Learned Indexes

- Motivation: use ML models in place of data structures and algorithmic tasks
- A general framework for learned indexes
- A practical and updateable learned index
- Multidimensional learned indexes

Learned Indexing Framework

Reading: Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, Neoklis Polyzotis: The Case for Learned Index Structures. SIGMOD Conference 2018: 489-504

Learned Indexing: Motivation

- General trend: replace and augment traditional algorithms with machine learning (ML) models
  - e.g. recommender systems
- Key idea: Indexes are “models” that predict the position of a key in a dataset

An Extreme Example

- Suppose that we want to index an array of 100M consecutive integers

  0 1 2 3 4 5 6 7 … 99999999

- We would not use a B+-tree, because the position of a number implies it value!
- This is an extreme example of using a model to index, i.e., a learned index

$$f(x) = x$$

model function

key

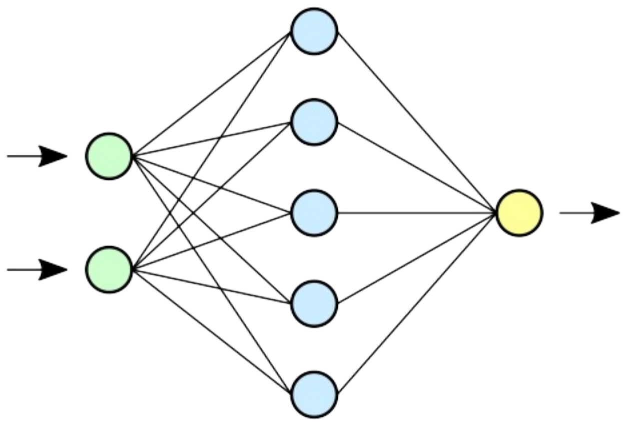
key position estimate

- Search cost: O(1), space complexity: O(1)

Practical Issues

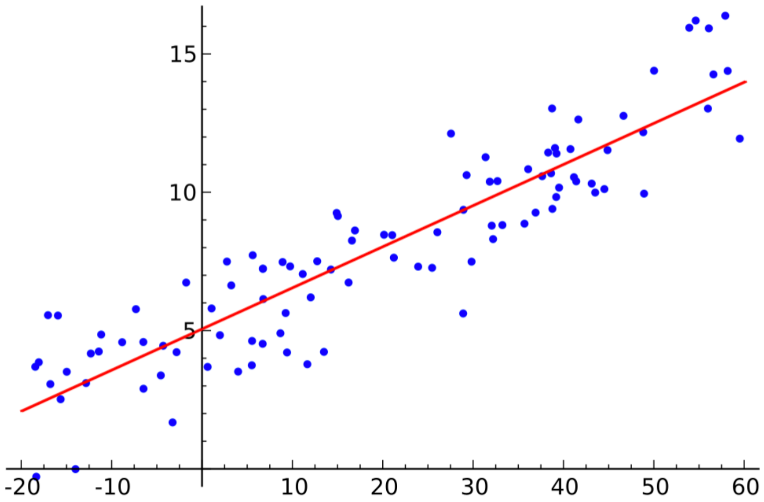
- The data do not always have a distribution that can be captured by a single model
- The effort to learn the data distribution can be high
- The data distribution may change
  - dynamic data
- Goal of Learned Indexing: Use ML to define models (e.g. neural networks) that can enhance or replace, traditional index structures (e.g. B^+-Trees, hash indexes, Bloom filters)

Black-box vs. Glass-box Models



Deep Neural Networks

(black-box)



Linear regression model

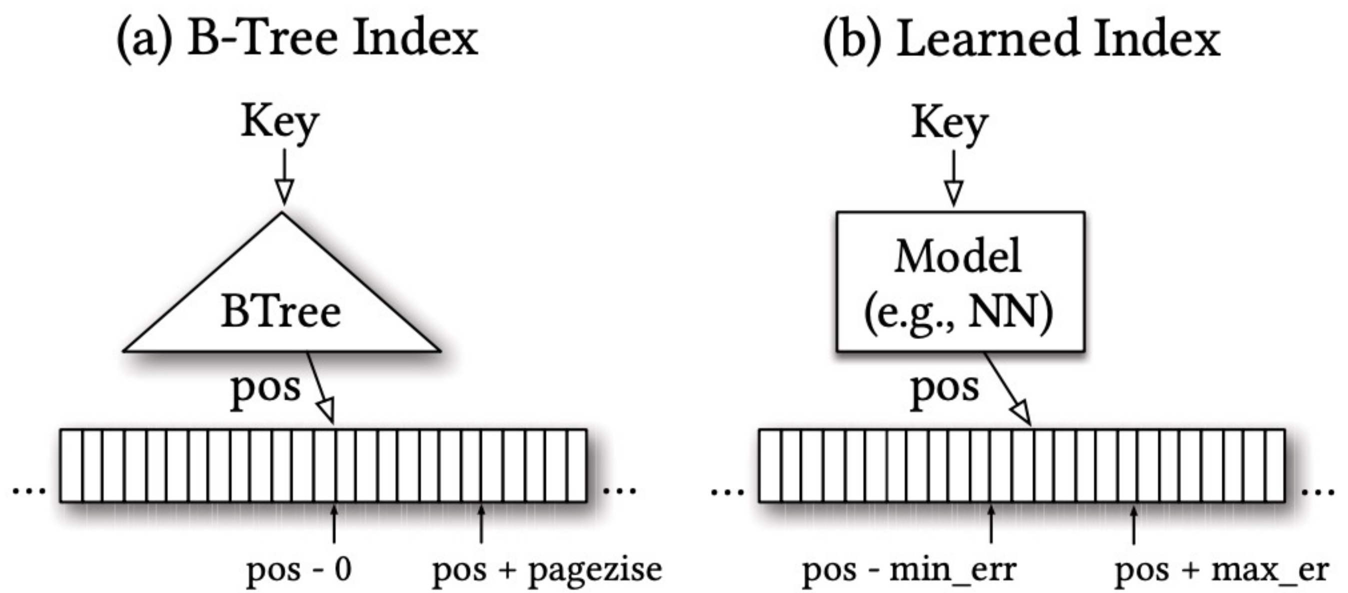
(glass-box)

