摘要 - 在本质上，许多物种灭绝了，因为它们无法快速适应环境。它们根本不足以适应越来越具有挑战性的环境。当算法太静态以应对其“环境”的特定挑战时，无论是工作负载，机器还是用户需求，都会发生类似的事情。在这方面，在本文中，我们探索了经过深入研究和引人入胜的自适应索引算法系列。经典自适应索引仅使数据的索引性适应工作负载。但是，我们将了解到目前为止我们已经忽略了二级索引的适应性，即指数算法本身。我们将证明这第二级适应性元适应性。基于仔细的实验分析，我们将开发一个自适应索引，它通过以下方式实现了自适应：（1）推广重组的方式，（2）对不断变化的索引性和不同的重组努力作出反应，以及（3）消除输入数据中的偏差分布。正如我们将要演示的那样，这使我们能够模拟大量专用自适应索引算法的特征。在一项广泛的实验研究中，我们将证明我们的元自适应索引在各种环境中都非常适用，并且在各种查询访问模式和密钥分发下优于大量专用自适应索引。

2018 IEEE 34th International Conference on Data Engineering

# Adaptive Adaptive Indexing

在最简单的自适应索引形式中，称为数据库Cracking或标准Cracking[3]，索引列相对于传入的查询谓词自适应地重新分区。如果选择[low, high]的范围查询进入，则低位的分区被分成两个分区，其中一个分区包含小于低的所有key，另一个分区包含大于或等于低的所有key。对于高分区重复相同的重组。完成这两个步骤后，可以通过对合格分区的扫描来回答范围查询。每个分区保存的键范围的信息存储在称为cracker index的单独索引结构中。以这种方式回答的查询越多，分区变得越精细。这样，查询响应时间逐渐收敛到传统索引之一。图1显示了这个概念。

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.**

目前存在大量的自适应索引算法。在我们最近的研究[1]，[2]中，我们分析了8篇论文，其中包括18种不同的索引方法。需要如此大量方法的原因在于适应性虽然提供了许多不错的特性，但也引入了令人惊讶的令人不快的问题[1],[2].例如，正如对这些工作的研究所表明的那样，自适应索引必须处理高方差，慢收敛速度，针对不同查询工作负载和数据分布的弱鲁棒性，以及个人和累积查询响应时间之间的权衡

## 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.

图1：数据库破解的概念重组多个查询并收敛到排序状态。

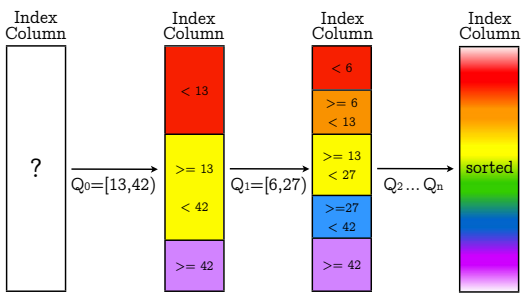



*Fig. 1: **Concept** of database cracking reorganizing for multiple queries and converging towards a sorted state.*

如果我们检查文献[4], [5], [6], [7], [8], [9], [10]提出所描述原理的变化，我们看到这些算法主要集中在减少单个 一次发行。例如，混合Cracking[5]试图提高朝向完整指数的收敛速度。随机破解[4]反而侧重于提高顺序查询工作负载的稳健性。因此，为了使系统具有自适应索引，实际上必须根据用户的需要和当前的工作量来进行扩展，并进行许多不同的实现。

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

这提出了这些算法实际上有多么不同的问题。在文献研究期间，我们做了两个观察：首先，每个Cracking算法的核心是简单的数据分区，将给定的key范围分成一定数量的分区。其次，算法之间的主要区别在于它们如何沿着查询序列分配索引工作量。有些方法倾向于在早期重新组织，而其他方法则在查询中尽可能地平衡工作量。基于这些观察，我们将提出一种广义的自适应索引算法，该算法适应专业方法的特征，同时优于它们。

（1）概括指数改进的方式。我们将数据分区识别为自适应索引中的常见重组形式。可以通过产生k个不相交分区的函数partition-in-k来表示各种类型的数据库破解以及排序。例如，我们可以使用k = 2来模拟标准破解(分别为二次Cracking )，而使用扇出k = 2^64可以表示对64位key进行排序。因此，partition-in-k将是唯一的 我们算法中重组的组成部分，使用高效基数分区技术的现场和非现场版本实现。

2375-026X/18/$31.00 ©2018 IEEE
DOI 10.1109/ICDE.2018.00066

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



（2）通过相对于要处理的分区的大小调整分区扇出k来调整重组工作。 经典方法使其重组工作在其生命周期中保持静止。 例如，crack-in-two将其输入总是分成两部分，与分区大小和索引状态无关。 但是，重组工作应该仔细调整到输入，以便将索引重新定义为可以在不确定查询响应时间的情况下进行个人操作。 为此，我们执行以下策略：通过减小必须重新定义的输入分区的大小，我们增加了分区k的扇出k。

除了使用的分区扇出外，partition-in-k的实际实现起着重要作用。 传统方法主要依赖于基于比较的方法，因为它们根据传入的查询谓词对key进行分区。 我们决定使用基于基数的分区算法，因为这种重组方法比基于比较的方法提供更高的分区吞吐量[11]。 当然，与基于比较的方法相比，基于基数的分区不生成关于给定谓词的分区，因此，需要为合格条目过滤生成的分区。 不过，考虑到性能优势，这是值得付出的代价

因此，我们利用重组努力的减少并重新组织更多细粒度以加速收敛，同时确保快速响应时间。 因此，如果分区达到足够小的大小，我们通过排序"完成"它，也可以在数据上启用有趣的订单。

（3）识别和化解倾斜的密钥分配并相应地调整重组机制以对抗它们。 默认情况下，仅当key分配是统一的时，基数分区才会创建平衡分区。 虽然经常存在均匀性，但依赖它是不小心的。 因此，我们引入了一种机制，能够解决由于第一个查询中存在偏差而导致的问题。 我们实现了两件事：首先，我们能够在没有开销的情况下检测输入中的偏差。 其次，在存在倾斜的情况下，递归拆分大于平均值的分区，以强制执行后续查询的平衡处理。

遵循这三个简单的概念，我们能够模拟特定的自适应索引算法。 通过七个配置参数，我们的通用算法可以专门用于关注诸如收敛速度，方差减小或偏斜阻力等属性，因此它可以模拟并可能替换大量专用索引。 我们将生成方法签名以可视化我们的仿真质量。 除了应用手动配置之外，我们还将使用模拟退火来优化参数集，以实现给定工作负载的最小累积查询响应时间。 现在让我们看看如何实现这种元自适应算法。

