Adaptive Adaptive Indexing

Felix Martin Schuhknecht[1], Jens Dittrich[2], Laurent Linden[3]

Saarland Informatics Campus
Saarland University, Germany
[1] felix.schuhknecht@infosys.uni-saarland.de
[2] jens.dittrich@infosys.uni-saarland.de
[3] laurent.linden@gmx.net

***Abstract*—In nature, many species became extinct as they could not adapt quickly enough to their environment. They were simply not fit enough to adapt to more and more challenging circumstances. Similar things happen when algorithms are too static to cope with particular challenges of their "environment", be it the workload, the machine, or the user requirements. In this regard, in this paper we explore the well-researched and fascinating family of adaptive indexing algorithms. Classical adaptive indexes solely adapt the indexedness of the data to the workload. However, we will learn that so far we have overlooked a second higher level of adaptivity, namely the one of the indexing algorithm itself. We will coin this second level of adaptivity *meta-adaptivity*.**

Based on a careful experimental analysis, we will develop an adaptive index, which realizes meta-adaptivity by (1) generalizing the way reorganization is performed, (2) reacting to the evolving indexedness and varying reorganization effort, and (3) defusing skewed distributions in the input data. As we will demonstrate, this allows us to emulate the characteristics of a large set of specialized adaptive indexing algorithms. In an extensive experimental study we will show that our meta-adaptive index is extremely fit in a variety of environments and outperforms a large amount of specialized adaptive indexes under various query access patterns and key distributions.

I. Introduction

An overwhelming amount of adaptive indexing algorithms exists today. In our recent studies [1], [2], we analyzed 8 papers including 18 different techniques on this type of indexing. The reason for the necessity of such a large number of methods is that adaptivity, while offering many nice properties, introduces a surprising amount of unpleasant problems [1], [2] as well. For instance, as the investigation of these works showed, adaptive indexing must deal with high variance, slow convergence speed, weak robustness against different query workloads and data distributions, and the trade-off between individual and accumulated query response time.

In the simplest form of adaptive indexing, called database cracking or standard cracking [3], the index column is repartitioned adaptively with respect to the incoming query predicates. If a range query selecting $[low, high)$ comes in, the partition into which low falls is split into two partitions where one partitions contains all keys less than low and the other partition all keys that are greater than or equal to low. The same reorganization is repeated for the partition into which $high$ falls. After these two steps, the range query can be answered by a simple scan of the qualifying partitions. The information which key ranges each partition holds is stored in a separate index structure called cracker index. The more queries are answered this way, the more fine granular the partitioning becomes. By this, the query response time incrementally converges towards the one of a traditional index. Figure 1 visualizes the concept.

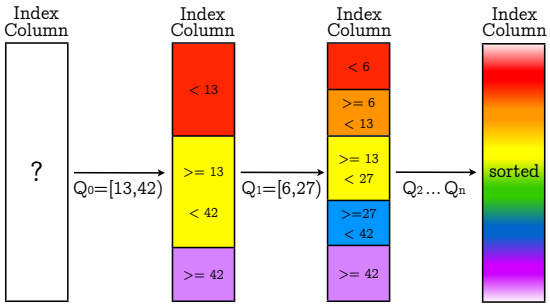


*Fig. 1: **Concept** of database cracking reorganizing for multiple queries and converging towards a sorted state.*

If we inspect the literature [4], [5], [6], [7], [8], [9], [10] proposing variations of the described principle, we see that these algorithms mostly focus on reducing a *single* issue at a time. For instance, hybrid cracking [5] tries to improve the convergence speed towards a full index. Stochastic cracking [4] instead focuses on improving the robustness on sequential query workloads. Thus, to equip a system with adaptive indexing, it actually has to be extended with numerous different implementations that must be switched depending on the needs of the user and the current workload.

This raises the question of how different these algorithms really are. During the study of the literature we made two observations: First, at the heart of every cracking algorithm is simple *data partitioning*, splitting a given key range into a certain number of partitions. Second, the main difference between the algorithms lies in how they *distribute* their indexing effort along the query sequence. Some methods tend to reorganize mostly early on, while others balance the effort as much as possible across the queries. Based on these observations, we will present a generalized adaptive indexing algorithm that *adapts itself* to the characteristics of specialized methods, while outperforming them at the same time.

(1) Generalize the way of index refinement. We identify *data partitioning* as the common form of reorganization in adaptive indexing. Various types of database cracking as well as sorting can be expressed via a function partition-in-k that

produces k disjoint partitions. For instance, we can emulate standard cracking (respectively crack-in-two) using $k = 2$, while sorting on 64-bit keys can be expressed using the fan-out $k = 2^{64}$. Consequently, partition-in-k will be the sole component of reorganization in our algorithm, realized using both in-place and out-of-place versions of highly efficient radix partitioning techniques.

(2) Adapt the reorganization effort by adjusting the partitioning fan-out k with respect to the size of the partition to work on. Classical approaches keep their reorganization effort static during their lifetime. For instance, crack-in-two splits its input always into two parts, independent of the partition size and the state of the index. However, the reorganization effort should be carefully adapted to the input to refine the index as much as possible in an individual step without deteriorating the query response time. To achieve this, we perform the following strategy: with a decrease in size of the input partition that has to be refined, we increase the fan-out k of partition-in-k. Thus, we exploit the decrease in reorganization effort and reorganize more fine-granular to speed up the convergence while ensuring fast response times. Consequently, if a partition reaches a sufficiently small size, we "finish" it via sorting, also enabling interesting orders on the data.

(3) Identify and defuse skewed key distributions and adjust the reorganization mechanism accordingly to counter them. By default, radix partitioning creates balanced partitions only if the key distribution is uniform. While uniformity is often present, it is careless to rely on it. Thus, we introduce a mechanism that is able to defuse the problems caused by the presence of skew in the very first query already. We achieve two things: First, we are able to detect skew in the input without overhead. Second, in the presence of skew, we recursively split partitions that are way larger than the average to enforce a balanced processing of subsequent queries.

Following these three simple concepts, we are able to emulate a large set of specialized adaptive indexing algorithms. Via seven configuration parameters, our general algorithm can be specialized to focus on properties such as convergence speed, variance reduction, or the resistance towards skew, and thus it can emulate and possibly replace a large number of specialized indexes. We will generate method signatures to visualize the quality of our emulation. Apart from applying manual configurations, we will use simulated annealing to optimize the parameter set towards the minimal accumulated query response time for a given workload. Let us now see how we can realize such a meta-adaptive algorithm.

II. Generalizing Index Refinement

Simple data partitioning is at the core of any adaptive indexing algorithm. The applied fan-out of the partitioning process dictates the characteristics of the method by influencing convergence speed, variance, and distribution of the indexing effort. Thus, 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. Consequently, we will solely use a partition-in-k step to perform the reorganization.

Apart from the used partitioning fan-out, the actual implementation of partition-in-k plays an important role. Classical approaches mostly rely on comparison-based methods, as they partition the keys with respect to the incoming query predicates. We decided to use a radix based partitioning algorithm as this type of reorganization method offers a higher partitioning throughput than comparison-based methods [11]. Of course, in contrast to comparison based methods, radix based partitioning does not generate partitions with respect to the given predicates, and thus, filtering the generated partitions for qualifying entries is required. Still, considering the performance advantage, this is a price worth paying.

