Comments on Social Media

“Machine Learning Just Ate Algorithms In One Large Bite….” [Christopher Manning, Professor at Stanford]
Fundamental Building Blocks

- Sorting
- B-Tree
- Hash-Map
- Scheduling
- Join
- Priority Queue
- Bloom Filter
- Range Filter
- Caching
Databases as an Example:

B-Trees

HDD HDD HDD HDD HDD
The Librarian
Key

Model
Fundamental Algorithms & Data Structures

- Join
- Sorting
- Tree
- Hash-Map
- Bloom-Filter

- Range-Filter
- Priority Queue
- Scheduling
- Cache Policy
Not convinced yet?
Another Example:

Index All Integers from 900 to 800M

```
900  901  902  903  904  905  906  907  908  909  ...  800M
```

B-Tree?
A More Concrete Example:

Index All Integers from 900 to 800M

| 900 | 901 | 902 | 903 | 904 | 905 | 906 | 907 | 908 | 909 | ... | 800M |

```
data_array[lookup_key - 900]
```
Goal:
Index All Integers from 900 to 800M

| 900 | 901 | 902 | 903 | 904 | 905 | 906 | 907 | 908 | 909 | ... | 800M |

Index All Even Integers from 900 to 800M

| 900 | 902 | 904 | 906 | 908 | 910 | 912 | 914 | 916 | 918 | ... | 800M |

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

Traditional data structures (typically) make no assumptions about the data

But knowing the data distribution might allow for significant performance gains and might even change the complexity of data structures (e.g., $O(\log n) \rightarrow O(1)$ for lookups or $O(n) \rightarrow O(1)$ for storage)
Building A System From Scratch For Every Use Case Is Not Economical
Conceptually a B-Tree maps a key to a page.

For simplicity assume all pages are continuously stored in main memory.
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

key

position

Sorted Array

pos

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

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

A form of a regression model

key → pos is equivalent of modeling the CDF of the (observed) key distribution:
Pos-estimate = P(X ≤ key) * #keys
A B-Tree Is A Model

Pos-estimate = F(key) * #keys
B-Trees Are Regression Trees

![Diagram of B-Tree and Sorted Array]

- Key
- Position
- Sorted Array
- B-Tree
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

TensorFlow
A First Attempt

Cache-Optimized B-Tree

≈ 250ns

TensorFlow

≈ 80,000ns
Reasons

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

**Problem II:** B-Trees are great for overfitting

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

**Problem IV:** Search does not take advantage of the prediction
Problem I: The Learning Index Framework (LIF)

• An index synthesis system
• Given an index configuration generate the best possible code
• Uses ideas from Tupleware [VLDB15]
• Simple models are trained “on-the-fly”, whereas for complex models we use Tensorflow and extract weights afterwards (i.e., no Tensorflow during inference time)
• Best index configuration is found using auto-tuning (e.g., see TuPAQ [SOCC15])
Problem II + III: Precision Gain per Node

Index over 100M records. Page-size: 100

Precision Gain: 100M \(\rightarrow\) 1M
(Min/Max-Error: 1M)

Precision Gain: 1M \(\rightarrow\) 10k

Precision Gain: 10k \(\rightarrow\) 100

100M records (i.e., 1M pages)
The Last Mile Problem
Solution:
Recursive Model Index (RMI)
How Does The Lookup-Code Look Like

Model on stage 1: \( f_0(\text{key\_type } \text{key}) \)
Models on stage two: \( f_1[] \) (e.g., the first model in the second stage is \( f_1[0](\text{key\_type } \text{key}) \))

Lookup Code:

```c
pos_estimate ← f1[f0(key)](key)
pos ← exp_search(key, pos_estimate, data);
```
How Does The Lookup-Code Look Like

Model on stage 1: \( f_0(\text{key\_type\ key}) \)

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

Lookup Code for a 2-stage RMI:

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

Operations with a 2-stage RMI with linear regression models

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

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

Worst-Case Performance is the one of a B-Tree
Problem IV: Min-/Max-Error vs Average Error
Binary Search

- **Actual Position**
- **Predicted Position**

Positions:
- **Left**
- **Middle**
- **Right**
Binary Search
Binary Search
Quaternary Search

- Actual Position
- Predicted Position

0...N

Left Q2 Right
Quaternary Search
Quaternary Search

0

N

Predicted Position

Actual Position

Left Q1 Q2 Q3 Right
Exponential Search
Does it have to be

DEEP LEARNING
## 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
My Own Comparison
A Comparison To ARTful Indexes (Radix-Tree)

Viktor Leis, Alfons Kemper, Thomas Neumann: The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases. ICDE 2013

Experimental setup:

- **Dense**: continuous keys from 0 to 256M
- **Sparse**: 256M keys where each bit is equally likely 0 or 1.
A Comparison To ARTful Indexes (Radix-Tree)

Viktor Leis, Alfons Kemper, Thomas Neumann: The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases. ICDE 2013

Experimental setup: continuous keys from 0 to 256M

Reported lookup throughput: 10M/s ≈ 100ns

Size: not measured, but paper says overhead of ≈8 Bytes per key (dense, best case): 256M * 8 Byte ≈ 1953MB

(1) Numbers from the paper
Learned Index

Generate Code:

```java
Record lookup(key) {
    return data[0 + 1 * key];
}
```
Learned Index

Generate Code:

```java
Record lookup(key) {
    return data[key];
}
```
Learned Index

Generate Code:
```java
Record lookup(key) {
    return data[key];
}
```

Lookup Latency: $10\text{ns}$ (learned index) vs $100\text{ns}^*$ (ARTfull) or one-order-of-magnitude better

Space: $0\text{MB}$ vs $1953\text{MB}$

Infinitely better :)
What about Updates and Inserts?
What about Updates and Inserts?

