2024

Tuning LSM trees using bayesian optimization

BOSTON UNIVERSITY

GRADUATE SCHOOL OF ARTS AND SCIENCES

Thesis

TUNING LSM TREES USING BAYESIAN OPTIMIZATION

by

ANWESHA SAHA

B.E., Visvesvaraya Technological University, 2019

Submitted in partial fulfillment of the

requirements for the degree of

Master of Science

2024

Approved by

First Reader

Manos Athanassoulis, PhD
Assistant Professor of Computer Science

Second Reader

George Kolios, PhD
Professor of Computer Science

Hold fast to dreams, for if dreams die, life is a broken-winged bird that cannot fly.

Langston Hughes

Acknowledgments

I extend my deepest thanks to Professor Manos Athanassoulis for his invaluable guidance and for offering me the chance to work with him. His support has been the cornerstone of my success at Boston University's Computer Science department.

Special thanks are owed to Andy Hyunh, whose invaluable assistance, support and willingness to share his expertise and time generously were critical to my growth, work and success. I extend my heartfelt gratitude to Professor Vasiliki Kalavri, who introduced me to the realm of research and believed in my potential. Thanks also to Professor Christine Papadakis for guiding me through the thesis requirements with a welcoming spirit, and to Chelsea Houlihan for her motivational support.

I am immensely thankful to Sakshi Sharma and Harsh Sharma, dear friends and fellow researchers, for pursuing their research journey alongside mine. To the entire DisC lab team, who have become like family over the past year, I owe a debt of gratitude. Their camaraderie, support, and encouragement have been a constant source of strength and motivation. My family, especially my parents, have been my unwavering support system, believing in me unconditionally. Their love has been the foundation of all my achievements.

I also thank the Computer Science Department staff and facilities for providing the necessary resources and opportunities for my academic journey. Lastly, my thanks to the many others whose support has been vital to my research success. Your collective encouragement has been a beacon of strength.

Anwesha Saha

Master's Student

Computer Science Department

TUNING LSM TREES USING BAYESIAN OPTIMIZATION

ANWESHA SAHA

ABSTRACT

With the exponential growth of data generation, optimizing databases and its underlying storage structure has emerged as an area of extensive and critical research. This thesis addresses an important aspect of this challenge by introducing an innovative approach to optimize Log Structured Merge Trees (LSM Trees), a state-of-the-art storage structure primarily created for write-heavy database applications without compromising on read operations. It uses Bayesian optimization via the BoTorch library to fine-tune the LSM tree configurations to balance across different workload configurations and address the longstanding challenge of dynamic workload adaptability. A pivotal aspect of this approach is the adaptation of Bayesian optimization to explore the LSM Tree parameter space intelligently by separately handling categorical and continuous variables and enabling a better, more complex examination of the cost surface.

This is done by comprehensively analyzing the LSM Tree structure, its amplification issues, and understanding the overall operational mechanics of this storage structure. The proposed solution is implemented not only on the classic LSM Tree module, but also on hybrid LSM Tree structures and their compaction strategies. The proposed solution approaches this problem by combining the BoTorch framework with an established analytical cost model for evaluation that serves as the objective function for the optimization process. This approach addresses a notable limitation of using the closed-form cost function to predict design decisions which solve a *Lin-*

ear Program instead of a *Linear Integer Program* and treats all values as continuous parameters, which does not accurately reflect the discrete nature of certain design decisions.

Experimental validation on diverse workloads demonstrate the efficiency of the proposed approach and show significant performance gains over traditional tuning methods. This thesis contributes to the growing research on database optimization strategies and help database administrators tune the performance of the LSM Tree structure with minimal manual intervention by providing an incremental step towards self-tuning database management systems, where tuning and optimization can be automated and help in paving the way for better, more reliable storage solutions.

Contents

List of Tables

List of Figures

List of Abbreviations

BO	.....	Bayesian Optimization
DBMS	.....	Database Management System
DDPG	.....	Deep Deterministic Policy Gradient
EI	.....	Expected Improvement
FLSM	.....	Flex Log Structured Merge Tree
GPU	.....	Graphics Processing Unit
I/O Operation	.....	Input/Output operation
LSM	.....	Log Structured Merge Tree
MC Acquisition	.....	Monte-Carlo Acquisition
MLOS	.....	Machine Learning for Systems
PDF	.....	Probability Distribution Function
qEI	.....	q-Expected Improvement
RBF	.....	Radial Basis Function
RL	.....	Reinforcement Learning
SE Kernel	.....	Squared Exponential Kernel
SLA	.....	Service Level Agreement
SMAC	.....	Sequential Model-Based Algorithm Configuration
SSTable	.....	Sorted String Table
T	.....	Size Ratio of a LSM Tree
UCB	.....	Upper Confidence Bound
VM	.....	Virtual Machine
$\mathbb{R}^2$	.....	the Real plane

Chapter 1

Introduction

Databases play an essential role in information storage and retrieval across multiple industries and are specifically designed to enable efficient data organization, access, and manipulation. Their structured nature enables operational efficiency and facilitates processes that require real-time analytics and cloud-based services. These systems serve as the backbone for decision-making and innovation, making them indispensable in the digital era.

Databases can be divided into two main categories - relational databases and non-relational databases. Relational databases [1] like PostgreSQL[2] and MySQL [3] follow the Structured Query Language (SQL) [4] to communicate with the database, whereas non-relational databases like MongoDB [5], Apache Cassandra [6] follow a more flexible data storage approach. Relational databases store information in a tabular structure, with rows and columns that usually represent the attributes and relationships between the data. Non-relational databases, on the other hand maintain a different set of more relaxed properties. This is because these databases usually prioritize availability and partition tolerance over strict consistency. They use a variety of data models like key-value pairs (Redis [7], Amazon DynamoDB [8]), document (MongoDB [5], CouchDB [9]), column-family(Apache Cassandra [6], Google BigTable [10]) and graph formats (Neo4j [11], Amazon Neptune [12]) to store data.

Due to the structured nature of relational databases, they provide robust data integrity and sophisticated query capabilities. Therefore, a fundamental question arises of when to use a non-relational database. Relational databases are difficult to scale especially as data volume increases and query capabilities become more complicated. Non-relational databases, due to their flexible structure can efficiently manage unstructured data and scale across distributed systems. This ability to handle large volumes of diverse data and scalability allow non-relational databases to be popular in big data and real time analytics based applications.

With the rise in amounts of data in the current digital era, the need for scalability and high-performance transaction systems have become crucial. This has led to the development of a new data structure called the **Log-Structured Merge Tree** or **LSM Tree** [13]. This structure was designed to provide low-cost indexing for files with high rates of record inserts and record deletes. The initial goal of the LSM Tree was to maintain efficient real-time indexes for large, rapidly growing datasets like history tables and log files without incurring high I/O costs associated with previous disk-based index structures like B-Trees [14]. These LSM Trees combine the benefits of memory-based and disk-based components and can handle high-throughput write workloads while maintaining high read-performance. This helps in scalability for large, rapidly growing datasets. LSM Trees achieve this performance by buffering writes in memory and periodically merging them into the disk-based storage since writing to a disk is an expensive operation.

While the structure and functioning of the LSM Trees is primarily designed for faster writes, it also uses strategies to enable read operations to be faster. One such strategy is to use a Bloom filter [15, 16] which is a probabilistic data structure that can determine whether an entry is present in the LSM Tree. Another structure used for efficient disk access is the fence pointer which is an in-memory index that stores

the range of the keys in each level which is generally the value of the lowest key at each level. The LSM Tree uses levels to store its data where the level size incrementally grows with levels. Data is written from the lower levels to the upper levels. To counteract the read time, LSM Trees employ a *Levelled* architecture where each level contains only one sorted run. While reading, the structure only needs to perform binary search to find the required key. However, this increases the time needed to write to the LSM Tree. Conversely, a *Tiered* architecture exists which allows multiple runs to accumulate at each level without merging to facilitate fast writes.

The LSM Tree allows the user to choose whether to sort merge data at each level for each run or allow all runs to accumulate without merging. These decisions that determine the speed of each operation that is left to the user are called *tunable parameters*. Configuration of these tunable parameters determine the efficiency of the database. As databases provide more flexibility, more tunable parameters become available. Tuning these design decisions require in-depth knowledge of the kind of operations being performed in the database, as well as a good understanding of the underlying architecture itself. This introduces the challenge of tuning database systems to balance speed, efficiency, and reliability under diverse workloads.

With the development of research in LSM Trees, more complex versions of this structure are now being used. While these allow better control of the system, they introduce further difficulty in tuning the database correctly. To combat these issues, database tuning, especially automated tuning, has been an active field of research. This thesis aims to delve into this field of automated tuning using Bayesian Optimization which is a statistical method that estimates the best parameters to identify optimal solutions with minimum computational expense and limited time.

To use Bayesian Optimization (BO) in this thesis, I utilize the Botorch framework, [17] which is an open-source framework developed by Meta. The advantages of

BoTorch include its flexible and powerful platform for implementing Bayesian Optimization and the ability to use computational capabilities of modern hardware like GPUs that enable scalable and efficient exploration of high dimensional configuration spaces. The capabilities of BoTorch and automated tuning are explored in the next Sections of this thesis.

There has been rapid development in the field of database tuning using both reinforcement learning and BO techniques in recent years. The fundamental problem with reinforcement learning techniques is that it requires an immense amount of data, time and resources to train. The aim is to solve this problem and contribute to the rapidly growing world of autonomous databases by using the statistical bayesian learning model on the Classic LSM Tree structure as well as current state of the art Hybrid LSM Trees. To our knowledge, no other research work has been done on the use of bayesian learning methods to specifically tune LSM Trees on all available models with the help of a closed form learned cost model in accordance with the state of the art BoTorch library. A fundamental issue with using classical solvers to predict design is its treatment of all values as continuous parameters. Therefore results suggested by this model are not practically applicable but provide a more theoretical or *ideal* values that cannot exist or be implemented in real databases. Therefore, the use of this model is limited to evaluation and comparison, but not for optimization. By using the cost model to save time on database runs, maintain consistency across runs and describe the target function and BO to incrementally predict better tuning configurations on the cost surface, the aim of this thesis is to be able to solve the problem of tuning through online learning with limited amount of previously seen data in a short amount of time.

In the following Chapters, this thesis outlines the detailed background of LSM Trees, their compaction strategies and the concepts associated to BO, that is used to

model the solution. Chapter 3 describes the problem definition using these concepts and formulates a cohesive expression to demonstrate the exact problem statement. Chapter 4, addresses the exact methods and steps followed to formulate the solution, and justify the design choices made. A detailed description of the results are presented in Chapter 5. Chapter 6 presents the key takeaways from these results. Chapter 7 discusses related work, focusing on data systems tuning using Machine Learning and Bayesian Optimization as well as work in this field using Reinforcement Learning (RL), and Chapter 8 concludes by discerning future work in this line of research.

Chapter 2

Background

This Section discusses the concepts used and applied through this thesis. It introduces the basic structure of the LSM Tree, discusses the amplification issues faced in storage structures and its types. It then discusses the varied models of the LSM Trees - Classic models and Hybrid models that are used in this thesis. It next details the Bayesian Optimization process and the formulae and concepts used and the choices available for the selection of the different components of this optimization process.

2.1 Log Structured Merge Trees

The *Log-Structured Merge Tree* (LSM Tree) is an advanced disk-based key-value data structure designed for efficient handling of both rapid data lookups and insertions. This data structure was initially created to address the challenge of the trade-off between optimizing for read operations and write operations. One of the key disadvantages faced by traditional database systems was the cost of writing to the disk after every write operation as in the case of B-trees. B-trees were efficient at providing fast data retrieval but did not do well with high-throughput write workloads. Every write operation in a B-tree requires writing to the disk. This causes a significant amount of overhead and performance bottlenecks.

The LSM tree achieves its efficiency by avoiding accessing and writing to disk after every write operation. Instead, it accumulates these writes in an in-memory buffer

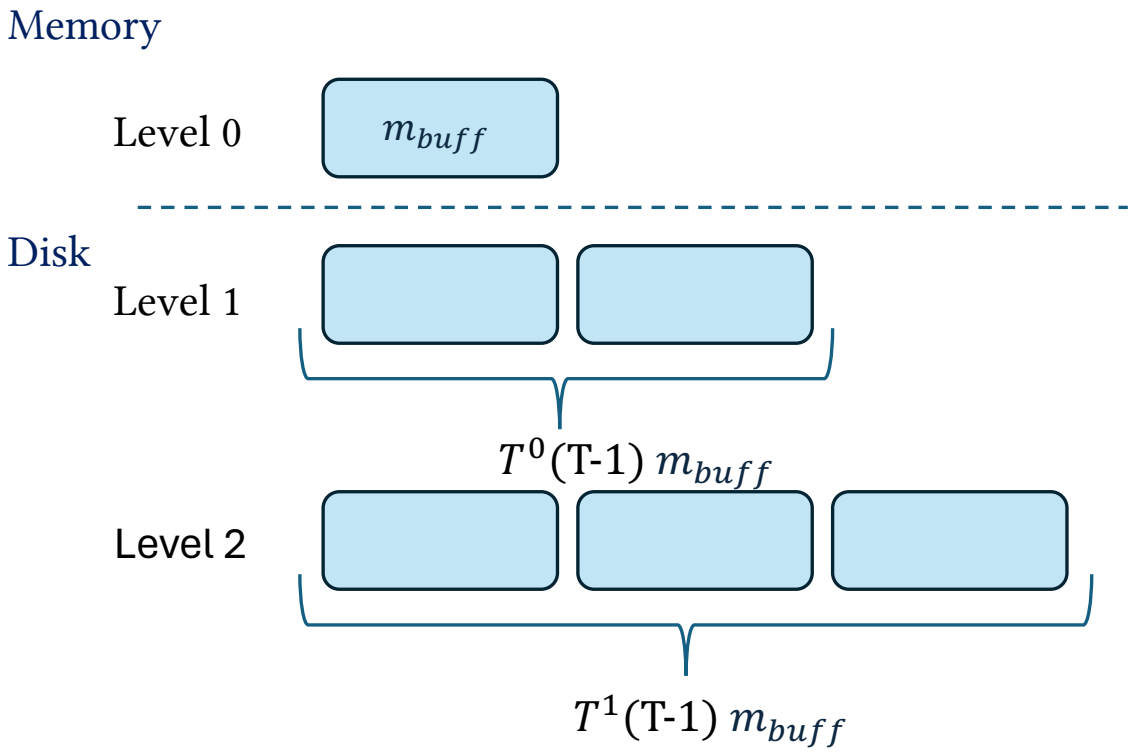

Figure 2·1: An LSM Tree with its memory and disk components. The memory component consists of an in-memory buffer at Level 0 with capacity m_{buff}. For the disk component, Level 1 and Level 2 represent on-disk data storage with capacities defined by the parameters $T^0(T-1)m_{\text{buff}}$ and $T^1(T-1)m_{\text{buff}}$ respectively where T denotes size ratio. Each level accumulates $T-1$ full m_{buff} (runs) before triggering a compaction.

called a MemTable and then periodically merges the contents of the buffer into a disk in batches. This data is stored in sorted order as Sorted String Tables or SSTables. This reduces the cost of expensive disk I/O that would be caused due to individual writes. When this in-memory buffer is filled up, the entire buffer is flushed to disk as a sequential write. This flush creates a *level* on the disk. We *refer to the in-memory buffer as level 0* and the first buffer in the disk to be created as level 1. The first level, or level 1, represents the lowest tier of the hierarchical LSM structure on disk. Data propagates from one level to the next higher level through periodic merge operations called compactions. For example, when level 1 is full, it gets sorted and merged with the data from level 2 via compaction.

The parameter that determines the size of each level, relative to the previous level is called the *size ratio* denoted by T. This tunable parameter, specified by the user or system administrator, plays an important role in the structural and operational efficiency of the tree. The LSM tree is structured such that each level increases exponentially in size compared to the previous level. It is important to determine the appropriate value of T to balance between the efficiency of data insertion and retrieval activities and the computational and storage costs associated with the compaction process.

These unique characteristics of the LSM tree, including its hierarchical structure, in-memory buffering, and compaction mechanisms, have made it a popular choice as a background data structure in multiple storage systems, specifically the database systems that deal with high-throughput write workloads or ingest large volumes of data over extended periods of time.

2.1.1 Amplification in LSM Trees

There are three kinds of amplifications associated with most storage structures including LSM Trees. Amplification is associated to disk access, which is one of the most expensive operations in a database. The aim of an efficient storage structure is to reduce disk access as much as possible. This is not simple, since it usually involves a trade-off when trying to reduce amplification of a particular kind - this means that reducing one type of amplification might increase the amplification elsewhere. The ideal configuration depends on the kind of operations the user performs on the database.

Read Amplification is defined as the number of the times the disk has to be accessed and read for one read query requested by the user. This value is different for a point read query and a range query. *Space Amplification* is the ratio of the space occupied in the disk to the actual size of data to be written. This means that if a user writes 10 MB of data into the database and this causes the database to occupy 100 MB of the disk space, then the space amplification of the structure is 10. *Write Amplification* is the amount or bytes of data written to disk when the user wants to insert one byte of data into the database.

2.1.2 Optimizing in LSM Trees

LSM Trees were originally designed to be optimized for write operations since other structures like Binary Trees or B-Trees were already read-optimized. To better the performance of reads in LSM Trees there are several optimizations that have been added in recent databases that use this storage structure. Typically in a *Read* operation, the LSM Tree would have to search every level, till the first time a matching key was found. This would mean a lot of redundant disk accesses especially if the key was at a very deep level which occurs if the LSM Tree is large or if the key does not exist

in the LSM Tree at all. To counteract this problem, LSM Trees use a combination of fence pointers and Bloom filters [18, 19]. The size of a fence pointer is usually insignificant compared to actual data and therefore can be stored in-memory. This is because the fence pointer just has information about the smallest and largest keys in every disk page [20]. Usually the fence pointer just stores the smallest key for that level which is enough information to denote the entire range. Since the pointer is present within the memory, searching fence pointers does not involve any costly overhead. So, for a point read query, once the level where the data is located is found, the fence pointer can be used to access directly the run which contains the available data in 1 I/O. For a range query, it starts the scan from this block.

