The Case for Learned Index Structures

John Yang | CS 294 | Feb 11, 2019
Outline

Background 3
Problem 5
Success Metrics 7
B-Trees 8
RM-Index 13
Hashmaps 21
Bloom Filters 25
Conclusions 28
Background

The State of System Design Today

Data Structures and Algorithms are

- General Purpose, “One Size Fits All”
- Assume nothing about data distribution
- Oblivious towards the nature of data
Background

Data Structure and Algorithm Domains

Join  Sort  Tree  Scheduling  Cache  Bloom Filter
Problem

“One Size Does Not Fit All”

1. Traditional Data Structures do not account for the nature of data
   a. Scales poorly with more data
   b. Do not take advantage of common patterns in real world data
   c. Suboptimal edge cases can fail with increases in computation time by orders of magnitude.

2. Learn the Data Distribution for Time, Space, Performance Improvements
   a. Scale with complexity, not size
   b. Machine Learning, Reinforcement Learning, and Neural Nets can replace, complement, improve existing heuristics and system operations.
Problem

Idea: Use Machine Learning Models to Learn Different Data Distributions and Create Adaptive Structures and Algorithms

In some sense, indexes are already models, so it’s worth exploring transitioning from rigid index structures to learned, more flexible models.
Success Metrics

Traditional Systems Metrics
- I/O Count
- Space + Memory Requirements
- Query + Lookup Time

Model Metrics
- Size of the Model
- Amount of Overhead
- Number of Training Iterations
- Amount of Training Data
B-Trees | Range Index

CPU Cache

Main Memory

Disk / Main Memory
Key Innovations

B-Trees as a Modeling Problem

- Smaller Index
- Faster Lookup
- More Parallelism
- Cheaper Insertion
- Hardware Acceleration
Key Innovations

B-Trees as a Cumulative Distribution Function

Predicted Position = \( P(x \leq \text{key}) \times \# \text{ of Keys} \)

What is the distribution of data?

Where is it coming from?

How does it look?
Key Innovations

Tensorflow Implementation of B-Tree Lookup

- 200M Web Server Log Records sorted by Timestamp
- 2 Layer Neural Network, 32-width fully connected, ReLU Activation Function
- Given the timestamp, predict the position!

Results:
- Tensorflow: 1250 Predictions / Sec ~ 80000 ns Lookup
- B-Trees: 300 ns Lookup, 900 ns Binary Search across entire data set
Key Results & Takeaways

1. Tensorflow is designed for running larger models. Python paired with significant invocation overhead equals slower execution. When is a model driven approach more appropriate than traditional indexes?

2. B Trees better at overfitting, more accurate at individual data instance level. How does a model solve the “last mile” problem - Narrow down a data set from large range to specific instance? (Overfitting?)

3. B Trees are cache efficient, keep relevant nodes and operations close by. On the other hand, neural nets require
Learning Index Framework (LIF)

Problem: How to better investigate different models for index replacement or optimization.

Solution: Learning Index Framework

- Index Synthesis System
- Given an Index => Generate, optimize, and test different index configurations
- For simple models (e.g. linear regression), learns values on the fly
- For complex models, extract model weights and generate C++ index structure
Recursive Model Index (RMI)

Problem: Accuracy of Last Mile Search

Solution: Recursive Regression Model
- Idea: Reduce error across a hierarchy of models focusing on subsets of data

\[
L_\ell = \sum_{(x,y)} \left( f_\ell^\left(\left\lfloor M_{\ell-1}(x)/N \right\rfloor\right)(x) - y \right)^2
\]

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

1.5 Million Records, ~60 Cycles
24K Records, 120 Cycles
Hybrid Recursive Model Index

Problem: Specific Data at the bottom of RMI may be harder to learn

Solution: Combine different models at different layers of RMI
- Neural Nets at the top
- Simple Linear Regression on the bottom
- Fall back on B-Trees if data is particularly difficult to learn
Search Strategies

- Binary Search
- Biased Quaternary Search
- Exponential Search
Experiments with LIF, RIM

Four Different Datasets

- Timestamps from weblogs (200 M)
- Longitudes from Maps (200 M)
- Data sample from log-normal distribution (190 M)
- String Document IDs (10 M, non linear!)
# Experiment Results

