Adaptive Adaptive Indexing

Authored by: Felix Martin Schuhknecht, Jens Dittrich , Laurent Linden

Presented by: Samir Farhat, Jianqi Ma, Ning Wang and Lanfeng Liu

What is the problem & why it is important?

Practical: Insufficiently Adaptive Indexing Algorithms

- Many techniques offset their advantages with their concessions
- Adaptive Indexing often encounters several issues
  - Struggles with diverse workloads
  - Balancing individual query vs aggregate query runtime
  - Escalating Variance
  - Slow convergence/sorting
- Single issue tunnel vision
  - How can solutions be combined and morphed to work in harmony?
- Static and no consideration of environment an algorithm runs in
- No effort to handle oddly skewed data in partitioning systems

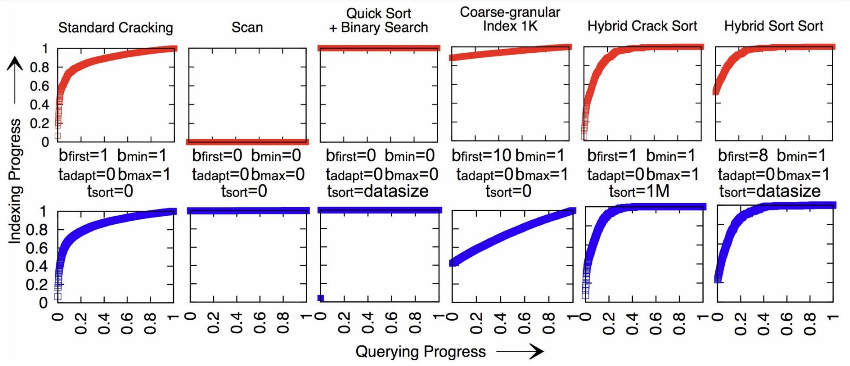


Fig. 10: ***Emulation of adaptive indexes and traditional methods.*** *The top row shows the signatures of the baselines from [1] in red. The bottom row shows the signatures of the corresponding emulations of our meta-adaptive index in blue, alongside with the parameter configurations that were used.*

Conceptual: Identifying Patterns and Generalizing Solutions

- Arguably the most important contribution the work presents is that identifies patterns and differences to bring these together
  - Data partitioning
  - Distributed indexing
- Previous work does not generalize data organization/partitioning
  - Classical solutions
- Previous work doesn't differentiate reorganization timing distributions

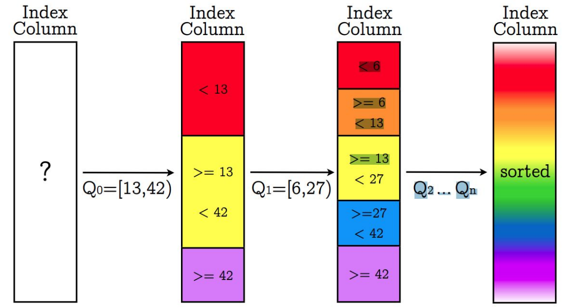


*Fig. 1: **Concept** of database cracking reorganizing for multiple queries and converging towards a sorted state.*

Importance: Fairly Evident

- By tackling the conceptual issues, researchers are able to wield and develop tools to improve adaptive indexing algorithms
  - Accelerating improvement
  - Focusing advancement paths
- By tackling the practical issues...
  - Financial benefits: Reduced cost for organizations by developing more efficient systems that accomplish the same in reduced time
  - Energy benefits: Less computational power to achieve desired results
  - Ecological benefits: Hand-in-hand with the latter two
    - Less “fuel burned”

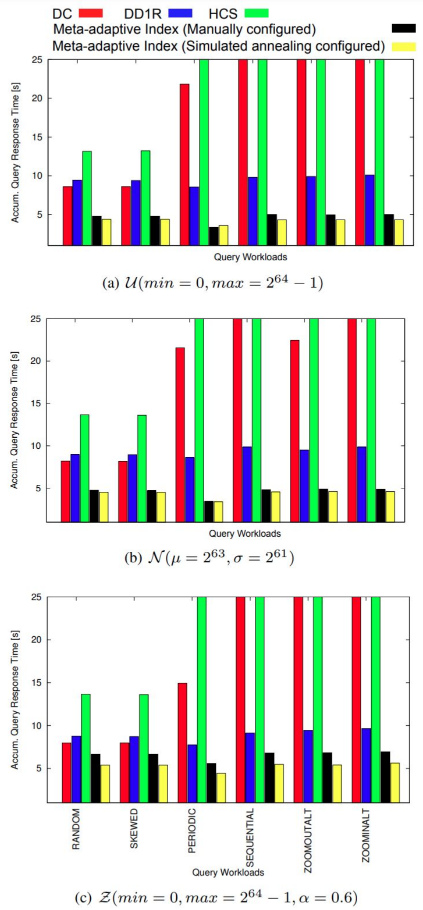


Fig. 12: ***Accumulated query response times*** *of the meta-adaptive index both manually configured (Section VIII-C1) as well automatically configured using simulated annealing (Section VIII-D1) under* ***uniform*** *(12(a)),* ***normal*** *(12(b)), and* ***Zipf-based*** *(12(c)) key distributions and* ***different query***

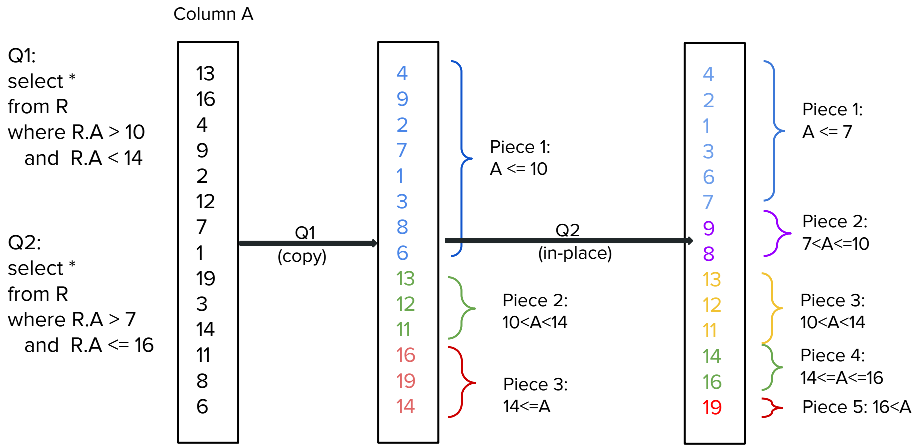
Why is it hard & why older approaches are not enough?

Recap: Common indexing techniques

- Tree
  - B+ Tree.
  - LSM Tree.
  - Radix Tree.
- Hash index
- Bitmap index
- Database cracking
  - Standard cracking
  - Stochastic cracking(DD1R is used in the paper)
  - Hybrid cracking(HCS and HSS are used in the paper)

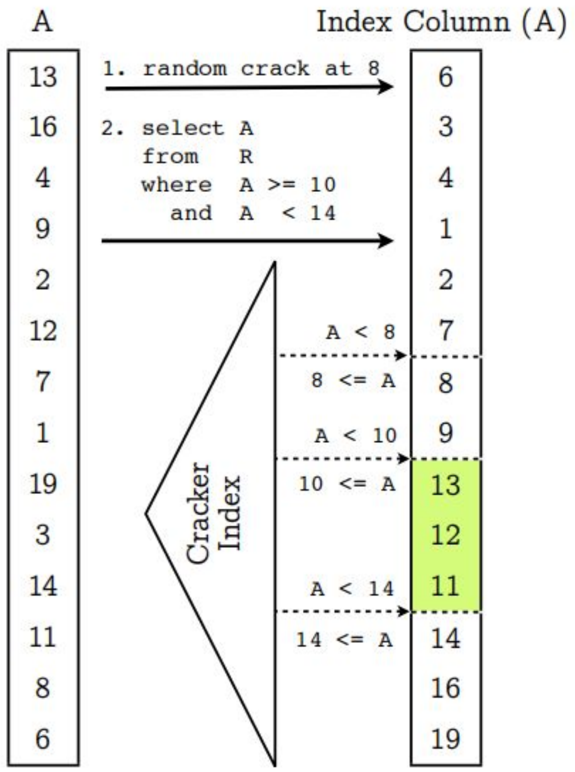
Details: Standard cracking(DC)

- The first time a range query is posed on an attribute A, a cracker database makes a copy of column A.
- The copied column is being continuously split into more and more pieces as queries arrive.



