Learned Index Structures: Practical Implementations and Future Directions

Author:
Taranpreet Kaur
July 17, 2018

Advisors:
Prof. Dr. rer. nat. habil. Gunter Saake,
Dr.-Ing. Eike Schallehn,
M.Sc. Gabriel Campero

Databases and Software Engineering Workgroup,
University of Magdeburg
Kaur, Taranpreet:

*Learned Index Structures: Practical Implementations and Future Directions*

Master’s Thesis, Otto-von-Guericke-Universität Magdeburg
Faculty of Computer Science, 2018.
Index structures like B-Trees, Hash index and bloom filters are data structures that are used in applications for efficient retrieval and faster access of data. If we try to create an analogy, we can think of these data structures as models where a B-Tree maps a key to a position of the record (in a sorted array), hash indexes map a key to a position in case of unsorted array and bloom filters for existence of a record. In this thesis we follow and enhance the research that was addressed in the paper “The Case of Learned Index Structures” \cite{1}. This paper begins with a presupposition that all traditional indexes could be replaced by Machine Learning models, like deep learning models and are called as Learned Index Structures (LIS). The idea behind this is to make the model learn or train the distribution function for keys and positions. Once the model is trained, it can now make predictions for the positions just like an index. The paper theoretically analyzed the use case where the Learned Index Structures outperformed the traditional Indexes and the efforts/challenges faced while designing the Learned Index Structure.

This thesis extends the analysis further by addressing some of the unanswered or open questions in the paper. We implemented the Recursive Model Index approach on the standard TPC-H Line Items dataset where the models were trained to predict position when a lookup key is fed to the model. A careful analysis has been done on designing of Learned Index Structures by experimentally implementing and evaluating the impact of hyper parameter tuning on the neural network. In order to analyze the efficiency between Learned Index Structures and traditional B-Trees/B+ Trees a comparison was made on the basis of build time, inference time and memory footprint. Our results show that Learned Index Structures performed slightly better than B+ Trees in inference time with a speed up of 3.27. The inference time is the total time taken for lookup and data fetch from index. In terms of memory consumption the size of Learned Index Structure was 126.97 KB where as B-Tree was 9,027.58 KB and B+ Trees was 2,818.05 KB in size. However due to the training time of neural networks the build time for Learned Index Structures was huge. We also believe that with the usage of GPU’s in the coming times this might be reduced further to a great extent. We also addressed the open question of insertions in Learned Index Structures by coming up with an idea and its implementation. Furthermore we do believe that replacing traditional indexes with Learned Index structures for data management will have more inferences in future and would prove beneficial while addressing other issues. As future works a novel idea has been presented to extend Learned Index structures for multi dimensional indexes.
Acknowledgements

Its a pleasure to thank all the people that made this thesis possible.

First and foremost I would like to express my gratitude to my supervisor M.Sc. Gabriel Campero. Its because of his motivation, inspiration, knowledge, his way of explaining things to me so clearly and in a simple manner that made this collaboration such an enriching, knowledgeable and fun experience. I am also thankful to him for addressing all my queries so quickly and promptly.

I am also very grateful to Dr.-Ing. Eike Schallehn for providing his valuable insights and guidance through out this thesis. He had always provided me with constructive comments and warm encouragement.

I would like to thank Prof. Dr. rer. nat. habil. Gunter Saake for giving me the opportunity to write my Master thesis under his chair. I wish to thank M.Sc. David Broneske and M.Sc. Balasubramanian Gurumurthy for their generous support.

I would also like to thank my family and friends for their constant love and support.

With this thesis my long term collaboration with Otto-von-Guericke-Universität Magdeburg comes to an end. It was a privilege to work in collaboration with the DBSE group.
vì
Declaration of Academic Integrity

I hereby declare that this thesis is solely my own work and I have cited all external sources used.

*Magdeburg, July 11th 2018*

__________________________________________

Taranpreet Kaur
# Contents

<table>
<thead>
<tr>
<th>List of Figures</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Tables</td>
<td>9</td>
</tr>
</tbody>
</table>

## 1 Introduction

1.1 Research aim .......................................................... 3
1.2 Research methodology .............................................. 3
1.3 Structure of thesis .................................................... 5

## 2 Technical background

2.1 Indexing and B-Trees .................................................. 7
  2.1.1 Binary Search Tree ............................................. 8
  2.1.2 Binary Tree ..................................................... 9
  2.1.3 Basic Operations of B-Trees ................................... 10
    2.1.3.1 Searching in a B-Tree ................................... 10
    2.1.3.2 Insertions in a B-Tree .................................. 11
    2.1.3.3 Deletions in B-Tree ....................................... 11
  2.1.4 Optimizations in B-Trees ....................................... 12
2.2 Neural Networks ...................................................... 13
  2.2.1 Machine Learning Algorithms: ................................ 13
    2.2.1.1 Linear Regression Model: ................................ 14
    2.2.1.2 Logistic Regression/Classifier Model: .................... 15
  2.2.2 Feed forward Multi-layered Neural Network ................. 15
    2.2.2.1 Perceptron ............................................... 16
    2.2.2.2 Artificial Neuron ........................................ 17
    2.2.2.3 Important terms to understand .......................... 17
  2.2.3 Training of the Neural Network - Back propagation Learning 18
    2.2.3.1 Gradient Descent ....................................... 19
    2.2.3.2 Updating the output layer weights ....................... 19
2.2.4 Activation Functions .............................................. 20
  2.2.4.1 Linear ..................................................... 20
  2.2.4.2 Sigmoid .................................................... 20
  2.2.4.3 Tanh ....................................................... 21
  2.2.4.4 Softmax .................................................... 21
## Contents

2.2.4.5 Reactified Liner .................................................. 22
2.2.5 Hyper parameters .................................................. 22
  2.2.5.1 Architecture based on the input data ...................... 23
  2.2.5.2 Layer Size .................................................. 23
  2.2.5.3 Magnitude : Momentum and Learning Rate .............. 24
  2.2.5.4 Regularization ............................................. 24
  2.2.5.5 Number of Epochs and Mini batches ..................... 24
2.2.6 Loss Functions .................................................. 25
  2.2.6.1 Loss Functions for Regression Models ................... 25
  2.2.6.2 Loss Functions for Classification Models ............... 26
  2.2.6.3 Activation Functions ...................................... 27
2.3 Learned Index Structures ......................................... 27
  2.3.1 Range Indexes ................................................ 28
    2.3.1.1 Naive Learned Index .................................. 29
    2.3.1.2 Recursive Model Index ................................ 29
    2.3.1.3 Hybrid Indexes ......................................... 30
    2.3.1.4 Search Strategies ...................................... 31
    2.3.1.5 Results ................................................ 31
  2.3.2 Point Indexes ................................................ 32
    2.3.2.1 Results ................................................ 33
  2.3.3 Existence Indexes ............................................. 33
    2.3.3.1 Learned Bloom Filters ................................ 34
2.4 Summary : .......................................................... 35

3 Prototypical implementation and research questions .......... 37
  3.1 Research question ................................................. 37
  3.2 Learned index structure implementation ..................... 39
    3.2.1 Experimental setup ....................................... 39
    3.2.2 Dataset .................................................. 40
    3.2.3 Learned B-Trees .......................................... 40
    3.2.4 Learned Bloom Filters ................................... 43
  3.3 Summary .......................................................... 44

4 Hyper parameter tuning and inference results ................. 47
  4.1 Research questions ............................................. 47
  4.2 Hyper parameter tuning results ................................ 48
    4.2.1 Impact of learning Rate .................................. 49
    4.2.2 Impact of density/architecture of neural network ....... 50
    4.2.3 Impact of batch size ..................................... 51
    4.2.4 Impact of epochs .......................................... 52
  4.3 Inference of learned index structures versus B-trees ....... 53
    4.3.1 Build time ................................................ 53
      4.3.1.1 Learned index structures ................................ 53
      4.3.1.2 B-Trees .............................................. 54
## Contents

4.3.2 Query execution time/inference time ........................................ 55
  4.3.2.1 Key lookups .......................................................... 56
  4.3.2.2 Lookup + fetch from array ........................................ 56
  4.3.2.3 Lookup + fetch from B-tree ....................................... 57
4.3.3 Memory utilization .......................................................... 58
  4.3.3.1 Learned index structures .......................................... 58
  4.3.3.2 B-trees ............................................................... 59
4.4 Summary ............................................................................. 59

5 Insertion in learned index structures .............................................. 63
  5.1 Research questions ........................................................... 63
  5.2 Our proposal: Batch insertions into the learned index structure ....... 64
    5.2.1 First approach .......................................................... 64
    5.2.2 Second approach, based on offsets ................................ 65
    5.2.3 Searching on the learned index structure after the batch insertions 66
    5.2.4 An end-to-end example ................................................. 66
  5.3 Evaluation of design alternatives ............................................ 69
    5.3.1 Model implementation with subtracting the offset ............... 69
      5.3.1.1 With random weights ............................................ 69
      5.3.1.2 With initialized weights ...................................... 70
    5.3.2 Model implementation without offset ................................ 70
      5.3.2.1 With random weights ............................................ 70
      5.3.2.2 With initialized weights ...................................... 71
  5.4 Summary ............................................................................. 72

6 Related work and Future Directions .............................................. 75
  6.1 Related Work ................................................................. 75
    6.1.1 B-Trees and its variants ............................................. 75
    6.1.2 Software 2.0 ............................................................. 76
    6.1.3 Hyper parameter tuning .............................................. 76
      6.1.3.1 Automatic Hyper parameter tuning ......................... 76
      6.1.3.2 AutoML .............................................................. 77
    6.1.4 Extensions to Learned Index Structures ......................... 77
      6.1.4.1 Criticism: LIS(blog post by Thomas Neumann) ............ 77
      6.1.4.2 Criticism: Classical Data Structures That Can Outperform Learned Indexes (by Peter Baillis) .... 77
      6.1.4.3 A Model for Learned Bloom Filters and Related Structures ............................................. 78
  6.2 Future Work ........................................................................ 78
    6.2.1 Multi Dimensional Index ............................................. 78

7 Conclusions ........................................................................... 81
  7.1 Our contributions ............................................................. 82
  7.2 Threats to validity .............................................................. 85
List of Figures

1.1 Process diagram of different phases in the CRISP-DM methodology [2] . 4

2.1 Binary Search Tree [3] ................................. 8
2.2 Binary Tree [4] ................................. 9
2.3 Insertions in a Binary Tree [4] ............................. 11
2.4 Deletion in B Trees [5] ................................. 12
2.5 Plot for Linear Regression Model [6] ........................ 14
2.6 Plot of Logistic Function [6] ............................. 15
2.7 Multi layered Neural Network Topology [7] .......................... 15
2.8 Single Layered Perceptron [8] ............................ 16
2.9 Algorithm for Back Propagation [6] .......................... 18
2.10 Gradient Descent [9] ................................. 19
2.11 Linear Activation Function [6] ............................ 20
2.15 Cross Entropy loss with respect to Predicted Value (Expected label is 1) [10] 26
2.16 B-Tress as Regression Models [1] .......................... 28
2.17 Recursive Model Index [1] .............................. 30
2.18 Hybrid End to End Training [1] ........................... 30
2.19 Hash Maps vs Learned Hash Maps [1] .......................... 32
2.20 Results of Lookup Time for Hash Maps [1] .......................... 33
2.21 Bloom filter with learned function [1] .......................... 34
2.22 Bloom filter as a Classification Problem .......................... 34
3.1 Recursive model index for implementation of Learned B-Trees ........ 41
3.2 Neural network for the parent classifier .............................. 41
3.3 Neural network implementation code for parent ....................... 41
3.4 Neural network for child linear regressors ........................... 42
3.5 Neural network implementation code for child model ................. 42
3.6 Result of predictions using the Recursive Model Index Framework .... 43
3.7 A classifier for a Learned Bloom Filter ............................. 43
3.8 Neural network implementation for classification ..................... 44
3.9 Results for bloom filters as a classification Problem ................. 44
4.1 RMSE and MSE for different learning rates .......................... 49
4.2 Impact of the number of hidden layers and neurons per layer on the learning [11] ............................................................. 50
4.3 RMSE and MSE for different architectures of the neural network ....... 51
4.4 RMSE and MSE for different batch sizes .............................. 52
4.6 Time taken by the parent model to train .............................. 53
4.5 RMSE and MSE for different number of Epochs ....................... 53
4.7 Time taken by the child model to train ............................... 53
4.8 Time taken by all the models to load .................................. 54
4.9 Time taken for creation of a B-Tree .................................... 54
4.10 Build time comparison between the learned index structures and B-trees 54
4.11 Build time comparison between the learned index structures and B-trees 55
4.12 Inference time comparison between learned index structure and B- trees 56
4.13 Inference for sequential access patterns between learned index structures and B-trees .............................................. 58
4.14 Inference for random access patterns between learned index structures and B-trees .............................................. 58
4.15 Memory footprint comparison between learned index structures and B-trees 58
4.16 Size comparison graph for all indexes ................................... 59
List of Figures

4.17 Summary table for comparing structures (Learned Index Structures V/S B-Trees) ........................................... 60

5.1 Model architecture for how we propose insertions could be done on the learned index structures .................................. 64

5.2 Model architecture for how we propose insertions could be done on the learned index structures, using an offset .................. 65

5.3 An example for insertions in learned index structures with our concept ................................................................. 67

5.4 Results for the insertion concept ......................................................... 68

5.5 Line items original, Line items new (Offset, Random weights) ............ 70

5.6 Line items original, Line items new (Offset, Initialized weights) ........ 70

5.7 Line items original, Line items new (without Offset, Random weights) . 71

5.8 Line items original, Line items new (without Offset, Initialized weights) 71

5.9 Table depicting MSE for sub child models following alternative designs for the insertion concept .................................. 71

6.1 Concept for Multi Dimensional Index using Learned Index Structures .... 79

6.2 Model Training for multi dimensional Index ........................................ 79

7.1 Summary table for comparing structures ........................................... 84

7.2 Table depicting MSE for Sub child Models ................................. 85
List of Tables

2.2 Architecture suggestions based of the input data [6] .................. 24
2.4 Activation functions based of the output data [6] ..................... 27
1. Introduction

Indexes are data structures which are used for efficient retrieval or access of data. They are crucially important for data management, as they speed up these retrieval tasks for data systems, by reducing the disk access or cache usage of queries. In relational databases indexes are created on columns of the tables present in the database. According to how they integrate with the database indexes can be either primary or secondary, based on the columns used (i.e., with primary ones associated with the primary key of items in a table). Closely related to this distinction, indexes can also be clustered or non-clustered, defining whether the index preserves the exact same order of the data in storage (i.e., a clustered index, for which there is usually one clustered index per table) or not (i.e., a non-clustered index which keeps a level of indirection, providing logical sorting on the data before).

We have different kinds of indexes built for specific queries:

- B-Tree [12] can be used for range-based lookups. These trees are self-balancing trees that aim to perform lookups and selections with logarithmic time complexity. Whenever a new item is inserted or deleted, the number of child nodes changes and a re-balancing the of B-Tree is performed. Usually, indexes of this kind offer a hierarchical arrangement of data, where the last level defines where data is located. The last level nodes in indexes can be like a lookup table of two blocks, wherein the first block is the lookup key which is stored in a sorted manner for easy retrieval of values, and the second block consists of pointers that point to the address or location of the data indexed through the lookup key.

- Hash Maps [13] are used for key-based lookups. This is supported by an associative array structure in which each relevant key points to a particular value. Hash functions are used to map each key to a significant position. A good hash function will map each key to a unique position. Else if the hash function maps the keys to the same position (also known as collisions) then the keys now need to be stored
in the form of a linked list at the same mapped position. Traversing through a linked list may be equivalent to a cache miss.

- Another kind of indexes are existential indexes, one of this kind is a Bloom Filter [14, 15], which have a guarantee for true positives (i.e., it can guarantee that if a value found to not be present in the filter, then it is not present in the database), but false positives could mean that there may be a chance that lookup keys might not be present.

In spite of their benefits, indexes come with the cost of more storage space, and extra work for their maintenance and additional writes. Furthermore, all these indexes are general purpose and will not fit well to the worst case distribution of data. For instance (as pointed out by Kraska et al. [1]) if there is a system that needs to store and query fixed length data that consist of continuous keys, then creating an index on keys would be useless since the keys themselves can serve the purpose of an offset. This transforms a lookup operation from $O(\log n)$ to $O(1)$, reducing memory from $O(n)$ to $O(1)$. The takeaway from such observation is that knowing the distribution of data could enable similar optimizations.

