Learning Bit Allocations for Z-Order Layouts in Analytic Data Systems

Learning Bit Allocations for Z-Order Layouts in Analytic Data Systems

Jenny Gao
Massachusetts Institute of Technology
jgao86@mit.edu

Jialin Ding*
Amazon Web Services
jialind@amazon.com

Sivaprasad Sudhir
Massachusetts Institute of Technology
siva@csail.mit.edu

Samuel Madden
Massachusetts Institute of Technology
madden@csail.mit.edu

ABSTRACT

To improve the performance of scanning and filtering, modern analytic data systems such as Amazon Redshift and Databricks Delta Lake give users the ability to sort a table using a Z-order, which maps each row to a "Z-value" by interleaving the binary representations of the row's attributes, then sorts rows by their Z-values. These Z-order layouts essentially sort the table by multiple columns simultaneously and can achieve superior performance to single-column sort orders when the user's queries filter over multiple columns. However, the user shoulders the burden of manually selecting the columns to include in the Z-order, and a poor choice of columns can significantly degrade performance. Furthermore, these systems treat all columns included in the Z-order as equally important, which often does not result in the best performance due to the unequal impact that different columns have on query performance. In this work, we investigate the performance impact of using Z-orders that place *unequal* importance on columns: instead of using an equal number of bits from each column in the Z-value interleaving, we allow unequal bit allocation. We introduce a technique that uses Bayesian optimization to automatically learn the best bit allocation for a Z-order layout on a given dataset and query workload. Z-order layouts using our learned bit allocations outperform equal-bit Z-orders by up to 1.6× in query runtime and up to 2× in rows scanned.

ACM Reference Format:
Jenny Gao, Jialin Ding, Sivaprasad Sudhir, and Samuel Madden. 2024. Learning Bit Allocations for Z-Order Layouts in Analytic Data Systems. In *Seventh International Workshop on Exploiting Artificial Intelligence Techniques for Data Management (aiDM '24), June 14, 2024, Santiago, AA, Chile.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3663742.3663975

1 INTRODUCTION

Scanning and filtering data is an important operation in analytical databases. To improve scan performance, analytic databases often horizontally partition tables into *blocks* and maintain a *zone map* for each block, which contains metadata such as minimum/maximum values per column [2]. When performing scans, the database engine first checks each zone map to determine if any relevant records might exist in the block and only scans the blocks that are relevant to a query.

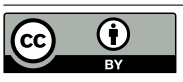
To increase the likelihood that blocks can be skipped, users typically sort their tables by a column that is commonly used in filters. For cases in which a table is commonly filtered by multiple different columns, some analytic databases support multi-column sort orders. One commonly used sort order technique involves a multi-dimensional space-filling curve called the Z-order, which maps multi-dimensional data to scalar "Z-values" while preserving the 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 their mapped Z-values. The Z-value for a record is calculated by interleaving the bits of the binary representation of the column values in a round-robin fashion. For example, Fig. 1a shows how a record with two three-bit columns is mapped to a four-bit Z-value by interleaving the first two bits from each column value.

Systems such as Amazon Redshift and Databricks Delta Lake allow users to sort on multiple columns using Z-orders. However, the Z-order is typically not constructed over all columns of the table (e.g., if sorting by a particular column has no impact on query performance, then there is no reason to include it in the Z-order). Therefore, users must manually choose which columns to include in the Z-order. One heuristic approach they might take is to choose the top n most important columns, where importance is defined by the average selectivity of filters over the column or the frequency with which the column is used in filters. However, a poor choice of columns might result in scanning many more rows than necessary that are not relevant to the query.

Furthermore, these systems give equal weight to the columns included in the Z-order, in the sense that a roughly equal number of bits from each column are included in the Z-order value due to the round-robin nature of bit interleaving. Such an approach, however, might not result in the best performance, since different columns have different amounts of impact on query performance.

