Background

In Nature

In Database System

Issues:
- Unknown Workloads / User Needs
- Large Amount of Raw Data
- Short Query Response Time
- etc.
How should we design a system to handle these issues?

- Manually Design? (painful)

- Any other solution that make the system do it for us? - Yes! Adaptive Indexing Algorithm.

Instead of making decision in the first place,

how about organizing the system when we see the workload?
Example - Standard Cracking
By Using Adaptive Indexing, Our System Is Allowed To:

- **Shift the cost of index maintenance** from Update to Query process
- **Reorganize data** based on workloads
- **Gradually construct index**
- **Continuously improve its performance** during queries
Performance

(a) Standard Cracking (DC)

(d) Sort + Binary Search.

(e) Scan.

(b) Reproducing Cracking Behaviour
However, it’s NOT perfect..

- Slow convergence speed
- “Sensitive” to workloads and data distributions
- Existing methods are specialized for different needs
Drawbacks of Existing Algorithms

(a) Standard Cracking (DC)  (b) Stochastic Cracking (DD1R)  (c) Hybrid Cracking (HCS). For HSS, the inputs are sorted.
Well. How about being more “adaptive”?

Just a little bit of Adaptivity

Oops

Adaptive Adaptive Indexing (meta-adaptivity)
Adaptive Adaptive Indexing

- A generalized adaptive algorithm
- Consider the second higher level of adaptivity
- Adaptive itself to the characteristics of existing methods
1. **Generalize Reorganization Method - Data Partitioning**

Two times partition-in-k (fan-out k = 2)
1. Generalize Reorganization Method - Data Partitioning

So, what if we are able to manage “k”?

- $k = 2$ with two times partition-in-k  
  **Standard Cracking**

- $k = 2^n$ for $n$ bits keys  
  **Sorted Data**

Change the system behavior to emulate any other existing algorithms. 😊
2. Adapt Reorganization Effort (dynamic fan-out k)
   - **Radix-based partitioning algorithm**
     - Put data into k “basket”
     - $k =$ amount of radix bit
   - Process the first query and subsequent queries separately
     - Out-of-place partitioning
     - In-place partitioning
   - Sort the data
2. Adapt Reorganization Effort (fan-out k)

Fig. 3: Out-of-place partitioning using software managed buffers [12].

Fig. 4: Enhancing software managed buffers using non-temporal streaming stores [12].

Out-of-place partitioning
2. Adapt Reorganization Effort (fan-out k)

Performance

Three Key Concepts
2. Adapt Reorganization Effort (fan-out k)

\[ f(s, q) = \begin{cases} 
  b_{\text{first}} & \text{if } q = 0 \\
  b_{\text{min}} & \text{else if } s > t_{\text{adapt}} \\
  b_{\text{min}} + \left[ (b_{\text{max}} - b_{\text{min}}) \cdot \left(1 - \frac{s}{t_{\text{adapt}}}\right) \right] & \text{else if } s > t_{\text{sort}} \\
  b_{\text{sort}} & \text{else.} 
\end{cases} \]

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>(b_{\text{first}})</td>
<td>Number of fan-out bits in the very first query.</td>
</tr>
<tr>
<td>(t_{\text{adapt}})</td>
<td>Threshold below which fan-out adaption starts.</td>
</tr>
<tr>
<td>(b_{\text{min}})</td>
<td>Minimal number of fan-out bits during adaption.</td>
</tr>
<tr>
<td>(b_{\text{max}})</td>
<td>Maximal number of fan-out bits during adaption.</td>
</tr>
<tr>
<td>(t_{\text{sort}})</td>
<td>Threshold below which sorting is triggered.</td>
</tr>
<tr>
<td>(b_{\text{sort}})</td>
<td>Number of fan-out bits required for sorting.</td>
</tr>
<tr>
<td>(\text{skewtol})</td>
<td>Threshold for tolerance of skew.</td>
</tr>
</tbody>
</table>
2. Adapt Reorganization Effort (fan-out k)

Fig. 5: The partitioning fan-out bits returned by $f(s, q)$ for partition sizes $s$ from 0MB to 80MB and $q > 0$ with $t_{adapt} = 64$MB, $b_{min} = 2$, $b_{max} = 10$, $t_{sort} = 2$MB, and $b_{sort} = 64$. 
3. Identify & Defuse Skewed Key Distributions

Not uniformly distributed

performance
Experimental Evaluation
Experimental Evaluation

**Fig. 8:** Different key distributions used in the experiments.

**Fig. 9:** Different query workloads. Blue dots represent the high keys whereas red dots represent the low keys.
Fig. 10: *Emulation of adaptive indexes and traditional methods.* The top row shows the signatures of the baselines from [1] in red. The bottom row shows the signatures of the corresponding emulations of our meta-adaptive index in blue, alongside with the parameter configurations that were used.
Fig. 11: **Individual query response times** of the meta-adaptive index (configured according to Section VIII-C1) in comparison to baselines for a **uniform** (11(a)), **normal** (11(b)), and **Zipf-based** (11(c)) key distribution. The used query workload is RANDOM with 1% selectivity on the key range.
Fig. 12: Accumulated query response times of the meta-adaptive index both manually configured (Section VIII-C1) as well automatically configured using simulated annealing (Section VIII-D1) under uniform (12(a)), normal (12(b)), and Zipf-based (12(c)) key distributions and different query workloads (see Section VIII-A).
partitioning item
use first character to partition into 'less', "equal", and "greater" subarrays

recursively sort subarrays, excluding first character for middle subarray

Appendix
Review

Innovative idea

Perfect substitute

Details in manual configuration

Other automatic configuration method