DI2

FYP Final Report

Learned Indexes for Commercial Databases

Presented By
LEE, Chui Shan
MO, Ka Lok
TSE, Ngo Chun
YIP, Tsz Hin

DI2

Advised by
Prof. Dimitris Papadias,
Mr. Max Prior

Submitted in partial fulfillment
of the requirement for COMP 4981
in the
Department of Computer Science and Engineering
The Hong Kong University of Science and Technology
2023-2024
Date of Submission: 18th Apr 2024

Abstract

The rapid advancement of artificial intelligence and deep learning has significantly impacted various industries and domains. One such domain is the field of databases, where researchers are actively exploring innovative ways to improve the query and write speed. One promising approach is to integrate learning algorithms into indexes (learned indexes). Moreover, the emergence of big data raises the need of high frequency data storage and retrieval, fostering the development of approaches that can increase data throughput of the databases. In recent literature, learned index algorithms and structures have been proposed.

In this project, we have designed and implemented a learned index called Sherry in RocksDB utilizing error-bounded Piecewise Linear Regression (PLR) algorithms. After successfully migrating Sherry into existing RocksDB, we run the stress test and performance benchmark under the newly proposed algorithms. Sherry outperforms the conventional index by saving up to 30% of space.

Table of Contents

List of Figures

List of Tables

1. Introduction

1.1 Overview

Indexing is used prevalently in databases to enhance query performances by avoiding a full table scan. An index is a data structure that organizes and stores data in a specific way that enables efficient searching and retrieval operations. In the context of disk-resident indexes, they improve the I/O cost of data retrieval by reducing the number of disk page reads and writes. Conventional indexes, such as B+ Tree [2], usually adopt a hierarchical structure where internal nodes are index nodes used to guide the search, and leaf nodes are data nodes storing data or pointers to data. With the properties of a tree structure, the number of disk accesses can be reduced significantly to a logarithmic scale.

Despite the performance enhancement, traditional indexes require additional space, and their structures and search algorithms are not tailored to the data distribution, implying that there is still room for improvement in optimizing query performance. With the advancement of machine learning, numerous learned indexes have been proposed and gained considerable interest in the research community, aiming to tackle the problems. In most cases, learned indexes are organized in a tree structure and utilize machine learning models, usually linear models and linear regression models, within their nodes to reduce query time and index size.

Currently, most learned indexes proposed are designed to reside in the main memory. Yet, databases typically store a large amount of data, usually up to millions of entries. It is, therefore, impossible for databases to load all data into the main memory, as the storage cost in memory is significantly higher than in secondary storage devices. This reveals the obstacles to employing learned indexes in production: (1) The majority of the learned indexes proposed are memory-resident, and (2) learned indexes have not been implemented into any commercial database.

In our work, we designed a space-efficient, disk-resident learned index, Sherry, and implemented the index on RocksDB.

1.2 Background

1.2.1 LSM Tree in RocksDB

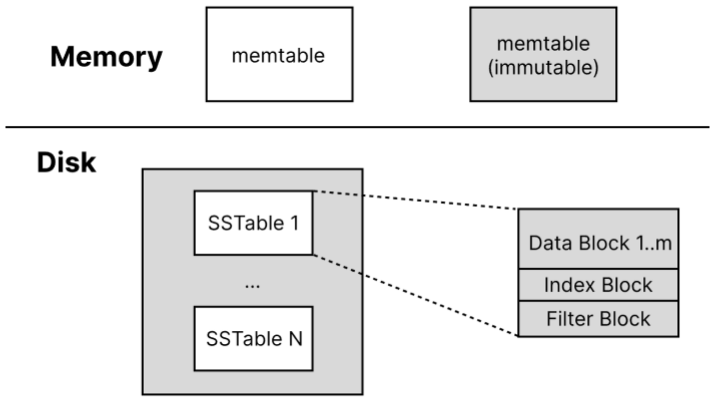


Figure 1: Structure Overview of LSM Tree in RockDB

RocksDB provides key-value pair storage and uses LSM Tree for data storage. Keys and values are arbitrary byte arrays, where keys are ordered within the key-value store according to a user-specified comparator function.

LSM Tree [5] consists of two important structures, *memtable* and *SSTable*, which are stored in memory and disk respectively. Figure 1 shows the overall structure of the LSM Tree. All newly inserted or updated data is first inserted into the memtable. When the memtable exceeds the size threshold, it becomes immutable, and its data is flushed into the disk for building the SSTable, where the data flushed will be sorted in the SSTable. Meanwhile, another memtable is created and used to handle all possible writes and updates during the flushing process. On the disk, SSTables are organized in several levels. Generally, data will flush to Level 0, then perform compaction for 2 SSTable at Level i to 1 SSTable at Level (i+1). More information could be referred to Figure 2.

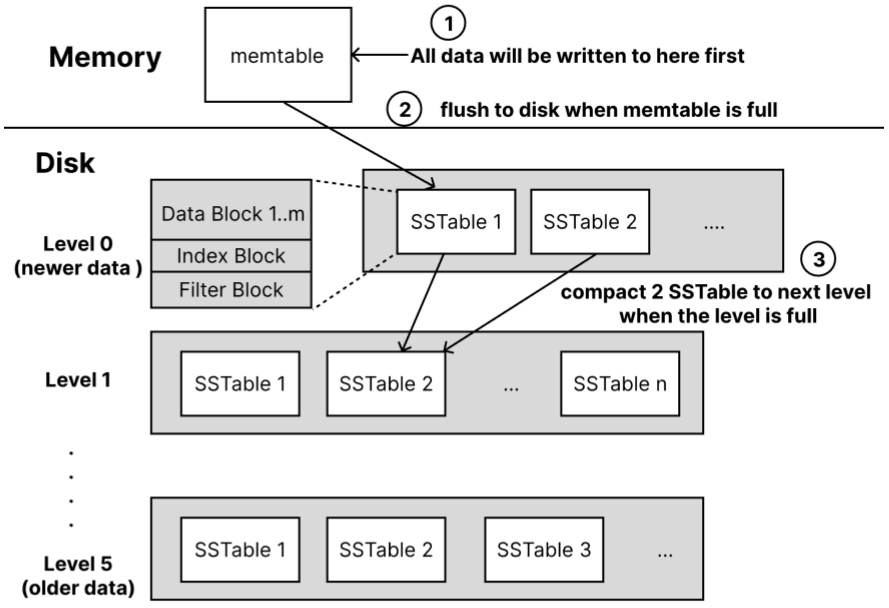


Figure 2: An example flow of writing a piece of data to LSM Tree

In write operations, all data written in the LSM Tree will **(1)** first go to the memtable. **(2)** If the memtable is full, RocksDB will flush the data to the disk and build an SSTable, which is a node in the LSM Tree. **(3)** If the number of SSTables is high in a certain level, a compaction process begins and merges two or more SSTables to build a larger SSTable.

To read a value from a key, the LSM Tree will **(1)** check whether the key exists in the memtable. **(2)** If not, it will access SSTable in each level to find the first occurrence of the value. **(3)** After searching for all levels, report no such an entry if not found.

1.2.2 SSTables in LSM Tree

Each node in the LSM Tree is an SSTable, a structure for storing data in disk properly. Each SSTable is immutable after created and contains several important components. Each component is dividend by blocks and written sequentially, as shown in Figure 3. Table 1 explains the usage of each block in the SSTable.

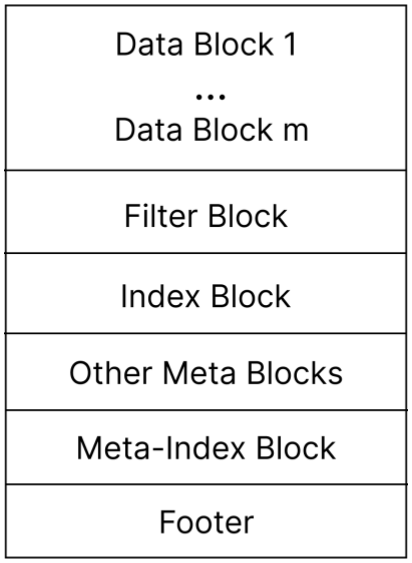


Figure 3: Structure of a typical SSTable

Block Type	Usage
Data Block	Area for data storage. All key-value pairs are stored in a sorted order of key.
Filter Block	Store a bloom filter which can determine whether a given key is resided in this SSTable.
Index Block	A meta block which stores the key range in each data block. It is used to improve the query time by locating the data block that may contain the query data.
Meta Blocks	Store other metadata about the SSTable.
Footer	A fixed constant for verification of integrity of this SSTable.

Table 1: Functions of different blocks in SSTable

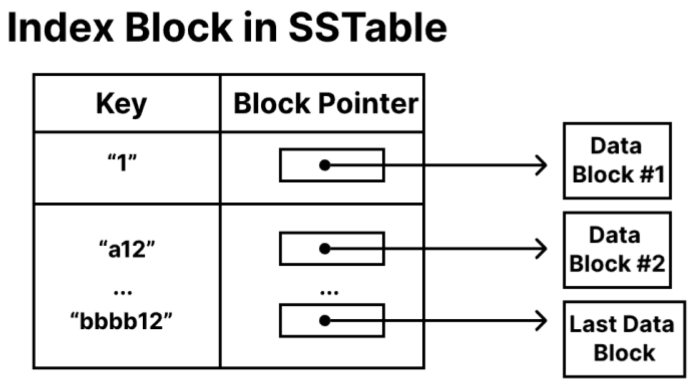


Figure 4: Overview of the Index Block Structure

The index block in an SSTable stores a key-value mapping *<key, block pointer>*. During the building process, all last keys of data blocks and the data block handles are collected. To locate the target data block of a query key, the index block performs a binary search internally to find the target data block as the key is sorted across the whole SSTable.

An SSTable is created in the following steps: **(1)** Block writes to the data block in a sorted order. **(2)** Build other blocks, such as the filter block and index block, after writing all data to the data block. Data blocks are partitioned approximately by a predetermined constant block size. **(3)** Assign the SSTable to the best level possible in the LSM-tree.

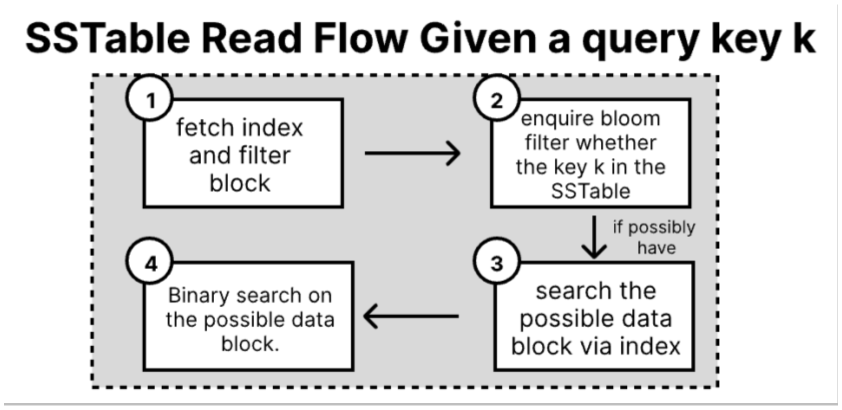
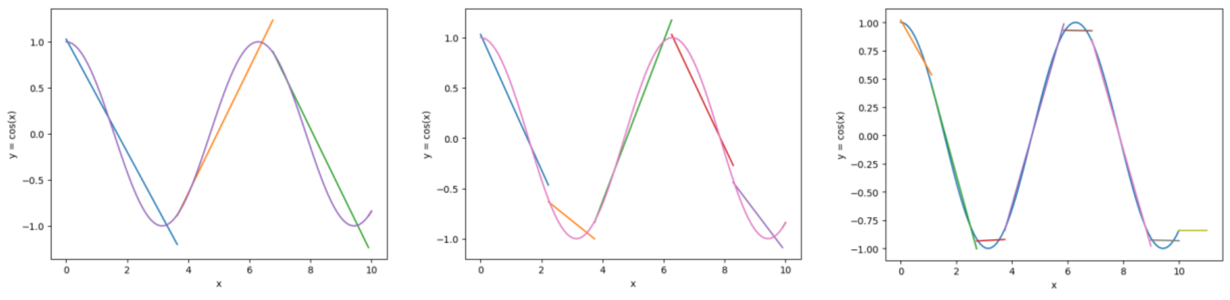

SSTable Read Flow Given a query key k
1
fetch index and filter block
2
enquire bloom filter whether the key k in the SSTable
if possibly have
3
search the possible data block via index
4
Binary search on the possible data block.

Figure 5: Read Flow of a SSTable

To read a piece of data from an SSTable, **(1)** it first fetches the filter block and index block. **(2)** it consults the filter block to determine whether the data is resided on this SSTable given a query key. **(3)** It then consults the index block to get the target data block using binary search. **(4)** Fetch the target data block and find the data using binary search with the query key.

1.2.3 Previous Work: Error-bounded Piecewise Linear Regression (PLR)

The error-bounded Piecewise Linear Regression (PLR) is a deterministic learning algorithm. It tries to find a piecewise linear function that approximates each point with a fixed error-bound ε. After training, the algorithm can be characterized by an array of segments. Using the model, all trained data points can be inferred within the range of $[actual - \epsilon, actual + \epsilon]$.



$\epsilon = 0.5$ $\epsilon = 0.25$ $\epsilon = 0.1$

Figure 6: Examples of PLR with different error-bound values

There are two versions of error-bound PLR Algorithms proposed by [10], Greedy PLR and Optimal PLR. Greedy PLR uses a greedy algorithm to determine the line segments while it is computationally lighter; Optimal PLR outputs optimal line segments while it is computationally heavy.

Algorithm 1.2.3.1: Greedy Error-bounded PLR.

Input: An array of data points dps, where each element is in the form (x, y). The model error-bound $gamma$.

Output: An array of segments $segs$ which fits all dps within the error-bound $gamma$.

```
1:  Create an empty array $segs$.
2:  Initialize $slope$ and $y\_offset$ for the current segment.
3:  Initialize $x\_start \leftarrow dps[0]$.
4:  for each data point $dp$ in $dps$ do
5:      Tune $slope$ and $y\_offset$ for the current segment, such that all trained data points fit on the current line segment.
6:      if tunning is impossible then
7:          Add the current line segment $seg \coloneqq < x\_start, slope, y\_offset >$ into $segs$.
8:          Record $x\_start \leftarrow dp.x$ for the new line segment.
9:      end if
10: end for
11: Output $segs$.
```

The PLR Algorithm could fit perfectly with immutable structures like SSTables as it only needs to be trained once and is able to use the output segments in the whole lifecycle. As an SSTable stores the key range for each data block in the index block, it is possible that a single segment outputted from the PLR Algorithm could represent several conventional index block entries, reducing the space consumption of an index block. To ensure efficient write operation, it is preferrable to choose an algorithm which is computationally light and able to build the model on-demand while building an SSTable.

Even though PLR Model fits in SSTables and LSM Tree, there are several limitations: **(1)** All data must be represented as an arithmetic representation. As typical index uses string representation, the PLR Algorithm imposes an arithmetic upper-bound of the index. Also, the index will be larger if we would like to expand the maximum representation bit. **(2)** It requires the consecutive data point to have different *x-values*. As the process will calculate slope, training the same *x-value* in consecutive data point will cause an issue of division by 0, hampering the model training process. **(3)** In terms of computational time, even though the Greedy PLR Algorithm is light in computational time, it is more expensive than conventional indexes that collect entries data and directly write to the index file.

1.2.4 Previous Work: Bourbon

We mainly reviewed Bourbon [3], which leverages the immutable data file of Log Structured Merge Tree (LSM Tree) to enable on-disk learned index storage and retrieval. Bourbon also utilizes the PLR Algorithm to learn the distribution of keys with respect to offsets in a file, resulting in reduced index size and faster learning speed.

Bourbon has two key features: **(1)** It modifies the access path of SSTables without accessing the conventional index block. Bourbon trains the model with the Greedy PLR Algorithm in one-pass using sorted *<key, offset_in_sstable>* pairs as data points. The resulting model could be inferred by inputting the *key* and accessing the precise address of the corresponding value in the SSTable. **(2)** It introduces a cost-benefit analyzer to determine whether to train a model or to use the conventional index during SSTable construction, to maximize the gain in access time.

Bourbon innovates and enables commercial databases to realize the potential time and space performance gain of learned index. It is inspirational for utilizing the immutable properties of SSTable and combining the PLR Algorithm into SSTable construction. Bourbon strikes a balance between the upfront cost of model training during SSTable construction and the potential improvement in read operations, resulting in boosting for overall read performance.

There are some concerns imposed by Bourbon that are worth addressing. Firstly, learned models are stored external to the index blocks in SSTables, not suiting the semantics of LSM Trees. When opening a database, Bourbon loads all on-disk learned models into the main memory, imposing an additional upfront cost as a result. In addition, Bourbon supports only unsigned integer as the key type, which can be improved by extending to other generic types, such as strings, in key-value stores. In our project, we have developed a learned index based on Bourbon and addressed these constraints. We utilize the PLR algorithm as our underlying learning algorithm and store the PLR segments into SSTables. We also employ a conversion scheme for string and unsigned integer keys.

1.3 Objectives

Our goal is to design a learned index based on Bourbon and implement it in RocksDB, which is a commercial, NoSQL, key-value database adopting the LSM Tree data structure for storage. Sherry supports read and write operations with comparable performance as conventional index. Furthermore, we aim to overcome some limitations in Bourbon, which are: **(1)** the learned index model parameters are stored outside SSTables, **(2)** they are loaded upfront but not on-demand, and **(3)** the key type is limited to unsigned integers.

Specifically, our objectives include:

(1) Designing a space-efficient learned index, such that less storage space is required for storing the learning index on the main memory and disk.

(2) Achieving comparable read and write performances against the conventional index in RocksDB without significant performance degradation.

(3) Integrating the learned index into SSTables to suit the semantics of LSM Tree and RocksDB.

(4) Supporting string as the key type to satisfy the requirement of RocksDB.

2. Design

Sherry utilizes the PLR algorithm to learn the distribution of data blocks. By using the PLR model parameters in index blocks, it eliminates the overhead cost of loading the learned models. Since the PLR algorithm represents data as lines and segments, our learned index potentially saves the overall storage space consumption with an efficient storage scheme.

2.1 Objectives

Our design replaces the original index block content in SSTables by the learned index representation and preserves the semantics of RocksDB, as depicted in Figure 7. As RocksDB is a complicated system, it is crucial to preserve the semantics for improving readability and maintainability.

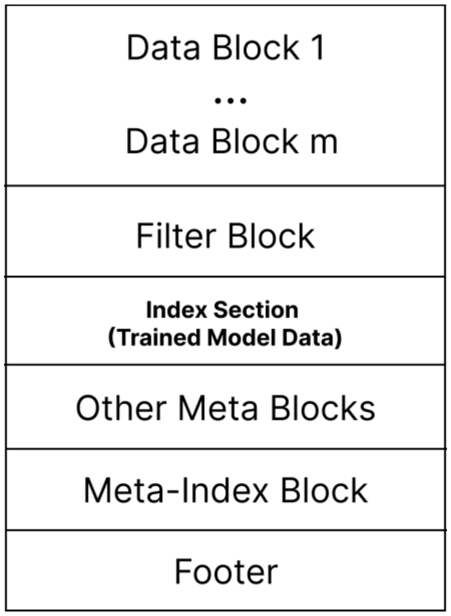


Figure 7: Structural Design of the SSTable

To achieve the index block replacement, it is vital to re-design the read and write process of the database to fit the new scheme. The following sections describe the read and write process involving Sherry, our learned index.

2.2 Data Storage Design

To store the data effectively under Sherry, we modify the index building procedure of an SSTable. In this design, all keys and the corresponding data block number are collected to train the model using PLR Algorithm.

As illustrated in Figure 8, the index building for Sherry consists of three steps: **(1)** Data Collection & Processing; **(2)** Model Training; **(3)** Model Encoding & Index Block Storage.