Existing Indexes as Models

- B^+-tree:
  - The sequence of leaf nodes can be thought of as a sorted array
  - Non-leaf nodes can be thought of as models that predict the position of a key in the array
- Bloom filter:
  - Think of it as a binary classifier: predicts if a key exists in a set or not
- Important differences:
  - B+-tree does not have errors
  - Bloom filter can have false positives but no false negatives

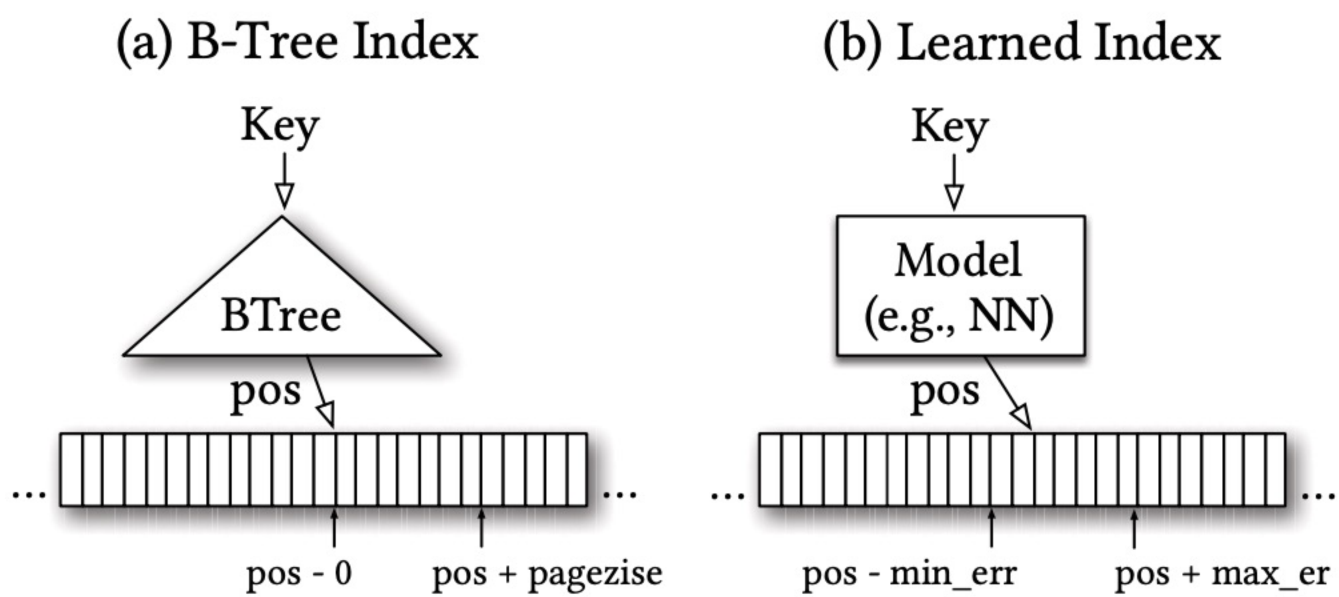
Learned Index for Range Searching

- Based on the B^+-tree
- Assumes static data, where <key,rid> pairs are sorted in an array
  - Leaf nodes of the tree are pieces in the array
- Replaces inner tree nodes by model(s)



