# Zone Map Layout Optimization

The Harvard community has made this article openly available. **Please share** how this access benefits you. Your story matters.

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Citable link</td>
<td><a href="https://nrs.harvard.edu/URN-3:HUL.INSTREPOS:37364697">https://nrs.harvard.edu/URN-3:HUL.INSTREPOS:37364697</a></td>
</tr>
<tr>
<td>Terms of Use</td>
<td>This article was downloaded from Harvard University’s DASH repository, and is made available under the terms and conditions applicable to Other Posted Material, as set forth at <a href="http://nrs.harvard.edu/urn-3:HUL.InstRepos:dash.current.terms-of-use#LAA">http://nrs.harvard.edu/urn-3:HUL.InstRepos:dash.current.terms-of-use#LAA</a></td>
</tr>
</tbody>
</table>
Zone Map Layout Optimization

by

Nenya Edjah
Advisor: Stratos Idreos

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE HONORS REQUIREMENTS FOR THE DEGREE OF

Bachelor of Arts
Computer Science

Harvard University
Cambridge, Massachusetts
May, 2020
Abstract

A zone map is a lightweight data structure that stores multiple summary statistic tuples (e.g., min/max) for multiple contiguous regions of an array called “zones”. The most common application of a zone map, and the one that we consider for most of this work, is as an index to accelerate range queries in relational databases by determining which zones should be skipped during a scan. Most prior academic work in zone maps has only discussed uniformly sized zones that were drawn without much consideration of the underlying data. However, a lot of real-world data exhibits a high degree of clustering and partial-orderedness. Uniform-sized zones are incapable of recognizing this structure which limits their ability to effectively skip data. To enable the maximum amount of data skipping, non-uniformly sized zones are required.

In this work, we propose a novel technique for statically generating optimal non-uniformly sized zone map layouts via the minimization of a cost function that models the expected amount of data that needs to be scanned for a random query. We show that the layouts generated by this technique have better expected performance than any uniformly-sized zone map, and depending on the distribution of the data, can obtain up to 6.4x lower query latencies than the best uniformly-sized zone map.

We additionally implement this system in a mature, open source, column-oriented database called MonetDB, and compare its performance to that of MonetDB’s native lightweight indexing data structures: imprints. We show that in comparison to MonetDB’s imprints, optimized zone maps can be built just as quickly, take up almost 100x less space, and, depending on the column’s data distribution, offer up to 10x lower latencies for medium to high selectivity queries.
Listing of figures

1 Uniform vs Non-Uniform Zones ........................................... 5
1.1 Visualizations of the data distributions used for testing .......... 18
1.2 Zone map construction times using different algorithms .......... 21
1.3 Zone map query times using different algorithms ................. 22
1.4 Zone map construction times using aggregate sorting ........... 24
1.5 Zone map query times using aggregate sorting ................. 25
1.6 Zone map query times as selectivity changes ..................... 27
1.7 Zone maps optimized for different selectivities .................. 30

2.1 MonetDB index build times ........................................... 34
2.2 MonetDB index space overheads ..................................... 35
2.3 MonetDB query latency speedup with indices ...................... 35
2.4 MonetDB impact of selectivity on query latency .................. 38
List of Tables

1.1 Data distribution descriptions ............................... 18
1.2 Average point query speedup over not using a zone map ........ 20
Acknowledgments

This thesis would not have been possible without the mentorship of Wilson Qin and Prof. Stratos Idreos. I am indebted to them for introducing me to a fascinating research topic and for the advice and encouragement they provided me along the way.
Empty your mind, be formless, shapeless – like water.

Bruce Lee

In the past several decades, much research in computer science has been conducted on methods for efficiently storing, querying, and processing data. The data that is usually considered is structured and comprised of tables that have multiple rows and columns. Traditional approaches have largely been centered on relational databases that store this
data on disk in a row-oriented format [5]. That is, each row is represented by an ordered tuple of the fields corresponding to each column, and that entire tuple is stored in a block of contiguous disk space. Some examples of popular databases which employ this strategy are PostgreSQL [2] and MySQL [1].

The most fundamental type of query on a relational database is the selection of one or more fields from all rows which match some criteria. The most common type of selection, and the one we will focus on in this work is the range query: select all tuples where the value of some field is within some range. For example, the following query would retrieve the names of all college-age students from a “users” table.

```
SELECT "name" FROM users WHERE age >= 18 AND age <= 22
```

Queries such as the one above are very common in industry applications. So, they need to be fast.

### 0.1 Data Layout and Indices

The most simple strategy for executing queries such as the one above is to scan. That is, iterate over every single row in the database, and check if the row’s values match the query predicate. Since the unit of I/O for disks is a large page (usually sized at 4096 bytes), a scan in a traditional row-oriented database would require reading every single page in the table. This is the case even if the table has 100 columns and the select predicate only needs access to a single one.

A more efficient scheme is to physically organize the data in fixed-width columns [6] rather than in rows. In such a column store, a query which selects based on a single column
only needs to scan that the pages associated with that column. In the case of a 100-column table, this corresponds to an up to 100x reduction in the amount of I/O and a considerably faster scan. Due to these and other benefits, column stores such as MonetDB [7], Vertica [11], Snowflake [8], and Amazon’s Redshift [3] have largely replaced row stores as the relational databases of choice for big data analytic workloads [6].

Whenever the selectivity of a query is high (i.e. the number of tuples selected from the table is low), scans, even in the case of column stores, are often a suboptimal strategy for executing range queries. In these cases, it is often better to take advantage of a secondary index. A secondary index is a supplementary data structure that is built on top of a column to accelerate select queries by either storing the locations of the relevant tuples or by allowing a scan to skip data which is not relevant to the query. A secondary index is in contrast to a primary index which does a similar thing, but alters the physical organization of the entire table. Multiple secondary indices can be built for different columns, but only a single primary index can be used unless the application wants to maintain another copy of the entire table [16]. So, we focus our attention on secondary indexing.

Over the past few decades, dozens of different kinds of data structures have been invented for database indexing. The most popular is the B+ Tree [9] which gives logarithm access time to any value in the data. However, B+ Trees tend to consume a large amount of space relative to the amount of data that they index, and they tend to scale poorly as query selectivity decreases.

Lighter weight secondary indices exist which trade off efficiency during very high-selectivity queries with lower memory/space usage and better scaling as query selectivity decreases. Indices in this family typically operate by storing summary statistics that indicate which pages
should be skipped during the execution of a scan. Some examples of such data structures are bitmaps [17], column sketches [10], column imprints [16], and the one we further analyze in this work: zone maps [13].

