The Case for Learned Index Structures

Google

SIGMOD'18

DB/AI Bootcamp

2018 Summer

DataLab, CS, NTHU

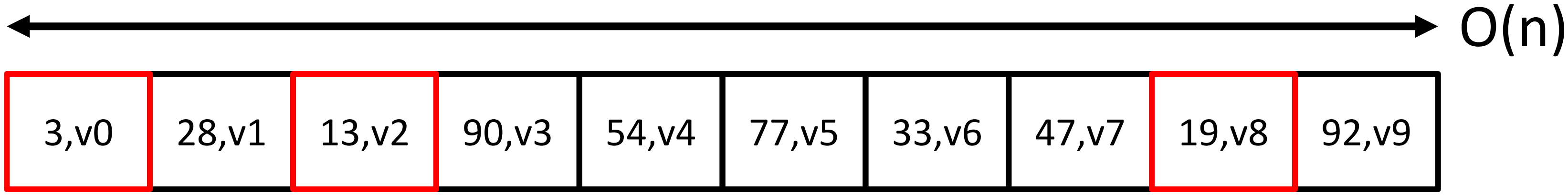
Database Indexes

- A database index is a data structure to improves the speed of search operations

An Example

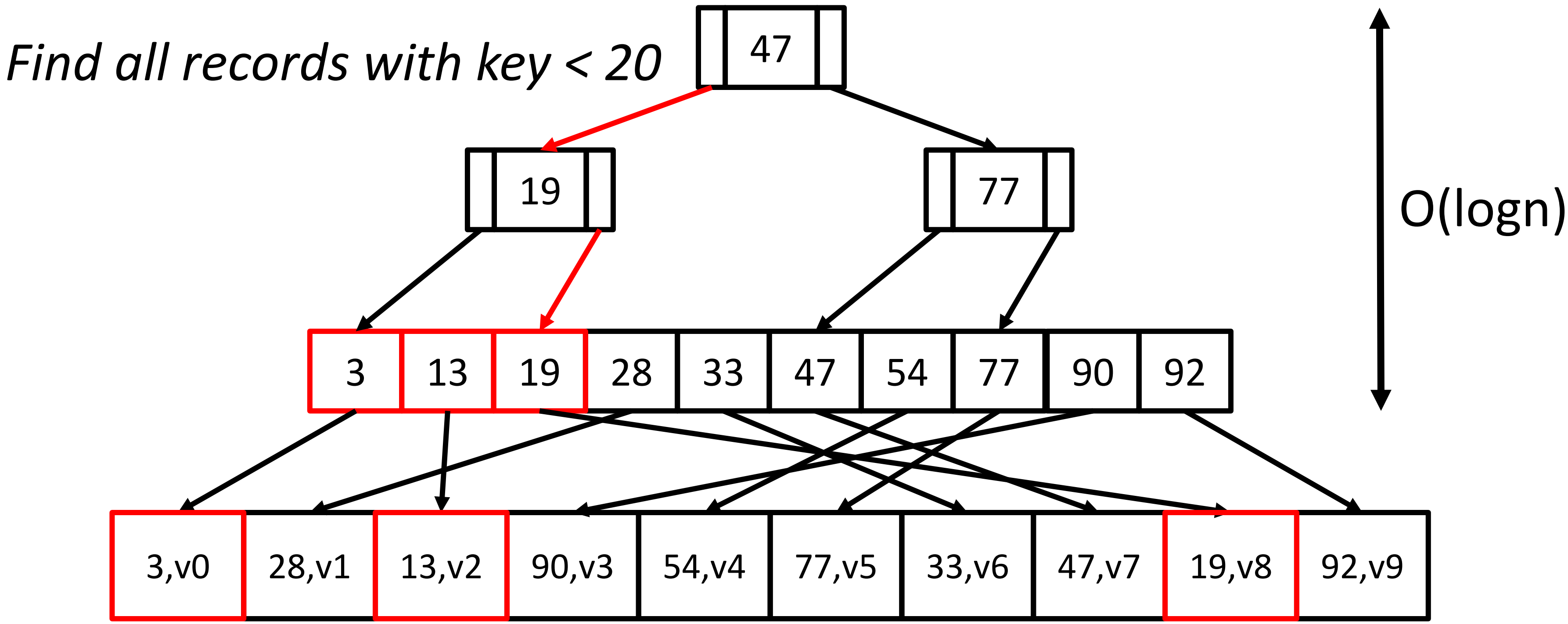
- 10 records, each record is a key-value pair
- Keys are selected from $\{k \in \mathrm{Z} | 0 \leq k \leq 99\}$

Find all records with key < 20



An Example

- 10 records, each record is a key-value pair
- Keys are selected from $\{k \in \mathrm{Z} | 0 \leq k \leq 99\}$



Motivation

- Hardware trends
  - Conventional index structures are branch-heavy: CPU
  - Learned index structures are computation-heavy: GPU
- Conventional index structures are *general*, and thus may omit some optimizations by leveraging on the data distribution
  - Consider a dataset with 1M unique keys with a value from 1M and 2M (so the value 1,000,009 is stored at position 10)

