Learning Z-Order Indexes with Dynamic Bit Allocation

by

Jenny Gao

S.B. Computer Science and Engineering, Mathematics
Massachusetts Institute of Technology, 2022

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2023

© Massachusetts Institute of Technology 2023. All rights reserved.

Author . . . .
Department of Electrical Engineering and Computer Science
January 20, 2023

Certified by . . . .
Samuel Madden
College of Computing Distinguished Professor of Computing
Thesis Supervisor

Accepted by . . . .
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Learning Z-Order Indexes with Dynamic Bit Allocation

by

Jenny Gao

Submitted to the Department of Electrical Engineering and Computer Science
on January 20, 2023, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

The Z-order curve is a space-filling curve that maps multi-dimensional data to single-dimensional values. Z-order has been used in databases to sort multi-dimensional data. Modern data management systems such as Amazon Redshift and Databricks Delta Lake give users the ability to sort on multiple columns using Z-order. However, the Z-order is difficult to tune, with tunable parameters such as which columns to include in the Z-order. Currently, users must specify the columns for Z-order when using the systems and might not necessarily achieve the best performance, as the choice of columns has a significant impact on performance. Another drawback is that the systems give equal weight to the columns, which often does not result in the best performance due to the unequal impact columns have on query performance. Our work aims to automatically determine the best Z-order configuration for a particular dataset and workload. In this thesis, we introduce learning Z-order indexes using an approach we refer to as dynamic bit allocation, which considers not only which columns to include, but also the weight to put on each column. Our learned Z-order indexes outperform existing techniques by up to 11× in query time and up to 30.2× in rows scanned, revealing the potential of tuning Z-order to improve query performance.

Thesis Supervisor: Samuel Madden
Title: College of Computing Distinguished Professor of Computing

Acknowledgments

First and foremost, I would like to thank my advisor, Professor Madden, for his support and guidance throughout the course of this thesis, without whom this work would not have been possible. I am also extremely grateful to my mentors Jialin and Siva for their invaluable advice and guidance throughout the entire research process.

Finally, I would like to thank my family for their encouragement and endless support.

Contents

1 Introduction **13**

2 Related Work **21**

3 Overview **25**

3.1 Problem Statement and Approach 27

3.1.1 Cost Model 27

3.1.2 Search Algorithms 28

3.2 Implementation 29

3.2.1 Z-Order 29

3.2.2 Search Algorithms 30

3.2.3 Recursive Cell Splits 30

4 Evaluation **33**

4.1 Baselines 34

4.2 Datasets and Query Workloads 35

4.3 Evaluation Setup 37

4.4 Results 37

4.4.1 Query Performance 37

4.4.2 Discussion 40

4.4.3 Search Algorithms 42

4.4.4 Z-Order Index Creation 44

4.4.5 Workloads in which Z-Order Indexes Work Well 44

4.4.6 Varying Total Number of Bits in Z-Order 48

5 Conclusion and Future Work **53**

List of Figures

1-1 Examples of Z-order configurations and data layouts 17

2-1 Examples of the Z-order curve and Z-order indexes 24

3-1 Performance of uneven bit allocation vs. equal bit allocation for synthetic dataset and query workload 26

3-2 Statistics obtained from sample dataset vs. entire dataset 28

4-1 Average number of rows scanned for the different indexing/partitioning methods 39

4-2 Average query time for the different indexing/partitioning methods . 40

4-3 Actual vs. estimated query time using the cost model 42

4-4 Lowest query time seen when running search algorithms 43

4-5 Z-order performance at varying number of blocks 47

4-6 Z-order performance at varying query selectivity 48

4-7 Z-order performance at varying number of columns included in the Z-order 48

4-8 Lowest query time seen for a specific number of bits in the Z-order value 50

List of Tables

3.1 Synthetic uniform workload 26

4.1 Dataset and query characteristics 36

4.2 Best 64-bit Z-order index creation time in seconds 44

4.3 k-bit Z-order index learning time in hours 44

4.4 Scan overhead for the different indexing methods 46

Chapter 1

Introduction

Scanning and filtering are important aspects of analytical databases. Several methods have been proposed for improving scan performance, such as dividing tables into contiguous *blocks* and using data structures called *zone maps*, which contain metadata such as minimum/maximum values per column, to skip blocks that are not relevant to a query [4]. Database systems can also organize the data according to an attribute by sorting on that attribute or creating a B-tree index on that attribute to quickly access records in the table.

To help with queries that filter over more than one column, multi-dimensional indexes or specialized sort orders can be used. Tree-based data structures include R-trees and k-d trees [13]. In addition, recent works have proposed multi-dimensional sorting schemes; one technique involves a multi-dimensional space-filling curve called the Z-order. Z-order is particularly suitable for sorting multi-dimensional data since it maps multi-dimensional data to one dimension while preserving locality of the data points: points that were close together in the multi-dimensional space would still be close to each other on a one-dimensional line after sorting by the Z-order values. The Z-order value for a record is calculated by interleaving the bits of the bitstring representation of the column values (see Definitions 1 to 3 below). Database systems can use Z-order to improve query performance by creating an index on the Z-order values or sorting on the Z-order values.

Definition 1 (Z-Order Value). *Consider a tuple* $(c_0, c_1, c_2, ..., c_{n-1})$*, where* c_i *is the value of column* i*. Let* $c_i[j]$ *be the* j*-th bit of the bitstring representation of column* i*, where the most significant nonzero bit has index 0 (i.e.,* $j = 0$*), and suppose each column value has* m *bits. The Z-order value is formed by interleaving the bits of the columns in round robin fashion and can have less than or equal to* $m * n$ *bits.*

Definition 2 (Z-Order Configuration). *We define a Z-order configuration to be an allocation of bits to columns. Suppose we have columns 0 to* $n - 1$*. A Z-order configuration can be written as an ordered list of key-value pairs: {*col_0*:* v_0*,* col_1*:* v_1*, ...,* col_{n-1}*:* v_{n-1}*}, where* v_i *denotes the number of "interesting" most significant bits (i.e., the bits starting from position* u*, where* u *is the index of the most significant nonzero bit of the largest* col_i *value in the dataset) from* col_i *to use in the Z-order bit interleaving. Placing more weight on a column is equivalent to having a higher* v_i *value for a column.*

Definition 3 (k-bit Z-Order Value). *Consider a Z-order configuration {*col_0*:* v_0*,* col_1*:* v_1*, ...,* col_{n-1}*:* v_{n-1}*}, where* $v_0 + v_1 + ... + v_{n-1} = k$*. Given a tuple of* n *column values* $(c_0, c_1, c_2, ..., c_{n-1})$*, the k-bit Z-order value can be constructed by evaluating* $m = min(c_0, c_1, c_2, ..., c_{n-1})$ *and interleaving* $\lfloor c_i/m \rfloor$ *bits of each column at a time. The order of the columns in the interleaving is in order of increasing column number, i.e.,* col_0*,* col_1*, ...,* col_{n-1}*.*

Example. *Consider the Z-order configuration {*col_0*: 2,* col_1*: 11,* col_2*: 7}. Here,* $m = min(2, 11, 7) = 2$*. We can construct the Z-order value by interleaving* $\lfloor 2/2 \rfloor = 1$ *bit of* col_0*,* $\lfloor 11/2 \rfloor = 5$ *bits of* col_1*, and* $\lfloor 7/2 \rfloor = 3$ *bits of* col_2 *at a time. Consider a tuple* $(c_0, c_1, c_2, ..., c_{n-1})$*, where* c_i *is the value of column* i*. As described in Definition 1, let* $c_i[j]$ *be the* j*-th bit of the bitstring representation of column* i*. The 20-bit Z-order value for the given tuple has the following bitstring representation:*

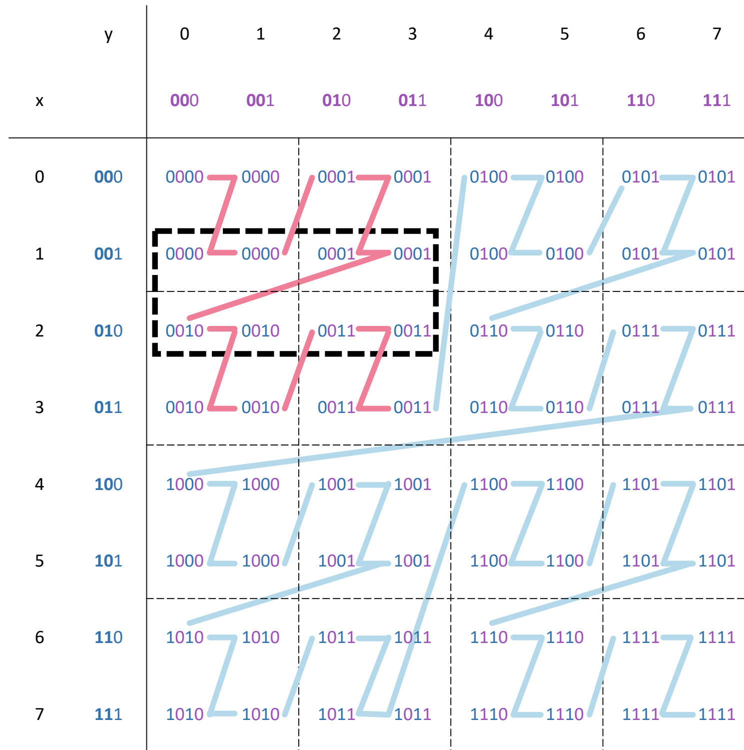
$$c_0[0], c_1[0], c_1[1], c_1[2], c_1[3], c_1[4], c_2[0], c_2[1], c_2[2], c_0[1], c_1[5], c_1[6], c_1[7], c_1[8], c_1[9],$$
$$c_2[3], c_2[4], c_2[5], c_0[2], c_1[10].$$

Complex sorting schemes like Z-order are difficult to tune, with tunable parameters such as which columns to include in the Z-order. Systems such as Amazon

Redshift and Databricks Delta Lake allow users to sort on multiple attributes, but it is unclear how best to determine the columns to include and how many columns to include. Poor tuning might result in scanning many more rows than necessary that are not relevant to the query. When choosing columns, database administrators might consider information such as which columns are accessed together and the selectivity of the columns. One approach they might take is to choose the top x most important columns, where importance is defined by selectivity or frequency. Existing systems give an equal weight to the columns included in the Z-order. Such an approach, however, might not result in the best performance due to reasons such as unequal column impact on query performance and column domain. Factors such as column selectivity and frequency (the number of queries a column appears in) might contribute to a column having a greater effect on query performance and thus should be taken into account when determining the weight to put on each column. For example, it may be better to give more weight to more selective columns. Producing the best Z-order configuration using all of this information remains a challenge.