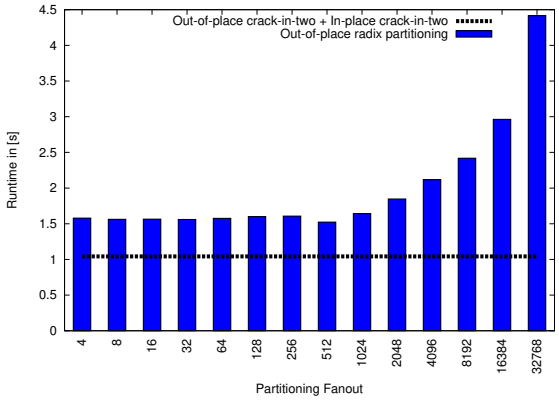
Further, we have to distinguish between the very first query, which can utilize an out-of-place partitioning algorithm, and subsequent queries, where the index column is reorganized solely in-place. In the former case, we can use a highly optimized out-of-place radix partitioning, that has shown its superior performance already in our study [12]. It enhances the partitioning process using software-managed buffers, non-temporal streaming stores, and an optimized micro-layout. In the latter case, we use an in-place radix partitioning algorithm, that swaps elements between partitions in a cuckoo-style fashion [13], without the need of additional memory. Both algorithms together build the core of reorganization in our meta-adaptive index and will be presented in detail in the next section.

III. Adapting Reorganization Effort

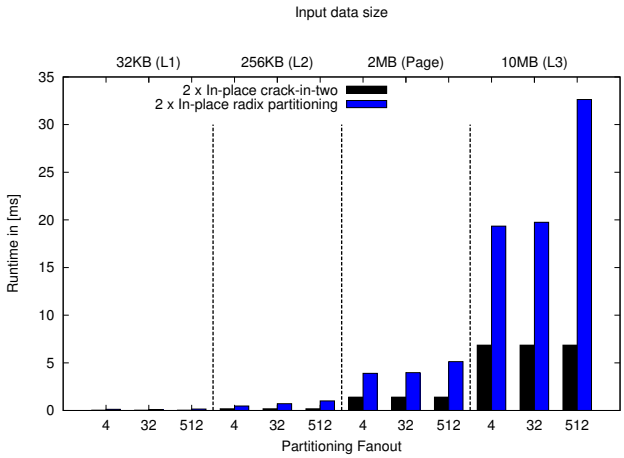
With a look at the previous section, it remains the question of how to steer the amount of reorganization. When should we invest how much into partitioning? To approach this question, we will run a set of experiments to investigate the impact of varying fan-outs on the partitioning process in different situations. We have to distinguish between the very first query, which can exploit out-of-place partitioning, and the remaining ones, which reorganize in-place. Further, we have to distinguish between different input partition sizes, as they highly influence the required cost of reorganization. Let us start by looking solely on the first query.

A. Data Partitioning in the Very First Query

For the very first query, we analyze the runtime of the out-of-place partitioning of 100 million entries of 8B key and 8B rowID. The used machine is a mid-range server that we also use in the experimental evaluation later on (see Section VIII-A for a detailed description). Thus, in total, around 1.5GB of data must be moved. The keys are picked in a uniform and random fashion from the entire unsigned 64-bit integer range. We reorganize for a single range query $[low, high)$, where the low predicate splits the key range into partitions of size $1/3$ and $2/3$ of the data size. The $high$ predicate splits the partition of size $2/3$ subsequently into two equal sized partitions. To reorganize for this query we consider two options: The classical way (as employed by standard cracking) is to partition the data out-of-place into two partitions with respect to low and then to perform in-place crack-in-two on the upper partition with respect to $high$. The created middle



(a) **Reorganization for the very first query**. Standard cracking performs an out-of-place crack-in-two step with respect to predicate low and an in-place crack-in-two step with respect to $high$. In comparison, we show out-of-place radix partitioning as presented in [12] under a varying fan-out of 4 to 32,768.



(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.

Fig. 2: ***Comparison of reorganization options*** *for a range query selecting $[low, high)$. We have to distinguish between the very first query (Figure 2(a)), where the keys are copied from the base table into the index column, and subsequent queries (Figure 2(b)), that reorganize in-place. We test the strategy applied by standard cracking and compare it with radix partitioning.*

partition answers the query. As an alternative, since we have to copy the entire column anyway, we can ignore the query predicates and instead directly partition the data out-of-place using our highly optimized radix based method [12] with a custom fan-out. Although this form of reorganization requires additional filtering to answer the query, it is a valid alternative as the partitions to filter are small for reasonable fan-outs.

1) Out-of-place Radix Partitioning: Let us have a look at how our out-of-place radix partitioning algorithm [12] precisely works. As input, the algorithm gets the source column as well as the requested number of partitions. As output, it produces the partitioned data in a (freshly allocated) destination column. The algorithm works in two passes: in the first pass, we scan the input and count how many entries will go into each partition. Based on this histogram, we initialize pointers to fill the partitions. In the second pass, we perform the actually partitioning by copying the entries into the designated partitions. Unfortunately, naively copying the entries from the base table into the partitions in the second pass can become quite costly for partitioning fan-outs larger than 32 [12]. As we write into the destination partitions in a random fashion, TLB misses are triggered if we partition into more than 32 partitions (since the CPU can cache only 32 address translations for huge pages). To overcome this problem, we employ a technique called *software-managed buffers*. Figure 3 visualizes the concept at an example that partitions into $k = 4$ partitions. Instead of writing entry `36` directly to the second partition, we first write it into the second *buffer*. The buffer for each partition has a size of $b = 2$ entries. Only if a buffer becomes full, i.e. after `42` has been written to it, we flush it in one go to the respective partition. As the buffers are likely to fit into the CPU caches, we effectively reduce the number of trips to main memory and thus the number of TLB misses by a factor of b. Although this technique doubles the amount of copied data, the reduction of TLB misses significantly reduces the runtime over the naive approach [12].

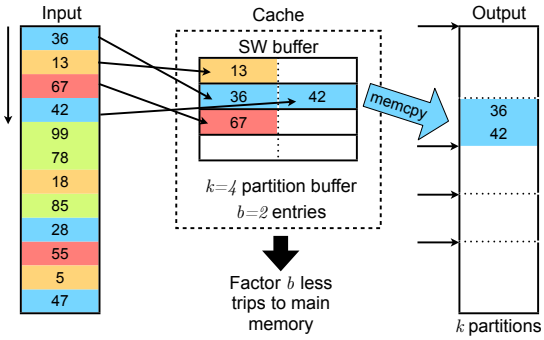


Fig. 3: ***Out-of-place partitioning*** *using* ***software managed buffers*** *[12].*

Additionally, we apply so called *non-temporal streaming stores*. These SIMD intrinsics allow to bypass the CPU caches when flushing the software-managed buffers to the destination partitions. Figure 4 shows the concept. To flush a single buffer of one cache-line, two calls to the AVX intrinsic `_mm256_stream_si256` are necessary, where each call writes half a cache-line. Internally, these two calls actually trigger the writing of the cache-line in one go as the CPU performs hardware write-combining.

Using these optimizations, we are able to significantly reduce the pressure on caches and TLB during partitioning. Let us now see how this optimized out-of-place radix partitioning algorithm performs in comparison with crack-in-two.

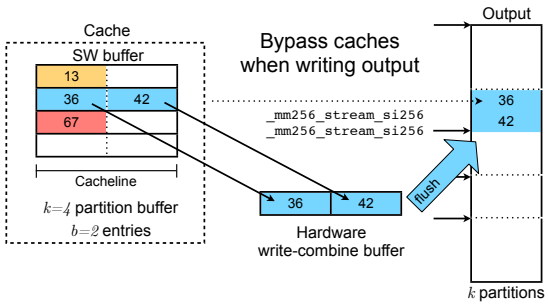


*Fig. 4: Enhancing **software managed buffers** using **non-temporal streaming stores** [12].*

2) Evaluation: In Figure 2(a) we test fan-outs from 4 to 32,768 for out-of-place radix partitioning. Interestingly, we are able to create a vast amount of partitions using radix partitioning with only slightly higher costs as two times crack-in-two which creates only three. For instance, creating 512 partitions is only 1.45x slower (respectively half a second slower) than creating three partitions using two times crack-in-two. At the same time, creating 512 partitions builds an index that is 170 times more fine granular than an index with only three partitions. Besides, for 512 generated partitions, the average partition size is around 3MB and thus easily fits into the L3 cache of the CPU when being processed in subsequent queries. In contrast to that, in the case of two times crack-in-two, each partition is still 500MB large. 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.**

B. Data Partitioning in Subsequent Queries

But how to continue for subsequent queries? First of all, since the data is now present in the index column, we can no longer use an out-of-place partitioning algorithm as in the first query. Instead, any reorganization must happen in-place. To evaluate the options, we again reorganize for a range query $[low, high)$. We consider the case where the low and the $high$ predicate fall into two different partitions, each of size s, where s equals a characteristic system size (size of L1, L2, Hugepage, and L3).