Nonetheless achieving such specialized optimizations is hard to manage for real world datasets, which are random and don not necessarily fit a particular pre-defined data pattern for which an optimization has been learned. If we try to build specialized solutions on a per case basis, the engineering efforts for the structures would be large and would amount to an additional cost for the index. Considering the engineering efforts required for building solutions specialized to a dataset, the authors of the paper “The Case of Learned Index Structures” suggested that specialized indexes could be created in a manageable way, by using machine learning models for learning the patterns and correlations within the data, while keeping comparatively low engineering costs [1].

The essential feature of such proposed structures is that the query results could be reconstructed with a limited memory footprint, with the machine learning models acting as a form of compression.

The central enabling idea for building learned index structures is the concept of a hierarchy of models, where at the root of the hierarchy a lookup key is fed to the model and that model predicts the next model for the look up. This process continues until a leaf node is found, which is akin to a page from a B-tree, where lookup can be performed and a position is returned that corresponds approximately to the position of the intended value. Building on this organization, which Kraska et al. [1] call a recursive model index, authors propose and evaluate implementations of learned B-trees, learned hash maps and learned bloom filters.

Unfortunately the paper of Kraska et al. provides limited information that could enable readers to replicate their findings. The datasets used are not traditional nor open source. Furthermore, there is insufficient information about the configuration of the neural
1.1. Research aim

There is an important list of unanswered questions raised by the work of Kraska et al. [1]. We have chosen a few, which serve as the focal points for this Thesis:

1. Hyper-parameter impact in accuracy: What is the impact of hyper parameters in the resulting accuracy for these structures? What is their resulting memory efficiency? What are the exact training times?
2. Comparison of training time and B-tree build times: How does the best training time achieved for the learned structures compare to building a B-Trees Index?
3. Performance comparison for Inference time: How does the inference time of a hand-tuned implementation, optimized through the use of SIMD, compare to the lookup time for B-trees? How does the performance compare when fetch time is added?
4. Insertions: How can insertions happen in learned indexes?

1.2. Research methodology

In order to solve data science problems the main hindrance for success is the gap between the ability to understand a problem and the methodology required to address the problem. To fill this gap, Data Science methodologies are required that provide some strategy for solving problems. These methodologies are completely independent of tools/technologies, and they serve as a structural framework for implementing methods in order to get the desired results. For this work we comply to the CRISP-DM methodology (one of the Data Science methodologies), in order to find answers to the above mentioned research questions.
CRISP-DM stands for cross-industry process for data mining. This is a structured framework to approach a data mining project.

The CRISP-DM model is shown in Figure 1.1. This model depicts sequential events that happen during a data mining process. These events may not happen in the order mentioned or may be required to perform again for back tracking and better results. The phases are as described below:

- Business understanding: In this phase, the goal is to determine the objective that needs to be achieved. There needs to be a clear distinction between what is desired and what can be achieved. This amounts to the identification of the important factors that influence the outcome. If this process is ignored then unrealistic requirements could lead to a waste of engineering efforts.

- Data understanding: This phase deals with the acquisition of the data for the project. Data description reporting, data quality and exploring data are some of the process that are taken care in this phase.

- Data preparation: In this phase selection, cleaning, construction and integration of the data take place.
1.3. Structure of thesis

This thesis is structured as follows:

- **Modeling**: During this phase the modeling technique to be used, along with the documentation of the model, and the assumptions made need to be determined. Here we also generate the test cases, build and access the model.

- **Evaluation**: In this stage we determine how close the model meets the business requirements. This phase is to determine if there is any business reason that makes the resulting model insufficient to meet the requirements. In this stage all aspects of quality assurance need to be considered.

- **Deployment**: All the knowledge gathered in the previous phases is put together and the final product is deployed for further use.

### 1.3 Structure of thesis

This thesis is structured as follows:

- **Chapter 2** provides an overview on the topics covered by our work, such as B-tree index structures and machine learning algorithms. We specifically take a close look at the aspects of hyper parameter tuning in neural networks. Following the CRISP-DM process model, in this chapter the business requirements are analyzed through a study of the paper “The Case for Learned Index Structures” [1].

- **Chapter 3** documents the prototypical implementation of the models used to construct the learned index structures. This addresses the modeling phase of the CRISP-DM process model, and it also explains the data preparation steps.

- **Chapter 4** provides our study for hyper parameter tuning. This corresponds to both the modeling and evaluation phases of CRISP-DM. In this chapter we also compare between the learned indexes and alternatives, such an implementation of B-trees, and the PostgreSQL indexes. We consider for this comparison parameters like build time, inference time and memory footprint.

- **Chapter 5** includes our concept for insertions, answering to the final of our research questions. In addition we evaluate, with regards to training time, two different configurations for this process.

- **Chapter 6** collects a brief selection of related work that provides context to our study. This section also includes our discussion of future work, where we outline a novel concept for managing multi-dimensional data using learned index structures.

- **Chapter 7** concludes our work by summarizing our study and findings. We disclose in this section threats to the validity of our work.
2. Technical background

In this chapter we provide a detailed technical background for our work. We present fundamental concepts on traditional indexes, such as B-Trees. We follow by introducing the building blocks of learned index structures (LIS), neural networks. In addition we discuss the impact of hyper-parameters (e.g. learning rate, number of hidden layers, epochs, activation functions, etc.) in the training of such networks. To conclude the chapter we review the learned index structures in detail, building on the work published thus far. With this chapter we cover the first phase of the CRISP-DM methodology which is business understanding.

We structure this chapter as follows:

- In Section 2.1 we discuss about indexing in general, and then we discuss a traditional indexing structure: B-Trees. We describe how B-Trees index the data and how insertion and deletion are performed in such structures. We also review optimizations proposed in the literature to the basic implementation.

- In Section 2.2 we discuss deep learning as a machine learning technique, based on neural networks. We present the basic architecture of a neural network, and how the model actually works for machine learning tasks. We also provide a careful list of hyper-parameters, and a review from the literature on how each hyper parameter can affect the neural network.

- Section 2.3 introduces in detail the learned index structures. Here we study carefully the proposal for this approach, highlighting some relevant aspects.

2.1 Indexing and B-Trees

For databases, indexes are a way to optimize the performance of common operations such as scanning, querying, manipulating or filtering data. As data structures, indexes
2. Technical background

serve a role similar to a book index: they can locate the position of a record faster than scanning the complete dataset in search for the record. Assuming a relational data model, indexes can be created using one or more columns of a database table. The basic structure of an index consists of a search key and some referenced data (sometimes also called the payload). The search key can be the primary key of a table (generally stored in a sorted order) and the referenced data consists of pointers that point to the address where the record with the given search key is located.

Jin discusses about index structures [16].

In this section we study one very popular type of index structure designed to work on storage devices. They are B-Trees and also known as Balanced Search Trees.

2.1.1 Binary Search Tree

A Binary Search Tree is a linked Data Structure wherein each node is an object and has a left node, right node and a pointer that points to its children and parent. Only the root node doesn’t have have a parent, otherwise if a node doesn’t have children then the attribute is set to NIL.

![Figure 2.1: Binary Search Tree](3)

The data(keys) in Binary Tree is stored in such a way that it should always satisfy the Binary search Tree Property [4]:

“Let x be a node in a binary search tree. If y is a node in the left subtree of x, then y.key ≤ x.key. If y is a node in the right subtree of x, then y.key ≥ x.key.”

This property is visible in the above diagram as the left tree (5,2,8,6) is less than the root node (10) and the right tree (15,13,19) is greater than root node. The most basic operation of a Binary Search Tree is the search operation within the tree. This property forms the basis of the B-Tree indexing. The algorithm for search operation in a Binary
Search Tree is pretty straightforward and easy. This is a top down approach and starts at the root node. The intention is to find the shortest path down towards the leaf nodes. For each node \( n \) it checks if \( n.\text{key} \) is equal to the key we are trying to find. If key is smaller than it goes to the left subtree, if key is greater than it moves to the right subtree and if key is equal the process is terminated.

### 2.1.2 Binary Tree

Binary Trees can have any number of children. Considering the example below if a node in a B-Tree has \( k \) number of keys then it will have \( k+1 \) children. Also the number of keys (\( k \)) within a node divides the whole range handled by the children of the node into subranges (\( k+1 \)).

![Figure 2.2: Binary Tree](image)

If we think of a typical B-Tree application the amount of data that is handled by B-tree is so huge that it cannot fit into the main memory. It is the responsibility of the B-Tree algorithm to copy the pages of the required data into the main memory in order to use it and then write it back to the disk with the updates that have been made. If the lookup object is on the disk and not in the main memory then B-Tree also has to perform DISK-READ and DISK-WRITE operations along with the search operations.

A B-Tree is a tree with the following:

- Every node \( n \) has the following properties:
  - Node \( n \) has \( n.k \) number of keys.
  - These keys (\( n.k \)) \( n:\text{key}_1; n:\text{key}_2; : : ; n:\text{key}_n \) are stored in decreasing order such that \( \text{key}_1 \leq \text{key}_2 \leq \text{key}_3 \) and so on.
  - \( n.\text{leaf} \) is a boolean value, set to TRUE if \( n \) is a leaf node otherwise set to FALSE.

- Each node has \( n.k \) keys and \( n.k+1 \) pointers pointing to its children. Leaf nodes don’t have children.

- The keys \( n.\text{key} \) separate the ranges of keys stored in the subtrees.
• The tree’s height h is equal to the leaves depth.

• All nodes have a lower and upper bound for the number of keys. These limits can be expressed in terms of a fixed integer t called the Minimum degree of the B-Tree.
  
  - Every node other than the root must have at least t-1 keys. Every internal node other than the root thus has at least t children. If the tree is nonempty, the root must have at least one key.
  
  - Every node may contain at most 2t-1 keys. Therefore, an internal node may have at most 2t children. We say that a node is full if it contains 2t-1 keys.

2.1.3 Basic Operations of B-Trees

2.1.3.1 Searching in a B-Tree

Searching in a B-Tree is similar to the Binary Search Tree with the only difference that in a Binary search tree we perform a two-way search and in a B-Tree we perform a multi-way search depending on the number of children. An algorithm for performing a search operation in a B-Tree is shown below:

B-TREE SEARCH (n,k) \[4\]

1. \(i=1\)
2. while \(i \leq n.p\) and \(k > n.key_i\)
3. \(i=i+1\)
4. if \(i \leq n.p\) and \(k == n.key_i\)
5. return \((n.i)\)
6. elseif \(n.leaf\)
7. return NIL
8. else DISK-READ \((n,c_i)\)
9. return B-TREE SEARCH \((n,c_i,k)\)

In this algorithm, B-tree takes the root node and the key to be searched as the input parameter. It then checks if the loop variable \(i\) is less than or equal to the number of keys in the root node and if the key to be searched is less than the node key. If both conditions are true it moves to the next key. It performs an if function on the condition that the loop variable is less than the number of nodes and the key we are searching is equal to the node key. If these conditions are true then the node and the index are returned. If its false then it checks if the node is the leaf node and doesn’t have any child node and returns NIL. Else it performs a disk read function. The B-TREE-SEARCH procedure therefore accesses \(O(h) = O(\log_t p)\) disk pages, where h is the height of the B-tree and p is the number of keys in the B-tree.
2.1. Indexing and B-Trees

2.1.3.2 Insertions in a B-Tree

Insertions in a B-Tree is not as simple as insertions in a Binary Search Tree because in a Binary Search Tree we just try to find the leaf position where we can insert the new key. So basically we are creating a new node and making an insertion. However in the case of a B-Tree this is not possible as it would be an invalid tree. Instead we try to insert a new key in the existing leaf node. This is not possible if the node is full, so there is a concept of splits for a full node \( n \) having \((2t-1)\) keys around the median key \( n.key_t \) into two nodes having \( t-1 \) keys in each node. The median key then moves up the tree into \( n \)'s parent in order to decide the new split. One thing that we need to make sure while splitting is that the parent is not full.

![Figure 2.3: Insertions in a Binary Tree](image)

The process of splitting up a child node takes input parameters as a non-full node \( n \) and index \( i \) such that \( n.c_i \) is a full child of \( n \). This procedure then splits the child into two and adjust \( n \) in a such a way that it has an additional child. Initially the root of the child is set to be an empty node thereby growing the tree by the height of 1. In the given example a full node \( y \) is split through the median point which is \( S \) into 2 child. The median \( S \) is then shifted to the parent of \( y \) as a new key of node \( x \).

The time complexity of the insertion algorithm is \( O(h) = O(t \log p) \), where the coefficient \( t \) is due to the operations performed for each node in the main memory. In practice, the time demand is dominated by the number of disk reads and writes needed, which is \( O(\log p) \).

2.1.3.3 Deletions in B-Tree

Deletions in B-tree are little more complicated but are analogous to insertions. Deleting a key from node can be done easily but we have to rearrange the child node as now the number of keys will be less and so should be the child nodes and the sub trees. The tree created after deletion of the nodes should be such that it doesn’t violate the B-Tree properties.

The leaf node consisting of the Value to be deleted is searched. The non-leaf nodes consisting of the same value are not deleted. The node consisting of the key is not
deleted since it may be a possibility that the particular node may consist of other keys, so only the key is deleted. We also have to keep in mind that the deletion doesn’t shorten the node much maintaining a minimum of $t-1$ keys (except root node). Once the deletion happens then we need to check the number of keys left in the leaf node. If the number of keys in the leaf node is equal to $t$, then we are good to go but if the number of keys are $t-1$ then a re-balancing operation is required.

Here in the above figure 53 is deleted and node $y$ needs a re-balancing as it has $t-1$ keys, so we make use of node $z$. If $z$ consist of $t+1$ keys, then all the keys are transferred to $y$. The pointer to $z$ and one router value from the parent of $y$ and $z$ nodes are removed. This is called fusing. If the parent $x$ has less than $t$ children after fusing, we must continue the re-balancing process in $x$, which is a non-leaf node. If fusing is applied to two non-leaf nodes, the router value between the pointers to these nodes in the parent is removed from the parent and added to the fused node. The time complexity of the deletion algorithm is $O(h) = O(t \log t)$.

2.1.4 Optimizations in B-Trees

So far we have studied the basic B-Tree structure, but there are several optimizations available for B-Trees for instance compression techniques like describing multiple records together, storing less bytes per record, for search operations comparing less bytes and avoiding fragmentation and wasted space. Some of the optimizations in B-Trees are listed below:

- **Interpolation Search**: Interpolation search predicts/estimates the position of the key by using linear interpolation based on the minimum and maximum key value of the interval. Worst case performance will be of a linear search in cases where the distribution of keys is non-uniform.

- **Variable length records**: B-Trees can support variable length records and variable length separator keys. The design of variable length records in a fixed page requires an indirection vector. This vector consists of fixed size entries where one record is represented by exactly one entry. Entry comprises of a byte offset (of the record) and some additional information like the size of the record. For search operations the search is performed on the vector.
2.2 Neural Networks

The Neural Network model is a machine learning model that tries to solve complex problems just the way a human brain does. The neural network comprises of neurons, weights and connections. The arrangement of the neurons within the neural network is in such a way that the neural network performs the learning on the data itself. Neural Network can be as simple as single layered neural network consisting of input layer and output and can as complex as the multi-layered neural network consisting of input layer, many hidden layers and output. The connections between neurons have certain weights. These weights are responsible for carrying out data from one neuron to another. The weights signify the degree to which each input neuron contributes in obtaining the output. More important features (neurons) will be assigned a higher weight and less important ones with lesser weight. Weights can be implemented as a row vector, or a square matrix.

Learning happens by updating weights in a neural Network. Neural Network architecture describes how a neural network will behave in a certain scenario or situation. A network architecture comprise of the following:

- Number of neurons
- Number of Layers
- Types of connections between the layers

2.2.1 Machine Learning Algorithms:

There are different kinds of Machine Learning algorithms to address different use cases.

