

Exploring Learned Indexes for Approximate Query Processing and Visual Interfaces

By Katharine N Sedlar

Submitted to the
Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of

Master of Electrical Engineering and Computer Science

at the

Massachusetts Institute of Technology

June 2019

© 2019 Katharine N Sedlar. All rights reserved.

The author hereby grants to M.I.T. permission to reproduce and to distribute publicly paper and electronic copies of this thesis document in whole and in part in any medium now known or hereafter created.

Author:

Department of Electrical Engineering and Computer Science
May 24, 2019

Certified by:

Tim Kraska, Thesis Supervisor
May 24, 2019

Approved by:

Katrina LaCurts, Chair, Master of Engineering Thesis Committee
May 24, 2019

Exploring Learned Indexes for Approximate Query Processing and Visual Interfaces

by Katharine Sedlar

Submitted to the Department of Electrical Engineering and Computer Science

May 24, 2019

In Partial Fulfillment of the Requirements for the Degree of Master of Engineering in
Electrical Engineering and Computer Science

Abstract

Learned index structures are a promising new direction for improving data access. They offer the ability to do fast lookups in very large data sets, such as the kind needed for visual interfaces, without taking up huge amounts of memory. This paper discusses the extension of research with learned index structures as applied to approximating range search queries for visualization, some of the unexpected theoretical challenges this task brings up, and how learned index structures compare with other modern techniques for fast visualization.

Table of Contents

1. Introduction	5
2. Related Work	8
3. Background	10
2.1 B-Trees and Other Basic Non-Learned Structures	10
2.2 Learned Index Structures	11
2.3 Comparison	13
4. Design and Implementation	15
4.1 KD-Trees and their Implementation	15
4.2 Initial Implementation of Learned Indices for Range Search	17
4.3 Learned Index Structures Using a Uniform Approximation	20
4.4 Issues that Arose and their Solutions	22
5. Results	28
5.1 Results of Learned Index Implementation	28
5.2 Comparison with Related Work	34
6. Conclusion and Future Work	38
7. Bibliography	41

List of Figures

1. Figure 1	10
2. Figure 2	11
3. Figure 3	12
4. Figure 4	14
5. Figure 5	15
6. Figure 6	17
7. Figure 7	19
8. Figure 8	21
9. Figure 9	23
10. Figure 10	25
11. Figure 11	30
12. Figure 12	31
13. Figure 13	31
14. Figure 14	33
15. Figure 15	35
16. Figure 16	36
17. Figure 17	37

1. Introduction

Last year, a unique new approach to data structures was proposed: learned index structures [1]. The premise was that all indices – from B-tree indices to hash indices – are models. However, the data structures for these indices are general purpose and don't take advantage of particular patterns in real-world data distributions. So, what if we replaced these indices with models that did make use of such patterns? Typically, this is impractical, because figuring out the exact pattern for some type of data takes a lot of human effort, not to mention the task of designing the specific structures themselves. However, using machine learning, a neural network could be trained on the data, learning the patterns of data and enhancing, or even replacing, traditional indices in data structures. Such a construction is known as a learned index structure.

According to “The Case for Learned Indices,” the paper which presented the approach, initial results for comparing learned index structures to traditional data structures are quite promising [1] . For range indices, learned index structures are 1.5 to 3 times faster than B-trees, and they use an astounding 1% of the memory! For point indices, learned index structures can reduce the number of conflicts by up to 77%, in comparison to normal HashMaps. For existence indices, learned index structures can improve Bloom Filters to be as much as 36% smaller.

Usually, learned index structures will show a greater improvement over traditional data structures as the dataset being used gets larger. Intuitively, this is both because the models can be trained on more data, and because learned models that work

by *predicting* the location of a data entry aren't great at finding the location exactly. For instance, a learned model might be very good at narrowing down the location of a data point from a data set of millions to a data set of thousands, but might struggle with the task of reducing the search space from thousands to hundreds.

Thus, these structures could ideally be applied to very big datasets that require fast lookups, particularly visualizations in which a user would want to quickly zoom in on or compare various attributes from a large dataset. For example, a user of a data visualization interface might want to analyze patterns of tweets in a particular year throughout the U.S. However, there might be several hundred million of these tweets [4], with all kinds of spatial, temporal, and categorical attributes, which can mean that pulling up a custom graph of the data aggregating the categories of data could take a prohibitive amount of time and space if not done carefully.

While learned index structures are well-suited for tasks of this magnitude, there has not yet been much research on expanding them for this use case. This type of data visualization requires querying the number of data points with particular attributes (e.g. with $x < 3$ and $20 \leq y < 32$), as one might do with a histogram. A query of this form is called a *range search*. Ideally, one could be able filter data by combinations of several different features, and quickly get a numerical estimation of how many data points satisfy the constraints. Thus, this thesis project was to expand the capability of learned index structures for use in range search queries, especially those used in the visualization of big data sets.

This paper will first discuss what **related work** there is for the task of fast query processing for visual interfaces with very large data sets. Next, it will provide **background** on how learned index structures work and how their properties compare with those of traditional data structures. Then, it will explain the **design and implementation** of the algorithm created for doing learned range search queries, as well as the challenges that were encountered. It will explain the **results** of comparing the algorithm with existing work in the field, before reaching the **conclusion** that summarizes the paper and present avenues of future research.

2 Related Work

While it is possible to use multi-dimensional data structures, such as KD-trees (described further in section 4.1) or R-trees, for the types of range search queries used in visualization, this section will focus on existing methods specifically developed for fast visualization of big data. Some of the most notable ones are described in the ImMens [2], NanoCubes [3], and HashedCubes [4] papers. Many of these methods make use of pre-aggregation of the data, in order to reduce later query times.

The imMens system, developed in 2013 was the first to enable interactive brushing of very large datasets [2]. It did this by using data reduction methods such as binned aggregation, precomputing multivariable data tiles, and using schemes to enable efficient parallel processing. However, while imMens has fast query times, it can't support compound brushing of more than four dimensions, it requires preprocessing that is sometimes computationally expensive, and it uses memory proportional to its key space, which limits key space.

The Nanocubes technology was developed slightly later [3]. Nanocubes are a type of data cube which perform a large number of aggregations of data, without taking up the extremely large amount of space that traditional data cubes require. Nanocubes can support more dimensions overall than imMens, though it only allows for one spatial dimension and one temporal dimension. It also does not allow querying down to the individual record.

Hashedcubes, developed in mid-2016, are the most recent innovation [4]. Hashedcubes is an alternative to Nanocubes which avoids a number of aggregations,

allowing it to have a simpler implementation and a more compact representation. Its goal was to provide a simpler implementation of a data structure for large-scale interactive visualization, while maintaining fast query times and low memory usage.

Later in section 5.2, this paper will discuss performance comparisons made between the Nanocubes code and the implementation for learned indices, particularly with respect to query times and memory usage. Because the approach of using learned indices is so different than previous ways of optimizing data visualization, there is also potential to combine it with existing methods in the future, making advances in speed and memory usages that are superior to either approach alone.

3 Background

What exactly are the relevant traditional data structures, and how do their analogous learned index structures compare? What can and can't learned indices do? Below this paper will summarize the existing work in learned index structures, in order to explain the necessary background needed to understand the work of expanding learned indices for range search queries in visual interfaces.