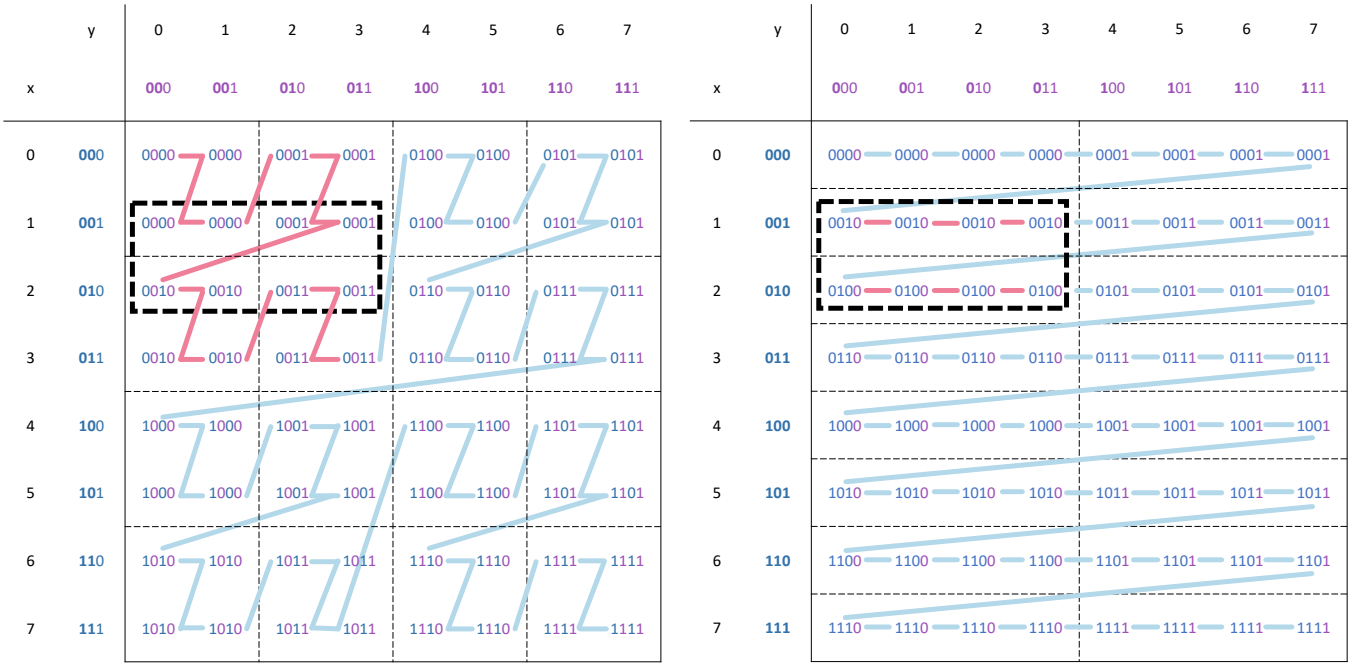
Therefore, in this paper, we consider Z-order layouts in which *unequal* weight is placed on different columns. Fig. 1 shows an example where allocating an unequal number of bits to each column results in better performance than equal bit allocation. This example table has 64 records and two columns, x and y, and each record is visualized as an (x,y) point on a two-dimensional plane. We consider four-bit Z-values: the Z-value for each (x,y) coordinate pair by interleaving the bits of the x and y columns. Using Z-ordering, we can arrange

*Work done prior to joining Amazon.



aiDM '24, June 14, 2024, Santiago, AA, Chile

ACM ISBN 979-8-4007-0680-6/24/06.
https://doi.org/10.1145/3663742.3663975



(a) A Z-order configuration with equal bit allocation (2 bits each from columns x and y) in four blocks being scanned.

(b) A Z-order configuration with unequal bit allocation (3 bits from column x and 1 bit from column y) results in two blocks being scanned.

Figure 1: Example of two possible Z-order data layouts on a table with two integer columns, x and y, each with domain between 0 and 7. The table contains 64 unique records, each of which is visualized as an (x,y) point on a two-dimensional plane, and each block contains 4 records. If a query needs to find all points that satisfy the filter ($x \geq 1$ `AND` $x \leq 2$ `AND` $y \geq 0$ `AND` $y \leq 3$), a Z-order configuration with an uneven bit allocation (b) results in fewer blocks scanned than one with an equal bit allocation (a).

the two-dimensional pairs on a one-dimensional line (Fig. 1a), which follows a "Z" shape, hence the name "Z-order curve." The block size is four records; block boundaries are indicated by the black dotted lines, which split the Z-shaped curve into blocks of four records each. 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. 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. 1b, we show another layout where we allocate three bits to the x column and one bit to the y column. The Z-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. 1a. This demonstrates the importance of considering the weight of each column in the Z-order: the filter over column x is more selective than the filter over column y, so for this particular query, sorting by column x is more impactful than sorting by column y, and so the Z-order bit allocation should reflect their relative importance.

In line with the insight that the default approach of allocating equal bits to columns might not result in the best performance, we propose an approach to automatically choose the best unequal-bit Z-order configuration for a given dataset and query workload. We use Bayesian optimization to search over possible configurations and evaluate each candidate configuration by using a cost model to estimate the average query runtime under that configuration.

One non-obvious insight we gained through our investigation is that only the first k bits of the Z-value really matter in improving query performance. However, the value of k varies depending on the dataset and workload characteristics, and it is important to set k appropriately: if k is too small, we miss out on performance, and if k is too large, then Bayesian optimization may not be able to explore the large search space efficiently. Instead of tuning k as a hyper-parameter, which would require running the Bayesian optimization multiple times, we instead directly embed k as an additional parameter in our Bayesian optimization formulation.

