


Adaptive Adaptive Indexing

Sumatra Dhimoyee, Ayesha Naeem, Joel
Franklin Stalin Vijayakumar, Peixu Xin

A dark blue diagonal gradient bar that starts from the bottom left corner and extends towards the top right corner, covering the bottom half of the slide.

Introduction

Till now have looked at different types of indexes in class...

- Tree
 - B+ Tree
 - LSM Tree
 - Radix Tree
- Hash index
- Bitmap index

What is the problem?

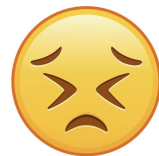
But our requirements are not constant

Workload, machine and user requirements are constantly changing

Problem Statement

How can we design an index that is able to handle these changing environment?

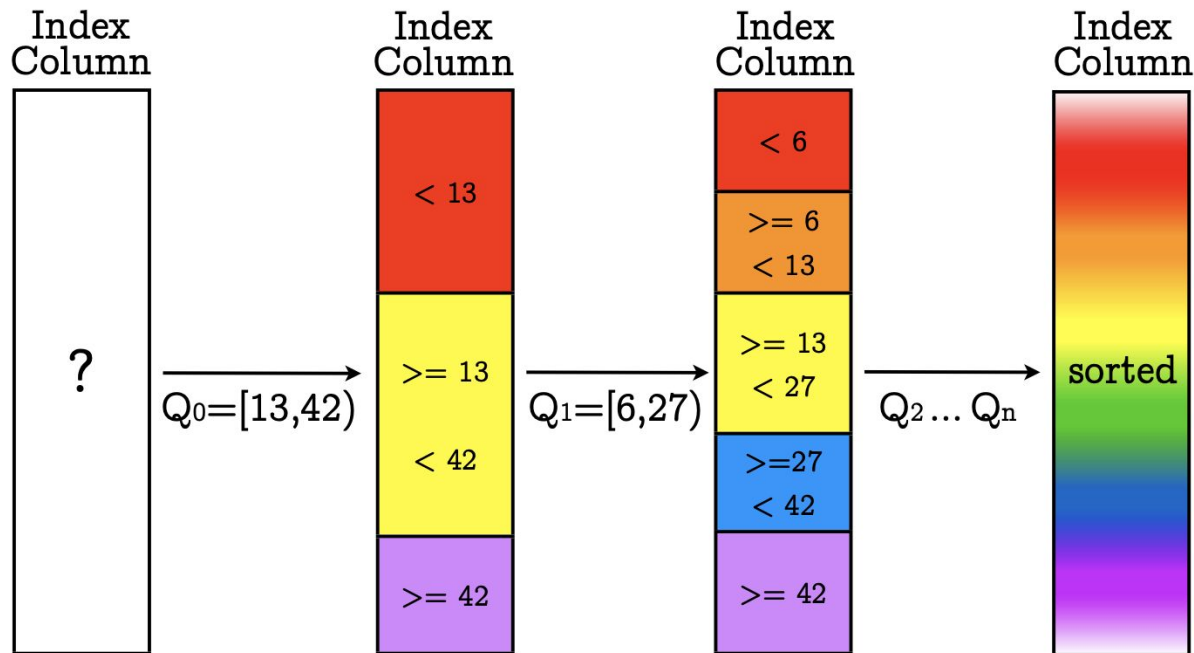
- Manual Tuning



- ***Adaptively build indexes***

Introducing the Concept of Cracking

What does it mean to build indexes adaptively in context of workload?

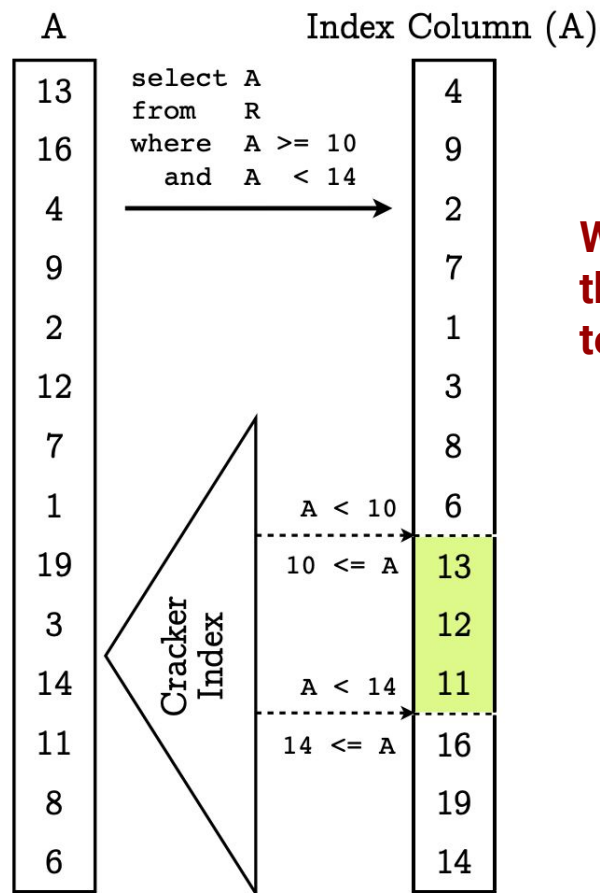


Standard Cracking

- Two times crack-in-two

Pros:

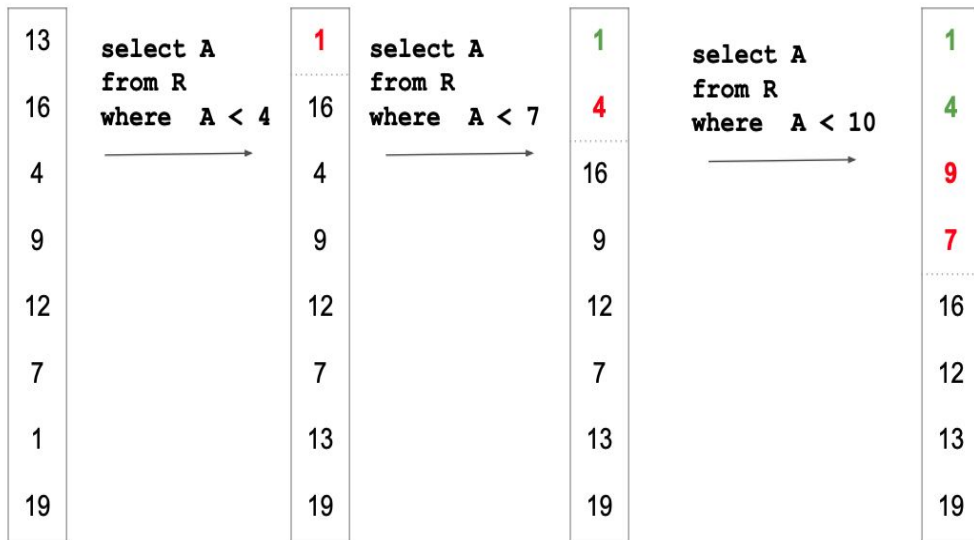
- Cheapest initialization cost
- Performs very well under uniform workload



What might be the issue with technique?

Sequential Workload

$N = 8$



C = # of comparisons required to answer query

N

N-1

N-2

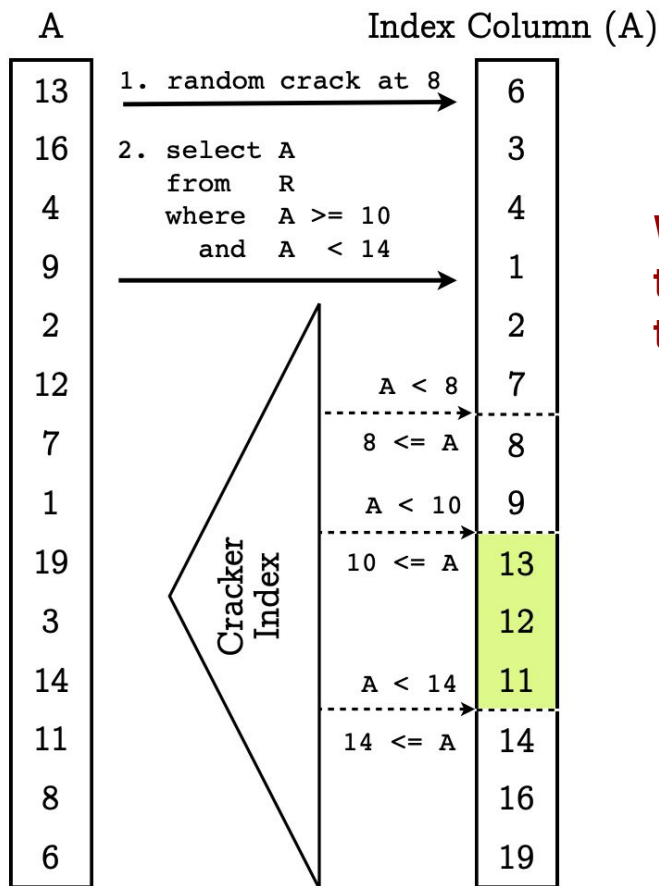
What is the problem?

- Long convergence time
- Least amount of reorganization

Can we do better?

