Learned Index Structures
(joint work with Alex Beutel, Ed H. Chi, Jeffrey Dean, Neoklis Polyzotis)

Tim Kraska <kraska@mit.edu>
[Disclaimer: I am NOT talking on behalf of Google]
Comments on Social Media

“Machine Learning Just Ate Algorithms In One Large Bite….” [Christopher Manning, Professor at Stanford]
Disclaimer
Fundamental Building Blocks Of Data Management Systems
(or almost any system/application)
NO ASSUMPTIONS
Goal:

Index All Integers from 900 to 800M

B-Tree?
Goal:

Index All Integers from 900 to 800M

```
data_array[lookup_key - 900]
```
Goal:

**Index All Integers from 900 to 800M**

| 900 | 901 | 902 | 903 | 904 | 905 | 906 | 907 | 908 | 909 |

**Index All Even Integers from 900 to 800M**

| 900 | 902 | 904 | 906 | 908 | 910 | 912 | 914 | 916 | 918 |

\[
data\_array[(\text{lookup\_key} - 900) / 2]
\]
Still holds for other data distributions
Key Insight

Knowing the (empirical) Data Distribution allows for Instance-based Optimizations

(e.g., lookups: $O(\log n) \rightarrow O(1)$
storage: $O(n) \rightarrow O(1)$)
Building A System From Scratch For Every Use Case Is Not Economical
B-Tree As An Example
B-Tree As An Example

For the moment, focus on in-memory immutable B-Trees.

Assumptions

No Inserts

No Paging

will talk about those issues later.
Conceptually a B-Tree maps a key to a page

Assume: Data is stored in a continuous main memory region
Alternative View

*B-Tree maps a key to a position with a fixed min/max error*

For simplicity assume all pages are continuously stored in main memory.
A B-Tree Is A Model

Model

Sorted Array

pos - err_{\text{min}}
pos + err_{\text{max}}
A B-Tree Is A Model

Finding an item
1. Any model: key → pos
2. Binary search in
   \[ [\text{pos} - \text{err}_{\text{min}}, \text{pos} + \text{err}_{\text{max}}] \]

\text{err}_{\text{min}} \text{ and } \text{err}_{\text{max}} \text{ are known from the training process}
A B-Tree Is A Model

Model

Sorted Array

key

position

pos - err_{min}  pos + err_{max}

A CDF model

Pos-estimate = F(key) * #keys
The B-Tree is Also A Model
What Does This Mean
What Does This Mean

Database people were the first to do large scale machine learning :)}
Potential Advantages of Learned B-Tree Models

- **Smaller indexes** → less (main-memory) storage
- **Faster Lookups?**
- **More parallelism** → Sequential if-statements are exchanged for multiplications
- **Hardware accelerators** → Lower power, better $/compute....
- **Cheaper inserts?** → more on that later. For the moment, assume read-only
A First Attempt

- 200M web-server log records by timestamp-sorted
- 2 layer NN, 32 width, ReLU activated
- Prediction task: timestamp $\rightarrow$ position within sorted array
A First Attempt

Cache-Optimized B-Tree

≈250ns

???
A First Attempt

Cache-Optimized B-Tree

≈250ns

≈80,000ns
Reasons

**Problem I:** Tensorflow is designed for large models

**Problem II:** Search does not take advantage of the prediction

**Problem III:** B-Trees are cache-efficient

**Problem IV:** B-Trees are great for overfitting
Reasons

Problem I: Tensorflow is designed for large models

Problem II: Search does not take advantage of the prediction

Problem III: B-Trees are cache-efficient

Problem IV: B-Trees are great for overfitting
Solution:
Recursive Model Index (RMI)

\[ L_0 = \sum_{(x,y)} (f_0(x) - y)^2 \]

\[ L_\ell = \sum_{(x,y)} (f_\ell(\lfloor M_\ell f_{\ell-1}(x)/N \rfloor))(x) - y)^2 \]
How Does The Lookup-Code Look Like

Model on stage 1: \( f_0(key\_type\ key) \)

Models on stage two: \( f_1[] \)
(e.g., the first model in the second stage is \( f_1[0](key\_type\ key) \))

Lookup Code for a 2-stage RMI:

\[
\begin{align*}
\text{pos}_{\text{estimate}} & \leftarrow f_1[f_0(key)](key) \\
pos & \leftarrow \text{exp}\_\text{search}(key, \text{pos}_{\text{estimate}}, data);
\end{align*}
\]
How Does The Lookup-Code Look Like

Model on stage 1: \( f_0(key\_type\_key) \)

Models on stage two: \( f_1[] \)
(e.g., the first model in the second stage is \( f_1[0](key\_type\_key) \))

Lookup Code for a 2-stage RMI:

\[
\begin{align*}
pos\_estimate & \leftarrow f_1[f_0(key)](key) \\
pos & \leftarrow \text{exp_search}(key, pos\_estimate, data);
\end{align*}
\]

Operations with a 2-stage RMI with linear regression models

\[
\begin{align*}
\text{offset} & \leftarrow a + b \times key \\
\text{weights2} & \leftarrow \text{weights\_stage2}[\text{offset}] \\
\text{pos\_estimate} & \leftarrow \text{weights2}.a + \\
& \quad \text{weights2}.b \times key \\
pos & \leftarrow \text{exp\_search}(key, pos\_estimate, data);
\end{align*}
\]

2x multiplies
2x additions
1x array-lookup
Hybrid RMI

Worst-Case Performance is the one of a B-Tree
Does it have to be
Does It Work?

