The Case For Learned Index Structures

Authors: Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, Neoklis Polyzotis

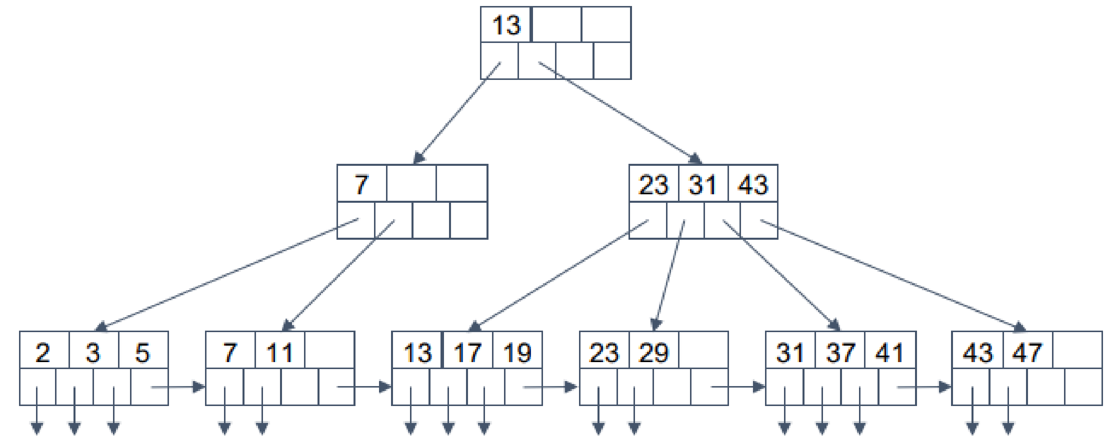
Presenters: Tommy Skodje, David Keefe, Tiffany Ma, Nikki Van Handel

Georgia Tech

Background/Overview

Background and Motivation

- Indexing is used in databases to increase the speed of data retrieval
  - Examples: B-Trees (covered in class), hash-maps, bloom filters
- Problem with indexes: Index efficiency scales with CPU and memory speed
  - Has slowed progress as of late due to Moore's Law no longer holding
- An inefficiency: They do not take advantage of patterns in data



Georgia Tech

Background and Motivation

- The authors argue that specialized indexes that reflect patterns in the data can be created using machine learning models

- Existing indexes can be replaced with ML models with similar semantic guarantees

- Using the distribution of the data, learned indexes can perform lookups faster and require less space to store

Georgia Tech

Related Work

- B+-Trees (discussed in class) and other tree indexes
  - FAST – Uses SIMD (Single Instruction Multiple Data) processing to take advantage of GPU compute power, like an ML model.
  - Tries/radix trees for text data
  - BF-Trees – A B+-Tree with a Bloom Filter on each leaf node
- Learning Hash Functions – Learning a locality-sensitive hash (LSH) function to build an Approximate Nearest Neighbor (ANN) index.
- Perfect Hashing – Tries to avoid conflicts, but the size of the hashing function grows with the size of the data.
- Mixture of experts – Multi-network architectures where layers are replaced with NN and improve computation speeds through introducing sparsity.

This paper aims to expand on this related work by learning the data distribution. The authors aim to use an ML model as the index itself

Georgia Tech

Overview

- Range Indexes and RM (Recursive-Model) Indexes
  - Related to B-Trees
- Point Indexes
  - Related to Hash-Maps
- Existence Indexes
  - Related to Bloom Filters

- Evaluation
  - Datasets: web-server logs, map dataset, log-normal distribution
  - Metrics: index size, lookup time, model execution time

Georgia Tech

Range Index

Range Index

- B-Tree – A tree that maps a look-up key to a position in a sorted array
  - Guarantee: The record's key is the first key greater than or equal to the lookup key
- Common practice to only index the first key of a page as a space-saving measure
  - Therefore, there is some level of "error" in each lookup, which is equal to the page size
- Error is analogous to error in a regression tree in machine learning!
- As page size error resembles model error, we can replace a B-Tree with a model