Outline

- Range index
- Point index
- Existence index

Outline

- Range index
- Point index
- Existence index

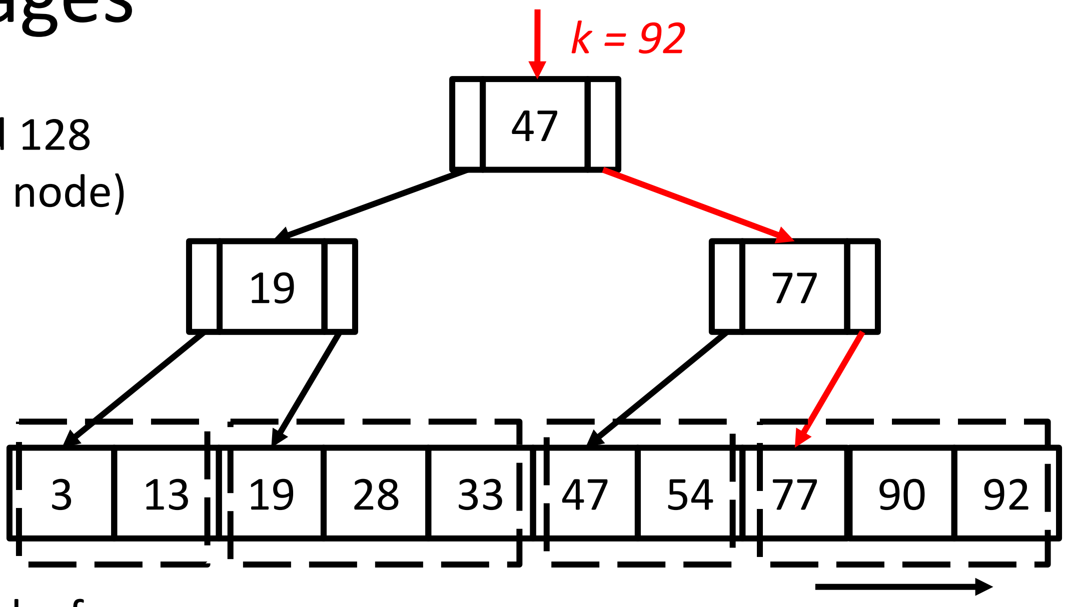
Assumptions

- Read-only analytic workloads
- Integer keys
- Experiments are done with the same hardware resources (i.e. CPU)

B-Tree Index

- Step 1: traverse the internal nodes to a leaf page
- Step 2: search for the specific index records in the leaf pages

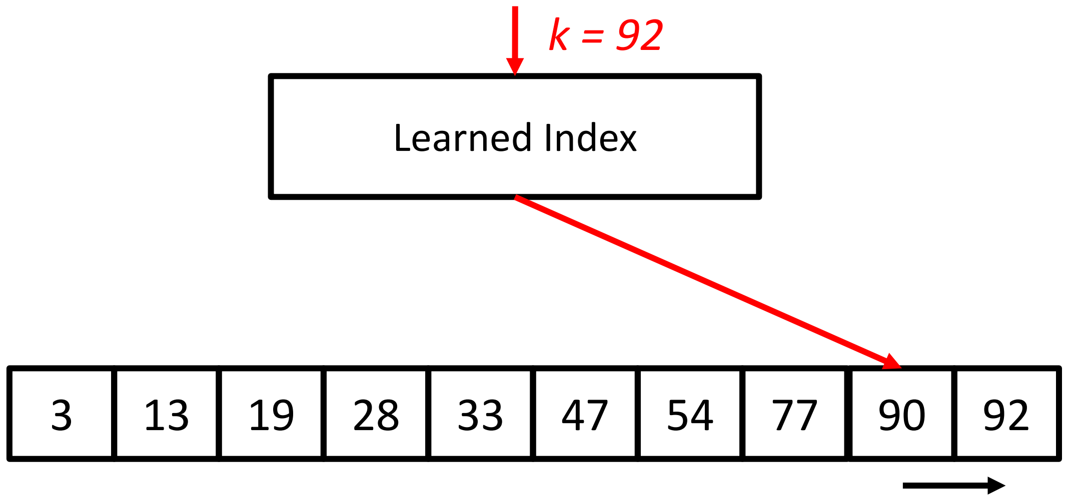
(in practice, around 128 entries per internal node)



