the results concerning this part of the experiments for the **wiki** dataset. Additionally, Figures D.1-D.5 of Appendix D report results for all the datasets. Such figures show that the previously mentioned boosting property is effective on

each dataset excluding **face**, for all the memory levels and Binning Dictionaries, except **BFE** and **BFT32k**. In particular, **BFE** is never boosted on **L1** and **L2**. Instead, **BFT32k** does not get any boosting on **L1**. These differences could be imputable to logarithmic and at most linear query time complexity of the operations related to $\mathcal{SD}$s, which remains unchanged in complexity order in their boosted versions (see Theorem 5.2.2), but with an increase in the multiplicative constants which penalizes the case of small Table data size (**L1** and **L2**). However, it is important to point out that the face dataset is a worst-case scenario. In fact, this dataset is characterized by an empirical CDF that for the cases of **L1**, **L2**, **L3** is close to uniform (see Figure 5.8. The consequence is that the Standard Interpolation Search procedure is already effective on the dataset making useless its Learned versions. Furthermore, the **L4** Table data end with some outliers, causing a totally unbalanced binning including all the elements into the first bin except the last 21 (the outliers). In this case, the consequence is that the Learned procedures do not boost the Standard ones. In addition, it is useful to note that, as the space used by the model grows, the gain of Binning increases, confirming the existence of a space/time trade-off already discussed for Learned Indexes in Chapter 3.

PGM

Figure 5.3 reports the results concerning this part of the experiments for the **wiki** dataset. Additionally, Figures D.6-D.10 of Appendix D report results for all the datasets. Surprisingly, for the **PGM** as Generic Learned Dictionary, no boosting effect is consistently found for each dataset. We now explain such behaviour with one dataset and one procedure as an example, i.e., **osm_L4** and **BBS** respectively. With reference to Figure 5.4, we can observe that, although the search procedure is improved in its Learned version, the navigation time of the **PGM** structure is greater than the one taken by the stand-alone version. In contrast, in Figure 5.5, we can observe that we can calculate the interval index for the Binning Model very fast, obtaining an important improvement of the search procedures.

5.5.2 Competitiveness of Generic Learned Dictionaries with respect to Specific ones

Query Time: No Bound on Space

The results concerning this scenario are reported in Figure 5.6. As it can be noted, if no bound is imposed on the space, a Binning Model is competitive with the State of the Art only in memory level **L4**, with the exception of the **face-L4** dataset for the reasons discussed in the previous Section. Moreover, in memory level **L3**, although not competitive with **RMI**, it performs consistently better than **PGM** and **RS**. Finally, it is useful to note that in most cases the best Binning Model has Branchy Binary Search as its final stage. This provides more evidence that Standard Binary Search is the best choice as Learned Index final stage, as already illustrated in the Chapter 4.

Query Time: Bounds on Space

The results concerning this scenario are reported in Figures D.11-D.15 of Appendix D for all datasets. In this Section, we report, as discussed next, the particular case of **osm** dataset in Figure 5.7. It should be noted that, by imposing bounds on the space occupied by the model, the Binning Models are almost never competitive with the State of the Art Models and the **SY-RMI** introduced in Chapter 3. An exception is

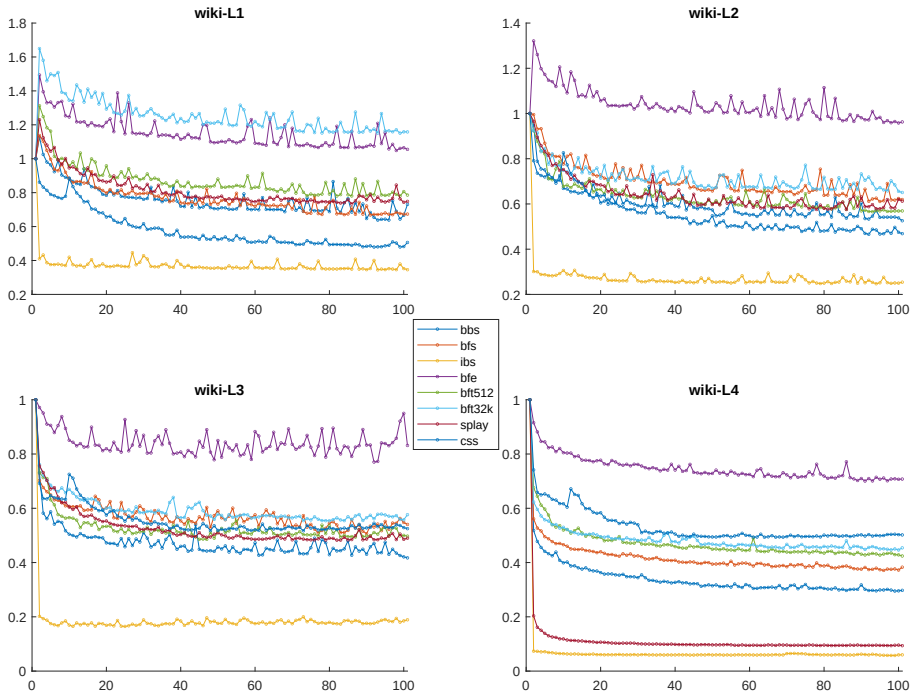


FIGURE 5.2: **Binning boosting property on wiki dataset**. For each memory level, we report in the abscissa axis the number of bins in percentage with respect of the number of elements in the Table. In the ordinate, we indicate the ratio between the mean query time of Binning and $\mathcal{SD}$ alone. For the sake of clarity, a ratio under one indicates that the Binning performs better than the simple ones.

the **osm** dataset in memory levels **L3** and **L4**. Only in these cases a Binning Model performs better than Learned Indexes in small space. This provides evidence that the Binning Models are able in a small space to well approximate very complex CDFs such as that of the **osm** dataset, shown in Figure 5.8.

5.6 Conclusions

In this Chapter, we have defined a new paradigm for Generic Learned Dictionaries for Static Sorted Search, capable of using a wide variety of Dictionaries. In particular, we have provided two families of them, the first based on the subdivision of the universe U into fixed-length intervals using a Binning technique, the second based on the use of the **PGM** Index to make explicit a partition into variable-length intervals. We have provided time bounds for the construction and the search for both. Then, considering that Learned Indexes in the Literature are boosters for the classic search procedures, we have studied the "boosting capacity" of these Generic Learned Dictionaries with different procedures. The results confirmed an excellent "boosting capacity" for Binning, while the **PGM** performs as an overcomplicated structure that defeats the purpose of improving search performance. Furthermore, we compared the Binning Dictionaries with the best Learned Indexes in the Literature, observing very good performance in large amounts of data, if space usage does not matter, and an ability to approximate very complex CDFs using very small space in addition to the Table.

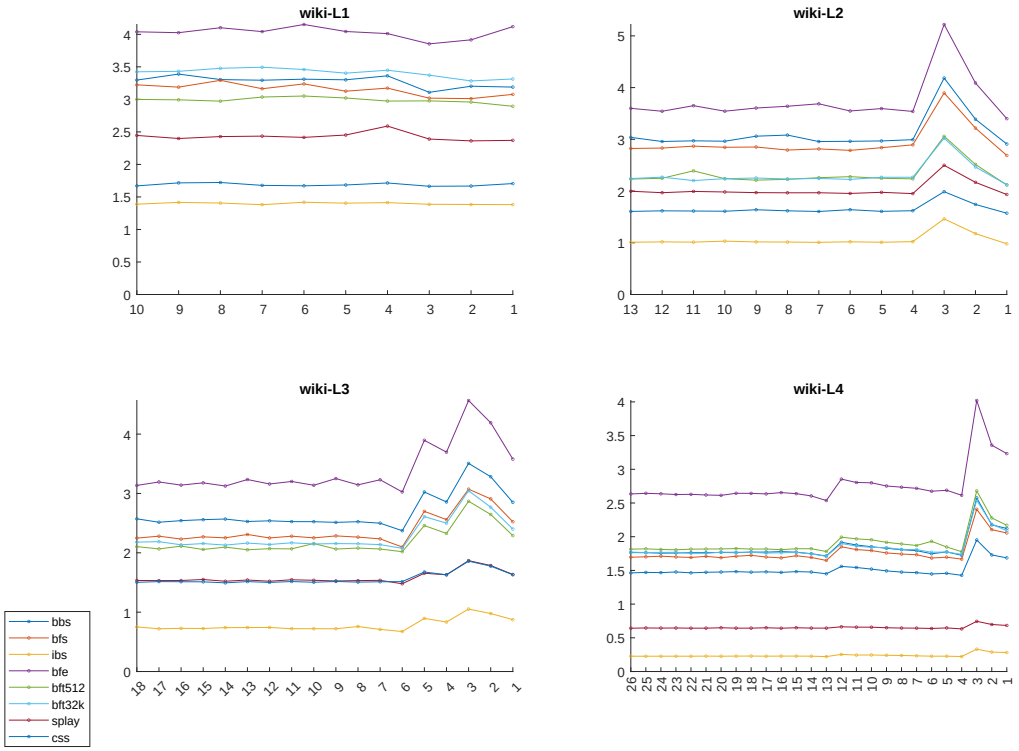


FIGURE 5.3: **PGM boosting property on wiki dataset**. For each memory level, we report in the abscissa axis the chosen ϵ for the **PGM** construction. In the ordinate, we indicate the ratio between the mean query time of the **PGM** and the $\mathcal{SD}$ alone. For the sake of clarity, a ratio under one indicates that the **PGM** performs better than the simple ones.

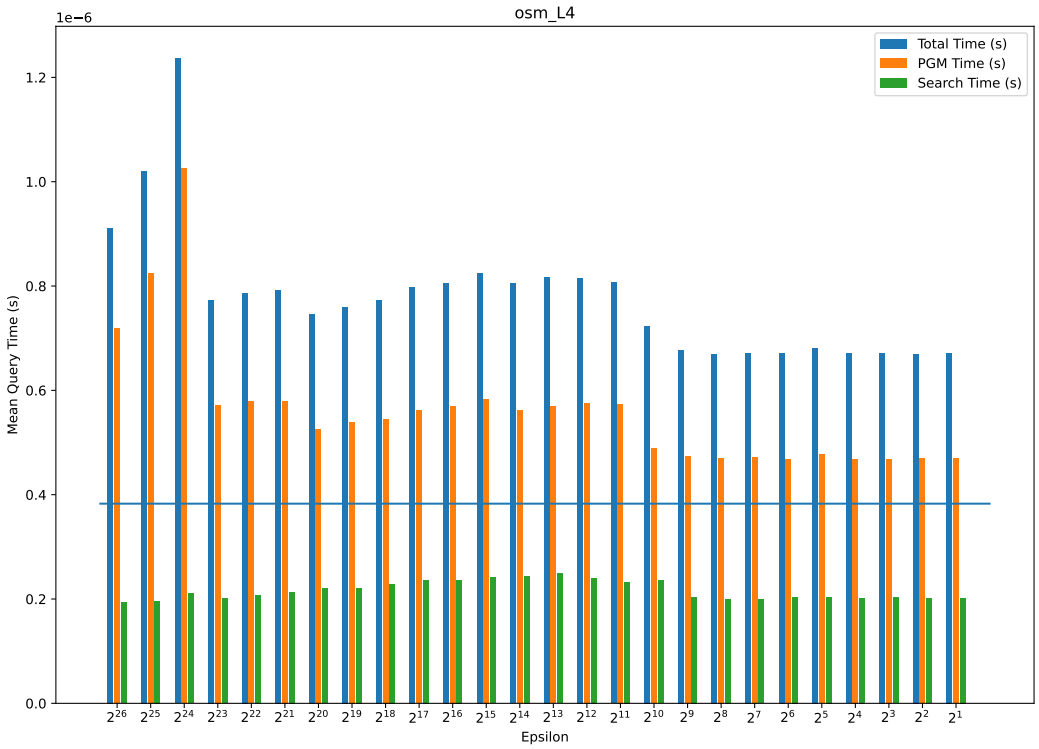


FIGURE 5.4: **PGM Query Time on osm_L4 dataset**. We report in the abscissa axis the chosen ϵ for the **PGM** construction, in the ordinate axis the mean query time expressed in seconds. The blue bars indicate the total time to execute a query using the **PGM** Dictionary. The orange bars show the time taken to navigate the **PGM** structure. The green bars report the time to search in the found interval using **BBS**.

Finally, the blue line is the **BBS** stand-alone mean query time.

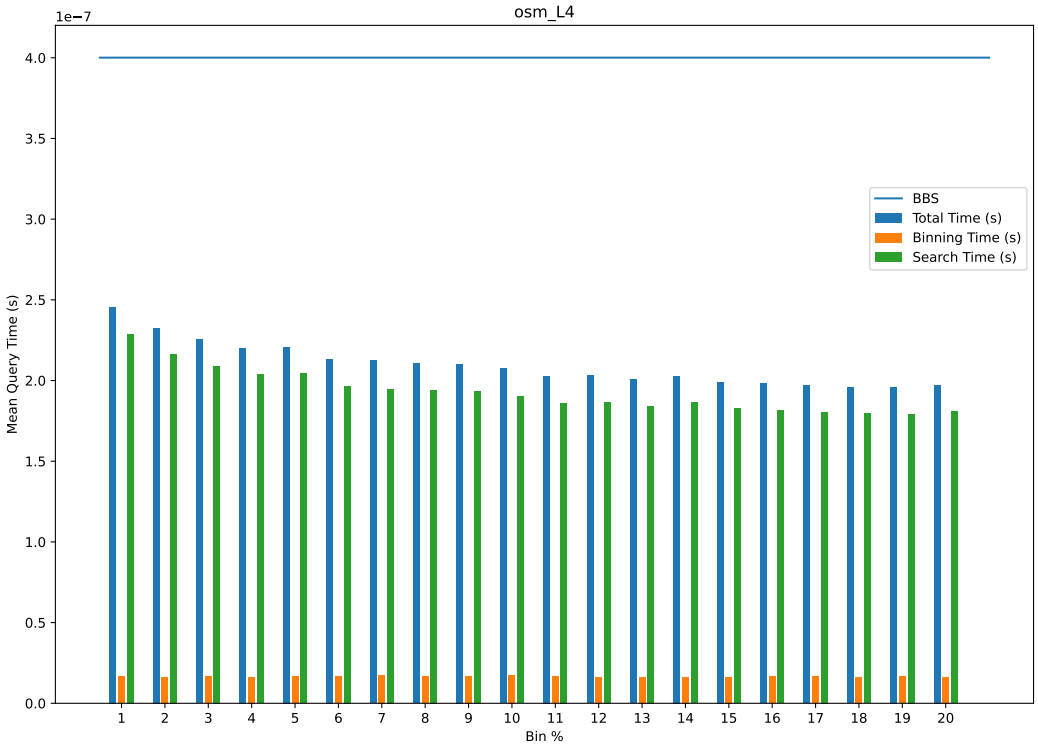


FIGURE 5.5: **Binning Query Time on osm_L4 dataset**. We report in the abscissa axis the chosen percentage for the Binning construction, in the ordinate axis the mean query time expressed in seconds. The blue bars indicate the total time to execute a query using the Binning Dictionary. The orange bars shows the time taken to calculate the bin index. The green bar reports the time to search in the found interval using **BBS**. Finally, the blue line is the **BBS** stand-alone mean query time.

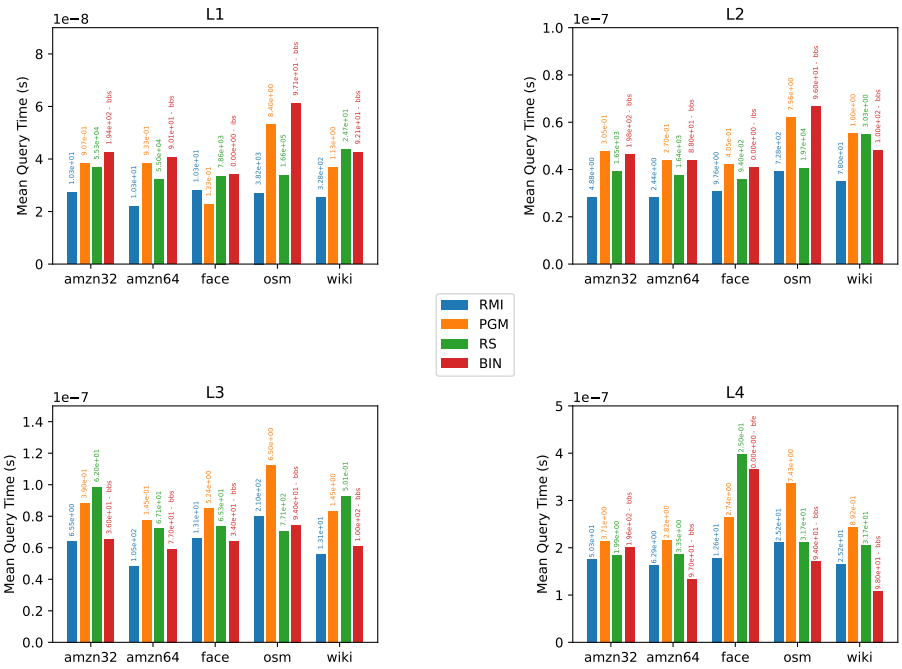


FIGURE 5.6: **Learned Indexes Query Time Without Bound on Space**. For each memory level and dataset, we report the mean query time of the best Learned Indexes including the Generic Learned Dictionary denoted with **BIN**. Above each bar we report the space in addition to the table in percentage. Indeed, we report the best search method in the **BIN** final stage.

