Progressive Indexes: Lightweight Indexing for Interactive Data Analysis

ABSTRACT

Interactive exploration of large volumes of data is increasingly common, as data scientists attempt to extract interesting information from large opaque data sets. This scenario presents a difficult challenge for traditional database systems, as (1) nothing is known about the query workload in advance, (2) the query workload is constantly changing, and (3) the system must provide fast responses to the issued queries. This environment is particularly challenging for index creation, as traditional database indexes require upfront creation, hence a priori workload knowledge, to be efficient.

In this paper, we introduce progressive indexing, a novel adaptive indexing technique that focuses on automatic index creation while providing interactive response times to incoming queries. Its design allows queries to have a limited budget to spend on index creation. This allows for systems to always provide interactive answers to queries during index creation while being robust against varying workloads and data distributions.

We provide a detailed analysis of state-of-the-art adaptive indexing techniques and perform an extensive experimental study that shows that progressive indexing is a more lightweight, robust (i.e., predictable) and convergent technique than adaptive indexing under various data distributions and workload patterns.

ACM Reference Format:

. 2019. Progressive Indexes: Lightweight Indexing for Interactive Data Analysis. In *Proceedings of ACM SIGMOD Conference (SIGMOD'19).* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Data scientists perform exploratory data analysis to discover unexpected patterns in large collections of data. This process is done with a hypothesis-driven trial-and-error approach [26]. They query segments that could potentially provide insights, test their hypothesis and either zoom in on the same segment or move to a different one depending on the insights gained.

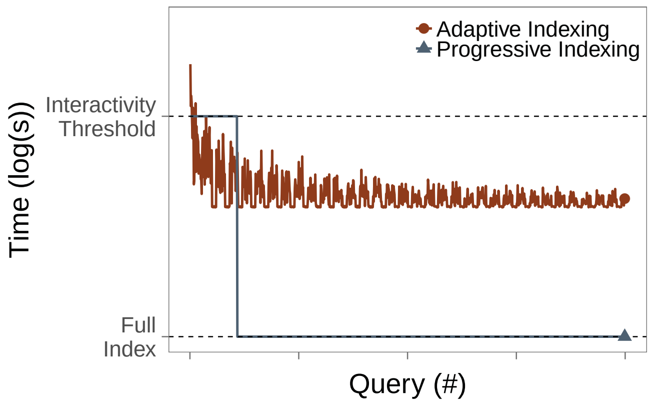


Figure 1: Indexing within the interactivity threshold.

Fast responses to queries are crucial to allow for interactive data exploration. The study by Liu et al. [19] shows that any delay larger than 500ms (the "interactivity threshold") significantly reduces the rate at which users make observations and generate hypotheses.

When dealing with small data sets, providing answers within this interactivity threshold is possible without utilizing indexes. However, exploratory data analysis is often performed on larger data sets as well. In these scenarios, indexes are required to speed up query response times.

Index creation is one of the major difficult decisions in database schema design [8]. Based on the expected workload, the database administrator (DBA) needs to decide whether creating a specific index is worth the overhead in creating and maintaining it. This considerable up-front cost creates a trade-off that requires careful consideration, experimentation, and knowledge about the database.

Creating indexes up-front is especially challenging in exploratory and interactive data analysis, where queries are not known in advance and workload patterns change frequently. In these scenarios, it is not certain whether or not creating a complete index is worth the large initial cost. In spite of these challenges, indexing remains crucial for improving database performance. When no indexes are present, even simple point and range selections require expensive full table scans. When these operations are performed on large data sets, indexes are essential to ensure interactive query response times. There are two main strategies that aim to release the DBA of having to manually choose which indexes to create.

(1) Automated index selection techniques [1, 4, 6, 7, 10, 20, 27, 30] accomplish this by attempting to find the optimal set of indexes given a query workload, taking into account the benefits of having an index versus the added costs of creating the index and maintaining it during modifications to the database. However, these techniques require a priori knowledge of the expected workloads and do not work well when the workload is not known or changes frequently, not being suitable for interactive data exploration.

(2) Adaptive indexing techniques such as database cracking [9, 11–17, 21, 22, 24, 25] are a more promising solution. They focus on automatically building an index as a side effect of querying the data. An index for a column is only initiated when it is first queried. As the column is queried more, the index is refined until it approaches the performance of a full index. In this way, the cost of creating an index is smeared out over the cost of querying the data many times, though not necessarily equally, and there is a smaller initial overhead for creating the index.

Since the index is refined in the areas targeted by the workload, database cracking is well suited for skewed workloads that focus on small partitions of the data. However, because of this workload adaptivity cracking does not offer robust performance against changes in workload patterns. Queries that deviate from the already targeted areas will target unrefined sections of the index, leading to significant and unpredictable spikes in performance. In data exploration scenarios where interactive responses are required, such performance spikes are undesirable.

Database cracking also penalizes the initial set of queries heavily, resulting in performance that is many times worse than that of a full scan. These spikes can cause performance to slow down past the interactivity threshold, as shown in Figure 1. As database cracking only works on individual columns, this performance spike occurs whenever a new column is queried, leading to considerable delays whenever a different partition of the data is queried.

In this paper, we introduce a new incremental indexing called *progressive indexing* that aims to solve these problems. It differs from adaptive indexing in that the amount of time that is spent on index creation and refinement can be automatically adapted, preventing queries from slowing down past the interactivity threshold. As a result, progressive indexing offers robust performance and convergence independent of sudden changes in the workload.

The main contributions of this paper are:

- We introduce several novel progressive indexing techniques that offer more predictable and robust performance than the state–of–the–art adaptive indexing techniques.
- We investigate their performance, convergence, and robustness in the face of different data distributions and querying patterns by evaluating them against existing adaptive indexing techniques.
- We provide a decision tree to assist in choosing which indexing technique to use.
- We provide a cost-model for each of the adaptive indexing techniques and use the cost-models to automatically adapt the indexing budget to prevent performance spikes from crossing the interactivity threshold.
- We describe how the indexes can be maintained when new entries are appended to the data set, and compare our technique against the techniques for handling updates in adaptive indexing.
- We provide Open-Source implementations of each of the techniques we describe.[1]

Outline. This paper is organized as follows. In Section 2, we investigate related research that has been performed on automatic/adaptive index creation. In Section 3, we describe our novel progressive indexing techniques and discuss their benefits and drawbacks. In Section 4, we perform an experimental evaluation of each of the novel methods we introduce, and we compare them against state of the art (adaptive) indexing techniques. In Section 5 we draw our conclusions and present a decision tree to assist in choosing which progressive indexing technique to use. Finally, in Section 6 we discuss future work.

2 RELATED WORK

Automatic index creation and maintenance has been a challenging and long-standing problem in database research. Even when the workload pattern is known, selecting and creating the set of optimal indexes is an NP-Hard problem [8]. When the querying pattern is not known in advance, optimal a-priori index creation is impossible. Automatic index techniques can be split into two categories, (1) automatic index selection and (2) adaptive index creation.

2.1 Automatic Index Selection Techniques

Automatic index selection techniques [1, 4, 6, 7, 10, 20, 27, 30] attempt to solve the problem by, given an existing (or expected) workload of the system, selecting the set of indexes that would result in optimal performance. The problem with these methods is that they can only be used when the workload of the system is known and stable. In an environment

[1]Our implementations and benchmark simulator will be made available for the camera-ready to not compromise double-blind review.

where the workload is unknown or rapidly changing, beyond what is known upfront, automatic index selection techniques do not offer much help. In addition, these techniques require sufficient time and space to invest in constructing a large full index upfront.

2.2 Adaptive Indexing Techniques

Adaptive indexing techniques are an alternative to a priori index creation. Instead of constructing the index upfront, the index is constructed as a by-product of querying the data. These techniques are designed for scenarios where the workload is unknown and there is no idle time to invest in index creation. The high investment of creating an up-front full index is smeared out over the cost of subsequent queries.

Database Cracking [16] (also known as "Standard Cracking") is the original adaptive indexing technique. It works by physically reordering the index while processing queries. It consists of two data structures: a cracker column and a cracker index. Each incoming query cracks the column into smaller pieces and then updates the cracker index with the reference to those pieces. As more queries are processed the cracker index converges towards a full index. This process is visualized in Figure 2a.