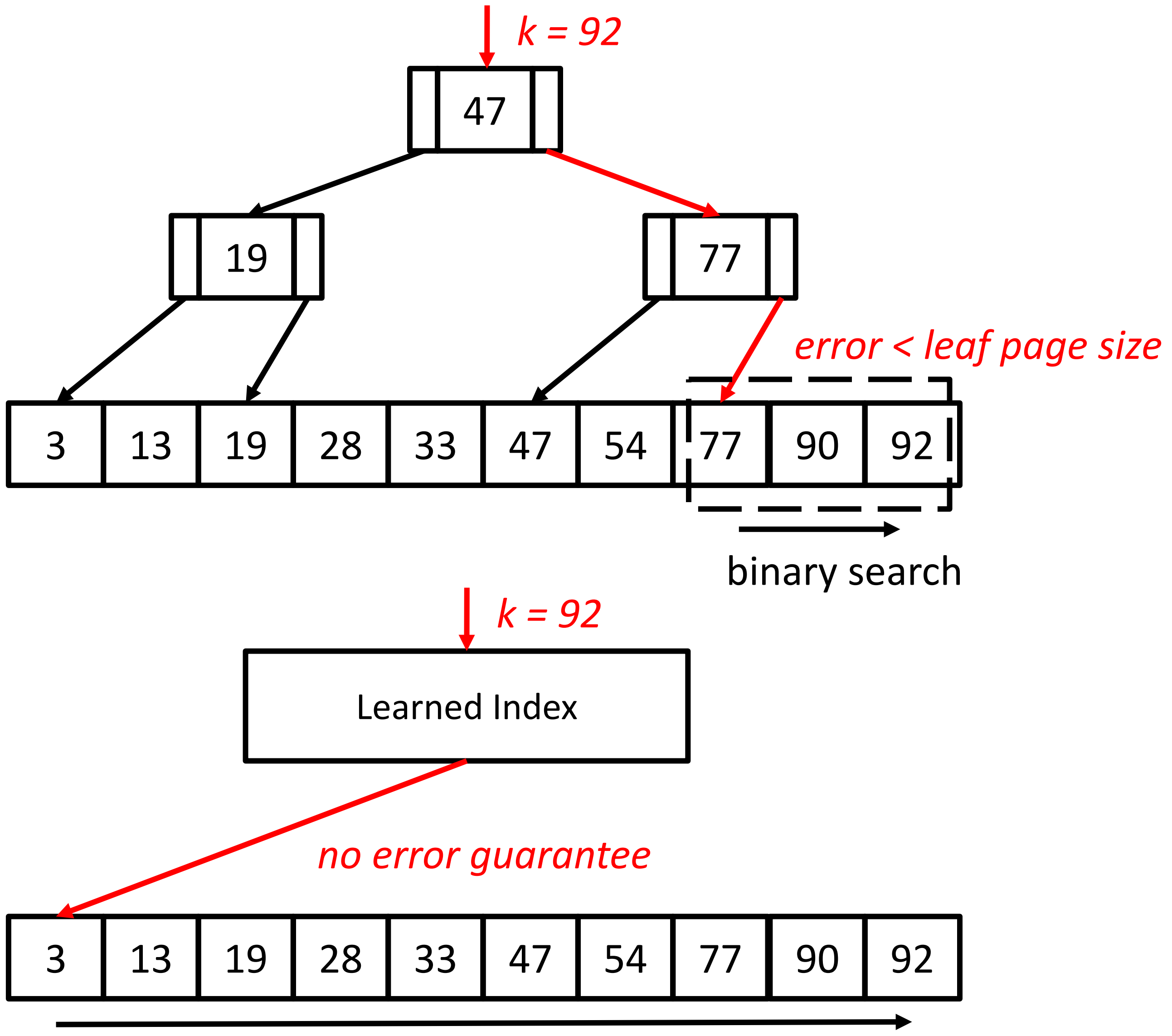
leaf page

Learned Index

- Predict the position of data entry with key *k*

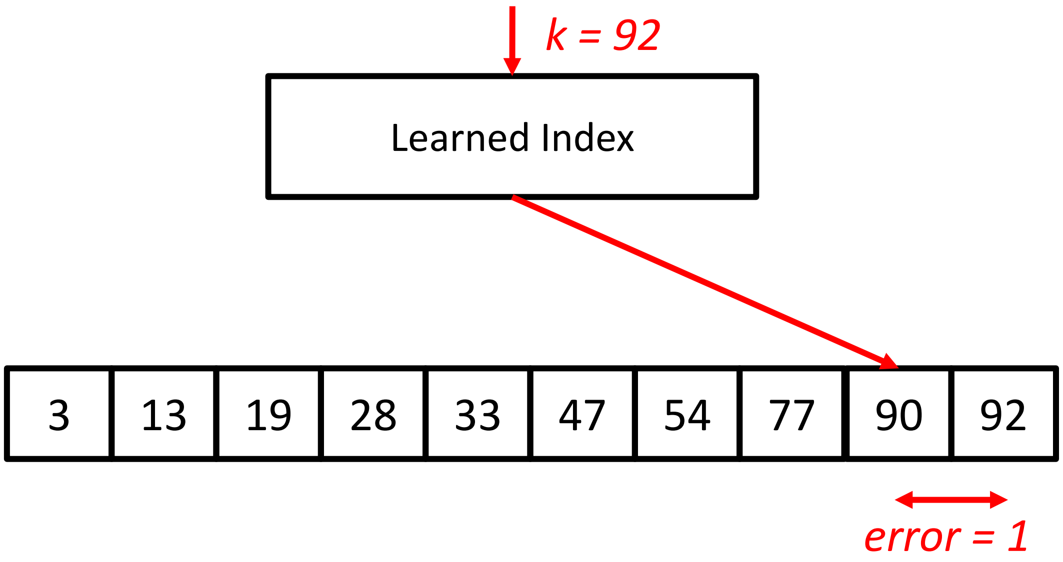


Last-Mile Accuracy



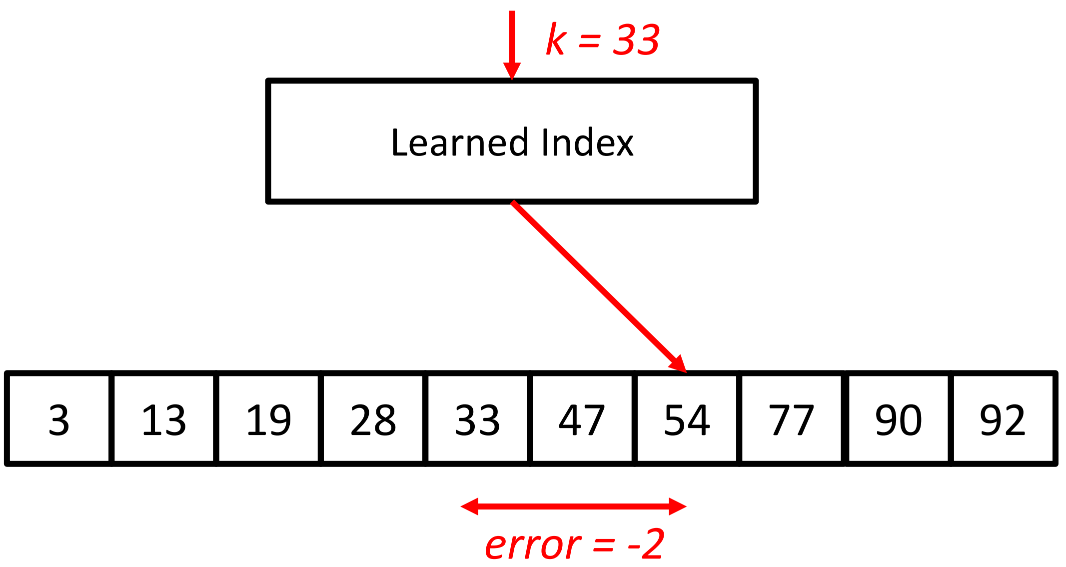
Min Max Errors

- Execute the model for every key and remember the worst over- and under-prediction of a position after the learned index is fixed



Min Max Errors

- Execute the model for every key and remember the worst over- and under-prediction of a position after the learned index is fixed