此外，我们必须区分可以利用out-of-place算法的第一个查询和后续查询，其中索引列仅在in-place重新组织。 在前一种情况下，我们可以使用高度优化的out-of-place基数分区，这已经在我们的研究中显示出其卓越的性能[12]。 它使用软件管理缓冲区，非时间流存储和优化的微布局来增强分区过程。 在后一种情况下，我们使用in-place基数分区算法，它以cuckoo-style [13]交换分区之间的元素，而不需要额外的内存。 两种算法共同构建了元自适应索引中的重组核心，并将在下一节中详细介绍。

看一下上一节，仍然是如何控制重组量的问题。 什么时候我们应该投入多少资源进行分区？ 为了解决这个问题，我们将进行一系列实验来研究不同扇出对不同情况下分区过程的影响。 我们必须区分可以利用out-of-place分区的首个查询和其他in-place重组的查询。 此外，我们必须区分不同的输入分区大小，因为它们极大地影响了重组所需的成本。 让我们首先看一下第一个查询。

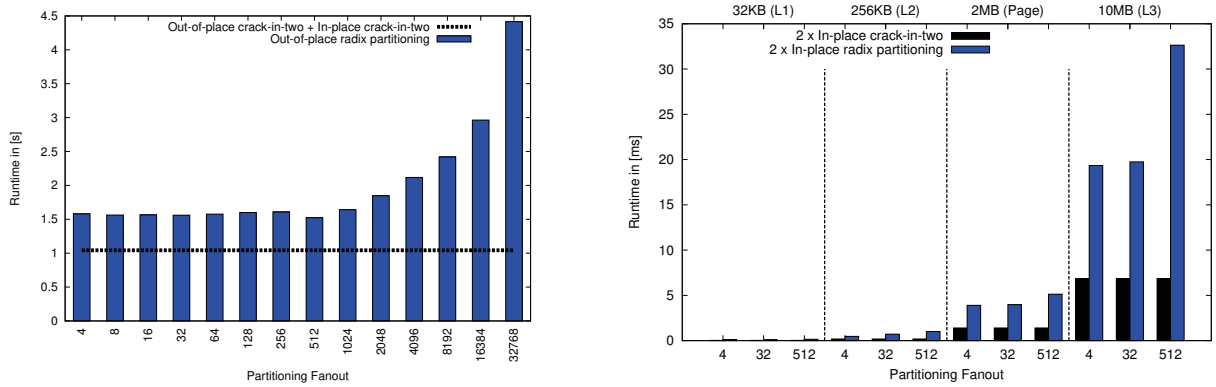
简单的数据分区是任何自适应索引算法的核心。 分区过程的应用扇出通过影响索引工作的收敛速度，方差和分布来决定方法的特征。 因此，能够自由地设置分区过程的扇出的算法能够适应各种自适应索引算法的行为。 因此，我们将仅使用k分区步骤来执行重组。

对于第一个查询，我们分析了1亿个8B键和8B个rowID条目的out-of-place分区的运行时间。 二手机器是一个中档服务器，我们稍后也会在实验评估中使用(详见第VIII-A节)。 因此，总共必须移动大约1.5GB的数据。 从整个无符号的64位整数范围以统一和随机的方式选择密钥。 我们重新组织单个范围查询[low, high)，其中低谓词将键范围分成大小为1/3和数据大小的2/3的分区。 高谓词将大小为2/3的分区随后分成两个大小相等的分区。 为了重新组织这个查询，我们考虑两个选项：经典方式（由标准破解使用）是将数据分离到两个分区相对于低，然后执行就地破解二进制 上部分区相对于高

重新组织首个查询。 标准cracking相对于高位预测低和in-place cracking两步执行不合适的二裂缝步骤。 相比之下，我们在[12]中以4到32,768的变化扇出显示了不合适的基数分区。

重组后续查询。 我们测试分区输入大小32KB(L1缓存), 256KB (L2缓存), 2MB(HugePage)和10MB (L3缓存)。 对于就地基数分区，我们将扇出4,32和512作为代表。

图2：选择[low, high]的范围查询的重新组织选择的比较。 我们必须区分非常的首个查询(图2(a))，其中的密钥从基础列中的基础复制，以及后续查询(图2(b))，它们in-place重组。 我们测试标准破解所应用的策略，并将其与基数分区进行比较。




(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.*

创建的中间分区回答查询。作为替代方案，由于我们无论如何都必须复制整个列，我们可以忽略查询谓词，而是使用我们高度优化的基于基数的方法[12]直接将数据分区到自定义扇出。虽然这种重组形式需要额外的过滤来回答查询，但它是一个有效的替代方案，因为过滤器的分区对于合理的扇出是很小的。

由于缓冲区可能会进入CPU缓存，因此我们有效地减少了到主存储器的次数，从而减少了因子b的TLB次数。 尽管这种技术使复制数据的数量翻了一番，但TLB未命中的减少显着降低了运行时间而不是天真的方法[12]。

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的基数分区：看看我们的out-of-place基数分区算法[12]是如何精确工作的。输入是算法获取源列以及所请求的分区数。输出是它在（新分配的）目标列中生成分区数据。该算法分两次通过：在第一遍中，我们扫描输入并计算每个分区将有多少条目。基于此直方图，我们初始化指针。在第二遍中，我们通过将条目复制到指定的分区来执行实际分区

*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].

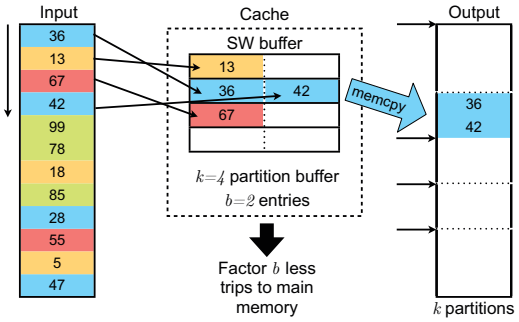



*Fig. 3:* ***Out-of-place partitioning*** *using* ***software managed buffers*** *[12].* 图3：使用软件管理缓冲区进行的out-of-place分区[12]

此外，我们应用所谓的非时间流式存储。这些SIMD内在函数允许在将软件管理的缓冲区迁移到目标分区时绕过CPU缓存。 图4显示了这个概念。 要使用一个缓存行的单个缓冲区，需要对AVX内部_mm256_stream_si256进行两次调用，其中每次调用写入一半缓存行。在内部，当CPU执行硬件写入组合时，这两个调用实际上一次触发高速缓存行的写入

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.