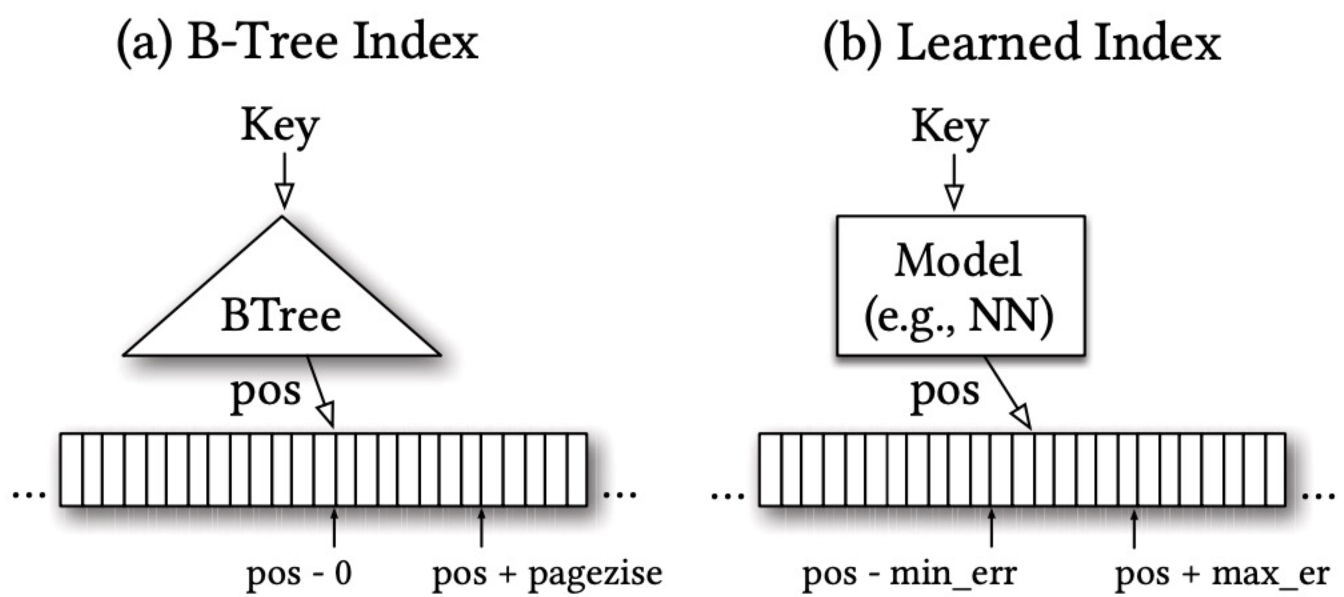
Learned Index for Range Searching

- The B+-tree is a regression tree model
  - maps a key to a position with a min_error=0 and max_error=page_size
  - guarantees that the key can be found in that region if it exists



Learned Index for Range Searching

- No need to have a fixed error bound
  - any error easily corrected by local search around the prediction (e.g., exponential search)
- Exploit modern hardware and ML accelerators

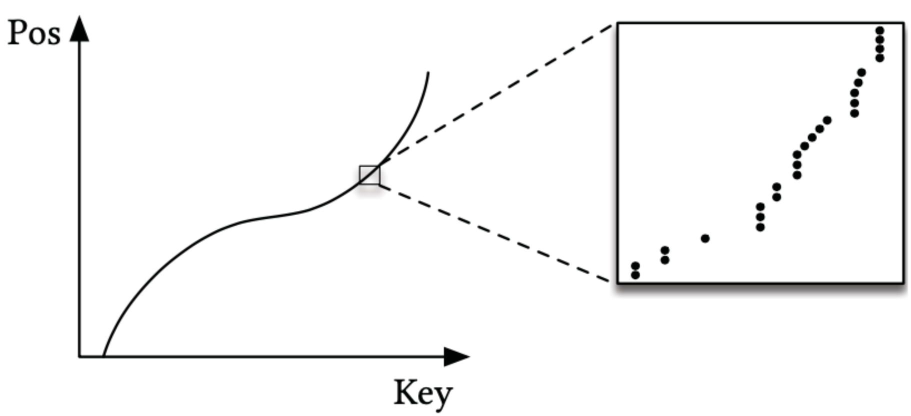


Range Index Models as CDF Models

- Model input: key
- Model output: position of key in the array
- Observation: a model for the position of a key in a sorted array approximates the cumulative distribution function (CDF)

$$p = F(\text{key}) * N$$

key position estimate — CDF $P(x \leq \text{key})$ — number of records

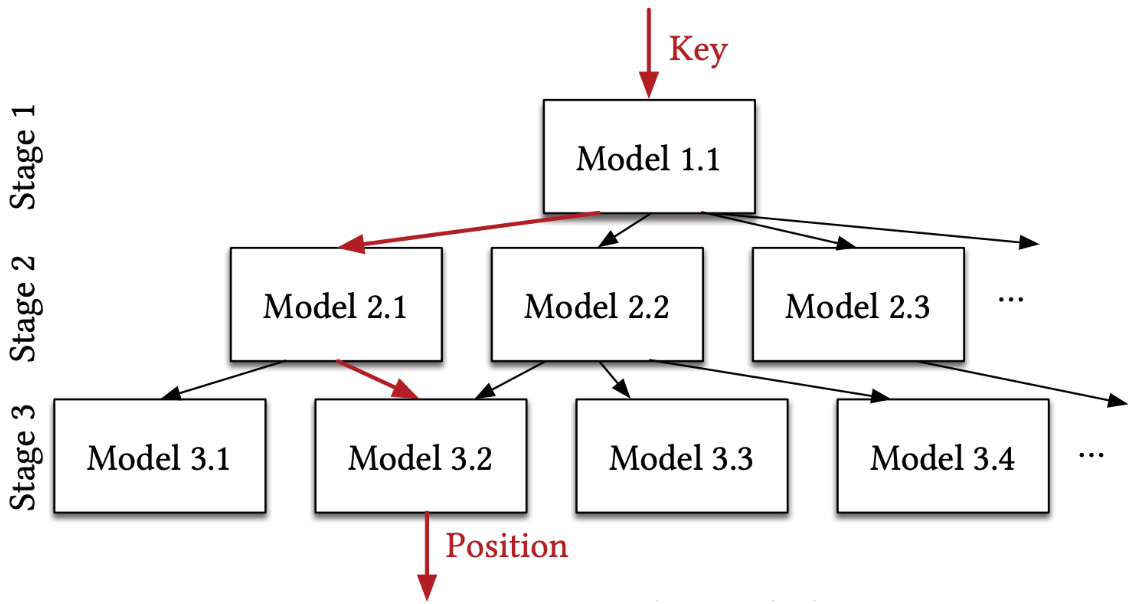


Range Index Models as CDF Models

- Possible solution: Model the CDF
  - Train a linear regression model to learn the CDF
  - Minimizing the (squared) error of a linear function
  - Alternative: use a neural net
- Using a single model for the whole CDF will not work
  - Slow training, poor “last mile” accuracy