We evaluate the performance of Z-order layouts using our learned bit allocations against several baselines, including equally-weighted Z-orders over manually selected columns and range partitioning over a single column, over four real-world workloads. Our benchmarks show that our learned Z-order layouts achieve up to 1.6× better query runtime performance and 2× improvement in rows

scanned compared to equal-bit Z-orders and up to 5× better query performance compared to range partitioning.

The remainder of the paper is organized as follows. Section 2 provides background. Section 3 presents our approach to producing the best Z-order layout for a particular dataset and query workload. We evaluate our learned Z-order indexes and discuss some of the insights gained in Section 4. Finally, Section 5 concludes the paper and identifies areas for future work.

2 BACKGROUND

In this section, we review the usage of zone maps and Z-order in commercial database systems and prior work on Z-order indexes and other multi-dimensional layouts.

Per-block zone maps, which track summary statistics such as the minimum and maximum value of each column in the block, are the predominant form of data-skipping indexes in modern analytic systems. Zone maps allow blocks that do not satisfy the scan filter predicates to be skipped, thereby improving scan performance. Zone maps are used by Amazon Redshift, Azure Synapse, and Snowflake, as well as data lakehouse systems such as Databricks Delta Lake that run on open formats such as Parquet and ORC that store zone map information in the file footer. Zone maps are coarse-grained indexes in the sense that they store metadata at the block level, and they are prevalent in analytic systems that support large scans. On the other hand, fine-grained row-level indexes such as the B+ tree are more prevalent in OLTP-oriented databases.

Zone maps are most effective when the table is sorted, so rows that are filtered by the same queries are typically co-located in the same blocks. Databricks Delta Lake [8] and Amazon Redshift [3] both give users the ability to sort tables over multiple columns using Z-orders. In the case of Amazon Redshift, this is called an interleaved sort key since Z-values are computed by interleaving the bit representations of column values. Currently, users of these systems must manually specify the columns to use in the Z-order.

Analytic systems also often support a form of multi-column sorting based on compound sort keys, which sort by multiple columns in succession. In compound sort keys, there is a clear order of importance to the columns, whereas in Z-order, all columns are treated roughly equally. Therefore, compound sort keys are ideal when there is one dominant column, but they suffer when multiple columns are important, or when the dominant column has a high number of distinct values (and therefore, there is little sorting effect on the other columns).

It is interesting to consider why Z-order curves became the predominant method for sorting equally by multiple columns in commercial analytic systems, even though a number of other space-filling curves with locality-preserving properties have been extensively explored in the broader scientific literature [15]. For example, the Hilbert curve has actually been shown to possess more desirable locality-preserving behavior than Z-orders under certain metrics [5, 9]. Although the design choice to use Z-order curves over other space-filling curves has not been explicitly explained by any commercial systems, we conjecture that it has to do with a combination of simplicity, interpretability, and performance. Z-orders are based on a bit interleaving, which is easily understandable and computationally efficient. On the other hand, computing analogous values for a Hilbert curve is much more complex [16].

Other works have proposed more complex index structures based on Z-order. The UB-tree [1, 14] is based on the standard B-tree but uses the Z-order curve to partition multi-dimensional space into regions called Z-regions, each of which covers an interval on the Z-order curve and is mapped 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 (i.e., the lower left and upper right vertices of the query rectangle) and retrieving all the Z-regions that intersect the query rectangle. Similar to UB-tree, Amazon DynamoDB allows users to create an index on the Z-order value [17]. 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. However, analytic data systems typically eschew fine-grained indexes in favor of zone maps, which are more suited to the large scan-heavy queries seen in most workloads, and so the UB-tree and its variants have not been implemented in commercial analytic systems.

Our work aims to learn the best Z-order configuration for a given dataset and query workload. There has been recent work on learned multi-dimensional indexes such as Flood [10] and Tsunami [4]. 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 [4]. Our work draws ideas from these systems to evaluate candidate Z-order bit allocation configurations.

Prior work on learned Z-order indexes primarily focus on how to use learned models to search for the record with a given Z-value efficiently, but they do not address how to select the columns to include in the Z-order, nor do they address the issue of unequal column importance. For example, the ZM-index [20] is a learned model index that combines the Z-order curve with a model that learns the distribution of Z-values.

A recent paper [13] proposes an instance-optimal variant of the Z-order index that does not use bit interleaving to construct Z-order values, but rather divides multi-dimensional space into cells by partitioning each dimension at selected values, then defines a curve that visits each cell in some locality-preserving order. However, we consider bit interleaving a fundamental characteristic of the Z-order: its simplicity and interpretability are part of the reason that modern data systems choose to support Z-order-based sorting. Therefore, we do not consider [13] a learned Z-order index, but rather a more general learned multi-dimensional index similar to Flood [10] and Tsunami [4].

Furthermore, there are various multi-dimensional index structures, such as R-trees and k-d trees [12], but these are generally specialized for spatial data and are not used for analytic databases.