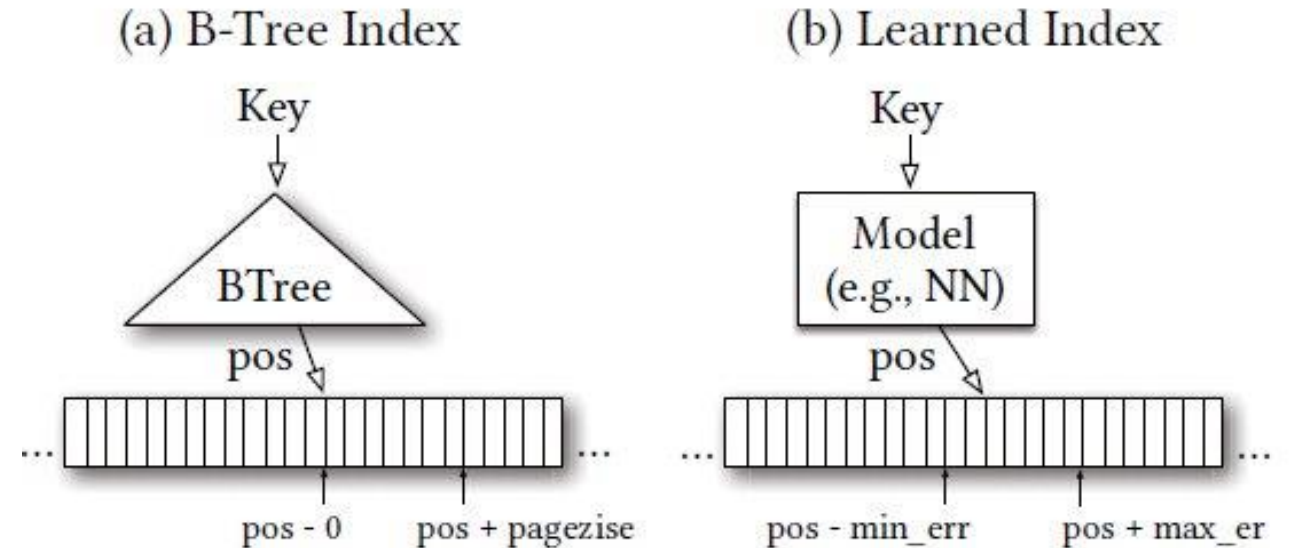


Figure 1: Why B-Trees are models

Georgia Tech

Model Complexity

- Each traversal of a level in a B-Tree can be thought of as "gaining precision"
  - The chance of finding a "positive result" (the matching record) increases the further traversal progresses
  - Reducing the search space at each level
- LogbaseB(N) nodes need to be traversed, with B being the branching factor of the tree
  - Have to add time for traversing a page with binary search (~50 cycles)
- Main timesaver of the model comes at the page level
  - Binary search is slow and hard to parallelize. Meanwhile, CPUs can handle 8-16 SIMD operations per cycle.
- More timesaving opportunity comes with SIMD processing and GPUs!

Range Indexes as CDF Models

- A model that predicts a position in a sorted array will model the Cumulative Distribution Function (CDF).

- B-Trees work like this too! They are models that "learn" the position of the data during construction

- P = F(Key) * N
  - P = position estimate
  - F(Key) = CDF to estimate the likelihood of observing a key less than or equal to the lookup key
  - N = Number of keys

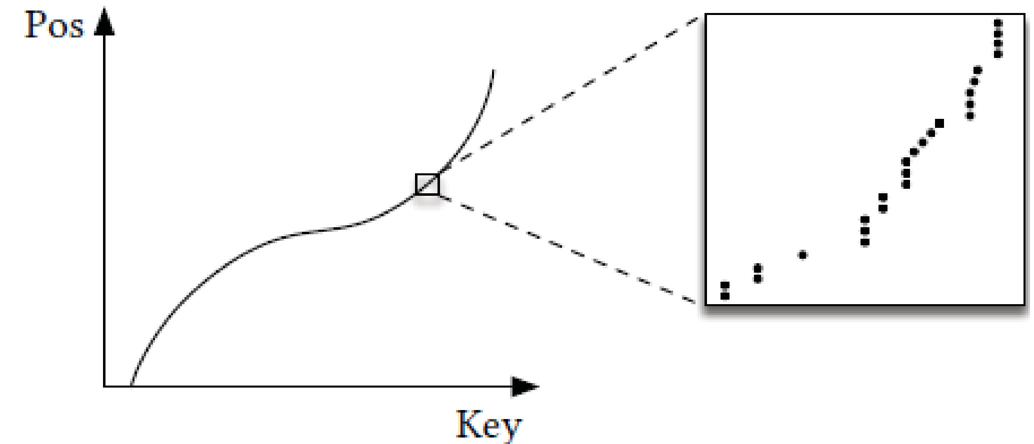


Figure 2: Indexes as CDFs

Georgia Tech

A First, Naïve Learned Index

- 200M web-server log records
- Neural network with ReLU activation functions
  - 32 neurons/layer
- Input: timestamps
- Labels (things to predict): positions in the sorted array

- Result: Worse performance than B-Trees. Why?
  - Tensorflow overhead
  - The model has trouble being accurate for individual data instances
  - B-Trees utilize the cache extremely well

Georgia Tech

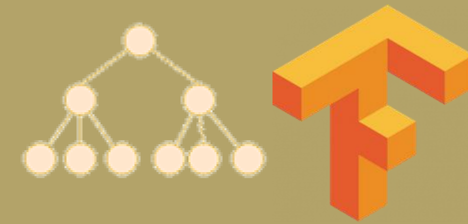
Recursive-Model Index

Learning Index Framework

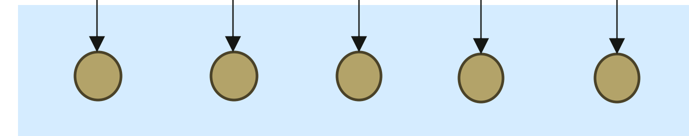
- Inputs: Index specification
- Framework:
    1. Index configuration generation
    2. Index optimization
        1. Model Choice
        2. Page Size
        3. Search Strategies
    3. Automatic Testing
- Usage
    - C++ index operations based on model weights.