Standard cracking invests the least amount of work to answer the query: two times in-place crack-in-two, reorganizing the two partitions according to low respectively $high$. In comparison, we evaluate again a radix based partitioning, but now in form of the in-place version. We apply in-place radix partitioning with a given fan-out to the two partitions into which the low and $high$ predicates fall. For our test we pick a small, a medium, and a high fan-out with 4, 32, and 512 partitions respectively.

1) In-place Radix Partitioning: Let us see how the in-place of radix partitioning works. As in the out-of-place version, a histogram generation phase is required, where we count how many entries go into each of the k partitions. With this information, we can determine the start of each partition. Now, we scan partition p_0 from the beginning and identify the first entry x that does *not* belong to partition p_0, but actually to another partition, let's say p_5. Then, we scan partition p_5 until we find the first entry y that does not belong to p_5. We replace y by x and continue the procedure of search and replace with entry y. This is done until we close a cycle by filling the hole that x left behind in partition p_0. We perform these cycles of swapping until all partitions contain the right entries.

2) Evaluation: In Figure 2(b), we can see that two times in-place crack-in-two is again the cheapest option. However, we can also observe that with a decrease in input size the absolute difference between the two tested methods decreases. While for 10MB creating 512 partitions using radix partitioning is still around 10ms more expensive than reorganizing into two partitions using crack-in-two, for 2MB it is only around 1.5ms more expensive. In other words, the smaller the input the more negligible the overhead of partitioning with higher fan-outs over cracking becomes. This gives us a strong hint on how we should adapt the partitioning fan-out k during the query sequence: **With a decrease in partition size, increase the fan-out k. At a sufficiently small size, finish the partition by sorting it as the cost is negligible.**

C. Adapting the Partitioning Fan-out

The conducted experiments of Section III-A and Section III-B indicate that the initial reorganization step can create a large number of partitions without deteriorating the runtime in comparison to lightweight methods. The remaining reorganization steps should adapt their effort with respect to the given partition size. Thus, let us now discuss how exactly we adaptively set the partitioning fan-out in the different situations we encounter. We can summarize our strategy in the following function $f(s, q)$, that receives the size s of the partition to reorganize, as well as the query sequence number q as an argument, and returns the number of bits by which the input should be partitioned. We coin this return value the number of *fan-out bits*, i.e. the actual partitioning fan-out $k = 2^{(\textit{fan-out bits})} = 2^{f(s,q)}$. The function depends on a set of parameters that configure the meta-adaptivity.

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

We realize the following high-level design goals in this function: (1) Treat the first query different than the remaining ones. (2) Increase the granularity of reorganization with a decrease of input partition size. (3) Finish the input partition by sorting it at a sufficiently small size.

Based on our observations of Figure 2(a) and Figure 2(b), we distinguish between the very first query and the remaining ones. If we are in the first query ($q = 0$), the function returns a manually set number of fan-out bits determined by the parameter b_{first}. If we are in a subsequent query, we first compare the partition size s with the threshold t_{adapt}. If $s > t_{adapt}$ we return the minimal amount of fan-out bits b_{min}, as the partition is still considered as too large for the application of higher partitioning fan-outs. If s is smaller

or equals than t_{adapt}, but still larger than the threshold for finishing the partition t_{sort}, we adaptively set the fan-out bits between b_{min} and b_{max}. The smaller the partition, the higher is the returned number of fan-out bits. If s is smaller or equals than t_{sort} the function finishes the partition by returning the maximal number of fan-out bits b_{sort} (e.g. 64 for 64-bit keys), which leads to a sorting of the partition. In total, the function $f(s, q)$ allows us to realize the strategies for adaptive partitioning which we discussed in the previous sections. Figure 5 visualizes the generated number of fan-out bits for a sample configuration. As we can see, the function smoothly adapts the number of generated fan-out bits to the size s of the input partition. Besides, we limit the reorganization using partition-in-k to partitions into which the current query predicates fall. In this regard, the reorganization is still focused on partitions of interest.

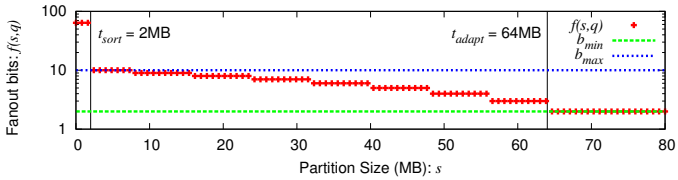


Fig. 5: The ***partitioning fan-out bits*** *returned by $f(s, q)$ for partition sizes s from $0MB$ to $80MB$ and $q > 0$ with $t_{adapt} = 64MB$, $b_{min} = 2$, $b_{max} = 10$, $t_{sort} = 2MB$, and $b_{sort} = 64$.*

IV. Handling Skew

As mentioned in Section II, we prefer a radix-based partitioning over a comparison based partitioning due to the runtime advantages offered by the former one. However, while radix-based partitioning offers a very fast way of assigning entries to their partitions, a valid argument against its use is that it performs badly when confronted with highly skewed key distributions. Skewed key distributions lead to the generation of non-uniform partition sizes, which can drastically limit the gain in index quality of a partitioning step. Extreme cases such as the Zipf distribution, where the most frequent key occurs around twice as often as the second most frequent key and so on, require the generation of so called *equi-depth histograms* to balance the partitions.

In our meta-adaptive index we address the problem of highly skewed data by introducing our own best effort *equi-depth out-of-place radix partitioning* algorithm, that is applied for the very first query. Traditionally, equi-depth partitioning only has to deal with computing equal sized partitions [14]. However, our solution also has to deal with the problem that further radix partitioning steps should still remain possible on said partitions. In other words, the boundaries of the generated partitions must split at radix bits. Therefore we cannot simply adapt a solution where we split and merge partitions on arbitrary keys [15] such that their sizes equalize. Instead we have to chose the partition keys according to the radix bits.

Our solution works as presented in Figure 6: First, we assume that the keys in the input column are uniform. Therefore, we build the initial histogram in phase 1 of the out-of-place partition-in-k algorithm as usual, using b_{first} many bits. Subsequently, we iterate over the newly build histogram and compare the size of each bucket against the theoretical optimum $(columnsize/k) * skewtol$. Here, $skewtol$ denotes the skew tolerance which gives the user control over the skew detection. Once a partition exceeds this threshold it is marked as skewed by the algorithm. In phase 2, we perform the out-of-place radix partition-in-k as normal with respect to the histogram built in phase 1. However, while we are copying tuples into their corresponding partitions, we simultaneously build new histograms on the partitions marked as skewed using b_{min} many bits. Thus, we piggy-back the histogram generation of the next partitioning phase onto the current out-of-place partitioning step. Once we have completed the out-of-place partitioning step, we have already generated this initial partitioning as well as the new histograms. Therefore, in phase 3, we iterate over all skewed partitions and further partition them in-place with respect to b_{min} many bits using the already build histograms of phase 2. Finally, we insert all the partitioning information into the index.