1. Supervised Learning: As the name suggests that this kind of learning is supervised where we have a set of outcomes that we wish to obtain as a prediction from set of inputs or independent variables. The model learns a function between
the input and desired output until a specific accuracy is reached. The data on which model is trained on is called the training data. Then the model is checked on validation data in order to check the accuracy. Examples are Linear Regression, Classifiers, Decision Trees etc

2. **Unsupervised Learning**: Here we don’t have a target or predicted audience. This is generally used for clustering problems. Examples are K-means, Apriori algorithm

3. **Reinforcement Learning**: In this algorithm the model is trained to work on specific problems/decisions. Here the model trains itself by trial and error method by exposing itself to a specific environment. The machine learns by taking good decisions and also from past decisions and tries to get the best decision. Examples are Markov Decision Process

Some of the most commonly used machine Learning algorithms are

- Linear Regression
- Logistic Regression/Classification Algorithm
- Random Forest
- K-Means
- Decision Tree

In Recursive Model Index we will be using linear regression models and classifier models. A short description for these models is given below:

### 2.2.1.1 Linear Regression Model:

Linear Regression Models are basically used to predict real value variables (example: positions, cost, discount etc). A learning relationship could be established between input and output. The relationship can be realized by a line called as regression line which fits the data and represented by a linear equation \( Y = w * X + b \), where \( Y \) is the output, \( w \) - weights, \( X \) - input, \( b \) - Bias. The regression line plot for a particular data is as shown below:

![Figure 2.5: Plot for Linear Regression Model](image)
2.2.1.2 Logistic Regression/Classifier Model:

Logistic Regression is a classification Model that is used to predict discrete values generally between 0,1 or yes,no or true,false based on expected output. Basically it determines the probability of certain event to happen. The end output being the probability generally lies between 0 and 1. As the name classifier suggest it classifies a new observation to a particular class. A plot for Logistic function (Sigmoid Activation Function) is as shown below:

![Figure 2.6: Plot of Logistic Function](image)

2.2.2 Feed forward Multi-layered Neural Network

![Figure 2.7: Multi layered Neural Network Topology](image)

This is the most basic form of the neural network in which we have an input layer consisting of one or more input nodes, one or more hidden layers and an output layer.
This is a fully connected acyclic neural network. By fully connected we mean that each and every neuron of the layer is connected to each and every neuron of the following layer. Training or the learning happens using the backpropagation algorithm using gradient descent.

### 2.2.2.1 Perceptron

A Perceptron may be defined as a linear model binary classifier with some input neurons, an output and with some threshold value for activation. It is a heaviside function with a certain threshold value. For a single layered perceptron we sum up the input times their respective weights (also known as the dot product of weights and inputs) and then the net sum is sent to the step function with a particular threshold. This function will give an output of either 1 or 0 (single binary value in case of step function), based on the input.

![Single Layered Perceptron](image)

A heaviside step function equation can be described as follows:

\[
\begin{align*}
f(x) &= \begin{cases} 
0 & x < 0 \\
1 & x \geq 0
\end{cases} 
\end{align*}
\]

The training begins with initializing random weights to the inputs, the dot product is calculated which is then passed through the activation function and the output is obtained. Output can be classes in case of a classifier. If the classification is correct then no weights are changed but if the classification is incorrect then weights are updated and checked again until the classification is correct. This loop continues until all the inputs are classified correctly. This algorithm is called the Perceptron Learning Algorithm. Limitation of a Perceptron Learning algorithm is the inability to solve nonlinear problems.
2.2. Neural Networks

2.2.2.2 Artificial Neuron

Artificial neuron is similar to perceptron but the only difference is in the activation function. Net input from an artificial neuron can be depicted as:

\[ \text{input\_sum}_i = W_i \cdot X_i \]

Where \( W_i \) is the vector of all weights coming into neuron \( i \) and \( X_i \) is the vector of all the input values from the input neuron. The whole equation can be seen as:

\[ \text{input\_sum}_i = W_i \cdot X_i + b \]

\( b \) stands for the bias term.

In order to get an output from the neuron this input sum is fed to the activation function and the corresponding output is generated.

\[ a_i = f(\text{input\_sum}) \]

\[ a_i = f(W_i \cdot X_i + b) \]

There are many types of activation function\( (f) \) that we can choose from depending on the data that we want to analyze.

2.2.2.3 Important terms to understand

1. **Connecting Weights** : With each and every neuron there are certain weights assigned to it. The assignment of weights are done on how influential the particular feature is in the network. The weights signify the amount by which the particular feature will amplify or will decrease while training the neural network. These are often denoted by the symbol \( w \).

2. **Biases** : Biases are basically nothing but scalar values that help triggering the neural network features in case of low signal strength. It forces the neural network to try out new combinations and interpretations. This is often denoted by the symbol \( b \).

3. **Activation Functions** : These functions are the functions that gives the output of the neural network. They control the behavior of the neural network and transmission of input from one layer to another. This whole cycle is called as forward propagation. They are responsible for the transformation of the combination of inputs, weights and biases as the input for the next layer. There are many types of activation functions that will be discussed in the coming sections.

4. **Input Layer** : In Figure 2.7 we can see an input layer with three artificial neurons. The raw data is fed into the input layer and then this input layer is connected to hidden layers having corresponding weights. The number of nodes in the input layer is equal to the number of input features that needs to be taken into consideration for the neural network. The input layer can either be fully connected or might not be fully connected.
5. **Hidden Layer**: There can be one or more hidden layers in a neural network. This basically depicts how information is being extracted from the raw data in order to give meaningful and correct output. In Figure 2.7 we can see there are two hidden layers each having 4 neurons each.

6. **Output Layer**: This is the layer where we get out results of the Neural Network. This is just like mapping input to output, the output layer given us the mapping to the input that has been provided. The output may be a real valued output or may be a probability, depending upon the activation function that is used. In Figure 2.7 there is one output layer consisting of two neurons.

### 2.2.3 Training of the Neural Network - Back propagation Learning

Training of a neural network happens by updating the weights of the neurons connecting edges and the biases. Weights basically describe the significance of an input feature in the neural network. Higher weight tells that the input has a larger role in the interpretation of the data. The process of learning involves re adjusting the weights and the biases, thereby making some input features more significant and some less.

Back propagation learning is also similar to perceptron learning algorithm but here we distribute the error between the contributing weights. With the perceptron algorithm its quite easy since each weight is corresponding to one neuron but with back propagation it becomes quite difficult as each weight will contribute to one or more neurons.

![Figure 2.9: Algorithm for Back Propagation](image)
Here in this Figure 2.9 we have an algorithm consisting of a network, training-records and a learning rate. Initially some random weights are assigned and the output is computed with a combination of weights, input, biases and activation functions. In order to calculate the error example\textsubscript{err} we subtract the output with actual output expected. This gives us the error which is example\textsubscript{err}. The weights and the bias are then updated by adding/subtracting the product of learning rate, example\textsubscript{err} and the derivative of input sum (of the neuron) to the weights or the bias, which then becomes the updated weights or the updated bias.

2.2.3.1 Gradient Descent:

The most basic optimization algorithm is Gradient Descent. Gradient means derivative of a function in one dimension to a function \( f \) in several directions. Generally our cost function will be convex and our main motive is to move as close as possible to the global minimum. Gradient descent follows this derivative approach to come down the hill to achieve global minimum.

![Gradient Descent](image)

Here we just start off with the random weights and taking derivatives we try to slope down and update weights each time we move down. The weight updates for the output and the hidden layers will be discussed in the section coming ahead.

2.2.3.2 Updating the output layer weights

The output layer weights can be updated by:

\[ W_{j,i} \leftarrow W_{j,i} \times \alpha \times a_j \times \Delta_i \times f'(input\_sum) \]

The error term \( e \) which was depicted as example\textsubscript{err} is denoted by \( Err_1 \). Derivative of activation function is denoted by \( f'(x) \). The error term is now given by:

\[ \Delta_i = Err_1 \times f'(input\_sum) \]
\[ W_{j,i} \leftarrow W_{j,i} \times \alpha \times a_j \times \Delta_i \]

**Propagation rule for the error value:**
\[ \Delta_j = f'(\text{input\_sum}) \sum_{i} W_{j,i} \Delta_i \]

**Updating the hidden layer weights:** To update the hidden layer we take the fractional error that has been computed for the output layer and multiply it by the learning rate and activation function from the previous layer.
\[ W_{k,j} \leftarrow W_{k,j} \times \alpha \times a_k \times \Delta_j \]

### 2.2.4 Activation Functions

Activation functions are responsible of transferring the input data from one layer to another layer. Generally most of the activation function are belonging to the sigmoid class. We would like to discuss some of the activation function that are generally used in Neural Networks.

#### 2.2.4.1 Linear

This is the most basic activation function as it passes the input as it is to the output without any changes. This is also known as the identity function and \( f(x) = Wx \).

![Figure 2.11: Linear Activation Function](image)

#### 2.2.4.2 Sigmoid

A sigmoid activation will convert the independent data variables coming from input or hidden layer into probabilities. The range of the output value will fall into 0 to 1. The sigmoid function is depicted by \( f(x) = \frac{1}{1+\exp^{-x}} \).
2.2.4.3 Tanh

Tanh is a trigonometric hyperbolic function and this function depicts the ratio of hyperbolic sine and hyperbolic cosine and the normalized range for a tanh function will be from -1 to 1. These kind of activation functions are good in dealing with negative numbers. The tanh function is depicted by $\tanh(x) = \frac{\sinh(x)}{\cosh(x)}$.

2.2.4.4 Softmax

This is a function that is generally used in classifier problems and returns a probability distribution over classes that are mutually exclusive. The algorithm of calculating a softmax function s as below:

1. $def softmax(L)$
2. $\expL = np.exp(L)$
3. $\sumExpL = \sum (\expL)$
4. $result = []$
5. for $i$ in $\expL$: $result.append(i * 1-0/\sumExpL)$
6. $return result$
2.2.4.5 Reactified Liner

This is a special function that will only activate the node when the value is greater than the threshold value. The ReLu function is depicted by \( f(x) = \max(0, x) \).

![Rectified Linear Activation Function](image)

Rectified Linear (ReLu) activation function has proved itself to train well in comparison to other activation functions without any use of training techniques. ReLu functions are the current state of art since the gradient will either be 0 or a constant thereby not suffering from the vanished gradient issue. We also have a leaky ReLu function wherein instead of giving a 0 we give a negative slope in order to avoid the dying ReLu issue. The function is:

\[
f(x) = \begin{cases} 
  x & x > 0 \\
  0.01x & \text{otherwise}
\end{cases}
\]  

(2.2)

2.2.5 Hyper parameters

In Neural networks there are certain parameters (both architectural parameters and training parameters) that we can adjust so that our training happens a bit faster, accurate and we can obtain better results. This particular tuning of parameters is called as Hyper parameters tuning. The basic motive of hyper parameters selection is to ensure that the model neither over fits nor under fits the training dataset, while making sure that the learning happens quickly. There are a number of parameters that can be tuned and they fall into several categories:

1. Architecture based on the input data
2.2. Neural Networks

2. Layer size
3. Magnitude (momentum, Learning Rate)
4. Regularization (dropout, drop connect, L1, L2)
5. Activation functions
6. Number of Epochs during training

The parameters defined above could also be differentiated into two categories, namely architectural parameters and training parameters.

Architectural Parameters includes the following:

1. Input Layer/Nodes
2. Hidden Layer/Nodes
3. Output Layer/Nodes
4. Bias
5. Activation functions

Training Parameters includes the following:

1. Weight Initialization
2. Error Function
3. Learning Rate
4. Epochs
5. Batch Size

2.2.5.1 Architecture based on the input data

Here is a small summary table that depicts the possible choice of architecture based on the input data of the Neural Networks.

2.2.5.2 Layer Size

Layer size is actually defined by the number of neurons in each layer. For the input and the output layer it is pretty simple, as the input layer will have neurons that are equal to the number of features on which the output depends and the output layer will have neurons equal to the classes or the result that we are trying to predict. The part that is tricky is the number of neurons in the hidden layers and the number of neurons in the hidden layers. The thumb rule is that more the data more number of hidden layers and more neurons are required such that the network doesn’t over fit or under fit the data. The network size is directly proportional to the dataset size. As an example the MNIST dataset has 3-4 hidden layers.

The number of neurons in the hidden layer also has a great impact on the training of the neural network. If we have to less neurons then training will not be proper as it wont be able to learn complex problems and if we have too many then we might run into over fitting issues. Here is how to start, the number of neurons in the next layer should be less that the previous one and also the number of neurons in the hidden layer should not be less than three fourth times the neuron in the previous layer. So in the end its the correct and optimal combination of amount of data, layer size and regularization.
### 2.2.5.3 Magnitude: Momentum and Learning Rate

Momentum is the parameter that helps the learning algorithm to come out if it is stuck in the local minimum. Learning Rate (LR) is the rate at which we want to change our parameters (weights) during the optimization of the network so that we can reduce the error between the output by the neural network and the actual output. In the back propagation algorithm when we try to update the weights we multiply the error gradient with the learning rate and then with the weights to get the new updated weights.

Large Learning rate (e.g., 0.8 or 1) will update parameters in large leaps whereas small Learning rate (e.g., 0.0001) will update parameters in small amount. Large LR’s will save time and will also train the neural network faster but this may overshoot out minimum. This can be stuck at the local minimum and may never reach the output that we want to get. In contrast small LR’s train slowly and can take too long to train the neural network but it will eventually lead to the local minimum.

### 2.2.5.4 Regularization

Over fitting in machine learning is controlled by regularization. It uses different methods for minimizing the parameter size in order to avoid the effects of out-of-control parameters. Regularization tries to maintain a trade off between a good fit by keeping weights of certain features low. Coefficients L1, L2 are penalty functions help making certain weights low, which eventually leads to simpler hypothesis. Simpler hypothesis are generalizable where as random unregularized weights leads to over fitting the data. These avoid in getting the neural space bigger in one direction. Drop out and Drop connect methods are used for regularization in which they either omit or mute one of the hidden neuron such that it doesn’t contribute to the network anymore.

### 2.2.5.5 Number of Epochs and Mini batches

Mini-batching means send one or more input vectors for training in the neural Network. It increases the efficiency of the Neural Network and also allows us to use the the
computer hardware and resources more efficiently. Mini batch-size is actually the number of records that we would pass through the algorithm for training at a time. The relationship between batch-size and training speed is generally a U-Shaped graph. As the batch size will become larger the training time will decrease and then eventually it increases when it exceeds a certain threshold for batch size. Epochs are full passes over the entire Neural Network (forward propagation as well as back propagation).

### 2.2.6 Loss Functions

Loss function tells us how close we are to the target. The intention here is to calculate the error function based on difference between the actual target and the predicted target. The error is then aggregated over the entire dataset and then averaged to get a single valued error. Hyperparameter tuning may help in reducing the error and optimizing the neural network.

#### 2.2.6.1 Loss Functions for Regression Models

In this section, we will go through different loss functions for Regression Models.

1. **Mean Squared Error Loss (MSE):** When we are working with a model that requires real valued outputs we generally use Mean Squared Error Loss. MSE is very sensitive to outliers, so one should be very careful while choosing a loss function for its model.

   \[
   L(W, b) = \frac{1}{N} \sum_{i=1}^{N} (\hat{Y}_i - Y_i)^2
   \]

   where N denotes the number of samples (inputs) with corresponding outputs that have been gathered, \( \hat{Y}_i \) denotes the output of the neural network and \( Y_i \) is the target output, M denotes the number of output features.

   In case when we have more than one output feature the Mean Squared Error loss now becomes:

   \[
   L(W, b) = \frac{1}{2N} \sum_{i=1}^{N} \sum_{j=1}^{M} (\hat{Y}_{ij} - Y_{ij})^2
   \]

2. **Mean Absolute Error Loss (MAE):** Just like the Mean Squared Error loss, the Mean Absolute Error Loss averages the absolute error.

   \[
   L(W, b) = \frac{1}{2N} \sum_{i=1}^{N} \sum_{j=1}^{M} |\hat{Y}_{ij} - Y_{ij}|
   \]

3. **Mean squared Log Loss (MSLE):**

   \[
   L(W, b) = \frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{M} (\log \hat{Y}_{ij} - \log Y_{ij})^2
   \]

4. **Mean Absolute Percentage Error Loss (MAPE)):**

   \[
   L(W, b) = \frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{M} \left| \frac{100 \times (\hat{Y}_{ij} - Y_{ij})}{Y_{ij}} \right|
   \]
2.2.6.2 Loss Functions for Classification Models:

In this section, we will go through different loss functions for Classification Models.

1. **Hinge Loss**: This is the most common loss function for classification models whenever there is a requirement of hard classification between 0,1 (binary classifier) or may be in -1,1. The loss equation of hinge loss when classification must happen between -1,1 is as below and used for binary classifications.

\[
L(W, b) = \frac{1}{N} \sum_{i=1}^{N} \max((0, 1), Y_{i,j} \times \hat{Y}_{i,j})
\]

2. **Logistic Loss**: Logistic loss functions are used when we actually want to model probabilities rather than an absolute value. The output for such models will be between 0 and 1. We also need to be sure that the last layer of the neural network has used a softmax or a sigmoid or any other classification activation function. Basically we are trying to predict the maximum likelihood for an outcome.

We try to understand the loss by assuming that our network is making a prediction for two classes, for instance spam and not spam (0-1 classifier). Assuming that output of \(X_i\) is given by \(h(W, b)\) and \(1-h(W, b)\)

\[
P(y_i = 1|X_i, W, b) = h(W, b)\]

\[
P(y_i = 0|X_i, W, b) = 1 - h(W, b)
\]

We can combine the above equations and express it as below:

\[
P(y_i|X_i, W, b) = (h(W, b))^y_i \times (1-(h(W, b))^{1-y_i})
\]

3. **Cross Entropy**: Cross Entropy losses are used for classification models that measure the output of a classifier which is a probability between 0 and 1.