3.1 B-Trees and Other Basic Non-Learned Structures

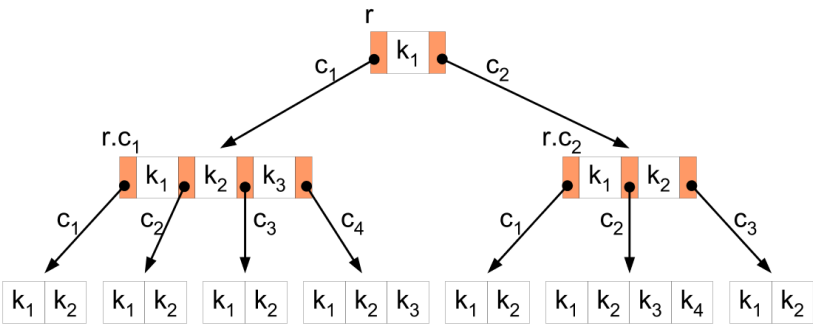


Figure 1: An example of a B-tree [9]

B-trees are currently a popular data structure in computer science. Conceptually, they are just generalized, self-balancing binary search trees with nodes that can have more than two children, as shown in figure 1. Because B-trees are shorter and wider than other similar search trees, they are better suited for tasks where one wishes to read a full disk block of data, as is often the case in databases. The nodes and leaves of a B-tree are often called pages, with each page potentially containing many values. Like a binary search tree, a B-tree has a search time of $O(\log n)$.

Because of the increased relevance, this paper will focus on learned index structures as applied to B-trees. However, other types of regular data structures, such as Hash-maps and Bloom filters, also can have an appropriate “learned” equivalent.

3.2 Learned Index Structures

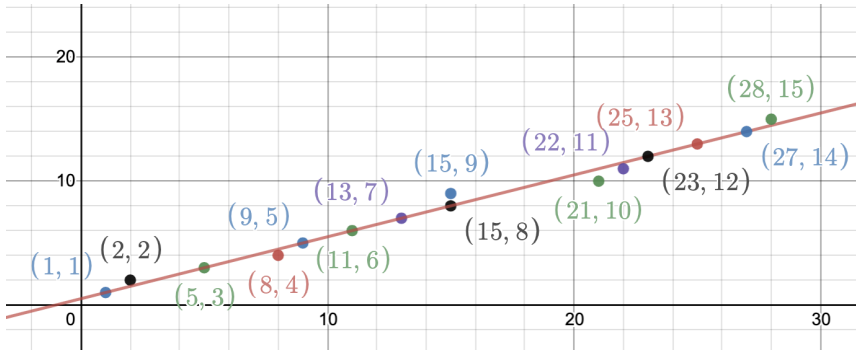


Figure 2: Using linear regression (i.e. finding a best-fit line) as a way of predicting the position (y axis) of data values (x axis)

Learned index structures are, at the most basic level, models that predict the positions of pieces of data. For example, we could imagine that for a sorted array of data [1, 2, 5, 8, 9, 11, 13, 15, 15, 21, 22, 23, 25, 27, 28], we might use linear regression to predict the position of data in the array, as shown in figure 2. In this case, training our model (i.e. using a line of best fit) would give us a model of $p = 0.5 * v + 0.5$ (where v is value and p is position). While the model would not perfectly predict the data, it would be a close approximation, and the model would use less space and generate a prediction more quickly when compared with storing the data in a B-tree. For much larger datasets with a more sophisticated method of training the model on the data, the memory and performance benefits would be even larger.

In practice, the use cases we are looking at might involve millions and millions of records. One consideration we now have is the fact that for a data distribution, the process of decreasing the search space from a region of e.g. 100 million records to 10 thousand records can be very different than the process of reducing the search space from 10 thousand records to 100 records. Thus, it can be beneficial to use a hierarchical structure of models.

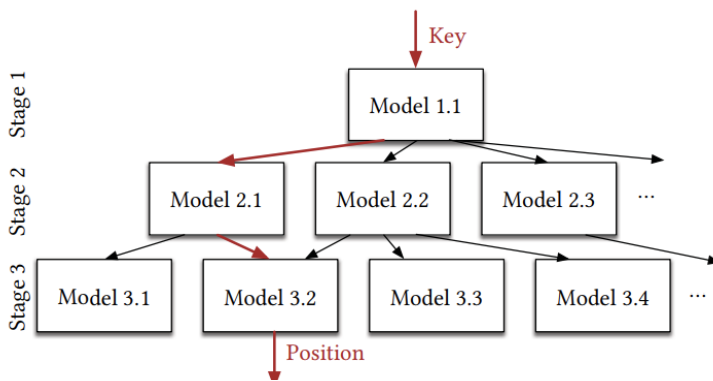


Figure 3: Using a hierarchy of models (taken from [1] with permission)

In such a hierarchy such as in figure 3, each model takes in a key as input and then outputs another model to continue the process, until the final model predicts the position of the record. This hierarchy notably does not need to be a tree; models can cover highly varying models of records, and a model could be used repeatedly in this process. Rather than proceeding to narrower and narrower locations, the hierarchy can be thought of each model being an “expert” for certain keys, with the models passing on the job to other models that have more knowledge than them when needed.

This also means that we can have models with different complexity, depending on the situation; for instance, the top model could be a complex neural net, while some of the bottom models could be simple linear regression models, or even B-trees

themselves, if the data proves particularly hard to learn. When training the models, we can pick the method best suited to the data distribution. So, for a particularly strange distribution, it could be possible that all models are B-trees (and thus the entire hierarchy is a giant B-tree), essentially setting B-trees as the worst-case performance of the process.

It is worth noting that currently, learned index structures are only being applied to lookups, and not data insertion or deletion. However, the use case we are optimizing for (doing fast attribute filtering in data visualizations) only requires lookups and searching; it's fine if insertion, deletion or model retraining takes more time using a more traditional method, because it doesn't affect the user.

3.3 Comparison

As we have already seen, learned indices outperform B-trees in terms of memory and lookup speed. Thus, accuracy is really the only metric in which learned structures can be outperformed. While we can successfully increase accuracy in learned indices by expanding the hierarchy of models, won't it be impossible to get the same *guarantees* of error boundaries that you would have with B-trees? If learned index structures are merely predicting the location of an entry, one might think this would always be outperformed by structures that track the exact location of each entry.

However, it turns out this isn't really relevant. Because the data must be sorted for supporting range requests, any error can be corrected with a brief local search. Furthermore, B-trees themselves can be thought of as also having an error range.

Namely, they have a min-error of 0 and a max error equal to the page size. Figure 4 shows this comparison.

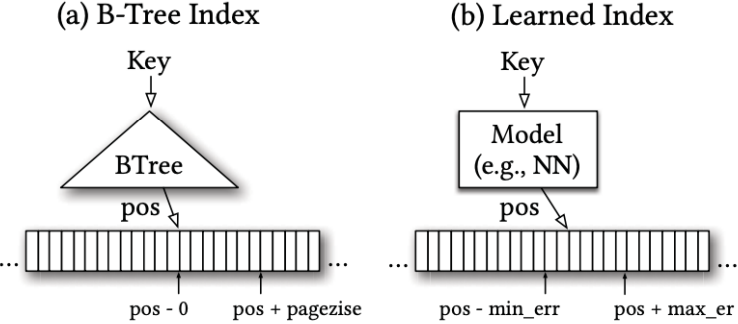


Figure 4: Error guarantees in B-trees vs learned indices (taken from [1])

Additionally, B-trees can only maintain their error guarantees for stored keys, not for all keys. When new data is acquired, a B-tree may need to rebalance itself, just as a learned model might need to retrain itself. If we consider learned models in the same way as B-trees, then we can simply execute the model for all keys ahead of time, and remember the worst results, in order to get the same type of min and max error guarantees. Because we are only concerned about performance in lookups, doing this additional pre-computation is not an issue. Thus, learned indices actually compare perfectly fine with traditional data structures with respect to accuracy and error guarantees.

4 Design and Implementation

Now, this paper will explain the details of how the learned index algorithm for range search works. It will first talk about KD-trees, which are the classical data structure for two-dimensional data and which are used in the hierarchy of models. Then, it will talk about how one might initially implement learned indices for range search based on their single-dimensional case, before explaining the main complication and the estimation used to compensate for it. Finally, it will discuss issues that came up during the implementation and how they were addressed. For those curious about the details of the implementations, code is available on Github upon request for access [5].

