Adaptive Adaptive Indexing

Paper Review by Manuja DeSilva & Michael Hendrick

Table of Contents

Meet The Authors

Felix Martin Schunknecht

Saarland University PostDoc

Jens Dittrich

Saarland University Prof.

Laurent Linden

Saarland University RA

The Problem

How do we quickly and efficiently answer range queries on a database, without performing manual tuning ?

The Problem

How do we quickly and efficiently answer range queries on a database, without performing manual tuning ?

Adaptively build indexes !

Why Even Index

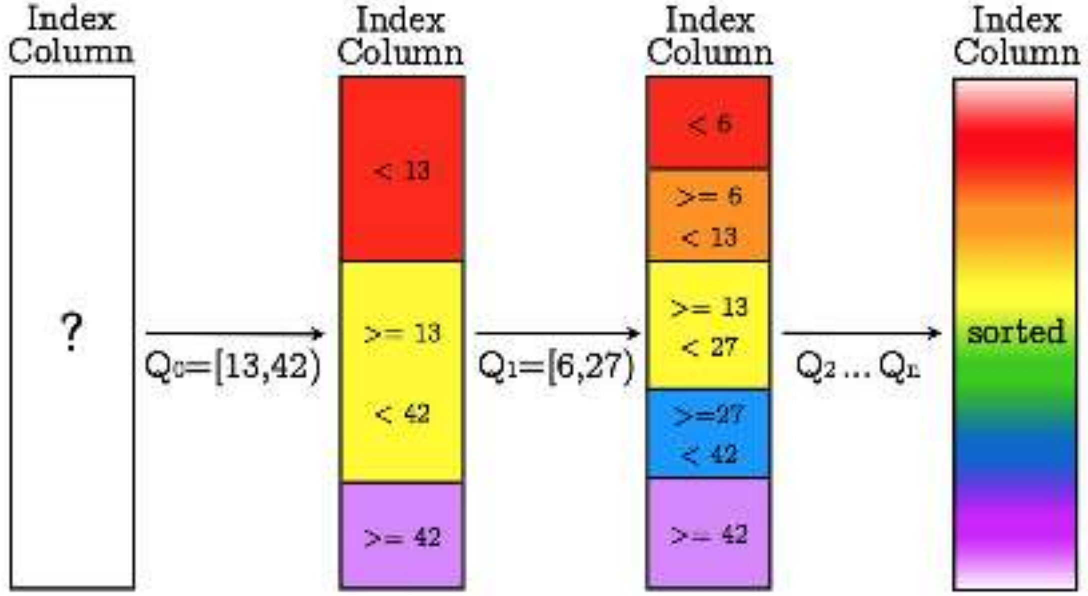


Fig. 1: ***Concept*** *of database cracking reorganizing for multiple queries and converging towards a sorted state.*

Existing work

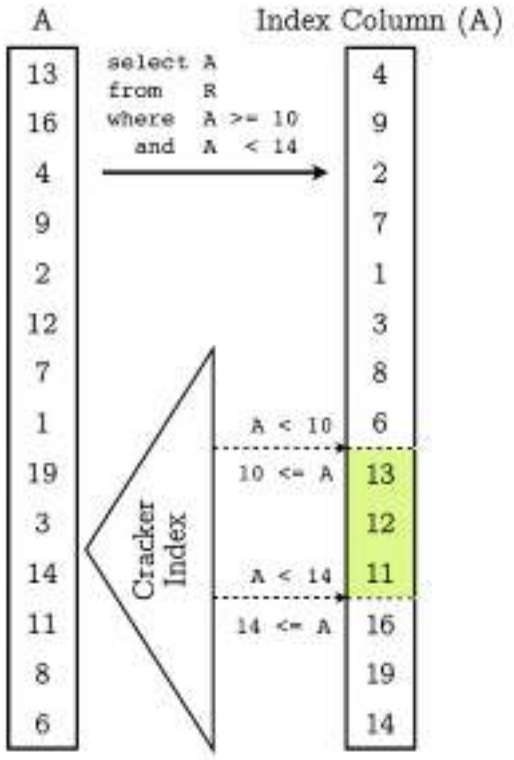
There already exists many types of adaptive indexes. Why do we need another one ?

Existing work

There already exists many types of adaptive indexes. Why do we need another one ?

Each of the other indexes only solves a very specific problem.

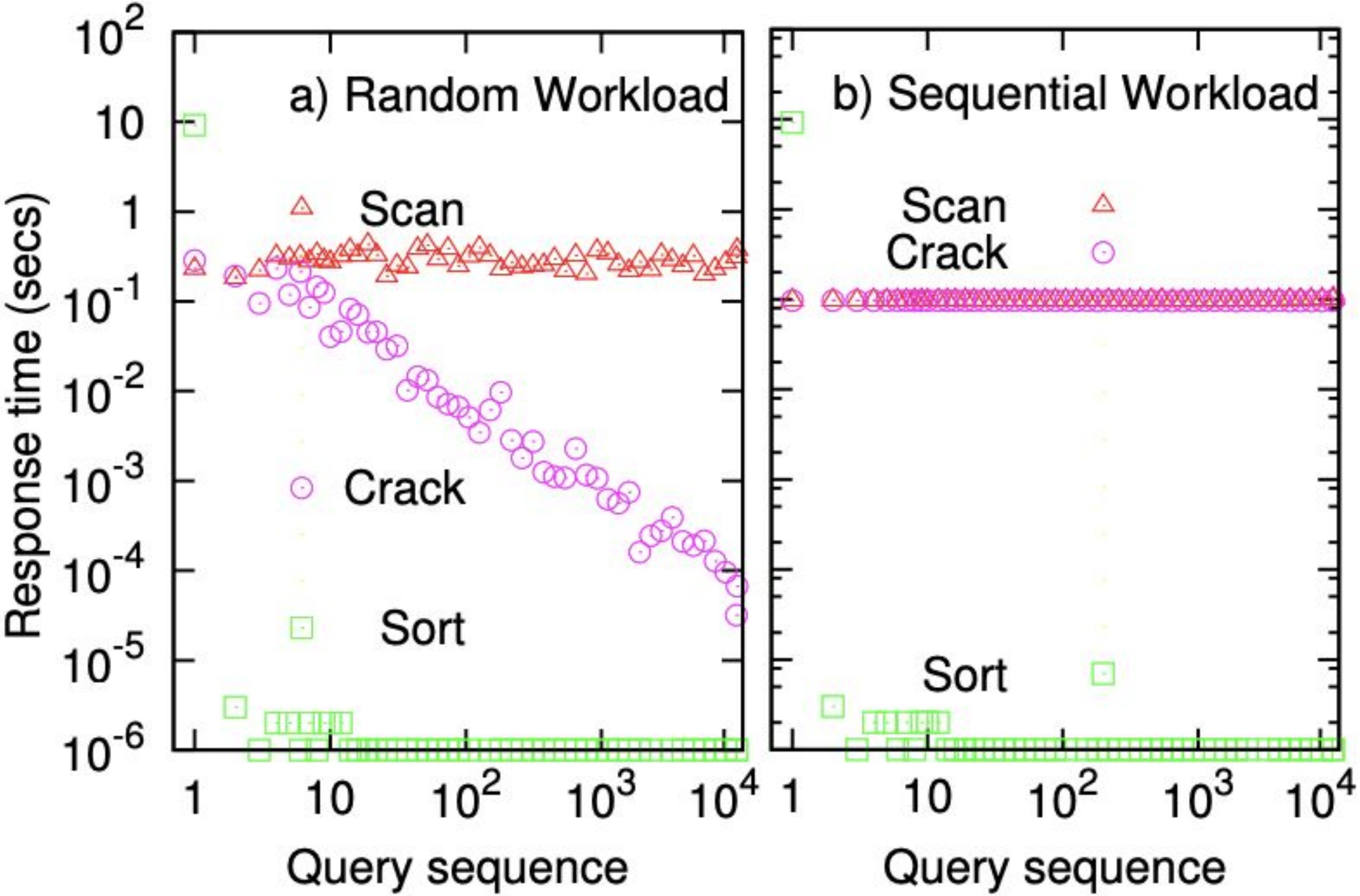
Standard Cracking (Database Cracking)



(a) Standard Cracking (**DC**)

$Cheap **→ performs the least amount of reorganization (crack in 2)**

Poor performance **-> Although it does well with random workloads, it performs the same as a *Scan* with sequential workloads**