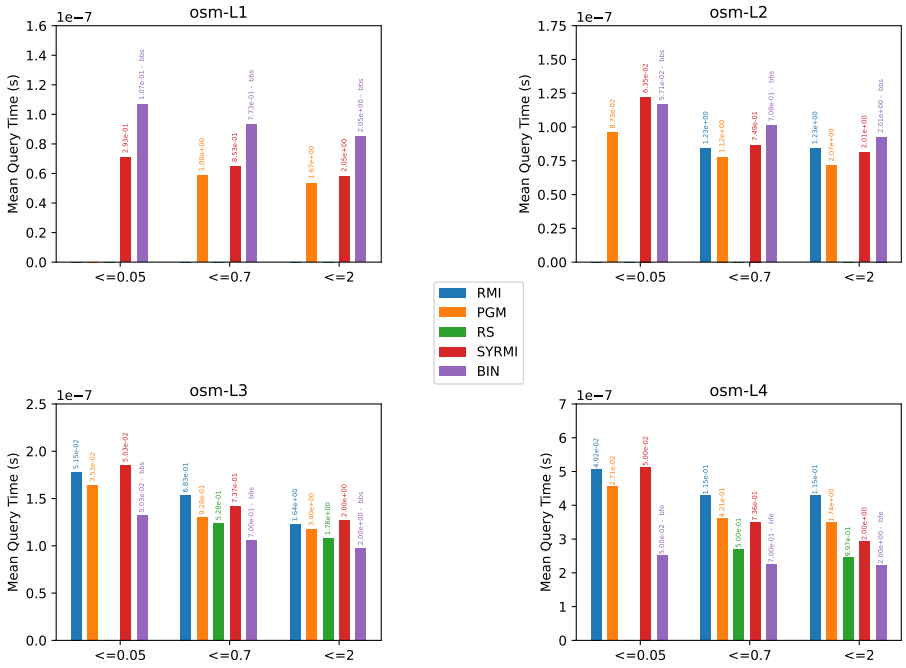


FIGURE 5.7: **Learned Indexes Query Time With Bounds on Space on osm dataset**. For each memory level, we choose three space bounds as in Chapter 3. For each space bound, from left to right, we report the mean query time of the best **RMI**, **PGM** and **RS** that satisfies the imposed bound. Next bar indicates the mean query time for the **SY-RMI** as in Chapter 3. The last bar is the mean query time for the best Generic Learned Dictionary with space inside the bound.

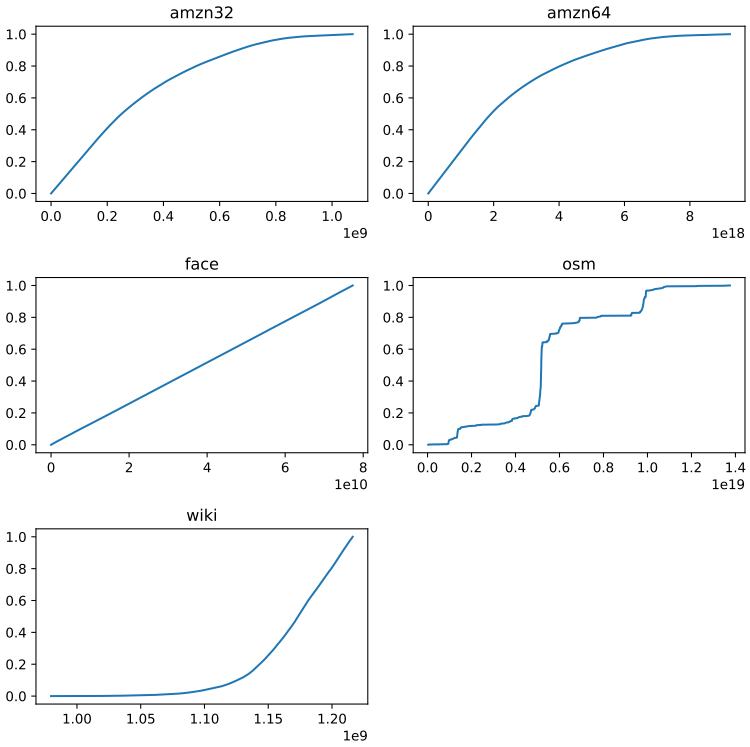


FIGURE 5.8: **Tables CDF**. For each dataset, we report the empirical *Cumulative Distribution Fuction* of the element in the Tables.

Bibliographic Notes

Most of the references in this Chapter are provided in the Bibliographic Notes Section of Chapter 1. The Fixed Length Generic Model in Section 5.2 is a generalization of the one proposed by Demaine, Jones, and Pătraşcu, 2004, who use it with a fixed number of bins equal to the number of the elements in the Table to improve Interpolation Search for non-Independent Data. The method to estimate a probability density function via histograms, mentioned in Section 5.1.2, it is well-known in the specialistic Literature that we refers to Freedman and Diaconis, 1981 for some details.

Chapter 6

Conclusions and Future Directions

In this Thesis, we have presented a new approach, which has become of growing importance in the Literature, for solving classic problems on Data Structures. In fact, although there are procedures such as Binary Search that are considered optimal according to many computational models, in the era of Big Data, it has become essential to seek ways to achieve substantial speed-ups even on procedures that are considered theoretically optimal. This search has led to the definition in the Literature of the so-called Learned Data Structures, which take advantage of models of Machine Learning, which, discovering particular patterns in the data, can be used as indexes for sorted sets. However, although these models seem to have an excellent performance in practice, some aspects, which could be very useful for designers and practitioners, have not been investigated so far in the Literature. In the following Sections, we summarise what have been the significant contributions in this Thesis and finally what are the future perspectives for extending these studies.

6.1 Advantage of Simple Models over Neural Networks

Starting from the considerations in the Literature on the use of new architectures such as GPU and TPU instead of CPU, we have provided for the first time experimental evidence on the existence of a data transfer bottleneck between these two kinds of architectures. This issue makes it premature to use very complex models, such as Neural Networks, that would benefit from the parallel computation performed by the GPU. In fact, in Chapter 2, although already known by specialists, we quantified for the first time through experimental results how very simple models can achieve significant table reductions and training and query times competitive in comparison to those obtained by Neural Networks models.

6.2 Learned Indexes in Small Space

In Chapter 3, we conducted a systematic experimental analysis of the ability of Learned Indexes to improve the performance of Binary and Interpolation Search queries. In particular, we studied the ability of those models to operate in small spaces. From the results obtained, we found that, although many of these models fail to achieve the best times in small space conditions, their prediction power is marginal in performance evaluation. Indeed, even if the query times between these models and the best ones differ by small constants, the difference in space can be several orders of magnitude. Therefore, this scenario reveals the need to ask not only which model is the best performing but also whether it is possible to obtain similar results but with a constant or small space with respect to the size of the Table.

6.3 On the Branchfreeness of Learned Indexes

Although there is a rather complex scenario in the Literature on the efficiency of Branchy versus Branch-free methods, mainly related to the memory hierarchy and data size, our results revealed in Chapter 4 that the choice of a Branchy procedure as the final stage of a Learned Index is the best one. However, we have pointed out that the Branch-free procedures in some cases, while presenting equivalent times to their Branchy counterparts, seems to use models that require less space in addition to the Table.

6.4 Generic Learned Dictionary

Starting from considering that Learned Indexes in the Literature can only use specific Dictionaries, i.e. Sorted Table Layouts, in Chapter 5, we have provided a new paradigm for the definition of Generic Learned Dictionaries, capable of operating on a wide range of Dictionaries. We then studied the boosting capabilities, already known for specific Learned Indexes, on two types of Generic Learned Indexes based on partitioning the Universe U into fixed or variable length intervals, respectively. From the results, we found that the former confirms an excellent boosting ability while the latter does not improve search performance due to a complicated indexing structure. An additional property of the Generic Learned Dictionaries with fixed-length intervals seems to be the ability to approximate very complex CDFs using very small space in addition to the Table.

6.5 Future Direction

From the contributions given in this Thesis comes a variegated scenario of models and procedures, whose performance in terms of time and space can depend on several factors, e.g. the computer architecture used or the distribution of the available data. Therefore, it could be useful for practitioners and designers to have access to a Benchmark platform that, given a sample of data and some customised options, could provide precise indications on which models, among all those most used in Literature, obtain the best performance in terms of time and additional space with respect to the Table.

A second consideration that could be made is that these Learned Indexes studied so far, except for a few, e.g. **PGM** Index, present a static structure that is poorly suited to many of today's scenarios where data is constantly evolving. So a new direction in this area could be to develop models, which, while maintaining the performance improvements of the search procedures, can handle the dynamic nature of the data.

Appendix A

Datasets

In this Thesis, we use five kinds of real datasets present in the Literature. In particular, we use 64 bits integers unless otherwise specified. We provide next a description of those datasets.

- **amzn**: Each key represents the popularity of a particular book from Amazon. We have two version of this dataset, one where each item is represent with 64 bits and another with 32.
- **face**: Each key represent an unique user ID from Facebook database.
- **osm** Each key represent an embedded location from Open Street Map database
- **wiki** Each key represent the time of editing from Wikipedia.

Furthermore, starting from those, we have produced, as described in Section A.1, datasets with different sizes. So that, we provide an experimentation, extensive and not available so far, of "how work" Learned Indexes across different memory levels.

Letting n the number of elements in a table, the details of the tables are the following.

- **Fitting L1 cache**. We choose $n = 3.7K$. For each dataset, the table corresponding to this type is denoted with the prefix **L1**, e.g., **L1_amzn**, when needed.
- **Fitting L2 cache**. We choose $n = 31.5K$. For each dataset, the table corresponding to this type is denoted with the prefix **L2**, when needed.
- **Fitting L3 cache**. We choose $n = 750K$. For each dataset, the table corresponding to this type is denoted with the prefix **L3**, when needed.
- **Fitting L4 cache**. We choose $n = 200M$, i.e. the entire datasets. For each dataset, the table corresponding to this type is denoted with the prefix **L4**, when needed.

We chose to fit only one memory line of L1 and L2 cache because our implementations use only one CPU Core. Instead, We decide to fit only half of the L3 cache because all CPU Core and processes share it.

As for query dataset generation, we extract uniformly and at random (with replacement) one million elements for each of the tables built as described above.

TABLE A.1: **The results of the Kolmogorov-Smirnov Test and of the KL divergence computation.**

	L1		L2		L3	
Datasets	%succ	KLdiv	%succ	KLdiv	%succ	KLdiv
amzn32	100	1.87e-05±1.41e-13	100	1.58e-04 ±8.76e-13	100	3.77e-03 ±2.20e-11
amzn64	100	9.54e-06±7.27e-14	100	7.88e-05±7.97e-13	100	1.88e-03±1.52e-11
face	100	1.98e-05±1.00e-12	100	7.98e-05±4.43e-13	100	1.88e-03±1.24e-11
osm	100	9.38e-06±4.51e-14	100	7.88e-05±3.46e-13	100	1.88e-03±9.55e-12
wiki	100	9.47e-06±5.27e-14	100	7.87e-05±5.64e-13	100	1.88e-03±1.25e-11

A.1 Kolmogorov-Smirnov Test and KL Divergence Computation

For each dataset, in order to obtain a CDF that resembles one of the original tables, we proceed as follows. First, we extract uniformly and at random a sample of the data of the required size. Then, we compute its CDF, and use a Kolmogorov-Smirnov test to assess whether the CDF of the sample is different from the dataset CDF. Finally, suppose the test returns that we cannot exclude such a possibility. In that case, we compute the sample's *probability density function* (PDF for short) and compute its KL divergence from the PDF of the dataset. Finally, we repeat this test 100 times for each table and choose the sample with the smallest KL divergence for our experiments.

In Table A.1, we report the percentage of times in which Kolmogorov-Smirnov test failed to find the difference between the two CDFs over the 100 extractions for each memory level and each dataset. Moreover, we also report the value of the KL divergence between the chosen generated dataset and the original one.

Appendix B

Learned Sorted Table Search and Static Indexes in Small Space: Supplementary Results

This Appendix summarizes the results discussed in Chapter 3.

B.1 Learning the CDF of a Sorted Table: Full Set of Experiments

Tables B.1-B.4 report the full set of experiments described and discussed in Section 3.4.1.

B.2 Constant Space Models: Full Set of Query Experiments

Figures B.1-B.5 report the full set of experiments described and discussed in Section 3.4.2.

B.3 Parametric Space Models: Full Set of Query Experiments

Figures B.6-B.10 report the full set of experiments described and discussed in Section 3.4.3. Table B.5 reports the average space, query time and Reduction Factor computed on all experiments performed in this study, normalized with respect to the best query time model coming out of **SOSD**.

TABLE B.1: **Training time for L1 tables, in seconds and per element**. The first column indicated the datasets. The remaining columns indicate the model used for the learning phase. Abbreviations are as in the main text. The **SOSD** columns refer to the entire output of that library, averaged over a number of models and elements in each table.

	L	Q	C	15O-BFS	SY-RMI 2%	SOSD RMI	SOSD RS	SOSD PGM
Datasets	**Training Time**							
amzn32	6.2e-07	4.3e-08	5.7e-08	8.0e-07	3.5e-06	1.2e-06	1.8e-06	5.4e-08
amzn64	7.3e-08	8.7e-08	1.0e-07	5.3e-07	2.6e-07	3.6e-07	2.8e-06	6.2e-08
face	3.0e-07	7.7e-08	8.5e-08	5.5e-07	7.5e-06	6.7e-07	8.2e-07	5.7e-08
osm	6.9e-08	7.4e-08	9.9e-08	4.6e-07	4e-05	4.6e-06	5.5e-06	5.0e-08
wiki	6.8e-08	1.4e-07	7.9e-08	9.0e-07	6.9e-06	3.7e-07	8.7e-06	4.6e-08

TABLE B.2: **Training time for L2 tables, in seconds and per element.**
The table legend is as in Table B.1

	L	Q	C	15O-BFS	SY-RMI 2%	SOSD RMI	SOSD RS	SOSD PGM
Datasets	**Training Time**							
amzn32	1.0e-08	7.7e-08	1.4e-07	1.1e-07	5.6e-06	4.6e-07	2.3e-07	4.2e-08
amzn64	1.6e-08	2.7e-07	3.1e-07	1.8e-07	5.2e-06	5.6e-07	3.5e-07	5.0e-08
face	1.4e-08	2.7e-07	2.8e-07	1.0e-07	4.1e-06	4.6e-07	1.1e-07	3.9e-08
osm	1.3e-08	2.7e-07	2.9e-07	1.2e-07	2.8e-04	2.9e-05	6.9e-07	4.0e-08
wiki	1.7e-08	2.7e-07	2.7e-07	1.0e-07	7.8e-06	9.3e-07	1.0e-06	3.7e-08

TABLE B.3: **Training time for L3 tables, in seconds and per element.**
The table legend is as in Table B.1

	L	Q	C	15O-BFS	SY-RMI 2%	SOSD RMI	SOSD RS	SOSD PGM
Datasets	**Training Time**							
amzn32	1.1e-08	1.7e-08	2.3e-08	5.3e-08	4.5e-06	5.0e-07	1.7e-08	2.7e-08
amzn64	1.6e-08	2.0e-08	1.9e-08	6.3e-08	1.5e-06	1.3e-07	2.4e-08	3.4e-08
face	8.2e-09	2.1e-08	1.9e-08	3.9e-08	1.5e-05	1.6e-06	1.4e-08	2.4e-08
osm	1.6e-08	1.9e-08	2.0e-08	4.4e-08	1.2e-05	1.3e-06	3.5e-08	3.8e-08
wiki	1.5e-08	2.0e-08	1.9e-08	4.1e-08	2.3e-06	2.2e-07	5.1e-08	3.7e-08

TABLE B.4: **Training time for L4 tables, in seconds and per element.**
The table legend is as in Table B.1