4.1 KD-Trees and their Implementation

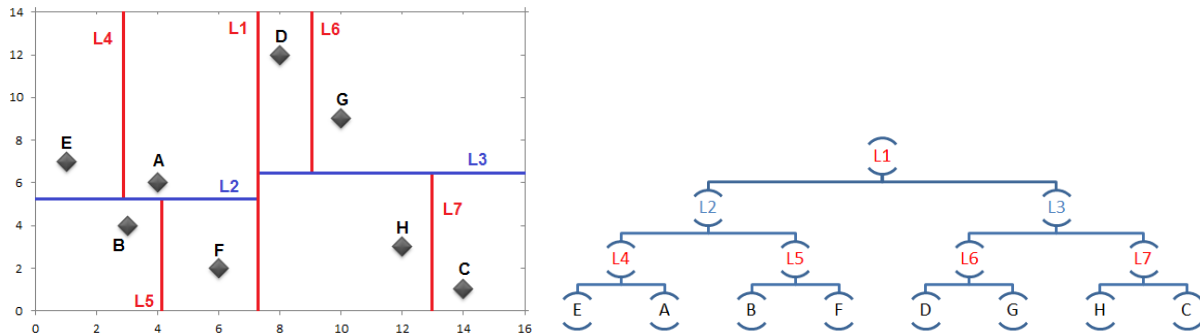


Figure 5: A two-dimensional KD-tree (right) [11], next to a plot of its points and the dimensions it is being split by (left) [10]

Before beginning the implementation of a learned index program for range search, it was first necessary to have a “non-learned” point of comparison: KD-trees. KD-trees are essentially just B-trees in multiple dimensions, where each level of the tree switches what dimension it divides the search space by [6]. For instance, a

two-dimensional KD would alternate every other level between dividing by “x” coordinate and by “y” coordinate. Figure 5 shows a cluster of points on a graph, and how a KD-Tree splits the overall area of the graph by repeatedly halving the point space until only one point is contained within a particular area. KD-trees can thus perform range search, because a region of the coordinate plane corresponds easily with the tree.

Thus, the first task in the learned index algorithm was implementing KD-trees with range search. Although python does have a scipy implementation of KD-trees [7], they don't include range search, so the algorithm utilized a version designed from scratch. The code focused on two-dimensional KD-trees, though the setup could be easily extended to arbitrary dimensions. The initial implementation would read from a csv file of data, and insert each point as a new leaf node in the KD-tree. Then, it would perform a range search by recursively walking down the tree, not pursuing any paths that would fall out of its range. As a performance improvement, the tree was augmented so that each node kept track of how many nodes were below it, which mean that if the recursive search found a branch entirely within the range search, it could just add the node count to its total sum without having to pursue any further paths down that branch.

However, without any balancing of the tree, this implementation of a KD-tree could become lopsided quickly. And indeed it always would become lopsided, because the data in the csv file would be sorted (thus the root of the KD-tree would be the point with the minimal x value). While a lopsided KD-Tree would still produce accurate results, eventually it would return a python recursion error for very large data sets or

data sets where the y value was also semi-sorted. Thus, the algorithm also ended up shuffling the data before generating a KD-tree. Python’s function for shuffling lists did not work on numpy arrays, so a custom shuffling function was implemented.

4.2 Initial Implementation of Learned Indices for Range Search

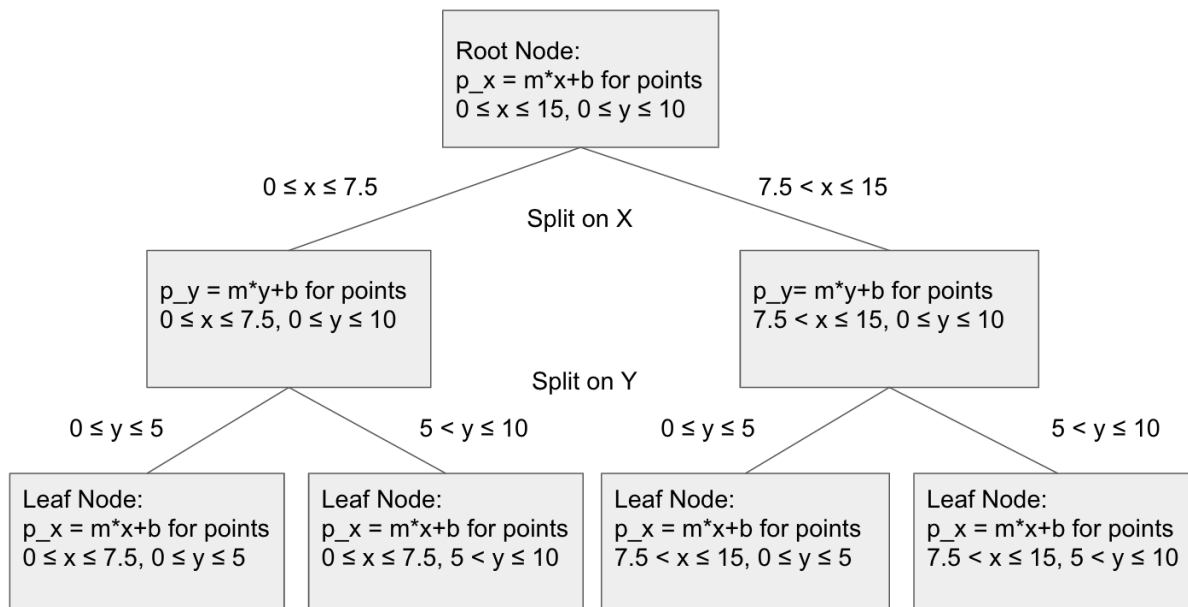


Figure 6: The KD-tree structure for a model’s hierarchy, consisting of various nodes and their sub-models

Next, we examine the initial approach one might use to do range search via learned indices. The program was set up to mirror the design for the single dimensional case, as described in the “The Case for Learned Index Structures” paper, using a learned model that would be trained on the data. For simplicity, this implementation of the algorithm used linear regression instead of any complex neural net.

The model also made use of the “hierarchy of models” concept from section 3.2, using a KD-tree for the hierarchical structure. A particular model of the data would consist of many *sub-models*, which corresponded to individual nodes in the model’s

hierarchy, trained on a particular subset of the data. The *depth* of a model is how many times the tree splits into having children (i.e. it is 1 plus the number of nodes in the height of the tree). The *width* of a model is the maximum number of children that a node can have. A particular sub-model would store the slope m and the intercept b corresponding to the best-fit line of the data it was trained on. Each level of the hierarchy would alternate whether or not it split the dataset along the x or the y attribute, and thus a particular submodel would store not only the m and b values for the attribute that it was trained on, but also the m and b values of its parent node, which corresponded to the best-fit line of the other attribute. The indices of a point would then be predicted by plugging in the x and y values of the point to the corresponding $p = m \times v + b$ equation, where p was the index position and v was the x or y value.

Figure 6 displays the KD-tree structure of the hierarchy of nodes for a depth 2 width 2 model trained on the range $0 \leq x \leq 15$, $0 \leq y \leq 10$. To query the indices of an individual point, the algorithm would start at the root node of the model, then walk down the hierarchy until reaching a leaf node, at which point the leaf node's submodel would produce an estimate for the indices of a point. For instance, to get the indices for the point (3, 4), an algorithm using the KD-tree hierarchy from figure 6 would use the sub-model of the bottom-left node for its estimation. However, attempting to actually do estimations for range search queries brought a critical problem to light.

In the single-dimensional case, you could count the number of values x , such that x is between a and b , by querying to get the index of a in the data and the index of b in the data, then subtracting their indices. So performing a search for all the points in a

range required merely two estimates, and if those were correct, then the overall result would be correct too.

