Adaptive Gap Placement for New Node Constructions in Tree-Based Learned Index

by

Sukphasuth Lipipan (1296306)

Supervisor: Dr. Junhao Gan

A thesis submitted in total fulfillment for the
degree of Master of Computer Science 100pts

in the
Faculty of Engineering and Information Technology
School of Computing and Information Systems
THE UNIVERSITY OF MELBOURNE

October 2024

THE UNIVERSITY OF MELBOURNE

Abstract

Faculty of Engineering and Information Technology
School of Computing and Information Systems

Master of Computer Science

by Sukphasuth Lipipan (1296306)
Supervisor: Dr. Junhao Gan

The B^+-tree has been the dominant indexing method in database management systems, used by most SQL databases. Recent enhancements take advantage of hardware optimizations such as caching and memory usage. However, the B^+-tree does not make use of existing data patterns and aims to work well for the worst cases. In contrast, the learned index offers a promising approach to optimizing indexing by leveraging machine learning models to predict the positions of keys and data in the data structure. Initial research on learned index showed significant improvements in search performance and memory consumption compared to traditional B^+-tree. However, the learned index has faced challenges in supporting insertion operations efficiently. While some approaches have tried to address this issue, they still require overhead for insertion and conflict resolution. This project proposes a new approach to strategically leaving gaps in the array based on the data distribution and optimizing node creation in the learned index. Experimental results show that our algorithm can significantly reduce conflicts and improve read and write operation performance. We believe this approach can be applied to other tree-based learned index structures and offer significant improvements in database management systems.

Declaration of Authorship

I, Sukphasuth Lipipan, declare that this thesis titled, 'Adaptive Gap Placement for New Node Constructions in Tree-Based Learned Index' and the work presented in it are my own. I confirm that:

- This thesis does not incorporate without acknowledgement any material previously submitted for a degree or diploma in any university; to the best of my knowledge and belief it does not contain any material previously published or written by another person where due reference is not made in the text.
- here necessary I have received clearance for this research from the University's Ethics Committee and have submitted all required data to the School.
- This thesis is 25736 words in length (excluding text in images, tables, bibliographies and appendices).

Signed: Sukphasuth Lipipan

Date: 1 June 2023

Acknowledgements

I want to express my sincere gratitude to my research advisor, Dr Junhao Gan, for the guidance, support, and encouragement throughout this project. I am grateful for his willingness to share his expertise and knowledge with me, and for his patience and understanding when I made mistakes. I am also grateful to my family and friends for their support and encouragement. I could not have completed this project without their love and support.

Contents

List of Figures

List of Tables

Acronyms

ALEX Adaptive Learned Index. vii, 3, 6, 11, 12, 14, 15, 18, 21, 22, 25, 27, 30, 99

CDF Cumulative Distribution Function. 7, 8, 18

FD rule Freedman-Diaconis rule. 31, 82–86, 91, 92

FITing-tree FITing-tree. 9–12, 22

FMCD Fastest Minimum Conflict Degree. 30, 40, 56, 60

IQR Interquantile Range. 32

LIPP Learned Index Precise Position. vii, 3, 14, 15, 22, 23, 25–27, 29–31, 33, 40, 41, 43, 55–58, 63, 69, 82, 91, 98, 101, 102

OLS Ordinary Least Square. 50, 51

RMI Recursive Model Index. 7, 8, 10–12, 14

TBLI Tree-Based Learned Index. 22, 27

Chapter 1

Introduction

The exponential growth of data has presented significant challenges for efficient data storage and retrieval, especially in the field of database management. Traditionally, indexing techniques such as B-trees and hash indexes have been used to organize data and facilitate fast retrieval. However, as data volumes have grown exponentially, these traditional indexing methods are becoming less effective, leading to performance degradation and increased storage costs.

In recent years, machine learning techniques have shown great promise in improving the efficiency and effectiveness of indexing in database management systems. Machine learned index is an emerging area of research that seeks to leverage the power of machine learning algorithms to create more efficient and accurate indexing methods.

The idea behind Learned Index is to train models that can learn the patterns in the data and build indexes that are optimized for the specific characteristics of that data. The models can identify and exploit correlations and dependencies within the data to produce more efficient indexing structures. By doing so, Learned Index can improve query performance and reduce storage costs, making it a promising area of research for the database management community.

The concept of machine Learned Index was first proposed by Kraska et al. in 2018 [14], who demonstrated that machine learned indexes could outperform traditional indexing methods in terms of query response time and storage efficiency. Since then, the field has rapidly evolved, with many researchers exploring different machine learning techniques and applications in indexing.

One of the significant advantages of Learned Index is its ability to handle high-dimensional and sparse data, which traditional indexing methods struggle with. In many real-world applications, data is often high-dimensional, and the number of features can far exceed the number of instances. Machine Learned Index can leverage the power of machine learning to create indexes that are optimized for such data, improving query performance and reducing storage costs. Another advantage of machine learned index is its ability to adapt to changing data distributions. Traditional indexing methods rely on pre-defined data structures that are optimized for specific data distributions. However, as data changes over time, these structures may no longer be optimal, leading to degraded performance. Machine learned index can adapt to changing data distributions by continually retraining the models, resulting in more effective and efficient indexing structures.

However, the drawback of the machine learning model is that it could miss predicting the location of the index. Therefore, the Learned Index will have to perform binary search on the look-up key within a bounded range, and the min and max error bounded range will have to be kept to support binary search within these ranges. Furthermore, the original Learned Index does not support updates. *i.e.*, insertions and deletions, which is crucial in real-world systems that contain read and write operations.

In addition to the challenges of insertion performance in Learned Index, there are also other issues that need to be addressed. For example, traditional B^+-tree indexes often rely on cache and memory optimization techniques to improve their performance. However, machine learned indexes have different requirements, and may need to consider other factors, such as the complexity of the machine learning model.

Moreover, machine learned indexes can be sensitive to changes in the underlying data distribution, which can affect the accuracy of the index. This is particularly relevant for datasets that have a high degree of variability or are subject to significant changes over time. As a result, machine learned indexes need to be carefully designed to be robust to changes in the data distribution, and may require retraining or updating periodically to maintain their accuracy.

The base Learned Index studies have open ways to many research areas such as optimising read learned index, immutable learned index on one-dimensional data [4, 6, 26], and multi-dimensional read-only learned index [5, 19].

The recent studies proposed a new Learned Index that supports read and write operations [4, 6, 7, 26]. These studies introduce a new concept to leave gaps in the array for subsequent insertion. These studies use model-based insert, a machine learning model, to insert items in the array. The main difference is how they handle conflicting elements when the model tries to insert an item in the location where the index could already occupy that space in the array. When there is a conflicting element, Adaptive Learned Index (ALEX) will shift the existing elements to the closest gap to make space for the new index. Learned Index Precise Position (LIPP) index creates a new leaf node when there are conflicting elements that could grow the height.

Furthermore, LIPP introduces ways to prune these leaf nodes to reduce the height of a tree, which will also reduce the cost of query operation. However, even though they provide strategies to improve query time, they could still suffer from poor insertion performance. These poor performances are brought from extra computation required to perform when conflict elements occur during insertion operation.

Nevertheless, these indexes that support updates operations aim to create a fast insertion and handle conflict when necessary, but it does not try to leave an empty spaces in the array strategically to reduce the number of conflicts which in turn could lead to a better search or insertion performance[4, 6, 7, 26].

To reduce extra computation incurred by conflicts, we propose a way to insert gaps strategically so that we save costs on the subsequent few insertions without having to do the extra computation, such as creating new nodes or shifting keys. Furthermore, it provides an efficient way to insert gaps between the existing data in the data structure so that the next insertion operation would not cause a conflict.

The main objective of this research is to optimize the performance of dynamic learned indexes, such as LIPP, and ALEX , on one-dimensional data by strategically leaving gaps for inserting new keys. In addition to this, the proposed algorithm at the end of this research aims to reduce the computation required to expand nodes in a tree-based learned index. Expansion occurs when the gapped array reaches full capacity. Our research will also focus on optimizing the gapped array to reduce the number of conflicts that occur during insertion, thereby reducing the computation required to shift or create a new child node. By achieving these objectives, we hope to improve the overall

efficiency and performance of mutable learned indexes, making them more viable for use in database management systems.

Chapter 2

Related Work

2.1 B$^+$-tree Indexes

The B$^+$-tree is a popular range index that has been adopted by SQL databases such as MYSQL and PostgreSQL [11]. Its structure is made up of internal and leaf nodes, with the internal nodes containing only indexes, and the leaf nodes containing all of the indexes [2]. Leaf nodes are linked together in a sequence set from left to right, which makes it possible to traverse the nodes in a specific order to retrieve data quickly and efficiently [2].

One of the key advantages of the B$^+$-tree is its ability to maintain height balance even when new data is inserted. This means that the depth of the left subtree is equal to that of the right subtree, ensuring that the search operation is efficient and the overall performance of the index is optimized [2].

In addition to its height-balancing capabilities, the B$^+$-tree is a dynamic data structure that can handle both read and write workloads. This makes it suitable for use in a variety of scenarios, including in-memory and on-disk storage [2]. Its versatility and reliability have made it a popular choice for developers and database administrators alike.

To search within a B$^+$-tree, the operation involves traversing to the leaf node and then performing a binary search within the node [2]. This process allows for fast and efficient data retrieval, which is essential for applications that require real-time data processing.

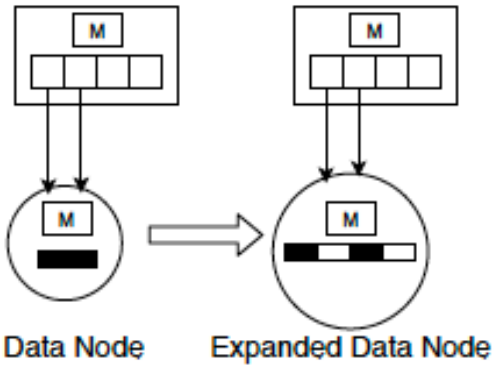

FIGURE 2.1: Adaptive Learned Index node's expansion machanism.

Researchers have developed various methods to optimize the performance of B^+-tree indexing, including taking advantage of hardware features such as CPU cache [22], multi-core processors [24], and SIMD [12]. These optimizations can significantly enhance the performance of the B^+-tree in specific use cases.

Despite its general-purpose nature, the B^+-tree can still be further optimized by taking into account existing data. Researchers have proposed machine learning-based techniques, such as learned indexes, that can leverage the statistical properties of the data to improve the performance of the B^+-tree in edge cases [14]. By using these techniques, developers and database administrators can improve the performance of the B^+-tree and optimize its use in specific applications.

2.2 Tree-based Learned Index

2.2.1 Original Paper on Learned Index

The Learned Index, a type of learned index structure, was first introduced in the paper titled "The Case for Learned Index Structures" by Tim Kraska [14]. Kraska's proposed method uses machine learning models to structure indexes and demonstrated its feasibility by creating a neural network using Python and TensorFlow. However, the study found that the neural network took $80,000ns$ to execute the model, while a standard B-tree only required $300ns$ for traversal and $900ns$ for the binary search. This discrepancy in execution time highlights the need for further optimization of the Learned Index structure.

Kraska identified three main factors that impact the query performance of the Learned Index. The first is the TensorFlow invocation overhead, which can be reduced through more efficient TensorFlow implementation. The second factor is the challenge of fitting cumulative distribution functions Cumulative Distribution Function (CDF) in neural networks as the zoom-out version may look smooth, but if looked closely, there may be more irregularities in the CDF. Lastly, the third factor is the high cost of executing complex neural networks, which can be mitigated by using the right techniques.

To address the issue of overhead, Kraska introduced the Learning Index Framework (LIF), which is implemented in C++. LIF allows for the extraction of the neural network's weights and hyperparameters to avoid the framework overhead, thereby reducing the execution time. Additionally, to address the accuracy problem on non-smooth CDFs, Kraska introduced the Recursive Model Index (RMI). The RMI is a hierarchical structure of models, where each model takes the key as input and predicts the next model to pick until the final stage, where it predicts the key's position. By using a hierarchical structure of models, RMI can effectively reduce the number of parameters needed to achieve high accuracy.

Overall, Kraska's study and subsequent research into the Learned Index have demonstrated the potential for machine learning models to improve the performance of index structures. The use of LIF and RMI has shown promising results in reducing overhead and improving accuracy, respectively. Further research in this area could lead to the development of even more efficient and accurate learned index structures, paving the way for exciting new developments in the field of database management.

2.2.2 Static Learned Index

RMI is a static Learned Index. It forms a hierachy of models such that the underlying models more accurate representation of the CDF. Experiments on RMI shows that it outperforms the B-trees in the index size and the query speed.

However, one limitation of RMI, however, is that it only supports read operations. In modern systems, write operations are equally important, and optimizing the performance of both read and write operations is critical for achieving high database performance

[16]. Therefore, while the RMI may offer benefits for read-optimized workloads, it may not be suitable for all applications.

Handling the updates in RMI is challenging due to the data structure used and the operation is computationally expensive therefore degrading the performnance of the system. The RMI can perform updates on keys but it has to shift the keys to the right to make space for the new key and also if the position of the keys are slightly modified, it will degrades the model's performance [9]. Most of Learned Index require a search within the error bound α to get the exact key position. For example, if p is the position predicted by the machine learning model, then the actual position of the key is within $[p - \alpha, p + \alpha]$. Hence using a binary search with the bounded error, α is guaranteed to get the actual position. Furthermore, re-training is required if there are too many shifts in the element's position or CDF changes, which is slow when there is a large amount of data in the data structure [4].

Hybrid Learned Index (HLI) is another notable static learned index that offers significant performance benefits over traditional indexing structures. HLI is called hybrid because it combines two modules: the Piecewise Linear Approximation Model (PLA) with an error bound and the Rank-Model Index (RMI) without an error bound. The RMI is used to index the PLA-model, with the inner node structure being similar to RMI and ALEX. The leaf node in HLI is the PLA-model with an error bound. The main idea behind HLI is to use RMI to predict the segment that the keys belong to, then traverse until reaching the leaf node, which uses the PLA-model to predict the position.

One significant advantage of HLI is that it outperforms traditional indexing structures. In particular, it achieves up to 4.4× the lookup performance of a B+tree. This performance boost is due to the use of a linear regression model as routing and the prediction only containing multiplication, which is bounded to $O(1)$. HLI also outperforms other learned indexes, such as RMI and PGM index, by about 50%.

However, there are some limitations to HLI. One significant drawback is that it does not support update operations such as insertion and deletion. This is because the PLA-model with error bound used in HLI is a static structure that cannot be updated. As a result, HLI is only suitable for scenarios where the data is static and not subject to frequent updates.

In summary, HLI is a static learned index that combines the RMI and PLA-model to achieve significant performance benefits over traditional indexing structures. The use of a linear regression model as routing and the prediction only containing multiplication, which is bounded to $O(1)$, makes HLI an efficient indexing structure. However, the lack of support for update operations limits the applicability of HLI to scenarios where the data is static and not subject to frequent updates.

2.2.3 Hybrid Indexes

In addition to the static learned index, Kraska also introduced the concept of hybrid indexes in his paper [14]. Hybrid indexes are a combination of the learned index and traditional index structures. By using the B-tree as the main index and incorporating the learned index, hybrid indexes aim to leverage the strengths of both methods. This combination can significantly improve the overall operation performance, especially for queries.

The learned index is used to predict the next child node such that it can jump to the next node directly, which improves the traversal process. However, if the distribution of the keys is difficult to learn, then the hybrid index falls back to using the traditional B-tree index. This hybrid approach provides a balance between the accuracy and performance of the index structure.

Moreover, hybrid indexes can benefit from the fact that the learned index can capture the patterns and dependencies among the keys, while the traditional index is efficient in handling updates. This combination provides a robust and flexible approach that can be adapted to different data distributions and scenarios.

The most notable hybrid indexes is called FITing-tree [7]. The FIT-tree uses B+tree as its index structure. What makes it a learned index is that its leaf nodes contain the linear function to predict the position of the keys. When building the index, FITing-tree divides the keys into several segments, which then it will generate the linear function for each of the segment, which is stored in a B+tree. The key difference between FITing-tree and B+tree is that FITing-tree leaf nodes contain a predictor function while B+tree contains a key. When performing a query, B+tree will have to traverse down until the leaf node finds the particular key. On the other hand, FITing-tree's leaf node does

not contain keys, as it only contains the linear predictor function that represents each segment of the keys. To query in FITing-tree, first, it has to traverse down until the leaf node that represents a particular segment that the key located in. Secondly, it will have to perform a search for the key in the segment within the error bound $pred\_pos \pm err$. The error bound is defined difference between the predicted position and true position of a particular [7].

In FITing-tree, keys are segmented which makes the process of segmenting very important. The algorithm FITing-tree uses is called "ShrinkingCone". It first forms a cone by defining an origin point, high slope (sl_{high}) and low slope (sl_{low}). The high slope and low slope represent the upper and lower bound of the segment. First the algorithm define $sl_{high} = \infty$ and $sl_{low} = 0$. Then if the key is inside the cone, then the high slope and low slope will be updated by using the key and key's position $\pm$ the error. In addition, the ShinkingCone is a non-optimal algorithm as it does not guarantee that it can produce the most optimal segment. However, there is still an upper bound on the maximum number of segments that the algorithm can produce $min(\frac{|keys|}{2}, \frac{|D|}{error+1})$ and $|D|$ is the number of datasets. This ensures that the amount FITing-tree will not produce more segments than the B+tree that uses a fixed size pages[7].

From the experiment result [7], the FITing-tree shows an improvement over index with fixed-size paging in query performance while also memory efficient when compared to full and fixed-size paging index. Eventhough FITing-tree outperforms traditional indexes in lookup and space efficient, there is still some disadvantages of the FITing-tree. Firstly, the performance compared to the other learned index like previously mentioned RMI. FITing-tree performance gained over B-tree is about 1.5× while RMI achieve almost 2× the performance of query when compared to B-tree. Secondly, the insertion performance on FITing-tree does not outperform the full B-tree index in the insertion experiment [7] as FITing-tree requires to perform splitting pages periodically and also performs a segmentation algorithm.

Apart from FITing-tree, the Interpolation-Friendly B-tree (IFB-tree) is also another hybrid indexes [10]. Similar to FITing-tree the IFB-tree also uses B-tree with the node interpolation. The main idea is that node interpolation is used to estimate the range that the keys located in. For example, if the look up key is s, then the interpolated

index of s is $\lfloor \frac{s-v_i}{v_{i+1}-v_i} \cdot n \rfloor$ where v_i is the keys before s, v_{i+1} is the key after s, and n is the node size.

The main advantage of IFB-tree is that it does not consume any extra memory when compared to FITing-tree. FITing-tree requires storing parameters like slope and intercept for the linear regression function. The IFB-tree structure is B-tree which makes it light memory usage when compared to other hybrid indexes.

However, the main drawback of the IFB-tree is similar to the FITing-tree that its query performs does outperforms a traditional B-tree but does not outperforms Learned Index like RMI. Based on the result shown by Hadian [10], the result is about up to 1.5× the query performance of traditional B-tree, which is similar while the RMI is almost $2-3\times$ faster on lookup performance compared to the traditional B-tree. However, if the result is compared to FITing-tree, there is barely any difference in term of lookup performance. The only benefit is that consumes lesser memory than the FITing-tree.

2.2.4 Dynamic Learned Index

The Adaptive Learned Index (ALEX) is a dynamic learned index structure proposed by Ding [4] that is designed to support both read and write operations such as insertion and deletion. In terms of structure, ALEX is similar to the RMI as it also has a hierarchy of models. The internal nodes only contain the linear regression model that points to the next child model to predict the partition of the keys. Meanwhile, the leaf nodes contain the keys and records in an array format. However, ALEX uses a special data structure called the Gapped Array to store its keys and records in the leaf node.

The Gapped Array is a modified version of an array that leaves gaps between the keys instead of placing the empty spaces at the end of the array. This data structure enables ALEX to use model-based insertion, where a model is used to predict the location of the new key, and insert it accordingly.

In addition to model-based insertion, ALEX also employs node expansion and retraining to maintain its accuracy. To expand a node, the node must be 80% full before performing the expansion. This criterion is checked when a new key is inserted. Furthermore, ALEX performs shifting when a key already exists in the predicted position. It can shift to the right or left, depending on the closest gap.

However, if the distribution of new keys does not follow the existing distribution, the linear regression model becomes inaccurate, leading to a more densely packed region. This can deteriorate the shifting performance, which can take $O(n)$ in the worst case. To solve this issue, ALEX requires expanding the data node or retraining the model.

One of the advantages of ALEX is its ability to support update operations, which is crucial in real-world systems. This advantage addresses the shortcomings of static learned indexes by using the Gapped Array and model-based insertion.

Furthermore, the use of gaps in the Gapped Array ensures that the average cost of insertion and shifting elements is bounded to $O(\log m)$ instead of $O(m)$ if ALEX has to shift keys to the closest gap.

Regarding the query operation, lookup in ALEX consists of two stages. First, it traverses down until it reaches the leaf node, which costs $O(\log_m P)$. Secondly, it performs an exponential search as the keys may have shifted around, which costs $O(\log m)$. Since keys can be in any position within the error bound, an exponential search is required. Therefore, the total cost of a query becomes $O(\log_m P + \log m)$.

Overall, ALEX is a dynamic learned index that supports both read and write operations with the use of the Gapped Array and model-based insertion. It also employs node expansion and retraining to maintain its accuracy, and its lookup operation has a worst-case time complexity of $O(\log_m P + \log m)$.

Aside from the ALEX learned index, the Piecewise Geometric Model index (PGM index) is another dynamic index that aims to support read and write operations [6]. The PGM index uses the Piecewise Linear Approximation Model, which segments the keys into multiple linear segments instead of using a machine learning model. This approach is different from FITing-tree, which stores keys in many nodes. However, the PGM index's structure is similar to ALEX, with the main difference being in its model.

One of the advantages of PGM index is that it outperforms FITing-tree in terms of operational performance and memory consumption by up to 75%. It also dominates RMI in terms of lookup and space consumption, and it even outperforms the B+tree in terms of space consumption. Moreover, the PGM index has a guaranteed worst-case bound, which is a significant advantage over the static learned index RMI, as the latter lacks a guaranteed bound. The worst-case time complexity is $O(\log m_{opt} + \log \epsilon)$, and

the space complexity is $\Theta(m_{opt})$. Here, m_{opt} refers to the number of ϵ-approximate segments. This makes the PGM index a superior dynamic learned index, as it can adapt based on the keys distribution, which may vary during lookup.

In addition, since the lookup operation is performed frequently in real-world systems, the PGM index's ability to adapt to different distributions during lookup makes it a very notable dynamic learned index. Furthermore, the PGM index also provides a smooth trade-off between operation performance and space consumption, allowing it to scale better in large datasets. In conclusion, the PGM index is a promising dynamic learned index that offers various advantages over other traditional and learned indexes. Future research can explore the PGM index's potential for further optimization and use in various real-world applications.

RadixSpline is a dynamic learned index introduced by Kipf that uses spline interpolation and a radix table to perform efficient lookup [13]. The index consists of two main components: a set of spline points and a radix table. The set of spline points is used to perform spline interpolation for any lookup key, resulting in a predicted position in the array. The radix table is then used to locate the accurate spline point for the queried key.

One of the advantages of RadixSpline is that it only has two hyperparameters to maintain: the number of radix bits r and the spline error e. The r parameter determines the size of the radix table and can be adjusted to balance accuracy and memory usage. A larger value of r will increase the size of the radix table exponentially 2^r, but it will also increase its accuracy.