Stochastic Cracking

- Introduces random cracks on top of Standard Cracking
- Pros:
 - Robust under various workload as it decouples reorganization from queries

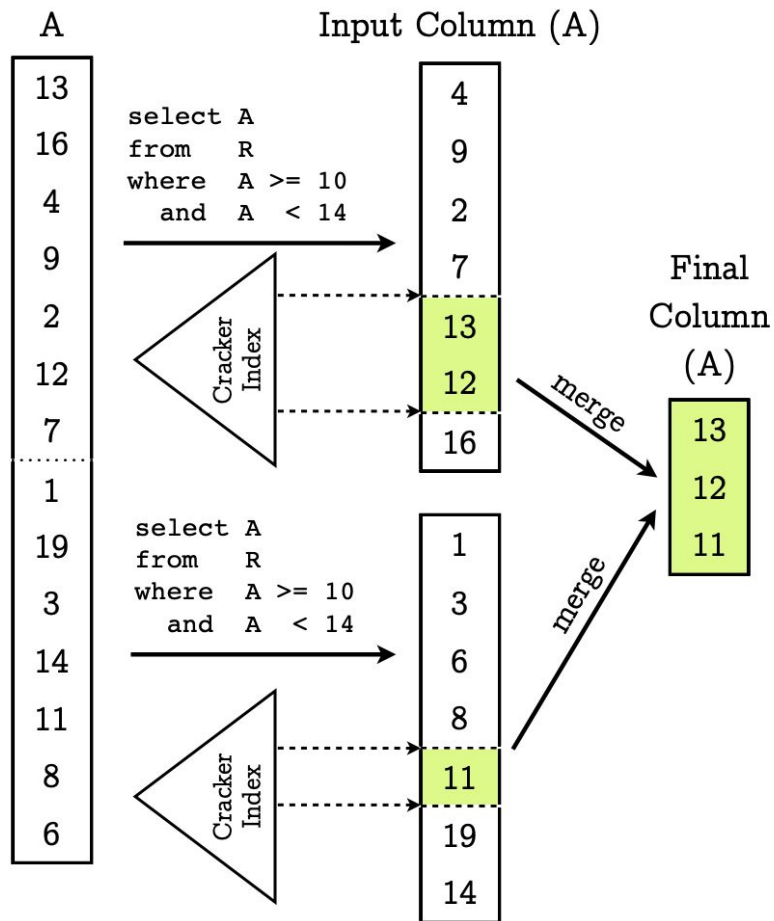


What might be the issue with technique?

High Initialization Cost

Hybrid Cracking

- Split the inputs into partitions, merge the final column
- Improve convergence speed and memory footprint



Issues with existing work

Adaptive indexes already exists. Why are do we need another one?

Each of the other indexes only addresses one specific problem!

Solution to all the problems?

What if we can create a system with more adaptivity?

**Adaptive Adaptive Indexing
(meta-adaptivity)**

Adaptive Adaptive Indexing

What do we mean by meta-adaptivity?

A system that can be extended with various implementations and can switch implementations based on user and current workload requirements.

Components of Adaptive Adaptive Indexing

Generalize
Index
Refinement

Adapt
Reorganization
Effort

Defuse
Skewed
Data

Generalize Index Refinement

The applied fan-out of a partitioning algorithm dictates the convergence speed, variance, and distribution of the indexing effort.

A system needs different implementations for adaptive indexing depending on the workload.

An algorithm that is able to set the fan-out of the partitioning procedure freely is able to adapt to the behavior of various adaptive indexing algorithms.

- $K=2$ -> Standard Cracking
- $K=2^{64}$ for 64 bit keys -> Sorting

Partition-in-K for reorganizing the index using a freely adaptive fanout

- Distinguish b/w the out of place algorithm for the 1st query, optimized inplace algorithm for the rest
- But use radix partitioning instead of comparison based methods

Problem with comparison based method?

Radix Partitioning

Radix based partitioning has a higher partitioning throughput.

Number	Binary Form
25	11001
9	1001
52	110100
6	0110
2	0010

Partition on two bits



Partitions	Number
00	2
01	6
10	9
11	25, 52

Adaptive the reorganization Effort

How much to actually partition?

Processing the First Query

Done out of place, copies the source column into a new array

Option 1: Using classical approach, copy the column and crack in 2.

If we are copying the entire column anyway, can we do something smarter?

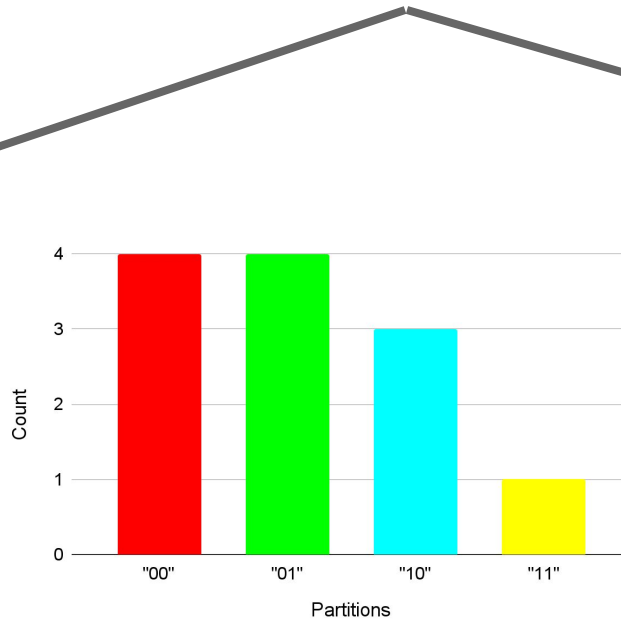
Option 2: Use the efficient radix based method with custom fan-out.

1. Count values in each partition

Input: Source column and $K = 2^b$ (number of partitions).

36	0100100
13	0001101
67	1000011
42	0101010
99	1100011
78	1001110
18	0010010
85	1010101
28	0011100
55	0110111
5	0000101
47	0101111

2. Create a histogram



3. Use the histogram to initialize pointers to fill the partitions

4. Copy values from the source column to the output column

A vertical stack of 12 colored boxes representing the output column. The colors and values from top to bottom are: red (13), red (18), red (28), red (5), green (36), green (42), green (55), green (47), cyan (67), cyan (78), cyan (85), and yellow (99). Arrows from the histogram point to these boxes: a large arrow from the '00' bar points to the top four red boxes; an arrow from the '01' bar points to the green boxes; an arrow from the '10' bar points to the cyan boxes; and an arrow from the '11' bar points to the yellow box.

13
18
28
5
36
42
55
47
67
78
85
99

What is the problem here?

Random copying leads to TLB misses with more than 32 partitions!

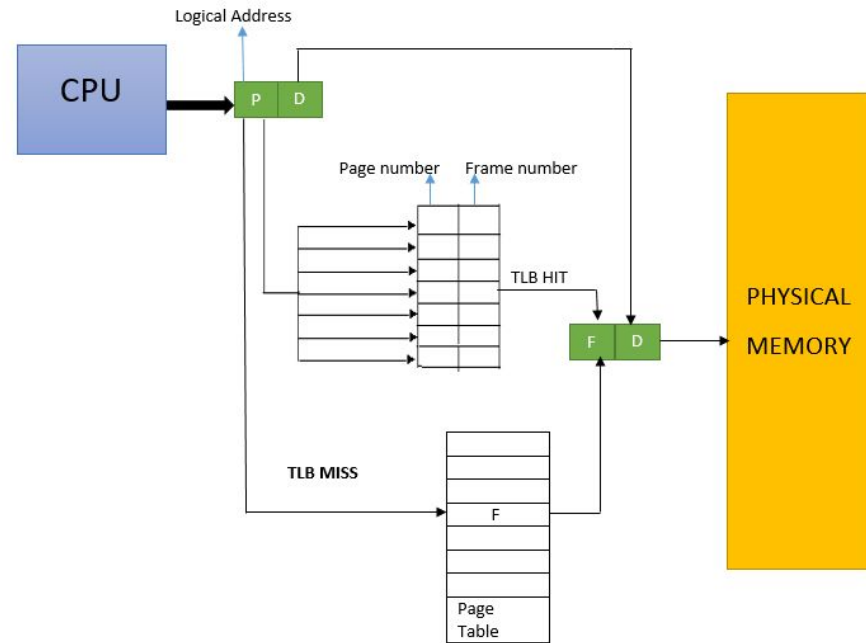
Translation Lookaside Buffer (TLB)

TLB stores a **mapping** of virtual memory to physical memory for quick lookups.

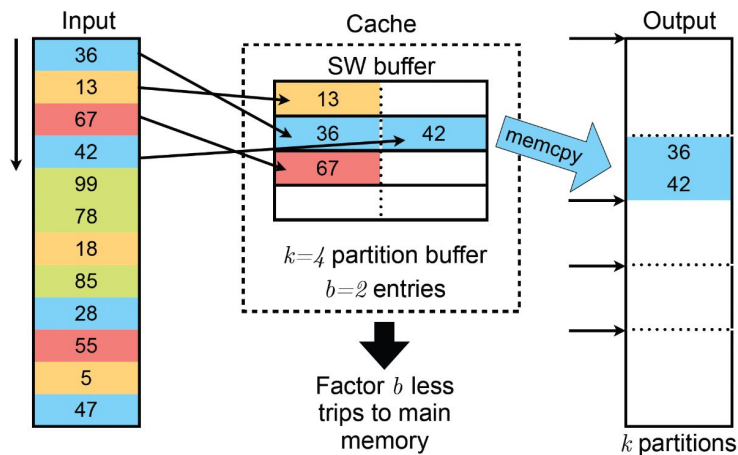
Separate entry for each partitions.

TLB can only store upto 32 PTEs at a time for huge pages.

How do we handle TLB misses? -
Software managed buffers!