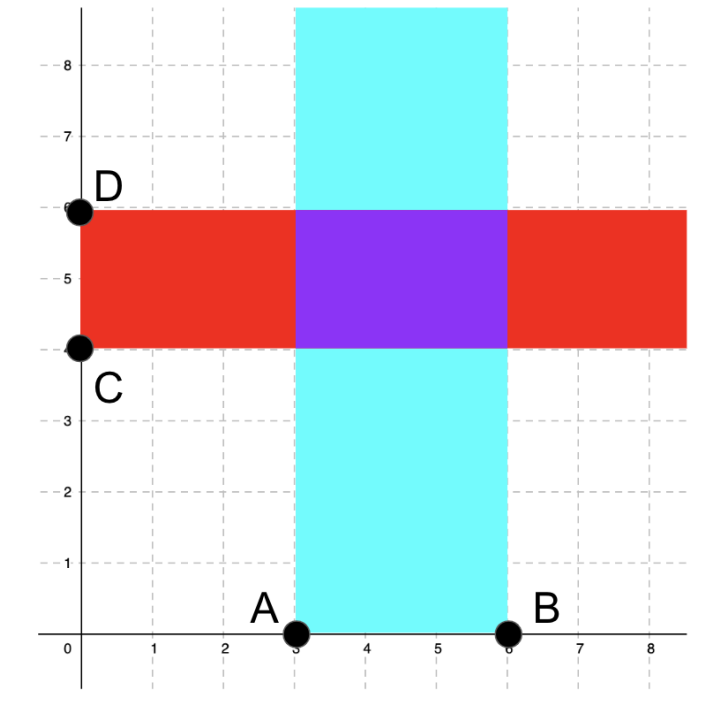


Figure 7: We can get an estimate for the number of points for the blue region, and an estimate for the red region, but not their purple intersection.

For the two-dimensional case of a range search, one might want to count the number of points (x, y) , such that $a \leq x \leq b$ and $c \leq y \leq d$. The analogous way to do this would be by getting the indices of a and b along the x dimension, then getting the indices of c and d along the y dimension, and subtracting the corresponding indices to get the number of points between them. However, while this tells us the number of points between a and b and the number of points between c and d , **we don't actually have any way of knowing how many are in their intersection.** Figure 7 visualizes an example of this problem.

Let's say we're trying to estimate how many points (x, y) there are such that $3 \leq x \leq 6$ and $4 \leq y \leq 6$. We can find out that there are 20 points with $3 \leq x \leq 6$ and 30 points with $4 \leq y \leq 6$. While we know that there can't be more than 20 points such that $3 \leq x \leq 6$ and $4 \leq y \leq 6$, we can't actually know anything more specific than that. Maybe **all** 20 points where $3 \leq x \leq 6$ also have the property that $4 \leq y \leq 6$. But maybe all the points with $3 \leq x \leq 6$ have $y < 4$, and all points with $4 \leq y \leq 6$ have $x > 6$. Using the naive method of attempting to do the same thing as in the single-dimensional case, we simply have a non-deterministic answer, and we can't do better than getting an upper bound.

4.3 Learned Index Structures Using a Uniform Approximation

However, the naive method is not simply a dead end. We can do a reasonably good job of approximating indices simply by using a *uniform estimation*. Basically, we assume that data is uniformly distributed within a sub-model. This means that when we get to a sub-model and are trying to estimate the number of points within the range search area, we calculate a count of the number of points within the bounds of one dimension, figure out what fraction of the total sub-model range is used in the range search in the other dimension, and then multiply the count times that fraction.

As an example, let's walk through a hypothetical range search. Imagine that we have the same data set and model as before, where the data set has all data such that $0 \leq x \leq 15$, and all data has $0 \leq y \leq 10$. Let's say we are trying to do a range search on the query from earlier, to find the number of points in the range such that $3 \leq x \leq 6$ and $4 \leq y \leq 6$. Our model has depth 2 and width 2, so like in figure 6, it has 4 leaf-nodes with sub-models that queries could be directed to within the model hierarchy: " $0 \leq x \leq 7.5$

and $0 \leq y \leq 5$ ", " $0 \leq x \leq 7.5$ and $5 < y \leq 10$ ", " $7.5 < x \leq 15$ and $0 \leq y \leq 5$ ", and " $7.5 < x \leq 15$ and $5 < y \leq 10$ ".

For our range search query, we must count the number of points estimated to be in each sub-model. As soon as we hit the intermediate sub-model with bounds " $7.5 < x \leq 15$ ", we know that there are 0 points found among all of its sub-models, because the bounds of the sub-model are entirely outside of our range search. But for the intermediate sub-model " $0 \leq x \leq 7.5$ ", we continue down to both leaf-nodes' sub-models.

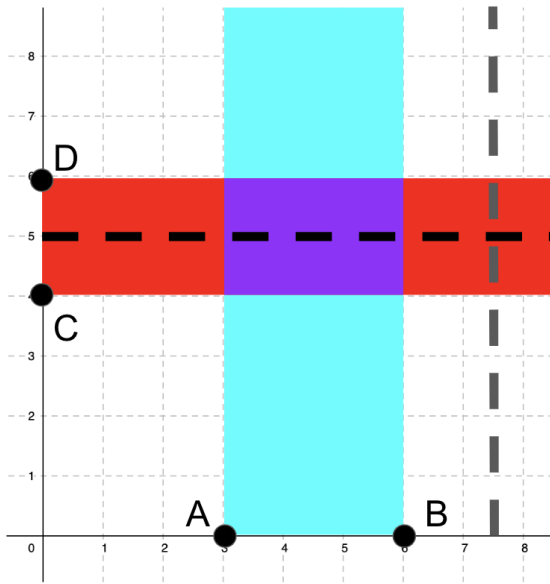


Figure 8: The algorithm divides the search space into subsections (divided by dashed lines), and we perform range search within the subsections. In the example, we query the for the number of points in the red region of the bottom left quadrant, then multiply by 0.4 to approximate the area of the purple region of the bottom left quadrant.

For the " $0 \leq x \leq 7.5$ and $0 \leq y \leq 5$ " sub-model, we query for the index of y-value 4 and of y-value 5, and we get might get "19" and "12" as responses, telling us that there are 8 nodes in the range $4 \leq y \leq 5$. Then, because our $3 \leq x \leq 6$ search is 40% of the

range of the sub-model's x-dimension, we multiply 8 by 0.4, estimating that the sub-model has 3.2 points in our search. This is visualized in figure 8. For the " $0 \leq x \leq 7.5$ and $5 < y \leq 10$ " sub-model, we query for the index of y-value 6 and of y-value 5, getting "22" and "1" as responses, telling us that there are 22 nodes in the range $5 < y \leq 6$. Again, we multiply this by 0.4 because $3 \leq x \leq 6$ search is half the range of the sub-model's x-dimension, getting 8.8 as a result. Thus, the model would estimate that there are $3.2 + 8.8 = 12$ nodes in the range $3 \leq x \leq 6$ and $4 \leq y \leq 6$.

Interestingly, the uniform estimation is not that inaccurate of an assumption to use, even for data with highly correlated x and y values (i.e. very non-uniform data). This is because highly correlated data is much easier to estimate using linear regression, so the index prediction for individual points will be particularly good, "counteracting" some of the error from the uniform estimation. Also, most real life data has a lot of scattering around whatever its "best fit" line is, so a "zoomed in" version of even very correlated data often won't be that far from uniformity anyway.

4.4 Issues that Arose and their Solutions

Unfortunately, the program had several issues in practice. This paper will now discuss how the algorithm encountered issues with data points that shared coordinates, errors with from the uniform estimation being applied to a zero-area model, and the density distortion effect that causes points to disproportionately be found on the edges for very small sub-models.

attr1	attr2	pos1	pos2
1	1	1	1
1	2	2	4
1	3	3	7
2	1	4	2
2	2	5	5
2	3	6	8
3	1	7	3
3	2	8	6
3	3	9	9

Figure 9: A table of the the points [(1, 1), (1, 2), (1, 3), (2, 1), (2, 2), (2, 3), (3, 1), (3, 2), (3, 3)] and their positions / indices in each dimension with respect to the other points.