Another advantage of RadixSpline is that it performs on par with traditional indexes in terms of build time. It is also competitive in terms of lookup performance when compared to other learned indexes. The index is built from the bottom up, constructing the error-bound spline on the keys and then indexing the spline points in the radix table.

However, RadixSpline has a significant drawback in that the radix table size r must be tuned. This can be a challenge as the amount of data grows and the data distribution changes. If the value of r is not carefully chosen, lookup performance can suffer.

Like other learned indexes, RadixSpline provides a good tradeoff between space and time complexity. It requires less space than traditional indexes while achieving comparable

lookup performance. The use of spline interpolation also allows it to accurately predict the location of a key in the array, even if the key is not in the set of keys used to construct the index.

Learned Index Precise Position (LIPP) introduced by Jiacheng Wu to support update operations. The primary aim of LIPP is to predict the exact position of a key and eliminate any extra local search required to perform when the model miss predicts the location of the key [26]. The structure of nodes in LIPP is different from other dynamic learned indexes such as the Adaptive Learned Index (ALEX). Unlike the ALEX, which has internal nodes and leaf nodes, LIPP only contains the data node, which consists of a gap, a pointer to the child node or a key.

One similarity between LIPP and ALEX is that they both use Gapped Arrays in the index structure to reserve gaps for new key insertion. However, during the insertion operation, the main difference is that ALEX shifts keys to the closest gaps available, while LIPP creates a new node and replaces the location as a pointer to the new node. The new node will have a gapped array with just two elements. However, the creation of a new node could grow the tree height, making the lookup operation time complexity to be bounded by $O(h)$, where h is the height of the tree.

To solve this problem, LIPP has to perform adjustments (branch pruning) method that helps to reduce the amount of tree height such that it is bounded by the $O(\log N)$. It compares the number of nodes from the previous adjustments, and if it is $2\times$ more nodes than the previous adjustments, then LIPP will trigger the adjustments. Even though the lookup is still bounded by the height of the tree, the adjustment causes the lookup to have a shorter traversal path.

One of the main advantages of the LIPP index is that it outperforms both PGM index and ALEX in read and write workloads by $13\times$ and $2.9\times$, respectively. Furthermore, LIPP also outperforms traditional indexes in the read and write operation as well. For the read-only workload, LIPP still outperforms ALEX, PGM index, and RMI. In addition, LIPP eliminates the last mile search completely by using the model to predict the precise position of the queried key.

However, ALEX performs better in terms of memory consumption, as it uses gaps to shift the elements, while LIPP consumes extra memory when there is a conflict, as it has

to hold two conflicting keys in the leaf nodes. Nonetheless, the LIPP index's benefits make it a promising solution for applications that require both high read and write performance.

In summary, LIPP is a dynamically learned index that uses a unique node structure and adjustments method to provide high read and write performance while eliminating the last mile search completely. Its performance in read and write workloads surpasses that of traditional indexes and other dynamic learned indexes such as ALEX and PGM index, making it a promising solution for modern data-driven applications.

Dynamically learned indexes are becoming increasingly popular among researchers. However, these indexes require the presence of empty space, known as gaps, to facilitate the insertion of new elements. This presents a significant challenge in optimizing the algorithms that learn where to place the gaps to improve the efficiency of search and range queries while preserving the sorted order of the data to ensure the correctness of the range query.

However, the success of these methods depends on how well they learn to place the gaps. Poorly placed gaps can negatively impact the search and range query performance, leading to decreased efficiency and degraded accuracy. Therefore, there is a need for machine learning models or algorithms that can effectively learn where to place the gaps to optimize these indexes.

2.3 Learned Hash Index

Hash-table is a fundamental data structure used in many computer science applications, including database management systems. Its primary purpose is to provide an efficient way of looking up data by using a hash function to map keys to positions in an array. However, when two or more keys are mapped to the same position, a conflict occurs, and the hash-table needs to handle it. This is usually done by using a linked list to store all conflicting elements. While Hash-tables are efficient in lookup time, with an average lookup cost of $O(1)$, the worst-case lookup time can be as high as $O(n)$, where n is the number of elements in the linked list.

To reduce the number of conflicts, various techniques have been developed, such as secondary probing or using multiple hash functions, like Cuckoo Hashing. While these techniques reduce the number of conflicts, conflicts can still impact lookup performance and memory consumption.

Kraska introduced the Hash-Model Index, which takes advantage of the underlying data distribution [14]. This index uses a machine learning model to learn the cumulative distribution function (CDF) of the data, providing a better hash function that reduces the number of conflicts by up to 77.5%. Unlike tree-based learned indexes, Hash-Model Indexes do not need to maintain sorted order, making them more suitable for supporting updates. When new items are inserted, they are stored in the array or a linked list if there is a conflict.

Hash-Model Indexes provide significant benefits over traditional Hash-tables, improving both lookup performance and memory consumption. Additionally, they provide a scalable solution that can handle large datasets efficiently. However, Hash-Model Indexes require more resources and computational power than traditional Hash tables, making them less suitable for applications with limited resources or computational power.

2.4 Multi-Dimensional Learned Index

Scanning and filtering are two typical usages in a database engine [19]. To create an order, most database systems use a B^+-tree index on a single attribute. However, when multiple attributes are filtered, using a B^+-tree may lead to latency as the search has to follow pointers and narrow down the search space. To overcome this issue, multi-dimensional indexes with tree-based data structures such as R-trees [8] or k-d trees [27] can be used to improve the scanning performance of queries over multiple columns.

R-trees are a data structure that is commonly used for spatial data [8]. They support multi-dimensional indexing for objects such as geographical coordinates, rectangles, and polygons. R-trees group nearby objects and represent them using rectangular bounding boxes. This helps in reducing the search space during querying, which results in faster query execution times. The R-tree data structure is shown in Figure 2.2

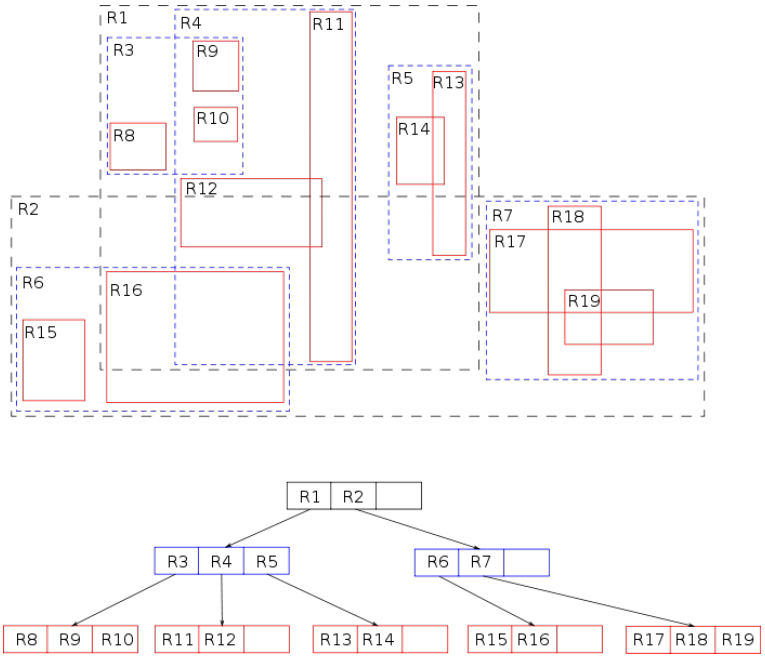

FIGURE 2.2: R-trees data structure and R-trees' bounding boxes

In contrast, k-d trees divide the data into subregions using hyperplanes to group nearby points [27]. This approach is commonly used for nearest neighbor search, where the query searches for the data point closest to the given point.

Using multi-dimensional indexes with tree-based data structures provides an efficient solution for querying databases with multiple attributes. These data structures enable faster and more efficient searching by reducing the search space and optimizing the query execution time. As such, they are widely used in database engines for scanning and filtering operations.

Despite the advantages of using multi-dimensional indexes with tree-based data structures, such as R-trees [8] and k-d trees [27], there are still some limitations that need to be addressed. One such limitation is the difficulty in tuning these indexes, which requires significant engineering efforts. Tuning the index for better performance in one scenario may not necessarily lead to better performance in another scenario [19]. Additionally, the indexes cannot adapt to specific workloads and data distributions, which may result in suboptimal performance.

In real-world scenarios, data distribution and query load can vary significantly, making it challenging to design an index that can adapt to changing conditions. Although traditional multi-dimensional indexes like R-trees and k-d trees are commonly used,

they have their limitations. In particular, tuning these indexes for optimal performance can be a time-consuming and difficult task that requires significant engineering effort. Moreover, even when these indexes are properly tuned, their performance may not always be optimal, especially when dealing with skewed data distributions or heavy query workloads.

To address these issues, Flood index was developed as an in-memory learned multi-dimensional index that overcomes some of the limitations of traditional indexes [19]. Flood index is based on the concept of Cumulative Distribution Function (CDF), which is used to learn the data distribution and project the multi-dimensional and skewed data into a uniform space. By doing so, the model can predict the data's location and provide fast access to it.

One of the main advantages of Flood index is that it outperforms traditional multi-dimensional indexes like R-trees and k-d trees by eliminating the need for search traversal times. Instead, Flood index relies on machine learning models to predict the location of the data. However, it should be noted that Flood index is designed to work only with read-only workloads. To support the insertion of new data, Flood index can leave gaps similar to ALEX or B^+-tree.

Despite its usefulness, Flood suffers from two major drawbacks: it is not well-equipped to handle a query workload that is skewed [5], and it does not maintain uniformly sized grid cells. To address these issues, the Tsunami was developed as an optimized extension of the Flood index [5]. However, the problem of efficient key insertion persists. One potential solution to this problem is to use gap reservation, which can aid in subsequent key insertions [19].

2.5 Learned Bloom Filters

The Existence index, utilizing a Bloom filter, is one of the most widely used types of indexes in storage systems, databases, and other applications where the fast and efficient determination of set membership is essential. A Bloom filter is a probabilistic data structure that allows for checking if a key is present in a set with an extremely low false negative rate. It does this by utilizing a bit array of size m and k hash functions to map keys to m array positions.

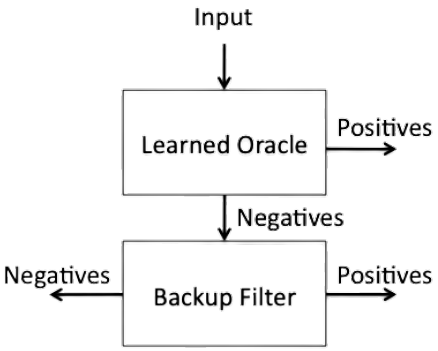

FIGURE 2.3: Learned oracle with backup Bloom filter (backup filter).

To add an element to the set, the key is inserted into the k hash functions, which returns k different positions in the m array. The bits in these returned positions are then set to 1, indicating the existence of the corresponding key in the set. During the insertion of a new key, the key is placed into the k hash functions, and the resulting positions in the m array bits are checked to determine if they are all 1, indicating the presence of the key in the set.

One of the primary advantages of using a Bloom filter is its speed and efficiency in checking for set membership. For instance, Postgres, a popular open-source relational database management system, utilizes a Bloom filter index that allows for the fast exclusion of non-matching tuples [21]. Bloom filters are also space-efficient as they only require a fixed amount of memory for a given false positive rate.

However, Bloom filters do have a potential drawback, namely the possibility of false positives. A false positive occurs when the filter returns that a key is in the set when it is not. This happens when the hash functions of two different keys result in the same bit positions in the bit array, causing the filter to erroneously report that the key is present. The probability of a false positive is dependent on the size of the bit array m, the number of hash functions k, and the number of keys inserted into the Bloom filter.

Bloom filters have been found to be highly effective in many applications that require fast and efficient membership checking. However, there are some limitations to traditional Bloom filters, such as their potential for false positives, as discussed earlier. One

potential solution to this problem is to augment Bloom filters with machine learning models, as proposed in the Learned Bloom filter [17].

The Learned Bloom filter is a probabilistic data structure that uses machine learning to make predictions about set membership. However, unlike traditional Bloom filters, the Learned Bloom filter does not provide a zero false negative rate. In other words, there is a chance that the machine learning model will not predict the correct value for a key. To overcome this limitation, the Learned Bloom filter requires a backup Bloom filter that can validate any false predictions made by the machine learning model (as shown in Figure 2.3).

It is worth noting that the backup Bloom filter only needs to contain at most $FNR \times n$ elements, where FNR is the false negative rate of the Learned Bloom filter and n is the number of items in the set [17]. This means that if the backup filter contains all the values similar to the traditional filter, it will take up more memory space as it needs to maintain both the backup filter and the Learned Bloom filter. By using this strategy, Bloom filters can potentially save memory space by storing only n bits of elements.

In addition to the Learned Bloom filter, another approach to reducing the false positive rate of Bloom filters is the Sandwich Bloom filter. The Sandwich Bloom filter consists of two layers of Bloom filters, with the Learned Bloom filter sandwiched between them [18].

The first layer of the Sandwich Bloom filter is the pre-filter, which is a traditional Bloom filter used to eliminate as many false positives as possible before the query reaches the Learned Bloom filter. The pre-filter reduces the workload of the Learned Bloom filter and helps to improve its accuracy. The pre-filter uses a set of independent hash functions to map keys to positions in the bit array, and each position is set to 1. When a key is queried, the pre-filter checks whether all of the corresponding positions in the bit array are set to 1. If any position is not set to 1, the query immediately returns false.

The second layer of the Sandwich Bloom filter is the backup filter, which is a traditional Bloom filter that is used to eliminate false negatives introduced by the Learned Bloom filter. As with the Learned Bloom filter, the backup filter only needs to store at most $FNR \times n$ elements, where FNR is the false negative rate of the Learned Bloom filter and n is the number of items in the set.

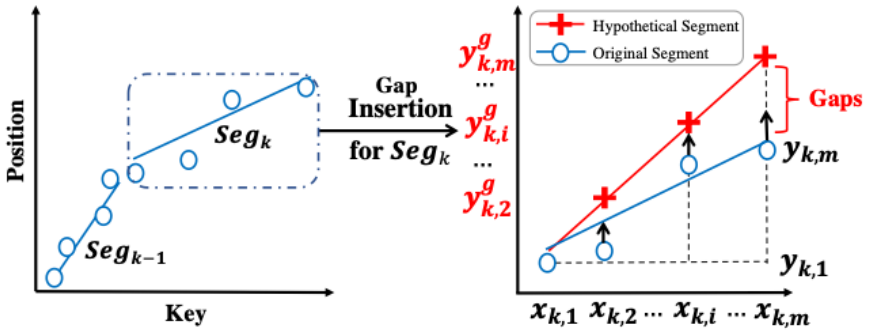

FIGURE 2.4: Gaps Insertion [15]

By sandwiching the Learned Bloom filter between the pre-filter and the backup filter[18], the Sandwich Bloom filter can provide high accuracy while minimizing false positives and false negatives. However, like the Learned Bloom filter, the Sandwich Bloom filter is not perfect and may still produce errors. It is important to carefully evaluate the specific requirements of the application before choosing to use a Sandwich Bloom filter.

2.6 Data Structure with Gaps

The concept of utilizing empty space, also known as gaps, for efficient insertion and deletion operations while maintaining the sorted order of keys is a common approach in various data structures, including the Learned Index [4–7, 15, 19, 26]. The initial introduction of gaps in data structures can be traced back to ALEX, where the Gapped Array technique was introduced. However, since then, several other data structures have utilized the concept of gaps, indicating its importance in designing efficient data structures.

For instance, the ubiquitous B+tree data structure, introduced by Comer in 1979, uses gaps at the end of its leaf nodes to allow for subsequent key insertion without significant overhead [2]. Another example of gap utilization is in the Packed Memory Array, a data structure designed for efficient searching and insertion of elements in compressed arrays, which also employs the use of gaps [1]. Similarly, the Packed Memory Quadtree, a variant of the Packed Memory Array designed for two-dimensional spatial data, uses gaps to maintain a balanced tree structure and ensure efficient insertion and deletion operations [25].

Another data structure, the FITing-tree, employs gaps at the end of its array to accommodate new key insertions and deletion operations while still guaranteeing a bounded error in query results [7]. The FITing-tree shifts keys either left or right, depending on the closest gap, to maintain a balanced tree structure while still achieving optimal query performance.

In addition to reserving space for new data, gapping can also be used strategically to reduce conflicts and improve the efficiency of data structures. For instance, ALEX places gaps in a way that reduces the number of conflicts between keys, which in turn minimizes the number of shifts required during insertion. This approach helps to reduce the worst-case shifting time to $O(m)$, where m is the number of keys in the gapped array [4].

Similarly, the LIPP tree [26] uses gaps to create new child nodes, which can improve the efficiency of the data structure. The PGM-index also uses gaps to reduce the size of its index and improve query times [6]. The Tsunami index extends the Flood index by using gaps to maintain uniformly sized grid cells, which in turn improves the query performance [5].

Therefore, gapping is a widely used technique in various data structures to ensure efficient insertion and deletion operations while maintaining sorted keys.

2.7 A Brief Summary

The mutable Tree-Based Learned Index (TBLI) have been shown to be effective in supporting insert operations while maintaining sorted keys by utilizing gaps. However, as we have observed from previous studies, conflicts arise when space is already occupied, leading to additional computation during insertion operations. To address this challenge, it is crucial to strategically place gaps in the array based on the available information, such as the distribution of existing data. In this project, we aim to minimize the number of conflicts in the LIPP TBLI by leveraging the information about the distribution of existing data, thereby optimizing the insertion overhead caused by conflicts. By strategically placing gaps, we aim to reduce the number of shifts and ensure that the model's error bound is still satisfied, leading to improved performance of TBLIs.

Chapter 3

Preliminary

The purpose of this section is to provide an introduction to the concepts and background of our research. By doing so, we hope to provide readers with a better understanding of the context in which our work is situated, and make it easier for them to grasp the key ideas and contributions of our research.

To achieve this goal, we provide an overview of the field of learned indexes and its relevance to data access and manipulation operations and the model used in Learned Index.

In addition, we provide background information on the LIPP approach, which is the foundation of our research. We discuss the structure of LIPP's node-based index and the components that make it work, such as the gapped array, bitmap index, and linear regression model. We also highlight some of the key features and benefits of LIPP, as well as its limitations.

3.1 Linear Regression

Linear regression is a widely used statistical technique for modeling the relationship between two variables, namely the independent variable, denoted as x, and the dependent variable, denoted as y. The objective of linear regression is to find the best fitting line that represents the relationship between the two variables in the most accurate way possible. The line is represented by the equation $y = mx + c$, where m represents the slope of the line, and c represents the y-intercept.

The main purpose of linear regression is to make predictions based on the relationship between the two variables. In the case of a machine-learned index, linear regression is used to predict the location of the keys in the gapped array. This is done by training a model on a dataset of key and its position. The model learns the relationship between the keys and its location, and can then be used to predict the key's location for new queries.

The process of finding the best fitting line involves minimizing the sum of the squared differences between the actual y-values and the predicted y-values for each data point. This method is known as the method of least squares, and it ensures that the line fits the data points as closely as possible.

One of the reasons why linear regression is used in machine-learned indexes is its simplicity and interpretability. The linear regression model can be easily understood and implemented, and the output can be used to build an index that can efficiently look up rows in the table. Moreover, linear regression can handle large datasets, making it an ideal choice for machine-learned indexes.

In contrast, neural network models can be more complex and computationally expensive, which may cause a large overhead in operations [14]. Therefore, linear regression is a more efficient and practical choice for machine-learned indexes. However, the choice of machine learning algorithm depends on the specific requirements and characteristics of the dataset.

3.2 Model-Based Insertion

Model-Based Insertion is essentially an insertion strategy that uses the learned model to determine where newly inserted keys should be placed in the gapped array, previously mentioned in previous section as an array that reserved spaces in between the keys for new insertion, that constitutes the learned index. The algorithm uses the model to predict the location of the new key, and then places it accordingly to ensure that the learned index can efficiently support insert operations. This is achieved by mapping the keys to the linear regression slope of the model, ensuring that the newly inserted keys fit seamlessly within the existing index structure (Figure 3.1).

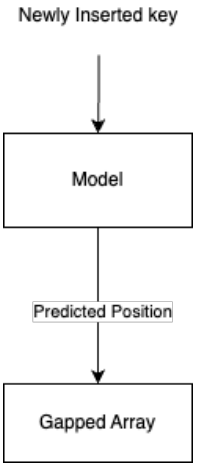

FIGURE 3.1: Model-Based Insertion

Learned Index Precise Position (LIPP) adopts a different approach to handling insertion operations compared to other learned indexes such as ALEX and PGM [6]. Rather than shifting elements or storing them in a temporary buffer to merge later, LIPP employs node creation. This technique involves creating a new node for each new key that is inserted into the index. By creating a new node for each key, LIPP is able to optimize the search operation by ensuring that the output position from the model is exact and precise. This means that LIPP does not need to perform extra searches to locate the key.

However, while LIPP's node creation approach has several benefits, it also has some limitations. One of the primary concerns with this approach is that it could result in the tree height growing, which could negatively impact the performance of the index. The height of the tree is a critical factor in determining the time complexity of the search operation, and an increase in the height of the tree could result in search operations becoming bound to the height of the tree.

3.3 Learned Index Precise Position

LIPP achieves prediction precision by using a node-based structure that includes a gapped array, a bitmap index, and a linear regression model (Figure 3.2). These components work together to enable accurate and efficient key access.

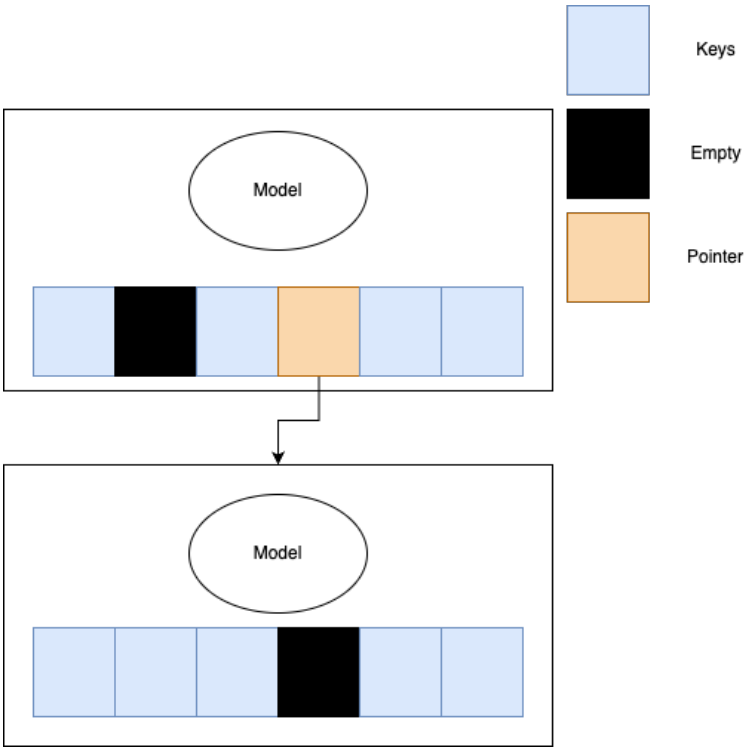

FIGURE 3.2: LIPP node structure

Furthermore, LIPP introduces the adjustments or branch pruning to reduce the height of the tree. The conditions to trigger the adjustments are (1) current depth after previous adjustments should not be more than β times (set to 2 by default) the previous adjustment, and (2) conflict number should not exceed α (set to 0.1 by default) from the previous adjustments [26].

In this research, we aim to build on the foundations of LIPP by developing an algorithm that further enhances its performance. Our algorithm will focus on optimizing performance of LIPP, specifically by improving the gap placement strategy. By doing so, we aim to improve the accuracy and efficiency of key access operations, enabling faster and more reliable data access and manipulation.