200M records of map data (e.g., restaurant locations). index on longitude
Intel-E5 CPU with 32GB RAM **without** GPU/TPUs **No Special SIMD optimization** (there is a lot of potential)

<table>
<thead>
<tr>
<th>Type</th>
<th>Config</th>
<th>Lookup time</th>
<th>Speedup vs. BTree</th>
<th>Size (MB)</th>
<th>Size vs. Btree</th>
</tr>
</thead>
<tbody>
<tr>
<td>BTree</td>
<td>page size: 128</td>
<td>260 ns</td>
<td>1.0X</td>
<td>12.98 MB</td>
<td>1.0X</td>
</tr>
</tbody>
</table>
### Does It Work?

200M records of map data (e.g., restaurant locations). Index on longitude

Intel-E5 CPU with 32GB RAM **without** GPU/TPUs **No Special SIMD optimization** (there is a lot of potential)

<table>
<thead>
<tr>
<th>Type</th>
<th>Config</th>
<th>Lookup time</th>
<th>Speedup vs. BTree</th>
<th>Size (MB)</th>
<th>Size vs. Btree</th>
</tr>
</thead>
<tbody>
<tr>
<td>BTree</td>
<td>page size: 128</td>
<td>260 ns</td>
<td>1.0X</td>
<td>12.98 MB</td>
<td>1.0X</td>
</tr>
<tr>
<td>Learned index</td>
<td>2nd stage size: 10000</td>
<td>222 ns</td>
<td>1.17X</td>
<td>0.15 MB</td>
<td>0.01X</td>
</tr>
<tr>
<td>Learned index</td>
<td>2nd stage size: 50000</td>
<td>162 ns</td>
<td>1.60X</td>
<td>0.76 MB</td>
<td>0.05X</td>
</tr>
<tr>
<td>Learned index</td>
<td>2nd stage size: 100000</td>
<td>144 ns</td>
<td>1.67X</td>
<td>1.53 MB</td>
<td>0.12X</td>
</tr>
<tr>
<td>Learned index</td>
<td>2nd stage size: 200000</td>
<td>126 ns</td>
<td>2.06X</td>
<td>3.05 MB</td>
<td>0.23X</td>
</tr>
</tbody>
</table>

60% faster at 1/20th the space, or 17% faster at 1/100th the space
You Might Have Seen Certain Blog Posts
Big thanks to Thomas Neumann as his blog post actually helped us a lot to improve our experiment section.
What About Our Assumptions

• Updates and Inserts\(^1\)
• Paging

Fundamental Algorithms & Data Structures

Join

Sorting

Tree

Hash-Map

Bloom-Filter

Range-Filter

Priority Queue

Scheduling

Cache Policy
Multi-Dimensional Index
Problems with (R-Tree / KD-Tree)
Machine Learning Is Good For Multi-Dimensional Data
There is Only 1-Dim Order On Disk*

*Sure the disk is more complicated, but the API and the scanning of records is usually 1-dim
Example
Equal Importance
Is It PCA?
Most Queries Are about **Order Amount**
Most Queries Are about **Order Zip Code**
Can I mix the projections?
Can I mix the projections?
2 Models

Projector

Locator

0 1 2 3
The projector

1. Root node define a primary direction
2. Project points on the root
3. Partition the space
4. Define directions for each sub-space
k. Recurse for any depth

This is an RMI Model not a BTree
After Projection Locator is a Normal BTree RMI
Early results (1M points, synthetic)

• ~200ns for point queries
• ~2x speed, ~10x space vs R-Trees
Future Work

Join

Sorting

Tree

Hash-Map

Bloom-Filter

Range-Filter

Priority Queue

Scheduling

Cache Policy

......
How Would You Design Your Algorithms/Data Structure If You Have a Model for the Empirical Data Distribution?

The Power of Continuous Functions
Learned Adaptation
Big Potential For TPUs/GPUs
Can Lower the Complexity Class

data_array[(lookup_key - 900)]
Warning
Not An Almighty Solution
Data System for AI Lab DSAIL@CSAIL

Data Systems for AI for Data Systems

Google
Microsoft
Intel
• A new approach to indexing
• Framework to rethink many existing data structures/algorithms
• Under certain conditions, it might allow to change the complexity class of data structures
• The idea might have implications within and outside of DBMS
Related Work

• **Succinct Data Structures** ➔ Most related, but succinct data structures usually are carefully, manually tuned for each use case
• **B-Trees with Interpolation search** ➔ Arbitrary worst-case performance
• **Perfect Hashing** ➔ Connection to our Hash-Map approach, but they usually increase in size with N
• **Mixture of Expert Models** ➔ Used as part of our solution
• **Adaptive Data Structures / Cracking** ➔ orthogonal problem
• **Local Sensitive Hashing (LSH) (e.g., learned by NN)** ➔ Has nothing to do with Learned Structures
Thank you!