Fig. 1-1 shows an example where giving an unequal number of bits to the columns results in better performance than equal bit allocation. Imagine a dataset with 64 tuples and two columns where each record is an (x, y) point on a two-dimensional plane. We consider four-bit Z-order values and compute the Z-order value for each (x, y) coordinate pair by interleaving the bits of the x and y columns. Using Z-ordering, we can arrange the two-dimensional pairs on a one-dimensional line, which follows a "Z" shape, hence the name "Z-order curve." The block size is four tuples; block boundaries are indicated by the black dotted lines, which split the Z-shaped curve into blocks of size four. Note that tuples with the same Z-order value can be arranged in any order on the one-dimensional line; the line in the figure shows one possible ordering. Imagine that we are searching for records with an x value between one and two and a y value between one and three. Since our data is two-dimensional, we are essentially looking for records that fall within the rectangular query space where the lower bound (upper left corner) is at $[x = 1, y = 0]$ and the upper bound (lower right corner) is at $[x = 2, y = 3]$. Fig. 1-1a shows an equal allocation of bits to

columns: we interleave the bits by taking the first bit of the x column, the first bit of the y column, the second bit of the x column, and the second bit of the y column. For this configuration, four blocks intersect the query rectangle and are scanned. In Fig. 1-1b, we show another layout where we give three bits to the x column and one bit to the y column. The Z-order value is computed by taking the first bit of the x column, the second bit of the x column, the third bit of the x column, and the first bit of the y column. This configuration results in improved query performance, with only two blocks being scanned, half of that in Fig. 1-1a. This reveals the importance of considering the weight of each column in the Z-order.



(a) Equal bit allocation results in four blocks being scanned.

(b) An uneven bit allocation results in two blocks being scanned.

Figure 1-1: Examples of Z-order configurations and data layouts. An uneven bit allocation results in fewer points scanned than equal bit allocation.

Our work seeks to improve query performance by optimizing the data layout and index structure for a particular dataset and query workload. Recent work on learned indexes has introduced the idea of automatically optimizing an index for a specific dataset and workload [6, 10]. In this thesis, we propose learning Z-order indexes using a dynamic bit allocation method. As described above, a naive approach of allocating equal bits to columns might not result in the best performance. We use Bayesian optimization to search over possible configurations and evaluate each candidate configuration using a cost model, which estimates the query time for that configuration. The best allocation of bits to columns is the one with the lowest estimated query time. We compute the Z-order values for each row by interleaving the bits according to the best configuration and sort the table by the Z-order values. The table is stored in the Parquet format, which consists of blocks and uses zone maps to skip blocks when performing query execution.

Our learned Z-order indexes achieve high performance on all our datasets: they are faster than, or on par with, every other indexing/partitioning method on all the workloads. The Z-order indexes are up to 1.4× faster than the next best index. In addition, they achieve a significant reduction in, or comparable, average number of rows scanned compared to every other technique. The best Z-order configurations produce data layouts that result in between 1.2× and 2× reduction in the number of rows scanned compared to the next best index. Moreover, the best 64-bit Z-order configuration achieves up to 52.4% better performance than the equal-bit Z-order configuration, revealing the importance of tuning Z-order. We also consider the k-bit Z-order problem, where we try a fixed set of k values. Results show that the best k-bit Z-order configuration results in up to 22.6% reduction in the average number of rows scanned compared to the best 64-bit Z-order configuration, which suggests that it may be helpful to tune k. Through our exploration of Z-order, we also learn that despite allocating a total of 64 bits to the columns, only the first x bits matter in improving query performance, where $x < 64$ and varies across the datasets.

The main contributions of this thesis are:

- Implementation of Z-order in C++ and Apache Arrow

- Design and implementation of a dynamic bit allocation method to finding the best Z-order configuration in Python, using search algorithms such as Bayesian optimization
- Evaluation of various partitioning/indexing methods on real-world datasets and query workloads

The rest of the thesis is organized as follows. Chapter 2 discusses related work. Chapter 3 presents our approach to producing the best Z-order index for a particular dataset and query workload. We evaluate our learned Z-order indexes and discuss some of the insights gained through our exploration of Z-order in Chapter 4. Finally, Chapter 5 concludes the thesis and identifies areas for future work.

Chapter 2

Related Work

The ability to sort by multiple dimensions can be highly beneficial, so modern database systems are increasingly turning to multi-dimensional sorting schemes. Z-order is a locality-preserving space-filling curve that maps multi-dimensional data to one dimension. The Z-order value for a tuple is calculated by interleaving the bits of the bitstring representation of the column values [15, 16].

Recent systems have introduced the idea of building indexes on top of the Z-order. Some analytical databases such as Databricks Delta Lake organize tables into blocks (horizontal partitions) and use a data structure called the zone map, which stores summary statistics of each block. Zone maps allow blocks that do not satisfy the predicates on columns to be skipped, thereby improving query performance. Databricks Delta Lake sorts the data by the Z-order values, which enables more effective block skipping and significantly reduces the amount of data scanned [9]. Amazon Redshift [5] is another commercial system that allows tables to be defined with an interleaved sort key, the Z-order value, to improve zone map effectiveness. These systems cache the zone map metadata in memory. Meanwhile, standard file formats such as Apache Parquet and Apache ORC store the minimum/maximum metadata in the file footer. During scans, if the range of values between the minimum and the maximum does not intersect the query, then the block is skipped.

Other works have proposed more complex indexes based on Z-order. The UB-tree [3, 15] is based on the standard B-tree but uses the Z-order curve to partition multi-

dimensional space into regions called Z-regions. A Z-region is the space covered by an interval on the Z-order curve. Each Z-region maps to one leaf page of the UB-tree. The UB-tree processes range queries by calculating the smallest and largest Z-order value in the query rectangle (lower left and upper right vertices of the query rectangle) and retrieving all the Z-regions that intersect the query rectangle. It does so by finding the physical indexes corresponding to the smallest and largest Z-order values and iterating through every physical index in that range. Whenever it encounters a Z-order value that is outside the query rectangle, it calculates the next Z-order value that falls within the query rectangle. Similar to UB-tree [15], Amazon DynamoDB allows users to create an index on the Z-order value [16]. The way it handles range queries is similar to that of UB-trees, in which it finds the minimum and maximum Z-order values for the query and iterates through the Z-order values in that range. For each Z-order value, it evaluates its relevance by calling the "isRelevant" function to check to see if the Z-order value falls within the query rectangle. If it does not, then it calls the "nextJumpIn" function to calculate the next smallest Z-order value that falls inside the query rectangle.

Some advantages of using Z-order just as a sort order include simplicity and low storage overhead, but one drawback is that the method of skipping blocks based on the minimum/maximum metadata might have false positives and result in more records being scanned that are not relevant to the query. Meanwhile, systems with more complex indexes can allow for more effective scans by only examining Z-order addresses that fall in the query rectangle, but this comes at the expense of increased execution time due to API calls (as mentioned in Amazon DynamoDB [16]) and storage overhead.

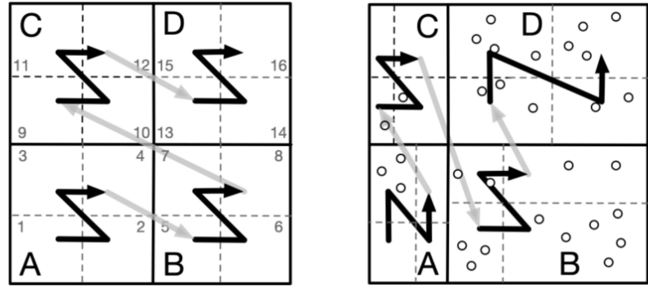
Currently, database administrators can use systems such as Amazon Redshift [5] and Databricks Delta Lake [9] to sort on multiple columns using Z-order by specifying the columns. However, the columns they select might not result in the best performance, as the choice of columns can greatly affect performance or lack thereof. Thus, determining the best columns for Z-order remains an open problem.

Our work aims to learn the Z-order layout from the data and query workload.

There has been recent work on learned multi-dimensional indices such as Flood [10] and Tsunami [6]. Flood is a multi-dimensional in-memory read-optimized index that automatically adapts itself to a particular dataset and workload. It works by using a sample query workload to glean information such as how often columns are used together in order to come up with the best layout [10]. Tsunami extends the ideas of Flood with new optimization techniques that allow it to adapt to skewed query workloads. It also uses a simple analytic linear cost model to predict the runtime of a query for a particular dataset and layout [6]. Our work draws ideas from these systems to evaluate candidate Z-order configurations.