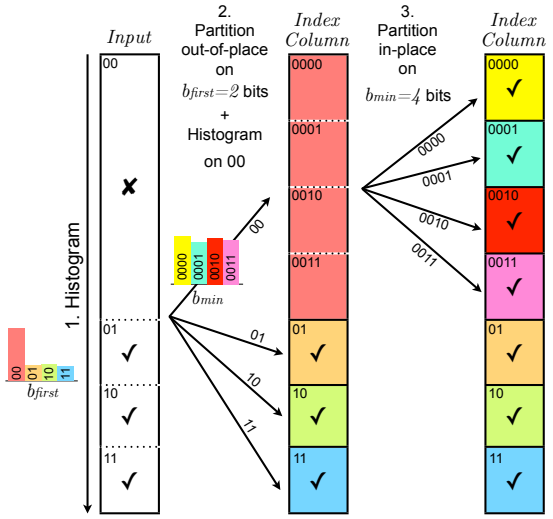


Fig. 6: ***Defusing of input skew****. In phase 1, we build a histogram on the input with respect to b_{first} many bits and locate the skewed partitions. In phase 2, we partition the input out-of-place into the index column based on the histogram of phase 1 while building new histograms only on the skewed partitions with respect to b_{min} many bits. In the final phase 3, we partition the skewed partitions in-place inside the index column based on the histograms of phase 2.*

Using such an approach has mainly two benefits: First, if the keys in the input column are *not skewed*, then the performance of the equi-depth radix partitioning basically equals the one of the standard radix partitioning algorithm, as the piggy-backed histogram creation comes almost for free. Second, if the input column is *heavily skewed* in a certain region, then the "resolution" of the radix partitioning is further increased in that region. Of course, this approach does not guarantee a perfectly uniform partitioning in any case. However, as we will see in the experiments, it offers a practical and lightweight method to defuse severe negative impact caused by skewed distributions.

V. Configuration Knobs

We have seen in the previous sections how the reorganizing procedure can be generalized (Section II), how the fan-out of partitioning is adapted (Section III) and how input skew is handled (Section IV). Along with that, we introduced a set of parameters that allow us to tweak the configuration of the algorithm towards the priorities of the user, the capabilities of the system, and the characteristics of existing adaptive indexes. Table I lists them again alongside with their meaning.

*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.

Of course, all these parameters can be set manually by the user according to the individual preferences. In Section VIII-B, we will setup the parameters to emulate characteristics of individual adaptive indexes. This shows that our meta-adaptive index is general enough to emulate existing adaptive indexes and thus is able to replace them. Further, we will demonstrate how to calibrate the parameters manually (see Section VIII-C1) and automatically using simulated annealing (see Section VIII-D1) to acquire a setup that aims at minimizing the accumulated query response time.

VI. Meta-Adaptive Index

As we have discussed the core topics of meta-adaptivity, we are now able to assemble all components in one single method — our *meta-adaptive index*. The primary goal is to include all discussed aspects *while* keeping the algorithm as simple and lightweight as possible. Algorithm 1 presents the pseudo-code of the it, which represents the logic by which we decide how to reorganize the index under incoming queries. Similar to the adaptive indexing algorithms known in the literature, our meta-adaptive index treats each query independently. As before, we denote the two predicates of a range-query as low and $high$ and use the terms $p[low]$ and $p[high]$ for the partitions, into which the respective predicates currently fall. Each query is now processed according to the same procedure, except of the initial one. For the very first query, the input has to be copied from the base table into a separate index column. Therefore, the algorithm employs out-of-place partition-in-k using $k = 2^{f(s,0)} = 2^{b_{first}}$ (see Section III for details) in order to copy over the data while also piggybacking partitioning work in the mean time (line 7). The created partition boundaries are inserted into the index. During the out-of-place partition-in-k, the aforementioned skew detection is performed as well (see Section IV for details). In case of skew in the distribution, an in-place partition-in-k using $k = 2^{b_{min}}$ is applied on the partitions which are significantly larger than the average partition size.

The output for the first query is then obtained via querying the updated index for the newly created $p[low]$ and $p[high]$

```
1  META_ADAPTIVE_INDEX(table, queries) {
2    // initialize empty index column
3    initializeEmptyIndex()
4    // process first query
5    // out-of-place partition,
6    // handle possible skew, and update index
7    oopPartitionInK(table, f(table.size, 0))
8    // answer query using filtering and scanning
9    // find border partitions
10   p[low] = getPartitionFromIndex(queries[0].low)
11   p[high] = getPartitionFromIndex(queries[0].high)
12   // determine result for lower, mid, upper partitions
13   filterGTE(p[low].begin, p[low].end, queries[0].low)
14   scan(p[low].end, p[high].begin)
15   filterLT(p[high].begin, p[high].end, queries[0].high)
16   // process remaining queries
17   for(all remaining queries q) {
18     // get query predicates
19     low = queries[q].low;
20     high = queries[q].high;
21     // find border partitions
22     p[low] = getPartitionFromIndex(low)
23     p[high] = getPartitionFromIndex(high)
24     // try to refine the largest partition first
25     if(p[low] is not finished) {
26       ipPartitionInK(p[low], f(p[low].size, q))
27       updateIndex()
28     }
29     // try to refine the smaller partition
30     if(p[high] is not finished) {
31       ipPartitionInK(p[high], f(p[high].size, q))
32       updateIndex()
33     }
34     // answer query using filtering and scanning
35     // find refined border partitions
36     p[low] = getPartitionFromIndex(low)
37     p[high] = getPartitionFromIndex(high)
38     // result for lower partition
39     if(p[low] is finished)
40       scan(binSearch(p[low], low), p[low].end)
41     else
42       filterGTE(p[low].begin, p[low].end, low)
43     // middle
44     scan(p[llow].end, p[high].begin)
45     // result for upper partition
46     if(p[high] is finished)
47       scan(p[high].begin, binSearch(p[high], high))
48     else
49       filterLT(p[high].begin, p[high].end, high)
50   }
51 }
```

Algorithm 1: Pseudo-code of the meta-adaptive index. Note that for simplicity, this code does not cover the case where two predicates fall into the same partition. The actual implementation covers this case.

partitions (lines 10 and 11), post-filtering said partitions (lines 13 and 15), and applying a scan to the region in-between (line 14). Please note that for simplicity, we do not discuss the cases where $p[low] = p[high]$. Of course, our actual implementation is aware of this case. Subsequent queries are processed very differently: First, the algorithm again queries the index for $p[low]$ and $p[high]$ (lines 22 and 23) to identify the partitions on which we want to limit the reorganization done by this query. Now, we first check for $p[low]$ whether the partition is already finished respectively sorted or not (line 25). If it is already finished, then no additional indexing effort needs to be spent on that partition. If however, the partition is not yet finished then additional effort needs to be invested. We call the fan-out function $f(s, q)$ with the size s of the partition $p[low]$ and the current query sequence

number q to determine which fan-out to apply for the in-place partition-in-k step and perform the reorganization (line 26). Subsequently, the same process is repeated for the $p[high]$ partition (lines 30 and 31). Finally, we have to obtain the query output. We first re-inspect the index for the updated $p[low]$ and $p[high]$ partitions (lines 36 and 37). Note that in contrast to e.g. standard cracking, the existing partition boundaries do not necessarily split at the given query predicates low and $high$. Thus, we have to filter the boundary partitions in case they are not finished yet. If they are finished, we can apply binary search and scanning on them (line 40 and 47). If they are not yet finished, we apply simple filtering (line 42 and 49). On the partitions in between, we use a simple scan (line 44), as they belong to the query result entirely.

VII. Background and Baselines

Before we put our meta-adaptive index experimentally under test against the state-of-the-art adaptive indexing algorithms that are present out there, let us recap the most prominent literature in the field. The most representative algorithms will serve as baselines for our meta-adaptive index in the following experimental evaluation.