One issue was that if the program was tested with a data set that was *actually* perfectly uniform (to establish a best-case scenario to make sure the method worked), using linear regression even with an arbitrary model width and depth would produce some error. Consider a data set with the nine points [(1, 1), (1, 2), (1, 3), (2, 1), (2, 2), (2, 3), (3, 1), (3, 2), (3, 3)], with index positions labelled as in figure 9. If one wanted to do a range search for $1 \leq x \leq 3$ and $1 \leq y \leq 3$ over the entire data set, you would expect to get an answer of 9. However, in practice one would get 7 as a result instead.

This is because the python functionality cannot distinguish what the correct index (out of multiple indices for a coordinate value) is when doing a linear regression fit. For instance, there are 3 points with x-coordinates of 1, with index positions and 1, 2, and 3 (essentially picked arbitrarily between the other points with the same x-index). Python can't tell which index the range search should be starting at, so it goes with the average, 2. Similarly, it uses the average index of x-coordinate 3, which is 8, when deciding the maximum index of the range search. So instead of counting the number of points between indices 1 and 9, it counts the number of points between indices 2 and 8.

Fortunately, this is not often an issue with more realistic data sets and range searches. The problem only arises when multiple points share a coordinate, and only when the border of the range search crosses the coordinate exactly, rather than strictly containing it. It is also small in magnitude, so large datasets that have a few overlapping coordinates by coincidence won't be affected very much. As far as improving the testing the algorithm, we can use randomly generated float numbers with 15 digits of precision. This makes it very unlikely to actually have multiple points with exactly the same index for either attribute, and for very large datasets, any collisions that do exist are negligible. Thus, for further testing, the program randomly generated custom data for any number of points within given ranges of x and y values, using a dataset of size 100,000 or larger. This essentially made the problem go away entirely.

Another puzzling issue with the algorithm was the fact that very large depths and widths somehow would result in less accurate results than small ones. This happened even when the depth and width was so large that an individual leaf model of the hierarchy would be trained on a **single** point, and thus there should have been no prediction error whatsoever. Increasing the depth or width of the model should have always been decreasing the error, because it would result in more precise training, so this was strange.

Eventually it became apparent that part of this problem was the math of making the uniform estimation with a zero-area model. Consider a model trained on simply the point (1, 2), and a range search query that includes that point. When the program goes to estimate the number of points in the model that are contained in the search query, it

looks at some attribute, e.g. the x-attribute and sees that the range of x values has minimum 1 and maximum 5, so there is 1 point that satisfies the x-constraint. It goes to satisfy the y constraint, and sees that the range has minimum 1 and maximum 4, so it should multiply the 1 point included in the x-constraint by the full area of the model. But, the area of the model is 0, so it concludes that there are 0 points in the range. The same problem would come up if you had a model trained on more than 1 point, but the points shared a coordinate.

Fortunately, this problem was solvable. Instead of always using the uniform estimation for a range search in a sub-model, there can be a special case for when the points that a sub-model is trained on are all completely contained in the range search. Each sub-model can keep track of the number of points it was trained on, and when a range search includes that entire set, the sub-model can just return that number, rather than needing to make an estimate.

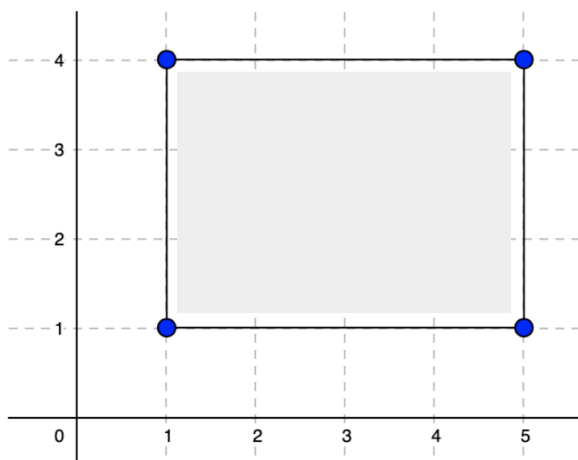


Figure 10: Doing a range search in a sub-model defined by its min/max values means that the points in the sub-model are disproportionately along the edges.

However, while this solution reduced the increased inaccuracy from models with large depths and widths, it did not eliminate it. There was also a harder-to-eliminate effect, which we will call the *density distortion effect*. Consider that in any sub-model, the bounds of the sub-model are determined by actual points in a sub-model. So if you know that a small sub-model has a min x value of 1, a max x value of 5, a min y value of 1, and a max y value of 4, then you know that some point within the sub-model has an x-value of 1, some point has an x value of 5, some point has a y value of 1, and some point has a y value of 4. Even if you query the area with $1.1 \leq x \leq 4.9$ and $1.1 \leq y \leq 3.9$, as shown in figure 10, you are much less likely to hit actual data points than the uniform estimation would predict.

This is not just a coincidental result from a particularly chosen type of sub-model. The fact that sub-models are defined by the minimum and maximum values of the points they are trained on means that points will *always* be located disproportionately at the edges of the sub-model, no matter how uniformly the data was distributed to begin with. For a large sub-model trained on thousands of points, this effect is very small; there are at most 4 points that can cause the distortion. However, for very small sub-models, the density distortion effect plays a major role in the accuracy of estimating the number of queried points in the sub-model.

Furthermore, models with large depths and widths don't just have smaller sub-models than models with moderate depths / widths; they also have *more* sub-models. Combined with the fact that accuracy improvements from increasing depth/width have a lesser magnitude with greater depth/width, the density distortion

effect ends up causing the accuracy of very large models to actually be lower than that of medium-sized models. It's also the case that even if a way to not use the uniform estimation for learned ranged search was developed (i.e. we could use an estimation more accurate for the individual data set), the density distortion would still mean that points in small sub-models are more dense around the edges of the sub-model, regardless of what the "real" distribution was.

The density distortion effect is not intractable; one could probably add some casework to the algorithm. It's guaranteed that a single sub-model (other than sub-models with exactly 1 point) is distorted by 2-4 points. So, the sub-model could track these points and separate them from the uniform estimation, individually adding them back if a range search contains them. However, there are some subtleties to implementing this, and it actually adds a significant increase to how much memory an individual sub-model takes up. It's also still a small effect for reasonably sized models that take up a practical amount of memory. The density distortion effect only becomes a major problem once models are already of a greater size than desired in practice.

5 Results

Now, this paper will discuss how the algorithm worked on a different types of data sets, explain the setup of the graphs, and examine how the results compare to those of Nanocubes, one of the existing methods of doing query processing on large multi-dimensional data sets.

5.1 Results of the Learned Index Implementation

The learned index implementation was tested on several of datasets of different size, with varied parameters of width and depth, to see how accurate the results were for different model sizes. For a given model, its size can essentially be calculated by taking the number of nodes in the model, multiplied by the byte size of the node.

The models used in practice were a bit larger than the theoretical models that were graphed. This was because several node attributes in the code were non-essential, but served to make the code easier for humans to understand and more flexible for different parameters. For instance, a list structure in python takes up more space than 9 floating point attributes put together, but having a list of the node's children means that the same code can be used for different model widths.

The actual implementation has 14 attributes plus a list of children. Each attribute is a float or a pointer, which would each be 4 bytes in a 32-bit system. However, 4 of those attributes could be removed, and one of them is a boolean, which could only use 1 byte. Instead of a list for the node's children, one could instead use one attribute for each child, given that the width of the model is known in advance. Thus, for the purposes of graphing size, a node was considered to be $9 \times 4 + 1 + 4 \times w$ bytes, where w

is the width of the model. For example, a model with width 2 would have 45 bytes per node.

