Adaptive-Adaptive Indexing

Zachary Gou
Roman G. Velez-Alicea

Table of Contents:

So Far…

	Data Organization	Point Queries	Short Range Queries	Long Range Queries	Data Skew	Updates	Affected by Physical Order
B+ Trees	Range	✓	✓	✓	✓	✓	—
LSM Trees	Insertion & Sorted	✓	✗	✓	✓	✓	—
Radix Trees	Radix	✓	✓	✓	✗	—	—
Hash Indexes	Hash	✓	—	✗	✗	✓	—
Bitmap Indexes	None	✓	—	✗	—	✗	*no*
Scan Accelerators	None	✗	—	✓	✓	—	*yes*

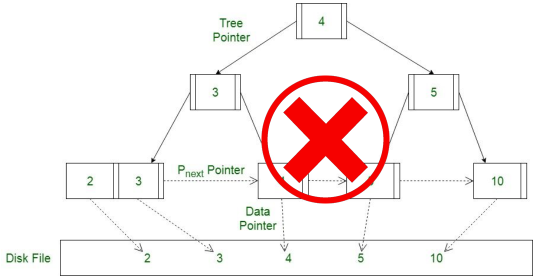
Better Solutions?

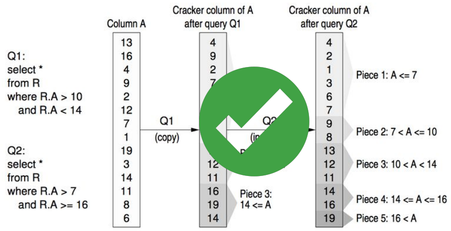
Database Researchers HATE Him!

Doctor's discovery revealed the secret to have the perfect index ordering with no tradeoffs! Watch this shocking video and learn how to do all queries in constant time using this one sneaky index trick! Free of initialization overhead!

Foundations: What is Database Cracking?

- Unlike traditional database indexing, which requires prior knowledge about the queries and the data distribution to create and maintain indexes, database cracking adjusts and optimizes indexes on-the-fly as queries are executed.
- With database cracking, we use the queries as *hints* to how the data has to be ordered.
- This is different from non-discriminative indices such as B-Trees and Hashtables.





Foundations: Quick Walkthrough

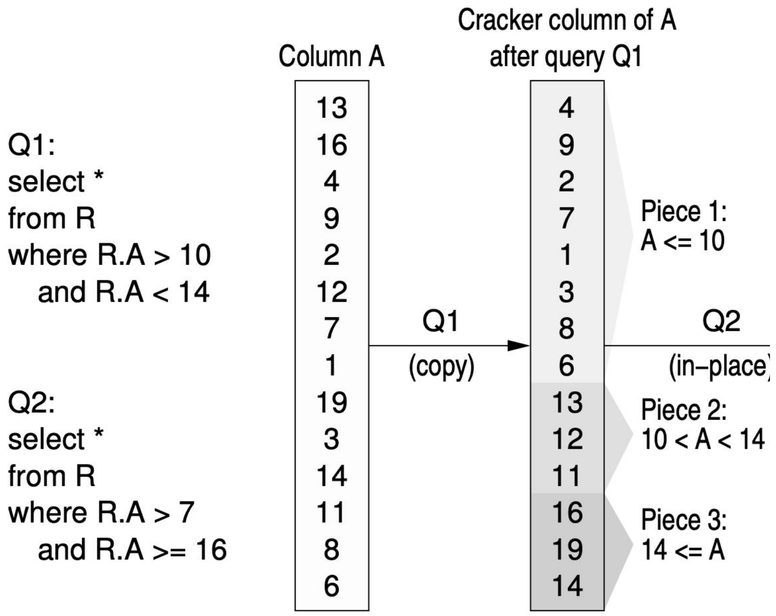
Q1:
```
select *
from R
where R.A > 10
   and R.A < 14
```

Q2:
```
select *
from R
where R.A > 7
   and R.A >= 16
```

Column A
13
16
4
9
2
12
7
1
19
3
14
11
8
6

We take the paper example. Given queries Q1 and Q2 we perform the following…

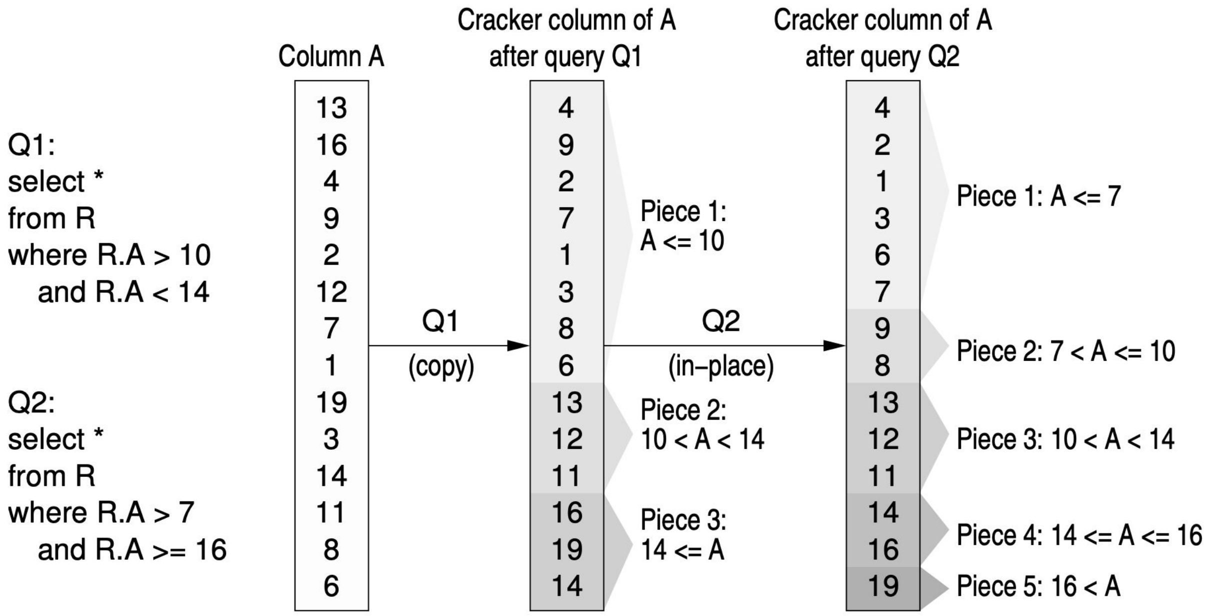
Foundations: Quick Walkthrough



Make a copy of Column of A, the Cracker column, which is used to take advantage of insertion order for reconstr.

Execute and Crack based on Q1.

Foundations: Quick Walkthrough

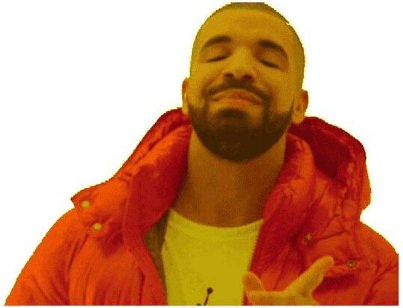


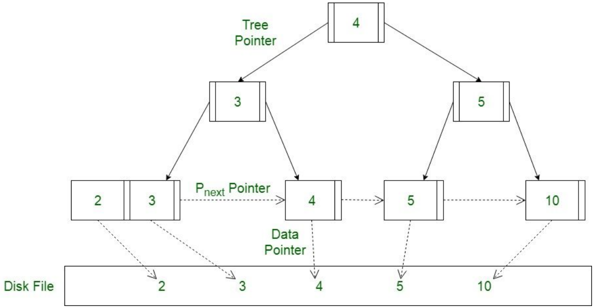
Execute and Crack based on Q2.

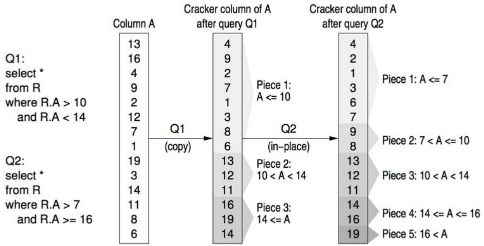
Piece 1 and 3 needs splitting. Piece 2 is free (Zero-Cost for Column Slice).

Piece 5 is free.

Why Doesn’t Everyone Use Trad. Database Cracking?







Why Doesn't Everyone Use Trad. Database Cracking?

- Composed of unoptimized and optimized partitions (sorting), the payoff is over time as more queries are made. Convergence issue. Solved by hybrid algorithms (adaptive merging + database cracking).
- CPU bound and not I-O Bounded, wasted CPU cycles on scanning. Optimized by branch-free cracking, SIMD instructions (more work per instruction), Vectorization, etc.
- Rate of performance per query depends on query pattern or order. A counting up sequential workload can have non-useful partitions. Solved by Stochastic Database Cracking.