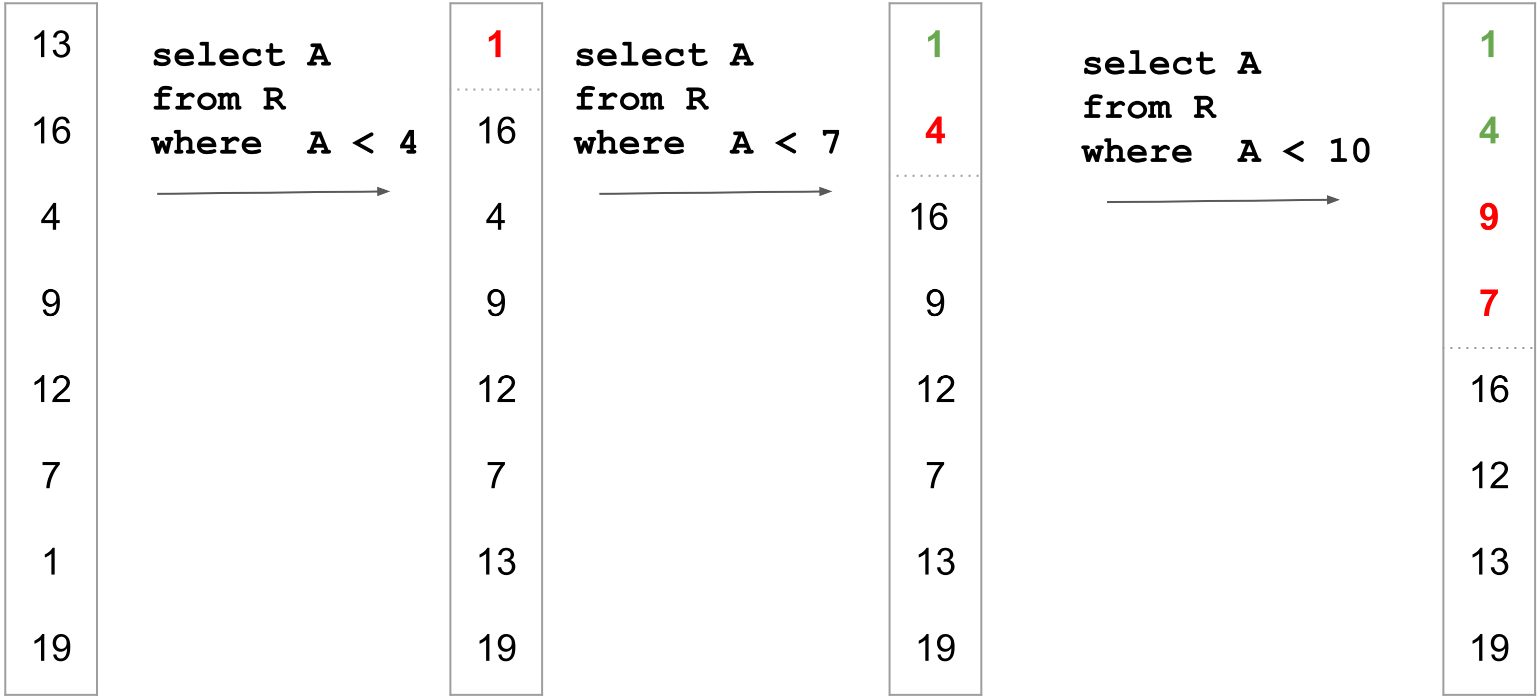


DC Performance (random vs sequential workloads)

[1]

What is a sequential workload ?

N = 8



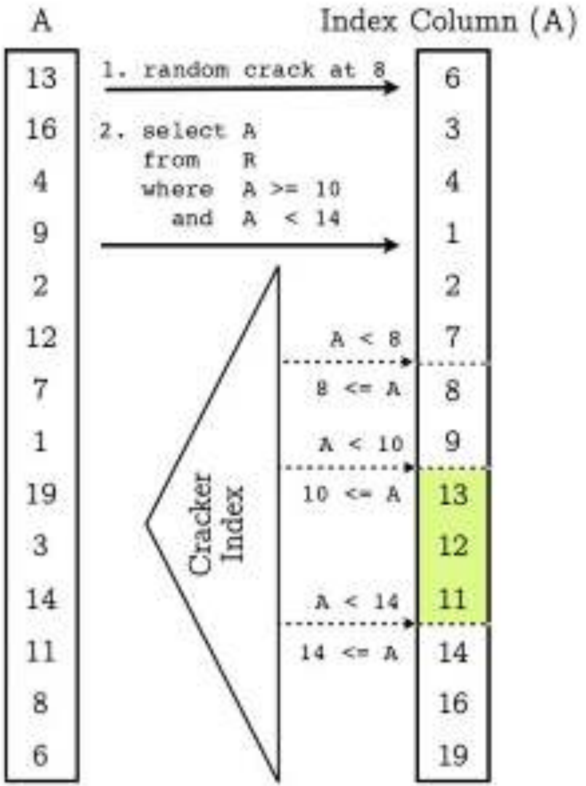
C = # of comparisons required to answer query

N

N -1

N - 2

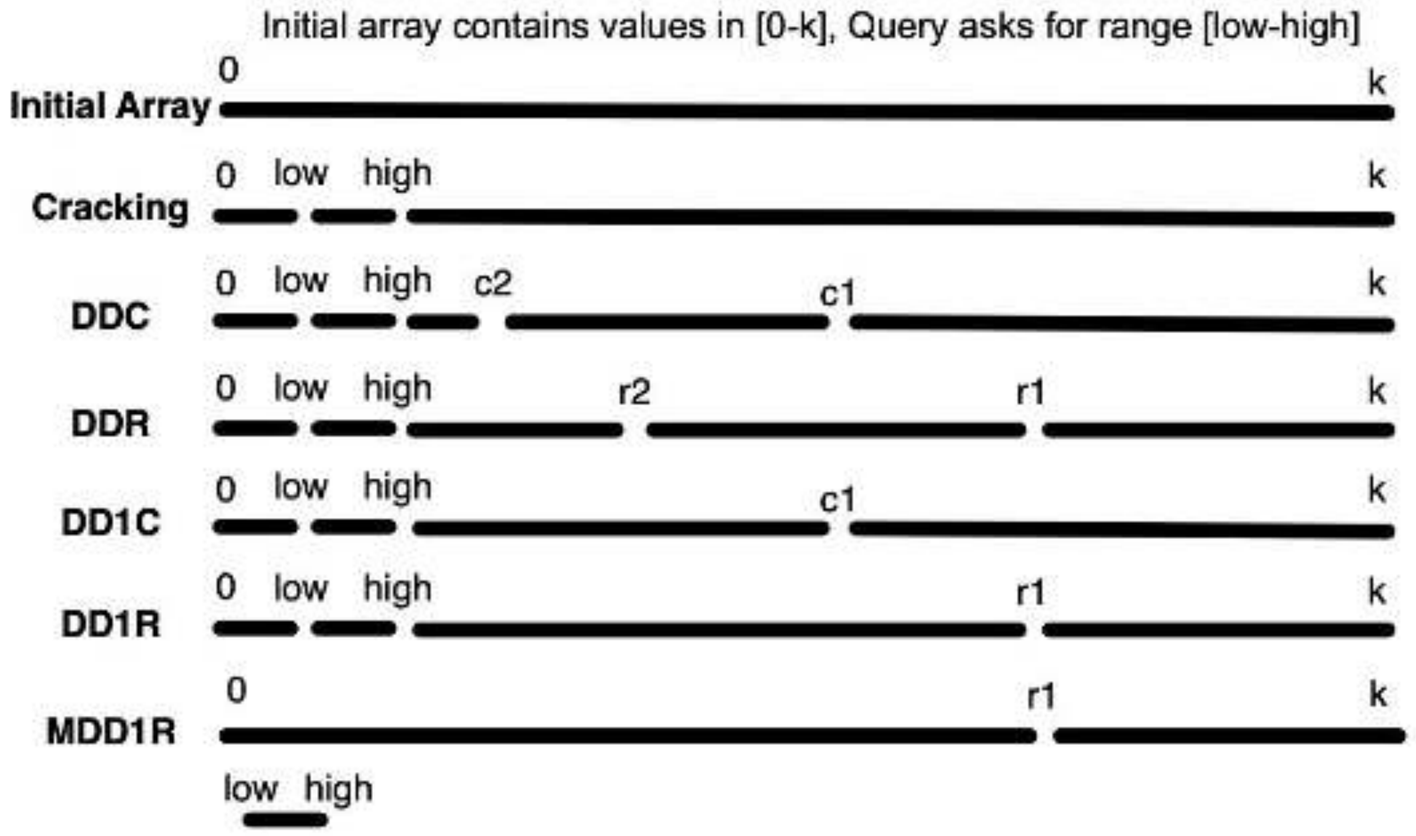
Stochastic Cracking



(b) Stochastic Cracking (**DD1R**)

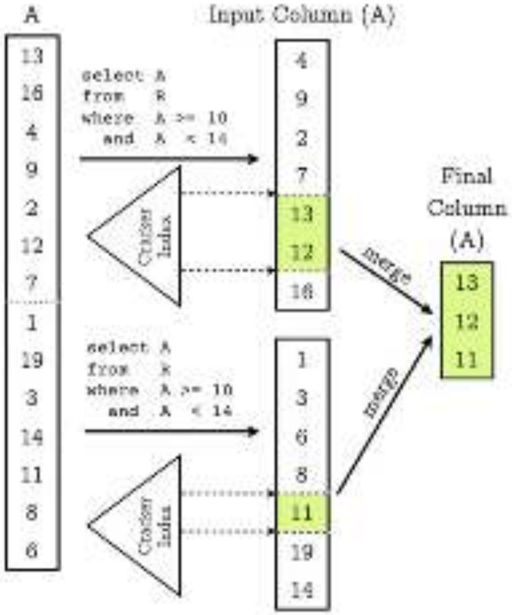
Picks up where standard cracking left off - e.g DC only partitions based on the query itself, which leaves a large part of the index still unsorted

Solution: ***Introduce random cracks in addition to the query crack***



Visual representation of Stochastic Cracking Algorithms

Hybrid Cracking



(c) Hybrid Cracking (**HCS**). For **HSS**, the inputs are sorted.