Chapter 4

Problem Formulation

Definition 4.1. ("*Conflict*") Consider an insertion of a new key to an array stored in a tree node; if the location where the new key should be put in has been occupied, then a *conflict* occurs.

Definition 4.2. ("*Gaps*") Gaps is reserved in the gapped array in between elements to support new keys insertion.

Where to place gaps such that the Learned Index is efficient to perform operations like insertion, deletion and query ?

The Learned Index Precise Position (LIPP) tree structure consists of nodes that hold a gapped array, which contains data, pointers to child nodes, and gaps. Gaps are particularly important for Tree-Based Learned Index (TBLI) as they enable support for subsequent key insertions and help reduce the number of conflicts and element shifts [4, 6, 7, 26]. TBLI handle conflicts in different ways, with ALEX using the shifting method [4] and LIPP creating a new child node [26]. However, while this method ensures that items remain sorted, it comes with the added computational cost of performing shifts or creating a new node during insertion operations. Shifting requires knowledge of the location of the nearest gap to perform the shift, while creating a new node requires following pointers during range queries.

How can we add gaps to a set of keys with increasing positions $\{x, y\}$ in a way that minimizes conflicts while still maintaining or partially maintaining the monotonicity of the key-position relationship? The number of gaps added should be limited to a maximum

value of M. The distribution of the density of keys in the set, $X_1, X_2, \cdots, X_n \mid X \in \mathbb{Q}$, is represented by $Pr[a \leq X_n \leq b]$. The number of gaps inserted between key X_i and X_{i+1} is m, and the total number of gaps in the gapped array should not exceed M (i.e., $\sum_{X_1}^{X_n} m \leq M$).

One of the main challenges in optimizing the performance of learned indexes is handling dynamic data. When new data is inserted into the data structure, it may follow a different distribution than the existing keys, making it difficult to determine where to insert gaps to minimize conflicts. However, if the newly inserted data follows the same distribution as the existing keys **static**, the problem becomes simpler to handle. In this case, we can focus on identifying places where the key density is highest and insert gaps accordingly. By doing so, we can reduce the number of conflicts and avoid the need to create new nodes.

On the other hand, for **dynamic** data, where the distribution of inserted keys may change over time, it is much harder to tackle the problem of minimizing conflicts. In this case, we need to be able to detect slowly changing distributions and adjust the placement of gaps accordingly. This can be achieved by monitoring the distribution of inserted keys and identifying areas where the density has shifted. Once these areas have been identified, we can insert gaps to reduce the number of conflicts.

Reducing the number of conflicts in each node is crucial to ensuring optimal performance of learned indexes. When conflicts occur, we may need to perform extra computations like creating new nodes, which can lead to a larger tree and poor lookup times. Therefore, by minimizing the number of conflicts, we can avoid these extra computations and keep the tree size manageable. Ultimately, this leads to faster lookup times and more efficient use of resources.

Furthermore, expanding gapped arrays and inserting gaps between keys can be a costly operation in terms of retraining the machine learning models used in the indexes. In order to learn the new positions of the keys after gaps have been inserted, the models need to be retrained. However, the amount of time required to retrain the models can vary depending on the complexity of the model used [14]. For instance, a simple linear regression model is used, the retraining process should be fast, as the model only contains multiplications [4, 14, 26]. However, more complex models, such as neural networks, may take longer to train on larger datasets, making it impractical to retrain

the models frequently [14]. Therefore, finding a balance between the complexity of the model and the time required to retrain it is essential when optimizing dynamic learned indexes.

The current state-of-the-art dynamic Learned Index, LIPP, supports updates by optimizing the keys to fit a linear model. This ensures that the model always predicts the correct position of the keys. However, LIPP creates a new node for each update, which can make operations like insertion and deletion bounded by the height of the tree.

The goal of this thesis is to design a dynamic Learned Index that supports updates and performs better than LIPP in terms of insertion and deletion time. To do this, we optimize the placement of gaps in the learnindex. This allows the Learned Index to adapt to the distribution of keys and place gaps in densely packed areas, which reduce the number of new child nodes that need to be created.

Chapter 5

Methodology

The use of gaps in LIPP trees is crucial for supporting subsequent key insertions and reducing the number of shifts and conflicts that occur during insertion operations. However, the handling of conflicts in LIPP trees differs from other tree structures like ALEX. While ALEX uses the shifting method to handle conflicts, LIPP trees create a new child node. This method guarantees that items remain sorted, but it introduces extra computation cost during insertion operations.

The objective of this study is to explore methods of optimizing mutable tree-based Learned Index by strategically creating gaps for future insertions to minimize conflicts. This approach differs from FMCD [26]. The study investigates two initial implementations: (1) Histogram, and (2) Partially sorted insertion strategy.

Our first implementation, Histogram, involves creating a histogram of the dataset to determine the frequency distribution of the keys. Based on this distribution, our algorithm leaves gaps in the appropriate places to accommodate future insertions, thereby reducing the number of conflicts that occur during insertion.

Our second implementation, Insert strategy partial sorted, focuses on identifying the most suitable location to insert a new key based on available space after the predicted position. The algorithm examines the dataset and creates a partially sorted list, which contains the keys in ascending order up to a certain point, and then the rest of the keys are left unsorted. This approach allows for more efficient insertion by taking advantage of the sorted section of the list while still leaving room for new keys to be inserted without causing conflicts.

5.1 Histogram

Histogram is a sequence of bin and bin is a partition of a range of keys which represents the distribution of the keys in the node. Maintaining a histogram in each node of the LIPP tree structure is a powerful method for optimizing the placement of gaps in the array. LIPP triggers rebuilding of tree nodes when the node's depth is two times. We use histogram to approximate the distribution of the items in the gapped array and insert gaps based on the distribution and then train the model based on the the keys' position.

However, the success of this method depends on selecting an appropriate bin size for the histogram. The challenge is to strike a balance between the overfitting and underfitting of the data in the histogram. Overfitting occurs when the bin size is too small, which leads to too many bins and lack of generalization to unseen data. Underfitting occurs when the bin size is too large, which leads to fewer bins and a loss of detail in the data.

There are several methods for selecting an appropriate bin size for the histogram, including the Freedman-Diaconis rule (FD rule) and Sturges' rule [23]. For our histogram algorithm, we use Freedman-Diaconis rule to attain the bin width for the histogram. Each nodes has its own bin width to reflect on the data in its gapped array and its subtree.

The Freedman-Diaconis rule is a method for determining an appropriate bin width for creating a histogram. Histogram is useful for visualizing the distribution of a dataset. They work by dividing the range of data into a series of bins and counting the number of data points that fall into each bin.

In addition, the bin width is an important parameter in constructing a histogram, as it determines the number of bins and the level of detail displayed in the frequencies of the keys. If the bin width is too small, the histogram may show too much detail and become difficult to read. If the bin width is too large, the histogram may lose important information and become overly simplified.

$$IQR = Q_3 - Q_1 \tag{5.1}$$

The Freedman-Diaconis rule aims to find an appropriate bin width that balances these trade-offs. It is based on the Interquantile Range (IQR) of the data (Equation 5.1), which is a measure of the spread of the data that is less sensitive to outliers than the range or standard deviation. The IQR is calculated by subtracting the value at the 25th percentile (Q_1) from the value at the 75th percentile (Q_3). It represents the range of the middle 50% of the data.

$$BinWidth = \frac{c \times IQR}{\sqrt[3]{n}} \tag{5.2}$$

The rule then calculates an "optimal" bin width as the product of a constant factor (usually set to 1.0) and the IQR, divided by the cube root of the number of data points (Equation 5.2). This formula scales the bin width with the sample size, so that the bin size becomes smaller as the sample size increases. The cube root is used to find a balance between having too few bins and too many bins.

The Freedman-Diaconis rule is particularly useful for datasets that have a skewed or multimodal distribution. These types of datasets can be difficult to visualize accurately with histograms using other methods. The rule adjusts the bin width to account for the spread of the data in the middle 50% of the distribution, rather than just the range of the data.

While our approach of using a histogram to analyze the distribution of collected data and selecting an optimal bin width has several advantages, it is important to note that there are certain limitations to this technique. One of the primary drawbacks is that it may not perform well on real-world datasets where the distribution of the data is random and does not follow a specific pattern or distribution.

In such cases, a histogram-based approach may not be as effective, as the data may be too complex and varied to be analyzed using a simple histogram. This may result in suboptimal bin width selection, which in turn can lead to less efficient adjustments to the Learned Index structure.

5.2 Partial Sorted Array

To address extra computation issue, we propose another method that uses of a partially sorted array in LIPP trees [3]. A partially sorted array allows the algorithm to continue the search for the correct position of a new key forward until it reaches a specified limit. For example, when inserting a key 1.2 into the gapped array $[1, 2, GAP, 3]$, the algorithm would ideally insert the key between one and two to maintain sorting. However, with partial sorting, the algorithm continues searching until it hits the limit $key + \epsilon$, where ϵ is a pre-defined limit. In this case, the algorithm scans forward until it reaches the number that is not more than three.

Partial sorting reduces the number of conflicts during insertion operations, but it does not completely eliminate them. Partially sorted arrays delay conflicts for future insertions, meaning conflicts will eventually occur when the available space runs out. For example, if the array is $[1, 2, 1.2, 3, 4, GAP]$ and the key 2.4 is inserted, the algorithm must search beyond 4 to find the gap. To address this issue, a limit ϵ is introduced to specify the amount of space to scan after the actual position where the insertion is intended. If space is exhausted after $key + \epsilon$, a new child node is created to hold the remaining keys.

When performing a range query on a partially sorted array, the algorithm does not scan from the beginning to the end of the range. Instead, it scans forward until it reaches the limit specified during the insertion operation to ensure the correctness of the range query. The use of partially sorted arrays in LIPP trees aims to amortize the cost of slow insertion, where the algorithm has to scan until it reaches the $key + \epsilon$ limit, with fast insertion where there is still space available.

Chapter 6

Histogram

In this section, we will delve into the implementation details of histograms and explore various operations such as insertion, update, and deletion. Understanding these fundamental aspects will provide you with a comprehensive understanding of how histograms are built and manipulated.

6.1 Implementation Details

6.1.1 The Insertion Algorithm

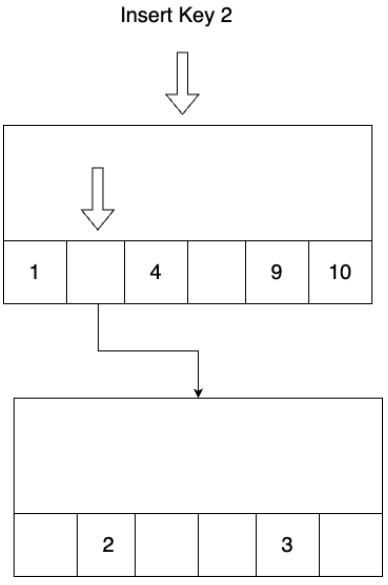

FIGURE 6.1: Example of Histogram Insertion

The strategy we employ for the insertion of histograms is very similar to that of the baseline strategy. However, there are some noteworthy differences in the way we rebuild nodes. Rather than relying on the traditional methods, we use a histogram approach to obtain an approximate distribution of the data. This allows us to generate keys, denoted by x, and their corresponding positions or labels, denoted by y.

To further refine the model, we use Ordinary Least Square (OLS) to obtain the slope and y-intercept for linear regression. This aids in the accurate identification of conflicts within the data. The rebuilding process occurs when a new node has a depth that is at least twice that of the previous node, in a similar fashion to the baseline strategy.

Algorithm 1 Histogram Insertion

```
1:  procedure INSERT(node, key)
2:      current_node ← node
3:      path_size[]
4:      while node ≠ null do
5:          pos ← PREDICT_POS(current_node, key)
6:          if pos is child node then
7:              currentnode ← node[pos]
8:              path_size[i] ← node
9:          else
10:             key_pos = PREDICT_POS(current_node, key)
11:             node[key_pos] ← key
12:             update frequency_array
13:     for i ← 0 to path_size − 1 do
14:         node ← path[i]
15:         if rebuilding criteria then
16:             keys ← collect_keys
17:             new_node ← rebuild_tree
18:             path_size[i] ← new_node
```

In cases where conflicting elements arise, a new node is created to ensure the integrity of the data. However, the creation of new nodes comes with an additional computation, which slows down the insertion time. This is because the model needs to be trained to accommodate these conflicting keys and accurately insert them into the gapped array.

As an example in Figure 6.1, if a key 2 is inserted into an existing tree, the predicted position will be 1. However, 1 is already occupied, so the tree will create a new node that contains $2, 3$ and replace the root node $pos[1]$ as the pointer pointing to the child node.

Overall, the insertion strategy for histograms relies on a combination of techniques such as the use of OLS and histogram analysis to ensure that the data is accurately represented and conflicts are resolved promptly. While it may result in slightly slower insertion times, the benefits of a more accurate representation of the data outweigh the costs.

6.1.2 The Query Algorithm

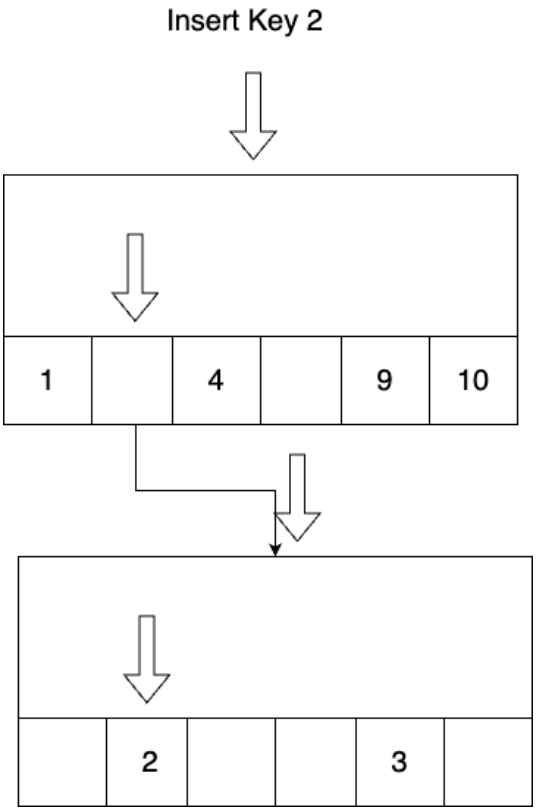

FIGURE 6.2: Example of Histogram Query

To perform a search operation in Learned Index, the algorithm starts at the root node of the tree and traverses down to the leaf nodes to find the data. One of the advantages of Learned Index is that it uses a model-based insertion technique, which allows it to perform search operations efficiently. When there is a conflict during insertion, Learned Index creates a new child node rather than shifting the keys to the closest gaps. This ensures that there are no misplacement in the index, which can slow down search operations by making performing a search after traversing down the tree.

The search algorithm in Learned Index starts by pushing the root node into the stack. Using the current node's linear regression model, the algorithm predicts the position of the keys it is searching for. It then checks if the predicted position is a child or a key. If the predicted position is a child, the algorithm pushes the child node onto the stack and continues the search. If the predicted position is a key, the algorithm returns the key.

Algorithm 2 Histogram Query

```
1:  procedure QUERY(node, key)
2:      current_node ← node
3:      while node ≠ null do
4:          pos ← PREDICT_POS(current_node, key)
5:          if pos is child node then
6:              currentnode ← node[pos]
7:          else
8:              key_pos = PREDICT_POS(current_node, key)
9:              return node[key_pos]
10:     return null
```

The child node continues to be pushed onto the stack until the algorithm reaches the node that contains the key it is searching for. This process allows for efficient search operations, as the algorithm only explores the parts of the tree that contain the key it is searching for.

As an example, in Figure 6.2, if we perform a query after the insertion for a key 2, the model predicts the location of key 2 in location 1 and it has to follow the pointer down to leaf node and use the model at leaf node to predict the location of key 2.

6.1.3 The Deletion Algorithm

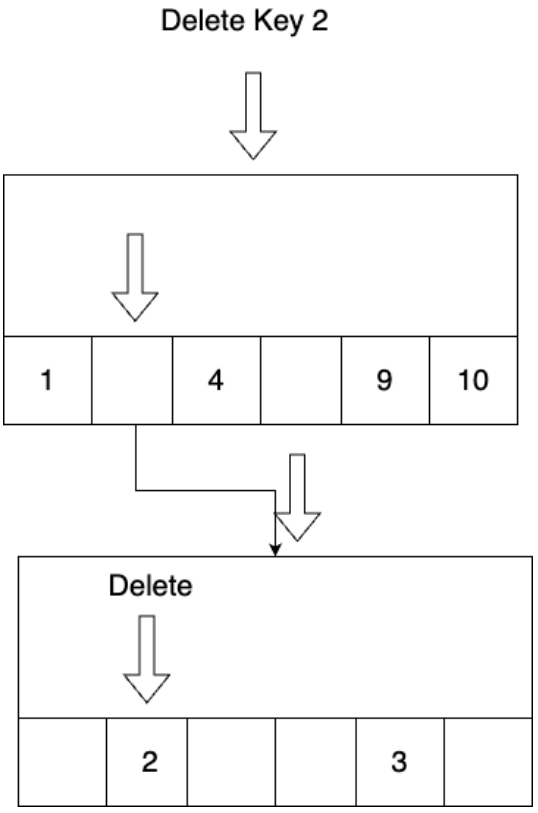

FIGURE 6.3: Example of Histogram Delete

Deletion is an essential operation for any data structure that stores data, and Learned Index is no exception. Deletion in Learned Index follows a similar strategy as search, where the algorithm has to traverse down the tree until reaching the node where the key is located. Once the node is found, the key can be removed from the tree. However, there is a critical difference between deletion and search in Learned Index. During deletion, the algorithm must also free up space in the gapped array, and also update the bitmap index that this position contains a gap instead of a key.

To perform the deletion, the algorithm starts at the root node and uses the same model-based approach to predict the position of the key. As the algorithm descends the tree, it checks each node's bitmap to determine whether the key is present in that node. Once the node containing the key is found, the key is deleted, and the algorithm updates the bitmap that stores it in the current node.

Algorithm 3 Histogram Delete

```
1: procedure DELETE(node, key)
2:     current_node ← node
3:     while node ≠ null do
4:         pos ← PREDICT_POS(current_node, key)
5:         if pos is child node then
6:             currentnode ← node[pos]
7:         else
8:             key_pos = PREDICT_POS(current_node, key)
9:             delete node[key_pos]
10:            update frequency_array
```

For an example Figure 6.3, deleting key 2 requires traversing down from root node until reaching key 2 nodes similar to query, then use the model to predict the location of key 2 before deleting it from the gapped array.

6.1.4 The Range Query Algorithm

Algorithm 4 Histogram Range Query

```
1: procedure RANGE_QUERY(root, lower, upper)
2:     stack ← empty stack
3:     current_node ← root
4:     keys ← empty array
5:     while current_node ≠ null  or  stack is not empty do
6:         while current_node ≠ null do
7:             push current_node to stack
8:             pos ← PREDICT_POS(current_node, lower)
9:             if pos is a child node then
10:                current_node ← current_node[pos]
11:            else
12:                break
13:        if stack is not empty then
14:            current_node ← pop a node from stack
15:            key_pos ← PREDICT_POS(current_node, lower)
16:            for i ← key_pos to node_size − 1 do
17:                if node[i] ≤ upper then
18:                    append node[i] to keys
19:                else
20:                    break
21:            current_node ← current_node[next position]
22:    return keys
```

Range queries are an essential operation in Learned Index, particularly when dealing with sorted collections of keys. Such queries involve searching for a range of keys

within a given dataset and are commonly used in a wide range of applications, including databases, search engines, and information retrieval systems.

One of the advantages of using a sorted dataset is that it makes range queries relatively easy to perform. By searching for the first key in the range and then iterating through the data structure to collect all keys until the end key is reached, the algorithm can efficiently retrieve all of the data within the specified range.

6.1.5 The Adjustments Algorithm

Algorithm 5 Histogram Gap Distribution

```
1:  procedure HISTOGRAM DISTRIBUTE(node, keys, size, gap_multiples)
2:      num_bins ← 0
3:      bin_size, bin_width ← node.hist
4:      total_gaps ← round(size × gap_multiples)
5:      gaps_per_bin ← empty vector
6:      sum_counts ← 0
7:      for i ← 0 to num_bins − 1 do
8:          gaps ← round(bin_size × total_gaps/size)
9:          gaps_per_bin.push_back(gaps)
10:         sum_counts ← sum_counts + bin_size
11:     result ← empty array of size 2 × size
12:     min_val ← keys[0]
13:     outSize ← 0
14:     for i ← 0 to size − 1 do
15:         bin_index ← floor((keys[i] − min_val)/bin_width
16:         result[outSize] ← keys[i]
17:         outSize ← outSize + 1
18:         if hist.first[bin_index] > 0 then
19:             result[outSize] ← 0
20:             hist.first[bin_index] ← hist.first[bin_index] − 1
21:             outSize ← outSize + 1
22:     return result
```

The strategy we employ to make adjustments to the Learned Index structure utilizes the same condition as LIPP. Prior to making any adjustments, our algorithm traverses the data structure and collects the keys that will be involved in the adjustments. We then create a new node to store these keys.

However, our approach differs in the way we analyze the collected data. Rather than using Fastest Minimum Conflict Degree (FMCD)[26], we use a histogram to examine the distribution of the data and select a bin width based on the characteristics of the

collected data. By doing so, we are able to make more informed decisions about the optimal size of the bin width, which can ultimately lead to more efficient and effective adjustments to the Learned Index.

6.2 Theoretical Analysis

When inserting a node into a gapped array, the algorithm must traverse down the tree until it finds an empty gap to fill. This process takes $O(\log N)$ time.

However, the linear regression model used to predict the position of keys is much more efficient, taking only $O(1)$ time due to its reliance on simple multiplication operations. Additionally, when a new node is inserted into the tree, the histogram in each node must be updated to reflect the new frequency of data points.

To accomplish this, we use an array of frequencies and calculate the bin index using the formula $bin\_index = floor((data\_point - min\_value)/bin\_width)$. The calculation of the bin index for a single data point takes only $O(1)$ time due to the simplicity of the arithmetic operations and the use of the floor function. However, insertions need to be readjusted when the condition is met, similar to the LIPP index, which will cost $O(N \log N)$ where N is the number of keys in the tree. In adjustments, gap redistribution is required and retrained the model to get the accurate prediction of keys and location. However, the adjustments do not occur every time we traverse down the tree path. In this case, we can amortize the cost such that we save $\log N$ credit every time we pass a path which makes the insertion costs similar to the baseline, which is $O(\log^2 N)$

However, the time complexity for updating the histogram frequency with bin width using an array varies depending on the number of data points that map to the same bin. In the best case, the time complexity for updating the frequency is $O(1)$, as updating an element in an array takes constant time. However, in the worst case, the time complexity for accessing the frequency element is $O(\epsilon)$, where ϵ is the number of data points that map to the same bin. This is because computing the index to access the array element can require linear time.

Therefore, the efficiency of updating the histogram frequency in each node during the insertion process can vary greatly depending on the number of data points that map to

each bin. However, the overall time complexity of inserting a new node into the gapped array is dominated by the traversal down the tree, which takes $O(\log N)$ time.

However, deleting a key in the Learned Index algorithm involves more than just traversing down the tree. Once the key is located, the algorithm needs to perform deletion and free the memory in the gapped array that was allocated for the key being deleted. This can be a more complex process than insertion, which only involves finding an empty gap in the gapped array and filling it with a new key.

