Slide Credits:

Assembled by Team Alpha Nebula (Yash Narendra Saraf, Mohammud Umair, Deepak Ranjan)

1. For the presentation, we used the following slide deck available on AI-Sys Spring 2019 page of UC Berkeley.
   - Course Page: https://ucbrise.github.io/cs294-ai-sys-sp19/#
   - Presentation Link: https://ucbrise.github.io/cs294-ai-sys-sp19/assets/lectures/lec05/learnedIndexes.pdf
2. We made some minor changes to the deck for our presentation. Following is the presentation deck attached - Name: AlphaNebulaDeck.pdf
3. Apart from the above presentation deck, we also used the author Prof Tim Kraska's presentation deck which we requested from the author. Link to the presentation deck:https://t.co/oh5yimy2er?amp=1
4. Also, we referred to the author's Stanford Presentation Video to prepare slides- Link: https://www.youtube.com/watch?v=NaqJO7rrXy0&t=2994s
5. The main reference was the original paper: https://dl-acm-org.gate.lib.buffalo.edu/citation.cfm?id=3196909

The Case for Learned Index Structures

John Yang | CS 294 | Feb 11, 2019

Outline

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

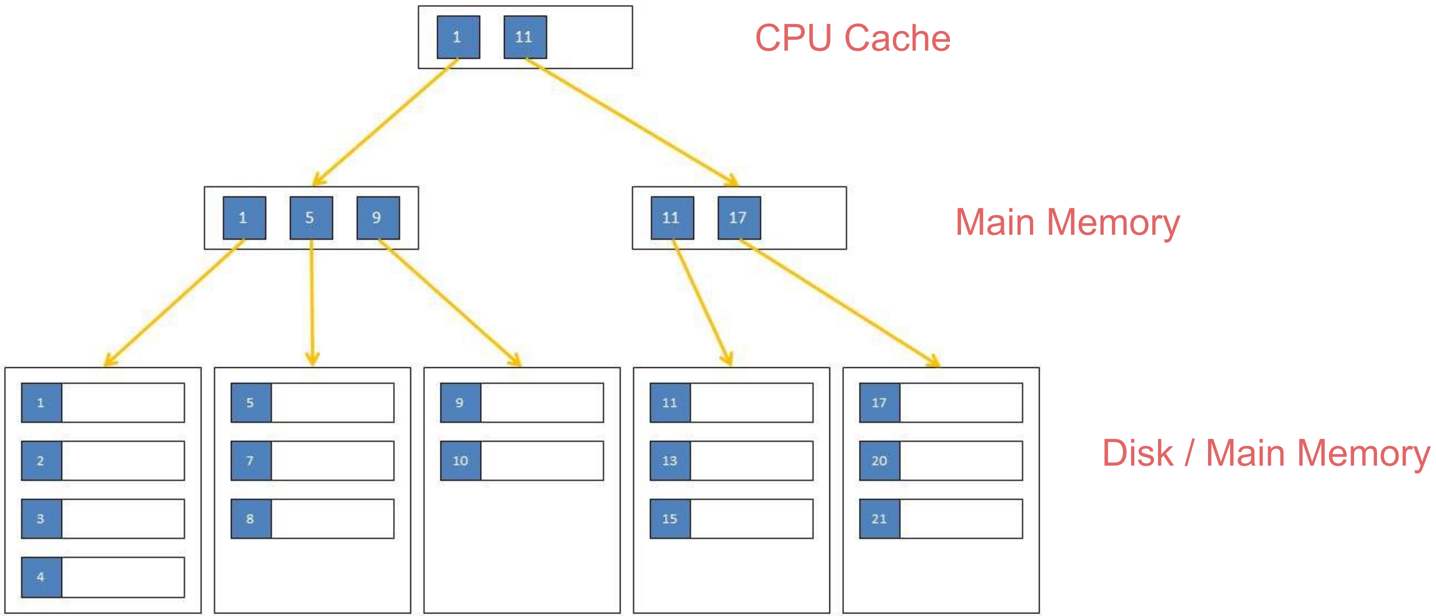
Traditional Systems Metrics

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

Model Metrics

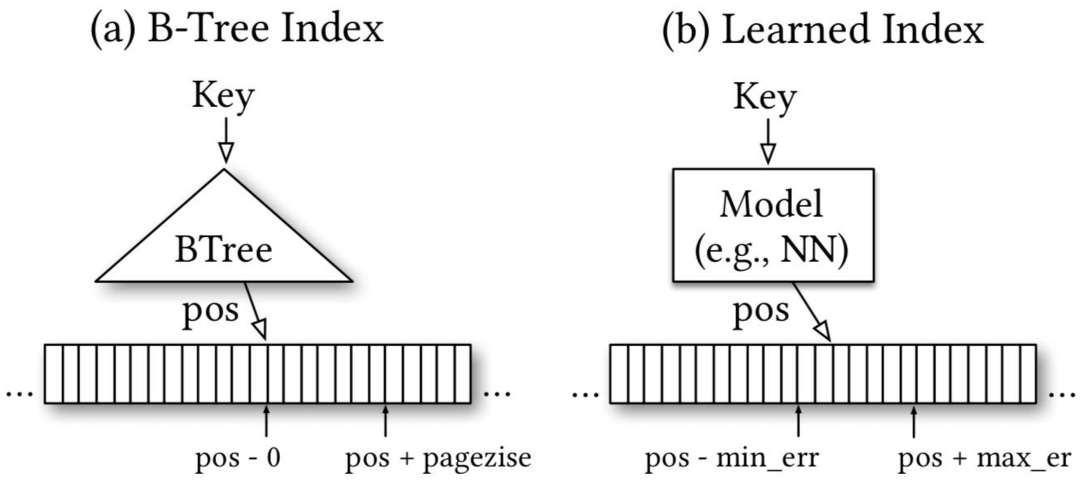
- Size of the Model
- Amount of Overhead
- Number of Training Iterations
- Amount of Training Data

B-Trees | Range Index



Key Innovations

B-Trees as a Modeling Problem

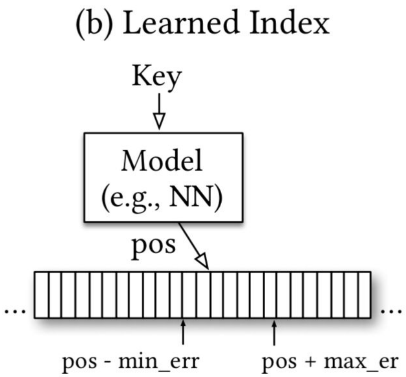


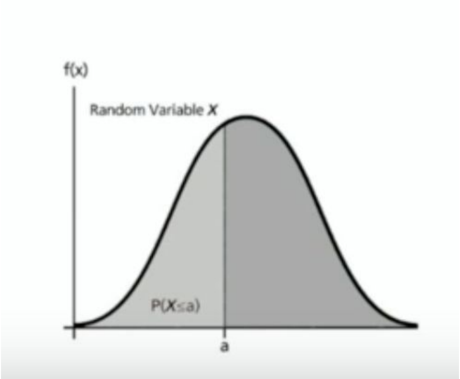
- Smaller Index
- Faster Lookup
- More Parallelism
- Cheaper Insertion
- Hardware Acceleration

Key Innovations

B-Trees as a Cumulative Distribution Function

Predicted Position = P(x $\le$ key) * # 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.

