TRIMMER: Transitive Indexing in Main-Memory Systems

ABSTRACT

We study main-memory indexing tailored for modern applications that need to process massive interleaved reads and writes efficiently. State-of-the-art main-memory static indexing requires too much time and workload knowledge to create memory-tailored data structure or sort a memory-resident array for a new batch of data. *Adaptive indexing* solves this problem by avoiding high initialization costs via incremental index refinement; however, we show that state-of-the-art adaptive indexing methods lack resilience, as they fail to maintain their adaptive properties when phased with massive sequences of queries and updates; no current method can support efficient interleaved reads and writes.

We identify the root of the problem in the rigidity of indexing. We observe that a different indexing design is the best fit for each stage of a workload. Motivated by this observation, we propose a new indexing paradigm which we call *transitive indexing*. Like adaptive indexing, transitive indexing also builds an index incrementally, in response to both queries and updates. However, transitive indexing differs from and builds upon adaptive indexing by progressively adjusting not only the content of an index, but also its design. We develop a specific instantiation of TRansitive Indexing in Main Memory, TRIMMER. Initially, the index is a collection of unordered vectors. As more queries arrive, Trimmer refines the hot vectors and starts inducing order and structure. Gradually, it morphs into a trie for the hot part of the data. In effect, Trimmer can absorb massive updates while at the same time providing fast lookup times without any initialization cost. A detailed experimental analysis on both synthetic and real-life data from the astronomy domain shows that Trimmer successfully overcomes the limitations of both adaptive and static main-memory indexing, enabling interactive data exploration under massive updates.

1. INTRODUCTION

Main memory sizes have grown to the degree that all data of a database instance can be memory-resident. This growth has led to increased attention on main-memory tailored techniques. Even though main memory allows for fast data access, proper indexing still provides significant query processing benefits. Such main-memory environments also allow for direct, interactive, data exploration in conditions of *data deluge* [2]. In such a setting, continuously arriving data can be kept in main memory, while a scientist or analyst poses queries in an interactive mode [6, 1, 11, 15].

The Problem: Read-Write Intensive Workloads. Past research has shown that main-memory-tailored indexing techniques can significantly outperform standard techniques such as binary search over a dense sorted array [13, 12]. However, even such solutions cannot efficiently support massive interleaved queries and updates. This problem is increasingly addressed at various levels; at the database architecture level with efforts such as SAP HANA [3] and Hyper [10] and at the index level with efforts such as ART main-memory indexing [12]. Still, such approaches employ static indexes, hence require an expensive in-advance indexing phase based on a priori knowledge of which data should be indexed. Real-time applications with ad-hoc query demands cannot afford such a priori indexing; they require quick access to new data. This need has been addressed by recently proposed adaptive indexing approaches [7, 14]. Adaptive indexing builds and refines an index incrementally during query processing, treating each query as an advice of how data should be stored. However, while adaptive indexing techniques provide significant improvements for ad-hoc workloads, alleviating initialization costs, they are not resilient as non-adaptive indexing is for read-write intensive workloads.

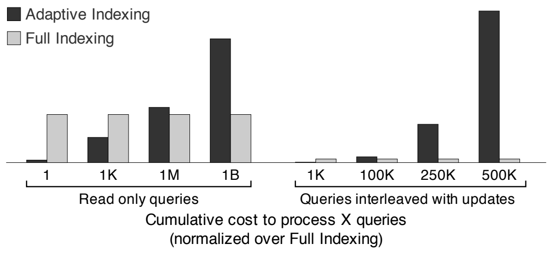

Figure 1: The need for transitive indexing.