In addition, a recent paper [14] proposes instance-optimal variants of the Z-order index that adapt to the data and anticipated workload of queries. The variants concern the ordering and partitioning of the cells. It considers normal Z-order index to partition the space into four cells (A, B, C, and D), with cells ordered as ABCD and the partitions happening at the coordinates corresponding to the median of the data distribution along each axis (see Fig. 2-1a). The variants it proposes allow the ordering of the cells to have either the ABCD pattern or the alternative ACBD pattern, as these are the only orderings that preserve monotonicity[1] for the points that fall in the different cells. It also allows the partitioning of the cells at any location, not just the median (see Fig. 2-1b). In this thesis, we implement the approach presented in [14] and show how it compares to our approach.

[1]If every component of a point u in cell M is less than every component of a point v in cell $N \neq M$, then point u appears earlier in the Z-ordering than point v.



(a) Standard two level Z-order index.

(b) Two level Z-order index with varying cell orderings and partitionings.

Figure 2-1: Examples of the Z-order curve and Z-order indexes. Figures taken from [14].

Chapter 3

Overview

Z-order indexing allows sorting by multiple columns, with each column having equal weight. It has drastically improved query performance and has been incorporated in several modern systems, which allow users to select the columns included in the Z-order. However, choosing the best Z-order configuration remains a challenge, as Z-order has many tunable parameters. We propose a dynamic bit allocation approach that considers not only which columns to include, but also the weight (number of bits) to put on each column. The optimal layout can vary depending on the dataset and query workload. Our work aims to produce the Z-order configuration that achieves the best performance for a particular dataset and query workload.

The number of most significant bits that we take from each column and use in bit interleaving affects Z-order performance. The naive approach of giving equal bits to columns does not always perform well. Consider the following synthetic dataset in Table 3.1, which consists of 10 million rows where each column's value is a random number chosen uniformly between zero and the maximum column value. The query workload contains 500 queries, with each query filtering over one column.

Column	Max Column Value	Selectivity (%)	Queries
col_0	10	30	10
col_1	8	40	10
col_2	1,000,000	0.1	150
col_3	1,000,000,000	0.1	180
col_4	1,000,000,000	0.1	150

Table 3.1: Synthetic uniform workload.

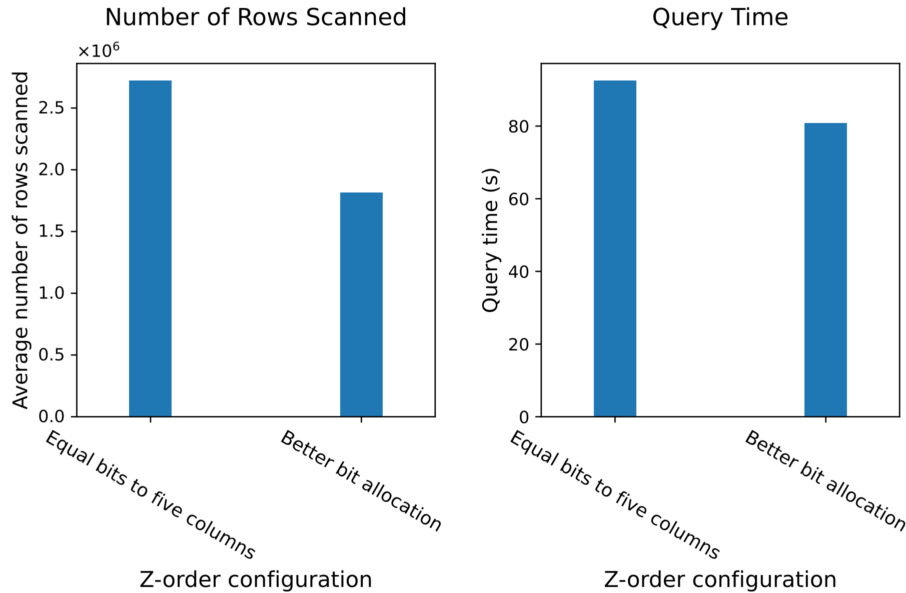


Figure 3-1: Uneven bit allocation results in better performance than equal bit allocation for the synthetic dataset and query workload.

In Fig. 3-1, we compare a Z-order configuration that gives each of the five columns an equal number of bits ($\{col_0 : 13, col_1 : 13, col_2 : 13, col_3 : 13, col_4 : 12\}$) and a better Z-order bit allocation ($\{col_0 : 3, col_1 : 3, col_2 : 17, col_3 : 22, col_4 : 19\}$). We can see that the better Z-order configuration results in a 33% reduction in the number of rows scanned. An uneven bit allocation helps in this example because column 0 and column 1 have a small domain and higher selectivity, so a small number of bits is sufficient. On the other hand, columns 2, 3, and 4 have a much larger domain, very low selectivity, and appear in many queries, so it is essential to give many bits to these columns. This motivated us to find the best Z-order mapping of bits to columns for a given dataset and workload, considering the columns to include in the Z-order and the number of bits for each column.

3.1 Problem Statement and Approach

The problem of finding the best k-bit Z-order configuration for a dataset with n columns can be framed as an optimization problem that seeks to find the best *mapping* of k bits to n columns. The rest of the thesis uses the terms *mapping* and *allocation* interchangeably. We distribute the k bits among the n columns, with zero bits signifying that the column is not included in the Z-order value. Each candidate mapping is an n-dimensional vector, and the best candidate is the one that results in the lowest query time.

3.1.1 Cost Model

A key component is the objective function/cost model, which takes a candidate mapping as input and produces the estimated query time. We come up with a cost model that estimates the query time based on certain statistics such as the number of blocks or rows that intersect a query, which can be efficiently computed by examining the minimum/maximum of each block. Plotting the statistics obtained from a sample dataset (10%) and those obtained from the entire dataset reveals that there is a strong linear relationship, as shown in Fig. 3-2. Hence, we create a sample dataset where rows are chosen uniformly from the dataset and obtain the statistics for the sample dataset, which is much more efficient than getting the statistics for the entire dataset. The cost model that we use for a single query is:

$$\text{Query Time} = w * (\text{number of rows scanned}) * (\text{number of filtered columns}).$$

The term w represents the time to scan a single column of a single point.

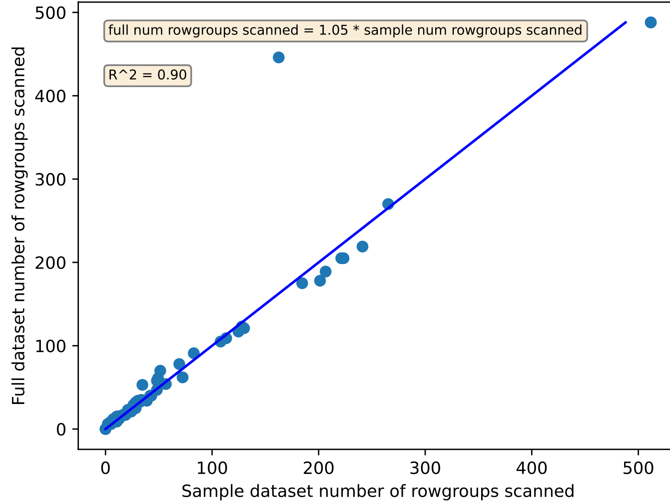


Figure 3-2: Statistics obtained from the sample dataset are an accurate estimate of those obtained from the entire dataset.

3.1.2 Search Algorithms

We consider the following search algorithms to find the optimal candidate mapping: random search, Bayesian optimization, and genetic algorithm. Each of the search algorithms follows the structure in Algorithm 1. In our evaluation, we found Bayesian optimization to be superior to the other search methods, so our technique uses Bayesian optimization to find the best mapping. We run Bayesian optimization on datasets collected from [12]: contributions, flights, taxi, and tweetmap.

Algorithm 1 Search algorithm structure (simplified)

for $i \leftarrow 1$ to N **do**
  $model = Model(objective_func, variable_bounds, num_iters, other_params)$
  $model.minimize()$
end for

During each iteration of the search, a candidate mapping in which the bits sum to k, the desired total number of bits, is generated. We consider allocating the bits to only the columns that appeared in the query workload; doing so should lead to better results since we are exploring a smaller search space. For some of the search algorithms, we do not control the generation of candidate mappings, so candidate mappings where the total number of bits is not k sometimes arise. For example,

when considering $k = 64$, the mappings may not have exactly 64 bits. In those cases, we scale the number of bits allocated to each column such that the resulting sum is around 64 bits. We ensure that the final mapping has exactly 64 bits through a process of randomly selecting a column and adding or removing a bit as needed to reach the desired 64 bits. The mapping is then inputted to the objective function, which calculates the estimated query time using the cost model described in Section 3.1.1.

In random search, a random mapping is produced for each iteration. Bayesian optimization seeks to find the global maximum (or minimum) and consists of a Bayesian statistical model for modeling the objective function and an acquisition function that proposes where to sample next [7]. Genetic algorithm is inspired by the process of natural selection and consists of a population of candidate solutions that evolve toward better solutions based on biological concepts such as mutations and crossovers [8].

3.2 Implementation

3.2.1 Z-Order

The Z-order code is implemented in C++. Tables are stored in Parquet format [2], a column-oriented file format, on disk; data is partitioned into sets of rows called *rowgroups*, and within each rowgroup, data from different columns is stored separately. The rest of this thesis uses the terms *rowgroup* and *block* interchangeably. The Apache Arrow library [1] is used to work with the Parquet files, such as loading the Parquet files in memory.

To sort the table according to a Z-order configuration, we wrote a function that takes as input a table, a list of columns, and a mapping of bits to columns. The function first determines the position of the most significant nonzero bit for each column. It then calculates the Z-order for each row in the table by interleaving the bits and finally sorts the table by the Z-order values. In this thesis, we consider k-bit Z-order values where $k \leq 64$ since 64 bits appeared sufficient for our datasets and

workloads, as query performance plateaus at x bits for some $x \leq 64$ (see Fig. 4-8). Specifically, we consider the following k values: 6, 13, 20, 30, 40, 50, and 64.