使用这些优化，我们能够显着降低分区期间cache和TLB的压力。现在让我们看看这种优化的out-of-place基数分区算法与crack-in-two的比较。

不幸的是，将数据从基表简单复制到第二遍中的分区对于分区大于32的扇出来说会变得非常昂贵[12]。当我们以随机方式写入目标分区时，如果我们分区到超过32个分区，则会触发TLB未命中（因为CPU只能为大页面缓存32个地址转换）。为了解决这个问题，我们采用了一种称为软件管理缓冲区的技术。图3在一个分为k = 4个分区的示例中可视化概念。我们首先将其写入第二个缓冲区，而不是将条目36直接写入第二个分区。每个分区的缓冲区大小为b = 2条。只有当一个缓冲区变满时，即在42写入缓冲区之后，我们才会将它一次性移动到相应的分区。

1）in-place基数分区：in-place基数分区是如何工作的。与在out-of-place的版本中一样，需要直方图生成阶段，其中我们计算进入每个k分区的条目数。有了这些信息，我们就可以确定每个分区的开始。现在，我们从头开始扫描分区p0并识别不属于分区p0的第一个条目x，但实际上是另一个分区，比方说p5。然后，我们扫描分区p5，直到我们找到不属于p5的第一个条目y。我们用x替换y并继续搜索过程并用条目y替换。这样做直到我们通过填充x在分区p0中留下的孔来关闭一个循环。我们执行这些交换循环，直到所有分区都包含正确的条目。

2）评估：在图2(a)中，我们测试从4到32再到768的扇出，用于out-of-place基数分区。有趣的是，我们能够使用基数分区创建大量分区，成本仅略高，而crack-in-two只产生三个分区。例如，创建512个分区比使用两次crackin-two创建三个分区慢1.45倍(分别慢半秒)。同时，创建512个分区构建的索引比只有3个分区的索引的粒度大170倍。此外，对于512个生成的分区，平均分区大小约为3MB，因此在后续查询中处理时很容易进入CPU的L3缓存。与此相反，在两次破解的情况下，每个分区仍然是500MB大。总的来说，首个查询的策略是明确的：创建一个显着更大的分区数量，从标准crack（仅创建三个分区）开始，并且随后会大幅减少平均分区大小

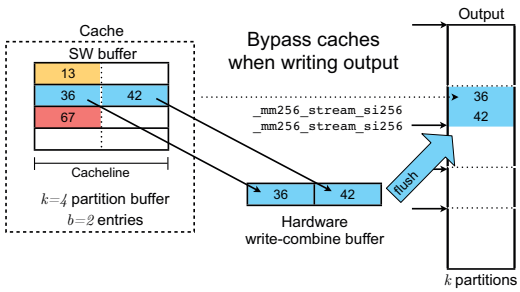



*Fig. 4: Enhancing **software managed buffers** using **non-temporal streaming stores** [12].*

图4：使用非时态流存储增强软件管理缓冲区[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).

但是如何继续进行后续查询？首先，由于数据现在存在于索引列中，我们不能再像第一个查询那样使用out-of-place分区算法。相反，任何重组都必须在in-place进行。为了评估选项，我们再次重新组织范围查询[low, high]. 我们考虑这样的情况：低谓词和高谓词分为两个不同的分区，每个分区大小为s，其中s等于特征系统大小(L1, L2, Hugepage和L3的大小)

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.

标准crack投入最少量的工作来回答查询：两次in-place cracking，根据low谓词分别重新组织两个分区。相比之下，我们再次评估基于基数的分区，但现在以in-place的形式。我们将具有给定扇出的in-place基数分区应用于低谓词和高谓词所属的两个分区。对于我们的测试，我们分别选择4,32和512个分区的扇出。

*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.**

2)评估：在图2(b)中，我们可以看到两次in-place crack再次成为最廉价的选择。然而，我们还可以观察到，随着输入尺寸的减小，两种测试方法之间的绝对差异减小。虽然10MB使用基数分区创建512个分区仍然比使用crack-in-two分解重组为两个分区要贵10ms左右，而对于2MB，它只需要大约1.5ms的成本。换句话说，输入越小，crack的开销就越可以忽略不计。这给了我们一个强有力的暗示我们应该如何在查询序列中调整分区扇出k：随着分区大小的减小，增大k。在足够小的尺寸下，通过对分区进行排序来完成分区，因为成本可以忽略不计。

### 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 $k = 2^{(\text{fan-out bits})} = 2^{f(s,q)}$. 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

我们在此函数中实现了以下高级设计目标：（1）将第一个查询与其余查询区别开来。（2）随着输入分区大小的减小，增加重组的粒度。（3）通过以足够小的尺寸对输入分区进行分类来完成输入分区

C.调整分区扇出第III-A节和第III-B节的进行实验表明，与轻量级方法相比，初始重组步骤可以创建大量分区而不会使运行时间恶化。其余的重组步骤应根据给定的分区大小调整其工作量。因此，现在让我们讨论在我们遇到的不同情况下，我们如何自适应地设置分区扇出。我们可以在下面的函数f(s,q)中总结我们的策略，它接收要重组的分区的大小s，以及作为参数的查询序列号q，并返回输入应该被分区的位数。我们将该返回值与扇出位的数量相加，即实际分区扇出k = 2(扇出位)= 2f(s,q)。该函数取决于一组配置元自适应性的参数

根据我们对图2(a)和图2(b)的观察，我们区分了第一个查询和其余查询。如果我们处于第一个查询(q = 0)，则该函数返回由参数bfirst确定的手动设置的扇出位数。如果我们在后续查询中，我们首先将分区大小s与阈值tadapt进行比较。如果s> tadapt，我们返回最小量的扇出位bmin，因为仍然认为分区对于更高分区扇出的应用而言太大。如果s小于或等于tadapt，但仍然大于完成分区tsort的阈值，我们自适应地将扇出位设置在bmin和bmax之间。分区越小，返回的扇出位数越高。如果s小于或等于tsort，则通过返回扇出位bsort的最大数量(例如，64位密钥为64)来实现分区，这导致分区的排序。总的来说，函数f(s,q)允许我们实现自适应分区的策略，我们在前面的章节中已经讨论过。图5显示了为样本配置生成的扇出位数。我们可以看到，该函数可以平滑地将生成的扇出位数调整为输入分区的大小。此外，我们使用partition-in-k将重组限制为当前查询谓词所属的分区。在这方面，重组仍然集中在感兴趣的分区上。

如第II部分所述，由于前者提供的运行时优势，我们更喜欢基于基数的分区而不是基于比较的分区。然而，虽然基于基数的分区提供了一种非常快速的方式来将条目分配给它们的分区，但是，当遇到高倾斜数据分布，它的效果较差。倾斜的数据分布导致产生非均匀分区，这可以极大地限制分区步骤的索引质量的增益。诸如Zipf分布之类的极端情况，其中最频繁的key出现的频率大约是第二频繁key的两倍，依此类推，需要生成所谓的等深度直方图以平衡分区

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.