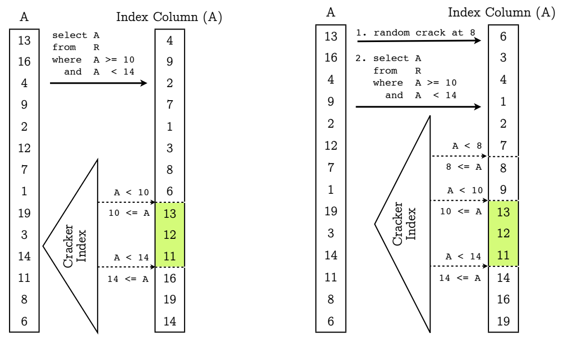
While database cracking accomplishes its mission of constructing an index as a by-product of querying, it suffers from several problems that make it unsuitable for interactive data analysis: (1) cracking adds a significant overhead over naive scans in the first iterations of the algorithm, (2) the performance of cracking is not robust, as sudden changes in workload cause spikes in performance, and (3) convergence towards a full index is slow and workload-dependent.

There is a large body of work on extending and improving database cracking. These improvements include better convergence towards a full index [9, 25], more predictable performance [12, 24], more efficient tuple reconstruction [15, 17, 25], better CPU efficiency [11, 21, 22], predictive query processing [29] and handling updates [13, 14]. Below, we give an overview of the work done on improving robustness as well as on handling updates.

2.2.1 *Improving Robustness.*

Stochastic Cracking [12] addresses the unpredictable performance problem by creating partitions using a random pivot element instead of pivoting around the query predicates. The pivot is used to perform arbitrary reorganization steps for more robust query performance. The actual cracking still only occurs in the pieces where the query predicates fall into. This is visualized in Figure 2b.

By using a random pivot element, it is less likely that large pieces of data remain uncracked. However, as stochastic cracking only cracks the pieces in which the query predicates fall, this can still occur, causing large performance



(a) Standard cracking. **(b) Stochastic cracking.**

Figure 2: Cracking algorithms [24].

spikes when later query predicates fall into these large yet uncracked pieces.

Progressive Stochastic Cracking [12] performs stochastic cracking in a partial fashion every iteration. It takes two input parameters, the size of the L2 cache and the number of swaps allowed in one iteration (i.e., a percentage of the total column size). When performing stochastic cracking, progressive stochastic cracking will only perform at most the maximum allowed number of swaps on pieces larger than the L2 cache. If the piece fits into the L2 cache, it will always perform a complete crack of the piece.

While progressive stochastic cracking reduces the initial cost of adaptive indexing, it still requires a full copy of the entire column, causing it to be significantly slower than only performing a full scan.

Coarse-Granular Index [25] improves stochastic cracking robustness by creating equal-sized partitions when the first query is executed. It also allows for the creation of any number of partitions instead of limiting the number of partitions to two, letting the DBA decide between the trade-off of the higher cost of the first query versus building a more robust index.

2.2.2 *Handling Updates.*

In database cracking, updates are merged to the index in a lazy fashion [13, 14]. They are stored in a separate "pending appends" vector that is kept sorted and are only merged into the index when needed for query processing. Three merge algorithms exist for cracking. They differ in the amount of data that is merged while executing one query.

Merge Complete Insertions (MCI). Once a query requests data from a cracked column, all elements from the append list are merged into the index. This algorithm has the disadvantage of performing all appends in a single query, heavily penalizing it. However, after the query, the appends list no longer needs to be considered until new entries are added or updated.

Merge Gradually Insertions (MGI). The MGI strategy only merges the elements from the append vector that match the current query. When a query requests values from a cracker column, the append vector is first checked for possible matches. All the matching elements in the append vector are then merged into the cracker index.

Merge Ripple Insertions (MRI). The MRI strategy minimizes the cost of merging the elements into the cracker index by swapping elements from previous pieces that do not match the query predicate back into the append list. This strategy has the lowest per query penalty as it avoids shifting pieces that are uninteresting to the query predicate. However, it is the strategy that takes the longest to fully merge the appends into the index.

3 PROGRESSIVE INDEXING

While there has been a lot of work done on improving robustness and convergence of database cracking, the current state–of–the–art adaptive indexing techniques still suffer from robustness problems, visualized as the large spikes in performance, workload-dependent convergence and significant initial query cost. These issues make current adaptive indexing techniques unsuitable for use in interactive data analysis, as the performance spikes from the initial queries or from querying large uncracked pieces will result in performance spikes that could cross the interactivity threshold.

Another issue with current adaptive indexing techniques is space utilization. Current adaptive indexing techniques all start by allocating a full copy of the column data with index positions, requiring $O(n)$ storage after the first query.

The large initial query cost and the large initial space requirement for database cracking are especially relevant in data exploration workloads. Whenever the data scientist inspects a new column in the data set for the first time, cracking immediately takes a significant amount of time and space to crack the column. If the data scientist then decides the column is not interesting or not relevant, a large amount of time and space has been wasted on creating this index.

Progressive Indexing. To address these issues, we propose a more robust adaptive indexing technique that we call *progressive indexing*. The core features of progressive indexing are that (1) it adds a low initial overhead over naive scans, both in terms of time and memory requirements, (2) the amount of indexing effort is automatically tuned so the interactivity threshold is never crossed, (3) it offers robust performance and convergence regardless of the underlying data distribution, workload patterns or query selectivity.

As a result of the small initial cost, progressive indexing occurs without significantly impacting worst-case query performance. Even if the column is only queried a single time, only a small penalty is incurred. On the other hand, if the column is queried hundreds of times, the index will reliably converge towards a full index and queries will be answered at the same speed as with an a-priori built full index.

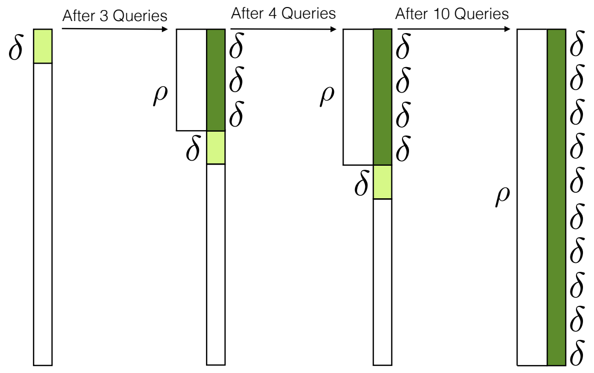


Figure 3: Creation phase of Progressive Indexing.

All progressive indexing algorithms progress through three canonical phases to eventually converge to a full B+-tree index: the *creation phase*, the *refinement phase*, and the *consolidation phase*. Each phase lasts for multiple queries, keeping the extra index-creation, -refinement, or -consolidation work per query strictly limited.

Creation Phase. The creation phase progressively builds an initial "crude" version of the index by adding another δ fraction of the original column to the index with each query. To do so, query execution during the creation phase is performed in three steps (visualized in Figure 3):

(1) Performing an index lookup on the ρ fraction of the data that has already been indexed;
(2) Scanning the not-yet-indexed $1 - \rho$ fraction of the original column;
and while doing so,
(3) Expanding the index by indexing another δ fraction of the total column.

As the index grows, and the fraction ρ of indexed data increases, an ever smaller fraction of the base column has to be scanned, progressively improving query performance. Once all data of the base column has been added to the index, the creation phase is succeeded by the refinement phase.

Refinement Phase. With the base column no longer required to answer queries, we only perform lookups into the index to answer queries. While doing these lookups, we further refine the index, progressively converging towards a fully ordered index. Once the index is fully ordered, the refinement phase is succeeded by the consolidation phase.

Consolidation Phase. With the index fully ordered, we progressively construct a B+-tree from it, since a B+-Tree is more efficient than binary search when executing very selective queries. Once the B+-tree is completed, we use it exclusively to answer all subsequent queries.

System	ω	cost of sequential page read (s)
	κ	cost of sequential page write (s)
	ϕ	cost of random page access (s)
	γ	elements per page
Dataset & Query	N	number of elements in the data set
	α	fraction of data to be scanned in index
Index	δ	fraction of data to-be-indexed
	ρ	fraction of data added to index
Progressive Cracking	h	height of the binary search tree
	σ	cost of swapping two elements (s)
Progressive Radixsort	b	number of buckets
	s_b	max elements per bucket block
	τ	cost of memory allocation (s)
B+-Tree	β	tree fanout

Table 1: Parameters used to model the cost of the progressive indexing techniques.