Database cracking has a slow convergence speed

Adaptive merging has a large memory footprint

***Solution:* Split the inputs into partitions (DC), merge the final column**

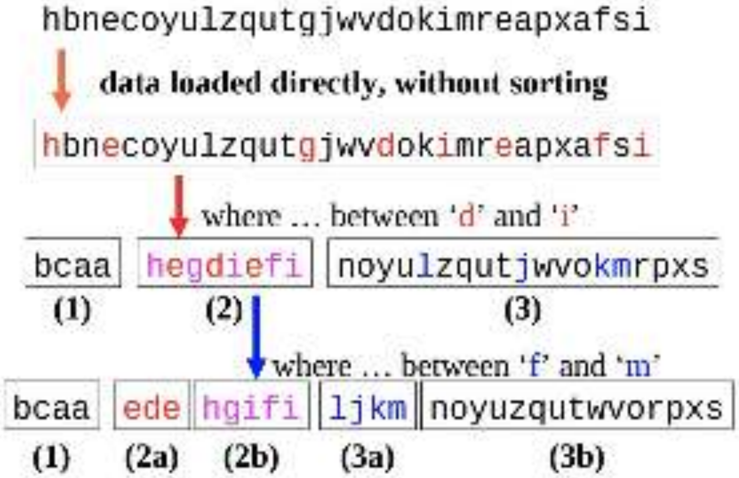


Figure 2: Database cracking.

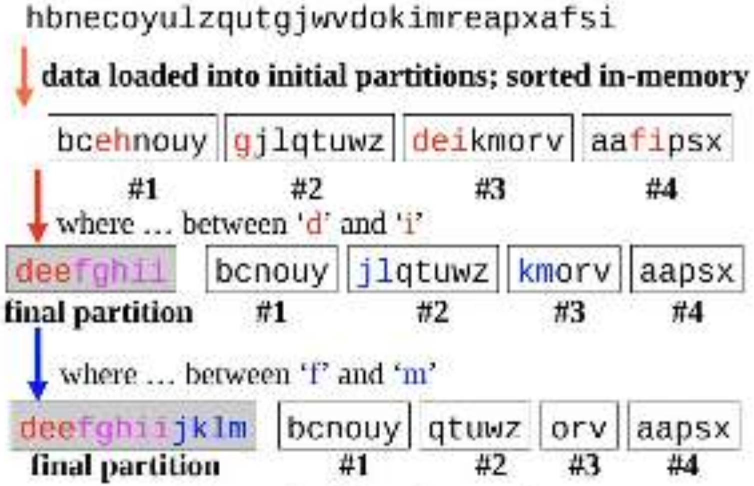
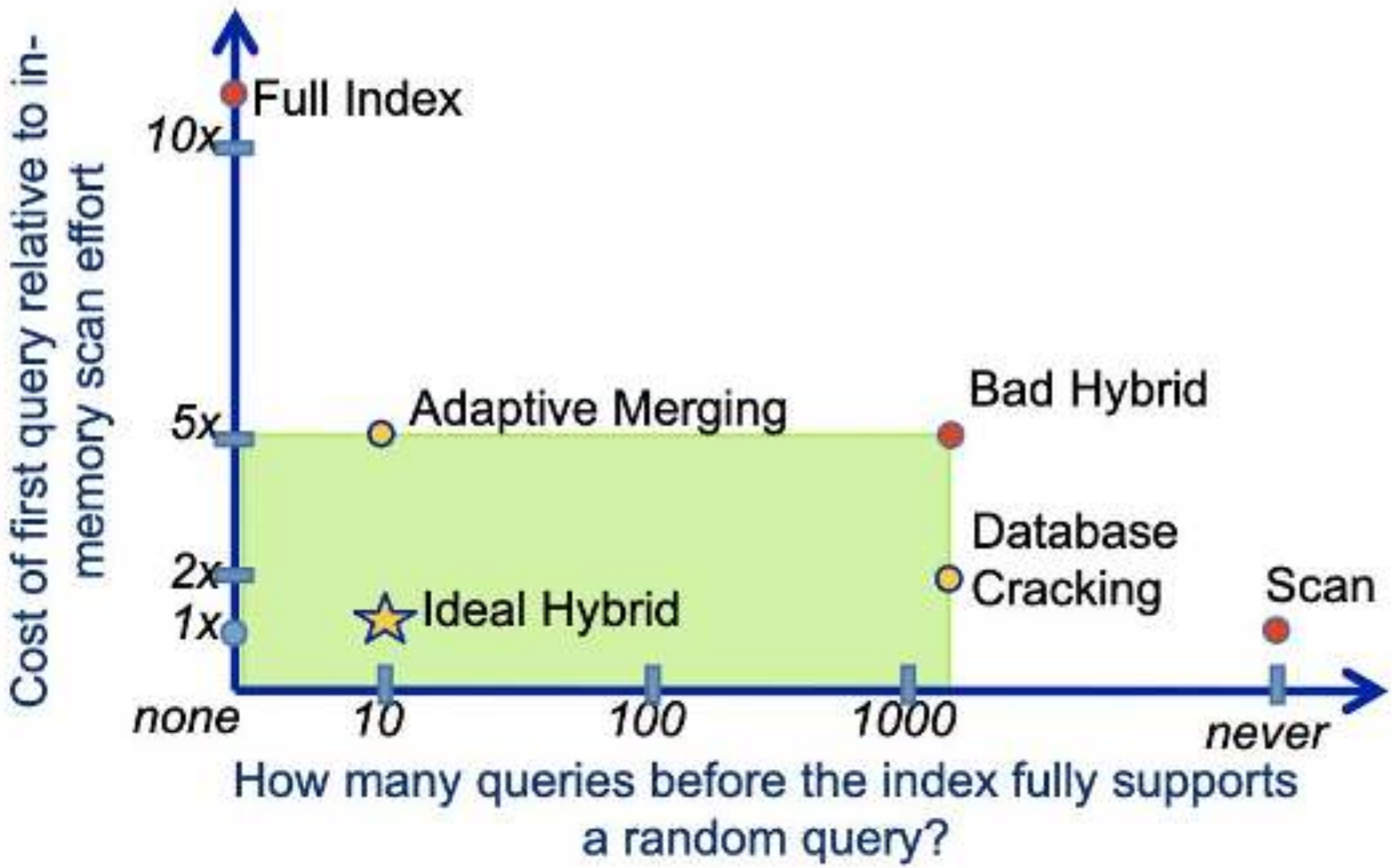


Figure 3: Adaptive merging.

[2]


Cost of first query relative to in-memory scan effort
Full Index
10x
5x
2x
1x
Adaptive Merging
Bad Hybrid
Database Cracking
Ideal Hybrid
Scan
none
10
100
1000
never
How many queries before the index fully supports a random query?

The solution to all other solutions:

Adaptive adaptive indexing

Partitioning

Classical approaches revolve around comparison based methods for calculating partitions.

What's the problem with this ?

Partitioning

Classical approaches revolve around comparison based methods for calculating partitions.

What's the problem with this ?

The partitions are solely dependent on the inputted queries and the raw data itself, which doesn't follow any schema

The Solution:

Radix Partitioning

What is Radix Partitioning?

Number	Binary
1	0001
2	0010
7	0111
5	0101
3	0011
4	0100

Number	Binary
1	**00** | 01
2	**00** | 10
7	**01** | 11
5	**01** | 01
3	**00** | 11
4	**01** | 00

Partition	Elements
Partition 1 (00)	1,2,3
Partition 2 (01)	7,5,4

Out of Place Radix Partitioning

Inputs: The ***source*** column and the ***number*** (k) of requested partitions

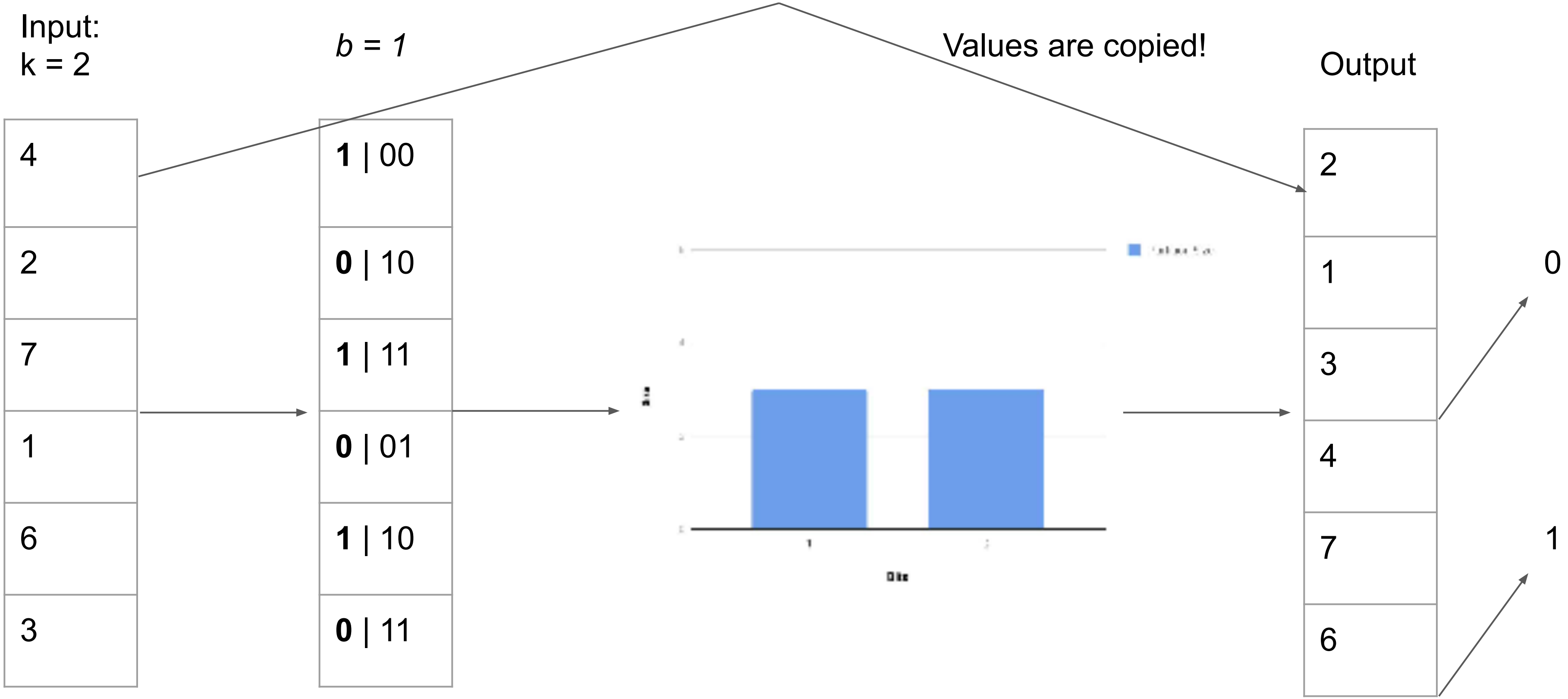
k is calculated as **k = 2^f** *(more on this in a bit)*

