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 Z-orders currently used by modern systems treat all columns 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 automatically learns 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 traditional Z-order layouts by up to 1.6× in query runtime and up to 2× in rows scanned.

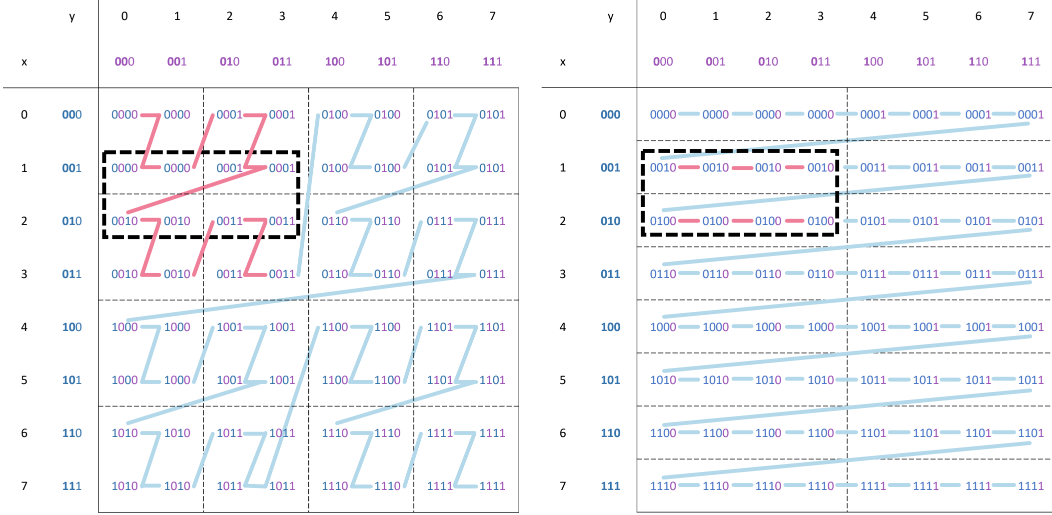
1 Introduction

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 [1, 2, 6]. 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 [8], 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.

*Work done prior to joining Amazon.

Machine Learning for Systems Workshop at 37th NeurIPS Conference, 2023, New Orleans, LA, USA.



(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 $(x \geq 1 \texttt{ AND } x \leq 2 \texttt{ AND } y \geq 0 \texttt{ AND } y \leq 3)$ (bold dotted-black rectangle), a Z-order configuration with an uneven bit allocation (b) results in fewer blocks (highlighted in red) scanned than one with an equal bit allocation (a).

However, the Z-order sort methods supported by existing systems like Amazon Redshift [2] and Databricks Delta Lake [5] give equal weight to the columns, 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 might not result in the best performance since different columns impact query performance differently.

Therefore, 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: the filter over column x is more selective than the filter over column y, so sorting by column x is more impactful than sorting by column y, and the Z-order bit allocation should reflect their relative importance.

We propose an approach to automatically learn the best unequal-bit Z-order configuration for a given dataset and query workload.

2 Learning Z-Order Bit Allocations

2.1 Problem Statement

We begin with some definitions:

Definition 1 (Z-Order configuration) *We define a Z-order configuration to be an allocation of bits to columns. Given 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 2 (Z-value under a configuration) *Given a Z-order configuration {col_0: v_0, col_1: v_1, ..., col_{n-1}: v_{n-1}}, the Z-value under this configuration for a given record $(c_0, c_1, \ldots, c_{n-1})$ 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).

Typically, systems place an upper limit on the total number of bits in a Z-value. A common limit is 64 bits, so Z-values can be represented as 8-byte integers, which can be sorted efficiently. For the remainder of this abstract, we assume 64-bit Z-values, though our techniques generalize to other bit limits.

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 the Z-order configuration (i.e., the *allocation* of 64 bits to n columns) that minimizes the total runtime of all queries in the workload.

2.2 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. 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 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.

Cost Model. Our cost model takes a candidate configuration as input and produces an estimated runtime of a query proportional to the amount of scanned data. The cost model that we use for a single query is:

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

The number of filtered columns is clear from the query itself. Instead of computing the exact number of rows scanned, which would require sorting the full dataset by the candidate configuration, we estimate the number of rows scanned by only sorting and creating zone maps over a sample of the dataset.