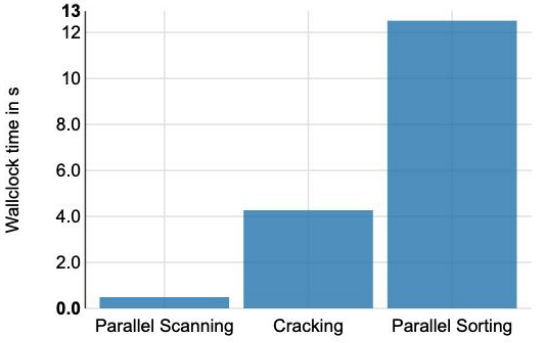
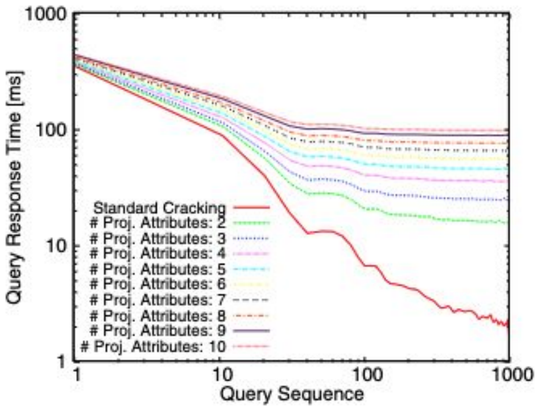


Figure 1: Costs of Database Operations

Comparison to Non I/O-bounded Database Cracking

Why doesn't everyone use Trad. Database Cracking?

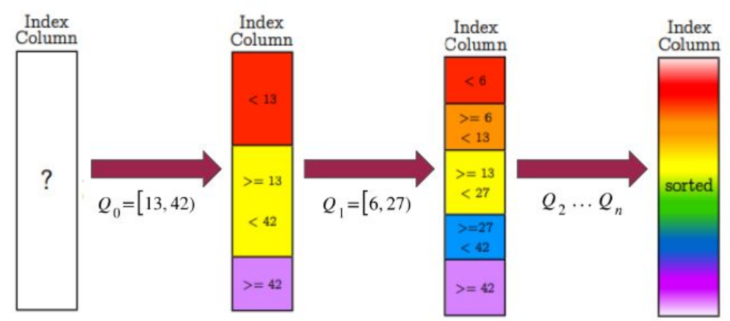
- What about cracking parallelization? The support of concurrency is crucial for performance on modern multi-core hardware. Therefore, the cracking algorithms must be extended to scale well with the available computing cores.
- If project attributes are a lot, then tuple reconstruction may be the bottleneck. Database cracking leads to an unclustered index, to which extra lookups are needed to fetch the projected attributes.

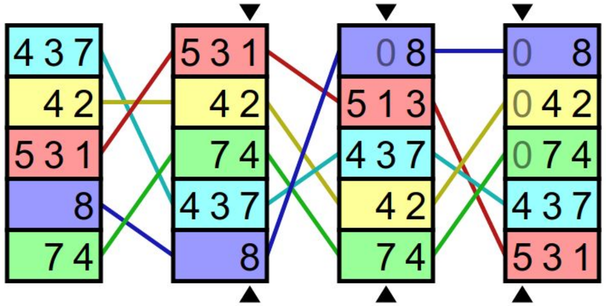


Comparison of Cracking against various number of attributes

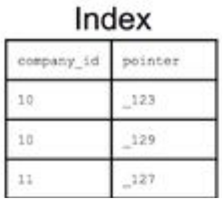
Observations: What do Cracking Algorithms Have in Common?

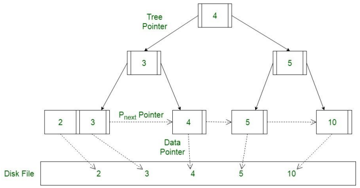
Commonality:	**Difference:**
Simple data partitioning.	Distribution of indexing effort across every query sequence.

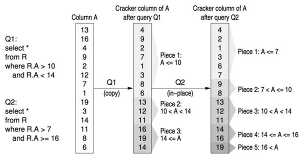


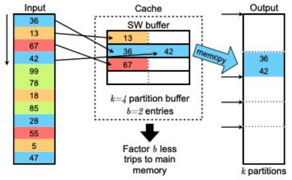


Introducing: ***Adaptive*** Adaptive Indexing









What's Common of All those Cracking Algorithms?

- At the heart of every cracking algorithm is simply just *data partitioning.*
- Given a sequence of queries (Q0, Q1, …, Qn), it matters how the indexing effort is distributed across the sequence of Queries!
- With enough partitioning, we can converge to the ideal data organization for the most optimal quieres!

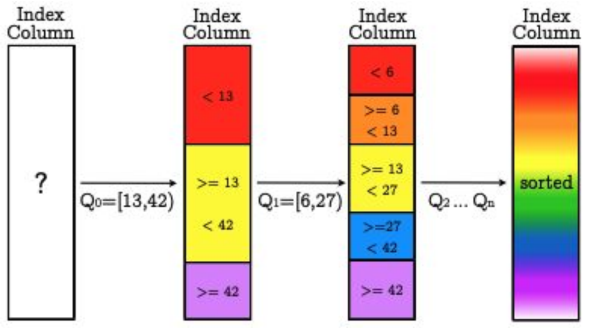


Fig. 1: ***Concept*** *of database cracking reorganizing for multiple queries and converging towards a sorted state.*

A Novel Approach – Adaptive-Adaptive Indexing

- Based observations of different cracking algorithms, the authors of Adaptive Adaptive Index (Schuhknecht et al.) sought to create a **generalized adaptive indexing algorithm** that *adapts itself* to the characteristics of specialized methods.

Features of the Adaptive-Adaptive Indexing Algorithm
1- Generalized Way of Index Refinement
2- Adaptive Reorganization Effort
3- Ability to Identify and Defuse Skewed Key Distributions

Feature: Generalized Index Refinement

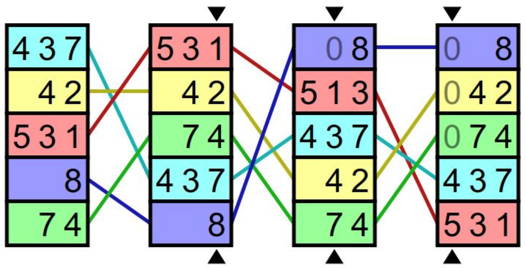
Partition-in-k:

- Each form of reorganization can be neatly represented via function that produces k disjoint partitions.
- Given a function f(k), **we can have granular influence over convergence speed, variance, distribution of the indexing effort.**
- We need an algorithm to set the fan-out so we can easily adapt to various adaptive indexing algorithms.

Reorganization Method	Partition-in-k Representation
Crack-in-Two	$k = 2$
64-bit Key Sort	$k = 2^{64}$

Feature: Generalized Index Refinement - Radix Partitioning

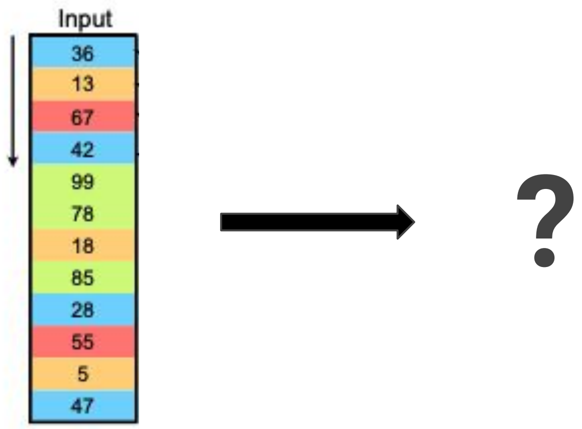
This implementation uses a specialized **radix-based partitioning** offering higher partitioning throughput than comparison-based methods.



Query Type	Radix Partitioning Method	Features
Very First Query	**Out-of-Place**	Temporary storage: software-managed buffers, non-temporal stores, optimized micro-layout.
Subsequent Queries	**In-Place**	Sorting within original data structure, ‘cuckoo-style’, no additional memory.