![Log Loss when true label = 1](image)

Figure 2.15: Cross Entropy loss with respect to Predicted Value (Expected label is 1)
Cross entropy loss increases as the predicted probability diverges from the expected label. The figure below depicts the change in the cross entropy loss with respect to the predicted value when the expected label is 1. It is quite evident that if the prediction is very far away from 1 the loss is huge. As we move closer to the expected output the loss decreases.

### 2.2.6.3 Activation Functions

Output of a neural Network depends on the activation functions and also the type of output that we are desiring from the neural Network. The table below gives the suggested activation functions to be used based on the use cases.

<table>
<thead>
<tr>
<th>Regression Model Output</th>
<th>Activation Functions</th>
<th>Other suggestions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear Activation Function</td>
<td>TanH functions - if data is in range $[-1,1]$, ReLu functions if data is in range $[0,\infty]$</td>
<td></td>
</tr>
<tr>
<td>Single Label Classification Problems (Binary Classifier)</td>
<td>Sigmoid Activation Function</td>
<td>In case of output layer with two outputs for a binary classifier, Softmax activation function may be used.</td>
</tr>
<tr>
<td>Multi-Class classification Problem</td>
<td>Softmax activation function</td>
<td>In cases where the number of classes are huge (thousands), then hierarchical Softmax activation function must be used.</td>
</tr>
<tr>
<td>Multi-label Classification Problem</td>
<td>Sigmoid Activation Function</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.4: Activation functions based of the output data [6]

### 2.3 Learned Index Structures

Indexes are the most efficient way to access the data and are especially important for databases. There has been a constant effort for index implementation being optimized
for GPU, cache and memory. Indexes are built for worst case scenarios. Index optimizations that exploit data distributions are usually very costly. “Case of Leaned Index Structure” [1] (a paper written by Tim Kraska and his colleagues) tries to address this problem using Machine Learning algorithms. Learned indexes has the goal of learning patterns and correlations in the data using Machine Learning Techniques and synthesizing efficient structures with low cost and engineering cost. However this paper also points out the challenges of semantic guarantees and performance demands of Learned Index in comparison to the traditional indexes. Semantic guarantees of indexes might be difficult to keep up to without using the traditional structures. Also, Machine Learning is performance intensive, this paper envisions that GPU will become more mainstream and performance will no longer be an issue. Furthermore Neural networks are more SIMD friendly in comparison to CPU-based Index structures. As future works they plan to extend the same approach for joins, sorts and write-heavy workloads.

If we talk about recent machine learning techniques for implementing databases internals then it would be a wise choice to introduce the idea given by Andrej Karpathy which is Software 2.0. Software 1.0 includes languages like JAVA, C++ where we the kind of operation that we need to implement. Software 2.0 is the code written by the optimization based on an evaluation criteria. Software 2.0 developers curate, maintain, clean and label datasets. They define the learning task, such that the program will learn via the optimization. Software 1.0 developers maintain the surrounding tools, analytics, visualizations, labeling interfaces, infrastructure.

2.3.1 Range Indexes

B-Trees provides a mapping of a key to a position with a guarantee that the record at this position is equal to or higher than the key provided we are looking in a sorted array. In machine learning terms B-Trees act like regression models since it maps a key to a position with certain amount of minimum and maximum error.

![Figure 2.16: B-Tress as Regression Models](image)

If we talk about the error guarantees of Machine Learning models, then we understand that B-Trees also provide error guarantees on stored data and for new data re-balancing.
is required which means that we are re-training the B-Trees with a certain min-max error. This is similar to a regression model in Machine Learning where in a key is given and the model gives a prediction and also remembers the worst over and under prediction. Hence we can replace B-Trees with a regression model. Other technical challenges are that B-Trees have a bounded cost for inserts and lookups and are quite efficient in cache usage. Although B-Trees come up with a lot of advantages but models like Deep Learning can transform the look up cost from log n to a constant operation.

2.3.1.1 Naive Learned Index

To further investigate the technical requirements of replacing traditional B-Trees with Learned Indexes the authors tried to implement a naive learned index and the performance was compared with that of the B-Trees. For implementation they took 200M web-server log records and using Tensor Flow tried to build a secondary index over timestamps. A two layered fully connected Neural network with 32 neurons in each layer and Rectified Linear Unit (ReLU) activation function was trained on this dataset to predict the positions. The input features were timestamps and position labels as the output function. They tried to calculate the lookup time for randomly selected keys using Tensor Flow and Python as the front end. It was found that search time of the trained Neural Network for predicting positions had no benefits over full search scan. Moreover B-Trees were 2-3 times faster on the same dataset.

A number of reasons were given in this respect:

1. Tensor Flow (python library for machine learning) has overheads while trying to learn small models.
2. B-Trees are good in over-fitting the data. Models are good at a high level (data distribution shape) but are bad at individual prediction.
3. B-Trees are cache efficient, whereas models are not.

2.3.1.2 Recursive Model Index

In order to overcome the above shortcomings, they proposed a new approach of Recursive Model Index (RMI), Learned Index framework (LIF) and standard error based search strategies. The learned index structure could be visualized as index synthesis system, when given an index specification. Learned Index Structures generates different configuration, optimizes them and then will test them automatically. Simple models could be learned on “the-fly", whereas complex models will be learned by Tensor flow. One thing to notice is that it never uses Tensor flow for inference. LIF extracts the weights and based on the model configuration creates efficient index structures in C++.

Recursive Model Index is a hierarchy of models where at each stage a lookup key is fed to a model and then the particular model predicts the next model until the leaf model is reached. The leaf model is basically a page where the search operation could be performed and it predicts a position for a lookup key.
Each model makes a prediction with a certain error about the position for the lookup key and that prediction is used for selection of next model. Recursive Model Indexes are not trees as it may be possible that different models at one stage may predict the same model at the next stage.

The advantages for Recursive Model Index are

1. It is easy to learn the overall distribution of data
2. RMI divides the space into smaller sub-ranges (like B-Trees).
3. No search process is required between the stages.

2.3.1.3 Hybrid Indexes

The beauty of RMI is that we can have different types of models in one architecture. For instance Rectified Linear Unit may be the top model which could be useful for learning
large and complex data distributions, whereas the leaf models might be simple linear models. Also, if the data couldn’t be learned then we could have a B-Tree model as the leaf model.

The training of the model as shown in Figure 2.18 begins with training of the top model, which then predicts the next model in line 9 and 10 until it reaches the leaf model to make a prediction. In case of hybrid indexes the min/max error is calculated based on the prediction made by the model. If the min/max error is greater than the threshold then the model is replaced by B-Trees. This becomes the selling point of their model, in which the worst case performance will that be of a B-Tree.

2.3.1.4 Search Strategies

The paper evaluates the lookup time, which is also affected by the search on the leaf node. Binary Search is one of the fastest search strategy. They also evaluate that a model can make a prediction which is much closer to the actual position whereas the min/max error might still be larger. The binary search case could be improved by the knowledge of standard deviation in the prediction of the model of the former stage. They present three strategies for it:

1. **Model Binary Search**: The prediction is served as a middle point and then they search right and left.

2. **Biased Search**: This is said to be the modified Binary Search. If the key is determined to be greater than the middle, the new middle is said to be min(middle + σ, (middle + right)/2), where σ is given by the former model.

3. **Biased Quaternary Search**: When the data distribution is not known, three points are chosen pos + σ, pos, pos - σ, where pos is the predicted position.

2.3.1.5 Results

In order to compare learned Indexes with B-Trees, authors created four secondary indexes over three real world or synthetic datasets.

1. **Web Server logs**: This dataset consists of 200M log entries, for every request to a major university website over the years, they created an index over all unique timestamps. This is a kind of worst learning problem as there is a complex pattern because of weekends, holidays, class schedules etc.

2. **Maps (Longitudes, Integers)**: For the map dataset they indexed the longitude of 200M user-maintained features (e.g., roads, museums, coffee shops) across the world. The longitude of locations are relatively linear and has less irregularities than the web server logs dataset.
3. **Web documents**: This dataset consists of 10M non-continuous document-ids of a large web index used as a part of a real product at a large company.

4. **Log normal** (**only synthetic one, Integers**): 190M unique values sampled from a log normal distribution with $\mu = 0$ and $\sigma = 2$. These values are scaled up to be integers by 1B. This data is highly non-linear making cumulative distribution function more difficult to learn using neural networks.

For all configurations, authors try to compare B-Trees with different page sizes with learned indexes using 2-stage RMI model and different second stage sizes (ie 10K, 50K, 100K and 200K). Neural Networks with 0..2 hidden layers and layer width ranging from 4..32 nodes were trained. The findings were that for the first stage simple (0 hidden layers) models to semi-complex (2 hidden layers and 8 or 16 wide) models gave the best results. For the second layer simple (0 hidden layers) models, which are linear models, had the best performance. B-Tree implementation is similar to stx::btree, with further cache-line optimization. Authors also report that in a micro benchmark against FAST, a state-of-the-art SIMD optimized B-Tree they did not observe large differences.

Conclusions of implementing Recursive Model Index for all datasets:

1. For larger datasets, the RMI are 3X faster, 10x smaller.
2. Hybrid Indexes did make sense for strings.
3. Less Model accuracy increases search time.
4. The last stages have higher influence on search and on memory sizes.

### 2.3.2 Point Indexes

Point indexes often referred as Hash Maps use hash functions to map a lookup key to a random position. The main problem is mapping of distinct lookups key to the same position which is termed as conflict. Whenever there is a conflict, for managing overflows the index has to use a linked list. Traversing through linked list in case of large datasets is equivalent to a cache miss costing up to 50-100 cycles.

![Figure 9: Traditional Hash-map vs Learned Hash-map](image)

Figure 2.19: Hash Maps vs Learned Hash Maps
In order to deal with the problem of overflow, machine learning provides a solution by learning the hash model itself that maps all the unique keys to different positions. If we try to highly utilize the traditional indexes, then the performance would degrade with added penalties. In case of learned Hash Maps it is possible to highly utilize the index but it is dependent on the data distribution.

Authors also suggested that learning a CDF of key distribution can help us to learn a better hash function. The data is not stored in a sorted manner neither compactly. They scaled the CDF by a target size M of the Hash map and used \( h(K) = F(K) \times M \), with key K as hash functions. If the model is able to learn the function F, then all conflicts will be avoided.

### 2.3.2.1 Results

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Slots</th>
<th>Hash Type</th>
<th>Search Time (ms)</th>
<th>Empty Slots</th>
<th>Space Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Map</td>
<td>75%</td>
<td>Model Hash</td>
<td>67.03 (95%)</td>
<td></td>
<td>-20%</td>
</tr>
<tr>
<td></td>
<td>90%</td>
<td>Model Hash</td>
<td>62.00 (95%)</td>
<td></td>
<td>-25%</td>
</tr>
<tr>
<td></td>
<td>100%</td>
<td>Model Hash</td>
<td>58.00 (95%)</td>
<td></td>
<td>-27%</td>
</tr>
<tr>
<td></td>
<td>Random Hash</td>
<td>52.00 (95%)</td>
<td>0.80 (25%)</td>
<td></td>
<td>-25%</td>
</tr>
<tr>
<td></td>
<td>Random Hash</td>
<td>48.00 (95%)</td>
<td>1.00 (25%)</td>
<td></td>
<td>-30%</td>
</tr>
<tr>
<td></td>
<td>Random Hash</td>
<td>44.00 (95%)</td>
<td>1.50 (25%)</td>
<td></td>
<td>-35%</td>
</tr>
<tr>
<td></td>
<td>Random Hash</td>
<td>40.00 (95%)</td>
<td>2.00 (25%)</td>
<td></td>
<td>-40%</td>
</tr>
<tr>
<td>Web Log</td>
<td>75%</td>
<td>Model Hash</td>
<td>78.00 (19%)</td>
<td></td>
<td>-20%</td>
</tr>
<tr>
<td></td>
<td>90%</td>
<td>Model Hash</td>
<td>73.00 (19%)</td>
<td></td>
<td>-25%</td>
</tr>
<tr>
<td></td>
<td>100%</td>
<td>Model Hash</td>
<td>68.00 (19%)</td>
<td></td>
<td>-30%</td>
</tr>
<tr>
<td></td>
<td>Random Hash</td>
<td>64.00 (19%)</td>
<td>0.80 (25%)</td>
<td></td>
<td>-20%</td>
</tr>
<tr>
<td></td>
<td>Random Hash</td>
<td>52.00 (19%)</td>
<td>1.00 (25%)</td>
<td></td>
<td>-25%</td>
</tr>
<tr>
<td></td>
<td>Random Hash</td>
<td>48.00 (19%)</td>
<td>1.50 (25%)</td>
<td></td>
<td>-30%</td>
</tr>
<tr>
<td></td>
<td>Random Hash</td>
<td>44.00 (19%)</td>
<td>2.00 (25%)</td>
<td></td>
<td>-40%</td>
</tr>
<tr>
<td>Log Normal</td>
<td>75%</td>
<td>Model Hash</td>
<td>79.00 (20%)</td>
<td></td>
<td>-20%</td>
</tr>
<tr>
<td></td>
<td>90%</td>
<td>Model Hash</td>
<td>75.00 (20%)</td>
<td></td>
<td>-25%</td>
</tr>
<tr>
<td></td>
<td>100%</td>
<td>Model Hash</td>
<td>71.00 (20%)</td>
<td></td>
<td>-30%</td>
</tr>
<tr>
<td></td>
<td>Random Hash</td>
<td>66.00 (20%)</td>
<td>1.00 (25%)</td>
<td></td>
<td>-25%</td>
</tr>
<tr>
<td></td>
<td>Random Hash</td>
<td>52.00 (20%)</td>
<td>1.50 (25%)</td>
<td></td>
<td>-30%</td>
</tr>
<tr>
<td></td>
<td>Random Hash</td>
<td>46.00 (20%)</td>
<td>2.00 (25%)</td>
<td></td>
<td>-40%</td>
</tr>
</tbody>
</table>

Figure 2.20: Results of Lookup Time for Hash Maps

In order to check the feasibility of Hash Map Indexes, the authors implemented a linked list based Hash Map. Here the records are stored normally just like the traditional Hash map but in case of a conflict it is then attached to a linked list. Hash Map implementation was a randomized hash function that uses two multiplications (3 XORS or 3 bit shifts). A two staged recursive Model Index was used with 100K models on the second stage with no hidden layers. The same datasets were used as in the B-Trees and the number of available slots were varied from 75 to 125 percent.

### 2.3.3 Existence Indexes

Existence indexes are one of the common types of index, Bloom Filters is a probabilistic data structure which is space efficient and it tells whether or not data is present. Bloom Filters use bit array of some size m and k hash functions, these hash functions are used to map key to one bit in the array.
In order to insert an element into a bloom filter, we feed the key to the k hash function and the bits of the returned positions are set to 1. In order to test when the key is again fed back to k hash function, it should receive k array position. The key is not a member if any of the bit of the positions returned is 0. Bloom filters always guarantee a False Negative Ratio (FNR) but a potential False Positive Rate (FPR).

### 2.3.3.1 Learned Bloom Filters

Existence indexes needs to learn a function that makes a clear distinction between keys and non-keys. A good hash function for a bloom-filter will be the one with loads and loads collisions within non-keys and non-keys, keys and keys, but least number of collisions within keys and non-keys. The set of keys can be denoted by $\kappa$ and set of non-keys by $\mu$. Authors propose to visualize Bloom filters as a classification problem. So all in all we learn a model that classifies a key in to one of the class (Key or No-Key). Authors train a neural network $D = \{(x_i, y_i = 1) | x_i \in \kappa\} \cup \{(x_i, y_i = 0) | x_i \in \mu\}$

We have sigmoid activation function, so the classifier will predict probabilities with the minimum loss function. Since the output will give us the probability of a key to be present in a set of keys a threshold $\tau$ needs to decides upon so that we can definitely ensure of the existence of the keys. Bloom filters are bound to have 0 FNR and potential FPR, but our model has non-zero FNR and FPR. So, in order to maintain 0 FNR, authors introduced a concept of Overflow Bloom Filter. Authors consider $\kappa = \{(x) \in \kappa \mid f(x) < \tau\}$ as a set of FNR from f and this set has to pass through another bloom filter. If $f(x) \geq \tau$ then the key definitely exists.
2.4 Summary:

In this section we would like to highlight the takeaways from the background chapter. These are listed below:

- Indexes are a way to optimize the performance of operations (retrieval, access, filter, select etc) in a database.

- In database we have different kinds of indexes which are specific for certain use cases. B-Trees are used for range based lookups, Hash Maps are used for key based lookups and bloom filters for existence queries.

- B-Tree index is the commonly used index in most of the applications.