Record data

Index Synthesis



Model Weights



Key → Inference Model → Position

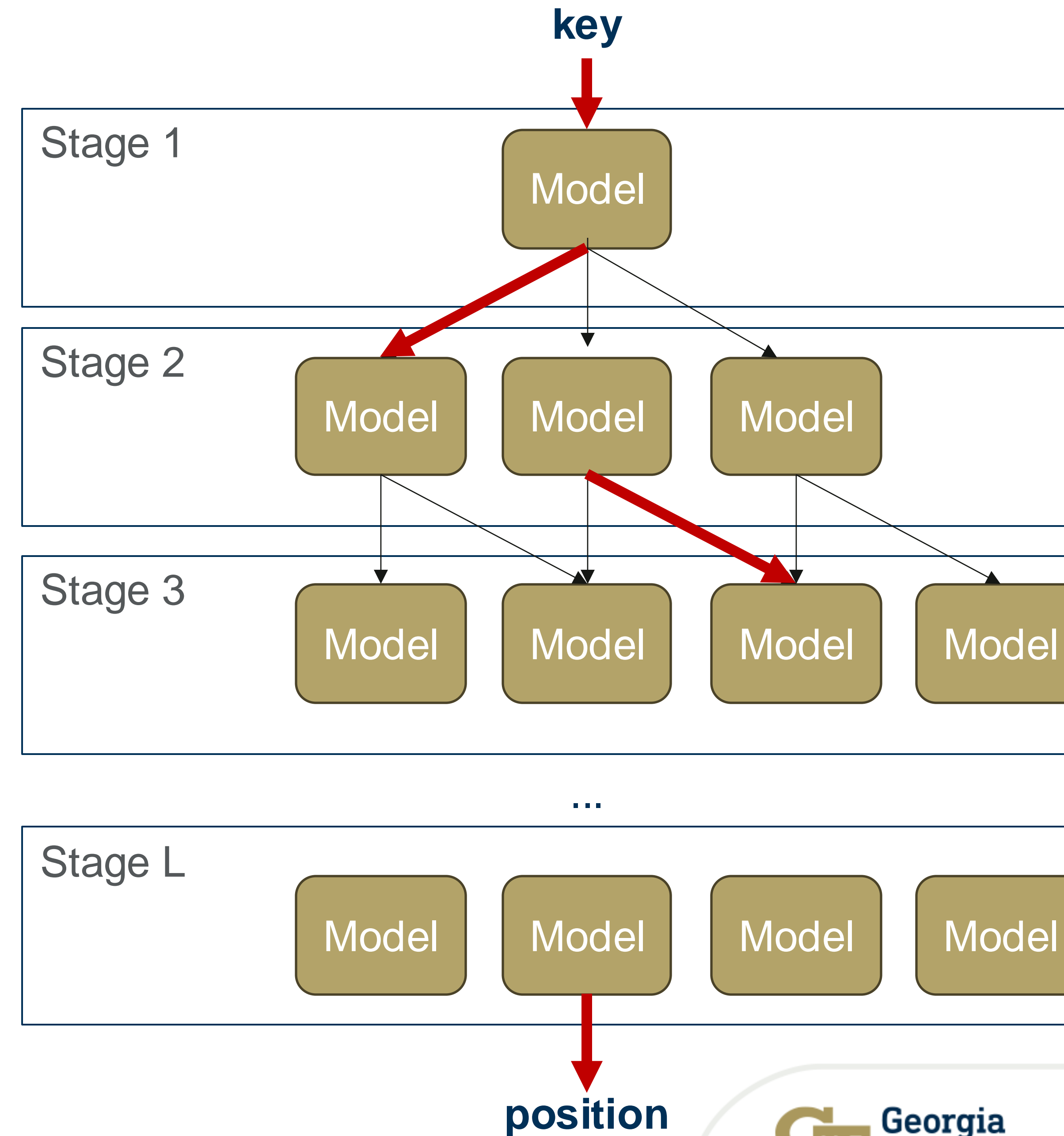
Georgia Tech

Recursive-Model Index

- **Problem**: How can a learned model match the accuracy of a B Tree?