Recursive Model (RM) Index

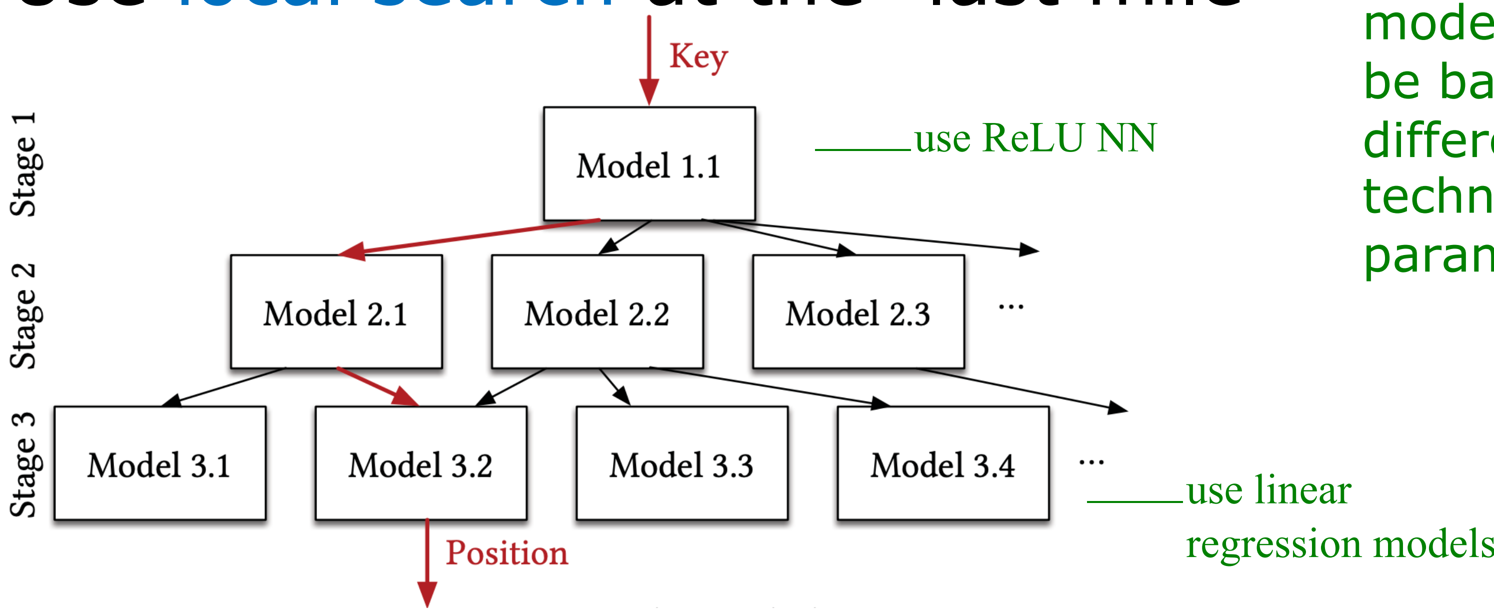
- A hierarchy of models
  - A non-leaf node in the hierarchy is a model that predicts which child model to use,
  - A leaf node is a model to predict the position of the key in an array



Recursive Model (RM) Index

- At each stage, a model takes the key as input and selects the model to use at the next level
  - Models need not form a tree
- Use local search at the "last mile"

Different models can be based on different techniques/ parameters



keep min- and max-error for every leaf model

Learned Index for Value Searching

- Hash-maps are effective for single-value search, but suffer from collisions
  - Hard to avoid bucket overflows
- ML has been used to find hash functions that avoid collisions
- Idea: Learning the CDF of keys can help to learn a good hash function
  - Scale the CDF by the targeted size M of the hash-map
  - Use h(key) = F(key)$*$M as hash function

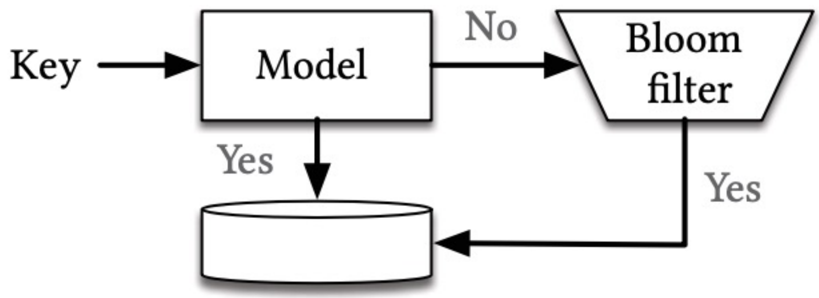
    CDF $P(x \leq \text{key})$
  - Use the RM-index for F
  - If CDF learning is accurate, collisions are avoided

Learned Index for Existence Search

- Search problem: test whether an element is a member of a set S
- Classic solution: Bloom filter
  - Use a bitarray of size m
  - Use k independent hash functions that map each key to one of the m positions
  - Use a single bitmap B for S; for each key in S, set B[$h_i(s)$] = 1, for each $i = 1,..,k$
  - For a search key x, apply all hash functions and get x's bitmap B_x
  - If B_x has 0 in any position where B has 1, x is not a member of S
  - Bloom filter guarantees there are no false negatives, but there may be false positives (x mistaken as a member)