	L	Q	C	15O-BFS	SY-RMI 2%	SOSD RMI	SOSD RS	SOSD PGM
Datasets	**Training Time**							
amzn32	7.9e-09	1.4e-08	1.4e-08	3.7e-08	1.2e-06	1.2e-07	9.5e-09	2.4e-08
amzn64	7.9e-09	1.4e-08	1.4e-08	3.7e-08	1.1e-06	2.2e-07	2.1e-08	5.0e-08
face	8.0e-09	1.4e-08	1.4e-08	3.6e-08	1.3e-06	2.5e-07	2.1e-08	6.5e-08
osm	8.0e-09	1.4e-08	1.4e-08	3.6e-08	1.2e-06	2.5e-07	2.2e-08	7.4e-08
wiki	7.9e-09	1.4e-08	1.4e-08	3.6e-08	1.1e-06	2.2e-07	1.9e-08	4.1e-08

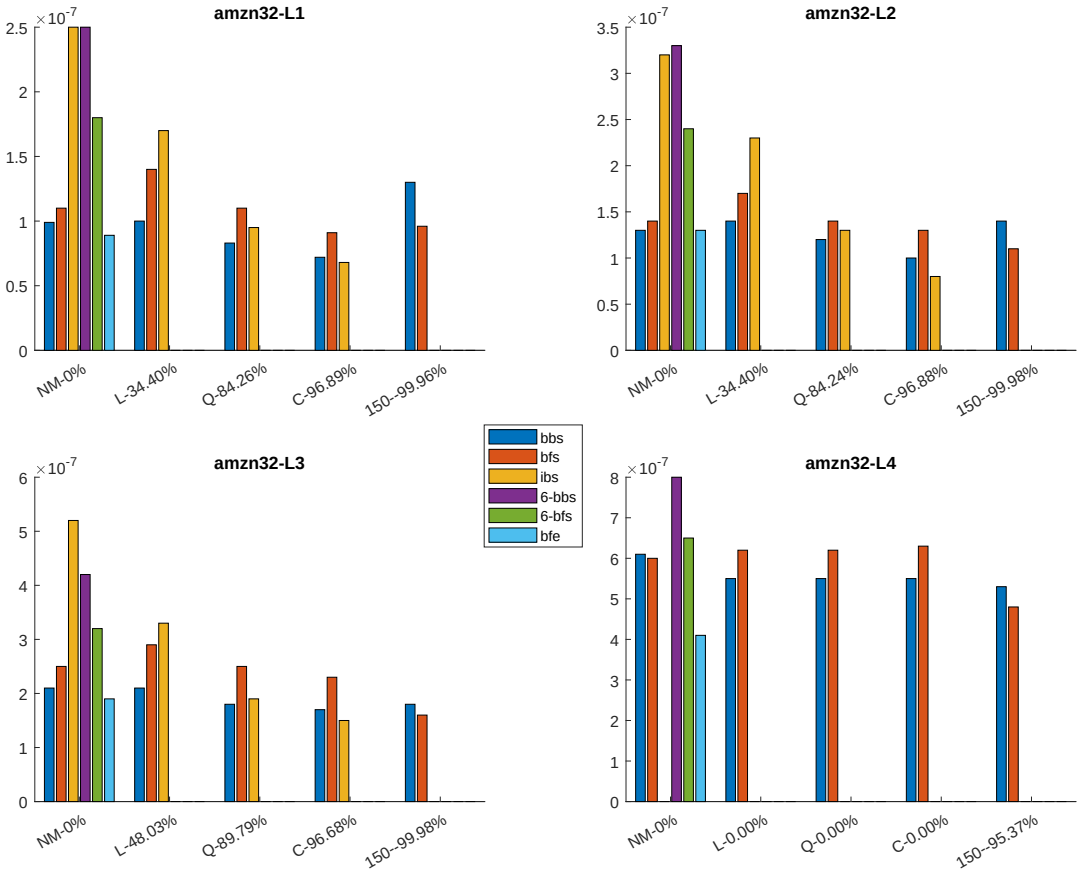


FIGURE B.1: **Query times for the amzn32 dataset on Sorted Table Search Procedures**. The methods are the ones in the legend. For each memory level, the abscissa reports methods grouped by model. From left to right, no model, linear, quadratic, cubic and **KO-**, with $k = 15$, and with **BFS** and **BBS** as search methods. **K-BFS** is reported with $k = 6$. For those latter, the Reduction Factor corresponding to the table is also reported. On the ordinate, it is reported the average query time, in seconds. For memory levels **L4**, **IBS**, **L-IBS**, **Q-IBS** and **C-IBS** have been excluded, since the inclusion of their query time values (1.4e-05, 1.6e-05, 1.6e-05, 1.4e-05, respectively) would make the histograms poorly legible.

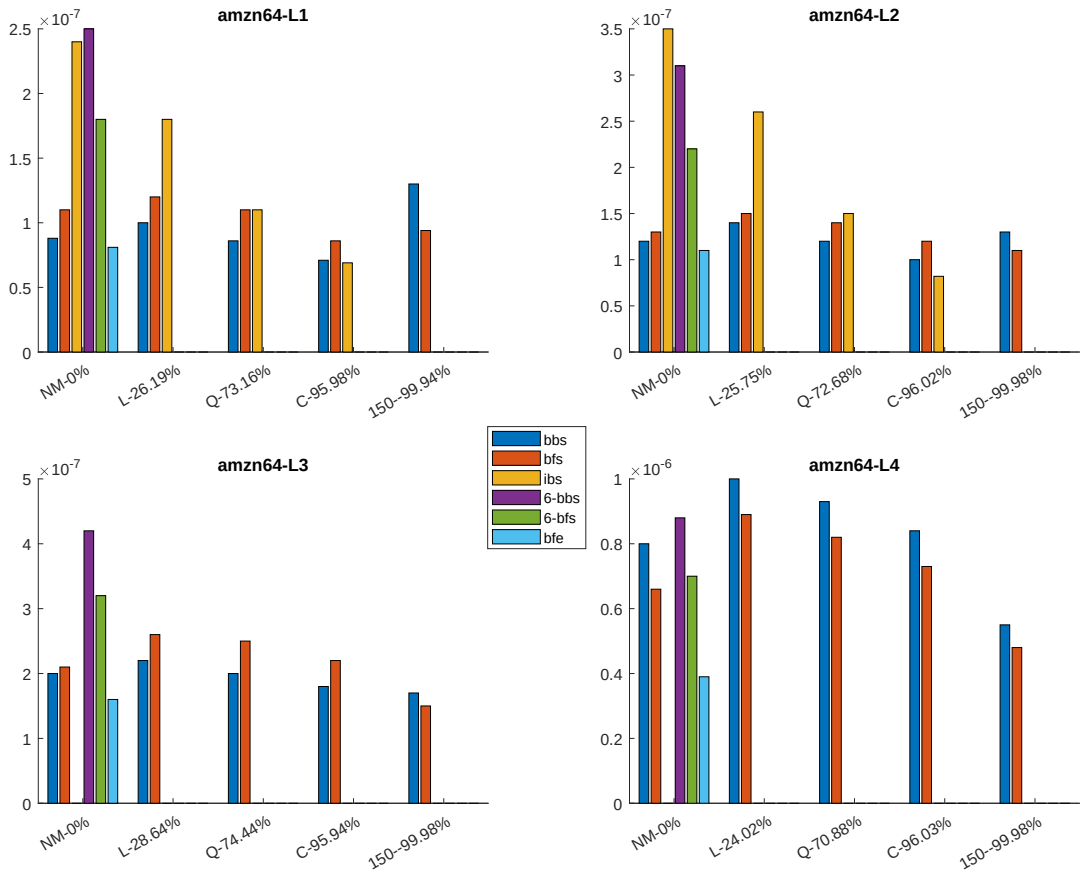


FIGURE B.2: **Query times for the amzn64 dataset on Sorted Table Search Procedures**. The figure legend is as in Figure B.1. For memory level **L4**, **IBS**, **L-IBS** and **Q-IBS** have been excluded, since the inclusion of their query time values (3.1e-06, 2.1e-06, 1.2e-06, respectively) would make the histograms poorly legible.

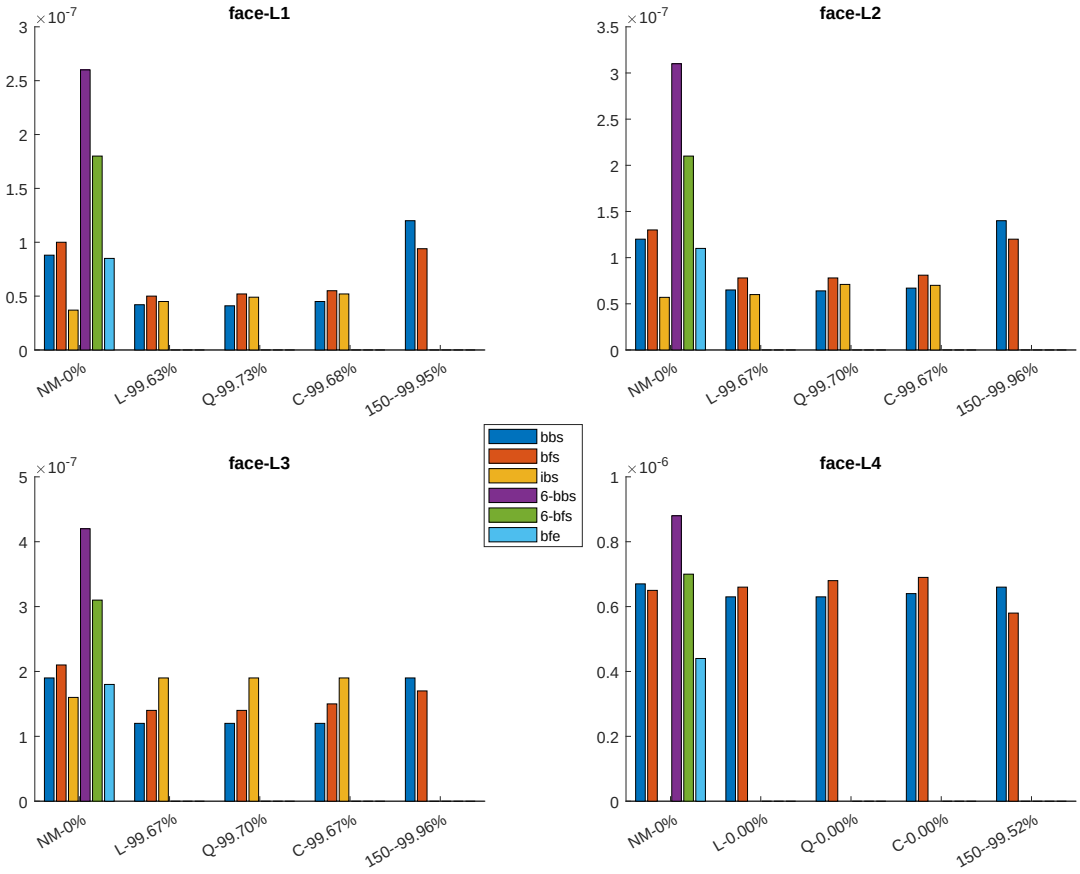


FIGURE B.3: **Query times for the face dataset on Sorted Table Search Procedures**. The figure legend is as in Figure B.1. For memory levels **L4**, **IBS** and its Learned versions have been excluded because of their poor performance (data not shown and available upon request).

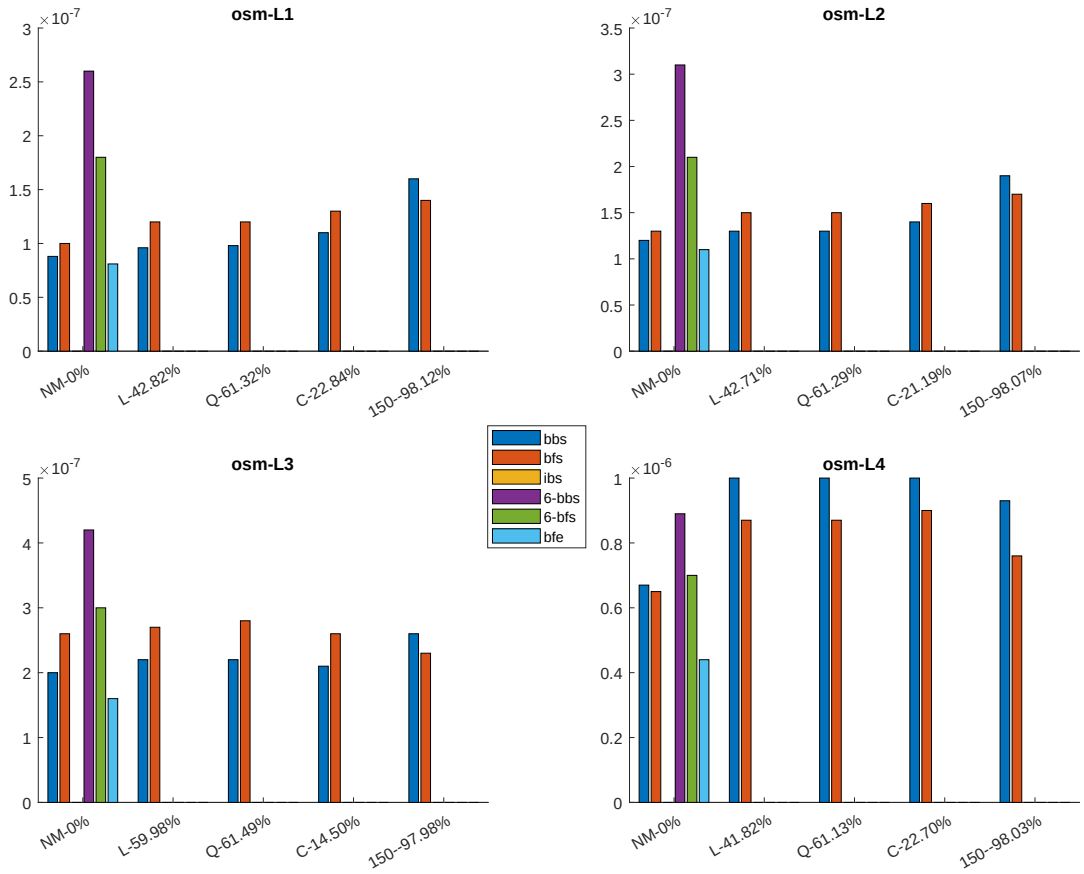


FIGURE B.4: **Query times for the osm dataset on Sorted Table Search Procedures**. The figure legend is as in Figure B.1. For all memory levels, **IBS** has been excluded, since the inclusion of its query time values (1.2e-06, 2.2e-06, 6.5e-06, 6.4e-05, respectively) would make the histograms poorly legible. Its regression-based Learned versions have been excluded for the same reason (data not shown and available upon request). However, they have better query time performance with respect to **IBS**, in particular **Q** and **C**.

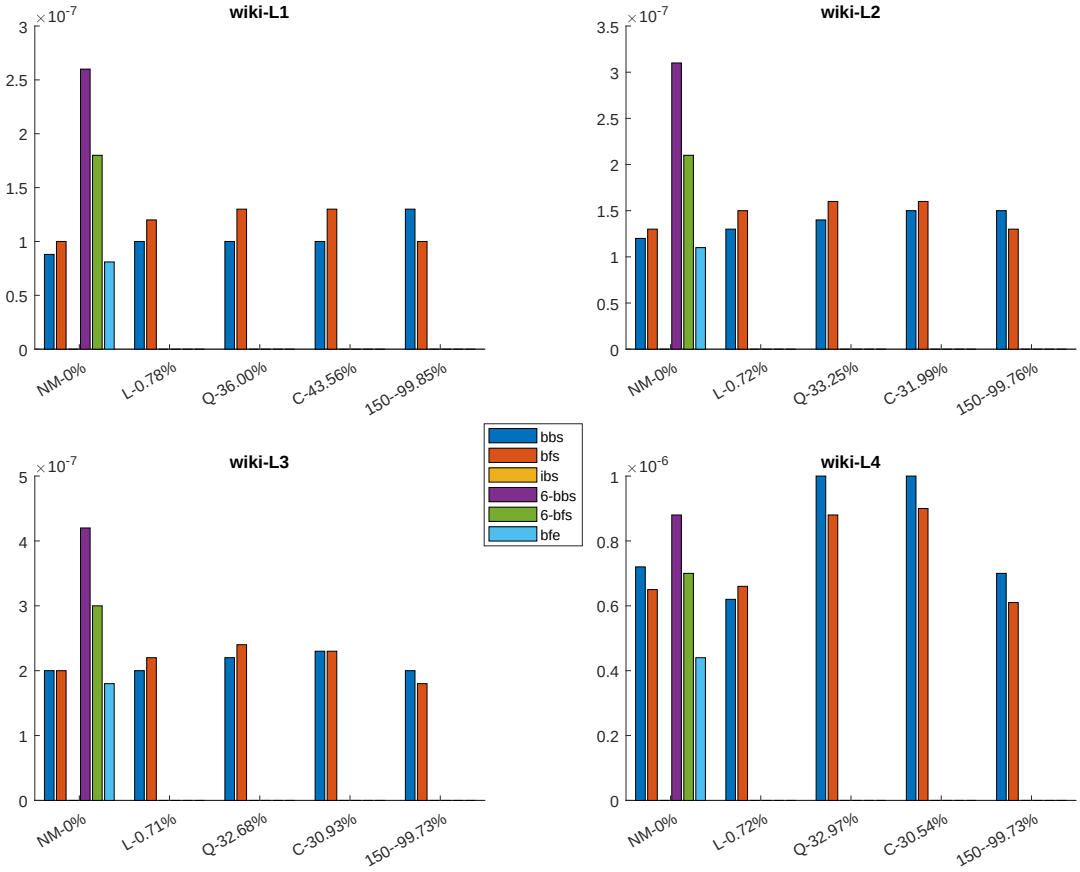


FIGURE B.5: **Query times for the wiki dataset on Sorted Table Search Procedures**. The figure legend is as in Figure B.1. For all memory levels, **IBS** has been excluded, since the inclusion of its query time values (3.1e-07, 5.1e-07, 9.5e-07, 5.1e-06, respectively) would make the histograms poorly legible. Its regression-based Learned versions have been excluded for the same reason (data not shown and available upon request). However, they have better query time performance with respect to **IBS**, in particular **Q** and **C**.