3.2.2 Search Algorithms

For Bayesian optimization, we use the Python package [11]. The package draws on the ideas of Bayesian inference and Gaussian processes and attempts to find the maximum value in as few iterations as possible.

We use the Python package [17] to run the genetic algorithm. The algorithm takes in a number of parameters, including population size and mutation probability. Below is the set of parameters we used.

```
{
    'max_num_iteration': 1500,
    'population_size': 30,
    'mutation_probability': 0.1,
    'elit_ratio': 0.2,
    'crossover_probability': 0.7,
    'parents_portion': 0.5,
    'crossover_type': 'uniform',
    'max_iteration_without_improv': None
}
```

We also set the variable type to be "int" so that the algorithm only considers configurations where each column is allocated an integer number of bits.

3.2.3 Recursive Cell Splits

The recursive approach presented in [14] omitted many details, so we filled in the gaps and came up with an approach detailed in Algorithm 2. At a high level, the algorithm greedily finds the best split points along each column sequentially, with the goal of arriving at a set of leaf cells that minimizes the number of points scanned

when running the query workload. The algorithm in [14] splits along all the columns in the dataset; we improve upon that by only considering the columns that appear in the query workload. One Z-order level is complete when we have found a split point for each of the columns. We recursively run the process of finding the split points one level at a time on each leaf cell resulting from a complete Z-order level; the cell splitting terminates when the cell contains fewer than 10 points or when the running time of the algorithm exceeds 24 hours.

After obtaining the split points, we build a tree-based index based on the resulting cells, with pages containing the points in the lowest level grid cells. We sort the full dataset by the index and calculate the query cost by counting the points traversed when scanning the pages that the query intersects.

Algorithm 2 Recursive cell splits algorithm (adapted from [14])

1. Given a dataset and query workload, determine the column split points.
   - Order the dimensions by selectivity; this will determine the order of the columns to split, which will be used for each level.
   - Choose the split point along each dimension sequentially.
     - We consider 10 candidate split points that divide the column space evenly.
     - For each candidate split point, assign points from the sample dataset to the leaf cells that result from splitting the column at the split point. We define the query cost to be the total number of points that fall in the cells that the query intersects. The objective function is the sum of the query costs for all the queries in the workload.
     - The best split point is the one that minimizes the objective function. After finding the best split point, we divide the region specified by each cell according to it.
   - We say that one Z-order level is complete after finding the best split points for each of the columns. We recursively run the process of finding the split points one level at a time on each leaf cell resulting from a complete Z-order level.
     - The algorithm terminates when the cell contains fewer than 10 points or when the running time of the algorithm exceeds 24 hours.
2. Build a tree-based index based on the resulting cells, with pages containing the points in the lowest level grid cells.
3. Sort the full dataset by the index. Calculate the query cost by counting the points traversed when scanning the pages that the query intersects.

Chapter 4

Evaluation

In this section, we evaluate the performance of our dynamic Z-order bit allocation approach. We present the results of an experimental study that compares our approach with other partitioning/indexing methods on several datasets and query workloads. Performance metrics include average number of rows scanned and query time. Our goal was to gain insight into how much performance gain Z-order can provide.

Results reveal that:

- Our learned Z-order indexes achieve high performance on all the datasets. They result in a significant decrease in, or comparable, average number of rows scanned compared to every other partitioning/indexing method. On our datasets, the best 64-bit Z-order configuration achieves up to 23.4× reduction in rows scanned compared to the default sort order, up to 7.8× reduction compared to range partitioning, and up to 3.7× reduction compared to the index resulting from the recursive cell splits algorithm (see Section 4.4.1).

- The Z-order configurations are faster than, or on par with, every other partitioning/indexing method on all the workloads. They are up to 1.4× faster than the next best index (see Section 4.4.1).

- The best 64-bit Z-order configuration achieves up to 1.8× reduction in the average number of rows scanned compared to the 64-bit Z-order configuration that gave an equal number of bits to the three most frequently appearing columns

(see Section 4.4.1). This highlights the importance of determining the best allocation of bits to columns.

- The best k-bit Z-order configuration results in up to 22.6% reduction in the average number of rows scanned compared to the best 64-bit Z-order configuration, which suggests that it may be helpful to tune k (see Section 4.4.1).
- When searching for the best 64-bit Z-order configuration, we allocate a total of 64 bits to the columns. However, only the first x bits matter in improving query performance, where $x < 64$ and varies across the datasets (see Section 4.4.6).

4.1 Baselines

We compare our approach to the following partitioning methods/indexes:

1. *Default sort order*: queries are performed on the original dataset. The data is still partitioned into blocks and uses zone maps to skip blocks.
2. *Range partitioning*: the dataset is sorted on a user picked column. We select the column with the lowest average selectivity[1]. Choosing the column with the lowest average selectivity appears to work well for our workloads since our definition of average selectivity takes into account both the number of queries the column appears in and the selectivity of the column, thus allowing for effective filtering over the column.
3. *Recursive cell splits*: [14] proposes an instance-optimized Z-order index that allows the partitioning of a dimension to happen at any location, not just the median as in standard Z-order. It divides the entire space into cells by recursively running the process of finding the best split point along each column sequentially (see Section 3.2.3 for more details). A tree-based index is created

[1]Average selectivity is calculated for each column by evaluating the mean of the selectivity of each query — if the col does not appear in the query, then it would be considered 100% selectivity; otherwise the selectivity would be CDF(right end of range) − CDF(left end of range). A query of the form $x < col < y$ would contribute $CDF(y) - CDF(x)$ to the average selectivity for that column.

based on the resulting cells, with pages containing the points in the lowest level grid cells. The dataset is then sorted by the index.

4. *Z-order: equal bits to three columns*: we distribute 64 bits, allocating an equal number of bits to the three most frequently occurring[2] columns in the query workload. We choose the three most frequently appearing columns since it seems plausible for a database administrator to easily identify these columns if they had to tune Z-order themselves. Specifically, we give 22 bits to the first column and 21 bits each to the other two columns. This is used as a baseline to see how a naive equal bits approach compares to an instance-optimized Z-order index.

The following are our approaches:

- *Z-order: best 64-bit allocation*: we sort the dataset according to the best 64-bit Z-order configuration found by Bayesian optimization.
- *Z-order: best bit allocation*: we consider the k-bit Z-order problem, where k is a hyperparameter. We run Bayesian optimization to find the best mapping for the following set of k values: 6, 13, 20, 30, 40, 50, and 64. The dataset is sorted according to the best Z-order bit allocation.

All the experiments are single-threaded and run on an Arch Linux machine with an Intel Xeon 2.1GHz CPU and 125GB RAM.

4.2 Datasets and Query Workloads

We evaluate indexes on four real-world datasets collected from [12]: contributions, flights, taxi, and tweetmap (see Table 4.1). The query workload consists of range and equality filters that filter over several columns. Each query has a selectivity of 1% or less. Query execution involves scanning/filtering, projection over the columns that appear in the query, and materialization of the output tuples.

[2]For each column, we count the number of queries it appears in.

	Contributions	Flights	Taxi	Tweetmap
Rows	86M	120M	175M	15M
Columns	9	21	13	16
Size (GB)	4.5	9.1	12	1.7
Queries	500	500	500	451

Table 4.1: Dataset and query characteristics.

Our first dataset, contributions, contains 25 years of political contributions data. The query workload consists of queries with range filters on donation amount, donation date, and location (latitude/longitude), and equality filters on recipient name and party.

The flights dataset consists of 120 million records of U.S. airline flights from 1987 to 2008. It has 21 columns, with flight data including origin and destination city and state, carrier, day of week, and flight distance. The queries answer analytics questions, such as "How many flights departed from a certain state at a certain time (defined by time of day, day of the week, and/or month) and arrived at a particular destination state?" and "How many flights were operated by a specific carrier and departed in a certain time interval?"

Our third dataset, taxi, contains information on taxi rides in NYC over a seven-year period, with data such as pickup time, trip distance, number of passengers, and stores within 30 meters of a pickup or dropoff location. The queries perform aggregations, such as counting the number of taxi rides in which passengers were picked up at a particular region defined by latitude/longitude in a particular time interval.

The tweetmap dataset contains geocoded tweets from November 2014 to February 2015, with information such as the tweet time, language the tweet was sent in, sender name, and country. The queries perform tasks such as gathering the tweets in a certain geographical location and sometimes also filtering by tweet time and language.

For each dataset, sample datasets were constructed by randomly selecting between 0.2% and 1% of the rows in the full dataset (except for tweetmap, which has 10% of

the rows in the full dataset since the full dataset is much smaller than those of other datasets). These sample datasets are used during optimization to find the best Z-order mapping. For each candidate mapping, statistics are obtained from the sample dataset and used to estimate the query time (see Section 3.1.1). The full dataset is divided into 16MB rowgroups during evaluation. Suppose the full dataset contains x 16MB rowgroups. To allow for more accurate estimates, we partition the sample dataset into at least x rowgroups when estimating the query time.

4.3 Evaluation Setup

We use the Google C++ benchmark to evaluate the performance of the partitioning/indexing methods detailed in Section 4.1. For all the methods, the sorted table is stored in the Parquet format and divided into rowgroups of size 16MB. Parquet stores the minimum/maximum statistics per column for each rowgroup and uses these statistics to skip rowgroups when executing queries. We calculate several metrics, including the query time and average number of rows scanned.

4.4 Results

4.4.1 Query Performance

We show how well our dynamic Z-order bit allocation approach performs when compared to the baseline indexes.