Usually, by using a fence pointer, the user can access the relevant key range at each run with only one storage access [20]. However, using just a fence pointer might not always be sufficient. Some read queries may still require going through all the runs in the LSM Tree [21, 18]. The Bloom filters can be used here to reduce this I/O cost. A Bloom filter is a space-efficient probabilistic data structure that returns a binary result for whether an entry is present in a particular level within the LSM Tree structure [15, 16]. Current databases that use LSM store these Bloom filters in-memory [22]. Every run has its own Bloom filter. If the Bloom filter returns *True* for that run, it indicates that the data might or might not be present in that run. However, if the Bloom filter returns *False*, it indicates that there is no possibility of the entry being present in that run. This prevents the LSM from probing that run for the entry. The greater the space allocated to the Bloom filter, the more accurate are the results. We see a measure of how the memory determines the accuracy of the Bloom filter in Chapter 4.

2.1.3 Operations in the LSM Tree

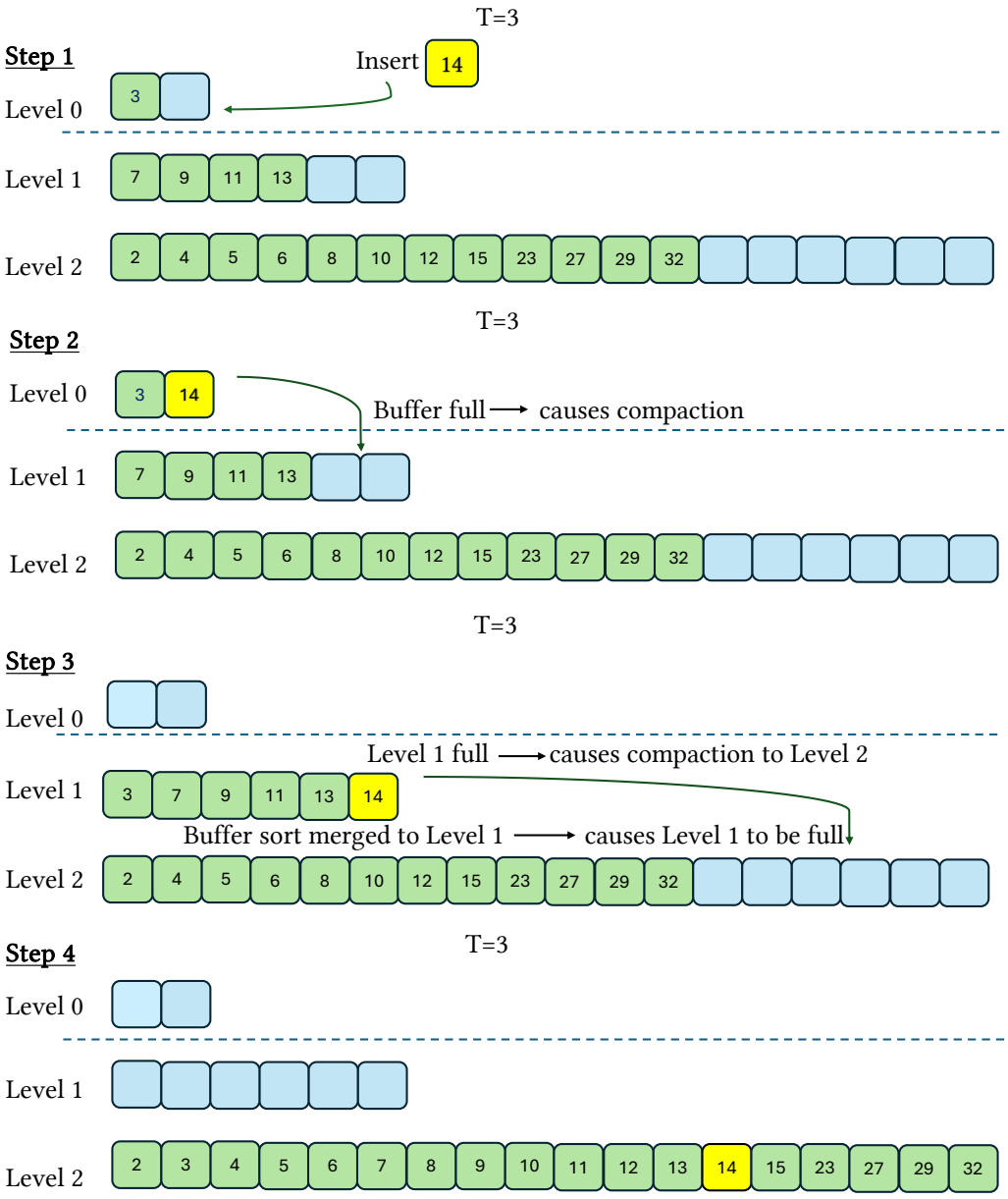

Figure 2·2: LSM Tree following the Leveled Compaction Policy: Insertion causes the Buffer (Level 0) and the next Level 1 to perform compactions and get merge sorted with the next levels. Since the LSM Tree follows the Leveled Compaction Strategy, each level has only 1 run. The T represents size ratio and has a value of 3. Therefore Level 1 is 3 times as large as Level 0 and Level 2 is 3 times larger than Level 1

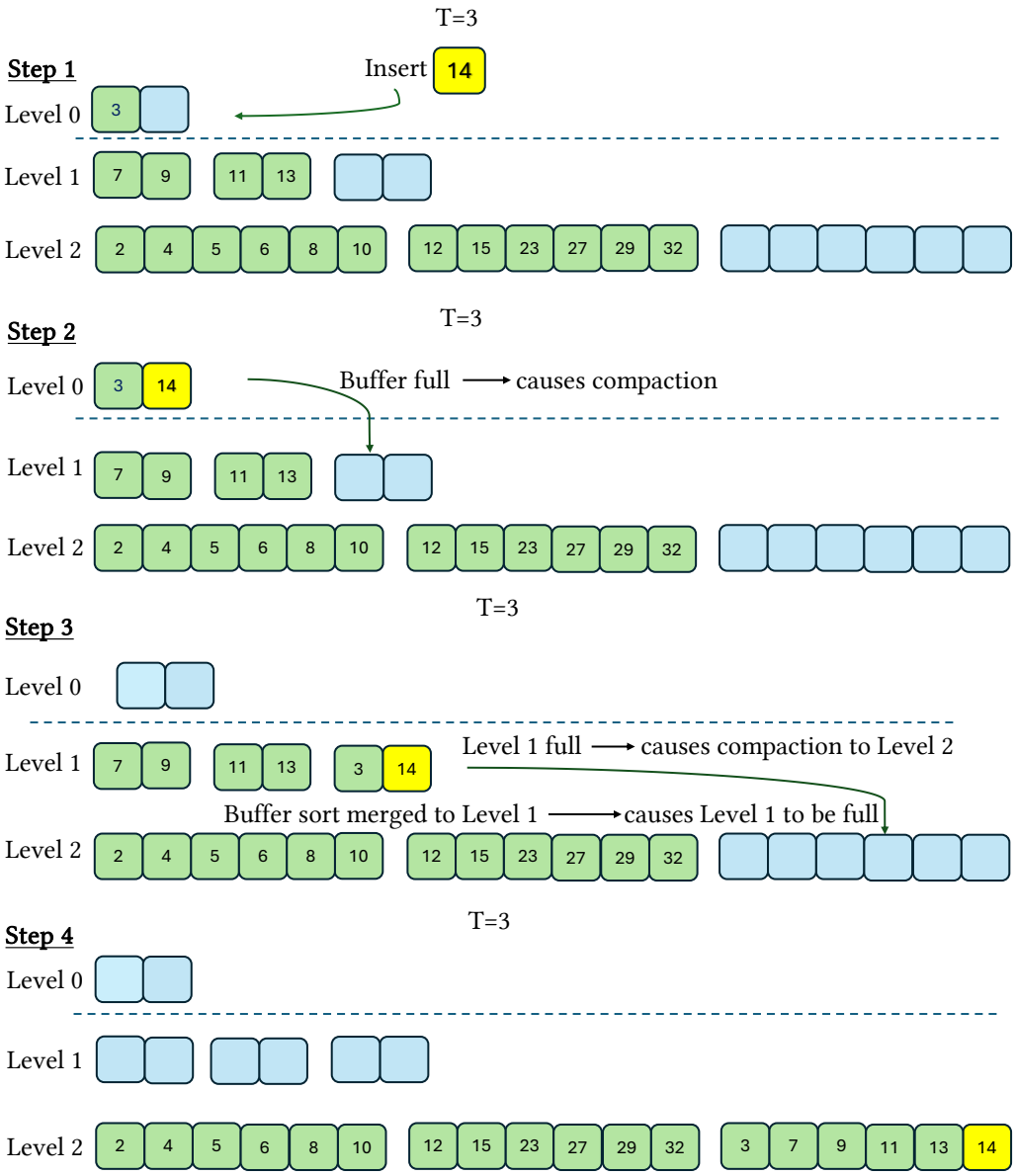

Figure 2·3: LSM Tree following the Tiered Compaction Policy: Insertion causes the Buffer (Level 0) and the next Level 1 to perform compactions and get merge sorted with the next levels. The T represents size ratio and has a value of 3. Since the LSM Tree follows the Tiered Compaction Strategy, each level has *T-1* runs. When it reaches the *T-th* run a compaction is triggered. The runs are individually sorted but two runs in the same level may contain duplicates and are not always sorted with respect to the entire level.

Writing to the LSM Tree

Writes in LSM Trees are performed out of place. When a user wants to write data, it is first inserted into the buffer as a key-value pair. When the buffer is full, this entire batch of data is written into disk together by a method known as *flushing*. The data traverses each layer via flushing by continuing this process till it reaches the last level. Each time a level is filled till capacity the data gets sort merged with the data from the next level.

Reading from LSM Trees

A read in a LSM Tree will always fetch the most recent version of the data that matches the key queried by the user. An LSM Tree might have multiple entries for the same key which is usually taken care of through compactions. But the most recent data will always be in the lower levels due to the structure and design of the LSM Tree. When a read query is made, the LSM Tree first checks its buffer for the presence of the item that is to be searched for. If the item is found, it indicates that this is the most recent copy and is returned to the user, not requiring any disk access. In case the entry is not found in the buffer, the LSM Tree checks the Bloom filter for each level to find the exact level where the entry might be located. If the Bloom filter returns *True*, it then verifies using the fence pointers to get the exact run location of the entry. However, in case the entry is not present in that run and the Bloom filter returned a false positive, the LSM Tree needlessly incurs an extra disk I/O. The worst case occurs if the Bloom filter returns *True* for all runs and the entry is not present in any run or if the entry is present in the last run. Then the LSM Tree will have to query all runs before it can return a definitive result to the user. The LSM Tree search occurs from the lower levels to the higher levels and since by virtue of the LSM Tree's property to store more recent data at lower levels, the data returned

to user is guaranteed to be the most recent.

Range Reads

A range read refers to a query defined by a start and end point that aims to retrieve all data between these two points. Range reads in LSM Trees work similar to normal read queries and return the most recent version of the data matching a key. LSM Trees are not particularly optimized for range queries. Bloom filters are not very effective for range queries either. This is because data in a given range can be scattered in multiple files over multiple levels. The LSM Tree has to seek and sort merge all these entries before it can return the result to the user. An iterator is then assigned to each run and multiple runs are searched in parallel to make the entire process faster. Recent research has improved efficiency of range queries in LSM Trees [23] especially by using filters to make the process faster [24, 25, 26].

Deleting in LSM Trees

Deletes in LSM Trees happen out-of-place. Instead of directly deleting an entry, a special type of key-value entry called a tombstone is used to mark an entry to be deleted. If the item to be deleted is present in the in-memory buffer, the item is deleted upon insertion of the tombstone. However, if the item to be deleted is within the tree present in the disk, the item will eventually be deleted when the tombstone is flushed and compacted with the item.

2.2 The Classic Log Structured Merge Trees

The Classic Log Structured Merge Tree offers two designs dependent on the compaction strategy used - a read optimized strategy called **Leveling** and a write-optimized strategy called **Tiering** [27].

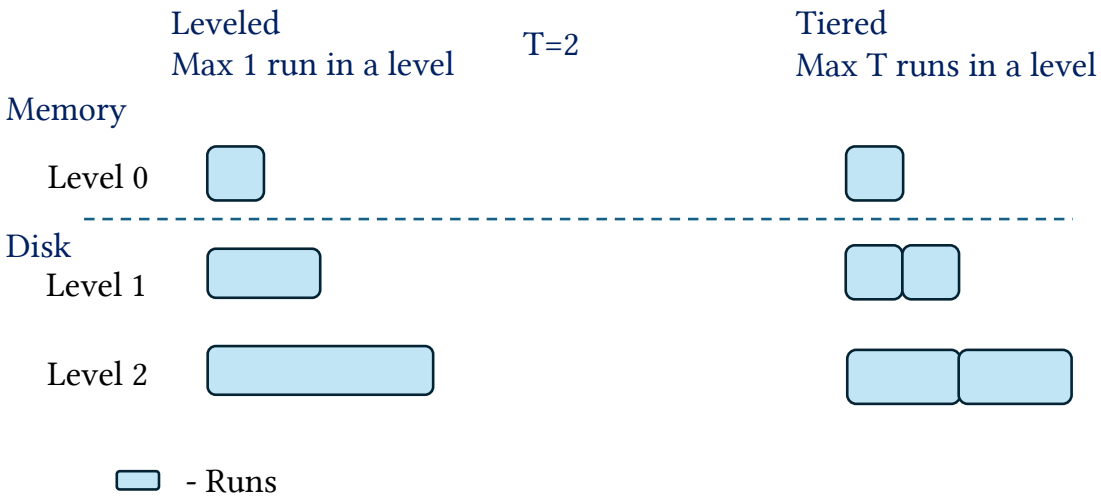

Figure 2·4: The LSM Tree with Tiered architecture on the right and the Leveled architecture on the left. The Leveled LSM Tree always has one sorted run at each level, whereas the Tiered LSM Tree always has 2 sorted runs at each level since the size ratio, represented as T, is 2.

2.2.1 Leveling

In this style of compaction, every run at a specific level is always merged and sorted while storing. Whenever a compaction is triggered, the level above the current level is greedily sort merged with the current level. This was the initial compaction strategy used in LSM Trees [13]. For a LSM tree where compactions follow Leveling policy, there is always a single merged and sorted run at each level. This helps in lookups because the Tree can use *binary search* at each level to find the key.

2.2.2 Tiering

The *Tiered* style of compaction [6] is primarily favored for write-heavy operations. In this style of compaction, each level can have multiple runs that are accumulated over time instead of being greedily sort merged after every run. This prevents the overhead of compactions after each run. The tiered design also allows duplicate keys to exist in a level since the runs are not merged. For a LSM tree where compactions

follow Tiering strategy as their compaction policy, there are T runs at each level where T signifies the size ratio of the LSM.

Comparison of Compaction Policies

Selection of compaction policy depends on the user and the kind of operations that the LSM Structure is being used for. The Leveling compaction policy allows each level of the LSM Tree to hold at maximum only one run per level. The entire level is sorted and stored for faster access. While this makes the reads much faster in the LSM Tree, it also increases the time taken to write because every new entry has to be sort merged to that level. This is explained in the Diagram 2·2.

In this Diagram, when the key to be written is inserted into the buffer, which is an in-memory storage, the buffer reaches capacity. This triggers the buffer to flush its contents into Level 1 which in turn triggers a compaction. In the case of leveling, it has to sort the data before it can merge with the contents of Level 1 which introduces a lot of overhead especially if the level is bigger in size. In the example provided in Figure 2·3 and Figure 2·2, the compactions are only demonstrated till Level 2, but since this Level also becomes full, the compaction chain is actually carried on to further levels.

In Tiering, this problem is resolved by choosing to not sort merge every run as it comes in. Instead the contents of the run from the previous level are stored as is till the level reaches capacity. Once the level reaches T runs, a compaction is triggered. The contents of the runs from the current level are then sort merged and put into the next level as explained in the Diagram 2·3. The problem with this strategy, however is that because all the runs in a level are not merged, reads will take more time since the entire data is no longer sorted for the whole level. The other issue is that, we might have duplicates in the same level within different runs which are not eliminated

since the runs are not merged together. This means the storage structure takes up more space than we envision leading to greater write amplifications.

In general, it can be argued, that both strategies have their own advantages and disadvantages. This leads to recent research exploring the space of hybrid compaction policies that merge Leveling and Tiering so as to be able to accommodate both reads and writes at a faster speed based on a specific workload [27]. To implement this procedure, data is organized into smaller files and compactions are performed on the *files* and not on the entire *level*. If *Level i* has to be compacted with *Level (i+1)* and there are files in both levels with overlapping key-range then the file or files which have this overlap are selected for the compaction.

Recent studies have further introduced hybrid LSM Tree structures which provide more flexibility by allowing different policies at different levels and some even allow the user to set a different number of runs for each level. It is generally agreed, that Leveling works best for a level with a very large quantity of data, as it is faster to search for items in this design and also prevents high numbers of duplicates. This idea has led to the development of the 1-leveling layout where the first level or Level 1 follows the Tiering policy and the others follow the Leveling policy or the Lazy Leveled architecture which follows the Tiering policy for all levels except the last one. These hybrid structures are discussed in detail in the next Section.

2.3 Hybrid Log Structured Merge Trees

Hybrid Log Structured Merge Trees refer to the LSM Trees that employ a mix of the Tiered and the Leveled compaction policy. These design decisions are made to allow more flexibility to the structure and allows the user more choice in terms of *tunable parameters*. This flexibility introduces more complexity to the model in terms of added hyper parameters and design decisions.

2.3.1 The Lazy Levelled Log Structured Merge Tree

The Lazy Levelled Log Structured Merge Tree [28] proposes a LSM Tree structure where only the last level follows the *Levelling* strategy for compaction whereas all other levels follow the *Tiering* compaction policy.

As stated in the paper, merge operations due to compactions in the lower levels are costly but do not benefit point read query or range query latency equivalently. Space amplification at these levels also are not significant enough to support frequent compactions. These observations led to the creation of the *Lazy Levelled LSM Tree* in order to address the inefficiencies associated with traditional compaction processes. It aims to minimize the frequency of merge operations in the lower levels since compactions at these levels are less impactful on query latency and do not contribute to space amplification.