Standard Cracking [3]: Of course, we compare the meta-adaptive index against the most lightweight form of database cracking (**DC**). It offers the cheapest upfront initialization and performs the least amount of reorganization per query to answer it using a scan. It performs very well under uniform random and skewed query distributions but it is prone to sequential workloads. Figure 7(a) visualizes the concept with an example. Let us say a query comes in that selects all entries greater than or equal to 10 and less than 14. In Standard Cracking, two times crack-in-two are applied using the given query predicates. This means, the index column is first partitioned with respect to 10. Then, the upper half containing all entries greater than or equal to 10 are partitioned with respect to 14. The information about the key ranges and the split lines is stored in a separated cracker index. Of course, subsequent queries partition only the areas into which the query predicates fall.

Stochastic Cracking [6]: The class of stochastic cracking algorithms aims at solving the major problem of the standard version, namely sequential query patterns. It is robust against various workloads as it decouples reorganization from the query predicates to a certain degree and introduces randomness. Various different forms of stochastic cracking exist — in this work, we will use **DD1R** as the baseline, which introduces one random crack per query, additionally to the reorganization done with respect to the predicates (see Figure 7(b)).

Hybrid Cracking [5]: The class of hybrid cracking algorithms aims at improving the convergence speed towards the fully sorted state. As with stochastic cracking, there are various different forms of hybrid cracking as well. In this work, we will inspect the most prominent forms called hybrid crack sort (**HCS**) and hybrid sort sort (**HSS**). As shown in Figure 7(c), hybrid cracking splits the input non-semantically into chunks (two in the example) and applies standard cracking for **HCS**, and sorting for **HSS**, on each chunk separately. Then, the qualifying entries of each chunk are merged and sorted in a final partition from which the query is answered.

Additionally, we evaluate the extremes **Sort + Binary Search** (full index, see Figure 7(d)) and **Scan** (no index, see Figure 7(e)). Please note that not all of the following evaluations and comparisons shows all baseline methods. We limit the presented investigation to those methods that are characteristic and that do not overload the visualization.

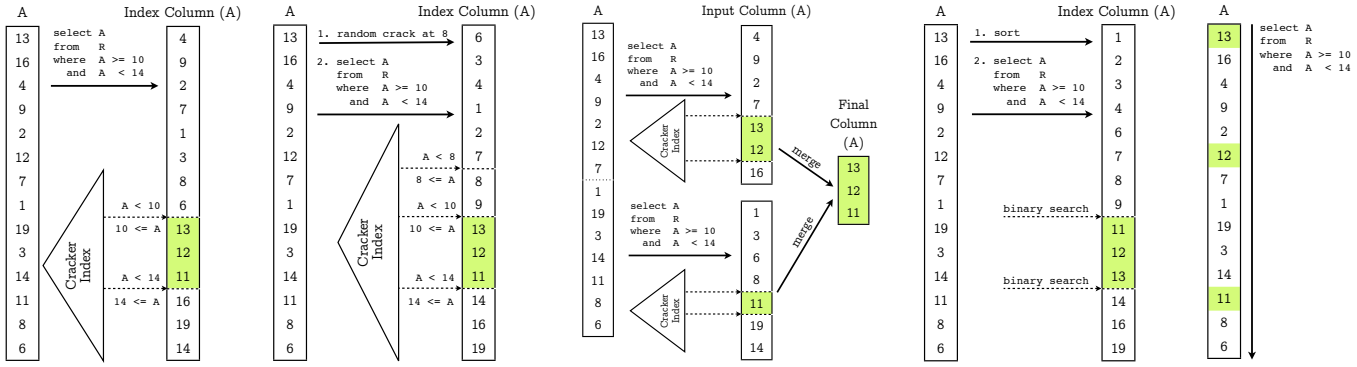
VIII. Experimental Evaluation

With the algorithm at hand, let us now see how our meta-adaptive index competes with the state-of-the-art methods in the field. We basically split the evaluation into two parts: In the first part, we evaluate whether our index can indeed *emulate* and possibly *replace* specialized adaptive indexes. To do so, we configure the meta-adaptive index to fit to the characteristics of other indexes and compare the signatures one by one. This evaluates whether the previously described generalization works and whether our meta-adaptive index is capable of replacing existing adaptive indexes. In the second part, we compare our meta-adaptive index with the baselines in terms of individual and accumulated query response time. We test both a manual configuration as well as configurations calibrated using simulated annealing.

A. Test Setup

The system we use throughout all the experiments consists of two *Intel(R) Xeon(R) CPU E5-2407 @ 2.2 GHz* with 32KB of L1 cache, 256KB of L2 cache, and 10MB of a shared L3 cache. 24GB of DDR3 ram are attached to each of the two NUMA regions. The operating system used in the experiments is a 64-bit *Debian GNU/Linux 8* with *kernel version 3.16*, configured to automatically use Transparent Huge Pages of size 2MB. The TLB can cache 32 virtual to physical address translations for huge pages. The program is compiled using *g++ version 4.8.4* with switches *-msse -msse2 -msee3 -msse4.1 -msse4.2 -mavx -O3 -lrt*. We repeat each experimental run three times and report the average.

The *index column* we use in the following experimental evaluation consists again of 100 million entries, where each entry is composed of a 8B key and a 8B rowID. Therefore the total data size of the index column is about 1.5GB. In total, we use three characteristic key distributions in our tests: First, a uniform distribution generating uniform keys between 0 and $2^{64}-1$. Second, a normal distribution with a mean of 2^{63} and a standard deviation of 2^{61}. And third, a (modified) Zipf distribution with a range of 0 to $2^{64}-1$ and a shape of $\alpha = 0.6$. To generate that distribution we first compute the frequencies for 10000 different values, following a Zipf distribution. Then, we split the unsigned 64-bit key range into 10000 equal sized parts and pick from each range as many keys as given by the previously calculated frequencies in a uniform and random fashion. Figure 8 visualizes the three distributions. The order of individual entries was randomized after workload generation using random shuffling.



(a) Standard Cracking (**DC**) (b) Stochastic Cracking (**DD1R**) (c) Hybrid Cracking (**HCS**). For **HSS**, the inputs are sorted. (d) **Sort + Binary Search**. (e) **Scan**.

Fig. 7: Answering the query `SELECT A FROM R WHERE A >= 10 AND A < 14` *using six different baseline methods.*

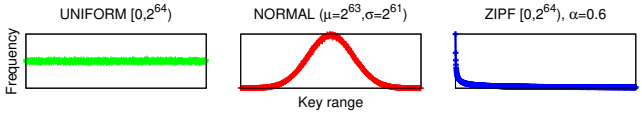


*Fig. 8: Different **key distributions** used in the experiments.*

The query workload we use in the experiments consists of 1000 range queries, each consisting of two 8B keys describing the lower respectively upper bound. To generate the individual queries, we use the workload patterns that have been described in [4] in detail. In Figure 9 we visualize these patterns. We use a fixed selectivity of 1% as common in the literature [1], which has two positive effects on the evaluation: First, such a higher selectivity challenges the convergence capabilities of the algorithms, as both cracks of a range query are located close to each other. Second, for a selectivity of 1% the querying time does not overshadow the cracking time.

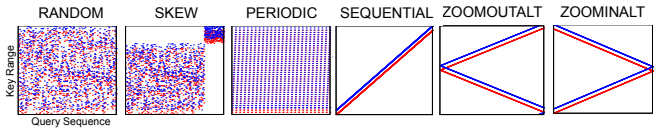


*Fig. 9: Different **query workloads**. Blue dots represent the $high$ keys whereas red dots represent the low keys.*

B. Emulation of Adaptive Indexes

Let us now first see whether the meta-adaptive index is capable of generalizing the principle of adaptive indexing. One main motivation of our algorithm was to replace the vast amount of existing adaptive indexes by a single method that can be configured to emulate different characteristics. In this section, we will evaluate whether this can be achieved and how the meta-adaptive index must be configured to emulate representative existing adaptive indexes. As baselines for the evaluation, we pick the signatures of four characteristic adaptive indexes, as presented in [1]. For a given query of the query sequence (x-axis) the plot of Figure 10 shows the amount of invested indexing effort (y-axis) that has been performed up to this query. We show the amount of indexing effort relative to the total indexing effort (indexing progress) and the queries relative to the total query sequence (querying progress). By this, we see for instance clearly that coarse-granular index, which pre-partitions the index in the first query with 1000 random cracks before applying standard cracking, performs 90% of its indexing progress already in the very first query, while standard cracking needs half of its querying progress to invest that much.