0.2 Zone Maps

Within the context of zone maps, a zone is simply a contiguous region of a column. A zone map stores the summary statistics for each zone, usually the minimum and maximum values, as an array of tuples. Since each zone can be large (potentially spanning tens of thousands of bytes of data), and each tuple is small (no more than a few dozen bytes), a zone map can be an extremely lightweight data structure.

To use a zone map to accelerate a scan, the zone map can be first be consulted to determine which zones overlap with the query’s predicate. The zones that do overlap will be linearly scanned for further processing, and the zones that do not can be skipped entirely.

On data which looks essentially random, zone maps don’t provide any benefit, but the overhead that they incur is usually small if there are not a significant number of zones [13]. However, on data which exhibits a high degree of clustering or semi-sortedness, zone maps can significantly accelerate queries.

Traditionally, zone maps have used uniform-sized zones, but this isn’t always the best choice. For example, in Figure 1, it’s clear that the variable width zone map would be a significantly better choice for assisting arbitrary queries.
This observation is not new. Some prior work has investigated the adaptive dividing and merging of zones \cite{15} over the course of the runtime of a sequence of queries. This process would tune the zone map to the access patterns of the particular workload which generated those queries. In this work however, we will strive to statically rather than dynamically generate zone map layouts which yield good performance for general workloads.

0.3 This Thesis

In this thesis, we will propose a technique to statically generate the mathematically “optimal” zone map layout for a particular column. In section 1.1, we define optimality in terms of the minimization of a cost function that represents the expected latency of a random query. It turns out that to generate such an “optimal” layout can take a prohibitively long time for large columns, so in sections 1.3 and 1.4, we also provide a number of heuristic algorithms based on the same cost minimization principle, but run far faster.

When our algorithms are applied to data that is semi-sorted or clustered, the zone bound-
aries that our algorithm draws are well tuned to the data, in a way similar to the variable-sized zones in Figure 1. This leads to a considerable amount of data skipping and consequently, a substantial drop in query latency. Based on a series of experiments in a standalone prototype column store, we will show in section 1.6 that the zone maps generated by our algorithms on such data have better performance than any uniformly-sized zone map for random query workloads.

In Chapter 2, we implement our system in a mature, open source, column-oriented database, MonetDB, and compare the performance of (optimized) zone maps to MonetDB’s built-in lightweight index: column imprints.
We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%.

Donald Knuth

1

Static Zone Map Layout Optimization

Given that different zone map layouts can yield radically different runtime performance characteristics, we would like to be able to statically generate a zone map layout – using non-uniform zone sizes – that optimizes performance for a general workload. To maximize the zone map’s ability to generalize, we employ randomness and consider the ex-
pected latency of a zone map assisted scan for a uniformly random point query. We choose point queries largely because they simplify the modeling considerations, and we later show in section 1.7 that the choice of point queries as opposed to a lower selectivity query does not significantly affect the results of our algorithm.

To statically optimize zone map layout for a given column, we must first define a function for the cost of a layout based on the point query model and then find a layout which minimizes this cost function.

1.1 The Cost Function

There are two types of costs that must be considered when performing a database query. The first is the cost of CPU computation, and the second is the cost of I/O. Over the past few decades, it has become the case that CPUs are often able to operate on data far faster than hard drives, solid state drives, or even DRAM are able to deliver it to the CPU. Consequently, I/O has become the primary bottleneck in database queries [6]. Because of this development, we would like our cost function to model the expected total I/O cost of a zone map scan for a random query.

Another relevant development in the past few decades has been the commodification of huge amounts of DRAM such that it is not uncommon to find servers with over 1 TB of memory [4]. This availability has led to the rise of main-memory optimized column stores – databases which essentially assume that all of their data is able fit into memory [6]. Instead of worrying about minimizing the number of disk I/Os that they must perform, these databases are concerned about minimizing the number of CPU cache misses.

In this work, we assume that we are operating within the context of a main-memory
optimized column store. So our goal is to model the number of cache lines that must be accessed to complete a scan. In this model, there are two types of operations we need to consider: scans of the zone map data structure and scans of column data.

Before we can begin modeling, we first need to define the core data structures which comprise a zone map. To support variable-sized zones, we can make the zone map data structure an array of zone structs. Each zone struct $z_i$ stores a minimum $z_i^{\min}$, a maximum $z_i^{\max}$, and a 64-bit integer $z_i^{size}$ which represents the number of elements covered by the zone. Consequently, if we assume the standard struct alignment rules in C, a column with 1-4 byte elements will have a zone struct of size 16, and a column with 8 byte elements will have a zone struct of size 24. We will represent the size of the zone struct for a particular data type $T$ as $\text{sizeof}(\text{Zone}<T>)$.

The cost to navigate the zone map data structure alone is the number of cache lines required to store the zone map. If we assume a cache line size of $s_{cache}$ and there are $N$ zones, then the cost of the navigation is

$$\frac{\text{sizeof}(\text{Zone}<T>) \cdot N}{s_{cache}}.$$ 

Similarly, the number of cache lines that need to be accessed to scan a column segment of data type $T$ that is represented by a zone $z_i$ is:

$$\frac{\text{sizeof}(T) \cdot z_i^{size}}{s_{cache}}.$$ 

Now suppose that we have a column $D$ of data type $T$, and we are considering one particular zone $z_i$. In a random point query, i.e. a select query (with inclusive lower and upper
bounds) which has a range of 1 element, the probability $p_i$ that the part of a column represented by the zone will be scanned is determined by $z_i^{\text{min}}$ and $z_i^{\text{max}}$ and can be computed as

$$p_i = \frac{\text{ubound}(z_i^{\text{max}}) - \text{lbound}(z_i^{\text{min}})}{|D|}$$

where \text{ubound}(x) counts the number of elements of $D$ which are less than or equal to $x$ and \text{lbound}(x) counts the number of elements of $D$ which are strictly less than $x$. Note that to efficiently compute \text{ubound} and \text{lbound}, it is necessary to sort $D$ which incurs an $O(|D| \log |D|)$ overhead to any zone map optimization algorithm.