Merging in an LSM Tree is expensive. The LSM Tree is designed such that the upper levels are exponentially smaller in size than the last level, therefore searching or querying at the last level would take equivalently a large amount of time owing to it's size. For this reason, it can be logically argued that the last level should be sorted such that time to query can be much smaller and space amplification is not severe. By reducing compaction frequency in the lower levels and ensuring the last level is efficiently organized for querying, the Lazy Levelled LSM maintains a balance between query performance and minimizing space and time overhead which leads to overall efficiency of the LSM Tree structure.

2.3.2 The Fluid Log Structured Merge Tree

The Fluid LSM Tree [28] builds upon the idea of the Lazy Levelled LSM Tree. It introduces more flexibility by allowing customizable compaction frequencies across different levels of the tree while retaining the fundamental behavior of the Lazy Lev-

elled LSM Tree.

This is done by introducing two new parameters - K and Z where K represents the maximum number of runs at every level except the last level and Z represents the maximum number of runs at the last level. This allows a user to finely tune the LSM Tree as per specific workload requirements while maintaining the ability to transition to the classic compaction policies like Tiering and Leveling by adjusting the K and Z values. This implies that the Fluid LSM Tree possesses the benefits of reduced merge operation costs from the Lazy Leveled LSM as well as finer control over compaction process which leads to better performance in many scenarios. Mathematically, if size ratio of the LSM Tree is denoted by T then the size of each run at all levels except the last level is given by the ratio T/K. When this size is reached, a new run is started at that level till there are K number of runs. When the level is entirely full, all the runs are merged and flushed to the next higher level. Similarly for the last level, each run is of size T/Z and there are maximum Z number of runs at the last level. Table 2.1 details how this model can be used to represent the other models by varying the values of K and Z.

Table 2.1: LSM Tree Policies Based on K and Z Settings

K	Z	**Resulting Policy**
1	1	Leveling
$T-1$	$T-1$	Tiering
$T-1$	1	Lazy Leveling

2.3.3 The K-Log Structured Merge Tree

The **K-LSM** [29] builds upon the Fluid LSM Tree structure [28]. In this design, each level in the LSM Tree structure has its own *K-value*. So for level i, the K-value for that level is determined by K_i. If there are a total of L-levels then there are L

number of K values, each determined by the user from K_1 to K_L. Therefore each run in the K-LSM tree is of size $\frac{T_i-1}{K}$. It is apparent that while this LSM provides a lot of flexibility to the user, it also introduces a lot more tunable parameters that have to be accurately determined by the user of the system for optimal performance of the LSM Tree.

A special case of the K-LSM based design can be generalized where all the K_i values are the same, that is if K_1=K_2=...=K_L=Q. We refer to this model as the *Q-LSM* model where Q represents the integer value given to K. This representation of all other models using the K-LSM model is shown in Table 2.2

Table 2.2: LSM Tree Policies obtained by Modifying K-values

LSM layout	Setting
Fluid LSM	$K_i = K, \forall i \neq L, K_L = Z$
Lazy Leveling	$K_L = 1, K_i = T-1, \forall i \neq L$
1-Leveling	$K_1 = T-1, K_i = 1, \forall i \neq 1$
Tiering	$K_i = T-1, \forall i$
Leveling	$K_i = 1, \forall i$
Q-LSM	$K_i = Q, 1 \leq Q \leq T-1, \forall i$

2.3.4 The Flex LSM Tree

The Flex LSM Tree model [30] was designed specifically for dynamic workloads. This model is built upon the Lazy Levelled LSM Tree and Fluid LSM Tree structure. It allows dynamic adjustment to the compaction policy which is required for performing performance optimization with varying workloads. This model uses similar notation to the Fluid LSM Model. The Flex LSM model is capable of adapting the frequency of merge operations at both the last level and every other level independently by

using two parameters - K and Z where K determines the runs at all levels except the last level and Z gives the maximum number of runs at the last level. However, in this model, the value of K and Z can change mid-run. It allows a portion of the run to have a different K value and the remaining portion to have a new value - K'. To accommodate the new K' runs, the runs at that level will have different lengths. The size of each run at every level before the last is determined by the ratio T/K where T is the size ratio. The size of the run at a particular level is determined by using reinforcement learning.

It is possible that the size of run changes even though there is some data already present at that level. To allow for this efficient transition between compaction policies, the F-LSM model introduces the concept of active run which is the current run to which entries are getting inserted into. This run, when full, is sealed and a new active run is created. If the compaction policy is changed, then it only affects the active run and not the already sealed runs. We consider that the current number of maximum runs at a level is denoted by K and the new number of runs due to change in policy is denoted by K'. If the new policy *decreases* the maximum runs at that level from K to K' then the active run's capacity is increased and the existing sealed runs are left untouched. If the new policy *increases* the maximum runs at that level from K to K' then the active run's capacity is decreased. If the active run in this case is already bigger than the predicted run size should be then the active run is immediately sealed and a new active run is created. Sealed runs remain till their level is compacted into the next level and gradually get merged into the structure. This design allows the user to immediately apply a new policy to the active run without reorganization of data and avoids expensive rebuilding from lazy transitions.

2.4 Bayesian Optimization

Bayesian optimization is a sequential design strategy used for optimizing complicated black-box functions, characterized by absence of any assumed functional form. It is used in areas where the functions are expensive to evaluate. Bayesian Optimization does not require any prior knowledge or assumption about the structure of the function and instead relies on a probabilistic model to search for the optimum.

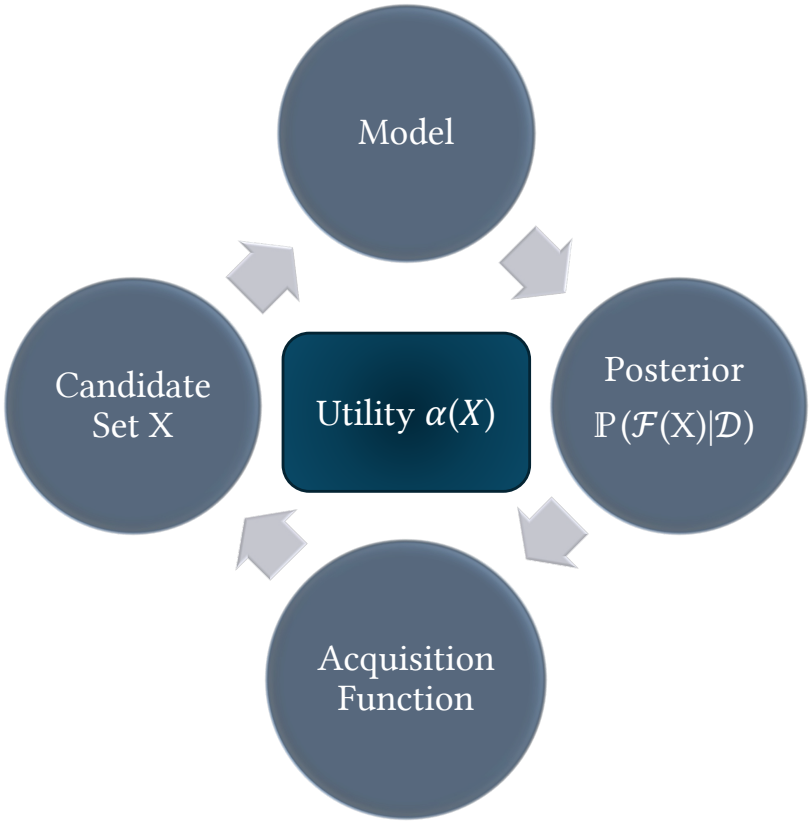

Figure 2·5: The iterative nature of the Bayesian Optimization process is represented in this diagram. The Bayesian process can be generally divided into these 5 main processes.

Diagram 2·5 represents the Bayesian process as a cycle that is indicative of its repetitive nature. The optimization starts from *Candidate Set X* which includes possible solutions to the optimization problem. It also includes the information gained

from the previous evaluations which serve as the new *training data* for the model. The *Model* represents the surrogate model for approximating the objective function and is updated using information gained from evaluation of candidate set. The *Posterior* reflects the updated state of the model after incorporating information from evaluation of *Candidate Set* and represents the updated beliefs about function's behavior. The *Acquisition Function* uses the *Posterior* to calculate the *Utility* for each candidate and guides the decision about where to sample next by weighing exploration against exploitation. *Utility* $\alpha(X)$ is the Utility score derived from the Acquisition function that quantifies the expected value of sampling each candidate.

2.4.1 Foundations of Bayesian Optimization

This Section describes the fundamental principles underlying the Bayesian optimization process. There are several statistical concepts that are used to design the optimization process. We work on providing and justifying the use of these statistical models to provide insight into how these are used and unified to design this process.

In the first subsection, we discuss the progression and the distribution's versatility, from simple univariate analysis in the Gaussian Distribution to the Gaussian Process which is used for sophisticated applications in machine learning and beyond.

The Gaussian Distribution

The **Gaussian distribution** can be described using it's probability density function (PDF) given by:

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2} \tag{2.1}$$

where μ and σ represent the mean and standard deviation of the distribution, respectively. Gaussian distribution is characterized by its bell-shaped curve, where

(μ) determines the center of the curve, and (σ) controls the width of the bell. This distribution is extensively used for representing the distribution of real-valued random variables with unknown distributions in both natural and social sciences.

Extension of the Gaussian Distribution to Multivariate Normal Distribution

While the univariate normal distribution describes random variables with a single dimension, real-world phenomena often involve multiple interrelated variables, necessitating a multivariate approach. The multivariate normal distribution extends the univariate distribution to represent these complex phenomena. The probability distribution function for such a distribution can be represented as:

$$f(\mathbf{x}; \mu, \mathbf{\Sigma}) = \frac{1}{\sqrt{(2\pi)^k |\mathbf{\Sigma}|}} e^{-\frac{1}{2}(\mathbf{x}-\mu)' \mathbf{\Sigma}^{-1} (\mathbf{x}-\mu)} \tag{2.2}$$

Here, $\mathbf{x}$ denotes a vector of k variables, μ is the mean vector, and $\mathbf{\Sigma}$ is the covariance matrix. This multivariate approach is important for analyzing datasets where variables are interconnected.

Gaussian Processes

Gaussian processes generalize the multivariate normal distribution to infinite dimensions, enabling the modeling of continuous functions.

Gaussian processes serve as a robust tool for various tasks in machine learning, like regression, classification, and optimization, specifically when the functional form is not explicitly known. A Gaussian Process (GP) can therefore be defined as a probabilistic model for functions, characterized by its mean function $m(x)$ and covariance function $k(x, x')$, where x and x' are points in the input space. The GP provides a distribution over possible functions that could fit the data, and allows uncertainty in

predictions. The GP is defined as:

$$f(x) \sim \mathcal{GP}(m(x), k(x, x')) \tag{2.3}$$

2.4.2 Components of the Bayesian Optimization Model

The *surrogate model* is used to understand the behavior of the unknown function f on untested points. The final goal is to optimize f with a minimal number of evaluations. The *surrogate model* is important for it's predictive capability and its ability to express the uncertainty of the predictions as a posterior distribution over the function values $f(x)$ at new points x.

The standard surrogate model usually used for f is the *Gaussian Process (GP)*, which is characterized by its ability to describe the posterior distribution for any finite set of points as a multivariate normal distribution. A GP can be defined using a mean function $\mu(x)$ and a covariance kernel $k(x, x')$, enabling the derivation of both a mean vector $(\mu(x_0), \ldots, \mu(x_k))$ and a covariance matrix Σ, where $\Sigma_{ij} = k(x_i, x_j)$, for any chosen set of points $(x_1, \ldots, x_k)$. Using a GP as the surrogate model implies the assumption that the vector $(f(x_1), \ldots, f(x_k))$ is multivariate normal, with its mean vector and covariance matrix determined by $\mu(x)$ and $k(x, x')$, respectively.

Posteriors are calculated using the dataset on which the model has been trained and encapsulate the model's confidence or *belief* about the potential values of a function at a specific point or across a range of points. The posterior distribution of a model represents it's updated predictions after incorporating the observed data up to that point. In the context of a Gaussian Process (GP) model, the posterior is explicitly defined as a multivariate Gaussian distribution. This distribution is completely characterized by two key parameters: its mean and its covariance matrix. These parameters are updated whenever new data is observed, thus refining the model's

predictions and uncertainty estimations about the function values.

Acquisition Functions guide the search process towards the areas in the parameter space that are most likely to provide the user with the expected outcome. These functions are used to evaluate the potential benefit of *candidate points* that will help determine the next points to evaluate in the optimization process. Acquisition functions are designed to balance the search space between *exploration* which means trying points where the uncertainty is high or trying points that are further away from the evaluated points and *exploitation* which refers to using the information already gathered from previous evaluation that are close to the optimum and the uncertainty in these points are low. Too much exploration by the acquisition function can cause more number of evaluations that do not result in a fruitful outcome especially if the answer lies at a point close to previously evaluated points and this leads to a waste of resources. Conversely, too much exploitation might lead to local optima as opposed to the actual solution in an area which has not been explored by the acquisition function. This trade-off can be balanced by quantifying the expected utility of sampling at a point in the input space and considering both aspects - the potential for discovery as well as the potential for optimization in the local area.

A popular choice of acquisition function is *Expected Improvement* as it balances the trade-off and performs well in most cases. It can be modeled by the equation [31]:

$$EI(x) = \sigma(x)\left(\gamma(x)\Phi(\gamma(x)) + \phi(\gamma(x))\right) \tag{2.4}$$

where:

$$\gamma(x) = \frac{f(x_{\text{best}}) - \mu(x)}{\sigma(x)} \tag{2.5}$$

$\mu(x)$ and $\sigma(x)$ are the predictive mean and standard deviation of the GP at point

x, and Φ and ϕ are the cumulative distribution function and probability density function of the standard normal distribution, respectively. The Expected Improvement (EI) acquisition function calculates the expected improvement over the current best observation x_{best}, taking into account both the mean and uncertainty (standard deviation) of the predictions. It balances exploration of new areas with the exploitation of known good areas.

Another choice of acquisition function that has proven to work well is the Upper Confidence Bound or UCB introduced for the *Multi-Arm Bandit Problem* [32, 33]. The Upper Confidence Bound (UCB), utilized in Bayesian optimization, balances exploration and exploitation by maximizing the function:

$$UCB(x) = \mu(x) + \sqrt{\beta}\sigma(x)$$

where $\mu(x)$ and $\sigma(x)$ denote the mean and standard deviation of the predictions at point x respectively, and β is a hyper parameter influencing the level of exploration. Users might prefer this acquisition function, as it allows for finer control of how much exploration is required for their specific use case.

EI and UCB fall in the general category of Analytic Acquisition functions. This means that they have a closed-form expressions and can be evaluated directly, without the need for numerical approximation or simulation-based estimation. Contrarily, Monte Carlo acquisition functions estimate acquisition values through simulations, ideal for handling complex scenarios like parallel evaluations. An example of such an acquisition function is the q-Expected Improvement (qEI), which extends the Expected Improvement framework for batch optimization and can be defined as:

$$qEI(x_1, \ldots, x_q) = \mathbb{E}\left[\max(f(x_1), \ldots, f(x_q)) - f(x^+)\right]$$

qEI is advantageous in environments allowing parallel computations, facilitating faster convergence by evaluating multiple points concurrently.

Covariance Functions are also known as kernels and are the main part of driving the Gaussian Process. They assume the behavior of the underlying function that we want to model. It stems from the idea that points in an input space that are close to each other are likely to yield similar target values than points which are far apart. The covariance function defines the covariance between two random variables as a function of their input values. For a Gaussian Process, the covariance function expresses the assumptions about the function that we want to learn from the data.

A valid covariance function $k(x, x')$ must be positive semi-definite for any finite set of points. This means for any number n of points $\{x_1, x_2, \ldots, x_n\}$, the $n \times n$ covariance matrix $\mathbf{K}$, with elements $K_{ij} = k(x_i, x_j)$, must be positive semi-definite. Some of the most popular kernels are defined below.

The **Squared Exponential (SE)** kernel, also known as Radial Basis Function (RBF), is a popular choice due to its smoothness and the property that functions sampled from the GP with this kernel are infinitely differentiable. It is defined as:

$$k_{SE}(x, x') = \sigma^2 \exp\left(-\frac{\|x - x'\|^2}{2l^2}\right) \tag{2.6}$$

where σ^2 is the variance and l is the characteristic length-scale.

The **Matern kernel** is one of the most frequently used kernels. The Matérn kernel is a generalization of the *Squared Exponential Kernel*, providing additional flexibility that allows the smoothness of the kernel to be controlled.

The Matérn kernel is parameterized by a positive smoothness parameter ν and a positive scale parameter ρ (sometimes called the length-scale). The kernel is defined

as follows:

$$k_{\text{Matérn}}(x, y) = \frac{2^{1-\nu}}{\Gamma(\nu)} \left(\sqrt{2\nu} \frac{\|x - y\|}{\rho} \right)^{\nu} K_{\nu} \left(\sqrt{2\nu} \frac{\|x - y\|}{\rho} \right)$$

where x and y are two input vectors, $\|x - y\|$ is the Euclidean distance between x and y, ν is the smoothness parameter, ρ is the scale parameter, K_{ν} is a modified Bessel function of the second kind, and Γ is the gamma function.

The smoothness parameter ν controls the smoothness of the function that the kernel represents. Higher values of ν lead to smoother functions. In the limit as ν approaches infinity, the Matérn kernel converges to the squared exponential kernel, which corresponds to an infinitely differentiable Gaussian process. For certain values of ν, especially when ν is half an integer, the kernel has a closed form because the Bessel function can be simplified.

The parameter ν can take specific values such as $\nu = \frac{1}{2}$, which corresponds to the exponential kernel, $\nu = \frac{3}{2}$, which results in a kernel function that is once differentiable, and $\nu = \frac{5}{2}$, which yields a kernel function that is twice differentiable.

This thesis uses the ideas detailed here to formulate the Problem Statement in the next Chapter 3 and the proposed solution in Chapter 4.

Chapter 3

Problem Definition

This section formally presents the problem, including a detailed description of the features used in its formulation. The problem relies on the cost model used in the *Endure* [34]. To maintain consistency, similar notation will be used for the problem definition. The problem is defined through an examination of three interrelated components: *Workload, System and Design.* The following Table is provided as easy reference for all the parameters used for the problem description.

3.1 Problem Components

The *Workload* component, denoted as **W** is a vector that represents the percentage of each operation type. It is constructed from *Empty Lookups* z_0, *Non-Empty Lookups* z_1, *Range Lookups* q, and *Writes* w. These represent the full range of allowed operations, and satisfies the equation

$$z_0 + z_1 + q + w = 100$$

.