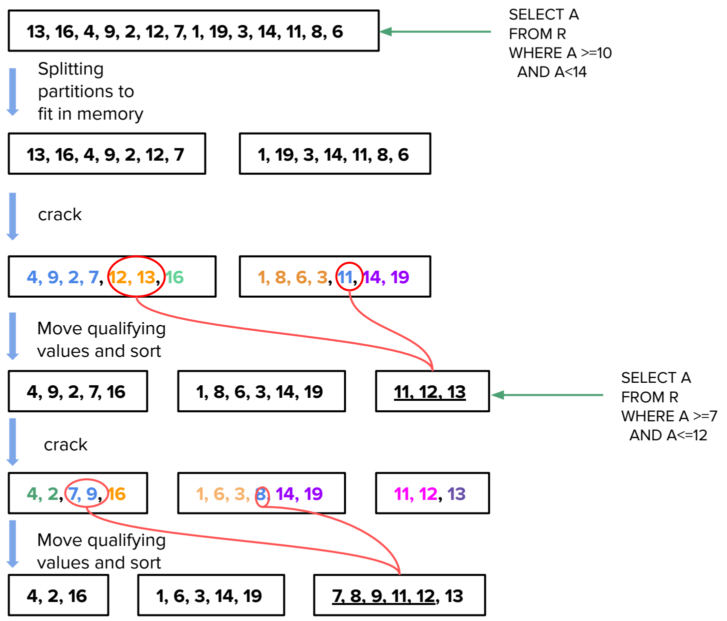
Details: Stochastic cracking(DD1R)

DD1R is an enhanced version of DC: before split the column, it will randomly choose a pivot, and split the column on the pivot in advance.



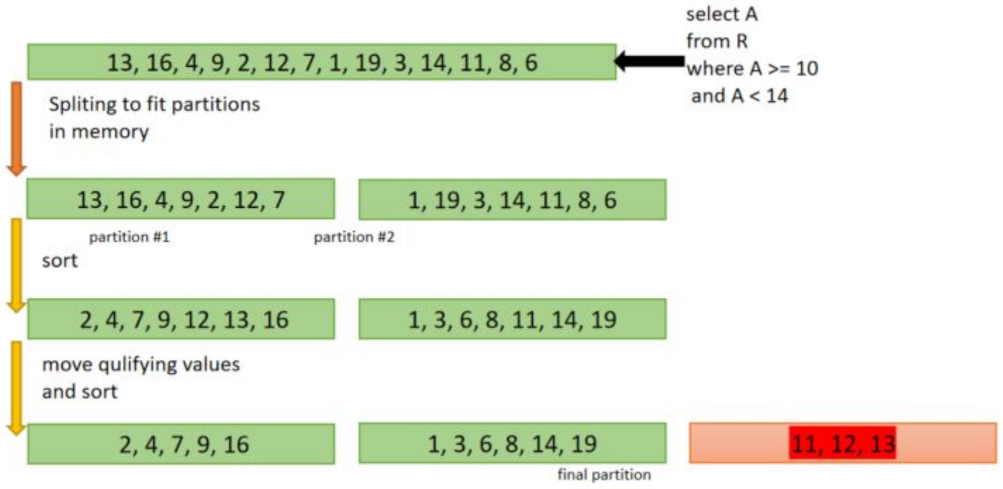
Details: Hybrid Cracking(HCS)

- The first query of each column splits the column's data into initial partitions that each fit in memory (or even in the CPU cache)
- For each query, the process can be divided into 2 phase:
  - Crack each partition according to the given query.
  - Move qualifying column values into the final partition(s) and sort each of them.



Details: Hybrid Cracking(HSS)

The HSS is a variant of the HCS which will sorts the partition instead of cracking the partition during phase 1.



Each of these algorithms mostly focus on reducing a single issue at a time !

What is key idea and why it works?

Components of Adaptive Adaptive Indexing

- Generalize Index Refinement
- Adapt Reorganize Effort
- Defuse Skewed Data

Components of Adaptive Adaptive Indexing

- Generalize Index Refinement
- Adapt Reorganize Effort
- Defuse Skewed Data

Generalize Index Refinement

- There are tons of different implementations of Adaptive Indexing:
  - Standard Database Cracking, Stochastic Cracking, Hybrid Cracking, etc.
- But the heart of every data reorganizing algorithm is **Data Partitioning**
- partition_in_k()
  - Fan-out: **k** (number of partitions you will get)
- Example
  - Standard Database Cracking is just partition_in_k() with k = 2
  - Sorting is just partition_in_k with k=size_of_dataset

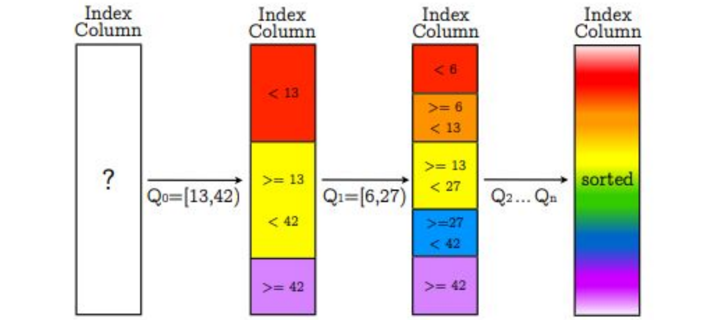
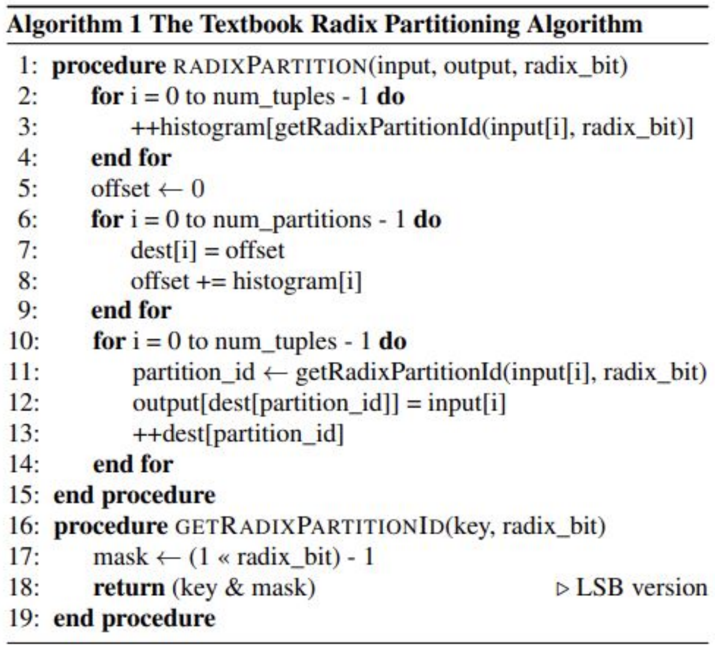


Fig. 1: ***Concept*** *of database cracking reorganizing for multiple queries and converging towards a sorted state.*

Fig. 1 from *Adaptive Adaptive Indexing*

Generalize Index Refinement Cont.