Setting δ. The value of δ can either be chosen by the DBA, in which case a fixed amount of data will be indexed every query, or chosen by the system such that the query performance will never degrade past the interactivity threshold ($t_{interactivity}$). In the automatic δ selection process, we use a cost model to determine how much time we can spend on indexing to guarantee that we never cross the interactivity threshold. The cost model takes into account the query predicates, the selectivity of the query and the state of the index in a way that is not sensitive to different data distributions or querying patterns and does not rely on having any statistics about the data available. To allow for robust query execution times regardless of the data, we avoid branches in the code and use predication when possible [3, 23]. The parameters and constants that are used in the cost models are shown in Table 1.

In the following sections, we will introduce four progressive indexing algorithms: Progressive Cracking, Progressive Radixsort (MSD), Progressive Bucketsort (Equi-Height), and Progressive Radixsort (LSD).

3.1 Progressive Cracking

Progressive Cracking is largely inspired by Database Cracking as discussed in Section 2.2, i.e., much like Database Cracking, Progressive Cracking can be seen as a Progressive QuickSort. However, Progressive Cracking is specifically designed to meet the robustness, convergence, and interactivity requirements for Progressive Indexing. Figure 4 depicts a snapshot of the creation phase and two snapshots of the refinement phase. We omit a snapshot of the consolidation phase as creating a B+-tree from a sorted array is rather straightforward. We discuss all three phases in detail in the following paragraphs.

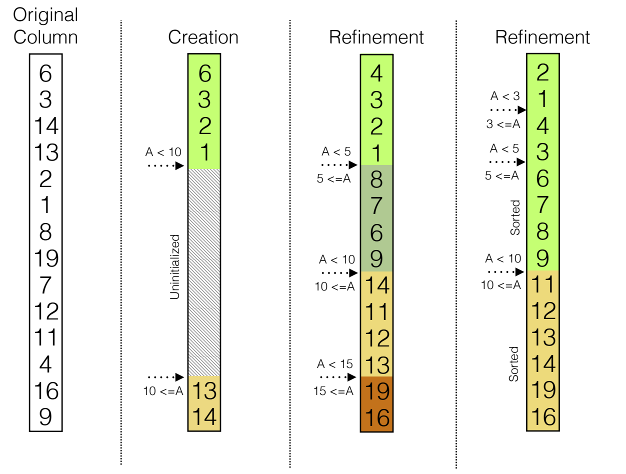


Figure 4: Progressive Cracking.

Creation Phase. In the first iteration, we allocate an uninitialized column of the same size as the original column and select a pivot, and we always use the element in the middle of the piece to be cracked as a pivot. We then scan the original column and copy the first $N * \delta$ elements to either the top or bottom of the index depending on their relation to the pivot. In this step, we also search for any elements that fulfill the query predicate and afterwards scan the not-yet-indexed $1 - \rho$ fraction of the column to compute the complete answer to the query. In subsequent iterations, we scan either the top, bottom or both parts of the index based on how the query predicate relates to the chosen pivot.

The total time taken is the sum of (1) the scan time, (2) the index lookup time and (3) the additional indexing time. Based on the query predicates and the initial pivot chosen, we know the fraction $[0, \rho]$ of the indexed data that we need to scan to correctly answer the query. The remaining $1 - \rho$ fraction of the data we always need to scan. However, while we are scanning, we pivot an additional δ fraction of the data. This results in the following cost for the initial indexing process:

$$t_{scan} = \omega * \frac{N}{\gamma}$$
$$t_{pivot} = (\kappa + \omega) * \frac{N}{\gamma}$$
$$t_{total} = (1 - \rho + \alpha - \delta) * t_{scan} + \delta * t_{pivot}$$

For the value of α for a given query, we obtain the following bound on δ to ensure we do not cross the interactivity threshold.

$$\delta \leq \frac{(1 - \rho + \alpha) * t_{scan} - t_{interactivity}}{t_{scan} - t_{pivot}} \quad (1)$$

Refinement Phase. We refine the index by recursively continuing the (quick-)sort in-place in the separate sections. We maintain a binary tree of the pivot points. In the nodes of this tree, we keep track of the pivot points and how far along the pivoting process we are. To do an index lookup, we use this binary tree to find the sections of the array that could potentially match the query predicate and only scan those, even when the full pivoting has not been completed yet.

In the refinement phase, we no longer need to scan the base table. Instead, we only need to scan the fraction α of the data in the index. However, we now need to (1) traverse the tree to figure out the bounds of α, and (2) swap elements in-place inside the index instead of sequentially writing them to refine the index. When we reach a node that is smaller than the L1 cache, we sort the entire node instead of recursing any further. After sorting a node entirely, we mark it as sorted. When two children of a node are sorted, the entire node itself is sorted, and we can prune the child nodes. As the algorithm progresses, leaf nodes will keep on being sorted and pruned until only a single fully sorted array remains. This results in the following cost for the refinement process:

$$t_{lookup} = h * \phi$$
$$t_{refine} = (\kappa + \omega) * \frac{N}{\gamma} + \lceil \sigma * \frac{N}{2} \rceil$$
$$t_{total} = t_{lookup} + (\alpha - \delta) * t_{scan} + \delta * t_{refine}$$

Which gives us the following upper bound value on δ.

$$\delta \leq \frac{t_{lookup} + \alpha * t_{scan} - t_{interactivity}}{t_{scan} - t_{refine}} \qquad (2)$$

Consolidation Phase. Finally, we can construct a B+-tree index on top of our sorted array. The consolidation phase is the same for all algorithms, hence the same cost model applies. This results in the following cost model for the B+-tree creation and lookup:

$$n_{nodes}(n) = \begin{cases} n_{nodes}(1) = 1 \\ n_{nodes}(n) = \lceil \log_\beta(n) \rceil + n_{nodes}(\lceil \log_\beta(n) \rceil) \end{cases}$$

$$t_{bp\_creation} = n_{nodes}(n) * \tau + n_{nodes}(n) * \beta * (\kappa + \phi)$$
$$t_{bp\_lookup} = 2 * (1 + \lceil \log_\beta(n) \rceil) * \phi + \frac{\alpha}{\gamma} * \omega$$

3.2 Progressive Radixsort (MSD)

The main idea of Progressive Radixsort Most Significant Digits (MSD) is to perform a progressive Radix (MSD) clustering

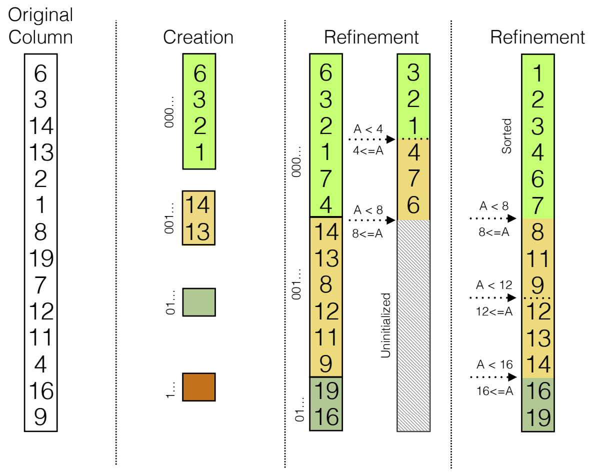


Figure 5: Progressive Radixsort (MSD).

during the creation phase. Compared to Progressive Cracking, this results in an initial "crude" index after the creation phase that consists of more than two partitions with disjoint value ranges, and thus a more fine-grained "bootstrap" for the further index refinement, which is then performed similarly as with Progressive Cracking. While inspired by "Hybrid Radix Crack" [17], Progressive Radixsort (MSD) is specifically designed to perform progressively. Figure 5 depicts a snapshot of the creation phase and two snapshots of the refinement phase. In the following, we discuss these two phases in detail. The consolidation phase is the same as with Progressive Cracking.

Creation Phase. In the first iteration, we start by allocating b empty buckets. Then, while scanning the original column, we place $N * \delta$ elements into the buckets based on their most significant $\log_2 b$ bits. We then scan the remaining $1 - \rho$ fraction of the base column. In subsequent iterations, we scan the $[0, b[$ buckets that could potentially contain elements matching the query predicate to answer the query in addition to scanning the remainder of the base column.

Bucket Count. Radix clustering performs a random memory access pattern that randomly writes in b output buckets. To avoid excessive cache- and TLB-misses, assuming that each bucket is at least of the size of a memory page, the number b of buckets, and thus the number of randomly accessed memory pages should not exceed the number of cache lines and TLB entries, whichever is smaller [2]. Since our machine has 512 L1 cache lines and 64 TLB entries, we use $b = 64$ buckets.