Software Managed Buffers

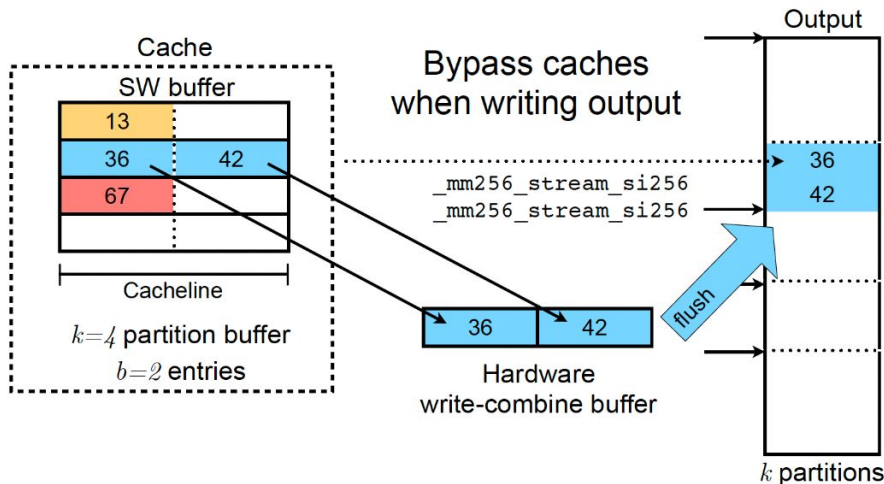


Out of place partitioning using software managed buffers

Buffer of size 2 for each partition

Likely to fit in CPU caches

Non temporal Streaming Stores and SIMD



Enhancing software managed buffers using non temporal streaming stores

SIMD intrinsics allow to bypass the CPU caches when flushing the software-managed buffers to the destination partitions.

Data Partitioning in Subsequent Queries

Already used out of place partitioning for the first query

In place reorganization must happen for subsequent queries

In-place radix partitioning

Step 1: Create a histogram for number of entries for each partition k

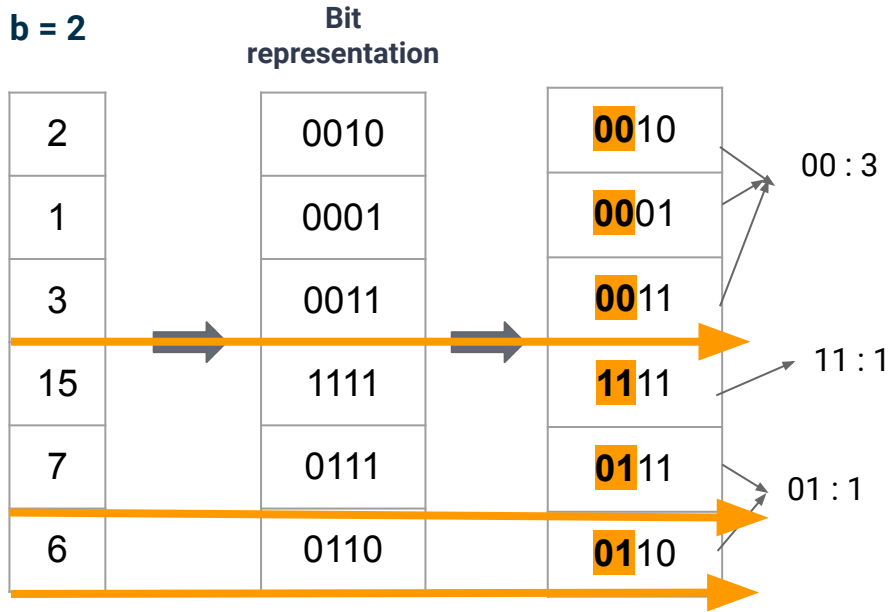
Step 2: Perform *search* and *replace* on the index column

Phase 1: Histogram Generation

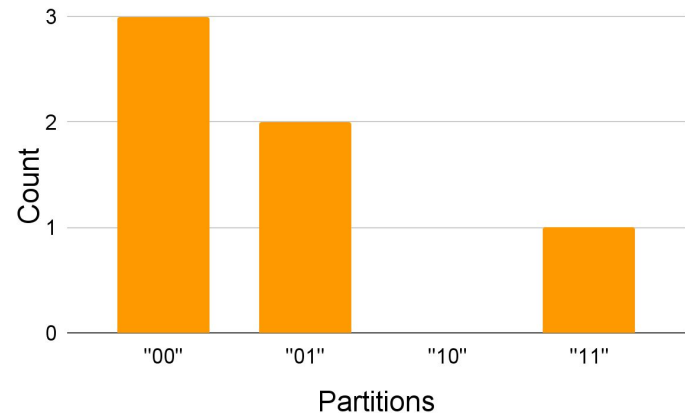
Determine the number of partitions **k** and the number of fanout bit **b**

B is the number of bits by which the input should be partitioned $K = 2^b$ (**More on this later**)

b = 2



Now we know the boundaries of the partitions (after 3rd, 5th and 6th element!)



Phase 2: Search and Replace

Index Column	Bit representation	
2	0010	✓
1	0001	✓
3	0011	✓
15	1111	?
7	0111	
6	0110	

00
01
11

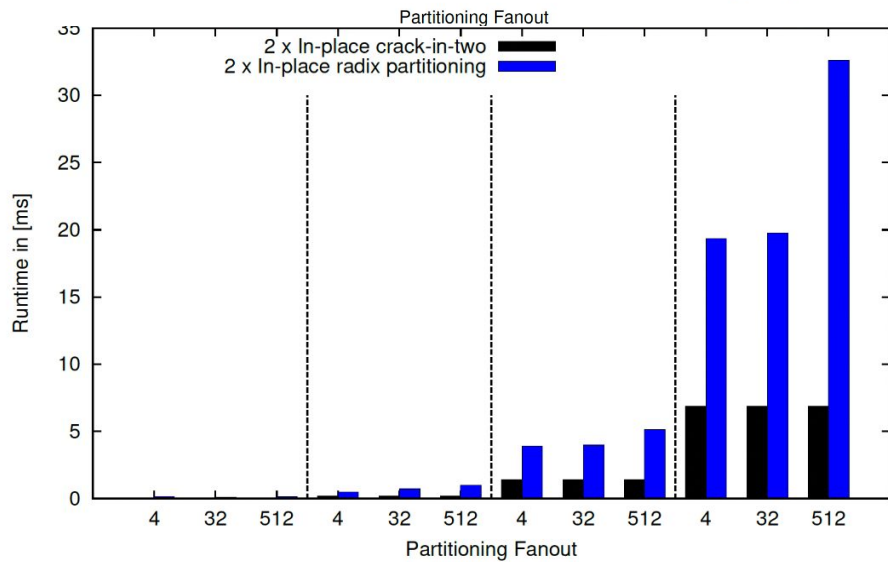
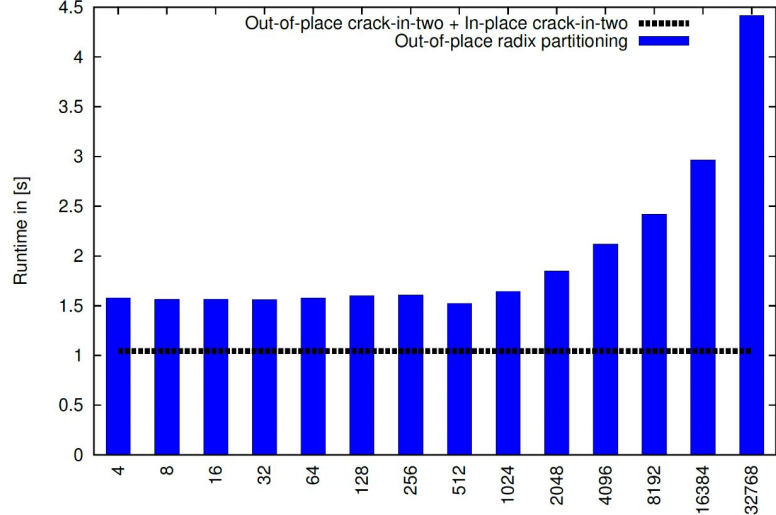
Index Column	Bit representation	
2	0010	✓
1	0001	✓
3	0011	✓
6	0110	✓
7	0111	✓
15	1111	✓

00
01
11

1. Check if elements belong to the right partition.
2. 15 belongs to partition 11. Scan partition 11 and find the first element that does not belong to 11. Swap it with 15

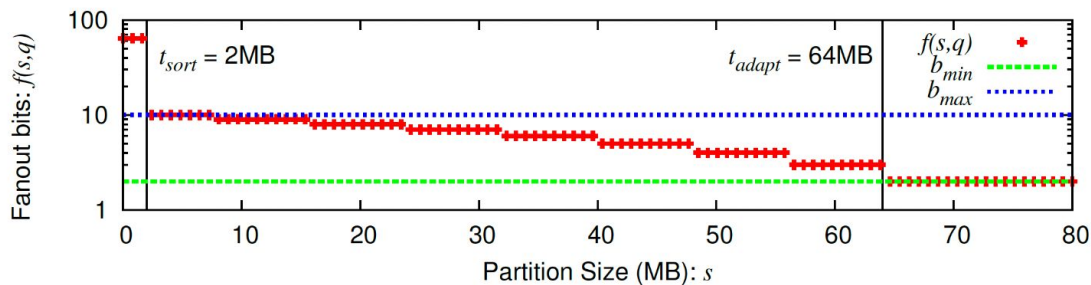
Evaluation of In and out of place Radix Partitioning

- Overhead of partitioning with higher fan-out k decreases with a decrease in partition size.
- Driving factor for deciding b bits for $K = 2^b$
- With a decrease in partition size, increase fan-out k . At a sufficiently small size, finish the partition by sorting it as the cost is negligible.
- How to find K ?



Adapting the Partitioning Fan-out $K = 2^b$

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



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

bfirst	Number of fanout bits in the first query.
s	Size of a partition
bmin	Minimal no. of fanout bits during adaption.
bmax	Maximal no. of fanout bits during adaption.
tadapt	Threshold below which fanout adaption starts.
tsort	Threshold below which sorting is triggered..
bsort	No. of fan-out bits required for sorting.