## Integer Datasets

<table>
<thead>
<tr>
<th>Type</th>
<th>Config</th>
<th>Map Data</th>
<th>Web Data</th>
<th>Log-Normal Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Size (MB)</td>
<td>Lookup (ns)</td>
<td>Model (ns)</td>
</tr>
<tr>
<td>Btree</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>page size:</td>
<td>32</td>
<td>52.45 (4.00x)</td>
<td>274 (0.97x)</td>
<td>198 (72.3%)</td>
</tr>
<tr>
<td>page size:</td>
<td>64</td>
<td>26.23 (2.00x)</td>
<td>277 (0.96x)</td>
<td>172 (62.0%)</td>
</tr>
<tr>
<td>page size:</td>
<td>128</td>
<td>13.11 (1.00x)</td>
<td>265 (1.00x)</td>
<td>134 (50.8%)</td>
</tr>
<tr>
<td>page size:</td>
<td>256</td>
<td>6.56 (0.50x)</td>
<td>267 (0.99x)</td>
<td>114 (42.7%)</td>
</tr>
<tr>
<td>page size:</td>
<td>512</td>
<td>3.28 (0.25x)</td>
<td>286 (0.93x)</td>
<td>101 (35.3%)</td>
</tr>
<tr>
<td>Learned</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Index</td>
<td></td>
<td>0.15 (0.01x)</td>
<td>98 (2.70x)</td>
<td>31 (31.6%)</td>
</tr>
<tr>
<td>2nd stage models: 10k</td>
<td></td>
<td>0.76 (0.06x)</td>
<td>85 (3.11x)</td>
<td>39 (45.9%)</td>
</tr>
<tr>
<td>2nd stage models: 50k</td>
<td></td>
<td>1.53 (0.12x)</td>
<td>82 (3.21x)</td>
<td>41 (50.2%)</td>
</tr>
<tr>
<td>2nd stage models: 100k</td>
<td></td>
<td>3.05 (0.23x)</td>
<td>86 (3.08x)</td>
<td>50 (58.1%)</td>
</tr>
</tbody>
</table>

Figure 4: Learned Index vs B-Tree
### Experiment Results

#### String Datasets

<table>
<thead>
<tr>
<th>Config</th>
<th>Size (MB)</th>
<th>Lookup (ns)</th>
<th>Model (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Btree</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>page size: 32</td>
<td>13.11 (4.00x)</td>
<td>1247 (1.03x)</td>
<td>643 (52%)</td>
</tr>
<tr>
<td>page size: 64</td>
<td>6.56 (2.00x)</td>
<td>1280 (1.01x)</td>
<td>500 (39%)</td>
</tr>
<tr>
<td>page size: 128</td>
<td>3.28 (1.00x)</td>
<td>1288 (1.00x)</td>
<td>377 (29%)</td>
</tr>
<tr>
<td>page size: 256</td>
<td>1.64 (0.50x)</td>
<td>1398 (0.92x)</td>
<td>330 (24%)</td>
</tr>
<tr>
<td><strong>Learned Index</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 hidden layer</td>
<td>1.22 (0.37x)</td>
<td>1605 (0.80x)</td>
<td>503 (31%)</td>
</tr>
<tr>
<td>2 hidden layers</td>
<td>2.26 (0.69x)</td>
<td>1660 (0.78x)</td>
<td>598 (36%)</td>
</tr>
<tr>
<td><strong>Hybrid Index</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t=128, 1 hidden layer</td>
<td>1.67 (0.51x)</td>
<td>1397 (0.92x)</td>
<td>472 (34%)</td>
</tr>
<tr>
<td>t=128, 2 hidden layers</td>
<td>2.33 (0.71x)</td>
<td>1620 (0.80x)</td>
<td>591 (36%)</td>
</tr>
<tr>
<td>t= 64, 1 hidden layer</td>
<td>2.50 (0.76x)</td>
<td>1220 (1.06x)</td>
<td>440 (36%)</td>
</tr>
<tr>
<td>t= 64, 2 hidden layers</td>
<td>2.79 (0.85x)</td>
<td>1447 (0.89x)</td>
<td>556 (38%)</td>
</tr>
<tr>
<td><strong>Learned QS</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 hidden layer</td>
<td>1.22 (0.37x)</td>
<td>1155 (1.12x)</td>
<td>496 (43%)</td>
</tr>
</tbody>
</table>

Figure 6: String data: Learned Index vs B-Tree
# Experiment Results

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Memory Savings</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Server Logs (Timestamps)</td>
<td>88%</td>
<td>1.88x</td>
</tr>
<tr>
<td>Longitudes</td>
<td>99%</td>
<td>2.7x</td>
</tr>
<tr>
<td>Synthetic Log Normal Data</td>
<td>88%</td>
<td>1.8x</td>
</tr>
<tr>
<td>Strings (Document IDs)</td>
<td>63%</td>
<td>1.1x</td>
</tr>
<tr>
<td></td>
<td>Lookup Table w/ AVX search</td>
<td>FAST</td>
</tr>
<tr>
<td>----------</td>
<td>-----------------------------</td>
<td>------</td>
</tr>
<tr>
<td><strong>Time</strong></td>
<td>199 ns</td>
<td>189 ns</td>
</tr>
<tr>
<td><strong>Size</strong></td>
<td>16.3 MB</td>
<td>1024 MB</td>
</tr>
</tbody>
</table>

Figure 5: Alternative Baselines
Hashmaps | Point Index

Keys
John Smith
Lisa Smith
Sam Doe
Sandra Dee
Ted Baker

Buckets
000 X
001
002 X
151 X
152
153
154 X
253 X
254
255 X

Entries
X Lisa Smith 521-8976
X John Smith 521-1234
X Sandra Dee 521-9655
X Ted Baker 418-4165
X Sam Doe 521-5030
Key Innovations