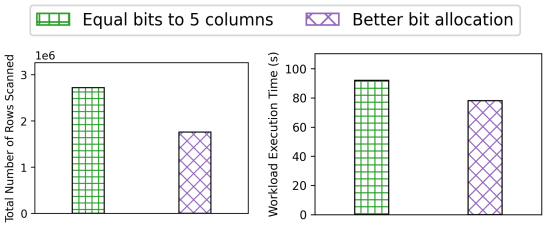


Figure 2: For the synthetic dataset and workload described in Table 1, a Z-order configuration with unequal bit allocation results in better performance, in terms of both number of rows scanned and query runtime, than one with equal bit allocation.

3 OVERVIEW

In this section, we first formally present the problem statement. We then motivate the problem statement through a concrete example of how Z-order performance can be improved through unequal bit allocation. We then explain our approach for automatically finding the best Z-order configuration for a given dataset and query workload.

3.1 Problem Statement

We begin with some definitions:

Definition 3.1 (Z-Order configuration). We define a Z-order configuration to be an allocation of bits to columns. Suppose we have columns col_0 to col_{n-1}. A Z-order configuration can be written as a set 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 bits 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 3.2 (Z-value under a configuration). 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 record $(c_0, c_1, \ldots, c_{n-1})$, the k-bit Z-value under this configuration can be constructed by evaluating $m = min(v_0, v_1, \ldots, v_{n-1})$ and interleaving $\lfloor v_i/m \rfloor$ bits of each column at a time in a round-robin fashion (if there are fewer than $\lfloor v_i/m \rfloor$ bits remaining in the allocation, we use all remaining allocated bits). 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 under this configuration 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 record* $(c_0,c_1,c_2,...,c_{n-1})$. *Let* $c_i[j]$ *denote the* j*-th bit of the binary representation of column* i. *Then the Z-value for the record has the following binary representation:*

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

Given a table with n columns (i.e., a dataset) and a query workload in which each query is a filter over the dataset followed by a projection over a subset of columns, the goal is to find a k-bit Z-order configuration that minimizes the total runtime of all queries in the workload. The value of k is not known beforehand, so finding an appropriate k is part of the problem statement. In our setup, we constrain that $k \leq 64$ as we use an 8-byte integer to represent the Z-value. Based on our experiments in Section 4, we expect 64 bits to be sufficient for many realistic datasets.

More concretely, the goal is to find the best *mapping* of k bits to n columns. A candidate mapping is represented as a length-n vector v, in which each element is an integer representing the number of bits allocated to the corresponding column, and the sum of all elements is k. A zero in the vector signifies that the corresponding column is not included in the Z-order. The rest of the paper uses the terms *mapping* and *allocation* interchangeably.

If v_i bits are allocated to column i, we use the most significant v_i bits from each column value in the Z-value. However, in reality, the values of a column often only span a small subset of the possible domain, e.g., a column with 4-byte integer type only has values between 0 and 1000, so the most significant 22 bits are always zero. Therefore, as an optimization, we use v_i to denote the number of "interesting" most significant bits, i.e., we ignore all leading bits whose values are the same for all records in the dataset.

Table 1: Synthetic workload characteristics.

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

3.2 Motivating Example

The choice of bit allocation in a Z-order configuration can have a significant impact on query performance. As a motivating example, consider the synthetic dataset described in Table 1, which consists of 10 million rows where each column's value is a random integer chosen uniformly between zero and the maximum column value. The query workload contains 500 queries, with each query filtering over one column and projecting over that same column.

In Fig. 2, we compare a 64-bit 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 ($\{col_0 : 3, col_1 : 3, col_2 : 17, col_3 : 22, col_4 : 19\}$). For each configuration, we sort the dataset's rows by the corresponding Z-order, then horizontally partition the dataset into 5000 equally-sized blocks, each with 2000 rows, and then build a zone map on each block that stores the min/max value per column in the block. For each of the 500 queries, we then use the zone maps to determine whether or not each block's rows need to be scanned. We 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 predicates over them are less selective, so a small number of bits is sufficient. On the other hand, columns 2, 3, and 4 have a much larger domain. Filters over them are very selective, and they appear in many queries, so it is essential to allocate more bits to these columns. This example demonstrates that considering the columns to include in the Z-order and the number of bits allocated to each column can have a significant impact on performance.

3.3 Approach

There are two parts to our approach to tackling the problem statement. First, we need a search algorithm to search the space of possible configurations, i.e., the n-dimensional space of candidate mappings. We use Bayesian optimization as our search algorithm. Second, the true objective function that we want to minimize as part of Bayesian optimization (total query runtime) is too expensive to evaluate since it would require sorting the dataset by the candidate configuration and running all the queries, so we need a proxy objective function that is cheaper to evaluate. We use an analytic cost model as our proxy objective function. We now describe these two parts in more detail.

