Learning Data System Components

6.830 Lecture 11
Sivaprasad Sudhir, Kapil Vaidya

Slides Courtesy: Prof. Tim Kraska
Building Blocks of Data Systems

- Indexes
- Joins
- Sorting
- Caching
- Hash Tables
An example

- Index all Integers from 100 to 1M

Data = 100 101 102 103 104 105 ... 1M

B-Tree?
An example

- Index all Integers from 100 to 1M

\[
\text{Data} = \begin{array}{cccccc}
100 & 101 & 102 & 103 & 104 & 105 & \ldots & 1M
\end{array}
\]

\[\text{Data[key – 100]}\]
An example

• What about even integers?

Data = 100 102 104 106 108 110 ... 1M

Data = 100 102 104 106 108 110 ... 1M

Data[(key – 100) / 2]
Other distributions?
Key Insight

• Traditional data structures make no assumption about data
• Optimized for worst case
• Building a system from scratch for every use case is not scalable
• What if we could learn the data distribution?
Setup

• In-memory
• Read only
B-Tree as a model

- Maps key to page
B-Tree as a model

- Maps key to pos
- Search between pos and pos + page size
B-Tree as a model

• Model that predicts the position of a key within some error bounds
Index as a model

- Model maps key to pos
- Search in $[\text{pos} - \text{err}_{\text{min}}, \text{pos} + \text{err}_{\text{max}}]$.
- $\text{err}_{\text{min}}$ and $\text{err}_{\text{max}}$ are known from training.
What is the model estimating?

- Predicting the position of a key
- Modeling the CDF of the keys
- \( \text{pos} = P(X \leq \text{key}) \times \#\text{keys} \)
What is the model estimating?

- Predicting the position of a key
- Modeling the CDF of the keys
- \( \text{pos} = P(X \leq \text{key}) \times \#\text{keys} \)
  
  \[ = F(\text{key}) \times \#\text{keys} \]
LookUp(key)

- Use the CDF model to predict the position of the key
- \( \text{pos} = F(\text{key}) \times \#\text{keys} \)
- Scan from \([\text{pos} - \text{errmin}, \text{pos} + \text{errmax}]\)
Key Idea

• When CPU cycles are cheap relative to memory accesses, compute-intensive function approximations can be beneficial for lookups
• ML models may be better at approximating some functions than existing data structures
• Power of continuous functions
Potential Advantages of Learned Models

• Smaller indexes
• Faster lookups
What Models?

• 200M web-server log records by timestamp-sorted
• 2-layer NN, 32 width, ReLU activated
• Prediction task: timestamp -> position within sorted array
Naïve Approach

250 ns

TensorFlow

???
Naïve Approach

250 ns

80,000 ns
Problems

1. Tensorflow is designed for large models
2. B-Trees are very good at overfitting
3. B-Trees are cache-efficient
4. Search does not take advantage of the prediction
Precision gain per node

Index over 100M records. Page-size: 100

Precision Gain: 100M --> 1M  
(Min/Max-Error: 1M)

Precision Gain: 1M --> 10k

Precision Gain: 10k --> 100

100M records (i.e., 1M pages)
Last Mile Problem
Recursive Model Index (RMI)

Fl.k - CDF of Ml.k
Ml – Number of models in level l
N – Number of keys

floor(F1.1(key) * M2) = 0
floor(F2.1(key) * M3) = 1
floor(F3.2(key) * N) = pos
Recursive Model Index (RMI)
Min-/Max-Error vs Average Error
Binary Search

Predicted Position

Actual Position

0

Left

Middle

Right

N
Binary Search
Binary Search
Exponential Search
Results

Updates?