L1			
	Time	Space	RF
Best **RMI**	2.5e-08	7.7e+03	99.86
PGM 0.70	1.8e00	6.5e-05	1e00
SY-RMI 0.05	2.6e00	3.8e-05	4.2e-01
PGM 0.05	2.6e00	1.7e-05	2.3e-01
Best **PGM**	1.5e00	1.2e-04	1e00
PGM $\leq$	1.5e00	3e-04	1e00
PGM 2	1.6e00	1.1e-04	1e00
SY-RMI 2	1.7e00	2.7e-04	9e-01
SY-RMI 0.70	2e00	1.1e-04	9e-01
RS ≤ 10	2e00	9.5e-04	9.5e-01
B-Tree ≤ 10	2e00	1.1e-03	9.9e-01
Best B-Tree	1.7e00	3.4e-02	1e00
Best **RS**	1.3e00	7.4e00	1e00
RMI ≤ 10	-	-	-

L2			
	Time	Space	RF
Best **RMI**	6.1e-08	1.3e+01	100
SY-RMI 0.05	1.8e00	3.9e-03	9.9e-01
PGM 0.05	2e00	3.5e-03	1e00
SY-RMI 0.70	1.4e00	5.7e-02	1e00
PGM 2	1.6e00	7.1e-02	1e00
PGM 0.70	1.6e00	3.4e-02	1e00
RMI ≤ 10	1.2e00	3.3-01	1e00
SY-RMI 2	1.4e00	1.5e-01	1e00
Best **PGM**	1.5e00	2.1e-01	1e00
PGM ≤ 10	1.5e00	2.1e-01	1e00
RS ≤ 10	1.5e00	3.15e-01	1e00
B-Tree ≤ 10	2.3e00	5.2e-01	1e00
Best B-Tree	2.3e00	9.2e-01	1e00
Best **RS**	1.2e00	1.4e+01	1e00

L3			
	Time	Space	RF
Best **RMI**	3.1e-08	1.4e+03	99.98
SY-RMI 0.05	3e00	4.5e-05	7.9e-01
SY-RMI 0.70	1.6e00	5.4e-04	9.6e-01
PGM 2	1.7e00	5.6e-04	1e00
PGM 0.70	1.8e00	3.1e-04	1e00
PGM 0.05	3.1e00	2.2e-04	7.7e-1
RMI ≤ 10	1.2e00	3.9e-03	1e00
SY-RMI 2	1.4e00	1.4e-03	9.9e-01
PGM $\leq$	1.6e00	1.3e-03	1e00
Best **PGM**	1.6e00	1.4e-03	1e00
RS ≤ 10	2e00	2.6e-03	9.9e-01
BTree ≤ 10	2.3e00	5.1e-03	1e00
Best B-Tree	2e00	1.5e-01	1e00
Best **RS**	1.3e00	3.4e00	1e00

L4			
	Time	Space	RF
Best **RMI**	1.8e-07	1.5e+01	99.97
PGM 0.05	1.8e00	3.3e-03	1e00
SY-RMI 0.05	2.1e00	3.3e-03	8e-01
PGM 2	1.5e00	8e-02	1e00
PGM 0.70	1.6e00	6.5e-02	1e00
SY-RMI 0.70	1.9e00	4.9e-02	8e-01
RMI ≤ 10	1.1e00	2.7e-01	1e00
Best **RS**	1.3e00	6.7e-01	1e00
RS ≤ 10	1.3e00	1.7e-01	1e00
Best **PGM**	1.4e00	2.3e-01	1e00
PGM ≤ 10	1.4e00	3e-01	1e00
SY-RMI 2	1.6e00	1.3e-01	8e-01
Best B-Tree	2.3e00	2.1e-01	1e00
BTree ≤ 10	2.3e00	1.7e-01	1e00

TABLE B.5: **A Synoptic Table of Space, Time and Accuracy of Models** For each memory level, the models are listed on the rows. The first one provides the best performing method for that memory level, on each of the datasets used in this research. The columns indicate average query time in seconds, average additional space used by the model and the average of the empirical Reduction Factor. The remaining column entries report analogous parameters, for each model, normalized with respect to the best one. For each parameter, we take the ratio Model/best model.

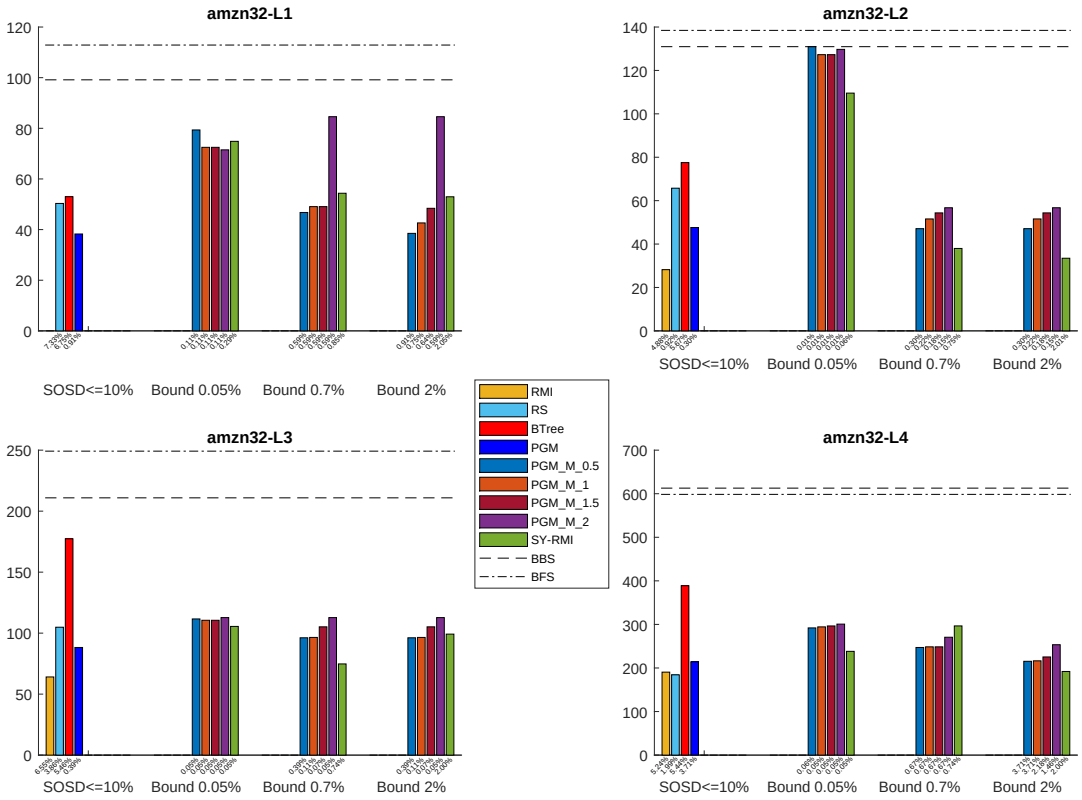


FIGURE B.6: **Query times for the amzn32 dataset on Learned Indexes in Small Space.** The methods are the ones in the legend (middle of the four panels, the notation is as in the main text and each method has a distinct colour). For each memory level, the abscissa reports methods grouped by space occupancy, as specified in the main text. When no model in a class output by **SOSD** takes at most 10% of additional space, that class is absent. The ordinate reports the average query time, with **BBS** and **BFS** executed in **SOSD** as baseline (horizontal lines).

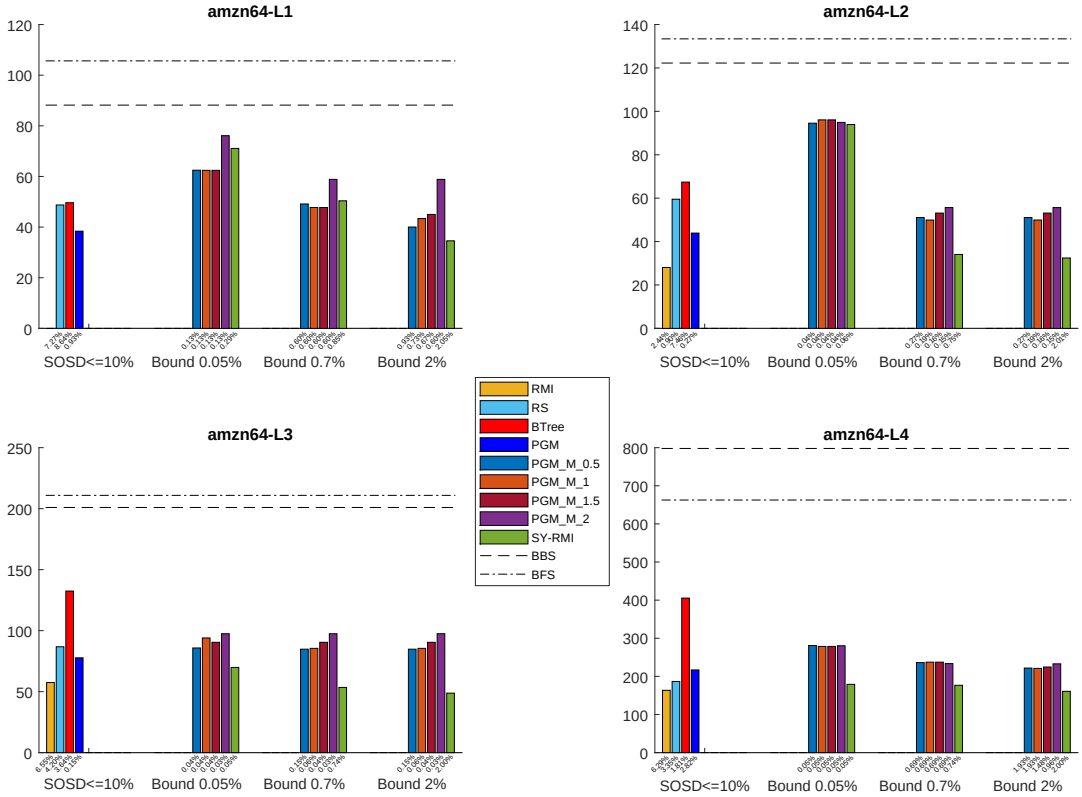


FIGURE B.7: **Query times for the amzn64 dataset on Learned Indexes in Small Space.** The figure legend is as in Figure B.6.

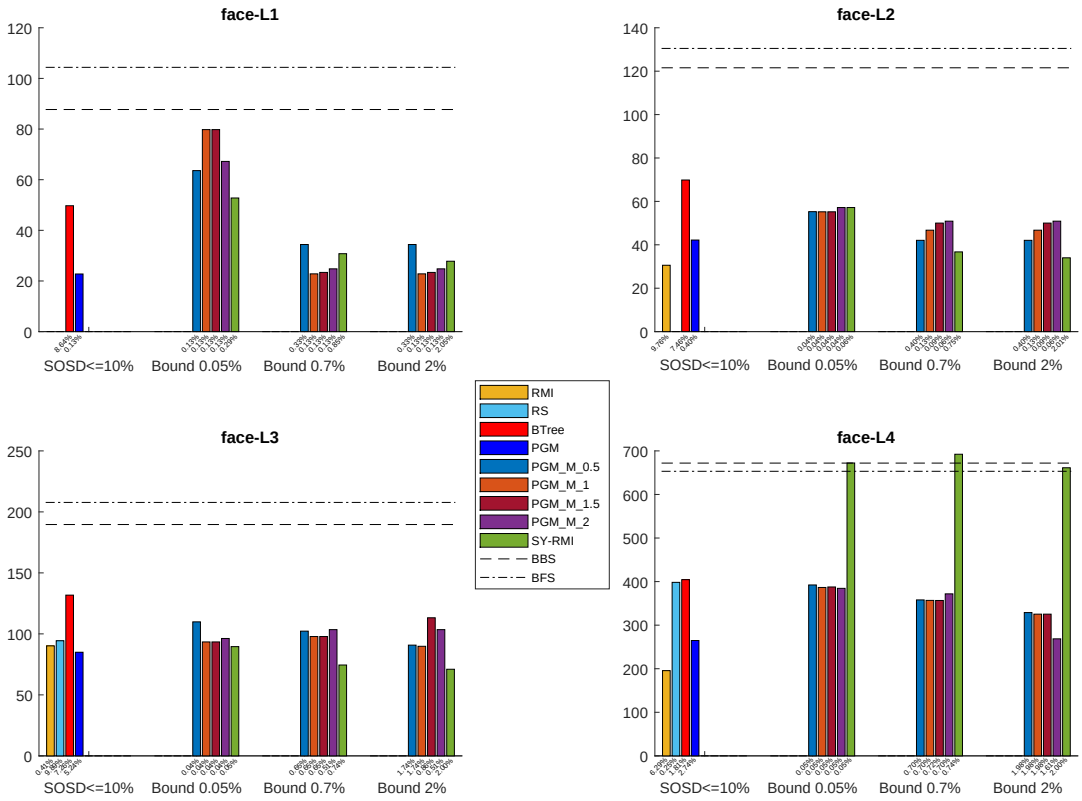


FIGURE B.8: **Query times for the face dataset on Learned Indexes in Small Space.** The figure legend is as in Figure B.6.

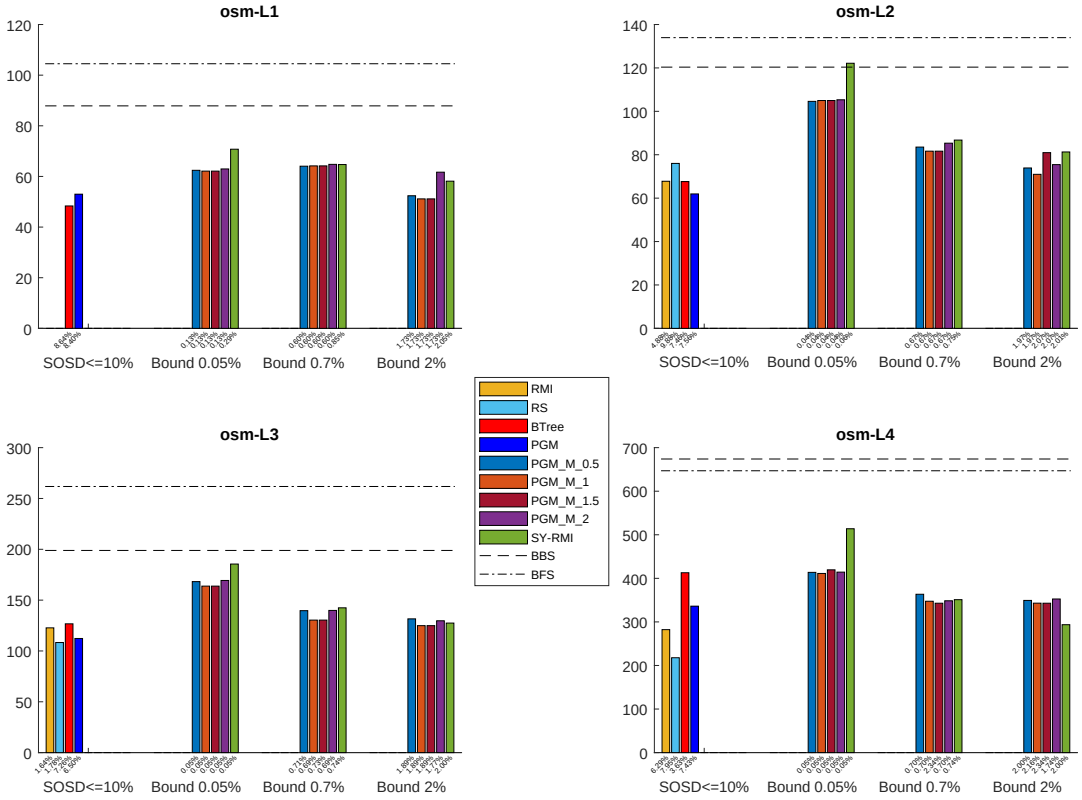


FIGURE B.9: **Query times for the osm dataset on Learned Indexes in Small Space.** The figure legend is as in Figure B.6.