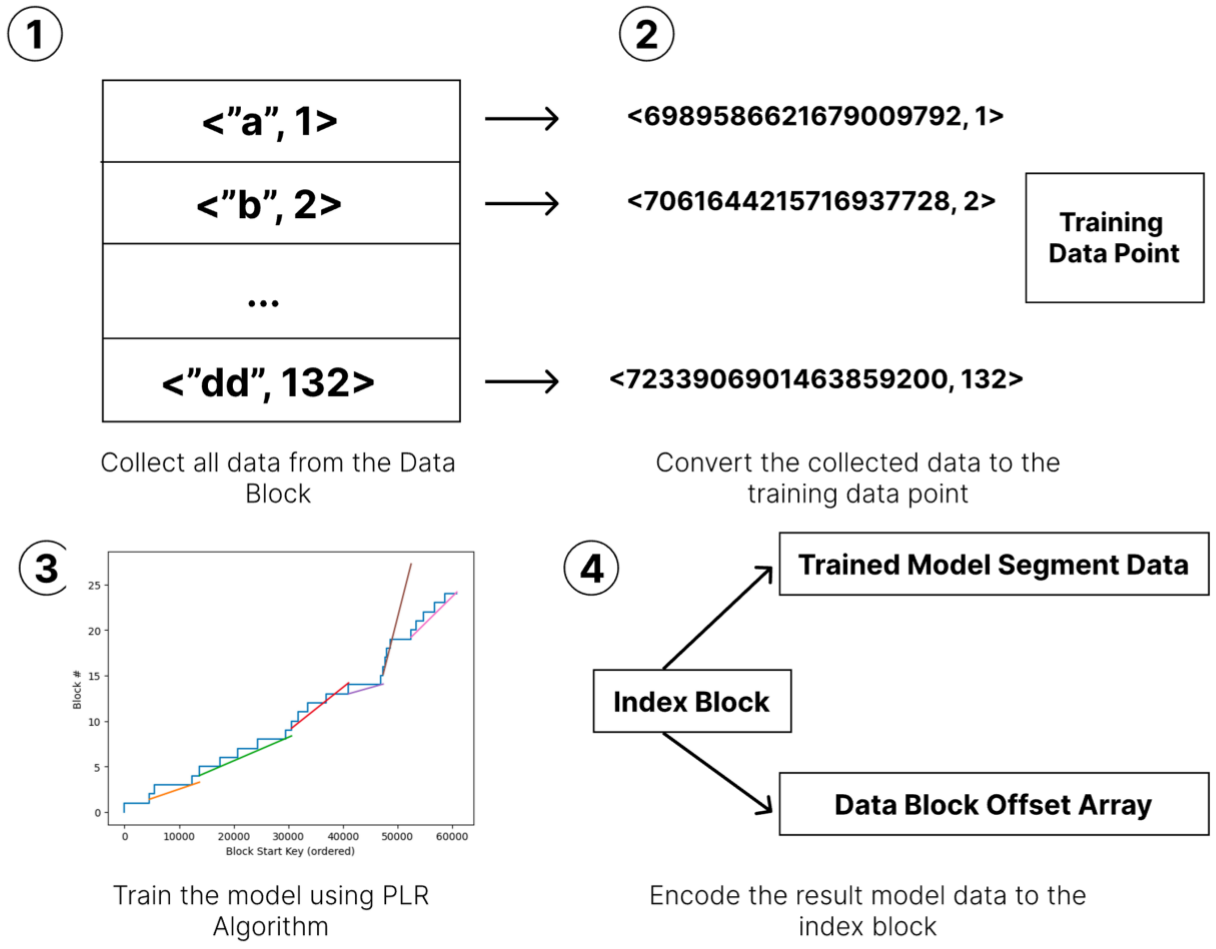

1
<"a", 1>
<"b", 2>
...
<"dd", 132>
Collect all data from the Data Block
2
<6989586621679009792, 1>
<7061644215716937728, 2>
<7233906901463859200, 132>
Training Data Point
Convert the collected data to the training data point
3
Block #
Block Start Key (ordered)
Train the model using PLR Algorithm
4
Index Block
Trained Model Segment Data
Data Block Offset Array
Encode the result model data to the index block

Figure 8: The design of the index building process

2.2.1 Data Collection & Processing

As required by the PLR Algorithm, we collect data points to train a model involving multiple segments. For every key written in the data block, we will collect and record its corresponding block number.

In RocksDB, keys are represented as strings while PLR Training algorithm requires arithmetic representation of data points. To accommodate the needs of the PLR algorithm, all keys will be converted to 64-bit unsigned integer (uint64_t), as illustrated in Figure 9. A one-to-one mapping conversion scheme is introduced to convert all keys within 8 bytes to a unique 64-bit unsigned integer. The key string will be converted directly to an unsigned integer with the same bit representation. Even though the scheme does not benefit from any form of encoding, it exhibits characteristics of one-to-one mapping and maintaining the sorted order of keys. Maintaining the sorted order is vital for the SSTable to perform efficient binary search in reading flow. It is also required by the PLR Algorithm to train in a greedy manner.

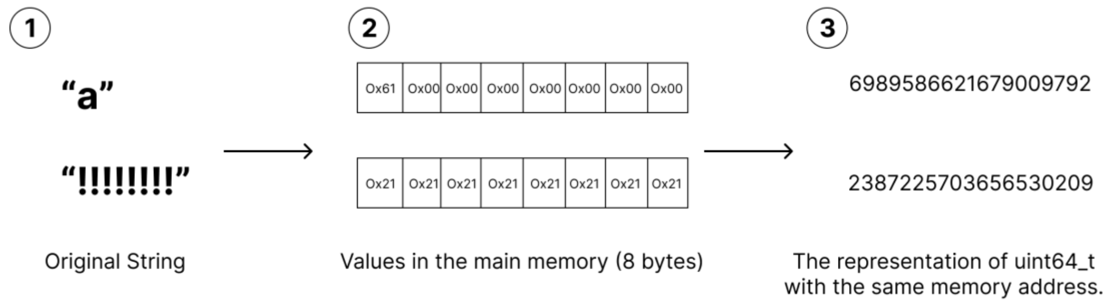

1
2
3
"a"
"!!!!!!!!"
Ox61 Ox00 Ox00 Ox00 Ox00 Ox00 Ox00 Ox00
Ox21 Ox21 Ox21 Ox21 Ox21 Ox21 Ox21 Ox21
6989586621679009792
2387225703656530209
Original String
Values in the main memory (8 bytes)
The representation of uint64_t with the same memory address.

Figure 9: An example of converting string key to unsigned integer

2.2.2 Model Training using PLR Algorithm

Sherry applies the PLR Algorithm to data points *<key, block number>*. All keys are inserted during the training process to increase accuracy. Sherry inputs a pair of unsigned integers of *<key, block number>*. After training, an array of segments will be returned. Each segment represents a line that all data points trained is within the error bound, which consists of *<start_key, line_slope (m), line_offset (c)>.* The segment can be interpreted as the predicted block range. If it is not possible to extend the line segment, a new segment will be created.

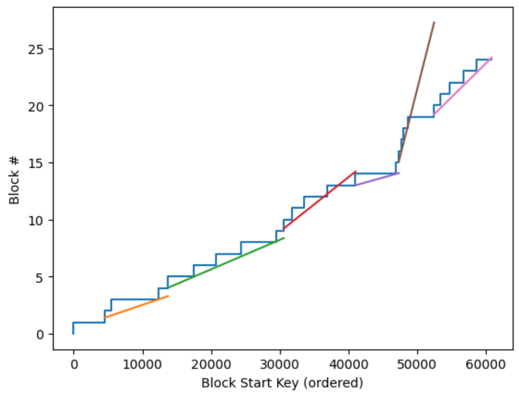


Figure 10: An example of trained segment results from PLR model training

As illustrated by Figure 10, the blue line is the step plot of the key and block number, the colored lines are the segments.

2.2.3 Model Encoding & Index Block Storage

To store the model data in the index block, Sherry encodes the array of segments to a string. Similar to the conversion scheme in Section 2.2.1, it converts each segment to a string with the same bit representation. Then every converted string is concatenated into one string and stored as the first part of the index block. For example, unsigned integer 2387225703656530209 will convert back to "!!!!!!!!" in the storage, which is the same conversion, but different direction as shown in Figure 9.

Other than the model data, a data block offset array is also stored inside the index block. As the model can only interpret the user key and output the possible block range, our learned index could not access the desired data block in constant time without the use of an offset array. The offset array serves as a mapper which maps a block number to a data block pointer, or handle, to the corresponding data block. The array is also encoded to the bitwise representation of string. The overall structure of the index block is shown in Figure 11.

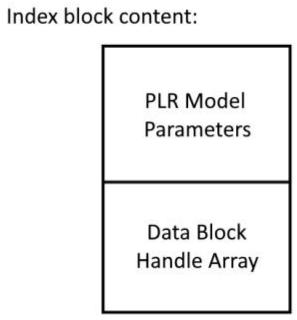


Figure 11: An example of the Index Block storage

2.3 Data Retrieval Design

To migrate the new index block to the existing RocksDB design, the data retrieval process is redesigned to interact with Sherry. As depicted in Figure 12, to retrieve a piece of data in an SSTable, there are several steps: **(1)** Index Block Decoding; **(2)** Model Inference; **(3)** Locating the data block with data block offset array; **(4)** Retrieval of the desired data in the data block.

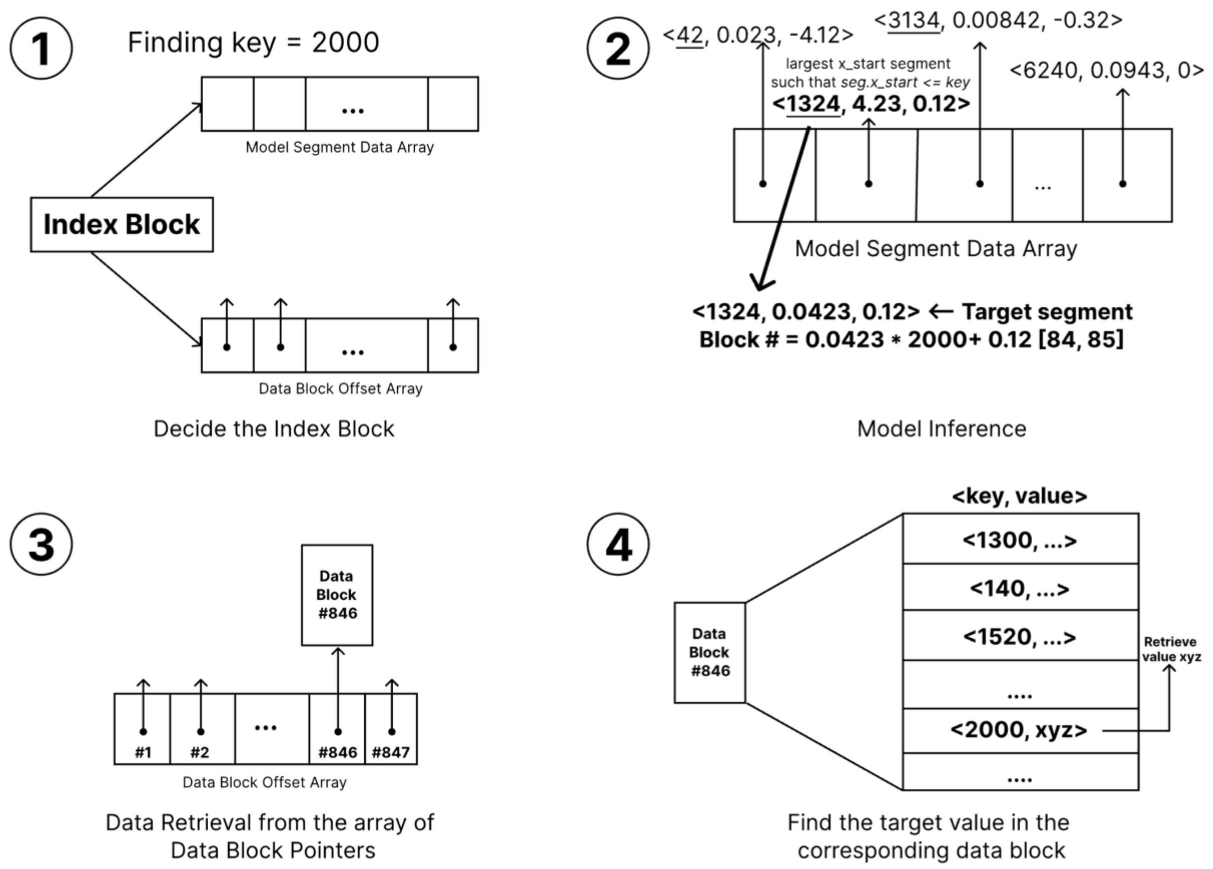


Figure 12: An example of Data Retrieval Process

2.3.1 Index Block Decoding

To make the model segment data and data block offset array usable, it decodes both arrays to the desired data type. Since Sherry encodes both data in the same bit representation with different types, it is trivial for the system by reinterpreting the bits to the original data type. When the decode request is sent, Sherry decodes all data at once and loads it into the main memory instead of on-demand fetching.

Both model segment data array and data block offset array are dynamic in size. The encoded string does not contain any information about the size. Our learned index develops two separate strategies for decoding model segment array and data offset array. To decode model segment array, our algorithm will detect how many segments the model contains by calculating the size of the whole encoded segments. Our learned index constructs the array by reading the value string sequentially. For the data block offset array, our learned index determines the size by the total number of data blocks in the SSTable, instead of determining by the encoded string size. All decoding algorithms will be verified by checking the size of encoding string is a valid integer multiplication of the original data type size.

2.3.2 Model Inference

After training by PLR Algorithm, the model output enables Sherry to make queries about the location of the data block by inputting the query key. The decoded model segment array is sorted as the training data point is processed in a non-increasing order. Therefore, Sherry can perform a binary search to find the largest segment where the *query key >= start_key* of the segment. After retrieving the segment data, Sherry infers the model by calculating the target block linearly. For the example shown in Figure 12, the first segment (start with key = 42) is not used because the second segment (1324) is also smaller than the query key 2000. The second segment is used because the third segment (3134) is larger than the target key.

The PLR Algorithm guarantees that the actual result and the inferred result will at most deviate with ***|error-bound|***. Therefore, our learned index will check the data block for the target key that is within ***[inferred data – error-bound, inferred data + error-bound]***. With the error-bound usually relatively small, the data retrieval process needs to read at most 2-3 data blocks.

2.3.3 Location of Data Block & Data Retrieval

After getting the predicted block range, Sherry consults the decoded Data Block Pointers to get the precise location of the data block. Then, each data block will be read sequentially until the end of the predicted range. In each of the data block, Sherry will perform a binary search to find the value. If the value could not be found after searching all data blocks within the range, then this SSTable does not contain the value.

3. Implementation

Section 3.1 highlights the high-level RocksDB architecture where Sherry is involved. Section 3.2 presents the algorithms related to PLR model training. Section 3.3 showcases the implementation for PLR index construction. Section 3.4 demonstrates the algorithms for PLR index decoding, PLR model inference and data block handle mapping. Section 3.5 depicts the algorithm for read operations in SSTables.

3.1 High-level RocksDB Architecture

Index block construction and reading are triggered in two workflows within the RocksDB architecture: SSTable construction and reading.

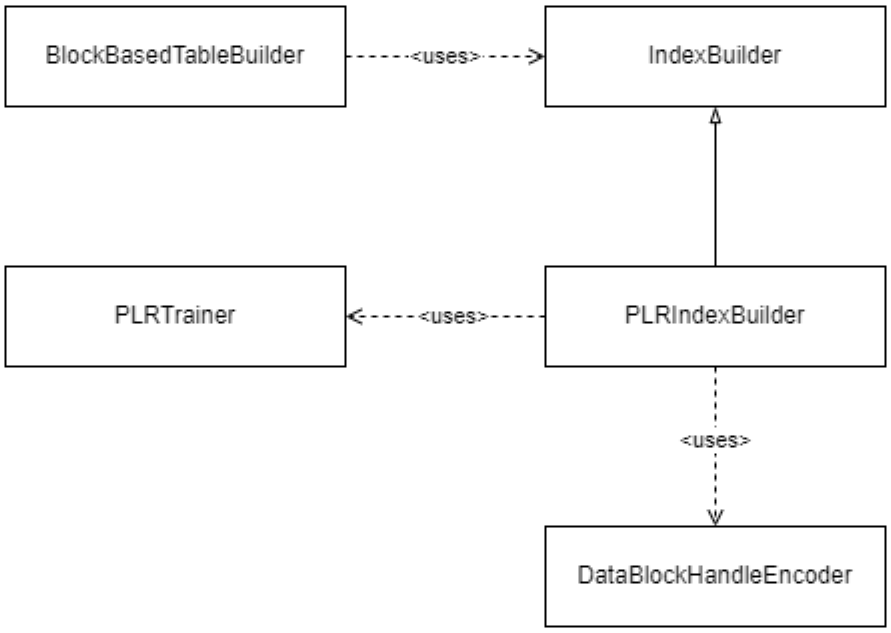


Figure 13: Class Diagram for SSTable and Index Construction

As depicted in Figure 13, BlockBasedTableBuilder is the module responsible for SSTable construction, which is triggered by a flush operation or compaction. In an SSTable building process, it constructs the index block simultaneously with a stream of data blocks. BlockBasedTableBuilder uses PLRIndexBuilder for creating Sherry, which achieves the functionality of index storage described in Section 2.2. PLRIndexBuilder depends on a PLR model training module, PLRTrainer, and a module for encoding data block handles, DataBlockHandleEncoder. The outputs from both PLRTrainer and DataBlockHandleEncoder jointly form the basis of an index block, which will be explained in-depth in Section 3.2 and Section 3.3.

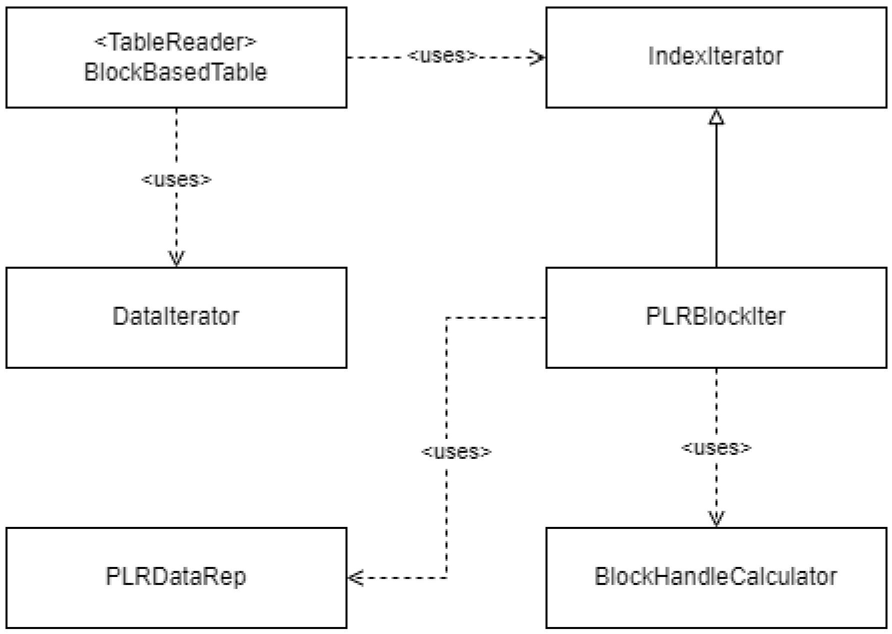


Figure 14: Class Diagram for Reading SSTable and Index Block

As illustrated in Figure 14, for SSTable reading, BlockBasedTable provides the interface for data retrieval, such as read operations. It utilizes two major types of iterators, index (block) iterators and data (block) iterators. As mentioned in Section 1, given a lookup key, the index iterator first locates the corresponding data block, then the data iterator searches for the corresponding key-value entry within that data block. Specifically, our project uses PLRBlockIter as the underlying implementation for index iterators, which adheres to the design described in Section 2.3. PLRBlockIter relies on PLRDataRep, a module responsible for PLR model inference predicting a data block number, and BlockHandleCalculator, a module that translates a data block number to a data block handle. The algorithms used by BlockBasedTable and PLRBlockIter in a read operation will be explained in Section 3.4 and Section 3.5.

3.2 PLR Model Training

As the introduction of the PLR Model in Section 2.2.2, Sherry is an application of using the PLR Algorithm to train a piecewise linear model within an error bound. Each SSTable uses the PLR Algorithm to train a model by inserting ***(user key, block number)*** data points. Yet, the computational cost is high to train the model if all available data points are inserted into PLR Model as a training data set. Also, some SSTables have different versions of the same user keys, causing the duplication of data points with different block number values, which violates the PLR Algorithm requirements.

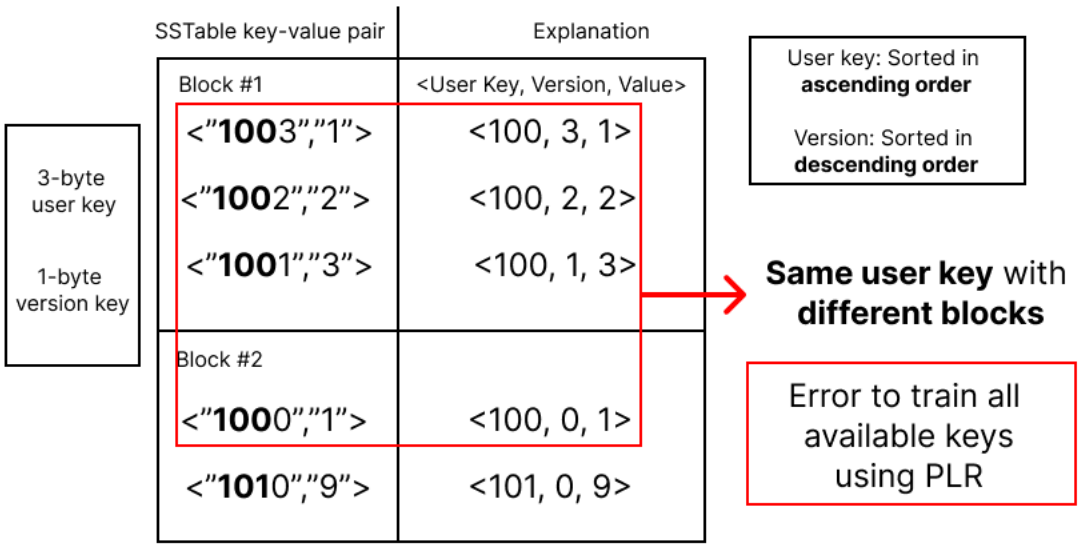


Figure 15: An example of SSTable Internal Structure

As shown in Figure 15, an SSTable may contain different versions of the same user key, pointing to different blocks. The version number (sequence number) is appended after the user key to maintain the SSTable sorted. Yet, the same user key may spam across blocks and cause issues to train a model using PLR Algorithm as it requires the x-coordinate to be different to prevent from 'undefined division by 0' when calculating the line slope.