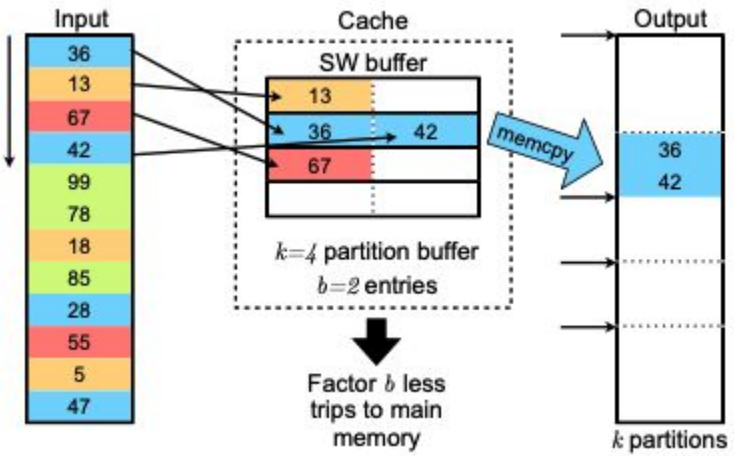
Feature: Adaptive Reorganization Effort - First Query Matters

- As mentioned before, both perform out-of-place partitioning (cracking column & radix-based column).
- Classical Database Cracking simply conducts k = 2 or k = 3 partition.
- Need to capitalize on k-partition advantage (k > 3). How can we do this efficiently on initialization?



Feature: Adaptive Reorganization Effort - First Query Matters

- Using *Out-of-Place Radix Partitioning,* leveraging ***software managed buffers*** and ***non-temporal streaming stores***, we can reduce the partitioning costs.
- We want to take advantage of the TLB cache! Fan-outs > 32 partitions (assuming huge pages), can't cache all address translations for ***each*** data entry in the input.
- Since most of the data is going to the **same** partition anyways, why don't we have an intermediary buffer and then flush them to a mem address all at once.

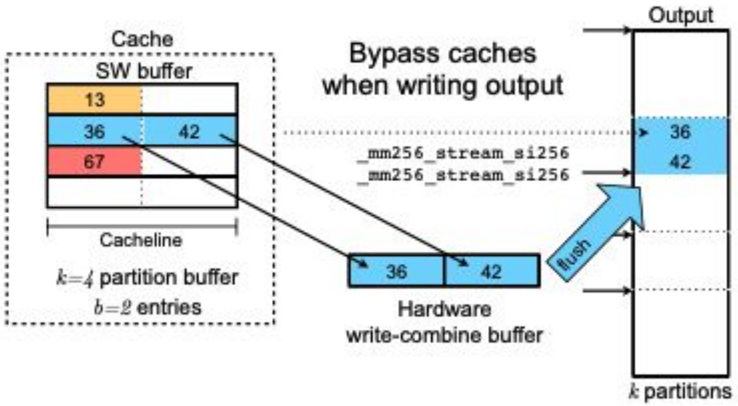


Feature: Adaptive Reorganization Effort - First Query Matters

- When flushing large amounts of data, we want to prevent cache pollution as the cache line may be large.

- Leverage the SIMD to bypass CPU caches. For a single buffer line, we need two calls to the AVX intrinsic "__mm256__stream_si256" in this example.

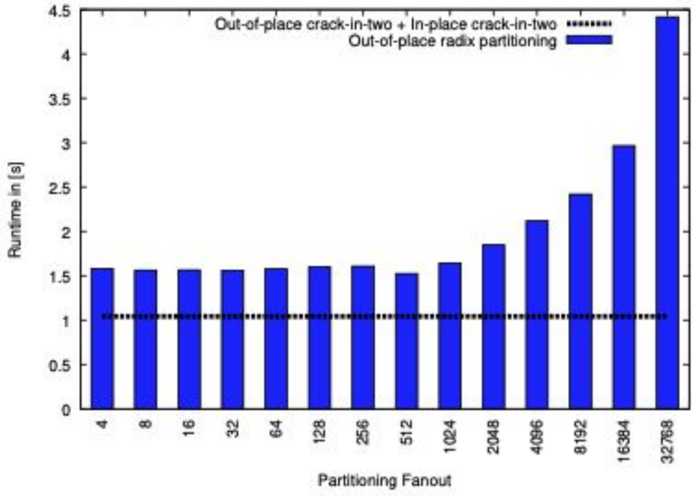
- Works well as these calls triggers a hardware write-combine.



Feature: Adaptive Reorganization Effort - First Query Matters

- Why does this even matter?
  What is the point of talking on all these optimizations that seem small?

- The key is on our first query, we can create a larger number (or dynamic as its control by b_min) than cracking with **negligible overhead** and **reducing the average partition size drastically!**
- Only about a 1.5x slow-down from 0 - 512 fan-out size compared to cracking initializations. We can fit these in caches nicely (like L3) (Data dependent).

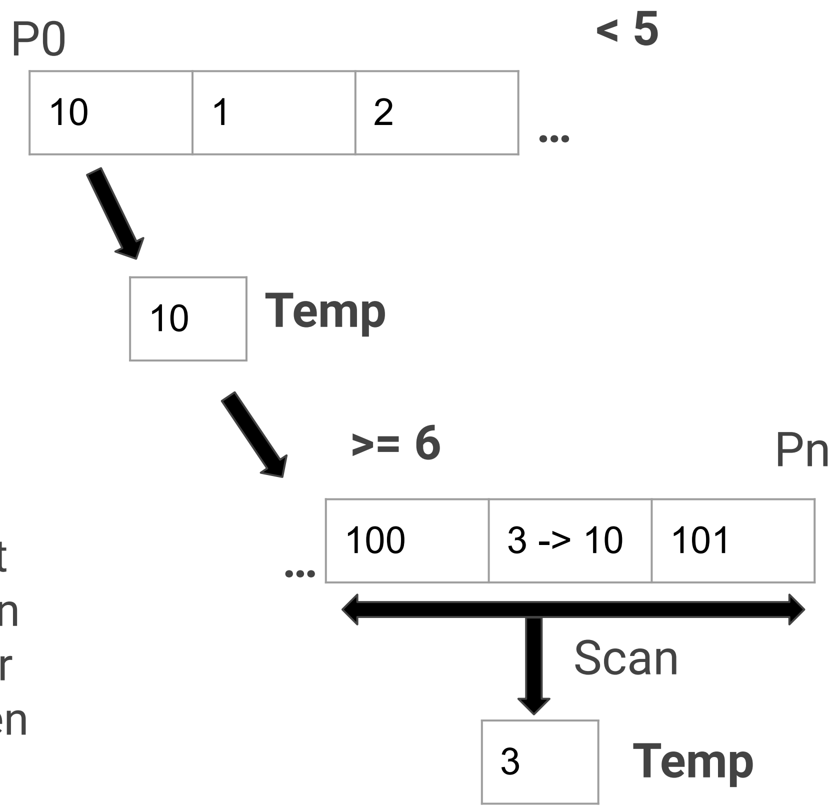


Feature: Adaptive Reorganization Effort - What about Subsequent Queries?

- Sounds all good!
  However, what about the *Qn* query?

- We use In-place Radix Partitioning.
  > Generate the histogram
  > Then perform inplace sorting (not complete sorting) with a replacement algorithm.

- Take a value lets say x0 in partition 0 that doesn't belong and place it in the partition that it does. Then in that partition look for wrong values and do the same. Stop when x0 is filled. Rinse and repeat the cycle.



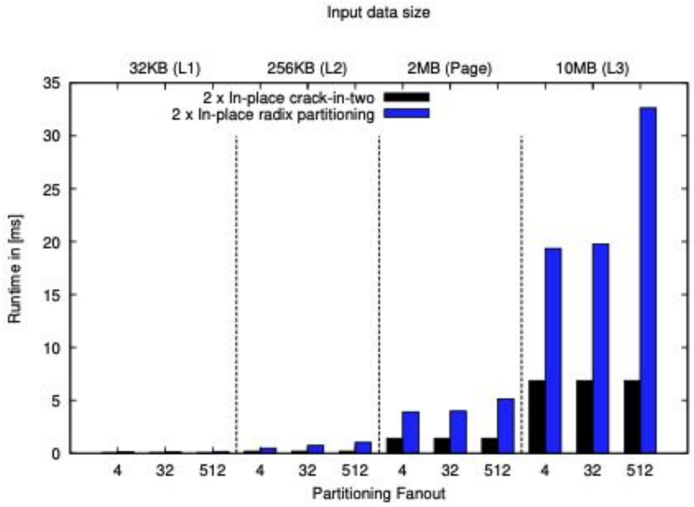
Feature: Adaptive Reorganization Effort - What about Subsequent Queries?

- Is this better (i.e cheaper) than two times in-place crack-in-two reorganizing?
- Unfortunately, No :(
- So why do we sacrifice performance in this in-place radix partitioning? What is the point if two times in-place crack-in-two reorganizing is better?

My reaction when expecting radix partitioning to beat cracking.

Feature: Adaptive Reorganization Effort - What about Subsequent Queries?

- The key is that **the overhead costs are negligible when the input sizes are small.** That means it cost doesn't matter when more partitioning happens!
- This hints that:
  With a decrease in partition size, increase of fan-out k. **At a sufficiently small size, finish the partition sorting cost is negligible.**
- Remember earlier? A query context **sorted index** is what is being converge to (in this case, we are not paying upfront sorting costs).



