The Scientific Journal of Cihan University – Sulaimaniya **PP: 186-209**
Volume (8), Issue (1), June 2024
ISSN 2520-7377 (Online), ISSN 2520-5102 (Print)

Learned-Indexes for Improving Query Performance A Comprehensive Survey with Taxonomy

Chiya Qadir Hama Faraj[1], Nzar A. Ali[2,3]

[1]Department of database Technology, Technical College of Infromatic, Sulaimani Polytechnic University, Sulaimani, Iraq
[2]Department of Statistics and Informatics, College of Administration and Economic, University of Sulaimani, Sulaimani, Iraq
[3]Department of Computer Science, Cihan University -Sulaimaniya, Sulaimaniya, Iraq

Email: chya.q.h@spu.edu.iq[1], nzar.ali@univsul.edu.iq[2], nzar.ali@sulicihan.edu.krd[3]

Abstract:

Database performance optimization involves intertwining developmental efforts with challenges. Core to this field are index structures, notably the B+-Tree technique, which enhances database performance by mapping keys to their locations regardless of data distribution. Although the B+-Tree improves query performance, it has inherent limitations affecting overall efficiency. The rise in data volume intensifies indexing complexities. Machine Learning (ML) emerges as a potent approach to rejuvenate legacy Database Management System (DBMS) components. A notable innovation is the "Learning Indexes" paradigm, viewing indexes as predictive models anticipating key locations in datasets, akin to Cumulative Distribution Functions (CDF). This study serves as a survey, exploring technologies underpinning learned-index paradigms and comparing them with traditional database indexing techniques. Through meticulous analysis, it unravels intricacies of both traditional and learned indexing paradigms, equipping aspiring analysts with a panoramic understanding. This underscores the imperative of charting a path for future advancements within this transformative domain.

Keywords: Index Terms, Learned-Indexes, Database Indexing, Query Performance, Complexity Analysis, Taxonomy.

الملخص:

يعد أداء و كفاءة قاعدة البيانات و ودقتها مجالًا أساسيًا، ويرتبط بشكل معقد بتحديات تطويرها. في جوهرها، تعمل هياكل الفهرس مثل (B+-Tree) القوية على تحسين أداء قاعدة البيانات بشكل كبير من خلال تعيين المفاتيح البيانات لمواقعها بغض النظر عن توزيع البيانات. بينما تعمل (B+-Tree) على تحسين أداء الاستعلام، إلا أنها تواجه قيودًا متأصلة تؤثر على كفاءة قاعدة البيانات. تصبح هذه القيود أكثر وضوحًا مع زيادة حجم البيانات ، مما يؤدي إلى تصاعد التعقيدات في الفهرسة. ولمواجهة هذه التحديات، تستفيد الأساليب الناشئة من التعلم الآلي (Machine Learning) لتنشيط أنظمة إدارة قواعد البيانات القديمة (DBMS). أحد الابتكارات الواعدة في هذا المجال هو ظهور "الفهارس المتعلمة"، وهو مجال بحثي متنامٍ وابتكار ثوري في بنية هياكل الفهرس. ويرى هذا المنظور الجديد أن "الفهارس هي نماذج"، يمكنها التنبؤ بذكاء مواقع المفاتيح باستخدام وظائف التوزيع التراكمي (CDF). يؤدي هذا إلى إعادة تصور الفهارس ككيانات قابلة للتدريب، مما قد يؤدي إلى تحسين كفاءة الاستعلام، خاصة بالنسبة لمجموعات البيانات الكبيرة. في حين أن الفهارس المتعلمة واعدة، فإن التحقيقات الشاملة في هذا المجال الناشئ محدودة. لذا هذه الدراسة بمثابة مراجعة شاملة ودراسة استقصائية شاملة مع تحليلٍ شامل وعميق، واستكشاف لأحدث تقنيات الفهارس المتعلمة، ومقارنتها بهياكل

DOI: http://dx.doi.org/10.25098/8.1.30

The Scientific Journal of Cihan University – Sulaimaniya **PP: 186-209**
Volume (8), Issue (1), June 2024
ISSN 2520-7377 (Online), ISSN 2520-5102 (Print)

الفهرسة التقليدية الشائعة. ومن خلال التحليل الدقيق، يكشف عن تعقيدات نماذج الفهارس المتعلمة والفهرسة التقليدية، مما يزود المحللين الطموحين بفهم بانورامي. وهذا يؤكد ضرورة رسم مسار للتقدم المستقبلي في هذا المجال التحويلي.

الكلمات المفتاحية: مصطلحات الفهرس، الفهارس المستفادة، فهرسة قاعدة البيانات، أداء الاستعلام، تحليل التعقيد، التصنيف.

پوختە:

کارایی و چووستیی داتابەیس بوارێکی بنەڕەتییە، بە شێوەیەکی ئاڵۆز پەیوەندیدارە بە ئاڵنگاریەکانی گەشەپێدان و پێشخستنیەوە. لە ناوەڕۆکی گەشەسەندنەکەیدا، پێکهاتەکانی فەرهەنگۆك (ئیندێکس) وەکو (B+-Tree) کە بە شێوەیەکی بەرچاو چووستیی داتابەیس بەرز دەکەنەوە بە نەخشەکێشانی کلیلی تۆمارەکان بۆ شوێنەکانیان بەبێ گوێدانە دابەشبوونی داتای کلیلەکان. لە کاتێکدا کە B+-Tree چووستیی پرسیار (کویری) باشتر دەکات، ڕووبەڕووی سنووردارببوونی سروشتی دەبێتەوە کە کاریگەری لەسەر کارایی داتابەیس هەیە. ئەم سنووردابوونانە زیاتر دەردەکەون لەگەڵ زیادبوونی قەبارەی داتاکان، کە دەبێتە هۆی زیادبوونی ئاڵۆزییەکان لە ئیندێکس کردندا. بۆ چارەسەرکردنی ئەم ئاڵنگاریانە، ڕێبازە نوێیە سەرهەڵداوەکان، فێربوونی ئامێر (Machine Learning) بەکارهێنن بۆ زیندووکردنەوەی سیستەمی بەڕێوەبردنی داتابەیس (DBMS). داهێنانێکی ئومێدبەخش لەم بوارەدا سەرهەڵدانی "ئیندێکسی فێربوو"ە (Learned Index)، کە بواریکی توێژینەوەی گەشەسەندووە ونوێگەریەکی شۆرش ئامێزە لە پێکهاتەی ئیندێکسەکاندا. ئەم دیدگا نوێیە پێی وایە کە "ئیندێکسەکان مۆدێلن"، کە دەتوانن بە شێوەیەکی زیرەك پێشبینی شوێنی کلیلەکان بکەن بە بەکارهێنانی نەخشەکانی دابەشبوونی کەڵەکەبوو (CDF). لەم گۆشەنیگایەوە ئیندێکسەکان وەك یەکەی ڕاهێنراو، بە ئەگەرێکی زۆرەوە کارایی کویرییەکان بەرز دەکەنەوە، بە تایبەت لەسەرکۆمەڵە داتا گەورەکان. لە کاتێکدا کە ئیندێکسە فێربووەکان ئومێدبەخشن، لێکۆڵینەوە گشتگیرەکان لەم ئاڕاستە نوێگەریەدا سنووردارە. هەر لەم سۆنگەیەوە، ئەم توێژینەوەیە پێداچوونەوە و لێکۆڵینەوەیەکی تەواو گشتگیرە شانبەشانی شیکاری قوڵ لەسەر نوێترین تەکنیکە پێشنیارکراوەکانی ئیندێکسە فێربووەکان و بەراوردکردنیان لەگەڵ پێکهاتە باوە دێرینەکانی ئیندێکسکردن. لە ڕێگەی شیکاری وردەوە، ئاڵۆزییەکانی مۆدێلی ئیندێکسە فێربووەکان و ئیندێکسە دێرینەکان ئاشکرا دەکات، کە تێگەیشتنێکی پانۆرامایی دابین دەکات بۆ شرۆڤەکار و شیکارە ئومێدخوازەکان. ئەمەش جەخت لەسەر پێویستی داڕشتنی ڕێگایەك بۆ پێشکەوتنی داهاتوو لەم بوارە گۆڕانکاریەدا دەکاتەوە.

کلیلە وشە: زاراوەی ئیندێکس، فێربوونی ئیندێکس، ئیندێکسکردنی بنکەدراوە، ئەدای پرسیار، شیکاری ئاڵۆزییەکان، تاکسۆنۆمی.