Learning Index Framework (LIF)

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

Solution: Learning Index Framework

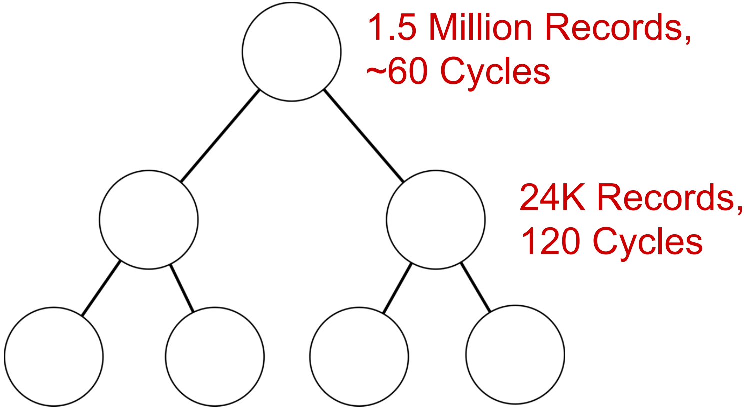
- Index Synthesis System
- Given an Index => Generate, optimize, and test different index configurations
- For simple models (e.g. linear regression), learns values on the fly
- For complex models, extract model weights and generate C++ index structure

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



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

Loss Function

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

Loss Function Initialization

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

Algorithm 1: Hybrid End-To-End Training

```
Input: int threshold, int stages[], NN_complexity
Data: record data[], Model index[][]
Result: trained index
M = stages.size;
tmp_records[][];
tmp_records[1][1] = all_data;
for i ← 1 to M do
    for j ← 1 to stages[i] do
        index[i][j] = new NN trained on tmp_records[i][j];
        if i < M then
            for r ∈ tmp_records[i][j] do
                p = index[i][j](r.key) / stages[i + 1];
                tmp_records[i + 1][p].add(r);
for j ← 1 to index[M].size do
    index[M][j].calc_err(tmp_records[M][j]);
    if index[M][j].max_abs_err > threshold then
        index[M][j] = new B-Tree trained on tmp_records[M][j];
return index;
```

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

		Map Data			Web Data			Log-Normal Data		
Type	**Config**	Size (MB)	Lookup (ns)	Model (ns)	Size (MB)	Lookup (ns)	Model (ns)	Size (MB)	Lookup (ns)	Model (ns)
Btree	page size: 32	52.45 (4.00x)	274 (0.97x)	198 (72.3%)	51.93 (4.00x)	276 (0.94x)	201 (72.7%)	49.83 (4.00x)	274 (0.96x)	198 (72.1%)
	page size: 64	26.23 (2.00x)	277 (0.96x)	172 (62.0%)	25.97 (2.00x)	274 (0.95x)	171 (62.4%)	24.92 (2.00x)	274 (0.96x)	169 (61.7%)
	page size: 128	13.11 (1.00x)	265 (1.00x)	134 (50.8%)	12.98 (1.00x)	260 (1.00x)	132 (50.8%)	12.46 (1.00x)	263 (1.00x)	131 (50.0%)
	page size: 256	6.56 (0.50x)	267 (0.99x)	114 (42.7%)	6.49 (0.50x)	266 (0.98x)	114 (42.9%)	6.23 (0.50x)	271 (0.97x)	117 (43.2%)
	page size: 512	3.28 (0.25x)	286 (0.93x)	101 (35.3%)	3.25 (0.25x)	291 (0.89x)	100 (34.3%)	3.11 (0.25x)	293 (0.90x)	101 (34.5%)
Learned Index	2nd stage models: 10k	0.15 (0.01x)	98 (2.70x)	31 (31.6%)	0.15 (0.01x)	222 (1.17x)	29 (13.1%)	0.15 (0.01x)	178 (1.47x)	26 (14.6%)
	2nd stage models: 50k	0.76 (0.06x)	85 (3.11x)	39 (45.9%)	0.76 (0.06x)	162 (1.60x)	36 (22.2%)	0.76 (0.06x)	162 (1.62x)	35 (21.6%)
	2nd stage models: 100k	1.53 (0.12x)	82 (3.21x)	41 (50.2%)	1.53 (0.12x)	144 (1.81x)	39 (26.9%)	1.53 (0.12x)	152 (1.73x)	36 (23.7%)
	2nd stage models: 200k	3.05 (0.23x)	86 (3.08x)	50 (58.1%)	3.05 (0.24x)	126 (2.07x)	41 (32.5%)	3.05 (0.24x)	146 (1.79x)	40 (27.6%)