Because the design of the model depended on the “uniform estimation” explained in section 4.3, the program was tested on data sets of varying degrees of uniformity, generated within a range of x and y values (in this case, the ranges $0 \leq x \leq 10$ and $10 \leq y \leq 20$ were arbitrarily chosen). For each case, both data sets of 100,000 points and data sets with 1,000,000 points were used. Notably, with this many points, the actual data points that happen to be generated are fairly unimportant, and variation between two datasets generated with the same method can be considered noise. The algorithm used model depths from 2-8 and model widths from 2-6, performing 100 range search queries for 10 equally spaced x bins and 10 equally spaced y bins in the dataset, and then computing the average error among those queries for each individual width and depth.

All plots for the learned index algorithm use a logarithmic scale for the model size. This is because the very large models with high depths and widths are so much larger than even the medium sized models, they would otherwise completely dominate the graph.

Additionally, the graphs use different colors for different widths of the model, and include line segment connections between the data points of the same width, in order to more clearly see the effect of depth. This does induce some confusion, because the graphs are not continuous (they are just a collection of 63 points), but it is clearer overall. In the graphs, blue is width 2, orange is width 3, green is width 4, red is width 5, and purple is width 6.

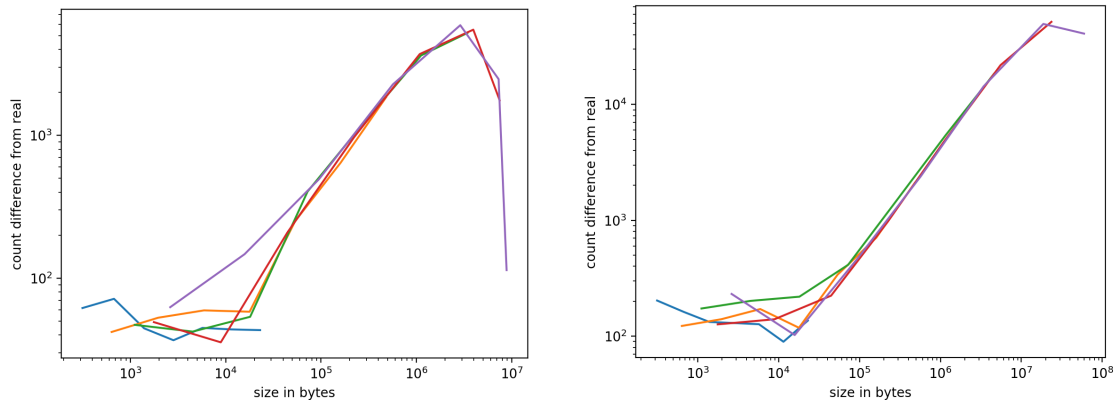


Figure 11: Memory usage for learned index models of 100,000 data points (left) and 1,000,000 data points (right), with data generated with **no** correlation.

First, there were data sets generated totally at random, which would on average have a uniform distribution. Figure 11 shows plots of these datasets. The small error in the middle of the graph is ~ 40 for the 100,000 point data set, and ~ 90 for the 1 million point data set.

Then there were data sets generated somewhat randomly, but without perfect correlation. In this case, random y values were generated, but multiplied by the the ratio of of the corresponding x value to the max x value, which meant that y values would tend to be smaller than their range would suggest, and that a small x value implied a small y value. Figure 12 shows plots of these datasets. The small error in the middle of the graph is ~ 130 for the 100,000 point data set, and ~ 680 for the 1 million point data set.

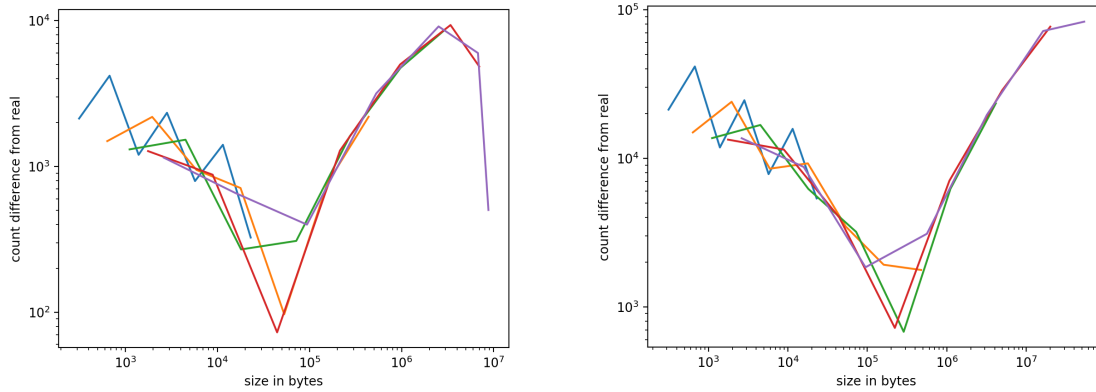


Figure 12: Memory usage for learned index models of 100,000 data points (left) and 1,000,000 data points (right), with data generated with **some** correlation.

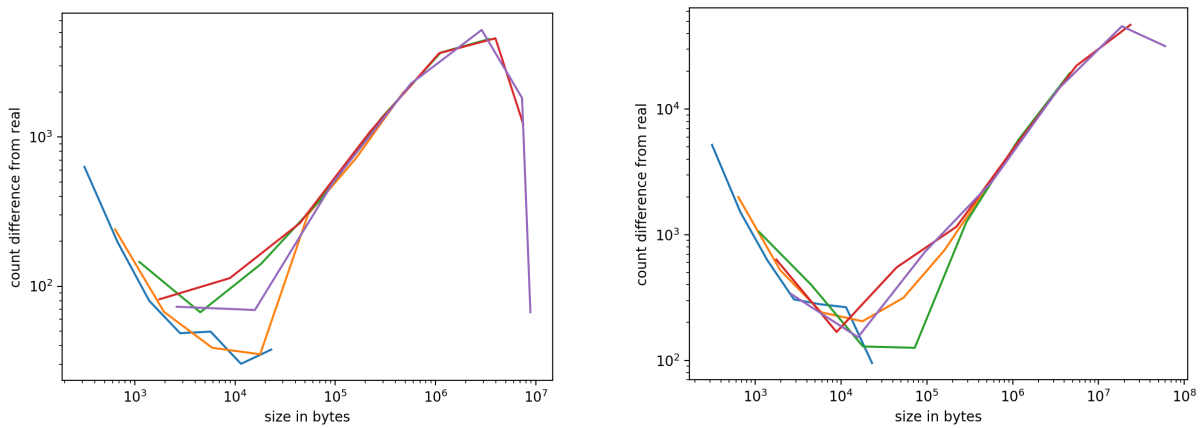


Figure 13: Memory usage for learned index models of 100,000 data points (left) and 1,000,000 data points (right), with data generated with **total** correlation.

Finally, there were data sets generated with perfect correlation; in this case, random x values were generated, and the y value was set to be twice the x value, plus the minimum y value. Figure 13 shows plots of these data sets. The small error in the middle of the graph is ~ 30 for the 100,000 point data set, and ~ 95 for the 1 million point data set.

Overall, the graphs follow a similar pattern: moderately high error for small model sizes, low error for medium sizes, very high error for large sizes, and eventually low error for very large sizes (though sometimes the graphs do not fully show the latter case). The first two features are what we expect from a learned model. The very high error from large sizes, although not a desired feature, is explained by the density distortion effect, mentioned in section 4.4 as a distortion that arises having a lot of very small sub-models. This effect is then becomes counteracted once sub-models get so small that the range search no longer intersects with only fractions of the model.

It was surprising how big of an error the density distortion effect caused once models started to get very large. Even while the algorithm is getting improvements to the error in the small, middle section, it's also the case that it is simultaneously having error added to it from the density distortion effect. This is most obvious from the plots of randomly generated data, where the data is nearly uniform to begin with, and to the degree that the model reduces error, it is from being trained on noise. While some initial decreases in error are visible at the start, the error quickly increases for larger depths and widths, because the increased layers of linear regression are not helping enough to counteract the density distortion effect. However, while the density distortion effect is a very prominent feature of the graphs, the model sizes for which the density distortion effect plays a major factor are already too large to be of practical use, and the comparison in section 5.2 focuses on the low center regions of the graphs.