DOI: http://dx.doi.org/10.25098/8.1.30

The Scientific Journal of Cihan University – Sulaimaniya **PP: 186-209**
Volume (8), Issue (1), June 2024
ISSN 2520-7377 (Online), ISSN 2520-5102 (Print)

INTRODUCTION

Modern databases are ubiquitous, impacting every facet of our lives. Data is considered a valuable organizational asset, with Database Management Systems (DBMS) storing, retrieving, and processing data to inform decisions efficiently and conveniently [1-3]. A key challenge in databases is performance, primarily tied to query processing. Enhancing query execution directly betters database performance. Indexes, supplementary structures linked to data files, enable efficient access methods. [1-5]. Indexes remain a potent technique significantly boosting query performance. Indexing links keys to related data record locations, associating each key with a reference to a full record in the database file. [1, 2, 5, 6]. The B+-tree and Hash-table are common index files, serving as models that map keys to records ***regardless of key distribution***. Although indexes reduce query response time, they have drawbacks impacting database performance. Additional space required for index files poses a size challenge, and creating multiple indexes for frequently used fields can burden the query optimizer. [7, 8]. Amidst escalating data volumes and diversity, indexing challenges grow. Recent research delves into machine learning to enhance legacy DBMS components, such as ***Learning Indexes***, a novel approach for query performance improvement. Learning Indexes deploy machine learning models to predict record positions for specific keys, leveraging data distribution. This approach considers indexes as models mapping keys to records, with potential for model upgrades [6, 9-11]. Being a new trend, there's limited high-level guidance available. This survey explores the popular state-of-the-art in Learned Indexes, encompassing principles, structure, procedures, database indexing, and traditional methods. This work aids researchers entering this field and offers a taxonomy for reviewed Learned Index Techniques. The paper's structure is as follows: Section two presents the Problem Description, followed by the literature review in section three. Section four offers the background review. Section five is divided into two parts: the first part includes critical discussion and illustrative explanation, while the second part presents theoretical results through summary tables and a taxonomy. The paper concludes with section six, which contains the conclusion and future work.

1. PROBLEM DESCRIPTION (SURVEY QUESTION

In nascent trends, initial resources like surveys or taxonomies might be lacking. Similarly, the emerging area of Learned Indexes faces a scarcity of comprehensive guidance. This study aims to illuminate the core concepts of Learned Indexes, presenting state-of-the-art advancements with crucial details. Additionally, a foundational taxonomy for these techniques is established..

2. REVIEWED LITERATURES

Learning-based structures, such as the learnt B-tree, are being investigated by the Database and Machine Learning communities, with the goal of upgrading conventional indexes with learning-based models to achieve higher time and space efficiency than previously known methods.

According to (Kraska et al., 2018)[6], indexes function as models, with the B^+-tree index resembling the (CDF) cumulative distribution function. They introduce the Recursive Model Index (RMI), which employs a learning model to predict page IDs in an in-memory framework. They further apply this to the Learned Hash-Map (Hash-Model Index), using hash functions to uniformly distribute keys in hash buckets, reducing conflicts [6]. Additional to the previous idea, they also proposed a Learned Bloom Filter that consists of a Neural

DOI: http://dx.doi.org/10.25098/8.1.30

The Scientific Journal of Cihan University – Sulaimaniya **PP: 186-209**
Volume (8), Issue (1), June 2024
ISSN 2520-7377 (Online), ISSN 2520-5102 (Print)

Network as initial-filter (pre-filter [12]) comes before an ultimately small Bloom Filter as a secondary-filter (backup-filter [12]). that also learned by observing the Query Distribution History for differentiation among the key and non-key [6].

Following (Michael Mitzenmacher, 2018)[12] 's analytical explanation of the proposed learned Bloom Filter in [6] he enhanced it further. In (Mitzenmacher, 2019)[13] he introduced the Optimized Learning Bloom Filter (Sandwiched Learned Bloom Filter) This approach employs an extra bloom filter before the learned prefilter, passing only positive queries to the learned prefilter, followed by the backup bloom filte [13].

A learnt index (Doraemon) was presented by (C. Tang, Z. Dong, M. Wang, Z. Wang, and H. J. a. p. a. Chen, 2019)[14] for dynamic workloads as a solution to the shifting data distribution issue that leads to model retraining. By utilizing the prior model structure for access patterns and data distribution that are comparable [14].

The learning index Fitting-tree, detailed in (Galakatos et al., 2019)[15], offers robust error boundaries, predictable efficiency, and two data insertion methods. Fitting-tree employs in-place insertion with additional space (ε) to avoid page errors. For large segments, insertion cost might be notable. The delta insertion technique maintains a fixed-size buffer where ordered keys are inserted. Upon buffer limit, segments split and merge [15].

The Alex-index proposed in (J. Ding et al., 2020)[16], similarly reserves space for inserted keys like Fitting-tree. However, in Alex-index, reserved space is distributed, directly placing keys in the predicted location. If occupied, gaps are added (gapped array) or the array grows (packed memory array). This design flexibility aids in balancing space and efficiency trade-offs [16].

Addressing the fully-dynamic indexable dictionary problem, (P. Ferragina and G. J. P. o. t. V. E. Vinciguerra, 2020)[17] introduced the PGM-index. This learning structure employs a bottom-up approach to recursively generate a multi-level index model. Three PGM-index versions are presented: one with ad-hoc compression for space efficiency, one adaptable to query distribution, and one optimizing itself within specified space or query time constraints [17].

(A. Kipf et al., 2020)[18] proposed the RadixSpline (RS); a learning index that could be constructed in a single-pass through a sorted data, whereas with a fixed amount of effort per additional element, unlike the prior techniques. In terms of both size and search efficiency, they have competed with the latest learning index models such as RMI from [6] Notably, RS has primarily two parameters, as highlighted in their evaluation [18].

DOI: http://dx.doi.org/10.25098/8.1.30

The Scientific Journal of Cihan University – Sulaimaniya **PP: 186-209**
Volume (8), Issue (1), June 2024
ISSN 2520-7377 (Online), ISSN 2520-5102 (Print)

3. BACKGROUND OVERVIEW

Databases hold data as records in files, often on secondary storage like hard disks for long-term storage. Transient data, parts of persistent data, are frequently accessed and processed in primary storage (e.g., main memory) during program runtime, with a short lifespan. [3]. Indexes are secondary access structures that aid in quick retrieval of field-based records from larger file records. Similar to a textbook's index aiding content search, a database's index fulfills a comparable role. Notably smaller than the book, the index reduces effort required [1]. Indexes typically store key values and a few attributes, using significantly less memory than the entire file. This enables creation of an index that loads into main memory, boosting processing efficiency for large disk-stored files [2].

3.1. MOST POPULAR TRADITIONAL INDEX

There are deferent kinds of index such as range index structure (e.g., B^+-Tree Index), point-index (e.g., Hash-map), and record existent indicators (e.g., BitMap-Index, Bloom filter) [6]. Below is the most two popular and common traditional index structures (B^+-Tree Index and Hash-map):

3.1.1. B+-TREE INDEX

The widely used B+-Tree index enhances query processing [16, 19]. It's a balanced-height lookup tree, directing record lookups based on field values [2, 3]. B^+-Tree maintains a balanced shape, with internal nodes having $\lceil n \ / \ 2 \rceil$ to n children, where n signifies the tree order. While the root holds 2 to n children [1, 2, 16], Leaf nodes store real data pointers and form a doubly linked list for random and sequential access [16, 19]. The B^+-Tree s a generic structure without assumptions about key distribution [19, 20].

3.1.2. BITMAP INDEX

Bitmap indexing is suitable for relations with a considerable number of records, especially for columns with limited unique values. In a bitmap index, each record is assigned an ID from 0 to n, which maps to a physical address including block numbers and offsets [3]. A bitmap index employs bit arrays, with an attribute-based index using a bitmap for each attribute value. These bitmaps have the same number of bits as the relation's record count. Initially set to 0, if record i of attribute A has value vj, the bitmap for vj sets the corresponding bit to 1. Bitmap indexes excel in selections, particularly involving multiple key selections [1]. To verify a selection, intersect bitmaps of key values, creating a new bitmap with a bit set to 1 if the corresponding bits in input bitmaps are all 1s, else it's 0. [1, 3].

3.1.3. BLOOM FILTER

A Bloom filter is a compact structure to check set membership. Using an m-bit array and k hashing functions, keys are mapped to array positions. Initially, all bits are 0. Adding a member sets bits at hash function addresses. To verify membership, a key's hash functions return positions. If any bit is 0, the key is not present. While Bloom filters prevent false negatives, they can lead to false positives. A strong hash function for point indexes has fewer collisions, while for a Bloom filter, it has more key and non-key collisions, and fewer key-non-key collisions [1, 6].

DOI: http://dx.doi.org/10.25098/8.1.30

The Scientific Journal of Cihan University – Sulaimaniya **PP: 186-209**
Volume (8), Issue (1), June 2024
ISSN 2520-7377 (Online), ISSN 2520-5102 (Print)

3.1.4. HASH INDEX (HASH MAP)

A Hash index consists of an array of pointers, or hash buckets, storing addresses pointing to linked lists of keys [21]. Hash indexes are crucial for point lookups in DBMS. They map keys to array positions using hash functions. Efficient implementation aims to avoid many keys mapping to the same location, termed collisions [6].

3.2. LEARNING BASE INDEX

Traditional indexes lack data distribution use. Learning-based indexes employ machine learning, diverging from B^+-Tree and Hash indexes. They aim for accurate data representation, enhancing efficient indexing [10]. Indexes are functions mapping keys to values for range, Hash, or Bitmap indexing [14].

3.2.1. MAIN CONCEPTS

3.2.1.1. UTILIZING MACHINE LEARNING TECHNIQUES

Machine Learning (ML) enables computers to learn without programming, encompassing AI, neuroscience, and more. It simplifies problem-solving by creating models from data [22, 23]. ML methods, including neural networks, construct the Learning Based Index by approximating CDF [6]. Figure 1 Demonstrates the Learned indexes compared to B-Tree index.