- **Adaptive** Adaptive Indexing
  - This algorithm is able to set the **fan-out k** of partition_in_k() freely
  - The fan-out k would heavily affect our policy to adapt the reorganization effort
- **Meta-Adaptive Indexing** uses partition_in_k() solely for data reorganization
- Implementation of partition_in_k()
  - Radix based partitioning
  - Pros: higher partitioning throughput than traditional comparison based methods
  - Cons: Does not generate partitions with respect to predicates –> requires filter to qualify entries



Algorithm 1 The Textbook Radix Partitioning Algorithm

```
procedure RADIXPARTITION(input, output, radix_bit)
    for i = 0 to num_tuples - 1 do
        ++histogram[getRadixPartitionId(input[i], radix_bit)]
    end for
    offset ← 0
    for i = 0 to num_partitions - 1 do
        dest[i] = offset
        offset += histogram[i]
    end for
    for i = 0 to num_tuples - 1 do
        partition_id ← getRadixPartitionId(input[i], radix_bit)
        output[dest[partition_id]] = input[i]
        ++dest[partition_id]
    end for
end procedure
procedure GETRADIXPARTITIONID(key, radix_bit)
    mask ← (1 « radix_bit) - 1
    return (key & mask)                                ▷ LSB version
end procedure
```

Components of Adaptive Adaptive Indexing

Generalize Index Refinement

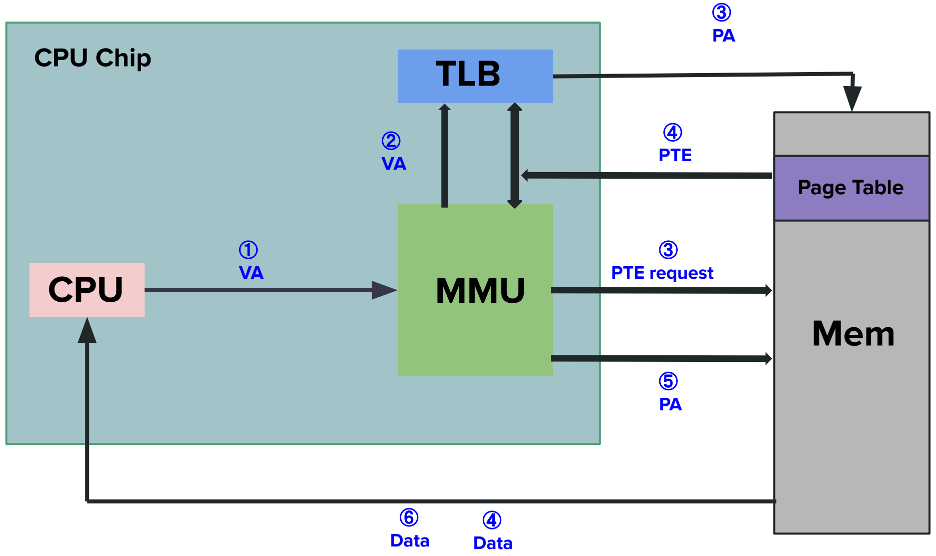
Adapt Reorganize Effort

Defuse Skewed Data

Data Partitioning in the Very First Query

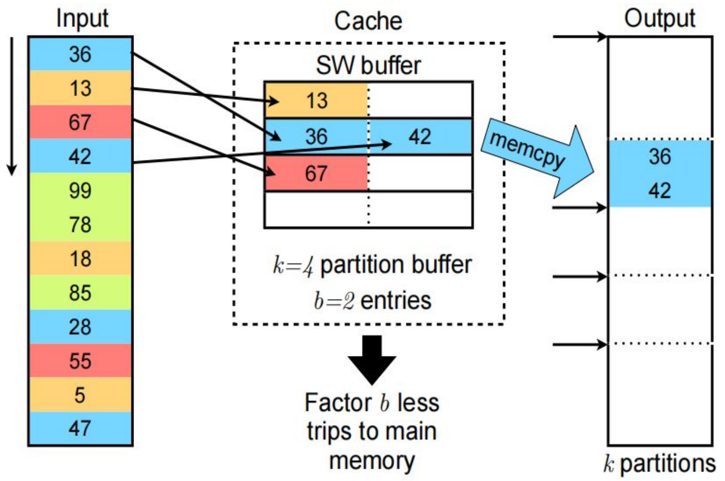
- **Out-of-place Radix Partitioning in two steps:**
  a. Scan the input and build a histogram. Use the histogram to initialize pointers to fill the partitions.
  b. Copy entries into designated partitions.

- Naive copy is costly: TLB misses are triggered if the number of partition is more than 32.



Data Partitioning in the Very First Query

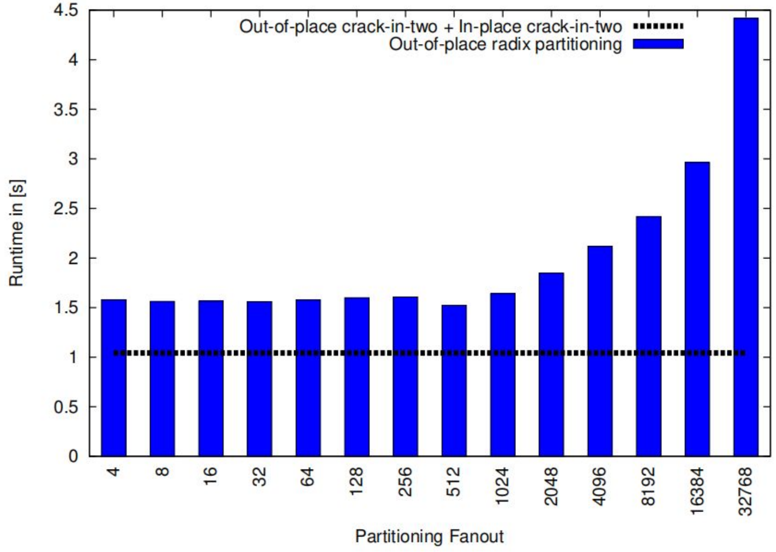
- To address the problem of too many TLB misses, use *software-managed buffers*



Data Partitioning in the Very First Query

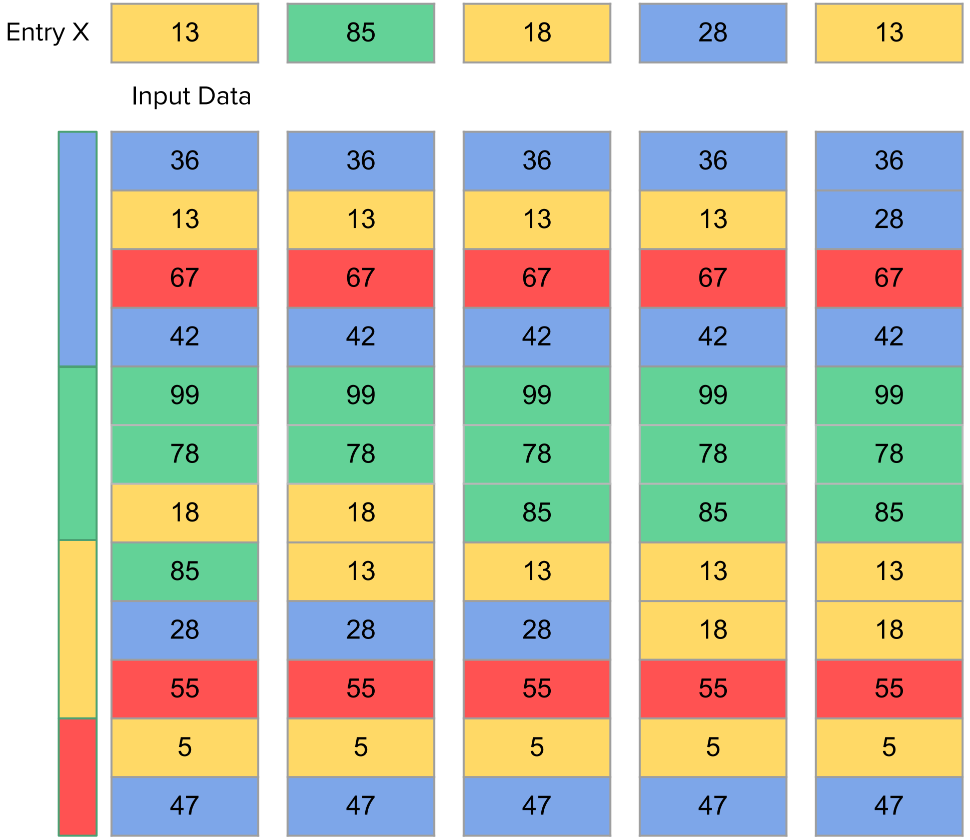
- Create a vast amount of partitions using radix partitioning with only slightly higher costs as two times crack-in-two which creates only three(or two, depend on the query).