Fig. 4-1 shows the average number of rows scanned for each index on each dataset. From Fig. 4-1, we can see that the best Z-order configurations achieve high performance on all the datasets: they result in fewer or comparable average number of rows scanned compared to every other index. On three of the datasets, the best 64-bit Z-order configuration produced by our approach achieves between 1.2× and 1.9× reduction in the number of rows scanned compared to the next best non-Z-order index. Meanwhile, the best Z-order configuration results in between 1.2× and 2× reduction in the number of rows scanned compared to the next best non-Z-order index.

For the contributions dataset, we can see that the best 64-bit Z-order mapping achieves a significant reduction (approximately 12.1×, or 92%) in the average number of rows scanned compared to the default sort order. Moreover, when compared to the Z-order mapping that allocated an equal number of bits to the three most frequently appearing columns, the best 64-bit Z-order mapping resulted in 23.2% reduction in the average number of rows scanned. This highlights the importance of determining the number of bits to give each column and the columns to include in the Z-order.

On the flights dataset, the best Z-order configuration achieves 46% reduction in the average number of rows scanned compared to the Z-order mapping that gave an equal number of bits to three columns.

For the taxi dataset, the best 64-bit Z-order mapping provides 37.5% improvement in the average number of rows scanned compared to the Z-order equal bit allocation to three columns. In addition, the best 64-bit Z-order mapping achieves 7.8× reduction in average number of rows scanned compared to range partitioning and 10.4× reduction compared to the default sort order.

On the tweetmap dataset, the best Z-order configuration scans 22.6% fewer rows than the best 64-bit Z-order configuration and 52.4% fewer rows than the Z-order equal bit allocation to three columns.

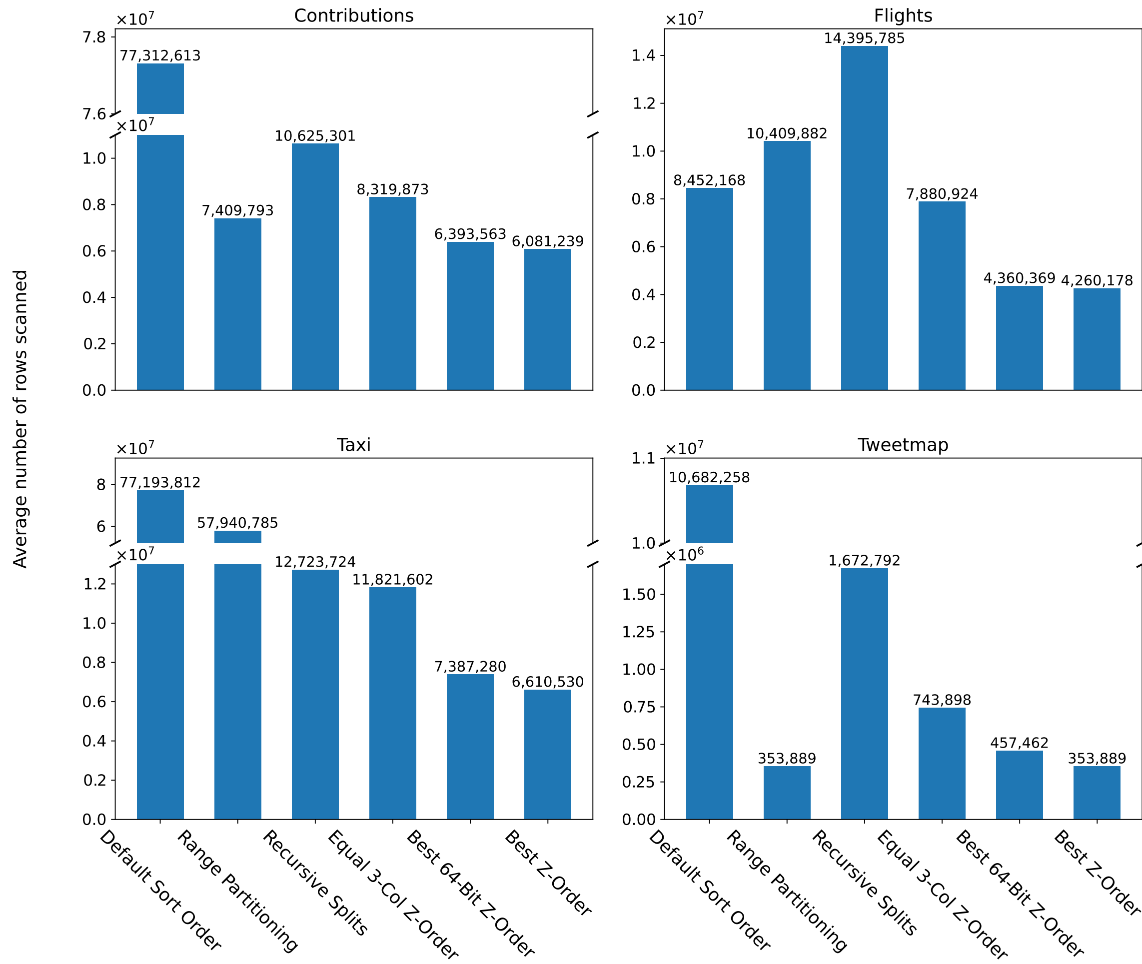


Figure 4-1: Average number of rows scanned for the different indexing/partitioning methods.

Fig. 4-2 shows the average query time for each index on each dataset, with error bars showing one standard deviation in each direction. The trends are similar to those in the plots showing average number of rows scanned (Fig. 4-1).

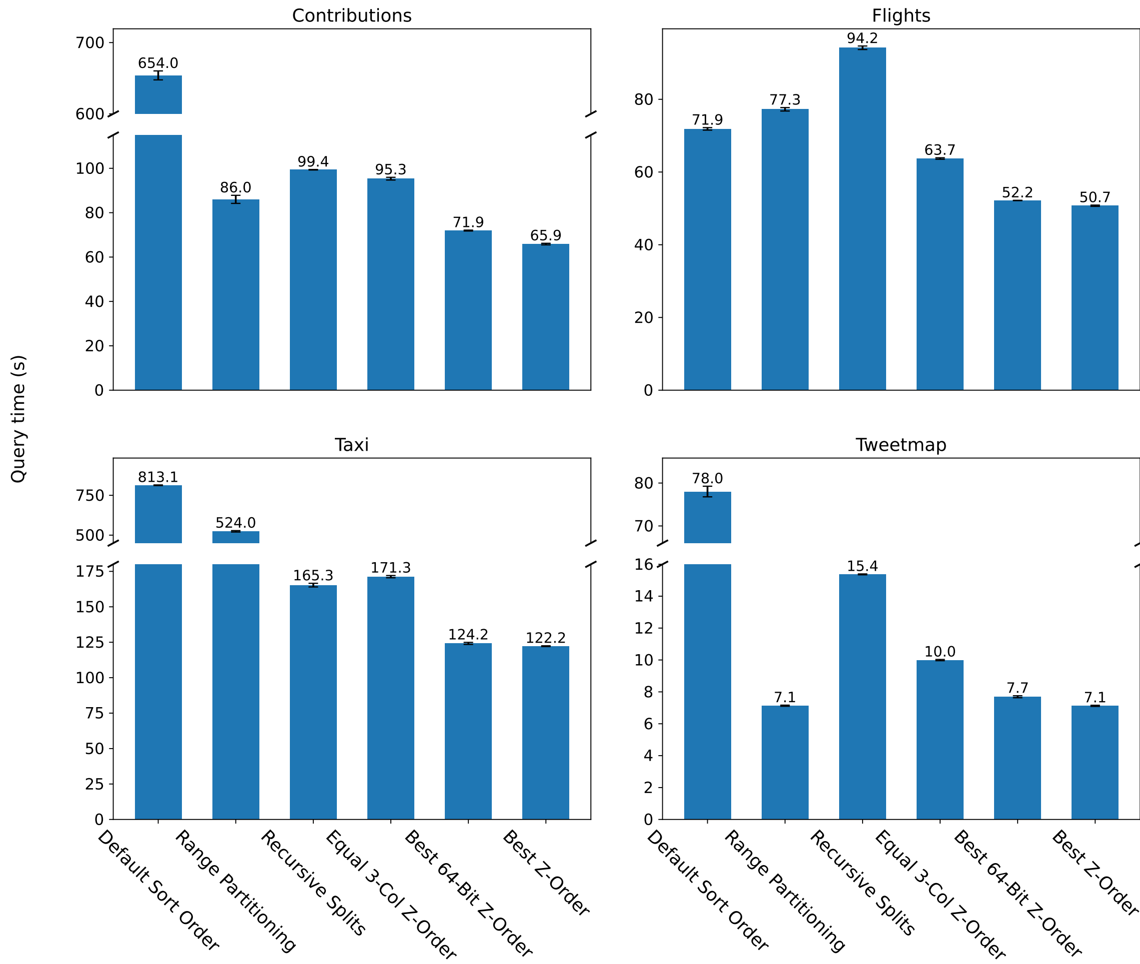


Figure 4-2: Average query time for the different indexing/partitioning methods.

4.4.2 Discussion

In Fig. 4-1, we can see that the best Z-order configurations found from Bayesian optimization generally resulted in a significant reduction in the average number of rows scanned when compared to the default sort order and range partitioning. This can be explained by the fact that Z-order allows for multi-dimensional sorting, which is particularly beneficial since the queries in the workload filter over multiple columns.

However, we might notice that range partitioning performs as well as the best Z-order mapping for the tweetmap dataset. This may be due to a characteristic of the tweetmap query workload, in which one of the columns appeared in almost

all of the queries in the workload and was the most selective among the columns: latitude. Range partitioning sorted the data by that column, which had an average selectivity of 1.51%. The importance of that column was also discovered during the Bayesian optimization search, which produced a best Z-order mapping that allocated all of the bits to that column. Another reason may be that it was challenging for Bayesian optimization to find a good Z-order configuration due to the large number of columns (ten) appearing in the query workload, a phenomenon known as the curse of dimensionality. As a result, range partitioning and the best Z-order index achieved similar performance.