Figure 4: Learned Index vs B-Tree

Experiment Results

String Datasets

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

Figure 6: String data: Learned Index vs B-Tree

Experiment Results

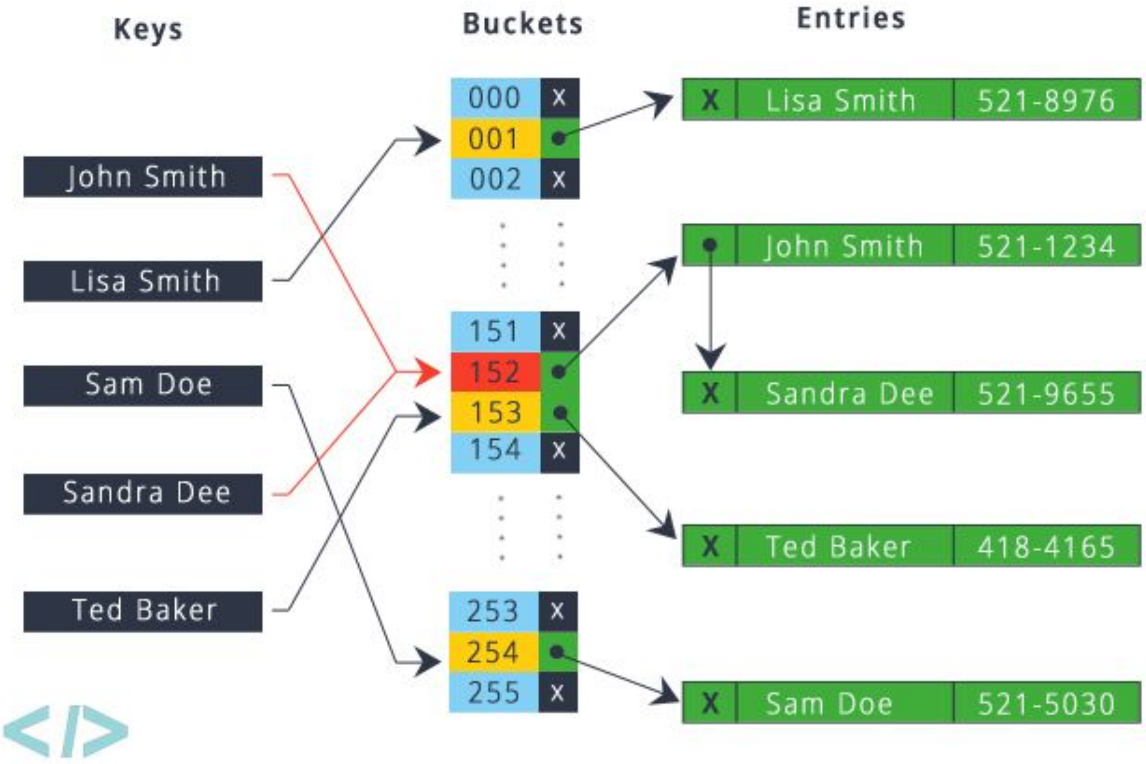
Dataset	Memory Savings	Speedup
Server Logs (Timestamps)	88%	1.88x
Longitudes	99%	2.7x
Synthetic Log Normal Data	88%	1.8x
Strings (Document IDs)	63%	1.1x

Experiment Results | Alternative Baselines

	Lookup Table w/ AVX search	FAST	Fixe-Size Btree w/ interpol. search	Multivariate Learned Index
Time	199 ns	189 ns	280 ns	105 ns
Size	16.3 MB	1024 MB	1.5 MB	1.5 MB