Table 3.1: Abbreviations used in the problem definition related to the *Workload W*

Term	Description
z_0	Percentage of empty reads
z_1	Percentage of non-empty reads
q	Percentage of range queries
w	Percentage of write queries

The *System* parameters, denoted as **S**, represent the constraints of the problem and reflect the limitations or the maximal utilizable resources within the given system. These immutable parameters include E, the size of an entry; s, the selectivity of range queries; B, the number of entries per page; N, the total number of elements in the tree; H, the budget for bits per element; and φ, the read/write asymmetry coefficient, which determines how much more expensive writes are than reads in the physical storage device. These parameters provide a fixed boundary within which the optimization operates.

Table 3.2: Abbreviations used in the problem definition related to the *System S*

Term	Description
E	Size of a key-value entry
s	Selectivity of range queries
B	Number of entries per page
N	Total elements in LSM Tree
H	Bits per element budget
φ	Read/Write asymmetry coefficient
m_{filt}	Memory allocated to the bloom filter
m_{buff}	Memory allocated to LSM Tree buffer
m_{buff}	Total memory $= m_{filt} + m_{buff}$

Design parameters represented by **D** are the adjustable parts of the system that this thesis aims to optimize. These parameters, which include policy choices and size ratios, are based on previous studies on LSM Trees and have been proven to have the maximum amount of impact on the performance of the system[21, 19]. Tuning these parameters help in balancing the LSM Tree structure to fit the kind of workload run on the system as well as account for the system constraint placed.

Classic Log Structured Merge Tree is characterized by the choice of compaction policy — Leveling or Tiering as well as the *bits allocated per element* h and the *size ratio*-T of the LSM tree. The *Lazy Leveled Log Structured Merge Tree* model introduces two additional parameters, K and Z, which signify the maximum number of runs at all levels except the last level and at the last level, respectively. These, along with the *bits per element* and the *size ratio*, define the design parameters for this LSM tree variant. The K-LSM makes use of an additional array called K that constitutes of all the K values for each level from 1 to L where L signifies the number

of levels. The simplified K-LSM model - Q-LSM uses a single parameter Q to denote the number of runs at each level.

Table 3.3: *Design Decisions D* available to each model

Design	Term	Description
General Design Decisions		
	h	Bits per element
	T	Size ratio
Classic Design		
	Policy	Choice between Level or Tiered structure
Fluid LSM Design		
	K	Capacity of runs at all levels except last
	Z	Capacity of runs at the last level
K-LSM Design		
	K	Array of capacity of runs at each level
Q-LSM Design		
	Q	All K values = Q

3.2 Cost Model

The LSM Tree uses *Bloom filters* to estimate whether an entry is present at a certain level. Because Bloom filters are probabilistic, there is a possibility that they give the user a false positive and the entry is never found on this level. In this case, the LSM Tree will have to go over all keys in that level to figure out that the Bloom filter result was incorrect resulting in a penalty in the form of extra disk I/O.

The measure of efficiency of a Bloom filter is through its *false positive rate*. This rate determines the frequency with which the Bloom filter gives a wrong indication that an entry is present in the LSM Tree. The rate of false positives in the LSM Tree is directly dependent on the amount of memory allocated to it. The more memory the Bloom filter is allocated, the better the accuracy of the filter and lesser the chances of false positives. This false positive rate is calculated assuming the use of an optimal number of hash functions in the Bloom filter [16]. This formula is given by the equation [35]:

$$\epsilon = e^{-\frac{m}{n}\cdot(\ln(2))^2} \tag{3.1}$$

In this formula, ϵ represents the false positive rate of the Bloom filter, e is the natural logarithm base, m denotes the number of bits in the Bloom filter and n signifies the number of items inserted into the Bloom filter.

If the same *bits per element* was used for all Bloom filters in the LSM tree, as was the previous norm, then whenever the memory m approached the extreme ends that is 0 or $infinity$, the false positive rate would be 1 and 0 respectively. This problem can be solved by using a different false positive rate for each level of the LSM Tree [21].

$$f_i(T) = \frac{T^{\frac{T}{T-1}}}{TL(T)+1-i} \cdot e^{-\frac{m_{flt}}{N}\cdot(\ln(2))^2} \tag{3.2}$$

Here, $f_i(T)$ is the probability of false positives for the i-th hash function in a Bloom filter, T represents the size ratio between two consecutive levels of the LSM Tree. This *size ratio* determines the number of total levels in the tree. If the size of the first level is a fixed quantity, then the size of the second level will be T multiplied by the size of the first level and so on. Therefore, the total number of levels can be determined by using the total number of elements in the tree given by N and the value of T. Therefore, the number of levels is presented as a function of T using the notation

$L(T)$. The value of i represents the index of the hash function and e represents the natural logarithm. The term m_{filt} determines the amount of memory assigned to the Bloom filters and is measured as number of bits. N as used previously, represents the number of elements inserted into the tree.

If the LSM Tree does not hold the value - that is if this is an *empty read*, the LSM Tree will have to check each level of the tree, and the value of false positives is directly proportional to the extra times that the structure will have to access the database unnecessarily. If the Bloom filter returns the correct result - that is the key actually exists in the database and the Bloom filter denotes this, the LSM Tree only incurs 1 I/O for the *non-empty read*. By comparison, the *cost of a non-empty read is generally lesser than an empty-read*.

Section §3 provided the problem formulation that this thesis aims to solve. In this definition, the system *Design* represented in the problem formulation denoted by D is kept as is instead of Φ that is used in the Endure paper. The rest of the notation used in the cost model are defined below and are kept as is from the *Endure* paper.

Expected cost for each operation is defined below using the notations from Table 3.4:

3.2.1 Empty Reads (Z_0)

Empty reads or *empty point queries* occur when a user initiates search for a key that does not exist in the database. In the worst case, all the Bloom filters return a false positive and the key's possible range would cover all levels due to which, the LSM Tree would have to probe every run in every level to look for the entry. An Input/Output operation (I/O) is performed every time the Bloom filter associated with that level returns a false positive indicating the presence of an element that is not actually there. Hence the expected number of I/O operations for each level in

Symbol	Meaning
$Z_0(D)$	Cost for empty reads
$Z_1(D)$	Cost for non-empty reads
$Q(D)$	Cost for range reads
$W(D)$	Cost for writes
$L(T)$	Number of levels to fill an LSM tree with size ratio T
$N_f(T)$	Number of entries to fill an LSM tree with size ratio T
K_i	Maximum number of overlapping files for level i
$f_i(T)$	Bloom filter false positive rate at Level i with size ratio T
f_a	Read/write asymmetry ratio for storage device
f_{seq}	Cost of a sequential read relative to a random read
S_{RQ}	Selectivity of a range query

Table 3.4: Description of symbols used in the cost model.

the database is directly dependent on the rate of the false positives from the Bloom filters:

$$Z_0(D) = \sum_{i=1}^{L(T)} K_i \cdot f_i(T). \tag{3.3}$$

3.2.2 Non-Empty Reads (Z_1)

Non-Empty Reads represent the scenario where a point query successfully retrieves the data associated with the key requested by the user from the database. The probability of a successful search at any level depends on the number of data points that the level contains given the total size of the LSM Tree. Normally, a level with a larger size - owing to more data points will have a higher probability of containing the data point that is being searched for compared to a smaller level due to the size difference.

A compaction is triggered when the T-th run reaches a particular level. This

moves entries to the next level and can continue till the last level in the worst case. At any given level i, the number of runs is given by the expression:

$$T^{i-1}(T-1)$$

Considering $N_f(T)$ is the total number of entries that the tree can accommodate up to level $L(T)$, the probability of finding the data point at a specific level i is proportional to the ratio of the number of items at that level to the total number of items up to that level. The number of elements that the buffer can hold is determined by the size of the buffer given by m_{buff} divided by the size of each entry within the buffer.

The equations above help frame the expression to calculate the probability that the given entry is located on level i as:

$$\frac{(T-1)T^{i-1} \cdot m_{\text{buff}}}{E \cdot N_f(T)} \tag{3.4}$$

Summing up the capacities of all levels up to level $L(T)$:

$$N_f(T) = \sum_{i=1}^{L(T)} (T-1) \cdot T^{i-1} \cdot \frac{m_{\text{buff}}}{E} \tag{3.5}$$

The cost for non-empty reads is similar to the logic applied for empty reads and constitutes the cumulative probabilities of unsuccessful I/O operations across all levels preceding level i, from level 1 to $i-1$. This is represented as the sum $\sum_{j=1}^{i-1} f_j(T)$.

In the K-LSM model, level i may contain at most K_i sorted runs. *The cost model considers that the target entry is positioned in the median of a run.* An additional $\frac{K_i-1}{2} \times f_i(T)$ I/O operations are added based on a probabilistic model of data distribution.

The cost for a non-empty point query, $Z_1(D)$, therefore depends on the probability of the entry being located at any level within the LSM Tree. Using all these equations

and the reasoning provided, the cost for non-empty reads is derived as:

$$Z_1(D) = \sum_{i=1}^{L(T)} \left(\frac{(T-1) \cdot T^{i-1}}{N_f(T)} \cdot \frac{m_{\text{buff}}}{E} \right) \cdot \left(1 + \sum_{j=1}^{i-1} K_j \cdot f_j(T) + \frac{K_i - 1}{2} \times f_i(T) \right). \tag{3.6}$$

3.2.3 Range Queries (Q)

Range queries refer to the queries that search for intervals or range of data instead of a single point entry. The first step in a range query is to determine the starting point of the required data range in the sorted runs. To do this, the LSM Tree has to go through every level, and probe every run to be able to find the starting location. For each level, a range query triggers at most one disk seek per sorted run.

The *number of sorted runs* at a level i is denoted by K_i. After the first seek, the query performs sequential scans to retrieve the rest of the data, which are less costly than random I/O operations because there is less latency involved in reading from contiguous memory locations. The *average number of pages scanned* is determined by the sum of page counts across all the runs which are within the query range, multiplied by S_{RQ}, which represents the selectivity of range query or the proportion of relevant entries in the data for range lookups. A scaling factor f_{seq} is included to represent the difference in cost between sequential and random I/O. The total cost for range lookups, $Q(D)$, has two main components - the sequential I/O cost to scan all the data and the sum of disk seeks needed to start the scan at each level. Using the above points, the equation for cost of range query lookup is calculated as:

$$Q(D) = f_{\text{seq}} \cdot S_{\text{RQ}} \cdot \frac{N}{B} + \sum_{i=1}^{L(T)} K_i. \tag{3.7}$$

Here, the term $f_{\text{seq}} \cdot S_{\text{RQ}} \cdot \frac{N}{B}$ calculates the cost of sequentially scanning the pages,

where N is the total number of elements, and B is the number of entries per page. The summation $\sum_{i=1}^{L(T)} K_i$ represents the sum of disk seeks for the sorted runs at each level.

3.2.4 Write Queries (W)

The write cost (W) is modeled for the worst-case scenario, where incoming entries do not overlap. As a result, most entries will traverse all levels of the Log-Structured Merge (LSM) tree. To estimate the expected number of I/O operations, the average number of merge operations is calculated for a single write at each level (i), and then summed across all levels. The total number of merges at a level is based on the fact that each level will receive flushes from the previous level until a full compaction event is initiated. The final flush of a sorted run is involved in just one merge because the level reaches capacity with that run. However, earlier flushes participate in an increasing number of merges, with the earliest flush undergoing the most number of merges. For K_i sorted runs at Level i,the total count of merge operations is calculated by summing the merges participated in by each flush and then adjusted by the number of sorted runs at that level. Using this approach the I/O cost for write operations is given by the equation:

$$K_i \cdot \sum_{j=1}^{(T-1)/K_i - 1} j = (T-1) \cdot \frac{(T-1-K_i)}{2K_i} \tag{3.8}$$

The average number of merges a single write participates in at Level i is based on the total merges for that level, adjusted by the flushes from Level $i-1$, and incremented by one because of the last flush that results in a full-level merge. This average number of merges translates to the I/O operations that a single write will contribute to given by: $\frac{T-1+K_i}{2K_i}$. For the cost of a single insert, the cost model considers the average number of merges per level along with the number of entries per page, denoted by

B. The potential asymmetry between reads and writes on the storage device is represented using a factor f_a. If a has a higher value, it implies that the LSM Tree has a greater cost for write operations relative to read operations. The term f_{seq} denotes the sequential I/O operations that represent the cost difference between sequential and random I/O operations. The total I/O cost for writes, represented as $W(D)$, is calculated as:

$$W(D) = f_{\text{seq}} \cdot \left(1 + f_a \cdot \frac{1}{B}\right) \cdot \sum_{i=1}^{L(T)} \frac{T - 1 + K_i}{2K_i}. \tag{3.9}$$

3.3 Defining the Problem

Problem 1 *Formally, the problem seeks to optimize the LSM tree configuration, under the constraint of a given System, to minimize the cost associated with processing a specific Workload. The cost function* $C(\mathbf{W}, \mathbf{D})$ *is defined as the number of I/O operations required by a Workload* $\mathbf{W}$*, under an LSM Design configuration* $\mathbf{D}$*, constrained by the System* $\mathbf{S}$*. The optimization challenge can thus be expressed as:*

$$\mathbf{D}^* = \arg\min_{\mathbf{D}} \; C_S(\mathbf{W}, \mathbf{D})$$

The cost model function, denoted by C as depicted in the equation above, is derived from the Endure framework and is one of the main parts of the optimization process. It captures the computational expense of I/O operations for a specific set of database activities denoted by workload W. This function will assess the performance impact of a potential design decision D within the bounds set by the system's limitations S without running these designs on a physical database system.

The cost is calculated by considering a combination of multiple attributes. The first comprises both frequency and the types of operation in the workload. For example, a design that supports a tiering-heavy structure might help decrease I/Os for a workload that contains more write operations than all other operations. In con-

trast, the design might lean more towards a levelling style compaction policy for a read-heavy workload. The system efficiency and limitations are the second factor that play a key role in determining the cost. For example, the more total memory available to the LSM Tree structure, the more buffer space and the more Bloom filter memory it can allocate. More total memory will lead to better write and read costs as there is a higher probability that the data to be read is in the buffer and more data can be written to the disk in one flush. The Bloom filters will give more accurate results when more memory is assigned to it [21], which will lead to less waster I/Os while accessing data. Ultimately, the tunable design parameters are balanced to adjust the LSM tree configuration to balance performance and resource utilization by minimizing the computed total cost.

This aim of this thesis is to minimize this cost function to find the most suitable design with the lowest computational expense while considering the system constraints for a specific workload. We discuss the Methodology used to solve the problem and its Implementation details in the next Chapter 4.

Chapter 4

Methodology

This thesis aims to solve the problem of finding the best tuning for the LSM Tree. Manual tuning is difficult because it requires intricate domain knowledge and traditional evaluation methods often fail to capture the complex behavior of modern systems. The solution proposed uses a twofold approach. It makes use of the *Analytical Cost model* [34] for evaluation to give a value that directly translates to *how good* the model performs. This cost model acts as a *target function* for the *Bayesian Optimization* module. By using this approach, this thesis aims to alleviate the issues associated with running a load directly on a deployed database system and avoid the potential time loss and inaccuracies associated with measuring each *tuned design* on the actual physical system. The motivation for this solution is that modern day solvers cannot search over categorical spaces. The cost model produces floating-point values for categorical attributes that leads to theoretically correct values, but solutions which cannot be applied to the models in the real world.

To overcome this, BO treats the design space as a holistic cost surface, separately accounting for categorical and continuous variables within this cost surface. This thesis uses the closed-form cost model only for evaluation purposes. Using this method, it aims to create a system that is not only fast and precise in its calculation, but also uses a fair and equal method to evaluate each design. In the experiments used in this thesis, evaluation of the design decision is done by comparing the cost of the

predicted design by *Bayesian Optimizer* with the cost of the design decision predicted by a solver using the *Cost Model*. We call the solution provided by the solver the *Analytic solution* and the associated cost the *Analytic cost* for the duration of this thesis document.

4.1 Bayesian Optimization Using BoTorch

There are a number of reasons for deciding to use the BoTorch [17] library for implementing the Bayesian Optimization process. BoTorch supports *Monte-Carlo (MC) Acquisition Functions*, which are important for dealing with complex optimization scenarios where normal analytic acquisition functions may not be available or could be difficult to compute. Another reason, why BoTorch is popular is because it *introduces an average approximation optimization approach*, which for the combination of deterministic higher-order optimization algorithms and variance reduction techniques. This is required for efficient optimization when noisy function evaluations are required. *BoTorch can also be integrated with PyTorch*, which allows users to fully utilize Pytorch's functionality. This lets users implement new acquisition functions and probabilistic models.

BoTorch is designed to *leverage modern hardware like GPUs for scalable parallel computation* by letting users make use of Tensors, which in turn leads to significant performance gains for large-scale problems. This is done by making efficient use of Tensors due to which there is notable performance improvements especially in large scale problems. Moreover, BoTorch introduces a new *one-shot formulation* of the Knowledge Gradient acquisition function. This helps in improving performance and are *backed by theoretical convergence proofs*. These proofs also validate the framework's sample average approximation technique for Monte Carlo acquisition functions, proving the robustness of Bayesian Optimization framework. BoTorch also *focuses on*

parallelism and hardware acceleration which allows it to perform rapid batch evaluations of acquisition functions. Empirical results show *BoTorch's algorithms achieve greater sample efficiency* compared to other packages, with the One-Shot Knowledge Gradient often outperforming other acquisition functions. Therefore, BoTorch represents a significant improvement while tackling a broad scale of problems.

4.2 Design Choices

There are 3 main design decisions made in the solution proposed in this thesis among others. This implementation, the possible alternatives and the application of each choice, along with the background on each of these general methods is detailed in section §2. This section focuses on BoTorch specific implementation and choices made in this thesis.

4.2.1 Acquisition Function in BoTorch

The first choice made is for the acquisition function. BoTorch treats acquisition function as *heuristics* to evaluate the *utility* or usefulness of one or more design points so that the objective of the underlying black-box function can be maximized. BoTorch provides a `AcquisitionFunction` API that supports most of the acquisition functions. These consist of both *analytic* and *(quasi-)Monte Carlo* based methodologies. Due to this flexibility, BoTorch can be used to optimize a diverse range of functions.