- **Solution**: Recursive Regression
  - Choose model the next model based on prior model output – “expert selection”
  - At each stage,
  - Prior stages learn data “shape” – distribution
  - Final stage predicts position.
  - Train k-th model by loss:

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



Georgia Tech

Hybrid Index

L=0 — Train the top-node model

L=L+1 — Pick the model for the next stage

Add keys to the next model

Post — Replace models with B trees if error > threshold

Algorithm 1: Hybrid End-To-End Training

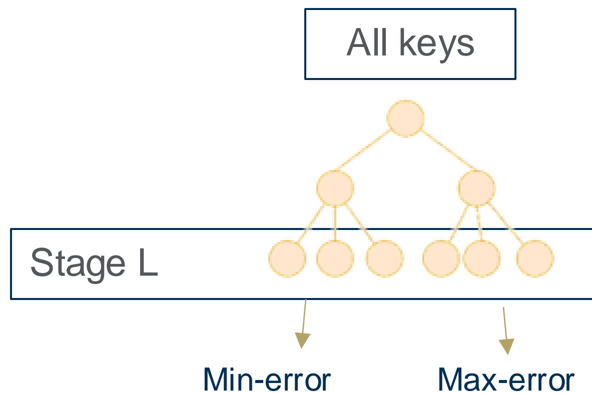
```
Input: int threshold, int stages[], NN_complexity
Data: record data[], Model index[][]
Result: trained index
1  M = stages.size;
2  tmp_records[][];
3  tmp_records[1][1] = all_data;
4  for i ← 1 to M do
5      for j ← 1 to stages[i] do
6          index[i][j] = new NN trained on tmp_records[i][j];
7          if i < M then
8              for r ∈ tmp_records[i][j] do
9                  p = index[i][j](r.key) / stages[i + 1];
10                 tmp_records[i + 1][p].add(r);
```

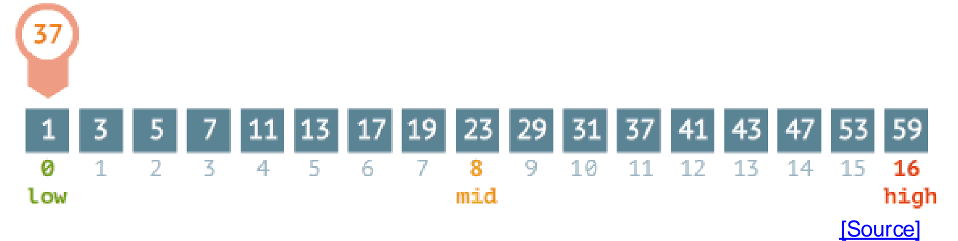
Worst case guarantee: B-Tree accuracy

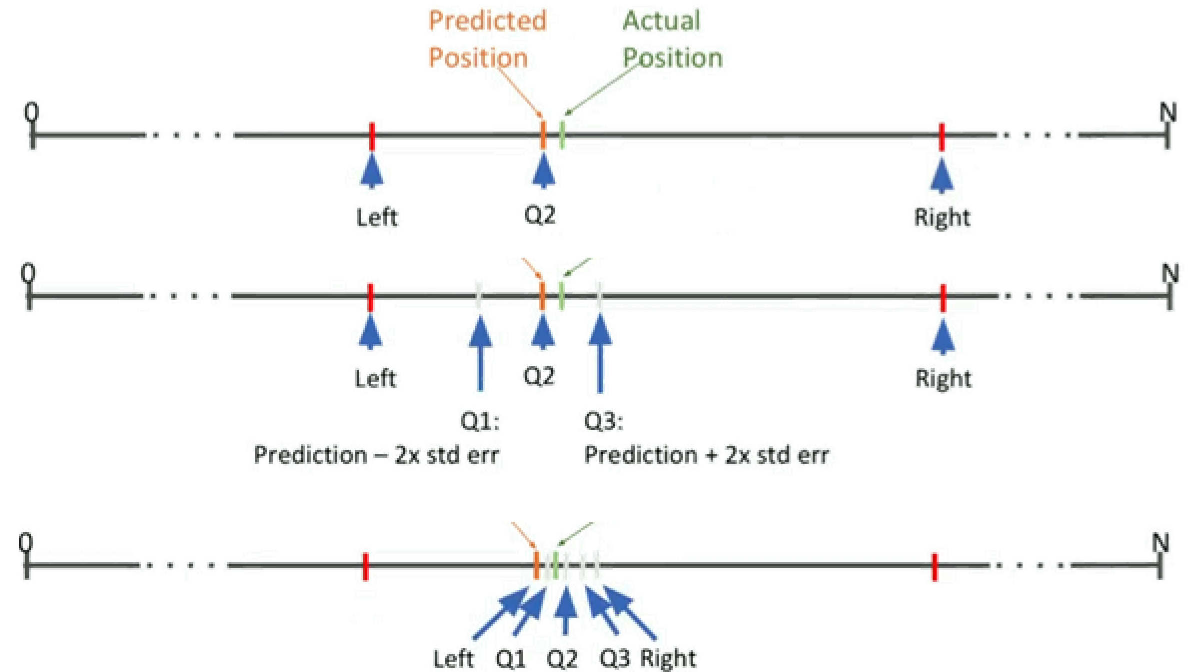
Georgia Tech

Search Strategies

- **Last mile search:** Finding exact position from model prediction
- Model Biased Search (default)
  - Like binary but centered about predicted position.
- Biased Quaternary Search
  - Save computation time