Phase 1: Create a ***Histogram***

Phase 2: Copy ***Entries***

Let’s look at an example!

Out of Place Radix Partitioning



What is TLB?

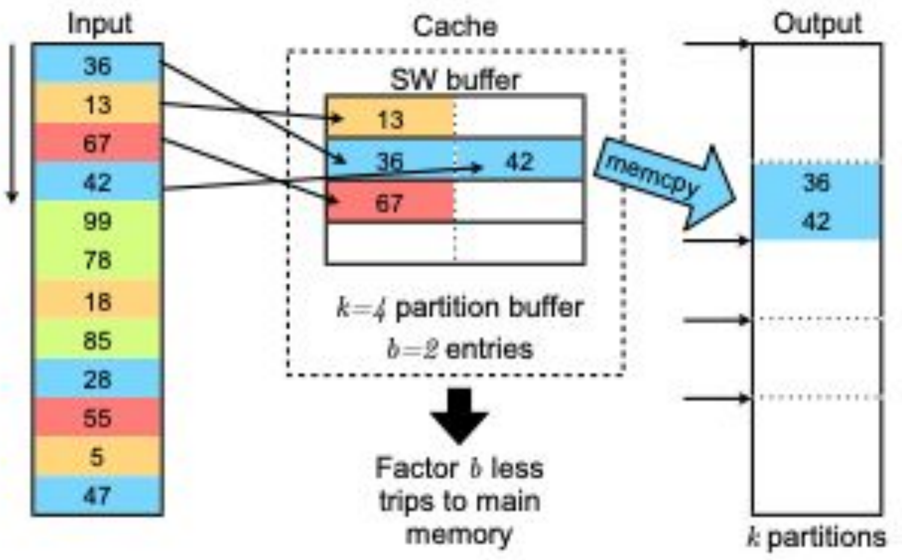
Translation Lookaside Buffer

TLB stores a **mapping** of <u>virtual mem to physical mem</u> for quick lookups

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

If there’s a TLB hit, great! But how to handle misses?

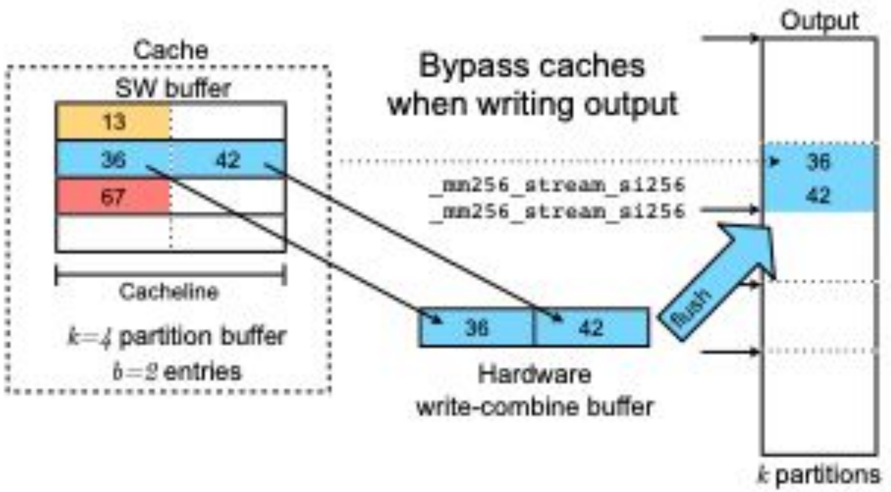
Software-Managed Buffers



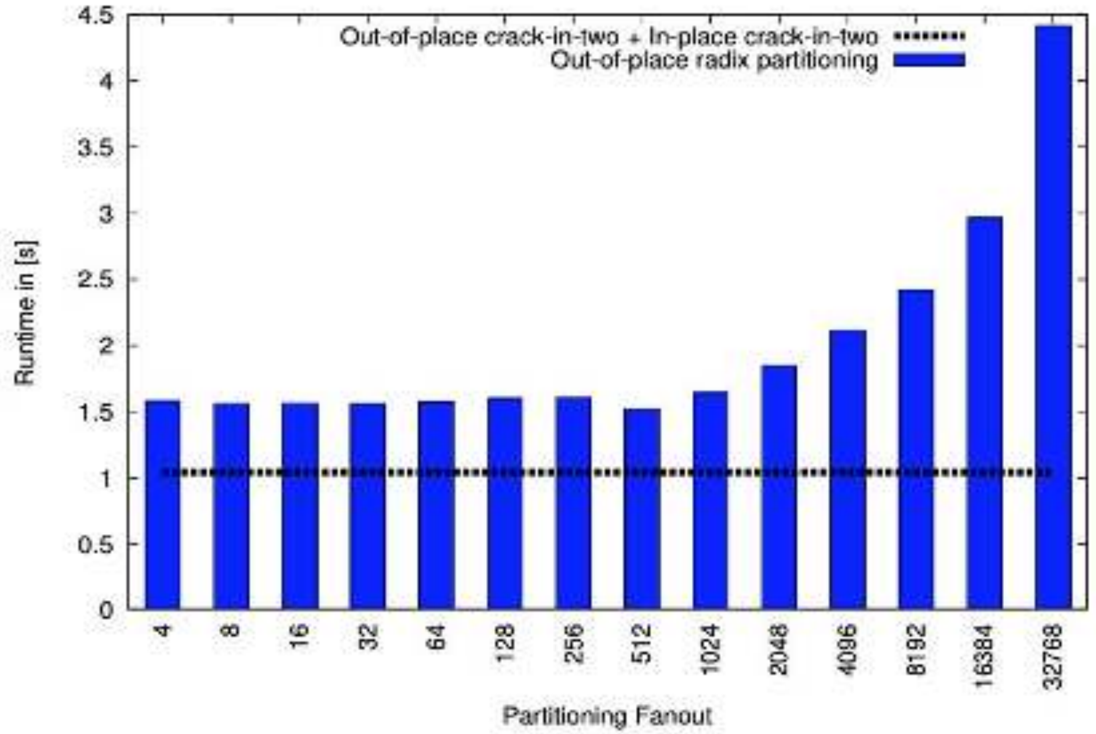
Non-temporal streaming stores and SIMD

add r0 r1 r2

add |r3| |r6] |r9|
|r4| |r7] |r10|
|r5| |r8] |r11|



Evaluation of Out of Place Radix Partitioning



In Place Radix Partitioning: Subsequent Queries

Contrary to **Out of Place**, all **subsequent** queries must **reorganize in-place**