The first kind of acquisition function offered by BoTorch is the *Monte Carlo* set of acquisition functions that are used for complex scenarios or when evaluating batches of design points ($q > 1$). Monte Carlo methods approximate the expected value of a function over the model's outputs, especially when the problem cannot be easily solved

or analyzed using analytical methods. The acquisition function is approximated as:

$$\alpha(X) \approx \frac{1}{N} \sum_{i=1}^{N} a(\xi_i), \quad \xi_i \sim \mathbb{P}(f(X)|\mathcal{D}), \tag{4.1}$$

where $\mathbb{P}(f(X)|\mathcal{D})$ is the posterior distribution of f at X, given observed data $\mathcal{D}$. *The q-Expected Improvement (qEI) is one of the Monte-Carlo acquisition functions used in this thesis.* The approximation used by (qEI) takes the form:

$$qEI(X) \approx \frac{1}{N} \sum_{i=1}^{N} \max_{j=1,\ldots,q} \{\max(\xi_{ij} - f^*, 0)\}, \quad \xi_i \sim \mathbb{P}(f(X)|\mathcal{D}), \tag{4.2}$$

This function leverages MC sampling to estimate integrals over the posterior distribution, thereby facilitating the evaluation of batch acquisition functions where analytical solutions are not feasible.

The second set of acquisition functions available to BoTorch is the *analytic acquisition functions*. Analytic acquisition functions are important for scenarios where a single design point ($q = 1$) is considered. This implies that analytic acquisition functions possess the capability to directly calculate the posterior distribution values at certain points by using the summarized characteristics or statistics of that distribution, without having to approximate or simulate the full distribution each time.

BoTorch provides support for multiple analytic acquisition functions out of the box. *This thesis uses the Upper Confidence Bound and Expected Improvement functions* for the solution to the optimization problem due to the proven results with EI and experimental tests showing that UCB has the potential to match the performance of EI. The *Expected Improvement* function aims to identify points that are expected to improve upon the current best observation, striking a balance between exploration and exploitation. Expected Improvement (EI) in BoTorch provides an expression

that captures the expected benefit over the current best observation:

$$EI(x) = (\mu(x) - f^* - \xi)\Phi(Z) + \sigma(x)\phi(Z), \tag{4.3}$$

where $Z = \frac{\mu(x) - f^* - \xi}{\sigma(x)}$, with $\mu(x)$ and $\sigma(x)$ denoting the posterior mean and standard deviation at point x, f^* representing the best observation, and ξ being a small positive number to foster exploration.

The second function used for experimentation is the *Upper Confidence Bound (UCB)* that allows explicit control over the exploration-exploitation trade-off through a tunable parameter.

$$UCB(x) = \mu(x) + \kappa\sigma(x), \tag{4.4}$$

where κ modulates the exploration-exploitation trade-off.

4.2.2 Models and Kernel in BoTorch

In Bayesian Optimization (BO), models act as surrogates to the actual underlying function that is to be optimized. BoTorch uses Gaussian Processes (GPs) for this purpose and maps the design points to a posterior probability distribution. The BoTorch framework supports a broad range of models through a flexible Model interface. In particular, *this thesis uses the Single Task Gaussian Process for optimization.* This is because the function to be modelled is predominantly simple which is well suited for the Single Task GP. Also, this model is made for situations where the training data used across all outputs is consistent which is true for the problem this thesis solves. A third criteria is that conditional independence should be maintained between outputs which this problem meets. The solution also considers the fact that the function to be modelled is well-defined and therefore does not require more complex models or modelling noise for optimization. Furthermore, as per the literature, GP has

worked best for most optimization problems [36, 37]. Gaussian Process is also good at quantifying uncertainty which is important for guiding the optimization process. In particular, `the MixedSingleTaskGP` is used as the problem requires modelling of both continuous as well as discrete search spaces. `MixedSingleTaskGP` can model both these spaces together along with maintaining support for just continuous search spaces as well. The model employs a composite kernel structure designed to account for the heterogeneity in the feature space, integrating both continuous and categorical variables. This kernel can be expressed as:

$$K((x_1, c_1), (x_2, c_2)) = K_{\text{cont},1}(x_1, x_2) + K_{\text{cat},1}(c_1, c_2) + K_{\text{cont},2}(x_1, x_2) \times K_{\text{cat},2}(c_1, c_2), \tag{4.5}$$

where (x_i, c_i) represents the pair of continuous (x_i) and categorical (c_i) features of the input. *The model differentiates between the continuous and categorical components by using two distinct kernels for each - a* ***`CategoricalKernel`****, based on Hamming distances for categorical features, and a regular kernel for continuous features.* The notation signifies that the model is equipped with the capability to fit different lengthscales for the kernels in both the additive and multiplicative terms, enhancing its flexibility to model complex interactions within the search space. As mentioned previously, the Matern kernel is a generalization of the Radial Basis Function (RBF) kernel with a smoothing parameter.

The model in BoTorch, uses a kernel to encode assumptions about the function's behavior over the input space. In particular, *use the Mixed Kernel* and the `optimize_acqf_mixed` function is used to get the next points via the acquisition function. This function requires the user to pass the bounds, the training data and the evaluated values of the training data using a target function, as well as a list of all possible combinations of the categorical attributes.

4.2.3 Selecting the Log Likelihood

The log likelihood is a measure of how likely the observed data is under the assumed probabilistic model. In Gaussian process regression, which this thesis uses to formulate the solution and is generally used in Bayesian optimization, the log likelihood quantifies how well the Gaussian process model fits the observed data points. Maximizing the log likelihood with respect to the kernel hyper parameters allows the model to adapt and better represent the underlying objective function. For a given GP $f \sim \mathcal{GP}(\mu, K)$ with mean function μ and covariance function K, and observed data $\mathbf{X}, \mathbf{y}$, the MLL is computed or approximated as:

$$\mathcal{L} = p(\mathbf{y}|\mathbf{X}) = \int p(\mathbf{y}|f(\mathbf{X}))p(f(\mathbf{X})|\mathbf{X})\, df \tag{4.6}$$

In BoTorch, when the GP inference is computed exactly, like in the case of regression with a Gaussian likelihood, the MLL can be evaluated directly. Alternatively, for cases involving approximate inference, the MLL may be approximated or bounded. For this specific problem, this thesis utilizes the exact MLL provided by BoTorch as it offers several advantages. Firstly, it ensures that the hyperparameters are tuned to provide the best explanation for the observed data under the GP prior and likelihood. The exactness of the MLL is preferable when dealing with noiseless or low-noise problems, where the assumptions of the Gaussian likelihood are closely aligned with the true data-generating process which is true for the problem this thesis aims to solve. It should also be noted that maximizing the exact MLL can lead to better generalization on unseen data, as it fully captures the uncertainty modeled by the GP. Moreover, the exact MLL forms an *interpretable loss function*. This thesis aims to minimize the negative MLL to achieve the Maximum Likelihood estimate during the optimization process.

The choice about which Log Likelihood to use depends on the size of the dataset, the noise characteristics, and the computational resources at hand. For this problem, given the dataset characteristics and the requirement for precision, the exact MLL from BoTorch is the most suitable choice.

4.3 Implementation

This Section describes the exact implementation of the processes described and the methods used throughout this thesis. The exact flow of the BO loop and the database schema used for implementation and storage of experimental results is detailed in Section 4.3.1 and Section 4.3.2 respectively.

4.3.1 Programmatic Implementation

In this thesis, the optimization process is started by using a sample of 20 data points. This sample of data points represents selecting a random point from a uniform distribution for *Workload* and another random point for *System* parameters from an uniformly distributed sample. These represent the parameters which will not change during the optimization process. These parameters are also the data points represented as training points which is the first parameter fed to the model after scaling. This thesis uses the default *Scaling* function in BoTorch to scale these points, and then determine via random sampling, a design subject to the system parameters. This is not an *ideal design* but rather a randomly selected one. The design cost is then evaluated using the cost model which also acts as the *target function*. This evaluated cost becomes the target set which is standardized and passed to the MixedSingleTaskGP model by using the default Standardize function provided by BoTorch. The other set of parameters required by the BoTorch model is the *bounds*. There are two considerations for the bounds of the data provided to the model. The first one are the

bounds set by the user and the second set is determined by the system capabilities. The bounds and standardized target set form the basis for initializing the BoTorch model, specifically a MixedSingleTaskGP model, which caters to mixed search spaces composed of discrete and continuous features. An Exact Marginal Log Likelihood (MLL) function is then applied to fit the model to the data. This is demonstrated in Diagram 4·1.

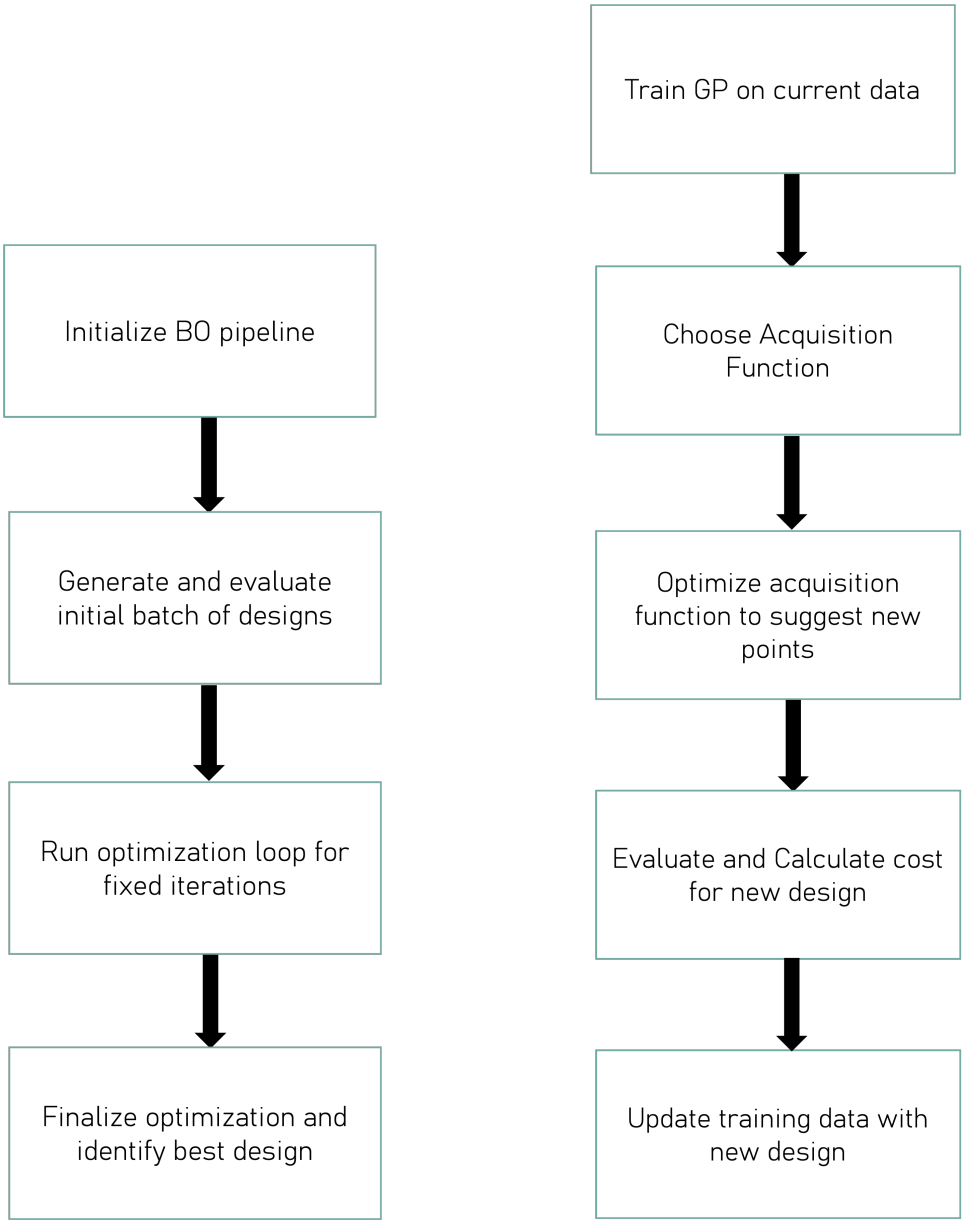

Figure 4·1: The Bayesian Optimization process used in the solution (a) gives the outline of the entire process and (b) expands the optimization loop

The proposed solution in this thesis uses BoTorch's `optimize_acqf_mixed` method to implement the acquisition function and get the next set of *candidates* to evaluate. This mixed-integer optimization algorithm efficiently manages the categorical features while optimizing the acquisition function, which in the case of the solution proposed in the paper could be Expected Improvement (EI), Upper Confidence Bound (UCB), or the q-Expected Improvement (qEI). The problem this thesis aims to address is a minimization problem whereas the Bayesian optimization considers a maximization problem by default. This can be handled by a parameter called *maximize* in the acquisition function. However, the Monte carlo acquisition function qEI does not provide this parameter and therefore this information is passed to the acquisition function by taking the negation of the highest target value observed so far to denote the best observed point. This loop is set to run fifteen times, finding a different design decision at each iteration.

The optimization process is also influenced by two parameters in the acquisition function - *num_restarts* and the *raw_samples*. This is set to the minimum possible value to account for faster optimization speed. However, doing this leads to lower accuracy as explained in the Chapter 5. The parameter *num_restarts* specifies the number of times to restart the optimization algorithm. This is required because the optimization of acquisition functions is usually non-convex and there could be multiple local minima. Therefore to ensure that the optimization function reaches an actual global minima and does not get stuck at a local minima, the optimization process is restarted several times from different initial points. The specific parameter *num_restarts* controls how many different sets of starting points will be used. If the value for this parameter is higher, then there is a higher possibility for the acquisition function to find the global minima but will cause an increase in the computational time. The parameter *raw_samples* controls the number of samples to get from the

acquisition function before the actual optimization begins. The best points from these samples are used for starting the optimization process by the acquisition function. If the value is higher, it allows the acquisition function to find better starting points leading to better design predictions but will add to a higher computation time. Upon finalizing the optimization hyper parameters, the proposed solution extracts the best-performing design as per the target function's evaluations. This design is theoretically optimal within the explored parameter space, considering the bounds and constraints imposed on the system.

4.3.2 Database Schema

This thesis uses a relational database managed via SQLite to structure and handle all experimental data efficiently. The choice of SQLite was made due to its robustness, ease of integration with Python, and its support for complex data relationships essential for the experimental analyses. This subsection details the schema of the database, which consists of several interconnected tables designed to log every experimental run, store configuration costs, and detail run outcomes. Each table serves specific functions, from storing raw experimental data to logging detailed cost analyses, thus ensuring comprehensive data capture essential for thorough performance evaluation and optimization studies. The database consists of four main tables: *runs, design_costs, run_details*, and *run_details_k_values*.

The relationship between each table, foreign keys and detailed column information is portrayed in the Diagram 4·2

The `runs` table records each experiment's run, capturing metrics for the *Workload* such as the ratio of empty reads, non-empty reads, range queries, writes as well as the *System* parameters like maximum number of bits per element, number of physical entries per page, the range selectivity, entries per page, total number of elements

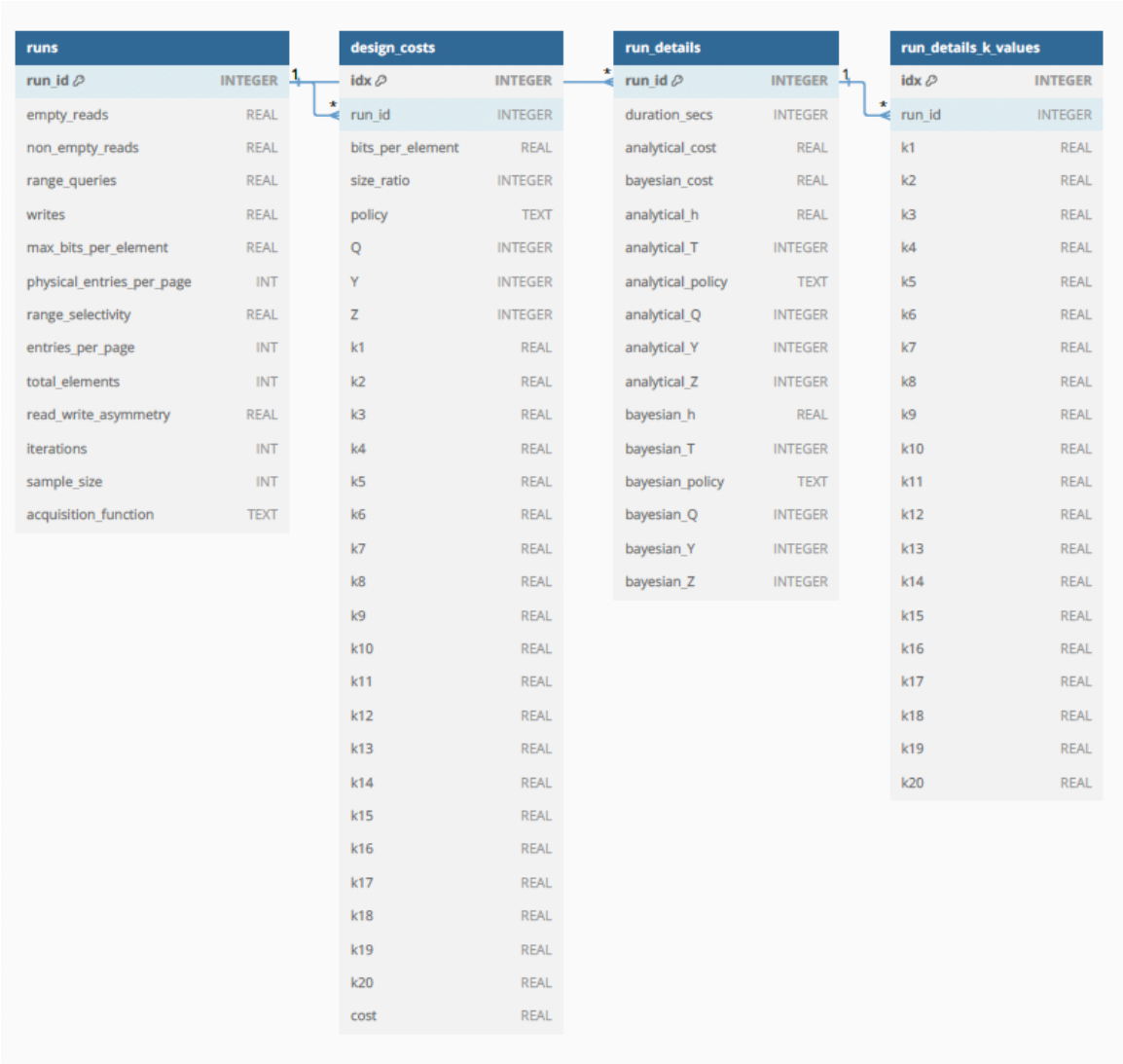

Figure 4·2: This diagram represents the exact schema for the database used for storage of different parameters during the optimization process.

stored in the tree and the read/write asymmetry of the data. It also stores the number of iterations run when the BO pipeline was executed, the sample size that was used while executing it and the acquisition function used for the run. These details represent the factors that will not change during the optimization process of BO.