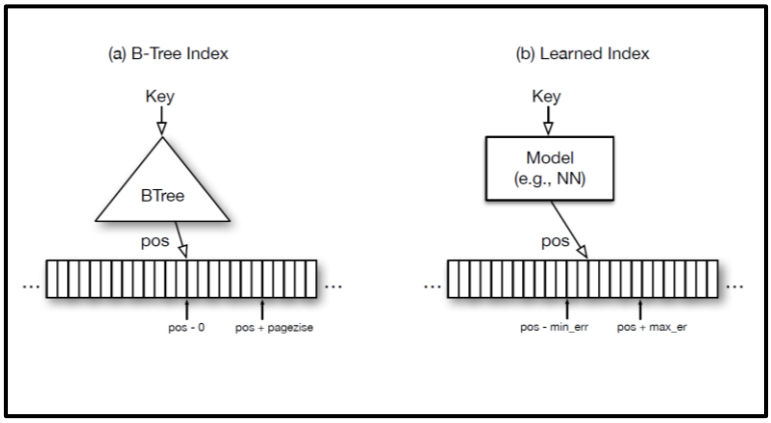

Figure 1: *(a) B-Trees indexes, (b) Learned indexes [6].*

Approximating the (CDF) Cumulative Distribution Function

The CDF approximation maps keys to positions in an array, resembling the standard statistical CDF. The traditional CDF, found in probability and statistics, represents the likelihood of values being less than a given key [10, 24, 25]. It uniquely characterizes probability distributions on real numbers[25, 26]. The CDF for some sample data from [10] is shown in Figure 2.

DOI: http://dx.doi.org/10.25098/8.1.30

The Scientific Journal of Cihan University – Sulaimaniya **PP: 186-209**
Volume (8), Issue (1), June 2024
ISSN 2520-7377 (Online), ISSN 2520-5102 (Print)

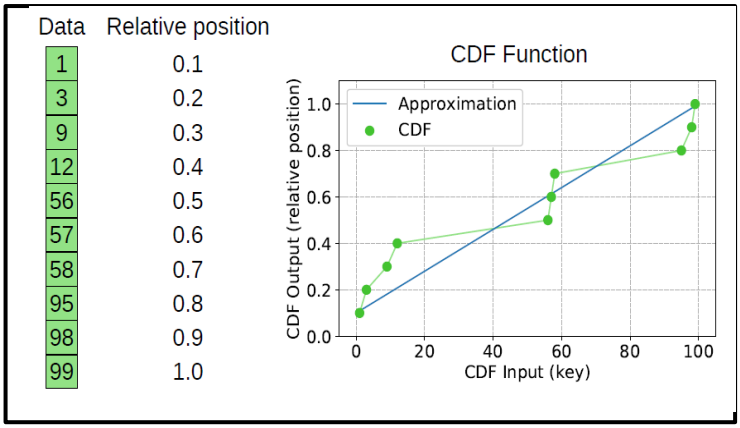

Figure 2: *The cumulative distribution function (CDF) view of a sorted array [10].*

3.3. LEARNING BASE INDEX STATE OF THE ART

Learned indexes, view indexes as distributions mapping keys to locations, approximated using ML models. They can be categorized into two groups: Fixed (Static) Learned Indexes and Dynamic Learned Indexes. Static indexes only support read operations, which limits their usability, while Dynamic indexes enable read-write operations and query pattern adaptation. However, Dynamic Learned Indexes face challenges, including retraining models for changing data distributions and the associated costs. This hinders their practicality in real-world dynamic workloads [14, 16, 17]. Therefore, as an attempt to overcome these defects, some of the following studies propose techniques within the framework of a Dynamic Learned Index.

3.3.1. RECURSIVE MODEL INDEX (RMI)

One key finding from [6] is that the complexity of the CDF requires a hierarchical approach for accuracy. The Recursive Model Index (RMI) is introduced, consisting of stages of regression models. When a query arrives, it's processed through each stage to estimate the key's position. At stage l, M_l models exist, with each stage's model trained iteratively with loss L_1 such that f_0 is initialized [6]. The RMI differs from traditional tree indexes: 1- It forms a Direct Acyclic Graph (DAG) structure. 2- Uneven coverage of records by models is allowed. 3- Stage predictions aren't position estimates, but expert selections. 4- Max-error is unpredictable. 5- Its size is fixed by total variables in models and final stage errors. 6- No inter-phase searches occur. RMI lacks data update support and is limited for secondary indexes [6]. See Figure 3 for visual reference

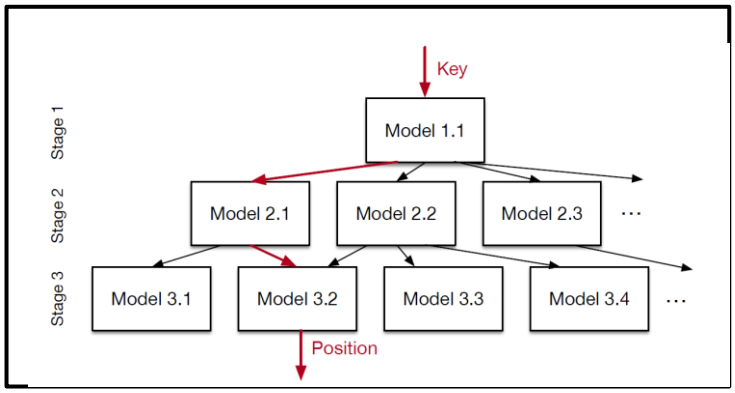

Figure 3: *Recursive Model Index (RMI) [6].*

DOI: http://dx.doi.org/10.25098/8.1.30

The Scientific Journal of Cihan University – Sulaimaniya **PP: 186-209**
Volume (8), Issue (1), June 2024
ISSN 2520-7377 (Online), ISSN 2520-5102 (Print)

3.3.2. HYBRID INDEXES

The Hybrid Index, another static learned index introduced in [6], builds on RMI's advantage of using different model structures across stages. Smaller neural nets (e.g., ReLU1) are suitable for complex data distributions in upper stages, while basic linear regression models are efficient for bottom stages. Extremely complex data might even resort to standard B-Trees at the lowest stage. See Figure 4 for the training process (Algorithm-1). Hybrid indexes can restrict search space per key based on the model used. Parameters like stage count, width, neural net configuration, and error threshold can be optimized through grid search. If learning data distribution is impractical, all models are swapped with B-Trees, making the worst-case performance comparable. Some overhead may occur between phases, but overall performance is similar [6].

ALGORITHM 1: HYBRID END-TO-END TRAINING [6]

Input: int threshold, int stages[], NN_complexity
Data: record data[], Model index[][]
Result: trained index
M = stages.size;
tmp_records[][];
tmp_records[1][1] = all data;
for $i \leftarrow 1$ **to** M **do**
  for $j \leftarrow 1$ **to** $stages[i]$ **do**
    index[i][j]= new NN trained on tmp_records[i][j];
    if $i < M$ then
      for $r \in tmp\_records[i][j]$ **do**
        $p = f(r.key) / stages[i+1]$;
        tmp_records[$i+1$][p].add(r);
for $j \leftarrow 1$ **to** $index[M].size$ **do**
  index[M][j].calc_err(tmp_records[M][j]);
  if $index[M][j].\max\_abs\_err > threshold$ **then**
    index[M][j] = new B-Tree trained on tmp_records[M][j];
return index;

Figure 4: *Algorithm 1: Hybrid End-To-End Training [6]*

3.3.3. HASH-MODEL INDEX

According to [6], a method for better hash function training involves learning the CDF of data key distribution. Contrary to range indexes, they avoid compactly storing records or strictly sorting them. Under static learned indexes, the CDF is scaled to the desired hash map size M using ($h(K) = F(K) * M$) with key K as the Hash-Model Index's hash function. Conflicts are minimized if the model F accurately learns the CDF. This hash function is independent of Hash-map architecture and compatible with various methods [6]. RMI's recursive model design is applied. Index size and efficiency have a trade-off influenced by the dataset and model, similar to RMI [6]. Inserts in hash-model indexes mirror traditional Hash-maps. A key is hashed using h(k), and conflicts are handled by the Hash-map algorithm. The learned hash function maintains efficiency, and inserts follow a distribution akin to data distribution [6].

DOI: http://dx.doi.org/10.25098/8.1.30

The Scientific Journal of Cihan University – Sulaimaniya **PP: 186-209**
Volume (8), Issue (1), June 2024
ISSN 2520-7377 (Online), ISSN 2520-5102 (Print)

3.3.4.LEARNED BLOOM FILTER (LBF)

Another static learned index, the Learned Bloom Filter (LBF) proposed at [6]. Utilized a Neural Network model as an initial-filter (pre-filter [12]), followed by a small Bloom Filter as a secondary-filter (backup-filter [12]). Unlike traditional indexes, which don't estimate key distribution, LBF takes both keys $\mathcal{K}$ and non-keys $\mathcal{U}$ datasets into account for machine learning. The neural network is trained for binary classification [6]. The output $f(x)$ represents the "probability" that key x belongs to $\mathcal{K}$. while in the model, the FPR is reduced to a non-zero value, resulting in an increase in the FNR also to a non-zero value. in contrast to Bloom filters. So, they set a threshold τ above which the key is considered present in $\mathcal{K}$,they establish $x \in \mathcal{K} | f(x) \geq \tau$. To ensure zero false negatives, a set of false negatives $\mathcal{K}_\tau^- = \{x \in \mathcal{K} | f(x) < \tau\}$, is derived for implementing an overflow Bloom filter (secondary-filter). The model assumes the key exists if $f(x) \geq \tau$, otherwise it's checked with the backup filter [6] as shown in Figure 11.