Then, the expected cost of including the zone $z_i$ in the zone map would be

$$C_D(z_i) = \frac{\text{sizeof(Zone<T>) + sizeof(T)} \cdot p_i \cdot z_i^{\text{size}}}{s_{\text{cache}}}$$

The total cost of a zone map layout $Z = (z_1, \ldots, z_N)$ on $D$ is then

$$C_D(Z) = \sum_{i=1}^{N} C(z_i)$$

We would like to minimize this cost.

### 1.2 Dynamic Programming

To minimize this cost function, we can employ a dynamic programming approach. But first, we compute a uniform-sized zone map $Z^{\text{base}}$ on $D$ using some configurable base size $m$. This base size is the granularity of our final zone map. The highest possible granularity is $m = 1$ and would allow for zones of size 1. However, since data is only read in discrete
cache lines, the smallest reasonable choice is \( m = \frac{s_{\text{cache}}}{\text{sizeof}(T)} \). Another reasonable choice if the data is stored on disk might be the size of an I/O page divided by \( \text{sizeof}(T) \).

We can minimize \( C_D(Z) \) by identifying specific groups of adjacent zones in \( Z_{\text{base}} \) and merging those groups using a \( \text{coalesce}(z_1, \ldots, z_k) \) function that combines multiple zones into one by taking the global minimum and maximum from the zones that it is given and summing their sizes.

Let \( n_{\text{base}} = \lceil \frac{|D|}{m} \rceil \) be the number of zones in the base zone map. We define the following recursion for the minimum cost that can be obtained by building a zone map on the segment of \( D \) that starts with base zone \( Z_{\text{base}}^i \):

\[
\text{DP}_D(i) = \begin{cases} 
0 & \text{if } i \geq n_{\text{base}} \\
\min_{k \in \{1, 2, \ldots, n_{\text{base}} - i\}} \left\{ C_D(\text{coalesce}(Z_{\text{base}}^i, \ldots, Z_{\text{base}}^{i+k-1})) + \text{DP}_D(i + k) \right\} & \text{otherwise}
\end{cases}
\]

The overall minimum cost is obtained by computing \( \text{DP}_D(0) \) with memoization. During the computation process, the zone group size \( k \) which minimizes the cost for each input is stored in a table. The full sequence of optimal zone group sizes can subsequently be found by tracing through that table.

Note that even though each \( \text{coalesce}(.) \) evaluation can take in \( O(n_{\text{base}}) \) arguments, every successive call can be computed in \( O(1) \) time by re-using the results from the previous call. This means that each iteration of the DP takes \( O(n_{\text{base}}) \) time since it has to select the minimum over \( O(n_{\text{base}}) \) elements. Since there are \( O(n_{\text{base}}) \) possible inputs to the DP, the dynamic programming ultimately requires \( O(n_{\text{base}}) \) memory and \( O(n_{\text{base}}^2) \) time to run.
Unfortunately, a quadratic time complexity is quite high if the base size $m$ is small or the column $D$ is long, so if we want our technique to be scalable, we need to develop some algorithms which run faster but only give approximately optimal results.

1.3 Constrained Dynamic Programming

One initial idea is that it seems unnecessary to allow the dynamic programming algorithm described above to be able to use any zone group size. In practice, it should still be possible to draw good zone map boundaries even if we only allow the algorithm to coalesce in groups whose sizes are drawn from some set $A \subset \{1, 2, \ldots, n_{\text{base}}\}$. One possible choice for $A$ might be powers of 2, i.e,

$$A_{\text{pow2}} = \{1, 2, 4, 8, 16, \ldots, 2^{\left\lfloor \log_2 n_{\text{base}} \right\rfloor}\}$$

With this modification, we can re-define the recursion as:

$$DP_D(i) = \begin{cases} 
0 & \text{if } i \geq n_{\text{base}} \\
\min_{k \in A} \left\{ C_D(\text{coalesce}(Z^{{\text{base}}}_i, \ldots, Z^{{\text{base}}}_{i+k-1})) + DP_D(i+k) \right\} & \text{otherwise}
\end{cases}$$

The time and space complexity of this variant is a little more difficult to analyze. If the largest element in $A$ is $n_{\text{base}}$, then for each input of the DP, we still have to perform the work of coalescing $O(n_{\text{base}})$ zones. Consequently, the time complexity for computing $DP_D(0)$ with memoization would still be $O(n_{\text{base}}^2)$ which isn’t an improvement over the original algorithm.
To get an improvement, we need to speed up the computation of coalesced zones. At a high level, this can be done by taking advantage of the execution order of the recursion. The DP computation starts at $\text{DP}_D(n_{\text{base}} - 1)$ and progresses downwards one step at a time. So, it is possible to maintain “running” zones that represent the coalesced zones for each zone group size $k \in A$ as the computation progresses down to $\text{DP}_D(0)$. We can track which base zones are supposed to be in the running coalesced zone by maintaining a queue of maximum length $k$ for each $k \in A$. We can incrementally update the minimum/maximum of the coalesced zone as the base zones get added to and removed from the queue by either performing a linear scan through the queue or by using a min/max heap. See Appendix A for the full description and pseudocode of this algorithm.

If a linear scan is used, then the total time complexity of computing a coalesced zone for all $n_{\text{base}}$ positions for a single zone size $k$ would be $O(n_{\text{base}} \cdot k)$, and if min/max heaps are used, then the time complexity would be $O(n_{\text{base}} \cdot \log k)$. The space complexity would be $O(k)$ because the queue and heaps would maintain a maximum of $k$ elements.

The total time complexity of the DP algorithm if heaps are used would then be

$$O \left( n_{\text{base}} \sum_{k \in A} \log k \right)$$

and the total space complexity would be

$$O \left( n_{\text{base}} + \sum_{k \in A} k \right)$$

If a set like $A_{\text{pow}2}$ is used, then this gives $O \left( n_{\text{base}} \log^2(n_{\text{base}}) \right)$ time complexity and $O(n_{\text{base}})$ space complexity.
1.4 Other Approximate Algorithms

The approaches described above are relatively complex, and can be quite slow for large datasets. So, it would be helpful to establish some simpler approximate algorithms for solving the optimization problem to determine if the more complex approaches are useful. The pseudocode for all of these algorithms can be found in Appendix A.