Bucket Layout. To avoid having to allocate large regions of sequential data for every bucket, the buckets are implemented as a linked list of blocks of memory that each hold up to s_b elements. When a block is filled, another block is added to the list and elements will be written to that block. This adds some overhead over sequential reads/writes as every s_b elements there will be a memory allocation and random access, and for every element that is added the bounds of the current block have to be checked.

The total time taken is the sum of (1) the scan time, (2) the index lookup time and (3) the time it takes to add elements to buckets. As we determine which bucket an element belongs to only based on the most significant bits, finding the relevant bucket for an element can be done using a single bitshift. As we are choosing the bucket count such that all bucket regions can fit in cache, writing elements to buckets is equivalent to sequentially writing them unless a bucket entry is filled, in which case we have to perform a memory allocation. Scanning the buckets for the already indexed data has equivalent performance to performing a sequential scan plus the random accesses we need to perform every s_b elements.

This results in the following cost for the initial indexing process:

$$t_{bscan} = t_{scan} + \phi * \frac{N}{s_b}$$
$$t_{bucket} = (\kappa + \omega) * \frac{N}{\gamma} + \tau * \frac{N}{s_b}$$
$$t_{total} = (1 - \rho - \delta) * t_{scan} + \alpha * t_{bscan} + \delta * t_{bucket}$$

Which gives us the following bound on δ.

$$\delta \leq \frac{(1-\rho) * t_{scan} + \alpha * t_{bscan} - t_{interactivity}}{t_{scan} - t_{bucket}} \tag{3}$$

Refinement Phase. In the refinement phase, all elements in the original column have been appended to the buckets. We then merge the buckets into a single sorted array. As the buckets themselves are ordered (i.e., for two buckets b_i and b_{i+1}, we know $e_i < e_{i+1} \forall e_i \in b_i, e_{i+1} \in b_{i+1}$), we can refine each bucket independently without considering the elements in other buckets in order to create a fully sorted list. For the sorting of the individual buckets into the final sorted list, we use Progressive Cracking. Using a progressive algorithm to sort individual buckets protects us from performance spikes caused by sorting large buckets.

The buckets are merged into the final sorted index in order, as such, we always have at most a single iteration of Progressive Cracking active at a time in which we are performing swaps.

For the elements that match the query, we either (1) need to scan the buckets for buckets that have not been merged, or (2) need to perform a binary search and scan on the sorted list. We define α as a fraction of elements that we have to scan in the buckets and β as the fraction of elements that we have to scan in the final index. As the number of merged buckets increases, fewer buckets will have to be scanned, i.e., α will increase, and β will increase. After all the buckets have been merged and sorted into the final index, we have a single fully sorted array from which we can construct our B+-tree index.

$$t_{bsearch} = \log_2(\rho * N)$$
$$t_{total} = \alpha * t_{bscan} + t_{bsearch} + \beta * t_{scan} + \delta * t_{refine}$$

Which gives us the following bound on δ.

$$\delta \leq \frac{\alpha * t_{bscan} + t_{bsearch} + \beta * t_{scan} - t_{interactivity}}{-t_{refine}} \tag{4}$$

3.3 Progressive Bucketsort

Progressive Bucketsort (Equi-Height) is very similar to Progressive Radixsort (MSD). The main difference is in the way the initial partitions (buckets) are determined. Instead of radix clustering, which is fast but yields equally sized partitions only with uniform data distributions, we perform a value-based range partitioning to yield equally sized partitions also with skewed data, at the expense that determining the bucket that a value belongs to is more expensive. Figure 6 depicts a snapshot of the creation phase and two snapshots of the refinement phase. In the following, we discuss these two phases in detail. The consolidation phase is the same as with Progressive Cracking and Progressive Radixsort (MSD).

Creation Phase. Progressive Bucketsort operates in a very similar way to Progressive Radixsort (MSD). Instead of choosing the bucket an element belongs to based only on the most significant bits, the bucket is chosen based on a set of

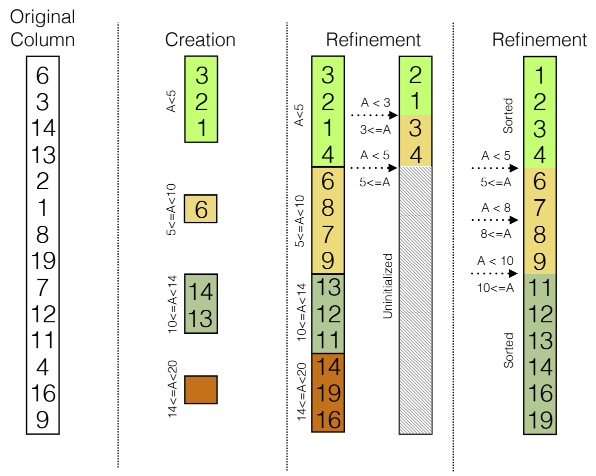


Figure 6: Progressive Bucket Sort

bounds that more-or-less evenly divide the elements of the set into the separate buckets. These bounds can be obtained either in the scan to answer the first query or from existing statistics in the database (e.g., a histogram).

In the creation phase, the cost of the algorithm is identical to that of Progressive Radixsort (MSD) except that determining which element a bucket belongs to now requires us to perform a binary search on the bucket boundaries, costing an additional $\log_2 b$ time per element we bucket. This results in the following cost for the initial indexing process:

$$t_{total} = (1 - \rho - \delta) * t_{scan} + \alpha * t_{bscan} + \delta * \log_2 b * t_{bucket}$$

Which gives us the following bound on δ.

$$\delta \leq \frac{(1 - \rho) * t_{scan} + \alpha * t_{bscan} - t_{interactivity}}{t_{scan} - \log_2 b * t_{bucket}} \quad (5)$$

Refinement Phase. The refinement phase of Progressive Bucketsort is equivalent to that of Progressive Radixsort (MSD). The only difference is that the distribution of elements inside the buckets is robust against skewed patterns. When dividing elements into buckets based only on the most significant bits, many elements can fall into the same bucket when dealing with skewed data distributions, which results in poor intermediate index performance.

3.4 Progressive Radixsort (LSD)

Progressive Radixsort Least Significant Digits (LSD) performs a progressive radix clustering on the least significant bits during the creation phase. Given that this does not result in a range partitioning, as with radix cluster (MSD), we cannot perform Progressive Cracking in the individual buckets to refine them in-place. Instead, we perform out-of-place radix (LSD) clustering also during the refinement phase, to achieve a "pure" Radixsort (LSD) in a progressive manner. Figure 7 depicts a snapshot of the creation phase and two snapshots of the refinement phase. In the following, we discuss these two phases in detail. The consolidation phase is the same as above.

Creation Phase. The creation phase of this algorithm is similar to the creation phase of Progressive Radixsort (MSD) except that we partition elements based on the least-significant bits instead of the most-significant bits. We can use the buckets that are created to speed up point queries because we only need to scan the bucket in which the query value falls. However, unlike the buckets created for the Progressive Radixsort (MSD) and Progressive Bucketsort, these intermediate buckets cannot be used to speed up range queries in many situations. Because the elements are inserted based on their least-significant bits, the buckets do not form a value-based range-partitioning of the data. Consequently, we will have to scan many buckets, depending on the domain covered by the range query.

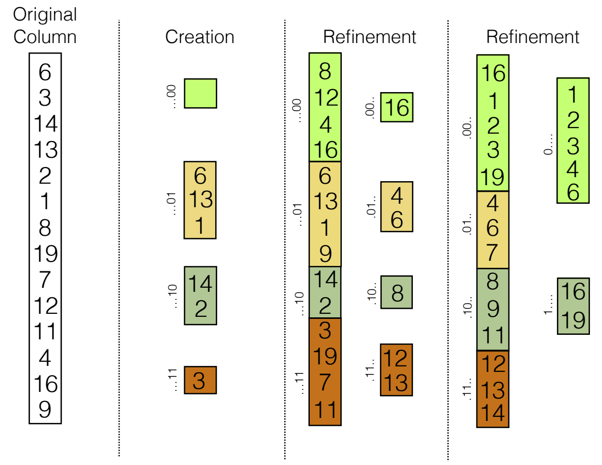


Figure 7: Progressive Radixsort (LSD).

The cost model for the Progressive Radixsort (LSD) is also equivalent to the cost model of the Progressive Radixsort (MSD), except the value of α is likely to be higher for range queries (depending on the query predicates) as the elements that answer the query predicate are spread in more buckets. As scanning the buckets is slower than scanning the original column, we also have a fallback that when $\alpha == \rho$ we scan the original column instead of using the buckets to answer the query.