(b) **Reorganization for a subsequent query**. We test the partition input sizes 32KB (L1 cache), 256KB (L2 cache), 2MB (HugePage), and 10MB (L3 cache). For in-place radix partitioning, we show fan-outs of 4, 32, and 512 as representatives.

Feature: Adaptive Reorganization Effort - So what's the policy?

- Haven't told you how to determine partition size for **Partition-in-k.**
  So How?

- With the following big bad Math equation:

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

You thought in Systems you could get away from math >:D

Feature: Adaptive Reorganization Effort - So what's the policy?

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

- Let ***s*** be the size of the partition to reorganize.
- Let ***q*** be the query sequence number.
- Outputs fanout-bits or the actual partitioning of the input.

TABLE I: ***Available parameters*** *for configuration.*

Parameter	Meaning
b_{first}	Number of fan-out bits in the very first query.
t_{adapt}	Threshold below which fan-out adaption starts.
b_{min}	Minimal number of fan-out bits during adaption.
b_{max}	Maximal number of fan-out bits during adaption.
t_{sort}	Threshold below which sorting is triggered.
b_{sort}	Number of fan-out bits required for sorting.
$skewtol$	Threshold for tolerance of skew.

Feature: Adaptive Reorganization Effort - So what's the policy?

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

b_first -> The *k* fanout for the first query.

TABLE I: **Available parameters** for configuration.

Parameter	Meaning
b_{first}	Number of fan-out bits in the very first query.
t_{adapt}	Threshold below which fan-out adaption starts.
b_{min}	Minimal number of fan-out bits during adaption.
b_{max}	Maximal number of fan-out bits during adaption.
t_{sort}	Threshold below which sorting is triggered.
b_{sort}	Number of fan-out bits required for sorting.
$skewtol$	Threshold for tolerance of skew.

Feature: Adaptive Reorganization Effort - So what's the policy?

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

If the partition size, s, is bigger than t_adapt
===>
then we return the minimum fanout bit as the partition is way too large for less partitions.

TABLE I: ***Available parameters*** *for configuration.*

Parameter	Meaning
b_{first}	Number of fan-out bits in the very first query.
t_{adapt}	Threshold below which fan-out adaption starts.
b_{min}	Minimal number of fan-out bits during adaption.
b_{max}	Maximal number of fan-out bits during adaption.
t_{sort}	Threshold below which sorting is triggered.
b_{sort}	Number of fan-out bits required for sorting.
$skewtol$	Threshold for tolerance of skew.

Feature: Adaptive Reorganization Effort - So what's the policy?

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

If the partition size, s, is smaller than t_adapt, and bigger than t_sort
===>
then we adaptively set the number of bits based on the equation above. The smaller the partition size, the higher returned number of fanout bits.

TABLE I: ***Available parameters*** for configuration.

Parameter	Meaning
b_{first}	Number of fan-out bits in the very first query.
t_{adapt}	Threshold below which fan-out adaption starts.
b_{min}	Minimal number of fan-out bits during adaption.
b_{max}	Maximal number of fan-out bits during adaption.
t_{sort}	Threshold below which sorting is triggered.
b_{sort}	Number of fan-out bits required for sorting.
$skewtol$	Threshold for tolerance of skew.

Feature: Adaptive Reorganization Effort - So what's the policy?

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

If the partition size, s, is smaller than t_adapt, and smaller than t_sort

===>

we then just sort that partition! Return maximum number of fan-out bits, trigger sorting. Remember before, given a small enough input, the sorting will be very cheap!

TABLE I: ***Available parameters*** *for configuration.*

Parameter	Meaning
b_{first}	Number of fan-out bits in the very first query.
t_{adapt}	Threshold below which fan-out adaption starts.
b_{min}	Minimal number of fan-out bits during adaption.
b_{max}	Maximal number of fan-out bits during adaption.
t_{sort}	Threshold below which sorting is triggered.
b_{sort}	Number of fan-out bits required for sorting.
$skewtol$	Threshold for tolerance of skew.

Feature: Adaptive Reorganization Effort - So what's the policy?

- This results in a smooth function. This can lead to possible optimizations on parameters to find the right descent. (Machine Learning?)

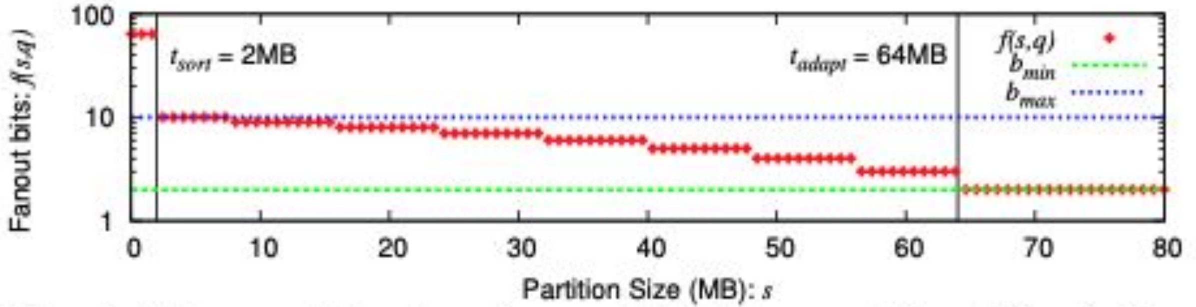


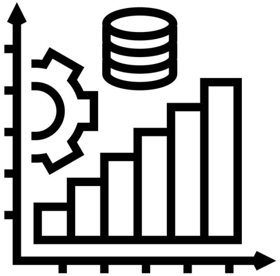
*Fig. 5: The **partitioning fan-out bits** returned by $f(s,q)$ for partition sizes s from 0MB to 80MB and $q > 0$ with $t_{adapt} = 64MB$, $b_{min} = 2$, $b_{max} = 10$, $t_{sort} = 2MB$, and $b_{sort} = 64$.*

Feature: Ability to Identify and Defuse Skewed Key Distributions

By default, radix partitioning creates balanced partitions *only if the key distribution is uniform*. The problem is, uniformity is not always present!

Proposed Solution: Defuse the problems cause by the presence of skew in the very first query.

- **Implementation Features:**
  - Detect skew <u>without overhead</u>
  - In the presence of skew, recursively split partitions that are much larger than the average to enforce balanced processing of subsequent queries.
- **Configuration:**
  - Seven configuration parameters – convergence speed, variance reduction, resistance toward skew, etc.



Feature: Why Is Skew An Issue?

- **Key Issue:**
  - Skewed key distributions lead to generation of non-uniform partition sizes.
  - Non-uniform partition sizes can severely limit the gain in index quality of a partitioning step.
- **Extreme Example:**
  - *Zipf distribution*
    - Most frequent key occurs twice as often as the second most frequent key.
- **Partition Balancing with Skew:**
  - Requires use of equi-depth histograms to balance the partitions.

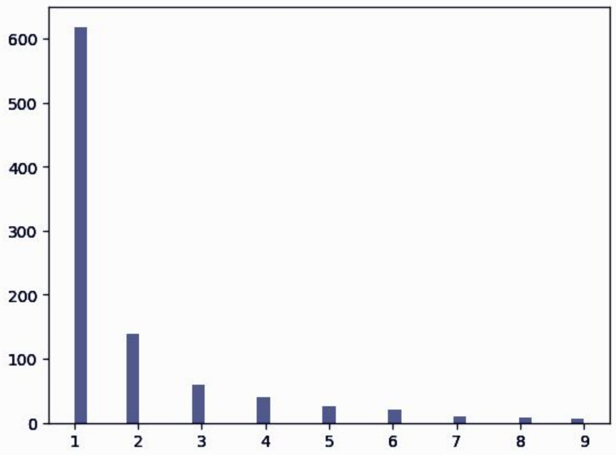
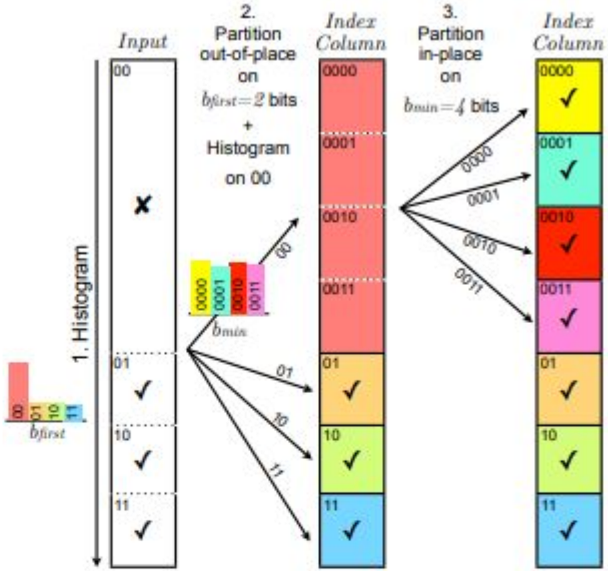


Figure: Example of Zipf distribution

Feature: Defusing Skew

How do we do it?