Similar to insertion, when deleting a key, the algorithm must traverse down the tree until it reaches the appropriate node. This process takes $O(\log N)$ time, where N is the number of elements in the tree. Once the node is reached, the algorithm must search for the key to be deleted. If the key is found, the algorithm must then perform deletion and free up the memory in the gapped array for future key insertion. In addition, it takes the same time complexity to update the histogram.

Querying a key in the Learned Index algorithm involves traversing down the tree until the model predicts that the key is located at the current node. This process takes $O(\log N)$ time, where N is the tree's height. Once the key is found, the algorithm returns its position in the gapped array.

The time complexity for querying a key in the Learned Index algorithm is dominated by the traversal down the tree. Since the linear regression model used by the algorithm can accurately predict the position of keys, the search process is generally efficient and does not require additional time complexity.

In summary, the Learned Index algorithm is designed to provide efficient querying capabilities with time complexity of $O(\log N)$, where N is the number of elements.

Chapter 7

Partial Sorted Insert Strategy

In this section, we will delve into the implementation details of partial sorted insertion strategy and explore various operations such as insertion, update, and deletion. Understanding these fundamental aspects will provide a comprehensive understanding of how partially sorted handles new key insertion.

7.1 Implementation Details

7.1.1 The Insertion Algorithm

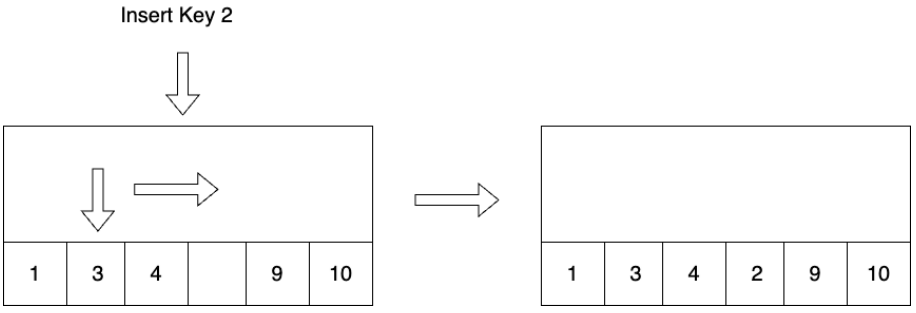

FIGURE 7.1: Example of Partially Sorted Insertion Strategy

When it comes to inserting a new key in the partially sorted gapped array of the LIPP index, the process differs from that of a fully sorted array. Instead of creating a new node in the tree for the key, the algorithm must search for the next available gap within the specified limit to insert the new key. This approach is designed to reduce the memory

overhead of the Learned Index data structure, as it eliminates the need to create new nodes for every new key.

Specifically, the algorithm traverses down the tree until it reaches a node containing a key that is within the predicted position of the new key. Once such a node is found, the algorithm scans forward within the array for the next available gap to insert the new key. This process ensures that the new key is inserted in the correct position in the partially sorted array, without requiring the creation of new nodes in the tree. However,

Algorithm 6 Partially Sorted Insertion Strategy

```
1:  procedure INSERT(node, key)
2:      current_node ← node
3:      path_size[]
4:      while node ≠ null do
5:          pos ← PREDICT_POS(current_node, key)
6:          if pos is child node then
7:              currentnode ← node[pos]
8:              path_size[i] ← node
9:          else
10:             key_pos = PREDICT_POS(current_node, key)
11:             if node[key_pos] is not empty then
12:                 key_pos ← search_next_ϵ(key_pos)
                node[key_pos] ← key
13:     for i ← 0 to path_size − 1 do
14:         node ← path[i]
15:         if rebuilding criteria then
16:             keys ← collect_keys
17:             new_node ← rebuild_tree
18:             path_size[i] ← new_node
```

this approach does have a potential downside in terms of computational overhead. Since the partially sorted array is not fully ordered, the algorithm may need to search further than it would in a fully sorted array to find an available gap to insert the key. This extra search can add additional computational complexity to the insertion process, as the algorithm must spend more time searching for an available gap in the array.

To handle conflicts, partially sorted Learned Index needs to iterate ϵ spaces after predicted position to collect all the partially sorted keys. This process can be time-consuming, especially if the data set is large. In some cases, the algorithm may need to rebuild the whole node entirely to properly sort the partially sorted keys. As seen from the example Figure 7.1, if a key 2 is inserted into an existing tree, it will search for an empty space

after the predicted position 1 and insert the key into the empty location. However, it will only search for ϵ amount of spaces.

A specific example of this can be seen in the context of a gapped array. When the predicted position contains a partially sorted keys, this requires the algorithm to recursive rebuild of both keys. In such cases, the algorithm needs to iterate through the gapped array to collect all the partially sorted keys that may be located ϵ distance away from the predicted position.

Moreover, rebuilding the gapped array becomes necessary if the predicted position is itself a partially sorted key. In this scenario, a recursive rebuilding process must be undertaken to ensure that the gapped array is correctly sorted, which will also affect the running time of this algorithm.

As mentioned earlier, one approach to handling conflicts in a gapped array data structure is to use a recursive rebuilding approach. However, an alternative approach is to use the percentage of partially sorted keys to determine whether to rebuild the whole node.

This approach takes advantage of the fact that when a high percentage of the keys in a gapped array is partially sorted, it may be more efficient to rebuild the whole node rather than to attempt to rebuild when the conflict happens. If the number of partially sorted keys is more than $\frac{1}{2}$ of the number of gapped array lengths, then rebuilding the whole node should be considered.

However, it is essential to note that this approach has limitations and may not always be the most efficient way to handle conflicts in a gapped array. It is also essential to consider other factors, such as the size of the data set and the computational resources available.

In this particular work, we will only be tested on the recursive rebuilding approach. This decision was likely made to simplify the testing process and to provide a clear comparison between baseline and partially sorted gapped array.

7.1.2 The Query Algorithm

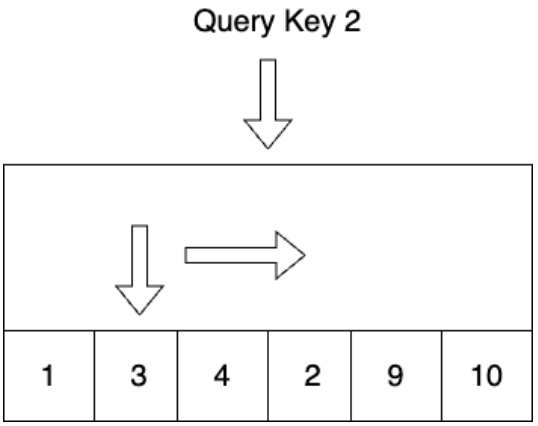

FIGURE 7.2: Example of Partially Sorted Query

When it comes to querying a key in the partially sorted gapped array of the Learned Index algorithm, the process is similar to that of insertion. Like insertion, the algorithm must traverse down the tree until it reaches a node containing a key that is within the predicted position of the queried key. If the key in the node is not the same as the queried key, the algorithm must then scan forward within the array up to the specified limit.

This approach efficiently locates the queried key within the partially sorted array while minimizing the memory overhead of the Learned Index data structure. By accurately predicting the position of the queried key using the linear regression model, the algorithm can quickly navigate the gapped array and locate the keys within the partially sorted limit. However, scanning forward in the array can potentially add computational

Algorithm 7 Partially Sorted Query

```
1:  procedure QUERY(node, key)
2:      current_node ← node
3:      while node ≠ null do
4:          pos ← PREDICT_POS(current_node, key)
5:          if pos is child node then
6:              currentnode ← node[pos]
7:          else
8:              key_pos = PREDICT_POS(current_node, key)
9:              if key_pos ≠ key then
10:                 key_pos ← search_next_epsilon(currentnode, key)
11:             return node[key_pos]
12:     return null
```

complexity to the querying process, mainly if the limit specified is far from the predicted position of the queried key. The algorithm must search through the array to find the next available gap within the limit, which can result in additional computational overhead.

As an example in Figure 7.2, if searching for 2, which is a partially sorted key, the predicted position by the model will be the same as insertion, which is $pos \leftarrow 1$. However, the algorithm must perform a sequential search for ϵ spaces after the predicted position to get key 2.

7.1.3 The Deletion Algorithm

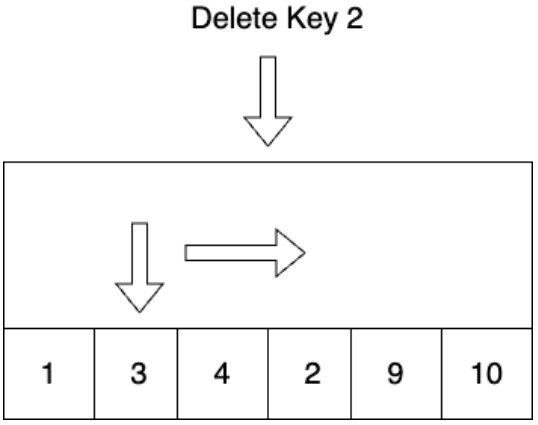

FIGURE 7.3: Example of Partially Sorted Delete

When it comes to deletion in a data structure such as a Learned Index, the process typically involves searching for specific keys that need removal. However, in the case of a partially sorted gapped array structure, this search process can be more complex, as the keys are not necessarily contiguous and may be spread out over multiple sections of the array.

To perform deletion efficiently in a partially sorted gapped array, the Learned Index algorithm must continue searching for the target keys beyond the predicted location. This typically involves searching until the ϵ partial sorted limit, where ϵ is the number of empty spaces after the predicted position the keys can be inserted into.

By continuing the search process this way, the Learned Index can ensure that it locates all of the relevant keys for deletion, regardless of their position within the array. This

Algorithm 8 Partially Sorted Delete

```
1:  procedure DELETE(node, key)
2:      current_node ← node
3:      while node ≠ null do
4:          pos ← PREDICT_POS(current_node, key)
5:          if pos is child node then
6:              currentnode ← node[pos]
7:          else
8:              key_pos = PREDICT_POS(current_node, key)
9:              if key_pos ≠ key then
10:                 key_pos ← search_next_epsilon(currentnode, key)
11:             delete node[key_pos]
```

can be a highly efficient method for performing deletions in large datasets, as it avoids the need for costly full scans of the entire array and instead focuses on the relevant sections of the data structure.

As an example, in Figure 7.3, deleting the keys works similar to query as it has to perform sequential search to find the key that has to be deleted. In this case, the predicted position of key 2 is 1 but the algorithm has to search for ϵ spaces after the predicted position to get the location of key 2.

7.1.4 The Range Query Algorithm

Typically, when performing a range query on a partially sorted gapped array, the algorithm will search for the first key within the specified range and then iterate through the array, collecting all keys until it reaches the end key. However, even after reaching the end key, there may still be additional keys that fall within the specified range but are located in the ϵ spaces beyond the end key.

The algorithm must continue searching beyond the end key, ensuring that all relevant keys are included in the range query, considering these additional ϵ spaces. By doing so, the algorithm can identify any keys not included in the initial search but are still within the specified range, providing a comprehensive and accurate result set.

Algorithm 9 Partially Sorted Query

```
1: procedure RANGE_QUERY(root, lower, upper)
2:     stack ← empty stack
3:     current_node ← root
4:     keys ← empty array
5:     while current_node ≠ null  or  stack is not empty do
6:         while current_node ≠ null do
7:             push current_node to stack
8:             pos ← PREDICT_POS(current_node, lower)
9:             if pos is a child node then
10:                current_node ← current_node[pos]
11:            else
12:                break
13:        if stack is not empty then
14:            current_node ← pop a node from stack
15:            key_pos ← PREDICT_POS(current_node, lower)
16:            for i ← key_pos to node_size − 1 do
17:                if node[i] ≤ upper then
18:                    append node[i] to keys
19:                else
20:                    break
21:            current_node ← current_node[next position]
22:    collect_keys_ϵ(currentnode, keys)
23:    return keys
```

7.1.5 The Adjustments Algorithm

When it comes to working with partially sorted items, several challenges can arise. One of the most significant is that the items are not sorted, and the keys may not follow a consistent, monotonically increasing pattern. This can make it difficult for models to accurately predict the partially sorted keys, leading to inaccuracies in any adjustments made to the item.

To address this issue, one practical approach is to collect all of the keys that need to be rebuilt and retrain the model based on the data collected. By carefully analyzing the information gathered, we can identify patterns and trends that can help us create a more accurate and reliable model. This can involve everything from reviewing historical data to conducting extensive data analysis and modelling exercises.

Another strategy that can be used in the adjustments process is to mix the histogram strategy. This statistical method is used to estimate the distribution of a set of data, which can help us to place gaps between different keys in the item more accurately. By

Time Complexity		
	Partially Sorted	Histogram
Insertion Operation	$O(\epsilon + \log N)$	$O(\log^2 N)$
Query Operation	$O(\epsilon + \log N)$	$O(\log N)$
Delete Operation	$O(\epsilon + \log N)$	$O(\log N)$
Adjustment Operation	$O(N \log N + \epsilon)$	$O(N \log N)$

TABLE 7.1: Time Complexity of Partially Sorted and HistogramLearned Index

doing so, we can ensure that our adjustments are more precise and that the item is better suited to meet the needs of its intended use.

To simplify the process, we have decided to use Ordinary Least Square (OLS) as the sole training method for rebuilding the partially sorted Learned Index.

7.2 Theoretical Analysis

Insertion operations for a partially sorted array require the algorithm to search for an empty space before inserting the new key. This search operation takes $O(\log N)$ time, where N is the number of keys in the Learned Index. The new key can be inserted immediately if an empty space is found.

However, if the space is not empty, the algorithm must perform a search for ϵ spaces after the predicted position to find a suitable location for the new key. This operation takes $O(\epsilon + \log N)$ time, where ϵ is the limited number of spaces after the predicted position.

In the case of rebuilding recursively, the algorithm needs to rebuild nodes with partially sorted keys. This operation takes $O(\frac{N}{2})$ time, where N is the number of keys in the rebuilding nodes. The algorithm needs to split the nodes into two groups and recursively rebuild them until all nodes are fully sorted.

The time complexity of a query operation in a partially sorted array with a Learned Index is dependent on the position of the key in the array. If the key is located in an empty space, the query operation takes $O(\log N)$ time. This is because the algorithm needs to traverse the Learned Index down to the appropriate leaf node and then search for the key within that node.

However, if the predicted position for the key already contains a key, the algorithm needs to perform additional searches to find the appropriate location for the new key. This takes an additional ϵ amount of time, resulting in a total time complexity of $O(\epsilon+\log N)$. The value of ϵ depends on the limit we set for the number of spaces the algorithm can search after the predicted position.

The time complexity of deletion in a partially sorted array with a Learned Index is also $O(\epsilon + \log N)$, where ϵ is the limited number of spaces after the predicted position that the algorithm searches for the key to be deleted.

The algorithm first traverses the Learned Index to find the appropriate leaf node and then searches for the key to be deleted. If the key is not found at the predicted position, the algorithm will continue searching for ϵ spaces after the predicted position to find the key. Once the key is found, the algorithm removes it from the partially sorted array.

Since deletion operations do not involve rebuilding the Learned Index, the time complexity remains the same regardless of the distribution of the keys in the Learned Index.

Adjustment operations in a partially sorted learn index involve collecting a specified number of keys and performing OLS to train the model. The time complexity for adjustment operations is $O(N \log N + \epsilon)$, where N is the number of keys collected for the adjustment process.

The most time-consuming part of the adjustment process is training the model using OLS. However, once the model is trained, it does not require retraining until insertion triggers the recursive rebuilding or once the condition of the adjustment is met.

Chapter 8

Evaluation

8.1 Experiment Setup

8.1.1 Environment

The programming language used for this experiment is C++ to implement the algorithm and perform the single-threaded operation on an M1 pro with ten core CPU and 16GB of RAM running on a Mac Operating System. In this section, we compare four implementations mentioned in the previous section 5. The experiment will test the following trials. (1) Operation performance compares how fast it performs on read, write and update operations **Static** scenario. (2) Memory Consumption compares the amount of memory it took to perform the algorithms. (3) Number of Conflicts compares the number conflict that happens during the insertion operations. (4) Average Tree Depth compares the number of average depths with the baseline. The test will perform on **Static** scenario where it assumes that the new data still follow the same distribution, and we will also test on some real-world datasets (**Dynamic** scenario) where the distribution of the new data is random.

	Dataset			
	Gaussian Distribution	Longitudes	Log Normal	Powerlaw
Num Keys	20M	20M	20M	20M
Total Size	1.6GB	1.6GB	1.6GB	1.6GB

TABLE 8.1: Characteristics of the dataset

8.1.2 Datasets

8.1.2.1 Synthetic Datasets

We conducted a comprehensive experimental study on synthetic datasets to evaluate the performance of different tree-based learned index implementations regarding insertion and search times. To generate the synthetic datasets, we randomly generated data based on Gaussian distribution, with the keys being of type double and 8 Bytes in size. We evaluated the different implementations based on the generated data distribution, which we characterized and analyzed. Table 8.1 presents the characteristics of the generated datasets.

To test the performance of the implementations, we generated a separate set of synthetic data based on the same Gaussian distribution for insertion operations. We also experimented with synthetic datasets that followed different distributions, such as power law and lognormal distributions and contained unique keys. We limit the number of keys a maximum of twenty million for simplicity in our evaluation. In the this test, we will perform bulk loading of maximum of twenty million and insertion another twenty million to test all of the operations, memory consumption and amount of new nodes creation.

A gaussian distribution, also known as a normal distribution, is a probability distribution which characterized by its symmetric and bell-shaped curved and the majority of the data is concentrated around the mean value. The gaussian distributions are commonly observed in heights of individuals in a population and etc.

On the other hand, power law distributon is a probability distribution that follows a powerlaw, which means that the frequency of event is inversely proportional to its size raised by the constant exponent. The power law has long tail, indicating a high

frequency of rare events. This distribution is commonly observed in natural such as the distribution of city sizes, the frequency of word usage and etc.

Lastly, the lognormal distribution is characterized by a skewed, right-tailed shape, where most of the concentrated at small values. The common scenario that relates to the lognormal is the distribution of incomes and salaries.

Our experiments focused on optimizing the insertion overhead caused by conflict that occur when space is already occupied in the tree-based learned indexes. We aimed to reduce the number of conflict by strategically placing gaps based on the existing data distribution. This optimization is crucial, as it can significantly improve the performance of insertion operations in tree-based learned indexes.

8.1.2.2 A Real World Dataset

The Open Street Map is a comprehensive platform that provides geographic data, including longitude, which contains information about locations around the world. We have collected the longitude data from Open Street Map and used it as a basis for our dataset. Specifically, we use the longitude information as our key and test it based on an 8-byte key size [20]. This data is completely random and does not follow any distribution, which accurately represents the real-world scenarios where systems insert data that may not follow any distributions.

8.2 Operation Performance

In the context of the partially sorted Learned Index and Histogram gap placement, the performance of each operation is crucial to ensure the efficiency and effectiveness of the data structure. To evaluate the performance of each operation, we measure their running time in milliseconds. Insertion, query, deletion, and adjustment are the four primary operations that we will evaluate.

To accurately evaluate the performance of each operation, we will measure their running time in milliseconds. The running time will be measured on a variety of input sizes to ensure that the data structure can handle different workloads efficiently. Additionally, we will measure the performance of each operation under different conditions.

By evaluating the performance of each operation, we can determine the strengths and weaknesses of the partially sorted Learned Index and Histogram gap placement and identify any areas where improvements can be made. This information will be valuable in optimizing the data structure and improving its performance in practical applications.

8.2.1 Insertion

In our recent empirical analysis, our primary objective was to comprehensively evaluate two algorithms that involve inserting partially sorted lists and gap optimization using a histogram, as compared to the baseline LIPP algorithm. Our overarching aim was to ascertain the extent to which these newly proposed algorithms confer a superior level of performance over the baseline and to assess the changing trend in performance compared to the baseline algorithm across different input sizes.

To achieve this aim, we subjected the algorithms to an array of tests using varying input sizes, ranging from one million to twenty million. By conducting tests across this spectrum of input sizes, we were able to determine the degree to which each algorithm's performance varied with respect to the input size.

To bolster the rigour of our assessment, we employed both static and dynamic scenarios in our evaluation. Through the use of different probability distributions, such as Gaussian, Log Normal, and Powerlaw, we sought to assess the efficacy of our algorithms in diverse and varied scenarios. Our analysis aimed to establish if the algorithms under consideration can confer a performance advantage over the baseline algorithm in these diverse scenarios.

Additionally, we conducted tests in the dynamic scenario to observe the algorithms' adaptability and robustness when dealing with random distributions. This allowed us to evaluate if the algorithms under consideration could maintain their performance gains when dealing with unforeseen circumstances.

Insertion Performance Result for Gaussian Distribution (in ms)			
Num Keys	Baseline (LIPP)	Partially Sorted	Histogram
1M	106	150	136
5M	696	845	742
10M	1556	2080	1547
20M	5875	7130	4575

TABLE 8.2: Insertion Results with Gaussian Distribution

8.2.1.1 Gaussian Distribution

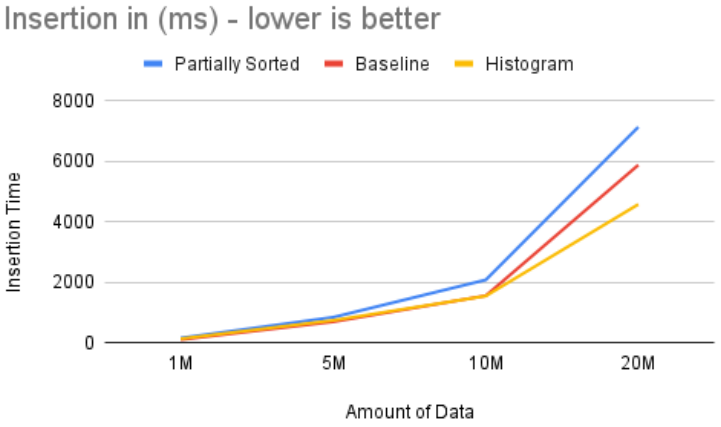

FIGURE 8.1: Overall Insertion Operation vs Number of Insertion Operation Following Gaussian Distribution.

The empirical analysis we conducted, which is presented in Table 8.2 and Figure 8.1, revealed that gap optimization using the histogram is a superior algorithm to the baseline with FMCD [26] and partially sorted gapped array algorithms when the data follows Gaussian Distribution. This was evident from the performance gains realized by the histogram-based algorithm, which was able to capture the distribution of the data and distribute gaps based on the data's distribution, thus performing better than the LIPP when newly inserted data followed the same distribution as the existing data.

Moreover, the histogram-based algorithm's performance superiority over the baseline algorithm was found to widen as the amount of data stored in the Learned Index tree increased, as depicted in Figure 8.1. This can be attributed to the histogram's efficient distribution of gaps in the data based on its distribution, which enables it to maintain optimal performance even with larger data sets.

Insertion Performance Result for Lognormal Distribution (in ms)			
Num Keys	Baseline (LIPP)	Partially Sorted	Histogram
1M	127	156	199
5M	1310	1593	1244
10M	2593	3580	2443
20M	5129	5930	4835

TABLE 8.3: Insertion Results with Lognormal Distribution

Conversely, the partially sorted gapped array algorithm performed the worst among the three algorithms. This was mainly due to its requirement to perform a sequential search within the gapped array to locate empty spaces before inserting new data. As a result, the partially sorted algorithm's performance deteriorated as the amount of data stored in the Learned Index tree increased, as illustrated in Figure 8.1. With larger amounts of data, there were more conflicts, which caused the partially sorted Learned Index to conduct more sequential searches, resulting in inferior performance compared to the baseline and histogram gap optimization algorithms.