1.4.1 Greedy Algorithm

There is a simple greedy approach. The idea is to initialize a special zone \( z_a := Z^\text{base}_0 \) to function as an aggregate. Then, we linearly iterate through all of the zones in \( Z^\text{base} \). If it would be better to merge zone \( Z^\text{base}_i \) into \( z_a \), i.e.

\[
C_D(\text{coalesce}(z_a, Z^\text{base}_i)) < C_D(z_a) + C_D(Z^\text{base}_i)
\]

, then do so. Otherwise, add the current \( z_a \) to a list of final zones, re-initialize \( z_a := Z^\text{base}_i \), and repeat the process starting at position \( i \) until all of \( Z^\text{base} \) has been processed. It’s easy to see that this algorithm requires \( O(n^\text{base}) \) time and \( O(n^\text{base}) \) space to store the final result, so it is as efficient of a build process as we can hope to obtain.

The greedy merging process described above can also be repeated on the output list of zones until a fixed point is reached.

1.4.2 Pairs Algorithm

Another algorithm might be to look at every pair of adjacent zones in \( Z^\text{base} \). That is, we look at \( \{(Z^\text{base}_0, Z^\text{base}_1), (Z^\text{base}_2, Z^\text{base}_3), \ldots, (Z^\text{base}_{n^\text{base}-2}, Z^\text{base}_{n^\text{base}-1})\} \). For each pair, we make
the decision of whether to merge them or to keep them separate, and we append the results of the each decision to an output list $L$ of zones. So, the length of $L$ will be between $n_{\text{base}}/2$ and $n_{\text{base}}$. If $L$ is identical to $Z_{\text{base}}$, then we terminate the algorithm and return $L$. Otherwise, we recursively perform the process on $L$.

It’s easy to see that the time complexity of each pass of the algorithm is linear, and in practice, the number of passes tends to be small, i.e. fewer than 15 on columns as large as 100m elements.

1.4.3 Bisection Algorithm

Yet another algorithm for this problem is based on bisection. The algorithm works by trying to split $Z_{0}^{\text{base}}, \ldots, Z_{n_{\text{base}}-1}^{\text{base}}$ into two large zones. That is, it determines the optimal split point $k \in \{0, \ldots, n_{\text{base}}-1\}$ such that

$$C_{D}(\text{coalesce}(Z_{0}^{\text{base}}, \ldots, Z_{k-1}^{\text{base}})) + C_{D}(\text{coalesce}(Z_{k}^{\text{base}}, \ldots, Z_{n_{\text{base}}-1}^{\text{base}}))$$

is minimized. If the split point is $k = 0$, then there is nothing that can be gained by splitting, so the algorithm would return that there should only be a single zone: $\text{coalesce}(Z_{0}^{\text{base}}, \ldots, Z_{n_{\text{base}}-1}^{\text{base}})$. Otherwise, the algorithm is recursively applied to the left and the right halves of the split and the two results are concatenated.

In practice, for most data distributions, the algorithm has a reasonable runtime and a small space requirement, but it is possible to construct some degenerate columns in which the performance degrades to a worst case of $O(n_{\text{base}}^{2})$. 

15
1.5 Other Optimizations

1.5.1 Partitioning

One way to accelerate the runtime of these algorithms at the cost of some correctness is to divide the column into partitions and blocks.

We define a partition to be a large chunk of the column which we can treat as independent of other chunks. This means that rather than using the same `lbound()` and `ubound()` functions for the entire column which would require an $O(|D| \log |D|)$ sorting cost, we could have separate `lbound()` and `ubound()` functions for separate partitions. If $P$ is the size of a partition, then this can reduce the total time spent sorting to $O(|D| \log P)$.

Additionally, we can further subdivide partitions into blocks. A block is a chunk of a partition that the optimization algorithm can work on independently from other blocks. If $B$ is the block size, then a partition of size $P$ would have $\lceil P/B \rceil$ blocks. This could considerably improve the runtimes of some algorithms. As an example, for the unconstrained dynamic programming, this could reduce the time complexity from $O\left(\frac{P^2}{m^2}\right)$ per partition to $O\left(\frac{PB}{m^2}\right)$. The full column time complexity would then be $O\left(\frac{|D|B}{m^2}\right)$.

While partitioning and blocking can considerably improve the time it takes to build optimized zone maps, the tradeoff is that it forces the algorithm to ignore global information and to only make locally optimal decisions. However, if the partition and block sizes are high enough, this doesn’t seem to significantly affect the quality of the zone maps.
1.5.2 Aggregate Sorting

For the simple linear time heuristic algorithms, the \( O(|D| \log P) \) time sort will be the biggest bottleneck to building the zone map. We can slightly improve this runtime by noting that the \texttt{lbound} and \texttt{ubound} functions are only needed to rank the min/max values within \( Z^{\text{base}} \). So, rather than sorting all \( P \) values within a partition, we can simply sort the \( \left\lceil \frac{2P}{m} \right\rceil \) values corresponding the mins and maxes of \( Z^{\text{base}} \) within that partition. If the data is sufficiently clustered, and \( m \) isn’t too large, then this gets us a good approximation of these values’ true ranks, and it brings down the total sorting cost to \( O\left( \frac{|D|}{m} \log \left( \frac{P}{m} \right) \right) \).

1.6 Prototype Evaluation

1.6.1 Experimental Setup

To evaluate the quality of our optimized zone maps, we implemented them within a single-threaded, standalone, column store prototype in C++. We then ran several experiments on a machine with a 2.2 GHz Intel i7 CPU with 6 physical and 12 logical cores, 32 GB of DDR4 RAM, and a 512 GB SSD.

As we’ve noted earlier, the algorithms described above build better zone maps by exploiting the structure within a column to draw zone boundaries that group similar elements together. So, the potential benefit of a zone map is largely dependent on a column’s data distribution. We tested our system on four different columns with different data distributions as listed in Table 1.1 and visualized in Figure 1.1. See Appendix A for the exact algorithms which generated the data.
We fixed our zone size granularity \( m \) to 8 since we have 64-bit elements and 64-byte cache lines, and we experimented with different strategies for partitioning, blocking, and
aggregate sorting as described in section 1.5. Because we used 100m element columns, neither the column nor the zone maps were able to fit inside of the CPU cache, so the timings that we collect for each trial are representative of the the worst case performance in a main-memory optimized column store.

1.6.2 Comparing the Optimization Algorithms

In this section, we compare the performances of the different algorithms. We measure the time it takes to generate zone maps for each algorithm and the average latency of 100 random point queries using these zone maps. We also measure the number of cache lines scanned for these queries. Each algorithm uses partitioning and blocking, so we vary the partition size and block size to investigate their effect. In this section, we do not use aggregate sorting, and we defer its analysis to section 1.6.3.