Slides Courtesy: Jialin Ding
• Why are inserts hard?
• How do B-Trees handle inserts?
ALEX

• Goals
  • Writes competitive with B+ Tree
  • Reads faster than B+ Tree and Learned Index
  • Index size smaller than B+ Tree and Learned Index

• Core structure
  • Dynamic tree structure
  • Models
# ALEX Core Ideas

<table>
<thead>
<tr>
<th></th>
<th>Faster Reads</th>
<th>Faster Writes</th>
<th>Adaptiveness</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Gapped Array</td>
<td>✔️</td>
<td>✔️</td>
<td></td>
</tr>
<tr>
<td>2. Model-based Inserts</td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3. Adaptive Tree Structure</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
</tbody>
</table>
1. Gapped Array

Dense Array

1 2 3 4 5 6 7 8
1. Gapped Array

Dense Array
1. Gapped Array

Dense Array
1. Gapped Array

Dense Array

```
1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8
```
1. Gapped Array

Dense Array

0 1 2 3 4 5 6 7 8
1. Gapped Array

Dense Array

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

Insertion Time

$O(n)$
1. Gapped Array

<table>
<thead>
<tr>
<th>Dense Array</th>
<th>Insertion Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7 8</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B+ Tree Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 4 5 6 7 8</td>
</tr>
</tbody>
</table>
1. Gapped Array

Dense Array

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

B+ Tree Node

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

Insertion Time

$O(n)$
1. Gapped Array

<table>
<thead>
<tr>
<th>Dense Array</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
</table>

Insertion Time: $O(n)$

| B+ Tree Node | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
1. Gapped Array

<table>
<thead>
<tr>
<th>Dense Array</th>
<th>Insertion Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7 8</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B+ Tree Node</th>
<th>Insertion Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7 8</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>
1. Gapped Array

<table>
<thead>
<tr>
<th>Array Type</th>
<th>Insertion Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dense Array</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>B+ Tree Node</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Gapped Array</td>
<td></td>
</tr>
</tbody>
</table>

Dense Array: 0 1 2 3 4 5 6 7 8

B+ Tree Node: 0 1 2 3 4 5 6 7 8

Gapped Array: 1 2 3 4 5 6 7 8
1. Gapped Array

<table>
<thead>
<tr>
<th>Insertion Time</th>
<th>Dense Array</th>
<th>B+ Tree Node</th>
<th>Gapped Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O(n)$</td>
<td><img src="image" alt="Dense Array" /></td>
<td><img src="image" alt="B+ Tree Node" /></td>
<td><img src="image" alt="Gapped Array" /></td>
</tr>
</tbody>
</table>
1. Gapped Array

<table>
<thead>
<tr>
<th>Dense Array</th>
<th>0 1 2 3 4 5 6 7 8</th>
<th>$O(n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>B+ Tree Node</td>
<td>0 1 2 3 4 5 6 7 8</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Gapped Array</td>
<td>0 1 2 3 4 5 6 7 8</td>
<td></td>
</tr>
</tbody>
</table>
# 1. Gapped Array

<table>
<thead>
<tr>
<th>Array Type</th>
<th>Insertion Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dense Array</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>B+ Tree Node</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Gapped Array</td>
<td>$O(\log n)$</td>
</tr>
</tbody>
</table>
1. Gapped Array

Gapped Array achieves inserts using fewer shifts, leading to faster writes.
2. Model-based Inserts

Gapped Array

1 2 3 4 5 6 7 8
2. Model-based Inserts

Model

Gapped Array

Key 1 2 3 4 5 6 7 8

1 2 3 4 5 6 7 8
2. Model-based Inserts

Model

Gapped Array
2. Model-based Inserts

Model

Gapped Array

Key: 1, 2, 3, 4, 5, 6, 7, 8

1
2. Model-based Inserts

Model

Gapped Array
2. Model-based Inserts

Model

Gapped Array
2. Model-based Inserts

Key

Model

Gapped Array
2. Model-based Inserts

Model

<table>
<thead>
<tr>
<th>Key</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
</table>

Gapped Array

1 2 3 4 5
2. Model-based Inserts

Model

Gapped Array
Model-based inserts achieve lower error, leading to faster reads

(a) Learned Index

(b) ALEX
3. Adaptive Structure

- Flexible tree structure
  - Split nodes sideways
  - Split nodes downwards
  - Expand nodes
  - Merge nodes, contract node