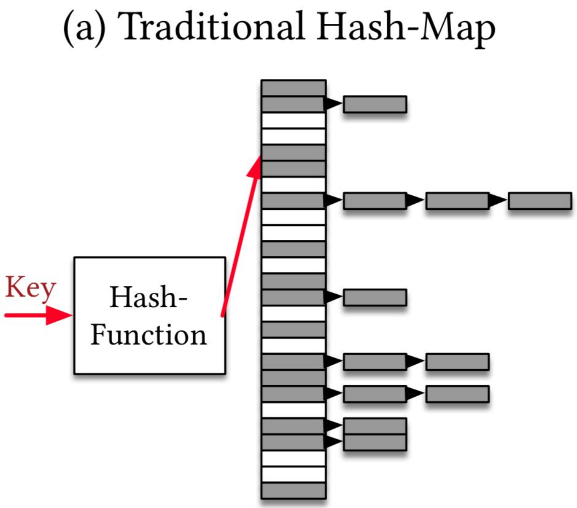
Figure 5: Alternative Baselines

Hashmaps | Point Index



Key Innovations

Hashmaps as a Model



Idea: Use Learned CDF as the Hash Function

Perfect CDF model should have **zero** collisions

Independent of type of hashmap

Key Results & Takeaways

	% Conflicts Hash Map	% Conflicts Model	Reduction
Map Data	35.3%	07.9%	77.5%
Web Data	35.3%	24.7%	30.0%
Log Normal	35.4%	25.9%	26.7%