The `run_details` Table extends the `runs` table using the `run_id` and provides more granular details on the run's duration and the costs calculated using the BO implementation and the cost and designs for the analytical method. It specifically stores all the detailed design decisions made by BO and the exact decisions suggested by the analytical solver for the same Workload and System values and evaluates and stores the cost for both. This is specifically useful for us because we use this for calculating the efficiency of the BO method over the closed form solution.

The `design_costs` Table is an extended implementation of the `run_details` specifically for each iteration in the BO process. While the `run_details` table stores the *best design* decision made by the optimization process, this table records every decision made as well as the cost for that design. This table helps store all the design decisions in one place and access cost of that design for any LSM structure including Q values for the Q-LSM design, each of the K-values for the KLSM design, The K and Z values for the Fluid LSM model and the size ratio and bits per element for all the models.

The last table `run_details_k_values` stores the analytical K values suggested by the closed form tuner for the K-LSM model specifically. We do not exclusively use these values and prefer to store them separately in case required for comparisons.

Chapter 5

Experiments and Results

5.1 Experimental setup

In this thesis, all experiments are conducted on the same machine to ensure the accuracy and to be able to reproduce results as and when required. The specific configurations used are detailed below:

5.1.1 Hardware and Software Configurations

The system is powered by a 13th Gen Intel(R) Core(TM) i9-13900KF processor, featuring a CPU family 6 and model 183, with 24 cores per socket and 2 threads per core. The CPU operates on a base frequency of 800 MHz and can turbo boost up to 5800 MHz, offering robust computational capability. The machine is also equipped with **1**25 GB of RAM, for accommodating memory-intensive applications. The VM possesses multiple storage devices, including a 3.5 TB SSD mounted on /scratchSSD, and two HDDs each with 7.3 TB capacity mounted on /scratchHDDa and /scratchHDDb, respectively. It also has a 1.7 TB NVMe drive mounted on /scratchNVMe, offering rapid data access speeds. The system runs on the a 64-bit system. More specifically it uses the Ubuntu operating system with kernel version 6.5.0-21-generic on x86_64 architecture. The experiments use a set of Python libraries essential for conducting Bayesian optimization, data manipulation, visualization, and other computational tasks.

These set of hardware specifications allow the proposed solution to meet the computational needs of the Bayesian Optimization process.

5.2 Evaluation

This section outlines the experiments that were run and compare the accuracy of various configurations of models that are used.

For each of the following experiments, the LSM Tree structure is varied along with the hyperparameters - *num_restarts* and *raw_samples*. The parameter raw_samples refers to the number of quasi-random points generated across the parameter space before the commencement of the optimization process. These points are crucial for the preliminary evaluation of the acquisition function, allowing for a broad assessment of the search landscape. The num_restarts parameter specifies the number of times the optimization algorithm is restarted with different sets of initial conditions. This approach is designed to mitigate the risks of converging to local minima and to enhance the probability of discovering a global optimum. In the experiments conducted, these two parameters have played a crucial role in variation of results and are therefore specifically varied. The experiments mainly make use of the Expected Improvement (EI) acquisition function due to its well-documented efficiency in balancing exploration and exploitation within the optimization landscape. This choice was particularly motivated by its performance in prior benchmarks within similar high-dimensional spaces [38, 39].

5.2.1 Evaluating Accuracy on the Classic LSM Tree

The experiments outlined in this section perform multiple experiments on the Classic LSM Tree to form a baseline for the more complex structures of the LSM. Some observations can be made from these experiments which are extended to the other

models.

The first graph represents the experiment run on the Classic LSM model with the Optimization loop set to use the **Expected Improvement** acquisition function which utilizes the MixedSingleTaskGP for modelling the function and set the num_restarts to 2 and raw_samples value to 3. The second graph demonstrates the same optimization problem but with the Acquisition function set to the **Upper Confidence Bound** function.

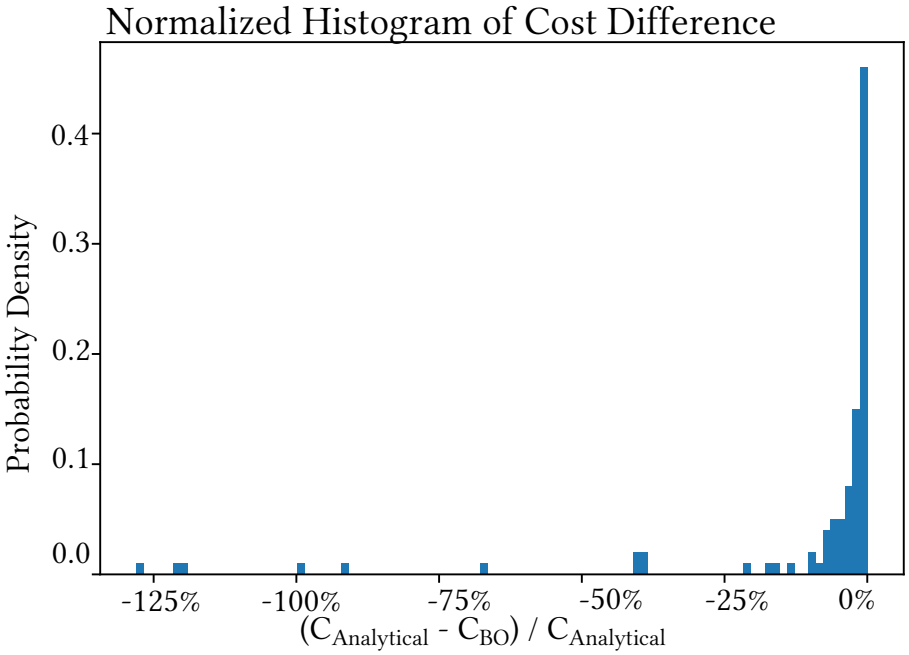

Figure 5·1: This graph presents the normalized histogram of cost difference for designs predicted for Classic LSM model by the bayesian optimization pipeline and the closed form analytical solver used in Endure. This experiment uses the Expected Improvement function and the hyperparameters - num_restarts is set to 2 and raw_samples is set to 3.

In the next graph, the experiment aims to compare the performance of the above Classic LSM Model using the EI function with the UCB function. Even though literature suggests that EI performs best for most database tuning problems, it can

be noted that UCB performs even better in terms of the maximum deviation from the *ideal* value predicted by the analytical model for the simple design of the *Classic LSM Tree*.

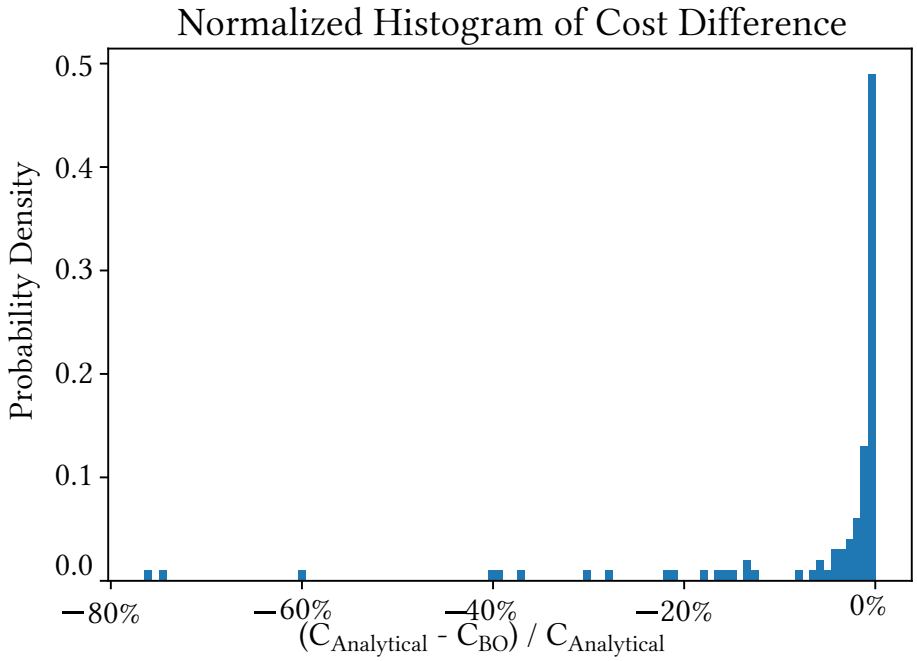

Figure 5·2: This graph presents the normalized histogram of cost difference for designs predicted for Classic LSM model by the bayesian optimization pipeline and the closed form analytical solver used in Endure. This experiment uses the Upper Confidence Bound function and the hyperparameters - num_restarts is set to 2 and raw_samples is set to 3.

The first two graphs, Figures 5·1 and 5·2, show that most values are centered around the 0% line, denoting that the designs predicted by the Bayesian optimization (BO) model are almost *as good as* the analytical results. However, upon further inspection, it can be noted that there are a few outliers in both graphs. The worst design performance for the EI function is approximately 125% more than the cost predicted by the analytical tuner, whereas for the UCB function, it is 80% more. Notably, even though the difference is marginal, more points lie on the 0% line for

the EI acquisition function than for the UCB acquisition function. This number of points on the 0% line decreases with the increase in complexity of the model for the UCB function; therefore, in subsequent experiments, the EI acquisition function is selected as the acquisition function of choice, as also demonstrated in the literature.

In the next experiment, this thesis tries the qExpectedImprovement function, hoping that introducing randomization into the optimization process might help in prediction of new design spaces.

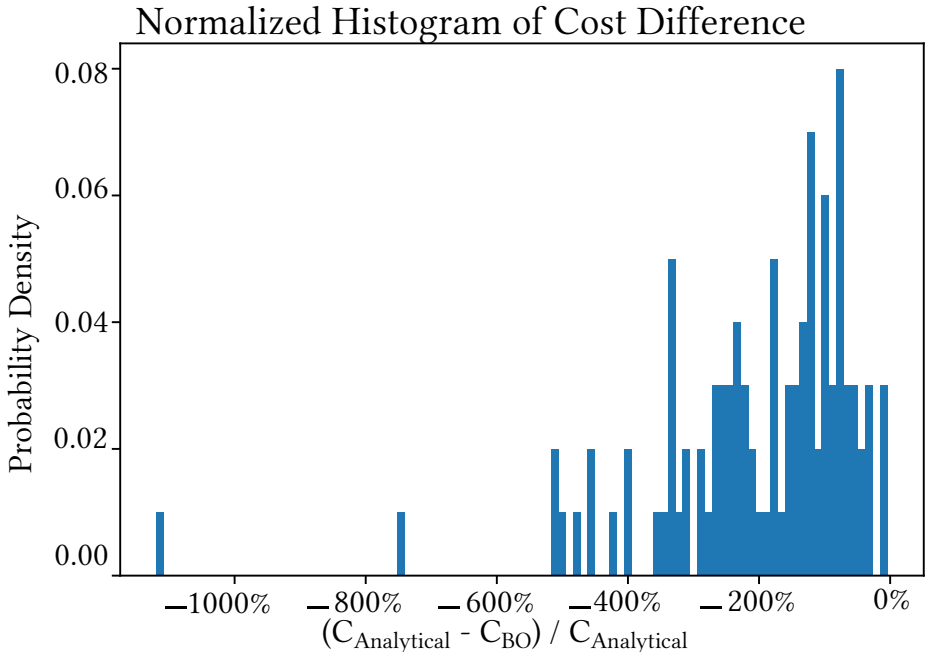

Figure 5·3: This graph presents the normalized histogram of cost difference for designs predicted for Classic LSM model by the bayesian optimization pipeline and the closed form analytical solver used in Endure. This experiment uses the qEI acquisition function and the hyperparameters - num_restarts is set to 2 and raw_samples is set to 3.

This experiment represented by Figure 5·3 indicates that qEI (quantitative Expected Improvement) performs poorly in optimizing the problem addressed in this thesis. This could be due to inherent design of the qEI function, which favors exploring random new points over exploiting known ones. This extensive exploration can

be inefficient for well-defined problem spaces where exploitation is more critical for locating the minimum cost design space effectively like the one described in this thesis. Additionally, qEI is generally better suited for batch processing and may struggle with the iterative progression required by this approach. Another contributing factor to its poor performance could be the sensitivity of qEI to hyper parameters, which in this case, may not have been optimal. It could be hypothesized that increasing the hyper parameters might lead qEI to do better, however, the goal of the thesis is to reduce computational complexity and be able to do predict best designs in a short amount of time. This goal cannot be realized by providing a large amount of time and resources to the qEI function. Due to these disadvantages and the poor performance of qEI, it is not used in future experiments.

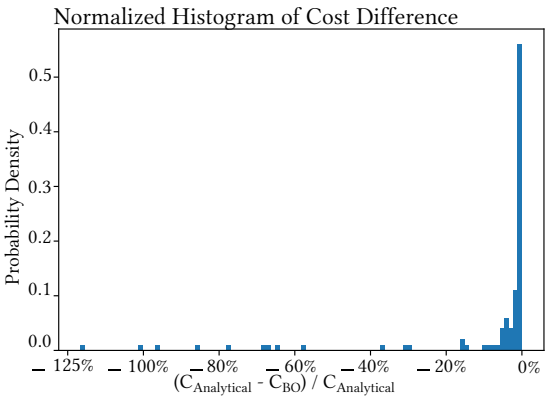

Figure 5·4: Comparison on Classic LSM between BO cost and analytical cost using EI with hyperparameters - num_restarts set to 30 and raw_samples set to 50.

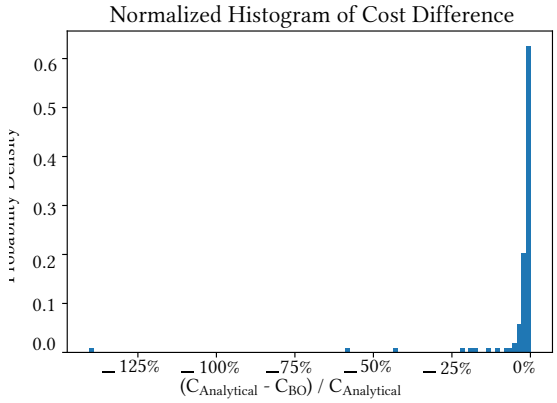

Figure 5·5: Comparison on Classic LSM between BO cost and analytical cost using EI with hyperparameters - num_restarts set to 100 and raw_samples set to 200.

The above two graphs - Figure 5·4 and Figure 5·5 demonstrate the effect of increasing the value of the hyper parameters *num_restarts* and *raw_samples*. It can be noted that while there is no drastic change in performance, the y-axis demonstrates

that when the values of the two parameters are increased, the probability density of the points at the 0% line increases denoting a subtle but positive increase in desired results. For a more complex model, it can be noted that this change is more pronounced as demonstrated later in this chapter. Since the Classic LSM Tree has a simpler structure, the higher amount of number of restarts and raw samples is not required to get a desired low-cost design from the BO model.

5.2.2 Evaluating Accuracy on the Q-LSM Tree

These experiments measure the performance of the proposed solution on the Q-LSM model. Two experiments are run on this model while varying the hyper parameters - raw_samples and num_restarts. The acquisition function is set to Expected Improvement.

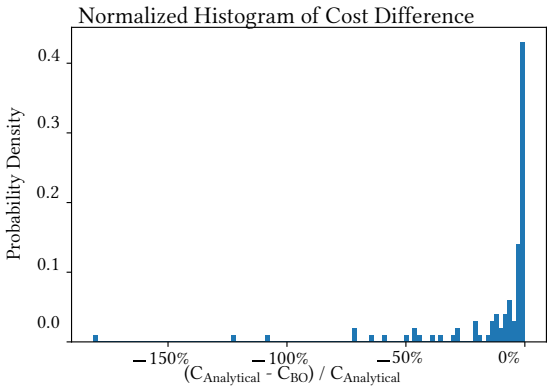

Figure 5·6: Comparison of Q-LSM cost with analytical using EI with hyperparameters - num_restarts set to 2 and raw_samples set to 3.

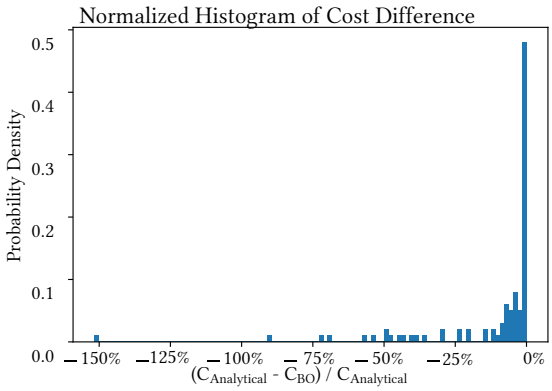

Figure 5·7: Comparison of Q-LSM cost with analytical using EI with hyperparameters - num_restarts set to 100 and raw_samples set to 200.

The graphs - Figure 5·6 and Figure 5·7 compare the performance of the proposed solution when the hyper parameters are increased. It can be observed that the Q-LSM similar to the Classic model does not show noticeable variations in performance

when the computational complexity is increased. This is because the Q-LSM model despite having one additional parameter compared to the Classic model, the overall optimization problem remains relatively simple.

5.2.3 Evaluating Accuracy on the Fluid LSM Tree

The following experiments are run on the Fluid LSM model to evaluate the performance of the proposed model on this Tree structure.