- All decisions are made to maximize performance
  - Uses a cost model of query runtime
Learned Data Structures

• A lot more work on different data structures
  • Multi-dimensional indexes
  • Hash Tables
  • Bloom Filters
  • Suffix Trees

• Different models to learn CDFs
Study Break

- How to extend the CDF idea to algorithms like sorting?
- Can you frame sorting as a prediction task?
Sorting
Sorting

• Puts elements of a list in a certain order.
Sorting

• Puts elements of a list in a certain order.
• Numerical order

\[
A = \begin{bmatrix}
9 & 1 & 17 & 18 & 3 & 0 & 1 & 4 \\
\end{bmatrix}
\]

\[
\text{Sort}(A) = \begin{bmatrix}
0 & 1 & 1 & 3 & 4 & 9 & 17 & 18 \\
\end{bmatrix}
\]
Sorting

• Puts elements of a list in a certain order.

• Numerical order

<table>
<thead>
<tr>
<th>A =</th>
<th>9</th>
<th>1</th>
<th>17</th>
<th>18</th>
<th>3</th>
<th>0</th>
<th>1</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sort(A) =</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>9</td>
<td>17</td>
<td>18</td>
</tr>
</tbody>
</table>

• Lexicographic order

<table>
<thead>
<tr>
<th>A =</th>
<th>uvw</th>
<th>aaa</th>
<th>bbc</th>
<th>efg</th>
<th>gjl</th>
<th>abc</th>
<th>cba</th>
<th>ghe</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sort(A) =</td>
<td>aaa</td>
<td>abc</td>
<td>bbc</td>
<td>cba</td>
<td>efg</td>
<td>ghe</td>
<td>gjl</td>
<td>uvw</td>
</tr>
</tbody>
</table>
Sorting

• Puts elements of a list in a certain order.
• Numerical order

\[
A = \begin{bmatrix}
9 & 1 & 17 & 18 & 3 & 0 & 1 & 4 \\
\end{bmatrix}
\]

Sort(A) = \[
\begin{bmatrix}
0 & 1 & 1 & 3 & 4 & 9 & 17 & 18 \\
\end{bmatrix}
\]

• Lexicographic order

\[
A = \begin{bmatrix}
uvw & aaa & bbc & efg & gjl & abc & cba & ghe \\
\end{bmatrix}
\]

Sort(A) = \[
\begin{bmatrix}
aaa & abc & bbc & cba & efg & ghe & gjl & uvw \\
\end{bmatrix}
\]

• Complex data types such as multi-dimensional, categorical etc have their own specific sort order.
Comparison based sorting algorithms
Comparison based Sorting

• Use a comparison function that determines which of two elements should occur first in the final sorted list.
• Compare two elements and then swap if needed.
• Some popular sorting functions: Quick Sort, Heap Sort, Insertion Sort, Tim Sort, Bubble Sort, Selection Sort
Bubble Sort

1. Compare each pair of adjacent elements
2. Swap the two if necessary
3. Repeat Until array is sorted
Bubble Sort Example

First pass

Next pass

Next pass

2 4 6 8 10
Insertion Sort

• Just like sorting a deck of cards.

1. Maintain a sorted deck of cards seen until now.
2. For a new card, find the position in the sorted deck
3. Place the card in the position by shifting the following cards
4. Repeat Until no new incoming card
Insertion Sort Example

Quite fast if numbers are close to actual positions (nearly sorted array)!!
Complexity of comparison-based algorithms

• Number of comparisons required is at least $O(N \log(N))$.