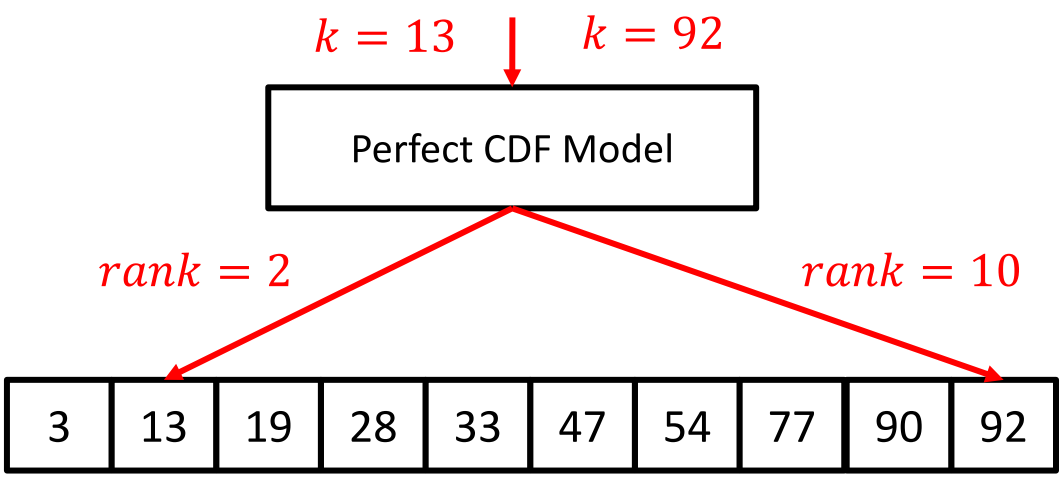


Learning Objective

- What should be learned to predict the positions of keys?
- Learn the data distribution, or more precisely, the CDF

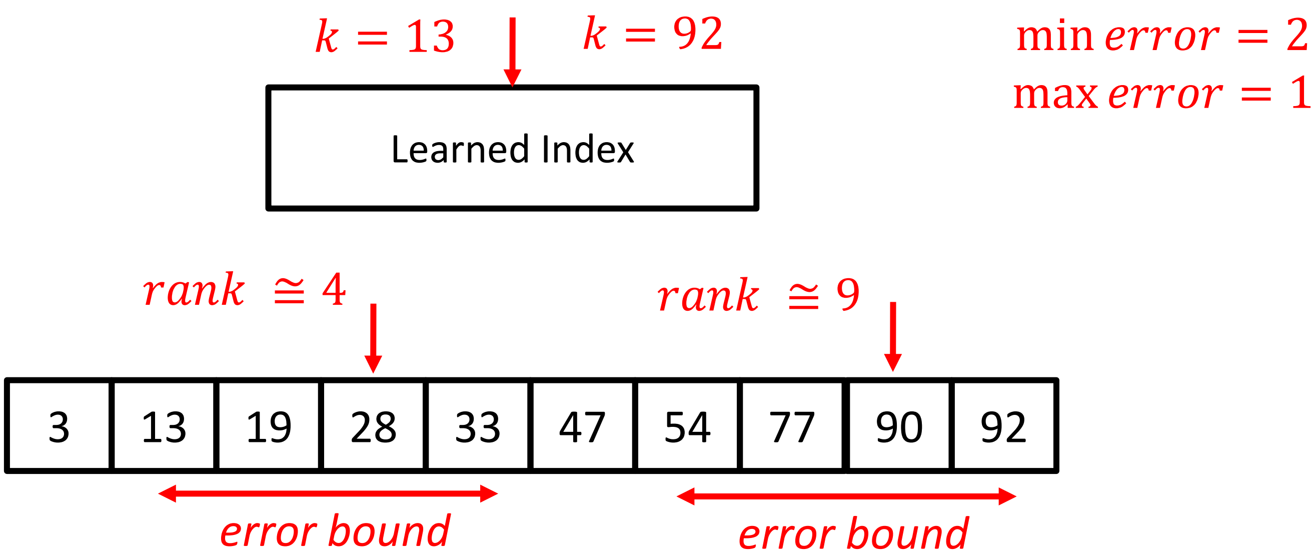
CDF

- Cumulative distribution function
- Consider a model that perfectly learn the CDF
  - given a key *k*, our model return the *exact rank* of the key among all the keys