3.3.1 Search Algorithm. To search the space of possible configurations, we use Bayesian optimization [6], which is a derivative-free global optimization method for black-box objective functions. Bayesian optimization is a natural choice of search algorithm for our problem for two reasons:

- Bayesian optimization is well-suited to expensive-to-evaluate objective functions as it typically requires fewer invocations of the objective function than other global optimization methods. Since our objective function is relatively expensive to evaluate (see Section 3.3.2), Bayesian optimization helps limit the amount of time we spend finding the best Z-order configuration.
- Bayesian optimization tolerates stochastic noise and uncertainty in the objective function. Since the objective function we use for the optimization (i.e., the output of our cost model) is only a proxy for the true objective function (i.e., total runtime of the query workload), there is uncertainty in our objective function evaluations.

Bayesian optimization is an iterative search algorithm: during each iteration of the search, the algorithm proposes a point to sample from its search space, evaluates this point using the objective function (see Section 3.3.2 for a description of our objective function), stores the evaluation result, and uses the history of sampled points and evaluation results to suggest a point to sample for the next iteration.

We formulate our Bayesian optimization search space as follows: for a given dataset and query workload, we first identify the columns that appeared in filters in the query workload (we do not consider any other columns since allocating bits to them would not improve the performance of scanning and filtering). If there are n such columns, then we create an $(n+1)$-dimensional search space. The first n dimensions have continuous domains between 0 and 1, and they represent the relative number of bits allocated to each corresponding column. The last dimension has a discrete domain of integers between 1 and 64, inclusive, and it represents k.

Given a sampled point $(p_0, p_1, ..., p_n)$ from the search space, we construct a candidate mapping that represents a p_n-bit Z-order configuration, in which the number of bits allocated to col_i is $\left(p_n \cdot \frac{p_i}{\sum_{j=0}^{n-1} p_j}\right)$. If the bit allocation is a fractional number, then we round it to the nearest integer. If the resulting candidate mapping has less than (or more than) k bits, which may happen due to skew during rounding, then we modify the candidate mapping to have exactly k bits through a process of randomly selecting a column and adding (or removing) a bit until reaching the desired number bits. The candidate mapping is then inputted to the objective function (see Section 3.3.2).

3.3.2 Cost Model. Our cost model is a proxy objective function: it takes a candidate mapping as input and produces the estimated runtime of a query. The cost model that we use for a single query is:

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

The constant term w represents the time to scan a single column of a single point and is set empirically based on experiments. The number of filtered columns is clear from the query itself. To compute the exact number of rows scanned, we would need to sort the dataset by the candidate configuration, horizontally partition the dataset into blocks, construct zone maps over each block, then examine each block's zone map to determine whether the query would skip the block (i.e., not scan its rows).

However, this step of sorting the entire dataset by a candidate configuration is time-consuming, especially when it must be done for each iteration of the search algorithm. Therefore, instead of computing the exact number of rows scanned, we estimate the number of rows scanned by only sorting and creating zone maps over a sample of the dataset. At the beginning of the overall Bayesian optimization, we create a sample dataset where rows are chosen uniformly from the dataset. Then, for each evaluation of the objective function, we estimate the rows scanned statistic using the sample dataset, which is much more efficient than computing the true statistic using the full dataset.

4 EVALUATION

In this section, we evaluate the performance of our learned Z-order bit allocation approach. We present the results of an experimental study that compares our approach with traditional equal-bit Z-orders and other sorting methods on several datasets and query workloads. Results reveal that:

- Our learned Z-order configurations are up to 1.6× faster in query runtime and scan up to 2× fewer rows than equal-bit Z-orders (Section 4.4). Our learned Z-order configurations are never slower than other sort techniques, whereas equal-bit Z-order sometimes underperforms a single-column range partitioning.
- Only the first k bits in a Z-value have a significant impact on query performance, and Z-order configurations with as few as 20 bits can be as good as 64-bit Z-order configurations (Section 4.5.1). To make the Bayesian optimization more tractable, it is useful to learn k as part of the optimization procedure.
- Our cost model accurately reflects true query execution time (Section 4.5.2), and learning the configuration on a dataset with 175M rows and 13 columns completes within 12 minutes (Section 4.5.3).

4.1 Baselines

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

(1) *Default sort order*: queries are performed on the original dataset, without any explicit sort order. In most cases, the original dataset is implicitly sorted on a timestamp column, since that is the order of ingestion. 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) *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 our learned unequal-bit Z-order configuration.

4.2 Implementation

4.2.1 Z-Order. Tables are stored in Parquet format [19], a column-oriented file format, on disk; data is partitioned into sets of rows called *rowgroups*, and within each rowgroup, data from different columns are stored separately. The rest of this paper uses the terms *rowgroup* and *block* interchangeably. The Apache Arrow C++ library [18] is used to work with the Parquet files, such as loading the Parquet files in memory. For Bayesian optimization, we use the Python package [11]. For each dataset and workload, we run Bayesian optimization for 600 iterations to find the best Z-order configuration.