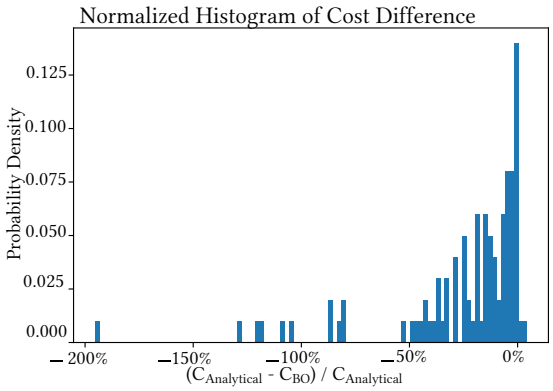

Figure 5·8: Comparison of Fluid LSM cost with analytical using EI with hyperparameters - num_restarts set to 2 and raw_samples set to 3.

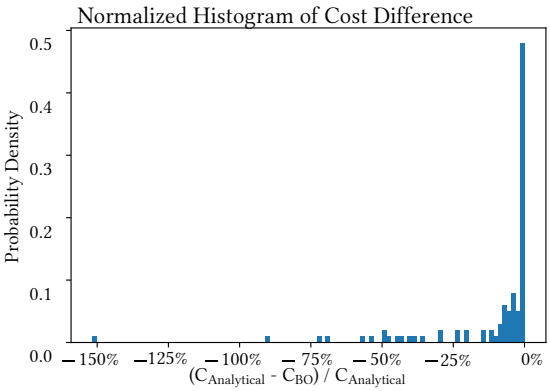

Figure 5·9: Comparison of Fluid LSM cost with analytical using EI with hyperparameters - num_restarts set to 100 and raw_samples set to 200.

As demonstrated in Figures 5·8 and 5·9, the impact of increasing the values in the Fluid LSM model is much more pronounced. It can be observed that for a model like this one which deals with much more categorical values and a bigger design space, the BO model does much better with the increase in computational complexity. The model does significantly better with the correct hyper parameter values and it can be expected that there is a possibility of improving performance even further with more increase in computational complexity.

5.2.4 Evaluating Design Cost Per Iteration

The next set of experiments serve as a demonstration of the cost surface explored by the Bayesian optimization process for each design that it suggested for the LSM Tree. We perform these experiments in two sets - the first graph in each set demonstrates the design costs for the *best designs* which are the five designs that have the least normalized cost difference from the cost of designs suggested by the analytical tuner whereas the second graph demonstrates the the design costs for the *worst designs* which are the five designs that have the most normalized cost difference from the cost of designs suggested by the analytical tuner. The x-axis demonstrates the design decisions over each iteration - where each optimization loop uses 15 iterations to find the best design point leading to the minimum possible cost. The y-axis gives the normalized cost of the design at each iteration stage.

Classic LSM

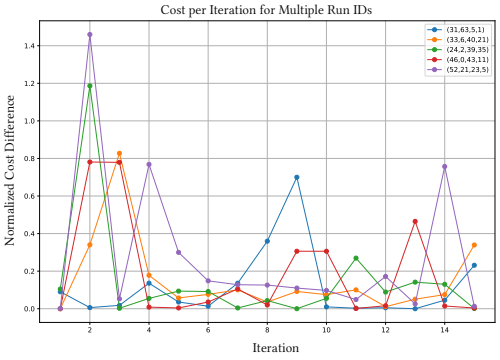

Figure 5·10: Best normalized design costs for the designs found by the BO loop over each iteration for 15 iterations for the Classic LSM model.

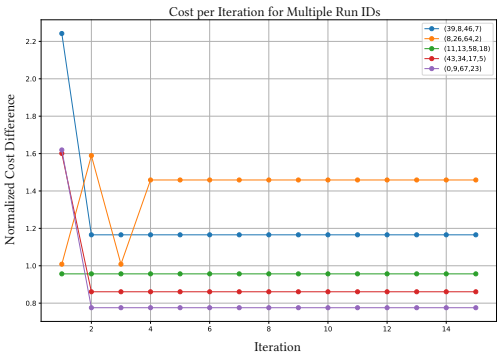

Figure 5·11: Worst normalized design costs for the designs found by the BO loop over each iteration for 15 iterations for the Classic LSM model.

Q-LSM

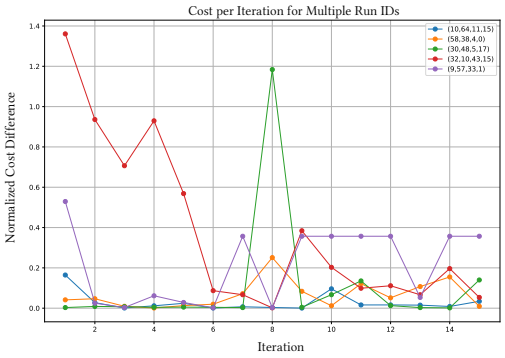

Figure 5·12: Best normalized design costs for the designs found by the BO loop over each iteration for 15 iterations for the Q-LSM model.

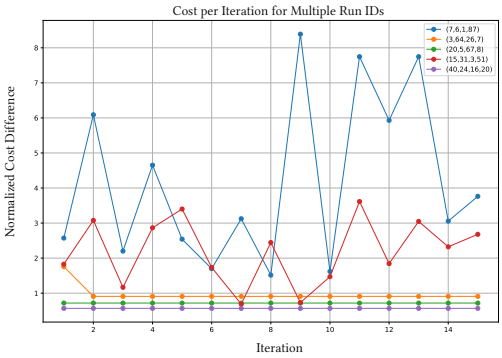

Figure 5·13: Worst design costs for the designs found by the BO loop over each iteration for 15 iterations for the Q-LSM model.

Fluid LSM

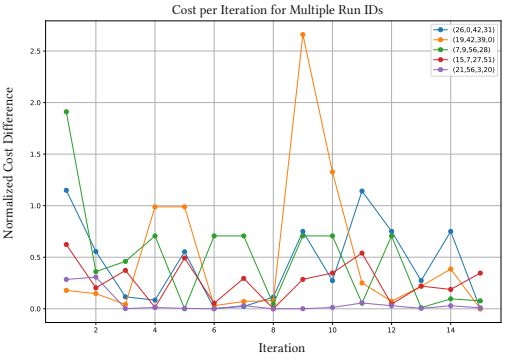

Figure 5·14: Best normalized design costs for the designs found by the BO loop over each iteration for 15 iterations for the Fluid LSM model.

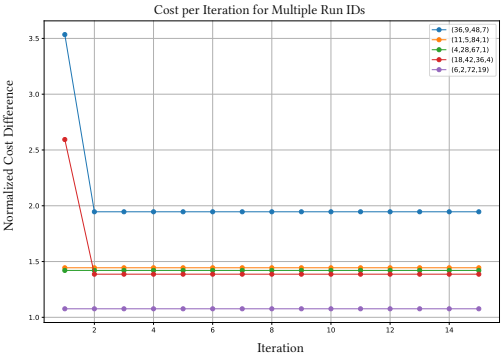

Figure 5·15: Worst normalized design costs for the designs found by the BO loop over each iteration for 15 iterations for the Fluid LSM model.

In the above graphs, it can be observed that in general, the *best design decisions* demonstrated in Figures 5·10, 5·12, 5·14 exhibit a high-low trend. This can be attributed to the fact that BO performs well and tries to explore more even though it finds a good design point, as evidenced by the graph reaching the 0 mark - denoting that the design cost predicted by BO has no difference from the cost of the design suggested by the analytical solver and still trying to do better. This behavior implies that the process is robust enough to get out of local minima and explore enough to be able to find a global minimum point.

In contrast, the *worst design decisions* demonstrated in Figures 5·11, 5·13, 5·15 exhibit a linear pattern which indicates that they explored and found a low point and got stuck at that point due to which it was not able to find the true global minima. The exploration tendency could be optimized by tuning the LSM Tree for higher exploration rates by using an acquisition function like Upper Confidence Bound and changing the β value, however, this might affect the runs that perform well with the current amount of exploration-exploitation trade-off. For the Q-LSM graph in 5·13, we notice a break in this pattern in the blue and red lines denoting that the optimization process does make an effort and try to find a different point. It is possible that a higher number of iterations in the BO process could in fact lead to the BO process to find the true minimum point. However, contrarily the increase in number of iterations would also lead to higher computational complexity and greater time to optimize which this thesis aims to reduce, and therefore it can be agreed that the current method does pretty well, keeping all these factors in mind.

The other advantage of using the proposed solution is the fact that it can be observed that the optimization loop takes around a few seconds for the Classic and Q-LSM models and upto a few minutes for the Fluid LSM model. This is a distinctive deduction in time from the previous methods described in Literature which implement

each design decision on the real database and observe the impact of each design for approximately five minutes. Using such methods, there is almost a 100x decrease in optimization time in the BO process compared to the other tuning procedures suggested in other papers. These discussions as well as more detailed observations of the results are continued in more detail in the next Section 6.

Chapter 6

Discussion

This section discusses the implications of the observations made in Chapter 5. The first advantage that is provided from using the proposed solution is that of flexibility. The modular design of the BoTorch library along with the Analytical cost function allows the user to substitute multiple predefined acquisition functions and even the target cost model with a different method of evaluation. These target functions could be a Learned ML model, a different analytical cost evaluator, a combination of the two or even the execution of the suggested design decision and workload on the actual database.

This thesis uses the analytical solver for evaluating design decisions, though it may not be a fair evaluator. This analytical solver is just a regular solver that uses the cost model to derive a solution for the closed form equation. The analytical tuner solves a much simpler linear integer programming problem compared to the harder linear program attempted using Bayesian Optimization (BO). The analytical solutions provided by the analytical solver are not ideal because floating point values cannot exist for inherently categorical parameters like size ratio and policy. However, by comparing the proposed solution to this ideal yet impractical model, a harder evaluation criteria is set, suggesting the solution may be stronger than experimental evaluations indicate.

Observations show that Upper Confidence Bound might perform as good as the

Expected Improvement acquisition function for the Classic LSM Design due to the simplicity of the Classic LSM Tree model making it easier to tune the hyper parameter β. This thesis experimented on the Classic LSM model with β values of 0.1, 0.2, and ultimately settled on 0.3 for effectively tuning the model's best observed evaluations. For more complex models, this hyper parameter became harder to tune, with UCB less likely to perform well.

A challenging aspect of this solution was selecting the abundant hyper parameters in Bayesian Optimization process. While previous research used BO, none calculated as many categorical attributes as attempted here, especially for K-LSM. It is unclear whether this problem is ill-suited for BO or if there was a shortcoming in accurately tuning the optimizer. Generally, the optimization worked but could not accurately predict a design after substantial time (a few hours), suggesting a disability of this method within the experimental scope.

However, harshly criticizing the method would be unfair given the small 20 design sample space used to learn the function's behavior. Simpler models like Classic LSM Tree and the Q-LSM design only needed to predict 3-5 decision parameters, while K-LSM Tree required predicting from 20 possible categorical values per level along with size ratios. BoTorch attempts this by considering all categorical attribute combinations, exponentially increasing choices. Historically, BO has struggled with categorical values and does not scale well, as stated in Literature.

Although potentially slower for very complex models with a very high categorical space, substantial proof exists that BO can perform better for regular Classic LSM designs and even more complex models like Fluid LSM and Q-LSM. For K-LSM, incrementally increasing values showed the acquisition function accurately predicts for upto 4 levels with K-values in this code. However, time constraints prevented producing accurate K-LSM results by running multiple loads.

Chapter 7

Related Work

The field of database tuning has been a topic of long research. Especially as new database storage structures develop, the need for tuning them becomes fundamental. This is because a storage structure when used in a database management system is only as good as the tuning applied to it. Many past research work on this subject has proved the monumental difference between the performance of a storage structure that has been tuned to harvest its optimal usage compared to the structure with only the default set. Database tuning has been an area of research even around 1998 where the necessity for tuning DBMSs for performance for the Web and other large-scale data stores was highlighted [40] .

This also introduced the concept of *self-tuning* databases that require no human interaction. The ultimate goal of autonomous database tuning has always been the same - to create a structure that is easy to use while being efficient for anyone without the need for in-depth knowledge in that specific field.

7.1 Overview

The first efforts in creating databases that can be self tuned or assess themselves can be dated back to as early as the 2000s. There have been efforts in tuning like RISC-style architecture [41] that talks about the *gain-pain ratio* and the importance of built-in self-assessment and auto-tuning capabilities. In reality, upon further exploration,

it can be found that this problem has existed since much earlier. It can be observed that the introduction of the *M&M* system that employs a feedback-based algorithm to determine the Multiprogramming Level (MPL) and memory settings for each class independently gives us the initial systems that employ a feedback mechanism that can be used to better our results [42].

Continuing these efforts, the Microsoft Research team detailed the work on self-tuning database systems over a period of ten years and assessed the challenges and solutions related to automated database design using data analysis and mining techniques [43]. There are other papers which detail the contributions in the field of self-tuning database which gives a review and assesses the advancements in this field [44]. Although, these works introduced the idea and the basic implementation approaches for such database systems, the actual implementation of robust self-tuned databases came much later.

7.2 Current Research

The reality is that the fundamental problem of tuning any database given a specific workload such that its performance is optimized is actually a NP-hard problem [45]. Most databases today are still tuned by specialized DBAs, who possess the technical expertise to tune the database manually. Therefore this field still remains an area of active research and multiple efforts have been made to use automatic tuning for this reason. This consistent effort is evident in multiple research papers in this field [46, 47, 48, 39, 49, 50, 51]. The insight gained from analyzing the current field of active research towards database optimization seems to be using two main technologies - the first using Bayesian Optimization and the second using Reinforcement Learning. The next section discusses the contributions in both these fields.

7.2.1 Bayesian optimization based optimization

In recent research BO has been a predominant method used for auto tuning databases. One such system called *Ottertune* provides an automated approach for tuning Database Management System (DBMS) using supervised and unsupervised machine learning methods [39]. It does this by selecting the most impactful knobs, then mapping new workloads to previously observed workloads and providing recommendations using a combination of this information. The Ottertune system makes use of the *Lasso* algorithm. The fundamental issue observed in this method is that it depends on historical data for the accuracy of its predictions.

The *iTuned* system is an automated tool that uses *Adaptive sampling* to detect *high impact parameters* and their optimal values [52]. It is specifically useful because it can be used across different database systems. iTuned uses vanilla BO models to search for a good configuration and then uses *Latin Hypercube Sampling (LHS)* [53] for initialization. It can be observed that iTuned works for a specific workload and the entire tuning process has to be redone when the workload changes. Even though, the proposed research implements tuning for a specific workload as well, it can be observed that this process is relatively simple and only takes a matter of minutes to be able to predict a *good* workload even for a high complexity model like the Fluid LSM.

The next system of particular interest is *Tuneful*, which addresses the issue of static workloads and introduces a system of incremental configuration tuning by using machine learning methods [54]. The paper employs a similar workload comparison as in *Ottertune* to provide a measure of *similarity* between two workloads. Tuneful uses Gini score as a measurement to gradually decrease the configuration space.

RelM introduces a modification of the black-box BO and instead uses a white box algorithm for memory allocation with significant lower overheads compared to

the black-box counterparts [55]. The focus of this paper is on applications running on modern distributed data processing systems. The *CGPTuner* introduces an auto-tuning framework based on *Contextual Gaussian Process Bandit Optimization (CGPBO)* that adapts to changing workloads [56]. CGPTuner effectively balances resource utilization and performance requirements and uses an external workload characterization module for its optimization process. This section finally discusses the observations made in *ResTune* which focuses on resource utilization for cloud databases [57]. This system introduces meta-learning for the optimization process along with constraints that it has to abide by provided by SLAs that it agrees on beforehand. It uses a constrained Bayesian optimization solver and uses an ensemble framework - RGPE.

The major issues in these systems stem from the fact that Bayesian Optimization processes do not handle heterogeneous values well and observe that most evaluations that use BO do not possess many categorical attributes. The proposed solution in this thesis handles these categorical values by using a combined kernel that can handle both continuous and categorical values. The other issue that can be observed is that as the optimization space increases, Bayesian optimization techniques tend to suffer from over over exploration as detailed in the paper [58]. This thesis approaches this problem by generally keeping our optimization space small - within the recommended 20 features and only exceeds the limit for the specific K-LSM model. Moreover, vanilla Bayesian optimization techniques assume a natural ordering of input value as stated in [59] which BoTorch fundamentally addresses by treating heterogeneous and continuous attributes separately and not assuming any order on the data.

7.2.2 Reinforcement Learning based optimization

The RL based systems use specific reinforcement learning algorithm called Deep Deterministic Policy Gradient (DDPG) [60]. The advantage of DDPG is that it can set a knob to any value within a fixed range even in a continuous space. DDPG does this by using two neural networks which uses an actor and a critic. The actor is used to select a configuration based on the input states. This is called the actor's *action*. The second part is the critic that is used for evaluation of the action by using a reward system. This thesis uses the support of the results observed in [61]. *CDBTune* provides an end-to-end automatic cloud database tuning system using deep reinforcement learning [62]. It uses DDPG for the high dimensional continuous parameters and directly optimizes from the input state comprising the database metrics to the output state which is the configuration settings using by representing the tuning process as a Markov Decision Process (MDP) [63]. This improves efficiency of the entire tuning process.

The next RL based tuner is *QTune* that uses Deep Reinforcement Learning to automatically tune database configurations [64]. QTune works specifically on SQL. It uses the actor-critic of DDPG with Deep RL for tuning based on the current database state and properties of incoming queries. QTune provides the user with three granularity options for tuning - at the query, workload and cluster level. This not only provides flexibility while tuning but also helps to balance latency and throughput. QTune does not have to be reconfigured every time workload and system hardware changes.

The issue with most RL-based systems is the requirement of a large training set to be able to build the model. Also the configurations generally provided are expensive to evaluate. To alleviate the cost of having to rerun each design decision, most methods focus on building complex systems that are fit for a range of workloads

and systems instead of selecting a design that is specifically suited for the current condition set. The proposed solution solves this problem by using a tuning technique that is extremely light weight. Even though the entire process of tuning is required for a change in the hardware system or workload, the entire tuning process can be completed in a few minutes as opposed to a few hours to several days as demonstrated in several papers. Moreover, the standardisation of a fixed cost function helps us avoid errors while evaluation. The solution proposed also avoid the computational expense required to train complex models since the BoTorch module used is comparatively lightweight.

Chapter 8

Future Work

This thesis lays the groundwork for using the BoTorch library for addressing the tuning challenges of the basic LSM Tree structure and some of the hybrid models. However, it should be noted that there is a lot of scope for further improvement in this field. Specifically, our future research aims to be able to tune the K-LSM variant without the need for constraints. This could be achieved by modifying the categorical kernel function which currently evaluates each combination of possible categorical values leading to exponential number of evaluations as the number of levels in the LSM Tree increases.