To avoid the above issues and save computational power, our learned index only feeds the first unique user key of each data block to train the model. If all user keys in a particular data block are duplicated and identical to the last data point's user key, no data points will be fed into the model trainer for the current data block.

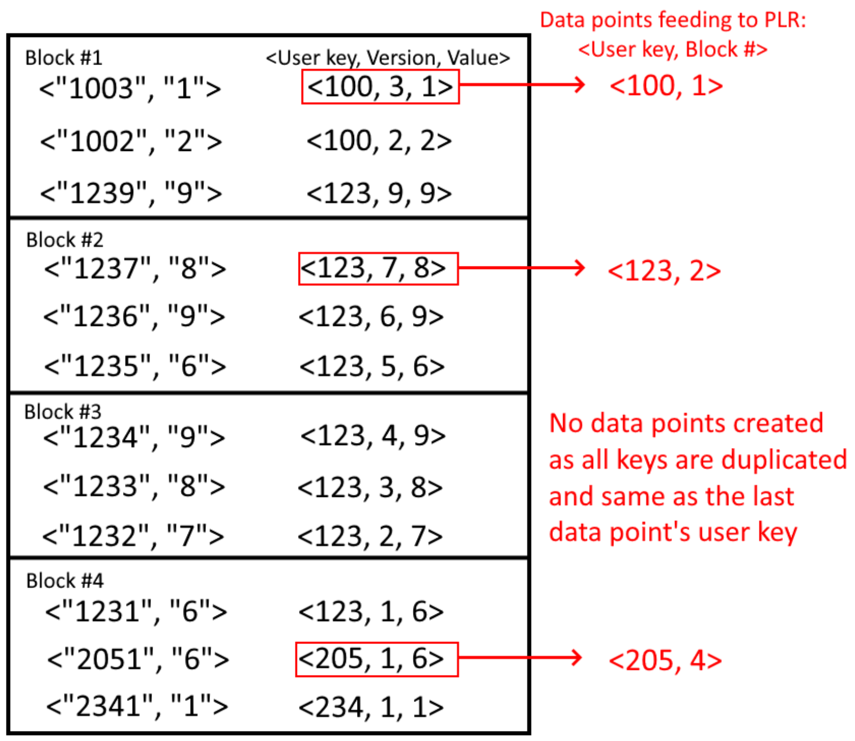


Figure 16: An example of PLR Model Training Process

Consider the example in Figure 16. In block #1, the data point <100, 1> is fed into the model trainer. In block #2, even though the user key “123” already exists in the last block, it has not been fed into the model trainer yet. So, <123, 2> is fed into the trainer. In block #3, all user keys are “123” and equal to the user key in the last training data point, so no data point is created. In block #4, the first non-duplicated user key is “205”, so <205, 4> is added to the trainer. The output of the model trainer is an array of segments which characterize the trained piecewise linear model.

3.3 Index Block Construction

As mentioned in Section 2.3, the actual index block is encoded into two parts: PLR model parameters and a data block handle array.

For each SSTable, the index block utilizes one and only one PLR model. Every PLR model consists of multiple line segments, which can be characterized by three parameters: Starting x-value, slope, and y-intercept. The model parameters are written into an encoded index block sequentially by PLRTrainer. An additional parameter gamma representing the error bound is also inserted at the beginning of the index block. For example, suppose there are 3 segments in total. The order of writing bytes arrays are: gamma, segment #1 starting x-value, segment #1 slope, segment #1 y-intercept, segment #2 starting x-value, segment #2 slope, segment #2 y-intercept, segment #3 starting x-value, segment #3 slope, and segment #3 y-intercept.

For the second part, the index block stores a mapping between data block number and block handle, which contains two parameters: Offset (within an SSTable) and size (of the data block). We have developed an efficient encoding scheme in the DataBlockHandleEncoder to store such mapping in the form of a data block handle array. The array contains the offset of the first data block, followed by an array of data block sizes for all blocks. For example, when there are 3 data blocks, the order of writing byte arrays are: handle #1 offset, handle #1 size, handle #2 size, and handle #3 size.

It is also worth noting that, when the user keys are stored with 8 bytes each, our design of data block handle array implicitly compresses about 33.33% of space consumption in the index block, as illustrated in Figure 17.

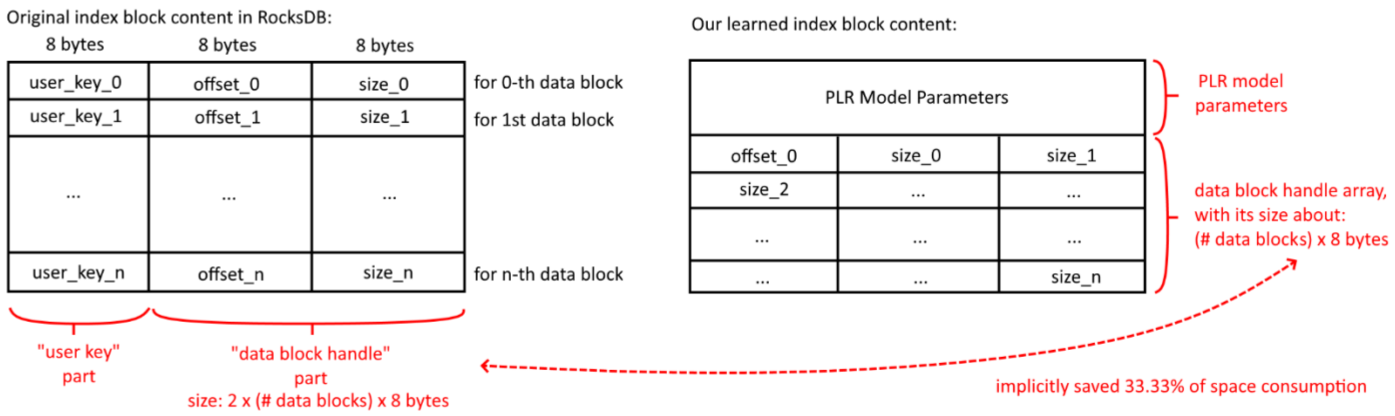


Figure 17: Implicit saving of space consumption when size of user key is 8 bytes

For each data block, the original index block creates an index entry using three 8-byte data (64-bit unsigned integer), which corresponds to a user key, data block offset, and data block size. However, in our learned index block design, we segregate the ‘user key’ part from the ‘data block handle’ part (offset and size). That is, the ‘user key’ part is represented by encoded PLR model parameters, while the ‘data block handle’ part is represented by the data block handle array. In specific, our data block handle array is a less-generic implementation for the ‘data block handle’ part. It uses approximately one 8-byte data per data block, as opposed to two 8-byte data per data block in the original index. This means that our data block handle array implicitly reduces the original index block by one 8-byte data per index entry, which is 33.33% of the original index block size.

3.4 Index Block Decoding and Data Block Retrieval

In a typical read operation within an SSTable, the table reader navigates through data blocks with the aid of an index block. As in Section 3.3, our learned index block is decoded into two parts: A model data trained by the PLR Algorithm, which predicts a data block number, and a mapper that translates block number into block handle for data block access.

The model data will be decoded as a PLRDataRep object, a class representing an array of segments. It enables encapsulation to interact with other programming interfaces. The decoding process follows the decoding scheme which converts the type directly with same bit representation. This can be accomplished by using *union* or *reinterpret_cast()*. The encoded string starts with the gamma error bound, then the sequence of segments in order, as shown in Figure 18. To verify the segments are valid, Sherry checks whether the total size of segments can be divisible by 40 (which is *sizeof(segment)*) after deducting the size of gamma.

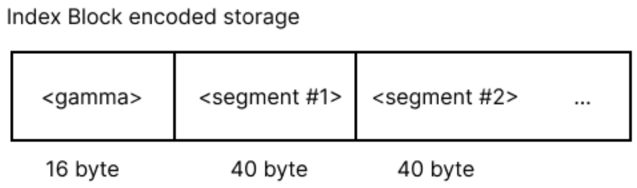


Figure 18: Encoded Segments Storage Layout

After decoding the segments, a PLRDataRep object encapsulating the array of segments is created. It serves the purpose of interacting with iterators and other modules easily. Inferencing the model is the main function of PLRDataRep. Other modules could easily infer the model by calling *GetValue(key)* to get the predicted block range.

During the model inference, PLRDataRep searches for the largest, closest segment whose x-start value is less than or equal to the query key. Using the segment's parameters (slope ***m*** and y-intercept ***c***), the predicted block number is computed by ***mx + c***. The predicted range is then calculated by adding and subtracting the result by gamma, the error bound.

Since the model training does not include intermediate points, the predicted block number may be overshot. To avoid using the overshot result, the function determines whether overshooting occurs, by comparing the current predicted block number with the smallest block number covered by the next segment. If the latter is smaller than the former, the inference is overshot in the current segment. In this case, PLRDataRep returns the predicted range using the next segment.

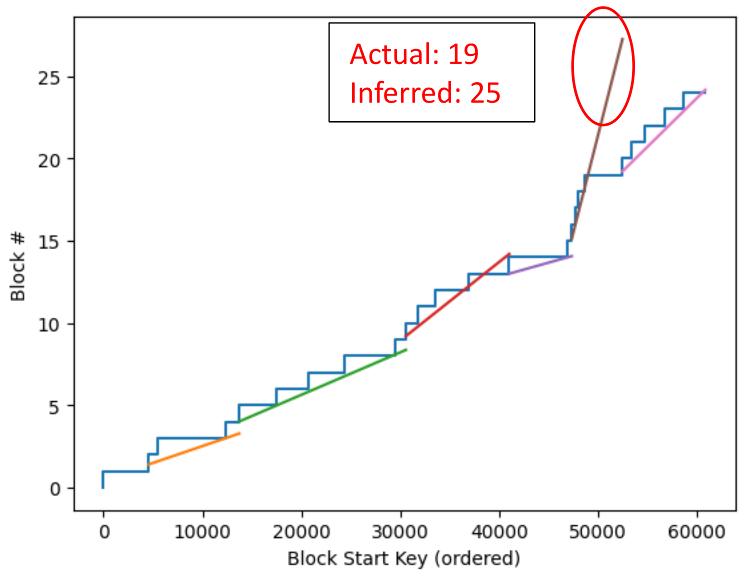


Figure 19: An example of Overshooting without training the intermediate key.

In Figure 19, segment #5 can lead to overshooting when the key is large. Given that *gamma* is *0.5*, and segments #5 and #6 are parameterized as *(47350.7, 0.0023675…, -96.9688…)* and *(52473.8, 0.00059225…, -11.857…)* in the form of *(starting x-value, slope, y-intercept)* respectively. Suppose the arithmetic representation of the input key is *52000*, close to the starting x-value of segment #6. The algorithm chooses segment #5 for initial calculation, then the predicted block number is: ***y = mx + c = 0.0023675 * 52000 – 96.9688 = 26.14***, and the predicted range is: ***[floor(26.14 – gamma), ceiling(26.14 + gamma)] = [25, 27]***. To determine if segment #5 overshoots, PLRDataRep compares the upper bound with the smallest block number covered by segment #6, which is: ***y = mx + c = 0.00059225 * 52473.8 – 11.857 = 19.22***. Since "the smallest block number covered by segment #6 minus by gamma" is less than the predicted upper bound by segment #5, the result is said to be overshot. As a result, we use ***19.22*** as the predicted block number instead, and ***[floor(19.22 – 0.5), ceiling(19.22 + 0.5)] = [18, 20]*** as the final predicted range.

For a normal, non-overshooting example, consider the input key 30000. PLRDataRep first chooses segment #2 *(13726.7, 0.00025827…, 0.46815…)* for prediction. Similar as the computation above, the predicted range by segment #2 is ***[7, 8]***. The first block number covered by the next segment, segment #3 *(30571.7, 0.00047770…, -5.3996…)*, is ***9.20***. Since the upper bound (***8***) is less than "the smallest block number covered by segment #3 minus by gamma" (**8.7**), the result is not overshot. Thus, ***[7, 8]*** is the final predicted range.

To construct the mapper, the BlockHandleCalculator decodes the second part of the index block, and, when a block number is given, it computes the data block handle. The computation of ***i***-th data block handle ***BH***$_i$ relies on the following relationship, where ***kBlockTrailerSize = 5*** is a RocksDB constant which preserves extra space for trailers in each data block:

$$BH_i \coloneqq (offset_i, size_i)$$
$$offset\ _i = offset_{i-1} + size_{i-1} + kBlockTrailerSize$$

Suppose the offset of the first data block is ***x***, then:

$$offset\ _i = x + \sum_{j=0}^{i-1} \left(size_j + kBlockTrailerSize\right)$$

Consider an example with 3 block handles, *(1000, 500)*, *(1505, 600)*, and *(2110, 550)*. This means the initial offset is *1000*, and the sizes of the three data blocks are *500*, *600*, and *550*. Thus, the encoded block handle array from the index block contains these four values. Suppose we are required to compute the 3rd block handle. Applying the formula, the offset of the 3rd block is: *1000 + 500 + 5 + 600 + 5 = 2110*. As a result, BlockHandleCalculator obtains *(2110, 550)* as the 3rd block handle.

3.5 Read Operation within an SSTable

Given a target key, the read operation in an SSTable aims to retrieve the corresponding key-value entry if it exists. Otherwise, it reports the key is not found. As mentioned in Section 2.3, read operations within an SSTable are divided into two major parts: Index block lookup and data block lookup. In the first part, index block lookup consists of two steps. The first step is PLR model inference, which predicts a range of data block numbers. The second step is data block handle lookup using the BlockHandleCalculator. The block handle is then used for data block retrieval and lookup, handled in the second part. Our work only focuses on index block lookup. BlockBasedTable performs a linear scan within the predicted block range and reports the first occurrence of the key-value entry, if it exists.

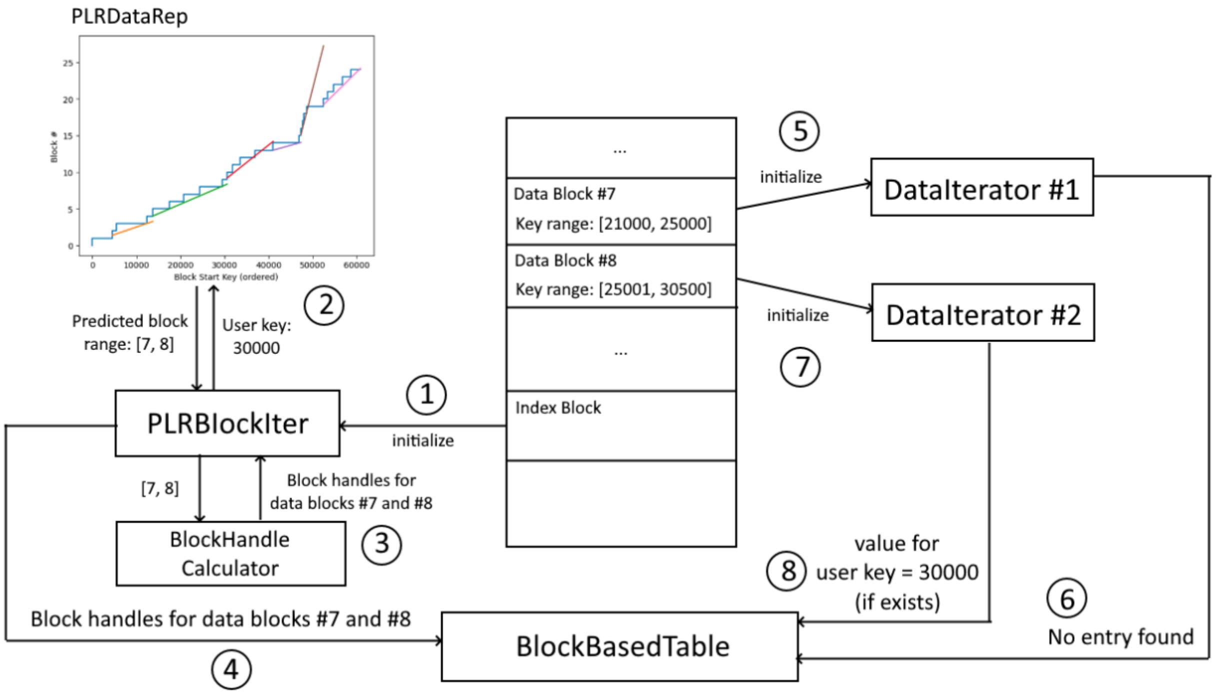

PLRDataRep
Block #
Block Start Key (ordered)
Predicted block range: [7, 8]
User key: 30000
2
1
initialize
PLRBlockIter
[7, 8]
Block handles for data blocks #7 and #8
BlockHandle Calculator
3
Block handles for data blocks #7 and #8
4
...
Data Block #7
Key range: [21000, 25000]
Data Block #8
Key range: [25001, 30500]
...
Index Block
5
initialize
DataIterator #1
initialize
DataIterator #2
7
value for user key = 30000 (if exists)
8
6
No entry found
BlockBasedTable

Figure 20: An example of read operation

Figure 20 illustrates an example flow in a typical read operation on SSTables. The example follows the one in Section 3.4, using the same PLR model as Figure 19 with the same user key, ***30000***. Firstly, BlockBasedTable initializes the index iterator, PLRBlockIter, decodes the PLR model, and initializes the BlockHandleCalculator. The PLR model returns the predicted range ***[7, 8]*** for the user key, ***30000***, and the BlockHandleCalculator returns the block handles for data blocks #7 and #8. BlockBasedTable then conducts a linear scan over the predicted range until the scanned data block covers ***30000***. The first data iterator searches within data block #7, but ***30000*** is out of its key range. The second data iterator searches within data block #8, which covers 30000. The iterator stops at the first key-value entry with ***key >= 30000*** and return the corresponding value if the entry's key is ***30000***. Otherwise, it reports that no entry exists for ***30000***.

The above algorithm can correctly return the value of the first entry whose user key is identical to the target user key. In the real implementation, since multiple entries with the same user key but varying versions coexist in RocksDB, the index lookup is not bounded by the upper bound from PLR model inference. Instead, the ending condition of the index lookup loop is determined by: (1) Whether the correct key-value entry with a visible version is found, or (2) whether the largest user key in the previous data block is larger than the target user key. Either condition being true will break the index lookup loop.

4. Testing

In this section, we discuss the unit tests, integration tests, and stress tests conducted, before benchmarking Sherry and comparing the performance with the default index used in RocksDB.

4.1 Unit Test

Table 2 summarizes the testing criteria and results for different artifacts:

Test Number	Testing Artifacts	Testing Scope	Results
1	Lines, Segments	Operations on basic data structure used in PLR training.	Passed
2	PLRTrainer	PLR model training and encoding.	Passed
3	PLRDataRep	PLR block decoding and inference.	Passed
4	DataBlockHandleEncoder	Data block handle array encoding.	Passed
5	BlockHandleCalculator	PLR block decoding and block handle calculation.	Passed
6	PLRIndexBuilder	PLR block building.	Passed
7	PLRBlockIter	PLR block decoding and navigation.	Passed

Table 2: Summary of Unit Test

4.1.1 Testing for PLR Model Training, Storage and Retrieval

We implemented the PLR algorithms and models as a separate module. We utilize the Google Test Framework (GTest) to verify the accuracy of the codebase for every module. For the underlying data structures (Segments and Lines), we conducted various tests, encompassing edge cases such as intersecting points, positive and negative segments, and segment offset calculations.

Since the PLR Model is a learning model, unit testing may not be the most effective way to verify its accuracy. Therefore, we generated a random dataset from normal and Zipfian distributions on keys and fed it to the PLR model. We utilized Python and matplotlib to visualize the trained segments and verify the correctness of the model.

In addition to testing the PLR model, we also test the PLRDataRep for model encoding and decoding. To test the PLRDataRep class, we created a PLRDataRep object and inserted randomized Segments. We then perform encoding and decoding operations to verify that the decoded object is equivalent to the original object. Furthermore, we utilized the visualized data shown above to simulate real-life operations and thoroughly tested the functions.

4.1.2 Testing for Index Block Decoding

The unit test for index block decoding encompasses two parts: Decoding of PLR model parameters and data block handle array. The first part is handled in Section 4.1.1.
The second part includes unit tests for BlockHandleCalculator, focusing on the correctness of its public functions, including encoding, decoding, and computing data block handles.

4.1.3 Testing for Index Block Navigation

In this section, we verify the correctness of the iterator for our learned index in terms of seek operations.

Testing our iterator required special attention. Other index implementations create a deterministic iterator which always seeks for the correct data block. However, our learned index creates a non-deterministic iterator. It approximates the range of data block numbers and requires a linear search to seek for the correct data block.

We define the correctness of the iterator's seeking behavior as follows: Given a non-negative floating point number ***gamma*** as the error bound of the PLR model, the trained PLR model takes a string key (encoded as integer) as input and predicts a range of data block numbers. If the string key exists, the predicted range must contain the data block which covers the string key.

At a high-level perspective, we parameterized the test suite with different ***gamma*** values. Each test first generates a sorted list of 100 pairs of string keys and data block handles. These pairs are added to PLRIndexBuilder sequentially, and PLRIndexBuilder encodes and outputs a string as the result. The encoded string resembles the actual index block contents written in a SSTable file.

Afterwards, the test then decodes the string and creates an index block iterator, PLRBlockIter. The test then verifies the correctness of the iterator's seek operation: Given a random string key from the sorted list, the iterator should output an interval covering the correct index number of the string key in the sorted list. We tested this behavior iteratively in each of the test case.