Another interesting effect is that the greatest error is not in the perfectly correlated data (which is the pessimal result possible for the uniform estimation), but in

the semi-correlated data. Presumably this arises from the fact that it is much easier to train linear regression models on linear data.

Another way to visualize the results of the learned index implementation is to plot the result of various range searches, compared with the real number of points that the range search would return (as determined by the KD-tree range search implementation). This is shown in figure 14, which shows the result of doing 20 range searches for consecutive x-values, but specifically for $y \geq 15$. This was performed on the 100,000 data point semi-correlated data from figure 12 (again with the full data set generated in the range $0 \leq x \leq 10$ and $10 \leq y \leq 20$). This was done with $\text{depth} = 4$ and $\text{width} = 5$, one of the optimal settings for this type of data set. As can be seen, the learned range search does a fairly close approximation of the real results.

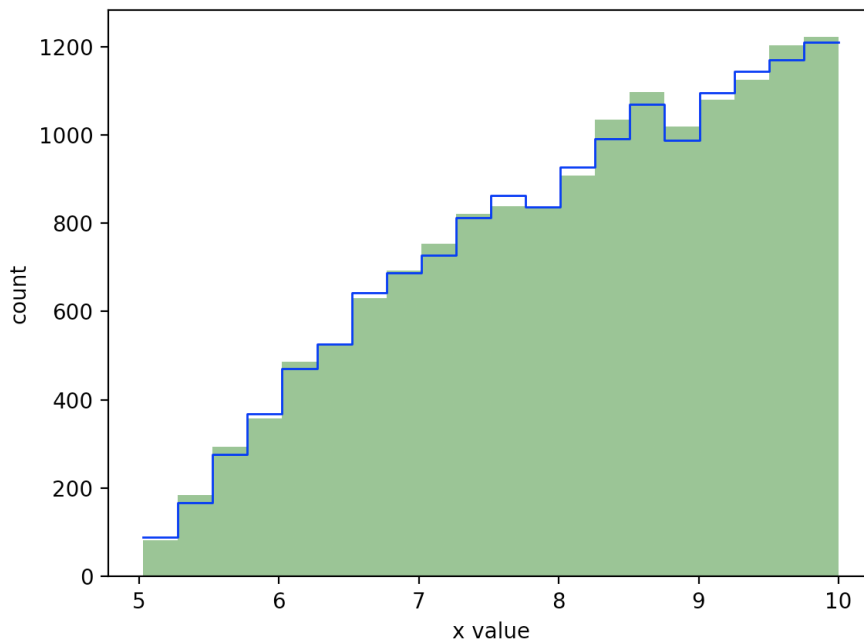


Figure 14: A plot of 20 different range searches, with the learned range search's predictions marked with the blue line and the real values shaded in green.

5.2 Comparison with Related Work

There were many challenges in trying to adequately compare the implementation with existing work. Undergraduate researcher Urmi Mustafi aided in the task of properly setting up the existing implementations,.

We first began by examining imMens, because it was the earliest work, and in some sense the simplest. However, while we could run the imMens implementation on its own sample data tiles, we found that the provided instructions for generating data tiles on a custom data set were insufficient and led to unresolvable errors.

Thus, we next tried to run the Nanocubes implementation. This had the additional difficulty of being an extremely large program written in C, which we had no prior experience with. There were no comments or instructions for how to use the implementation with custom data, so figuring out how to modify the code in appropriate sections was tricky.

We also encountered an issue where the Nanocubes implementation was always done on latitude / longitude data, and thus applied a Mercator projection to all of the data points, making the data not directly comparable. Unfortunately, we were unable to find a way in the code to remove the Mercator projection, so we decided to use scale down the data points to correspond to a very small geographic area (e.g. the city of Cambridge), such that the Mercator projection would have a negligible effect on the data points.

However, ultimately these issues were overcome. Though Nanocubes is not predictive in the same way that the learned index implementation is, it was still possible

to do a comparison of accuracy vs memory usage. Nanocubes has two modifiable parameters for this: depth and resolution. Depth refers to the depth of their model, similar to the learned index implementation. Resolution is what one might expect; it basically refers to how much individual data points are distinguished from another, or if they are all grouped together.

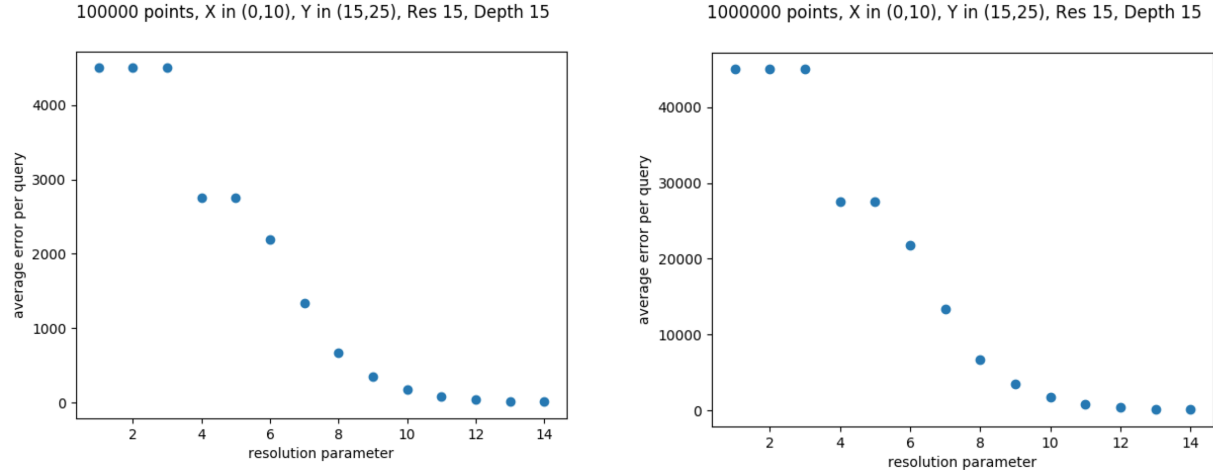


Figure 15: The relationship between resolution and error size in Nanocubes, when set at depth 15, for 100,000 points (left) and 1,000,000 points (right).

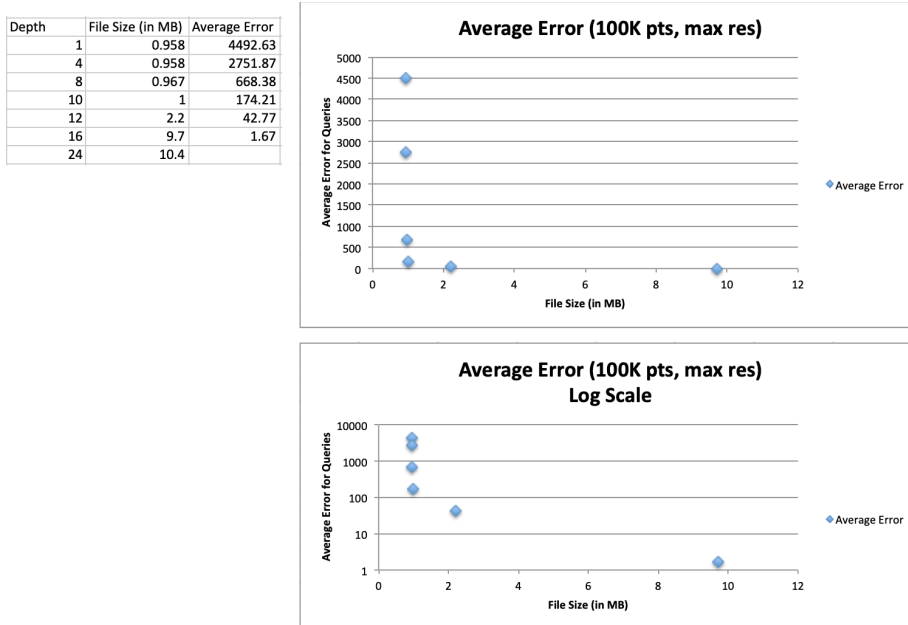


Figure 16: The relationship between file size (governed by depth) and error in Nanocubes, given maximum resolution.