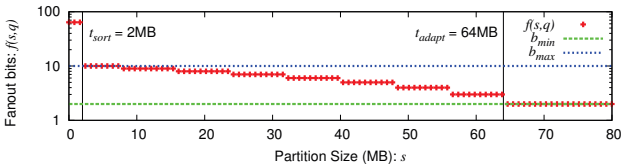



*Fig. 5: The **partitioning fan-out bits** returned by $f(s, q)$ for partition sizes $s$ from $0MB$ to $80MB$ and $q > 0$ with $t_{adapt} = 64MB$, $b_{min} = 2$, $b_{max} = 10$, $t_{sort} = 2MB$, and $b_{sort} = 64$.*

## 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.

在我们的元自适应索引中，我们通过引入等深out-of-place基数分区算法来解决高度偏斜数据的问题，该算法适用于第一个查询。传统上，等深度分区只需处理计算相同大小的分区[14]。但是，我们的解决方案还必须解决在所述分区上仍应继续进行基数分区步骤的问题。换句话说，生成的分区的边界必须以基数位分割。因此，我们不能简单地调整我们在任意键[15]上拆分和合并分区的解决方案，以使它们的大小相等。相反，我们必须根据基数位选择分区键。

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.

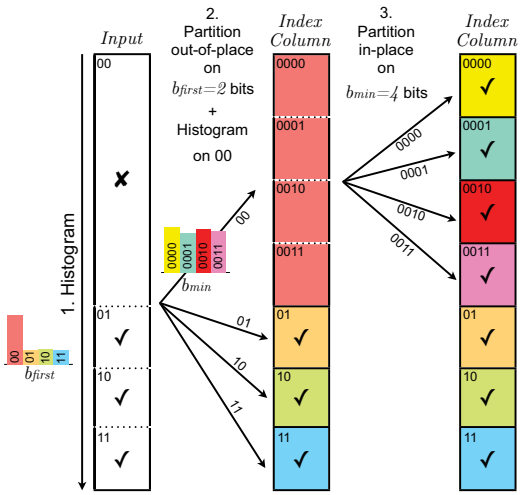



图6：消除输入偏斜。在阶段1中，我们在输入上构建关于b_first多位的直方图并找到偏斜的分区。在阶段2中，我们基于阶段1的直方图将输入分配到索引列中，同时仅在倾斜的分区上构建新的直方图，并且关注多个比特。对于最终阶段3，我们将分别在第2阶段的直方图中的索引列中的分区。

*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.

使用这种方法主要有两个好处：首先，如果输入列中的key没有偏斜，那么等深度基数分区的性能基本上等于标准基数分区算法的性能，因为piggy-backed直方图创建几乎无成本。其次，如果输入列在某个区域严重偏斜，则在该区域中基数分区的"分辨率"进一步增加。当然，这种方法在任何情况下都不能保证完全统一的分区。然而，正如我们将在实验中看到的，它提供了一种实用且轻量级的方法来消除由偏斜分布引起的严重负面影响

我们的解决方案如图6所示：首先，我们假设输入列中的key是统一的。因此，我们像往常一样使用b_first多位来构建out-place-in-k算法的阶段1中的初始直方图。随后，我们迭代新构建的直方图，并将每个bucket的大小与理论最优值（列大小/ k）* skewtol进行比较。这里，skewtol表示偏斜容差，使用户可以控制偏斜检测。一旦分区超过此阈值，它就会被算法标记为偏斜。在阶段2中，我们相对于阶段1中构建的直方图正常执行out-of-place radix partition-in-k。但是，当我们将元组复制到相应的分区时，同时在标记为的分区上构建新的直方图使用b_min很多位偏斜。因此，我们将下一个分区阶段的直方图生成捎带到当前的异地分区步骤。一旦我们完成了异地分区步骤，我们就已经生成了这个初始分区以及新的直方图。因此，在阶段3中，我们迭代所有偏斜的分区，并使用已构建的阶段2的直方图，相对于b_min许多位进一步对它们进行in-place分区。最后，我们将所有分区信息插入到索引中。

我们在前面的章节中已经看到了如何推广重组过程（第II部分），如何调整分区的扇出（第III部分）以及如何处理输入偏差（第IV部分）。与此同时，我们引入了一组参数，允许我们根据用户的优先级，系统的功能以及现有自适应索引的特征来调整算法的配置。表I再次列出了它们的含义。

## 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. |

当然，所有这些参数都可以由用户根据个人偏好手动设置。在第VIII-B节中，我们将设置参数以模拟各个自适应索引的特征。这表明我们的元自适应索引足以模拟现有的自适应索引，因此能够替换它们。此外，我们将演示如何手动校准参数（参见第VIII-C1节）并自动使用模拟退火（参见第VIII-D1节）来获取最小化累积查询响应时间的设置

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 $p[high]$

正如我们已经讨论了元自适应的核心主题，我们现在能够在一个方法中组合所有组件 - 我们的元自适应索引。主要目标是包括所有讨论的方面，同时保持算法尽可能简单和轻量级。算法1呈现了它的伪代码，它表示我们决定如何在传入查询下重组索引的逻辑。与文献中已知的自适应索引算法类似，我们的元自适应索引独立地处理每个查询。和以前一样，我们将范围查询的两个谓词表示为low和high，并对用p [low]和p [high]表示分区