Handling Skew

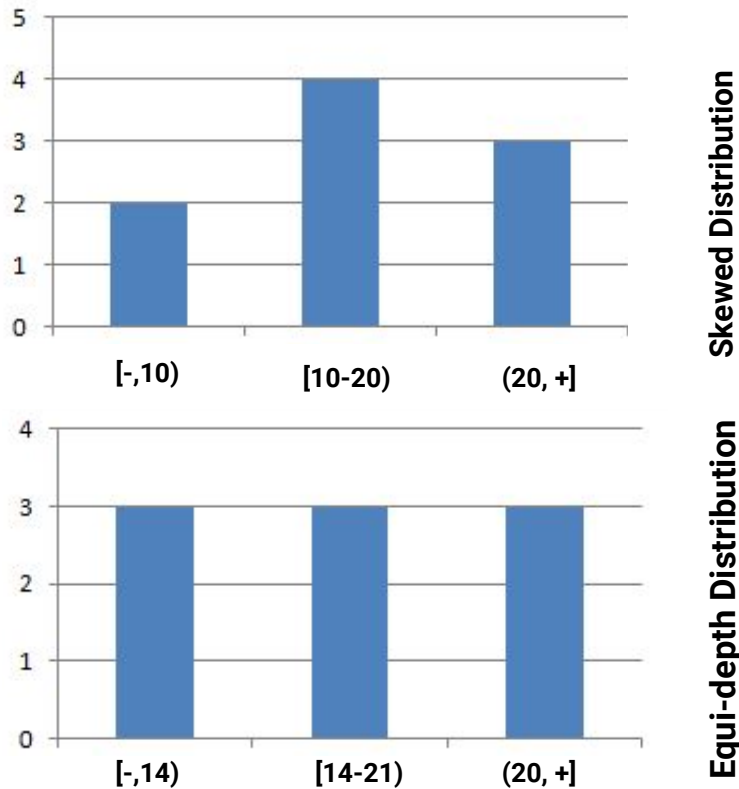
Radix based partitioning cannot handle skewed input distributions well - **Why?**

Solution: Equi-depth partitioning

Not suitable for radix based partitioning - **Why?**

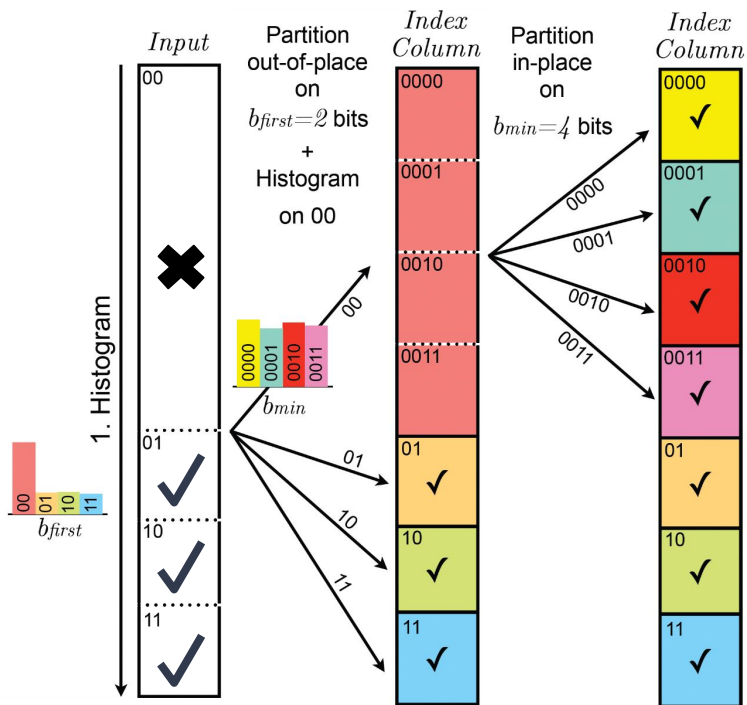
Meta Adaptive Index's Solution:

Equi-depth out of place radix partitioning algo



Handling Skew

- Create a histogram for the first query ($b = b_{first}$)
- $Skewtol$ denotes skew tolerance of a partition. If $s > (columnsize/k) * skewtol$, mark partition as skewed.
- Complete out-of place partitions (b_{first}) and build histograms on skewed partitions (b_{min})
- Sub-partition skewed partitions (b_{min}) in-place.



Experimental Evaluation

Two Tests

```
graph TD; A[Two Tests] --> B[Emulation of other indexes by meta-adaptive index]; A --> C[Comparison of response times of the meta-adaptive index to other indexes]
```

**Emulation of other indexes
by meta-adaptive index**

**Comparison of response
times of the meta-adaptive
index to other indexes**

Emulation of the below Indexing Algorithms

1. **Standard Cracking**
2. **Hybrid Crack Sort (HCS)**
3. **Hybrid Sort Sort (HSS)**
4. **Scan**
5. **Quick Sort + Binary Search**
6. **Coarse-granular Index 1K**

Memory Requirements for the Experiment

32KB of *L1* cache

256KB of *L2* cache

10MB of shared *L3* cache

2MB Page Size

24GB of DDR3 RAM

Our Dataset

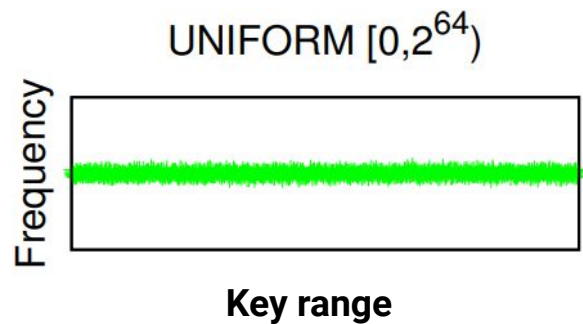
100 million entries

Each entry - 8B Key and 8B rowID

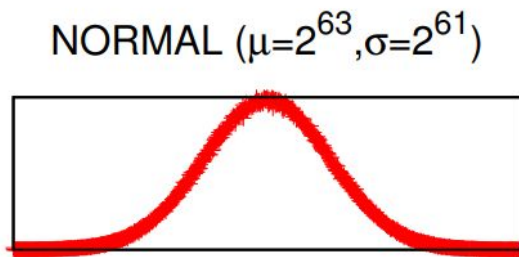
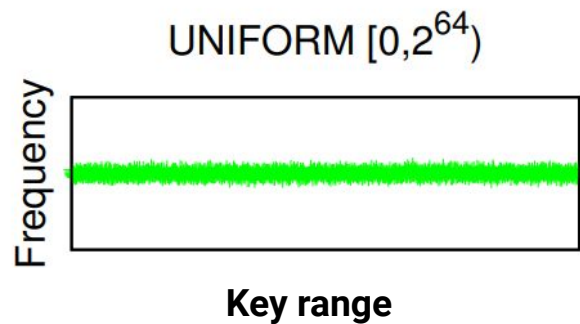
1.5 GB of data

Different Key Distributions used in Experiment

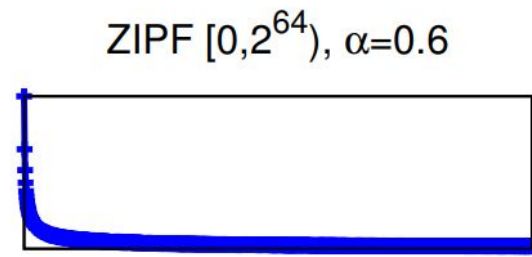
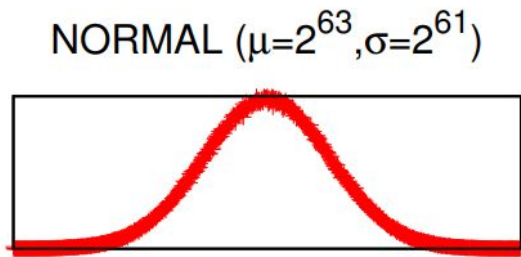
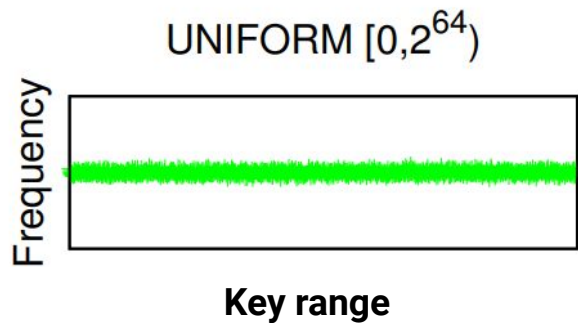
Different Key Distributions used in Experiment



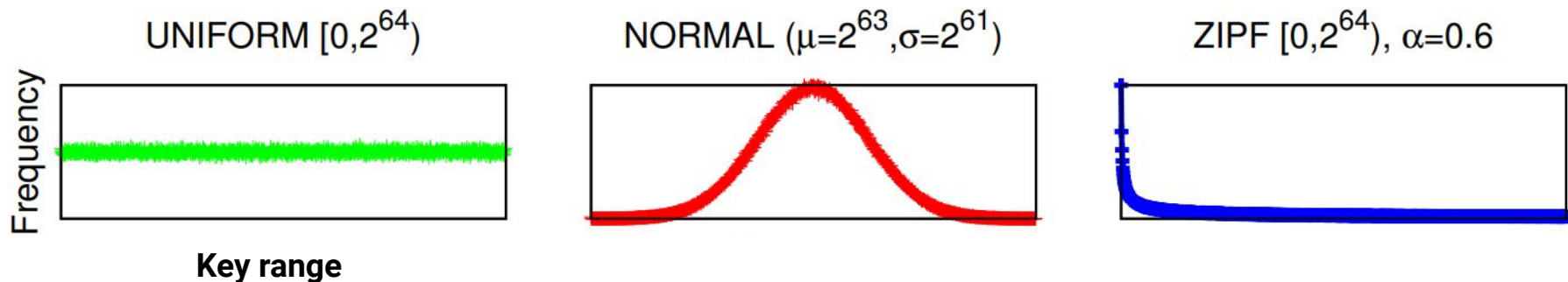
Different Key Distributions used in Experiment