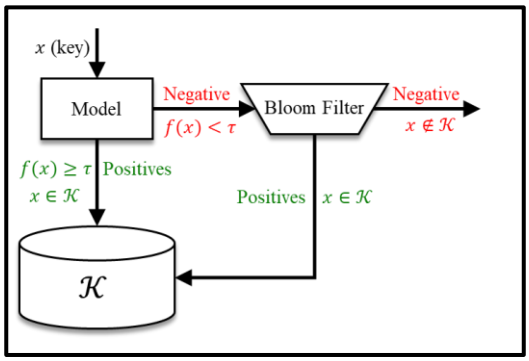

Figure 5: Learned Bloom filter [6].

3.3.5.SANDWICHED LEARNED BLOOM FILTER

Sandwiched Learned Bloom Filter (Sandwiched LBF) also static learned index proposed by [13] as an optimization of the Learned Bloom Filter from [6], by using additional Bloom filter (Initial Bloom filter) in front of function f, so as to exclude almost all queries for keys not in $\mathcal{K}$. Instead of declaring that input x is in $\mathcal{K}$, this initial Bloom filter forwards all matching elements to the learnt function f. Otherwise, it produces an instant negative response ($x \notin \mathcal{K}$). Then, as previously (Learned Bloom Filter from [6]), they utilize the function f to try to eliminate false positives from the initial Bloom filter. The backup filter at next step, used to return back keys from $\mathcal{K}$ that were false negatives from f. If the initial bloom-filter is set up to eliminate more false positives at first, the backup Bloom filter could be weak, and allowing almost everything to pass through, so it will be reasonably small, so, any extra bits have to go toward the initial Bloom filter, where the budget (amount) of the allocated bits for Bloom filters grows [13]. Figure 12 illustrate the Sandwiched Learned Bloom Filter.

DOI: http://dx.doi.org/10.25098/8.1.30

The Scientific Journal of Cihan University – Sulaimaniya **PP: 186-209**
Volume (8), Issue (1), June 2024
ISSN 2520-7377 (Online), ISSN 2520-5102 (Print)

3.3.6. DORAEMON LEARNT INDEX

Proposed for dynamic workloads, the dynamic learned index (C. Tang, Z. Dong, M. Wang, Z. Wang, and H. J. a. p. a. Chen, 2019) [14] improves latency by extending training data with access frequencies and addressing access pattern and skewed queries. Doraemon caches learned models and fine-tunes them when similar input distributions are encountered. It comprises three components: Training Set Generator, Counselor, and Finalizer, adapting read access patterns [14]. To integrate read access patterns, frequently accessed keys are duplicated in the training set. For keys with higher access, their positions are shifted, enhancing accuracy without improving error boundaries. Counselor tunes the model, while Finalizer retrains final stage models using the original dataset. Quick due to linearity, this procedure ensures corrected position information [14].

3.3.7. FITTING-TREE

Another dynamic learned indexes called FITing-Tree proposed by (Galakatos et al., 2019) [15], innovatively incorporates data awareness. Using piece-wise linear functions, it approximates an index with limited error at creation, balancing lookup performance and space consumption. Employing an adjustable error threshold, FITing-Tree adapts to datasets and workloads. A cost model helps determine error factors for search latency or storage budget [15]. It models index as a monotonically growing function mapping keys to storage locations, contrasting clustered B+ trees. FITing-Tree segments data into variable-sized parts meeting error thresholds, maintaining a fixed-size array for each segment. ShrinkingCone algorithm creates segments while expanding them within error constraints, ensuring efficient segment length and insertion. Segments are arranged in a B+-Tree structure for efficient retrieval. FITing-Tree stores only start keys and slopes for linear interpolation, adapting to sorted data and non-primary key attributes, using indirection layers for the latter. Point and range queries involve B+-tree searches followed by key location within segments [15].

3.3.8.ALEX-INDEX

The dynamic learned index ALEX [16] was designed to tackle challenges posed by short-range queries, point lookups, data modification (inserts, updates, deletes), and bulk loading. It merges learned index principles with established storage and indexing techniques. Unlike [6], ALEX adjusts RMI height and shape dynamically depending on the workload. Storing data at leaf levels as in B+ Trees enables individual nodes to extend and split more efficiently, while exponential search corrects RMI mispredictions. ALEX employs model-based insertion, enhancing search performance by minimizing model mispredictions. Unlike [6], ALEX eliminates the need to adjust model count parameters for different datasets or workloads [16]. ALEX's design involves a tree similar to a B+Tree, with expanding/shrinking nodes using Gapped Arrays to absorb insertions and allow accurate data placement through model-based insertion (Figure 6). Gaps are filled by adjacent keys for optimal search efficiency [16].

ALEX combines dynamic expansion, node splitting, and selective model retraining based on cost models adapting to changing workloads, ensuring efficiency despite dynamic data distribution changes. These advantages are achieved without manual parameter adjustments [16]. Leaf nodes, or "data nodes," store linear regression models, two Gapped Arrays for keys and payloads, similar to

DOI: http://dx.doi.org/10.25098/8.1.30

The Scientific Journal of Cihan University – Sulaimaniya **PP: 186-209**
Volume (8), Issue (1), June 2024
ISSN 2520-7377 (Online), ISSN 2520-5102 (Print)

B+Tree leaf nodes. Internal nodes predict the position of child pointers using models, aiding traversal and partitioning the key space flexibly (Figure 6) [6].

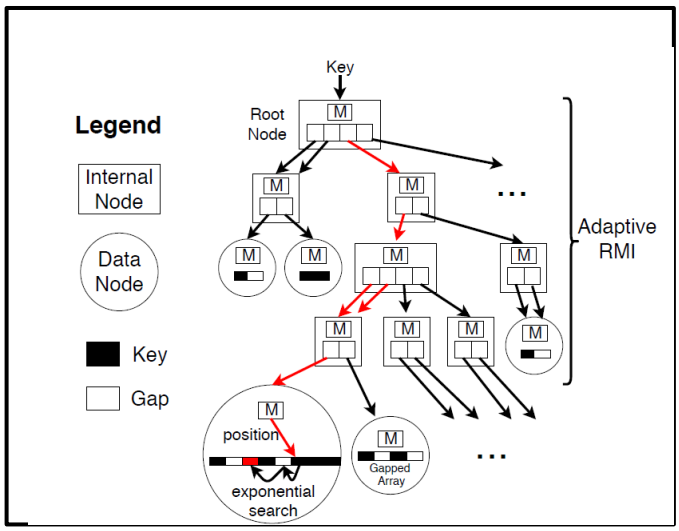

Figure 6: *ALEX Design [15].*

3.3.9.PIECEWISE GEOMETRIC MODEL INDEXES (PGM)

The Piecewise Geometric Model index (PGM-index) by (P. Ferragina and G. J. P. o. t. V. E. Vinciguerra, 2020) [17] is a fully dynamic and compressed learning index. PGM adapts linear models to keys with an error tolerance ε in a recursive structure. PGM adjusts to space or latency constraints and excels in predecessor, range, and update queries under high-performance limits [17]. The PGM-index is parameterized with $\varepsilon \geq 1$, solving the indexable dictionary issue on a multiset S of n keys from universe U. A linear model approximates key locations, and binary search corrects predictions ($\varepsilon = 2$) [17]. (Algorithm 2) in Figure 7 demonstrate PGM's recursive construction. PGM's first component is the Piecewise Linear Approximation model (PLA-model), using an optimal streaming algorithm for minimal segments $O(n)$. It maps keys to predicted array locations, maintaining ε

distance. Recursive construction transforms the optimum PLA-model into segments, adapting to key distribution. This recursive structure forms the PGM-index's levels and nodes. PGM's unique construction differs from FITing-Tree and RMI. Each PGM level has a PLA-model, and nodes hold segments from that model [17]. (Algorithm 3) in Figure 7 demonstrate PGM's recursive search and construction [17].

DOI: http://dx.doi.org/10.25098/8.1.30

Figure 7: (a) Algorithm 2 the build of PGM Index [17], (b) Algorithm 3 Top-Down PGM Recursive search [17].

The Scientific Journal of Cihan University – Sulaimaniya **PP: 186-209**
Volume (8), Issue (1), June 2024
ISSN 2520-7377 (Online), ISSN 2520-5102 (Print)

ALGORITHM 2 BUILD-PGM-INDEX(A, n, ε) [17]

1 $levels$ = an empty dynamic array
2 $i = 0$; $keys = A$
3 **repeat**
4 M = BUILD-PLA-MODEL$(keys, \varepsilon)$
5 $levels[i] = M$; $i = i + 1$
6 m = SIZE(M)
7 $keys = [M[0].key, \ldots, M[m-1].key]$
8 **until** $m = 1$
9 **return** $levels$ in reverse order

(a)

Algorithm 3 QUERY$(A, n, \varepsilon, levels, k)$ [17]

1 $pos = f_r(k)$, where $r = levels[0][0]$
2 **for** $i = 1$ **to** Size$(levels) - 1$
3 $lo = max\{pos - \varepsilon, 0\}$
4 $hi = min\{pos + \varepsilon, Size(levels[i]) - 1\}$
5 s = the rightmost segment s' in $levels[i][lo, hi]$ such that $s'.key \leq k$
6 t = the segment at the right of s
7 $pos = \lfloor \min\{f_s(k), f_t(t.key)\} \rfloor$
8 $lo = \max\{pos - \varepsilon, 0\}$
9 $hi = \min\{pos + \varepsilon, n - 1\}$
10 **return** search for k in $A[lo, hi]$

(b)

3.3.10. RADIXSPLINE INDEXES (RS) (READ-ONLY)

The RadixSpline (RS) index, introduced by (A. Kipf et al., 2020) [18], maps keys to their data locations. RS is a static learned index that doesn't support single updates. It involves spline points approximating data distribution and a radix table for efficient lookup [18].