4.2.2 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 4MB. Parquet stores the minimum/maximum statistics per column for each rowgroup and uses these statistics to skip rowgroups when executing queries. All the experiments are single-threaded and run on an Arch Linux machine with an Intel Xeon 2.1GHz CPU and 125GB RAM.

4.3 Datasets and Query Workloads

We evaluate indexes on four real-world datasets collected from [7]: contributions, flights, taxi, and tweets (see Table 2). 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.

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 tweets 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 tweets, which samples 10% of the rows from the full dataset since the full dataset is much smaller than those of other datasets). These sample datasets are used during optimization: for each candidate mapping, the rows scanned statistic is estimated using the sample dataset and inputted into the cost model to estimate the query time (see Section 3.3.2). If the full dataset contains x rowgroups, we also partition the sample dataset into x rowgroups when estimating the rows scanned statistic.

Table 2: Dataset and query characteristics.

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

4.4 Overall Results

We show how well our learned Z-order bit allocation approach performs when compared to the baseline sort order approaches.

Fig. 3 shows the average number of rows scanned for each sort order on each dataset. We observe that our learned 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 layout. On three of the datasets, the Z-order configuration produced by our approach achieves between 1.2× and 2× reduction in the number of rows scanned compared to the next-best layout.

Fig. 4 shows the average query time for each sort order on each dataset. The trends for query runtime are similar to those for the average number of rows scanned (Fig. 3).

Overall, these results show that our learned Z-order configurations generally resulted in a significant improvement in performance when compared to the default sort order and range partitioning. This is due to the fact that Z-order allows for multi-dimensional sorting, which is particularly beneficial since the queries in the workload filter over multiple columns.

[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 $CDF(\text{right end of range}) - CDF(\text{left end of range})$. A query of the form $x < col < y$ would contribute $CDF(y) - CDF(x)$ to the average selectivity for that column.
[2] For each column, we count the number of queries it appears in.

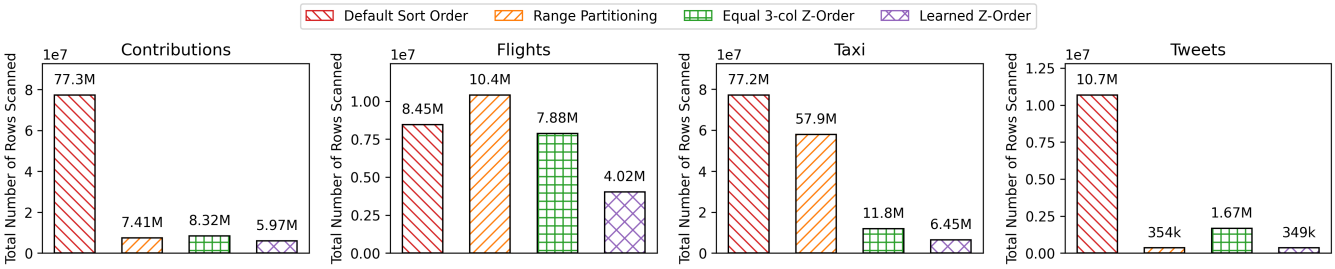


Figure 3: Total number of rows scanned for the different indexing/partitioning methods.

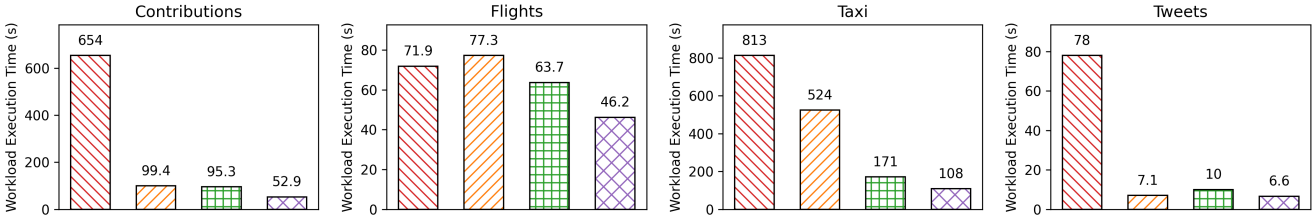


Figure 4: Total workload execution time for the different indexing/partitioning methods.

In general, Z-order layouts are most effective for workloads where most of the queries filter over a relatively small number of columns. For workloads in which queries filter primarily over one column, range partitioning performs as well as Z-order. For workloads in which queries filter over a large number of columns, we can only allocate a few bits to each column, which may not be enough to allow for effective query filtering, especially since only the first k bits truly matter for performance (see Section 4.5.1).

For example, Z-order performs particularly well for the Flights dataset, in which the learned Z-order configuration allocated bits to five total columns, and the queries in the Flights workload each filter over two to four of the columns among the five columns used in the Z-order. The Z-order configuration for the Taxi dataset displayed similar behavior.