Different Key Distributions used in Experiment

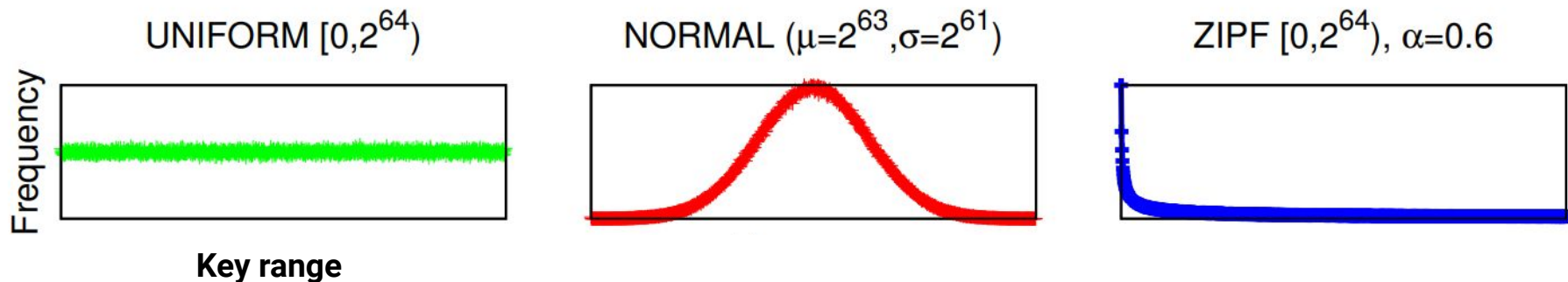


Different Key Distributions used in Experiment



What is the effect of differing key distributions?

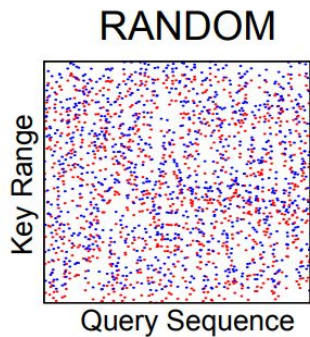
Different Key Distributions used in Experiment