Equi-Depth Histograms: Statistical tools which summarize the distribution of data across a given attribute - partitions data in buckets such that each bin contains approximately the same number of records.



Feature: Equi-Depth Out-of-Place Radix Partitioning Algorithm

Function: An algorithm that leverages equi-depth histograms designed to handle skewed data distributions from the initial query.

Procedure	
Phase	**Action**
1	Initial Assumption and Histogram Construction
2	- Iterative Comparison and Marking Skew - Partitioning with Respect to Histogram
3	In-Place Partitioning of Skewed Partitions

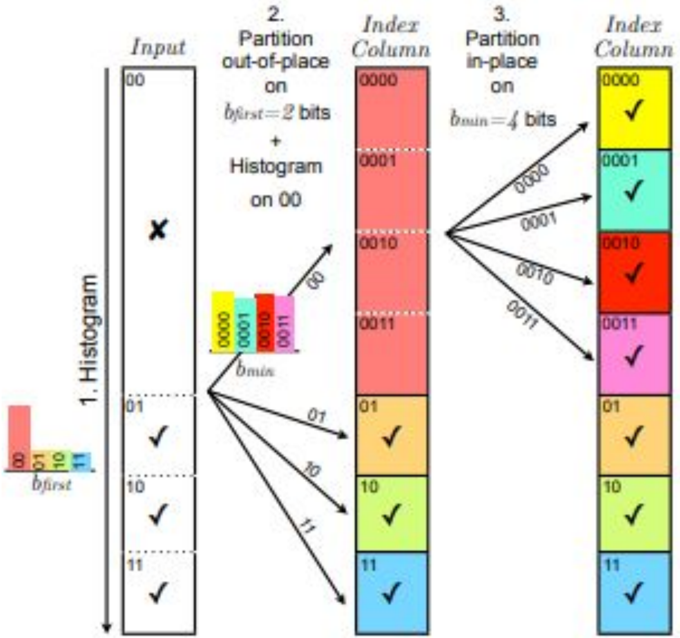


Figure: Defusing of input skew

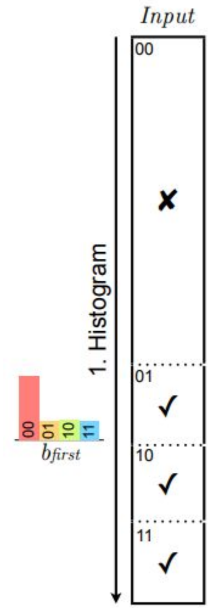
Feature: Equi-Depth Out-of-Place Partitioning - Phase 1

Phase 1 Procedure:

1) **Initial Assumption**
   a) Uniformly distributed keys in the input column
   .
2) **Construct Histogram** .
   a) First phase of the out-of-place partition-in-k algorithm
   b) ‘bfirst’ bits for partitioning

3) **Iterate** –
   a) $\left(\dfrac{column\ size}{k}\right) \cdot skew\ tolerance$

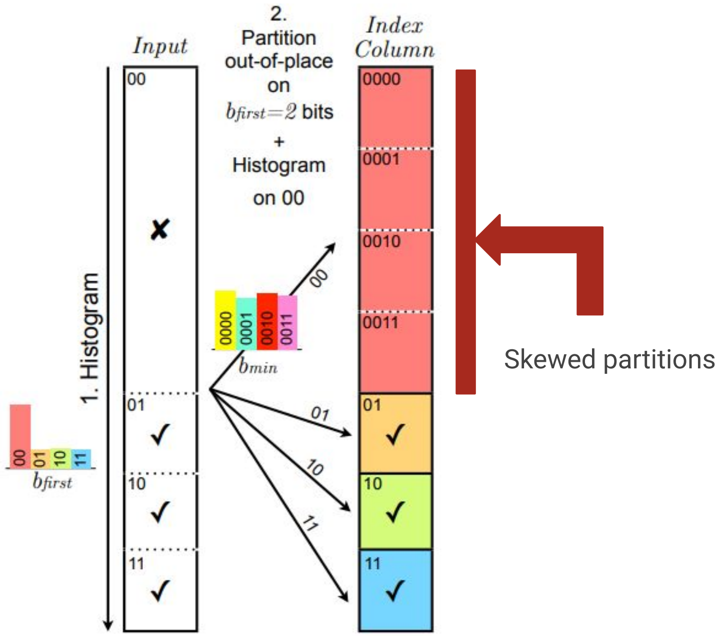
4) Identify and mark partitions that exceed this threshold as skewed.



Feature: Equi-Depth Out-of-Place Partitioning - Phase 2

Phase 2 Procedure:

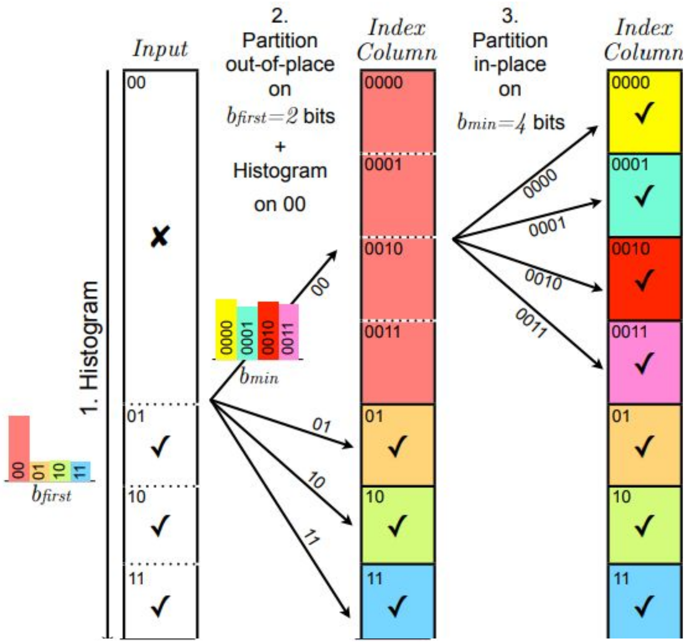
1) **Partition with Respect to the Histogram**
   a) Out-of-place partition-in-k
   b) Copy tuples into corresponding partitions
   c) New histograms built for skewed partitions, using minimum number of bits ‘bmin’.
   d) Piggyback histogram generation for next partition phase into current step



Feature: Equi-Depth Out-of-Place Partitioning - Phase 3

Phase 3 Procedure:

1) **In-Place Partitioning of Skewed Partitions**
   a) Iterate over all skewed partitions
   b) Partition in-place according to ‘bmin’ many bits



Feature: Configuration Knobs

- The adaptiveness is that we can take our parameters to influence the degree of **partitioning. Optimal parameters leads to optimal query response times.**

- How do we figure it out optimal params.?

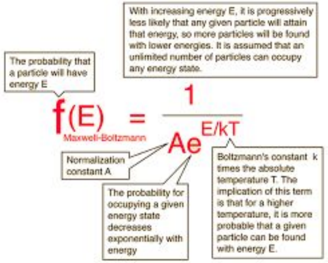
- Trial and Error (Manual Config.)

- Smarter Trial and Error => Simulated Annealing (Following Boltzmann distributions due to using a Boltzmann probability)

*TABLE I: **Available parameters** for configuration.*

Parameter	Meaning
b_{first}	Number of fan-out bits in the very first query.
t_{adapt}	Threshold below which fan-out adaption starts.
b_{min}	Minimal number of fan-out bits during adaption.
b_{max}	Maximal number of fan-out bits during adaption.
t_{sort}	Threshold below which sorting is triggered.
b_{sort}	Number of fan-out bits required for sorting.
$skewtol$	Threshold for tolerance of skew.

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



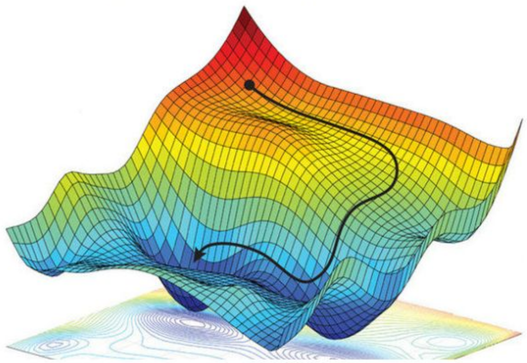
Food for Thought: Thinking Outside of Configuration Knobs

- Possible even better. Remember our Guest Lecturer Andy Huynh’s research? Perhaps we can transform into an optimization problem.