- However, traditional index structures are not general purpose and don’t fit the worst case distribution. Also, re balancing of B-Trees is required whenever we add or delete an item from B-Trees.

- Real world data don’t necessarily follow a specific data distribution. It would cost a lot of engineering effort if we try to build specialized indexed structures.

- Learned Index Structures are specialized solutions that makes use of Machine Learning algorithm (Neural Networks) in order to make use of patterns and correlations with in the data.

- Learned Index Structures could be build at a lower engineering cost and results could be reconstructed using matrix multiplication of weights of trained models. Hence results for an index are obtained through Learned Index Structures with a lesser memory footprint.

- The authors propose that the traditional indexes could be replaced by learned Index Structures.

- Recursive Model Index approach (hierarchy of model) was proposed by the authors to predict positions of a lookup key.

- The benefits of RMI is easy to learn the overall data distribution, divides spaces into smaller sub ranges (like B-Trees), no search process required between stages.

- An additional selling point was that this approach was hybrid in nature. We could use different index models at each stage and calculate the error. If the error was greater than the threshold then the model could be replaced by B-Trees. So, the worst case performance will that be of a B-Tree.

- A hybrid end to end training algorithm was used to train the Recursive Model Index.
• With the approach above the authors tried to build a leaned B-Tree and trained the model on four datasets (web server logs, maps dataset, synthetic logs). The implementation of Neural Network in the paper outperforms B-Trees by 70%.

• Similar approaches could be applied on Hash Maps and Existence Indexes. For Hash Maps conflicts could be avoided by making a model learn a function that would uniquely maps a data set to a position.

• The authors suggested to implement Bloom filters as a classification problem in order to avoid the problem of potential False positive Ratio.
3. Prototypical implementation and research questions

In this chapter we present the research questions, the experimental setup and the prototypical implementation of the Learned Index Structures, including hyper parameter tuning and design choices. We structure this chapter as follows:

- In **Section 3.1** we attempt to provide a list of questions that remained unanswered in the paper by Kraska et al. [1]. Out of those questions we have chosen a few questions as focal points to address in this thesis.

- In **Section 3.2** we describe the details of prototypical implementation of Recursive Model Index (RMI) of our Learned B-Trees along with the details of Keras classifiers and Keras Regression Models.

- In **Section 3.2.4** We discuss about the issue of Bloom filter regarding potential False Positive Rate. We try to interpret bloom filters as a classification problem and discuss the prototype implementation details and the results.

### 3.1 Research question

We can list some of the unanswered questions in the work of Kraska et al.[1], by dividing them between methodological and design questions:

**Methodological questions:**

This set of questions refer to limitations to the reproducibility of the findings, as well as aspects that could be further evaluated.
• **Datasets:** Standard datasets were not used for the test. Datasets are not publicly available. Furthermore, the datasets are insufficiently described, preventing their reconstruction or approximation.

• **Model details:** How was the encoding done for feeding the input to the network? What was the architecture employed? Except for the naive approach, there was no mention of the number of nodes/hidden layer. Was there any bias added? What about the error functions?

• **Training details:** The learning rate has a great impact on the training and accuracy of a model. What learning rate was used? What kind of Optimizers were used? How many epochs/iterations was the model trained on? No information about training Loss and training and validation data set was given.

• **Comparisons:** How does training time for all the models, really compare to building a complete index? Comparison with other B-tree configurations and other structures?

• **Evaluation with other platforms:** Authors report inference time on CPUs, how about GPUs?

In summary, we consider that further study is needed to understand their results in a more general manner, and to consider the parameter settings. Also, more data on the training time seems important.

**Design questions:**

This set of questions refer to the design of the learned index structures, proposing aspects that need further development. Among them we can suggest the following:

• **Generalizability across datasets:** Can learned indexes be made general for all datasets? (This includes more work on indexing strings)

• **Usage:** How do we deal with batch queries?

• **Insertion:** Concept for updating these structures: How to add or delete nodes? What needs retraining and delta updates?

From this list we have chosen a few research questions to be addressed in this thesis which are as follows:

1. Hyper-parameter impact in accuracy: What is the impact of hyper parameters in the resulting accuracy for these structures? What is their resulting memory efficiency? What are the exact training times?
2. Comparison of training time and B-tree build times: How does the best training time achieved for the learned structures compare to building a B-Trees Index?
3. Performance comparison for Inference time: How does the inference time of a hand-tuned implementation, optimized through the use of SIMD, compare to the lookup time for B-trees? How does the performance compare when fetch time is added?
4. Insertions: How can insertions happen in learned indexes?
3.2 Learned index structure implementation

3.2.1 Experimental setup

We have used the following configurations for the prototypical implementation (training of models):

- **Machine configuration:**
  - Operating System: Windows 10 Home 64-bit (10.0, Build 17134)
  - Processor: Intel(R) Core(TM) i3-6006U CPU @ 2.00GHz (4 CPUs), 2.0GHz
  - Memory: 8GB RAM
- **Python version:** 3.6
- **Anaconda version:** 4.4.7
- **Anaconda build version:** 3.0.27
- **Tensorflow version:** 1.7.0

In addition we have employed Keras, a library for neural networks in Python which uses Tensor Flow in the back end (Tensor Flow is a library for machine learning in Python).

We have used the following configurations for testing the parameters between LIS and B-Trees:

- **Machine configuration:**
  - Operating System: Ubuntu 16.04 LTS
  - Processor: Intel(R) Core(TM) i7-6700U CPU @ 3.40GHz × 8
  - Disk: 488.1 GB

- **B+ Tree:** We have used STX B+Tree package which is a set of C++ template classes implementing a B+ tree key/data container in main memory. This package consist of five main classes within the STX namespace. The package consists of a base class Btree which implements B+ Tree algorithms using leaf nodes and inner nodes in main memory. The current source package can be downloaded from: [http://panthema.net/2007/stx-btree/](http://panthema.net/2007/stx-btree/) By default this tree has nodes of size 128 bytes, with 10 items per node (when only storing positions) or 8 items per node (when storing more data in the tree), in our evaluations.

- **For comparing with the learned index structure we implement a small C++ program in charge of performing the inference, while using SIMD instructions, and aiming for efficiency.**

---

1 As per the default parameters. See: [https://github.com/bingmann/stx-btree](https://github.com/bingmann/stx-btree)
We measure time using the call to `gettime` in C++. For python we use the built-in time library. For all cases related to performance we report the results of several repetitions. For inference we perform lookups on 200 keys and report the total time elapsed.

### 3.2.2 Dataset

Making using of the phases of CRISP-DM model this section deals with data understanding and data preparation.

We have used the standard TPC-H line items table with scale factor 1. This table comprised of ORDERKEY, PARTKEY, SUPPKEY, LINENUMBER, QUANTITY, EXTENDED PRICE, DISCOUNT, TAX, RETURN FLAG, LINE STATUS, SHIP-DATE, COMMITDATE, RECEIPTDATE, SHIPINSTRUCT, SHIPMODE, COMMENT columns. The dataset was generated using the DBGEN tool provided by the TPC-H council.

We have created an index on ORDERKEYS. To this end duplicate keys were removed (reducing the tuples to 200,000).

### 3.2.3 Learned B-Trees

We have taken the standard TPC-H dataset for line items and used it to try to predict the position for particular ORDERKEY entries. With this we deal with the modeling phase of CRISP-DM model.

We have followed the Recursive Index Model approach in which we have a parent model that is fed with a look up key (ORDER KEY) and based on the lookup key it decides the next model for prediction. In our case we have taken a two level Recursive Model Index where the first level consists of a Keras classifier model (parent model) which classifies the lookup key into one of the child models, and furthermore the search operation can be performed in that child model. The whole dataset is quantized into four subranges (or children) and the parent model is trained to classify the look up key (ORDER KEY) into one of the child models (based on the subrange that the lookup up key falls in).

The leaf nodes/child model are implemented as Keras linear regression models that are trained on the respective sub ranges and give a prediction of the position for the lookup key. Once an Order Key is given to the parent model, it then classifies the Order Key into one of the child models which further predict the position based on the data distribution curve learned by the child model.

The classifier is a 2-layered fully connected neural network with 12 and 8 neurons in each hidden layer and Rectified Linear Unit and Sigmoid activation functions. The Regressors model is a 1 layered fully connected neural network with 8 neurons in the hidden layer and Rectified Linear Unit activation function. The recursive model Index for Learned B-Trees is as shown in Figure 3.1.
3.2. Learned index structure implementation

Figure 3.1: Recursive model index for implementation of Learned B-Trees

Figure 3.2: Neural network for the parent classifier

```python
# define baseline model
def create_baseline():
    # create model
    model = Sequential()
    model.add(Dense(12, input_dim=1, init='uniform', activation='relu'))
    model.add(Dense(8, init='uniform', activation='relu'))
    model.add(Dense(4, init='uniform', activation='sigmoid'))

    # Compile model. We use the the binary cross entropy loss function, and the Adam gradient optimizer.
    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
    return model
```

Figure 3.3: Neural network implementation code for parent
In case of real world data the positions of some data items is not stored in a sequential order, data items are stored in random positions. For a neural network to learn a completely random function in which the data doesn’t follow a pattern of distribution function or in which there is no correlation between input and output might be very difficult and might need a lot of training and hyper parameter tuning, or a highly complex neural network design.

In our implementation the model predicts a sorted position (the sequence of these positions represents a sparse progressive sequence, with gaps between numbers, to potentially allow for insertions). In turn the sorted position serves as a lookup key in an array, which is mapped to the actual random position in a key-based lookup table. We have a key value table where keys are the sorted position (that are learned by the model) which is given by the model and the values are the actual random position. Once we predict the sorted positions after passing the lookup key through the parent model (classifier) and the child model (regressors), we pass the keys to a look up table and fetch the random positions. The results of the implementation of Recursive Model Index can be seen in Figure 3.6.

```python
def baseline_model():
    # create model
    model = Sequential()
    model.add(Dense(8, input_dim=1, kernel_initializer='normal', activation='relu'))
    model.add(Dense(1, kernel_initializer='normal'))

    # Compile model and loss function as mean squared error
    model.compile(loss='mean_squared_error', optimizer=optimizer)
    return model
```

Figure 3.5: Neural network implementation code for child model

Figure 3.4: Neural network for child linear regressors
3.2. Learned index structure implementation

3.2.4 Learned Bloom Filters

Bloom Filters have a zero False Negative Rate and a potential False Positive Rate. As suggested by the authors the problem of a potential False Positive could be avoided by considering the Bloom filter as a classification problem. In our implementation we have a two layered neural network with 12 and 8 neurons in the hidden layer. The output layer has 2 nodes, having a sigmoid activation function in the output layer and all other layers have Rectified Linear Unit activation function. All the keys are given a classification as 1 and the non keys a classification as 0. The neural network architecture of Bloom filter is shown in Figure 3.7. A neural network is trained...
\[ D = \{(x_i, y_i = 1) \mid x_i \in \kappa\} \cup \{(x_i, y_i = 0) \mid x_i \in \mu\} \]

where \( x_i \) is the input, \( y_i \) is the output class, \( \kappa \) is the set of keys, \( \mu \) is the set of non keys. The implementation code for the Bloom Filter Classifier is shown in Figure 3.8.

The classification model is trained for Order Keys from 1 to 200000. The classification model acts like an existence index. The results of the bloom filter are shown in Figure 3.9.

```python
# define bloom filter baseline model
def create_baseline():
    # create model
    model = Sequential()
    model.add(Dense(12, input_dim=1, init='uniform', activation='relu'))
    model.add(Dense(8, init='uniform', activation='relu'))
    model.add(Dense(2, init='uniform', activation='sigmoid'))
    # Compile model, we use the the binary cross entropy loss function, and the Adam gradient optimizer.
    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])

    return model
```

Figure 3.8: Neural network implementation for classification

```
Enter the value to get Prediction:56730
The Order Key is present

Enter the value to get Prediction:300000
The Order Key is not present

Enter the value to get Prediction:76453
The Order Key is present

Enter the value to get Prediction:458362
The Order Key is not present

Enter the value to get Prediction:7489
The Order Key is present

Enter the value to get Prediction:987654
The Order Key is not present
```

Figure 3.9: Results for bloom filters as a classification Problem

### 3.3 Summary

- Our implementation for the Recursive Model Index consist of a parent model (classifier) and four child models (regression).
- To train classifiers and regression models, hyper parameter tuning is required.
- The best configuration in our study, for the parent model (classifier) was:
  - Architecture: Input (1 Neuron) \(\rightarrow\) Hidden Layer 1 (12 Neurons) \(\rightarrow\) Hidden Layer 2 (8 Neurons) \(\rightarrow\) Output Layer (4 Neurons)
  - Learning Rate - 0.0001
  - Batch Size - 128
  - Optimizer - Adam
3.3. Summary

- Epochs - 100

• The best configuration in our study, for the child models (regression) was:
  - Architecture: Input (1 Neuron) \(\rightarrow\) Hidden Layer 1 (8 Neurons) \(\rightarrow\) Output Layer (1 Neurons)
  - Learning Rate - 0.0001
  - Batch Size - 128
  - Optimizer - Adam
  - Epochs - 100

• The best configuration for implementing a Bloom Filter as a classification problem, in our study, was:
  - Architecture: Input (1 Neuron) \(\rightarrow\) Hidden Layer 1 (8 Neurons) \(\rightarrow\) Output Layer (1 Neurons)
  - Learning Rate - 0.0001
  - Batch Size - 128
  - Optimizer - Adam
  - Epochs - 100
3. Prototypical implementation and research questions
4. Hyper parameter tuning and inference results

In this chapter we evaluate the impact of hyper parameter tuning on the learned model, by measuring the resulting error (root mean square error, mean square error) of the model under different configurations. We also provide an experimental comparison between our implementation of the learned index structures and the traditional B-Trees on parameters like build time, memory footprint, lookup/inference time and fetch time. In this chapter we document the evaluation phase of the CRISP-DM model.

- In Section 4.2 we discuss the impact of hyper parameter tuning on the model by calculating the errors of the predictions made by the already trained models on different configurations (i.e., different learning rates, epochs, batch size and network architectures). Through these tests we are able to achieve principled choices regarding the hyper parameters to use for the subsequent evaluations.

- In Section 4.3 we discuss the build/training time for B-trees and learned index structures, while using the parameters selected in the previous section. A comparison between these alternatives is also done on aspects such as build time, memory footprint required by each structure and the query execution time/inference time.

4.1 Research questions

Recapitulating the research questions we established for this work, in this chapter we will be addressing the following specific research question:

1. Hyper-parameter impact in accuracy: What is the impact of hyper parameters in the resulting accuracy for these structures? What is their resulting memory efficiency? What are the exact training times?
2. Comparison of training time and B-tree build times: How does the best training time achieved for the learned structures compare to building a B-Trees Index?

3. Performance comparison for inference time: How does the inference time of a hand-tuned implementation, optimized through the use of SIMD, compare to the lookup time for B-trees? How does the performance compare when fetch time is added?

4.2 Hyper parameter tuning results

In this section we address the first of our research questions.

As discussed in the background chapter, designing a neural network requires a lot of hyper parameter tuning in order to build a model able to provide results with the least errors. As far as we know, there are no defined rules about how to tune a neural network in order to reach the optimal configuration. There are numerous aspects for which to decide. Some of them are listed below:

1. Type of problem: Is it a classification problem or a regression problem? What is the essential goal of the network? This is decisive for the structure of the network.
2. Activation function/expected output: What kind of output is expected from the neural network. If probabilities are expected, then sigmoid activation functions should be applied in the output layer, if values between -1 and 1 are expected, then tanh is fitting; if absolute values are appropriate then a rectified linear unit could be the choice.
3. Complexity of the problem: If the learning problem is very complex, then more hidden layers, number of neurons, or even specialized kinds of neurons will be required to learn the problem. Density can be expected to be proportional to the complexity of the problem\cite{6}.
4. Learning rate: More complex problems require more learning and examples, hence a smaller learning rate. The learning rate affects how much the weights are updated during back propagation.
5. Epochs/iterations: Complex problems will need more iterations to reach good accuracy.
6. Loss function: It depends on the type of problem. Commonly, mean square error is used for regression models and binary cross entropy is used for classifiers\cite{6}.

In order to understand the impact of hyper parameter tuning for the neural network that acts as a storage layer for the learned index structures, we performed some small tests. Apart from answering to our research questions, a core purpose of these test is to provide a reproducible evaluation to justify our choice of hyper parameters for the learned index structure that we build on the TPC-H dataset.

With this goal we trained the neural networks destined to be the child models of a learned index structure for range queries, using different configurations. These neural
4.2. Hyper parameter tuning results

networks were then fed with some test input data (order keys from the TPC-H lineitem table), and they predicted some positions. We tracked down the the root mean square error and the mean square error for all these different configurations. The configurations upon which training was done and the results are shown in the coming sections.