RadixSpline's two components are: (1) spline points approximating data, ensuring predicted lookup position within an error bound, and (2) a radix table with r-bit prefixes as indices, narrowing spline search space. At lookup, spline points around the key are located using linear interpolation within a small data region [18]. RS construction integrates GreedySplineCorridor algorithm for spline and radix table creation in one pass [18].

In RS lookup (illustrated in Figure 8): an r-bit prefix determines radix table pointers for a limited spline search range. Binary search locates spline points around the key, linear interpolation generates approximate position, and a final binary search within error bounds refines the position [18].

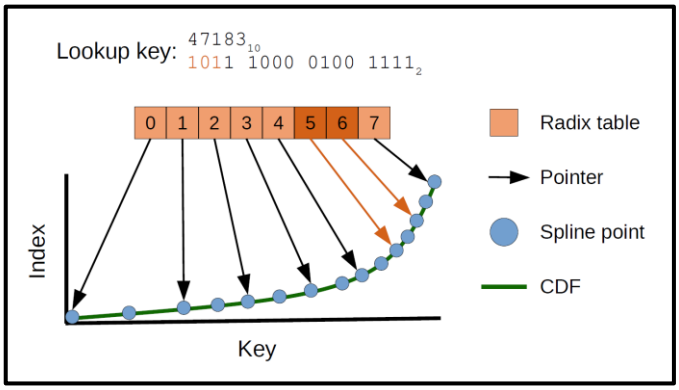

Figure 8: *Illustrates an example of radix spline index lookup [18].*

DOI: http://dx.doi.org/10.25098/8.1.30

The Scientific Journal of Cihan University – Sulaimaniya **PP: 186-209**
Volume (8), Issue (1), June 2024
ISSN 2520-7377 (Online), ISSN 2520-5102 (Print)

4. DISCUSSION AND RESULTS

This section is split into two parts, each with a distinct purpose. In the first part, we comprehensively explain and analyze the reviewed techniques, covering time complexities, structures, strengths, and limitations. This provides an understanding of each technique's applicability and use cases. The second part summarizes the theoretical analyses using two tables, making the information concise and accessible. Additionally, we present a taxonomy categorizing Learned Indexes into static and dynamic categories, further enhancing clarity. This structured approach ensures readers grasp the techniques' nuances, catering to those seeking both in-depth insights and quick reference.

4.1. CRITICAL DISCUSSION AND EXPLANATION

The section has two parts: Part one explains the techniques, including time complexities, structures, strengths, and limitations, giving insights into their use cases. Part two summarizes the analyses in concise tables, adds a taxonomy categorizing techniques into static and dynamic Learned Indexes, enhancing clarity. This structured approach aids readers in understanding nuances and caters to different reading preferences.

When analyzing theoretical aspects of indexing techniques, including complexities and Big O notation calculation for traditional database indexing and Learned Indexes, exact analysis can be challenging due to lack of explicit details, comparison operations, and implementation specifics in the research. Factors contributing to this challenge are explained later in this section. Instead, an estimated Big O notation can be used based on fundamental characteristics of the technique [26]. This estimation helps approximate performance and understand scalability. Big O notation is a mathematical way to describe an algorithm's upper bound complexity. It aids in comparing techniques, understanding scalability, and making informed decisions for specific applications [27, 28]. This approximation, though valuable, is not exact, focusing on idealized computational models and ignoring hardware and software complexities.

Actual performance is influenced by hardware constraints (CPU architectures, memory hierarchies, disk access times) and implementation details, introducing deviations from theoretical analysis [6, 16, 26-28]. Key factors include:

1. Algorithmic Variations: Real-world implementations may differ due to optimizations or heuristics, affecting operation counts [6, 16, 26-28].
2. Dataset Characteristics: Data type, size, distribution, and skewness impact complexity and indexing strategies [5, 6, 16, 21].
3. Implementation Details: Specific choices in optimization, data structures, and implementation affect time and space complexities [6, 16, 26-28].
4. Machine Learning Technique: Choice of ML algorithm impacts computational demands, training complexities, and prediction accuracy [6, 10, 23].
5. Number of Models: Using multiple models or auto-generation based on data/query workload adds computational complexity [6, 10].

DOI: http://dx.doi.org/10.25098/8.1.30

The Scientific Journal of Cihan University – Sulaimaniya **PP: 186-209**
Volume (8), Issue (1), June 2024
ISSN 2520-7377 (Online), ISSN 2520-5102 (Print)

6. Retraining: Frequent model retraining impacts performance, adding time/resource demands [6, 10, 14-16].
7. Error Bound Factor: Error bounds affect prediction accuracy and efficiency trade-off, influencing complexity and performance guarantees [6, 10, 15-18].
8. Query Type and Workload: Query type and workload characteristics introduce variations in complexity based on indexing strategies [6, 10, 15-18].

Performance can vary based on factors mentioned earlier. The following offers a detailed overview of the reviewed techniques

On a hand, traditional indexing techniques like B$^+$tree have limitations. B$^+$tree has $O(B \log_B n)$ lookup and I/O complexity, also $O(B \log_B n)$ complexity for dataset modification (insert, delete, update) in worst and best cases. It consumes $O(n)$ space and has building time of $O(n \log^2{}_B n)$ in worst-case considering maximum rebalancing of the tree, and building time of $O(n)$ in best-case scenarios. B$^+$-Tree is adaptive and versatile, but lacks adaptive learning on CDF, struggles with skewed distributions, and can have overhead for large datasets. It suits relational databases, file systems, key-value storage, transaction processing, and search engines.

While on the other hand, Learned Indexes enhance query efficiency using machine learning and CDF-based statistical models to replace traditional structures. Despite challenges in theoretical analysis and $Big\ O$ notation, state-of-the-art learned index techniques are examined as follows:

The Recursive Model Index (RMI), a Learned Index with a Directed Acyclic Graph (DAG) structure, aims to reduce space complexity. In worst-case scenarios, high irregularity of key distribution and complex models hinder accurate predictions. Conversely, best-case scenarios with regular CDF patterns and lightweight models enable efficient key partitioning. Let n be dataset size, s stages, and m model space complexity. RMI space complexity for both scenarios approximates $O(s * m)$. In best-case scenarios, it can minimize if models are lightweight, metadata requirements are minimal, CDF enables efficient partitioning, and models can be shared. As a result, RMI can approximate $O(1)$ without guarantees, indicating constant space usage. Lookup time complexity approximates $O(s)$ for both scenarios. In worst cases, traversing all stages may be needed due to irregular CDF or skewed distribution, leading to $O(s)$ time complexity. In best cases, with regular CDF or accurate predictions within a stage, lookup time may be $O(1)$ without guarantees, indicating constant time complexity. RMI's I/O time complexity mirrors lookup time for worst cases, leading to $O(s)$ complexity. However, in best cases, optimal disk access patterns result in efficient data block access and $O(1)$ complexity. RMI has a build time complexity of $O(s * n)$ in both cases due to traversing the dataset at each stage. It lacks data modification capabilities, supports limited query types, has training overhead, and balances accuracy and performance trade-offs. Overall, RMI is suitable for large datasets, in-memory databases, data warehousing, analytics, read-heavy workloads, and decision support systems.

DOI: http://dx.doi.org/10.25098/8.1.30

The Scientific Journal of Cihan University – Sulaimaniya **PP: 186-209**
Volume (8), Issue (1), June 2024
ISSN 2520-7377 (Online), ISSN 2520-5102 (Print)

Hybrid Indexes, a subset of Learned Indexes, combine RMI and traditional B-tree structures. Last-stage models may be replaced by B-trees if learning is challenging. It inherits characteristics of both methods, leaning towards one depending on the scenario. Similar to RMI, Hybrid Indexes aim to minimize space complexity, approximating $O(sm)$ at best-case and $O(n)$ at worst-case scenarios. In best cases, lookup complexity is around $O(s)$, and I/O complexity approximates $O(1)$. In worst cases, lookup and I/O complexities are approximately $O(s + log\ n)$ when last-stage models are replaced by B-trees. Like RMI, Hybrid Indexes have a significant build time of $O(s * n)$ in both scenarios. Hybrid Indexes, like RMI, depend on key distribution, lack data modification, support limited queries, and have training overhead. Balancing accuracy, size, and performance is crucial. These characteristics make Hybrid Indexes suitable for large databases, data analytics, read-heavy workloads, and latency-sensitive applications.

The Hash-Model Index is a Learned Index used for point queries, replacing traditional hash functions with a learned model from key data's CDF. Its structure mostly retains traditional hash index components. Key factors affecting its time and space complexities are the hash model, dataset and hash table sizes, and collision handling. For dataset size represented by n and hash table size by d (buckets), the best-case occurs with ideal collisions (even distribution or no collision). Lookup and I/O complexities approximate $O(1)$ in the best-case, while build time complexity is $O(n)$ regardless of the scenario. Worst-case lookup and I/O complexities could approximate $O(n)$. The Hash-Model Index lacks structural adaptability, supports only point queries, has longer build times than traditional hash indexes due to model learning, and larger size due to added storage for the hash model. Suitable use cases include scenarios requiring high-speed querying.