Another observation is that the recursive splits algorithm resulted in more rows scanned than the Z-order configurations (see Fig. 4-1). The algorithm specified in [14] considered splitting along all the columns in the dataset. We improve upon the algorithm by considering only the columns that appeared in the query workload, which resulted in better performance compared to the original version since the original version explores columns/search space not relevant to the query workload. Even after optimizing the recursive splits algorithm, it performs worse than the Z-order configurations produced by our approach. This might be attributed to the way it splits along a dimension, the greedy approach not producing the best grid cell layout, and the equal weight to all the columns. The paper does not specify details such as how many candidate splits to consider for each column; we decided to consider ten candidate when splitting along a dimension. It is unclear how best to explore the search space for a column, such as how many candidates to consider, in a reasonable amount of time. In addition, it is possible that the greedy approach detailed in [14] fails to find the globally optimal solution due to nature of the greedy algorithm, where it makes a locally optimal choice at every step. Furthermore, since each column is split the same number of times (ignoring terminating conditions), the algorithm essentially puts the same weight on each column, including columns that do not matter much for query performance. As a result, the index produced by the algorithm sometimes performs worse than the Z-order configuration that gives equal bits to three columns.

In addition, from Fig. 4-1 and Fig. 4-2, we can observe that the decrease in query

time is not as significant as the decrease in the average number of rows scanned. One reason is that query execution involves not only scanning the rowgroups to find the rows that intersect the queries, but also materializing the output tuples. Although the former is improved when using the Z-order index since it results in fewer rowgroups being scanned, the latter is constant among all the different partitioning methods/indexes. Moreover, Parquet stores rowgroup metadata such as minimum/maximum for each column in the footer. Each partitioning/indexing method has to iterate over the metadata for all the rowgroups in the footer in order to read the rowgroup metadata, thereby incurring I/O cost. As a result, the reduction in query time is not as drastic as the reduction in the average number of rows scanned.

4.4.3 Search Algorithms

In Fig. 4-3, we plot the estimated and actual times of individual queries in the workload. Fig. 4-3 reveals that the cost model produces accurate estimates of the total query time, which is calculated as the sum of the estimated query time of the individual queries. This is particularly useful for comparing candidate mappings during the search for the best Z-order configuration.

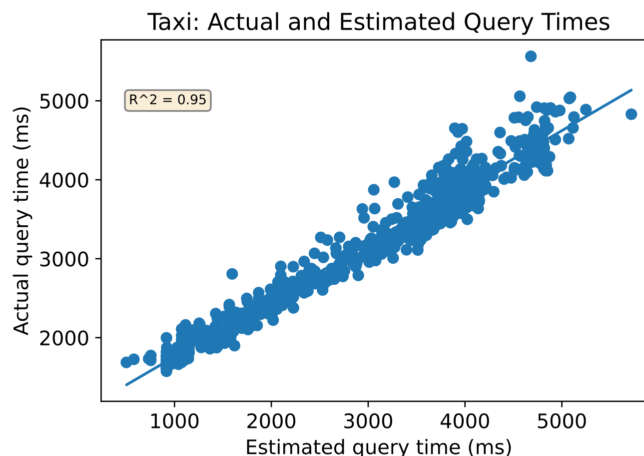


Figure 4-3: Actual vs. estimated query times (using the cost model in Section 3.1.1) for a workload.

We initially explore the following search algorithms for finding the best allocation of bits to columns: random search, Bayesian optimization, and genetic algorithm.

We had each search algorithm run for 600 iterations and repeat this 5 times. In Fig. 4-4, we plot the lowest query time seen so far (averaged across the runs) as iteration number increases. Bayesian optimization generally performs the best among the search algorithms. The gradient of the objective function cannot be accurately computed, supporting the use of Bayesian optimization in finding the best Z-order configuration. In addition, having a well-behaved objective function gives algorithms such as Bayesian optimization an advantage over random-based algorithms such as random search and genetic algorithm, which helps explain why Bayesian optimization performs well for the Z-order bit allocation problem. Our dynamic bit allocation approach uses Bayesian optimization to find the best Z-order configuration.

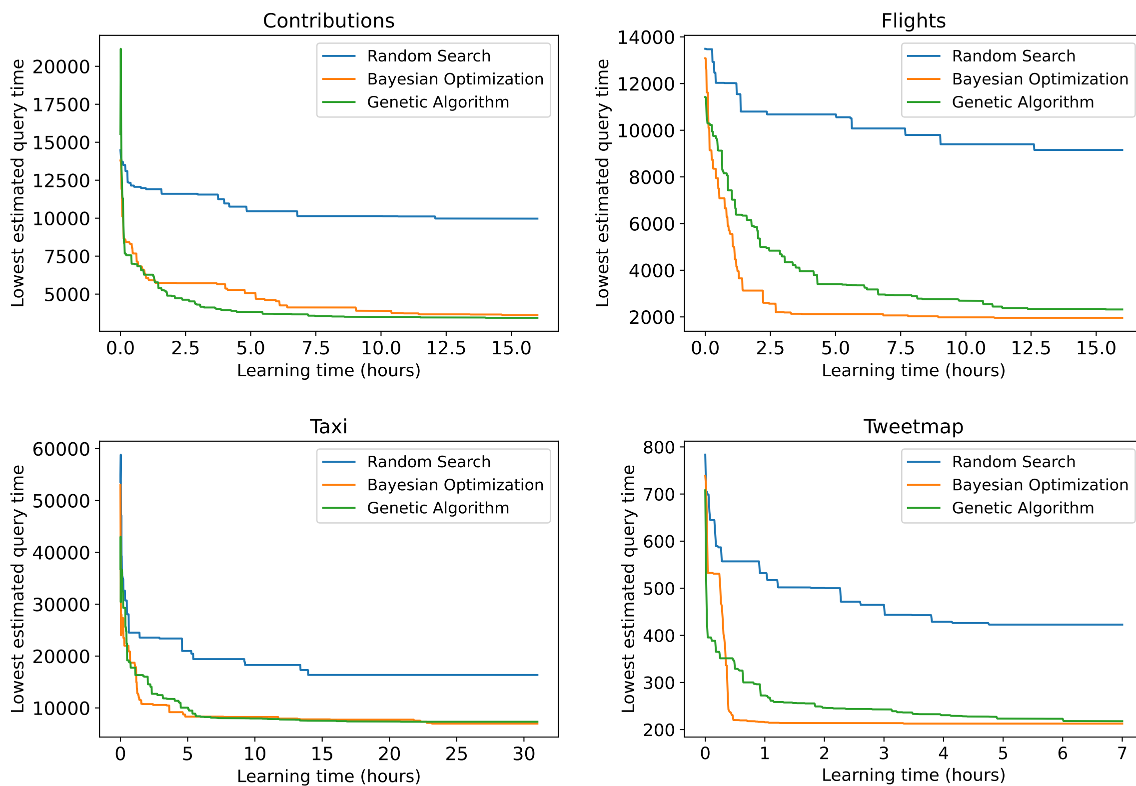


Figure 4-4: Lowest query time seen when running several search algorithms: random search, Bayesian optimization, and genetic algorithm.

4.4.4 Z-Order Index Creation

Table 4.2 shows the time it takes to create the best 64-bit Z-order index. We separate time into learning time, which is the time taken for Bayesian optimization to find the best Z-order configuration, and repartitioning time, which is the time to compute the Z-order values for the full dataset and sort by that. Note that we did not fully optimize the performance of the learning code; with further optimization, the learning time should decrease.

	Contributions	**Flights**	**Taxi**	**Tweetmap**
Learning	57.2K (15.9 h)	54.2K (15.1 h)	108K (30 h)	18.7K (5.2 h)
Repartitioning	135.4	265	271.5	31.2
Total	57.3K (15.9 h)	54.5K (15.1 h)	108.3K (30.1 h)	18.7K (5.2 h)

Table 4.2: Best 64-bit Z-order index creation time in seconds.

As mentioned in Section 4.1, we consider the following set of k values for the best k-bit Z-order problem: 6, 13, 20, 30, 40, 50, and 64. Table 4.3 shows the learning time when running the searches for these k values in parallel. The repartitioning times are comparable to those in Table 4.2.

	Contributions	**Flights**	**Taxi**	**Tweetmap**
Learning	28.3	29.3	44.3	14.3

Table 4.3: k-bit Z-order index learning time in hours.

4.4.5 Workloads in which Z-Order Indexes Work Well

Using a Z-order index works well for workloads where most of the queries filter over a few columns. On the other hand, the performance improvements that come with using a Z-order index are greatly reduced when bits are allocated to more columns: for the 64-bit Z-order problem, even though 64 bits are allocated, only the first x bits truly matter, where $x < 64$ and varies across the datasets (see Section 4.4.6). This means that if we have more columns, each column can only have a few bits, which

may not be enough to allow for effective query filtering. This is further exemplified in Fig. 4-7, which shows that including more columns in the Z-order can adversely affect performance when there is a limited number of blocks. We have observed that the Z-order index performs particularly well for workloads where a majority of the filters are over three or fewer columns and that the best Z-order mappings found by Bayesian optimization only allocate bits to at most six columns.

We use scan overhead as a metric to understand the workloads Z-order does well in. Scan overhead is defined as the ratio between points scanned and the actual result size. The optimal scan overhead is one, and we aim to get as close to one as possible. Table 4.4 shows the scan overhead for the various indexing methods when evaluated on datasets that are divided into 4MB rowgroups. Note that we can achieve perfect scan overhead if rowgroup size is one, but that is not practical.