The tests were performed with the configurations listed below:

1. Learning rate : 0.1, 0.01, 0.001, 0.0001.
2. Density/architecture of the neural network : 1 hidden layer (8 neurons), 2 hidden layers (8 and 4 neurons, respectively), 2 hidden layers (with 4 neurons each), 4 hidden layers (with 2 neurons each).
3. Batch size : 16, 32, 64, 128, 256.
4. Number of epochs: 50, 100, 150, 250, 500.

Our choice of the configurations responds to the goal of testing separately each aspect.

4.2.1 Impact of learning Rate

![RMSE and MSE for different Learning Rate](image)

Figure 4.1: RMSE and MSE for different learning rates

The learning rate determines the extent of each update to the weights during the back propagation process. A very high learning rate makes it easy to change the mind of the network. For instance, if a network is learning about the eye color of humans, and there are 10 instances where it learns that human eye color is black and only 1 instance where it learns that human eye color is brown. With a very high learning rate the network could predict that all human eye colors are brown, given the fact that it has seen only one instance of brown color in the end, and 10 instances of black. In case of a lower learning rate where the network learns according to all data, given the example above the neural network might predict that all eye colors are black considering brown to be an outlier.
The results for our evaluation are shown in Figure 7.1. We report that, in practice, the learning rate is directly proportional to the root mean square error and mean square error: at a high learning rate the predictions tend to be incorrect and the error increases substantially, whereas when the learning rate is reduced, the predictions get better and better, and hence the error decreases.

One more observation that we can make from this brief test is that the tuning of the learning rate has indeed a high practical impact in the error, since the mean square error for a learning rate of 0.1 is 186.49 and that of 0.0001 is merely 0.3693.

As a result from this test we can select for our further studies the smallest learning rate observed, since it is noted to produce less errors in the learned model.

### 4.2.2 Impact of density/architecture of neural network

The density assigned to a neural network or the number of neurons placed per layer should be dependent on the complexity of the problem. So, by trying to increase the number of hidden layers or the number of neurons for a fixed problem, the model is liable to over-fitting the data. This means that it will learn for the training data but will not be able to establish the trend for new unseen data and will not be able to generalize.

![Figure 4.2: Impact of the number of hidden layers and neurons per layer on the learning](image)

Figure 4.2 shows the scenario of under fitting wherein the model is not complex enough to learn the problem and suffers from bias condition. The second image shows that the model is complex enough to learn the problem and generalize a trend for the future unseen values. The third image depicts the scenario of over fitting where the model learns the value but is not able to establish a trend for generalization.

Figure 4.3 depicts our observation on the impact of the architecture on the accuracy achieved by the neural network in learning the positions for the learned index structure.
4.2. Hyper parameter tuning results

By increasing the number of hidden layers, the accuracy on the task might or might not increase. By moving from one hidden layer with 12 neurons, to two hidden layers with 4 neurons each, there is a drop of the mean squared error. As a linear regression problem we have the minimum error with the configuration where we have just one hidden layer and 8 neurons. When we increase the number of neurons from 8 to 12 the error increases as well. If we increase the number of hidden layers the error slightly increases. With more hidden layers added we can see a notable increase in the error. Therefore, the conclusion is that for our kind of dataset, which is small and with a simple trend, adding additional hidden layers or neurons does not provide any benefits.

As a result we can select the simplest configuration from those we tested, for the further evaluations.

4.2.3 Impact of batch size

The batch size is the number of training samples the neural network will take in one training epoch (i.e., one forward pass and one backward pass). The batch size indicates how often the weights are adjusted.

Batch size needs to be chosen wisely since a smaller batch size will lead to less accurate estimates for gradients and also training will be very slow. Larger batch sizes may cause degradation of the model in terms of generalization beyond the training data set, but the training will be faster.
Figure 4.4: RMSE and MSE for different batch sizes

Figure 4.4 displays that the error for batch size 32 and 128 are smaller, when compared to the batch sizes of 64 and 128. From our results it is not possible to detect a specific trend.

Hence, for further evaluation we simply select the best batch size observed, 32.

### 4.2.4 Impact of epochs

An epoch means one forward pass and one back propagation pass. Epochs actually determine the total number of times the neural network will see the entire test dataset. Suggesting an optimum number of epochs for a problem is difficult as the number of epochs depends on the diversity of the dataset and the complexity of the problem. Few epochs lead to a neural network that underfits the data and gives large errors (in our observations, as shown in Figure 4.5, this happens for 50 epochs).

In our evaluation, shown in Figure 4.5, the optimum number of epochs is reached in the case of 250 since the error reaches its minimum in our observations. If we further increase the number of epochs (to 500 epochs), the model overfits the data, basically memorizing the data and failing to learn the trend, therefore giving inadequate predictions for new data.

From our results we can determine that 250 epochs is a reasonable configuration.

With this evaluation we conclude the presentation of our observations regarding the hyper parameter tuning process. We were able to determine the parameters that lead to lower errors in the model, and we also considered the impact that each has on the process. By far the architecture was the parameter that showed the highest impact on the errors observed (i.e., under mis-configurations, it lead to higher errors). This was followed by the learning rate, the batch size and the number of epochs. We noted that a fairly simple design with a single layer and 8 neurons, a low learning rate, with moderate batch sizes and epochs produce the best results. We summarize the findings in Section 4.4.
4.3 Inference of learned index structures versus B-trees

In this section we address the second and third of our research questions. Specifically, we compare the learned index structures with the B-trees, in terms of the build time, execution time of the query (or inference time) and the memory consumption.

4.3.1 Build time

4.3.1.1 Learned index structures

Learned index Structures require training before these models can be used for predicting the positions. We have measured the training times for the best possible configurations for both the parent model (classifier) and the child model (linear regression). The time taken by the parent model for training with the configuration, as shown in Figure 3.5, is 1461.61 seconds. The time taken by each child model, as shown in Figure 3.4, is 1710.87 seconds.

Figure 4.5: RMSE and MSE for different number of Epochs

In the next section we compare experimentally the B-trees with the learned index structure that we implemented.
We can save the configuration and learned weights of the best model trained in an hdf5 file. This can be reused later using the load model command to make predictions, or to train it further. We tracked the loading time of the models in order to make predictions, as shown in Figure 4.8. The total load time for all models is 8.24 seconds.

4.3.1.2 B-Trees

We chose two B-Tree implementations, one being inside a database, namely PostgreSQL. Here we loaded the line items table and created an index on the order keys. By default the index that is built in Postgres is a B-Tree. The time taken by PostgreSQL to create an index on B-Tree is 1.054 seconds. Whenever we try to create an index on Postgres, it uses the operating system’s cache. Once a query is executed the results are returned from the cache itself and not from the disk. In cases where the database is large and cannot fit into the memory then buffer managing strategies are adopted.

Implementation of B+ Tree in main memory: We used the STX B+ Tree package which is a set of C++ template classes. This class implements the B+ Tree in the main memory itself. The build time for B+ Tree on the same data set is 0.43511 seconds, as shown in Figure 4.9.

```
taran@LAPTOP-U9P01PNM ~/stx-btree
$ ./breetest.exe
It took 0.435111 seconds to build the B+ Tree:
```

The graph shown in Figure 4.11 summarizes our comparison of the build time between the learned index structures, the PostgreSQL index (B-trees) and the STX implementation of B+trees. We can see that the build time of the learned index structures is markedly high (with a difference between 3 and 4 orders of magnitude), in comparison to B-Trees or B+ Trees. This is mainly due to the training time of the models.

When considering only the loading of the model, compared with the build time of the trees, the models are still less efficient, with the tree build times being 8 and 20x faster.

<table>
<thead>
<tr>
<th>Index</th>
<th>Build Time (in seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Learned Index Structure</td>
<td>3172.48</td>
</tr>
<tr>
<td>PostgreSQL Index</td>
<td>1.054</td>
</tr>
<tr>
<td>B+Tree</td>
<td>0.435111</td>
</tr>
</tbody>
</table>

Figure 4.10: Build time comparison between the learned index structures and B-trees
4.3. Inference of learned index structures versus B-trees

Figure 4.11: Build time comparison between the learned index structures and B-trees

Thus from our comparison we note that, with our implementations B-trees are more efficient in their building process than learned index structures.

4.3.2 Query execution time/inference time

In order to compare the inference time taken by all index structures we have defined three specific cases, which are defined as follows:

- Look up time for a position when a key is given.
- Look up time + Data fetch (payload) from an array: In order to be comparable, since learned index structures do not store the payload, we employed a separate array to store the data and refer to it by position. We used an STL array, and we left randomly empty positions, corresponding to the sequence with gaps that represent the learned positions for the sorted keys. For B-trees we employed the same array in this configuration.
- Look up time + Data fetch (payload) from the B-tree: For this case we kept the data for the learned index structures in the array, but we stored the payload for the B-trees in as the value (eliminating the position).

The payload is defined as the remaining columns from the line item table, apart from the order key. We should also note that when fetching data, in our configuration the learned index structure looks one position to the right and one position to the left of the predicted position. We note that in our dataset this is sufficient to guarantee finding the expected key.

All the scenarios were tested by feeding 200 sequential and 200 random existing order keys, with 100 repetitions. We report the average execution time for these 100 repetitions was taken into consideration.
The best learned models were stored in hdf5 files. These files record information like the type of model, the number of hidden layers and neurons, learning rate, weights, etc. Such model can predict the position of a given key. The results of queries can be re-constructed using simple matrix multiplication of the weights, and applying the activation functions that we included. For efficiency we implemented this inference step in C++, using SIMD instructions. We did not perform optimizations such as weight quantization. Hence this corresponds to our implementation for the learned index structures.

### 4.3.2.1 Key lookups

In Figure 4.12 we show the results for our evaluation. B-Trees perform better than learned index structures in the case of looking-up the positions of sequential keys, taking 399 microseconds to get positions of 200 order keys whereas learned index structures take 1516 microseconds. This means that the former is roughly 4 times better than the latter. Hence B-Trees appear to be more efficient for lookups over sequential data, and they win over learned index structures. One of the reasons for B-trees to provide benefits for sequential data is that they are leveraging the cache, thanks to the adjacency of keys, and also they have a bounded cost for lookups. For learned index structures there is not such thing as adjacent keys, so this cannot be leveraged for improvements in memory usage. Nonetheless, learned index structures can gain from their small memory footprint.

If we take a look at the performance of the learned index structures for random keys, then we see that there is not much of a difference between the lookup times of learned index structures for sequential and random keys. In contrast, for the case of B-trees we see a jump in the lookup time (4.62 times slower) when going from sequential keys to random ones. So, for lookups the performance of the learned index structure is robust to random lookups, whereas the B-trees are impacted by this access pattern. In addition, learned index structures perform slightly better than B-trees for random keys (close to 20% better).

### 4.3.2.2 Lookup + fetch from array

The next case in our study was to retrieve the position of the order key, next to fetching the payload/data from an array (where the data is stored at the respective positions).
For the learned index structures the performance for sequential keys and random keys is roughly the same, which are 3408 microseconds and 3524 microseconds respectively. In case of the B-trees, the performance deteriorates more markedly with changes in the access patterns. For the sequential keys the time taken was 2972 microseconds, and for the random keys it was 4241 microseconds (40% worst).

If we compare the lookup time with the time of data being fetched, then both structures are observed to take a performance impact. Fetching data from an array causes more cache misses and hence leads to performance impact. However, the performance of the learned index structure does not vary with the randomness of the keys, and it also outperforms B-trees by a speed up of 1.2.

4.3.2.3 Lookup + fetch from B-tree

The third case proposed in our evaluation is retrieve the position and data from the index (i.e., the payload is stored in the B+ tree, and, for the case of the learned index structures we test with the payload in an array). When the data is stored in the B-tree, the performance deteriorates for both sequential and random cases, with a difference between them (10140 micro seconds to 11555 micro seconds), as the data fetch leads to more cache misses. The performance for the learned index structures remains the same.

A comparison of learned index structures and B-trees when fetching data from the index shows that the learned index structure outperforms B-trees, achieving speed ups of 3.2x.

In Figure 4.15 and Figure 4.14 we provide a visualization for our comparison between the learned index structures and B-trees as discussed in this section.

In our experiments B-trees are shown to perform better for look ups in case of sequential data, since B-trees are cache-optimized data structures. The performance starts deteriorating as soon as we start fetching data from an array or from the index, because the fetch would lead to cache misses. The performance of learned index structures remains the same irrespective of the fact whether the order keys are sequential or random. We note that this is an important factor to observe, which can have implications for using this structure.

In the case of random keys the gap between the inference time of learned index structures and B-trees grows, since the whole data might not fit into the cache and fetching all data would lead to cache misses.

In conclusion, as the data increases and with randomness of the accesses, the performance of B-trees deteriorates whereas learned index structures will be unaffected by changes in access patterns, leading to a comparably better performance when data grows.
Figure 4.13: Inference for sequential access patterns between learned index structures and B-trees

Figure 4.14: Inference for random access patterns between learned index structures and B-trees

4.3.3 Memory utilization

Figure 4.15: Memory footprint comparison between learned index structures and B-trees

4.3.3.1 Learned index structures

Memory utilization is an important selling point of learned index structures. The models that are already trained on the training data set and saved in a HDF5 file and can be re
loaded again for reuse. The size of the HDF5 files (parent and child model) on the disk is about 124KB (126,976 Bytes).

4.3.3.2 B-trees

Memory utilization in B-Trees is certainly greater than learned index structures. The size of the PostgreSQL index that was created on the order key column is 9,027,584 Bytes. For reference, The size of the dataset that was used for B+ Tree implementation was 2.68 MB (2,818,048 Bytes).

Figure 4.16 gives a comparison of memory space for each index.

It was expected that learned index structures would act as a form of compression, since query results could just be reconstructed by simple matrix multiplication of weights that are stored in the model files. The memory consumption of the learned index structure is the lowest (124KB), which means that the structure is 71 times more space-efficient than PostgreSQL indexes, and 21 times more than B+ trees.

With this we conclude our evaluations contrasting learned index structures and B-trees. In the next section we summarize the results.

4.4 Summary

- According to our evaluation, the architecture of a neural network has the maximum impact on the accuracy of the learned model, followed by the learning rate.

- If we try to increase the number of hidden layers or the number of neurons, it might increase the accuracy but will cause the model to over fit the data. Decreases, correspondingly, may cause under fitting of data.
• With a very high learning rate it is easy to change the decision of the neural network, with a very low learning rate some of the instances may be considered as outliers.

• Hyper parameters like epochs and batch size do have an outsized impact on accuracy, but they need to be configured after some evaluation too.

• In our study we find that child nodes for the learned index structures designed to work on the order key of the line item table of the TPC-H dataset, (with the addition of a column to indicate the positions), can be designed with 1 hidden layer only, a small learning rate and moderate values for batch sizes and epochs.

• If we compare the build time of learned index structures and B-trees, then the training time of the learned index structures is remarkably high (3172.48 seconds). Even the load time of the models is 8.24 seconds, whereas a B-trees can be build in PostgreSQL for the same use case in 1.054 seconds and B+ trees can be similarly build in 5.014 seconds. This is a notable gap separating the models.

• B-trees are better than learned index structures when sequential data is given and for lookups only. But if the data is fetched along with the lookup, then the performance of the B-tree deteriorates. Cache misses contribute to this deterioration.

• In case of random data, the learned index structures outperform B-Trees for lookups and data fetch. B-Trees continue to deteriorate when data is increased, whereas the learned index structures are less affected. The robustness of learned index structures to different access patterns is an important property that should be noted from these structures.

• Memory consumption of learned index structures is the lowest amongst all the considered alternatives. We just need the model files of already trained models in order to predict the position of the Order key. LIS is 71 times more space efficient than PostgreSQL indexes.

• Hence the comparison of learned index structures and traditional indexes can be summarized as in the table below:

<table>
<thead>
<tr>
<th>Index</th>
<th>Build Time</th>
<th>Lookup Inference Time (Random Keys)</th>
<th>Memory Consumption</th>
</tr>
</thead>
<tbody>
<tr>
<td>Learned Index Structure</td>
<td>✗ ✗</td>
<td>✅ ✅</td>
<td>✅ ✅</td>
</tr>
<tr>
<td>B+ Tree</td>
<td>✅ ✅</td>
<td>✅</td>
<td>✗</td>
</tr>
</tbody>
</table>

Figure 4.17: Summary table for comparing structures (Learned Index Structures V/S B-Trees)
4.4. Summary

With this summary we conclude this chapter on hyper-parameter tuning and in comparing the learned index structures with B-trees. In the next chapter we discuss and assess a novel concept for insertions in learned index structures.
5. Insertion in learned index structures

One of the research questions that was not addressed by Kraska et al.[1] was how to insert new data items into the learned index structures. In this chapter we propose a simple way in which this could occur. In addition we evaluate some design alternatives, such as the employment or not of pre-trained weights.