We also compare our optimized zone maps to the best performing uniform-sized zone map (which varied for different columns), as well to no zone map.

In general, the larger partitions/block sizes had the best results. The point query speedups over not using a zone map for the zone maps generated using a partition size of 100m and a block size of 8192 are summarized in Table 1.2.
<table>
<thead>
<tr>
<th></th>
<th>sorted</th>
<th>random</th>
<th>correlated</th>
<th>rand_clusters</th>
</tr>
</thead>
<tbody>
<tr>
<td>best uniform zone map</td>
<td>uniform_8192</td>
<td>uniform_8192</td>
<td>uniform_256</td>
<td>uniform_128</td>
</tr>
<tr>
<td></td>
<td>4760.76x</td>
<td>1.00x</td>
<td>23.12x</td>
<td>26.83x</td>
</tr>
<tr>
<td>quadratic_dp</td>
<td>4760.76x</td>
<td>0.97x</td>
<td>25.89x</td>
<td>172.42x</td>
</tr>
<tr>
<td>constrained_dp</td>
<td>4701.25x</td>
<td>0.97x</td>
<td>24.30x</td>
<td>130.75x</td>
</tr>
<tr>
<td>multipass_greedy</td>
<td>4760.76x</td>
<td>0.97x</td>
<td>18.14x</td>
<td>13.91x</td>
</tr>
<tr>
<td>multipass_pairs</td>
<td>4760.76x</td>
<td>0.96x</td>
<td>24.12x</td>
<td>144.81x</td>
</tr>
<tr>
<td>bisection</td>
<td>4760.76x</td>
<td>1.00x</td>
<td>25.36x</td>
<td>167.81x</td>
</tr>
</tbody>
</table>

Table 1.2: Average point query latency speedup over not using a zone map. This table’s results are for a partition size of 100m, a block size of 8192, and no aggregate sorting.

We visualize the time it takes to build an optimized zone map in Figure 1.2, and we visualize the average query latency for random point queries and the number of cache lines accessed by these queries in Figure 1.3.
Figure 1.2: Measurements of the time to construct zone maps on different columns with 100m elements. Included in these are measurements of the preprocessing work required to compute the $\text{lbound()}$ and $\text{ubound()}$ functions.
We observe that our optimized zone map layouts take a non-trivial amount of time to build, especially as we increase the partition and block sizes. However, they also have a non-trivial edge in query latency. In particular, the zone map layout produced by the unconstrained dynamic programming algorithm (quadratic_dp in the charts) with $P = 10^8$ and $B = 8192$ performed considerably better than the best uniform-sized zone map for columns which had a lot of structure (1.12x better for the correlated column and 6.42x better for the rand_clusters column). For these columns, even though the zone map took almost 60 seconds to build, over the course of thousands of queries, the initial build time
quickly becomes amortized by the considerably faster latency.

For the sorted column, the optimized zone map layouts matched the performance of the best uniform layout, and for the random column, the performance of the optimized layout was not much worse (0.97x). This suggests that the optimized zone map layouts are worthwhile for columns which have a reasonable amount of clustering.

Additionally, for all but the multipass_greedy algorithms, we see an increase in performance as the partition and block sizes increase which is to be expected. As expected, there is a tradeoff between the quality of the zone maps and the amount of time spent building them. If the build time is a concern, several of the heuristic algorithms such as the bisection technique obtain performance comparable to that of the quadratic_dp at a fraction of the build time.

Since we are not using aggregate sorting in this set of experiments, the preprocessing is a considerable amount of this build time, and it is the bottleneck in all algorithms except for the quadratic_dp. The preprocessing is important for making the system agnostic to data type and for producing accurate estimates of the probability of a scan. In this work, we tolerate expensive build processes since we are ultimately more concerned about the resulting query latency. But in future work, further optimization of the build process would be an interesting area to explore.

Finally, note that there is a strong correlation between the query time and the number of cache lines accessed. This suggests the cost function we defined accounts for a large percentage of runtime performance, but the relationship is not perfectly linear. The remaining runtime is likely accounted for by factors such as the time it takes the CPU to execute instructions. Branch mispredictions may initially seem to be another factor, but in this ex-
periment, the selectivity is so high that the CPU does in fact manage to correctly predict most branches.

1.6.3 Impact of Aggregate Sorting

In subsection 1.5.2, we discussed aggregate sorting as a potential way to speed up the pre-processing times for zone maps, but we also mentioned that it leads to approximate rankings whose quality are largely dependent on the distribution of that data.

To evaluate this effect, we repeated the experiments from the previous section with the aggregate sorting optimization enabled. The results are located in Figures 1.4 and 1.5.

![Figure 1.4: Zone map construction times using aggregate sorting.](image)
Figure 1.5: Zone map query times using aggregate sorting.

Compared to the results in Figure 1.2, the results of Figure 1.4 show that zone map construction times are considerably lower when aggregate sorting is enabled. With aggregate sorting, the amount of time spent on the preprocessing step is on average around 6x lower. For the simple algorithms which spent most of their time on preprocessing, this also corresponds to an approximately 6x improvement in the total zone map construction time. For the \texttt{quadratic_dp} algorithm however, this corresponds to a smaller, but still appreciable, 1.5x improvement in the $P = 10^8$ case.

If we look at Figure 1.5 and compare it to Figure 1.3, we see that the query times with aggregate sorting are comparable to the query times without. The only exception is with the
random column on which the aggregate sorting yields 7-10% higher query latencies. This makes sense since the aggregate sorting will only produce good approximations of the true lbound() and ubound() functions if the data is sufficiently clustered or partially ordered. Since optimized zone map performance for random data is poor anyway, this doesn’t seem like a barrier to using the optimization given that it can produce optimized zone maps considerably faster without harming performance on clustered data.

1.6.4 Impact of Query Selectivity

Since our zone maps have technically been optimized for point queries, one might expect that their performance might degrade to be worse than the non-optimized uniform zone maps as the selectivity increases. To test this out, we executed random range queries of various selectivities using different zone map layouts. The results of Figure 1.6 show that the optimized zone map continues to perform as well or better than all other uniform-sized maps until selectivities very close to 1.
Figure 1.6: Measurements of the average latency as a function of selectivity over 100 random range queries for a zone-map assisted scan of different data distributions with 100m elements. While building the quadratic_dp zone map, we used a partition size of 100m and a block size of 8192.