Different key distributions affect the SKEW

Different Query Workloads

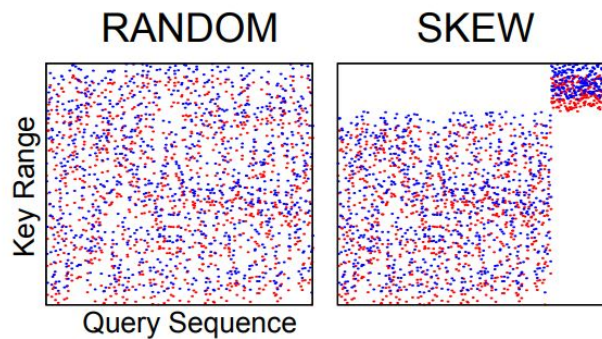
Different Query Workloads



Blue dots represent the high keys whereas red dots represent the low keys

Each Query Workload consists of 1000 range queries

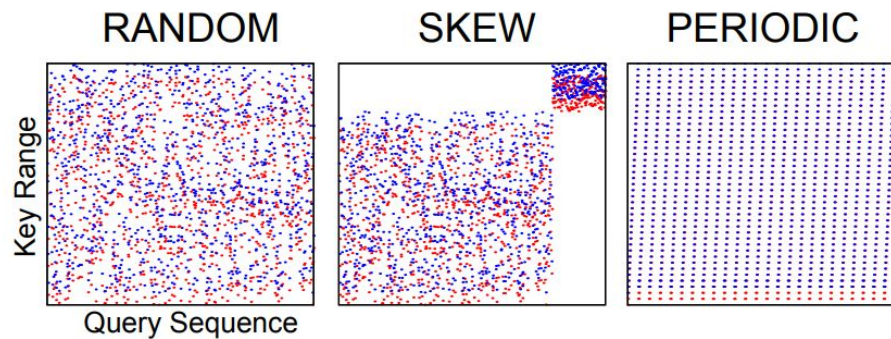
Different Query Workloads



Blue dots represent the high keys whereas red dots represent the low keys

Each Query Workload consists of 1000 range queries

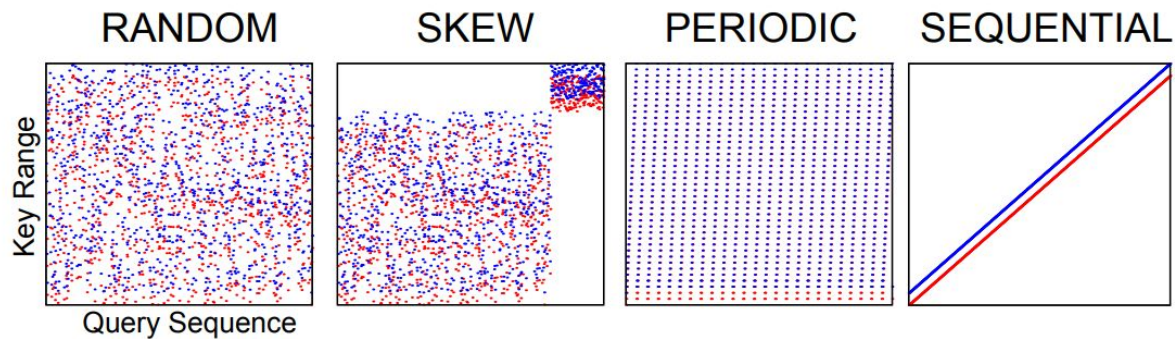
Different Query Workloads



Blue dots represent the high keys whereas red dots represent the low keys

Each Query Workload consists of 1000 range queries

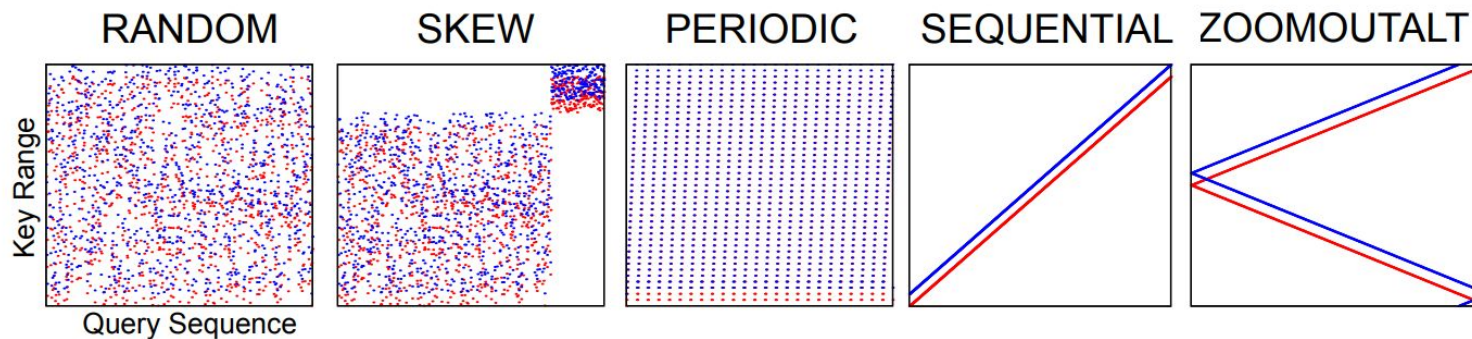
Different Query Workloads



Blue dots represent the high keys whereas red dots represent the low keys

Each Query Workload consists of 1000 range queries

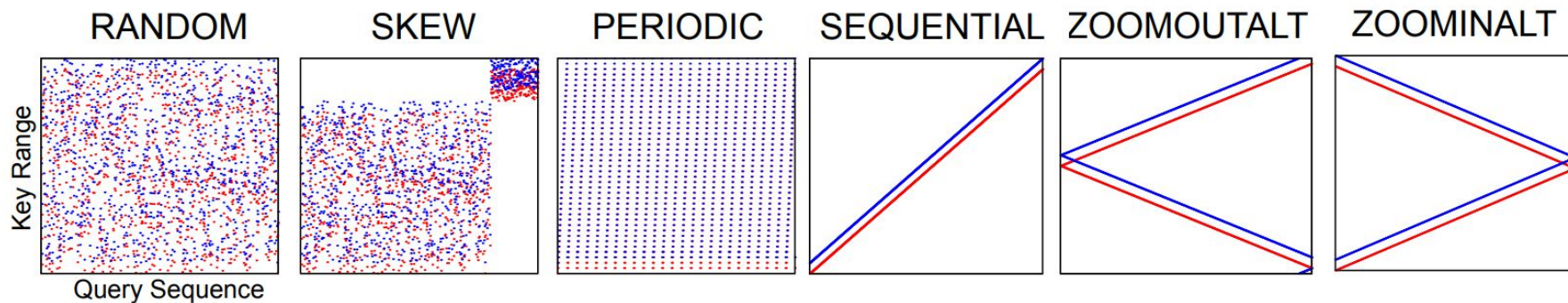
Different Query Workloads



Blue dots represent the high keys whereas red dots represent the low keys

Each Query Workload consists of 1000 range queries

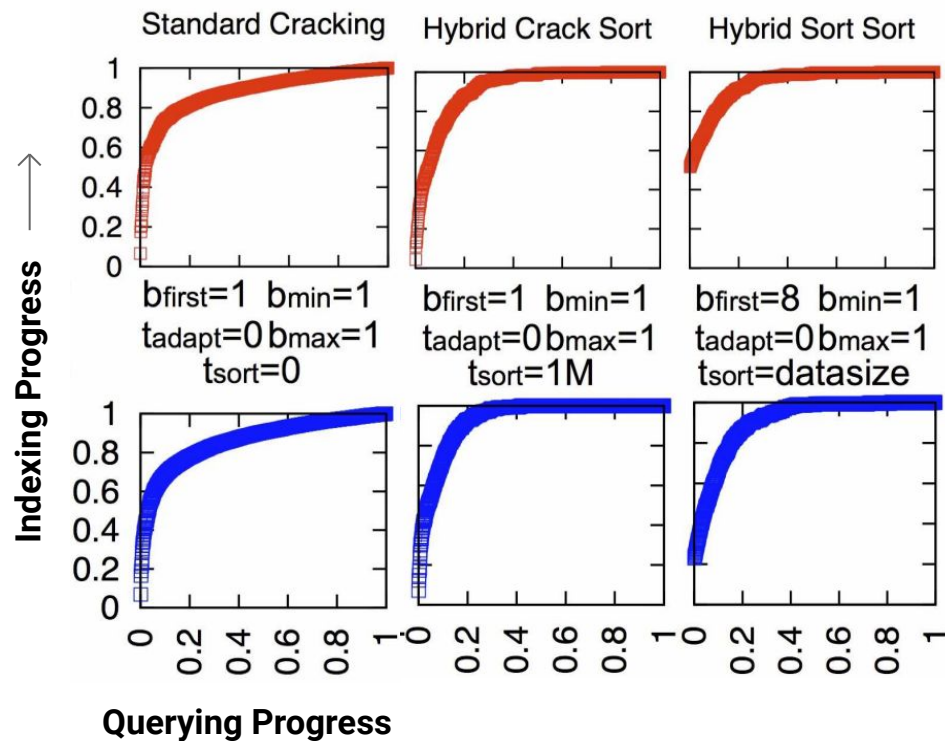
Different Query Workloads



Blue dots represent the high keys whereas red dots represent the low keys

Each Query Workload consists of 1000 range queries

Emulation of Adaptive Indexes and Traditional methods

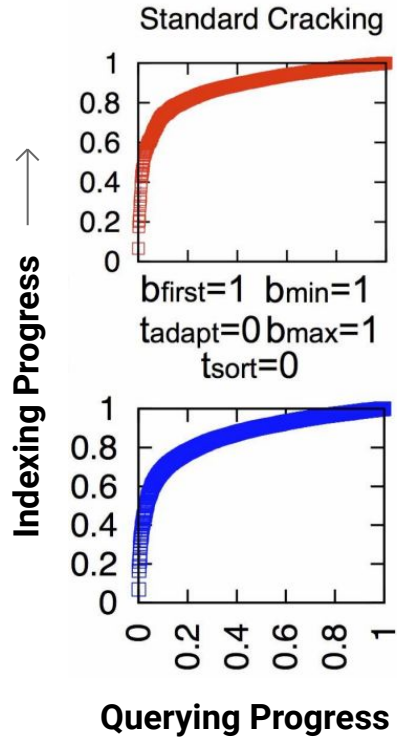


$$f(s, q) = \begin{cases} b_{first} & \text{if } q = 0 \\ b_{min} & \text{else if } s > t_{adapt} \\ b_{min} + \left[(b_{max} - b_{min}) \cdot \left(1 - \frac{s}{t_{adapt}} \right) \right] & \text{else if } s > t_{sort} \\ b_{sort} & \text{else.} \end{cases}$$

Question

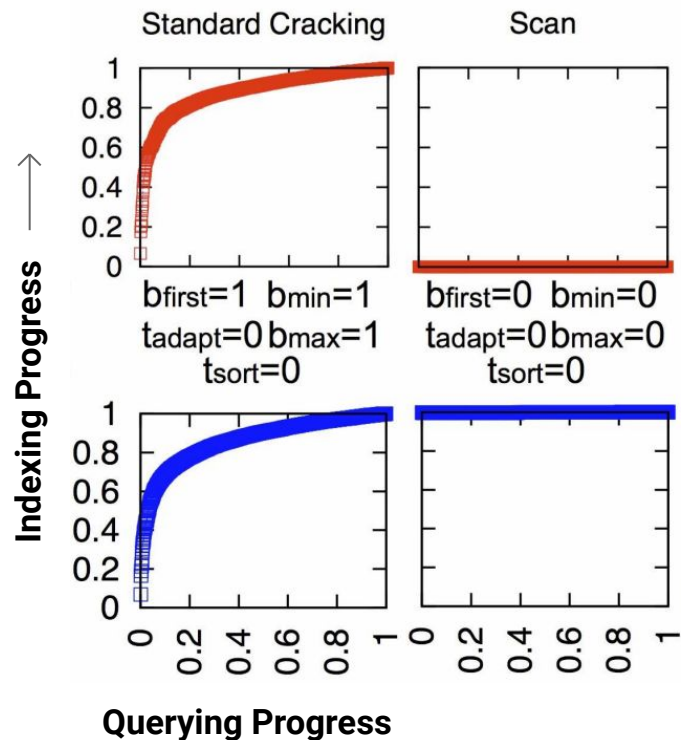
From the graph, why is Hybrid Crack Sort and Hybrid Sort Sort better than Standard Cracking?

Emulation of Adaptive Indexes and Traditional methods



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

Emulation of Adaptive Indexes and Traditional methods

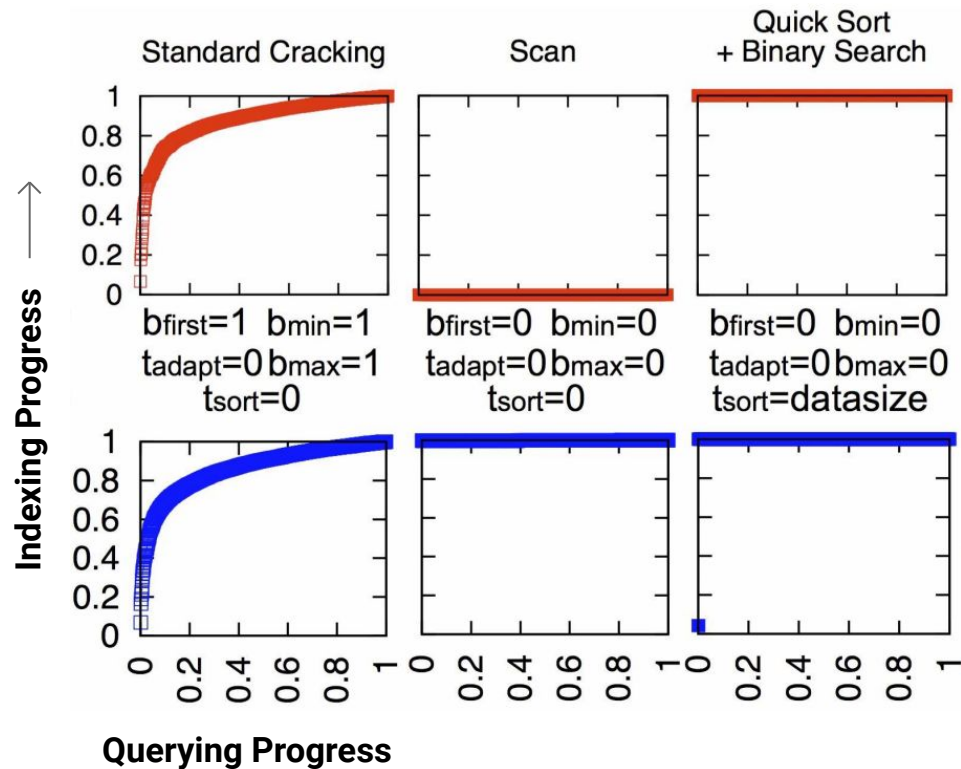


$$f(s, q) = \begin{cases} b_{first} & \text{if } q = 0 \\ b_{min} & \text{else if } s > t_{adapt} \\ b_{min} + \left[(b_{max} - b_{min}) \cdot \left(1 - \frac{s}{t_{adapt}} \right) \right] & \text{else if } s > t_{sort} \\ b_{sort} & \text{else.} \end{cases}$$

Question

In emulation of Scan (blue graph), all parameters are set to 0. So no partitions are created. Still the graph is a line at $y = 1$. Why?

Emulation of Adaptive Indexes and Traditional methods

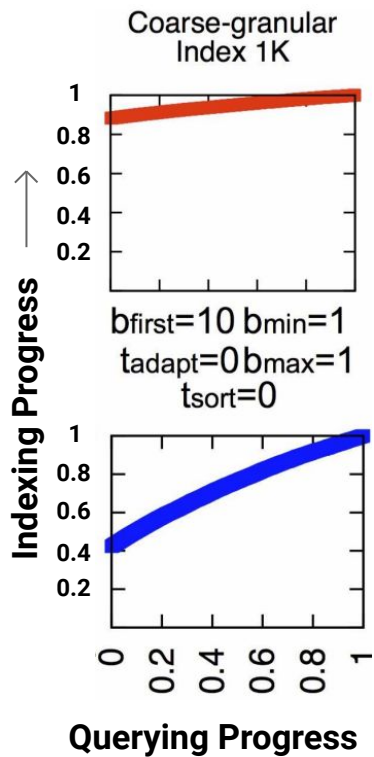


$$f(s, q) = \begin{cases} b_{first} & \text{if } q = 0 \\ b_{min} & \text{else if } s > t_{adapt} \\ b_{min} + \left[(b_{max} - b_{min}) \cdot \left(1 - \frac{s}{t_{adapt}} \right) \right] & \text{else if } s > t_{sort} \\ b_{sort} & \text{else.} \end{cases}$$