- In total, the strategy for the very first query is clear: Create a significantly larger number of partitions than standard cracking (creating only three partitions) with negligible overhead and consequently reduce the average partition size drastically



Data Partitioning in Subsequent Queries

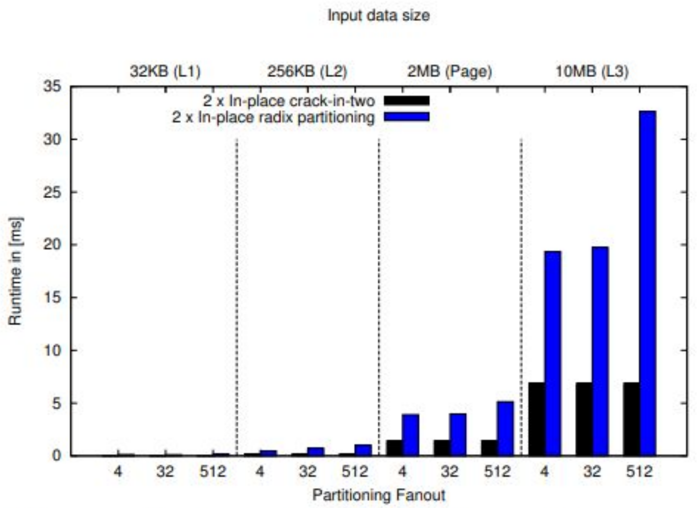
- **In-place Radix Partitioning** since the data is now present in the index column
  - Cuckoo-style: Search and Replace
- Scan partition p0 from the beginning and identify the first entry x that does not belong to partition p0 but actually to another partition p2
- we scan partition p2 until we find the first entry y that does not belong to p2
- Replace entry with x with entry y
- Repeat this process until a cycle is detected



Entry X	🟨 13	🟩 85	🟨 18	🟦 28	🟨 13
🟦	🟦 36	🟦 36	🟦 36	🟦 36	🟦 36
🟦	🟨 13	🟨 13	🟨 13	🟨 13	🟦 28
🟦	🟥 67	🟥 67	🟥 67	🟥 67	🟥 67
🟦	🟦 42	🟦 42	🟦 42	🟦 42	🟦 42
🟩	🟩 99	🟩 99	🟩 99	🟩 99	🟩 99
🟩	🟩 78	🟩 78	🟩 78	🟩 78	🟩 78
🟩	🟨 18	🟨 18	🟩 85	🟩 85	🟩 85
🟨	🟩 85	🟨 13	🟨 13	🟨 13	🟨 13
🟨	🟦 28	🟦 28	🟦 28	🟨 18	🟨 18
🟨	🟥 55	🟥 55	🟥 55	🟥 55	🟥 55
🟥	🟨 5	🟨 5	🟨 5	🟨 5	🟨 5
🟥	🟦 47	🟦 47	🟦 47	🟦 47	🟦 47

Data Partitioning in Subsequent Queries

- **In-place Radix Partitioning** since the data is now present in the index column
  - Cuckoo-style: Search and Replace
- With a decrease in **partition size**, increase the **fan-out k**. At a sufficiently small size, finish the partition by sorting it as the additional cost is negligible
- Where modifications on fan-out **k** happens! But how?



(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.

Adapting the Partitioning Fan-out

- How to set Fan-out **K**?
  - $k = 2^{(\textit{fan-out bits})} = 2^{f(s,q)}$
  - Fan-out bits: $f(s,q)$
  - *s* = size of the partition to reorgnize | *q* = the query sequence number
- What is inside $f(s,q)$
  - $$f(s,q) = \begin{cases} b_{first} & \text{if } q = 0 \\ b_{min} & \text{else if } s > t_{adapt} \\ b_{min} + \left\lceil (b_{max} - b_{min}) \cdot \left(1 - \frac{s}{t_{adapt}}\right) \right\rceil & \text{else if } s > t_{sort} \\ b_{sort} & \text{else.} \end{cases}$$
- b_first, b_min, b_max, b_sort, t_adapt, t_sort are all manually set

Adapting the Partitioning Fan-out Cont.

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

- **first query** <u>different</u> than others
- Increase the granularity of reorganization with a decrease of input partition size
- Finish the input partition by sorting it at a sufficiently small size

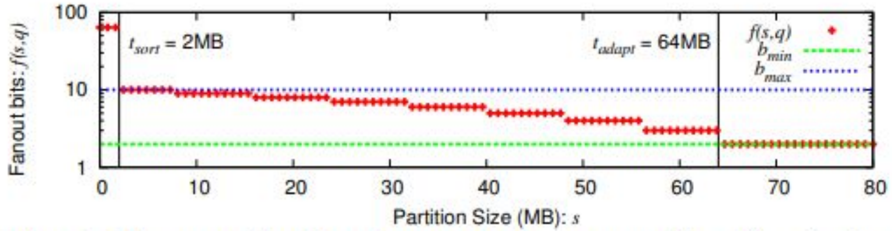


Fig. 5: The ***partitioning fan-out bits*** *returned by $f(s,q)$ for partition sizes s from 0MB to 80MB and $q > 0$ with t_{adapt} = 64MB, $b_{min} = 2$, $b_{max} = 10$, t_{sort} = 2MB, and $b_{sort} = 64$.*

Components of Adaptive Adaptive Indexing

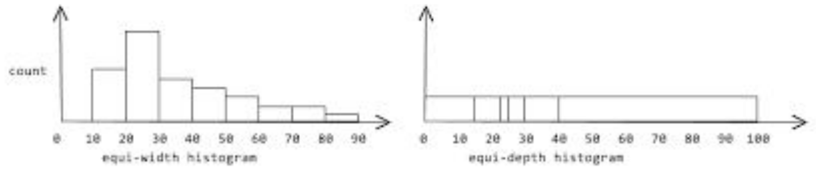
Generalize Index Refinement

Adapt Reorganize Effort

Defuse Skewed Data

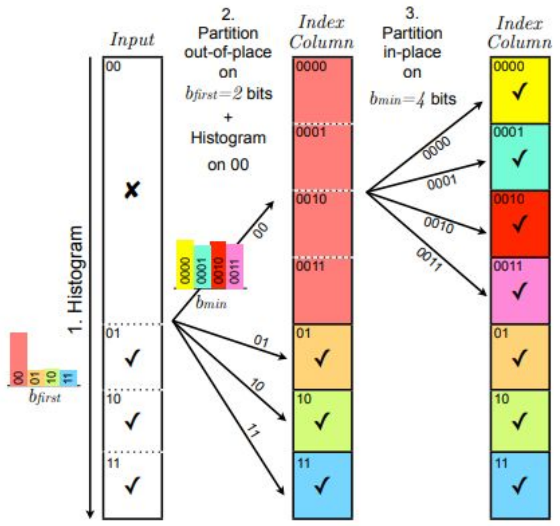
Handling Skewed Distributions

- Radix-based partition performs badly when key is highly skewed
  - Non-uniform partition sizes —> index quality of partitioning steps decreases
- Equi-depth out-of-place partitioning
  - Similar with Equi-depth histogram
- ***skewtol*** determines how many data can be split into one partition



Equi-depth out-of-place partitioning

- Happens During **Data Partitioning in the very First Query**
- Build the histogram with respect to b_first
- Perform out-of-place Radix partitioning and build a new histogram on the skewed partitions with respect to the b_min
- Perform in-place Radix partitioning on the skewed partitions.