- Can we define a cost function (which is query response time) and then define neighborhood uncertainty, following a iterative method such as Stochastic Gradient Descent (SGD)?

TABLE I: ***Available parameters*** *for configuration.*

Parameter	Meaning
b_{first}	Number of fan-out bits in the very first query.
t_{adapt}	Threshold below which fan-out adaption starts.
b_{min}	Minimal number of fan-out bits during adaption.
b_{max}	Maximal number of fan-out bits during adaption.
t_{sort}	Threshold below which sorting is triggered.
b_{sort}	Number of fan-out bits required for sorting.
$skewtol$	Threshold for tolerance of skew.



Can this be turned into a Convex Optimization problem? Probably not, perhaps its another case of non-convex optimization.

Meta-Adaptive Index

Meta-Adaptive Index - Pseudocode

```
META_ADAPTIVE_INDEX(table, queries) {
  // initialize empty index column
  initializeEmptyIndex()
  // process first query
  // out-of-place partition,
  // handle possible skew, and update index
  oopPartitionInK(table, f(table.size, 0))
  // answer query using filtering and scanning
  // find border partitions
  p[low] = getPartitionFromIndex(queries[0].low)
  p[high] = getPartitionFromIndex(queries[0].high)
  // determine result for lower, mid, upper partitions
  filterGTE(p[low].begin, p[low].end, queries[0].low)
  scan(p[low].end, p[high].begin)
  filterLT(p[high].begin, p[high].end, queries[0].high)
  // process remaining queries
  for(all remaining queries q) {
    // get query predicates
    low = queries[q].low;
    high = queries[q].high;
    // find border partitions
    p[low] = getPartitionFromIndex(low)
    p[high] = getPartitionFromIndex(high)
    // try to refine the largest partition first
    if(p[low] is not finished) {
      ipPartitionInK(p[low], f(p[low].size, q))
      updateIndex()
    }
    // try to refine the smaller partition
    if(p[high] is not finished) {
      ipPartitionInK(p[high], f(p[high].size, q))
      updateIndex()
    }
    // answer query using filtering and scanning
    // find refined border partitions
    p[low] = getPartitionFromIndex(low)
    p[high] = getPartitionFromIndex(high)
    // result for lower partition
    if(p[low] is finished)
      scan(binSearch(p[low], low), p[low].end)
    else
      filterGTE(p[low].begin, p[low].end, low)
    // middle
    scan(p[1low].end, p[high].begin)
    // result for upper partition
    if(p[high] is finished)
      scan(p[high].begin, binSearch(p[high], high))
    else
      filterLT(p[high].begin, p[high].end, high)
  }
}
```

Meta-Adaptive Index - Feature 1

Function: **Generalize the Way of Refinement**

- This portion highlights the generalized way of index refinement through both out-of-place and in-place-partitioning.
- Updates the index based on the initial query and iteratively refines it with subsequent queries.

```
META_ADAPTIVE_INDEX(table, queries) {
  // initialize empty index column
  initializeEmptyIndex()
  // process first query
  // out-of-place partition,
  // handle possible skew, and update index
  oopPartitionInK(table, f(table.size, 0))
...
   for(all remaining queries q) {
      // get query predicates
      low = queries[q].low;
      high = queries[q].high;
...
      if(p[low] is not finished) {
         ipPartitionInK(p[low], f(p[low].size, q))
         updateIndex()
      }
      // try to refine the smaller partition
      if(p[high] is not finished) {
         ipPartitionInK(p[high], f(p[high].size, q))
         updateIndex()
      }
...
}
```

Meta-Adaptive Index - Feature 2

Function: Adaptive Reorganization Effort

- This portion highlights the adaptive reorganization effort, where priority is given to refining the largest partition first, and moving to the smaller if needed.
- The effort to adapt is based on each query's requirements and current state of partitions.

```
for(all remaining queries q) {
    // get query predicates
    low = queries[q].low;
    high = queries[q].high;
...
    // try to refine the largest partition first
    if(p[low] is not finished) {
        ipPartitionInK(p[low], f(p[low].size, q))
        updateIndex()
    }
    // try to refine the smaller partition
    if(p[high] is not finished) {
        ipPartitionInK(p[high], f(p[high].size, q))
        updateIndex()
    }
...
}
```

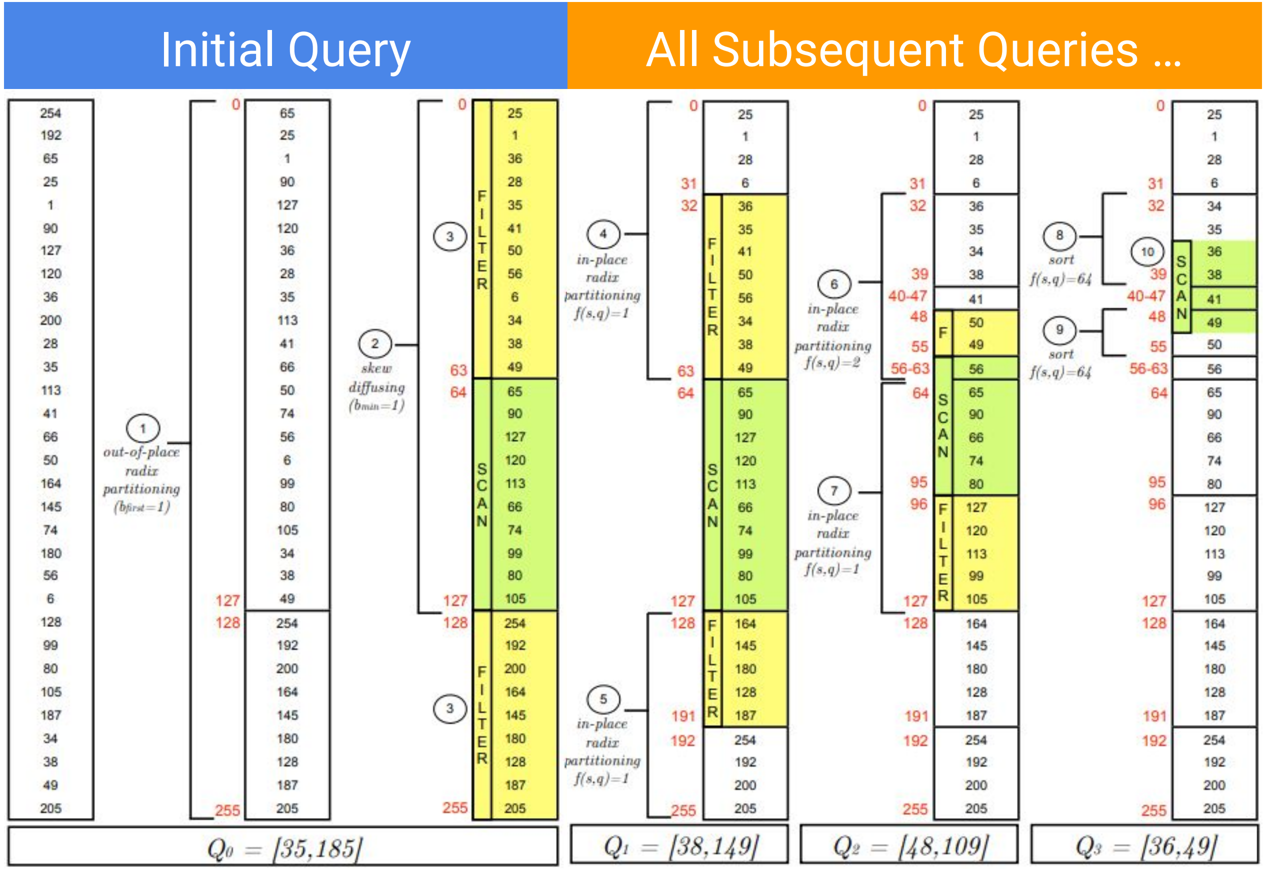
Meta-Adaptive Index - Feature 3

Function: Identify and Defuse Skewed Key Distributions

- This portion highlights the algorithm’s ability to identify and address skewed key distributions through its partition strategy.
- Initially - OOP partitioning that allows handling potential skew.
- Subsequently - IP partitioning to refine further.

```
// out-of-place partition,
// handle possible skew, and update index
oopPartitionInK(table, f(table.size, 0))
...
if(p[low] is not finished) {
    ipPartitionInK(p[low], f(p[low].size, q))
    updateIndex()
}
...
if(p[high] is not finished) {
    ipPartitionInK(p[high], f(p[high].size, q))
    updateIndex()
}
```

Meta-Adaptive Index - Diagram View



Baseline Comparisons

The authors tested against:

> **Standard Cracking** (Standard)

> **Stochastic Cracking** (Combat against Sequential Query Patterns)

> **Hybrid Cracking [HSS & HCS]** (Combat against Convergence issues)

> **Sort + Binary Search** (Extreme Case)

> **Linear Scan with no Index** (Extreme Case)

Experimental Evaluation: Setup

> Data: 100 million entries, each entry is 8B key and 8B RowID. Total about 1.5 GB.