Question

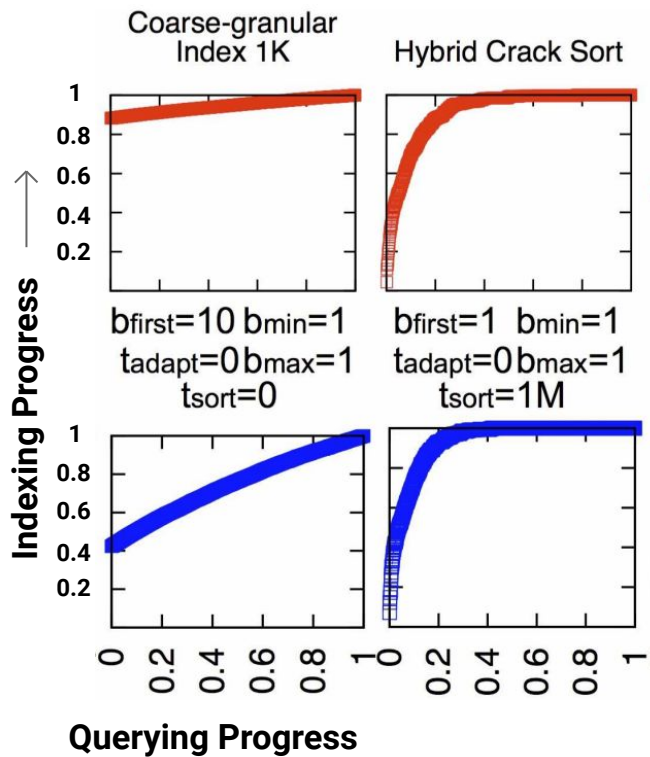
Can you guess the parameters (b_{first} , b_{min} , t_{adapt} , b_{max} , t_{sort}) for emulation of Coarse-granular Index 1K?

Emulation of Adaptive Indexes and Traditional methods



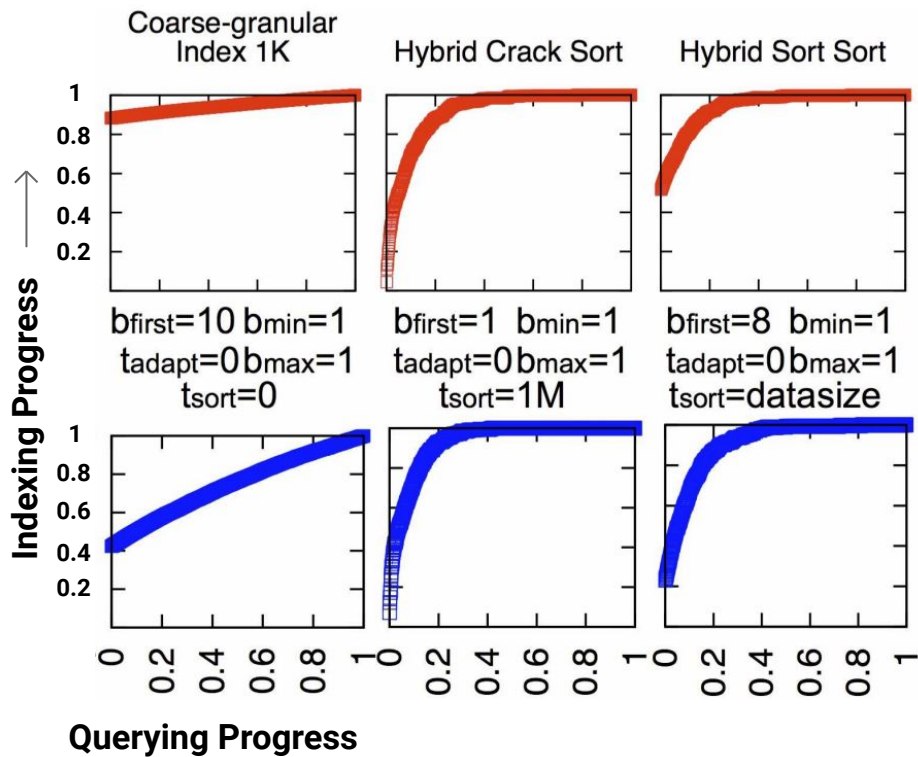
$$f(s, q) = \begin{cases} b_{first} & \text{if } q = 0 \\ b_{min} & \text{else if } s > t_{adapt} \\ b_{min} + \left[(b_{max} - b_{min}) \cdot \left(1 - \frac{s}{t_{adapt}} \right) \right] & \text{else if } s > t_{sort} \\ b_{sort} & \text{else.} \end{cases}$$

Emulation of Adaptive Indexes and Traditional methods



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

Emulation of Adaptive Indexes and Traditional methods

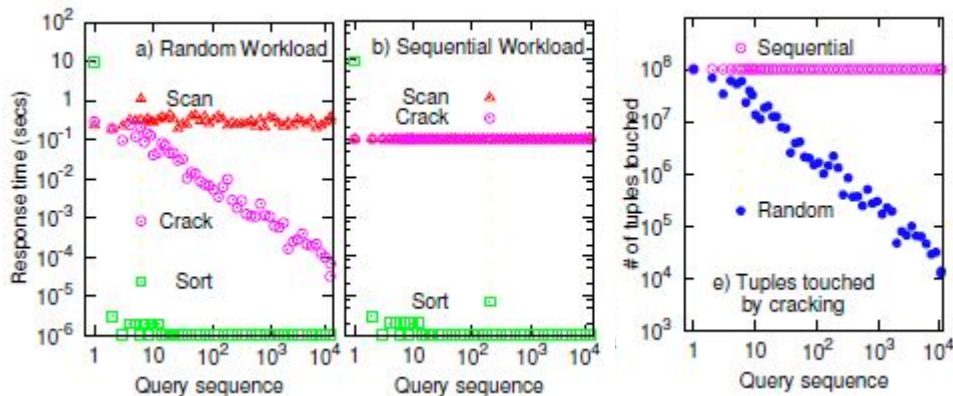


$$f(s, q) = \begin{cases} b_{first} & \text{if } q = 0 \\ b_{min} & \text{else if } s > t_{adapt} \\ b_{min} + \left[(b_{max} - b_{min}) \cdot \left(1 - \frac{s}{t_{adapt}} \right) \right] & \text{else if } s > t_{sort} \\ b_{sort} & \text{else.} \end{cases}$$

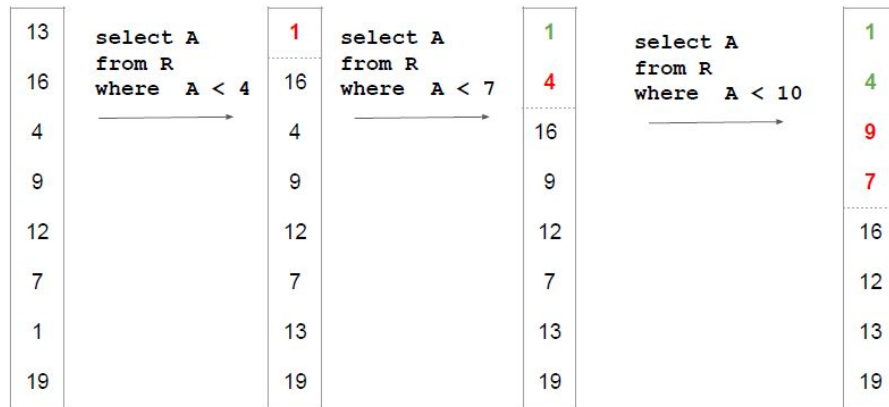
Cracking Performance

Crack fails to deliver the performance improvements in sequential workload.

With the sequential workload, Crack touches a large number of tuples, which **falls only negligibly as new queries arrive**, whereas with the random workload the number of touched tuples drops swiftly after only a few queries.



$N = 8$



C = # of comparisons required to answer query

N

N - 1

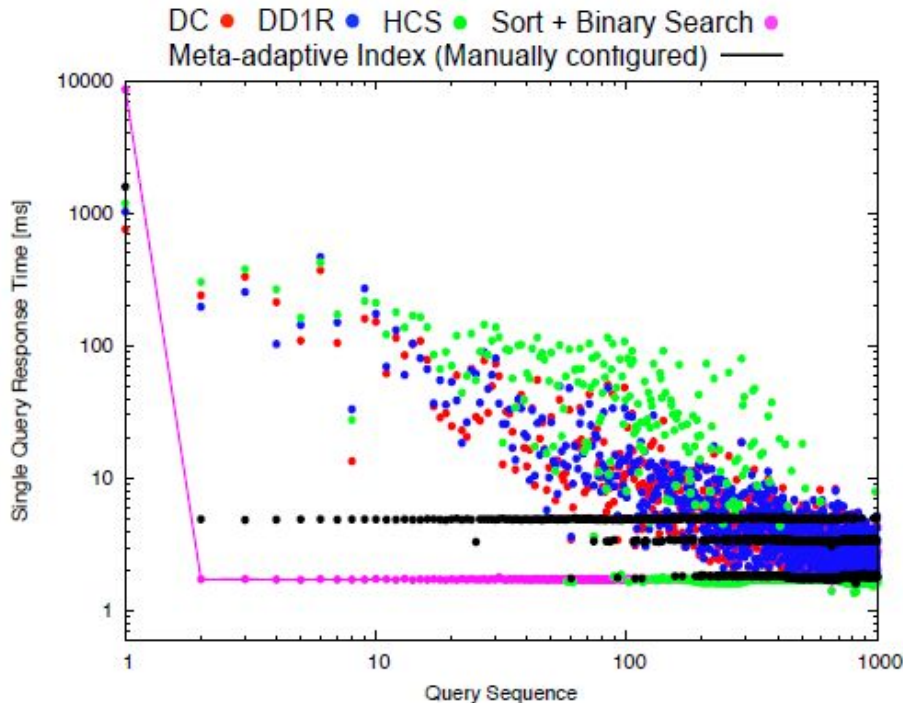
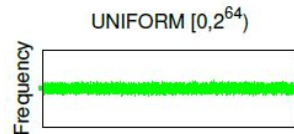
N - 2

Evaluation of Out-of-place Radix Partitioning

The meta-adaptive index shows the most stable performance and offers early on fast individual response times, similar to the full index.