Standard cracking reorganizes data using [low,high] inputs given by the query

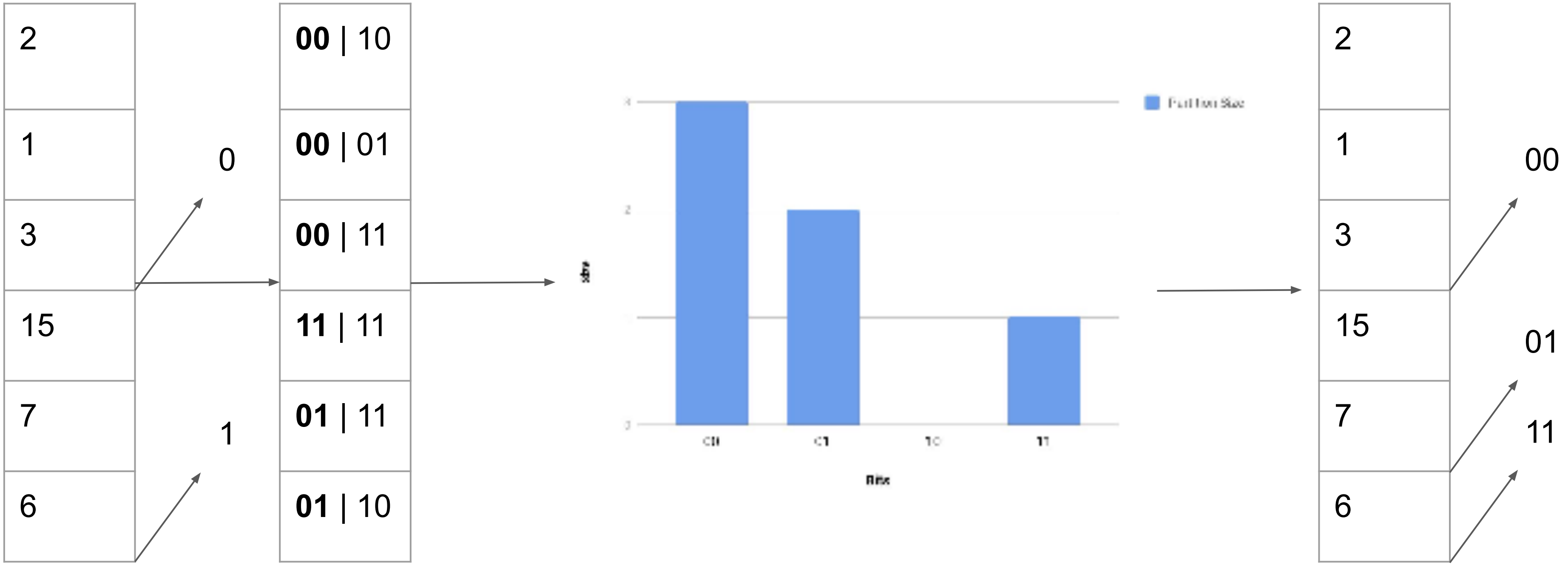
Phase 1: Create a ***histogram*** that tells us the amount of values in each partition

Phase 2: Perform a ***search and replace*** through the index column.

In-Place Example

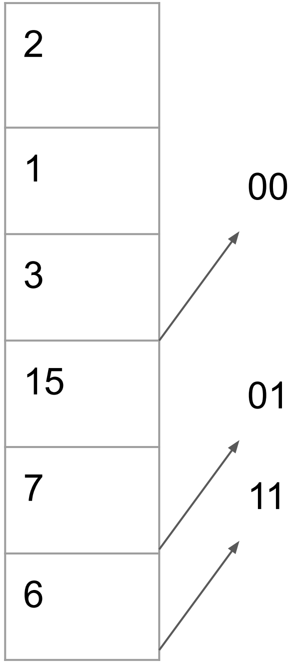
Input:
k = 4

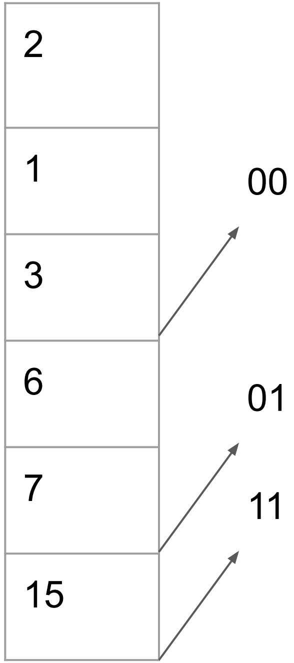
b = 2



In-Place Example

2	**00** | 10
1	**00** | 01
3	**00** | 11
15	**11** | 10
7	**01** | 11
6	**01** | 10





Is 2 in the right place? **Yes!**

Is 1 in the right place? **Yes!**

Is 3 in the right place? **Yes! 00 is done!**

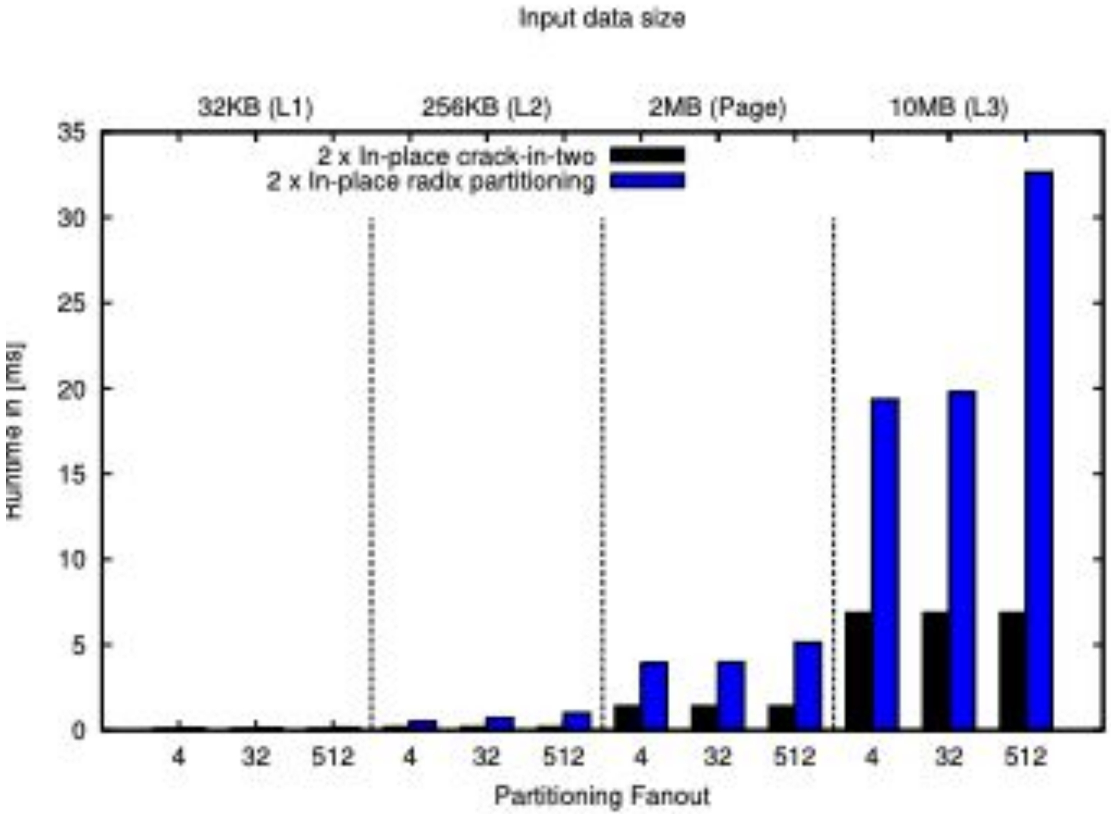
Is 15 in the right place? **No! Swap within 11.**

Is 6 in the right place? **Yes!**

Is 7 in the right place? **Yes, 01 is done!**

Is 15 in the right place? **Yes, 11 is done!**

Evaluation of In Place Radix Partitioning



The meta-adaptive indexing algorithm

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.

2

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

The meta-adaptive indexing algorithm in-action

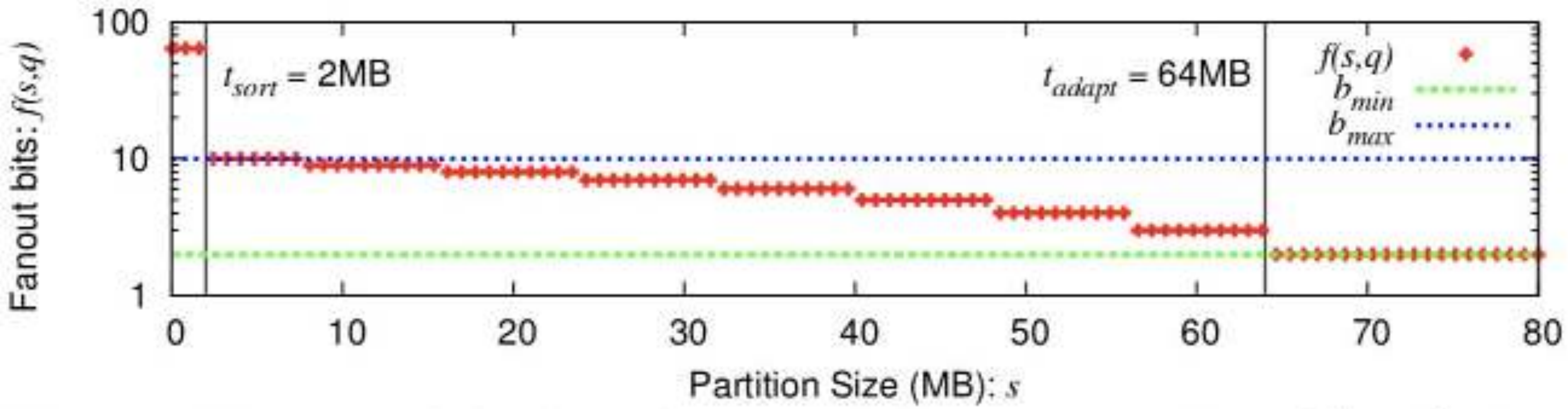
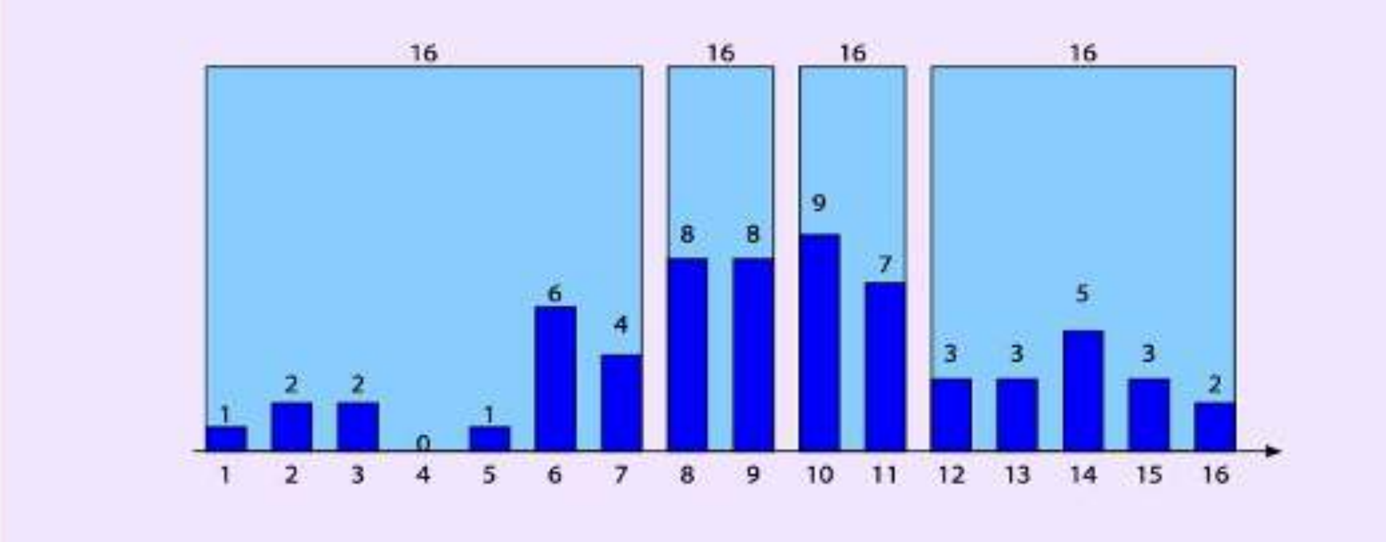