Learned Index for Existence Search

- Issue: a good Bloom filter takes too much space
- Solution: Learned Bloom filter
  - Goal: learn a good hash function with lots of collisions among keys and lots of collisions among non-keys, but few collisions of keys and non-keys
  - Training needs not only S, but also a set U of non-keys
    - U obtained from past queries or could be random
  - Possible models: binary classifiers (e.g. RNN or CNN)
    - $D = \{(x_i, y_i = 1) | x_i \in S\} \cup \{(x_i, y_i = 0) | x_i \in U\}$
  - Choose a threshold τ; assume that x exists if $f(x) \geq \tau$
  - Create an "overflow" Bloom filter for false negatives from f

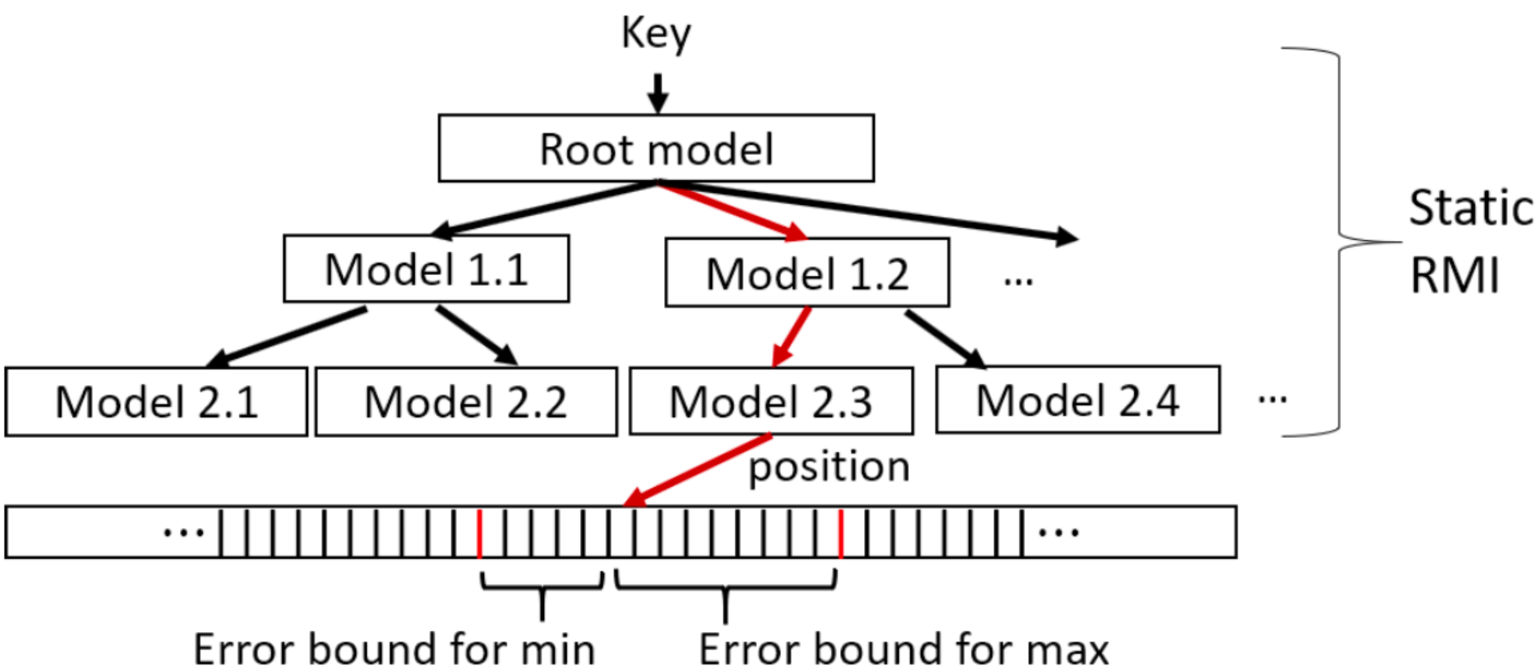


Adaptive Learned Index

Reading: J. Ding, U. F. Minhas, J. Yu, C. Wang, J. Do, Y. Li, H. Zhang, B. Chandramouli, J. Gehrke, D. Kossmann, D. B. Lomet, T. Kraska: ALEX: An Updatable Adaptive Learned Index. SIGMOD Conference 2020: 969-984

