Introduction:

This lecture was given by Professor Tim Kraska of MIT’s Computer Science and Artificial Intelligence Laboratory. His research focuses on building systems for machine learning and using machine learning for systems. In this lecture, he discusses his work about learned index structures.

Basic data structures such as index structures, priority queues, and sorting algorithms are used in nearly all of the systems and applications. These basic structures have been integral to computer science since the very beginning, but recent advances suggest that machine learning may alter the way these structures are implemented and how they perform.

More specifically, existing index structures can be replaced with other types of models, known as learned indexes. The idea behind learned indexes is that a model can learn structure of lookup keys and use this signal to effectively predict the position or existence of records.

As a disclaimer, learning very rarely helps our algorithms now, even though there is a lot of potential. Here, we’re concerned with exactly how to make our data structures practically better using learning.

So here’s the idea: We’ll go through Knuth’s "Art of Computer Programming," and look for where there is potentially room for improvement using learning.

1 B-Trees

A B-Tree is a tree data structure that allows for searches and sequential accesses in logarithmic time. Because it is well suited for storage systems that read and write relatively large blocks of data, it is commonly used in databases and file systems.

B-trees are similar to Red Black trees, but with larger pages and greater than two children per node. The main idea behind a B-tree is to exploit locality in a tree structure. B-trees allow for this by placing multiple elements in each tree node. Because of this, B-trees ensure that few disk reads are needed to find the place where data is stored.

Lookup in a B-tree generally works as follows: Given a starting node, we use simply linear or binary search to find whether the element we are looking for is in the node. If it is not, we know which child pointer to follow from the current node to find it.
2 Learned B-Trees

Overview and Inspiration: Suppose we’re in a library, and we’re looking for a particular book. So, we’ll look through the catalog! First, you would consult a directory catalog and look for the letter "H". This would point you to another drawer and eventually you would get the index card of where the book is actually located. This is what a B-Tree does. In essence, a B-tree is like a red-black tree, but with > 2 branches, and is used by nearly all databases.

Alternatively, though, we could just ask the librarian. The librarian would tell you to "go up the stairs and turn right at the corner" to find your book. The library could be organized in a way such that all children’s books are in one area, all mysteries are in another, etc., making it easier for her to tell you these directions. The librarian has a mental model for where the books are. Additionally, an (idealized) librarian can also reorganize the library so that it better fits a mental model. What organization is best is going to depend on the normal pattern of use of the library.

Let’s consider an illustrative example: suppose we want to index the integers from 900 to 800M. While we can use a B-tree to get to a desired integer in this case, it probably makes more sense to do something like:

\[
data\text{\_array}[\text{lookup\_key} - 900].\]

Suppose we have even integers from 900 to 800M? We use essentially the same trick, and just divide by 2:

\[
data\text{\_array}[\{(\text{lookup\_key} - 900)/2\}].\]

To figure out how to look something up, we can get certain patterns and data and calculate how to get the offset. From this, we form a model.

An index is a model that takes a key as an input and predicts the position of the record. The B-tree is already a model. Given a key, it "predicts" the location of a value within a key-sorted set. We can replace this model with any other model that does position prediction. In other words, from a given key, you can be directed to a specific model.

Conceptually, a B-Tree maps a key to a page. (For simplicity, assume all pages are sorted and continuously stored in main memory.) So, to find something, we use the B-tree to map a key to

---

1 This may seem like a ridiculous, impossible case, but it really isn’t; dates are a good example of where this kind of thing comes up.

2 We actually only index the first key in a page.
a position, and then binary search within the min/max error. More specifically, from the page, you know that the key must be in this page. Therefore, your maximum error (or number of searches you would have to do within is this page) is the page size.

However, with a database, we only care about the data we already have on disc. A model that predicts the position given a key inside a sorted array effectively approximates the cumulative distribution function. The predicted position can be modeled as:

\[ p = F(\text{Key}) \times N, \]

where \( F(\text{key}) \) is the estimated cumulative distribution function for the data (\( P(X < \text{Key}) \)) and \( N \) is the total number of trees.