Addressing the limitations of this study, particularly evaluating and handling categorical attributes is a crucial step in future work and one of the major setbacks faced while working on this research. While this thesis acknowledges that Bayesian optimization is more catered towards continuous values[65], it can be observed that there is potential to better tune the LSM Tree even with it's mixed design space. The main advantages observed of using the Bayesian Optimization process via the BoTorch library is the ease of use and the requirement of very little training data as well as a very fast tuning process time. In accordance with this, training and predicting results for our model is quite fast requiring much lesser time than state of the art RL models. It is important to note, that it is possible that Upper Confidence Bound could work better than Expected Improvement when the correct hyper parameter value for β is

selected for the acquisition function. If the β value is balanced, the UCB acquisition function will be equally balanced between exploitation and exploration.

In future work, the proposed solution could be compared to the results observed by using various other Bayesian optimization technique implementations - especially frameworks like MLOS which have proved to be successful in the domain of database tuning and optimization. The MLOS design paradigm provides an interesting option for tuning LSM Trees. Underneath the MLOS framework, a SMAC optimizer is used which can be beneficial for categorical attributes as it uses a random forest based optimizer for its optimization process. BoTorch in itself lays the foundation for a much more flexible implementation where our limitations can be addressed by using a custom acquisition function that is better suited for our design space. This thesis also acknowledges the possibility of further experimentation, which might lead to better results than the ones proposed in this research.

In conclusion, while this thesis marks a significant step towards understanding the ability of BoTorch to be able to tune a database structure, it also provides the opportunity to explore new tools that could potentially lead to a major breakthrough in the process of automating new and emerging data structures for databases.

Chapter 9

Conclusion

This thesis presents an innovative addition to the already active world of automated database tuning. The focus of this work is on the Log Structured Merge Tree (LSM Tree) by using Bayesian Optimization (BO). The method focuses on not only tackling the basic LSM Tree structure, but other complex hybrid forms of the classic tree model like the Fluid LSM from *Doestoevsky* [28], the K-LSM from *Endure* [34] and the Q-LSM (a simplified version of the K-LSM tree). This thesis improvises the normal Bayesian optimization process by using a detailed cost model as the objective function. The lightweight and fast nature of the tuning process enables efficient design forecasts despite changing workloads. Although conventional database tuning has existed for a long time, applying Bayesian optimization to the LSM Tree structure especially for the hybrid tree models has not yet been explored. The sophisticated cost model from Endure is utilized to address the limitation of high run times under varying workloads faced by other approaches - a natural problem in database tuning. The main challenges attempted to be addressed are approaching a complex heterogeneous tuning space that not only has different sets of parameters for different models, but also has the issue of having to handle both categorical and continuous attributes. This problem is overcome by using a hybrid kernel to handle each space separately. This allows navigating the complex space of the hybrid structures and highlights the flexibility and efficacy of the solution.

This work consolidates multiple LSM Tree structures and demonstrates the efficacy of the solution approach for automated database tuning. The motivation for this research is to extend the concept of database tuning towards autonomous database systems that can self-optimize in real-time intelligently without human intervention and are able to accommodate changes to parameters that can and will change like hardware specifications and workload types. The prospect of being able to auto tune is particularly attractive for hybrid LSM structures which due to their flexibility are also significantly more complex to tune through traditional and user dependent methods. This thesis ultimately hopes to be the initial step towards the exploration of completely auto-tuned systems that can scale and adapt to a world of ever-increasing data.

Bibliography

[1] E. F. Codd, "A relational model of data for large shared data banks," *Communications of the ACM*, vol. 13, no. 6, pp. 377–387, 1970. [Online]. Available: https://www.seas.upenn.edu/~zives/03f/cis550/codd.pdf

[2] PostgreSQL Global Development Group, "Postgresql," https://www.postgresql.org, 2 2024, version 16.2.

[3] "Mysql," MySQL Website, 2024, accessed: 2024-04-10. [Online]. Available: https://www.mysql.com/

[4] International Organization for Standardization, "Sql:2023," ISO/IEC JTC 1/SC 32/WG 3, 2023, accessed: 2024-04-10. [Online]. Available: https://www.iso.org/standard/76583.html

[5] MongoDB Inc., *MongoDB*, 2023, accessed: 2024-04-10. [Online]. Available: https://www.mongodb.com/

[6] "Apache Cassandra," http://cassandra.apache.org, accessed: 2024.

[7] "Redis," https://redis.io/.

[8] Amazon Web Services, Inc., "Amazon dynamodb," https://aws.amazon.com/dynamodb/, 2024, accessed: 2024-04-10.

[9] The Apache Software Foundation, "Apache couchdb," https://couchdb.apache.org/, 2024, accessed: 2024-04-10.

[10] Google Cloud, "Cloud bigtable," https://cloud.google.com/bigtable, 2024, accessed: 2024-04-10.

[11] Neo4j, Inc., "Neo4j graph database," https://neo4j.com/, 2024, accessed: 2024-04-10.

[12] Amazon Web Services, Inc., "Amazon neptune," https://aws.amazon.com/neptune/, 2024, accessed: 2024-04-10.

[13] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree (lsm-tree)," *Acta Inf.*, vol. 33, no. 4, p. 351–385, jun 1996. [Online]. Available: https://doi.org/10.1007/s002360050048

[14] R. Bayer and E. M. McCreight, "Organization and maintenance of large ordered indexes," *Acta Informatica*, vol. 1, no. 3, pp. 173–189, 1972.

[15] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, p. 422–426, jul 1970. [Online]. Available: https://doi.org/10.1145/362686.362692

[16] Wikipedia contributors, "Bloom filter — Wikipedia, the free encyclopedia," 2021, [Online; accessed 8-March-2024]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Bloom_filter

[17] M. Balandat, B. Karrer, D. R. Jiang, S. Daulton, B. Letham, A. G. Wilson, and E. Bakshy, "Botorch: A framework for efficient monte-carlo bayesian optimization," 2020. *NeurIPS Proceedings*, 33. https://proceedings.neurips.cc/paper_files/paper/2020/file/f5b1b89d98b7286673128a5fb112cb9a-Paper.pdf

[18] C. Luo and M. J. Carey, "Lsm-based storage techniques: A survey," *CoRR*, vol. abs/1812.07527, 2018. [Online]. Available: http://arxiv.org/abs/1812.07527

[19] N. Dayan, M. Athanassoulis, and S. Idreos, "Optimal bloom filters and adaptive merging for lsm-trees," *ACM Trans. Database Syst.*, vol. 43, no. 4, dec 2018. [Online]. Available: https://doi.org/10.1145/3276980

[20] S. Dong, M. Callaghan, L. Galanis, D. Borthakur, T. Savor, and M. Strum, "Optimizing space amplification in rocksdb," in *Proceedings of the 8th Bien-nial Conference on Innovative Data Systems Research (CIDR '17)*, Chaminade, California, USA, January 8-11 2017, this article is published under a Creative Commons Attribution License (http://creativecommons.org/licenses/by/3.0/), which permits distribution and reproduction in any medium, as well allowing derivative works, provided that you attribute the original work to the author(s) and CIDR 2017.

[21] N. Dayan, M. Athanassoulis, and S. Idreos, "Monkey: Optimal navigable key-value store," in *Proceedings of the 2017 ACM International Conference on Management of Data*, ser. SIGMOD '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 79–94. [Online]. Available: https://doi.org/10.1145/3035918.3064054

[22] Facebook, "Rocksdb," https://github.com/facebook/rocksdb, 2021, accessed: March 27, 2024.

[23] W. Zhong, C. Chen, X. Wu, and S. Jiang, "REMIX: Efficient range query for LSM-trees," in *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, Feb. 2021, pp. 51–64. [Online]. Available: https://www.usenix.org/conference/fast21/presentation/zhong

[24] S. Luo, S. Chatterjee, R. Ketsetsidis, N. Dayan, W. Qin, and S. Idreos, "Rosetta: A robust space-time optimized range filter for key-value stores," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 2071–2086. [Online]. Available: https://doi.org/10.1145/3318464.3389731

[25] H. Zhang, H. Lim, V. Leis, D. G. Andersen, M. Kaminsky, K. Keeton, and A. Pavlo, "Succinct range filters," *Commun. ACM*, vol. 64, no. 4, p. 166–173, mar 2021. [Online]. Available: https://doi.org/10.1145/3450262

[26] H. Zhang, H. Zhang, V. Leis, D. G. Andersen, M. Kaminsky, K. Keeton, and A. Pavlo, "Succinct rangefilters," *ACM Transactions on Database Systems (TODS)*, vol. 45, no. 2, pp. 5:1–5:31, 2020.

[27] S. Sarkar and M. Athanassoulis, "Dissecting, designing, and optimizing lsm-based data stores," in *Proceedings of the 2022 International Conference on Management of Data*, ser. SIGMOD '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 2489–2497. [Online]. Available: https://doi.org/10.1145/3514221.3522563

[28] N. Dayan and S. Idreos, "Dostoevsky: Better space-time trade-offs for lsm-tree based key-value stores via adaptive removal of superfluous merging," in *Proceedings of the 2018 International Conference on Management of Data*, ser. SIGMOD '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 505–520. [Online]. Available: https://doi.org/10.1145/3183713.3196927

[29] A. Huynh, H. A. Chaudhari, E. Terzi, and M. Athanassoulis, "Towards flexibility and robustness of lsm trees," 2023. https://arxiv.org/abs/2311.10005

[30] D. Mo, F. Chen, S. Luo, and C. Shan, "Learning to optimize lsm-trees: Towards a reinforcement learning based key-value store for dynamic workloads," *Proc. ACM Manag. Data*, vol. 1, no. 3, nov 2023. [Online]. Available: https://doi.org/10.1145/3617333

[31] J. Snoek, H. Larochelle, and R. P. Adams, "Practical bayesian optimization of machine learning algorithms," 2012. https://arxiv.org/abs/1206.2944

[32] A. Slivkins, "Introduction to multi-armed bandits," 2024.

[33] P. Auer, "Using confidence bounds for exploitation-exploration trade-offs," *J. Machine Learning Research*, vol. 3, no. null, p. 397–422, mar 2003.

[34] A. Huynh, H. A. Chaudhari, E. Terzi, and M. Athanassoulis, "Endure: A robust tuning paradigm for lsm trees under workload uncertainty," 2021.

[35] S. Tarkoma, C. E. Rothenberg, and E. Lagerspetz, "Theory and practice of bloom filters for distributed systems," *IEEE Communications Surveys & Tutorials*, vol. 14, no. 1, pp. 131–155, 2012.

[36] J. Snoek, H. Larochelle, and R. P. Adams, "Practical bayesian optimization of machine learning algorithms," 2012. https://arxiv.org/abs/1206.2944

[37] R. Martinez-Cantin, "Bayesopt: A bayesian optimization library for nonlinear optimization, experimental design and bandits," 2014.

[38] K. Kanellis, C. Ding, B. Kroth, A. Müller, C. Curino, and S. Venkataraman, "Llamatune: Sample-efficient dbms configuration tuning," 2022.

[39] D. Van Aken, A. Pavlo, G. J. Gordon, and B. Zhang, "Automatic database management system tuning through large-scale machine learning," in *Proceedings of the 2017 ACM International Conference on Management of Data*, ser. SIGMOD '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 1009–1024. [Online]. Available: https://doi.org/10.1145/3035918.3064029

[40] P. Bernstein, M. Brodie, S. Ceri, D. DeWitt, M. Franklin, H. Garcia-Molina, J. Gray, J. Held, J. Hellerstein, H. V. Jagadish, M. Lesk, D. Maier, J. Naughton, H. Pirahesh, M. Stonebraker, and J. Ullman, "The asilomar report on database research," 1998. https://arxiv.org/abs/cs/9811013

[41] S. Chaudhuri and G. Weikum, "Rethinking database system architecture: Towards a self-tuning risc-style database system," *Proceedings of the 26th International Conference on Very Large Data Bases, VLDB'00*, 12 2000.

[42] K. P. Brown, M. Mehta, M. J. Carey, and M. Livny, "Towards automated performance tuning for complex workloads," in *Proceedings of the 20th International Conference on Very Large Data Bases*, ser. VLDB '94. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1994, p. 72–84.

[43] S. Chaudhuri and V. Narasayya, "Self-tuning database systems: a decade of progress," in *Proceedings of the 33rd International Conference on Very Large Data Bases*, ser. VLDB '07. VLDB Endowment, 2007, p. 3–14.

[44] G. Weikum, A. Mönkeberg, C. Hasse, and P. Zabback, "Self-tuning database technology and information services: from wishful thinking to viable engineering," *28th International Conference on Very Large Databases (VLDB 2002), Morgan Kaufmann, 20-31 (2002)*, 08 2002.

[45] D. G. Sullivan, M. I. Seltzer, and A. Pfeffer, "Using probabilistic reasoning to automate software tuning," *SIGMETRICS Perform. Eval.*

Rev., vol. 32, no. 1, p. 404–405, jun 2004. [Online]. Available: https://doi.org/10.1145/1012888.1005739

[46] S. Agrawal, S. Chaudhuri, L. Kollár, A. P. Marathe, V. R. Narasayya, and M. Syamala, "Database tuning advisor for microsoft SQL server 2005," in *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, VLDB 2004, Toronto, Canada, August 31 - September 3 2004*, M. A. Nascimento, M. T. Özsu, D. Kossmann, R. J. Miller, J. A. Blakeley, and K. B. Schiefer, Eds. Morgan Kaufmann, 2004, pp. 1110–1121. [Online]. Available: https://doi.org/10.1016/B978-012088469-8.50097-8

[47] S. Chaudhuri and G. Weikum, "Foundations of automated database tuning," in *Proceedings of the 32nd International Conference on Very Large Data Bases*, ser. VLDB '06. VLDB Endowment, 2006, p. 1265.

[48] J. Kossmann and R. Schlosser, "Self-driving database systems: A conceptual approach," *Distributed and Parallel Databases*, vol. 38, 12 2020.

[49] D. Shasha and P. Bonnet, *Database tuning: principles, experiments, and troubleshooting techniques*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002.

[50] *VLDB '92: Proceedings of the 18th International Conference on Very Large Data Bases*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1992.

[51] A. J. Storm, C. Garcia-Arellano, S. S. Lightstone, Y. Diao, and M. Surendra, "Adaptive self-tuning memory in db2," in *Proceedings of the 32nd International Conference on Very Large Data Bases*, ser. VLDB '06. VLDB Endowment, 2006, p. 1081–1092.

[52] S. Duan, V. Thummala, and S. Babu, "Tuning database configuration parameters with ituned." *PVLDB*, vol. 2, pp. 1246–1257, 08 2009.

[53] M. D. McKay, "Latin hypercube sampling as a tool in uncertainty analysis of computer models," in *Proceedings of the 24th Conference on Winter Simulation*, ser. WSC '92. New York, NY, USA: Association for Computing Machinery, 1992, p. 557–564. [Online]. Available: https://doi.org/10.1145/167293.167637

[54] A. Fekry, L. Carata, T. Pasquier, A. Rice, and A. Hopper, "To tune or not to tune? in search of optimal configurations for data analytics," in *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, ser. KDD '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 2494–2504. [Online]. Available: https://doi.org/10.1145/3394486.3403299

[55] M. Kunjir and S. Babu, "Black or white? how to develop an autotuner for memory-based analytics," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1667–1683.[Online]. Available: https://doi.org/10.1145/3318464.3380591

[56] S. Cereda, S. Valladares, P. Cremonesi, and S. Doni, "Cgptuner: a contextual gaussian process bandit approach for the automatic tuning of it configurations under varying workload conditions," *Proc. VLDB Endow.*, vol. 14, no. 8, p. 1401–1413, apr 2021. [Online]. Available: https://doi.org/10.14778/3457390.3457404

[57] X. Zhang, H. Wu, Z. Chang, S. Jin, J. Tan, F. Li, T. Zhang, and B. Cui, "Restune: Resource oriented tuning boosted by meta-learning for cloud databases," in *Proceedings of the 2021 International Conference on Management of Data*, ser. SIGMOD '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 2102–2114. [Online]. Available: https://doi.org/10.1145/3448016.3457291

[58] B. Shahriari, K. Swersky, Z. Wang, R. Adams, and N. De Freitas, "Taking the human out of the loop: A review of bayesian optimization," *Proceedings of the IEEE*, vol. 104, no. 1, pp. 148–175, Jan. 2016, publisher Copyright: © 1963-2012 IEEE.

[59] X. Wan, V. Nguyen, H. Ha, B. Ru, C. Lu, and M. A. Osborne, "Think global and act local: Bayesian optimisation over high-dimensional categorical and mixed search spaces," 2021. https://arxiv.org/abs/2102.07188

[60] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," 2019.

[61] X. Zhang, Z. Chang, Y. Li, H. Wu, J. Tan, F. Li, and B. Cui, "Facilitating database tuning with hyper-parameter optimization: a comprehensive experimental evaluation," *Proc. VLDB Endow.*, vol. 15, no. 9, p. 1808–1821, may 2022. [Online]. Available: https://doi.org/10.14778/3538598.3538604

[62] J. Zhang, Y. Liu, K. Zhou, G. Li, Z. Xiao, B. Cheng, J. Xing, Y. Wang, T. Cheng, L. Liu, M. Ran, and Z. Li, "An end-to-end automatic cloud database tuning system using deep reinforcement learning," in *Proceedings of the 2019 International Conference on Management of Data*, ser. SIGMOD '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 415–432.[Online]. Available: https://doi.org/10.1145/3299869.3300085

[63] R. Bellman, "A markovian decision process," *Journal of Mathematics and Mechanics*, vol. 6, pp. 679–684, 1957. [Online]. Available: https: //doi.org/10.1512/iumj.1957.6.56038

[64] G. Li, X. Zhou, S. Li, and B. Gao, "Qtune: a query-aware database tuning system with deep reinforcement learning," *Proc. VLDB Endow.*, vol. 12, no. 12, p. 2118–2130, aug 2019. [Online]. Available: https: //doi.org/10.14778/3352063.3352129

[65] P. I. Frazier, "A tutorial on bayesian optimization," 2018. https://arxiv.org/abs/1807.02811

CURRICULUM VITAE