8.2.1.2 Lognormal Distribution

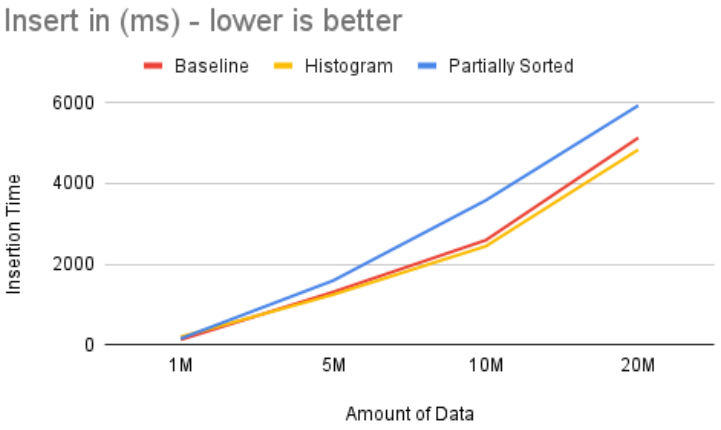

FIGURE 8.2: Overall Insertion Operation vs Number of Insertion Operation Following Lognormal Distribution.

Upon analyzing the results presented in Table 8.3 and Figure 8.2, we can conclude that the gap optimization using histogram algorithm still performs better than the baseline and partially sorted gapped array algorithms when using Log Normal Distribution. The results reveal that the histogram algorithm's superior performance is due to its ability

to capture the distribution of the data and distribute gaps accordingly, thereby achieving better performance than the LIPP when the newly inserted data follows the same distribution as the existing data.

Similar to the findings with Gaussian Distribution, as the amount of data stored in the Learned Index tree increases, the performance gap between the histogram-based algorithm and the other two algorithms widens, as illustrated in Figure 8.2. This can be attributed to the histogram algorithm's efficient distribution of gaps in the data, which enables it to maintain optimal performance even with larger data sets.

On the other hand, the partially sorted gapped array algorithm performed the worst among the three algorithms, as observed in the results presented in Figure 8.2. The reason for this inferior performance can be attributed to the extra computations required to search for empty spaces after the predicted position. With larger amounts of data, the partially sorted algorithm's performance further deteriorates due to the increased amount of recursive rebuilding required when the spaces after the predicted position run out. Additionally, since each node contains more partially sorted items, each node takes longer to perform recursive rebuilding, leading to reduced performance as the amount of data increases.

The findings from our evaluation underscore the importance of considering the distribution of data when designing algorithms for processing and storing data. The results indicate that gap optimization using the histogram is a more efficient algorithm than the baseline and partially sorted gapped array algorithms, particularly when the newly inserted data follows the same distribution as the existing data. Leveraging techniques such as gap optimization using histograms can optimize performance and efficiency in data processing and storage.

8.2.1.3 Powerlaw Distribution

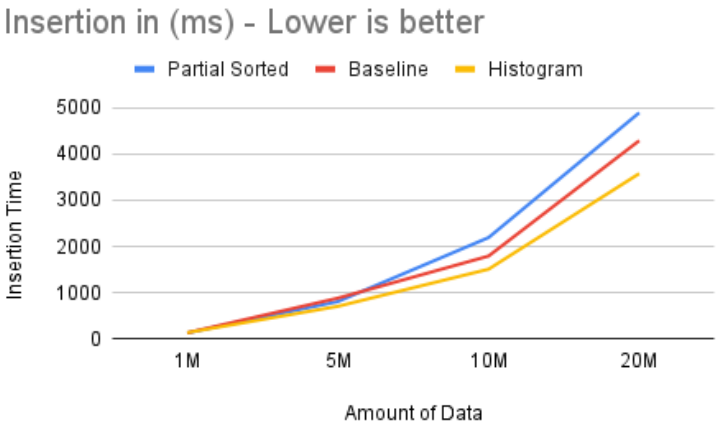

FIGURE 8.3: Overall Insertion Operation vs Number of Insertion Operation Following Powerlaw Distribution.

It is important to note that power-law distribution is different from the other distributions tested in that it has a long tail, meaning that there are a few elements that frequently occur while most elements are rare. This property can make it more challenging to capture the distribution using a histogram-based approach accurately.

Additionally, the wider spread of the performance gap in Figure 8.3 compared to the other distributions indicates that the insertion operation is more unpredictable in power-law distributed data. This may be due to the highly skewed nature of the distribution, making it difficult to predict where new elements should be inserted accurately.

Despite these challenges, the histogram-based gap optimization approach still outperforms the baseline and partially sorted gapped array in power-law distribution. This suggests that the approach is still effective in capturing the distribution and distributing gaps accordingly, leading to better insertion performance.

The poor performance of the partially sorted gapped array in power-law distribution further supports the idea that this approach is unsuitable for indexing this data type. As the amount of partially sorted keys in each node increases with larger datasets, the amount of recursive rebuilding required also increases, leading to worse performance.

8.2.1.4 Real-World Dataset (Longitudes)

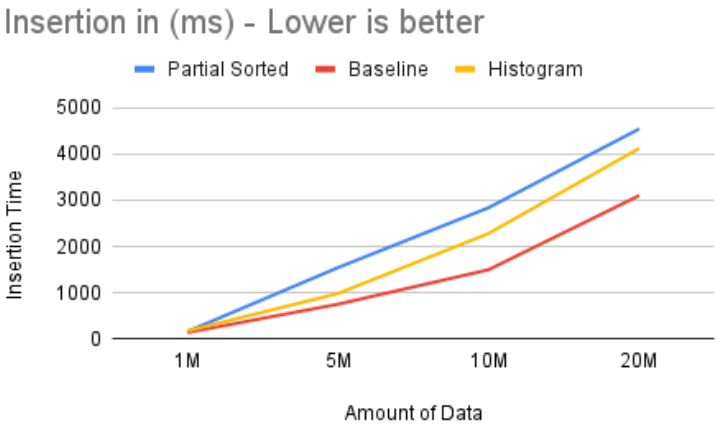

FIGURE 8.4: Overall Insertion Operation vs Number of Insertion Operation using Longitude Dataset.

To further analyze the result on a real-world dataset in Figure 8.4, it is important to consider the nature of real-world data and how it can affect the performance of the algorithms. Real-world data is often diverse and can have a wide range of distributions, from uniform to skewed, and can also have varying degrees of correlation and clustering.

The baseline algorithm is able to perform better as it uses a FMCD that distributes gaps and keys to optimize the linear regression model, and it does not require rebuilding during the insertion operation or does not have any assumption on the dataset. In other words, the baseline can be a good general purpose which performs well in any scenario.

However, the histogram algorithm may not perform as well on real-world datasets as it does not account for the changing distribution of the data. The algorithm assumes that the data distribution is consistent and places gaps based on that assumption. This can lead to slower insertion times when the new data's distribution differs from the existing data.

Moreover, the partially sorted gapped array is not suitable for real-world datasets as it does not help the linear regression model improve its accuracy. Instead, it only delays the new node creation at a later stage to amortize the cost, which can be costly in terms of performance when there is a huge amount of data in the node.

The histogram algorithm is designed to work well when the distribution of the data is known, and it is able to adapt the gaps based on that distribution. But in real-world

scenarios, where the distribution is not known or random, the baseline performs better as it is able to generalize to fit the randomness of the data.

8.2.2 Query

To evaluate the performance of the query operation, we will test varying input sizes ranging from one million to twenty million, in order to determine which algorithms perform best in different scenarios and input sizes. Our tests will include both static (Gaussian, Lognormal, and Powerlaw) and dynamic (Longitudes) scenarios.

Testing static scenarios is important for query performance because it allows us to analyze the algorithms' behavior under specific data distributions. By using static datasets such as Gaussian, Lognormal and Powerlaw, we can evaluate how well the algorithms perform when the data follows a known distribution, and use this information to optimize the algorithm. For example, if an algorithm performs well on a Gaussian dataset, which is a commonly occurring distribution in real-world data, we can be confident that it will perform well on similar datasets in the future. This allows us to make more informed decisions when selecting which algorithms to use for specific tasks, and can help improve overall system performance.

Furthermore, testing static datasets allows us to measure the impact of the distribution on query performance, and identify any potential areas for optimization. This is particularly useful for algorithms that are designed to work with specific types of data distributions, such as the histogram-based algorithm that we tested earlier.

For each scenario, we will measure the query time taken by the baseline, histogram, and partially sorted algorithms. The results will be recorded and analyzed to determine which algorithm performs the best in each scenario.

This approach is necessary because different datasets may have different characteristics that affect the performance of the query operation. By testing different scenarios with varying input sizes, we can gain a better understanding of the strengths and weaknesses of each algorithm and determine which algorithm is the most suitable for a particular scenario.

Furthermore, by including both static and dynamic scenarios, we can compare the performance of the algorithms in scenarios where the data distribution is known beforehand

and scenarios where the data distribution is constantly changing. This will provide valuable insights into the suitability of each algorithm in real-world scenarios where the data distribution is not known beforehand.

8.2.2.1 Gaussian Distribution

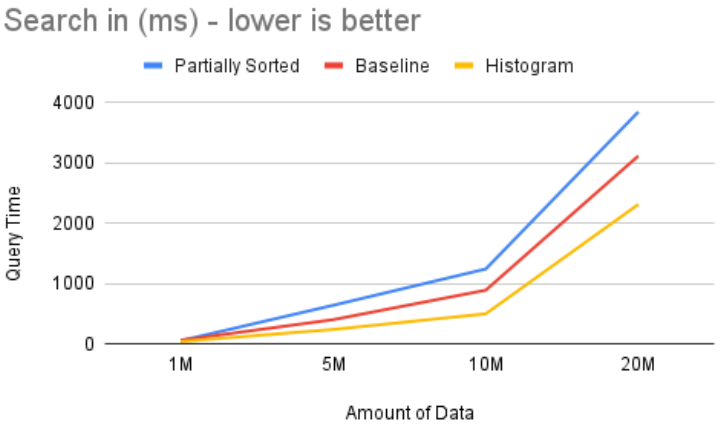

FIGURE 8.5: Overall Query Operation vs Number of Query Operaton Following Gaussian Distribution

The results of the experiment show that the performance of Learned Index algorithms varies significantly depending on the distribution of the input data (Figure 8.5). When it comes to query operations on the Gaussian Distribution, the histogram algorithm outperforms the other algorithms. This is because the histogram algorithm makes use of the gaps between keys that are specifically optimized for the given distribution. By doing so, it traverses lesser depth, which in turn improves the overall query performance.

On the other hand, the baseline algorithm performs relatively well compared to the partially sorted Learned Index algorithm. However, the baseline algorithm tries to generalize and perform well in most cases without taking advantage of the prior data distribution. This means its performance is less than the histogram algorithm in certain distribution types.

The partially sorted algorithm performs the worst of the other two algorithms in query operations. This is due to the extra computational overhead required when the keys are partially sorted. Even though the amount of conflicts is reduced with the partially sorted gapped array, the algorithm still spends a significant amount of time doing sequential

searches due to potential miss predictions by the model and the partially sorted gapped array.

It is important to note that the above results were obtained in static scenarios, where the input data distribution remains constant. The histogram algorithm performed best in static scenarios because it was specifically optimized for the given distribution.

8.2.2.2 Lognormal Distribution

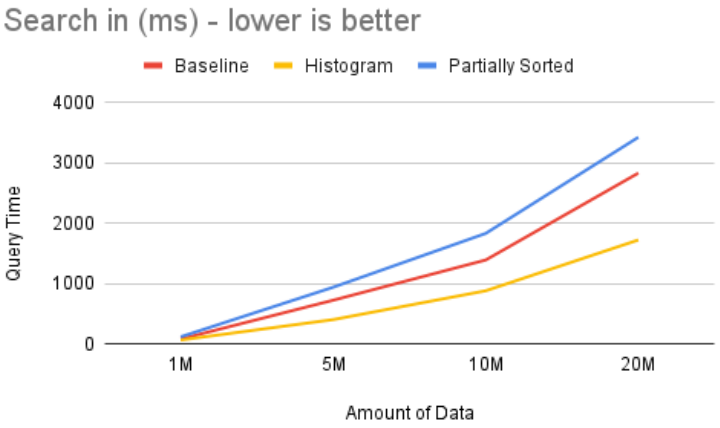

FIGURE 8.6: Overall Query Operation vs Number of Query Operation Following Lognormal Distribution.

The experiment was also conducted on a lognormal distribution dataset. Similar to the Gaussian distribution (Figure 8.6), the histogram algorithm outperformed the baseline LIPP and the partially sorted array in terms of query performance. The results shown in Figure 2 demonstrate that as the amount of data increases, the performance gap between the histogram and the baseline widens. This indicates that the histogram algorithm takes advantage of the data distribution and optimizes gaps in a way that allows the query to traverse fewer levels in the tree, leading to better performance.

On the other hand, the baseline algorithm still outperformed the partially sorted array even though the latter had a lower average tree depth. This is due to the extra computational overhead required for the ϵ linear search when there are a large number of partially sorted items in the tree. Despite having a lower average tree depth, the partially sorted array's slower search process prevented it from performing as well as the baseline in terms of query performance.

8.2.2.3 Powerlaw Distribution

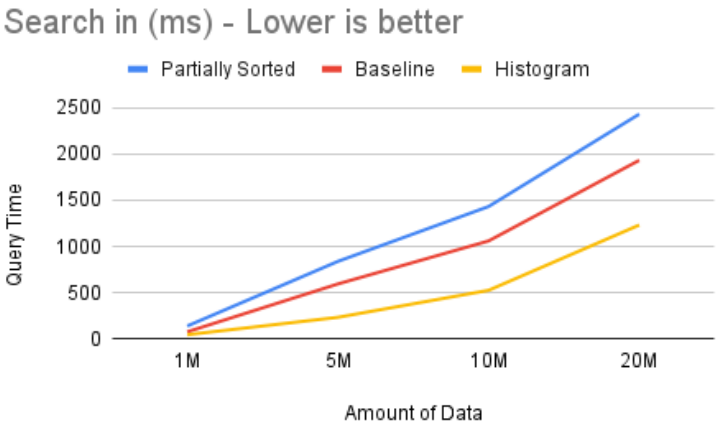

FIGURE 8.7: Overall Query Operation vs Number of Query Operation Following Powerlaw Distribution.

The study also evaluated the performance of the three data structures on Zipfian distribution (Figure 8.7). The results were similar to the other distributions, with the histogram performing the best and the partially sorted gapped array performing the worst.

The histogram data structure again outperformed the baseline and partially sorted Learned Index structures due to its optimization for the Zipfian distribution. The gaps in the histogram structure are also specifically designed to account for the high frequency of certain keys in the distribution, resulting in a lower average tree depth and improved query performance.

In contrast, the baseline Learned Index performed relatively well in querying Zipfian distribution, but still could not match the efficiency of the histogram. This is because the baseline tries to generalize and perform well in most cases, but does not take full advantage of the specific distribution characteristics.

The partially sorted gapped array again struggled with the Zipfian distribution, with the extra computational overhead of the ϵ linear search causing a significant performance gap from the other two structures. As with the other distributions, the partially sorted Learned Index performed better than the baseline Learned Index in terms of average tree depth. However, this improvement was outweighed by the increased search time from the linear search.

8.2.2.4 Real-World Dataset (Longitudes)

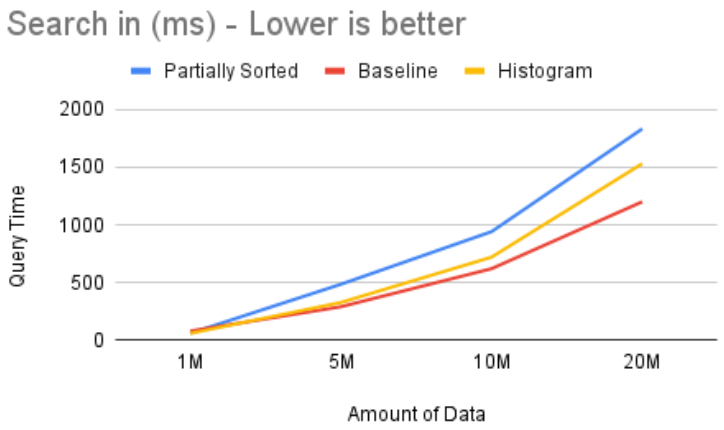

FIGURE 8.8: Overall Query Operation vs Number of Query Operation using Longitude Dataset.

Moving to a real-world dataset, the experimental result in Figure 8.8 shows that the histogram performs on par with the baseline when it comes to range query. This indicates that both the baseline and histogram have a similar average tree depth which contributes to the similar query performance.

It is important to note that the query performance is heavily dependent on the height of the tree. If the tree height is greater, it leads to poor query performance. This can be attributed to the fact that a larger height would require the query algorithm to traverse through more levels of the tree, hence making the query more time-consuming.

However, when it comes to the partially sorted gapped array, the performance is still not up to the mark. Despite having a lower average tree depth compared to both baseline and histogram, the extra computation required for the linear search, after traversing the tree depth, hampers its overall query performance.

8.2.3 Deletion

In order to gain a deeper understanding of the performance of partially sorted and histograms compared to the baseline, we conduct further experiments involving delete operations. This setup is similar to the previous tests, with varying input sizes ranging from one to twenty million, using different distributions and a real-world dataset.

In the deletion operation, the performance is expected to be similar to that of the query operation, as the tree must be traversed to locate the keys before they can be deleted. This provides us with additional insight into the efficiency of each algorithm and how they handle deletions.

These experiments will allow us to further analyze the performance of each algorithm in different scenarios, providing a more comprehensive understanding of their strengths and weaknesses.

8.2.3.1 Gaussian Distribution

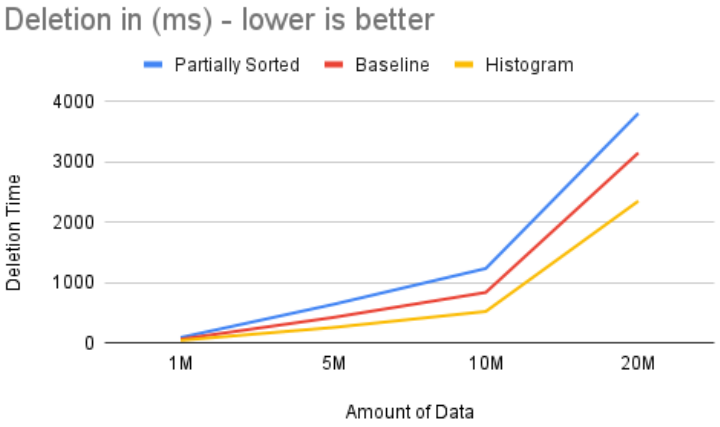

FIGURE 8.9: Overall Deletion Operation vs Number of Deletion Operation Following Gaussian Distribution.

The results Figure 8.9 showed that the histogram outperformed both the baseline and the partially sorted array in Gaussian distribution. This was due to the lesser average depth of the tree, which made traversing down the tree faster, allowing for faster deletions.

It was also observed that the baseline still performed better than the partially sorted, even though it had a higher average tree depth compared to the partially sorted array. This was due to the precise position of the keys from the model and not requiring an extra linear search like the partially sorted gapped array. Furthermore, the sharp spike is due to increasing data from 10 M to 20 M.

8.2.3.2 Powerlaw Distribution

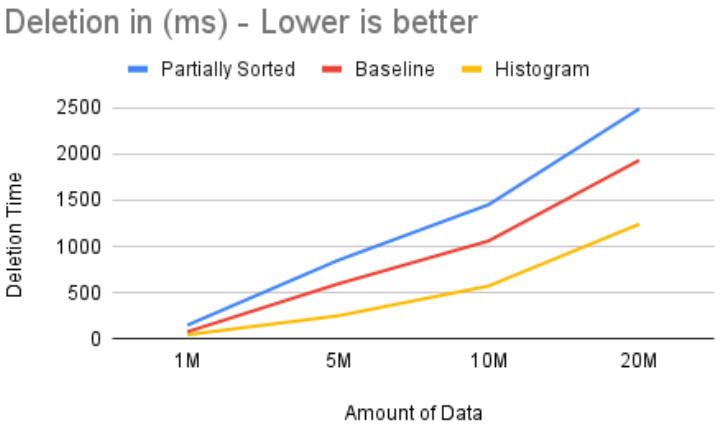

FIGURE 8.10: Overall Deletion Operation vs Number of Deletion Operation Following Powerlaw Distribution.

For the Powerlaw Distribution, we use a similar strategy as other distributions where we bulk load the keys and then insert the same amount into the tree. The resulting Figure 8.10 shows that the histogram still outperforms both baseline and partially sorted. The deletion works similarly to the query as it has to traverse down until it reaches a key before performing deletion which makes the deletion process bounded by the height of the tree. In this case, the reason why histogram achieve better performance is because it is able to make use of the gaps available in upper node and create lesser child node than the baseline. However partially sorted is slower even with similar tree depth is because it is bounded by the ϵ spaces and the height of the tree. The amount of time is mostly spent on the linear ϵ search which causes the gap in performance between the histogram and the partially sorted.

Overall, the results in Figure 8.10 showed that the histogram indexing algorithm was consistently the best performer in terms of query, insertion, and deletion operations across various distributions and real-world datasets. However, it is worth noting that the baseline algorithm also performed relatively well and could be considered as a viable alternative in scenarios where the data distribution is unknown. On the other hand, the partially sorted gapped array algorithm performed poorly and could be improved by reducing the number of partially sorted keys in the gapped array.

8.2.3.3 Real-World Dataset (Longitudes)

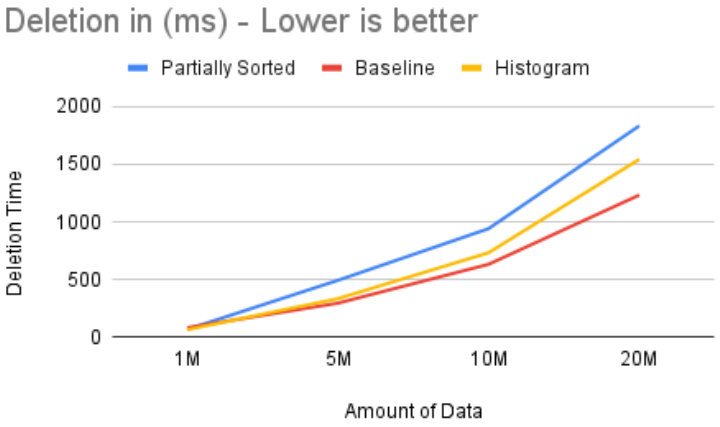

FIGURE 8.11: Overall Deletion Operation vs Number of Deletion Operation using Longitude Dataset.

The real-world dataset serves as a more practical and relevant evaluation for these algorithms. The deletion operation is expected to have a similar performance to the query operation since both require traversing down the tree to locate the keys to be deleted or queriesd. As expected, the results in Figure 8.11 show that the histogram and baseline perform similarly when deleting keys from the tree. This is due to their similar number of average tree height, which is a crucial factor that influences the query or deletion performance.

In contrast, the partially sorted gapped array performs worse than the histogram and baseline. Even though the partially sorted array has a lower average tree depth than the baseline, it requires an extra linear search after traversing down the tree to search for the predicted position of the key. This search is necessary because the keys could be located anywhere within a range of x to $x + \epsilon$ around the predicted position. Consequently, the partially sorted array is bounded by the tree's height and the ϵ limit spaces after the predicted position. These factors contribute to its inferior performance compared to the histogram and baseline.