We can see that the scan overhead is lowest for the flights dataset. This can be explained by the fact that the queries in the workload filter over only a few columns (two to four) and almost all of the queries filter over a subset of the columns that were given bits in the best Z-order mapping. Specifically, the best 64-bit Z-order mapping for the flights dataset allocates bits to five columns, and the query filters mostly involve those five columns (e.g., if the five columns in the Z-order are col_a, col_b, col_c, col_d, and col_e, one query type is over col_b, col_c, and col_d and another query type is over col_a and col_b). Therefore, the data layout resulting from the best Z-order configuration allows for effective rowgroup skipping, hence the low scan overhead.

The taxi dataset also has a relatively low scan overhead for reasons similar to that of flights, where many of the queries filter over only a few columns and those columns are included in the Z-order.

In contrast, the contributions and tweetmap datasets have higher scan overhead. The contributions dataset has many queries that filter over four or more columns. Based on analysis of the rows scanned for each query, we can see that queries with four or more columns contribute to the greatest increase in scan overhead.

Meanwhile, most of the queries in the tweetmap workload filter over three or more columns. The Z-order index does not work well for this workload because the queries

involve many different columns, a total of ten columns. It is not possible to allocate an adequate number of bits to many columns due to the limited number of bits (and as discussed in Section 4.4.6 below, only the first x bits matter for query performance, where $x < 64$). As a result, the queries in the workload have high scan overhead.

	Contributions	**Flights**	**Taxi**	**Tweetmap**
Default sort order	610.29	20.23	208.22	1315.81
Range partitioning	59.69	19.02	158.87	29.57
Recursive cell splits	49.44	29.06	20.73	112.11
Z-order: equal bits to three columns	61.68	12.27	18.87	34.32
Z-order: best 64-bit allocation	32.40	8.98	12.44	29.79
Z-order: best allocation	31.43	8.46	11.90	29.57

Table 4.4: Scan overhead for the different indexing methods.

Factors Affecting Z-order Performance

To better understand how certain factors such as the number of blocks that the dataset is partitioned into, selectivity of queries, and number of columns used in the Z-order affect query performance, we examine how performance changes when these factors are varied.

We create a simple synthetic dataset containing one million rows and five columns, where each column's value is a random number chosen uniformly between zero and one million. We also create a random workload containing 100 queries, each with 10% selectivity. Each query filters over one column, and there are 20 queries per column.

Results show that increasing the number of blocks results in fewer rows scanned (see Fig. 4-5). For range partitioning, performance improves but eventually levels off as the number of blocks increases. This is because range partitioning can only help queries that filter over the column it is partitioned by, so performance no longer improves after having a sufficient number of blocks. On the other hand, having more blocks for Z-order results in a significant improvement in query performance because each block becomes more fine-grained, containing fewer points and therefore allowing for more effective block skipping. Increasing the number of blocks past a certain

point, however, would probably have a negative impact on query performance due to the high I/O cost.

In addition, performance increases when queries have a lower average selectivity, as shown in Fig. 4-6. This makes sense since fewer rows match the query filter, so more blocks can be skipped when scanning the dataset. Query performance is further improved when the data is divided into more blocks, which matches what was described above regarding Fig. 4-5.

Furthermore, the number of columns to include in the Z-order has an impact on performance, which varies based on the number of blocks the data is divided into (see Fig. 4-7). For this scenario of a uniform dataset and query workload, including more columns in the Z-order generally results in better performance since it allows for more effective query filtering for more columns and hence more queries. However, when the number of blocks is very small (10-20), performance initially improves but ultimately worsens. This is because when there are very few blocks, many Z-order values fall in the same block, rendering block skipping less effective. Allocating bits to many columns under those circumstances results in a data layout that further negatively impacts block skipping since each column has fewer bits, leading to ineffective filtering over the columns.

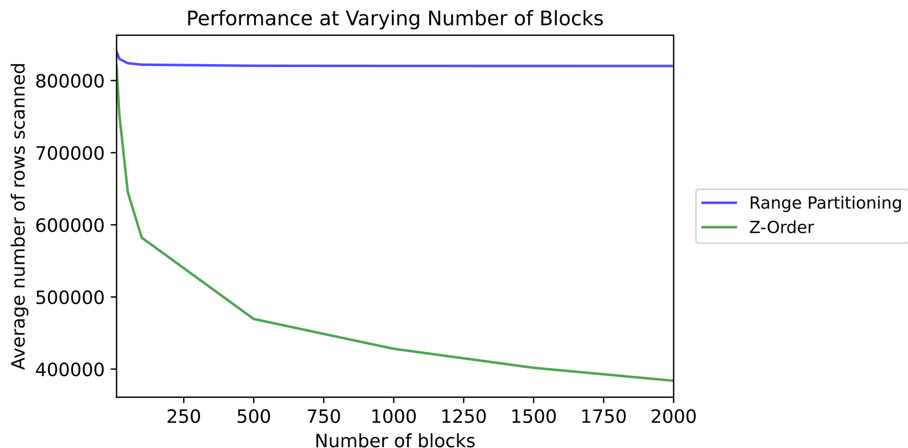


Figure 4-5: Performance increases as the number of blocks increases.

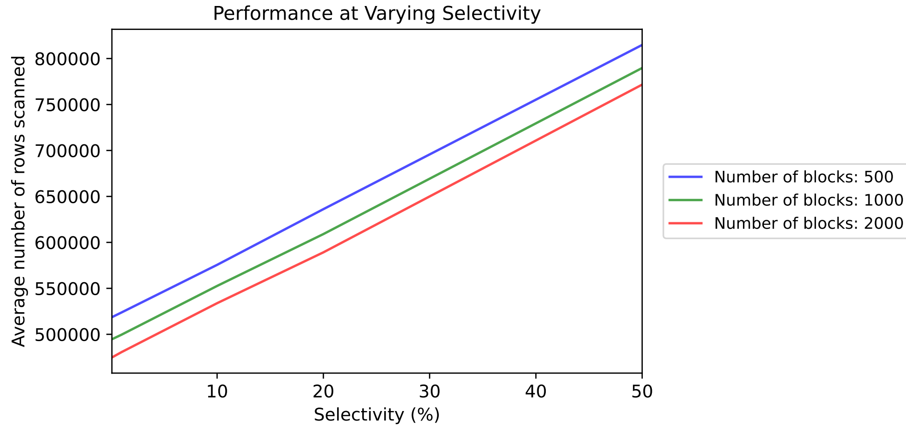


Figure 4-6: Performance increases for queries with lower average selectivity.

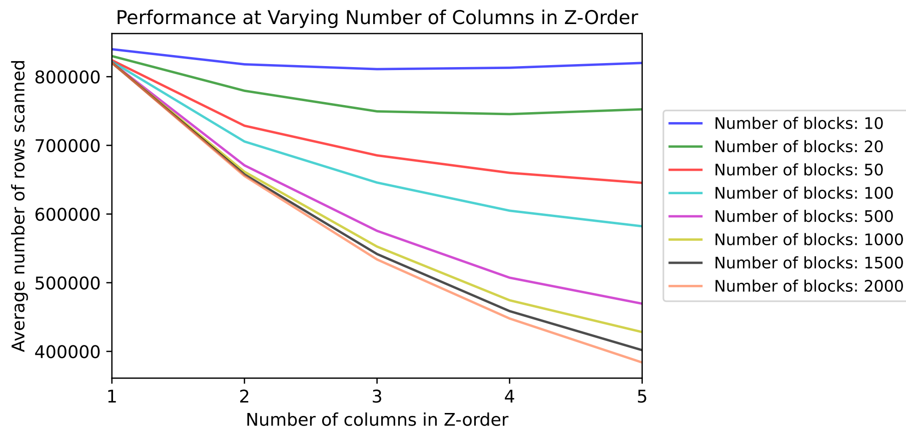


Figure 4-7: Performance generally increases when more columns are included in the Z-order. However, when there is a limited number of blocks, performance improves and eventually worsens.

4.4.6 Varying Total Number of Bits in Z-Order

For the 64-bit Z-order problem, although we allocate a total of 64 bits to the columns in the dataset, it appears that only the first x bits matter for query performance, where $x < 64$. This can be demonstrated by an example using the flights dataset.

A Z-order configuration that allocates bits to the three most selective columns is $\{col_7 : 31, col_5 : 5, col_{15} : 6\}$. The number of bits for each column is sufficient to cover the entire domain of the column. For example, column 15 values range from 0 to 52; it is possible to uniquely define every value in that range using six bits, as $2^6 = 64$.

Thus, adding bits to reach a total of 64 bits should result in an equivalent mapping.

One might think that since we have more bits left, we can give bits to the other columns that appear in the workload. For example, a possible configuration is $\{col_7 : 31, col_5 : 5, col_{15} : 6, col_6 : 3, col_{17} : 4, col_{19} : 9, col_1 : 6\}$. Although one might hypothesize that this would lead to better query performance, this Z-order mapping actually results in worse performance. This can be explained by the limited number of rowgroups that the dataset is divided into. Conceptually, having 64 bits means that there are 2^{64} possible Z-order values, so we would need 2^{64} rowgroups in order for each rowgroup to be associated with a unique Z-order value (we might not actually need 2^{64} rowgroups or each rowgroup to only correspond to one Z-order value to get good performance, but we can assume this for simplicity). If we consider a dataset that is divided into 5000 rowgroups, then it is as if this can only support 5000 Z-order values, which means approximately 13 bits in total. Thus, only the first 13 bits truly matter and help when scanning datasets. However, as mentioned earlier, it might not be necessary to have each rowgroup correspond to only one Z-order value, so it is possible that some x number of bits that is greater than 13 matter.