Problem Formulation

- Input
  - a key *k*
- Output
  - the *estimated rank* of *k* among all the keys



Model Choice

- We want a model that is large enough to learn the data distribution
- We want a model that is small enough that has execution time comparably to B-tree index

A Naïve Learned Index

- 2 fully-connected layer with 32 neurons per layer
- B-tree index: 300 ns
- Learned index: 80000 ns
- The challenges
  - Tensorflow overhead
  - low last-mile accuracy

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

Tensorflow Overhead

- Tensorflow is more suited for large model than for small model
- Compile Tensorflow model into C++ program
  - can execute a small model in 30 ns

Low Last-Mile Accuracy

- If the error is too large, the last mile search would be very costly
- Key observation
  - reducing error to 100 from 100M is hard
  - reducing error to 10K from 100M is much easier; similarly, reducing error to 100 from 10K is much easier
- Recursive model index (RMI)

Recursive Model Index

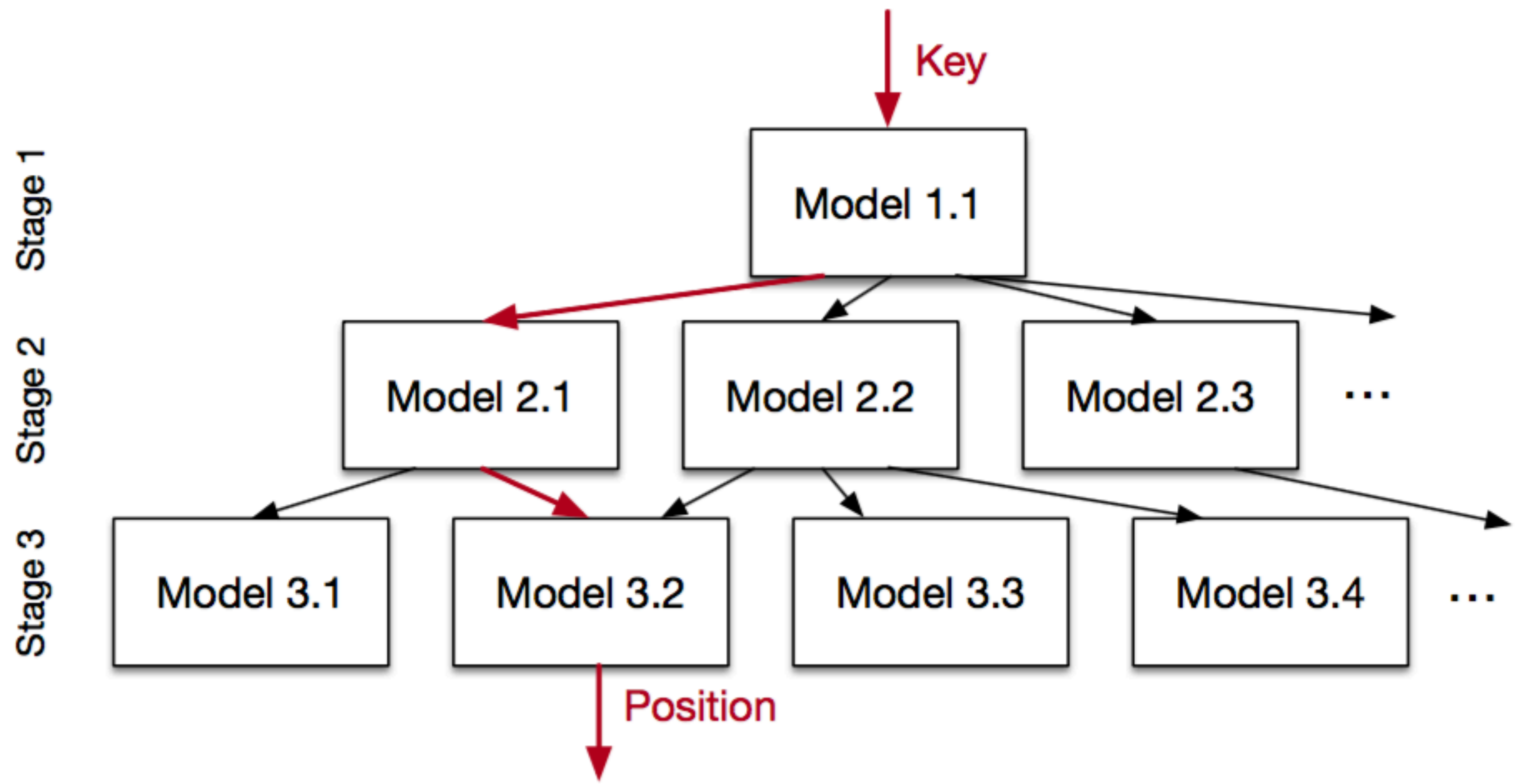
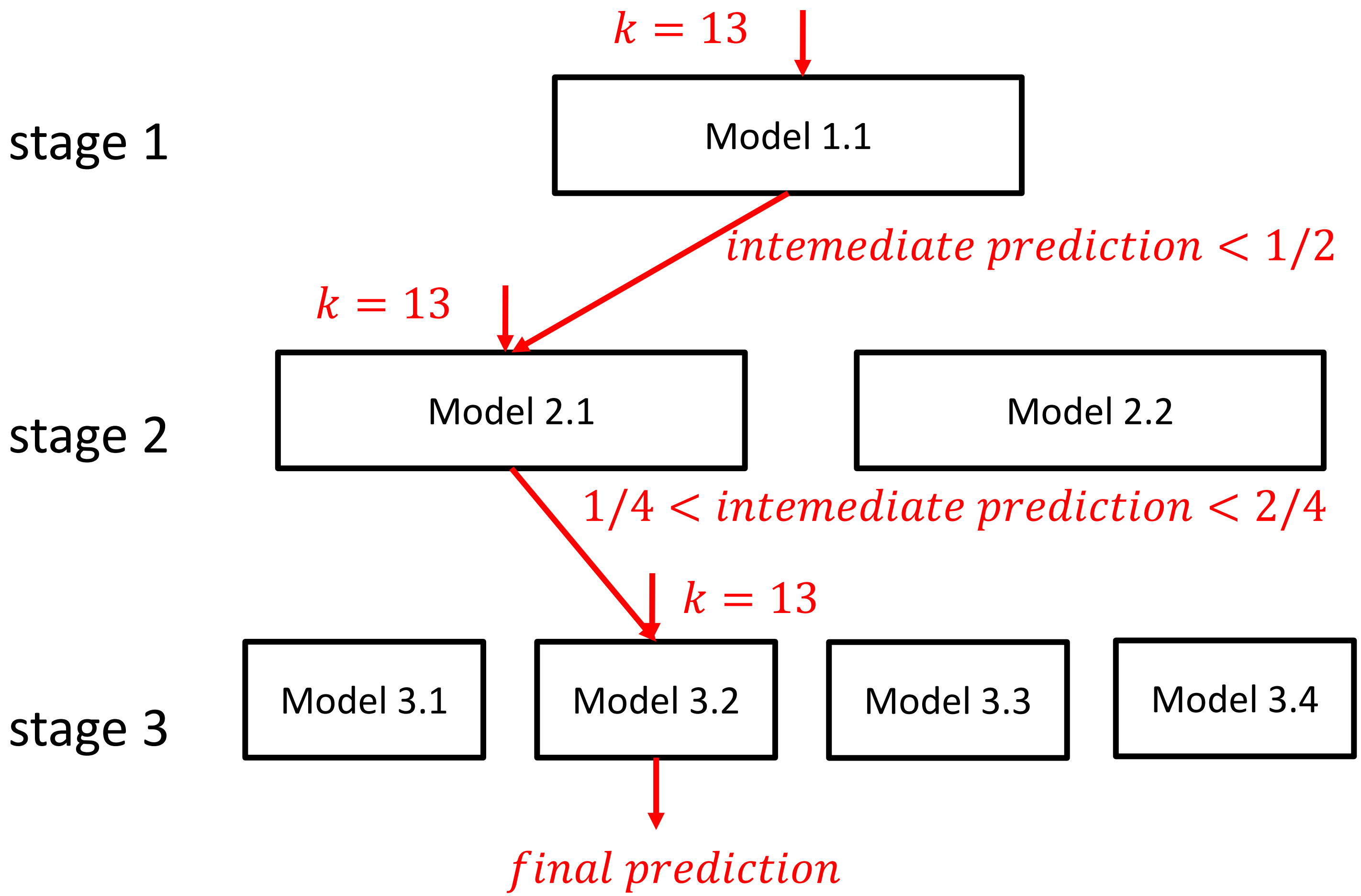