• Proof:
  • Let's say sorting algorithm makes at most $D$ comparison function calls every run.
  • There are $N!$ different arrays that we can feed to this algorithm.
  • Every different order feed should result in a new set of $D$ decisions.

$$N! \leq 2^D$$
Complexity of comparison-based algorithms

• The number of comparisons required is proportional to $O(N \log(N))$.
• Proof:
  • Lets say sorting algorithm makes atmost $D$ comparison function calls every run.
  • There are $N!$ different arrays that we can feed to this algorithm.
  • Every different order feed should result in new set of $D$ decisions.

$$N! \leq 2^D$$

Is $N \log(N)$ the best we can do?
Question

• Given a 1 million sized integer array, if you knew the integers are between 1-100, how would you sort the array?
Question

• Given a 1 million sized integer array, if you knew the integers are between 1-100, how would you sort the array?
• Use a 100 sized array to count occurrences of each value
Distribution based sorting algorithms
Distribution based algorithms

• General idea of distribution-based sorting algorithms:
  "Build a histogram of the data and place the data using it"

• Easy to build a histogram if range of the data is limited
Counting Sort

• Useful when the data range is small.

1. Initialize an empty count array for the data range
2. Parse the array and increment count of value seen
3. This generates a histogram of the data
4. Calculate the CDF of each value using the histogram
5. Parse the array and place elements based on their CDF
6. Decrement the CDF count
## Counting Sort:

Data Range is from [0,7]

<table>
<thead>
<tr>
<th>Input Array</th>
<th>5</th>
<th>7</th>
<th>5</th>
<th>2</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Count array</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Counting Sort:

Data Range is from [0,7]

<table>
<thead>
<tr>
<th>Input Array</th>
<th>5</th>
<th>7</th>
<th>5</th>
<th>2</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Count array</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>
Counting Sort:

Data Range is from [0, 7]

<table>
<thead>
<tr>
<th>Input Array</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5</td>
<td>7</td>
<td>5</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Count array (Histogram)</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CDF</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>
Counting Sort:

Data Range is from [0,7]

<table>
<thead>
<tr>
<th>Input Array</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5</td>
<td>7</td>
<td>5</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Count array (Histogram)</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CDF</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

| Output Array | 1 | 1 | 2 | 5 | 5 | 7 |
Counting Sort

• Only works if data range small
Counting Sort

• Only works if data range small

• How to generalize counting sort?
Counting Sort

• Only works if data range small

• How to generalize counting sort?

Operate on small segments of data
Radix Sort: Generalized Count Sort

• Divide element into segments
• Does count sort one segment at a time.
• Repeats this procedure on following segments until finished

• Segments can be anything 10 bits, 2 digits, one character, etc

237
318
216
462
211
268
460
Radix Sort: Generalized Count Sort

• Divide element into segments
• Does count sort one segment at a time.
• Repeats this procedure on following segments until finished

• Segments can be anything 10 bits, 2 digits, one character, etc
Radix Sort: Generalized Count Sort

• Divide element into segments
• Does count sort one segment at a time.
• Repeats this procedure on following segments until finished

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>237</td>
<td>237</td>
<td>216</td>
</tr>
<tr>
<td>318</td>
<td>216</td>
<td>211</td>
</tr>
<tr>
<td>216</td>
<td>211</td>
<td>237</td>
</tr>
<tr>
<td>462</td>
<td>268</td>
<td>268</td>
</tr>
<tr>
<td>211</td>
<td>318</td>
<td>318</td>
</tr>
<tr>
<td>268</td>
<td>462</td>
<td>462</td>
</tr>
<tr>
<td>460</td>
<td>460</td>
<td>460</td>
</tr>
</tbody>
</table>

• Segments can be anything 10 bits, 2 digits, one character, etc
Radix Sort: Generalized Count Sort

- Divide element into segments
- Does count sort one segment at a time.
- Repeats this procedure on following segments until finished

<table>
<thead>
<tr>
<th>237</th>
<th>237</th>
<th>216</th>
<th>211</th>
</tr>
</thead>
<tbody>
<tr>
<td>318</td>
<td>216</td>
<td>211</td>
<td>216</td>
</tr>
<tr>
<td>216</td>
<td>211</td>
<td>237</td>
<td>237</td>
</tr>
<tr>
<td>462</td>
<td>268</td>
<td>268</td>
<td>268</td>
</tr>
<tr>
<td>211</td>
<td>318</td>
<td>318</td>
<td>318</td>
</tr>
<tr>
<td>268</td>
<td>462</td>
<td>462</td>
<td>460</td>
</tr>
<tr>
<td>460</td>
<td>460</td>
<td>460</td>
<td>462</td>
</tr>
</tbody>
</table>