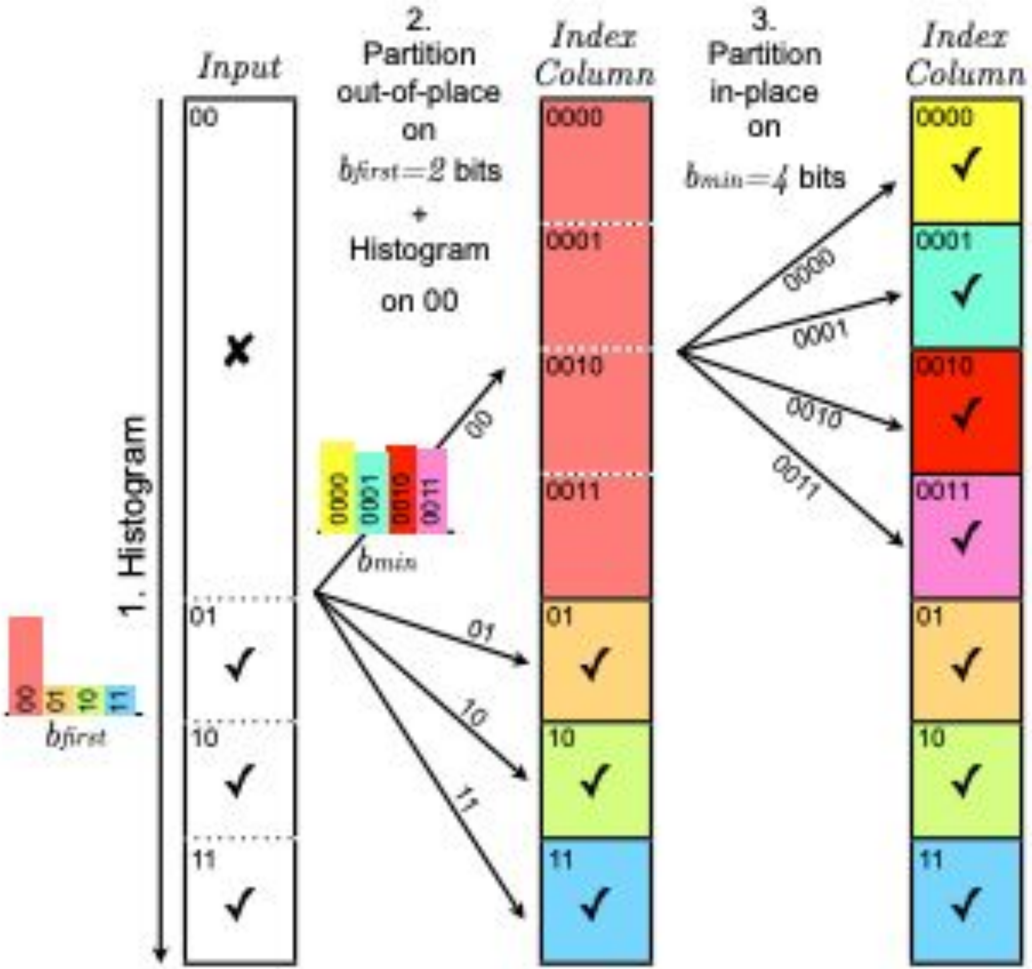
Fig. 5: The **partitioning fan-out bits** *returned by* $f(s,q)$ *for partition sizes* s *from* $0MB$ *to* $80MB$ *and* $q > 0$ *with* $t_{adapt} = 64MB$, $b_{min} = 2$, $b_{max} = 10$, $t_{sort} = 2MB$, *and* $b_{sort} = 64$.

Handling Skew

- Radix partitioning might not handle skewed distributions well (**Why?**)
- Solution: **Equi-depth** histograms and out of place radix partitioning.
- Not ***quite*** perfect for radix partitioning (**Why**?).



Handling Skew



What is the effect of differing key distributions ?

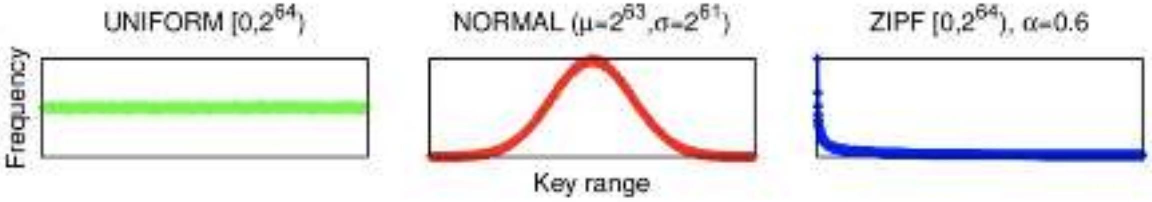


Fig. 8: Different **key distributions** used in the experiments.

What is the effect of differing key distributions ?

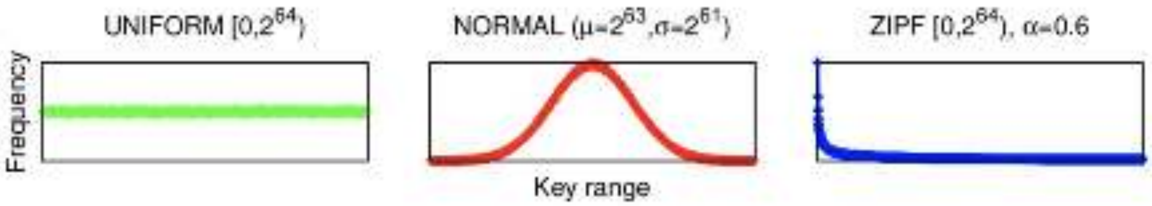


Fig. 8: Different **key distributions** used in the experiments.

Different key distributions affect the *skew.*

Query workload

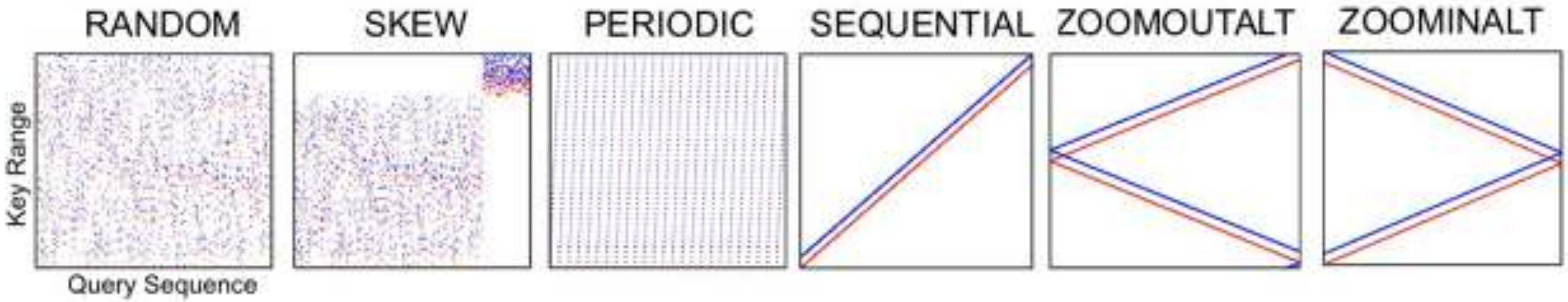


Fig. 9: Different ***query workloads****. Blue dots represent the $high$ keys whereas red dots represent the low keys.*

Experimental Evaluation

Two Tests

How well can the meta-adaptive index emulate other indexes ?