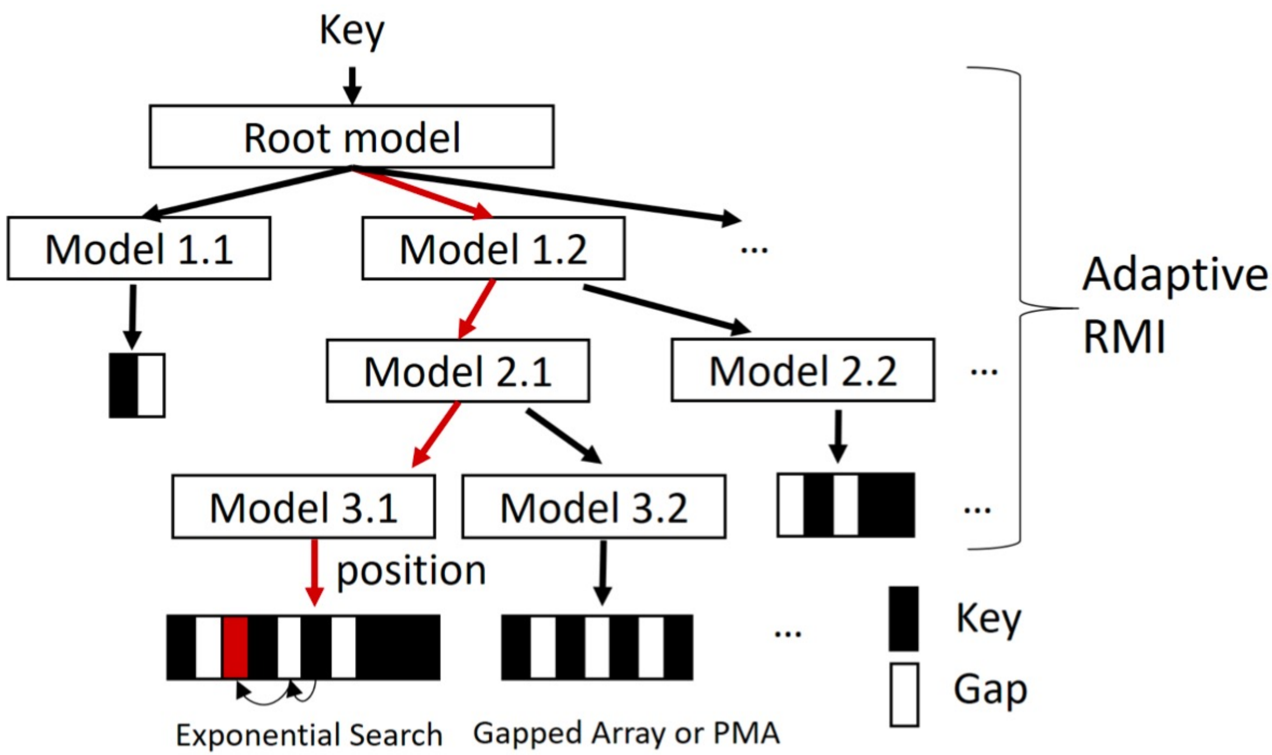
Drawback of RM-index

- Static depth and static models in the index
  - Models become ineffective as data are updated
- The keys are in a sorted static array
  - Updates are expensive



ALEX: Adaptive Learned Index

- Primary goal: support updates
- Differences to RM-index:
  - Dynamic leaves of different structure
  - Adaptive RM-index based on the workload



Performance goals of ALEX

- Insert time should be competitive with B^+-tree
- Search time should be faster than B^+-tree and similar to RM-index
- Index storage space should be similar to RM-index and much smaller than B^+-Tree
- Data storage space (leaf level) should be comparable to B^+-Tree
  - Low index storage space is important in cases where index storage is in (limited) memory and data storage on disk

Features of ALEX

- Use exponential search instead of binary search
- Use dedicated arrays for leaf nodes instead of a single sorted data array
- Use gaps in dedicated arrays to facilitate fast insertion
- Avoids partitions with very few keys and partitions fully-packed with keys
- Models of non-leaf nodes adapt as data are updated

Exponential vs. Binary Search

- ALEX uses exponential search without error bounds to find keys at the leaf level
  - RM-index uses binary search with error bounds
- Rationale: if the models are good, their prediction is close enough to the final position, so exponential search will be efficient
- ALEX does not need to store error bounds for RMI models, which saves space

Exponential search example:

search for 37, prediction at position of 14

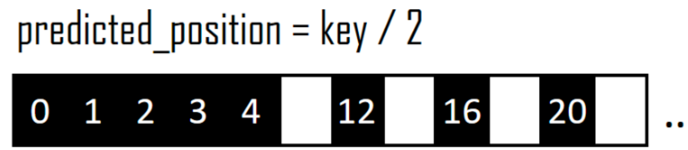
…,6, 8, 14, 17, 25, 37, 52, 63, 66, 80, …

Leaf Nodes vs. Single Array

- ALEX uses a dedicated array for each leaf
  - RM-index keeps all keys in a single sorted array
  - Insertions are very expensive
- Each leaf node is an array with gaps between elements to facilitate cheap insertions
- Two alternative array layouts for leaves: Gapped Array vs. Packed Memory Array

Gapped Array Leaf Node Layout

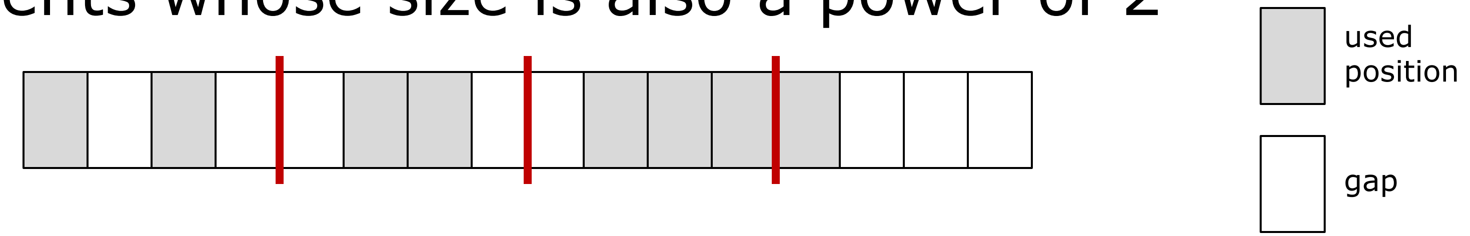
- Allow "naturally" distributed gaps between array elements



- To insert a new key
  - use the RMI to predict the insertion position
  - if a gap and inserting there keeps the sorted order, all done
  - else use exponential search to find correct position
    - If not a gap, make a gap by shifting keys and insert there
- O(log n) insertion time with high probability
- If numkeys/leafsize ≥ density bound d
  - expand leafsize by 1/d, re-train model for leaf
  - (must keep enough gaps to prevent shifting)
- Drawback of gapped array: a possible long continuous region with no gaps (expensive shifting)