Figure 3: Staged models

An Example



$k = 13$
stage 1
Model 1.1
$intemediate\ prediction < 1/2$
$k = 13$
stage 2
Model 2.1
Model 2.2
$1/4 < intemediate\ prediction < 2/4$
$k = 13$
stage 3
Model 3.1
Model 3.2
Model 3.3
Model 3.4
$final\ prediction$

Hybrid Index

- In some cases, the model may fail to learn the data distribution well
- If the min- max-error is higher than a threshold, replace the model with B-tree

Result

Type	Config	Search	Total (ns)	Model (ns)	Search (ns)	Speedup	Size (MB)	Size Savings	Model Err ± Err Var.
Btree	page size: 16	Binary	280	229	51	6%	104.91	700%	4 ± 0
	page size: 32	Binary	274	198	76	4%	52.45	300%	16 ± 0
	page size: 64	Binary	277	172	105	5%	26.23	100%	32 ± 0
	page size: 128	Binary	265	134	130	0%	13.11	0%	64 ± 0
	page size: 256	Binary	267	114	153	1%	6.56	-50%	128 ± 0
Learned Index	2nd stage size: 10,000	Binary	98	31	67	-63%	0.15	-99%	8 ± 45
		Quaternary	101	31	70	-62%	0.15	-99%	8 ± 45
	2nd stage size: 50,000	Binary	85	39	46	-68%	0.76	-94%	3 ± 36
		Quaternary	93	38	55	-65%	0.76	-94%	3 ± 36
	2nd stage size: 100,000	Binary	82	41	41	-69%	1.53	-88%	2 ± 36
		Quaternary	91	41	50	-66%	1.53	-88%	2 ± 36
	2nd stage size: 200,000	Binary	86	50	36	-68%	3.05	-77%	2 ± 36
		Quaternary	95	49	46	-64%	3.05	-77%	2 ± 36
Learned Index Complex	2nd stage size: 100,000	Binary	157	116	41	-41%	1.53	-88%	2 ± 30
		Quaternary	161	111	50	-39%	1.53	-88%	2 ± 30