1.7 Selectivity-Specific Zone Map Optimization

In subsection 1.6.4, we observed that the zone map query performance degrades as selectivity decreases. This is to be expected since decreasing selectivity means that the lower bound for how much data must be scanned will increase. But, since we generated the optimized zone maps in subsection 1.6.4 using a point query model, one might wonder if it’s possible to get better performance for range queries of different selectivities if we optimize zone maps specifically for those selectivities. In order to do this, we first need to redefine the cost
function.

1.7.1 Updating the Cost Function

Remember that for point queries, the probability of a scan for a particular zone $z_i$ was

$$p_i = \frac{\text{ubound}(z^\text{max}_i) - \text{lbound}(z^\text{min}_i)}{|D|}.$$ 

For a random range query with selectivity $f$, computing the probability of a scan is a little more complicated. First, we must consider that in order for a range query to actually have a selectivity of $f$, then its possible starting points may only range from the 0-th percentile to the $(1 - f)$-th percentile element of $D$. Consequently, we assume that the starting points are uniformly distributed along that interval.

Now, we must consider the number of range queries that will actually overlap with the values in $z_i$. In terms of rankings, the smallest possible starting point for a query that overlaps is going to be

$$r^\text{min} = \max \left(0, \text{lbound}(z^\text{min}_i) - |D| \cdot f\right).$$

Similarly, the largest possible starting point is going to be

$$r^\text{max} = \min \left((1 - f) \cdot |D|, \text{ubound}(z^\text{max}_i)\right).$$

Since we also assume that the starting point of the the range queries are uniformly dis-
tributed from the 0-th to the \((1 - f)\)-th percentiles, the total probability of a scan is

\[
p_{i,f} = \frac{r_{max} - r_{min}}{(1 - f) \cdot |D|}
\]

\[
= \min \left( (1 - f) \cdot |D|, \ubound(z_i^{max}) \right) - \max \left( 0, \lbound(z_i^{min}) - |D| \cdot f \right)
\]

\[
= \min \left( (1 - f) \cdot |D|, \ubound(z_i^{max}) \right) - \max \left( 0, \lbound(z_i^{min}) - |D| \cdot f \right)
\]

Note that if we substitute a selectivity of \(f = 0\) into the formula above, we get the same probability that we had for point queries.

We can now substitute this new formula for the probability of a scan into the cost function from section 1.1. The expected cost of including an individual zone \(z_i\) for column \(D\) of data type \(T\) for a random range query of selectivity \(f\) would be:

\[
C_{D,f}(z_i) = \frac{\text{sizeof}(\text{Zone}<T>) + \text{sizeof}(T) \cdot p_{i,f} \cdot z_i^{size}}{s_{cache}}
\]

and the total cost of a zone map layout \(Z\) on column \(D\) with \(N\) zones \(z_1, \ldots, z_N\) is

\[
C_{D,f}(Z) = \sum_{i=1}^{N} C_{D,f}(z_i).
\]

We can use the same techniques discussed earlier in this chapter to minimize this cost function.

1.7.2 Results

In Figure 1.7 we present the results of an experiment in which we evaluate quadratic_dp zone maps which were optimized for different selectivities. The results show that for data
distributions with a lot of structure such as the correlated and rand_clusters columns, there is little to be gained by optimizing for different selectivities since there is no significant speedup over the zone map which was optimized for point queries.

For data which is effectively random, we seem to have a significant improvement if we use the non-point-query zone maps as selectivities decrease, but in reality, this is just because those zone maps have fewer zones and are simply getting closer to not using any zone map at all. Since it would have been better to not use a zone map in this case, it looks like there isn’t much of a benefit to optimizing for multiple different selectivities.

Figure 1.7: The relative speedup for zone maps which were optimized for different selectivities over the quadratic_dp zone map which was optimized was point queries. All optimized zone maps were generated using a partition size of 100m, a block size of 8192, and no aggregate sorting.
To get a better sense of how our strategy for zone map layout optimization fares against other secondary indexing strategies, we implemented the system described in Chapter 1 in MonetDB [7], a mature, open source, column-oriented database. MonetDB is a main-memory optimized database in that it uses memory-mapped files to access data. As
discussed in Chapter 1, limiting the number of cache lines accessed during a scan is an important goal for such a system, and MonetDB’s native data structures of choice for accomplishing this are column imprints [16].

2.1 Column Imprints

A column imprint is a simple, cache conscious secondary index that consists of a collection of many small bitvectors, each indexing the data points of a single cache line. Each bit in an imprint vector corresponds to a region of the column’s domain. If a bit is on, then the cache line contains a value from that region of domain, and if a bit is off there are no values from that region present within that cache line. So, in a sense, imprint vectors act as miniature histograms for individual cache lines. The process of performing a range query using imprints involves first constructing a query-specific bitvector based on the query’s predicate. This query-specific bitvector is then checked with every imprint vector using a bitwise AND operation, and if at least one match is found, then the entire cache line that corresponds to that imprint will be fetched and scanned. Column imprints are constructed using a linear time algorithm that first generates a histogram of the data. This histogram is subsequently used to divide the domain of the column into at most 64 evenly sized bins – each bin corresponding to bit in an imprint vector [16].

Like zone maps, imprints are most useful when the data exhibits a good degree of clustering. In such cases, each imprint vector will only contain a small number of bits which decreases the probability that a cache line will be retrieved for a random query. However, imprints do improve upon zone maps in that they are still useful for data distributions where there is little clustering. Consider an example where the only values in a cache line
are near to both the minimum and maximum of the domain. A zone map would always scan this cache line, but a corresponding imprint would simply have two set bits (one for the min and one for the max) which makes the probability of scanning the cache line low.

Imprints are relatively space efficient since they only store a single vector for each cache line where each vector ranges from 8-64 bits depending on the cardinality of the column’s domain. However, in the worst case of 64-bit vectors and 64-byte cache lines, imprints must incur a space overhead of at least 12.5% the size of the base data in order to index every cache line. In such cases, the maximum reduction in cache line reads than an imprint can provide is 8x, however CPU friendly compression schemes can bring this factor up without significantly slowing down the imprint checking process [16].

Zone maps, and in particular, the optimized zone maps we’ve proposed, do not suffer from this deficiency. As we will see in section 2.2, optimized zone maps can require as little as a 0.1% space overhead for well structured data such as the correlated and rand_clusters columns. Because of this, zone maps give a better lower bound on the number of cache lines which must be read to complete a scan. This can lead to dramatic differences in the performances of the two indices at medium to high selectivities.