Additionally to the adaptive indexes, we look at the signatures of *Scan* and of *Quick Sort + Binary Search* as representatives of the extreme cases using no index at all or a fully evolved index. All baseline signatures in the top row of Figure 10 originate from the work of [1], where we generated them using uniformly distributed keys and queries, where each query selects 1% of the data. In the bottom row of Figure 10, we show the corresponding signatures of our meta-adaptive index. For each baseline method we configure the meta-adaptive index in a way to emulate its characteristics as much as possible. Our technique is configured entirely via the configuration parameters, as discussed in Section V and works on uniformly distributed keys and random range queries (see UNIFORM respectively RANDOM in Section VIII-A).

We start with **standard cracking** as the classic representative of adaptive indexing. To emulate its behavior, we fix all fan-out bits to $b_{first} = b_{min} = b_{max} = 1$. Like this every reorganization emulates crack-in-two and no adaption of the partitioning fan-out is performed. The sorting threshold t_{sort} is set to 0 such that cracking continues no matter how small the partitions become. Using this configuration, we are able to nearly replicate the signature and thus the behavior of standard cracking.

Next, let us look at classical **scanning** and filtering. For the baseline, the indexing stays at 0 over the entire query sequence and the original column is processed. For the meta-adaptive index, we are almost able to emulate that behavior. We set all parameters to 0 such that no reorganization is happening — except for the very first query, that copies the keys over from the base table to a separate index column. Thus, all the indexing effort (the copying) is done in the beginning.

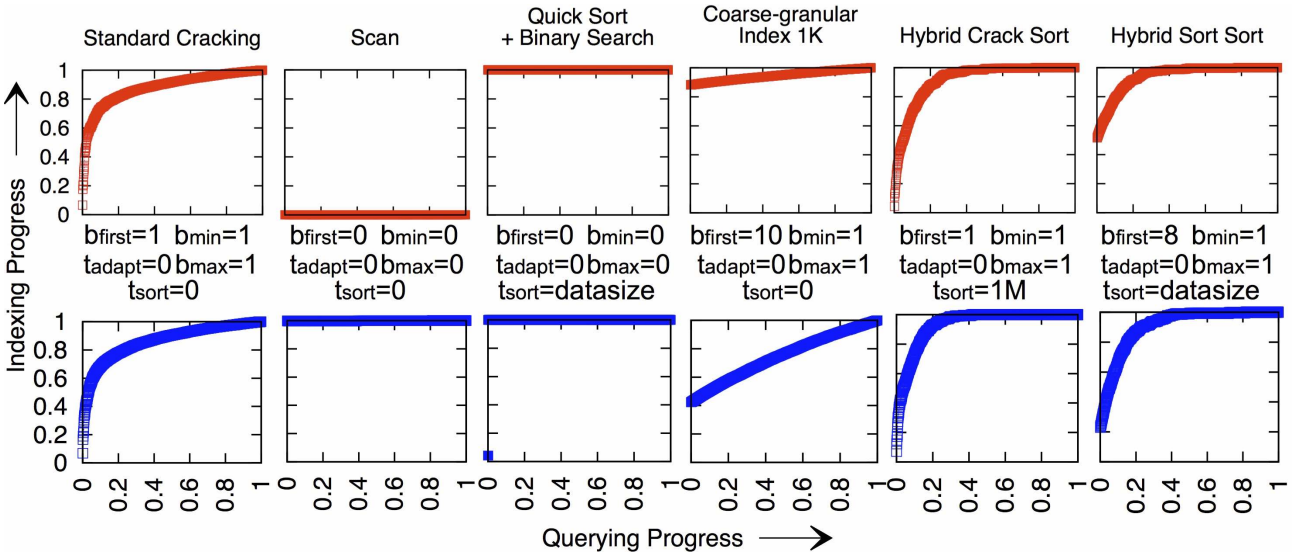


Fig. 10: ***Emulation of adaptive indexes and traditional methods.*** *The top row shows the signatures of the baselines from [1] in red. The bottom row shows the signatures of the corresponding emulations of our meta-adaptive index in blue, alongside with the parameter configurations that were used.*

Subsequent queries simply scan and filter the index column exactly as the baseline is doing it.

The other extreme is **full indexing** by completely sorting the keys and then searching for the boundaries to answer the queries. Thus, for the baseline, all indexing effort happens in the very first query that copies and fully sorts the index entries. Afterwards, no more indexing effort is invested. To emulate this behavior, we set $b_{first} = 0$ and $t_{sort} = 100\text{M}$. With this setting, the remaining parameters do not have any impact. The first query copies the keys over into the index column. The second query triggers the sorting of all keys as t_{sort} is set to the size of the entire column.

Coarse-granular index prepends a partitioning step to the very first query and subsequently continues query answering in the same way as standard cracking. The index is basically bulk-loaded with 1000 partitions. To emulate this behavior, we first set $b_{first} = 10$. This creates 1024 partitions during the copying from the base table into the index column. Afterwards, we continue emulating standard cracking by setting $b_{min} = b_{max} = 1$, leading to crack-in-two applications. The completion threshold t_{sort} is set to 0 to avoid the sorting of small partitions. As we can see in Figure 10, the shapes of the curves are quite similar — in both cases, a large portion of the indexing effort is performed in the very first query. For the baseline, more than 80% is invested into the initial range partitioning — for the meta-adaptive index, only around 40%. This is simply caused by the fact that the out-of-place radix partitioning implementation is faster than the comparison based range-partitioning implementation that was used in [1] and thus takes a smaller portion of the total indexing time.

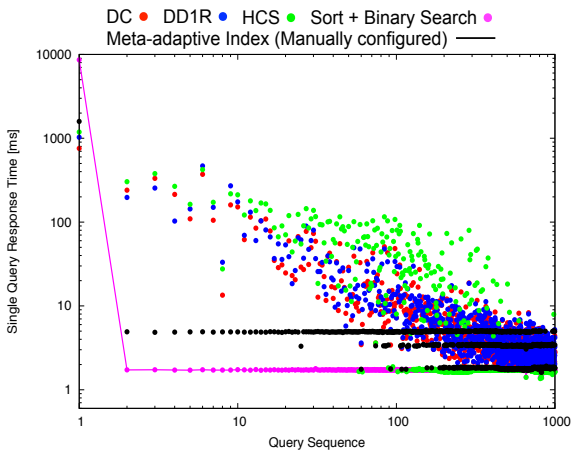
Hybrid Crack Sort generates a higher convergence speed as the results of a range query are directly sorted and subsequent queries can benefit. Of course, our meta-adaptive index can not replicate the exact processing flow of the hybrid methods. However, we can observe that this is not necessary at all to generate a similar behavior. To do so, we first set $b_{first} = b_{min} = b_{max} = 1$. This guarantees that at least the reorganization early on in the query sequence is as lightweight as for standard cracking. However, we also set $t_{sort} = 1\text{M}$. Thus, if a partition size reaches 1% of the column size, it is sorted. This ensures a much faster convergence than for standard cracking. As we can see, with this configuration we are able to emulate hybrid crack sort very well *while* providing a much simpler processing flow.

Finally, let us look at another representative of the hybrid methods, namely **hybrid sort sort**. In this case, sorting is also used as the way of reorganization for the initial column. This behavior speeds up convergence towards the fully sorted state even more. To emulate that, we first increase the amount of fan-out bits for the initial reorganization b_{first} to 8. This does not fully sort the column, but increases the amount of invested indexing effort in the very first query. Second, we set $t_{sort} = 100\text{M}$ such that any further access of a partition triggers sorting. By this, we are able to closely resemble the signature of hybrid sort sort using our meta-adaptive index.