Refinement Phase. In the refinement phase, we move elements from the current set of buckets to a new set of buckets based on the next set of significant bits. We repeat this process until the entire column is sorted.

In this phase, we scan α fraction of the original buckets to answer the query and move δ fraction of the elements into the new set of buckets. This results in the following cost for the refinement process.

$$t_{total} = \alpha * t_{bscan} + \delta * t_{bucket}$$

Which gives us the following bound on δ.

$$\delta \leq \frac{\alpha * t_{bscan} - t_{interactivity}}{-t_{bucket}} \quad (6)$$

3.5 Handling Appends

We use a Progressive Mergesort strategy to update our index. Appends are added to an extra vector that is separate from the base column. When a query is executed the append vector is fully scanned to answer the query. When the append vector

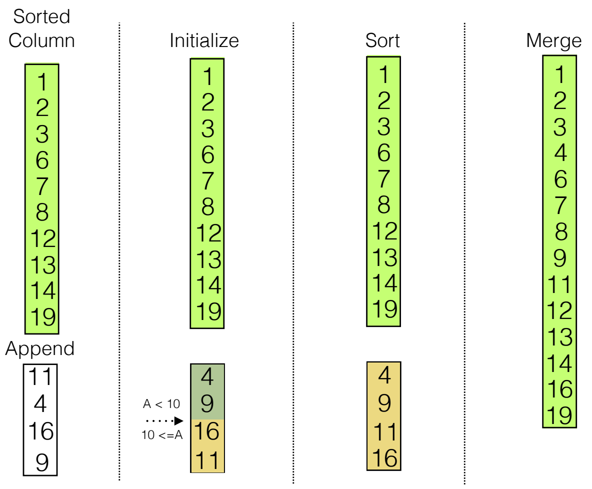


Figure 8: Handling appends & merging append vector.

grows past a certain threshold, we move the append vector to a separate list of to–be–merged vectors and create a new empty append vector. We progressively sort the vectors in the to–be–merged list using a fraction of the sorting budget. When the base vector and vectors in the to–be–merged list are sorted, the B+-Tree nodes are removed, and the lists are merged using a cascading two-way merge. Note that we do not necessarily complete a full merge in one query, as this would result in large drops in performance when we merge two large chunks. Instead, we merge at most $N * 2\delta$ elements and keep track of how far along the merge we are. After a merge is completed, we replace the two original chunks with the merged chunk. Figure 8 depicts the manner in which the append vectors are merged with the base columns.

4 EXPERIMENTAL EVALUATION

In this section, we provide an evaluation of the proposed progressive indexing methods and the performance characteristics they exhibit. In addition, we provide an in-depth comparison of the performance of the proposed methods with existing (adaptive) indexing methods.

For each of the techniques, we are interested in measuring the following properties:

(1) The robustness of the algorithm against different query workloads.
(2) The performance penalty applied to the first query.
(3) The number of queries required before the indexing technique becomes faster than only performing full scans (i.e., indexing pay-off).
(4) How fast the indexing technique converges towards a full index.

Setup. We implemented all our progressive indexing algorithms in a stand-alone program written in C++ and compiled with GNU g++ version 7.2.1 using optimization level `-O3`. All experiments were conducted on a machine equipped with 16 GB main memory and an 8-core Intel Core i7-2600K CPU @ 3.40 GHz with 8192 KB L3 cache.

We use an 8-byte integer array with 10^8 uniformly distributed values as our dataset. All queries are of the form: `SELECT SUM(R.A) FROM R WHERE R.A BETWEEN` V_1 `AND` V_2.

To test how different workload patterns influence the different algorithms, we use the following four different canonical workload patterns. Note that, for all workloads, given V_1 as determined below, V_2 is chosen such that each query has the specified workload selectivity.

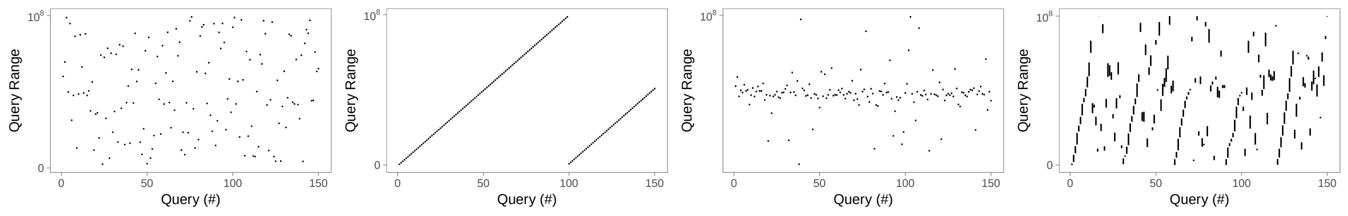
- **Random Workload** consists of a set of queries where V_1 is chosen completely randomly over the domain (See Figure 9a).
- **Sequential Workload** consists of a set of queries where V_1 is chosen such that subsequent range queries have significant overlap (See Figure 9b).
- **Skewed Workload** consists of a set of queries where V_1 is chosen according to a Zipfian distribution (See Figure 9c).
- **Mixed Workload** consists of a set of queries where the pattern changes between random, sequential and skewed every 10 queries. In addition, the query selectivity randomly varies within [1,10]% (See Figure 9d).

With the exception of the mixed workload, for each generated workload, the selectivity of all queries within the workload is identical. Also, for all experiments, unless stated otherwise, the workload size is 10,000 queries, and the query selectivity is 1%. We repeat the entire workload 10 times and take the median runtime of each query as the reported time.

4.1 Adaptive Indexing Methods

For our performance assessment, we choose a two-stage approach. First, we evaluate several state–of–the–art adaptive indexing methods. In case this allows us to identify a single overall best-performing method, we can largely simplify the presentation of the performance comparison of our progressive indexing methods with state–of–the–art adaptive indexing methods in the subsequent sections.

Here, we test Standard Cracking, (Progressive) Stochastic Cracking and Coarse Granular Index. We also compare with a B+-tree implementation, as a Full Index, that immediately constructs the entire index upon receiving the first query. In addition, we measure how long it takes to answer all the queries using only Full Scans. We compare the performance penalty on the first query, the number of queries it takes for the indexing to pay-off, and the total query response time. We do not evaluate convergence since these algorithms only fully converge to an ordered index when all values from the



(a) Random. (b) Sequential. (c) Skewed. (d) Mixed.

Figure 9: Workload Patterns.

Index	1^{st} Query (s)	Pay-off	Total Time (s)
P. Stc 1%	0.19	198	77
P. Stc 10%	**0.30**	**50**	**62**
Cracking	0.82	49	116
Stochastic	0.95	54	62
Coarse	1.50	32	114
Index	7.22	72	7.24

Table 2: Adaptive Indexing Summary (less is better)

column have been issued as query predicates. In this experiment, a mixed workload is used due to the unpredictability in its pattern and selectivity.

The implementations for the Full Index, Standard Cracking, Stochastic Cracking, and Coarse Granular Index were inspired from the work done in Schuhknecht et al. [25][2]. The implementation for Progressive Stochastic Cracking was inspired from the work done in Halim et al. [12][3]. The Full Scan implementation uses predication to avoid branching.

First Query. Table 2 depicts a summary of the results. The Full Index has the highest penalty on the first query due to the a-priori index creation. It is followed by Coarse Granular Index that is penalized due to the creation of multiple balanced initial partitions. Stochastic is slightly more expensive than Cracking because it performs an extra random crack on the first query. Progressive Stochastic incurs the lowest initial query cost due to its limits in the number of swaps per query.

Pay-Off. The Coarse Granular Index needs the lowest number of queries to pay-off (See Table 2) due to the creation of more balanced partitions. Its later followed by Standard Cracking, Progressive Stochastic 10% and Stochastic Cracking that take about the same number of initial queries to return the initial indexing investment. Finally, Progressive Stochastic 1% presents the highest pay-off due to its low number of swaps on the initial queries, taking longer to build an index that is useful in speeding up subsequent queries.

[2]https://infosys.uni-saarland.de/publications/uncracked_pieces_sourcecode.zip
[3]https://github.com/felix-halim/scrack

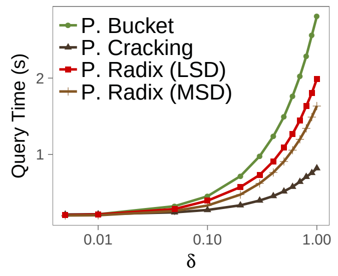
Total Time. Table 2 also depicts the cumulative cost of the entire workload for all algorithms. The Full Index has the lowest total time due to the a-priori index creation. From the adaptive algorithms, the stochastic approaches were the fastest ones due to their partition strategy, being the most robust strategies. Standard Cracking and Coarse Granular Index suffer more from the unpredictable workload since they always index the exact query predicates.