This chapter documents the evaluation phase of the CRISP-DM methodology.

We structure the chapter as follows:

- In Section 5.2 we discuss our concept for data insertion into the learned index structures. This section describes in detail when training is required for the new data, what will be the configuration of the new model, etc. In Section 5.2.3 we propose an algorithm for the search operation after insertions have been performed and the models have been trained on the new dataset.

- In Section 5.3 we present a prototypical implementation for how to carry out the batch insertions, and we use it to evaluate design alternatives, such as training the model with/without random weights and with/without the offset concept.

5.1 Research questions

In this chapter we aim to answer to the following question:

4. Insertions: How can insertions happen in learned indexes?

We specifically consider the case of insertions happening in batches, with new child models being created in the same style as described in the original paper. However, to
avoid hyper parameter tuning we expect that the same architecture and learned weights could be used as a starting point for the training.

Under this setup we observe that the new set of keys can either map to the same range of positions from the previous set, or to a new range determined by an offset. In the first case the network needs to relearn the mapping, and thus it can be more costly and perhaps require a different architecture and hyper parameter tuning.

In the following sections we present our concept for inserting data in batches to the learned index structure, we also discuss and evaluate the design alternatives of training the new models with/without random weights and with/without the offset concept.

5.2 Our proposal: Batch insertions into the learned index structure

5.2.1 First approach

![Figure 5.1: Model architecture for how we propose insertions could be done on the learned index structures](image)

Figure 5.1: Model architecture for how we propose insertions could be done on the learned index structures

Lets recall the learned index structures, as shown in Figure 5.1, we have a series of parent models (classifier models) and children models which are the leaves (regression models). Each parent model is trained to classify keys into child models and the child models are responsible for predicting the positions.

On top of this structure we have introduced a small sorted array per child model, which corresponds to corrections on the dataset. Once a new key arrives, it is first mapped to a child model by the parent, then it is inserted in the array corresponding to this child model, and this array is kept in a sorted manner. We propose that once the amount of data in this array reaches a threshold, a training process can be started such that a
new child model is added, as an appendix, to the original one, being trained only on the corrected dataset. The new child models are trained using the same configurations (architecture, learning rate, epochs, batch size, activation functions) as that of the child models.

In order to support this lifecycle, lookups will take place in the following way: A lookup key is fed into the parent model. First a check is performed at the corrected data set to check if the model is trained or not trained on the lookup key. If the key is in the corrected data set then the values from the corrected data set are returned. If the key is not present in the corrected dataset, then the key is passed to the corresponding child model, where the position could be looked for. Since the model will not predict with 100% accuracy, we add and subtract an epsilon value (very small value) to the prediction made by the child and new child models (same as done by Kraska et al. [1]). The range to be looked up in the data dictionary will be from prediction - epsilon to prediction + epsilon. The range of prediction for positions is given by both the child model and the new child model (model trained on the corrected dataset). Each model gives a range prediction based on the data distribution learned by the model itself. This range is then fed to key based lookup for random key to get a set of random key positions.

### 5.2.2 Second approach, based on offsets

Figure 5.2: Model architecture for how we propose insertions could be done on the learned index structures, using an offset

While working on the above mentioned concept an interesting observation was noticed: The model mentioned above works reasonably well for data having the same distribution. However, if the new corrected data set has a different data distribution than the previous child model then we again need hyper parameter tuning in order to give predictions with the same accuracy. This means that every time training happens on the corrected dataset hyper parameter tuning is required. In order to avoid this
situation we introduce the simple concept of an offset. An offset is a constant number that is subtracted from the positions (which needs to be learned by the models) such that the data distribution trend is similar to the above trained child model. This is done in order to avoid hyper parameter tuning.

The prediction that we obtain from the new data child model is again added back to the offset. This sum value (prediction + offset) is then looked upon in the data dictionary and a position is returned. The new recursive model index is now as shown in Figure 5.2.

### 5.2.3 Searching on the learned index structure after the batch insertions

The algorithm for prediction of position for a lookup key is as defined below.

**Algorithm 1: Search in learned index structure, with our concept for batch insertions**

Require: key, epsilon

1. if key in CorrectedDataset then
2. return CorrectedDataset.position((key)) //The position given
3. end if
4. PredictedModel = ParentModel.predict(key)
5. PositionCM = ChildModel[PredictedModel].predict(key)
6. PositionSCM = SubChildModel[PredictedModel].predict(key)
8. PositionCMArray[] = all positions in range (PositionCM - epsilon, PositionCM + epsilon)
   PositionSCMArray[] = all positions in range (PositionSCM - epsilon, PositionSCM + epsilon)
9. return PositionSCMArray[], PositionCMArray[] //These are the positions where the key could be found

In the algorithm above we expect our model to return the positions where the key could be found. If the key is in the corrected dataset (and not in any subchild model), then it is returned from there (steps 1 and 2). Otherwise it is necessary to consider which child model is likely to contain the key (step 4), and we will ask for positions from the child model and the subchild model (steps 5 and 6). In order to search for the positions of the subchild model in the table we must add the offset (step 7).

The offset helps the algorithm to learn for different data distributions. Epsilon is the range value (Prediction-Epsilon, Prediction+Epsilon) that we set where a lookup will be performed in the lookup table.

### 5.2.4 An end-to-end example

This could be better explained with an end-to-end example, as shown in Figure 5.3.
5.2. Our proposal: Batch insertions into the learned index structure

In the example Figure 5.3 we can see that we have a trained dataset (Order keys = 1,3,5,..., 199997,199999) which consists of data items for which the child neural network has already been trained. The second column in this dataset corresponds to the real positions of the values. These, for example, could be very scattered. The sorted position corresponds to a value that the learned index model can learn such as to find the keys. The input for the recursive model index will be the order key and the output will be the sorted positions (1,4,6,...). The neural network in RMI learns the distribution function between the order key and the sorted position. A look up table needs to be maintained, where the key is the sorted position and the value is the random position at which the order key is located, hence the lookup proceeds here and then the fetch needs to go to the position assigned.

Once the training happens, we consider the case that there is a new data set (order key = 2,4,6,...,199998,200000) having a completely different data distribution (300002,300004,300006,...). These data items remain in the corrected dataset until a threshold of the number of items is reached. Once the threshold is reached a new sub child neural network is now trained upon this data set and the corrected data set is now emptied. Since the data distribution is completely different (starting from the difference in the ranges) and the neural network currently has the configuration of the child model, an offset value is subtracted from the SortedPos and the resulting positions are now fed to the neural network as the output. The order keys and the new position which is SortedPos - offset have at least comparable data ranges, which aids in training the new model.

The sub child model neural network now learns the distribution function between order key and SortedPos - offset. The lookup table is updated with the key being the SortedPos
and the value being the value stored of the physical position.

Once the sub child neural network is trained on the new data set, the overall model is ready for the search operation.

For clarity we emphasize that the child model is the model that is trained on the trained dataset and the sub child model is trained on the corrected data set.

As an example for the search functionality, lets consider retrieving the key 2 from the model. The value 2 goes through the parent model which classifies it to the child model 1 and sub child model 1. Both the models (child model and the sub child model) will predict positions as per their own data distribution. Child model will predict some value between 1 and 4, for example 3. The sub child model predicts 2 (whereas the actual value should be 300002, here the offset is 300000). Two range values will be fed to the look-up table, one will be from child model which is 3 (3 - epsilon, 3 + epsilon) and another will be offset+2 ((offset+2) - epsilon, (offset+2) + epsilon)).

In Figure 5.4 we can see that for the first key = 200001, the model is not trained, and hence we just get the results from the corrected dataset. In the second prediction where key = 70960, the model is trained on these values. First we get the index of the child model which is one, and the key is now fed to both the child(first) and the sub child(second) models with the index 1. The predictions that we get from both the models are then fed to the key-based lookup for the positions.

---

**Figure 5.4: Results for the insertion concept**
5.3 Evaluation of design alternatives

For assessing the impact of design alternatives on the accuracy of the models for inserting data, we have two different types of data sets:

1. Line items original: Where the keys have sorted positions like 1, 3, 6, 7, ..., etc.
2. Line items new: Where the keys have positions like 300001, 300003, 300006, 300007, ..., etc.

The parent model (classifier) is trained on the original data only. The child models are then trained on the same data set. The input parameter is the key, and the output parameter is the position. For the second data set the parent model classifies the new line items data set into new data child models (sub child models). Since the data distribution is different for this new data set (positions now are 300001, 300003, ...) the neural network needs re-tuning of hyper parameters for it to learn with the same accuracy.

In order for the new model to be comparable, at least in ranges, to the original one (such that the hyper-parameter tuning, and learned weights can be used as starting points to build the new model), an offset of 300000 can be established and the position to be learned can now be mapped to position-offset. The new data child models can be trained with the key as input and the difference position-offset as the output, from the new data set while using the same configuration of the original child model. The search can proceed as explained in Section 5.2.4.

In this section we provide the results of our tests for the concept of insertion that we propose, under the following design alternatives:

1. Without subtracting the offset and taking random weights as initialization for the new data child model.
2. Without subtracting the offset and initializing the weights of new data child model with the weights of the already trained child models.
3. Subtracting the offset and taking random weights for the new data child model.
4. Subtracting the offset and initializing the weights of new data child model with the weights of the trained child models.

5.3.1 Model implementation with subtracting the offset

5.3.1.1 With random weights

[Figure 5.6] depicts the accuracy of the learned models and the mean square error, for the first data set, plus the results for the new models on a new data set. In this case the new model uses the concept of the offset for building the new data child model, which has been initialized with random weights. The left image corresponds to the results after training the models for the original line items dataset. The right image is for the models (sub child models and parent) on the new line items dataset (with the offset and random weights assigned to the new trained model).
5. Insertion in learned index structures

5.3.1.2 With initialized weights

Figure 5.6 depicts the accuracy of the learned models and the mean square error, for the first data set, plus the results for the new models on a new data set. In this case the model has the concept of offset on the new data child models, but they have been initialized with the weights of the former child model. The left image is for the original line items dataset. The right image is for the new line items dataset (with offset and weights initialized from the previous child model). Since this approach has the least errors, this will be our proposal for the insertion operation for the learned index structures. Specifically we propose that the method to add new models and hence insert data into the learned index structure should adopt offsets and use the initialized weights from the child model.

Figure 5.6: Line items original, Line items new (Offset,Initialized weights)

However, we note that there is not a big difference between random and initialized weights when the offset is employed.

5.3.2 Model implementation without offset

5.3.2.1 With random weights

Figure 5.7 depicts the accuracy of the learned models and the mean square error, for the first data set, plus the results for the new models on a new data set. In this case the model doesn’t have the concept of offset on the new data child model which has been initialized with random weights. The configuration for the model is the same across all tests. The left image shows the results for the original dataset. The right image portrays the results for the new line items data (without offset and with random weights for the newly trained model).

Figure 5.7: Line items original, Line items new (Random,Random weights)
5.3. Evaluation of design alternatives

5.3.2.2 With initialized weights

Figure 5.8 shows the accuracy of the learned models and the mean square error, for the first data set, plus the results for the new models on a new data set. In this case the new model doesn’t have the concept of offset, and it has been initialized with the weights of the child model. The left image is for the original line items. The right image is for the new line items (without offset and weights initialized from the previous child model to the new trained model).

We would like to summarize the results in a table as shown in Figure 7.2. The table depicts the mean square error of the sub children when the trained sub child model tries to predict the position for certain dataset. Here it is clearly visible that the best performance is that of the sub child models with the offset implementation. However, after such distinction, the use of initialized or random weights does not have a notable impact.

![Table depicting MSE for sub child models following alternative designs for the insertion concept](image-url)
With this we conclude our presentation for our concept for inserting data in the learned index structure, and our evaluation of design alternatives that we considered for training the models.

5.4 Summary

- In this chapter we presented a novel concept for inserting data in the learned index structures: newly arrived data items are stored in a separate storage until a threshold (e.g. number of data items) is reached. A new child model can then be built with the same configurations as the model already trained. This model can be added as a sub child model to the recursive model index structure already built. Now we will have two position predictions, as seen in Figure 5.4, one from a child model and another from a sub child.

- In order to aid the training and configuration of the new model, we propose the use of an offset such that the positions already learned are similar in range to the positions to be learned. This is mainly done because learning a different data distribution will require a careful re-tuning of the hyper parameters. Regarding this insight we tested the training for new data sets with two scenarios, as listed below:

  - The first scenario is that we subtract the offset from the positions that need to be learned. This further has two possibilities where the model further is trained using random weights or initialized weights (of previously trained child model). We can see from Figure 7.2 that the error achieved is low (in green).
  
  - The second scenario is that we don’t subtract the offset from the positions that need to be learned. Sub child models are further trained with randominitialized weights and the error is checked. From Figure 7.2 we can see that the error is huge.

- The results of training the models based on the above scenarios prove that insertions on new dataset are possible by subtracting the offset from the positions (in case of different data distributions) and then training the models (with random weights or with initialized weights)

- Some considerations can be made regarding our approach, we include them in the following:

  - If we consider parameters like build time, inference time and memory consumption for insertions, then the build time will be less since the only addition is the new child model and training of child model is going to take 1710 seconds. In our design the parent model is already trained to classify the new child models.
– Memory utilization will also increase since we will have to store an extra model file that contains the weights for the new sub child model.

– The execution time can be made to be roughly the same, since the predictions for both models can happen in parallel.
5. Insertion in learned index structures
6. Related work and Future Directions

6.1 Related Work

Learned index structures could be extended to a wide range of use cases and applications. In order to get more knowledge and to provide a systematic, reproducible study, we carried out the following keyword searches in Google Scholar: “B-Trees”, “Deep Learning”, “Learned Index Structures”, “Hyper parameter tuning”, “Software 2.0” and “Auto ML”. Some of the search results and related work in this field are mentioned below:

6.1.1 B-Trees and its variants

The B-Tree is one of the most commonly used index structures. Apart from the basic implementation a lot of variants of B-Trees have been designed for optimization and performance benefits. These specialized kind of indexes are employed for specific use cases, mostly related to supporting range queries. One of the variants of the B-Tree is the B+ Tree [17], the only difference is that the non leaf nodes stores that keys and the leaf nodes stores all the records. B+ Tree offers faster record traversal, record search and it also avoid overflowing pages. It is generally used for data that is stored in RAM. UB-Trees [18] or the Universal B-Trees (linear space for storage and logarithmic time for insertion, updates and deletes) are balanced just like B-Trees, provides the same semantic guarantees and can be used for multi-dimensional indexing. Here records are being stored in the form of Z-Order. Several secondary indexes could be replaced by a single UB-Tree and can join indexes which would reduce the storage cost and runtime. H-Trees [19] are specialized indexes that are used to index the superclass subclass relationship and their instances. These kind of specialized index structures are build to support Object Oriented Databases. H-Tree indexes are the indexes which are maintained just like B+ Trees. A concept of nested indexes is introduced into it where $H_{superclass}$ is one index structure within which there is another index structure.
For instances of subclass. In order to improve the efficiency of index structures several cache optimized index structures like Cache sensitive search Trees [20] have been introduced which have better lookup in terms of execution time as compared to B-Trees. Even though B+ Tree is more cache efficient than other index structures still most of the space is being used for storing pointers. CSS tree is a variant of B+ Tree that stores every child node adjacent to each other and just stores the address of the first child node. The address of rest of the children could be found by adding an offset. However a lot of work is also going on in the direction of making use of SIMD instructions [21] and making use of GPU capabilities [21] in order to improve index structures. HB+ Trees [22] is one of the specialized index that makes use of CPU-GPU together for computing and memory resources.

6.1.2 Software 2.0

Software 2.0 is an idea introduced by Andrej Karpathy where the author considers that neural network marks the beginning of the way we write the software, termed as Software 2.0. We all are already familiar with Software 1.0 which includes language like Java, C++, Python etc. We know what kind of operations we want to do and accordingly we write the code. In contrast software 2.0 is abstract (example weights of the neural network). Here we just create a rough sketch which identifies a search space to search for a program that works in the direction of achieving a particular goal. Back propagation and stochastic gradient descent algorithms are used in order to make the search efficient.