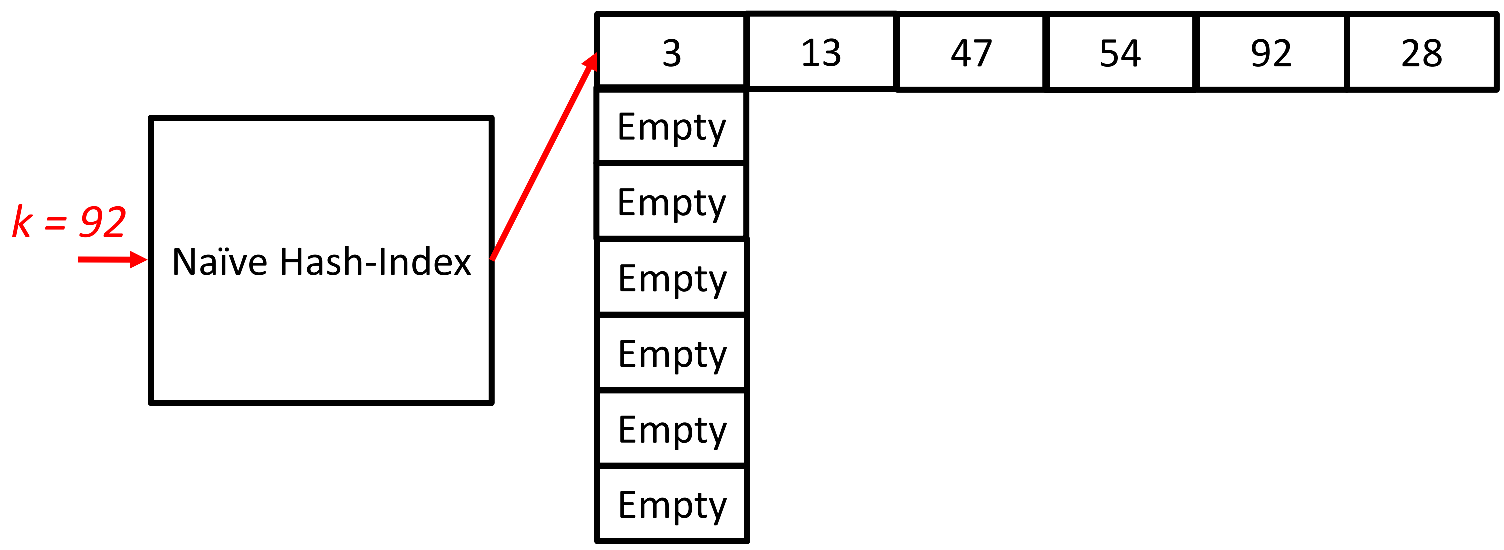
Figure 4: Map data: Learned Index vs B-Tree

Outline

- Range index
- Point index
- Existence index

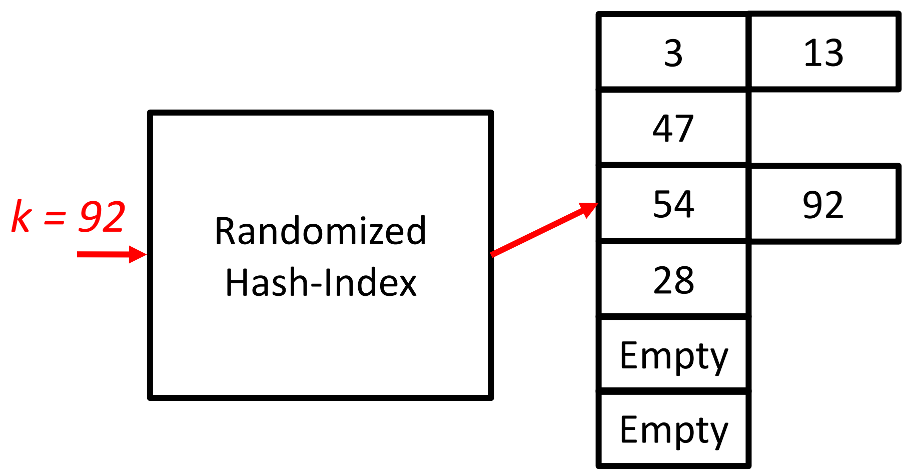
Naïve Hash-Index

- Consider a bad hash function that maps all the objects to the same slot



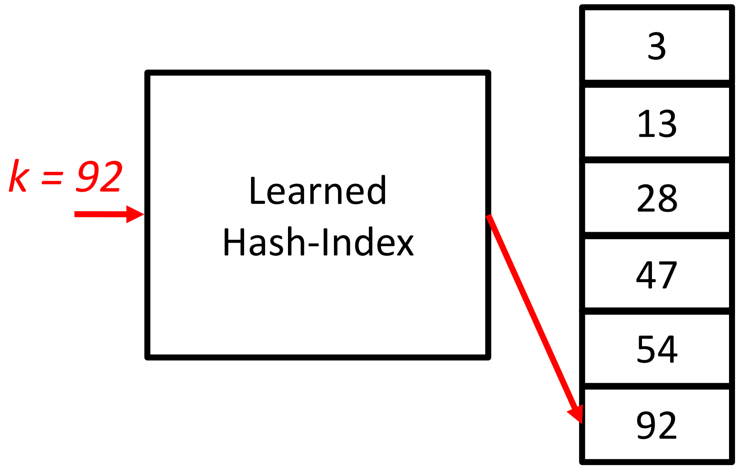
Randomized Hash-Index

- 2 multiplications, 3 bit shifts, 3 XORs
- Assume # of slots = # of records, collision rate is often around 33%