Based on this assessment of state–of–the-art adaptive indexing methods, we decide to use Progressive Stochastic Cracking 10% (second row in Table 2, highlighted in boldface) as reference to compare our progressive indexing methods to throughout the remainder of the benchmarks. Progressive Stochastic Cracking 10% has the lowest total cost for the entire workload of all tested adaptive indexing methods, indicating its robust performance. Moreover, it incurs a considerably smaller penalty on the initial queries than Standard Cracking, Stochastic Cracking, and Coarse Granular Index. Finally, it yields significantly earlier indexing pay-off than Progressive Stochastic 1%.

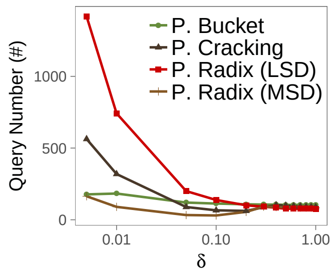
4.2 Delta Parameter Impact

In Section 3, we described four different progressive indexing techniques. Each of these techniques functions by indexing a small fraction of the data each step until the algorithm has converged towards a Full Index. The amount of data that is indexed each step depends on the δ parameter. Hence this parameter is crucial to the performance characteristics shown by these algorithms. For $\delta = 0$, no indexing is performed, meaning that algorithms resort to merely performing Full Scans on the data, never converging to a Full Index. For $\delta = 1$, the entire creation phase will be completed immediately during the first query execution. Between these two extremes, we are interested in seeing how exactly different values for the δ parameter influence the performance characteristics of the different algorithms. For the experiments in this section, we execute a mixed workload.

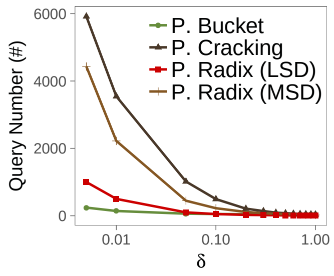
First Query. Figure 10a shows the performance of the first query for varying values of δ. As expected, the performance of the first query degrades as δ increases since each



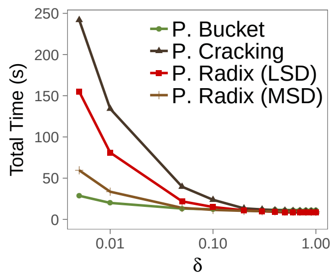
(a) First query.



(b) Pay-off.



(c) Convergence.



(d) Total time.

Figure 10: Progressive indexing experiments with varying deltas (X-axes in logarithmic scale).

query does extra work proportional to δ. For every algorithm, however, the amount of extra work done differs.

We can see that Bucketsort is impacted the most by increasing δ. This is because determining which bucket an element falls into costs $O(log\, b)$ time, followed by a random write for inserting the element in the bucket. Radixsort, despite its similar nature to Bucketsort, is impacted much less heavily by an increased δ. This is because determining which bucket an element falls into costs constant $O(1)$ time. Cracking experiences the lowest impact from an increasing δ, as elements are always written to only two memory locations (the top and bottom of the array), the extra sequential writes are not very expensive.

Pay-Off. Figure 10b shows for varying values of δ the number of queries required until the progressive indexing technique becomes worth the investment. We observe that with a very small δ, it takes many queries until the indexing pays off. While a small δ ensures low first query costs, it significantly limits the progress of index-creation, -refinement and -consolidation per query, and consequently the speed-up of query processing. With increasing δ, the number of queries required til pay-off quickly drops to a stable level. This non-linear trade-off between indexing investment and indexing benefit enables us to identify an optimal δ for early pay-off, i.e., the smallest δ that yields the minimal number of queries required til pay-off.

Convergence. An obvious effect from increasing the δ parameter is the convergence speed of the progressive index towards a full index. When $\delta = 0$ the index will never converge, and a higher value for δ will cause the index to converge faster as more work is done per query on building the index.

Figure 10c shows the number of queries required until the index converges towards a Full Index. We see that Bucketsort converges quickly, even with a low δ. This is followed by Radixsort (LSD), Radixsort (MSD) and then Cracking.

A design-feature of all variants of our progressing indexing technique, with the exception of Cracking, is that the convergence speed towards a full index is independent of the workload. We achieve this by having each query index a fixed δ fraction of the data, regardless the query predicate. With $\delta > 0$, convergence to a full index will finish within a finite number of queries. This is in contrast to adaptive indexing techniques, where the convergence to a full index inherently depends on the actual query workload, and an eventual full index is not guaranteed.

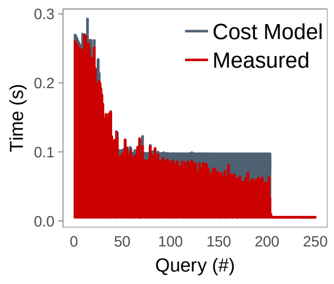
Cumulative Time. As we have seen before, a high value for δ means that more time is spent on constructing the index, meaning that the index converges towards a Full Index faster. While earlier queries will take longer with a higher value of δ, subsequent queries will take less time. Another interesting measurement, therefore, is the cumulative time spent on answering a large number of queries. Does the increased investment in index creation earlier on pay off in the long run?

Figure 10d depicts the cumulative query cost. We can see that a higher value of δ leads to a lower cumulative time. Converging towards a Full Index requires the same amount of time spent on constructing the index, regardless of the value of δ. However, when δ is higher, that work is spent earlier on (during fewer queries), and queries can benefit from the constructed index earlier.

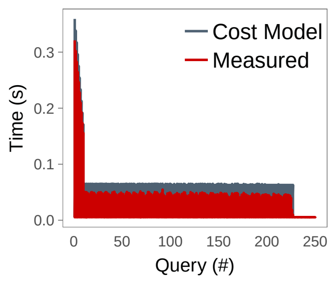
Cracking and Radixsort (LSD) perform poorly when the delta is low. For cracking, this is because it will take many queries to finish our pivoting in one element. While in Radixsort (LSD) the intermediate index that is created cannot be effectively used to answer range queries before it fully converges, meaning a long time until convergence results in poor cumulative time.

Another observation here is that the cumulative time converges rather quickly with an increasing delta. The cumulative time with $\delta = 0.25$ and $\delta = 1$ are almost identical for all algorithms, while the penalization of the initial query continues to increase significantly (recall Figure 10a).

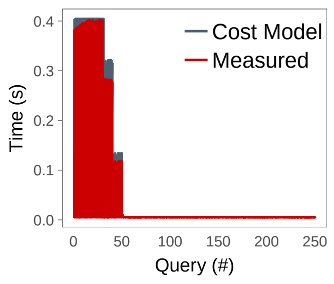
Choosing δ. In the end, the best value of δ depends on the expected use case of the database. If columns are expected



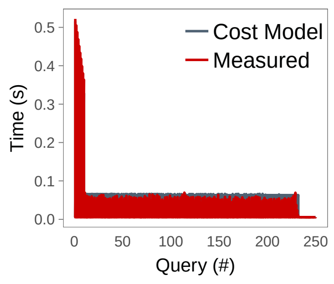
(a) P. Cracking.



(b) P. Radixsort (MSD).



(c) P. Radixsort (LSD).



(d) P. Bucketsort.

Figure 11: Cost Model vs Measured Cost.

to be used only a few times, a low value of δ makes sense so early queries are not penalized heavily. When many queries are performed and the increased time of initial queries is not an issue, a higher value for δ might be preferable for faster index convergence.

However, we do note that increasing δ past 0.25 does not seem to have a big effect on either the cumulative time spent on answering queries or the time until pay-off, regardless of the progressive indexing technique used. Hence, selecting a δ higher than 0.25 does not seem reasonable.

Cost Model. In the previous section, we presented a cost model for each of the progressive indexing approaches, to let the system automatically pick a delta, taking the interactivity threshold time in consideration.

Figure 11 depicts a comparison of our cost model and the real measured cost. The data and the workload follow a uniform random distribution, and we execute 250 queries. Our cost model calculates the maximum number of swaps for each indexing step and uses it as an upper bound. We can see that our cost model accurately gives a tight upper bound of the query costs with the real cost rarely surpassing it.