现在，每个查询都按照相同的过程进行处理，初始查询除外。对于第一个查询，必须将输入从基表复制到单独的索引列中。因此，该算法使用k=2 ^ f(s,0)=2 ^ b_first（详见第III节）来使用k外的分区，以便复制数据，同时还可以捎带分区工作。平均时间（第7行）。创建的分区边界包含在索引中。在k的外部分区中，也执行上述的偏斜检测（详见第IV节）。在分布中的偏斜的情况下，使用k = 2 ^ b_min的就地分区在分区上应用，该分区明显大于平均分区大小。

```
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

随后，对p [high]分区重复相同的过程（30和31行）。最后，我们必须获取查询输出。我们首先重新检查更新的p [low]和p [high]分区的索引（36和37行）。注意，与例如标准破解，现有的分区边界不一定在给定的查询谓词低和高分割。因此，我们必须过滤边界分区，以防它们尚未完成。如果它们完成，我们可以对它们应用二进制搜索和扫描（40和47行）。如果它们还没有完成，我们应用简单的过滤（42和49行）。在它们之间的分区上，我们使用简单的扫描（44行），因为它们完全属于查询结果。

然后通过查询新建的p [low]和p [high]分区(10和11行)的更新索引，后过滤所述分区(13和15行)，并应用一个查询来获得首个查询的输出。扫描到中间区域(14行)。在这里为了简单起见，我们不讨论p [low] = p [high]的情况。当然，我们的实际实现意识到了这种情况。后续查询的处理方式截然不同：首先，算法再次查询索引的p [low]和p [high](22和23行)，以标识我们希望限制此查询完成的重组的分区。现在，我们首先检查p [low]分区是否已经分别完成分类(第25行)。如果它已经完成，则不需要在该分区上花费额外的索引工作。但是，如果尚未完成分区，则需要投入额外的工作。我们将扇出函数f(s, q)称为分区p [low]的大小s和当前查询序列号q，以确定哪个扇出应用于就地分区-k步骤并进行重组(第26行)

在我们将实验性的元自适应指数与现有的最先进的自适应索引算法进行测试之前，让我们回顾一下该领域最着名的文献。 在下面的实验评估中，最具代表性的算法将作为我们的元自适应指数的基线。

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.



标准破解[3]：当然，我们将元适应索引与最轻量级的数据库破解(DC)进行比较。 它提供最便宜的前期初始化，并为每个查询执行最少量的重组，以使用扫描来回答它。 它在统一的随机和偏斜查询分布下表现很好，但它很容易出现顺序工作负载。图7(a)以一个例子可视化概念。 让我们说一个查询来选择大于或等于10且小于14的所有条目。在标准破解中，使用给定的查询谓词应用两次破解二次。这意味着，索引列首先相对于10进行分区。然后，包含所有大于或等于10的条目的上半部分相对于14进行分区。关于键范围和分割线的信息以分开的方式存储 饼干指数。 当然，后续查询仅分区查询谓词所属的区域。

随机破解[6]：随机破解算法类旨在解决标准版本的主要问题，即顺序查询模式。 它可以抵御各种工作负载，因为它将重组与查询谓词分离到一定程度并引入随机性。存在各种不同形式的随机破解 - 在这项工作中，我们将使用DD1R作为基线，每个查询引入一个随机破解，另外还有关于谓词的重组(见图7(b))

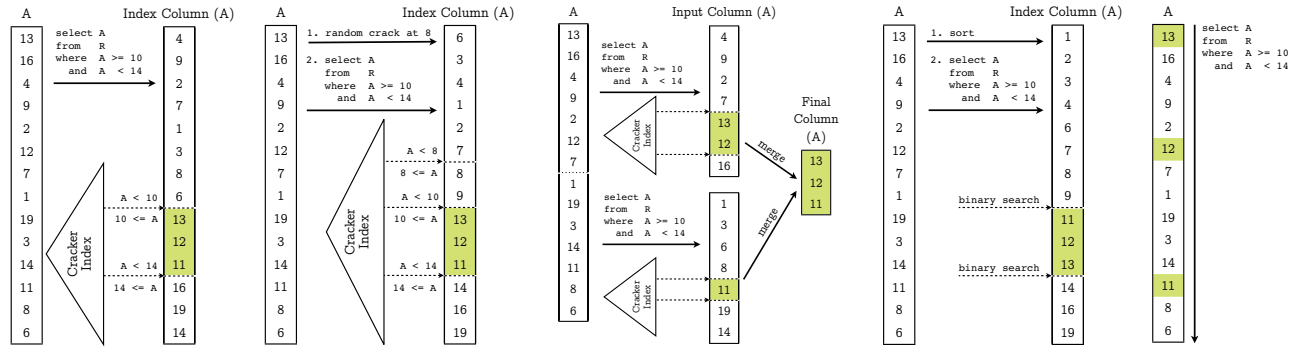
混合破解[5]：混合破解算法的目的是提高朝向完全分类状态的收敛速度。 与随机cracking 一样，也存在各种不同形式的混合cracking。 在这项工作中，我们将检查最突出的形式，称为混合cracking排序(HCS)和混合排序排序(HSS)。 如图7(c)所示，混合破解将输入非语义分成块(示例中为两个)，并对HCS应用标准破解，并对HSS进行单独分类，然后将每个块的合格条目合并并分类为最终。 从中回答查询的分区。

此外，我们评估极端排序+二进制搜索(完整索引，见图7(d))和扫描(无索引，见图7(e))。 请注意，并非所有以下评估和比较都显示所有基线方法。 我们将所提出的调查限制在那些具有特征且不会使可视化过载的方法上。

有了这个算法，现在让我们看看我们的元适应性指数如何与现场最先进的方法竞争。 我们基本上将评估分为两部分：在第一部分中，我们评估我们的索引是否确实可以模拟并可能替换专门的自适应索引。为此，我们将元自适应索引配置为与其他索引的特征相比较，并逐一比较签名。 这将评估先前描述的泛化是否有效以及我们的元自适应索引是否能够替换现有的自适应索引。 在第二部分中，我们将个人和累积查询响应时间的元自适应指数与基线进行比较。 我们测试了手动配置以及使用模拟退火校准的配置。

我们在所有实验中使用的系统包括两个Intel(R)Xeon(R) CPU E5-2407 @ 2.2 GHz，具有32KB的L1高速缓存，256KB的L2高速缓存和10MB的共享L3高速缓存。两个NUMA区域中的每一个都连接有24GB的DDR3内存。 实验中使用的操作系统是64位Debian GNU / Linux 8，内核版本为3.16，配置为自动使用大小为2MB的透明大页面。 TLB可以为大页面缓存32个虚拟到物理地址的转换。 该程序使用g ++版本4.8.4和开关-msse -msse2 -msee3 msse4.1 -msse4.2 -mavx -O3 -lrt进行编译。 我们重复每次实验运行三次并报告平均值。