Input
1. Histogram
b_{first}
00
01
10
11
2. Partition out-of-place on $b_{first}=2$ bits + Histogram on 00
b_{min}
0000
0001
0010
0011
Index Column
3. Partition in-place on $b_{min}=4$ bits
Index Column

Experiments: How Does this Paper Supports its Claim

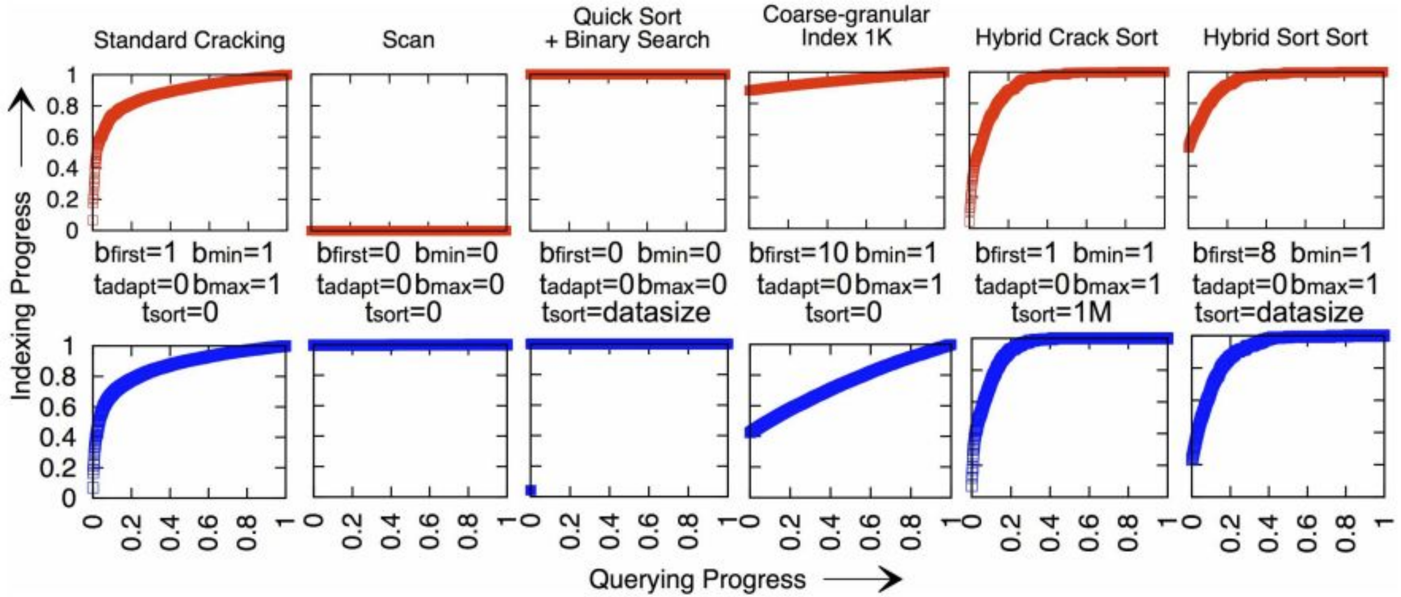
Configuration parameters

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

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.

Emulation of adaptive indexes

- Motivation is to replace existing adaptive indexes by a single method
- Emulation of some adaptive indexes can be achieved by configuring parameters

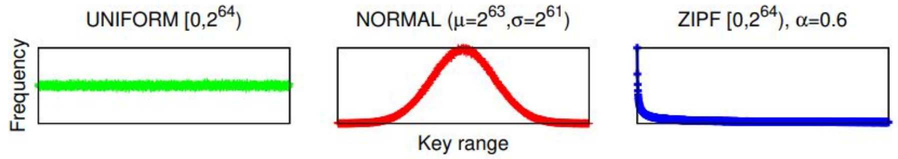


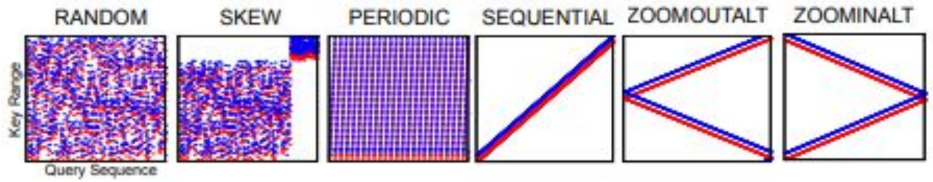
Baselines

- Standard cracking: simplest adaptive indexing
- Stochastic cracking: decouples reorganization from the query predicates to a certain degree and introduce randomness
- Hybrid cracking: hybrid cracking splits the input non-semantically into chunks and applies standard cracking
- Sort + Binary search
- Scan

Experimental Setup

- 100 million entries, 8B key, 8B rowID, 1.5GB index column
- Three characteristic key distributions (key range from 0 to 2^{64} - 1)
  - Uniform distribution
  - Normal distribution (mean = 2^{63}, standard deviation = 2^{61})
  - Zipf distribution (α = 0.6)
- Query workload: 1000 range queries with 1% selectivity and varying workload patterns