In contrast, the Contributions workload consisted of many queries that filter over four or more columns. Based on analysis of the rows scanned for each query, we observed that queries with four or more columns contribute to the greatest increase in query runtime and rows scanned.

Meanwhile, most of the queries in the Tweets workload filter over a total of ten different columns, but one column clearly appears most frequently in filters and is the most selective. This makes Z-order doubly ineffective: first, it is not possible to allocate an adequate number of bits to all ten columns, especially since only the first k total bits really matter for improving performance. Secondly, since there is one dominant column in filters, a range partitioning over that column achieves performance that is nearly as good as Z-order.

In addition, we 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. The former is improved through using better Z-order configurations, but the latter is constant among all the different partitioning methods. Moreover, Parquet stores rowgroup metadata such as the minimum/maximum value for each column in the file footer. Each partitioning 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.

4.5 Microbenchmarks

We now use microbenchmarks to evaluate more detailed performance characteristics.

4.5.1 Number of Bits in the Z-Order. Although we are allowed to allocate up to 64 bits to the columns in the dataset, it turns out that only the first k bits really matter for query performance, where k varies for each dataset and workload but is typically less than 64.

Fig. 5 shows, for each value of k between 1 and 64, the estimated performance of the best k-bit Z-order configuration found during Bayesian optimization from Section 4.4. The plot demonstrates that many different values of k are explored during the 600 iterations of Bayesian optimization. More importantly, they demonstrate that Z-order configurations with as few as 20 bits are able to achieve performance that is roughly on par with 64-bit configurations.

This implies that only the first k bits of a Z-value are important for improving performance. There are two reasons for this:

- For datasets in which columns have a low number of distinct values, a small number of bits is already sufficient to differentiate all unique values. For example, on the Flights dataset, one of the frequently-filtered columns consists of integers that range from 0 to 52. Therefore, it is possible to uniquely identify every value in that range by allocating 6 bits to the column, since $2^6 \geq 53$, and there is no reason to allocate more than 6 bits to the column.
- Once a set of rows are placed in the same block/rowgroup, there is no benefit (in terms of skipping blocks) to further sorting the rows within the rowgroup. For example, imagine that under k-bit Z-order configuration, the min/max Z-values

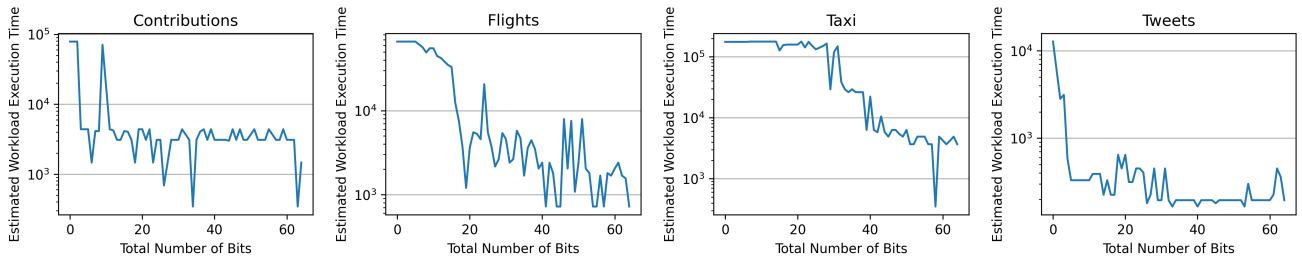


Figure 5: Estimated performance of the best k-bit Z-order configuration found during Bayesian optimization for each value of k. For most datasets, k much less than 64 is sufficient for achieving good performance. Note the log scale on the y-axis.

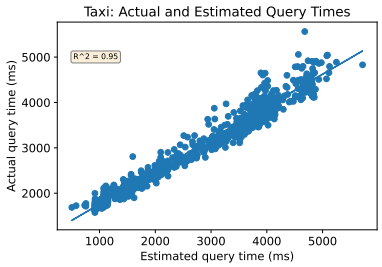


Figure 6: Actual vs. estimated query times (using the cost model in Section 3.3.2) for a workload.

in each rowgroup are non-overlapping, i.e., there is no case in which rows in different blocks have the same Z-value. Then, increasing the number of bits used in the Z-value would only shuffle the order of rows within a rowgroup, but would not result in rows being shuffled between rowgroups, so the zone maps for each rowgroup remain the same, and there is no impact on rowgroup skipping.

The value of k at which more bits have no impact on performance is difficult to compute upfront, since it depends on not only the size of the dataset and the number of rowgroups it is divided into, but also the domain, cardinality, and data distribution of each column.