我们在以下实验评估中使用的索引列再次包含1亿个条目，其中每个条目由8B键和8B rowID组成。因此，索引列的总数据大小约为1.5GB。总的来说，我们在测试中使用了三个特征密钥分布：第一，均匀分布生成0到264 -1之间的统一密钥。第二，正态分布，平均值为263，标准差为261.第三，（修改）Zipf分布，范围为0到264-1，形状为α= 0.6。为了生成该分布，我们首先根据Zipf分布计算10000个不同值的频率。然后，我们将无符号的64位密钥范围分成10000个相等大小的部分，并从每个范围中选择由先前计算的频率以均匀和随机方式给出的密钥。图8显示了三种分布。在使用随机shuf fling生成工作负载之后，单个条目的顺序是随机的。




(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.*

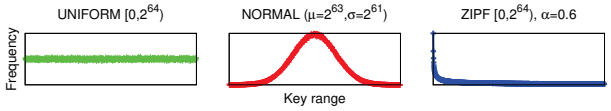



*Fig. 8: Different **key distributions** used in the experiments.*

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.

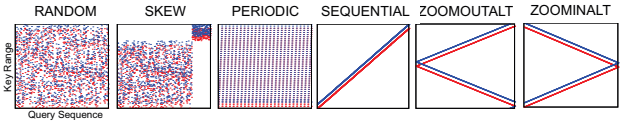



*Fig. 9: Different **query workloads**. Blue dots represent the high keys whereas red dots represent the low keys.*

### 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.



我们在实验中使用的查询工作负载由**1000**个范围查询组成，每个查询包括两个**8B**键，用于描述较低的上限。为了生成单个查询，我们使用**[4]**中详细描述的工作负载模式。 在图**9**中，我们可视化这些模式。我们使用**1%**的固定选择性在文献**[1]**中常见，这对评估有两个正面影响：首先，这种更高的选择性挑战了算法的收敛能力，因为范围查询的两个裂缝都位于附近 对彼此。 其次，对于**1%**的选择性，查询时间不会掩盖开裂时间。

现在让我们首先看看元自适应索引是否能够推广自适应索引的原理。我们的算法的一个主要动机是用一种方法替换大量现有的自适应索引，这种方法可以被配置为模拟不同的特征

在本节中，我们将评估是否可以实现这一点，以及如何配置元自适应索引来模拟代表性的现有自适应索引。作为评估的基线，我们选择了四个特征自适应指数的特征，如[1]中所示。对于查询序列（x轴）的给定查询，图10的图示出了在该查询之前已经执行的投资索引工作量（y轴）。我们显示了相对于总索引工作量（索引进度）和相对于总查询序列（查询进度）的查询的索引工作量。通过这种方式，我们清楚地看到粗粒度索引在应用标准破解之前使用1000个随机裂缝对第一个查询中的索引进行预分区，在第一个查询中已经执行了90%的索引进度，而标准破解需要一半的查询进度来投入那么多。

除自适应索引外，我们还将扫描和快速排序+二进制搜索的签名视为完全不使用索引或完全演化索引的极端情况的代表。 图10顶行中的所有基线签名都源自[1]的工作，我们使用均匀分布的密钥和查询生成它们，其中每个查询选择1%的数据。 在图10的底行，我们显示了我们的元自适应索引的相应签名。对于每种基线方法，我们以尽可能模拟其特征的方式配置元自适应索引。 我们的技术完全通过配置参数进行配置，如第V节所述，适用于均匀分布的密钥和随机范围查询（参见第VIII-A节中的UNIFORM和RANDOM）

我们从标准破解开始，作为自适应索引的经典代表。 为了模拟其行为，我们将所有扇出位复制到bfirst = bmin = bmax = 1。像这样，每次重组都会模拟二次破解，并且不会执行分区扇出的调整。 排序阈值tsort设置为0，这样无论分区变小多少，都会继续破解。使用这种配置.我们几乎可以复制签名，从而复制标准破解的行为

接下来，让我们看看经典的扫描和过滤。 对于基线，索引在整个查询序列中保持为0，并处理原始列。 对于元自适应索引，我们几乎能够模拟该行为。 我们将所有参数都设置为0，这样就不会发生任何重组 - 除了第一个查询，它将密钥从基表复制到单独的索引列。 因此，所有索引工作（复制）都在开始时完成。

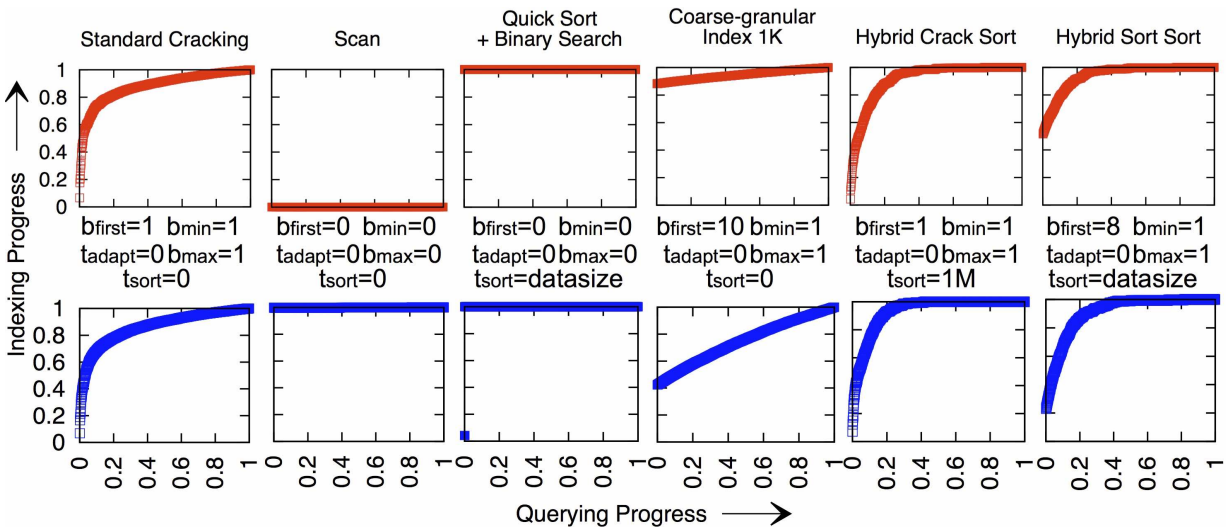



*Fig. 10:* ***Emulation of adaptive indexes and traditional methods.*** *The top row shows the signatures of the baselines from [1] in red. The bottom row shows the signatures of the corresponding emulations of our meta-adaptive index in blue, alongside with the parameter configurations that were used.*

后续查询只是在基线执行时完全扫描并过滤索引

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



另一个极端是完全索引，方法是对键进行完全排序，然后搜索边界以回答查询。因此，对于基线，所有索引工作都发生在复制和完全排序索引条目的第一个查询中。之后，不再投入索引工作。要模拟此行为，我们设置bfirst = 0和tsort = 100M。使用此设置，其余参数不会产生任何影响。第一个查询将密钥复制到索引列中。第二个查询触发所有键的排序，因为tsort设置为整个列的大小