How do the response times of the meta-adaptive index compare to other indexes ?

How much memory are we working with ?

32KB of *L1* cache

256KB of *L2* cache

10MB of shared *L3* cache

2MB Page Size

24GB of DDR3 RAM

How much memory are we working with ?

32KB of *L1* cache

256KB of *L2* cache

10MB of shared *L3* cache

2MB Page Size

24GB of DDR3 RAM

Why are these numbers important ?

How much memory are we working with ?

32KB of *L1* cache

256KB of *L2* cache

10MB of shared *L3* cache

2MB Page Size

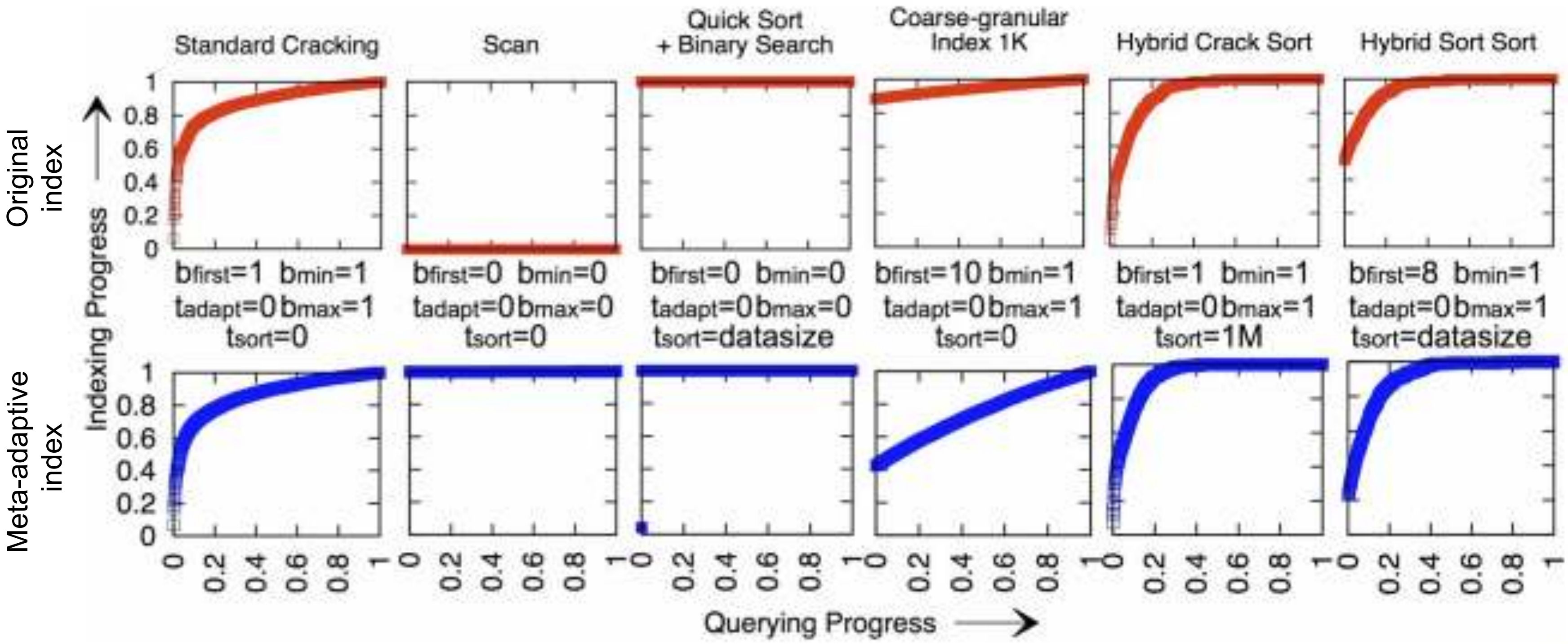
24GB of DDR3 RAM

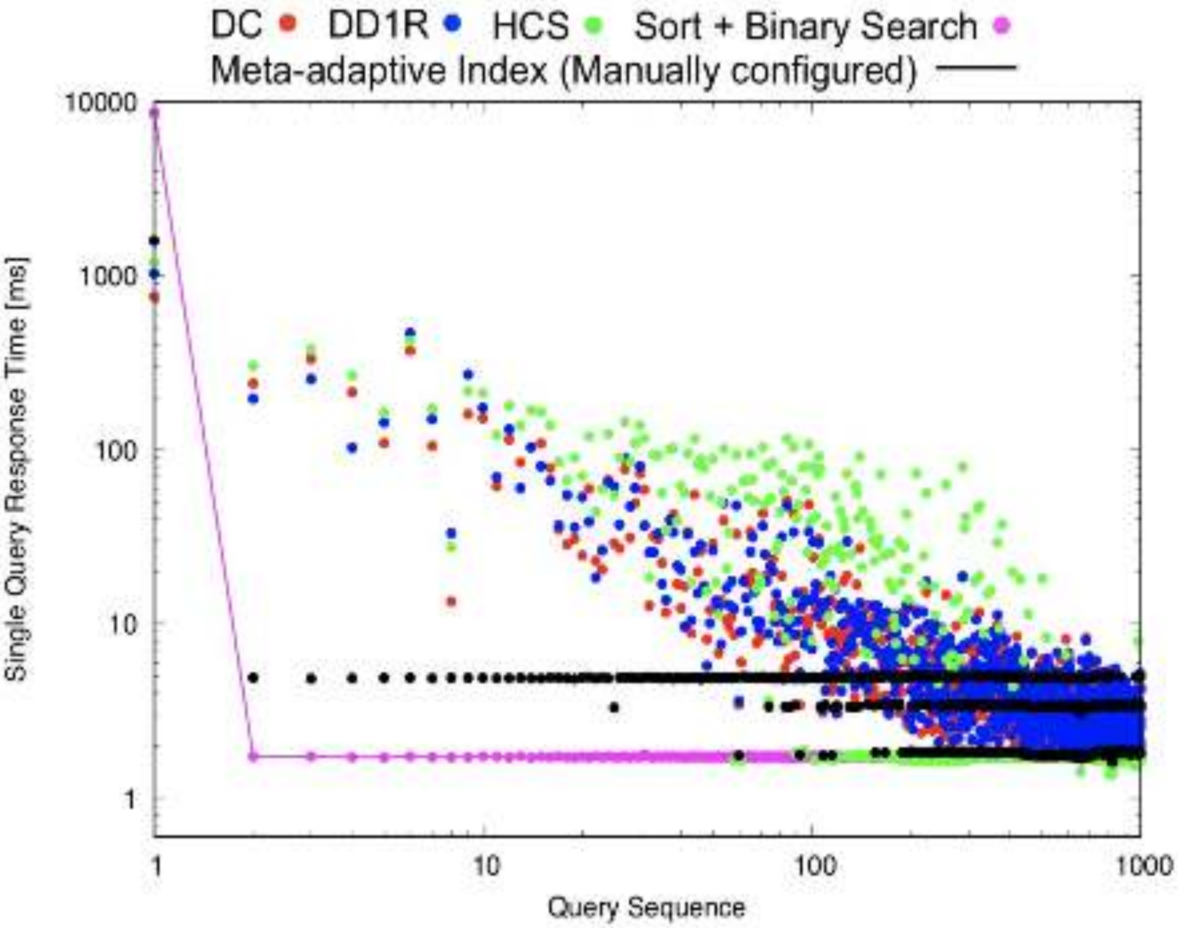
These could potentially be the values of t_{adapt} and t_{sort}

Our dataset

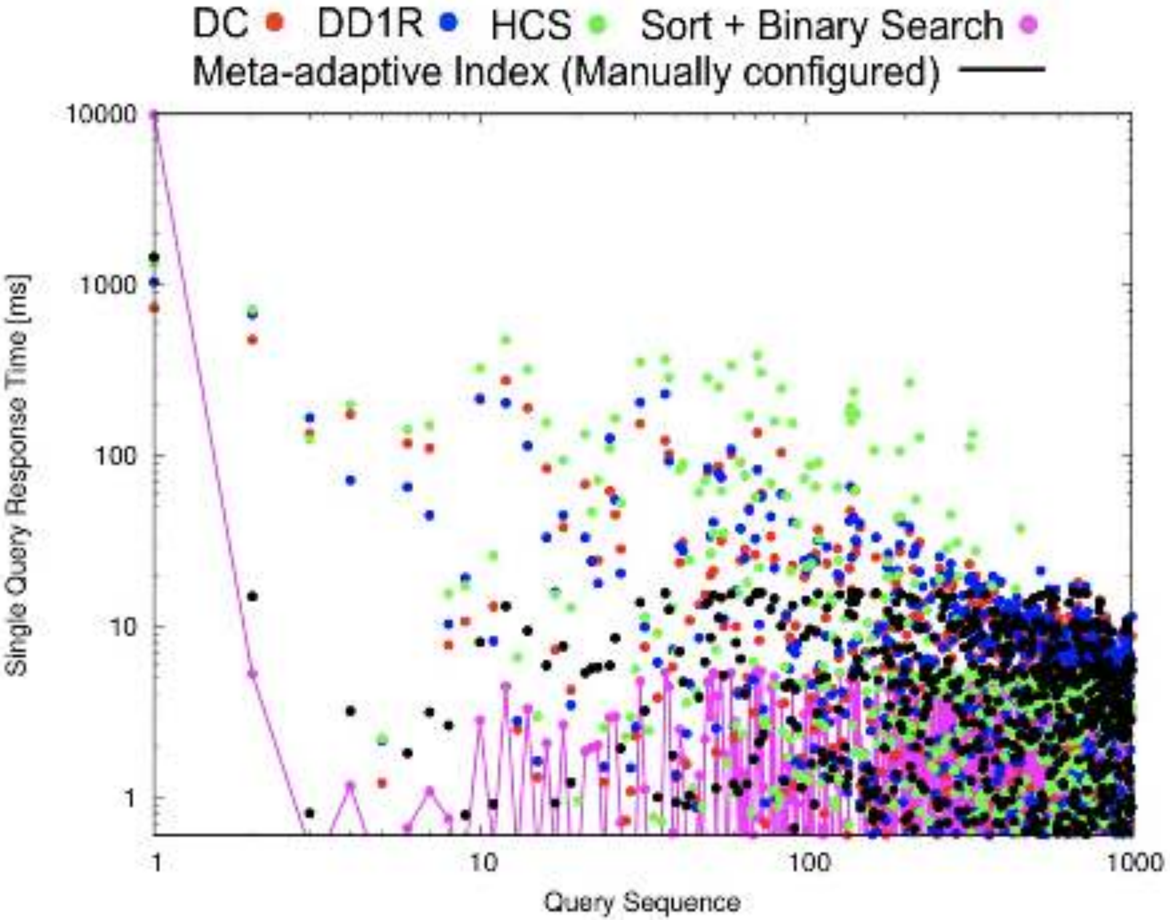
About 1.5GB of data, around 100 million entries consisting of 8B keys