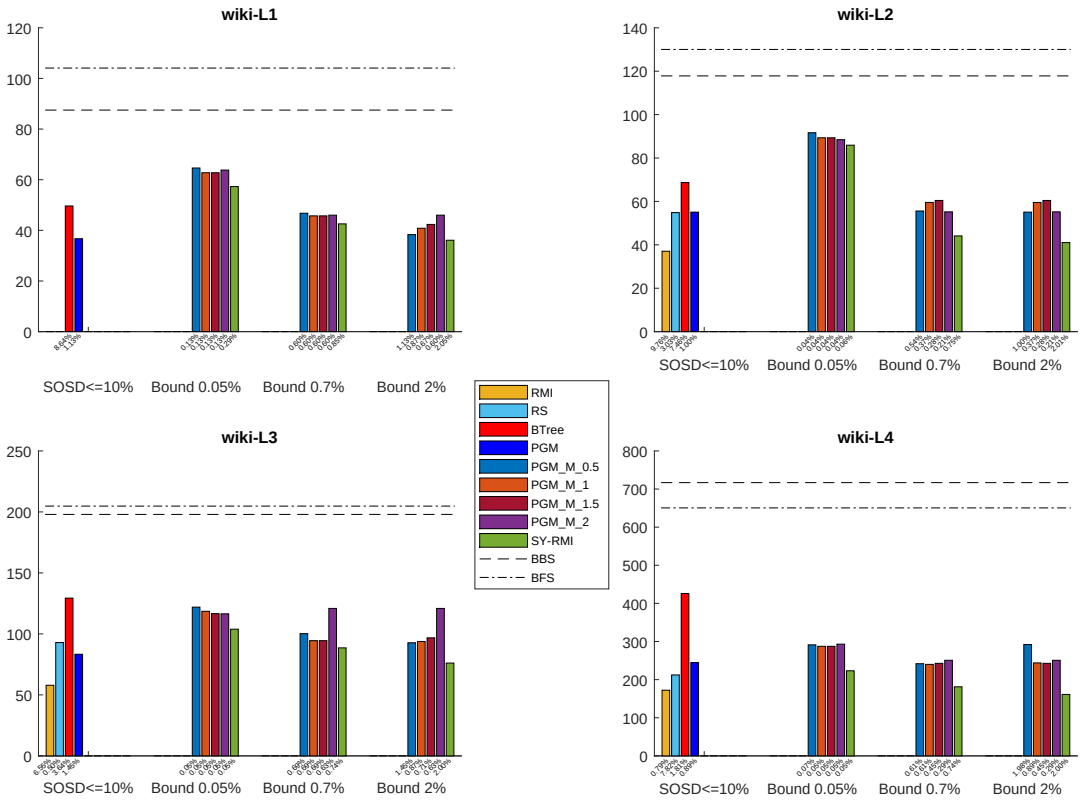


FIGURE B.10: **Query times for the wiki dataset on Learned Indexes in Small Space.** The figure legend is as in Figure B.6.

Appendix C

Standard Vs Uniform Binary Search and Their Variants in Learned Static Indexing: Supplementary Results

This Appendix summarizes the results discussed in Chapter 4.

C.1 Experiments with SOSD

For all the considered datasets, Figures C.1-C.5 plot, from left to right, the query times ratio between Branchy and Branch-free Binary Searches (**BFS/BBS**) as final stage of best **RMI**, **RS**, **PGM** models. The following blue bar shows the **BFS/BBS** query times without using any index. Next, we have analogous bar for k-ary Search instead of Binary (**KRMI**, **KRS**, **KPGM**). In magenta, the ratio of the two versions of k-ary alone (indicated as KARY). The following two bars are the average query times ratios of **BFE/BBS** and **BFE/BFS** respectively.The last two bars report the homologous ratios for k-ary Search.

Tables C.1-C.5 report Reduction Factors (in percentage) and the number of elements of the interval reduced by the best performing **RMI**, **RS**, **PGM** indexes, when using **BBS**, **BFS**, **K-BBS** and **K-BFS**, for all the considered datasets.

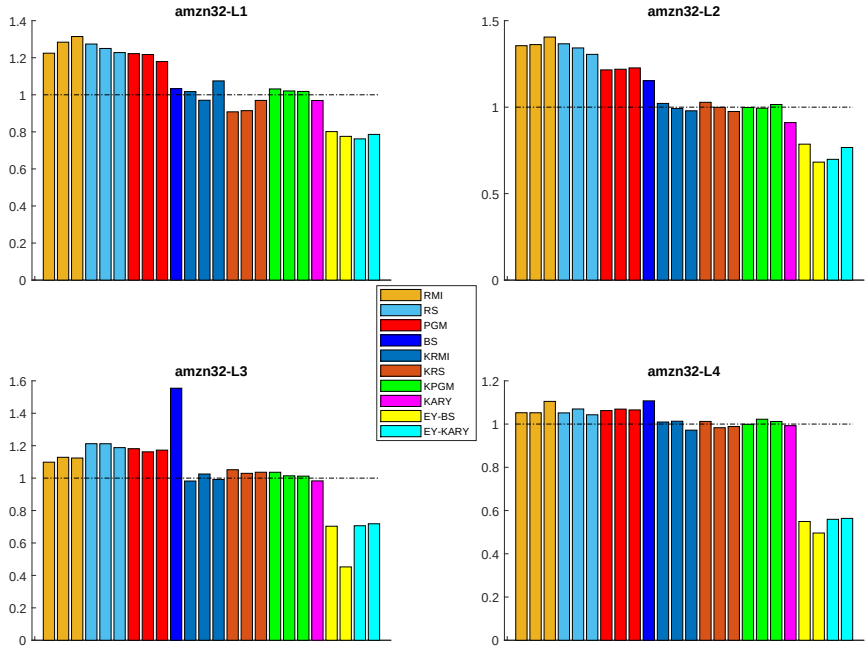


FIGURE C.1: **Branch-free vs Branchy on the amzn32 dataset**. From left to right, **RMI**, **RS** and **PGM**, highest rank first. For each, and according to rank, the bar height indicates the ratio of Branch-free/Branchy Binary Search average query times for the three best models, reported by memory level. The following blue bar shows the same ratio for the two versions of Binary Search (indicated as **BS**). Next, we have analogous bar heights for *k*-ary instead of Binary Search (**KRMI**, **KRS** and **KPGM**). In magenta, the ratio of the two versions of *k*-ary alone (indicated as **KARY**). The last two bars are the average query times ratios of **BFE/BBS** and **BFE/BFS** respectively. The last two bars report the homologous ratios for *k*-ary Search. A bar height below one indicates that Branch-free indexing is better than its Branchy counterpart.

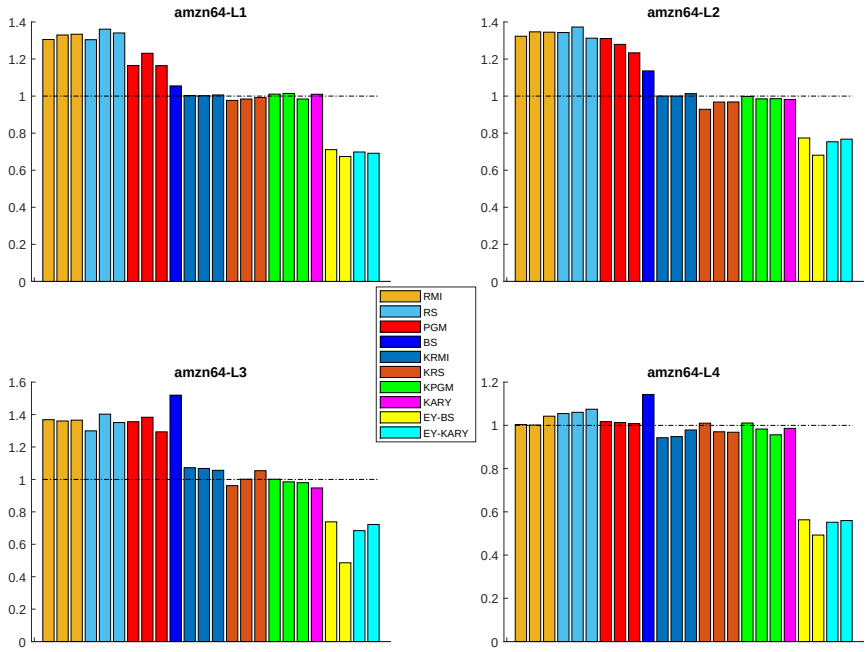


FIGURE C.2: **Branch-free vs Branchy on the amzn64 dataset**. The figure legend is as in Figure C.1.

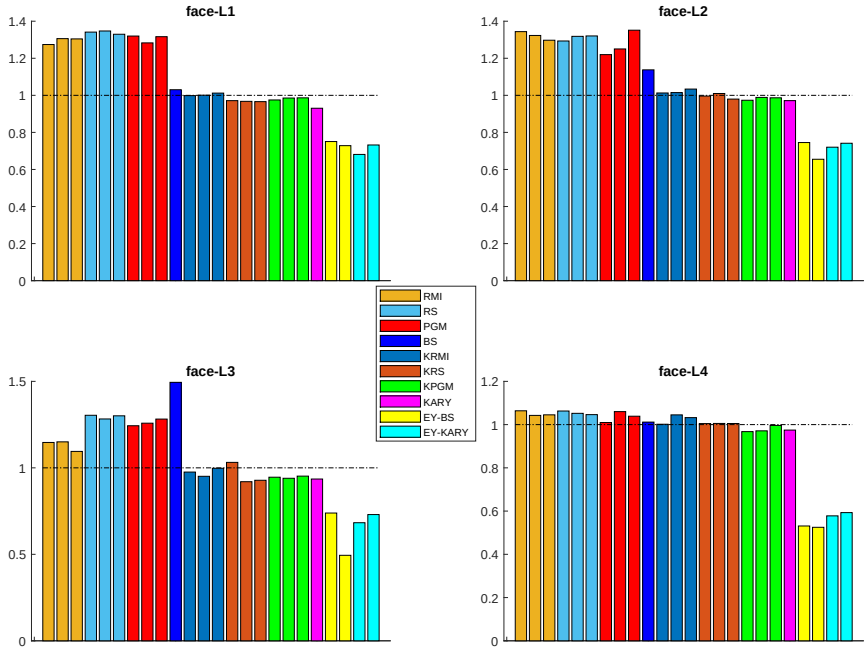


FIGURE C.3: **Branch-free vs Branchy on the face dataset**. The figure legend is as in Figure C.1.

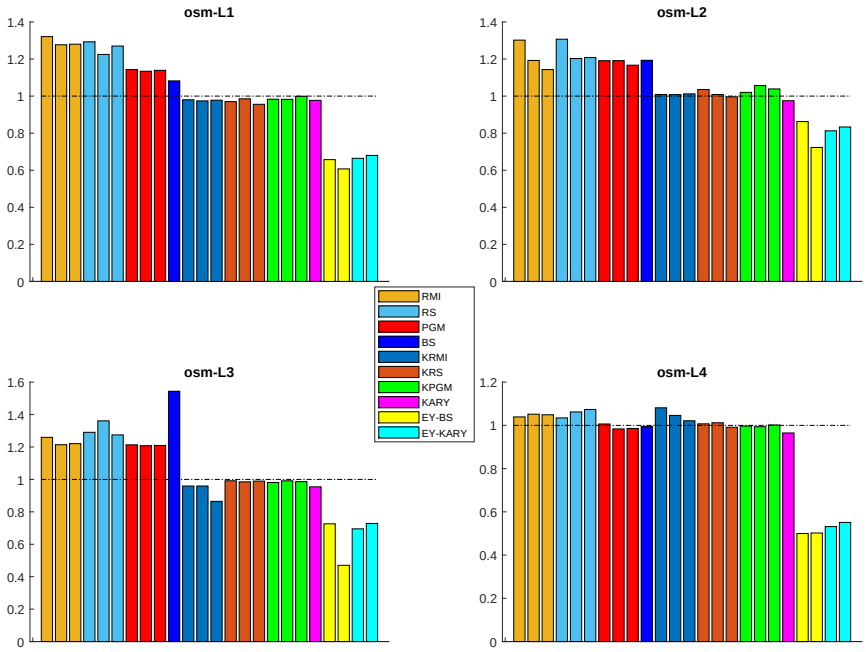


FIGURE C.4: **Branch-free vs Branchy on the osm dataset**. The figure legend is as in Figure C.1.

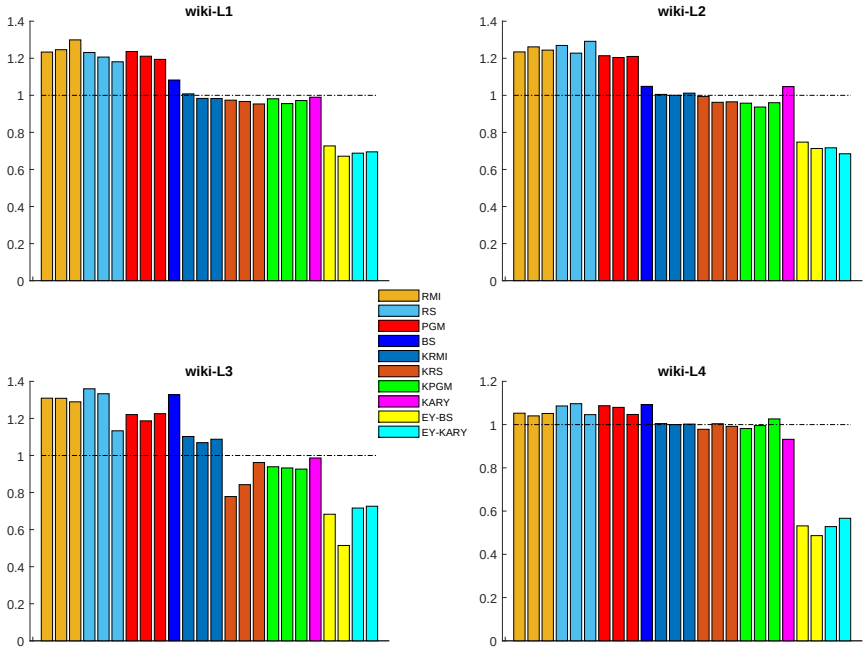


FIGURE C.5: **Branch-free vs Branchy on the wiki dataset**. The figure legend is as in Figure C.1.

	RMI		RS		PGM	
	BBS	**BFS**	**BBS**	**BFS**	**BBS**	**BFS**
L1	99.89%-7	99.89%-7	99.89%-7	96.35%-269	99.87%-9	99.87%-9
L2	99.98%-12	99.98%-12	99.98%-11	99.87%-81	99.98%-9	99.98%-9
L3	100.00%-5	100.00%-6	100.00%-11	100.00%-11	100.00%-9	100.00%-9
L4	99.84%-323209	99.84%-323211	100.00%-81	100.00%-191	100.00%-65	100.00%-34
	K-BBS	**K-BFS**	**K-BBS**	**K-BFS**	**K-BBS**	**K-BFS**
L1	99.89%-7	99.89%-7	98.92%-81	99.84%-11	99.87%-9	99.87%-9
L2	99.98%-12	99.98%-12	99.98%-11	99.98%-11	99.98%-9	99.98%-9
L3	100.00%-5	100.00%-6	100.00%-11	99.99%-81	100.00%-9	100.00%-17
L4	99.84%-323209	99.84%-323211	100.00%-191	100.00%-61	100.00%-129	100.00%-65

TABLE C.1: **Search Range for the amazon32 dataset.** The columns of table report the model classes. Each model class is divided into Branchy (**BBS**) and Branch-free (**BFS**) versions of Binary and *k*-ary Searches (in this latter case **K-BBS** and **K-BFS**). In each class, we consider the best performing models.The rows report the memory levels. Each memory level corresponds to a row in the table. For those rows, each entry contains the pair Reduction Factors in percentage - number of elements to search after a prediction is made.