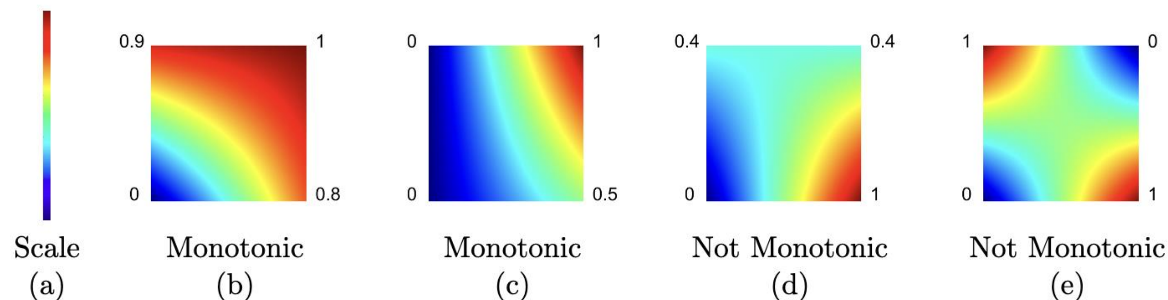
Stanford Seminar, US, Stanford Seminar - The Case for Learned Index Structures.(Oct 18 2018). Accessed: Nov 19 2024. [Online Video]. Available: https://www.youtube.com/watch?v=NaqJO7rrXy0

Georgia Tech

Implementation

Monotonicity

- Monotonic: varying in such a way that it either never decreases or never increases.
- Forcing monotonicity guarantees correct accurate error bounds
- Non-monotonic models may make use of exponential search.



Scale (a) Monotonic (b) Monotonic (c) Not Monotonic (d) Not Monotonic (e)

Training

- 200M records training takes a few seconds to train for RMI
- More complex models require training on the order of minutes.
  - Stochastic gradient descent may converge quickly for simple neural nets.
  - Hyperparameter search performed with grid search.
- Convergence criterion may be set relatively large.

M. Gupta, A. Cotter, J. Pfeifer, K. Voevodski, K. Canini, A. Mangylov, W. Moczydlowski, and A. Van Esbroeck. Monotonic calibrated interpolated look-up tables. The Journal of Machine Learning Research, 17(1):3790–3836, 2016.

Georgia Tech

Results: RMI

- Compared performance to a two stage optimized B-Tree
- Additional compressions are possible for both Btrees but even more for NN!

Type	Config	Map Data Size (MB)	Map Data Lookup (ns)	Map Data Model (ns)	Web Data Size (MB)	Web Data Lookup (ns)	Web Data Model (ns)	Log-Normal Data Size (MB)	Log-Normal Data Lookup (ns)	Log-Normal Data 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%)

- Alternative baselines compared

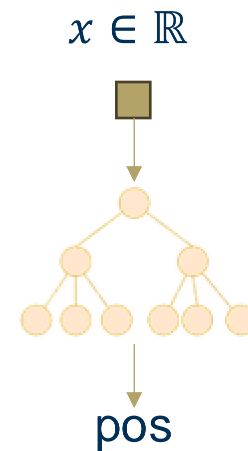
	Lookup Table w/ AVX search	FAST	Fixe-Size Btree w/ interpol. search	Multivariate Learned Index
Time	199 ns	189 ns	280 ns	105 ns
Size	16.3 MB	1024 MB	1.5 MB	1.5 MB

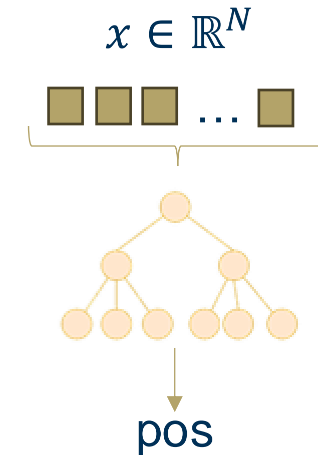
Figure 5: Alternative Baselines

Georgia Tech

String Indexing

- Many databases require strings to be indexed
- Tokenize strings to into feature vectors using ASCII character values
  - All vectors set to length N, longer vectors truncated, smaller vectors filled with 0's
- Similar neural network structure used, with input being a vector instead of a single value





	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

Georgia Tech

Point Index

Hash-Model Index

- Primary challenge with hash maps is avoiding collisions
- Hash functions do not consider data distribution, often leading to high occurrence of collisions
- A model can learn the CDF, mapping keys more uniformly across the output space according to the distribution
- End goal is to reduce collision occurrence