Packed Memory Array (PMA) Layout

- Node size is a power of 2 and is divided to segments whose size is also a power of 2



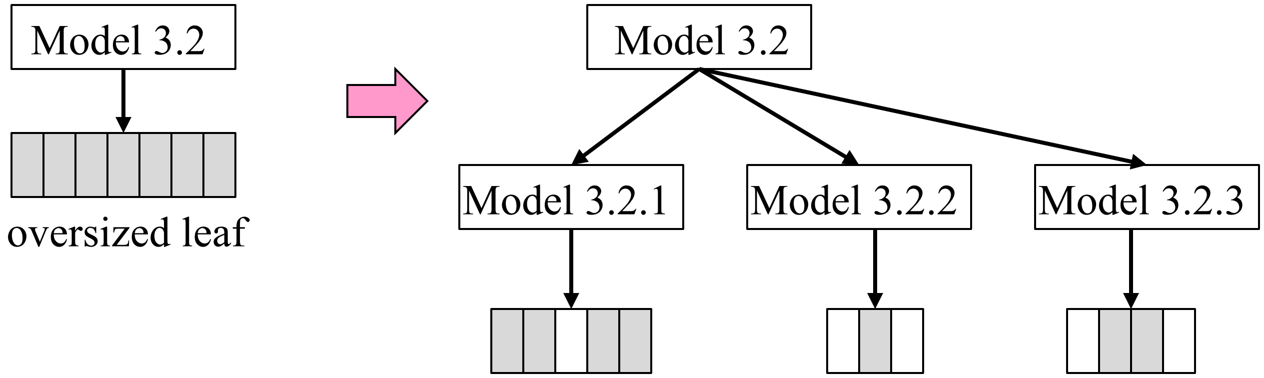
- An implicit binary tree on top of the array
  - Each node has a density bound for non-gaps
  - Nodes near leaves have higher density bounds
  - If insertion to a segment violates bound, redistribute keys to ensure no bound is violated
  - If no redistribution solves problem, double size of array
- Redistribution may affect model's accuracy
  - Perform model-based redistribution
  - Retrain model after node expansion

Adaptive RM-Index

- ALEX tries to avoid two problems
  - Wasted models: leaf nodes with very few keys
  - Fully-packed leaves: expensive insertions
- RM-index initialization in ALEX
  - Goal: adaptively determine RMI depth and leaves
  - Recursive building:
    - Train a model for root, evenly partition keys to root's children
    - Oversized partitions (in number of keys) are recursively split
    - Each non-root node has a fixed number of partitions
    - Leaves with very few keys are progressively merged with siblings

Adaptive RM-Index (cont'd)

- RM-index updates in ALEX
  - Goal: adapt RMI to changes
  - Node splits: if a leaf becomes oversized, it is split to a number of children (new leaves)
    - The model for the split leaf becomes the model for the new parent
    - Data from split leaf node distributed to the new (children) leaves
    - New models are trained for the created children



Multidimensional Learned Indexes

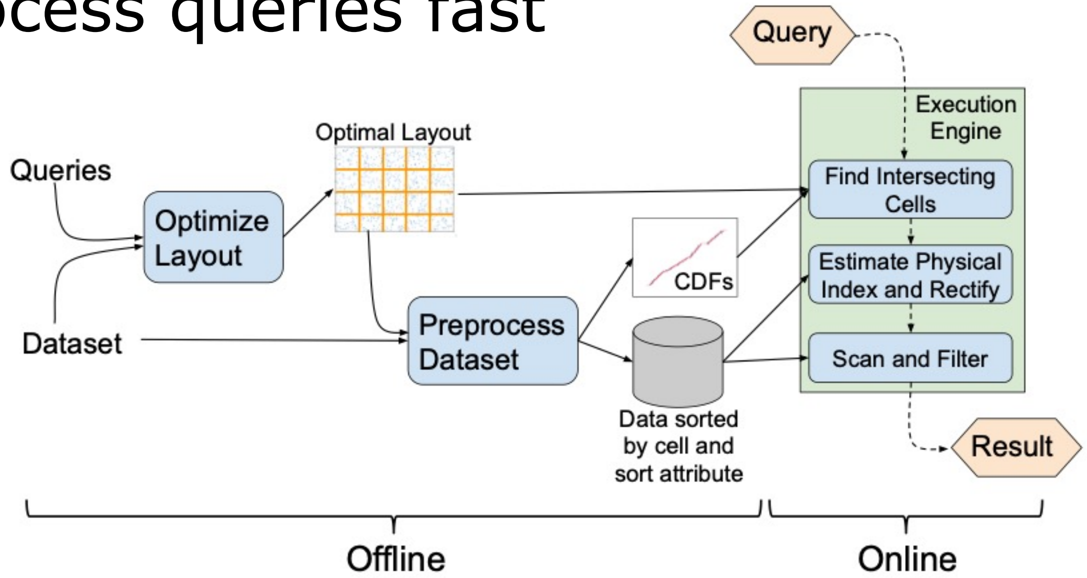
Reading: Vikram Nathan, Jialin Ding, Mohammad Alizadeh, Tim Kraska: Learning Multi-Dimensional Indexes. SIGMOD Conference 2020: 985-1000

Multidimensional Indexes

- Objective: index multiple attributes simultaneously
- Challenge: data cannot be totally ordered
- There is room for improving existing multidimensional indexes using ML models, such as in the 1D case
- Possible solution: define a 1D order of the data (e.g. z-order) and then use a 1D learned index
  - Issue: z-order CDFs are hard to learn