	RMI		RS		PGM	
	BBS	**BFS**	**BBS**	**BFS**	**BBS**	**BFS**
L1	99.88%-4	99.84%-6	99.79%-7	99.68%-11	99.73%-9	99.73%-9
L2	99.97%-8	99.98%-5	99.96%-11	99.74%-81	99.97%-9	99.89%-33
L3	100.00%-14	100.00%-4	99.99%-81	100.00%-11	100.00%-9	100.00%-9
L4	100.00%-16	100.00%-22	100.00%-81	100.00%-81	100.00%-65	100.00%-34
	K-BBS	**K-BFS**	**K-BBS**	**K-BFS**	**K-BBS**	**K-BFS**
L1	99.88%-4	99.84%-6	99.79%-7	99.68%-11	99.73%-9	99.73%-9
L2	99.97%-8	99.98%-5	99.96%-11	99.96%-11	99.97%-9	99.97%-9
L3	100.00%-14	100.00%-4	99.96%-271	99.99%-81	100.00%-9	100.00%-9
L4	100.00%-16	100.00%-22	100.00%-81	100.00%-81	100.00%-129	100.00%-129

TABLE C.2: **Search Range for the amazon64 dataset.** The table legend is as in Table C.1.

	RMI		RS		PGM	
	BBS	**BFS**	**BBS**	**BFS**	**BBS**	**BFS**
L1	99.85%-5	99.86%-5	99.84%-5	99.84%-5	99.52%-17	99.52%-17
L2	99.98%-6	99.98%-5	98.35%-519	99.55%-141	99.97%-9	99.94%-17
L3	100.00%-10	100.00%-10	99.98%-141	100.00%-5	99.98%-129	100.00%-9
L4	100.00%-50	100.00%-50	100.00%-521	100.00%-531	100.00%-33	100.00%-33
	K-BBS	**K-BFS**	**K-BBS**	**K-BFS**	**K-BBS**	**K-BFS**
L1	99.85%-5	99.86%-5	99.84%-5	99.84%-5	99.08%-33	99.08%-33
L2	99.98%-6	99.98%-5	99.98%-5	99.98%-5	99.97%-9	99.97%-9
L3	100.00%-10	100.00%-10	100.00%-5	100.00%-5	100.00%-9	100.00%-9
L4	100.00%-50	100.00%-50	100.00%-521	100.00%-521	100.00%-65	100.00%-33

TABLE C.3: **Search Range for the facebook dataset.** The table legend is as in Table C.1.

	RMI		RS		PGM	
	BBS	**BFS**	**BBS**	**BFS**	**BBS**	**BFS**
L1	99.88%-4	99.85%-5	99.73%-9	99.57%-15	98.27%-65	-0.34%-3751
L2	99.98%-6	99.99%-3	99.97%-9	98.97%-330	99.97%-9	99.97%-9
L3	99.99%-80	100.00%-30	100.00%-9	99.96%-331	100.00%-9	100.00%-9
L4	100.00%-883	100.00%-883	100.00%-52	100.00%-16	100.00%-10	100.00%-34
	K-BBS	**K-BFS**	**K-BBS**	**K-BFS**	**K-BBS**	**K-BFS**
L1	99.88%-4	99.85%-5	99.73%-9	99.73%-9	99.73%-9	99.73%-9
L2	99.98%-6	99.99%-3	99.97%-9	98.97%-330	99.97%-9	99.94%-17
L3	99.99%-80	100.00%-30	99.96%-331	99.91%-651	100.00%-9	100.00%-17
L4	100.00%-883	100.00%-883	100.00%-16	100.00%-52	100.00%-18	100.00%-18

TABLE C.4: **Search Range for the osm dataset.** The table legend is as in Table C.1.

	RMI		RS		PGM	
	BBS	**BFS**	**BBS**	**BFS**	**BBS**	**BFS**
L1	99.84%-5	99.86%-5	97.53%-91	97.53%-91	99.73%-9	45.48%-2033
L2	99.97%-9	99.95%-16	99.71%-91	99.71%-91	99.89%-33	99.89%-33
L3	100.00%-5	100.00%-9	99.99%-91	99.99%-91	100.00%-33	100.00%-9
L4	100.00%-30	100.00%-39	100.00%-91	100.00%-91	100.00%-33	100.00%-33
	K-BBS	**K-BFS**	**K-BBS**	**K-BFS**	**K-BBS**	**K-BFS**
L1	99.84%-5	99.86%-5	97.53%-91	97.53%-91	99.73%-9	99.73%-9
L2	99.97%-9	99.95%-16	99.71%-91	99.71%-91	99.94%-17	99.89%-33
L3	100.00%-5	100.00%-9	99.99%-91	99.99%-91	100.00%-9	100.00%-9
L4	100.00%-30	100.00%-39	100.00%-91	100.00%-91	100.00%-33	100.00%-65

TABLE C.5: **Search Range for the wiki dataset.** The table legend is as in Table C.1.

Appendix D

Generic Learned Static Sorted Sets Dictionaries: Supplementary Results

This Appendix summarizes the results discussed in Chapter 5.

D.1 Boosting

Figures D.1-D.5 report the full set of experiments described and discussed in Section 5.5.1.

D.2 Comparison with the State of the Art

Figures D.11-D.15 report the full set of experiments described and discussed in Section 5.5.2.

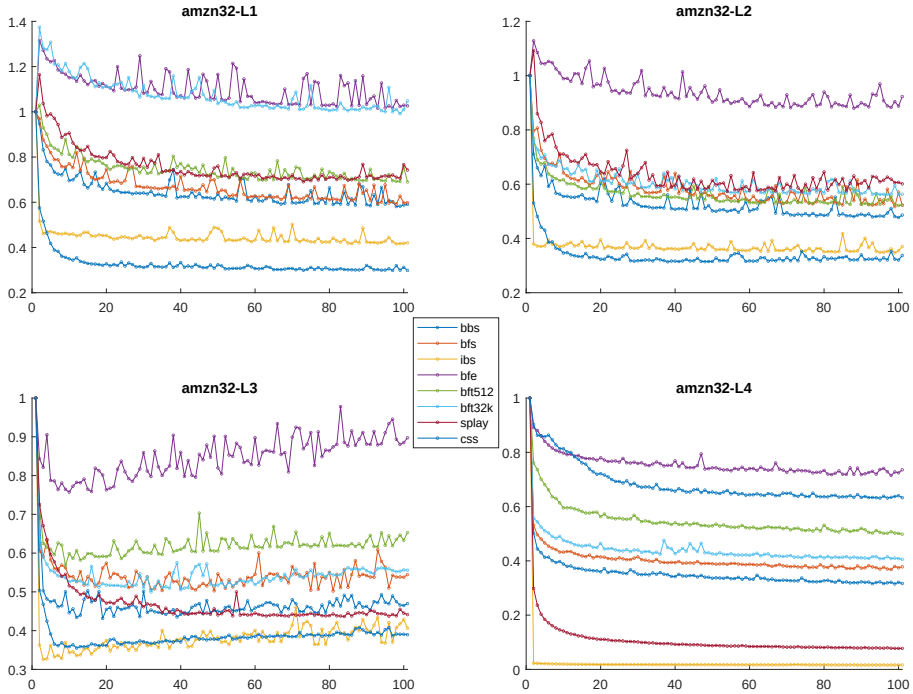


FIGURE D.1: **Binning boosting property on amzn32 dataset**. For each memory level, we report in the abscissa axis the number of bins in percentage with respect of the number of elements in the Table. In the ordinate, we indicate the ratio between the mean query time of Generic Learned Dictionaries and $\mathcal{SD}$ alone. For the sake of clarity, a ratio under one indicates that the Generic Learned Dictionary performs better than the simple ones.

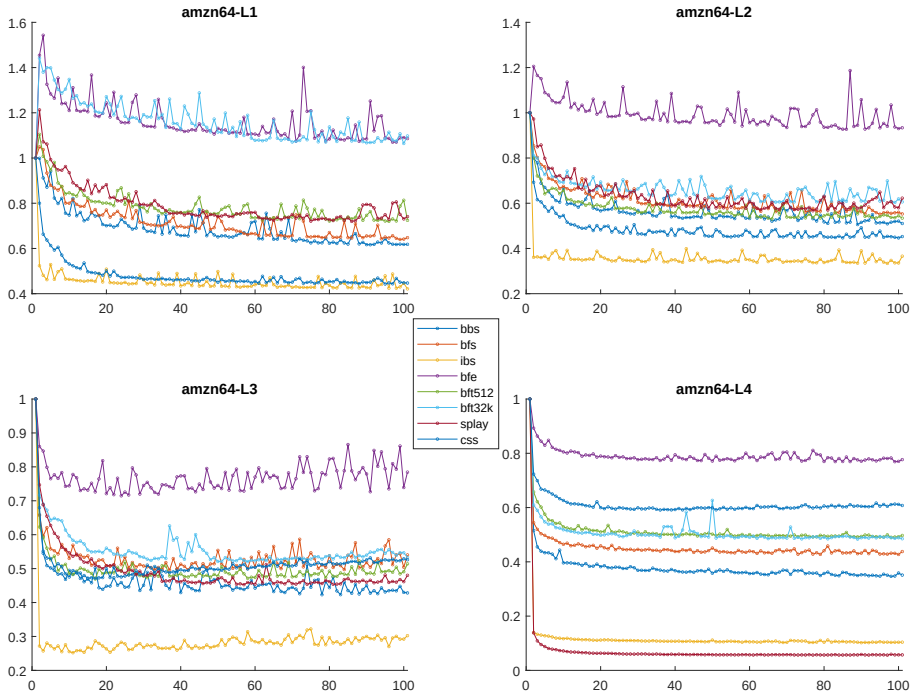


FIGURE D.2: **Binning boosting property on amzn64 dataset**. The legend is as in D.1.

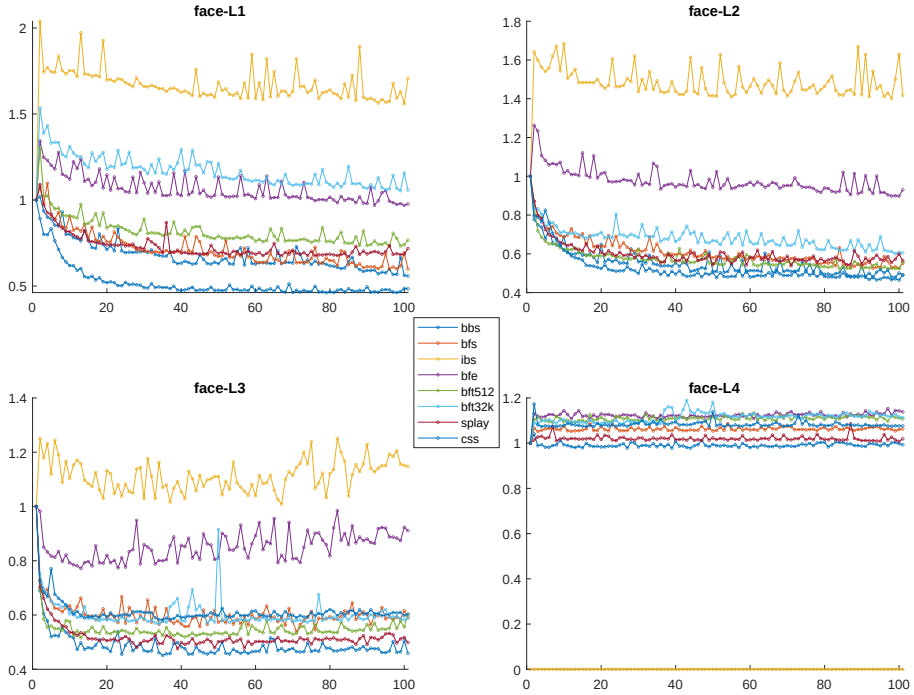


FIGURE D.3: **Binning boosting property on face dataset**. The legend is as in D.1.

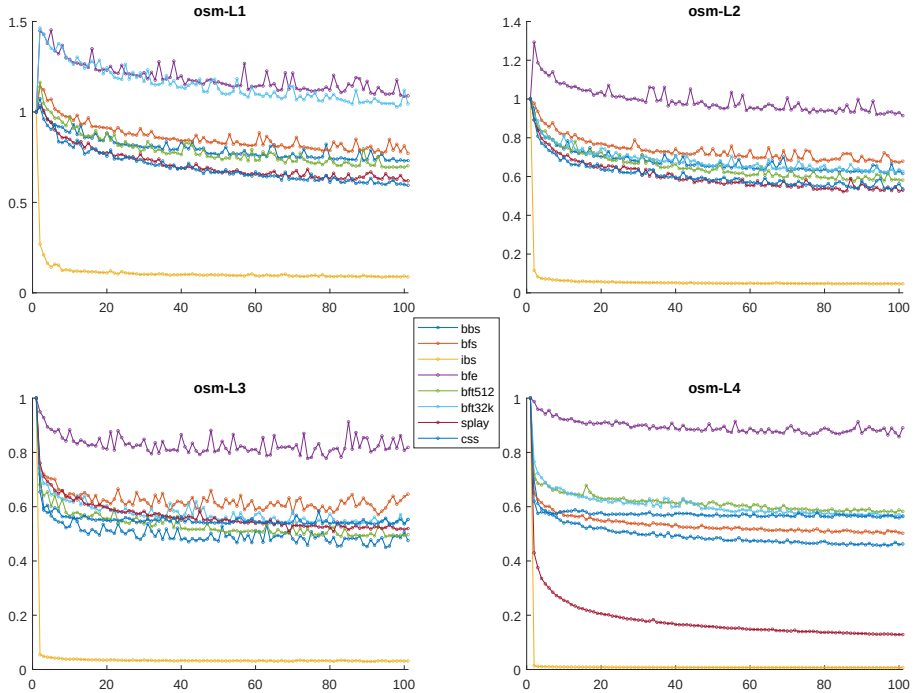


FIGURE D.4: **Binning boosting property on osm dataset**. The legend is as in D.1.

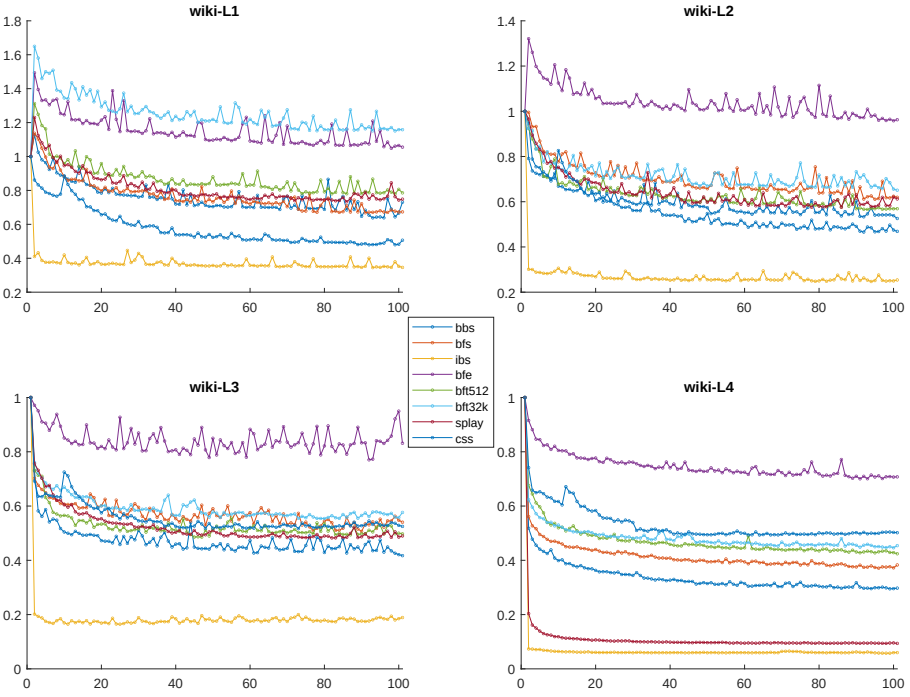


FIGURE D.5: **Binning boosting property on wiki dataset**. The legend is as in D.1.