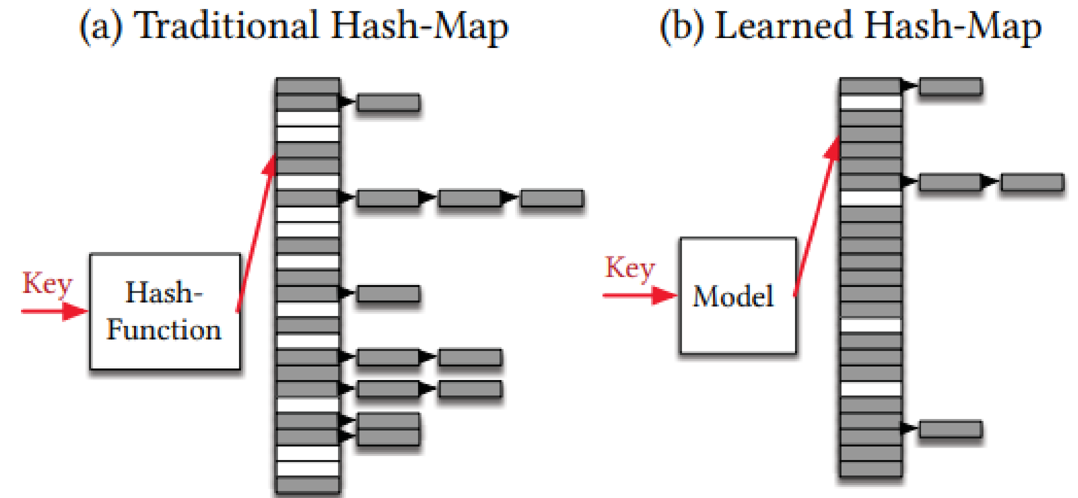


Figure 7: Traditional Hash-map vs Learned Hash-map

Georgia Tech

Evaluation – Point Index

- Traditional hash functions lead to equal collision occurrence across datasets
- Point indexes can perform better on datasets with more learnable CDFs
- Hash-Model Index reduced collisions significantly across 3 datasets

	% 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

Georgia Tech

Existence Index

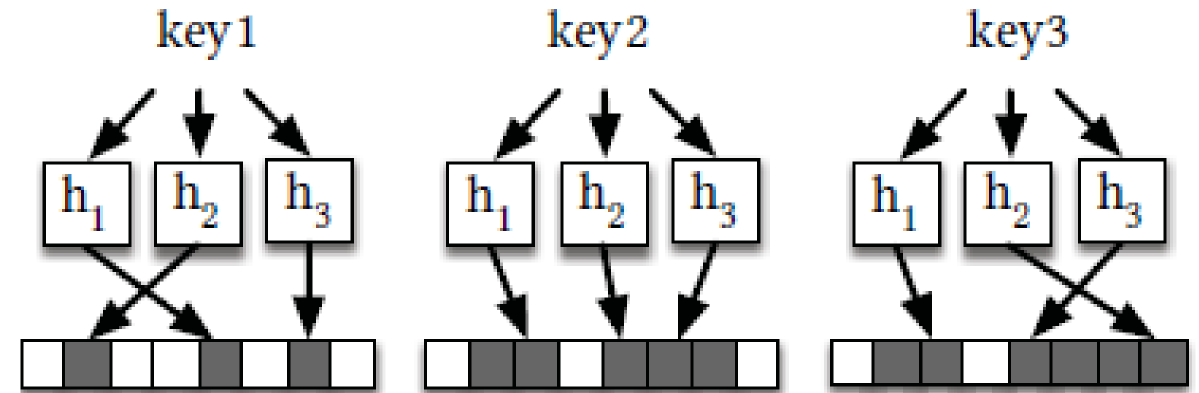
Existence Index

- Crucial for determining if an element **exists** within a data set
- **Bloom filters –** space efficient probabilistic data structure
- Primary use – verifying if a key is present in cold storage, e.g. SSTables

Georgia Tech

Existence Index – Bloom Filter Basics

- Core components:
  - **Bit array** of size *m* to store bits indicating the presence of elements
  - **Hash functions** (n=k) that map elements to positions in the bit array
- Process:
  - Insertion is performed by setting bits to 1 at positions returned by hash functions
  - Membership check is performed by checking if any bits are 0 (=absent)
- Characteristics:
  - Guarantees no false negatives, but false positives are possible



Georgia Tech

Existence Index - Challenges

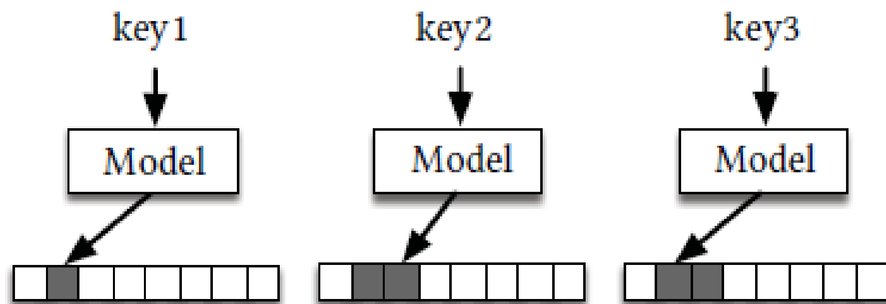
- **Latency and Space Trade-offs**
  - Cold storage access latency allows for more complex models.
    - **Example**: For 1 billion records, around 1.76 GB is needed.
    - For a 0.01% False Positive Rate (FPR), approximately 2.23 GB is required.
    - Ongoing research to reduce memory consumption without sacrificing performance
- **Optimization Goal**: Minimize index space and false positives.
- **Modeling Techniques**
  - Possibility of leveraging learned models for more efficient existence indexes