The Learned Bloom Filter (LBF) within the Learned Index framework combines a learned model and a traditional Bloom Filter. In lookup, the worst-case scenario occurs when the queried key isn't in the filter, resulting in a false negative. This directs the lookup to the back filter, incurring extra computation. In the best case, the queried key is present in the filter, yielding a correct positive prediction by the model. Despite model and back filter complexities, LBF's lookup, I/O, and space complexities are approximately $O(1)$. However, LBF's complexities are greater than traditional Bloom Filters due to model intricacies and training overhead. Building LBF has a complexity of $O(n)$, regardless of the scenario. LBF doesn't support dataset modification, handles only existence queries, and while it reduces false positive rates, some still occur. It's suitable for large-scale datasets with complex patterns and applications with low tolerance for false positives.

The Sandwiched Learned Bloom Filter enhances the Learned Bloom Filter by adding a Bloom filter before the learned model. This reduces negative queries sent to the model, improving query performance. In the worst-case scenario, if the initial Bloom filter has high false positives or the model struggles, the front filter's benefits lessen. In the best case, with a good initial Bloom filter and accurate model predictions, the approach shines. The time and space complexities remain similar to Learned Bloom Filter. The front filter can enhance query performance by avoiding model overhead for negatives in the best case. However, it introduces additional space and potential computational complexity during construction and lookup in the worst case. Like LBF, the Sandwiched Learned

DOI: http://dx.doi.org/10.25098/8.1.30

The Scientific Journal of Cihan University – Sulaimaniya **PP: 186-209**
Volume (8), Issue (1), June 2024
ISSN 2520-7377 (Online), ISSN 2520-5102 (Print)

Bloom Filter involves a trade-off between efficiency and accuracy. It's suited for LBF use cases and scenarios with high query throughput.

Doraemon, a Learned Index solution for dynamic workloads, introduces complexity with components like Training Set Generator, Counselor, and Finalizer, augmenting data based on access patterns. In the worst case, if data distribution changes drastically or becomes skewed, predictions falter, impacting lookup times. A major distribution shift may demand extensive model retraining, degrading performance and efficiency. Conversely, Doraemon excels in stable or predictable distribution scenarios, leveraging cached models for accurate predictions and faster lookups. Build time complexity approximates $O(n)$ for both cases, considering extra complexity in the worst case. Lookup and I/O time complexity depend on model complexity, approximating $O(1)$ in the best case. Space complexity is model-dependent regardless of the structure. Dataset modifications' impact depends on structural implementation. Doraemon's sensitivity to data distribution and reliance on a representative training dataset can lead to computational overhead and complexity. Adapting the structure for dynamic modifications can be challenging. Doraemon requires storage for trained models and cached data. It suits database indexing, time-series data, log analytics, IoT data, and real-time analytics.

FITing-Tree, a mixed Learned Index, embeds Learned Models (Linear Segments) within a B^+-Tree structure. With n as dataset size, m models, B B+-Tree order, $buff$ segment buffer size, and ε error threshold, build time complexity at worst-case (max segments) is approximates $O(n + log_B m)$, and at best-case (min segments) is $O(n)$. Lookup time complexity (worst-case) is $O(log_B\, m\, +\, log_2\, \varepsilon\, +\, log_2\, buff\,)$, and (best-case) is $O(log_B\, m)$. I/O time complexity is $O(log_B\, m)$ regardless of the scenario. Space complexity worst-case is $O(log_\varepsilon\, n)$ and best-case is $O(log_B\, m)$, Modification (insert, delete, update) time complexity is Modification (insert, delete, update) time complexity is $O(log_B\, m\, +\, buff)$ at worst-case, and $O(log_B\, m)$ at best-case. FITing-Tree supports common query types and dataset modifications, adapting to distribution changes. Sensitive to data distribution, it can have a large model count, affecting B+-Tree height. The error threshold balances size and performance. FITing-Tree suits large-scale datasets, complex patterns, data exploration, analytics, read-heavy workloads, latency-sensitive apps, clustered and non-clustered indexes, and applications with dataset modification.

ALEX, a mixed Learned Index, balances size and performance dynamically using cost models, eliminating manual tuning. Overhead before RMI initialization and index construction for node expansion and splitting. Given n (dataset size), B (max node size), m (min models/partitions), ALEX space complexity is $O(n\, +\, m)$. Build time complexity worst-case approximates $(n\, log_B\, m\, +\, log\, B\, +\, m)$, best-case $O(n\, log_B\, m)$. Lookup time complexity $O(log_B\, m\, +\, log\, B)$, , best-case $O(log_B\, m)$. I/O time complexity worst-case $O(log_B\, m)$, best-case $O(1)$. All modification (insert, delete, update) time complexities approximate $O(log_B\, m\, +\, log\, B\, +\, m)$ worst-case, $O(log_B\, m\, +\, log\, B)$ best-case. ALEX supports point and range queries. Each level partition requires model training, introducing initial overhead. Sensitive to data distribution, skewed data leads to more splits/merges, impacting performance. Adaptive to changing data distributions, yet rapid changes may affect model-based pointers and performance. ALEX reduces memory/storage but still needs

DOI: http://dx.doi.org/10.25098/8.1.30

The Scientific Journal of Cihan University – Sulaimaniya **PP: 186-209**
Volume (8), Issue (1), June 2024
ISSN 2520-7377 (Online), ISSN 2520-5102 (Print)

additional storage. While efficient with queries and changing distributions, it lacks explicit optimization based on query patterns.

PGM-index, a Learned Index for point and range queries, balances index size and performance through dynamic recursive structure. With n (dataset size), ε (error threshold), $c \geq 2\varepsilon$, B_z (block size), m_opt (optimal models/segments), PGM-Index aims to minimize models while specifying ε. Regardless of model complexities, PGM-Index's build time complexity approximates $O(n)$. Lookup time worst-case $(log_{B_z} m_{opt} + log_2 \varepsilon)$, best-case $O(log_{B_z} m_{opt})$. I/O time worst-case $O(log_c m_{opt})$, best-case $O(1)$. Insertion, deletion time complexities $O(log\, n)$ $amortized$ worst-case, $O(1)$ best-case. PGM-Index's dynamic array introduces overhead, inefficient memory use during resizing. Query distribution impacts performance; real-time learning may not keep up with data changes. PGM-Index versatile and powerful, suitable for range queries, time series data, low-latency applications. Fits in-memory databases, space-efficient systems. Handles high-dimensional data efficiently for multi-dimensional indexing.

RadixSpline is a static learned index designed for efficient index building, focusing on read-only scenarios. It can be built in a single pass over sorted data, with complexities depending on parameters like n (dataset size), ε (error bound), r (prefix radix bit), and k (spline points). While suitable for read-heavy workloads, it lacks support for dynamic dataset updates. Its build time complexity is O(n) regardless to scenario. Space complexity approximates $O(k + 2^r)$ when $k \leq n$ for both scenario with significant different based on ε. Lookup time worst-case is $(log_2 k)$, best-case $O(1)$. I/O time is similar, approximating $O(log_2 k)$ worst-case, and, $O(1)$ best-case. RadixSpline doesn't adapt to changing data distributions, focusing on individual key lookups. Its construction process is complex, involving various checks and adjustments. It's well-suited for batch processing of large datasets, archival data with infrequent changes, and data warehousing scenarios. Also useful for historical data analysis.

DOI: http://dx.doi.org/10.25098/8.1.30

The Scientific Journal of Cihan University – Sulaimaniya **PP: 186-209**
Volume (8), Issue (1), June 2024
ISSN 2520-7377 (Online), ISSN 2520-5102 (Print)

4.2. THEORETICAL RESULTS

This section includes two components: a summary table compiling theoretical analyses and a taxonomy classifying reviewed Learned Indexes.

4.2.1. SUMMARY TABLES

We summarize the theoretical analyses from the initial result component. in two tables for clarity. These tables condense key findings, making them easily understandable and accessible. They offer a quick overview of each reviewed technique, aiding comparison. The summary tables are presented in (Table 1 and Table 2).

Key features of the reviewed Learned Indexes are divided into two summary tables. The first table covers structure, constraints, and use cases. The second table includes complexities, dataset modification (insert and delete), and enabled properties. Symbols used for representing complexities are:{n (number of keys), B (tree order/fanout for B^+-Tree, Fitting-Tree, and ALEX), B_z (page-size for PGM), s (stages/level for RMI and Hybrid Index), ε (error bound), m (number of generated model),m_{greedy} (number of the linear models generated by Fitting-Tree's greedy algorithms), m_{opt} (optimal number of linear models generated by PGM 's optimum algorithm, $buff$ (buffer location in Fitting-Tree), c (variable fan-out for PGM data structure where $c \geq \varepsilon$, and k (number of spline knot generated by RadixeSpline's spline algorithm of }.

It's important to note that these complexities are simplified assumptions, approximating the actual model complexity and training process. In reality, the build time complexity would also be affected by the previously mentioned factors such as the quality of the dataset, the complexity of the learned model architecture, and the available computational resources.