So, a B-Tree is really a model which approximately maps a key to a position; that is, modeling the CDF of the (observed) key distribution. That is, we can formulate a position-estimate as the product of the # of keys, and some modeling function \( F \) which acts on a key. An important distinguishing fact is we only care about the data we have stored, rather than the entire underlying distribution. In other words, we only care about the empirical CDF. Therefore, overfitting is not only fine but desirable.

Inverting the analogy, a B-Tree is really just a regression tree. They essentially learn this data distribution by building a regression tree. Using learned models, we can achieve adaptation to the tenant’s data. So, in the case of the 900-800M, we can model that! Additionally, suppose our database is for a warehouse, and our keys are dates. When looking at data from last year, users may only be interested in looking at monthly data. However, for this year, they may want to look at specific dates. In this case, as orders might be seasonal, a learned model could really shine compared to a standard B-Tree. And, our model can be whatever we want! In an ideal case, the model granularity can be adapted to the work load.

Another perspective on this is that what we really want is system customization. As customizing a system with expert knowledge is expensive and time-consuming (and therefore usually infeasible) we have come to accept B-Trees as an okay universal implementation. Using machine learning, we can get a bit closer to true system customization than we can feasibly do using traditional methods. We also can run our new algorithms on GPUs/FPGAs/TPUs, which may have different cost trade-offs. Traditional B-Trees are a bunch of if statements, which are really hard to parallelize, and this can make a big difference in performance. Non-B-Tree models can also potentially also be run more efficiently on CPUs, too.

\[ ^3 \text{So, Database people were really the first to do large-scale machine learning.} \]
So, does it work? A first attempt: As a first attempt, a two-layer fully connected neural network with 32 neurons per layer using ReLU activation functions was compared to a standard B-tree. The key or input features in this case were timestamps.

- State-of-the-art B-Tree takes 260 ns for a lookup.
- Tensorflow (basic perceptron, normal implementation, etc) takes > 80,000 ns.

So, this doesn’t look particularly promising at a first glance. We observe a couple of things:

- Traditional model architectures do not work, as they’re too expensive.
- ML Frameworks are not designed for nanosecond execution. (Tensorflow has a large overhead that makes it not well-suited for smaller models)
- Over-fitting actually is desirable here (which B-trees are really good at. They’re also cache-efficient)
- ML + System co-design is necessary.

In response to the results of the naive attempt, the LIF (learning index framework), recursive-model indexes (RMI), and standard-error-based search strategies were implemented.

Problem 1: The Learning Index Framework:

The Learning Index Framework (LIF) is an essential component of this whole idea of ML + System co-design, in that it allows learned index models to be run with very little overhead compared to traditional frameworks. LIF can either generate its own linear regression models (discussed more below) or extract weights from Tensorflow graphs and perform automatic code-generation to create an efficient model for learning indexes (by generating efficient index structures in C++ based on the model specification). LIF is still in the experimental phase.

Problem 2: Precision Gain per Node:

In a B-Tree with a 100 key page size, within even the first step we might go from 100M possibilities to 1M. So, within even a couple of layers, B-Trees work quite well and can handle considerable complexity.

Also of great importance is the last mile problem: even when data looks nice and smooth at the large scale, at the small scale, it may be very rough and random. In other words, this issue arises when models are more efficient to approximate the general shape of a CDF, but have problems being accurate at the individual data instance level. In other words, the data looks nice and smooth overall, but can appear irregular when zoomed in. B-Trees handle this quite well as they increase in resolution as they go down.

---

4Prof. Kraska mostly skipped over this in the lecture, but I think it is relevant for complete understanding, so I’m including it anyway.
Because of the last-mile search problem, reducing the prediction error to the order of hundreds from 100 million records is challenging. However, reducing the error from 100 million to 10,000 and from 10,000 to 100 is simpler.