In Figure 11a we can see that the cost model for Progressive Cracking, when compared to the real cost, has a considerable variance, this is because of our model always considering the maximum number of possible swaps happening. Similarly, the same happens for the merge phases of Progressive Radixsort (MSD) (See Figure 11b) and Progressive Bucketsort (See Figure 11d).

4.3 Performance Comparison

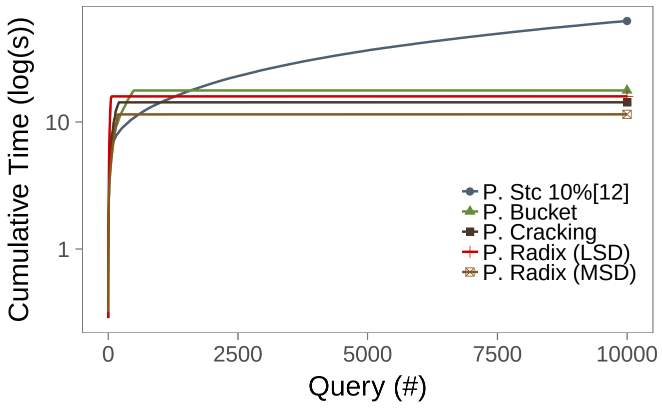
In this section we run experiments comparing progressive indexing against Progressive Stochastic Cracking 10%, we have chosen δ such that the first query is as slow as the first query of Progressive Stochastic 10% for each of the progressive indexing methods, for the fairness of comparison. We run experiments with various workload patterns and data distributions. Also, we also experiment with workloads with only point queries and workloads that interleave read-only queries and appends.

4.3.1 Mixed Workload.

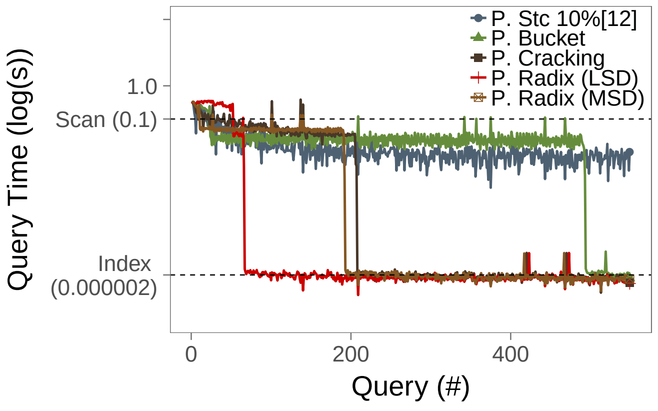
Cumulative Time. Figure 12a depicts the cumulative time required to answer the entire workload for each of the indexing methods. We can see that there is a relation between full index convergence and lower cumulative time. The algorithms that perform more work on index construction early on leads to a faster cumulative time, as more of the issued queries can benefit from the constructed index, while Progressive Stochastic 10% suffers from never achieving full convergence. We can see that Progressive Radixsort (LSD) converges fast, but the fact that it performs so poorly before convergence leads to a high cumulative time. Progressive Bucketsort performs poorly prior to convergence since the construction of the index is expensive due to the $O(logb)$ insertions. Progressive Cracking pays a penalty due to its workload dependence being affected by the unpredictability of the mixed workload. Finally, Radixsort (MSD) presents the lowest cumulative time for mixed workloads, since its intermediate results can be used to answer range queries, it has a low insertion cost and produces initial balanced partitions indifferent of the query workload.

Performance Over Time. Figure 12b depicts the time required to answer individual queries as the index is refined. Progressive Stochastic 10% exhibit very volatile performance. Sudden order of magnitude performance jumps are not uncommon. This is mainly due to two reasons. First, the adaptive methods refine the index depending on the bounds used in the query predicate, which can result in either a large refinement or no refinement at all depending on the predicates and the state of the current index. Second, since the adaptive indexing techniques never converge towards a full index but keep on using the (potentially unbalanced) cracker pieces, the performance keeps varying significantly.

All progressive indexing methods exhibit a much more monotonous decrease in query response time than Progressive Cracking 10% until each of them has converged to a



(a) Cumulative cost.



(b) Performance Over Time.

Figure 12: Mixed Workload.

Indexing Method	Mix Qry	Rnd Qry	Seq Qry	Skw Qry	Pnt Qry	Skw Data
P. Stc 10% [12]	62.42	21.24	20.63	18.23	8.67	21.97
P. Bucket	17.71	14.21	14.89	13.04	13.22	11.73
P. Cracking	14.29	10.82	12.17	9.79	10.91	10.97
P. Rdx(LSD)	15.91	12.64	13.77	12.10	7.22	61.76
P. Rdx(MSD)	11.48	10.12	10.37	9.85	9.83	12.42

Table 3: Total time for various workloads (s).

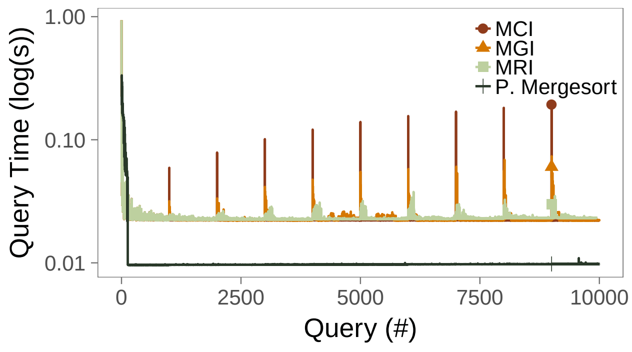


Figure 13: Progressive vs Adaptive Updates.

Full Index, after which they require a low constant time to answer new queries with its variance depending mostly on the query selectivity.

Progressive Cracking exhibits the most volatile performance among the progressive indexing methods while the index is being constructed. This is because the query adaptivity suffers from the erratic workload. Progressive Radixsort (LSD) also has spikes in performance while the index is being constructed. This is because how effectively the partial radix index can be used to speed up queries depends on the query. We have to scan one bucket in the best case or all buckets in the worst case to answer the query, leading to volatile performance before convergence.

Progressive Radixsort (MSD) and Progressive Bucketsort, on the other hand, show a very stable decrease in performance over the number of queries. Since the elements are partitioned relatively equally over the buckets, meaning there is never a large bucket that needs to be scanned entirely to answer a query.

4.3.2 Other Workloads & Data Distributions.
Cumulative Time. Table 3 depicts the cumulative time required to answer different experiments for each of the indexing methods. We execute random, sequential and skewed workloads, a workload with randomly selected point queries and one with a skewed distribution and a skewed workload. The skewed data for the column is generated by selecting 80% of the elements as the nearest ones to the median value of the domain. The remaining values are chosen randomly. We can see that progressive Radixsort (MSD) is very efficient for uniformly distributed datasets in general. Progressive Cracking is very efficient when dealing with very skewed distributions, due to its workload dependence, even being more efficient than Bucketsort for skewed data distribution, since the latter still suffers from the high insertion costs. Finally, Progressive Radixsort (LSD) present the best performance for point queries since we only scan the bucket where the point query is located.

4.4 Appends

We present one experiment to compare Progressive Mergesort with the update strategies from adaptive indexing. We want to see if our strategy updates our index without producing many performance spikes. In this scenario, the column, workload, and appends are uniformly randomly distributed. We use 10^4 appends every 1000 queries. We use Progressive Cracking on the first 1000 queries and Progressive Mergesort after that, both with $\delta = 0.2$. Figure 13 depicts the results. MCI (Merge Complete), has one huge performance spike every 1000 queries, due to the complete merge of the appends. MGI (Merge Gradually) presents smaller but more spikes than MCI since it only effectively merges data if it is between the query predicate. MRI (Merge Ripple) presents the smallest but most frequent performance spikes on the subsequent queries since it mostly swaps the data from the append list and the index. Finally, Progressive Mergesort presents the least number of performance spikes, due to its indexing limitation, only indexing small and predictable pieces of data per query.

5 CONCLUSION

In this paper, we introduced progressive indexing, a novel adaptive indexing technique that offers robust and predictable query performance under different workloads. Progressive techniques offer a balance between fast convergence towards a full index together with a small performance penalty for the initial queries on an unindexed column. We have implemented several progressive indexing techniques and shown how they perform against existing adaptive indexing techniques for various data distributions and query patterns. We also developed cost models for all our progressive indexing techniques, to allow for an automatic delta tuning and provided a new strategy for efficient merging appends into our index. Finally, selecting which progressive algorithm to use is not straightforward. Based on the main characteristics of each algorithm and the results in our experimental evaluation, we conclude our work with the decision tree shown in Figure 14, that provides recommendations on which technique to use in different situations.