8.2.4 Adjustment / Branch Pruning

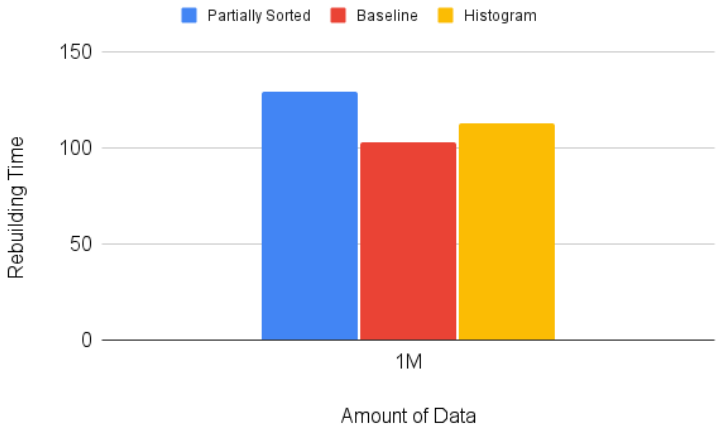

FIGURE 8.12: Overall Rebuilding Time vs Amount of Keys in the Tree Following Gaussian Distribution

The Learned Index Precise Position (LIPP) algorithm is designed to optimize the performance of learned indexes by making adjustments to the tree's structure under specific conditions. These adjustments aim to reduce the tree's height, thereby reducing the time it takes to access and retrieve keys. During our testing of the learned indexes, we did not modify any of these conditions since they were already optimized for performance. However, we recognized that each of the algorithms (baseline, histogram, and partially sorted) has a unique implementation of these adjustments, which needed to be tested to determine the most effective implementation.

To test the effectiveness of these adjustments, we employed a rigorous testing strategy. We tested the algorithms on the Gaussian distribution dataset and measured the execution time in milliseconds. By comparing the performance of the three algorithms, we aimed to identify which implementation of the adjustments was most effective.

From the result Figure 8.12, we observed that the histograms perform slower than the baseline on the test data. This is because the gap redistribution, retraining of the model, and rebuilding new histogram from the newly collected keys required for the histogram adjustments result in an operating cost of $O(N \log N)$. It costs the same as the baseline $O(N \log N)$ [26], but the baseline does not need to rebuild the histogram, making it faster than the histogram algorithm. The partially sorted algorithm also performed worse than the baseline and histogram algorithms. This can be attributed

to the challenges associated with collecting partially sorted keys. Nevertheless, like the histogram algorithm, the partially sorted algorithm's adjustments were also dominated by the retraining of the model and the collection of keys.

8.3 Memory Consumption

In addition to measuring the execution time, it is also essential to analyze the memory consumption of each implementation. Memory consumption refers to the amount of memory that is required to perform each operation. Therefore, in this section, we will be using the same data set as before but measuring the memory consumption of each implementation.

Measuring memory consumption is crucial as it helps us determine the amount of memory resources required to run each algorithm effectively. By analyzing the memory consumption, we can optimize the algorithm to reduce its memory footprint, which is important in scenarios where memory is a scarce resource.

We will use the same testing strategy as before, running each algorithm on the same data set and measuring the memory consumption in bytes. By comparing the memory consumption of the three algorithms, we can determine which implementation is the most memory-efficient.

It is important to note that memory consumption is affected by various factors, such as the size of the data set, the structure of the data, and the algorithm's implementation. Therefore, the results obtained in this section will provide valuable insights into the memory consumption of each algorithm and help us optimize the algorithm for better performance.

8.3.1 Gaussian Distribution

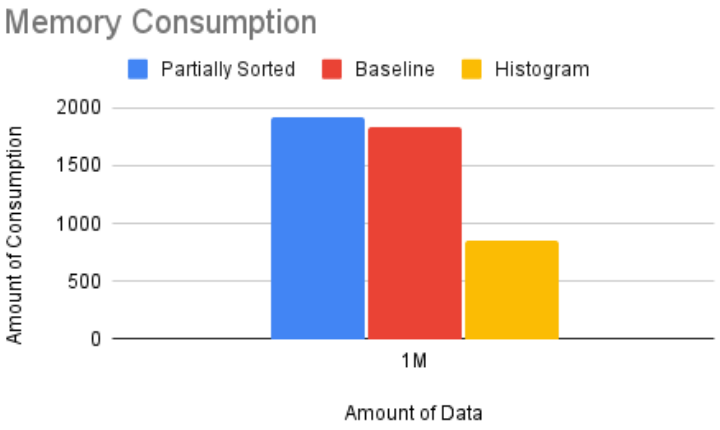

FIGURE 8.13: Results of Memory Consumption with Gaussian Dataset 1M

From the result Figure 8.13 of the test on Gaussian Distribution, we can observe that the histogram performs best in terms of memory consumption, consuming the least compared to the partially sorted array and the baseline. The reason behind this is that the histogram maintains an array of frequencies in each node, allowing it to capture the distribution of the keys. As a result, it can predict the exact position of a key without having to perform a linear search when there is a miss prediction, leading to fewer conflicts and the creation of fewer child nodes, ultimately reducing memory consumption. Even though, maintain frequencies in each node consumes extra memory but having lesser child node and average tree depth (Figure 8.17) makes the histogram performs better in term of memory consumption.

In contrast, the partially sorted array performs worse than the histogram, even though the main idea behind the partially sorted is to delay the creation of child nodes as much as possible. As the data set grows, more partially sorted keys will be in each node. When a conflict occurs, and ϵ empty spaces run out, the algorithm still has to perform recursive rebuilding, creating numerous child nodes that consume memory. Therefore, the result is expected.

Moreover, the baseline implementation performs similarly to the partially sorted. The baseline algorithm aims to work well in most scenarios, which makes it perform worse when there are assumptions that can be taken advantage of. From the result Figure 8.17, we can see that the baseline creates more child nodes, which in turn consumes

more memory than the partially sorted and histogram as the new nodes have to reserve some memory as gaps to be used for new keys.

8.3.2 Powerlaw Distribution

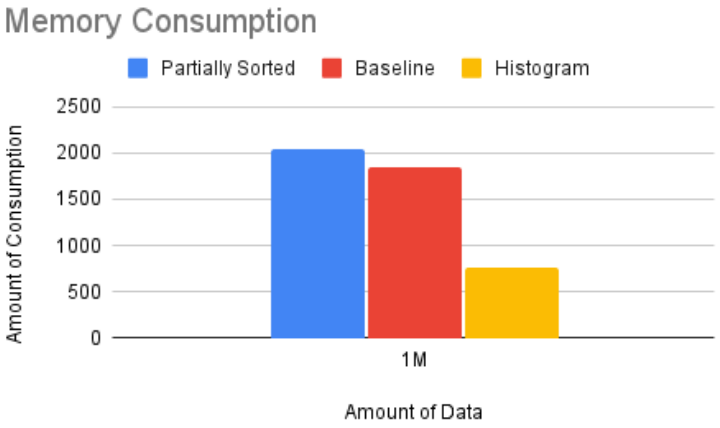

FIGURE 8.14: Results of Memory Consumption with Powerlaw Distribution Dataset 1M

Our memory consumption test on the Powerlaw Distribution shows a similar trend to that of the Gaussian Distribution (Figure 8.14). The histogram outperforms both the baseline and partially sorted array in memory consumption. The histogram consumes less memory, making it more efficient for use in a static scenario where the data distribution is known. It is able to create fewer child nodes, utilizing the gaps efficiently.

On the other hand, the partially sorted array delays the creation of new nodes to amortize the cost. However, as the amount of data increases, it will eventually have to spend a significant amount of time rebuilding nodes recursively. This means that the number of nodes that need to be rebuilt will continue to increase as more data is added to the Learned Index.

Similarly, the baseline performs similarly to the partially sorted array. It does not make any assumptions about the data distribution, unlike the histogram, which creates more child nodes than the baseline.

8.3.3 Lognormal Distribution

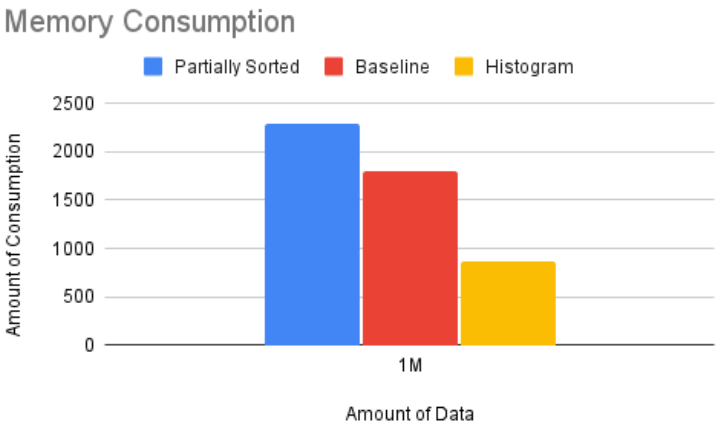

FIGURE 8.15: Results of Memory Consumption with Lognormal Distribution Dataset 1M

The memory consumption test showed that the histogram outperformed the baseline and partially sorted Learned Index, which is consistent with the results from the Powerlaw Distribution (Figure 8.15). The histogram is able to maintain the distribution and gaps based on the data distribution, which allows it to have a lower average tree depth and consume less memory compared to the other two implementations.

On the other hand, the baseline only optimizes the keys based on the linear regression slope, which does not take into account the distribution of the data. As a result, it creates more child nodes and consumes more memory than the histogram. The partially sorted array delays the creation of new nodes to amortize the cost, but as the data becomes larger, it still needs to rebuild nodes recursively, resulting in more memory consumption.

Overall, the histogram is a more efficient implementation for static scenarios where the data distribution is known. Its ability to maintain gaps and distribution reduces the need for creating new child nodes and, thus, results in lower memory consumption.

8.3.4 Real-World Dataset (Longitudes)

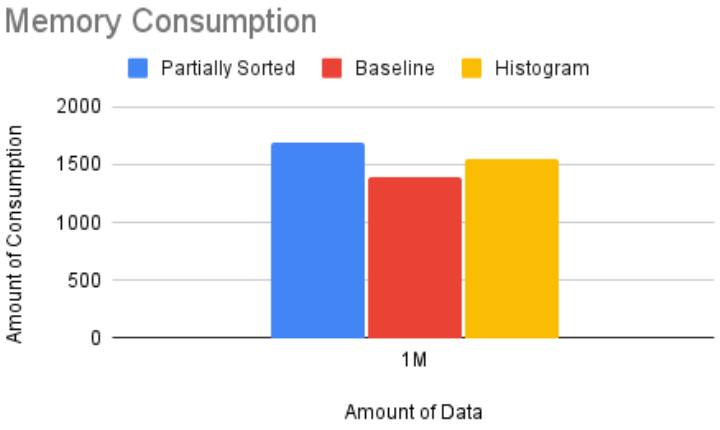

FIGURE 8.16: Results of Memory Consumption with Longitudes Distribution Dataset 1M

However, for real-world scenarios where distribution is not known, the histogram performs worse than the baseline and is not able to capture the distribution of the data since the assumption is not valid for real-world data where distribution is known for newly insert keys (Figure 8.16). Since the histogram is not able to optimize based on the distribution, the histogram will cause more conflict than the baseline which is expected and hence consume more memory.

Similar to the histogram, the partially sorted method does not depend on any particular distribution. However, it defers resolving conflicts until all available empty spaces are filled, at which point it creates new child nodes. The results indicate that the partially sorted method performs slightly better than the histogram technique because it delays creating new nodes until a later stage. However, when there is a larger amount of data in the tree, the partially sorted method does consume more resources than the histogram. This is because it waits until the current node is completely filled before recursively rebuilding it, which uses up more memory.

When dealing with real-world datasets, it is better to use the baseline approach if memory consumption is a top priority. This method is highly adaptable and generally produces better outcomes than the histogram and partially sorted techniques. Nevertheless, if the data distribution is familiar, the histogram approach can be fine-tuned to better fit the data.

8.4 Number of New Node Creation / Tree depth

In this section, we investigate the performance of three different Learned Index implementations, namely the histogram, partially sorted, and baseline, in terms of the number of new node creations or conflict when using different datasets.

The number of conflict is crucial in evaluating the performance of Learned Index algorithms, especially in dynamic scenarios, where the data distribution can change over time. For instance, when a new key is inserted into the tree, it may cause conflict and result in new node creations to maintain the performance of the algorithm. Therefore, measuring the number of conflict can provide insights into how well each Learned Index implementation adapts to changes in the data distribution.

We tested the three algorithms by performing bulk loading of one million to twenty million. After performing bulk loading we perform insertion for another one million to twenty million to test the trend and new node creation on different algorithms. Basically, we bulk load fifty percent of the keys in and insert another fifty percent later to perform the experiment, which means that one million will have two million in the tree.

8.4.1 Gaussian Distribution

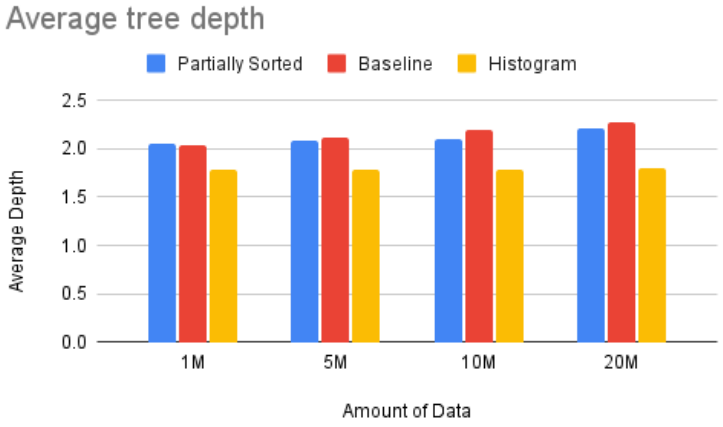

FIGURE 8.17: Results of Average tree depth after a number of insertions (Gaussian)

From the results Figure 8.17, we found that the histogram performs better on the average tree depth compared to the other two algorithms. Furthermore it has lesser conflict number (new node creation) compared to partially sorted and baseline. This is because

histogram is able to capture distributions of the data while partially sorted only delays conflict until running out of space. This is inline with other experiments from previous sections. As lesser average tree depth improve operations performance as all of the operations are bounded by the height of the tree. For example, if we are able to reduce the depth of the tree, we will be able to reduce the time to perform read and write operations.

On the other hand, from result Figure 8.17, the partially performs similarly to the baseline due, this is because the main idea of partially sorted is to store keys after the predicted position and reduce the number of new node creation. However, this is only delaying creating new node until later which will still be incured. This results inline with the operation experiments above as the amount of tree depth increase, the longer time it takes to perform read and write operation. Furthermore, the poor result on operations for partially sorted is due to the cost of recusively rebuilding the keys to be sorted.

In addition, the baseline performs worse than the histogram as the baseline implementation does not consider the data distribution, resulting in a higher number of average tree depth as it fails to optimize key positioning. From the results Figure 8.17, can see that the baseline has higher average tree depth compared to histogram, which is inline with operations performance as histogram outperform the baseline in operations as well due to having lesser depth to traverse.

8.4.2 Powerlaw Distribution

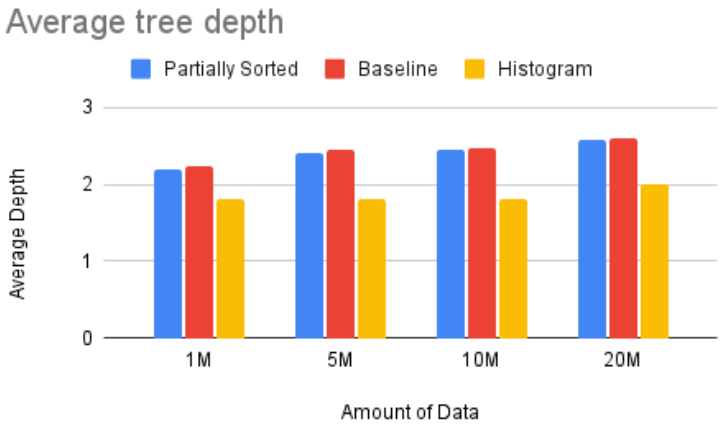

FIGURE 8.18: Results of Average tree depth after a number of insertions (Powerlaw)

The result tested on Powerlaw distribution is not different from the Gaussian distribution (Figure 8.18) where the histogram is able to outperforms both algorithm. This is because histogram can capture the distribution of the data and make use of the upper nodes gaps while the baseline just create new nodes whenever there is a conflict and only wait until the next branch pruning (or adjustment) to occupy the upper nodes.

Furthermore, partially sorted performs on par with the baseline as it only delays the new node creation until ϵ spaces runs out. From the result Figure 8.18, we can see that the trend of average tree depth slowly increasing as more keys inserted in the tree. Even though the amount of data increase almost $10\times$, but the average tree depth does not increase much because of the branch pruning method that helps to reduce the amount of tree depth. However, we can see that the amount of average tree depth still increase for both partially sorted and baseline. While histogram is able to take advantage of the empty gaps in the upper node, which will create lesser new child node.

8.4.3 Lognormal Distribution

The result on Lognormal Distribution follows the same trend as previous distributions (Figure 8.19). However the main difference from the previous test is that the baseline seem to have lesser average tree depth from five million to ten million, this is because when it reaches a condition, the baseline will trigger branch pruning which reduces the tree depth.

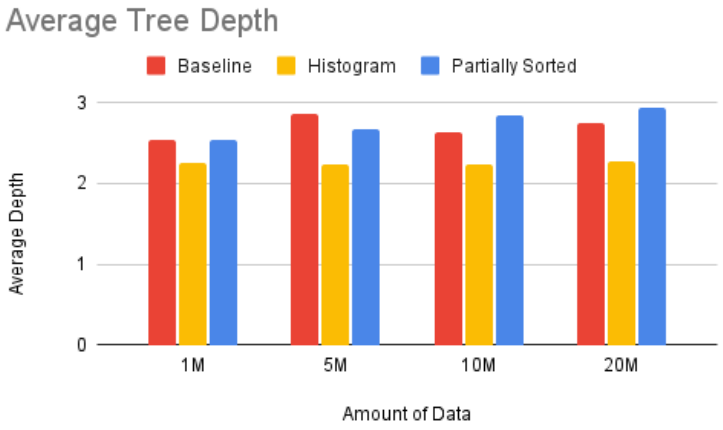

FIGURE 8.19: Results of Average tree depth after a number of insertions (Lognormal)

For histogram, we can see a similar trend where the histogram able to maintain the average tree height throughout the data increases. Which is inline with the result of insertion and search operations where it outperforms the baseline and partially sorted array. If it is able to make use of the upper nodes gaps, it will not need to wait until the branch pruning to make use of the available gaps in the upper nodes.

8.4.4 Real-World Dataset (Longitudes)

On the other hand, the results on dynamic scenario (Figure 8.20) shows that the baseline outperforms histogram and the partially sorted. Based from the result Figure 8.20, we can see the baseline is able to make use of the gaps to insert and does not increase in the average depth while histogram does not perform when it comes to dynamic scenario as histogram can not capture the distribution which is crucial for histogram to work well. This results inline with the previous test on operations where baseline is able to out shine the histogram and partially sorted. This is because the search operation is only bounded by $O(h)$ where h is the height of the tree. This mean that reducing the amount of average tree depth will make the performance of the operations faster. Furthermore, the partially sorted perform similarly to both histogram and the baseline, as it make uses of the gaps by inserted keys after the predicted position which delays the new node creation, thus having lesser depth.

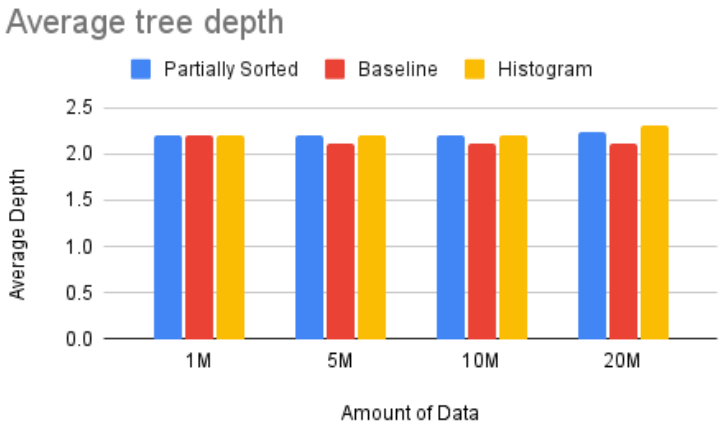

FIGURE 8.20: Results of Average tree depth after a number of insertions (Longitudes)

However this does not inline with the operations performance, this is because the operation performance of the partially sorted is bounded not just by the height of the tree,

but also ϵ spaces. Which is similar to the section on Theoretical Analysis where we mentioned that the time complexity of the operations is $O(\epsilon + \log N)$

8.5 Evaluation Summary

In this section, we have compared the performance of three learned index algorithms: baseline, histogram, and partially sorted. Our testing showed that the histogram implementation outperforms the baseline in operations such as insertion, deletion, and query when the distribution is static, and new keys follow the same distribution. This is due to the histogram's ability to minimize the height of the tree by grouping keys with similar values into a single bin, leading to faster traversal times.

In addition, measuring the number of conflict can provide valuable insights into the performance of Learned Index algorithms in dynamic scenarios. The histogram implementation performs the best in terms of the number of conflict, followed by the partially sorted implementation, while the baseline implementation performs the worst on the static scenario. On the other hand, the baseline outperforms the histogram and the partially sorted on the dynamic scenario. If the data distribution is known, histogram seems to be a better fit to optimize the gaps while performing faster on operations. If the data distribution is not known, baseline seem to perform better as it generalize to fit in many scenarios. Furthermore, the partially sorted still need more experiments and improve as the searches is still bounded by the extra search spaces after the predicted position which makes it slower during read and write operations.

However, our testing also revealed that the histogram implementation needed further exploration to improve its performance in real-world scenarios where data is not static and can change over time. This is an area where the partially sorted algorithm falls short, as it requires extra computation that causes operations to slow down as the data increases in the tree.

In conclusion, the choice of the learned index algorithm depends on the characteristics of the dataset being indexed and the expected usage patterns. For static distributions, the histogram algorithm offers the best performance, while the partially sorted algorithm may be more suitable for dynamic data. Nonetheless, further research is needed to

explore the potential of these algorithms in more diverse scenarios and to develop more advanced learned index structures that can handle more complex data distributions.

Chapter 9

Study on Parameter Tuning

In this section, we will perform in depth analysis on different parameters and the trade off between the two algorithm (Histogram and Partially Sorted). Different parameters will yield different result and base on the different dataset. The experiment method will be the same as the evaluation section where we will be using the same dataset. However, we will only limits to real-world dataset and one of the distribution due to the computation power in testing all of the distribution. We expect that the result will help gain more understanding on different parameters that the histogram and partially sorted introduced and its flaws and tradeoff.

9.1 Histogram Bin

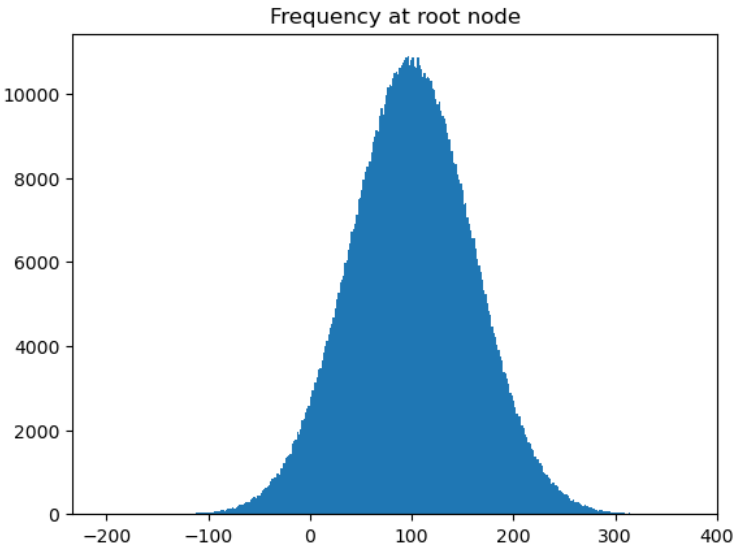

FIGURE 9.1: Distribution at root node

The choice of the bin size is important consideration when maintaining a histogram in each node. The bin size determines the number of bins used in the histogram and the range of values includes in each bin. The bin size should be chosen carefully to ensure that the resulting histogram accurately reflects the distribution of the data.