Learned Hash-Index

- CDF as hash function
- If CDF is perfectly learned: no collision



Result

Dataset	Slots	Hash Type	Search Time (ns)	Empty Slots	Space Improvement
Map	75%	Model Hash	67	0.63GB (05%)	-20%
		Random Hash	52	0.80GB (25%)	
	100%	Model Hash	53	1.10GB (08%)	-27%
		Random Hash	48	1.50GB (35%)	
	125%	Model Hash	64	2.16GB (26%)	-6%
		Random Hash	49	2.31GB (43%)	
Web Log	75%	Model Hash	78	0.18GB (19%)	-78%
		Random Hash	53	0.84GB (25%)	
	100%	Model Hash	63	0.35GB (25%)	-78%
		Random Hash	50	1.58GB (35%)	
	125%	Model Hash	77	1.47GB (40%)	-39%
		Random Hash	50	2.43GB (43%)	
Log Normal	75%	Model Hash	79	0.63GB (20%)	-22%
		Random Hash	52	0.80GB (25%)	
	100%	Model Hash	66	1.10GB (26%)	-30%
		Random Hash	46	1.50GB (35%)	
	125%	Model Hash	77	2.16GB (41%)	-9%
		Random Hash	46	2.31GB (44%)	

Figure 10: Model vs Random Hash-map

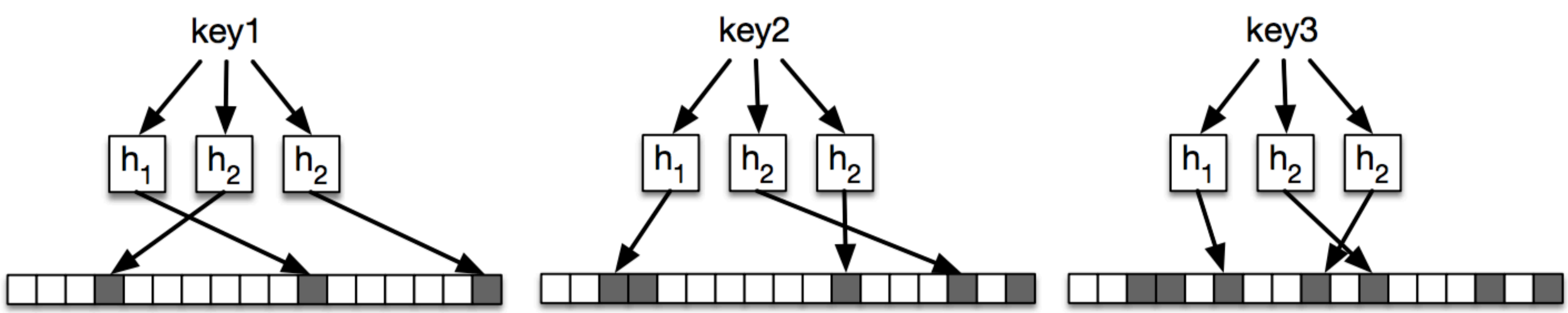
Outline

- Range index
- Point index
- Existence index

Bloom Filter

- Bit array of size m and k hash functions
- Insertion: a key is fed to the k hash-functions and the bits of the returned positions are set to 1
- Query: If any of the bits at those k positions is

(a) Bloom-Filter Insertion



Learned Bloom Filters

- Bloom filters as a binary classification problem
- Input
  - key *k*
- Output
  - probability that record with key *k* exists

The Challenge

- Bloom filter allows false positive, but not false negative

Solution

- Define a threshold τ which we *believe* if $f(k) > \tau$, then *k* exist in the database
- Feed all keys in the model and build a bloom filter for those with probability less than τ

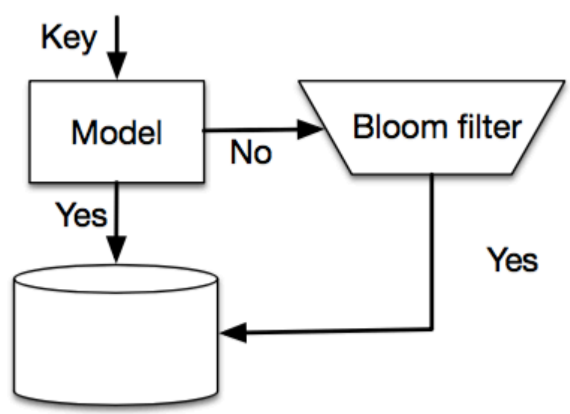


Figure 11: Bloom filters as a classification problem

Choosing τ

- Observe that as τ decreases, the false positive rate increases; meanwhile, the size of bloom filter decreases
- Given a target FPR, tune τ to achieve the target FPR

Result

- In contrast to learned range indexes and point indexes that aim to improve the performance, learned existence indexes aim to reduce the size of bloom filter
- 47% reduction in bloom filter size with the same false positive rate

Lab 1 – Learned Index Structures

- Build a 2-stage recursive model index
- Simulate with python, numpy and Tensorflow
- Both synthetic and real-work workloads will be provided