Emulation of adaptive indexes and traditional methods

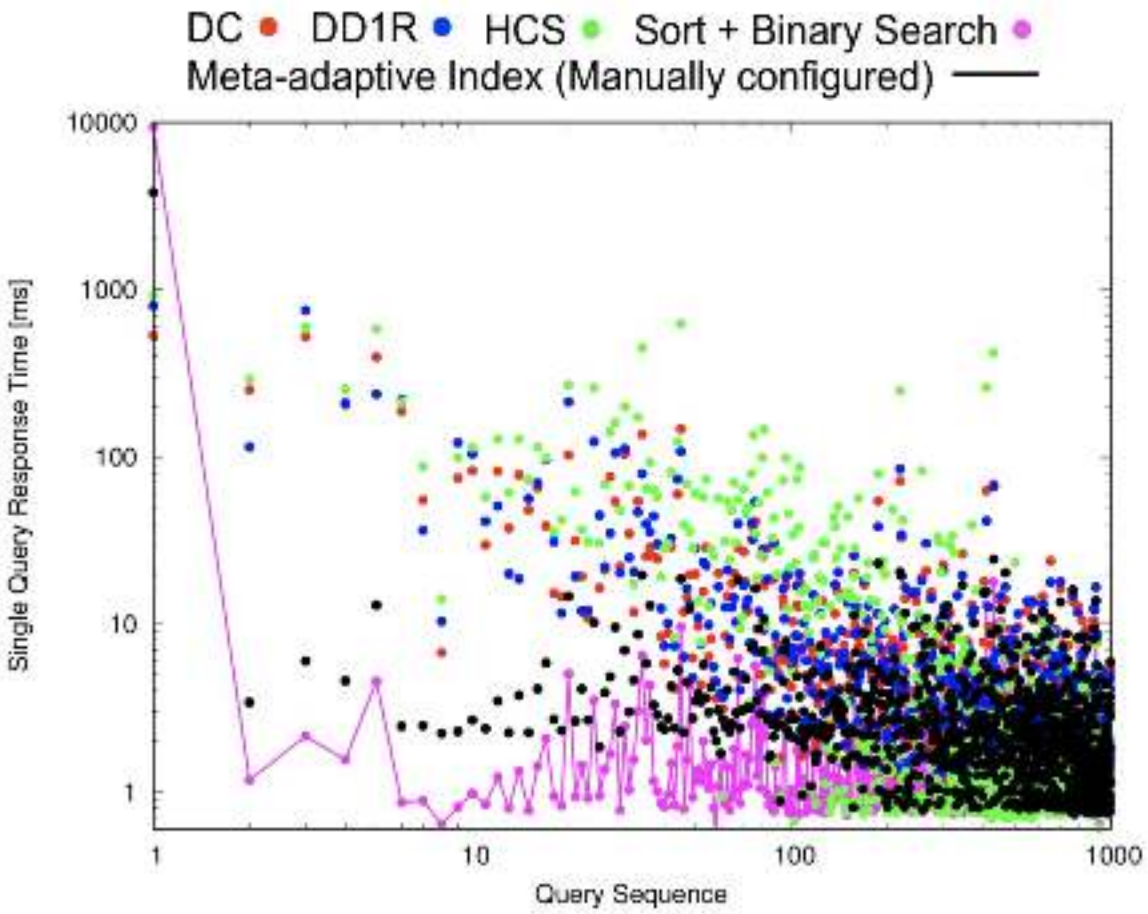




(a) $\mathcal{U}(min = 0, max = 2^{64} - 1)$



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

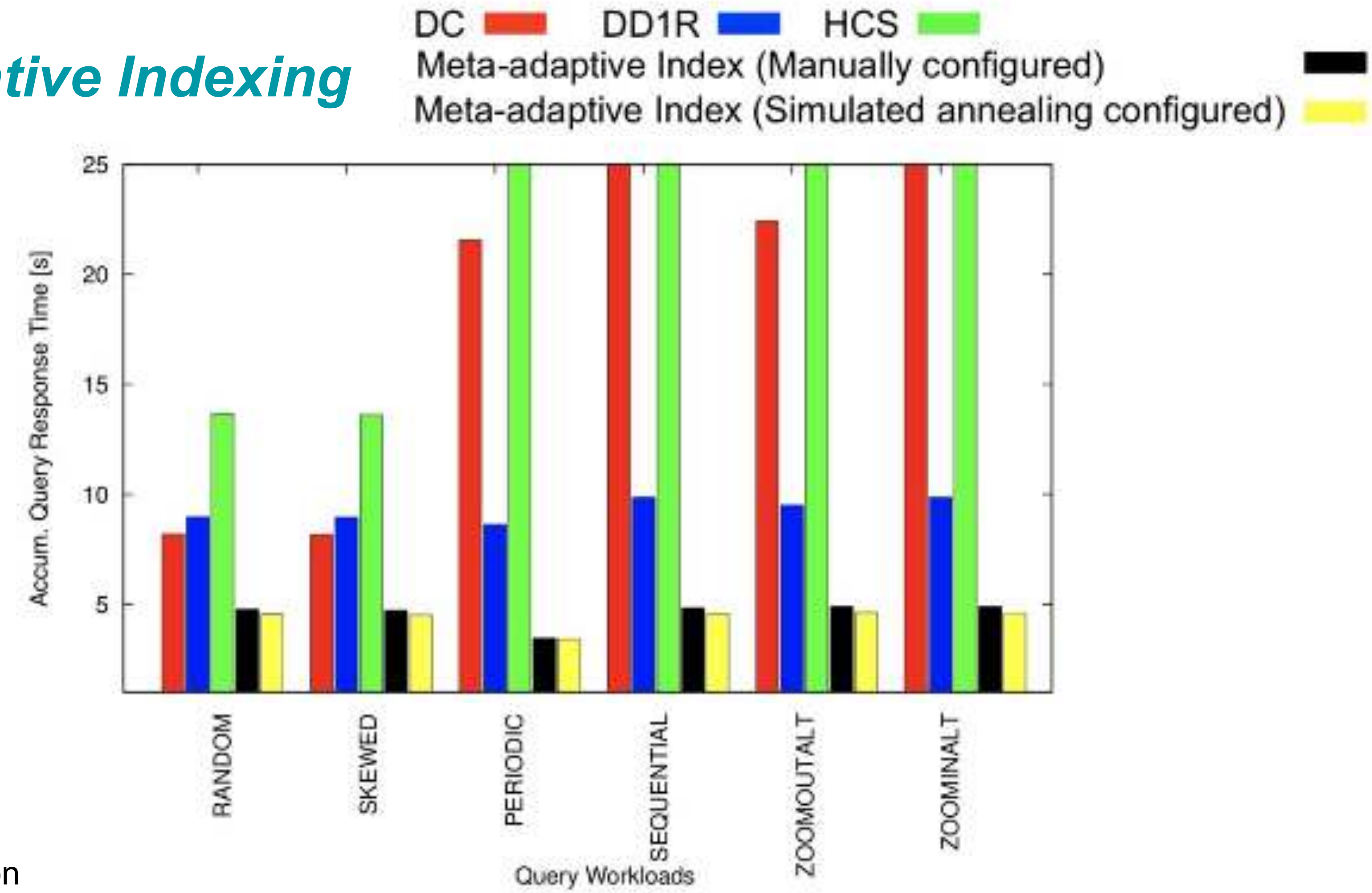


(c) $\mathcal{Z}(min = 0, max = 2^{64} - 1, \alpha = 0.6)$

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

Cumulative Indexing



*Normal distribution

Final Thoughts

- Tackles more than one problem
- Minimal overhead compared to previous work, with better results
- Consistently performs well under varying workloads, in comparison to varying results of other indexes.
- Takes advantage of unique optimizations, such as Simulated Annealing, Software-Managed Buffers, and Non-Temporal Streaming Stores

References

1. 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.

2. 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