If the bin size is too small, the histogram may not be able to capture the distribution of keys in the node and difficult to interpret. This is because each bin will contain only a small number of data points, and the resulting bar heights will be highly variable. Which makes it difficult to see the underlying pattern of the data.

On the other hand, if the bin size is too large, the resulting of the histogram in each node may be too coarse and may not accurately capture the fine details of the distribution. This will result in important details to be missed out.

We tested small static bin and the FD rule to gain some insights on the data patterns and how bins affect the gaps optimization. The figure 9.1 is taken from the histogram LIPP root node with a Gaussian distribution dataset on one million keys. we can see that the root node will contain the whole tree distribution.

In this section, we will test the static bin size and Freedman-Diaconis rule to find the appropriate bin size and its tradeoff using the same dataset and setup as the previous

section. However, in our calculation of the *bin_index*, we have to find the *bin_width* which determines the range of keys that fall into this index. For example, if *bin_width* is equal to 1, that means the range of first index can be in the range of keys 0 to 1. With static bin size, we can calculate *bin_width* by $\frac{Key_{max}-Key_{min}}{bin\_size}$.

9.1.1 Gaussian Distribution

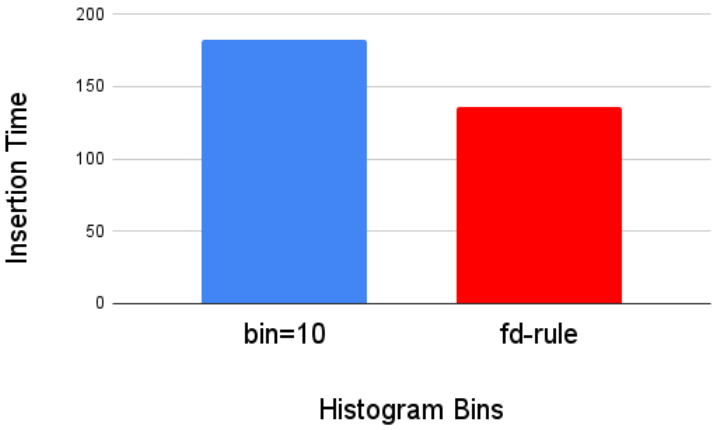

FIGURE 9.2: Results of Insertion on Different Histogram Bin Setting

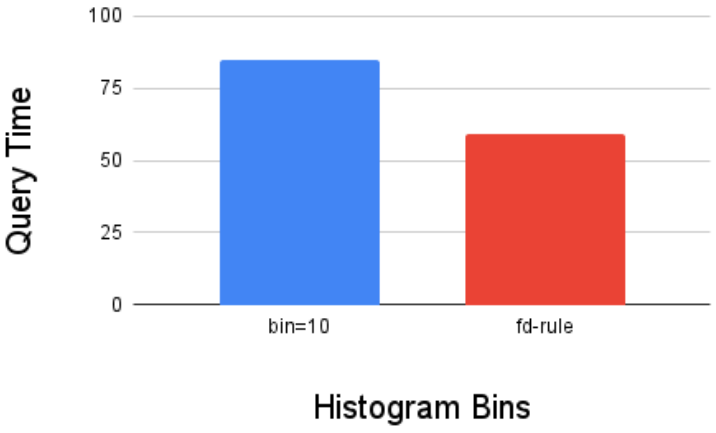

FIGURE 9.3: Results of Query on Different Histogram Bin Setting

In this section, we aim to compare the performance of a static bin size of 10 with the FD rule when used in the histogram-based Learned Index algorithm on Gaussian Distribution. We carried out the same tests as in the previous section, but with the addition of comparing the performance of the different bin sizes.

From the resulting figure 9.2 and 9.3, we can observe that using a static bin size of 10 does not perform well when there is a large amount of data in the tree. This is because the root node, which represents the entire distribution of the tree, does not capture the pattern well when the bin size is too small. As a result, the gap optimization in the tree performs poorly, causing the tree to grow deeper and impacting the read and write operations since they are bounded by $O(h)$ where h is the height of the tree. Furthermore, we can also see that both the insertion and query operations perform poorly due to the increasing height of the tree when the gap optimization does not work well. Additionally, the number of conflict (or new nodes created) also increases, as the histogram-based Learned Index relies heavily on the histogram in each node to distribute gaps according to the frequencies.

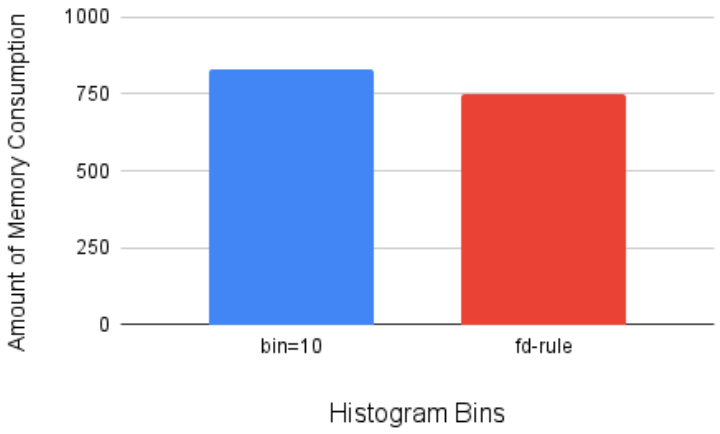

FIGURE 9.4: Results of Memory Consumption on different histogram bin setting

In contrast, the FD rule maintains a longer frequency (or bin) size, which consumes more memory compared to the static bin size. From the resulting figure 9.4, we can see that the FD rule maintains larger number of bin sizes, while the static bin size only maintains 10. This cost will be incurred when the adjustment triggers, and the algorithm has to rebuild the nodes. However, it offsets the operation cost, as the insertion operation has to traverse less depth compared to the static bin size.

This makes the FD rule performs well in term of representing the underlying patterns and able to make our gaps optimization works well on different distributions. It outperforms the static bin size when perform operations like the insertion, query and deletion due to the smaller depth that it has to traverse down.

9.1.2 Real-World Dataset (Longitudes)

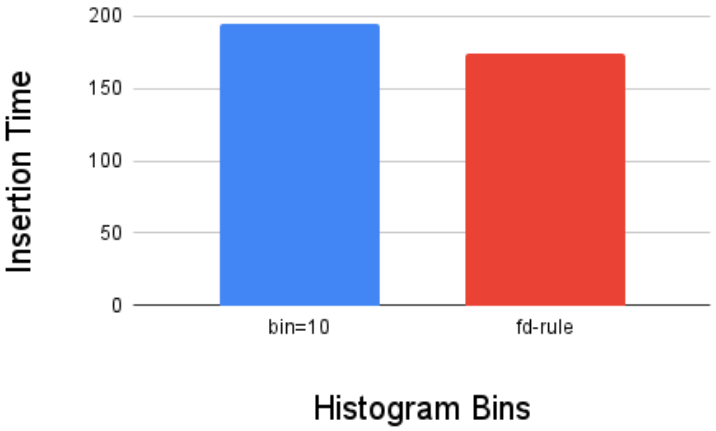

FIGURE 9.5: Results of Insertion Operation on different histogram bin setting

Real-world datasets can have a complex distribution, and unlike synthetic datasets with known distributions, it can be challenging to predict the distribution of new keys. Therefore, it is important to evaluate the performance of the Learned Index algorithms on real-world datasets to understand how they adapt to the underlying patterns.

Based on the results of our experiments on real-world datasets, we observed that the performance of the histogram Learned Index was poor compared to the baseline, as discussed in the previous section. However, in this experiment, we evaluated the performance of two different approaches, namely the static bin size and the FD rule.

The results showed that both the static bin size and the FD rule did not perform well when the distribution of new keys was unknown, as they were not able to capture the underlying patterns of the dataset. As a result, both approaches performed poorly on insertion and query operations compared to the baseline. However, the FD rule still outperforms the static bin size as it is able to capture keys' distribution better than the static bin size which overfits to only a few scenarios (Figure 9.5).

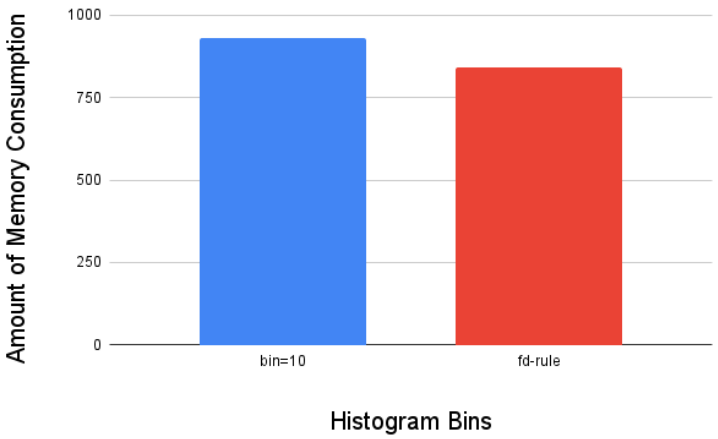

FIGURE 9.6: Results of Memory Consumption on different histogram bin setting

For memory consumption (Figure 9.6), the FD rule still performs better than the static bin size. The FD rule is able to capture the distribution better than the static bin size. Static bin size may only work in a few scenarios where there is not large amount of data. In this case, it makes static bin size create more child nodes than the FD rule which consumes more memory to maintain the child nodes.

Therefore, based on our experiments on one million real-world keys, the FD rule approach is a better solution for both real-world dataset and static dataset. This is because the FD rule is able to capture distribution and distribute gaps based on the detailed distribution while the static bin size will only works if there is lesser data and require a lot of engineering efforts to tune this parameter to work on different scenarios.

9.2 Partially Sorted ϵ Spaces

The ϵ spaces are a crucial factor in determining the performance of Learned Index, as it influences the traversal of the tree and the operation's runtime complexity. The ϵ parameter determines the number of gaps to search for in the gapped array when the predicted position is not empty. As all the operations are bounded by the ϵ spaces, the number of ϵ sets affects the search performance.

To determine the impact of the ϵ parameter on the search operation, we conducted experiments on different parameter values, namely 1, 5, and 10, using the same datasets as

in the previous section, i.e., Gaussian and Real-World datasets. We evaluated the performance of the insertion, query, and deletion operations using these different parameter values.

The ϵ parameter has a significant impact on the search operation's performance. In general, a larger value of ϵ will result in a higher number of gaps to search for, making the search operation more efficient as query has to traverse down lesser depth due to smaller tree heigit. However, a larger value of ϵ also increases the memory overhead of the index, as more gaps need to be stored.

On the other hand, a smaller value of ϵ will result in fewer gaps to search for, making the new node creation higher which will make the search operation in efficient. However, a smaller value of ϵ also reduces the number of search after the non-empty spaces.

9.2.1 Gaussian Distribution

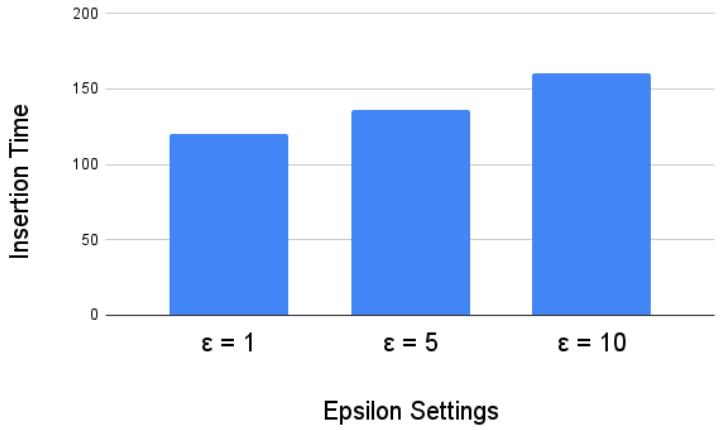

FIGURE 9.7: Results of Insertion Operation on different Partially Sorted ϵ spaces

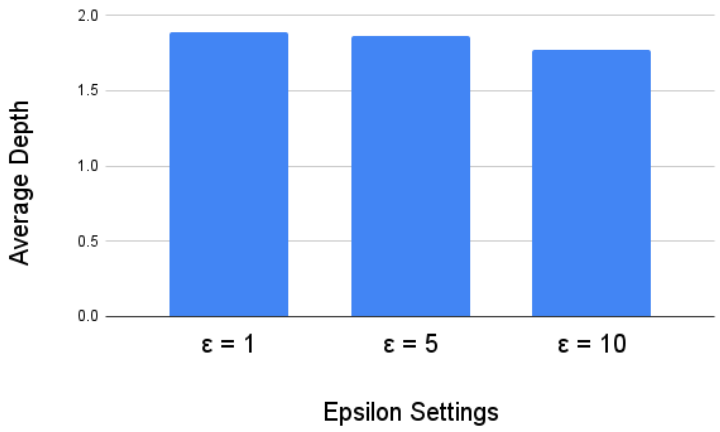

FIGURE 9.8: Results of Average Tree Depth on different Partially Sorted ϵ spaces

In the case of the Gaussian Distribution, we inserted keys with the same distribution as the keys that were already present in the tree. However, we do not expect the partially sorted data structure to perform well on this distribution as it only delays the slow operation of new node creation until the gaps are full. The results from the Insertion operation Figure 9.7 shows that the trend of results changes as the number of ϵ decreases, indicating that the insertion operation becomes faster as the number of searches it has to perform decreases. However, if we look at the Figure 9.8, we can see that the number of new node creations increases as the number of ϵ decreases. This is because it has lesser ϵ spaces for the partially sorted to delay new node creation.

In addition, querying keys in the partially sorted data structure has the same performance trend as the insertion operation. This is because they both have the same time complexity, which is $O(\epsilon + \log N)$. This complexity is bounded by the search after the predicted position. Based on our results, it seems that the time to search a key is mostly dominated by the ϵ linear search as the trend of the search performance seem to decreases while the ϵ also decreases. However the number of nodes creation inversely related to the ϵ because the smaller the ϵ, the number of lesser gaps to delay new node creation.

The importance of the ϵ parameter in the partially sorted data structure is enormous, as all operations are mostly bounded by the traversals down the tree height and the ϵ spaces (or $O(\epsilon + \log N)$). The ϵ parameter determines the number of gaps in the gapped array to search for when the predicted position is not empty. Since all operations are bounded by the ϵ spaces, the number of ϵ sets will affect the search performance.

Furthermore, its effectiveness heavily depends on the choice of the parameter ϵ, which determines the number of gaps to search for when the predicted position of a key is not empty. This choice can require significant engineering effort, and even with a well-chosen ϵ, partially sorted may not outperform other data structures.

In comparison to baseline and histogram data structures, which are only bounded by the height of the tree, partially sorted suffers from the need to delay the creation of new nodes until gaps are filled. This means that partially sorted does not assume any underlying pattern in the data distribution, but instead aims to optimize search and insertion by using the gaps as a buffer.

9.2.2 Real-World Dataset (Longitudes)

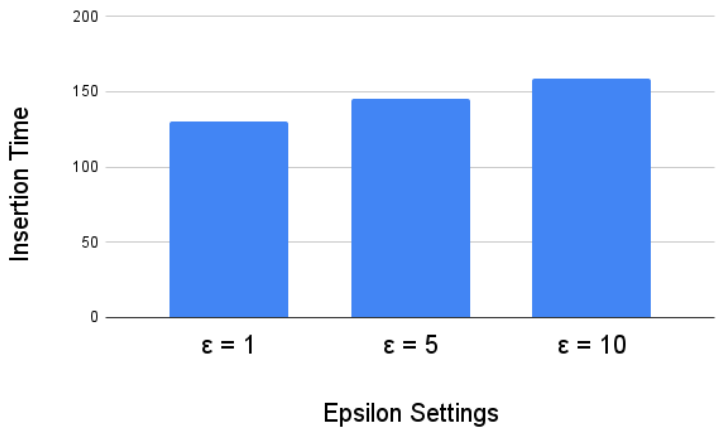

FIGURE 9.9: Results of Insertion Operation on Different Partially Sorted ϵ Spaces

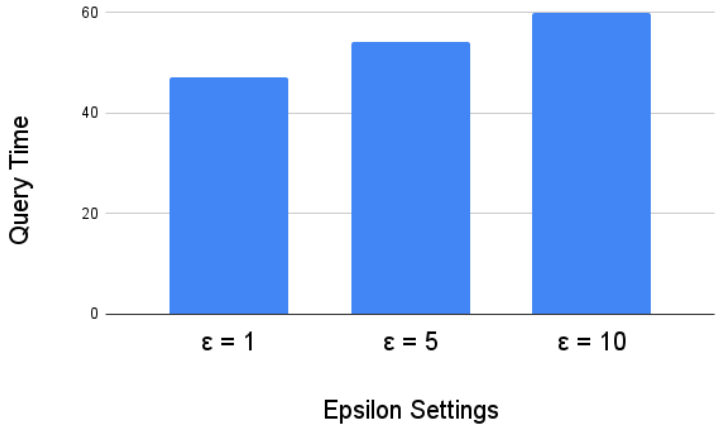

FIGURE 9.10: Results of Query Operation on Different Partially Sorted ϵ Spaces

When applied to real-world datasets, the partially sorted algorithm demonstrates a similar performance trend to that seen in static scenarios. When compared to other algorithms such as baseline and histogram, the partially sorted algorithm may not always be the most efficient. Nevertheless, this section of our study focuses on comparing the performance of different partially sorted parameter settings.

The results presented in the figure 9.9 and 9.10 clearly show that partially sorted with $\epsilon = 1$ performs best in operations such as insertion, deletion, and query. This is because this setting only requires one space search after the predicted position, which can significantly improve the algorithm's performance. Since the time complexity of the algorithm is bounded by $\epsilon + \log N$, a lower value of ϵ leads to faster operations.

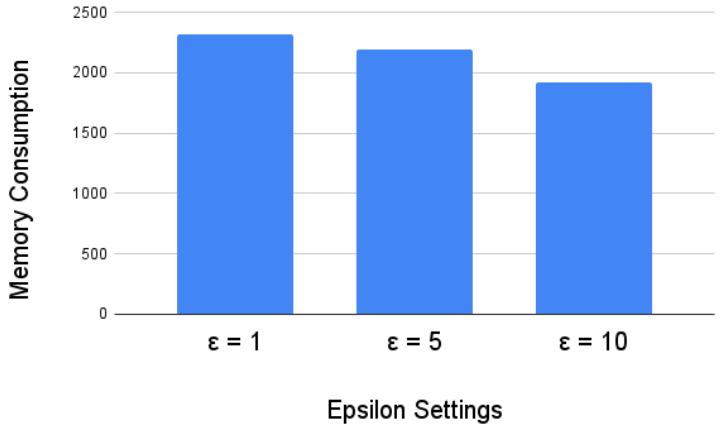

FIGURE 9.11: Results of Memory Consumption on Different Partially Sorted ϵ Spaces

However, when ϵ is decreased, the memory consumption of the algorithm increases (Figure 9.11). This is in line with our expectations, as reducing the number of spaces only delays the new node creation process for one conflict. As a result, there are more child nodes with lower ϵ values, leading to increased memory usage.

In summary, the value of ϵ in partially sorted refers to the number of limits that the algorithm can locate, which only delays the new node creation process. This algorithm does not take advantage of the underlying pattern of the keys, making its results similar for both real-world datasets and static scenarios. Therefore, selecting the optimal value of ϵ for a given dataset is critical to achieving the best performance of the partially sorted algorithm.

9.3 Tradeoff

Tradeoff is a crucial aspect of designing a Learned Index structure as it involves balancing size, accuracy, and efficiency. However, the accuracy can be eliminated by introducing new node creation in LIPP algorithm [26], which leads to exact predictions but increases the size of the structure as a new node is created each time the predicted location is already occupied. In contrast, the partially sorted algorithm introduces the accuracy tradeoff in terms of ϵ which determines the number of keys that can be located in the structure.

In this section, we will compare the tradeoffs between Histogram and Partially Sorted parameters using the results obtained in the previous section. Although the Histogram algorithm does not require an accuracy tradeoff as the predicted position is exact and the time complexity is bounded by the height of the tree. In contrast, Partially Sorted provides a balance between accuracy, size and efficiency tradeoffs, but it may not perform as efficiently as Histogram or other algorithms in certain scenarios. Therefore, it is important to consider the specific use case and requirements when deciding which algorithm to use.

9.3.1 Histogram

In the previous section on Histogram Bin, we compared the performance of different bin sizes in a static scenario with a Gaussian Distribution. We found that the histogram outperformed the baseline and partially sorted algorithms. However, we also found that the choice of bin size plays a significant role in the accuracy and efficiency of the algorithm. If the bin size is too small, the histogram will not be able to capture the distribution of keys in the tree, resulting in poor accuracy. On the other hand, if the bin size is too large, the distribution will be too coarse, also resulting in poor accuracy. Hence, there is a tradeoff between bin size and the visibility of underlying patterns.

In terms of memory consumption, a FD rule is the best option. However, this approach sacrifices the node size as it has to maintain higher size of frequencies array. Furthermore, the static bin size require fine tuning which consequently lead to more conflicts, and the Learned Index will have to create more new nodes, making operations less efficient. Tuning for a good enough bin size becomes challenging as the tree grows larger, and each

node has its own distribution, which cannot use a "one size fits all" approach. However, if we compare with the baseline, the histogram algorithm consumes more memory as it has to maintain the frequency array to distribute gaps while baseline does not have extra array in each node.

In addition, FD rule helps determine the bin size based on the keys in each node. This approach decides the bin size in each node such that it is good enough to distribute gaps based on the frequencies array. However, the tradeoff for this implementation is that it will consume more memory in each node as it has to maintain the array of frequencies. Due to variable bin size, the accuracy of distributing gaps increases, which decreases the number of conflicts. Thus, operations performance increases, as seen from the previous section.

From our experiment, we conclude that using FD rule is the best approach as it determines a good enough bin size for each node, resulting in higher accuracy and fewer conflicts. However, controlling the amount of memory consumption can be challenging. It is essential to balance memory consumption and performance to achieve the optimal solution. We argue that optimizing operations performance is the best approach since insertion and query operations occur frequently in real-world systems where users have to keep the data updated. Thus, the efficiency of these operations plays a critical role in the overall system performance.

9.3.2 Partially Sorted

In this section, we discuss the tradeoff between size, efficiency and accuracy in the context of Learned Index with partially sorted arrays. Unlike histogram bins, partially sorted arrays use the concept of partial limit spaces or ϵ to delay the slow new node creation by storing it a few ϵ spaces after predicted position. ϵ defines the number of spaces that the algorithm has to search for after the predicted position. Therefore, the operation performance like insertion is bounded by $\epsilon + \log N$, where N is the number of keys in the node. In this section, we will be experimenting with different values of ϵ to determine the tradeoff between size, efficiency, and accuracy.