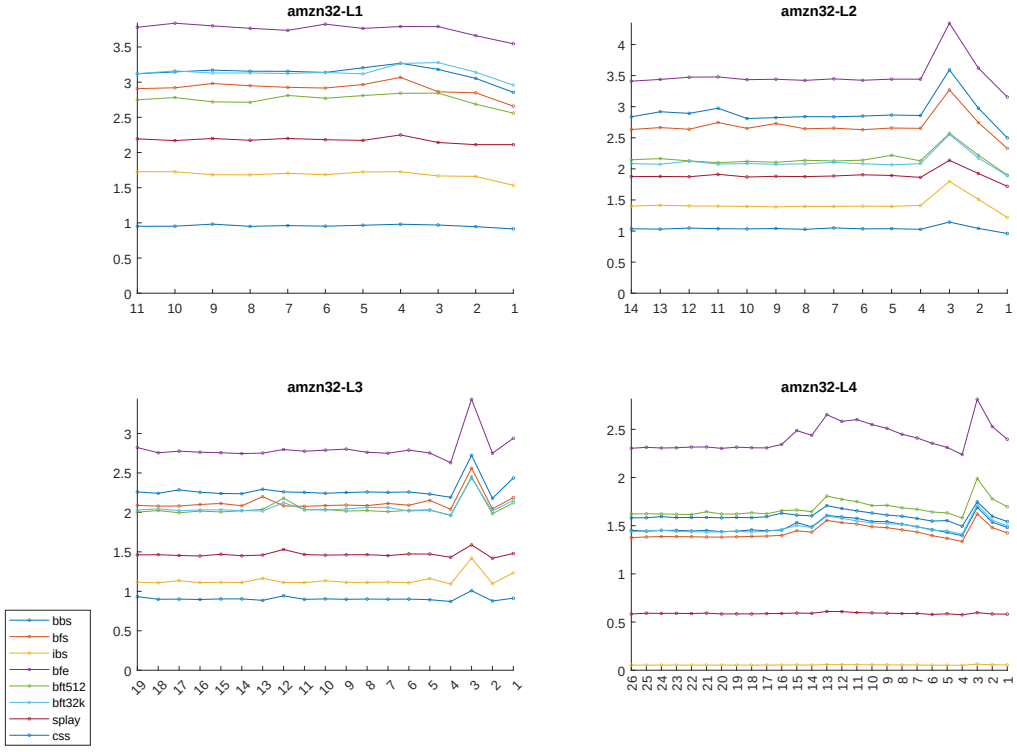


FIGURE D.6: **PGM boosting property on amzn32 dataset**. For each memory level, we report in the abscissa axis the choosen ϵ for the **PGM** construction. In the ordinate, we indicate the ratio between the mean query time of the **PGM** and the $\mathcal{SD}$ alone. For the sake of clarity, a ratio under one indicates that the **PGM** performs better than the simple ones.

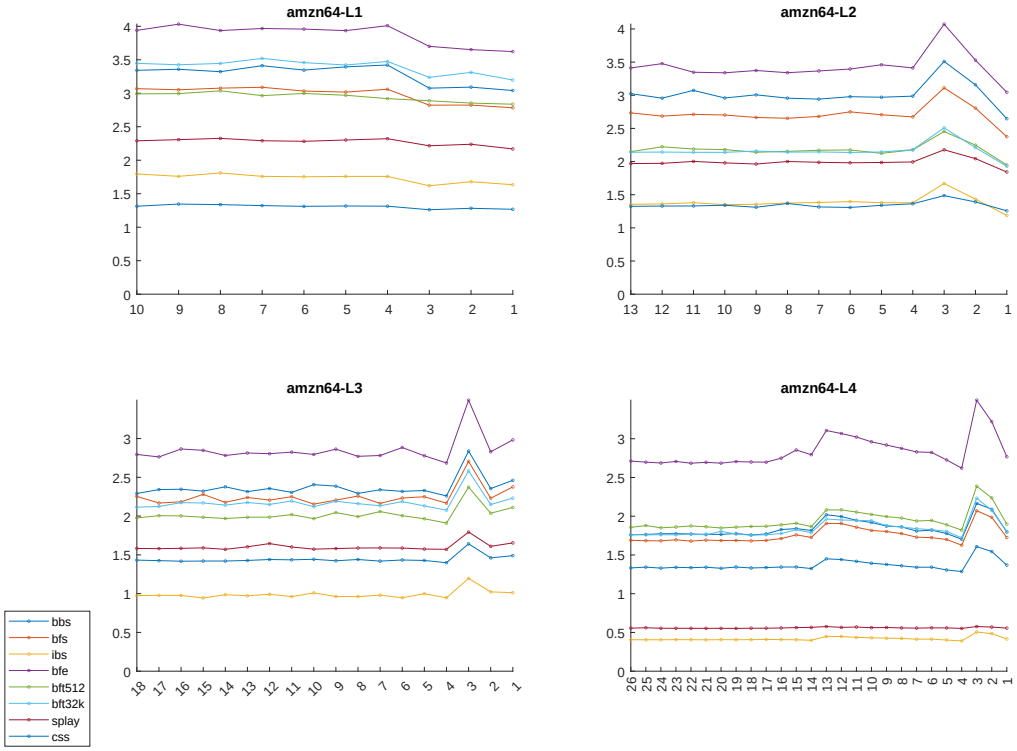


FIGURE D.7: **PGM boosting property on amzn64 dataset**. The legend is as in D.6.

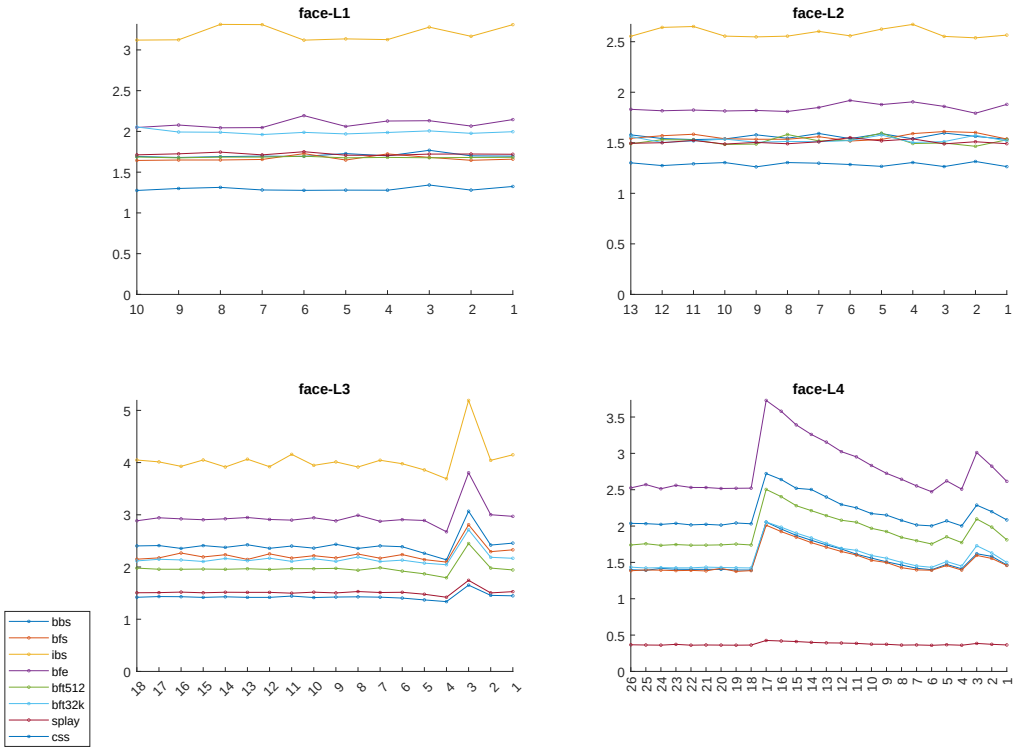


FIGURE D.8: **PGM boosting property on face dataset**. The legend is as in D.6.

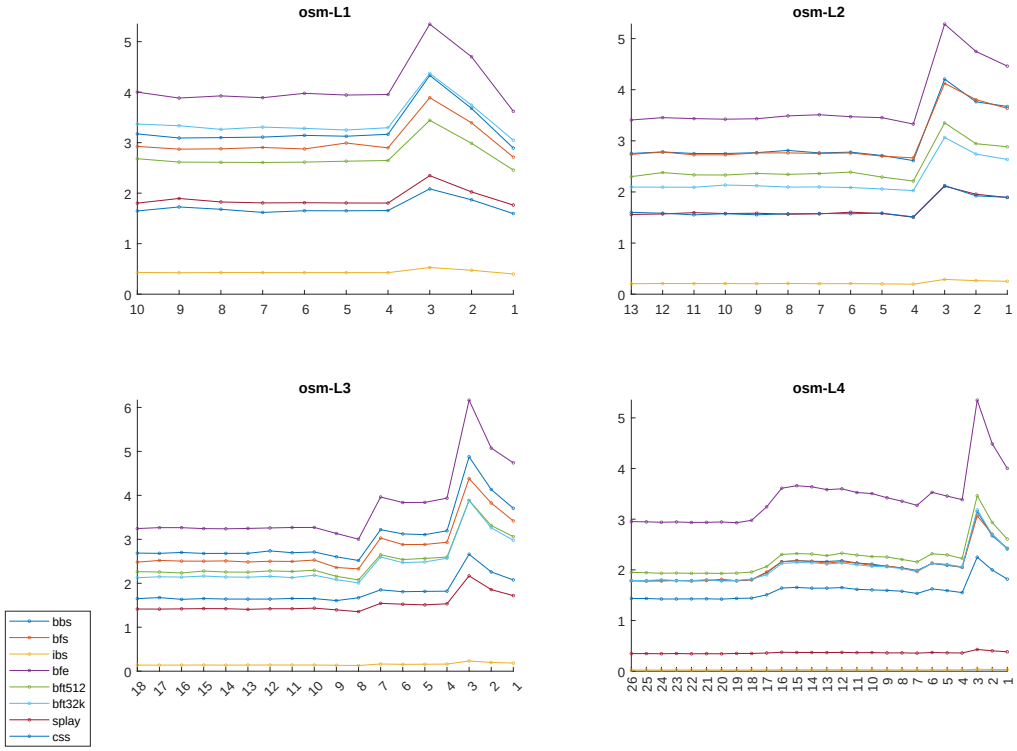


FIGURE D.9: **PGM boosting property on osm dataset**. The legend is as in D.6.

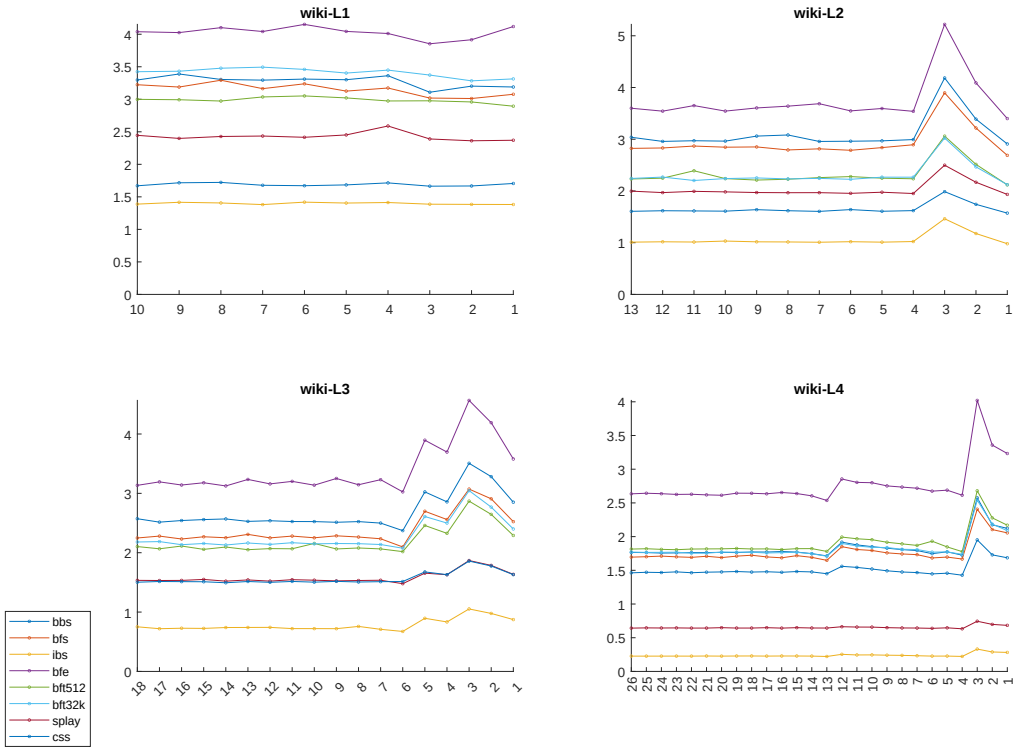


FIGURE D.10: **PGM boosting property on wiki dataset**. The legend is as in D.6.

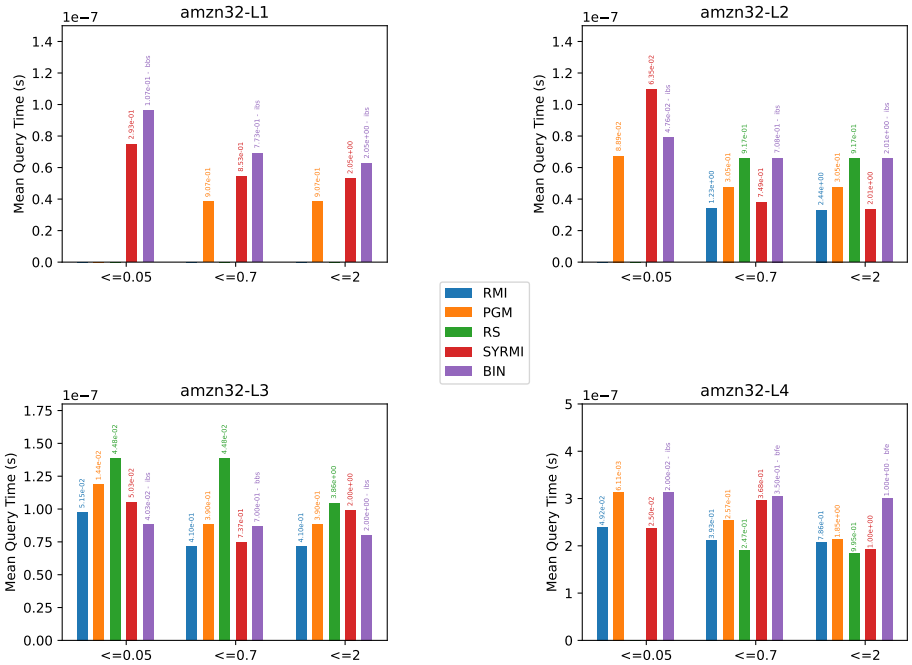


FIGURE D.11: **Learned Indexes Query Time With Bounds on Space on amzn32 dataset**. For each memory level, we choose three space bounds as in Chapter 3. For each space bound, from left to right, we report the mean query time of the best **RMI**, **PGM** and **RS** that satisfies the imposed bound. Next bar indicates the mean query time for the **SY-RMI** as in Chapter 3, The last bar is the mean query time for the best Generic Learned Dictionary with space inside the bound.

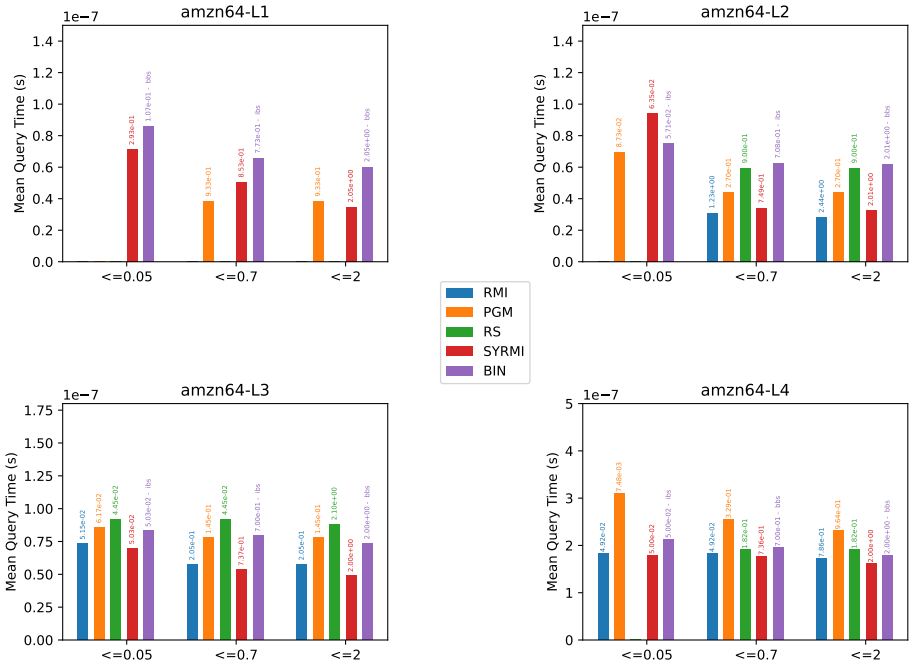


FIGURE D.12: **Learned Indexes Query Time With Bounds on Space on amzn64 dataset**. The legend is as in Figure D.11.