### 2.2 MonetDB Experiments

We implemented uniform zone maps as well as the quadratic_dp and multipass_pairs algorithms in MonetDB. To speed up build times, the optimization algorithms use a partition size of 100m, a block size of 8192, and aggregate sorting. To evaluate them, we performed a similar set of experiments to the ones conducted in Section 1.6. Unlike our standalone prototype which was single-threaded, MonetDB uses multiple threads to parallelize
the execution of its queries. However, we found that a high number of threads ended up having a significant overhead for high selectivity queries, so we fixed the number of threads to 2. We used the same hardware for these experiments that we used in Section 1.6.

2.2.1 Build Time, Index Size, and Point Query Latency

In this section, we analyze the build times, index sizes, and query latencies (for high selectivity queries) for imprints, uniform zone maps, and optimized zone maps in MonetDB. We use the same four columns that were described in Table 1.1. The results are visualized in Figures 2.1, 2.2, and 2.3.

Figure 2.1: MonetDB index build times.
Figure 2.2: MonetDB index space overhead. The percentage is relative to the size of the column.

Figure 2.3: MonetDB query latency speedup over not using an index. These numbers are averaged over 100 random high selectivity queries.
In Figure 2.1, we can see that uniform zone maps can in general be built 20x more quickly than imprints indices. Optimized zone maps take more time, and the factor depends on the algorithm and the data distribution. In the case of the `multipass_pairs` algorithm, the construction time ranges from 0.85x to 3.5x that of imprints, and in the case of the `quadratic_dp` algorithm, construction time ranges from 5.7x to 8.1x that of the imprints.

The imprints index construction time is fairly consistent across the data distributions. This makes sense as the imprints construction algorithm has a fixed linear runtime [16]. For the optimized zone map algorithms however, construction time is partly dependent on the preprocessing time for the `lbound()` and `ubound()` functions which are likely to have more random memory accesses for less structured data. Additionally, in the case of the `multipass_pairs` algorithm, runtime is also dependent on how many pairs of zones can be coalesced together in each iteration of the algorithm. If that number is small – as it is likely to be in the case of a highly disordered column – then we can expect a slower algorithm.

In Figure 2.2, we see that the imprints indices are usually the most costly in terms of space overhead. They consistently use 18.8% additional space relative to the size of the column. The zone maps on the other hand are very lightweight. The smallest uniform zone size we tested with used no more than 0.6% extra space, and the largest ones used 0.02% extra space. For clustered columns, the optimized zone maps similarly achieved low space utilization, capping out at 1.6% extra space for the correlated column. For the random column however, it appears the optimized zone maps were unable to coalesce many of the original zones together, so they ended up producing an output zone map which looked very similar to the size-8 uniform zone map. Consequently, their space overhead was on the order of 30%.
In terms of the high-selectivity query latency, Figure 2.3 shows that the optimized zone maps once again come out on top for the clustered columns. These zone maps are consistently as fast as the best uniform zone maps for the sorted and correlated columns, and they are up to 5x faster than the best uniform zone maps and up to 100x faster than the imprints index for the rand_clusters column. An interesting result is that for the random column, none of the zone maps perform much better than a full, no-index scan. In fact, the optimized zone maps perform noticeably worse at 0.8x (a possible side effect of aggregate sorting). But, the imprints index is able to get a 20% improvement over the full scan. This is likely due to the ability of columns imprints to perform effective filtering even in the case of a very wide range of values within a cache line.

2.2.2 Impact of Query Selectivity

In this section, we replicate the experiment from section 1.6.4. For each column and each index, we ran 100 random queries for 20 different selectivities. The results in Figure 2.4 contain the results of these tests.
Figure 2.4 shows that despite zone maps having superior performance to imprints for high selectivities, as the selectivity decreases, imprints gradually begin to have an edge. At a selectivity of 0.15, imprints are better than the quadratic_dp zonemap for the sorted and rand_clusters columns. It takes until a selectivity of 0.35 for the same crossover to occur for the correlated column. For the random column, imprints are marginally faster than no-index scans until around 0.05 selectivity at which point no-index scans become the fastest method.

All three query methods have an approximately linear growth curve until very low selectivity at which point there is a dramatic drop in latency. This is likely due to the number of
CPU branch mispredictions rapidly going down at around 0.95 selectivity.

Another observation is that the slope of the growth curve for imprints is notably lower than that for the optimized zone maps. A possible explanation for this is that imprints have a more efficient method of testing if data should be scanned or not. Zone maps must perform two comparisons while imprints simply have to perform the much faster operation of a single bitwise AND. It’s possible that at lower selectivities, this ends up compensating for the imprints’ overhead of having to scan through a data structure which is 18% the size of the column.
In literature and in life we ultimately pursue, not conclusions, but beginnings.

Sam Tanenhaus

In this thesis, we discussed the use of zone maps as a data structure for secondary indexing in relational databases. We motivated the need for non-uniform zones, formulated a cost function that models the expected latency of a zone map scan for a random query, and presented a number of different algorithms which can generate zone map layouts that
minimize this cost. We evaluated our algorithms in a standalone column store prototype as well as in an open source column-oriented database called MonetDB. In both testbeds, we found that our optimized zone maps function effectively as lightweight secondary indices, obtaining performances that, depending on the column’s data distribution, range from as fast to much faster than the best uniformly-sized zone maps. In the MonetDB experiments, we also found that at medium to high levels of selectivity, optimized zone maps can yield considerably better performance than another secondary indexing data structure, column imprints. However, the relative performance of these zone maps drops off as query selectivity decreases.

These conclusions inspire a lot of potential new directions of research. One idea is to apply the same static optimization approach from this work to other secondary indexing data structures. In particular, based on the results of Chapter 2, it is clear that column imprints are a robust data structure, but they fail to achieve good performance for high-selectivity queries. A potential solution might be to allow for each imprint to cover a variable number of cache lines. To determine the optimal imprint layout, similar strategies to the ones in Chapter 1 could be applied.