4.2 Integration Test: Read Operations in SSTables

The integration test with SSTables aims to verify the correctness of interactions between BlockBasedTable and PLRBlockIter. It is conducted by loading randomized key-value entries into an SSTable and specifying the use of Sherry for the index block in table options.

4.3 Stress Test

The stress test in RocksDB aims to validate the reliability at scale. It is a randomized test that covers different combinations of database features, including data integrity and crash recovery. To test the correctness of our learned index, we reuse the stress test module by specifying Sherry as the index block type in block-based tables.

The stress test is divided into two parts: Whitebox and Blackbox tests, which crash the database at pre-defined checkpoints and at random timings respectively, each time with different database options. Passing through such crash tests implies Sherry is reliable in terms of data integrity and recovery. Table 3 summarizes the stress test results under different access patterns, with other database options randomized.

Test Number	Workload	With delete operations?	With overwriting keys?	Results
1	Read-heavy	Yes	Yes	Passed
2			No	Passed
3		No	Yes	Passed
4			No	Passed
5	Write-heavy	Yes	Yes	Passed
6			No	Passed
7		No	Yes	Passed
8			No	Passed
9	Scan-heavy	Yes	Yes	Passed
10			No	Passed
11		No	Yes	Passed
12			No	Passed

Table 3: Summary of Stress Test using Sherry

5. Evaluation

After verifying the correctness of Sherry, we perform benchmarking to evaluate the performance of Sherry and compare the result with the original index used in RocksDB to evaluate if Sherry can fulfill our objectives or not.

5.1 Evaluation Settings

In order to minimize the differences in performance due to benchmarking environment, we perform benchmarking on virtual machines on Google Cloud Platform (GCP). The specification of the virtual machine instance is listed as follows:

- Machine Type: C2
- CPU: Intel® Xeon® Gold 6253CL
- vCPUs: 8
- Memory: 32 GB
- Operating System: Ubuntu 20.04 LTS
- Disk Type: SSD persistent disk
- Disk Size: 500 GB

5.2 Time Performance

5.2.1 Time Performance Benchmark Procedure

The benchmark is divided into 5 tests with different scenarios, including write-only, read-only, write-heavy, read-heavy, and read-write-balanced. And for each scenario, we tested with 3 different key distributions, including uniform, linear, and exponential. We used each of these settings compare the performance of Sherry with the default index in RocksDB, creating a total of 30 different combinations of tests as shown below:

- Benchmark:
  - fillrandom (Write:100%)
  - readrandom (Read:100%)
  - readrandomwriterandom (Write: 90%, Read: 10%)
  - readrandomwriterandom (Write: 10%, Read: 90%)
  - readrandomwriterandom (Write: 50%, Read: 50%)
- Distribution: Uniform, Linear, Exponential
- Index used:
  - Default Index
  - Sherry

For the benchmarks, there are some crucial parameters that are shared to keep the result more consistent across benchmarks more consistent. The parameters are listed as follows:

- Duration: 1200 seconds (20 minutes)
- Key Range: [0, 900000000]
- Key Size: 8 bytes
- Value Size: 400 bytes
- Data Block Size: 400 bytes
- Fixed Seed for random generation
- Uses 16 threads

We run each test for 20 minutes to ensure the statistics obtained are stable and can represent the actual performance of the current settings. Also, we use fixed seed to ensure the random generated contents between the benchmarks are the same. The statistics are generated every 30 seconds, allowing us to observe abnormal patterns if they exist.

For each test, we evaluate the performance with the following metrics:

- Number of operations per seconds
- Microseconds per read / write
- Cumulative writes (if applicable)

For microseconds per read / write, we compare the values at P50 (50- percentile), P75 (75- percentile), and P99 (99- percentile) to observe the performance on average and at the worst cases. For cumulative writes, we compare the writing speed in terms of MB/s directly, which gives an intuitive understanding of how well the index is performing.

5.2.2 Time Performance Results

For uniform distribution, Sherry has a performance gain in most of the settings for write operations in terms of microseconds per write. As shown in Table 4, the performance gain is significant under less write-intensive workload, with worse cases (P99) of being almost 6x worse. This suggests that Sherry can perform better than the default index consistently in terms of write operations with rare exceptional case under read-heavy workload.

Comparison Grid for **Time Required per Write Request** in **Uniform Distribution** (%)			
Benchmark Case (Read-Write Percentage)	P50	P75	P99
`Fiilrandom` (Write:100%)	6.95%	8.70%	2.50%
`Readrandomwriterandom` (Read:90%, Write:10%)	49.16%	67.17%	-581.03%
`Readrandomwriterandom` (Read:10%, Write:90%)	4.08%	2.24%	26.64%
`Readrandomwriterandom` (Read:50%, Write:50%)	4.16%	2.44%	29.23%

Table 4: Comparison of Microseconds Per Write with Uniform Distributed Keys (Measured in Percentage Decrease in Microseconds Used, Larger Means Better)

From Figure 21, we can observe that the performance gain is diminishing in general when increasing the write percentage. Such a phenomenon is due to the increasing number of compactions triggered by frequent write operations.

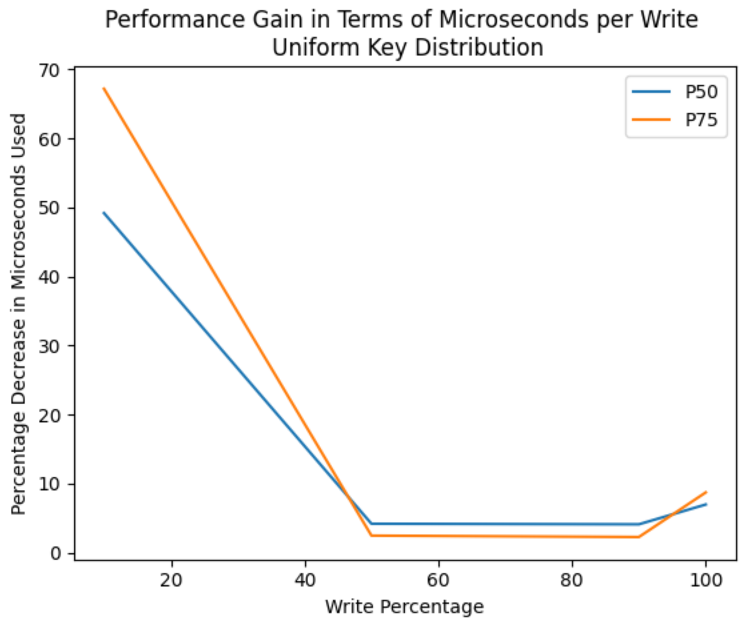


Figure 21: Trend of Performance Gain when Increasing the Write Percentage, Uniform Key Distribution

For read operations, Table 5 suggests that Sherry does not have an advantage when compared to the default index in terms of microseconds per read. As proportion of read increases against write, the performance of Sherry deteriorates. All mixed workload settings suffer from extreme cases that significantly slow down the reading process as indicated by the value at P99.

Comparison Grid for **Time Required per Read Request** in **Uniform Distribution** (%)			
Test Case (Read-Write Percentge)	P50	P75	P99
`Readrandom` (Read:100%)	-43.47%	-36.07%	-48.16%
`Readrandomwriterandom` (Read:90%, Write:10%)	-13.01%	-32.89%	-3960.97%
`Readrandomwriterandom` (Read:10%, Write:90%)	-7.69%	-5.63%	-1491.79%
`Readrandomwriterandom` (Read:50%, Write:50%)	-8.23%	-13.99%	-1979.44%

Table 5: Comparison of Microseconds Per Read with Uniform Distributed Keys (Measured in Percentage Decrease in Microseconds Used, Larger Means Better)

We can observe that the read performance is declining when increasing the read percentage in Figure 22. Such phenomenon suggests that Sherry is having a bottleneck when reading.

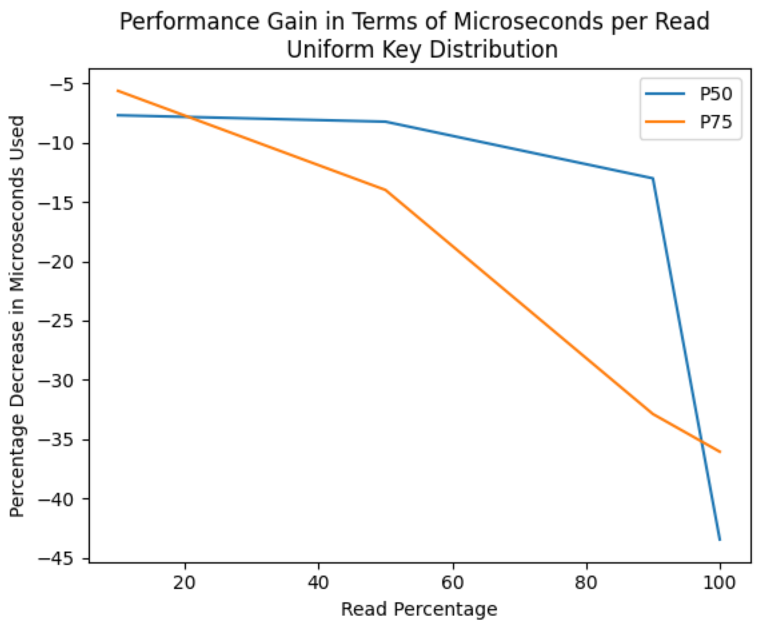


Figure 22: Trend of Performance Gain when Increasing the Read Percentage, Uniform Key Distribution

For linear distribution, Table 6 shows that Sherry has similar or slightly worse write performance in most of the settings. Although the results above suggest that Sherry favors write operations, the data in Table 6 report a performance decline in write operations for write percentage >= 50% (Almost the same as the default performance for 100% write at P75)

Comparison Grid for **Time Required per Write Request** in **Linear Distribution** (%)			
Test Case (Read-Write Percentage)	P50	P75	P99
`Fiilrandom` (Write:100%)	1.01%	-0.20%	-11.89%
`Readrandomwriterandom` (Read:90%, Write:10%)	71.41%	71.28%	-737.78%
`Readrandomwriterandom` (Read:10%, Write:90%)	-6.15%	-10.26%	-8.30%
`Readrandomwriterandom` (Read:50%, Write:50%)	-16.84%	-11.66%	-203.93%

Table 6: Comparison of Microseconds Per Write with Linear Distributed Keys (Measured in Percentage Decrease in Microseconds Used, Larger Means Better)

In Figure 23, we can see a similar pattern as the case in uniform key distribution, with the difference that settings with write percentage >= 50% is suffering from performance loss instead.

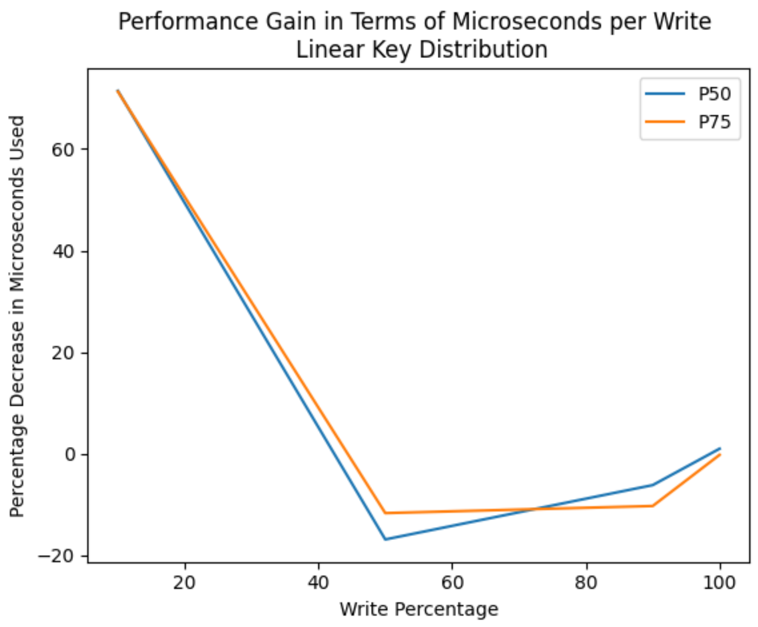


Figure 23: Trend of Performance Gain when Increasing the Write Percentage, Linear Key Distribution

For read operations, Table 7 suggests that Sherry does not have an advantage when compared to the default index in terms of microseconds per read. It is worth noticing that Sherry has some extreme cases under mixed workload.

Comparison Grid for **Time Required per Read Request** in **Linear Distribution** (%)			
Test Case (Read-write Percentage)	P50	P75	P99
`Readrandom` (Read:100%)	-49.59%	-50.22%	-53.27%
`Readrandomwriterandom` (Read:90%, Write:10%)	-1.59%	-0.37%	-4597.53%
`Readrandomwriterandom` (Read:10%, Write:90%)	-9.43%	-20.38%	-2130.11%
`Readrandomwriterandom` (Read:50%, Write:50%)	-7.49%	-12.77%	-1969.25%

Table 7: Comparison of Microseconds Per Read with Linear Distributed Keys (Measured in Percentage Decrease in Microseconds Used, Larger Means Better)

We can observe a different pattern for read performance gain in Figure 24. The performance does not decline as the read percentage increases. A peak closely resembles the performance of the default index when read percentage = 90%. The current data could not draw any conclusion and the reason behind such observation requires more in-depth investigation.

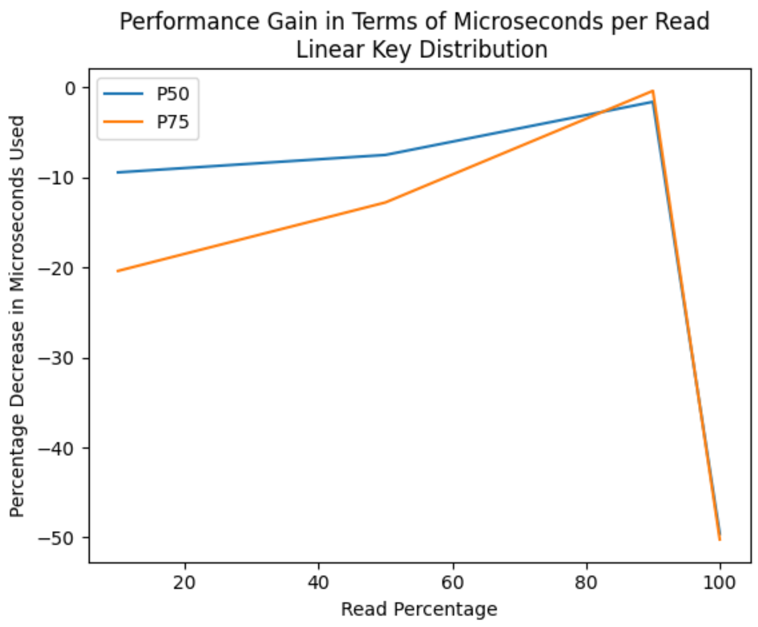


Figure 24: Trend of Performance Gain when Increasing the Read Percentage, Linear Key Distribution

For exponential distribution, Sherry obtains a similar performance when compared with the default index at a balanced workload, as shown in Table 8. As for other settings, the performance gains are similar to the cases in linear key distribution. The values at P99 are better than the ones using the linear key distribution. This implies that Sherry performs more consistently with exponential key distribution when compared with using linear key distribution.

Comparison Grid for **Time Required per Write Request** in **Exponential Distribution** (%)			
Test Case (Read-Write Percentage)	P50	P75	P99
`Fiilrandom` (Write:100%)	3.44%	-0.67%	-73.09%
`Readrandomwriterandom` (Read:90%, Write:10%)	74.47%	70.61%	-792.89%
`Readrandomwriterandom` (Read:10%, Write:90%)	-16.95%	-6.86%	-19.63%
`Readrandomwriterandom` (Read:50%, Write:50%)	0.04%	0.12%	0.17%

Table 8: Comparison of Microseconds Per Write with Exponential Distributed Keys (Measured in Percentage Decrease in Microseconds Used, Larger Means Better)

Drawing insights from Figure 21, Figure 23, and Figure 25, we observed that the trend of performance gain with the increase in write percentage is similar across the 3 types of key distribution, with a decline from 10% write to 90% write, and the performance at pure writing is better than having 10% read and 90% write. From our investigation, the decline of write performance in mixed workload is due to the increasing amount of compaction required, as training the PLR model requires extra computations. As for the case of pure writing, the bottleneck of read operations is removed and therefore Sherry can utilize all available resources for writing, resulting in a slightly better performance.

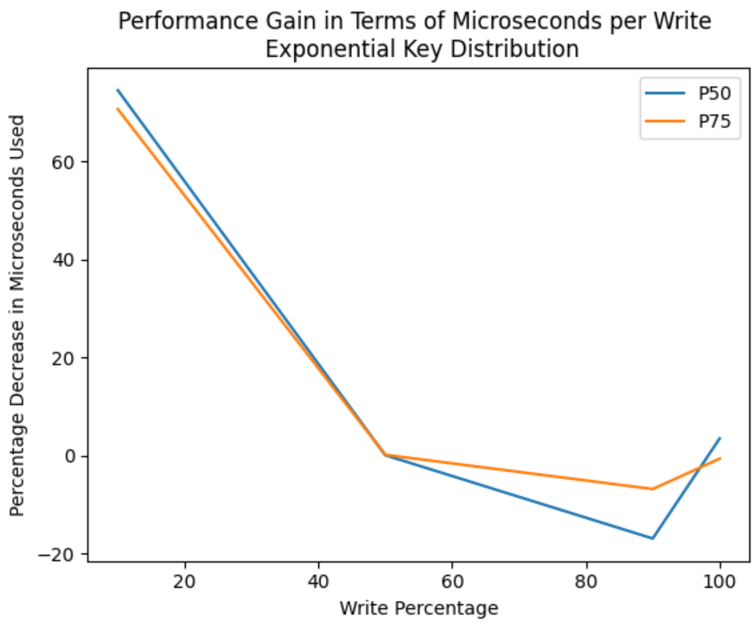


Figure 25: Trend of Performance Gain when Increasing the Write Percentage, Exponential Key Distribution

For read operations, Table 9 suggests that Sherry does not have an advantage in mixed workload (reading and writing concurrently) when compared to the default index in terms of microseconds per read, having a similar or slightly worse read performance. However, it is worth noticing that Sherry is outperforming the default index with a time reduction of over 25% in terms of microseconds per read in pure read workload. Further investigations are needed to understand such a phenomenon.

Comparison Grid for **Time Required per Read Request** in **Exponential Distribution** (%)			
Test Case (Read-Write Percentage)	P50	P75	P99
`Readrandom` (Read:100%)	25.88%	33.26%	31.27%
`Readrandomwriterandom` (Read:90%, Write:10%)	-3.62%	-1.95%	-4849.80%
`Readrandomwriterandom` (Read:10%, Write:90%)	-14.89%	-19.38%	-1923.02%
`Readrandomwriterandom` (Read:50%, Write:50%)	-17.08%	-24.93%	-2602.58%

Table 9: Comparison of Microseconds Per Read with Exponential Distributed Keys (Measured in Percentage Decrease in Microseconds Used, Larger Means Better)

In Figure 26, we have a completely different trend of performance gain in microseconds per read when using exponential key distribution. The performance gain increases as the read percentage increases. Since previous key distributions (Uniform, Linear) do not give similar results, we believe that exponential key distribution plays a crucial role in the performance gain here. Further benchmarking is required to understand the effect of exponential key distribution more thoroughly.

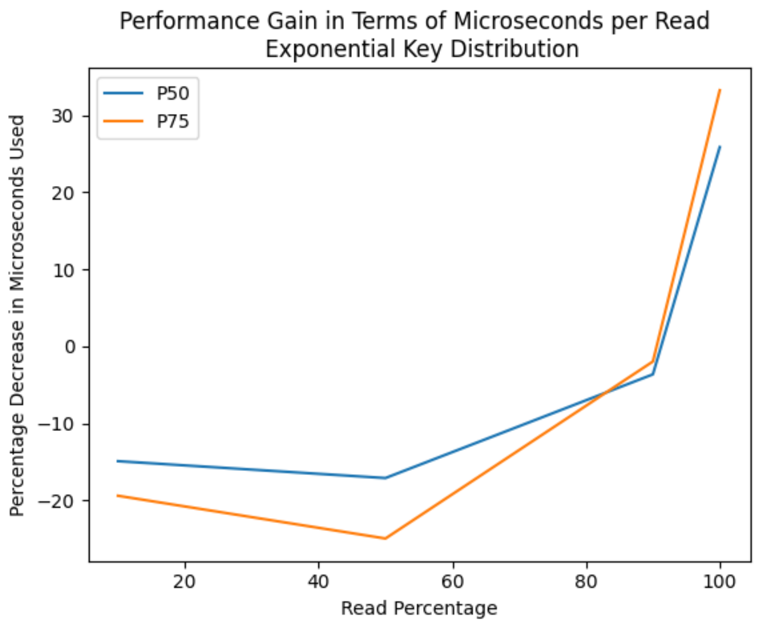


Figure 26: Trend of Performance Gain when Increasing the Read Percentage, Exponential Key Distribution

Finally, in Table 10, Table 11, and Table 12, we consider the cumulative writes speed between the default index and Sherry under different settings. In general, Sherry is having a better performance in write-heavy workloads. The exception comes from Fiilrandom using exponentially distributed keys, which writes 60.88% slower than the default index, while other Fiilrandom using keys with different distribution have performance differences ranging from -5.25% to 1.24% only.