- Segments can be anything 10 bits, 2 digits, one character, etc
Distribution based algorithms complexity

• Assuming the data elements have at most $W$ digits, the complexity of radix sort is at most $O(NW)$.
• This might be better than $N\log N$ based on $W$
• Generally, for numerical data 32, 64, 128 bit integers radix sort is much faster than comparison-based sorts.
Distribution based algorithms

- Essentially generate CDF of the data by counting

Can we use ML models to model CDF?
A Learned Sorting Algorithm
Sorting as a prediction task

• Sorting is essentially a prediction task
• Sorting involves predicting the correct final position of the element based on its value
  \[ pos ← [\text{ECDF}(key) \cdot N] \]
• The algorithms indirectly calculate the position of the element using some operations and place it there
• We can potentially use models for doing this instead of algorithms.
Sorting as a prediction task

Unsorted array A:

| 8 | 10 | 15 | 24 | 19 | 4 | 62 | 30 | 43 |

Sorted array A’:

| 4 | 8 | 10 | 15 | 19 | 24 | 30 | 43 | 62 |

CDF Model Mapping:

- $F_A(x) = \frac{0.000 + 0.125 + 0.250 + 0.375 + 0.500 + 0.625 + 0.750 + 0.875 + 1.000}{9} = 0.656$
- $F_A(x) \cdot |A| = \{0, 1, 2, 3, 4, 5, 6, 7, 8\}$
Model

• Fast in training and inference
  • Can't use Neural networks and other expensive models
  • Use RMI from indexing
  • They are fast to train and infer
What are the potential issues with suing RMI's?

Unsorted array A:

8 10 15 24 19 4 62 30 43

Sorted Array:

Model Mapping

RMI
Sorting with RMIs

Unsorted array A:

| 8 | 10 | 15 | 24 | 19 | 4 | 62 | 30 | 43 |

Nearly Sorted Array:

| 4 | 8 | 15 | 10 | 30 | 62 | 19 | 24 | 43 |
Issues with directly using model

- Collisions: two elements get mapped to the same place
- Imperfect mapping: array may not be monotonic
Fixing the collisions

• Standard Problem in Hash Maps
• Solution use bucketing. Sort within buckets later.

8 10 15 24 19 4 62 30 43

RMI

8
Fixing the collisions

• Standard Problem in Hash Maps
• Solution use bucketing. Sort within buckets later.

| 8 | 10 | 15 | 24 | 19 | 4 | 62 | 30 | 43 |

RMI
Fixing the collisions

• Standard Problem in Hash Maps
• Solution use bucketing. Sort within buckets later.
Fixing the collisions

- Standard Problem in Hash Maps
- Solution use bucketing. Sort within buckets later.
Fixing the collisions

- Standard Problem in Hash Maps
- Solution use bucketing. Sort within buckets later.

| 8  | 10 | 15 | 24 | 19 | 4  | 62 | 30 | 43 |

RMI

| 10 | 8  | 15 | 19 | 24 |
Fixing the collisions

• Standard Problem in Hash Maps
• Solution use bucketing. Below bucket Size=3

| 8 | 10 | 15 | 24 | 19 | 4 | 62 | 30 | 43 |

RMI

| 10 | 4 | 8 | 15 | 19 | 24 |
Fixing the collisions

• Standard Problem in Hash Maps
• Solution use bucketing. Sort within buckets later.
Fixing the collisions

• Standard Problem in Hash Maps
• Solution use bucketing. Sort within buckets later.

| 8 | 10 | 15 | 24 | 19 | 4 | 62 | 30 | 43 |

RMI

| 10 | 4 | 8 | 15 | 19 | 24 | 62 |
Fixing the collisions

• Standard Problem in Hash Maps
• Solution use bucketing. Sort within buckets later.
Fixing the collisions

• Standard Problem in Hash Maps
• Solution use bucketing. Sort within buckets later.