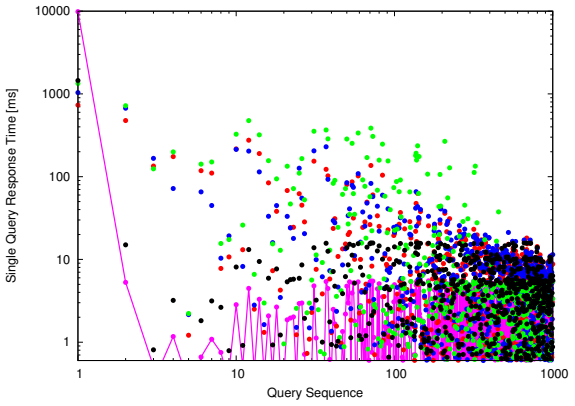
Using proper configurations, we are able to tune the index into one or the other direction and distribute the indexing effort along the query sequence in different ways. The question is now: does the ability to adapt to the characteristics of various adaptive indexes also help in terms of query response times?

C. Individual Query Response Time

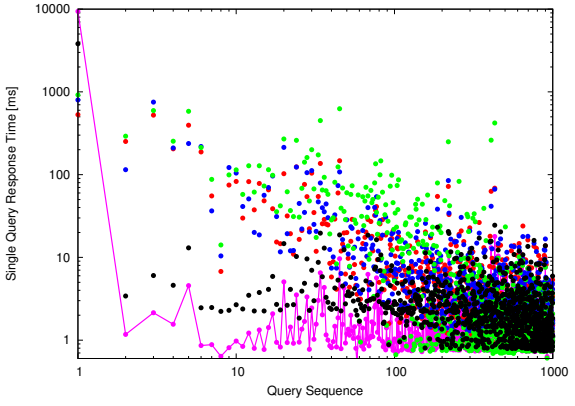
First, we focus on the *individual* query response time. The main goal of basically any adaptive index is to keep the pressure on the individual queries as low as possible. Therefore, for instance standard cracking invests the least amount of reorganizational work to answer a query. However, choosing the amount of reorganizational effort per query is not that trivial. It can pay off to penalize a single query



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



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



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

Fig. 11: ***Individual query response times*** *of the meta-adaptive index (configured according to Section VIII-C1) in comparison to baselines for a* ***uniform*** *(11(a)),* ***normal*** *(11(b)), and* ***Zipf-based*** *(11(c)) key distribution. The used query workload is RANDOM with* 1% *selectivity on the key range.*

a bit more to significantly speed up subsequent queries. To find out how our meta-adaptive index behaves in terms of individual query response time, we put it to the test against the main representatives of the different adaptive indexing classes: standard cracking (DC), stochastic cracking (DD1R), and hybrid crack sort (HCS). Additionally, we test sorting with binary search. Let us now see how we can configure the index.

1) Manual Configuration: Our primary goal is to keep the individual query response times low. The indexing effort should be nicely distributed along the query sequence. However, we should also have the accumulated query response time in mind as a secondary goal. Therefore, we choose the following configuration: For the first query, we use $b_{first} = 10$ bits as according to Figure 2(a), higher fan-outs make the partitioning significantly more expensive. Thus, with individual response time in mind, 10 bits are the limit. For subsequent queries, we balance between convergence speed and pressure on the individual queries as well, by setting $b_{min} = 3$ and $b_{max} = 6$. Thus, for partitions larger than $t_{adapt} = 64$MB, we keep the partitioning fan-out low as they do not fit into the TLB. As soon as the partition is smaller than $t_{sort} = 256$KB and thus fits into the L2 cache, we sort it. The skew tolerance is set to a high value of 5x to ensure that severe skew is defused and moderate skew is tolerated.

2) Experimental Evaluation: Let us now inspect the individual query response times of the meta-adaptive index in comparison with the baselines. We focus on the RANDOM query workload with a selectivity of 1% and test the uniform, normal, and Zipf distributed dataset.

Let us start with the results of the uniform workload in Figure 11(a). As we can see, the first query of the meta-adaptive index is slightly more expensive than that of the baselines. However, we can see that this investment certainly pays off as from the second query on, the individual response time dropped permanently below 10ms. In comparison to that, all the adaptive indexing baselines show significantly higher response times till around 100 queries and obviously converge much slower towards the sorted state. Especially hybrid crack sort shows very high response times even after 100 seen queries if it has to merge entries into the final column. Overall, the meta-adaptive index shows the most stable performance and offers early on fast individual response times, similar to the full index. Under a normal distribution in Figure 11(b), the very first query response times equal pretty much the ones under the uniform distribution, where the meta-adaptive index is only slightly slower than the baselines. For the rest of the query sequence, we clearly see a higher variance in response times for all methods, which is caused by the key concentration around the middle of the 64-bit space (2^{63}). However, only the meta-adaptive index achieves to stay below 20ms per query for each query, while the remaining adaptive methods cause response times that are an order of magnitude higher till around 100 seen queries. Finally, let us inspect the behavior under the Zipf distribution in Figure 11(c). This workload is basically the worst case for a radix based partitioning algorithm, as most values fall into few partitions. Here, indeed the meta-adaptive index is around four times slower in the first query than the three adaptive baselines. This is due to the necessary skew handling for this highly skewed distribution. Nevertheless, we can see that the investment pays off: From the second query on, we stay below around 30ms

per query, while the remaining methods show the spread we have seen previously already. Overall, we can see how well the meta-adaptive index behaves in terms of individual query response time under these extreme key distributions. It is able to outperform the three main representatives of the major adaptive indexing classes. Let us now see how it behaves in terms of accumulated query response times.

D. Accumulated Query Response Time

To test the performance with respect to accumulated query response time, we use again the manual configuration of Section VIII-C1. The evaluation of the individual response time indicated already that this configuration is also a very valid choice in terms of accumulated time. Nevertheless, we also want to evaluate how well an automatically generated configuration can perform. Thus, we use *simulated annealing* to come up with a configuration, that tries to optimize the parameters with respect to accumulated response time. Thus, let us first discuss how simulated annealing works conceptually and how it can be applied in our case.

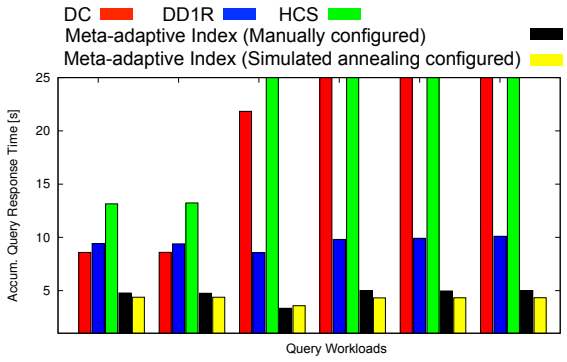
1) Automatic Configuration: As the parameters to configure depend on each other, we use simulated annealing [16] to confirm that a particular set of parameters indeed results in short accumulated query response times. We implement simulated annealing as described in [17]. It is a well known technique for approximating the global optimum of a function via stochastic probing. The general idea is to start with an initial configuration and a hot temperature. The temperature is decreased every few steps. While the temperature continues to decrease, the configuration is varied in every step. The magnitude of change in the configuration depends on (1) the temperature $temp$, (2) a random number $r \in [0, 1)$, and (3) manually set minimum and maximum values for the parameters to vary. After a certain temperature threshold is reached the algorithm stops. The final configuration is considered to be a reasonable approximation of the global minimum.