Comparison of **Cumulative Writes** to disk with **Uniform Distribution** (%)	
Test Case (Read-Write Percentage)	Cumulative Writes
`Fiilrandom` (Write:100%)	-5.25%
`Readrandomwriterandom` (Read:90%, Write:10%)	-69.53%
`Readrandomwriterandom` (Read:10%, Write:90%)	15.44%
`Readrandomwriterandom` (Read:50%, Write:50%)	14.40%

Table 10: Comparison of Cumulative Writes with Uniform Distributed Keys (Measured in Percentage Increased in MB/s, Larger Means Better)

Comparison of **Cumulative Writes** to disk with **Linear Distribution** (%)	
Test Case (Read-Write Percentage)	Cumulative Writes
`Fiilrandom` (Write:100%)	1.24%
`Readrandomwriterandom` (Read:90%, Write:10%)	-72.37%
`Readrandomwriterandom` (Read:10%, Write:90%)	-7.45%
`Readrandomwriterandom` (Read:50%, Write:50%)	-29.35%

Table 11: Comparison of Cumulative Writes with Linear Distributed Keys (Measured in Percentage Increased in MB/s, Larger Means Better)

Comparison of **Cumulative Writes** to disk with **Exponential Distribution** (%)	
Test Case (Read-Write Percentage)	Cumulative Writes
`Fiilrandom` (Write:100%)	-60.88%
`Readrandomwriterandom` (Read:90%, Write:10%)	-72.67%
`Readrandomwriterandom` (Read:10%, Write:90%)	-7.86%
`Readrandomwriterandom` (Read:50%, Write:50%)	-1.53%

Table 12: Comparison of Cumulative Writes with Exponential Distributed Keys (Measured in Percentage Increased in MB/s, Larger Means Better)

We can interpret the actual writing performance by referring to the "Cumulative Writes" statistics. However, the cumulative writes' speed does not align with the result of microseconds per write shown above. We observed inconsistency between "Microseconds per Write" and "Cumulative Writes". A time reduction in "Microseconds per Write" does not imply an increase in "Cumulative Writes" in our benchmarks. We believe that there are other factors contributing to the "Cumulative Writes" results.

Therefore, we investigated the relationship among "Microseconds per Write", "Cumulative Writes", and "Ops/Sec". We found out the performance of "Cumulative Writes" is positively correlated to "Ops/Sec". The relations of these 2 metrics are shown in Table 13 (Uniform Key Distribution), Table 14 (Linear Key Distribution), and Table 15 (Exponential Key Distribution).

Uniform Distribution	Cumulative Writes (GB)			Ops/Sec		
Test Case	Default	Sherry	Difference	Default	Sherry	Difference
Fiilrandom (Write:100%)	321.12	304.25	-5.25%	800094	758082	-5.25%
Readrandom writerandom (Read:90%, Write:10%)	14.57	4.44	-69.53%	365033	110649	-69.69%
Readrandom writerandom (Read:10%, Write:90%)	2.59	2.99	15.44%	7302	8430	15.45%
Readrandom writerandom (Read:50%, Write:50%)	2.57	2.94	14.40%	13017	14934	14.73%

Table 13: Relationship Between Change in Cumulative Writes and Change in Ops/Sec, Uniform Key Distribution

Linear Distribution	Cumulative Writes (GB)			Ops/Sec		
Test Case	Default	Sherry	Difference	Default	Sherry	Difference
Fiilrandom (Write:100%)	321.99	325.98	1.24%	801984	811816	1.23%
Readrandom writerandom (Read:90%, Write:10%)	16.47	4.55	-72.37%	412518	113150	-72.57%
Readrandom writerandom (Read:10%, Write:90%)	3.76	3.48	-7.45%	10583	9802	-7.38%
Readrandom writerandom (Read:50%, Write:50%)	3.1	2.19	-29.35%	15737	11098	-29.48%

Table 14: Relationship Between Change in Cumulative Writes and Change in Ops/Sec, Linear Key Distribution

Exponential Distribution	Cumulative Writes (GB)			Ops/Sec		
Test Case	Default	Sherry	Difference	Default	Sherry	Difference
`Fiilrandom` (Write:100%)	342.61	134.04	-60.88%	853642	333786	-60.90%
`Readrandom writerandom` (Read:90%, Write:10%)	16.94	4.63	-72.67%	424161	115272	-72.82%
`Readrandom writerandom` (Read:10%, Write:90%)	3.18	2.93	-7.86%	8951	8255	-7.78%
`Readrandom writerandom` (Read:50%, Write:50%)	3.26	3.21	-1.53%	16509	16284	-1.36%

Table 15: Relationship Between Change in Cumulative Writes and Change in Ops/Sec, Exponential Key Distribution

To conclude, Sherry has advantages in write operations under read-heavy workload, with rare exceptions. In general, Sherry does not have advantage over the default index in read operations. An exception is that it outperforms the default index under pure reading workload. The distribution of keys has a significant impact on Sherry's performance, where a simpler distribution favors the write operations while a more complex distribution favors the read operations.

5.3 Space Performance

5.3.1 Space Performance Benchmark Procedure

To measure the performance in terms of space, we measure the size of index block as the indicator of space performance. To make a fair test environment, all tests are performed in the following procedure:

(1) **Generate test files**: We use a C++ Program to generate the files to be put in the database. There are three key distributions: Uniform, Linear, Exponential.

(2) **Test with database**: Select one dataset, read the dataset, and put all key-value pair sequentially. The same dataset is used to test with database with default and PLR.

(3) **Calculate Index Size**: Calculate the index size for all SSTables in one database using *SstFileReader* API exposed to user.

The goal of this test is to measure the space saving rate of the PLR model compared to the conventional index. As mentioned in Section 2.3, Sherry adopts the data block handle array design. With our implementation, it implicitly saves one-third of the index size compared to the conventional index. To maintain the fairness of the test, the benchmark presents are "adjusted saving rate", with the following formula.

$$\text{Adjusted Saving Rate} = \frac{\text{Conventional Index Size - Learned Index Size}}{\text{Conventional Index Size}} - 33.33\%$$

5.3.2 Space Performance Results

In the benchmark, we would like to investigate different possible factors that affect the space consumption of our learned index. We test our space consumption against the default index in RocksDB under the factors described in Table 16.

Test Number	Dependent Variable	Independent Variable
1	Fixed Value Size (400 bytes, 200 bytes, 20 bytes)	Key Size, Total ingested file size, number of key-value pair inserted
2	Value Size Type (400 bytes, 1-400 bytes)	Key size, number of key-value pair inserted, sequence of the value-size
3	Number of entries of key-value pair (1 M, 5 M, 10 M)	Key Size, Value Size, Sequence of Insertion
4	Data Block Size (4KB, 8KB, 16KB)	Key Size, Value Size, Number of entries, Sequence of insertion
5	Error-bound in PLR Algorithm (0.5, 1, 2)	Key Size, Value Size, Number of entries, Sequence of insertion, Data Block size

Table 16: Summary of Space Benchmarking Test Plan

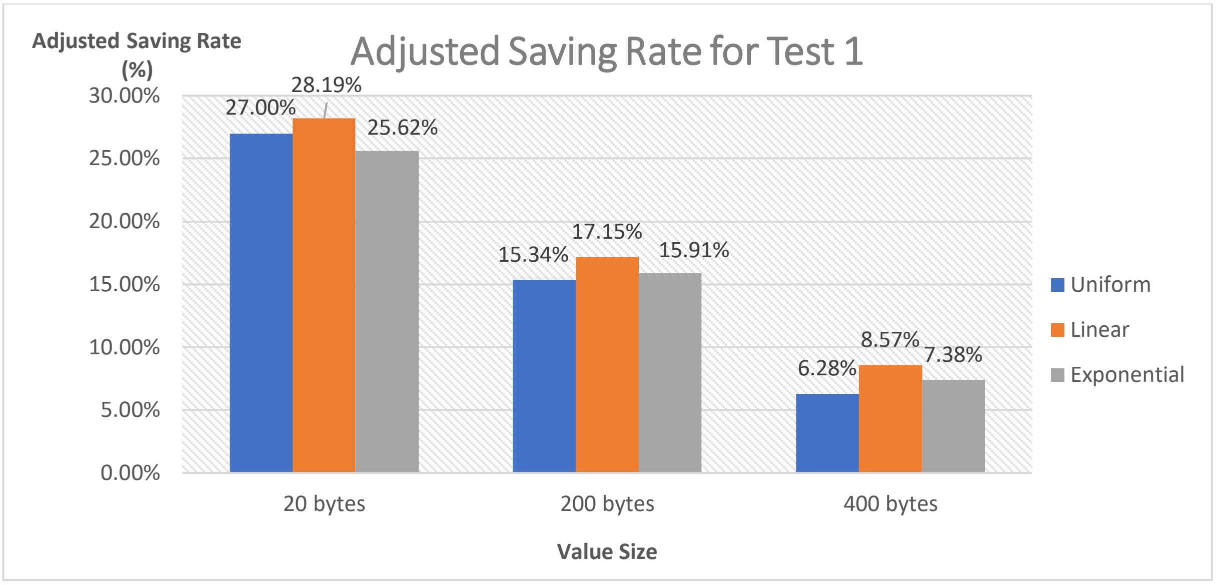


Figure 27: Space Benchmark Result for Test 1

The goal of Test 1 is to investigate the effect of the value size under the same database size. From Figure 27, we can see that the key distribution is not the main factor of the saving rate of our index. Instead, the lower the value size, the higher saving rate Sherry gained. If all values are 20 bytes long, Sherry saves up to 30% index size compared to the conventional index.

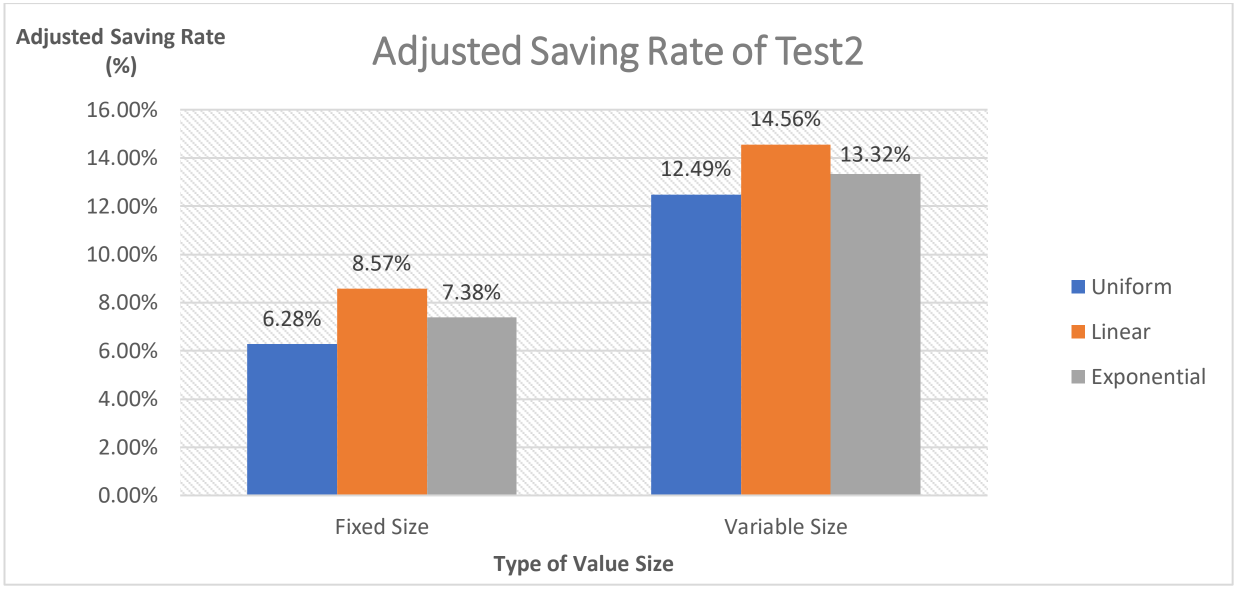


Figure 28: Space Benchmark Result for Test 2

In Test 2, we focus on whether a variable key-value pair will affect the performance of Sherry. In Figure 28, given that the average size of the value in the variable size is around 200 bytes, the boost of the saving rate is likely due to the decrease of the key size (Test 1). Compared to Test 1 value size = 200 bytes, the saving rate on average decreases 3%. Therefore, the variable size key increases the index size slightly.

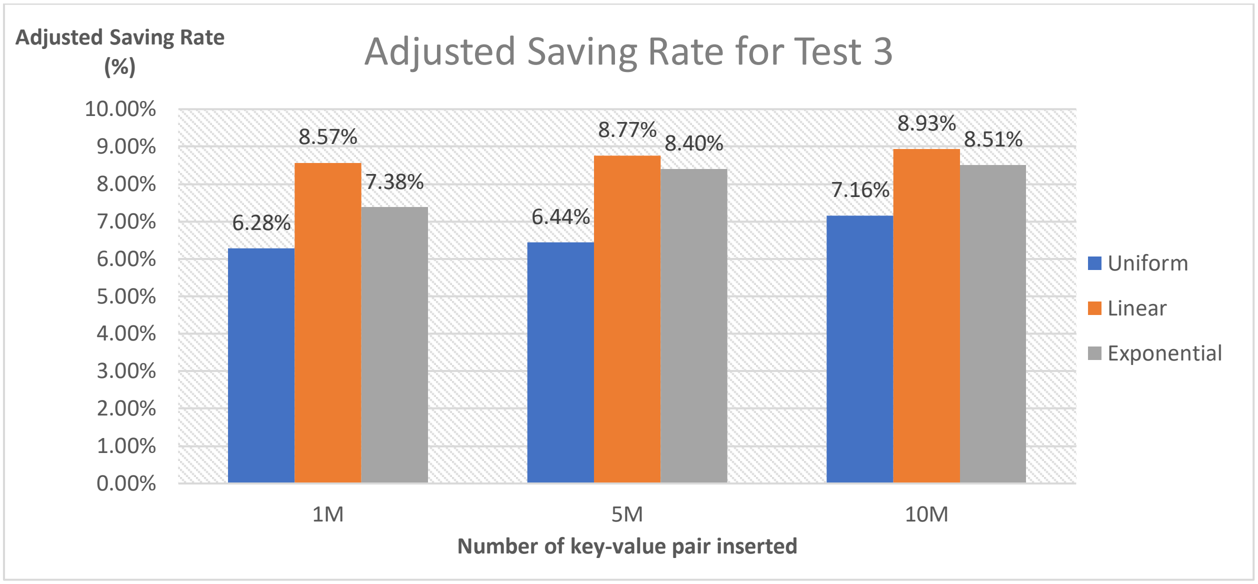


Figure 29: Space Benchmark Result for Test 3

In Figure 29, Test 3 shows that the space saving rate is independent of the number of key-value pairs in the database. That is, the saving rate will not improve / deteriorate when the database grows over time. Our saving rate is stable and independent of the database elapse time.

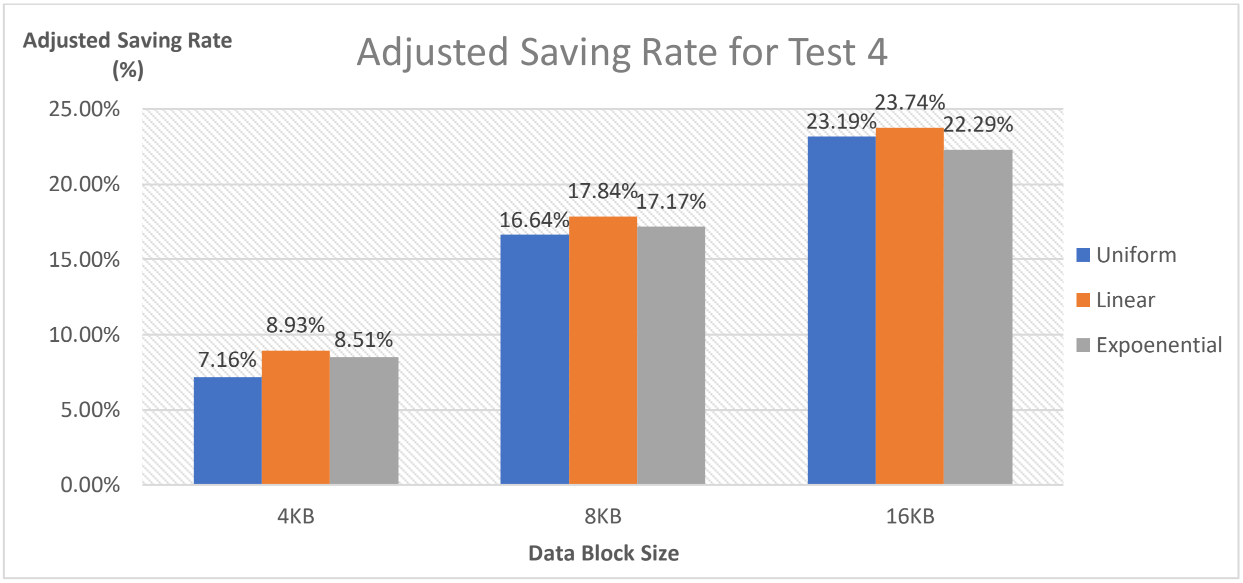


Figure 30: Space Benchmark Result for Test 4

In Figure 30, Test 4 shows that increasing data block size results in increasing saving rate. With the result in Test 1, we could draw a conclusion that with increasing number of entries within the data block, Sherry could save much more space.

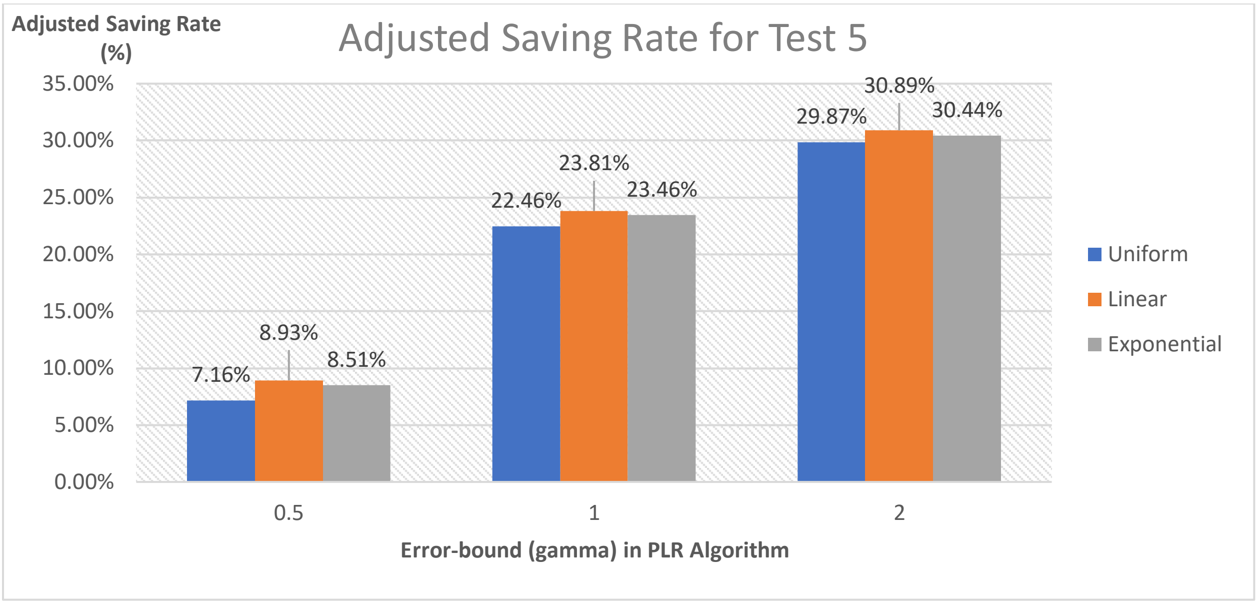


Figure 31: Space Benchmark Result for Test 5

As expected, in Figure 31, after increasing the error-bound, the saving rate in Test 5 increases. It is because fewer segments are generated in the building process. Yet, the marginal benefits are decreasing in increasing error bound.

To summarize, Sherry produces promising results in reducing space consumption compared with the conventional index block. The most critical factors in improving space performance are: (1) The number of key-value entries per data block, and (2) the error bound.

6. Discussion

6.1 Choice of learned index & database

Learned index is considered as a new research area, a limited number of learned indexes are proposed and implemented. This increases the difficulty for us to select a suitable index as our starting point.

After choosing Bourbon as our reference work, we would like to develop Sherry with LSM Tree, while suiting the need of commercial database. The databases which support LSM Tree are usually key-value, NoSQL database. However, the majority of commercial databases are in the form of relational, SQL database. We struggled to find a suitable database for our implementation. After pivoting several times, we decided to implement Sherry in RocksDB, as it has the broadest connection to the conventional SQL databases. Our developed index in RocksDB could be further connected to MariaDB to suit the needs of using relational database while enjoying the benefits of Sherry. As shown in the Table 17, both MariaDB and PostgreSQL can use RocksDB as the underlying storage engine. It can connect to RocksDB via Inter-Process Communication (IPC). Our work in RocksDB can be extended to other commercial databases, resulting in a wider application and usage in the future.