Bucketing Reduces Collisions!!
But does not eliminate them!!
Fixing the collisions: monotonicity not garantueed

• Standard Problem in Hash Maps
• Solution use bucketing. Sort within buckets later.
• When bucket full, throw the extra bucket in a separate array.

RMI

Bucketing Reduces Collisions!!
But does not eliminate them!!
Imperfect Array

• Since the array is nearly sorted.
• We use a fast-sorting algorithm for nearly sorted array.
• Insertion Sort is good for such cases.

![Diagram of sorting process](image)
Learned Sort: After Fixing All Issues
Learned Sort: After Fixing All Issues

Step 1
Model-based bucketization

Step 2
In-bucket reordering
Learned Sort: After Fixing All Issues

Step 1
Model-based bucketization

Step 2
In-bucket reordering

Step 3
Touch-up & Compaction

Bucket Full
Learned Sort: After Fixing All Issues
Performance of Learned Sort

• How do you expect it to perform w.r.t Radix Sort
Performance of Learned Sort

• How do you expect it to perform w.r.t Radix Sort
• It is much Slower than Radix Sort!!

• But why is it slow?
Random access by the model

Unsorted array $A$:

| 8 | 10 | 15 | 24 | 19 | 4 | 62 | 30 | 43 |

Sorted array $A'$:

| 4 | 8 | 10 | 15 | 19 | 24 | 30 | 43 | 62 |

$F_A(x)$:

| 0.000 | 0.125 | 0.250 | 0.375 | 0.500 | 0.625 | 0.750 | 0.875 | 1.000 |

$F_A(x) \cdot |A|$

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
Performance of Learned Sort

• How do you expect it to perform w.r.t Radix Sort

• It is much Slower than Radix Sort!!

• Model does a lot of random accesses while mapping elements in buckets

• Solution: Use small number buckets (~1000) so they fit in cache. Then recursively divide bucket into smaller buckets.
Recursive Partition: fanout=2

<table>
<thead>
<tr>
<th>8</th>
<th>10</th>
<th>15</th>
<th>24</th>
<th>19</th>
<th>4</th>
<th>62</th>
<th>30</th>
</tr>
</thead>
</table>

8 keys
Recursive Partition: fanout=2

<table>
<thead>
<tr>
<th></th>
<th>8</th>
<th>10</th>
<th>15</th>
<th>24</th>
<th>19</th>
<th>4</th>
<th>62</th>
<th>30</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 keys</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>8</th>
<th>10</th>
<th>15</th>
<th>4</th>
<th>24</th>
<th>19</th>
<th>62</th>
<th>30</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 buckets x 4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Recursive Partition: fanout=2

8 keys

| 8 | 10 | 15 | 24 | 19 | 4 | 62 | 30 |

2 buckets x 4

| 8 | 10 | 15 | 4 | 24 | 19 | 62 | 30 |

4 buckets x 2

| 8 | 4 | 10 | 15 | 24 | 19 | 62 | 30 |

Learn Sort Fanout depends on L2 cache size!! (Around 1k-5k)
Multi layered Learned Sort algorithm

Step 1
Model-based bucketization

Step 2
In-bucket reordering

Step 3
Touch-up & Compaction

Step 4
Sort & Merge
Cache Efficient Learned Sort

• How would this perform compared to Radix Sort?
Radix Sort vs Learned Sort:

• Learned Sort is 1.5x faster than Radix Sort on 64 bit integers
• Reason: Learned Sort does fewer number of passes over the data
Summary

• Performance of data structures and algorithms can be enhanced by learned models of data and query distribution
• ML models cannot be simply plugged in
• ML enhanced data structures and algorithms require careful design
• Need to consider Models + Systems + Algorithms