4.2.2. TAXONOMY

In the next section, a taxonomy of the reviewed Learned Index techniques is presented. It divides these methods into two groups: static and dynamic Learned Indexes. The static category includes Existent Indicators, Point Indexes, and Point/Range Indexes. Within this, Learned Bloom Filter and Sandwiched Learned Bloom Filter fall under Existent Indicators, Hash-Model under Point Index, and RMI, Hybrid, and RadixSpline under Point/Range Index. In the dynamic category, all techniques fall under Point/Range Indexes: Doraemon, Fitting-tree, Alex, and PGM. This taxonomy offers a structured way to understand these techniques, aiding in their organization and providing a comprehensive view of Learned Indexes. See Figure 9 for the taxonomy illustration.

DOI: http://dx.doi.org/10.25098/8.1.30

The Scientific Journal of Cihan University – Sulaimaniya **PP: 186-209**
Volume (8), Issue (1), June 2024
ISSN 2520-7377 (Online), ISSN 2520-5102 (Print)

5. CONCLUSION AND FUTURE WORK

In this comprehensive survey, we've explored the emerging realm of learned indexes, analyzing their complexities, capabilities, and limitations. Through a well-defined taxonomy, we've illuminated the diverse landscape of learned indexes, revealing their potential and challenges. The survey began by addressing fundamental questions about learned indexes, uncovering their advantages and drawbacks. Learned indexes offer improved performance, reduced memory overhead, and potential to revolutionize data indexing and querying. Efficient point and range queries, adaptability to data shifts, and reduced I/O operations showcase their applicability to modern data challenges. Yet, as the survey progressed, it became evident that existing techniques are not fully equipped to meet various application and dataset needs. While promising, learned indexes face challenges such as variable query patterns, sensitivity to data distribution, training overhead, memory usage, and the accuracy-size-performance trade-off. Further innovation and adaptive techniques are necessary to bridge theoretical analyses with practical implementation. This survey points to the future of learned indexes. New techniques are needed, blending traditional index structures with the intelligence of learned models. Dynamic data requires indexes that evolve with it, responding to distribution and query changes effectively. In conclusion, this survey offers an expansive view of learned indexes, highlighting their potential and limitations while emphasizing the need for focused research and innovation. As data evolves, the learned index paradigm must discover techniques that harmoniously blend tradition and innovation. Through exploration, we aim to propel learned indexes toward a future that empowers data-driven applications with efficiency, adaptability, and lasting relevance.

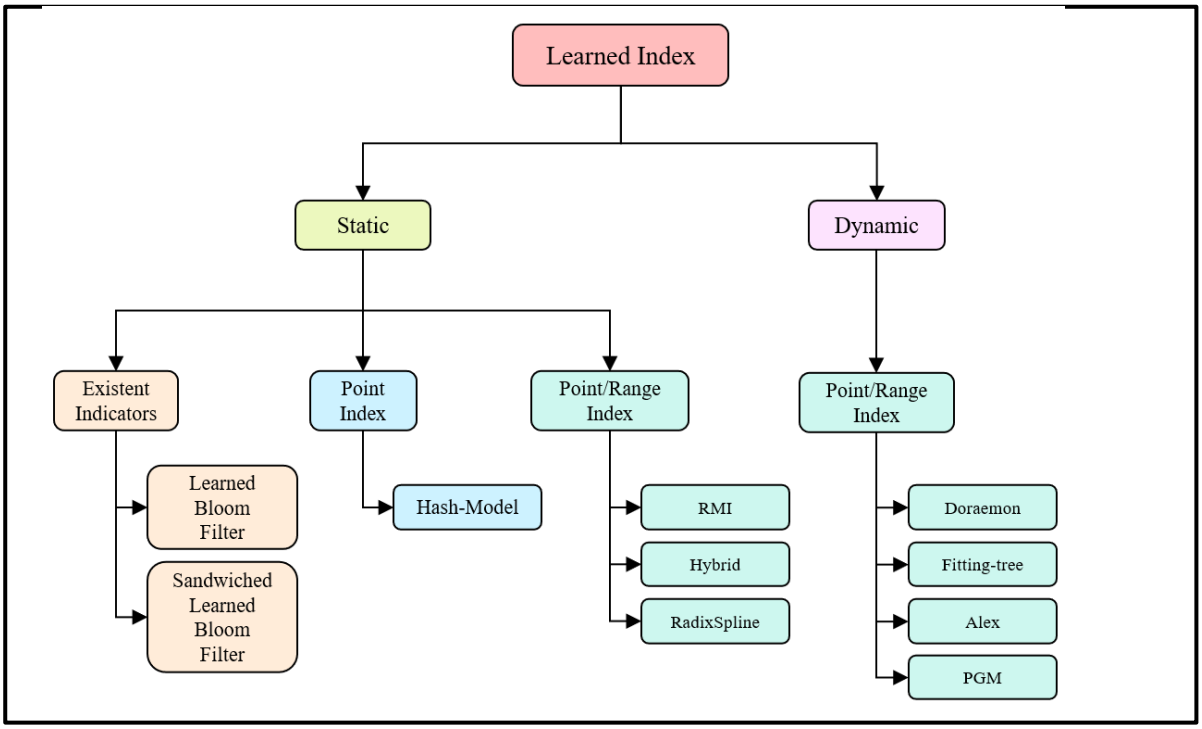

Figure 9: Taxonomy of the reviewed state of the art Learned Index

DOI: http://dx.doi.org/10.25098/8.1.30

The Scientific Journal of Cihan University – Sulaimaniya **PP: 186-209**
Volume (8), Issue (1), June 2024
ISSN 2520-7377 (Online), ISSN 2520-5102 (Print)

Table 1: Summary Table 1

Technique	Structure	Constraints	Suitable Use Cases
B+-Tree	Tree	Lack of Adaptive Learning on the CDF or Prediction-based Optimization. Suboptimal for Skewed Distributions. Takes up Space according to the Dataset Size. Potential Overhead for Large Datasets	Database Indexing mainly Relational Databases. File Systems. Key-Value Stores. Transaction Processing Systems. Search Engines.
RMI	Directed Acyclic Graph (DAG)	Data Distribution Dependency. Dataset Updates Inability. Limited Query Types. Training Overhead. Trade-off Between Accuracy and Performance.	Large Datasets. Caching and In-Memory Databases. Data Warehousing and Analytics. Read-Heavy Workloads. Decision Support Systems.
Hybrid	mixed (RMI Models + Algo-Index)	Data Distribution Dependency. Dataset Updates Inability. Limited Query Types. Training Overhead. Increased Memory Usage. Trade-off Between (Accuracy. Size. and Performance).	Large-Scale Databases. Data Exploration and Analytics. Read-Heavy Workloads. Latency-Sensitive Applications.
Hash-Model	Hash-like model	Data Distribution Dependency. Dataset Updates Inability. Limited Query Types. Training Overhead. Increased Memory Usage. Trade-Off Between Performance and Index Size	High-Speed Querying
LBF	mixed (Mode + Bloom-Filter)	Data Distribution Dependency. Dataset Updates Inability. Limited Query Types. Training Overhead. Memory Consumption Related to Traditional BF. False Positive Rate	Large-Scale Datasets. Complex Data Patterns. Applications With Lower Tolerance False Positives
Sandwiched LBF	mixed (Mode + Bloom-Filters)	Data Distribution Dependency. Dataset Updates Inability. Limited Query Types. Training Overhead. Addition Memory Consumption Related to LBF. False Positive Rate. Trade-Off between Efficiency and Accuracy	Large-Scale Datasets. Complex Data Patterns. Applications With Lower Tolerance False Positives. High Query Throughput Scenarios
Doraemon	Learned Model	Data Distribution Sensitivity. Training Overhead. Model Retraining due to Dataset Modifications.	Database Systems. Time-Series Data. Log Analytics Applications. Iot Data Processing. Real-Time Analytics Scenarios
Fitting-Tree	mixed (B+-Tree of Mode)	Data Distribution Dependency. Training Overhead. Number of Linear Models can be Large and lead to an Increase in its B+-tree Height. Model Retraining due to Dataset Modifications. Trade-off Between (Size. and Performance).	Large-Scale Datasets. Complex Data Patterns. Data Exploration and Analytics. Read-Heavy Workloads. Latency-Sensitive Applications. (clustered) index. Secondary (Non-Clustered) indexes. Applications With Dataset Modification
Alex	Tree-like Model	Data Distribution Dependency. Training Overhead. Model Scaling or Retraining due to Dataset Modifications. Additional Memory and Storage. Rapid and Frequent Dataset Changes may Decrease Performance. doesn't Adapt to Query Patterns.	Large-Scale Datasets. Time-Series Data. Log Analytics Applications. Iot Data Processing. Real-Time Analytics Scenarios
PGM	Tree-like Model	Data Distribution Dependency. Training Overhead. Additional Memory Overhead. Query Sensitivity. Insertion/Deletion Overhead. Parameter Selection. Dynamic Data Changes. Inability to keep Real-time Learning	Database Indexing. Range Queries. Dynamic and Compressed Indexing. In-Memory Databases. Time Series Data. Low-Latency Indexing. Streaming Data. Online Recommender Systems. High-Dimensional Data
RadixSpline	Two Level (Radix Table Array + Array of Spline-Knot Model)	Read-Only. Static Construction. Complex Construction Proces and implementation. Limited Support for Range Queries.	Batch Data Processing. Archival Data. Data Warehousing.