Search Algorithm. To search the space of possible configurations, we use Bayesian optimization [3], which is a derivative-free global optimization method for black-box objective functions. We chose Bayesian optimization because it is well-suited to expensive-to-evaluate objective functions as well as objective functions with stochastic noise and uncertainty, both of which are true for our cost model.

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-dimensional search space, where each dimension has a continuous domain between 0 and 1, which represents the relative number of bits allocated to each corresponding column.

Given a sampled point $(p_0, p_1, \ldots, p_{n-1})$ from the search space, we construct a candidate mapping that represents a 64-bit Z-order configuration, in which the number of bits allocated to col_i is $\left(64 \cdot \frac{p_i}{\sum_{j=0}^{n-1} p_j}\right)$. The candidate mapping is then inputted to the objective function.

3 Evaluation and Next Steps

We evaluate the performance of our learned Z-order bit allocation approach against three alternative sorting methods: (1) *Default sort order*: queries are performed on the original dataset, without any explicit sort order. (2) *Range partitioning*: the dataset is sorted on the column with the lowest average filter selectivity. (3) *Equal-weight Z-order*: we distribute 64 bits, allocating an equal number of bits to the three most frequently occurring columns in the query workload, resembling what is possible in today's systems [2, 5].

We evaluate indexes on four real-world datasets collected from [4]: contributions, flights, taxi, and tweets. The query workload for each dataset consists of around 500 queries, each with a filter

Table 1: Evaluation results on four datasets. we compare *Default* sort order, *Range* partitioning, *Equal*-weight Z-order over three columns, and *Ours* (Z-order with learned bit allocation), in terms of average rows scanned per query and end-to-end workload execution time.

	Data Traits		Rows Scanned				Workload Execution Time (s)			
	Rows	Cols	Default	Range	Equal	Ours	Default	Range	Equal	Ours
Contrib.	86M	9	77.3M	7.41M	8.32M	5.97M	654	99.4	95.3	52.9
Flights	120M	21	8.45M	10.4M	7.88M	4.02M	71.9	77.3	63.7	46.2
Taxi	175M	13	77.2M	57.9M	11.8M	6.45M	813	524	171	108
Tweets	15M	16	10.7M	364K	1.67M	349K	78	7.1	10	6.6

selectivity of 1% or less. Query execution involves scanning and filtering, projection over the columns that appear in the query, and materialization of the output tuples.

For all methods, the data is stored in Parquet format [9] and partitioned into 4MB rowgroups (i.e., blocks). Parquet natively uses zone maps to skip blocks. For Bayesian optimization, we use the Python package [7]. For each dataset and workload, we run Bayesian optimization for 600 iterations to find the best Z-order configuration. All the experiments are single-threaded and run on an Arch Linux machine with an Intel Xeon 2.1GHz CPU and 125GB RAM.

Results. Table 1 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$\times$ and 2$\times$ reduction in the number of rows scanned compared to the next-best layout.

Table 1 also shows the overall workload execution time for each sort order on each dataset. The trends for query runtime are similar to those for the average number of rows scanned, with our learned Z-order configurations achieving up to 1.6$\times$ faster runtimes. Improvements in query time are not as dramatic as improvements in rows scanned because query execution involves performing other operations after scanning the rowgroups, such as materializing the output tuples.

Discussion. 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 data skipping.

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 displays similar behavior.

In contrast, the Contributions workload consists 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 contributed 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 cannot allocate an adequate number of bits to all ten columns. 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.

Next Steps. Z-order data layouts are a rich area for future work. One open question is how to select the order in which the bits from different columns are interleaved to form the Z-value; ordering matters because bits in more significant positions have a larger impact on sort order than less significant bits. Another open question is how to handle dynamic datasets, where changes in the data distribution may cause the optimal bit allocation to also change. In order to maintain high performance, we would either need to switch to a new Z-order configuration, anticipate the data distribution evolution during the initial Z-order optimization, or a combination of both.

References

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

[2] 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.

[3] Peter I. Frazier. A Tutorial on Bayesian Optimization, 2018. URL https://arxiv.org/abs/1807.02811.

[4] HEAVY.AI. Omnisci. https://www.omnisci.com/, 2023.

[5] 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.

[6] Guido Moerkotte. Small materialized aggregates: A light weight index structure for data warehousing. In *Proceedings of the 24rd International Conference on Very Large Data Bases*, VLDB '98, page 476–487, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc. ISBN 1558605665.

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

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

[9] The Apache Software Foundation. Apache parquet. https://parquet.apache.org/, 2023.