Figure 15 shows the relationship between resolution and accuracy when fixed at a particular depth (in this case 15). Interestingly, resolution scales almost perfectly with the number of points in the dataset, and with resolution fixed, a 10x larger data set will have 10x as much error. Figure 16 shows the relationship between different file sizes (corresponding to different depths) and accuracy when fixed at maximum resolution. Notably, getting to the error of 174.21 can be done without much cost to model size, though near-zero error can be achieved with a model's file size is 10x as large.

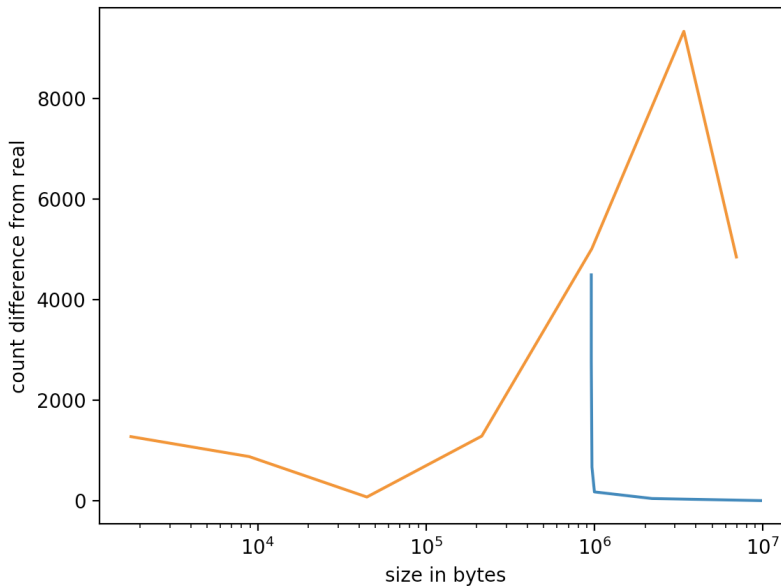


Figure 17: A side by side comparison of the learned index implementation (orange), with width 5 on the semi-correlated data, vs. the Nanocubes results from figure 16 (blue). Both used 100,000 data point data sets. Again, plots are not continuous, but lines were added between points for visual clarity.

These results do seem to bode well for the learned index implementation. As shown in figure 17, even in the worst result for the learned index model (semi-correlated data), a model with a properly chosen depth and width takes up about 200x less space than the Nanocubes model with a similar amount of error. Though comparisons with speed are impractical between a program written in C and a program written in Python, the speed for range search in the learned index algorithm is proportional to the size of the model (due to needing to traverse more nodes in larger models), so ultimately a comparison with a C-written version of the learned index algorithm seems promising. At the very least, even this initial prototype algorithm of the learned index method seems able to get comparable results with a modern tool for fast data lookups.

6 Conclusion and Future Work

This paper began by introducing the concept of learned indices and their potential impact on the task of quickly visualizing filtered data queries. Then, it talked about what related work in this area already existed, before explaining in more detail how learned indices work. Next, it explained the design and implementation of an algorithm for applying learned indices to range search, along with what kind of interesting problems arose. Finally, it explained the results of running the algorithm and how those results compared with modern alternatives. Though this research has been yielded interesting results, there also are several avenues for improving upon the algorithm, continuing comparison with the existing methods, and incorporating it within real visualization tools.

One next step would be to fix the density distortion effect where very large models (which thus have sub-models trained on a small number of data points) give very inaccurate results. Although most of the widths and depths used in practice would be smaller than what would produce these results, it would be useful to have a fully accurate program that produced a smoother graph for plotting models of various depths / widths.

Another next step would be to be able to compare the speed of the learned index implementation versus an existing technique such as Nanocubes. This would require converting the program into a faster language, such as C, because the Python version would be far too slow to make a fair comparison. Of course, properly being able to run the imMens and Hashcubes algorithms would also allow for this type of comparison.

Similarly, it could be interesting to try implementing queries for more than 2 dimensions, although comparisons would be more difficult and the data would be less accurate (due to needing to make even more of a “uniform distribution” assumption).

A more challenging avenue for future work would be to train the model not just on the indices of the x and y values, but also on the correlation between x and y within the dataset each sub-model was trained on. While this wouldn’t make much of a difference for large models that are using datasets where data is scattered fairly randomly once you “zoom in” enough, this would significantly improve accuracy when that is not the case. While it is unclear what exactly training on the correlation means in this case, this is probably the avenue of work that provides the biggest practical improvement to the algorithm.

Finally, the ultimate goal of this research is to incorporate into real world visualization tools. Thus, the final step of this work would be to integrate it with something like Vizdom [8], a data visualization tool being worked on in MIT’s CSAIL database group. It would be similarly interesting to see if it could be combined with existing techniques for big data visualization, rather than merely serving as an alternative.

This research supports the potential for vastly improving the both memory usage and performance of big data lookups. Work with learned index structures has offered an exciting look at unexplored directions for using machine learning to improve data structures, and hopefully in the future, we will see these methods utilized for data queries in huge visualizations.

7 Bibliography

- [1] Kraska, Tim, Beutel, Alex, Chi, Ed H., Dean, Jeffrey, & Polyzotis, Neoklis. 2017. The Case for Learned Index Structures. ArXiv e-prints, Dec., arXiv:1712.01208.
- [2] Z. Liu, B. Jiang, J. Heer, "immens: Real-time visual querying of big data", *Proceedings of the 15th Eurographics Conference on Visualization*, pp. 421-430, 2013.
- [3] L. Lins, J. T. Klosowski, C. Scheidegger, "Nanocubes for real-time exploration of spatiotemporal datasets", *IEEE Transactions on Visualization and Computer Graphics*, vol. 19, no. 12, pp. 2456-2465, Dec. 2013.
- [4] C. A. L. Pahins, S. A. Stephens, C. E. Scheidegger, J. L. D. Comba. "Hashedcubes: Simple, Low Memory, Real-Time Visual Exploration of Big Data", *IEEE Transactions on Visualization and Computer Graphics*, 23(1), 671-680. [7539326].
<https://doi.org/10.1109/TVCG.2016.2598624>
- [5] Sedlar, K. (2019). Account Provider Selection. [online] Github.mit.edu. Available at: <https://github.mit.edu/ksedlar/learnedindices> [Accessed 22 May 2019].
- [6] GeeksforGeeks. (2019). K Dimensional Tree | Set 1 (Search and Insert) - GeeksforGeeks. [online] Available at: <https://www.geeksforgeeks.org/k-dimensional-tree/> [Accessed 22 May 2019].
- [7] Docs.scipy.org. (2019). scipy.spatial.KDTree — SciPy v0.14.0 Reference Guide. [online] Available at: <https://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.spatial.KDTree.html> [Accessed 22 May 2019].

[8] Crotty, A., Galakatos, A., Zraggen, E., Binnig, C. and Kraska, T. (2015). Vizdom: Interactive Analytics through Pen and Touch. Proceedings of the VLDB Endowment, 8(12), pp.2024-2027.

[9] Anon. (2012). [image] Available at:

<https://commons.wikimedia.org/wiki/File:B-tree-definition.png> [Accessed 22 May 2019].

[10] A. Delesse (2014). [image] Available at:

https://commons.wikimedia.org/wiki/File:KD-Tree_part1.png [Accessed 23 May 2019].

[11] A. Delesse (2014). [image] Available at:

https://commons.wikimedia.org/wiki/File:KD-Tree_part2.png [Accessed 23 May 2019].