	MariaDB	Oracle SQL	MySQL	SQLite	PostgreSQL	Cassandra
Open source?	Yes	No	Yes	Yes	Yes	Yes
Use in RDBMS Context?	Yes	Yes	Yes	Yes	Yes	No
Use LSM Tree?	Yes*	No	No	No	No	Yes
Existing IPC to interact with RocksDB?	Yes	No	No	No	Yes	No

* Natively supports MyRocks, which uses RocksDB as the underlying storage engine

Table 17: Database Comparison Table

6.2 Implementation

As we modified the index block in SSTable, we needed to alter the index block related classes to ensure the operations related to the index blocks in Sherry are correct. In particular, PLRIndexBuilder, PLRIndexReader, PLRIndexIter classes are implemented. For translation between the PLR segments and the actual data blocks, the BlockHandleCalculator class is implemented. We faced some challenges while implementing the classes as RocksDB is a complex and massive database that contains a large number of libraries and classes, we have to review many of the existing classes to make sure we are using the classes and understand the semantics correctly. Also, it was difficult to verify if our implementation was reliable. We did unit test on the above classes to ensure the implementation of each class is correct, then we performed integration tests on the newly implemented classes to check if the classes are working as expected, and finally we performed stress test to verify the newly implemented classes are reliable by going through different crash scenarios.

6.3 Benchmarking

HKUST CSE COMPUTE service has been busy due to high demand. It took us more time than expected to perform benchmarks. We have also encountered some unexpected situations, such as reaching the storage limit of the server, limited number containers that can be run concurrently, and our testing progress ended due to running out of file descriptors. To overcome the issues, we switched to Google Cloud Platform (GCP) for a more consistent and reliable benchmarking environment.

Although GCP provides a more stable environment, we still encountered some difficulties. The virtual machines (VMs) that we use to perform benchmark have limited computational power. We faced the issue of running out of RAM in some of the benchmarks, as shown in Figure 47, Figure 48, Figure 49 from Appendix B: Time Performance Evaluation. Although we performed benchmarks of the default and Sherry on the same VM, giving fair results for comparison, we believe that the performance can be further improved by running on a machine without bottlenecks in computational resources.

7. Conclusion

7.1 Summary of Achievements

Sherry meets our objectives stated in Section 1.3. We designed a new learned index structure which supports read and write paths with the existing LSM Tree. Our team achieves the following in this project:

- Design a learned index Sherry in RocksDB:
  Thanks to the previous work, our team adopted the idea of Bourbon to design a new learned index structure by using the PLR Algorithm. Our team successfully replaced the index block and modified the write and read processes to make it compatible with RocksDB. By introducing new design and structure in the index block, we successfully developed a learned index with on-demand model fetching, and no constraints on the key data type. Sherry is fully migrated to RocksDB and have easy-to-use feature by switching the options when opening RocksDB.

- Space Gain using Sherry:
  The primary goal of using learned index is to improve the space performance while keeping the time performance comparable to the existing index. Sherry could save up to 30% of space compared to the conventional index. The space saving and boosted writing performance makes Sherry perform better in write-heavy workload, such as logging and the like, in enterprise level.

- Comprehensive Testing of Sherry:
  Our team conducts a set of comprehensive test cases to ensure Sherry will not cause any unexpected behavior in extreme circumstances. Unit tests and integration tests were carried out to make sure all subsystems work properly. Official stress tests have also been conducted to simulate the scenarios of sudden crash, multi-threading, and extreme read-write throughput. As we passed all the tests, we believe that Sherry is reliable to use in production.

- Performance benchmarking:
  To compare the result quantitively in a fair environment, our team planned a series of performance benchmarks in time and space. By comparing the performance with different factors, our team successfully interprets the two key factors dominating the space reduction. Even though it is relatively weak and inconclusive in the time performance benchmark, our team could see the correlation and relation between different tests. Yet, we

are currently unable to explain the root cause of the extreme cases in performance benchmark in Sherry. We hope that further investigations could explain the root cause.

7.2 Future Work

Even though Sherry has a great success in terms of space, it is far from perfect. There are some rooms of improvements in terms of time performance and certain limitations. As shown in the benchmark section, our time performance is inconsistent and require further investigation. Except the benchmarking issue, there are several directions that worth to research as a continuation of this project:

- Support of larger key size:
  Since the current limitation of Sherry on key size is 8 bytes, further research direction could dig deep on how to encode or structure the input keys to training points more efficiently such that more general key could be used, such as UUID.

- Hybrid Index:
  The PLR Algorithm used by Sherry has a relatively higher computational cost. To optimize the database by using Sherry, a hybrid index should be developed in the future. The hybrid index has a controller in which could identify whether there will be marginal benefits to use Sherry when building the SSTable based on the previous statistics. The computational time and space saved could be more flexible than the current implementation.

8. References

[1] Apache Cassandra, “Open source NoSQL database,” 2024. [Online]. Available: https://cassandra.apache.org/_/index.html

[2] D. Comer, "Ubiquitous B-Tree," ACM Computing Surveys, pp. 121-137, 1979.

[3] Y. Dai, Y. Xu, A. Ganesan, R. Alagappan, B. Kroth, A. Arpaci-Dusseau and R. Arpaci-Dusseau, "From WiscKey to Bourbon: A Learned Index for Log-Structured Merge Trees," in 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20), USENIX Association, 2020, pp. 155--171.

[4] MariaDB Foundation, “MariaDB,” 2024. [Online]. Available: https://mariadb.org/

[5] P. O’Neil, E. Cheng, D. Gawlick and E. O’Neil, "The log-structured merge-tree (LSM-tree)," Acta Informatica, vol. 33, no. 4, pp. 351-385, 1996.

[6] Oracle, “Cost-optimized high performance database,” 2024. [Online]. Available: https://www.oracle.com/database/

[7] Oracle, "mysql-server," 2023. [Online]. Available: https://github.com/mysql/mysql-server.

[8] PostgreSQL, "postgres," 2023. [Online]. Available: https://github.com/postgres/postgres/tree/master.

[9] SQLite, 2011. [Online]. Available: https://github.com/smparkes/sqlite/tree/master.

[10] Q. Xie, C. Pang, X. Zhou, X. Zhang, and K. Deng, “Maximum error-bounded piecewise linear representation for online stream approximation,” The VLDB Journal, vol. 23, no. 6, pp. 915–937, Apr. 2014. doi:10.1007/s00778-014-0355-0

9. Appendix

9.1 Appendix A: Meeting Minutes

Date: 19/08/23
Time: 22:00 – 22:30
Location: Zoom
Attendees: Alex, Andy, Jade, Oscar
Recorder: Jade

1. **Report on Progress**
    a. N/A for the first meeting

2. **Items Discussed**
    a. Teammates' background
    b. Communication plan
        i. Zoom for internal meetings
        ii. Face-to-face for meetings with supervisors
    c. Team norms
    d. Sources for materials related to the topic
        i. Database course notes online
        ii. Paper related to learned index
    e. Project scope
        i. Implement 1 learned index into a commercial database, can be existing one

3. **Goals Before Next Meeting**
    a. Read materials related to the topic and have a basic understanding of it
    b. Create a drive for storing documents and resources
    c. Schedule a meeting with supervisors

4. **Next Meeting**
    a. Meeting with supervisors for clarification
    b. Date: TBC
    c. Time: TBC
    d. Location: TBC

Date: 06/09/23
Time: 12:30 – 13:30
Location: Room 3555
Attendees: Prof. Papadias, Max, Alex, Andy, Jade, Oscar
Recorder: Andy

1. **Report on Progress**
    a. All members have read some materials related to database and learned index

2. **Items Discussed**
    a. Project scope
        i. Prof. Papadias mentioned that the topics is open, we can do whatever topics related to learned index and implement it into commercial database
        ii. Sherry needs to be on-disk instead of in main memory
    b. Choice of commercial database
        i. Focus on relational database
        ii. MySQL does not have good documentation
    c. Important algorithms to understand
        i. B tree
        ii. B+ Tree
        iii. LSM tree
    d. Possible approach
        i. Implement several learned indexes and compare their performance in commercial database
        ii. Create a new on disk learned index for commercial database
    e. Related literature
        i. Max provided a list of literatures that are related to learned index
        ii. Highlighted literatures that are the most important

3. **Goals Before Next Meeting**
    a. Read the literature list provided by Max
    b. Compare differences between learned index in main memory and on disk
    c. Understand existing indexes used by different databases
    d. Brainstorm ideas on the project approach

4. **Next Meeting**
    a. Internal Meeting
    b. Date: 09/09/23
    c. Time: 2100
    d. Location: Zoom

Date: 09/09/23
Time: 21:00 – 22:30
Location: Zoom
Attendees: Alex, Andy, Jade, Oscar
Recorder: Alex

1. **Report on Progress**
   a. All members finished reading the literature list provided by Max
   b. Brainstormed some ideas on possible approach based on the literatures

2. **Items Discussed**
   a. Features of existing learned indexes
      i. In main memory or on disk
      ii. Index Structure
      iii. Construction
      iv. Advantages and disadvantages
   b. Comparison of learned indexes
      i. Time and space complexity
      ii. Practical performance compared to B+ Tree
   c. Possible approaches
      i. Implement 1 learned index into commercial database and benchmark with B+ Tree
      ii. Implement Distribution-aware PGM index into commercial database
      iii. Implement and improve Learned LSM Tree index for commercial database
      iv. Comparison of existing learned indexes in commercial database

3. **Goals Before Next Meeting**
   a. Finish literature review part in the proposal report
   b. Set up a framework for the project
   c. Brainstorm more ideas on possible approaches

4. **Next Meeting**
   a. Meeting with Max
   b. Date: 13/09/23
   c. Time: 1400
   d. Location: RPG Hub

Date: 13/09/23
Time: 14:00 – 14:50
Location: RPG Hub
Attendees: Max, Alex, Andy, Jade, Oscar
Recorder: Oscar

1. **Report on Progress**
    - **a.** Finished literature review of proposal report
    - **b.** Sent email to Max for opinions on the brainstormed approaches

2. **Items Discussed**
    - **a.** Brainstormed approaches
        - **i.** Max suggested that we should focus on an index structure that can utilize distribution of data
        - **ii.** Sherry should assume data stored on disk, instead of main memory
        - **iii.** Not reasonable to adapt most of the existing learned index into commercial database directly due to the assumption
    - **b.** Brainstorming new ideas
        - **i.** Modify an existing learned index to be used on disk, with attention to the distribution of data, and implement it into a commercial database
        - **ii.** Using a hybrid approach, use a learned index if time below threshold, sis witch to B+ Tree otherwise
        - **iii.** Design a new on disk learned index, with specific use case (minimize query time/minimize space)

3. **Goals Before Next Meeting**
    - **a.** Finish the proposal report and submit it by 15/9
    - **b.** Finalize choice of commercial database

4. **Next Meeting**
    - a. Internal Meeting
    - b. Date: 8/10/23
    - c. Time: 2100
    - d. Location: Zoom

Date: 8/10/23
Time: 21:00 – 22:30
Location: Zoom
Attendees: Alex, Andy, Jade, Oscar
Recorder: Jade

1. **Report on Progress**
    a. Reviewed discussion with Max, decided to adopt the approach of modifying an existing learned index
    b. Chose PostgreSQL as target database
    c. Found learned index candidate: Wisckey
2. **Items Discussed**
    a. Interim report
        i. Finalized contents to be included in the interim report
        ii. Discussed the division of labour and set the internal deadline
    b. October TODO
        i. Investigate implementation of LSM Tree in PostgreSQL
        ii. Differentiate Wisckey and LSM Tree
        iii. Investigate the infrastructure of PostgreSQL to facilitate CICD
3. **Goals Before Next Meeting**
    a. Finish the assigned parts in the interim report
    b. Understand PostgreSQL LSM Tree implementation and its difference with Wisckey
4. **Next Meeting**
    a. Internal Meeting
    b. Date: 23/10/23
    c. Time: 2130
    d. Location: Zoom

Date: 23/10/23
Time: 21:30 – 22:30
Location: Zoom
Attendees: Alex, Andy, Jade, Oscar
Recorder: Andy

1. **Report on Progress**
    a. Finished all assigned parts in the interim report
    b. Summarized features of Wisckey
    c. Found more suitable candidate: Bourbon
2. **Items Discussed**
    a. Learned index
        i. Wisckey employs LSM system but based on SSD.
        ii. Bourbon serves as the successor to Wisckey, being disk-resident as well. Need further investigation into the feasibility of implementing it.
    b. Poster
        i. Discussed contents to be included in the poster
        ii. Discussed division of labour and set the internal deadline
    c. Monthly report
        i. Summarized progress over the previous month
        ii. Finished the report during meeting
    d. Development environment
        i. Need to test for how to compile PostgreSQL
3. **Goals Before Next Meeting**
    a. Investigate the possibility of implementing Bourbon into a commercial database
    b. Setup development environment
4. **Next Meeting**
    a. Meeting with Max
    b. Date: 15/11/23
    c. Time: 0930
    d. Location: RPG Hub

Date: 15/11/23
Time: 09:30 – 10:30
Location: RPG Hub
Attendees: Max, Alex, Andy, Jade, Oscar
Recorder: Alex

1. **Report on Progress**
    a. Changed to LSM Tree-based storage engine (PostgreSQL)
    b. Sent email to Max for opinions on the brainstormed approaches
2. **Items Discussed**
    a. Brainstormed approaches
        i. Informed Max regarding our current planned approach, which try to use FDW wrapper to incorporate Bourdon implemented into RocksDB with PostgreSQL
        ii. Discussed the feasibility of making our approach ACID-compliant.
    b. Feedback from Max
        i. Need to show evidence that integrating NoSQL to relational database will show improvement
        ii. If no existing relational database uses LSM Tree natively, we can directly implement it
        iii. Keep minimal steps being involved.
        iv. Acceptable if showing good result in some dataset and not strictly worse than others
3. **Goals Before Next Meeting**
    a. Investigate if there are results showing that integrating NoSQL to relational database results in performance improvement
    b. Look for LSM Tree implementations in existing relational database
4. **Next Meeting**
    e. Internal meeting
    f. Date: 25/11/23
    g. Time: 2130
    h. Location: Zoom

Date: 25/11/23
Time: 21:30 – 22:30
Location: Zoom
Attendees: Alex, Andy, Jade, Oscar
Recorder: Oscar

1. **Report on Progress**
    a. Found MyRocks, a relational database that includes LSM Tree implementation
    b. Studied LevelDB, which is similar to RocksDB.
2. **Items Discussed**
    a. Database choice
        i. Discussed the feasibility of switching to MyRocks, which saves the step of using FDW wrapper to bridge RocksDB and PostgreSQL.
    b. Learned index implementation
        i. Investigated the implementation of Bourbon. Existing implementation is not well commentated. Need more time to interpret how to implement it as external modules
3. **Goals Before Next Meeting**
    a. Map LevelDB's function with RocksDB's function
    b. Investigate where should the learned model be stored
    c. Implement PLR module from Bourbon as an external module
4. **Next Meeting**
    a. TBC due to final period

Date: 21/12/23
Time: 22:00 – 23:30
Location: Zoom
Attendees: Alex, Andy, Jade, Oscar
Recorder: Jade

1. **Report on Progress**
    a. Established mapping of functions between LevelDB and RocksDB
    b. Studied storage format of RocksDB for interpreting possible locations to store learned models
    c. Implemented PLR module
2. **Items Discussed**
    a. Learned model storage location
        i. Storing in existing SSTable might not be optimal as blocks are accessed based on offset
        ii. Need to explore possible ways to implement it outside of SSTable
    b. PLR module
        i. Implemented PLR based on reference found on Github
        ii. Need to perform unit test
3. **Goals Before Next Meeting**
    a. Design the module that stores learned models, with interface that connects with corresponding SSTable
    b. Perform unit test on PLR module
4. **Next Meeting**
    a. Meeting with Max
    b. Date: 22/1/24
    c. Time: 1030
    d. Location: RPG Hub
    e. Note: Most of us not in HK until mid-January. No internal meetings, share progress on WhatsApp instead.

Date: 22/1/23
Time: 10:30 – 12:00
Location: RPG Hub
Attendees: Max, Alex, Andy, Jade, Oscar
Recorder: Andy

1. **Report on Progress**
    a. Confirmed using RocksDB as target commercial database
    b. Studied Bourbon implementation detailedly
    c. PLR implementation is not working as expected, trying to identify the problem currently
    d. MariaDB is used to run RocksDB instead of using MyRocks due to difficulties when compilating
2. **Items Discussed**
    a. Brainstormed approaches
        i. Informed max regarding our progress since last meeting
        ii. Remove cost-benefit analyzer first, investigate the performance difference
        iii. Replacing existing index block to store the PLR model parameters.
    b. Feedback from Max
        i. Predict data block instead of exact offset might save model space
        ii. Should investigate into the decision rule when predicted range spread across multiple data blocks (whether checking previous or next data block first)
3. **Goals Before Next Meeting**
    a. Investigate the possibility to predict data block instead of exact offset, which is different from Bourbon's original implementation
    b. Evaluate the space gained from predicting data block instead of exact offset
    c. Fix PLR module's error
    d. Perform and pass unit test for PLR module
4. **Next Meeting**
    a. TBC

Date: 11/4/23
Time: 4:15 – 5:00
Location: RPG Hub
Attendees: Max, Alex, Andy, Jade, Oscar
Recorder: Jade

5. **Report on Progress**
    a. All stress test passed
    b. Moving on to benchmarking
    c. Interpretation of the benchmark result
6. **Items Discussed**
    a. Interpretation of benchmark result
        i. median of read latency is performing better than default
        ii. Max read latency is performing worse than default
        iii. What data should be used to benchmark our result (min, median, max, 99 percentile)
    b. Feedback from Max
        i. There is no standard data in the representing read or write latency, we can reference from published paper
7. **Goals Before Next Meeting**
    a. Finish final report
    b. Get significant result from benchmarking
    c. Get significant result from testing
8. **Next Meeting**
    a. 16/4/2024

Date: 16/4/23
Time: 21:30 – 23:00
Location: via discord
Attendees: Alex, Andy, Jade, Oscar
Recorder: Jade

1. **Report on Progress**
    a. Divided workload on final report
    b. First Draft on Final Report
2. **Items Discussed**
    a. Test result
        i. Read latency perform worse than default
        ii. How to represent the results (line chart or bar chart should be used)
        iii. Storage of index blocks is reduced significantly, but we have to double confirm the result
    b. Feedback on final report
        i. More diagrams should be implemented to illustrate the concept of Bourbon and PLR and our design
        ii. Simplify our wordings to make the final report more comprehensible
3. **Goals Before Next Meeting**
    a. Finish amending final report and sent it to Max for feedback
    b. Think about the oral presentation
4. **Next Meeting**
    a. TBC

9.2 Appendix B: Time Performance Evaluation

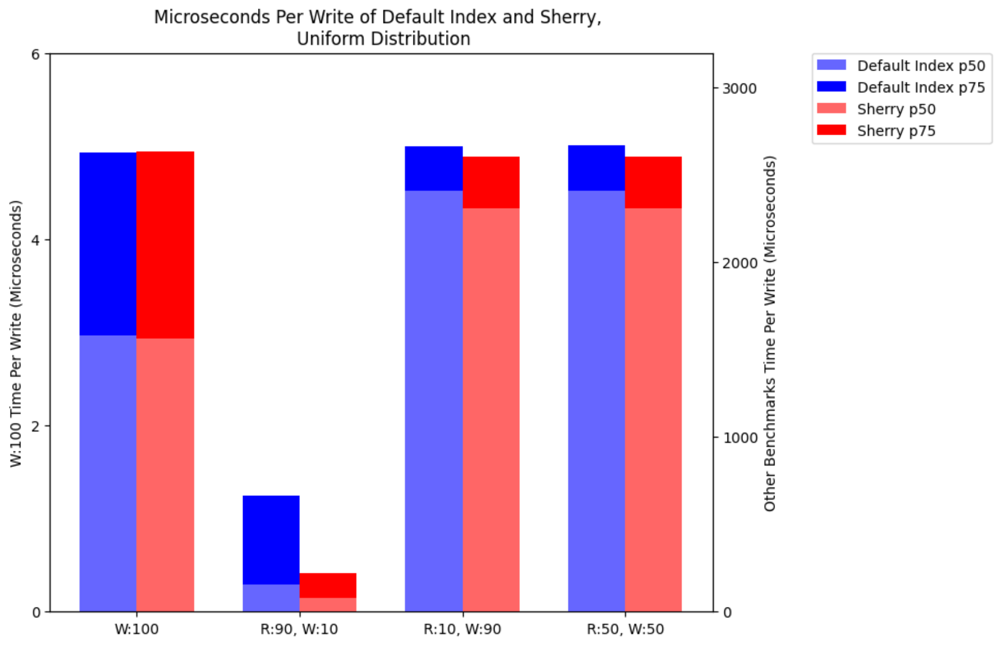


Figure 32: Comparison of Microseconds Per Write of Default and Sherry at P50 & P75, Uniform Key Distribution