So, our solution is the Recursive Model Index (RMI). It leverages the above idea by implementing a recursive regression model. An RMI is a tree of models, which direct to their children, which eventually direct to positions. In other words, this is a hierarchy of models, where a model at a given level in the hierarchy takes the key as an input and based on its output, picks another model. This repeats until the final stage where a position is predicted. This gives us great freedom to vary parameters of models at different levels, number of models, etc. One approach to this is to do it top-down. That is, train the top model first, and then break up its predictions and train the next models for each of its ranges. This turns out to be really fast to train.

Recursive model indexes do not have to be trees. It is possible for different models of one level to pick the same models in the next level. Furthermore, the predictions between the different levels are not position estimates. They simply point to another more “expertized” model that has better knowledge about a given collection of keys.

With these models, we find a trade-off between model depth and complexity: a more complex model may fit the data better, but also be more expensive to run. So, here we’ll just use linear regression models stacked two-deep. We can also vary the number of children models to trade between memory usage and accuracy.

How Does the Lookup-Code Look Like?

Model on stage 1: \( f(\text{key\_type\ key}) \)
Models on stage 2: \( g = [g_n(\text{key\_type\ key})] \)

Lookup code for 2-stage RMI:
\[
\text{pos\_estimate} = g[f(\text{key})](\text{key})
\]
\[
\text{pos} = \text{exp\_search}(\text{key}, \text{pos\_estimate}, \text{data})
\]

2-stage RMI with linear regression models
\[
\text{offset} = a + n*\text{key}
\]
\[
\text{weights2} = \text{weights\_stage2}[	ext{offset}]
\]
\[\text{pos\_estimate} = \text{weights2\_a} + \text{weights2\_b} \times \text{key}\]
\[\text{pos} = \exp\_\text{search}(\text{key}, \text{pos\_estimate}, \text{data})\]

The above code is really fast, as it’s just 2 multiplies, 2 additions, and 1 array lookup.

**Hybrid RMI**

Sometimes the models don’t have the gain that you want. So, you replace the models with a B-Tree, and then the worst-case performance is that of a B-Tree!

Recursive models allow us to build mixtures of models. Hybrid indexes allow us to bound the worst case performance of learned indexes to the performance of B-trees. This is because if the data distribution is exceptionally difficult to learn, all the models by default would become B-trees, making the entire model a B-tree.

Learned indexes have an advantage in that models can actually predict the position of the key, not just the region. To make the final search better, you can use searches other than binary search.  

**Search Strategies:**

**Model Biased Search:** This is similar to traditional binary search, but the middle value is the value predicted by the model.

**Biased Quaternary Search:** This takes three split points so that the hardware gets all three data points at once to achieve better performance if the data is not in the cache. Quaternary search can do better using RMI because we have a predicted position rather than a min key. An exponential search (assuming normally distributed error) may do even better.

So, to answer the question "does it have to be deep learning?" the answer is no. Maybe it could be, but it probably isn’t.

- Tensorflow (basic perceptron, normal implementation, etc) > 80,000 ns.
- State-Of-The-Art B-Tree takes 260 ns and 13 MB.
- Learned Index structure: 85 ns and 0.7 MB

Here, we can also play around with trade-offs of speed vs memory, etc.

Also, versus just about any comparable technique (beyond pure B-Trees), the learned index structure continues to win on both memory and speed. Of course, it is slower to generate than the B-Tree, but we usually spend a lot more time querying these structures than we do building them, so that really doesn’t matter much very often.

Another criticism is that of handling inserts. RMI assumes sorted keys, which could make insertions slow. However, the solution for RMI-based methods is exactly the same as the standard solution to this problem, which is Delta indexing. We put new inserts into a buffer, and then with new queries we simultaneously search the tree and buffer, and then eventually merge them. Retraining is also just a matter of fine-tuning, so it is not as expensive as training from scratch.

In fact, if our learned model can approximate the distribution for appends, then the insert complexity becomes O(1) rather than O(log n) because you can pre-allocate for the space required.
However, to do this, you need to generalize (rather than intentionally over-fitting.) More on this general topic is discussed in A-Tree: A Bounded Approximate Index Structure

Can we do more?