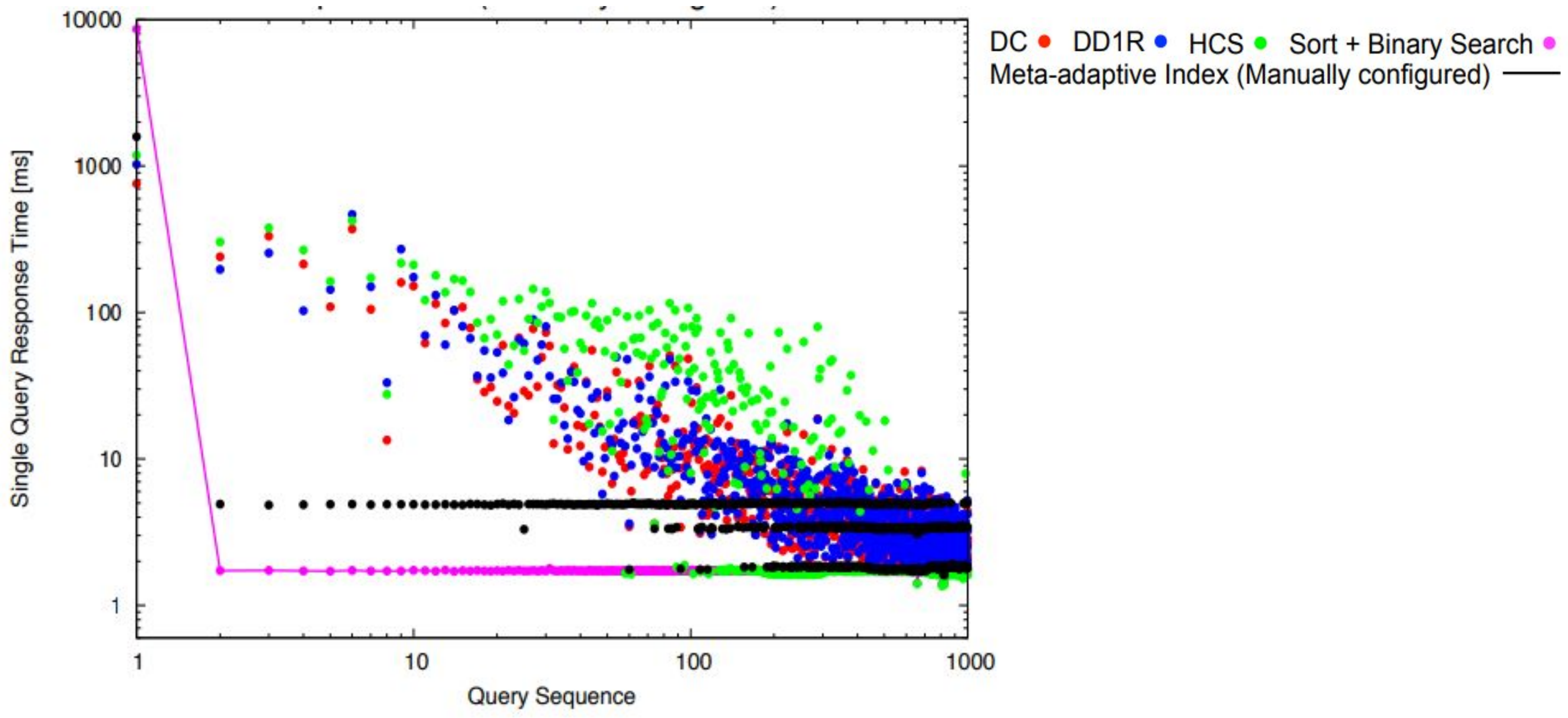




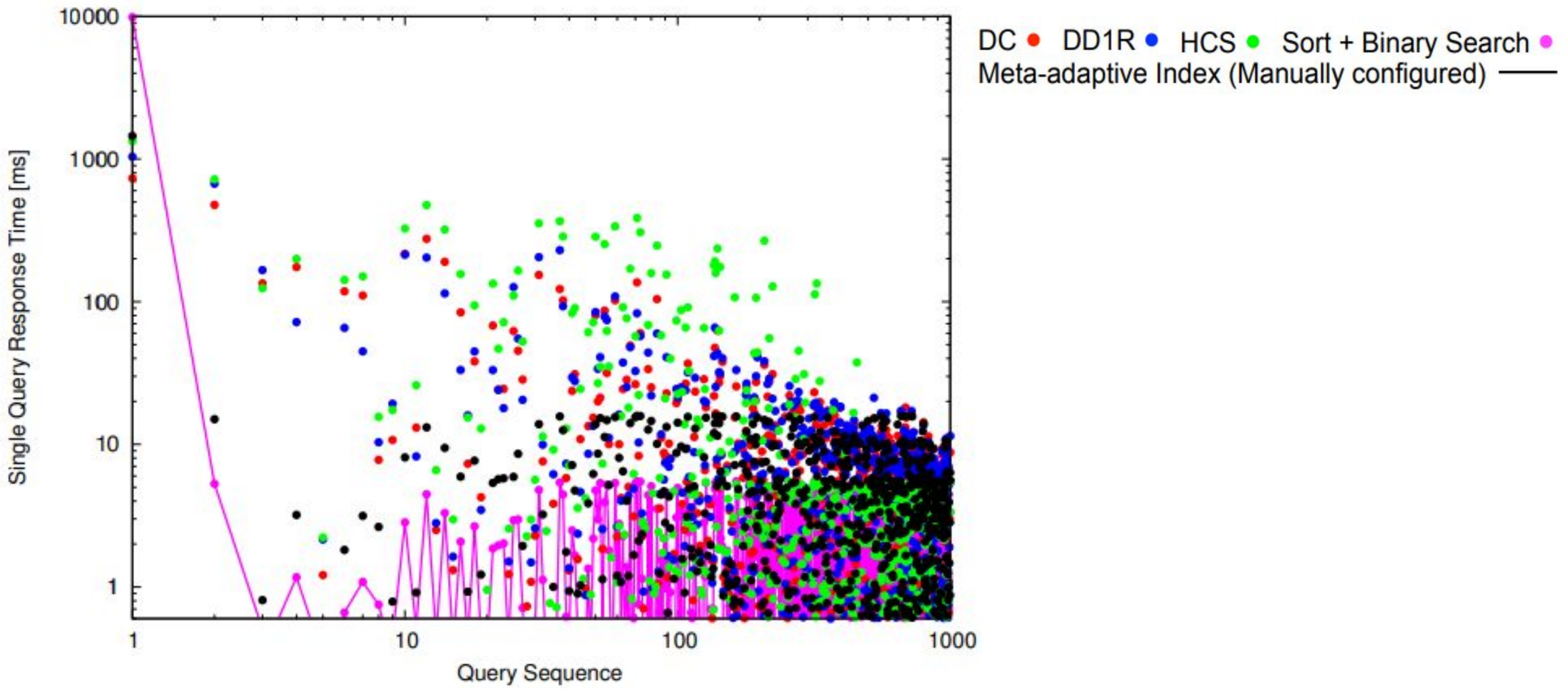
Individual query response time

- Manual configuration
  - b_first = 10, more fan-outs make partitioning significantly more expensive
  - b_min = 3, b_max = 6, to balance convergence speed and pressure on individual query
  - t_adapt = 64MB, the size of the translation lookaside buffer
  - t_sort = 256KB, the size of L2 cache
  - skewtol = 5, moderate skews are tolerated

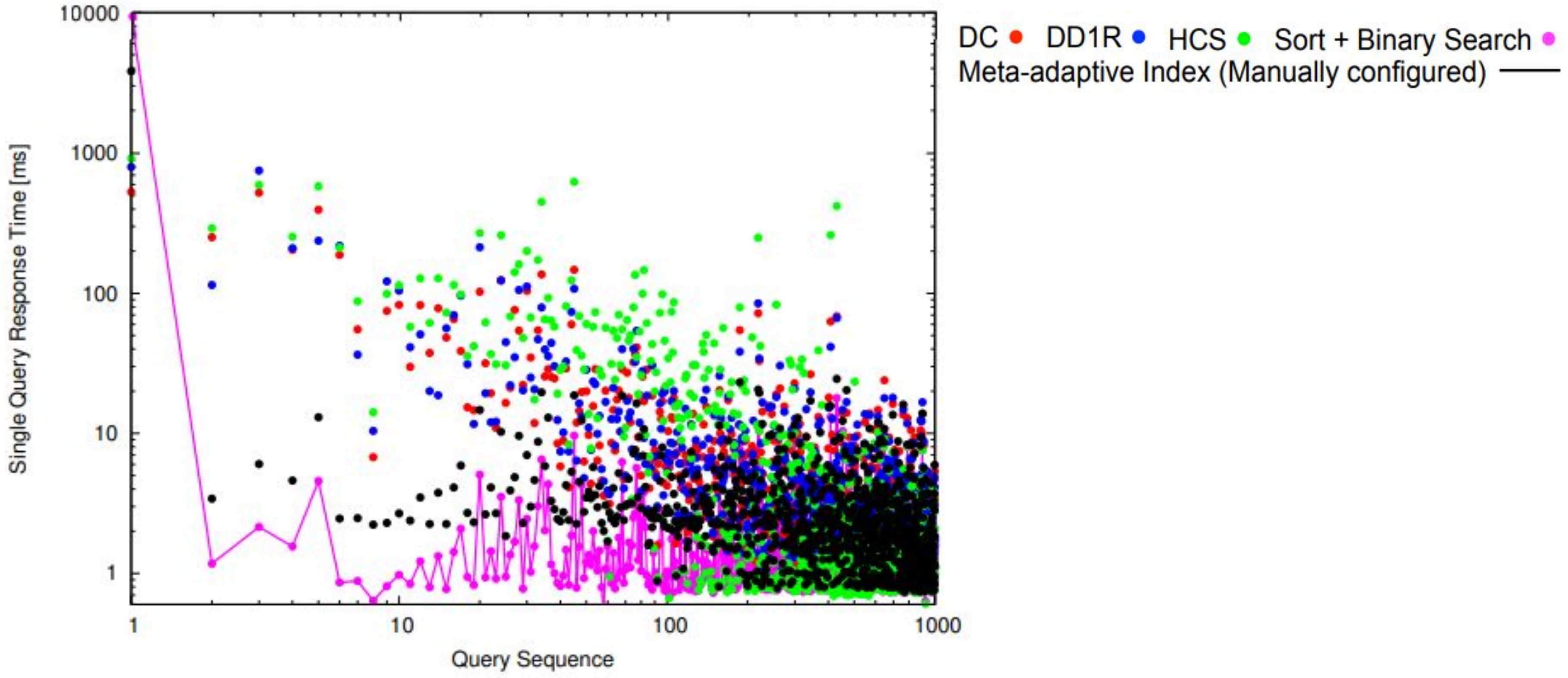
Individual query response time - Uniform distribution