Including k as a parameter in the Bayesian optimization is important because it allows us to directly learn the appropriate value of k and thereby restrict the search space to a more tractable size. On the other hand, we found that running Bayesian optimization for the same number of iterations, but with k fixed to 64, resulted in finding Z-order configurations that performed up to 20% worse, because it is harder for Bayesian optimization to efficiently explore the unnecessarily large search space of 64-bit Z-order configurations.

4.5.2 Cost Model. In Fig. 6, we plot the estimated and actual times of individual queries in the Taxi workload, under the learned Z-order configuration. The results show that the cost model produces relatively accurate estimates of query time, which is important for comparing candidate mappings during the search for the best Z-order configuration.

4.5.3 Z-Order Index Creation. Table 3 shows the time it takes to create the learned Z-order configuration. We separate time into learning time, which is the time taken for Bayesian optimization to run 600 iterations, and repartitioning time, which is the time to compute the Z-values for the full dataset and sort by them.

Table 3: Best Z-order index creation time in seconds.

	Contributions	Flights	Taxi	Tweets
Learning	438.2	1015.6	718.5	93.1
Sorting	135.4	265	271.5	31.2
Total	573.6	1280.6	990.0	124.3

Note that we did not fully optimize the performance of the learning code; with further optimization, the learning time should decrease. For example, instead of sampling and evaluating one point at a time from the search space, parallel Bayesian optimization techniques can process multiple points simultaneously [6]. Furthermore, Bayesian optimization is an anytime algorithm: we can stop it at any number of iterations and take the best configuration found so far. If decreasing the training time is important, then it is always possible to run Bayesian optimization for fewer iterations, though this would mean that we might find a sub-optimal Z-order configuration.

5 CONCLUSION AND FUTURE WORK

Analytic data systems such as Databricks Delta Lake and Amazon Redshift give users the ability to sort a table by multiple columns using Z-orders. However, Z-orders traditionally place equal weight on the table's columns, which does not result in the best performance when different columns have an unequal impact on performance. Our work uses a Bayesian optimization approach to learn the best unequal-weight Z-order configuration for a particular dataset and query workload. Our learned Z-order configurations outperform other sort order methods, including traditional equal-weight Z-orders and single-column range partitioning, by up to 1.6× in query runtime and 2× in rows scanned.

Z-order data layouts are a rich area for future work. In particular, one open question is how to handle dynamic datasets, in which new rows inserted into the dataset expand the domain of certain columns, such that the bits used to compute the Z-value are no longer the most significant bits of that column. In order to maintain the invariant that the most significant "interesting" bits are used from each column, we would either need to re-compute Z-values using the new set of bits, or anticipate the domain expansion during the initial Z-order optimization and over-allocate bits to the column, or a combination of both.

REFERENCES

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

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

[3] Zach Christopherson. 2016. 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/.

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

[5] C. Faloutsos and S. Roseman. 1989. Fractals for Secondary Key Retrieval. In *Proceedings of the Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (Philadelphia, Pennsylvania, USA) *(PODS '89)*. Association for Computing Machinery, New York, NY, USA, 247–252. https://doi.org/10.1145/73721.73746

[6] Peter I. Frazier. 2018. A Tutorial on Bayesian Optimization. https://doi.org/10.48550/ARXIV.1807.02811

[7] HEAVY.AI. [n.d.]. OmniSci. Retrieved March 24, 2021 from https://www.omnisci.com/

[8] Adrian Ionescu. 2018. 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.

[9] Jonathan K. Lawder and Peter J. H. King. 2000. Using Space-Filling Curves for Multi-Dimensional Indexing. In *Proceedings of the 17th British National Conferenc on Databases: Advances in Databases (BNCOD 17)*. Springer-Verlag, Berlin, Heidelberg, 20–35.

[10] Vikram Nathan, Jialin Ding, Mohammad Alizadeh, and Tim Kraska. 2019. Learning Multi-dimensional Indexes. *CoRR* abs/1912.01668 (2019). arXiv:1912.01668 http://arxiv.org/abs/1912.01668

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

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

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

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

[15] Hans Sagan. 2012. *Space-filling curves.* Springer Science & Business Media.

[16] John Skilling. 2004. Programming the Hilbert curve. *AIP Conference Proceedings* 707, 1 (04 2004), 381–387. https://doi.org/10.1063/1.1751381 arXiv:https://pubs.aip.org/aip/acp/article-pdf/707/1/381/11557416/381_1_online.pdf

[17] Zack Slayton. 2017. 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/.

[18] The Apache Software Foundation. [n.d.]. Apache Arrow. https://arrow.apache.org/

[19] The Apache Software Foundation. [n.d.]. Apache Parquet. https://parquet.apache.org/

[20] Haixin Wang, Xiaoyi Fu, Jianliang Xu, and Hua Lu. 2019. Learned Index for Spatial Queries. In *20th IEEE International Conference on Mobile Data Management, MDM 2019, Hong Kong, SAR, China, June 10-13, 2019*. IEEE, 569–574. https://doi.org/10.1109/MDM.2019.00121