For the initial configuration we choose the parameters based on the manual configuration of our previous experiments. The temperature $temp$ is initialized to 1.0, and is reduced via division (by a constant α) of 2.0 in this case. The number of $steps$ performed per temperature is set to 12, which is twice the number of parameters to optimize (b_{sort} is fixed to 64 and thus not considered). The parameters to change are chosen based on a rotation. The probability $pAccept$ of accepting a "worse" configuration is set to $e^{-(dQRT/temp)}$, where $dQRT$ represents the change in accumulated query response times. The stopping criterion is set so that the final configuration is obtained if either $temp$ reaches approximately 0.0 or the configuration does not change between 20 temperature changes. As a quality function we simply use the accumulated query response time of the meta-adaptive index under the given configuration. The time to reach the final configuration is essentially dominated by the execution of the workload using the individual configurations. For example, for the uniform random workload, reaching the final configuration took 28 minutes. For each of the three key distributions, we perform an individual simulated annealing run to obtain a specialized configuration. In each case, we use the random query pattern as a representative workload. Table II presents the three obtained configurations.

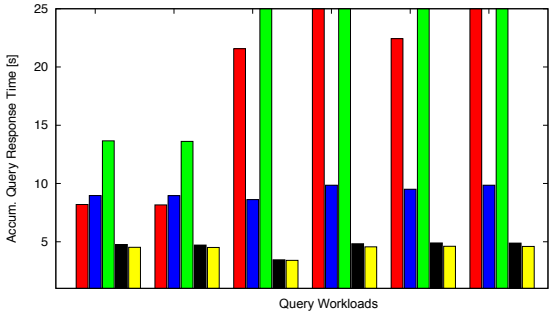
*TABLE II: Configuration to minimize accumulated query response time as determined by **simulated annealing**.*

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

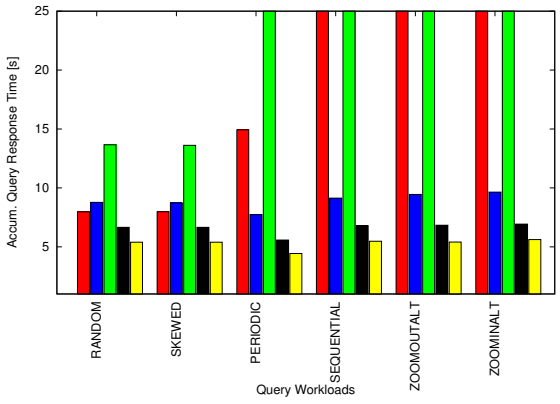
2) Experimental Evaluation: Let us now evaluate how our meta-adaptive index performs with respect to accumulated query response time under the 18 tested workloads. In Figure 12 we show the results for the meta-adaptive index as well as the three adaptive indexes standard cracking, stochastic cracking, and hybrid crack sort. As we can see, the meta-adaptive index behaves very well under the uniform key distribution in Figure 12(a). This holds for both the manual as well as the automatic configuration. The automatic configuration is slightly better for all workloads except of PERIODIC. Apparently, the higher initial fan-out using $b_{first} = 12$ bits is a better choice in terms of accumulated query response time. We can also see that t_{adapt} is configured significantly larger by simulated annealing, which basically causes using the maximum fan-out bits $b_{max} = 5$ for the next access. Therefore, simulated annealing identified fast convergence as the way to optimize for accumulated query response time. With respect to the baselines, we can also see that the meta-adaptive index performs well under all patterns. It is not prone to the workload like DC and HCS. Let us now look at the normal distribution in Figure 12(b). Again, the meta-adaptive index wins under all patterns clearly. The difference between manual and automatic configuration is very small, as simulated annealing produced a configuration that is similar to the manual one. Again, DC and HCS fail to handle the sequential query patterns. DD1R, which introduces a random crack per query, is pretty much resistant to the query patterns. However, it is still around twice as slow as the meta-adaptive index. Finally, let us inspect the Zipf distribution in Figure 12(c). Here, we can see the largest difference between the manual and the automatic configuration, where the latter one is significantly faster. Interestingly, the simulated annealing sets b_{first} only to 5 bits, leading to a small initial fan-out of 32 partitions. This makes sense in the presence of heavy skew. It is wasted effort to partition using a higher number of partitions if basically all entries end up in the first one. Thus, it is more efficient to use a smaller fan-out and then to recursively reorganize the first overly full partition again. We can also see that DD1R is still the closest competitor over all patterns. Still, no method is as robust and fast as our meta-adaptive index. Before concluding, let us investigate the scaling capabilities of our approach. Table III shows the runtime when varying the dataset size and the factor of slowdown with respect to a size of 100M under



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



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



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

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

the random uniform workload. As we can see, our approach scales linearly with respect to the datasize.

TABLE III: Scaling of the Meta-adaptive Index (manually configured) under uniform random workload.

Size	25M	50M	**100M**	200M	300M	400M	500M
Runtime	1.17s	2.39s	**4.77s**	9.63s	14.37s	19.82s	24.47s
Scaling	0.24x	0.50x	**1x**	2.01x	3.01x	4.15x	5.13x

IX. Conclusion

Our initial goal of the meta-adaptive index was to develop a technique which can fulfill several of the core needs of adaptive indexing at once. Firstly, we wanted to unify the large amount of specialized adaptive indexes that aim at improving a specific problem at a time in a single general method. We achieved this by identifying the fact that partitioning is at the core of any adaptive indexing algorithm. We proposed a meta-adaptive index that can emulate a large set of specialized indexes, which we were able to show by inspecting the indexing signatures. Based on this, we secondly looked at how the meta-adaptive index compares with respect to the classical adaptive indexing baselines and showed its superior performance under 18 different workloads with an average speedup of around 2x over the best baseline. Thirdly, we looked at how to manually and automatically configure the meta-adaptive index. Using simulated annealing, we were able to push the performance of the meta-adaptive index to the limits. Overall, 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.

References

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

[2] Felix Martin Schuhknecht, Alekh Jindal and Jens Dittrich, "An experimental evaluation and analysis of database cracking," *VLDBJ*, 2015.

[3] S. Idreos, M. L. Kersten, and S. Manegold, "Database cracking," *CIDR*, pp. 68–78, 2007.

[4] 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.

[5] 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.

[6] S. Idreos, M. Kersten, and S. Manegold, "Self-organizing tuple reconstruction in column-stores," in *SIGMOD 2009*, pp. 297–308.

[7] H. Pirk, E. Petraki, S. Idreos, S. Manegold, and M. L. Kersten, "Database cracking: fancy scan, not poor man's sort!" in *DaMoN, Snowbird, UT, USA, June 23, 2014*, pp. 4:1–4:8.

[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.

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

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

[11] O. Polychroniou and K. A. Ross, "A comprehensive study of main-memory partitioning and its application to large-scale comparison- and radix-sort," in *SIGMOD 2014*, pp. 755–766.

[12] F. M. Schuhknecht, P. Khanchandani, and J. Dittrich, "On the surprising difficulty of simple things: the case of radix partitioning," *PVLDB*, vol. 8, no. 9, pp. 934–937, 2015.

[13] A. Maus, "Arl, a faster in-place, cache friendly sorting algorithm," *Norsk Informatik konferranse NIK*, vol. 2002, pp. 85–95, November 2002.

[14] Y. Ioannidis and V. Poosala, "Balancing histogram optimality and practicality for query result size estimation," in *SIGMOD 1995*, pp. 233–244.

[15] A. Aboulnaga and S. Chaudhuri, "Self-tuning histograms: Building histograms without looking at data," in *SIGMOD 1999*, pp. 181–192.

[16] A. Belloni, T. Liang, H. Narayanan, and A. Rakhlin, "Escaping the local minima via simulated annealing: Optimization of approximately convex functions," in *COLT 2015*, pp. 240–265.

[17] R. Eglese, "Simulated annealing: A tool for operational research," *European Journal of Operational Research*, vol. 46, no. 3, pp. 271 – 281, 1990.