粗粒度索引将分区步骤预先设置为第一个查询，然后以与标准破解相同的方式继续查询应答。该索引基本上是批量加载1000个分区。为了模拟这种行为，我们首先设置bfirst = 10. 这在从基表复制到索引列的过程中创建了1024个分区。之后，我们通过设置bmin = bmax = 1继续模拟标准开裂，从而导致二次裂缝应用。完成阈值tsort设置为0以避免对小分区进行排序。

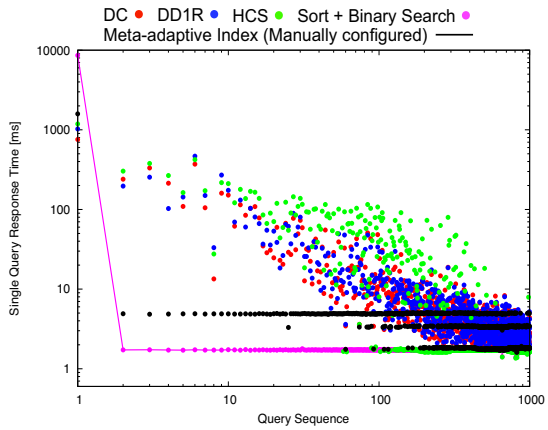
正如我们在图10中看到的，曲线的形状非常相似 - 在这两种情况下，大部分索引工作都是在第一个查询中执行的。对于基线，超过80%投入到初始范围划分中 - 对于元自适应指数，仅约40%。这仅仅是因为异地基数分区实现比[1]中使用的基于比较的范围分区实现更快，因此占用了总索引时间的较小部分。

使用适当的配置，我们能够将索引调整到一个或另一个方向，并以不同的方式沿着查询序列分配索引工作。现在的问题是：适应各种自适应索引特征的能力是否也有助于查询响应时间？

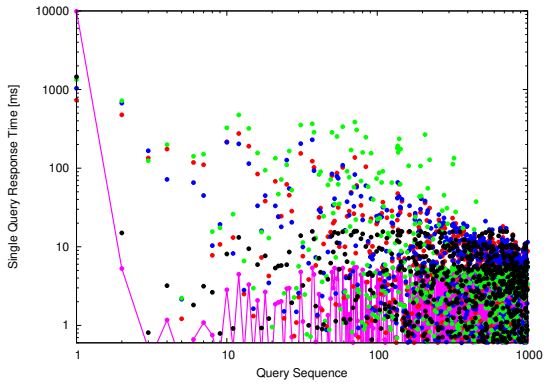
混合裂缝排序产生更高的收敛速度，因为范围查询的结果被直接排序并且后续查询可以是有益的。当然，我们的元自适应索引不能复制混合方法的精确处理流程。但是，我们可以观察到，根本不需要产生类似的行为。为此，我们首先设置bfirst = bmin = bmax = 1。这保证了查询序列中至少早期的重组与标准破解一样轻量级。但是，我们还设置了tsort = 1M。因此，如果分区大小达到列大小的1%，则对其进行排序。这确保了比标准破解更快的收敛。正如我们所看到的，通过这种配置，我们能够很好地模拟混合裂缝排序，同时提供更简单的处理流程。

最后，让我们看一下混合方法的另一个代表，即混合排序。在这种情况下，排序也用作初始列的重组方式。此行为会加速向完全排序状态的收敛。为了模拟这一点，我们首先将初始重组的扇出位数增加到8，这不会对列进行完全排序，但会增加第一个查询中投入的索引工作量。其次，我们设置tsort = 100M，以便分区的任何进一步访问都会触发排序。通过这种方式，我们能够使用我们的元自适应索引非常类似于混合排序排序的签名。

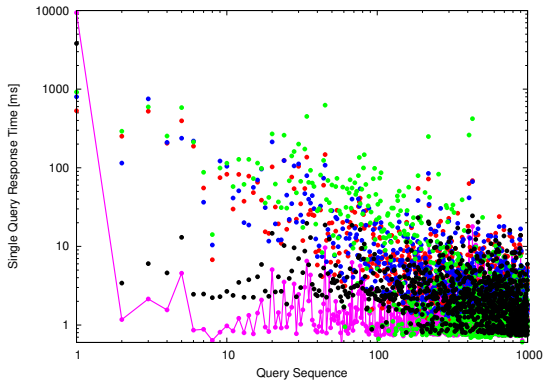
首先，我们关注单个查询响应时间。基本上任何自适应索引的主要目标是尽可能降低各个查询的压力。因此，例如标准破解投入最少量的重组工作来回答查询。但是，选择每个查询的重组工作量并不是那么简单。它可以为单个查询提供更多的惩罚，以显着加快后续查询。为了找出我们的元自适应索引在单个查询响应时间方面的行为，我们针对不同自适应索引类的主要代表进行测试：标准破解（DC），随机破解（DD1R）和混合破解 排序（HCS）。另外，我们用二分搜索测试排序。现在让我们看看如何配置索引。




(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

2）实验评估：现在让我们检查元自适应索引的各个查询响应时间与基线的比较。我们专注于RANDOM查询工作负载，选择性为1%，并测试统一，普通和Zipf分布式数据集。

1）手动配置：我们的主要目标是保持个人查询响应时间较短。索引工作应该沿着查询序列很好地分布。但是，我们还应将累积的查询响应时间作为次要目标。因此，我们选择以下配置：对于第一个查询，我们使用bfirst = 10bits，如图2（a）所示，更高的扇出使得分区显着更加昂贵。因此，考虑到个人响应时间，10位是限制。对于后续查询，我们通过设置bmin = 3和bmax = 6来平衡各个查询的收敛速度和压力。因此，对于大于tadapt = 64MB的分区，我们将分区扇出低，因为它们不会进入TLB。一旦分区小于tsort = 256KB并因此进入L2缓存，我们就对它进行排序。偏斜容差设置为5x的高值，以确保消除严重的偏斜并允许适度的偏斜

让我们从图11(a)中统一工作量的结果开始。我们可以看到，元适应性指数的第一个查询比基线的查询略贵。但是，我们可以看到，这项投资肯定会从第二次查询中得到回报，个人响应时间永久地下降到10ms以下。与此相比，所有自适应索引基线在大约100个查询之前显示出显着更高的响应时间，并且明显地朝向排序状态收敛得慢得多。

特别是混合破解排序即使在100次查询之后也显示出非常高的响应时间，如果它必须将条目合并到最终列中。总体而言，元自适应指数显示最稳定的性能，并提供快速的个人响应时间，类似于完整索引。在图11（b）中的正态分布下，第一个查询响应时间几乎与均匀分布下的查询响应时间相等，其中元自适应索引仅略慢于基线。对于查询序列的其余部分，我们清楚地看到所有方法的响应时间的变化更大，这是由64位空间中间的密钥集中引起的（263）。但是，对于每个查询，只有元自适应索引实现每个查询保持在20ms以下，而剩余的自适应方法导致响应时间高出一个数量级，直到大约100个查看的查询。最后，让我们在图11（c）中检查Zipfdistribution中的行为。对于基于基数的分区算法，此工作负载基本上是最坏的情况，因为大多数值都属于少数分区。在这里，实际上，第一个查询中的元自适应索引比三个自适应基线慢大约四倍。这是由于对这种高度偏斜的分布进行必要的偏斜处理。尽管如此，我们可以看到投资得到回报：从第二个查询开始，我们保持在每个查询约30毫秒以下，而其余方法显示我们之前已经看到的传播。总的来说，我们可以看到元自适应索引在这些极端密钥分布下的个体查询响应时间方面的表现如何。它能够超越主要自适应索引类的三个主要代表。现在让我们看看它在累积查询响应时间方面的表现。