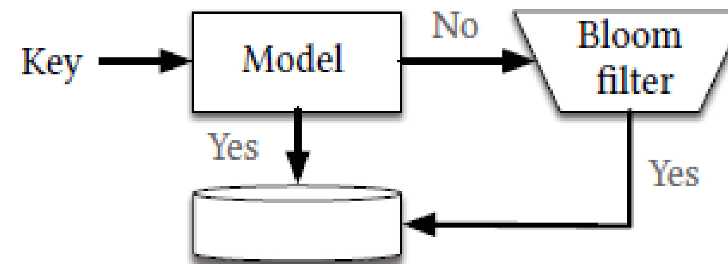
Existence Index – Learned Bloom Filters

- **Approach**:
- Unlike traditional indexes, learned Bloom filters use machine learning to predict key presence.
- The model differentiates keys from non-keys by learning their distributions.
- **Key Advantage**: Potential to optimize for specific query patterns, observed from historical data

Existence Index – Learned Filter Methods

- **Classification as a Solution**:
  - Treat existence as a binary classification task (key or non-key).
  - Models like Recurrent Neural Networks (RNN) or Convolutional Neural Networks (CNN) can classify keys with minimal log loss.
- **Overflow Bloom Filter**:
  - Used to handle false negatives by setting a threshold (τ) for classification accuracy

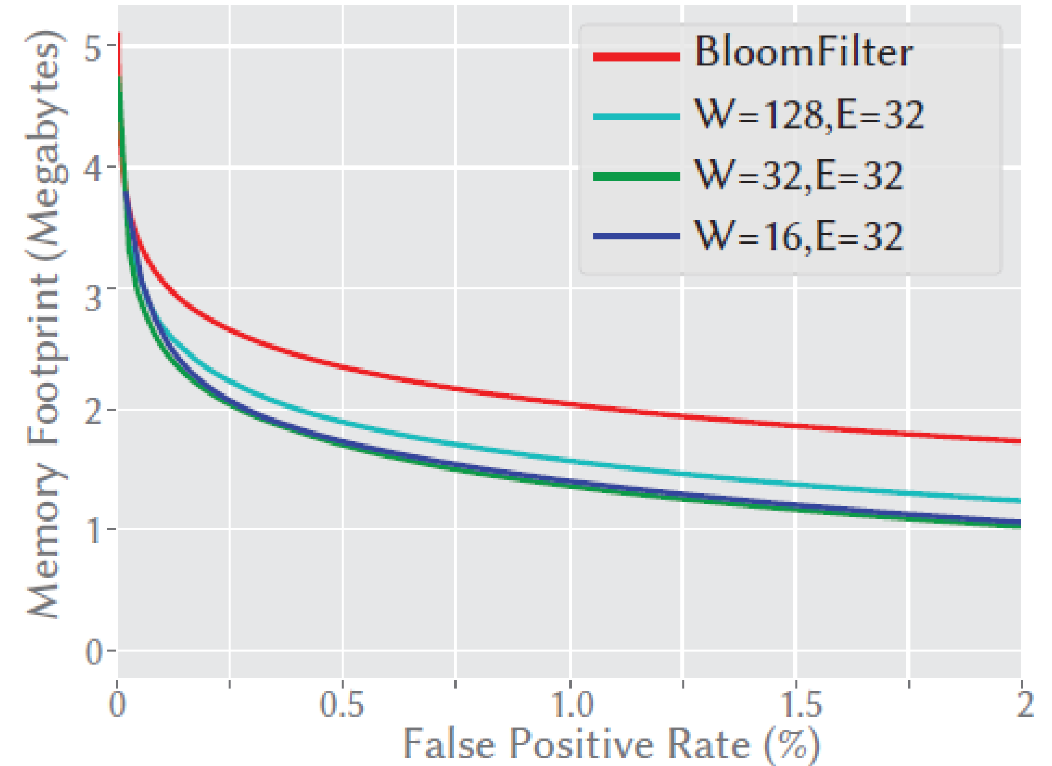




Georgia Tech

Evaluation - Existence Index

- **Goal**: Maximize collisions among non-keys, minimize collisions between keys and non-keys.
- **Mechanism**: Learned model maps values to bit positions, improving space efficiency and accuracy
- **Case Study**: Applied to blacklist phishing URLs using a dataset of 1.7M URLs.
- **Results**:
    - Achieved a significant reduction in memory usage compared to standard Bloom filters.
    - Maintained desired FPR while reducing false negatives using a smaller model size