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

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

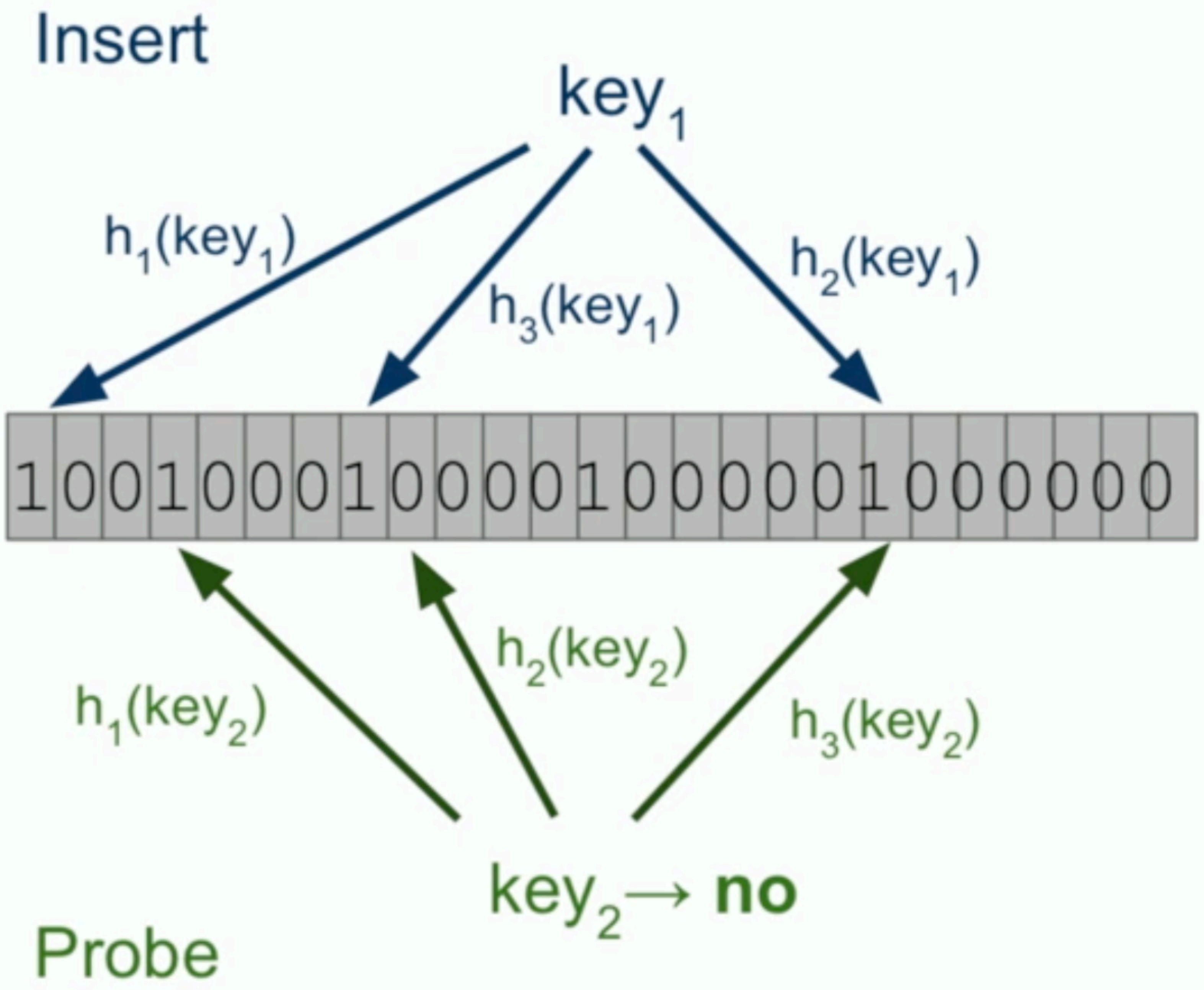
Figure 11: Model vs Random Hash-map

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

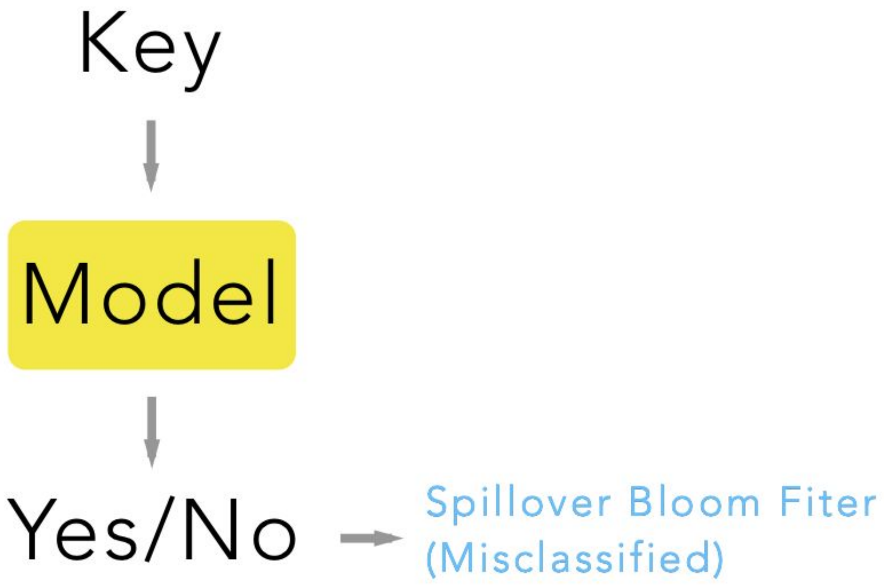
Bloom Filters - Existence Index



Guarantees FNR=0; small (chosen) FPR.