Generated Key distr.: Uniform, Normal, and Zipf.

> Workload: 1000 Range Queries with 8B upper and lower bounds with selectivity 1%.

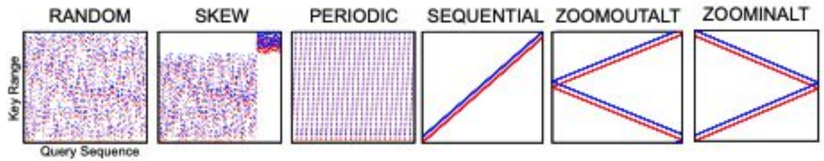


*Fig. 9: Different **query workloads**. Blue dots represent the high keys whereas red dots represent the low keys.*

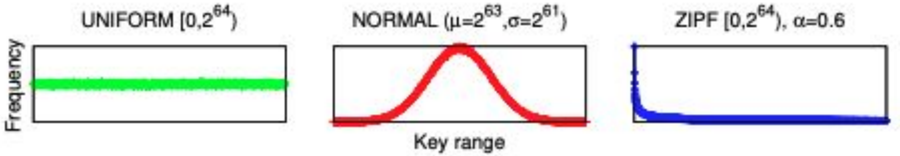


*Fig. 8: Different **key distributions** used in the experiments.*

Experimental Evaluation: Individual Query Response Time Test Setup

> The main goal of basically any adaptive index is to keep the pressure on the individual queries as low as possible.

> Tested with inspection on individual queries. Given the parameters of:

b_first = 10, T_adapt = 64 MB,

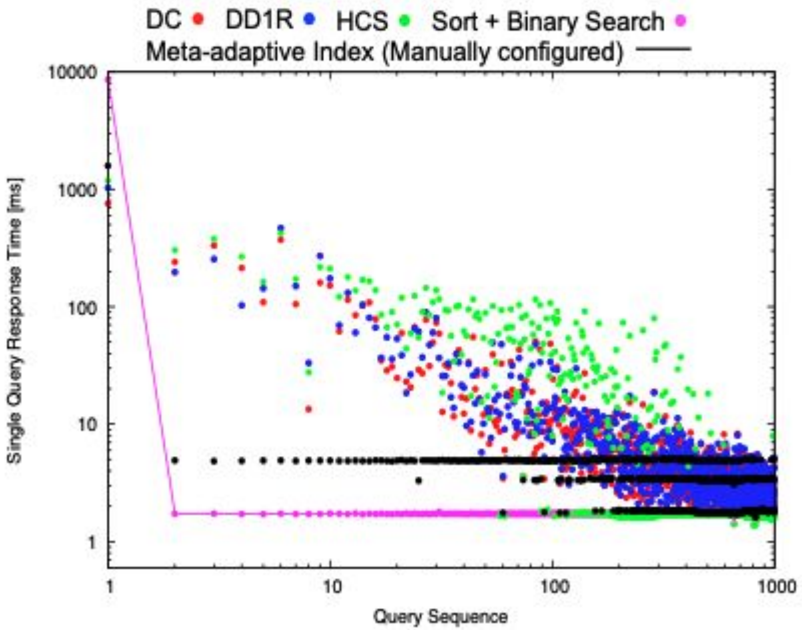
b_min = 3, T_sort = 256 KB

b_max = 6 Skew_Tol = 5x

> Focused on RANDOM Query Workload with 1% Selectivity.

Experimental Evaluation: Individual Query Response Time - Uniform

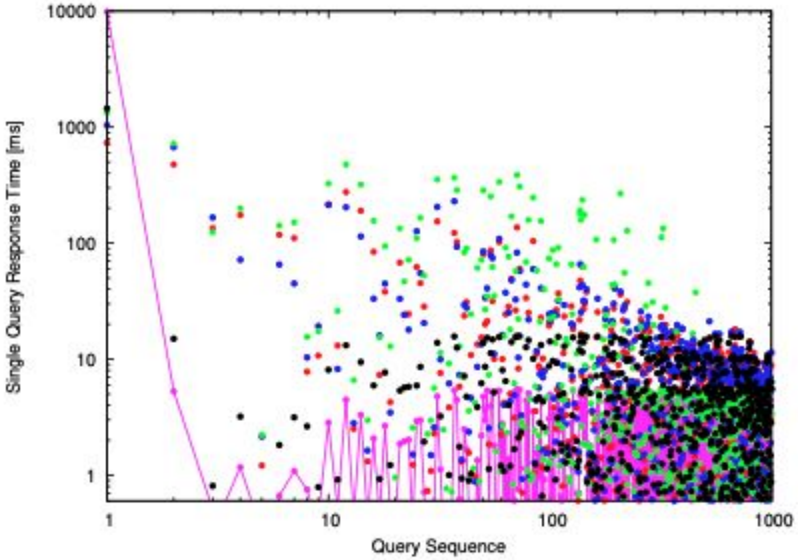
- Start: Meta-Adaptive Index is expensive compared to other baselines.
- Pays off over time.
- Robust and Stable performance below 10ms.



(a) $\mathcal{U}(min = 0, max = 2^{64} - 1)$

Experimental Evaluation: Individual Query Response Time - Normal

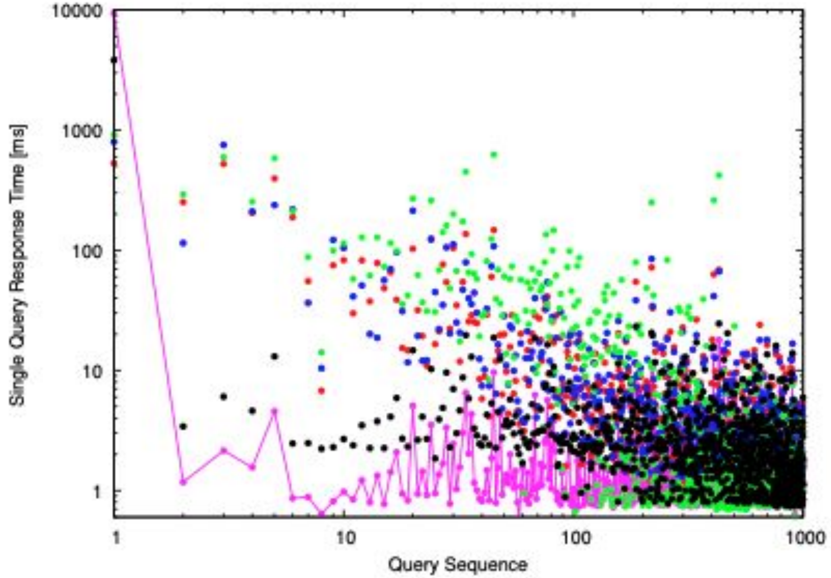
- Start: Meta-Adaptive Index is almost equivalent to other baselines, slightly slower.
- High variance of response times for other methods (concentration)
- Robust and Stable performance below 20ms.



(b) $\mathcal{N}(\mu = 2^{63}, \sigma = 2^{61})$

Experimental Evaluation: Individual Query Response Time - Zipf

- Worse case for radix based partitioning. Much slower (4x) to other baselines in the beginning.
- Skewed distribution impacts first query.
- Robust and Stable performance below 30ms.