Motivating Example. Figure 1 illustrates this problem: it depicts the relative time needed (y-axis) to answer the first X queries (x-axis) in a long sequence of random range queries and updates over a single column of 10^8 tuples (10 updates arrive after every 10 read queries). Each graph is normalized (independently) over static indexing costs. In this case, for static indexing we use the state-of-the-art in-memory index ART [12] while for adaptive indexing we use the state-of-the-art stochastic cracking [5]. As Figure 1 shows, while early in the query sequence adaptive indexing significantly outperforms static indexing, as the sequence evolves, adaptive indexing loses its advantage: it is not resilient. The first query is more than 10 times faster with adaptive indexing; full indexing needs to fully index the column, while adaptive indexing performs only small index refinement actions er query. However, as more queries and updates arrive, adaptive indexing eventually becomes more than 10 times slower than static indexing. Ideally, we would like to acquire the performance of adaptive indexing early in the query sequence and that of static indexing later on. In this paper we propose an indexing mechanism that achieves this objective.

Contributions: Transitive Indexing. Our contributions are outlined as follows.

- We propose a new main-memory indexing paradigm, *transitive indexing*, whereby not only the contents, but also the *design* of an index data structure is dynamic; it morphs as the workload evolves.
- We present TRIMMER, a particular instantiation of transitive indexing which requires no initialization. Data is initially simply appended in the form of vectors with no order forced inside each vector or across vectors. As the workload evolves, Trimmer starts forcing order and structure. Each vector is independently indexed and refined, while data flows from vector to vector. Gradually, hot data start morphing into a trie structure optimized for main memory, while cold data remain in vector form; all this happens during query processing without any need for any administration or control.
- We show that Trimmer maintains all desirable properties of adaptive indexing: it is lightweight, continuously adapts by incrementally refining indexes, and requires zero workload knowledge and preparation costs. At the same time, it is resilient: it maintains its performance advantages unfettered even under long strings of queries and continuous data updates; hence, it is geared towards continuous on-the-fly data exploration under massive read and write operations.

We experimentally demonstrate the advantages of Trimmer over existing static and adaptive indexing techniques. We use both synthetic benchmarks as well as real-world data and queries. For example, in experiments with a 4 Terabyte instance of data and queries from the Sloan Digital Sky Survey/SkyServer from the astronomy domain, Trimmer can handle a combined workload of $15 \cdot 10^4$ queries and $5 \cdot 10^8$ updates in roughly 1 minute, while state-of-the-art adaptive indexing needed more than 16 hours and state-of-the-art non-adaptive indexing needs ???.

Paper Organization. The rest of the paper is organized as follows. Section 2 discusses background and related work. Then, Section 3 presents in detail the resilience problem of existing adaptive indexing techniques and their inability to cope with the requirements of continuous exploration, and presents a series of possible patches that deal with the problem only to a certain degree. Then, Section 4 introduces transitive indexing for full resilient and adaptive exploration, and presents our particular instantiation thereof. In Section **??**, we present a detailed experimental analysis, demonstrating the effectiveness and the advantages of Trimmer both on synthetic and on real workloads. Finally, Section **??** discusses future work and concludes the paper.

2. RELATED WORK

Important points to be made here:

2.1 Distinction form previous work

This work is distinguished from previous work on adaptive indexing for column-stores, as it proposes a general-purpose main-memory indexing paradigm. In particular, this work differs from the recent work bringing together the best of all adaptive indexing methods [9]. The work in [9] examines the issues of fast initialization and convergence to a fully-refined index state, and provides adaptive indexing designs that achieve both. In other words, it has a short-term orientation: it looks at what happens during the first few queries and tries to shorten the time to convergence as well. On the other hand, our work, while preserving these short-term concerns of initialization and convergence, also has a long-term orientation: it examines scenarios in which the index never gets the chance to settle to a steady, "fully-refined" state for any part of the data, but instead needs to be continuously revamped due to continuous queries interleaved with updates.