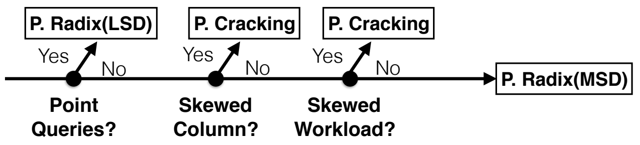


Figure 14: Progressive Indexing Decision Tree.

6 FUTURE WORK

We point out the following as the main aspects to be explored in progressive indexing future work:

- **Indexing Methods.** Other techniques can be adapted to work progressively with different benefits. For example, instead of constructing the complete hash table, we only insert $n * \delta$ elements and scan the remainder of the column. The partial hash table can be used to answer point queries on the indexed part of the data. Another example is column imprints [28] where instead of immediately building imprints for the entire column, only build them for the first fraction δ of the data.
- **Indexing Structures.** Different data structures can be used to exploit modern hardware and boost access to more selective queries. In this paper, we choose to progressively build a B+-Tree in our consolidation phase. However, other structures like the ART-tree [18] can also be built progressively, with more careful considerations on their creations costs and query performance.
- **Complex Database Operations.** Much like regular indexes, progressive indexes could also be used for other database operations such as joins and aggregations.
- **Multi-Column Indexes.** Multidimensional indexes are necessary to improve the performance of queries that involve multiple columns, and these could also be created in a progressive fashion.
- **Approximate Query Processing.** Until now, we considered that we always have time to invest in index creation during query processing. However, this is not true in scenarios where a simple full scan crosses the interactivity threshold, approximate query processing [5] is used to allow for a query response under the threshold. Building progressive indexes can speed query response and accuracy with a trade-off between less accurate queries in the beginning.
- **Interleaving Progressive Strategies.** As depicted in our decision tree, different progressive strategies can be more efficient in different scenarios. When deciding for very small deltas, the indexes can take longer to fully converge, and the workload patterns might change dramatically. Detecting these changes and changing the progressive strategy on-the-fly can be beneficial for this cases.

REFERENCES

[1] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R. Narasayya. 2000. Automated Selection of Materialized Views and Indexes in SQL Databases. In *Proceedings of the 26th International Conference on Very Large Data Bases (VLDB '00)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 496–505. http://dl.acm.org/citation.cfm?id=645926.671701

[2] Peter A. Boncz, Stefan Manegold, and Martin L. Kersten. 1999. Database Architecture Optimized for the New Bottleneck: Memory Access. In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*. 54–65. http:

//www.vldb.org/conf/1999/P5.pdf

[3] Peter A. Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR*.

[4] Nicolas Bruno. 2011. *Automated Physical Database Design and Tunning*. CRC-Press.

[5] Kaushik Chakrabarti, Minos Garofalakis, Rajeev Rastogi, and Kyuseok Shim. 2001. Approximate query processing using wavelets. *The VLDB JournalâĂŤThe International Journal on Very Large Data Bases* 10, 2-3 (2001), 199–223.

[6] Surajit Chaudhuri and Vivek Narasayya. 1998. AutoAdmin "What-if" Index Analysis Utility. *ACM SIGMOD Record* 27, 2 (1998), 367–378.

[7] Surajit Chaudhuri and Vivek R Narasayya. 1997. An Efficient, Cost-Driven Index Selection Tool for Microsoft SQL Server. In *VLDB*, Vol. 97. 146–155.

[8] Douglas Comer. 1978. The Difficulty of Optimum Index Selection. *ACM Transactions on Database Systems (TODS)* 3, 4 (1978), 440–445.

[9] Goetz Graefe and Harumi Kuno. 2010. Self-selecting, self-tuning, incrementally optimized indexes. In *Proceedings of the 13th International Conference on Extending Database Technology*. ACM, 371–381.

[10] Himanshu Gupta, Venky Harinarayan, Anand Rajaraman, and Jeffrey D Ullman. 1997. Index Selection for OLAP. In *Data Engineering, 1997. Proceedings. 13th International Conference on*. IEEE, 208–219.

[11] Immanuel Haffner, Felix Martin Schuhknecht, and Jens Dittrich. 2018. An Analysis and Comparison of Database Cracking Kernels. In *Proceedings of the 14th International Workshop on Data Management on New Hardware (DAMON '18)*. ACM, New York, NY, USA, Article 10, 10 pages. https://doi.org/10.1145/3211922.3211930

[12] Felix Halim, Stratos Idreos, Panagiotis Karras, and Roland HC Yap. 2012. Stochastic Database Cracking: Towards Robust Adaptive Indexing in Main-Memory Column-Stores. *VLDB* 5, 6 (2012), 502–513.

[13] Pedro Holanda and Eduardo Cunha de Almeida. 2017. SPST-Index: A Self-Pruning Splay Tree Index for Caching Database Cracking. In *EDBT*. 458–461.

[14] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. 2007. Updating a Cracked Database. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data (SIGMOD '07)*. ACM, New York, NY, USA, 413–424. https://doi.org/10.1145/1247480.1247527

[15] Stratos Idreos, Martin L Kersten, and Stefan Manegold. 2009. Self-organizing Tuple Reconstruction in Column-stores. *SIGMOD* (2009), 297–308.

[16] Stratos Idreos, Martin L Kersten, Stefan Manegold, et al. 2007. Database Cracking. In *CIDR*, Vol. 3. 1–8.

[17] Stratos Idreos, Stefan Manegold, Harumi Kuno, and Goetz Graefe. 2011. Merging What's Cracked, Cracking What's Merged: Adaptive Indexing in Main-Memory Column-Stores. *VLDB* 4, 9 (2011), 586–597.

[18] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*. IEEE, 38–49.

[19] Zhicheng Liu and Jeffrey Heer. 2014. The Effects of Interactive Latency on Exploratory Visual Analysis. 20 (12 2014), 2122–2131.

[20] Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, Jiexi Lin, Lin Ma, Prashanth Menon, Todd C Mowry, Matthew Perron, Ian Quah, et al. 2017. Self-Driving Database Management Systems. In *CIDR*.

[21] Eleni Petraki, Stratos Idreos, and Stefan Manegold. 2015. Holistic Indexing in Main-memory Column-stores. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 1153–1166.

[22] Holger Pirk, Eleni Petraki, Stratos Idreos, Stefan Manegold, and Martin Kersten. 2014. Database Cracking: Fancy Scan, not Poor Man's Sort!. In *Proceedings of the Tenth International Workshop on Data Management on New Hardware*. ACM, 4.

[23] Kenneth A. Ross. 2002. Conjunctive Selection Conditions in Main Memory. In *Proceedings of the Twenty-first ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 3-5, Madison, Wisconsin, USA*. 109–120. https://doi.org/10.1145/543613.543628

[24] Felix Martin Schuhknecht, Jens Dittrich, and Laurent Linden. 2018. Adaptive Adaptive Indexing. *ICDE* (2018).

[25] Felix Martin Schuhknecht, Alekh Jindal, and Jens Dittrich. 2013. The Uncracked Pieces in Database Cracking. *Proc. VLDB Endow.* 7, 2 (Oct. 2013), 97–108. https://doi.org/10.14778/2732228.2732229

[26] Thibault Sellam, Emmanuel MÃijller, and Martin Kersten. 2015. Semi-Automated Exploration of Data Warehouses. (10 2015), 1321–1330.

[27] Ankur Sharma, Felix Martin Schuhknecht, and Jens Dittrich. 2018. The Case for Automatic Database Administration using Deep Reinforcement Learning. *arXiv preprint arXiv:1801.05643* (2018).

[28] Lefteris Sidirourgos and Martin Kersten. 2013. Column Imprints: A Secondary Index Structure. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD '13)*. ACM, New York, NY, USA, 893–904. https://doi.org/10.1145/2463676.2465306

[29] Elvis Teixeira, Paulo Amora, and Javam C Machado. 2018. MetisIDX-From Adaptive to Predictive Data Indexing. (2018).

[30] Gary Valentin, Michael Zuliani, Daniel C Zilio, Guy Lohman, and Alan Skelley. 2000. DB2 Advisor: An Optimizer Smart Enough to Recommend Its Own Indexes. In *Data Engineering, 2000. Proceedings. 16th International Conference on*. IEEE, 101–110.