(c) $\mathcal{Z}(min = 0, max = 2^{64} - 1, \alpha = 0.6)$

Experimental Evaluation: Accumulated Query Response Time - Uniform

- The automatic configuration is slightly better for all workloads except of PERIODIC.
- Largest b_first

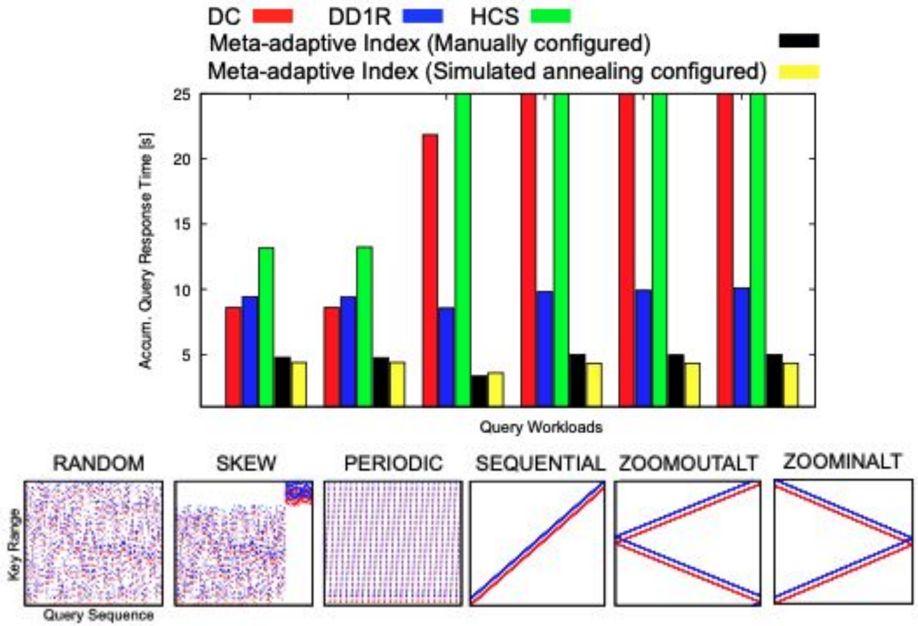
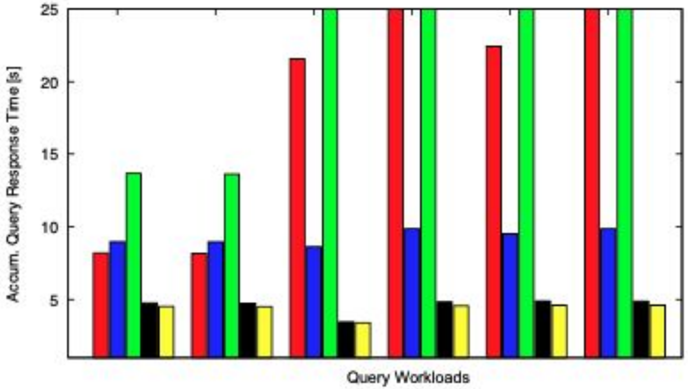


TABLE II: Configuration to minimize accumulated query response time as determined by ***simulated annealing****.*

Parameter	Uniform	Normal	Zipf
b_{first}	12 bits	10 bits	5 bits
b_{min}	2 bits	1 bit	3 bits
b_{max}	5 bits	5 bits	5 bits
t_{adapt}	218MB	102MB	211MB
t_{sort}	354KB	32KB	32KB
$skewtol$	4x	5x	5x

Experimental Evaluation: Accumulated Query Response Time - Normal

- The accumulated query time difference between manual and automatic are very small.
- DC (Red) slightly better than HCS. DD1R the same.



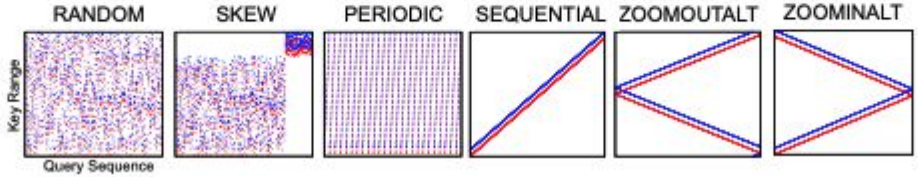


*TABLE II: Configuration to minimize accumulated query response time as determined by **simulated annealing**.*

Parameter	Uniform	Normal	Zipf
b_{first}	12 bits	10 bits	5 bits
b_{min}	2 bits	1 bit	3 bits
b_{max}	5 bits	5 bits	5 bits
t_{adapt}	218MB	102MB	211MB
t_{sort}	354KB	32KB	32KB
$skewtol$	4x	5x	5x

Experimental Evaluation: Accumulated Query Response Time - Zipf

- Biggest difference between manual config and automatic config
- Small b_first

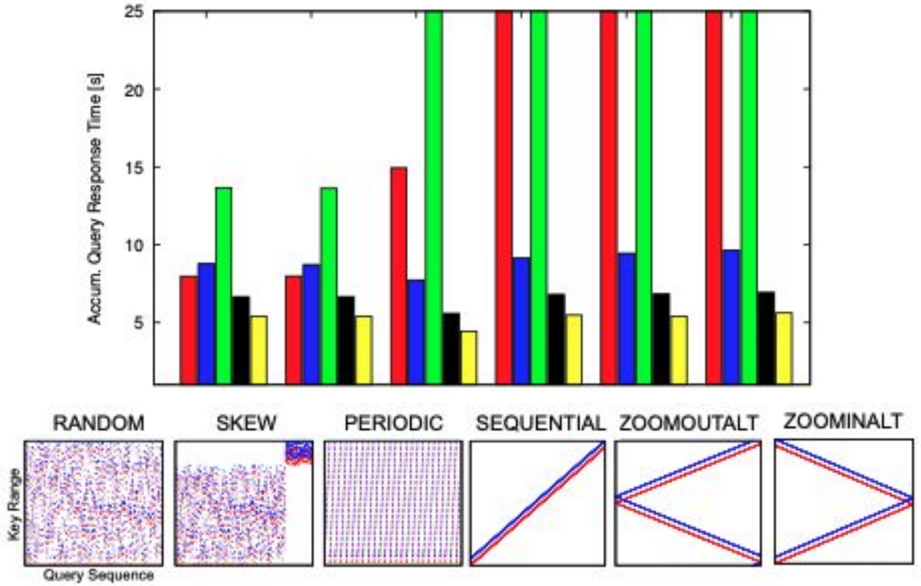


TABLE II: Configuration to minimize accumulated query response time as determined by ***simulated annealing****.*

Parameter	Uniform	Normal	Zipf
b_{first}	12 bits	10 bits	5 bits
b_{min}	2 bits	1 bit	3 bits
b_{max}	5 bits	5 bits	5 bits
t_{adapt}	218MB	102MB	211MB
t_{sort}	354KB	32KB	32KB
$skewtol$	4x	5x	5x

Conclusions

- The authors unified the large amount of specialized adaptive indexes that aim at improving a specific problem at a time in a single general method. This was achieved this by identifying the fact that partitioning is at the core of any adaptive indexing algorithm.
- A meta-index is proposed that can emulate a large set of specialized in-dexes, which we were able to show by inspecting the indexing signatures.
- The meta-adaptive index serves as a valid alternative for a large number of specialized indexes and is able to improve in terms of robustness, runtime, and convergence speed over the state-of- the-art methods.
- Used automatic methods such as Simulated annealing to push the performance of the meta-adaptive index to the limits.

Future Work and Suggestions

- **Write-Heavy Workloads**
  - Investigate the performance of adaptive-adaptive indexing under write-heavy scenarios, develop strategies to mitigate potential overhead from frequent index updates.
- **Enhanced Skew Handling**
  - Investigate ways to extend skew detection and mitigation beyond initial partitioning strategies to improve index performance.
- **Real-World Benchmarks**
  - Test the indexing algorithm on real-world applications, particularly those with mixed and unpredictable workloads, to validate and refine the approach.

Sources

F. M. Schuhknecht, A. Jindal, and J. Dittrich, “The uncracked pieces in database cracking,” *PVLDB*, vol. 7, no. 2, pp. 97–108, 2013.

Felix Martin Schuhknecht, Alekh Jindal and Jens Dittrich, “An experi- mental evaluation and analysis of database cracking,” *VLDBJ*, 2015.

S. Idreos, M. L. Kersten, and S. Manegold, “Database cracking,” *CIDR*, pp. 68–78, 2007.

F. Halim, S. Idreos, P. Karras, and R. H. C. Yap, “Stochastic database cracking: Towards robust adaptive indexing in main-memory column- stores,” *PVLDB*, vol. 5, no. 6, pp. 502–513, 2012.

S. Idreos, S. Manegold, H. Kuno, and G. Graefe, “Merging what’s cracked, cracking what’s merged: Adaptive indexing in main-memory column-stores,” *PVLDB*, vol. 4, no. 9, pp. 585–597, 2011.

S. Idreos, M. Kersten, and S. Manegold, “Self-organizing tuple recon- struction in column-stores,” in *SIGMOD 2009*, pp. 297–308.

Sources

H.Pirk,E.Petraki,S.Idreos,S.Manegold,andM.L.Kersten,“Database cracking: fancy scan, not poor man’s sort!” in *DaMoN, Snowbird, UT, USA, June 23, 2014*, pp. 4:1–4:8.

V. Alvarez, F. M. Schuhknecht, J. Dittrich, and S. Richter, “Main memory adaptive indexing for multi-core systems,” in *DaMoN 2014, Snowbird, UT, USA, June 23, 2014*, pp. 3:1–3:10.

G. Graefe, F. Halim, S. Idreos *et al.*, “Concurrency control for adaptive indexing,” *PVLDB*, vol. 5, no. 7, pp. 656–667, 2012.

Goetz Graefe, Felix Halim, Stratos Idreos et al, “Transactional support for adaptive indexing,” *VLDBJ*, vol. 23, no. 2, pp. 303–328, 2014.

Thank You!

Any Questions?