We distinguish 3 classes of learned algorithms:

- **CDF**: empirical CDF model of the data. (Like the B-tree).
- **Oracle**: prediction model (scheduling algorithms, for example.)
- **Full-Model**: Learning the entire algorithm / data structure.

Almost every data structure in Knuth’s "Art of Computer Programming" can probably improved with learning:

- **CDF-based**: trees, hashmaps, bloom filters, multidim indexes, sorting, range filters, dna searches
- **Oracle/full-model**: data cubes, scheduling, sql query optimizer

Let’s consider hash maps next.

### 3 Hash Maps

Here, our goal is to reduce conflicts.

Existing techniques of using learned models as a hash-function do not take advantage of the underlying data distribution.

Instead of a hash function, we can use a model which uses the empirical CDF which tries to minimize collisions. More specifically, we can learn the CDF of the key distribution in order to learn a better hash function. Instead of aiming to store records compactly or in order, we can scale the CDF by the targeted size $M$ of the Hash-map and use $h(K) = F(K) * M$

Using the same RMI (In-place chained Hash-map, 20-byte record, learned hash functions) slightly improves on both time and utilization. On the other hand, it’s a small enough improvement that it’s unlikely to replace traditional hash maps soon, though it could conceivably be used in distributed applications.

How well we benefit from the learned hash function depends on:

1. How accurately the model represents the observed CDF
2. Hash map architecture: how inserts, look-ups, conflicts are handled

We can also consider multidimensional hashing: How do we index several dimensions at once? We only have one order on disc, though. We return to the librarian analogy from the beginning, in that we want to reorganize to better fit learned modeling.
Let’s assume just 2 dimensions: amount and date. We could project down to date, and then all date queries are fast, but amount queries will be terrible. There are structures, such as an R-tree, which is the higher-dimensional analog of the B-tree, which can do this in theory, but in practice, they’re so terrible since there is too much overhead that they’re never used.

Let’s try a learned solution! Suppose we generate a grid over our data. This gives us a mean to create a more reasonable order on disc. (Let’s say, bottom to top within a column, left to right among columns). So, if you can estimate where something is (the probability mask) using a learned CDF, you can probably find it pretty fast. The CDF essentially tells you ”how much data can you skip over”. We don’t have any guarantees, but it is a good heuristic, as the data shows:

**Average query time:**

- Full Column Scan: 396 ms
- R-Tree: 591 ms
- Index-Organized Table: 12.6 ms
- Learned Multi-dimensional index: 0.37 ms

### 4 Sorting

For sorting, we can use our CDF model to pre-sort by generating an expected position of an element in the (finally) sorted list and putting it there. As our CDF is approximate, we can expect for our pre-sorted list to be imperfect in that there may be holes or wrong orderings in the pre-sorted list. So, after we’ve presorted the list approximately, then we can quickly compact the list (remove empty slots) and finish sorting using a traditional algorithm. If we use a monotonic model, elements won’t ever be wrongly ordered, so we don’t even need to sort at the end – we just need to compact the list.

In testing, it turns out that std::sort is slightly faster than quicksort, and that radix sort is several times faster than either of them, but the learned sort is faster than even radix sort for all but the smallest sorts (radix sort is limited to small data). Radix sort also does not have the same advantage when sorting strings.

---

5 Also, holes are easier to deal with than collisions that require lots of re-shuffling, so it’s better to over-allocate.
6 Timsort is the preferred sorting algorithm for nearly-sorted ranges, so it is used here.
5 Conclusions

Practically every standard algorithm has assumptions built in which could be improved through learning. For example, std::vector doubles in size whenever it runs out of space. Maybe we should automatically adjust the threshold based on past usage? That way we can save memory, or time spent copying, or perhaps even both! There is a great number of algorithms and data structures that could stand to benefit from these adaptive, learned algorithms.

And, of course, we are still at the beginning! GPUS/FPGAs, ML Compression, re-generation of IDs, and sparse vs dense storage are all unexplored opportunities which have great potential.

References