Individual query response time - Normal distribution

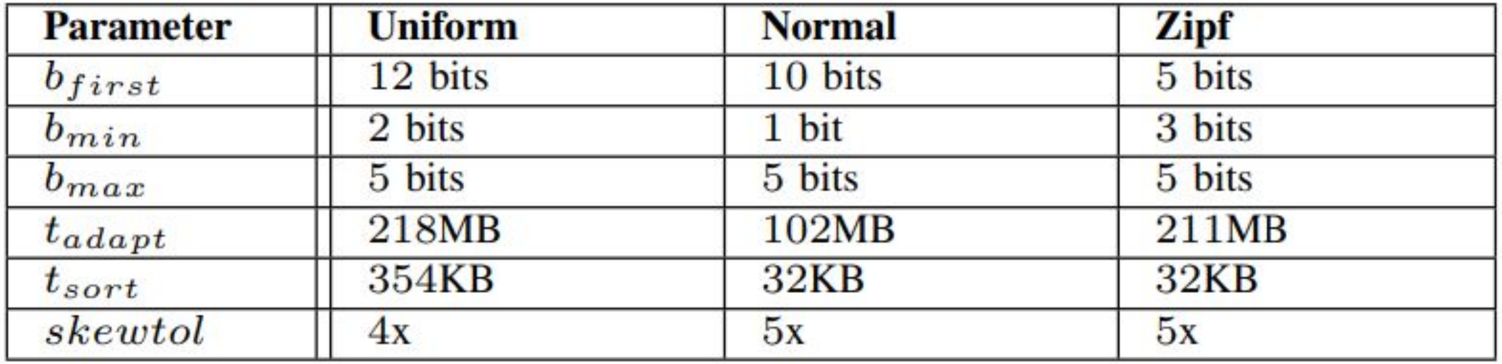


Individual query response time - Zipf distribution



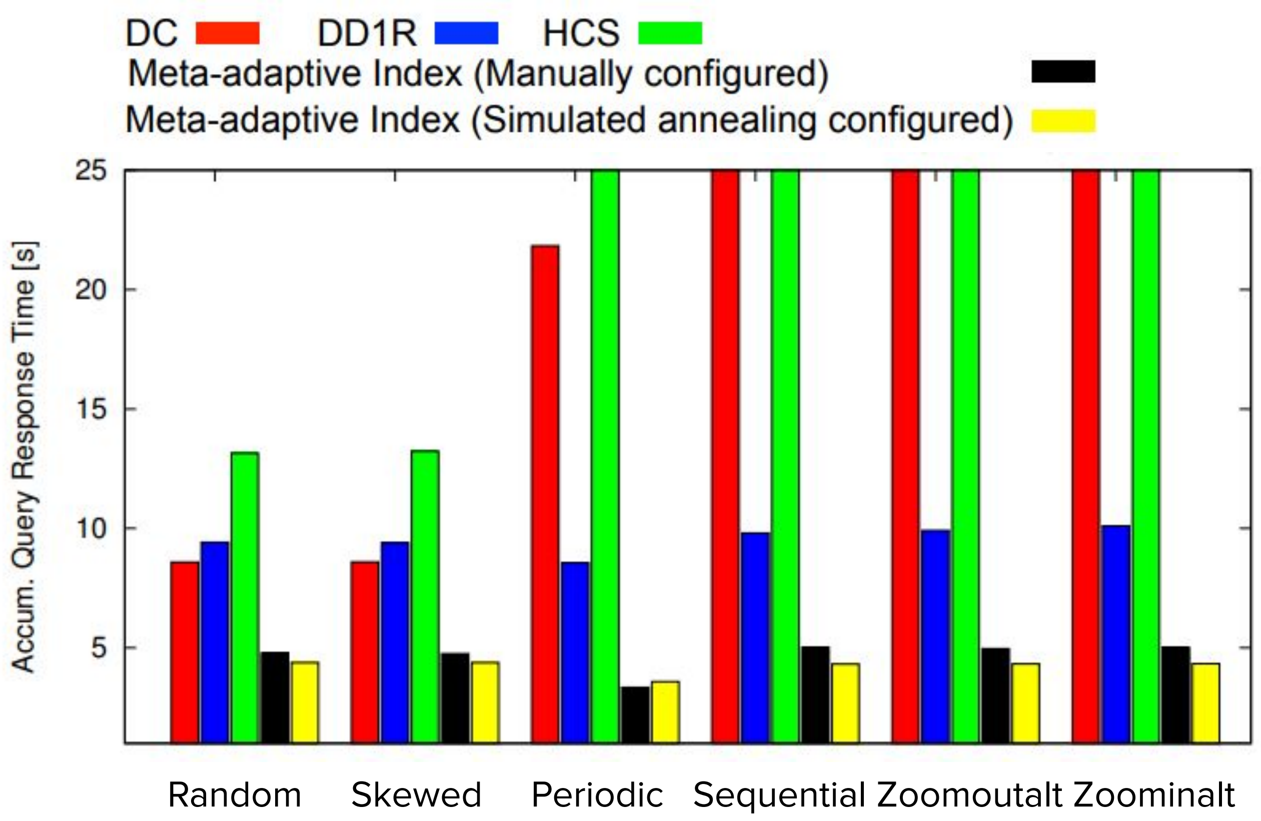
Accumulated Query Response Time

- Manuel configuration same as before
- Simulated annealing configuration

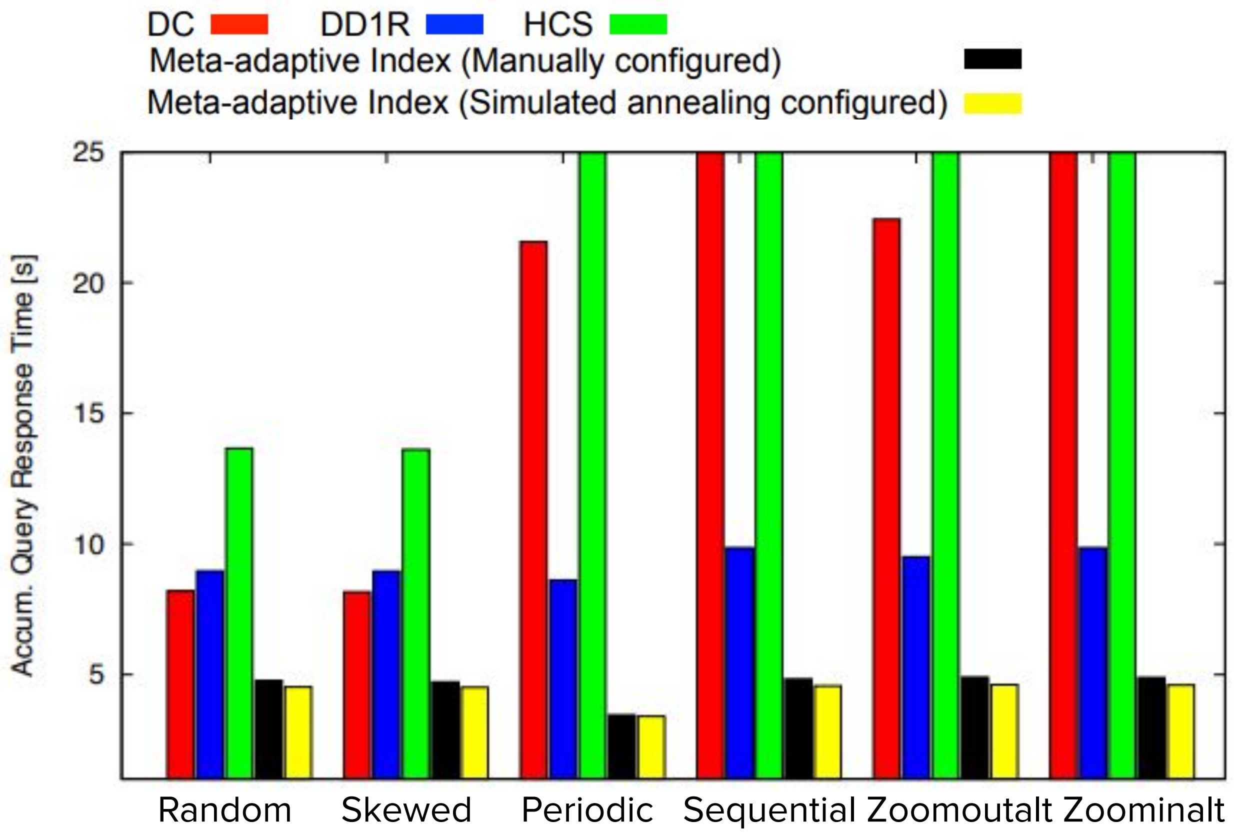


Parameter	Uniform	Normal	Zipf
b_{first}	12 bits	10 bits	5 bits
b_{min}	2 bits	1 bit	3 bits
b_{max}	5 bits	5 bits	5 bits
t_{adapt}	218MB	102MB	211MB
t_{sort}	354KB	32KB	32KB
$skewtol$	4x	5x	5x

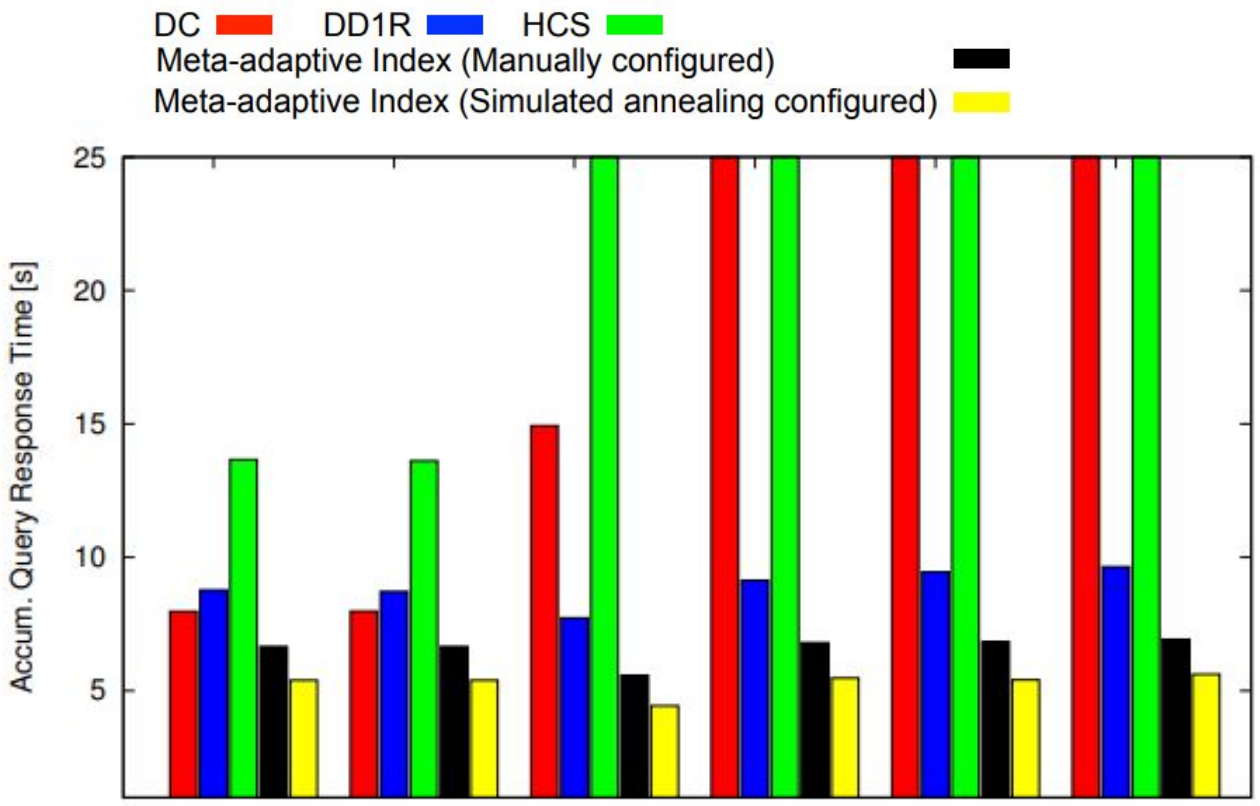
Accumulated Query Response Time - Uniform distribution



Accumulated Query Response Time - Normal distribution



Accumulated Query Response Time - Zipf distribution



Random Skewed Periodic Sequential Zoomoutalt Zoominalt

What is missing and how can we improve this idea?

What’s missing

- The paper does not discuss the update schema and performance under adaptive adaptive indexing
    - Frequent insertions could undermine query performance since it might increase the amount of invested indexing effort
- Individual query response time only examines the random query workload, performance could vary under other types of query workload
    - For example, if query frequently access the same range, meta adaptive indexing won’t perform as well as standard cracking since it’s paying unnecessary indexing effort for every query
- Little discussion on the limitations of manual parameter setting
    - Simulating annealing still requires manual configuration
    - How do we tell the future of workloads or dynamically adjust to single workload changing

Possible next steps of the work presented in the paper?

Natural Progressions and Escalations