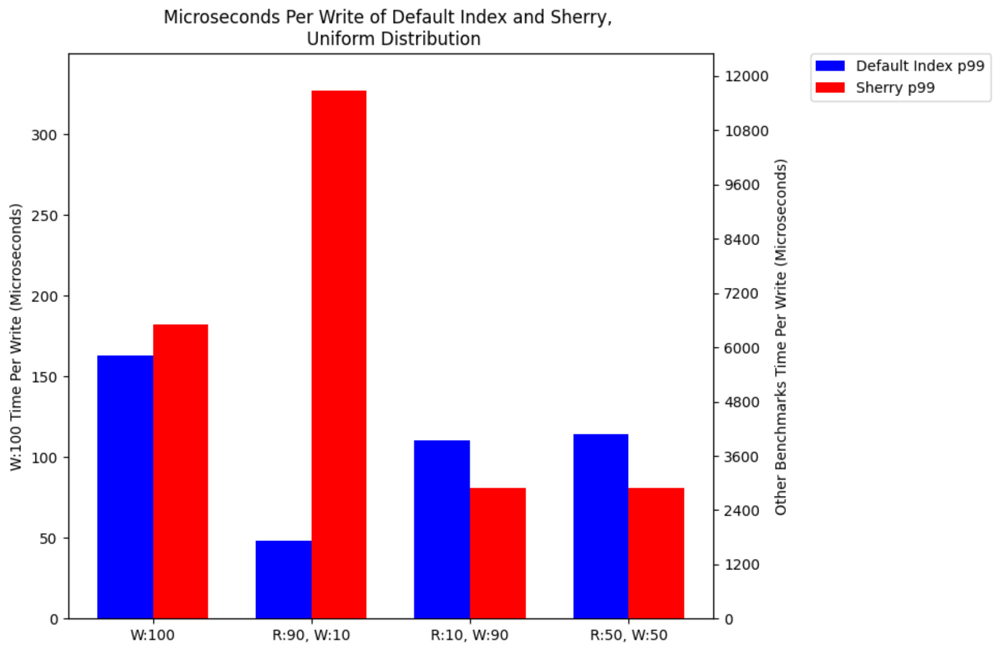


Figure 33: Comparison of Microseconds Per Write of Default and Sherry at P99, Uniform Key Distribution

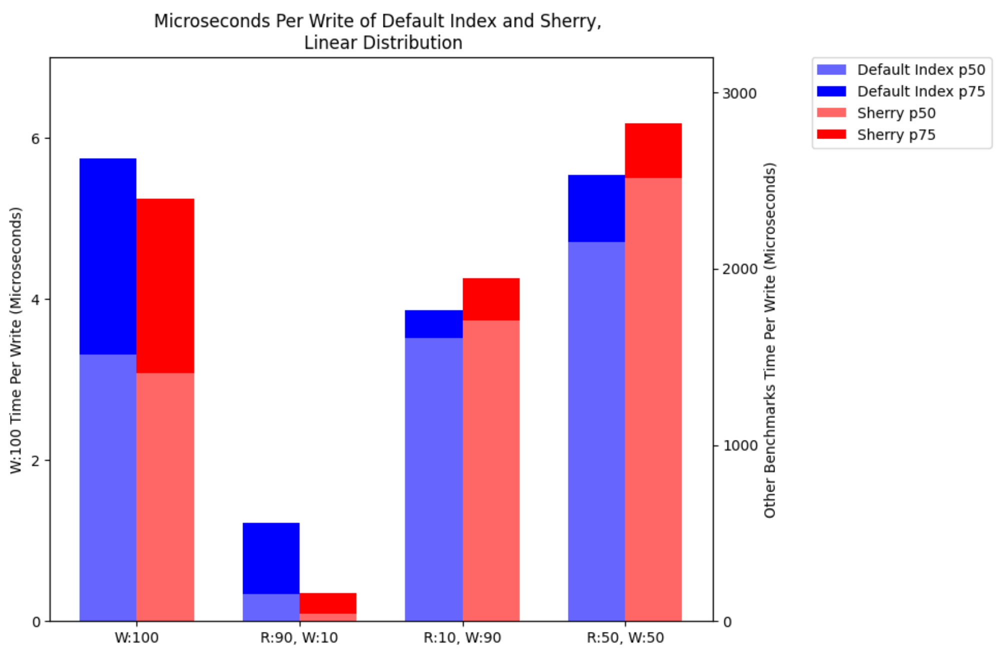


Figure 34: Comparison of Microseconds Per Write of Default and Sherry at P50 & P75, Linear Key Distribution

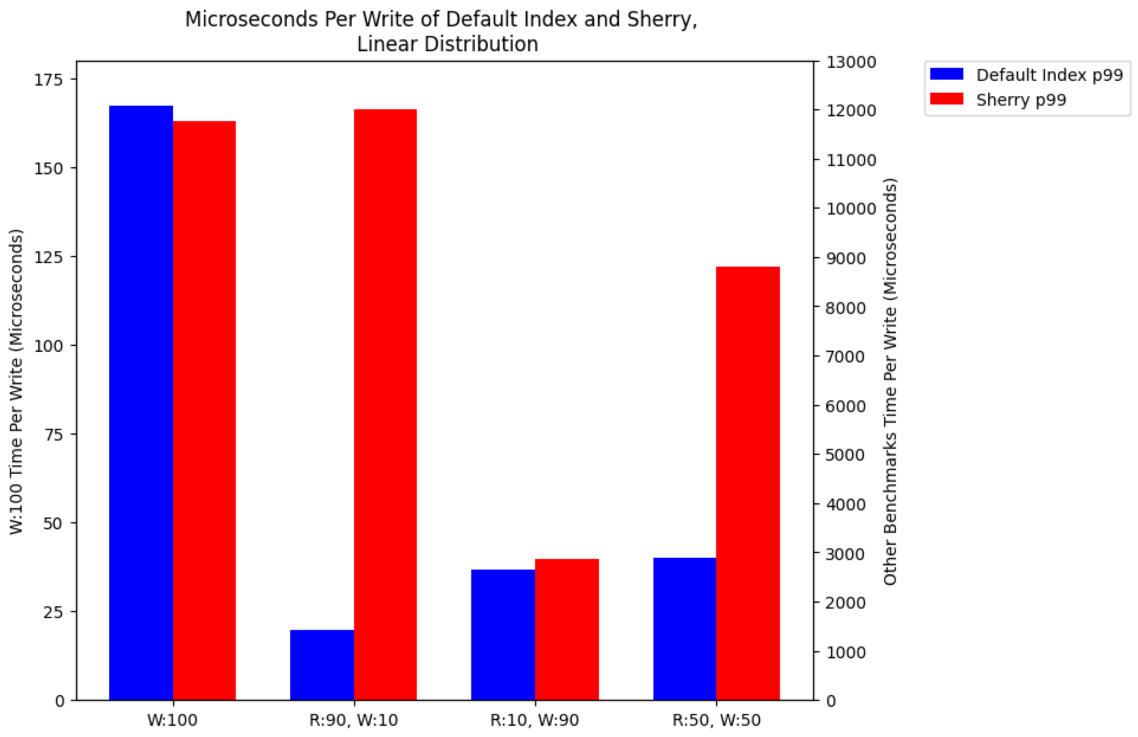


Figure 35: Comparison of Microseconds Per Write of Default and Sherry at P99, Linear Key Distribution

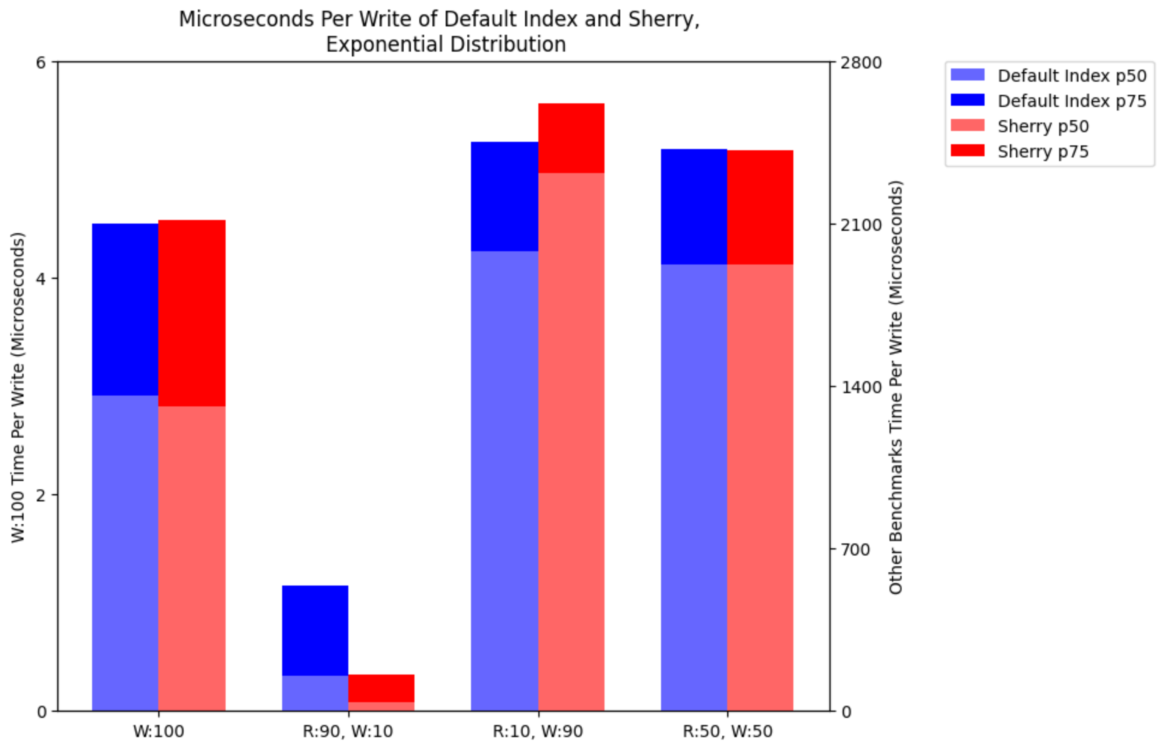


Figure 36: Comparison of Microseconds Per Write of Default and Sherry at P50 & P75, Exponential Key Distribution

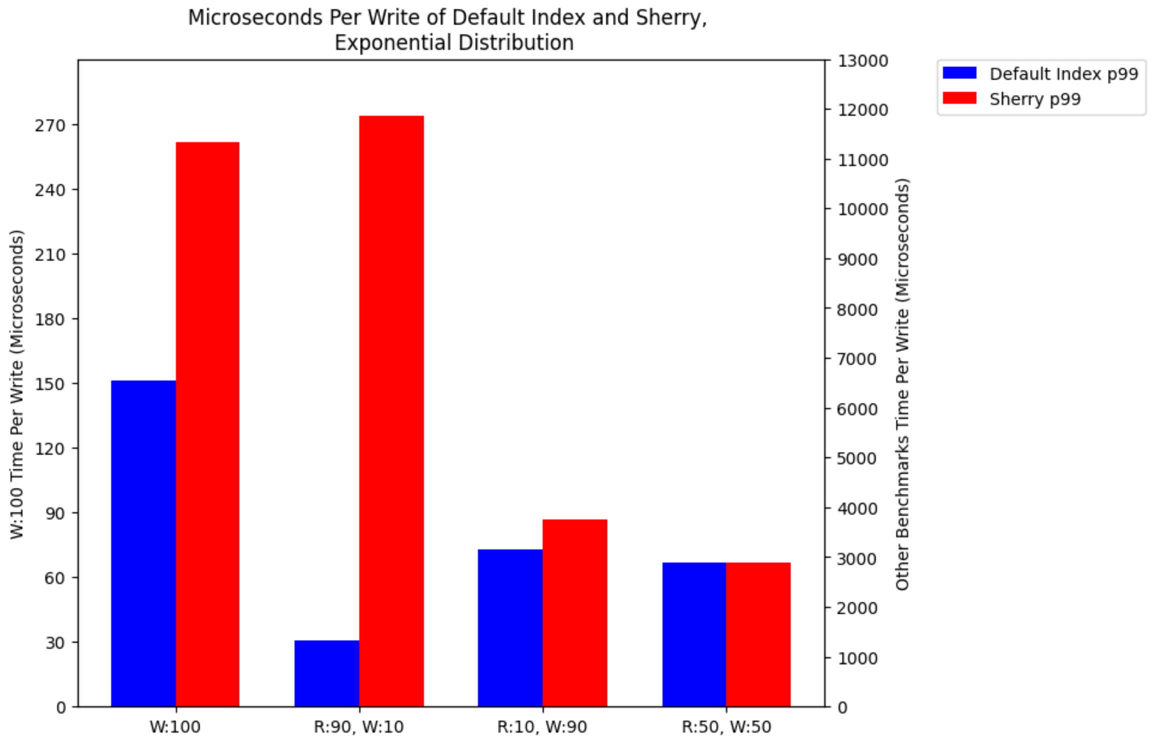


Figure 37: Comparison of Microseconds Per Write of Default and Sherry at P99, Exponential Key Distribution

Microseconds per Read

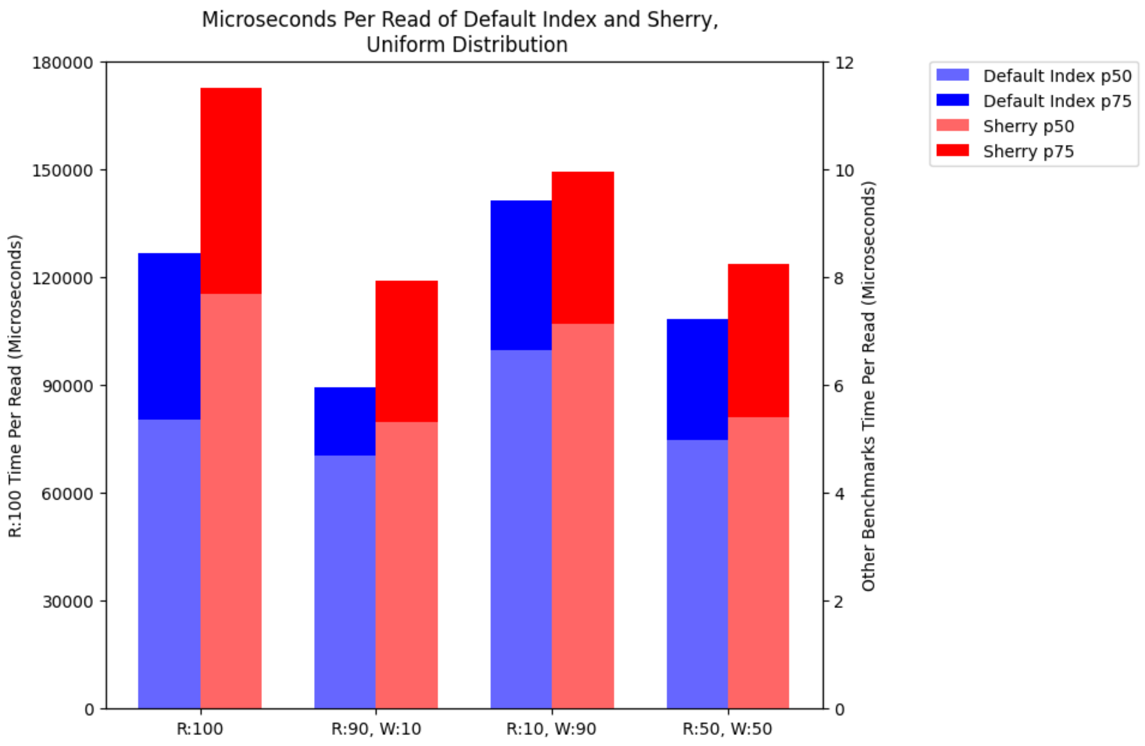


Figure 38: Comparison of Microseconds Per Read of Default and Sherry at P50 & P75, Uniform Key Distribution

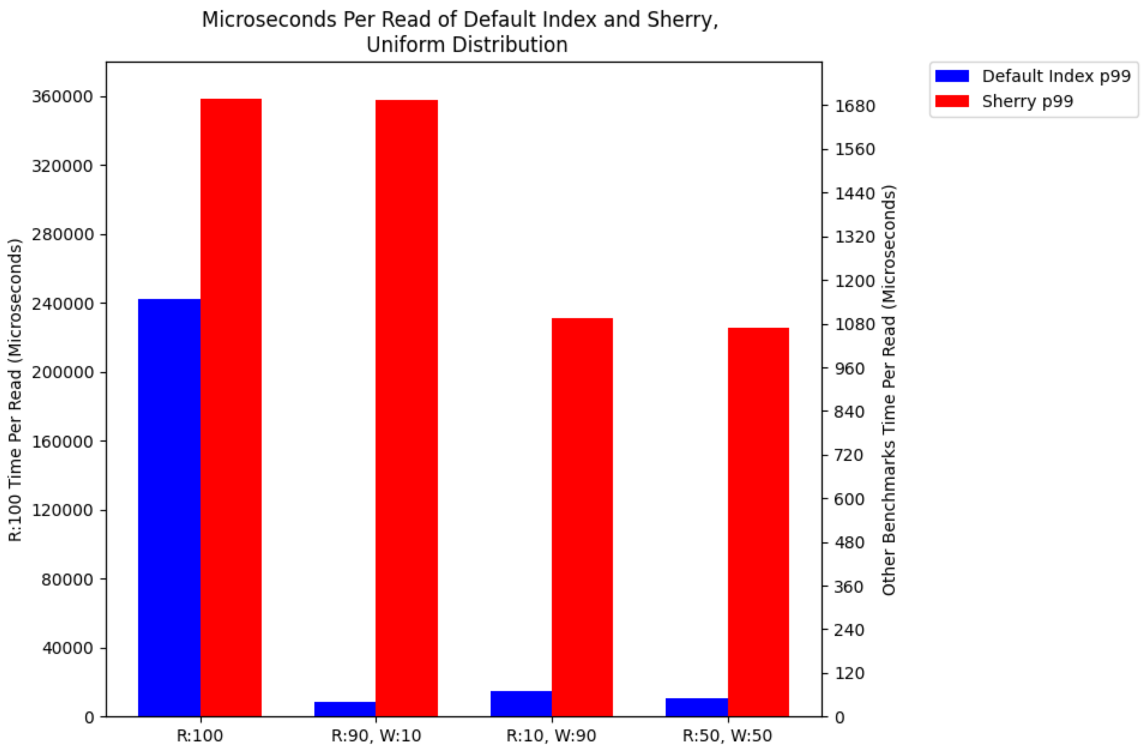


Figure 39: Comparison of Microseconds Per Read of Default and Sherry at P99, Uniform Key Distribution

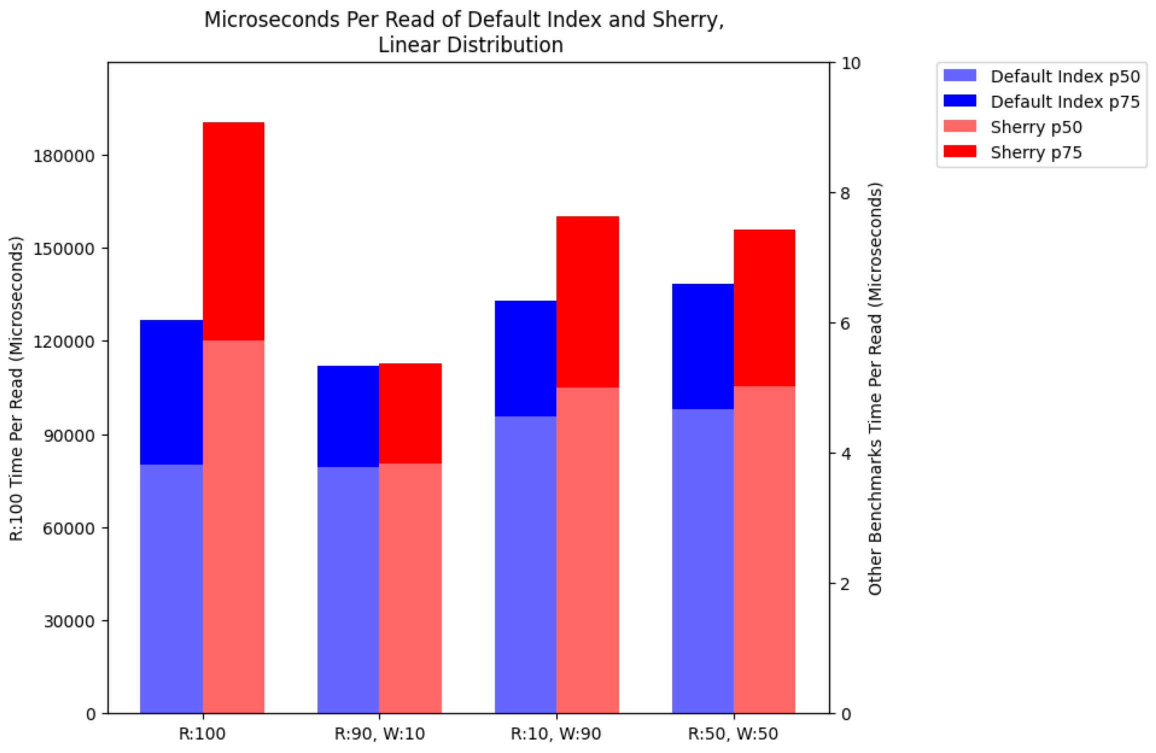


Figure 40: Comparison of Microseconds Per Read of Default and Sherry at P50 & P75, Linear Key Distribution