Key Innovations

Bloom Filters as Binary Classification

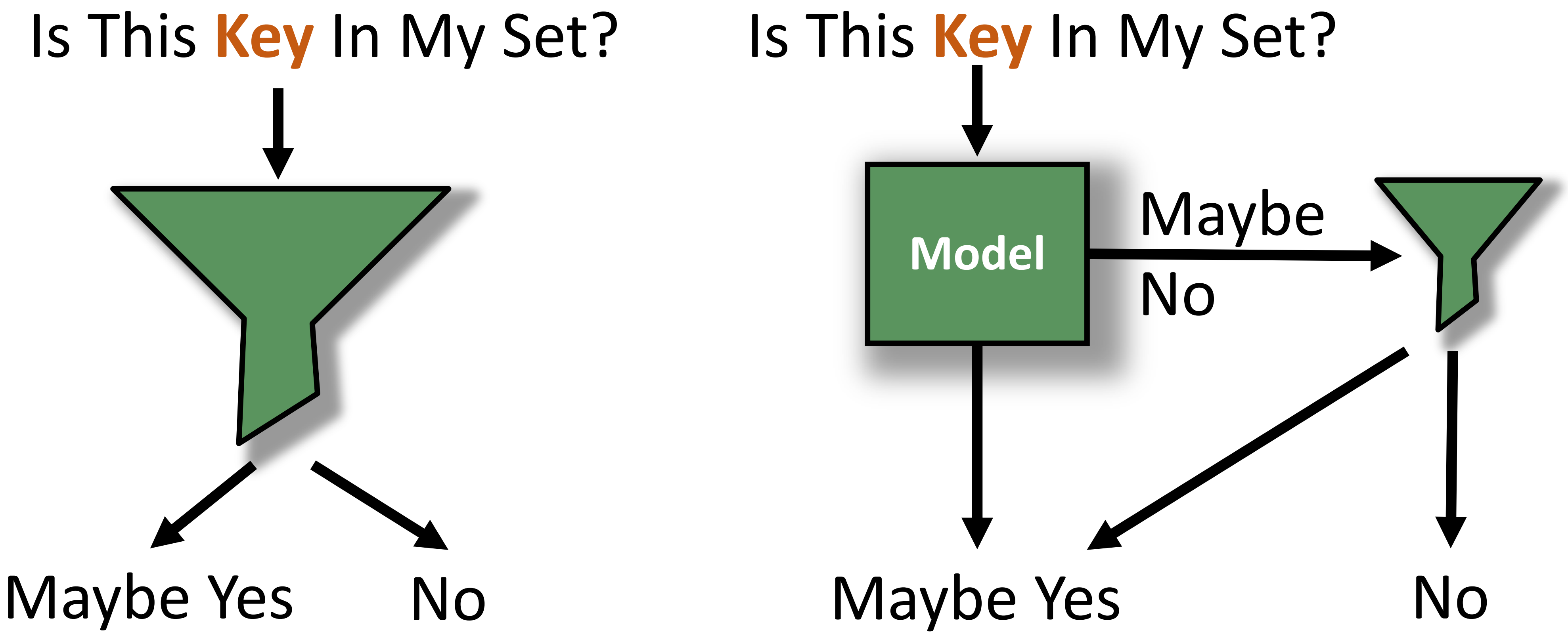


Idea: Binary Classification

Problem: False Negatives

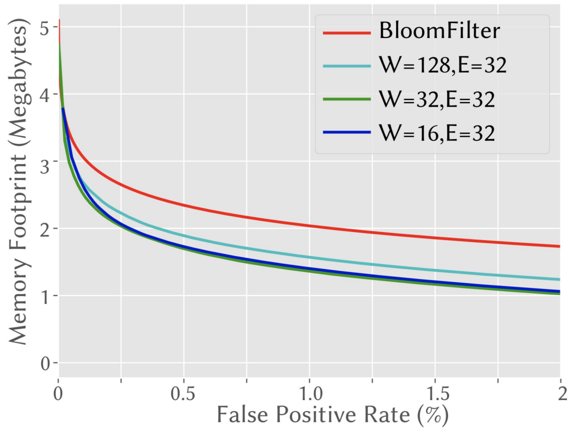
Solution: Hybrid Model / Bloom Filter

Bloom Filter- Approach 1



36% Space Improvement over Bloom Filter at Same False Positive Rate

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 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?