Georgia Tech

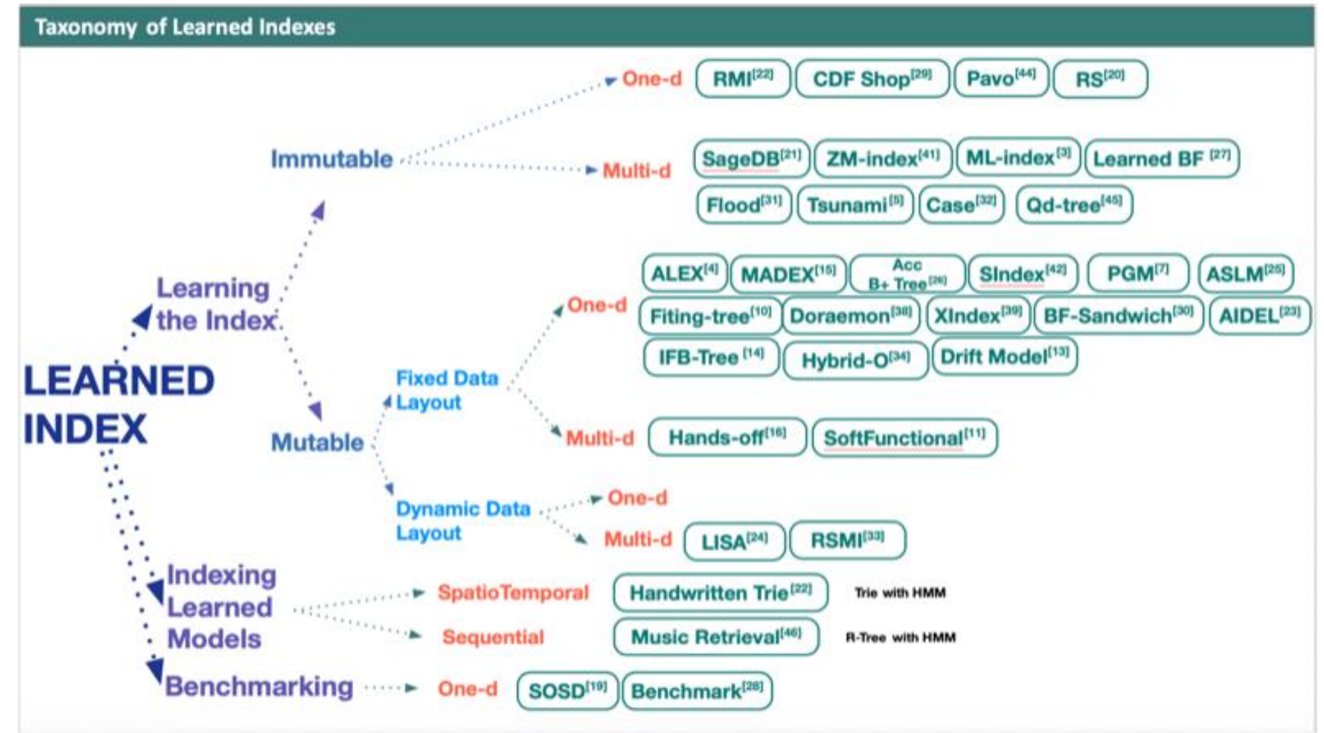
Conclusion

Conclusions

- **Contributions:**
  - Introduced how machine learning models like neural networks can be used to replace traditional index structures (e.g., B-Trees, Bloom Filters, Hash-maps)
  - Presented recursive model index (RMI) and hybrid approaches, balancing complexity and accuracy for practical use
- **Limitations:**
  - Paper suggests high theoretical performance with **appends** and **inserts**, however, no benchmarking is given.
  - Performance dependency on data distribution; irregular or complex patterns need exploration.
- **Future work:**
  - Extend the exploration of learned indexes to incorporate a broader range of machine learning models beyond linear models and neural networks.
  - Develop learned indexes for multi-dimensional data, leveraging the ability of neural networks and other models to capture complex high-dimensional relationships.
  - Investigate the application of learned models in other database operations, such as sorting and join algorithms.

Current Work

- Derivatives
  - [Learning Multi-Dimensional Indexes]()
  - [ALEX: An Updatable Adaptive Learned Index]()
  - [Partitioned Learned Bloom Filters]()
- Block Range Indexes (BRINs) - Splits values in a table into blocks, then summarizes the data in that block. Reduces data volume, but doesn't take advantage of data structure like learned indexes



Abdullah Al-Mamun, Hao Wu, Walid G. Aref: A Tutorial on Learned Multi-dimensional Indexes. ACM SIGSPATIAL Conference, pp. 1-4, Nov. 2020.

Georgia Tech

Study Questions

- What were the main reasons for poor performance of the authors' first naïve learned index?
- How/Why can the CDF of a data distribution be used to model an index structure?

Georgia Tech

Resources

Lectures by the authors:

- Alex Beutel and Ed Chi: Seminar at Standford
- Tim Kraska at Sigmod 2018

Georgia Tech