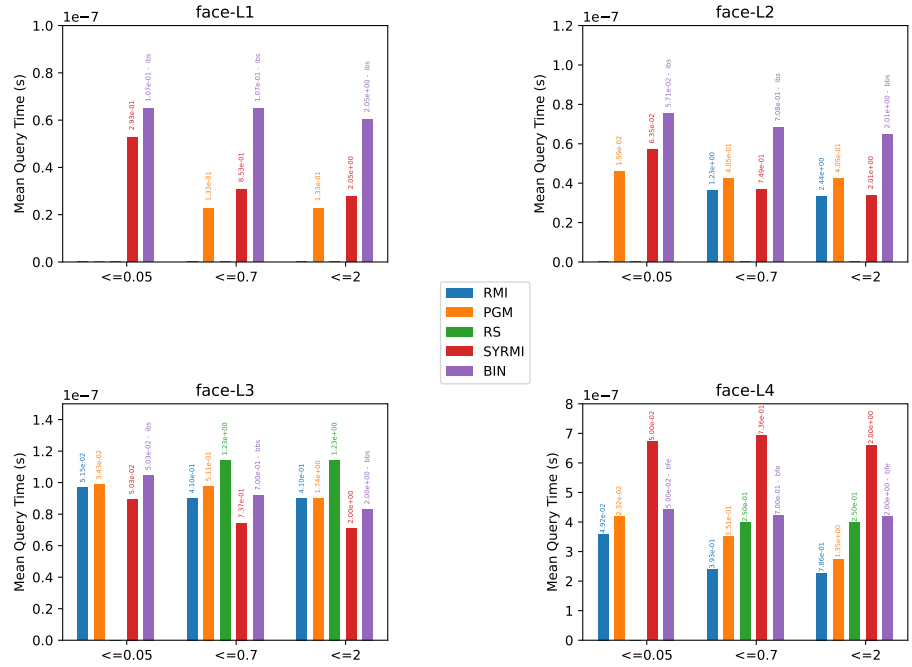


FIGURE D.13: **Learned Indexes Query Time With Bounds on Space on face dataset**. The legend is as in Figure D.11.

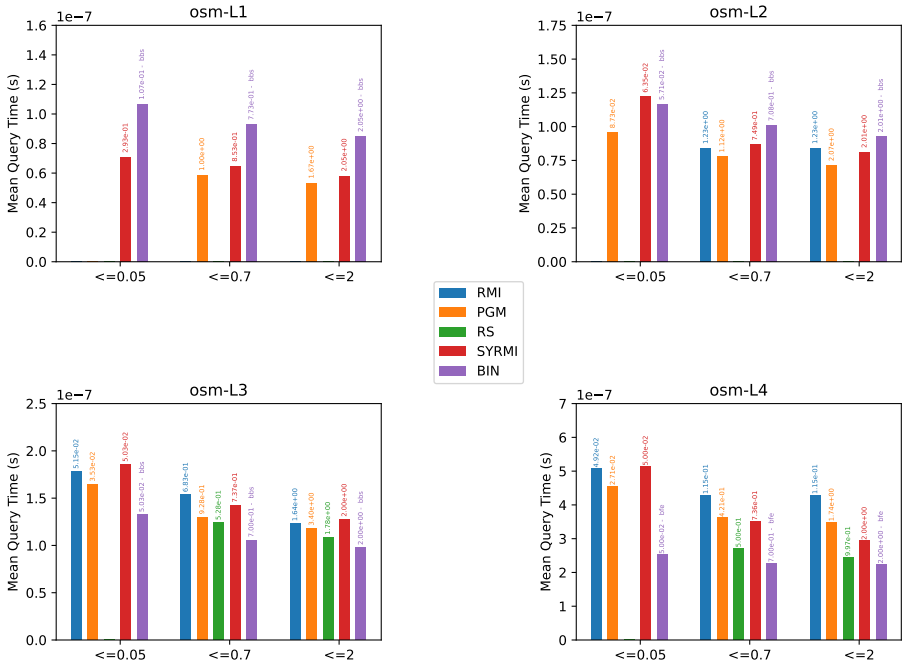


FIGURE D.14: **Learned Indexes Query Time With Bounds on Space on osm dataset**. The legend is as in Figure D.11.

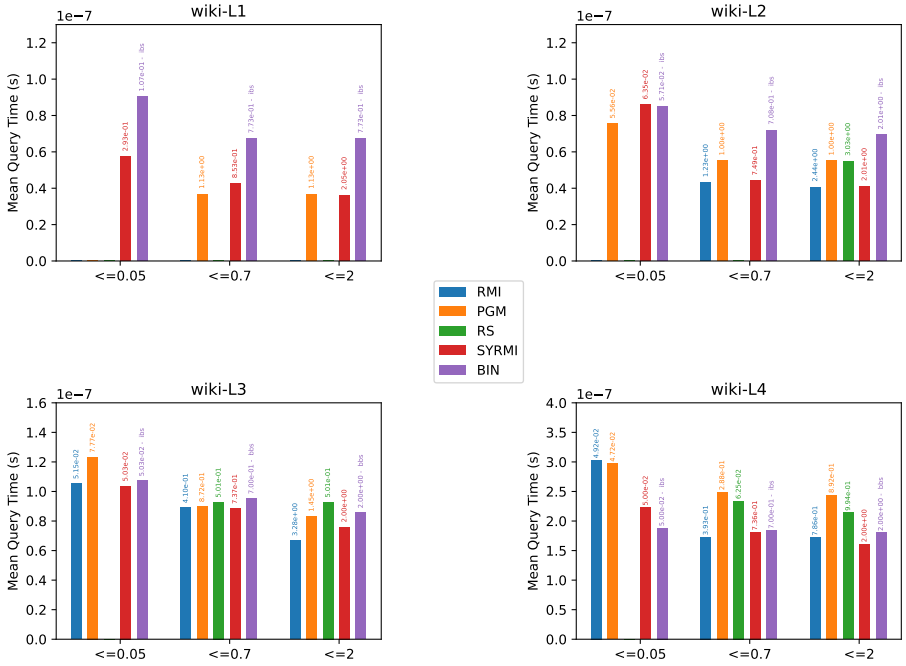


FIGURE D.15: **Learned Indexes Query Time With Bounds on Space on wiki dataset**. The legend is as in Figure D.11.

Bibliography

Abadi, Martín et al. (2015). *TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems*. http://download.tensorflow.org/paper/whitepaper2015.pdf.

Aho, Alfred V., John E. Hopcroft, and Jeffrey D. Ullman (1974). "The Design and Analysis of Computer Algorithms". In.

Ao, Naiyong et al. (May 2011). "Efficient Parallel Lists Intersection and index Compression Algorithms Using Graphics Processing Units". In: *Proc. VLDB Endow.* 4.8, pp. 470–481. ISSN: 2150-8097. DOI: 10.14778/2002974.2002975. URL: https://doi.org/10.14778/2002974.2002975.

Bayer, R. and E. McCreight (1970). "Organization and Maintenance of Large Ordered Indices". In: *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*. SIGFIDET '70. Houston, Texas: Association for Computing Machinery, pp. 107–141. ISBN: 9781450379410. DOI: 10.1145/1734663.1734671. URL: https://doi.org/10.1145/1734663.1734671.

Bloom, Burton H. (1970). "Space/Time Trade-Offs in Hash Coding with Allowable Errors". In: *Commun. ACM* 13, pp. 422–426.

Comer, Douglas (June 1979). "Ubiquitous B-Tree". In: *ACM Comput. Surv.* 11.2, pp. 121–137. ISSN: 0360-0300. DOI: 10.1145/356770.356776. URL: https://doi.org/10.1145/356770.356776.

Dai, Zhenwei and Anshumali Shrivastava (2020). "Adaptive Learned Bloom Filter (Ada-BF): Efficient Utilization of the Classifier with Application to Real-Time Information Filtering on the Web". In: *Advances in Neural Information Processing Systems*. Ed. by H. Larochelle et al. Vol. 33. Curran Associates, Inc., pp. 11700–11710.

Demaine, Erik D., Thouis Jones, and Mihai Pătraşcu (2004). "Interpolation Search for Non-Independent Data". In: *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA '04. New Orleans, Louisiana: Society for Industrial and Applied Mathematics, pp. 529–530. ISBN: 089871558X.

Ferragina, Paolo, Fabrizio Lillo, and Giorgio Vinciguerra (July 2020). "Why Are Learned Indexes So Effective?" In: *Proceedings of the 37th International Conference on Machine Learning*. Ed. by Hal Daumé III and Aarti Singh. Vol. 119. Proceedings of Machine Learning Research. PMLR, pp. 3123–3132. URL: http://proceedings.mlr.press/v119/ferragina20a.html.

Ferragina, Paolo and Giorgio Vinciguerra (2020a). "Learned Data Structures". In: *Recent Trends in Learning From Data*. Ed. by Luca Oneto et al. Springer International Publishing, pp. 5–41. ISBN: 978-3-030-43883-8. DOI: 10.1007/978-3-030-43883-8_2. URL: https://doi.org/10.1007/978-3-030-43883-8_2.

— (2020b). "The PGM-index: a Fully-Dynamic Compressed Learned Index with Provable Worst-case Bounds". In: *PVLDB* 13.8, pp. 1162–1175. ISSN: 2150-8097. DOI: 10.14778/3389133.3389135. URL: https://pgm.di.unipi.it.

Freedman, David (Aug. 2005). *Statistical Models : Theory and Practice*. Cambridge University Press. ISBN: 0521854830.

Freedman, David and Persi Diaconis (Dec. 1981). "On the Histogram as a Density Estimator:L2 Theory". In: *Zeitschrift für Wahrscheinlichkeitstheorie und Verwandte*

Gebiete 57.4, pp. 453–476. ISSN: 1432-2064. DOI: 10.1007/BF01025868. URL: https://doi.org/10.1007/BF01025868.

Fumagalli, Giacomo et al. (2022). "On the Choice of General Purpose Classifiers in Learned Bloom Filters: An Initial Analysis Within Basic Filters". In: *Proceedings of the 11th International Conference on Pattern Recognition Applications and Methods (ICPRAM)*, pp. 675–682.

Galakatos, Alex et al. (2019). "FITing-Tree: A Data-Aware Index Structure". In: *Proceedings of the 2019 International Conference on Management of Data*. SIGMOD '19. Amsterdam, Netherlands: Association for Computing Machinery, pp. 1189–1206. ISBN: 9781450356435. DOI: 10.1145/3299869.3319860. URL: https://doi.org/10.1145/3299869.3319860.

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville (2016a). "Deep Feedfoward Networks". In: *Deep Learning*. http://www.deeplearningbook.org. MIT Press, pp. 164–223.

— (2016b). "Example: Linear Regression". In: *Deep Learning*. MIT Press, pp. 105–108.

Khuong, Paul-Virak and Pat Morin (2017). "Array Layouts for Comparison-based Searching". In: *J. Exp. Algorithmics* 22, 1.3:1–1.3:39.

Kipf, Andreas et al. (2020). "RadixSpline: a Single-pass Learned Index". In: *Proceedings of the Third International Workshop on Exploiting Artificial Intelligence Techniques for Data Management, aiDM@SIGMOD 2020, Portland, Oregon, USA, June 19, 2020*, 5:1–5:5. DOI: 10.1145/3401071.3401659. URL: https://doi.org/10.1145/3401071.3401659.

Knuth, Donald E. (1973). "The Art of Computer Programming, Vol. 3 (Sorting and Searching)". In: *Addison-Wesley Publishing Company* 3, pp. 481–489.

Kraska, Tim et al. (2018). "The Case for Learned Index Structures". In: *Proceedings of the 2018 International Conference on Management of Data*. SIGMOD '18. Houston, TX, USA: Association for Computing Machinery, pp. 489–504. ISBN: 9781450347037. DOI: 10.1145/3183713.3196909. URL: https://doi.org/10.1145/3183713.3196909.

Marcus, Ryan, Andreas Kipf, et al. (2020). "Benchmarking Learned Indexes". In: *Proc. VLDB Endow.* 14.1, pp. 1–13.

Marcus, Ryan, Emily Zhang, and Tim Kraska (2020). "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, pp. 2789–2792. ISBN: 9781450367356. DOI: 10.1145/3318464.3384706. URL: https://doi.org/10.1145/3318464.3384706.

Mehlhorn, Kurt and Athanasios K. Tsakalidis (1990). "Algorithm for Finding Patterns in Strings". In: *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity*, pp. 255–300.

Mitzenmacher, Michael (2018). "A Model for Learned Bloom Filters, and Optimizing by Sandwiching". In: *Proceedings of the 32nd International Conference on Neural Information Processing Systems*. NIPS'18. Montréal, Canada: Curran Associates Inc., pp. 462–471.

Moore, Gordon E et al. (1965). *Cramming More Components Onto Integrated Circuits*.

Neumann, Thomas and Sebastian Michel (2008). "Smooth Interpolating Histograms with Error Guarantees". In: *Sharing Data, Information and Knowledge*. Ed. by Alex Gray, Keith Jeffery, and Jianhua Shao. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 126–138. ISBN: 978-3-540-70504-8.

Ohn, Ilsang and Yongdai Kim (2019). "Smooth Function Approximation by Deep Neural Networks with General Activation Functions". In: *Entropy* 21.7, p. 627.

Pătraşcu, Mihai (2008). "Predecessor Search". In: *Encyclopedia of Algorithms*. Ed. by Ming-Yang Kao. Boston, MA: Springer US, pp. 661–664. ISBN: 978-0-387-30162-4. DOI: 10.1007/978-0-387-30162-4_298. URL: https://doi.org/10.1007/978-0-387-30162-4_298.

Pătraşcu, Mihai and Mikkel Thorup (2006). "Time-Space Trade-Offs for Predecessor Search". In: *Proceedings of the Thirty-Eighth Annual ACM Symposium on Theory of Computing*. STOC '06. Seattle, WA, USA: Association for Computing Machinery, pp. 232–240. ISBN: 1595931341. DOI: 10.1145/1132516.1132551. URL: https://doi.org/10.1145/1132516.1132551.

Peterson, W. W. (1957). "Addressing for Random-Access Storage". In: *IBM Journal of Research and Development* 1.2, pp. 130–146. DOI: 10.1147/rd.12.0130.

Prokop, H. (July 1999). "Cache-oblivious Algorithms". MA thesis. Massachusetts Institute of Technology.

Rao, Jun and Kenneth A. Ross (1999). "Cache Conscious Indexing for Decision-Support in Main Memory". In: *Proceedings of the 25th International Conference on Very Large Data Bases*. VLDB '99. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., pp. 78–89. ISBN: 1558606157.

Sato, Kaz, Cliff Young, and David Patterson (2017). *An in-depth Look at Google's First Tensor Processing Unit (TPU)*. https://cloud.google.com/blog/products/ai-machine-learning/an-in-depth-look-at-googles-first-tensor-processing-unit-tpu.

Schlegel, Benjamin, Rainer Gemulla, and Wolfgang Lehner (2009). "K-ary Search on Modern Processors". In: *Proceedings of the Fifth International Workshop on Data Management on New Hardware*. DaMoN '09. Providence, Rhode Island: Association for Computing Machinery, pp. 52–60. ISBN: 9781605587011. DOI: 10.1145/1565694.1565705. URL: https://doi.org/10.1145/1565694.1565705.

Schulz, Lars-Christian, David Broneske, and Gunter Saake (2018). "An Eight- Dimensional Systematic Evaluation of Optimized Search Algorithms on Modern Processors". In: *Proc. VLDB Endow.* 11, pp. 1550–1562.

Sleator, Daniel Dominic and Robert Endre Tarjan (July 1985). "Self-Adjusting Binary Search Trees". In: *J. ACM* 32.3, pp. 652–686. ISSN: 0004-5411. DOI: 10.1145/3828.3835. URL: https://doi.org/10.1145/3828.3835.

Vaidya, Kapil et al. (2020). *Partitioned Learned Bloom Filter*. arXiv: 2006.03176 [cs.DS].

Van Sandt, Peter, Yannis Chronis, and Jignesh M. Patel (2019). "Efficiently Searching In-Memory Sorted Arrays: Revenge of the Interpolation Search?" In: *Proceedings of the 2019 International Conference on Management of Data*. SIGMOD '19. Amsterdam, Netherlands: Association for Computing Machinery, pp. 36–53. ISBN: 9781450356435. DOI: 10.1145/3299869.3300075. URL: https://doi.org/10.1145/3299869.3300075.

Wang, Brian (2017). *Moore Law is Dead but GPU will get 1000X faster by 2025*. https://www.nextbigfuture.com/2017/06/moore-law-is-dead-but-gpu-will-get-1000x-faster-by-2025.html.