Alex Galakatos, Michael Markovitch, Carsten Binnig, Rodrigo Fonseca, Tim Kraska:
A–Tree: A Bounded Approximate Index Structure
https://arxiv.org/abs/1801.10207
The Simple Approach: Delta Indexing

Training a simple Multi-Variate Regression Model Can be done in one pass over the data
Leverage the Distribution
If the Learned Model Can Generalize to Inserts
Insert complexity is $O(1)$ not $O(\log N)$
Updates/Inserts

• Less beneficial as the data still has to be stored sorted
• Idea: Leave space in the array where more updates/inserts are expected
• Can also be done with traditional trees.
• But, the error of learned indexes should increase with $\sqrt{N}$ per node in RMI whereas traditional indexes with $N$
Still at the Beginning!

- Can we provide bounds for inserts?
- When to retrain?
- How to retrain models on the fly?
- ...

Fundamental Algorithms & Data Structures

Join

Sorting

Tree

Hash-Map

Bloom-Filter

Range-Filter

Priority Queue

Scheduling

Cache Policy

......
Fundamental Algorithms & Data Structures

Join  Sorting  Tree  Hash-Map  Bloom-Filter

Range-Filter  Priority Queue  Scheduling  Cache Policy
Hash Map

Goal: Reduce Conflicts
### Hash Map - Results

<table>
<thead>
<tr>
<th></th>
<th>% Conflicts Hash Map</th>
<th>% Conflicts Model</th>
<th>Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Map Data</td>
<td>35.3%</td>
<td>07.9%</td>
<td>77.5%</td>
</tr>
<tr>
<td>Web Data</td>
<td>35.3%</td>
<td>24.7%</td>
<td>30.0%</td>
</tr>
<tr>
<td>Log Normal</td>
<td>35.4%</td>
<td>25.9%</td>
<td>26.7%</td>
</tr>
</tbody>
</table>

**25% - 70% Reduction in Hash-Map Conflicts**
You Might Have Seen Certain Blog Posts
Independent of Hash-Map Architecture
# Hash Map – Example Results

<table>
<thead>
<tr>
<th>Type</th>
<th>Time (ns)</th>
<th>Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stanford AVX Cuckoo, 4 Byte value</td>
<td>31ns</td>
<td>99%</td>
</tr>
<tr>
<td>Stanford AVX Cuckoo, 20 Byte record - Standard Hash</td>
<td>43ns</td>
<td>99%</td>
</tr>
<tr>
<td>Commercial Cuckoo, 20Byte record - Standard Hash</td>
<td>90ns</td>
<td>95%</td>
</tr>
<tr>
<td>In-place chained Hash-map, 20Byte record, <strong>learned hash functions</strong></td>
<td>35ns</td>
<td>100%</td>
</tr>
</tbody>
</table>
Fundamental Algorithms & Data Structures

Join

Sorting

Tree

Hash-Map

Bloom-Filter

Range-Filter

Priority Queue

Scheduling

Cache Policy
Fundamental Algorithms & Data Structures

- Join
- Sorting
- Tree
- Hash-Map
- Bloom-Filter
- Range-Filter
- Priority Queue
- Scheduling
- Cache Policy
Is This **Key** In My Set?   Is This **Key** In My Set?

Bloom Filter- Approach 1

Maybe Yes   No   Maybe Yes   No

36% Space Improvement over Bloom Filter at Same False Positive Rate
Bloom Filter - Approach 2 (Future Work)
Fundamental Algorithms & Data Structures

- Join
- Sorting
- Tree
- Hash-Map
- Bloom-Filter
- Range-Filter
- Priority Queue
- Scheduling
- Cache Policy
Future Work

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

- Join
- Sorting
- Tree
- Hash-Map
- Bloom-Filter
- Range-Filter
- Priority Queue
- Scheduling
- Cache Policy
Future work: Multi-Dim Indexes
Future work: Data Cubes
Other Database Components

- Cardinality Estimation
- Cost Model
- Query Scheduling
- Storage Layout
- Query Optimizer
- ...

How Would You Design Your Algorithms/Data Structure If You Have a Model for the Empirical Data Distribution?
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
Local Sensitive Hashing (LSH)

Thanks Alkis for the analogy
How Would You Design Your Algorithms/Data Structure If You Have a Model for the Empirical Data Distribution?
Adapts To Your Data
Big Potential For TPUs/GPUs
Can Lower the Complexity Class

data_array[(lookup_key - 900)]
Warning
Not An Almighty Solution
Data Systems for AI for Data Systems

Founding Sponsors
Google
Microsoft
Intel

ML Faculty

System Faculty

Research Area
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 database systems

Technical Report:
Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, Neoklis Polyzotis: The Case for Learned Index Structures