Question: why does all algos become faster and faster?

The touched tuple become smaller and smaller.



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

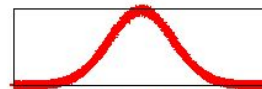
Evaluation of Out-of-place Radix Partitioning

The meta-adaptive index becomes unstable, but still more stable than others.

Question: why does it becomes less stable?

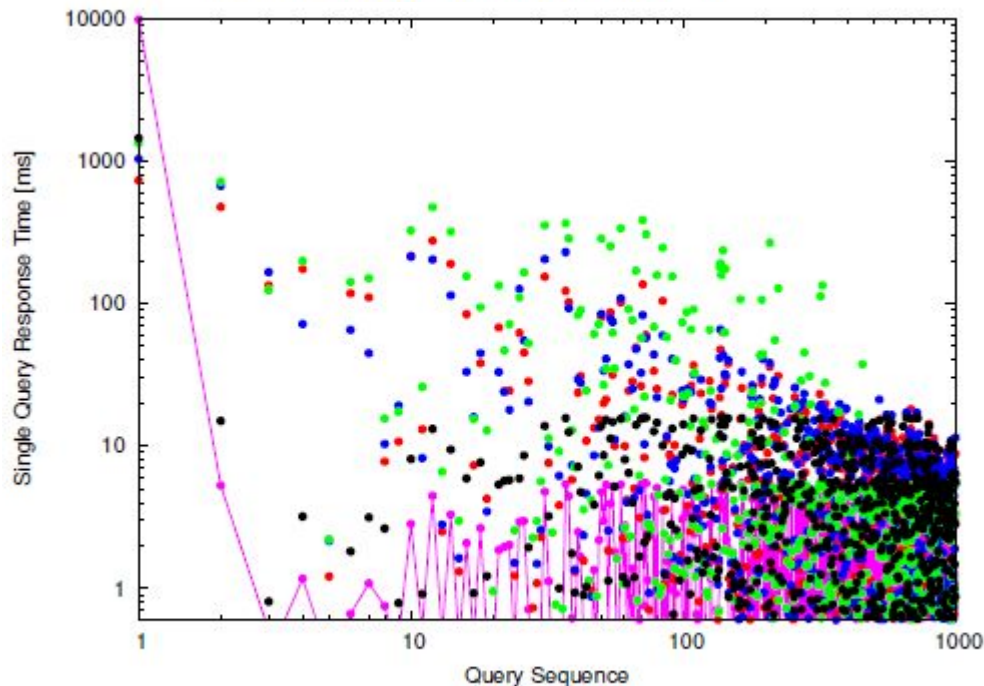
Because many values fall into a narrow range.

NORMAL ($\mu=2^{63}, \sigma=2^{61}$)



Key range

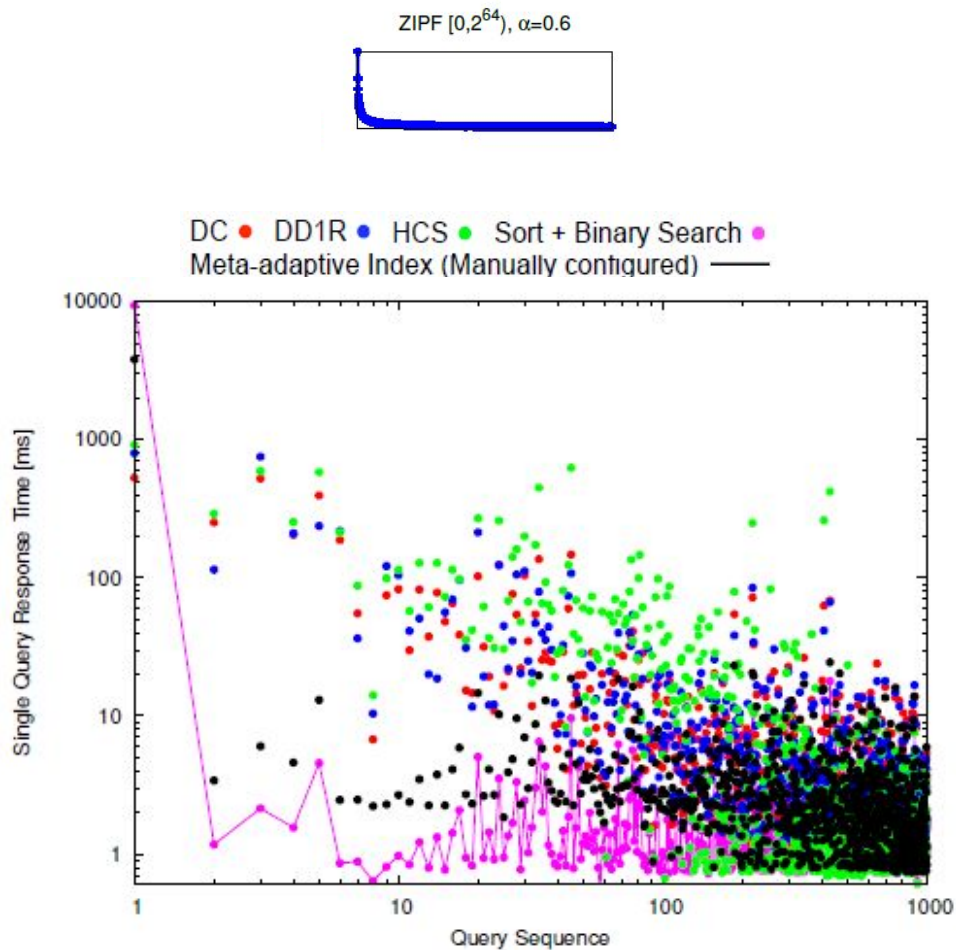
DC ● DD1R ● HCS ● Sort + Binary Search ●
Meta-adaptive Index (Manually configured) —



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

Evaluation of Out-of-place Radix Partitioning

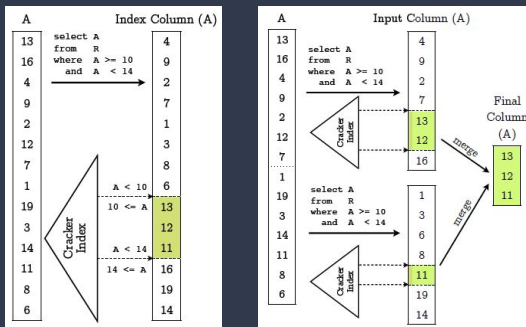
The worse case, as most values fall into few partitions. But meta-adaptive index is still the best one.



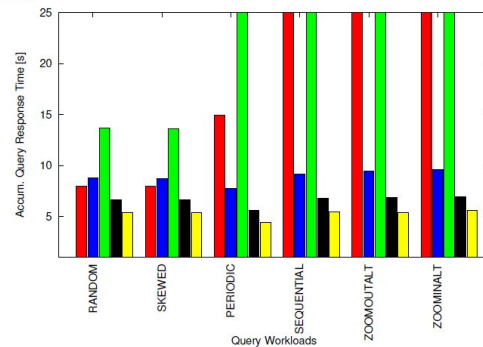
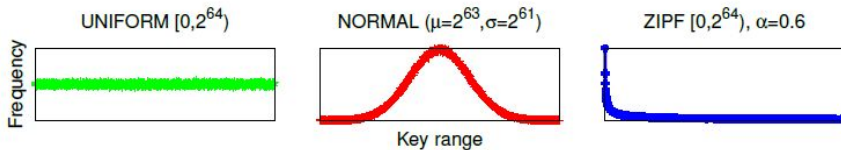
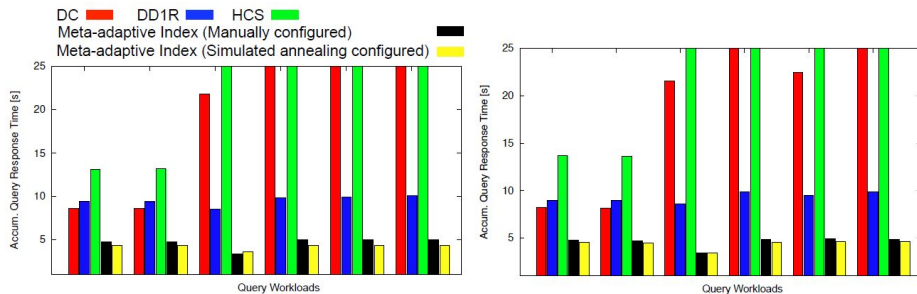
(c) $\mathcal{Z}(\min = 0, \max = 2^{64} - 1, \alpha = 0.6)$

Accumulated Query Response Time

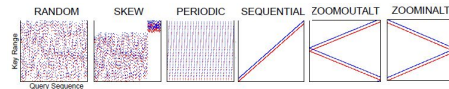
Question: Why does the HCS even slower than DC?



Because HCS is calibrated for sequential queries?? Also, merge is not free.



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



Conclusion

1. Minimal overhead compared to previous work, with better results.
2. Performs well in almost every workloads (uniform, normal and zipf distribution).
3. Optimize parameters using simulated annealing algorithm.

What can we improve?

1. Simulating annealing algorithms has some cons. If the cooling process is slow enough, the performance of more solutions will be better, but in contrast to this, **the convergence speed is too slow**. If the cooling process is too fast, **the global optimal solution may not be obtained**.
2. **Introduce more adaptive index algorithms** to fully enhanced the robustness of the meta-adaptive index. Also, they could try **MDD1R** stochastic cracking algorithm instead of DD1R.