A Learned Multidimensional Index

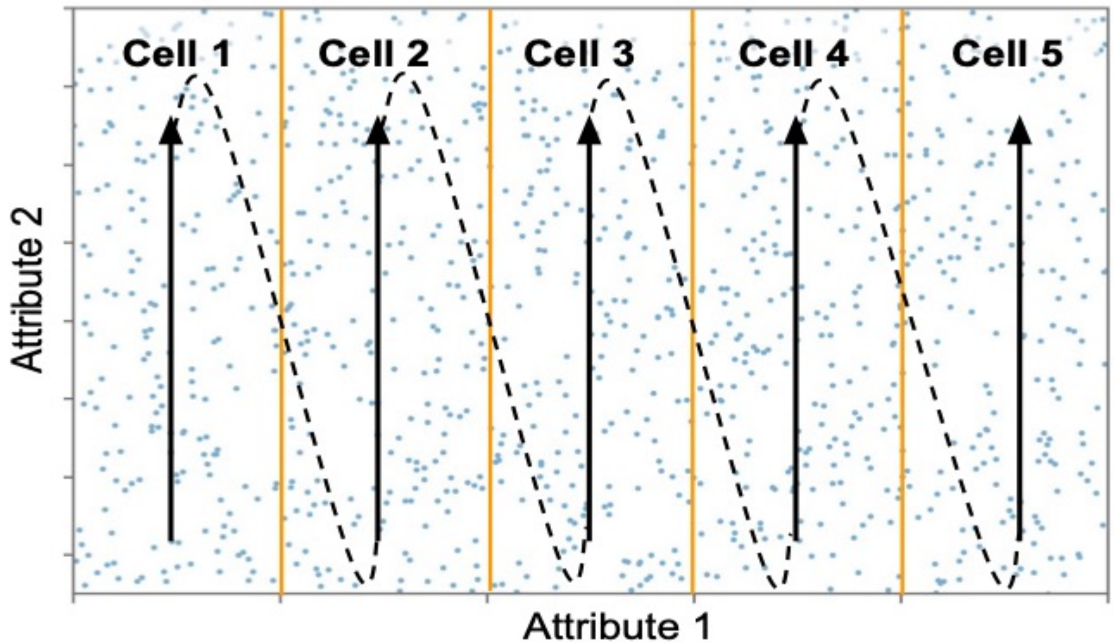
- Flood is a multi-dimensional learned index that adapts to the query workload
  - Designed mostly for static data
  - Suitable for multidimensional range queries
  - Created in a preprocessing phase, then used to process queries fast



Flood: preprocessing

- Define an order on the d dimensions
- Use the first d-1 dimensions to define a (d-1)-dimensional grid on the data space
  - Dimension i is divided to c_i equi-sized partitions
  - ML model is used to determine partitioning
- Each data point is mapped to a grid cell
  - Point coordinates and ML models are used to locate cell
- Cells are sorted in based on a dimension ordering
- In each cell, points are sorted based on d-th dimension
- Flood uses ML and query workload to find the best dimension ordering and grid granularity

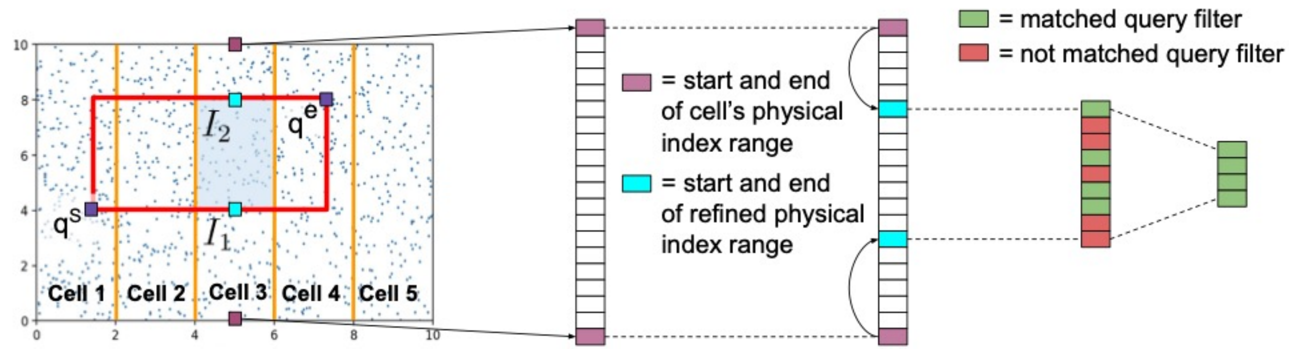
Flood: 2D data example



- Dimension order (x, y) and $c_0 = 5$
- Points are partitioned to columns along x and then sorted by their y-values
- The serialization order is indicated by the arrows

Flood: query processing

- Find cells that overlap with multidimensional query range, using the ML models
- Use point ordering in each cell to confine the range of points to be scanned
  - Done using a piecewise linear model
- Scan range and filter



(1a) Projection finds 4 intersecting cells (cells 1-4) → (1b) Identify physical index range of third cell. (Repeat for other cells.) → (2) Refine the physical index range → (3) Scan and Filter

Summary

- Learned indexes aim at reducing the memory footprint of indexing while also enabling fast data accesses
- Performance depends on how well the ML models fit the data distribution
- Dynamic modeling and gapped array storage is proposed for adaptive indexing
- Multidimensional learned indexes partition data, then create a 1D sort order at each partition