From the experimental results in Figure 9.7, we observe that the operation performance of partially sorted arrays decreases as ϵ increases. This is consistent with the theoretical

analysis that suggests that the performance is not only bounded by $\log N$, but also by the ϵ spaces that the algorithm has to search. On the other hand, partially sorted arrays do not need to store any extra information like histogram bins, where a frequency array is maintained for each node. To determine if a key is partially sorted or not, we simply use the model to predict the position of the key and compare it with the current position. If the predicted position is different from the current position, then it is a partially sorted key.

In terms of operation performance, we observe that insertion is the most impacted operation by the choice of ϵ. The reason for this is the recursive rebuild operation that is triggered when the ϵ spaces are exhausted. In particular, when the number of partially sorted keys in the node is high, the number of recursive rebuild operations required also increases, thereby reducing the efficiency of the insertion operation.

Based on the experimental results, we can conclude that $\epsilon = 1$ offers the best trade-off between size and performance in terms of operations like insertion. However, as ϵ increases, the performance of the partially sorted array also improves, but at the cost of increased memory consumption due to the need to store more partially sorted keys. For example, when $\epsilon = 5$, the performance of insertion is significantly worse than when $\epsilon = 1$ due to the increased number of recursive rebuild operations required to insert a new key.

In conclusion, the choice of ϵ in partially sorted arrays determines the tradeoff between size, efficiency, and accuracy. While a smaller value of ϵ improves the performance of operations like insertion, a larger value of ϵ improves the memory consumption of the algorithm at the cost of increases in operation time. Therefore, it is important to strike a balance between these tradeoffs when implementing Learned Index with partially sorted arrays.

9.4 Maximum Gaps

In this section, we will delve deeper into an experiment to explore how different gaps distributions affect the performance and memory consumption of Histograms. This experiment is essential because the number of gaps that we distribute will affect the overall usage of memory as the gaps have to be reserved for key insertion.

In this experiment, we will test different datasets, such as Gaussian distributions and real-world datasets, to observe the changing trends and the trade-offs on different settings of gap distribution. We will also test different gap distributions such as 2×, 2.5×, and 3×. The 2× refers to double the size of keys currently collected in the adjustments. The adjustment or branch pruning is only done based on the mentioned condition. For example, if we collected keys of $[1, 2, 3]$, the size of the expanded array will be 2×3 which is 6 so there will be 3 keys and 3 gaps. However, for 2.5× the size, if the outcome of multiplication contains a decimal, we will round it up and place the leftover gaps at the end of the array.

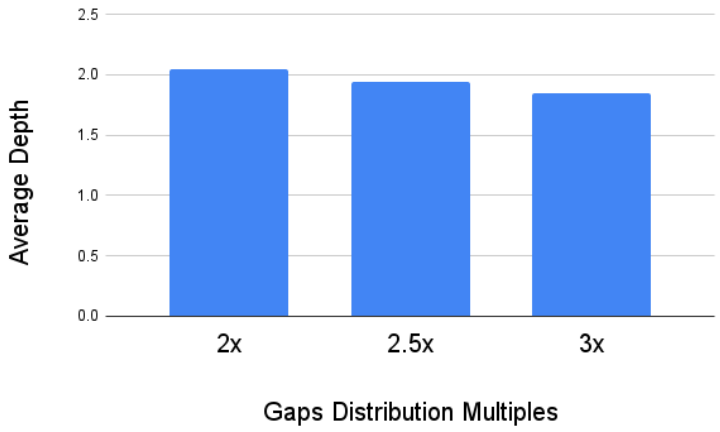

FIGURE 9.12: Results of Average Tree Height on Different Maximum Gaps Parameters (Gaussian Distribution)

In the experiment on the Gaussian distribution, the results in figure 9.12 show that distributing gaps 3× the size of keys collected performs best in terms of new node creation, as it has more gaps to locate keys in, which will reduce the number of child nodes. This setting can also increase the operation performance like insertion, as it traverses to a lesser depth than the 2× and 2.5× settings.

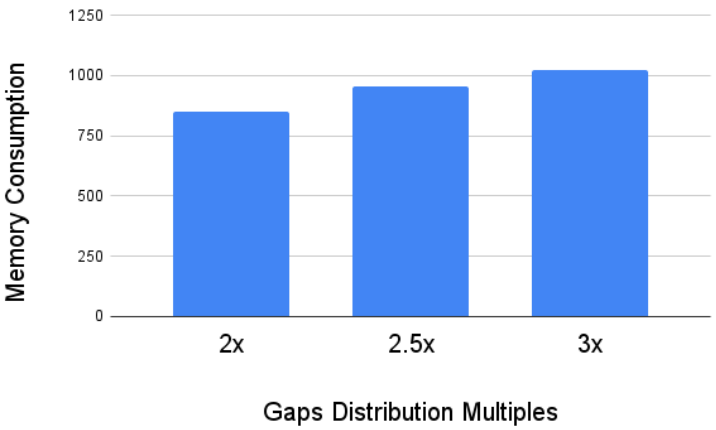

FIGURE 9.13: Results of Memory Consumption on Different Maximum Gaps Parameters (Gaussian Distribution)

However, the memory consumption is affected by the increasing number of gaps, which is a tradeoff between size and efficiency. From the memory consumption figure 9.13, we can see that the amount of memory consumed by 3× increases from 2× and 2.5×. Even though 3× has lesser tree depth, but reserving more gaps in each nodes consumes more memory than the amount of new child nodes created by 2×.

If the operation performance is the main concern, distributing more gaps can help reduce the number of new node creations while consuming more memory, as the gaps must be reserved for new keys, which is in line with the theoretical analysis that the performance is bounded by the height of the tree ($O(\log N)$). That means if we can reduce the height of the tree, we will gain performance in operations like insertion, deletion, and query. However, if memory is the primary concern, distributing more gaps will consume more memory.

From our testing, it seems that the 2× setting is the best tradeoff between operation performance and memory consumption, as the average tree height trend does not increase as much as the memory consumption. From the figure 9.13, we can see that the 3× gaps distribution increases memory consumption much more than it increases operation performance. Which makes 2× the best tradeoff in performance and size.

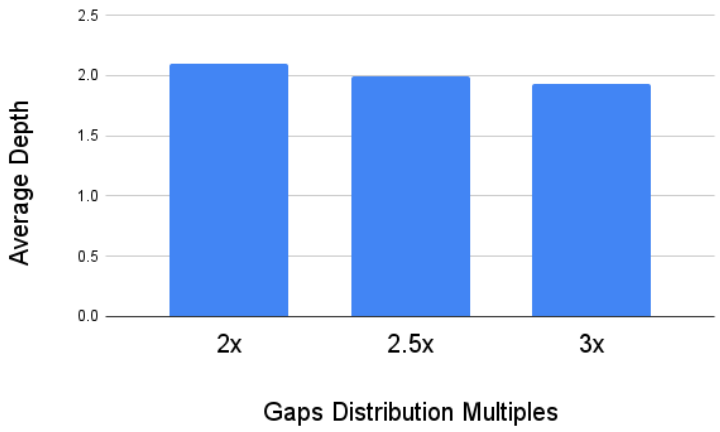

FIGURE 9.14: Results of Average Tree Height on different Maximum Gaps Parameters (Longitudes Distribution)

For the real-world dataset, a similar trend appears that the maximum number of gaps to distribute will have more gaps for the model to insert new keys into. Based on the figure 9.14, we can see that the 3× performs better in terms of average tree height as it reduces the number of new child nodes because it has more gaps for the machine learn index to place new keys in. When there are fewer child nodes, the operation performance increases, which is in line with the theoretical analysis, where the operation performance for histograms is bounded by how much the height of the tree grows.

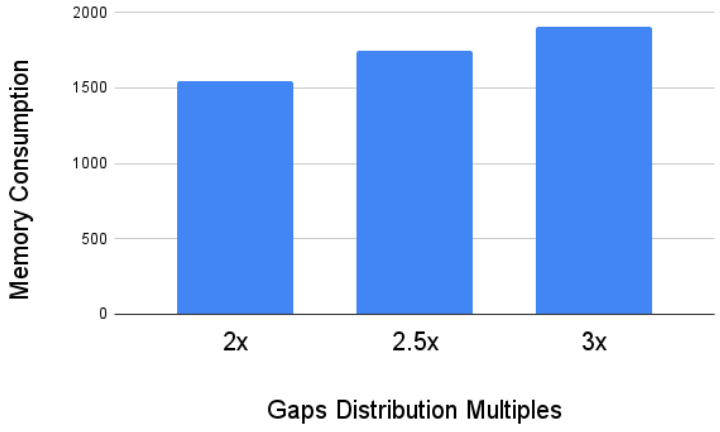

FIGURE 9.15: Results of Memory Consumption on different Maximum Gaps Parameters (Longitudes Distribution)

However, expanding more gaps 3× consumes extra memory (figure 9.15) as the gaps have to be reserved for new keys, which will consume memory. Furthermore, we can see from the figure 9.15 that the amount of memory consumption increases when we

distribute more gaps. However, the gain in operation performance is not significant and does not outweigh the fact that memory consumption trend increases higher than the operation performance. Which makes 2× the number of collected keys the best tradeoff between operation performance and memory consumption in real-world datasets.

In conclusion, distributing more gaps can help reduce the number of new node creations and increase operation performance, but at the expense of memory consumption. The optimal gap distribution is a tradeoff between operation performance and memory consumption, with 2× being the best setting for most scenarios.

9.5 Scalability

In this section, we will delve into the scalability of two indexing algorithms, namely the histogram and partially sorted index, by analyzing their performance on datasets of varying sizes. We will be using the same datasets as the previous section, but this time with an increase in the number of insertions. We will test on datasets with sizes of $10M$, $20M$, and $50M$ and also perform tests on both Gaussian distribution and real-world datasets.

The performance of an indexing algorithm is crucial when dealing with large datasets as it determines the speed of insertion, query, and building. A good indexing algorithm should be able to handle large datasets efficiently, without sacrificing performance. The scalability of an indexing algorithm is, therefore, an important factor to consider when selecting an indexing algorithm.

To test the scalability of the histogram and partially sorted index, we will perform insertion, query, and building operations on datasets of varying sizes. We will start by inserting $10M$ keys into both indexing algorithms and measure their performance. We will then repeat the process with datasets of sizes $20M$ and $50M$.

Additionally, we will test the scalability of the algorithms on two different types of datasets. The first type is the Gaussian distribution, which is a well-known distribution often used to test algorithms. The second type is a real-world dataset, which is more complex and closer to the kind of data an algorithm would encounter in a practical scenario.

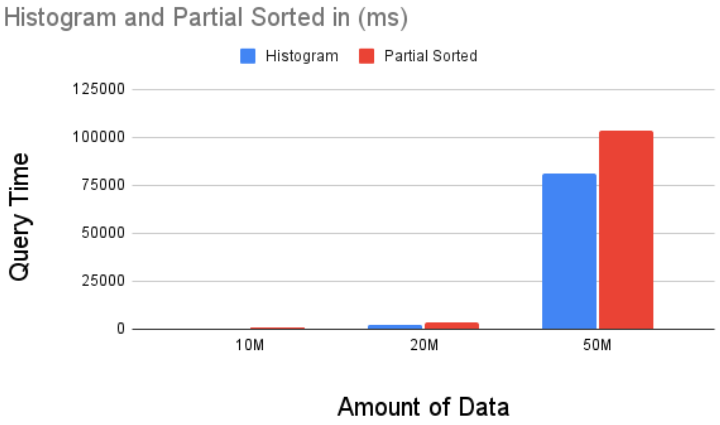

FIGURE 9.16: Results of Query Operation on different test sizes (Gaussian Distribution)

When it comes to query performance on Gaussian Distribution, the histogram has a clear advantage over the partially sorted approach (Figure 9.16). This is because the histogram, as an extension of LIPP, only needs to traverse down the tree depth to locate the keys, while partially sorted requires a search just like other Learned Index algorithms. In our tests using the Gaussian Distribution dataset, we found that query time increases as the amount of data increases and the tree height grows larger for the histogram. This is consistent with the theoretical prediction that query time is bounded by the height of the tree.

On the other hand, the partially sorted approach showed the highest growth rate compared to the baseline and histogram. This is because the algorithm has to perform a local search for ϵ after traversing down the tree depth. As per the theoretical analysis, query time is bounded by $O(\epsilon + \log N)$, which makes the partially sorted approach not scalable as the amount of data increases. When there are many partially sorted keys within each node, the performance of the partially sorted approach is affected, which makes it slower than other Learned Index algorithms.

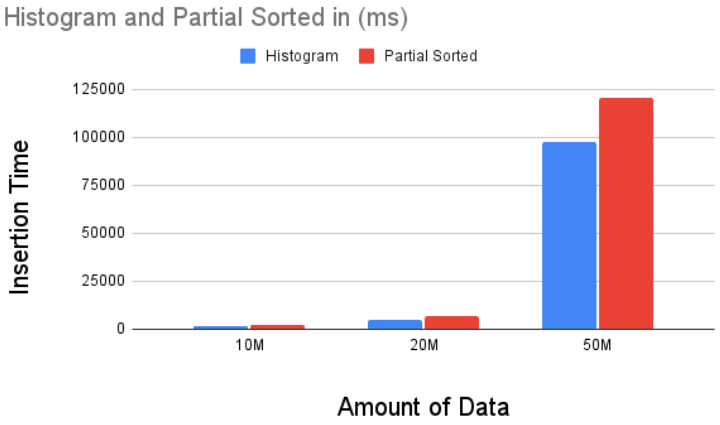

FIGURE 9.17: Results of Insertion Operation on different test sizes (Gaussian Distribution)

Insertion time shows a similar trend to query performance (Figure 9.17), where the histogram outperforms the partially sorted approach as the data size increases. When the data size is small, the partially sorted approach tends to outperform the histogram because it delays new node creation, while the histogram creates a new node only when there is a conflict. However, when the amount of data increases, and more partially sorted keys are inserted into each node, the performance of the partially sorted approach worsens as it has to perform recursive rebuilding. This slows down the insertion performance compared to the histogram since the recursive rebuilding cost is $O(\frac{N}{2})$, where N is the number of keys in the node.

When it comes to query and insertion performance in real-world datasets, we have seen that the histogram approach may not be as effective as it is in the Gaussian distribution. However, the general trend of histogram outperforming partially sorted is still observed. This is because, in histogram, we only need to traverse down the tree depth to locate the keys, whereas in partially sorted, we need to perform local search similar to ALEX[4]. As the amount of data increases, the number of partially sorted keys also increases, which in turn increases the depth of the tree. Therefore, we observe a linear increase in partially sorted performance with increasing data size. However, histogram can still create new nodes and increase the tree depth without the need for local search, making it more scalable than the partially sorted approach.

Furthermore, the trend in insertion performance is also similar to that of query performance. When the amount of data is small, partially sorted tends to outperform the

histogram as it delays the creation of new nodes. However, as more data is inserted into the tree, the number of partially sorted keys in each node increases, which triggers recursive rebuilding that can slow down the insertion performance. In contrast, the histogram approach still outperforms partially sorted because it only needs to traverse down the tree depth and create new nodes when there is a conflict, without the need for local search. This makes the insertion operation faster in histogram than in partially sorted.

In summary, the histogram approach is more scalable than the partially sorted approach for both query and insertion operations, especially when dealing with large amounts of data, as histogram does not require any extra searches after traversing down the tree compared to partially sorted keys.

Chapter 10

Conclusion

10.1 Contributions

In summary, this thesis proposes two new strategies to optimize Learned Index: the Histogram and Partially Sorted Insertion Strategy. The Histogram method aims to capture the distribution and pattern of the data by maintaining a frequency array in each node. The gaps are distributed where they are needed the most, based on the data distribution, which helps to minimize the number of child nodes and improve the operation performance.

The Partially Sorted Insertion Strategy works by delaying the creation of new nodes until the amount of ϵ is full. This approach helps to reduce the number of child nodes created and optimize the memory consumption of the Learned Index.

To evaluate the performance of the proposed methods, we conducted experiments on various parameters such as memory consumption, the number of child nodes, and operation performance, and compared the results with the baseline LIPP method. The experiments show that the Histogram outperform the baseline LIPP method and partially sorted insertion strategy in terms of memory consumption and operation performance by up to $2\times - 3\times$. We hope that our algorithms complement the existing studies on the dynamic learned index.

10.2 Limitations

Despite the advantages of the histogram and partially sorted, there are still some limitations to consider. One limitation of the histogram is its performance on real-world datasets, where it does not capture the distribution as well as the raw LIPP. This is because histogram creates child nodes whenever there is a conflict, which can significantly affect the operation performance. In contrast, LIPP creates less child nodes and performs better in capturing the distribution of real-world datasets.

Another limitation of partially sorted is that it still does not perform as well as LIPP and histogram due to the extra local search that it has to perform. While partially sorted delays new node creation, it requires additional operations to maintain its partially sorted property, making it slower than LIPP and histogram. Additionally, our research mainly focuses on the efficiency and memory consumption of these Learned Index structures, without delving into their model accuracy. Future research could explore the triangular tradeoff between model accuracy, size, and efficiency in partially sorted.

In summary, while histogram and partially sorted have advantages over other Learned Index structures, they still have limitations that need to be considered. Understanding these limitations can help researchers and practitioners choose the best Learned Index structure for their specific use case.

10.3 Future Work

In this section, we put forward some potential avenues for future research that could enhance both the histogram and partially sorted array, with the goal of further improving the performance of the Learned Index. By doing so, we hope to provide guidance for future studies that can build upon our findings and push the limits of what is possible with the Learned Index.

10.3.1 Histogram

One limitation of the histogram is that it assumes the dataset has a fixed and known distribution. In practice, this is often not the case, as real-world datasets may have

outliers and do not follow a known distribution. This limitation can affect the accuracy of the histogram, leading to suboptimal query processing performance. Therefore, there is a need to improve the histogram to support dynamic datasets with changing distributions.

To address this limitation, one possible improvement is to add a mechanism to the histogram that triggers the rebuilding process when certain conditions are met. This mechanism can be based on statistical measures such as the variance of the data or the deviation from the expected distribution. When the histogram detects that the data's distribution has significantly changed, it can rebuild the histogram with the new distribution parameters.

This improvement can benefit many database systems that rely on histograms for query processing. By adapting to the changing distribution of the data, the histogram can provide more accurate estimates of the frequency distribution, leading to better query performance.

10.3.2 Partially Sorted

One possible improvement for the current work of the Partially Sorted Insertion Strategy is to introduce a parameter to keep track of the number of partially sorted keys in each node. With this parameter, we can set a threshold for the percentage of partially sorted keys in each node. For example, if the percentage of partially sorted keys in a node exceeds 50% of the node size, then the node will perform a recursive rebuild. This threshold can be adjusted based on the specific dataset and the performance requirements of the application.

By introducing this improvement, we can reduce the number of partially sorted keys that have to be collected and rebuilt during insertion, which can improve the overall insertion performance of the partially sorted array. Moreover, it can also prevent the accumulation of too many partially sorted keys in a node, which can cause the tree to grow too deep and negatively affect the query performance.

Bibliography

[1] Bender, M. A., and Hu, H. An adaptive packed-memory array. *ACM Trans. Database Syst. 32*, 4 (nov 2007), 26–es.

[2] Comer, D. Ubiquitous b-tree. *ACM Computing Surveys (CSUR) 11*, 2 (1979), 121–137.

[3] Daniels, H., and Velikova, M. Monotone and partially monotone neural networks. *IEEE Transactions on Neural Networks 21*, 6 (2010), 906–917.

[4] Ding, J., Minhas, U. F., Yu, J., Wang, C., Do, J., Li, Y., Zhang, H., Chandramouli, B., Gehrke, J., Kossmann, D., Lomet, D., and Kraska, T. ALEX: An updatable adaptive learned index. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (may 2020), ACM.

[5] Ding, J., Nathan, V., Alizadeh, M., and Kraska, T. Tsunami: A learned multi-dimensional index for correlated data and skewed workloads, 2020.

[6] Ferragina, P., and Vinciguerra, G. The pgm-index: A fully-dynamic compressed learned index with provable worst-case bounds. *Proc. VLDB Endow. 13*, 8 (apr 2020), 1162–1175.

[7] Galakatos, A., Markovitch, M., Binnig, C., Fonseca, R., and Kraska, T. FITing-tree. In *Proceedings of the 2019 International Conference on Management of Data* (jun 2019), ACM.

[8] Guttman, A. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD international conference on Management of data* (1984), pp. 47–57.

[9] Hadian, A., and Heinis, T. Considerations for handling updates in learned index structures. In *Proceedings of the Second International Workshop on Exploiting Artificial Intelligence Techniques for Data Management* (New York, NY, USA, 2019), aiDM '19, Association for Computing Machinery.

[10] Hadian, A., and Heinis, T. Interp olation-friendly b-tr ees: Bridging the gap betw een algorithmicandlearnedinde xes.

[11] Kieseberg, P., Schrittwieser, S., Frühwirt, P., and Weippl, E. Analysis of the internals of mysql/innodb b+ tree index navigation from a forensic perspective. In *2019 International Conference on Software Security and Assurance (ICSSA)* (2019), IEEE, pp. 46–51.

[12] Kim, C., Chhugani, J., Satish, N., Sedlar, E., Nguyen, A. D., Kaldewey, T., Lee, V. W., Brandt, S. A., and Dubey, P. Designing fast architecture-sensitive tree search on modern multicore/many-core processors. *ACM Trans. Database Syst. 36*, 4 (dec 2011).

[13] Kipf, A., Marcus, R., van Renen, A., Stoian, M., Kemper, A., Kraska, T., and Neumann, T. Radixspline: A single-pass learned index, 2020.

[14] Kraska, T., Beutel, A., Chi, E. H., Dean, J., and Polyzotis, N. The case for learned index structures, 2017.

[15] Li, Y., Chen, D., Ding, B., Zeng, K., and Zhou, J. A pluggable learned index method via sampling and gap insertion, 2021.

[16] Lourenço, J. R., Abramova, V., Cabral, B., Bernardino, J., Carreiro, P., and Vieira, M. No sql in practice: A write-heavy enterprise application. In *2015 IEEE International Congress on Big Data* (2015), IEEE, pp. 584–591.

[17] Mitzenmacher, M. A model for learned bloom filters and optimizing by sandwiching. *Advances in Neural Information Processing Systems 31* (2018).

[18] Mitzenmacher, M. Optimizing learned bloom filters by sandwiching, 2018.

[19] Nathan, V., Ding, J., Alizadeh, M., and Kraska, T. Learning multi-dimensional indexes. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (may 2020), ACM.

[20] OpenStreetMap. Open street map on aws https://registry.opendata.aws/osm/.

[21] Postgres. Postgressql bloom filter index, Aug 2022.

[22] Rao, J., and Ross, K. A. Making b+- trees cache conscious in main memory. *SIGMOD Rec. 29*, 2 (may 2000), 475–486.

[23] Sahann, R., Müller, T., and Schmidt, J. Histogram binning revisited with a focus on human perception. In *2021 IEEE Visualization Conference (VIS)* (2021), IEEE, pp. 66–70.

[24] Srinivasan, V., and Carey, M. J. Performance of b-tree concurrency control algorithms. In *Proceedings of the 1991 ACM SIGMOD international conference on Management of data* (1991), pp. 416–425.

[25] Toss, J., Pahins, C. A., Raffin, B., and Comba, J. L. Packed-memory quadtree: A cache-oblivious data structure for visual exploration of streaming spatiotemporal big data. *Computers & Graphics 76* (2018), 117–128.

[26] Wu, J., Zhang, Y., Chen, S., Wang, J., Chen, Y., and Xing, C. Updatable learned index with precise positions, 2021.

[27] Zhou, K., Hou, Q., Wang, R., and Guo, B. Real-time kd-tree construction on graphics hardware. *ACM Transactions on Graphics (TOG) 27*, 5 (2008), 1–11.