Hashmaps as a Model

(a) Traditional Hash-Map
(b) Learned Hash-Map

Idea: Use Learned CDF as the Hash Function

Perfect CDF model should have zero collisions

Independent of type of hashmap
## Key Results & Takeaways

<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>

**Figure 8: Reduction of Conflicts**

Control / Base: MurmurHash3-like Hash Function  
Model: 2-Stage RMI Models, 100k models on 2nd stage, no hidden layers
**Key Results & Takeaways**

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Slots</th>
<th>Hash Type</th>
<th>Time (ns)</th>
<th>Empty Slots</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Map</strong></td>
<td>75%</td>
<td>Model Hash</td>
<td>67</td>
<td>0.18GB</td>
<td>0.21x</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Random Has</td>
<td>52</td>
<td>0.84GB</td>
<td></td>
</tr>
<tr>
<td>100%</td>
<td></td>
<td>Model Hash</td>
<td>53</td>
<td>0.35GB</td>
<td>0.22x</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Random Has</td>
<td>48</td>
<td>1.58GB</td>
<td></td>
</tr>
<tr>
<td>125%</td>
<td></td>
<td>Model Hash</td>
<td>64</td>
<td>1.47GB</td>
<td>0.60x</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Random Has</td>
<td>49</td>
<td>2.43GB</td>
<td></td>
</tr>
<tr>
<td><strong>Web</strong></td>
<td>75%</td>
<td>Model Hash</td>
<td>78</td>
<td>0.64GB</td>
<td>0.77x</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Random Has</td>
<td>53</td>
<td>0.83GB</td>
<td></td>
</tr>
<tr>
<td>100%</td>
<td></td>
<td>Model Hash</td>
<td>63</td>
<td>1.09GB</td>
<td>0.70x</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Random Has</td>
<td>50</td>
<td>1.56GB</td>
<td></td>
</tr>
<tr>
<td>125%</td>
<td></td>
<td>Model Hash</td>
<td>77</td>
<td>2.20GB</td>
<td>0.91x</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Random Has</td>
<td>50</td>
<td>2.41GB</td>
<td></td>
</tr>
<tr>
<td><strong>Log Normal</strong></td>
<td>75%</td>
<td>Model Hash</td>
<td>79</td>
<td>0.63GB</td>
<td>0.79x</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Random Has</td>
<td>52</td>
<td>0.80GB</td>
<td></td>
</tr>
<tr>
<td>100%</td>
<td></td>
<td>Model Hash</td>
<td>66</td>
<td>1.10GB</td>
<td>0.73x</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Random Has</td>
<td>46</td>
<td>1.50GB</td>
<td></td>
</tr>
<tr>
<td>125%</td>
<td></td>
<td>Model Hash</td>
<td>77</td>
<td>2.16GB</td>
<td>0.94x</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Random Has</td>
<td>46</td>
<td>2.31GB</td>
<td></td>
</tr>
</tbody>
</table>

**Conclusion:** Actual benefits from reducing conflicts depends on a variety of factors (e.g. architecture, payload), complexity not guaranteed to pay off

**Small Payloads - Traditional Cuckoo hashing works best**

**Larger Payloads + Distributed Settings**
- Increased latency okay when considering cache miss, conflict costs

Figure 11: Model vs Random Hash-map
Bloom Filter | Existence Index

FILTER

Do you have 'key1'?  
Filter: No

Do you have 'key2'?  
Filter: Yes
Yes: here is key2

Do you have 'key3'?  
Filter: False Positive

STORAGE

Storage: Yes
necessary disk access
Yes: here is key2
unnecessary disk access

Storage: No

Key Innovations

Bloom Filters as Binary Classification

Idea: Binary Classification

Problem: False Negatives

Solution: Hybrid Model / Bloom Filter

Spillover Bloom Filter (Misclassified)
Key Results & Takeaways

Task: Determine if URLs are “good”. If bad, warn about phishing / hacked
Built with RNN, $W$ is number of neurons, $E$ is embedding size

36% Reduction in Memory
Future Implications & Research Potentials

- Trees: Insertion, B-Tree Variants, when should a model be retrained?
- Joins: Estimating Cardinality (Next Paper!)
- Sorting: CDF provides approximate sort order, use sandwiched / hybrid approach alongside traditional sorting algorithms
- Modeling Data Structures or Datasets with Non-Linear Behavior
- Neural Networks for Multidimensional Indexed Structures: Creating mapping function to capture more complicated, multidimensional keys (3+ Dimensions)
- TPUs/GPUs: Run learning indices on vector processors, replace if-then statements with multiplexers.
Future Implications & Research Areas

Conclusions

- Benefits of learned indexes are dependent upon the usage and architecture of the data structure or algorithm in question
- Don’t necessarily replace, use traditional indexes alongside learned models

Questions

- What factors can help guide the transition from a data structure or an algorithm to an appropriate model?
- How can we effectively scale accuracy with size?
- What are some principles for designing hybrid models?