6.1.3 Hyper parameter tuning

6.1.3.1 Automatic Hyper parameter tuning

Keeping in mind the manual effort that takes place for hyper parameter tuning. We all realize the that hyper parameter tuning is a crucial part in machine learning algorithms. Over the time manual hyper parameter tuning is a tedious and crucial task. Several automatic hyper parameters tuning methods are also available, for instance surrogate based method [23] or local search based methods [24] etc. These kind of automatic hyper parameter tuning methods have been successfully applied on neural networks. An example of this can be seen in the paper [25], where surrogate based methods have been applied on Deep belief Networks (DBNs) for tuning its hyper parameters. On the similar grounds this paper [26] also applied the same methods on large number of classifiers for a huge set of problems. Even after the automatic hyper parameter tuning, something better could come out if these optimization strategies could be made general and can be applied to similar learning problems. An easy explanation to the above scenario is the way our brain takes hints from the past learning experience. If a classifier has been trained to problem A and a new problem B which is similar to A needs training, then the optimization strategies should take hints from already tuned hyper parameters for problem A. This is the main contribution of this paper where it tries to take hints from previous optimization while trying to implement the surrogate
6.1. Related Work

6.1.3.2 AutoML

AutoML [28] stands for automatic machine learning is a suite of Machine Learning Algorithm with which one is able to train and validate the machine learning models without much knowledge of algorithms nuances. A standard work process for a machine learning model comprise of few stages like data acquisition, exploration, feature selection, model and experimentation, and finally with prediction. AutoML will just focus on two steps which are data acquisition and preparation. Whatever steps are required between these two process will be taken care by AutoML. Ideally a user will have his on data set and the labels are defined, which is fed to AutoML. As an output the user will get a trained model that is ready to predict. All the relevant processes like choosing the best algorithms, hyper parameter tuning etc happens at the back end of AutoML.

6.1.4 Extensions to Learned Index Structures

6.1.4.1 Criticism : LIS(blog post by Thomas Neumann) :

In this blog [29] the author argues that instead of using neural network for approximating the cumulative distribution function other functions could be used for approximation (for example spline interpolation). The author creates an analogy between traditional B-Trees and splines. In case of spline we can assume search key as the position between keys and we can interpolate the position of the search key. However this search position will have a little error and the interpolation strategy could be used while navigating through the keys or the offsets at the lower levels. Tests were done on map dataset and log normal data set of same size as that of neural network. As reported by the author for map dataset the learned indexes perform more accurately and the interpolating B-Trees were also comparable enough. For log normal dataset the interpolating B-Trees perform more accurately than Learned Index Structures. Author also points out the performance of Learned Index structure with that of B-Tree. With the result set that the author has it appears that B-Tree performs fairly well, so the author still believes that B-Trees are the best choice for indexing.

6.1.4.2 Criticism : Classical Data Structures That Can Outperform Learned Indexes (by Peter Baillis) :

This blog [30] addresses the second use case of the paper [1] where LIS promises a memory reduction of Hash tables by 80%. The author suggests another structure called cuckoo hash tables [31] which proves to be more space efficient (5-20X less overhead) and 2X times faster than the LIS. Learned Hash Maps implementation suggested that if these structures learn the Hash Function that maps the keys to position, then the distribution of the keys would be even and it will lead to less collisions. Hence, being more space efficient and having faster access. The authors of this blog implemented a bucketized cuckoo hashing (a technique that achieves 99% occupancy and all lookups
just in two memory access). It doesn’t require any training unlike the LIS and can also accommodate changes to the key distribution during run time itself. They tested this approach on a map data set and on a log normal data set. For both the data sets their table took 36ns per access and wasting 0.015 GB of space. The learned functions had around 5% to 20% of slots empty. As closing thoughts the authors do suggest that memory access is expensive as compared to cycles and also that Machine learning models can approximate a function very well but there are other structures that can perform significantly well and we should also consider them as better representatives of a baseline alternative to learned index structures.

6.1.4.3 A Model for Learned Bloom Filters and Related Structures:

This paper [32] is written by Michael Mitzenmacher where he tries to build a more concrete model for the Learned Bloom filters proposed by Kraska et al., [1]. The author points out the merits as well as the demerits of the Learned Bloom filters and asks whether they will actually be suitable to address real world problems. This paper addresses the same problem of potential false Positive Rate which as per the authors of Learned Index Structures [1] could be addressed by adding an additional filter for the values that fall below a threshold. The question still remains the same as how we decide the false positive rate and the threshold value. One of the suggested way is to empirically find the optimal false positive rate over a test data but this approach seems feasible only in case of standard bloom filters where the false positive rate is close enough to the expected output. This paper focuses on how false positive rate could impact the query set and how this may affect the application that is using this kind of approach. The authors also suggest to use variations of bloom filter for comparison with Learned Bloom Filters.

6.2 Future Work

Learned Index Structures have proved that there is potential by utilizing correlations and patterns within the data using Machine Learning algorithms. Such approach has opened doors for many research questions, and for addressing real world problems. One of the interesting research question would be multi-dimensional indexing. We would like to present a novel idea of how Learned Index approaches could be extended for multi dimensional indexing.

6.2.1 Multi Dimensional Index

Traditional 1 dimensional index has already been implemented using Learned Index Structures and the results have been shown in Chapter 4. In a 1 dimensional index we learn the Sorted Linear positions which are then fed to a global lookup table where we have the Sorted Linear positions and real position stored as a key value pair.

In the [Figure 6.1] above we can see that if we query 1D indexes for a range of Order Keys between 10 and 25, it returns range for the learned positions between 1.5 to 5.2.
We get an array of learned positions (2,3,4,5) which is fed to the lookup table and the real positions (300003,300004,300005,300006) are obtained. Keeping the same concept of Learned Index Structures we have built a conceptual idea of extending the same approach to multi dimensional indexes.

In Figure 6.1 we have another table that has positions (sorted) for 1D Keys, there are positions for 2D Keys which are unsorted. We introduce a concept of junction which comprises of the sorted positioning of the 2D Keys and the actual positioning as well. When we train the 2D model, the model learns the sorted positioning which is later fed to the lookup table for real value. If there is a query that is asking for positions for keys between range 5 and 10, the 2D index model will return a range of learned positions 1 to 5 (2D Key 5 is at sorted position 1 and 2D Key 10 is at sorted position 5). These sorted positions are checked for the corresponding original Positions. The array
of original positions \((0,2,3,5,6)\) is fed to the look up table for real positions. The last kind of query that we can construct is the join query between 1D and 2D. In this query the result will be the inner join set of the positions returned by both 1D index and 2D index. In our example the return positions for join on 1D \((2,3,4,5)\) and 2D \((0,2,3,5,6)\) is an array of positions \((2,3,5)\).

The question that arises is that how many models do we need to train for Multi dimensional Indexes. The answer is that the number of models is equal to the number of dimensions [Figure 6.2] of the indexes since every time for a new dimension a model needs to learn the sorted positioning present in the junction data set.

With this we conclude our discussion of some possible future work (essentially the extension to multi-dimensional cases) and of recent related work to the learned index structures.
7. Conclusions

In this thesis we study techniques for indexing. Indexes are indisputably relevant for efficient data retrieval in today’s big data context. Traditional indexes, which are generic in nature but have some configurable parameters (e.g. page sizes for B-trees), are commonly used in mainstream applications. However, authors have noted that, due to their generality these indexes might not be able to fully exploit knowledge about data distribution and could not be suitable for worst case scenarios [1]. In fact, authors have made the case that learned index structures, an approach in which indexes are substituted by learned models, could be a potentially valuable solution towards developing data structures. Concretely, the authors of “The Case for Learned Index Structures” [1] present the implementation of specialized solutions that adopt machine learning algorithms, identifying patterns and correlations within the data with limited engineering cost.

In this Thesis we considered this approach, aiming to address open, unanswered questions in their work. More specifically, we presented a principled approach towards building one of such structures (i.e., a learned index for range queries, comparable to a B-tree) for the TPC-H dataset, using a single attribute from the line items table (i.e., order key). To this end we followed the CRISP-DM methodology. We followed the stages of data and business understanding, modeling and evaluation, describing our approach to hyper parameter tuning, leading to a configuration that we adopt for our tests. Next we compared the performance of the learned index structures with that of B-tree implementations, in build and search times. We found that the build times are hardly comparable, with B-trees being by far the most efficient. However, with regards to searches we found both approaches to have their competences. Learned index structures where robust for access patterns, providing a similar behavior for random or sequential lookups; building on this fact they could offer speedups for data fetch, reaching 3x differences when compared with data stored in B-trees. B-trees on the other hand proved to be efficient for sequential lookups. Finally learned index structures were shown to achieve a notably lower memory footprint.
Next we proposed and evaluated a concept for inserting data into the learned structure. We tested different design alternatives and, by the end, were able to propose that offset-based solutions are worthwhile of considerations. To conclude we also proposed a concept for extending learned index structures to handle multi-dimensional indexing, a topic which we look forward to investigating in future work.

Thus our core contribution is a practical evaluation of learned index structures, highlighting their robustness to access patterns and limited memory footprint. Next we contribute with novel concepts for insertions and multi-dimensional data management with these structures.

In this chapter we conclude our work as follows:

- We summarize our contributions (Section 7.1).
- We disclose threats to the validity of our work, which should be considered (Section 7.2).
- Finally we close this Thesis with some small remarks (Section 7.3).

### 7.1 Our contributions

There is a large list of unanswered questions from the work of Kraska et al. [1] out of which we have chosen a few research questions that we addressed in this thesis:

1. Hyper-parameter impact in accuracy: What is the impact of hyper parameters in the resulting accuracy for these structures? What is their resulting memory efficiency? What are the exact training times?
2. Comparison of training time and B-tree build times: How does the best training time achieved for the learned structures compare to building a B-Trees Index?
3. Performance comparison for Inference time: How does the inference time of a hand-tuned implementation, optimized through the use of SIMD, compare to the lookup time for B-trees? How does the performance compare when fetch time is added?
4. Insertions: How can insertions happen in learned indexes?

In order to answer these questions we assumed the following:

- We employed the recursive model index approach (hierarchy of models) to implement a learned index structure in charge of supporting range queries.
- Our dataset was the line items dataset with scale factor 1, as generated by DBgen and consisting of just one column (order key), unless specified otherwise.

\[1\] We highlight some of these questions in Section 3.1
7.1. Our contributions

• The implementation for our recursive model index consists of a parent model (classifier) and four child models (regression). The configuration for all models was determined through a hyper parameter tuning process, as described in Chapter 4. Specifically the following was tested:

1. Learning Rate: 0.1, 0.01, 0.001, 0.0001
2. Density/Architecture of Neural Network: 1 Hidden Layer (8 Neurons), 2 Hidden Layer (8 and 4 Neurons), 2 Hidden Layer (4 Neurons each), 4 Hidden Layer (2 Neurons each)
3. Batch Size: 16, 32, 64, 128, 256
4. Number of Epochs: 50, 100, 150, 250, 500

• The results that we obtained proved that the architecture of a neural network has the maximum impact on the accuracy of the predictions and errors which is followed by the learning rate, number of epochs and batch size.

• The best configuration for the parent model (classifier) with an accuracy of 93.35% was:

  – Architecture: Input (1 Neuron) → Hidden Layer 1(12 Neurons) → Hidden Layer 2 (8 Neurons) → Output Layer (4 Neurons)
  – Learning Rate - 0.0001
  – Batch Size - 128
  – Optimizer - Adam
  – Epochs - 100

• The best configuration for the child models (regression) was:

  – Architecture: Input (1 Neuron) → Hidden Layer 1(8 Neurons) → Output Layer (1 Neurons)
  – Learning Rate - 0.0001
  – Batch Size - 128
  – Optimizer - Adam
  – Epochs - 100

• The best configuration for implementation of a Bloom Filter as a classification problem was:

  – Architecture: Input (1 Neuron) → Hidden Layer 1(8 Neurons) → Output Layer (1 Neurons)
  – Learning Rate - 0.0001
  – Batch Size - 128
  – Optimizer - Adam
7. Conclusions

- Epochs - 100

- In the context of parameters like build time, memory consumption and inference time we evaluated the performance of learned index structures with B+ trees and B-trees (index created on order key, in PostgreSQL).

- Learned index structures took 3172.48 seconds for training and 8.24 seconds for loading, whereas B-trees took 0.43511 seconds for building.

- The inference time of learned index structures is less in comparison to B-trees when a lookup is performed on random data. In fact, we see that learned index structures are robust to differences in access patterns. Also the performance of B-trees worsens when a data needs to be fetched, due to cache misses. A speed up of 1.18, 1.20, 3.27 could be seen for LIS for lookups, lookups with data fetch from and array and look ups with data fetch from the index (i.e., where the data for the row was stored in the B-tree), respectively. B-trees performed better for lookups on sequential data.

- The memory footprint of learned index structures was studied by considering the generated model files consisting of the model weights. These structures are 21 times more efficient than B-trees in their memory usage.

- A summary on the evaluations comparing B-trees with the learned index structures is shown below:

<table>
<thead>
<tr>
<th>Index</th>
<th>Build Time</th>
<th>Lookup Inference Time (Random Keys)</th>
<th>Memory Consumption</th>
</tr>
</thead>
<tbody>
<tr>
<td>Learned Index Structure</td>
<td>✗ ✗</td>
<td>✓ ✓</td>
<td>✓ ✓</td>
</tr>
<tr>
<td>B+ Tree</td>
<td>✓ ✓</td>
<td>✓</td>
<td>✗</td>
</tr>
</tbody>
</table>

Figure 7.1: Summary table for comparing structures

- We presented a concept for inserting data in batches to learned index structures, by appending new child models, in charge of learning the positions for the new data. Our study in this regard can be summarized as follows:

  - New data is added to a separate storage until a threshold is reached. Once the threshold is reached, a new child model can be created and trained on the new dataset.

  - Since the data distribution of the new dataset (especially the ranges for the positions) can be expected to be different from the original one, hyper parameter tuning might be required to find the optimal architecture. One practical approach that we evaluated was to determine an offset, as a mechanism to mapping the new positions to a range that could be similar to the original ones (hence, enabling the reuse of the architecture and configuration). By using the offset we map all positions to the position with the offset subtracted,
and we train the dataset with the position - offset. Once the model is trained this offset is added back to the prediction given by the model.

- We explored several possibilities for the insertion with/without subtracting the offset and also using random weights or initialized weights (weights from the previous child model). Results showed that the maximum accuracy is obtained by subtracting the offset (colored in green) in case the data distribution of the corrected dataset is different from the old dataset. The table below shows the resultant errors on the newly trained models with respective scenarios.

![Table depicting MSE for Sub child Models](image)

Thus we can summarize our work. We discuss future directions in Section 6.2.

### 7.2 Threats to validity

In this section we list some threats to the validity of our study.

- **Internal threats :**
  - In our work we performed and documented some tests for hyper parameter tuning to determine the most fitting model for the task. Related works in the field of automatic hyper parameter \[23\] could constitute an improvement over our approach, helping us to achieve comparatively accurate models, with potentially simpler architectures leading us to a lower cost in inference. Similarly, more advanced supporting tools to understand the impact of parameters could have lead us to better configurations.
  
  - The training of the models was done on a off-the-shelf computing platform (Processor (i3-6006U CPU @ 2GHz), RAM (8GB)). We consider that with a more powerful machine the models could be trained in a similar time, to achieve better accuracy, and hence produce a better build time.
  
  - In our implementation of inference for the learned models we did not employ optimizations like quantization. Furthermore, other optimizations in the use of SIMD, or in exploiting multiple threads, could have been explored to further improve the performance. Hence our implementation for inference might not be sufficient to reflect the learned index structure’s ability to perform speedy lookups.
• **External threats :**

  - Datasets, the choice of machine learning algorithms, or the technologies used (e.g. the STL array implementation) may have affected the performance of the results failing to make them sufficiently representative of the intended application. Specially tests with other datasets and types of keys seem necessary.

  - Although our tests have proved that learned index structure’s performance and memory footprint can outperform B-Trees and B+ Trees, specially in data fetch and lookups with random keys, for the use case that we evaluated, we also cannot rule out that there might be other structures that may perform better, and hence would be better candidates for a performance comparison (e.g. for point queries, skiplists or BW-trees). Peter Bailis gives an example of this [30], which we discuss in the related work section, where cuckoo hash implementations are shown to perform better than the learned hash maps. Therefore, an internal threat to the validity of our observations is the possibility of having chosen inadequate data structures for comparison. This also includes the possibility of us choosing inadequate configurations (e.g., the size of nodes, for which we employed the default of 128 bytes), or implementations.

  - Regarding the insertion concept, in this solution a new model is created every time batches of data are inserted. Each model gives its own prediction for the lookup key. We believe that since this model will grow linearly with addition of new models (when training happens). Due to the growth of the recursive model index, B-Trees can be expected to perform better than learned index structures at some point of time after a sufficient amount of models have been created. Hence our observation on the training accuracy might reflect insufficiently on how these models should be built for a sustainable and efficient use.

### 7.3 Concluding remarks

Based on our study we are able to offer an open-source implementation of a learned index structure, next to reproducible steps towards building it. We hope that our research can contribute towards more comprehensive evaluation of these technologies, and the understanding of their limits and potentials.
Bibliography