We ran an experiment to see how query performance changes as we vary the number of bits in the Z-order. For each k value (total number of bits in the Z-order value), we run Bayesian optimization five times, each with 600 iterations. In the figures below, we plot the lowest estimated query time seen for each of the k values considered.

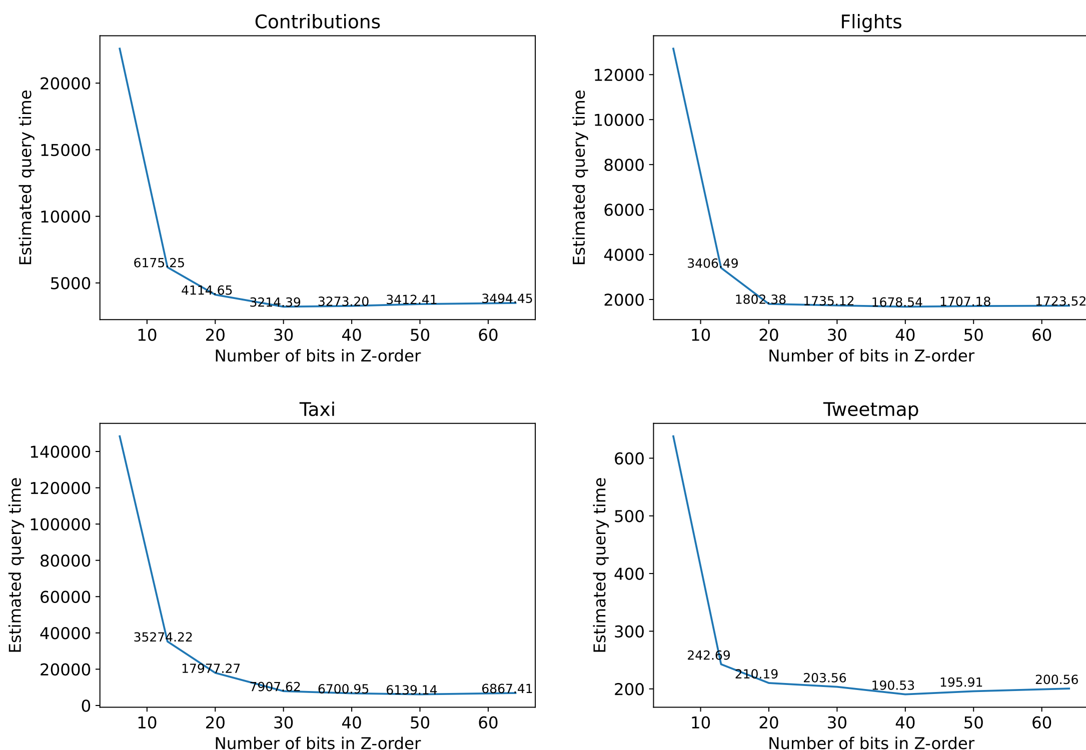


Figure 4-8: Lowest query time seen for a specific number of bits in the Z-order value.

From Fig. 4-8, we can see that the best performance is achieved at varying total number of bits in the Z-order. For the contributions dataset, the lowest query time occurs at 30 bits in the Z-order. For the flights and tweetmap dataset, the best Z-order mapping occurs at 40 bits in the Z-order. Meanwhile, for the taxi dataset, the lowest query time is observed at 50 bits in the Z-order. The difference in the optimal number of bits in the Z-order might be due to factors such as differences in the data distributions. For example, a more skewed/non-uniform data distribution might result in the optimal configuration occurring at a larger number of bits in the Z-order since having more bits can be highly beneficial, enabling greater distinction between the records.

It is worth noting that even though the best performance is achieved at x total number of bits, the best 64-bit configuration should be no worse than the best x-bit configuration. This is because it is always possible to fill the latter $64 - x$ bits

with "dummy bits" that do not matter. In the example above, instead of giving the remaining bits to columns 6, 17, 19, and 1, the remaining bits could all be given to column 7, which would be equivalent to the initial three-column Z-order configuration above. However, the larger search space makes the Bayesian optimization problem much more difficult. As the Bayesian optimization search is constrained by a certain number of iterations, it is easier for Bayesian optimization to find a good configuration if the total number of bits is smaller.

Another interesting observation is that having as little as 30 bits is often sufficient and results in only slightly worse performance than the optimal number of bits in the Z-order. This might be related to the discussion above where Z-order performance is constrained by the number of blocks that the dataset is divided into: because of the limited number of blocks that Z-order values can fall into, only the first x (e.g., 30) bits have a meaningful impact on performance; additional bits only offer slight improvements.

Our results suggest that allocating a total of 64 bits by default achieves high performance and that varying the value of k allows for some (up to 22.6%) improvement in practice.

Chapter 5

Conclusion and Future Work

Existing systems such as Databricks Delta Lake and Amazon Redshift give users the ability to sort multiple columns using Z-order. However, some drawbacks include the need to specify the columns and the equal weight given to the columns, which may affect query performance or lack thereof. Our work aims to determine the best Z-order index for a particular dataset and query workload. In this thesis, we present a learning-based approach to Z-order layout optimization using dynamic bit allocation, which allows different weights on the columns through the number of bits allocated to the columns. Learning from the query workload allows our Z-order indexes to outperform other partitioning/indexing methods. Our learned Z-order indexes achieve up to 30.2× reduction in rows scanned compared to the default sort order, up to 8.8× reduction compared to range partitioning, and up to 4.7× reduction compared to the index resulting from the recursive cell splits algorithm. Moreover, our Z-order indexes are up to 1.4× faster than the next best index. Our results suggest that tuning Z-order can lead to significant improvements in query performance. In addition, tuning k, the total number of bits in the Z-order, may result in improved performance (up to 22.6% on our datasets). Through our exploration of Z-order, we also learn that only the first x bits matter for query performance when allocating 64 bits to the columns, where $x < 64$ and varies across the datasets.

Our work can be extended to consider dynamic workloads, where the goal is to determine whether or not the workload has changed and when to reorganize the data

according to a new Z-order configuration. One possibility is to periodically evaluate the performance of the Z-order configuration on queries over a recent time window and replace the Z-order configuration if performance falls below a certain threshold. Another possible approach consists of forecasting the future query workload and determining the best candidate Z-order configuration for that predicted future workload. If the performance gain exceeds the repartitioning cost, then one would decide to repartition the data using the new Z-order configuration.

In addition, another area of exploration can be considering the context of changing datasets, in which rows are added, updated, or removed. For instance, one idea is logically adjusting the Z-order such as changing the domain of each bit or giving more bits to a column in response to row updates/insertions. This can either be reactive (only changing after a violating row has been inserted) or proactive (e.g., anticipating that the domain of a column will expand in the future and therefore giving the column more bits than initially needed).

Bibliography

[1] Apache Arrow. https://arrow.apache.org/.

[2] Apache Parquet. https://parquet.apache.org/.

[3] Rudolf Bayer. The Universal B-Tree for Multidimensional Indexing: General Concepts. In Takashi Masuda, Yoshifumi Masunaga, and Michiharu Tsukamoto, editors, *Worldwide Computing and Its Applications*, pages 198–209. Springer Berlin Heidelberg, 1997.

[4] Nigel Bayliss. Optimizing Table Scans with Zone Maps. https://blogs.oracle.com/datawarehousing/post/optimizing-table-scans-with-zone-maps, 2014.

[5] Zach Christopherson. Amazon Redshift Engineering's Advanced Table Design Playbook: Compound and Interleaved Sort Keys. https://aws.amazon.com/blogs/big-data/amazon-redshift-engineerings-advanced-table-design-playbook-compound-and-interleaved-sort-keys/, 2016.

[6] Jialin Ding, Vikram Nathan, Mohammad Alizadeh, and Tim Kraska. Tsunami: A Learned Multi-dimensional Index for Correlated Data and Skewed Workloads. *CoRR*, abs/2006.13282, 2020.

[7] Peter I. Frazier. A Tutorial on Bayesian Optimization, 2018.

[8] John H. Holland. Genetic Algorithms. *Scientific American*, 267(1):66–73, 1992.

[9] Adrian Ionescu. Processing Petabytes of Data in Seconds with Databricks Delta. https://databricks.com/blog/2018/07/31/processing-petabytes-of-data-in-seconds-with-databricks-delta.html, 2018.

[10] Vikram Nathan, Jialin Ding, Mohammad Alizadeh, and Tim Kraska. Learning Multi-dimensional Indexes. *CoRR*, abs/1912.01668, 2019.

[11] Fernando Nogueira. Bayesian Optimization: Open source constrained global optimization tool for Python. https://github.com/fmfn/BayesianOptimization, 2014–.

[12] OmniSci. https://www.omnisci.com/. Accessed: 2021-03-24.

[13] Beng Chin Ooi, Ron Sacks-Davis, and Jiawei Han. Indexing in Spatial Databases, 2019.

[14] Sachith Pai, Michael Mathioudakis, and Yanhao Wang. Towards an Instance-Optimal Z-Index [Extended Abstract]. AIDB, 2022.

[15] Frank Ramsak, Volker Markl, Robert Fenk, Martin Zirkel, Klaus Elhardt, and Rudolf Bayer. Integrating the UB-Tree into a Database System Kernel. In *Proceedings of the 26th International Conference on Very Large Data Bases*, VLDB '00, page 263–272, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[16] Zack Slayton. Z-Order Indexing for Multifaceted Queries in Amazon DynamoDB. https://aws.amazon.com/blogs/database/z-order-indexing-for-multifaceted-queries-in-amazon-dynamodb-part-1/, 2017.

[17] Ryan Solgi. Geneticalgorithm. https://github.com/rmsolgi/geneticalgorithm, 2020.