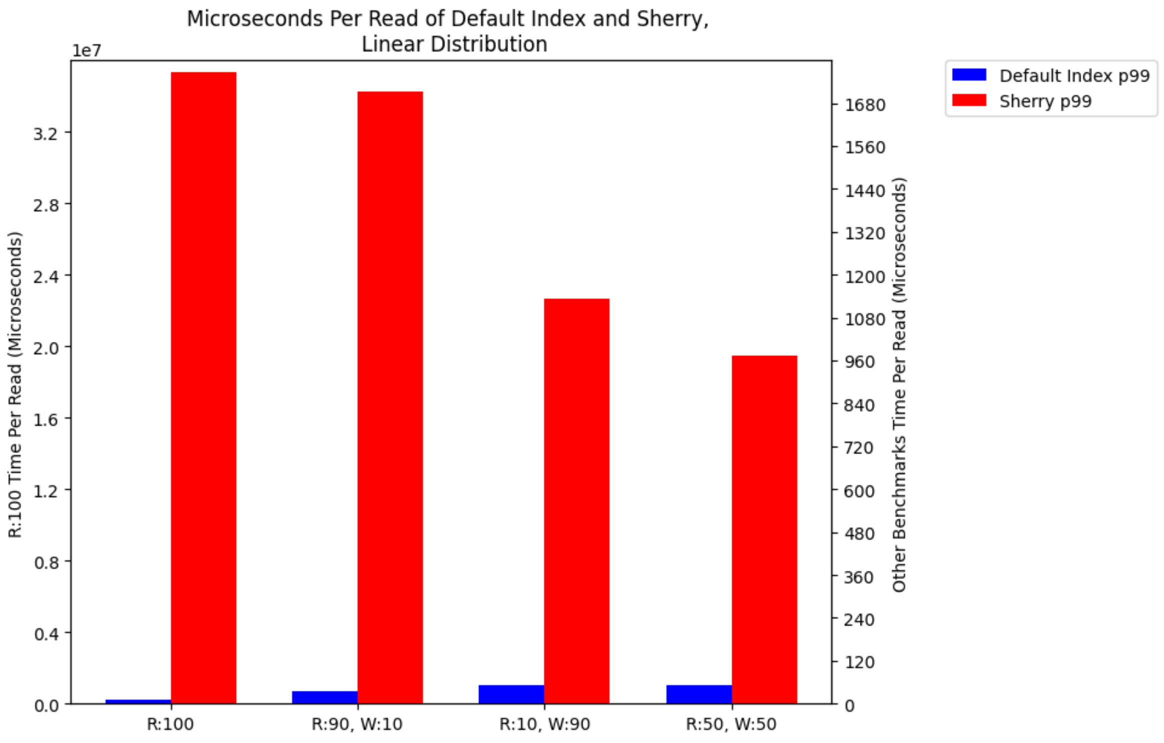


Figure 41: Comparison of Microseconds Per Read of Default and Sherry at P99, Linear Key Distribution

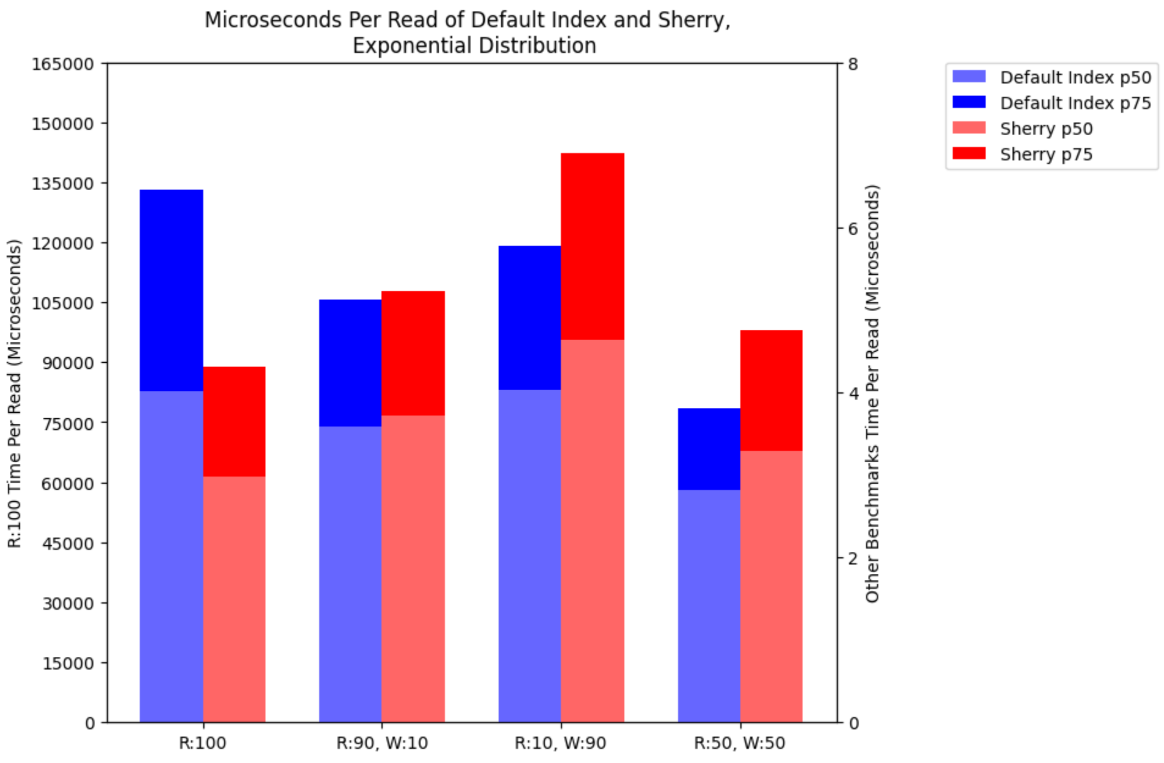


Figure 42: Comparison of Microseconds Per Read of Default and Sherry at P50 & P75, Exponential Key Distribution

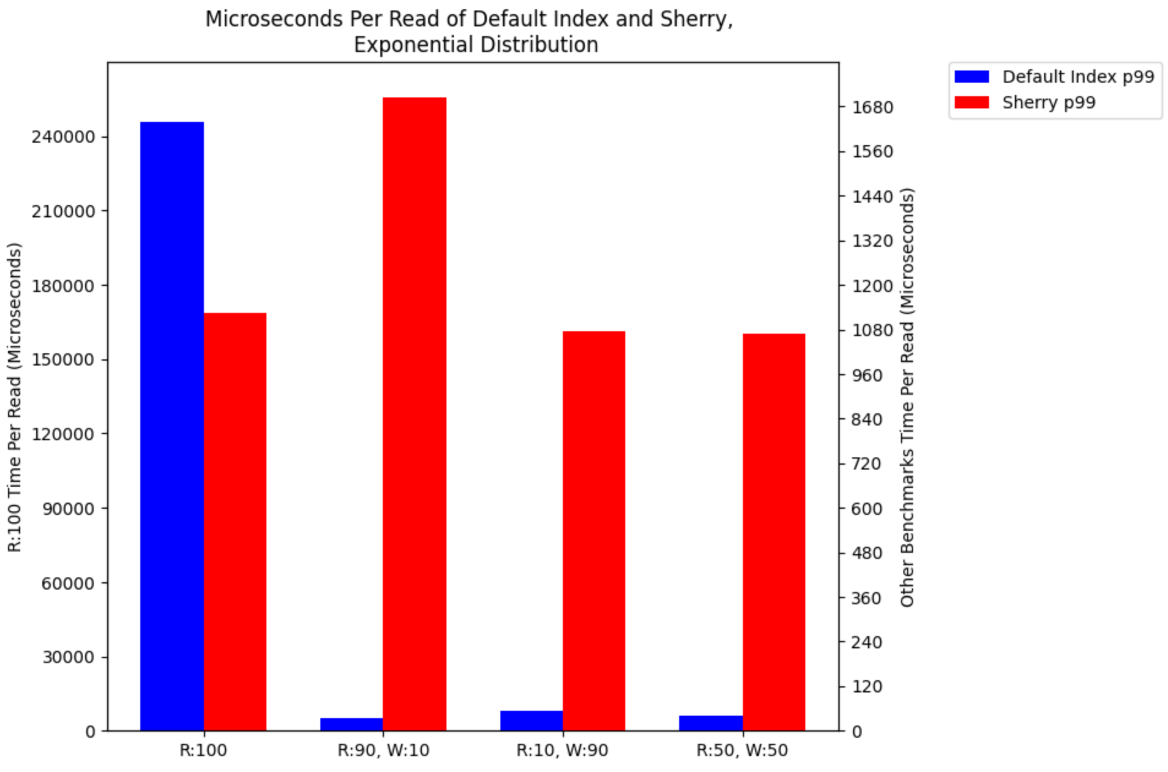


Figure 43: Comparison of Microseconds Per Read of Default and Sherry at P99, Exponential Key Distribution

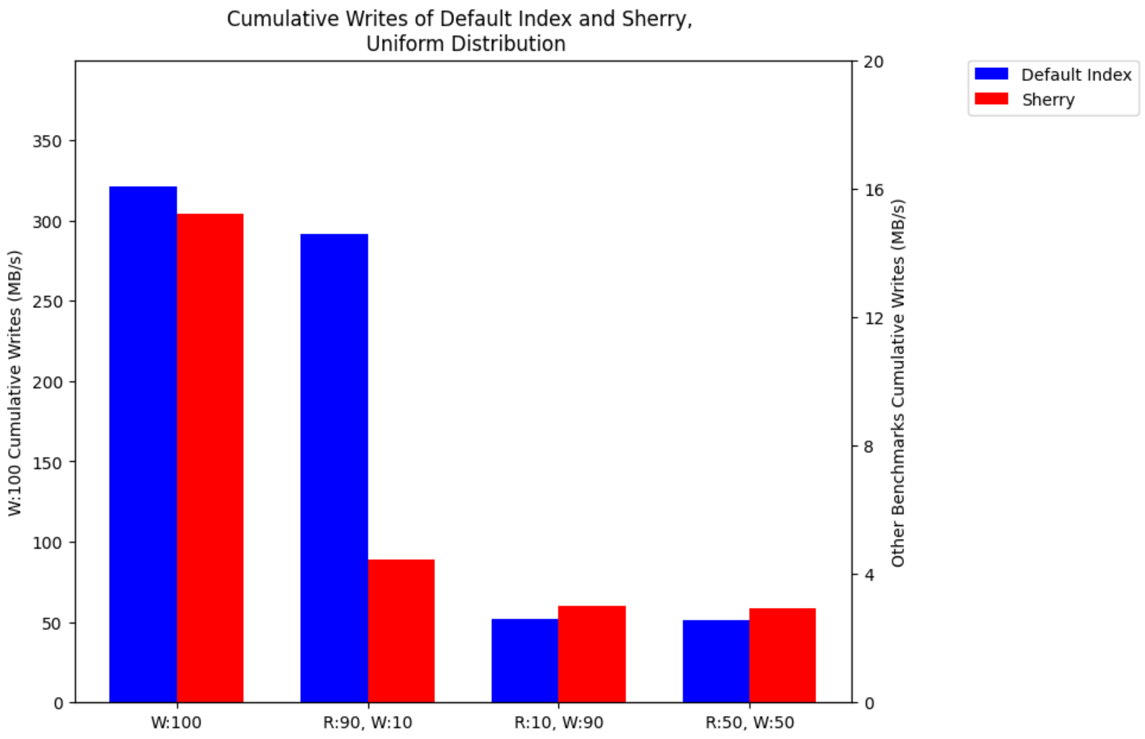


Figure 44: Comparison of Cumulative Writes of Default and Sherry, Uniform Key Distribution

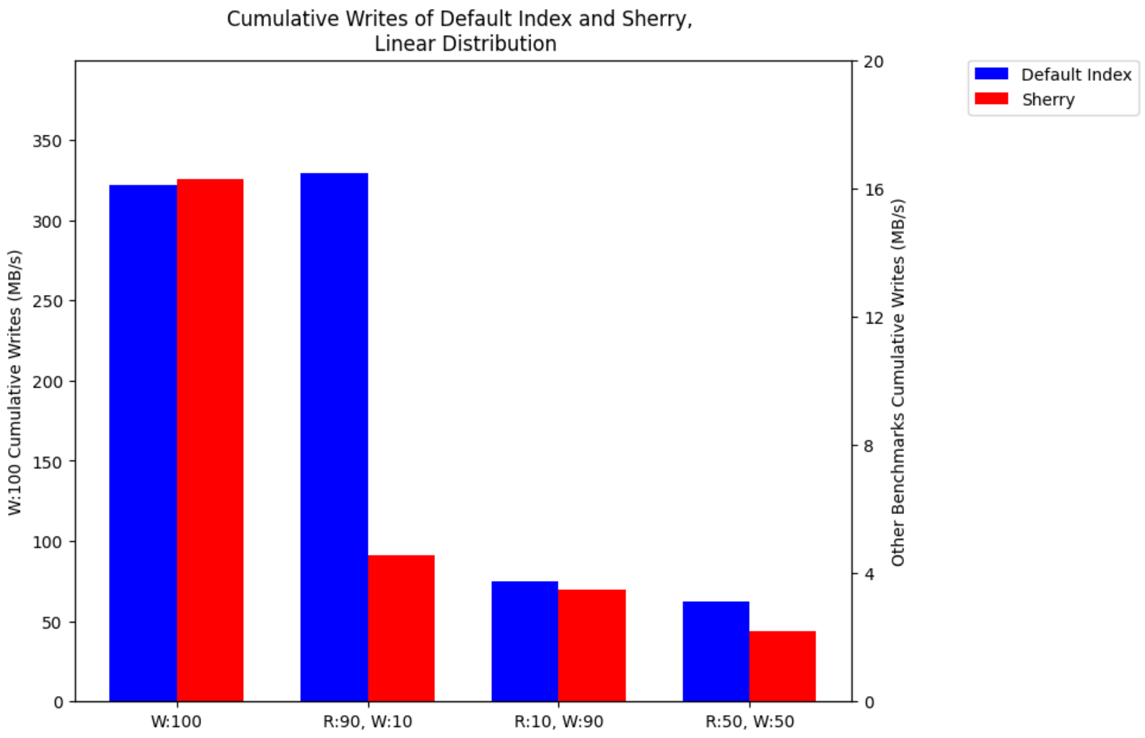


Figure 45: Comparison of Cumulative Writes of Default and Sherry, Linear Key Distribution

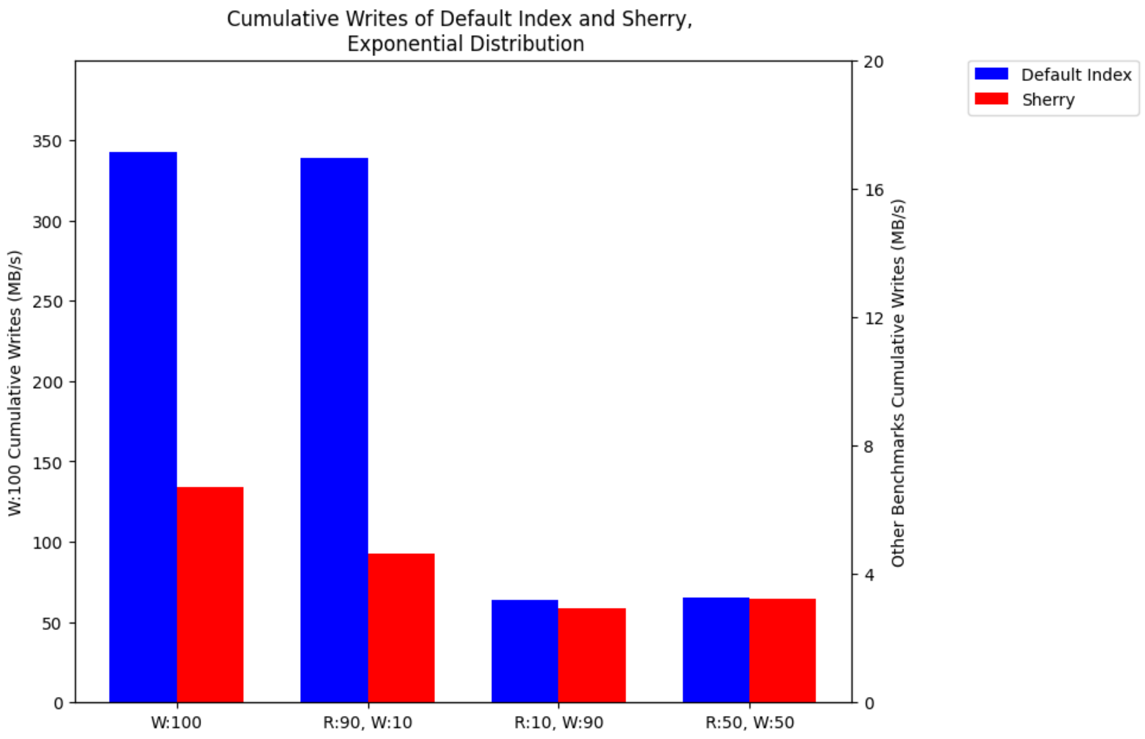
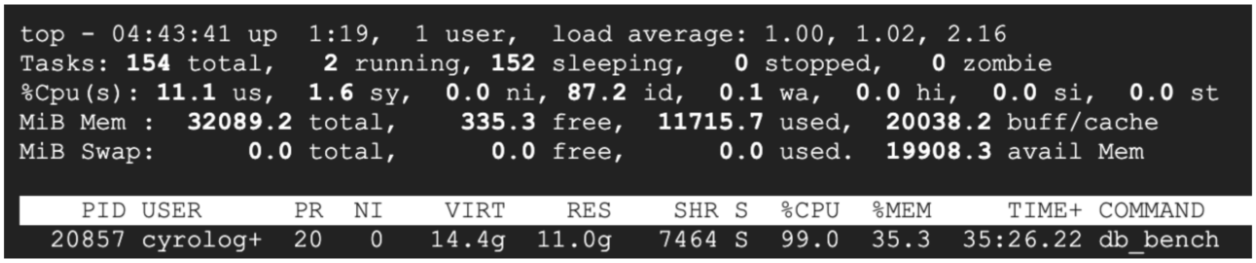


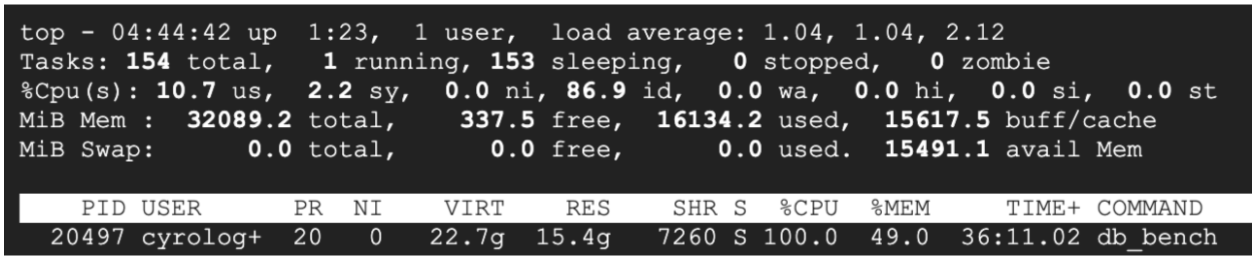
Figure 46: Comparison of Cumulative Writes of Default and Sherry, Exponential Key Distribution



```
top - 04:43:41 up  1:19,  1 user,  load average: 1.00, 1.02, 2.16
Tasks: 154 total,   2 running, 152 sleeping,   0 stopped,   0 zombie
%Cpu(s): 11.1 us,  1.6 sy,  0.0 ni, 87.2 id,  0.1 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :  32089.2 total,    335.3 free,  11715.7 used,  20038.2 buff/cache
MiB Swap:      0.0 total,      0.0 free,      0.0 used.  19908.3 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
  20857 cyrolog+  20   0   14.4g  11.0g   7464 S  99.0  35.3  35:26.22 db_bench
```

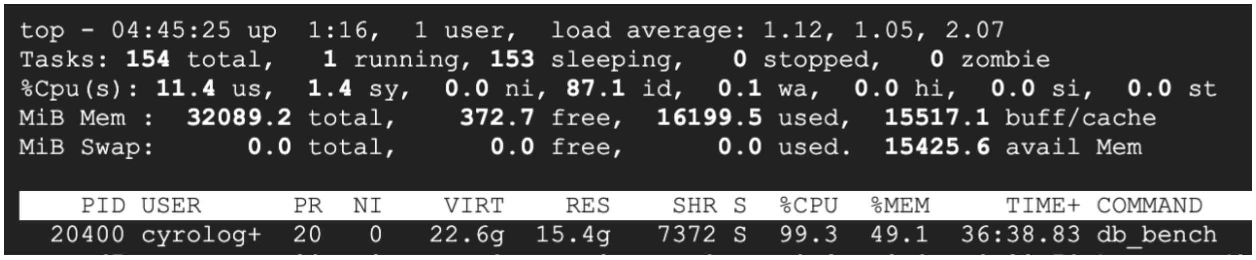
Figure 47: System Resource Usage on GCP when Running benchmarks with Uniform Key Distribution



```
top - 04:44:42 up  1:23,  1 user,  load average: 1.04, 1.04, 2.12
Tasks: 154 total,   1 running, 153 sleeping,   0 stopped,   0 zombie
%Cpu(s): 10.7 us,  2.2 sy,  0.0 ni, 86.9 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :  32089.2 total,    337.5 free,  16134.2 used,  15617.5 buff/cache
MiB Swap:      0.0 total,      0.0 free,      0.0 used.  15491.1 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
  20497 cyrolog+  20   0   22.7g  15.4g   7260 S 100.0  49.0  36:11.02 db_bench
```

Figure 48: System Resource Usage on GCP when Running benchmarks with Linear Key Distribution



```
top - 04:45:25 up  1:16,  1 user,  load average: 1.12, 1.05, 2.07
Tasks: 154 total,   1 running, 153 sleeping,   0 stopped,   0 zombie
%Cpu(s): 11.4 us,  1.4 sy,  0.0 ni, 87.1 id,  0.1 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :  32089.2 total,    372.7 free,  16199.5 used,  15517.1 buff/cache
MiB Swap:      0.0 total,      0.0 free,      0.0 used.  15425.6 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
  20400 cyrolog+  20   0   22.6g  15.4g   7372 S  99.3  49.1  36:38.83 db_bench
```

Figure 49: System Resource Usage on GCP when Running benchmarks with Exponential Key Distribution