DOI: http://dx.doi.org/10.25098/8.1.30

The Scientific Journal of Cihan University – Sulaimaniya **PP: 186-209**
Volume (8), Issue (1), June 2024
ISSN 2520-7377 (Online), ISSN 2520-5102 (Print)

Table 2: Summary Table 2

Technique	Estimated Complexities										Enables Propertie					
	Build time		Lookup time		I/Os	size		Dataset Modification								
	Worst Case	Best Case	Worst Case	Best Case	Worst Case	Best Case	Worst Case	Insert		Delete		Common Query Types	Structural Adaptation	Error limit	space Optimization	Dataset Updating
								Worst Case	Best Case	Worst Case	Best Case					
B+-Tree	$O(n \log_B^2 n)$	$O(n)$	$O(B \log_B n)$	$O(B \log_B n)$	$O(B \log_B n)$	$O(n)$	$O(n)$	$O(B \log_B n)$	$O(B \log_B n)$	$O(B \log_B n)$	$O(B \log_B n)$	Almost All Types	Yes	(N/A)	No	Yes
RMI	$O(n*s)$	$O(n*s)$	$O(s)$	$O(s)$	$O(s)$	$O(s*m)$	$O(s*m)$	(N/A)	(N/A)	(N/A)	(N/A)	Point and Range Queries	No	No	No	No
Hybrid	$O(s*n)$	$O(s*n)$	$O(s + \log n)$	$O(s)$	$O(s + \log n)$	$O(1)$	$O(n)$	(N/A)	(N/A)	(N/A)	(N/A)	Point and Range Queries	No	partial	No	No
Hash-Model	$O(n)$	$O(n)$	$O(n)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$	(N/A)	(N/A)	(N/A)	(N/A)	Point Queries	No	No	No	No
LBF	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	(N/A)	(N/A)	(N/A)	(N/A)	Existence Queries	No	No	No	No
Sandwiched LBF	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	(N/A)	(N/A)	(N/A)	(N/A)	Existence Queries	No	No	No	No

DOI: http://dx.doi.org/10.25098/8.1.30

The Scientific Journal of Cihan University – Sulaimaniya **PP: 186-209**
Volume (8), Issue (1), June 2024
ISSN 2520-7377 (Online), ISSN 2520-5102 (Print)

Sandwiched LBF	O(n)	O(1)	O(1)	O(1)	O(1)	O(1)	(N/A)	(N/A)	(N/A)	Existence Queries	No	No	No
Doraemon	O(n)	O(n)	Model Dependent	O(1)	Model Dependent	O(1)	Unrelated	Unrelated	Unrelated	Allmost All Types	Yes	No	No
Fitting-Tree	O(n+$\log_B m_{greedy}$)	O(n)	O($\log_B \varepsilon + \log_2$ buff)	O($\log_B m_{greedy}$)	O($\log_B m_{greedy}$)	O(m_{greedy})	O($\log_B m_{greedy}$)	O($\log_B m_{greedy}$ + buff)	O($\log_B m_{greedy}$)	Allmost All Types	Yes	Yes	Yes
Alex	O($n \log_B m$ + $\log B + m$)	O($n \log_B m$)	O($\log_B m$ + $\log B$)	O($\log_B m$)	O($\log_B m$)	O($n + m$)	O($\log B$ m + log B)	O(logB m + log B + m)	O(logB m + log B)	Point and Range Queries	Yes	No	Yes
PGM	O(n)	O(n)	O($\log_{B^2} m_{opt}$ + $\log_2 \varepsilon$)	O($\log_{B^2} m_{opt}$)	O($\log_c m_{opt}$)	O(m_{opt})	O(1)	O($\log n$) *amortised*	O(1)	Point and Range Queries	Yes	Yes	Yes
RadixSpline	O(n)	O(n)	O($\log_2 k$)	O(1)	O($\log_2 k$)	O($k + 2^r$)	O(1)	(N/A)	(N/A)	Point and Range Queries	No	Yes	Partially

DOI: http://dx.doi.org/10.25098/8.1.30

REFERENCES

[1] A. Silberschatz *et al.*, *Database System Concepts*, SEVENTH ed. McGraw-Hill Education, 2 Penn Plaza, New York, NY 10121, 2020.

[2] G. K. Gupta, *DATABASE MANAGEMENT SYSTEMS*. McGraw Hill Education (India) Private Limited, 2018.

[3] R. Elmasri and S. B. Navathe, *Fundamentals of Database Systems*, Seventh ed. Pearson, 2016.

[4] C. Wijesiriwardana and M. F. Firdhous, "An Innovative Query Tuning Scheme for Large Databases," in *2019 International Conference on Data Science and Engineering (ICDSE)*, 2019, pp. 154-159: IEEE.

[5] A. aminuddin *et al.*, "ANALYZING THE EFFECT OF DATA SIZE VARIATION ON THE PERFORMANCE OF B-TREE AND HASH MAP INDEXING IN MYSQL AND POSTGRESQL PLATFORMS," *Journal of Critical Reviews JCR*, vol. 7, no. 12, 2020.

[6] T. Kraska *et al.*, "The Case for Learned Index Structures," presented at the Proceedings of the 2018 International Conference on Management of Data, Houston, TX, USA, 2018. Available: https://doi.org/10.1145/3183713.3196909

[7] M. Valavala and W. Alhamdani, "A Survey on Database Index Tuning and Defragmentation," *International Journal of Engineering Research & Technology*, vol. 9, no. 12, pp. 317--321, 2020.

[8] P. Neuhaus *et al.*, "GADIS: A genetic algorithm for database index selection," in *The 31st International Conference on Software Engineering \& Knowledge Engineering*, Portugal, 2019.

[9] J. Zhang and Y. Gao, "CARMI: A Cache-Aware Learned Index with a Cost-based Construction Algorithm," *Proceedings of the VLDB Endowment*, vol. 15, no. 11, pp. 2679-2691, 2022.

[10] R. Marcus *et al.*, "Benchmarking learned indexes," *Proceedings of the VLDB Endowment*, vol. 14, no. 1, pp. 1-13, 2020.

[11] P. Ferragina *et al.*, "Why Are Learned Indexes So Effective?," in *International Conference on Machine Learning*, 2020, vol. 119, pp. 3123-3132: PMLR.

[12] M. Mitzenmacher, "A Model for Learned Bloom Filters and Related Structures," *arXiv preprint arXiv:1802.00884*, 2018.

[13] M. Mitzenmacher, "A Model for Learned Bloom Filters and Optimizing by Sandwiching," *Advances in Neural Information Processing Systems*, vol. 31, 2018.

[14] C. Tang *et al.*, "Learned indexes for dynamic workloads," *arXiv preprint arXiv:1902.00655*, 2019.

[15] A. Galakatos *et al.*, "Fiting-Tree: A data-aware index structure," in *Proceedings of the 2019 International Conference on Management of Data*, 2019, pp. 1189-1206.

[16] J. Ding *et al.*, "ALEX: an updatable adaptive learned index," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 969-984.

[17] P. Ferragina and G. Vinciguerra, "The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds," *Proceedings of the VLDB Endowment*, vol. 13, no. 8, pp. 1162-1175, 2020.

DOI: http://dx.doi.org/10.25098/8.1.30

[18] A. Kipf *et al.*, "RadixSpline: a single-pass learned index," in *Proceedings of the Third International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*, Portland, Oregon, 2020, pp. 1-5: Association for Computing Machinery.

[19] Y. Wang *et al.*, "Treator: a Fast Centralized Cluster Scheduling at Scale Based on B+ Tree and BSP," in *2021 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom)*, New York City, NY, USA, 2021, pp. 324-335.

[20] A. Hadian and T. Heinis, "Interpolation-friendly B-trees: Bridging the Gap Between Algorithmic and Learned Indexes," in *Proceedings of the 22nd Internationa lConferenceon Extending Database Technology (EDBT)*, 2019, pp. 710-713: OpenProceedings.org.

[21] S. Mukherjee, "Indexes in Microsoft SQL Server," March 6, 2019. Available: at SSRN: https://ssrn.com/abstract=3415957 or http://dx.doi.org/10.2139/ssrn.3415957

[22] V. Nasteski, "An overview of the supervised machine learning methods," *Horizons*, vol. 4, pp. 51-62, 2017.

[23] R. Muhamedyev, "Machine learning methods: An overview," *Computer Modelling & New Technologies*, vol. 19, no. 6, pp. 14-29, 2015.

[24] J. Sánchez, *Probability for Data Scientists*, 1st ed. San Diego: Cognella Academic Publishing, 2020, p. 362.

[25] A. Spanos, *Probability theory and statistical inference: Empirical modeling with observational data*, Second ed. Cambridge, United Kingdom: Cambridge University Press, 2019.

[26] M. Agenis-Nevers *et al.*, "An empirical estimation for time and memory algorithm complexities: newly developed R package," *Multimedia Tools and Applications*, vol. 80, no. 2, pp. 2997-3015, 2021/01/01 2021.

[27] S. Phalke *et al.*, "Big-O Time Complexity Analysis Of Algorithm," presented at the 2022 International Conference on Signal and Information Processing (IConSIP), Pune, India, 2022.

[28] F. Dedov, *The Bible of Algorithms and Data Structures: A Complex Subject Simply Explained*. 2020.

DOI: http://dx.doi.org/10.25098/8.1.30