为了测试累积查询响应时间的性能，我们再次使用第VIII-C1节的手动配置。 对个体响应时间的评估已经表明，这种配置在累积时间方面也是非常有效的选择。 然而，我们还想评估自动生成的配置可以执行的程度。 因此，我们使用模拟退火来提出一种配置，试图根据累积的响应时间来优化参数。 因此，让我们首先讨论模拟退火工作如何以及如何在我们的案例中应用它。

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 $\alpha$) 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

1）自动配置：由于配置的参数相互依赖，我们使用模拟退火[16]来确认一组特定的参数确实导致短的累积查询响应时间。 我们实现了[17]中描述的模拟退火。 通过随机探测来近似函数的全局最优是众所周知的技术。 一般的想法是从初始配置和高温开始。 每隔几步降低温度。 当温度继续降低时，每个步骤的配置都会发生变化。 配置的变化幅度取决于(1)温度温度，(2)随机数r∈[0,1)，以及(3)手动设置参数变化的最小值和最大值。 达到某个温度阈值后，算法停止。 最终的配置被认为是全局最小值的合理近似值

对于初始配置，我们根据之前实验的手动配置选择参数。温度温度初始化为1.0，并且在这种情况下通过除法（通过常数α）减小2.0，每个温度执行的步数设置为12，这是要优化的参数数量的两倍（bsort固定为64，因此不予考虑）

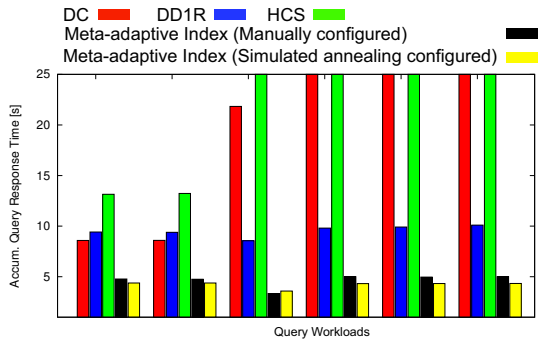
2）实验评估：现在让我们评估我们的元自适应索引在18个测试工作负载下的累计查询响应时间方面的表现。 在图12中，我们展示了元自适应指数以及标准裂缝，随机裂缝和混合裂缝排序这三个自适应指标的结果。 我们可以看到，在图12（a）中统一密钥分布下，元适应性指数表现得非常好。 这既适用于手动配置，也适用于自动配置。 除PERIODIC外，所有工作负载的自动配置略好一些。 显然，在累积查询响应时间方面，使用bfirst = 12bits的较高初始扇出是更好的选择。 我们还可以看到，通过模拟退火，tadapt的配置显着增大，这基本上导致使用最大扇出位bmax = 5进行下一次访问。

因此，模拟退火识别快速收敛作为优化累积查询响应时间的方法。 关于基线，我们还可以看到元适应性指数在所有模式下都表现良好。 它不容易像DC和HCS那样的工作量。 现在让我们看一下图12（b）中的正态分布。

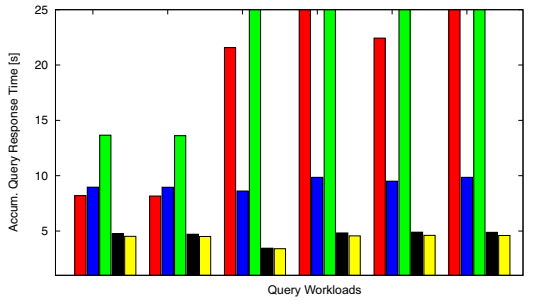
要根据旋转选择要更改的参数。接受"更差"配置的概率pAc被设置为e-（dQRT / temp），其中dQRT表示累积查询响应时间的变化。设置停止标准，以便如果温度达到约0.0或者在20个温度变化之间配置不发生变化，则获得最终配置。作为质量函数，我们简单地使用给定配置下的元自适应索引的累积查询响应时间。达到最终配置的时间主要取决于使用各个配置执行工作量。例如，对于统一的随机工作负载，达到最终配置需要28分钟。对于三个关键分布中的每一个，我们执行单独的模拟退火运行以获得专门的配置。在每种情况下，我们使用随机查询模式作为代表性工作负载。表II列出了三种获得的配置。

同样，元适应性指数在所有模式下都清晰地获胜。手动和自动配置之间的差异非常小，因为模拟退火产生的配置类似于手动配置。同样，DC和HCS无法处理顺序查询模式。 DD1R，每个查询引入一个随机破解，几乎抵抗查询模式。但是，它仍然是元自适应索引的两倍慢。最后，让我们检查图12（c）中的Zipf分布。在这里，我们可以看到手动和自动配置之间的最大差异，后者明显更快。有趣的是，模拟退火仅设置为5位，导致32个分区的初始扇出小。这在有严重偏斜的情况下是有意义的。如果基本上所有条目最终都在第一个分区中，则使用更多数量的分区进行分区是浪费精力。

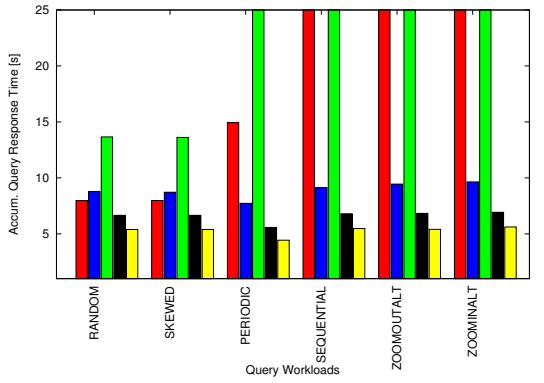
因此，使用较小的扇出然后再次递归地重新组织第一个过满的分区是更有效的。我们还可以看到DD1R仍然是所有模式中最接近的竞争对手。尽管如此，没有任何方法像我们的元自适应索引一样强大和快速。在结束之前，让我们研究一下我们方法的扩展能力。表III显示了在随机统一工作负载下改变数据集大小和减速因子相对于100M大小的运行时间。 正如我们所看到的，我们的方法相对于数据量线性扩展。




(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.

我们的元自适应索引的最初目标是开发一种技术，可以同时满足自适应索引的几个核心需求。首先，我们希望统一大量专门的自适应索引，这些索引旨在通过单一通用方法一次改进特定问题。我们通过确定分区是任何自适应索引算法的核心这一事实来实现这一目标。我们提出了一个可以模拟大量专用索引的元适应性索引，我们可以通过检查索引签名来显示这些索引。基于此，我们再次看到了相对于经典自适应索引基线的自适应索引，并显示了它在18种不同工作负载下的优越性能。平均加速比最佳基线高出约2倍。第三，我们研究了如何手动和自动配置元自适应索引。使用模拟退火，我们能够将元自适应索引的性能推向极限。总的来说，元适应性指数可以作为大量专用索引的有效替代指标，并且能够在最先进的方法上提高稳健性，运行时和收敛速度

## 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.