Another potential direction of research is based on the observation that optimized zone maps are capable of significantly accelerating scans in relational databases. Since this is the case, it might be worthwhile to try to apply them to other categories of databases where scans are traditionally slow such as NoSQL key-values stores [12]. Specifically, there is the potential to build range-query-accelerating secondary indices on top of fundamental data structures in that domain, namely the log-structured merge (LSM) tree [14].
A.1 Pseudocode for Zone Map Optimization Algorithms

These are some high-level Python implementations of the algorithms described in sections 1.2, 1.3, and 1.4.
A.1.1 Quadratic Dynamic Programming

def quadratic_dp(z_base):
    n = len(z_base)

    DP_zones = [None] * (n + 1)
    DP_costs = [0] * (n + 1)
    DP_step_size = [0] * (n + 1)

    for pos in reversed(range(n)):
        best_zone = z_base[pos]
        best_cost = cost(best_zone) + DP_costs[pos + 1]
        best_step_size = 1

        curr_zone = z_base[pos]
        for i in range(pos + 1, n):
            curr_zone = coalesce(curr_zone, z_base[i])
            curr_cost = cost(curr_zone) + DP_costs[i + 1]

            if curr_cost < best_cost:
                best_zone = curr_zone
                best_cost = curr_cost
                best_step_size = i - pos + 1

        DP_zones[pos] = best_zone
        DP_costs[pos] = best_cost
        DP_step_size[pos] = best_step_size

    pos = 0
    optimal_zones = []
    while pos < n:
        optimal_zones.append(DP_zones[pos])
        pos += DP_step_size[pos]
    return optimal_zones
A.1.2 Constrained Dynamic Programming

```python
def constrained_dp(z_base, group_sizes):
    n = len(z_base)
    DP_zones = [None] * (n + 1)
    DP_costs = [0] * (n + 1)
    DP_step_size = [0] * (n + 1)

    queues = [Queue() for k in group_sizes]
    min_heaps = [MinHeap() for k in group_sizes]
    max_heaps = [MaxHeap() for k in group_sizes]
    curr_zones = [EmptyZone() for k in group_sizes]

    for pos in reversed(range(n)):
        best_cost = float('inf')
        best_step_size = 1
        best_zone = z_base[pos]

        for i in range(len(group_sizes)):
            # update curr_zones[i] in O(log(group_sizes[i])) time
            queues[i].append(z_base[pos])
            min_heaps[i].add(z_base[i].min)
            max_heaps[i].add(z_base[i].max)
            curr_zones[i] = coalesce(curr_zones[i], z_base[i])

            if len(queues[i]) > group_sizes[i]:
                item = queues[i].pop_front()
                min_heaps[i].remove(item.min)
                max_heaps[i].remove(item.max)

                curr_zones[i].size -= item.size
                curr_zones[i].min = min_heaps[i].get_min()
                curr_zones[i].max = max_heaps[i].get_max()

        # update the best zone
        curr_cost = cost(curr_zones[i]) + DP_costs[pos + group_sizes[i]]
        if curr_cost < best_cost:
            best_cost = curr_cost
            best_zone = curr_zones[i]
            best_step_size = group_sizes[i]

        DP_zones[pos] = best_zone
        DP_costs[pos] = best_cost
        DP_step_size[pos] = best_step_size

    pos = 0
    optimal_zones = []
    while pos < n:
        optimal_zones.append(DP_zones[pos])
        pos += DP_step_size[pos]
    return optimal_zones
```

```
44
A.1.3 Greedy Algorithm

```python
def multipass_greedy(z_base):
    optimal_zones = []
    curr_zone = z_base[0]

    for i in range(1, len(z_base)):
        potential_merge = coalesce(curr_zone, z_base[i])
        if cost(potential_merge) < cost(curr_zone) + cost(z_base[i]):
            curr_zone = potential_merge
        else:
            optimal_zones.append(curr_zone)
            curr_zone = z_base[i]
    optimal_zones.append(curr_zone)

    # if there might be room to improve, do another pass
    if len(optimal_zones) != len(z_base):
        return multipass_greedy(optimal_zones)
    return optimal_zones
```

A.1.4 Pairs Algorithm

```python
def multipass_pairs(z_base):
    optimal_zones = []

    for i in range(0, len(z_base), 2):
        potential_merge = coalesce(z_base[i], z_base[i + 1])
        if cost(potential_merge) < cost(z_base[i]) + cost(z_base[i + 1]):
            optimal_zones.append(potential_merge)
        else:
            optimal_zones.append(z_base[i])
            optimal_zones.append(z_base[i + 1])

    if len(z_base) % 2 == 1:
        optimal_zones.append(z_base[-1])

    # if there might be room to improve, do another pass
    if len(optimal_zones) != len(z_base):
        return multipass_pairs(optimal_zones)
    return optimal_zones
```
A.1.5 Bisection Algorithm

def bisection(z_base):
    n = len(z_base)

    # compute coalesced zones in the forward and backward directions
    forward = [EmptyZone()]
    for i in range(n):
        forward.append(coalesce(forward[-1], z_base[i]))

    backward = [EmptyZone()]
    for i in reversed(range(n)):
        backward.append(coalesce(backward[-1], z_base[i]))

    # figuring out the optimal split point
    best_split = 0
    best_cost = float('inf')
    for i in range(n):
        split_cost = cost(forward[i]) + cost(backward[n - i])
        if split_cost < best_cost:
            best_cost = split_cost
            best_split = i

    # recursively bisect the left and right halves if it might be worthwhile
    # otherwise, return a single fully coalesced zone
    if best_split != 0:
        return bisection(z_base[:best_split]) + bisection(z_base[best_split:])
    else:
        return [forward[n]]
A.2 Data Generation Algorithms

These are the exact algorithms which generated the data described in Table 1.1.

```python
def generate_sorted(n):
    return [i/n for i in range(n)]

def generate_random(n):
    return [random.uniform(0, 1) for i in range(n)]

def generate_correlated(n):
    stddev = 0.001

    res = [0] * n
    for i in range(1, n):
        while res[i] <= 0 or res[i] >= 1:
            res[i] = res[i - 1] + random.gauss(0, stddev)
    return res

def generate_rand_clusters(n):
    max_cluster_size = 8192
    cluster_alignment = 1
    cluster_stddev = 0.001

    res = []
    while len(res) < n:
        cluster_size = cluster_alignment * random.randint(1, max_cluster_size)
        cluster_mean = random.uniform(0, 1)

        while cluster_size > 0 and len(res) < n:
            value = cluster_mean + random.gauss(0, cluster_stddev)
            value = max(0, min(1, value, value))
            res.append(value)
            cluster_size -= 1
    return res
```

47
References


   now-available-amazon-ec2-high-memory-instances-with-6-9-and-12-tb-of-memory-perfect-for-s/