- Meshing the generalized solution with different data structures
  - Grid Tables
  - KD trees
  - Hashing structures
- Adaptive indexing with ML
  - CNNs
  - Deep learning methods
  - Workload classification
- Experiment performance on different or adaptive sorting keys
- Dynamic parameter setting on partition counts and

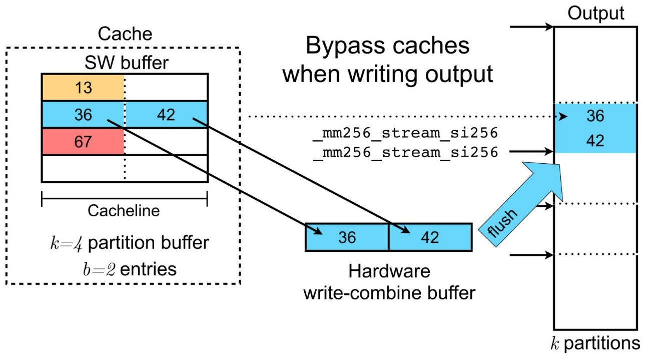


*Fig. 4: Enhancing **software managed buffers** using **non-temporal streaming stores** [12].*

Actual Follow Ups

- “MetisIDX-From Adaptive to Predictive Data Indexing” - focus
  - Machine Learning
  - Key range forecasting and undergo
  - Continuous training by the indexing thread
- “Predictive Indexing”
  - Problem: Retrospectively making computationally expensive physical design changes at once
  - Continuously improves physical design using lightweight physical design changes
- SPST-Index: A Self-Pruning Splay Tree Index for Caching Database Cracking.
- “Progressive indexes: indexing for interactive data analysis”
  - Automatic index creation while providing interactive response times to incoming queries
  - Design allows queries to have a limited budget to spend on index creation
- “Cracking KD-Tree: The First Multidimensional Adaptive Indexing”
  - Generates a KD-Tree based on multidimensional range query predicates
  - Incrementally creates partial multidimensional indexes as a by-product of query processing.
- “GridTables: A One-Size-Fits-Most H2TAP Data Store’

Discussion: Technical Question

The authors of adaptive adaptive indexing show that their technique performs well in multiple generic workloads. What type of workloads would adaptive adaptive indexing not work well for and why? Can you think of examples of reads or updates that would be inefficient in this schema?