The hybrid techniques of [9] use several buckets with overlapping value ranges and physically move/merge data to a second collection of buckets as queries arrive. They do not attempt to bring initial buckets in order - order is only brought in the final buckets (partitions) where query-qualifying data are moved, and they keep searching the initial, unordered partitions for query ranges not covered by final partitions. On the contrary, in Trimmer, even before transiting to a trie, we perform most index refinement actions in-place and continuously maintain buckets on ordered, non-overlapping value ranges for the hot workload set. Most importantly, we introduce transitivity for the sake of long-term resilience.

This discussion may need some experimental justification.

The previous related work section can be integrated here.

3. THE PROBLEM

In this section, we motivate the need for resilience in a main-memory adaptive indexing environemnt. We first demonstrate the main advantages that adaptive indexing brings and then we highlight the core non-resilience problems that appear when dealing with long query-and-update sequences. We use the original database cracking technique to highlight these benefits and shortcomings of adaptive indexing. In Section **??**, we show that the same properties are still true for more complex adaptive indexing techniques that were introduced more recently such as stochastic cracking [5], adaptive merging [4], and hybrid adaptive indexing [9].

3.1 Adaptive Behavior

The power of database cracking is its ability to self-organize automatically and at low cost. *Automaticity* eschews the need for special decision-making as to when self-organizing actions should be performed; the index self-organizes *continuously* by default. *Low cost* sets cracking apart from approaches such as online indexing, as indexing is efficiently integrated with query processing.

Cracking Continuous Adaptation. As we saw in the example of Figure **??**, cracking uses selection predicates to drive the way data is stored. After each query, data is clustered in a way such that the qualifying values for are in a contiguous area in the attribute column. The more queries are processed, the more structure is introduced.

Cracking Cost. Let us now discuss the cost of cracking, which includes the cost for identifying what kind of physical reorganizations are needed and performing them. An index shows which piece of the array holds which value range, in a tree structure; original cracking uses AVL-trees [7]. Then the cost of a query is the cost to search the tree in order to determine the portion of the array which needs to be cracked plus the cost to perform the actual data reorganization. In Figure **??** $Q1$ needs to analyze all tuples in the column in order to achieve the initial clustering, as there is no prior knowledge about the structure of the data. The second query, $Q2$, can exploit the knowledge gained by $Q1$ and avoid touching part of the data. With $Q1$ having already clustered the data into three pieces, $Q2$ touches only two of those, namely the first and third

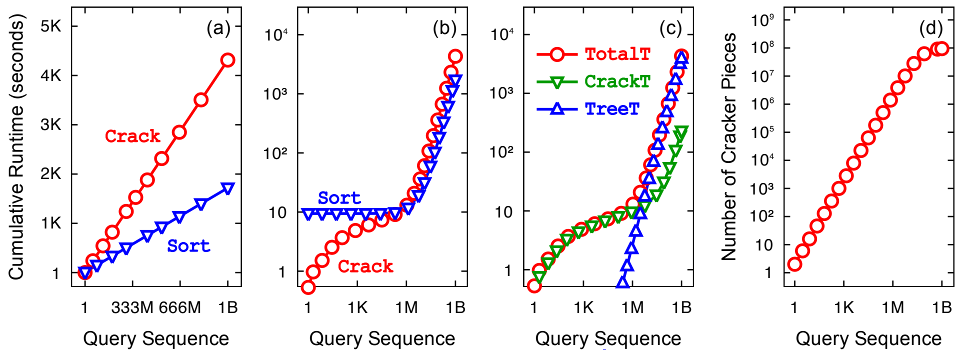

Figure 2: Non-resilience during long query sequences.

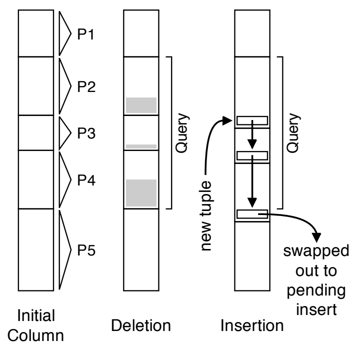

Figure 3: Database cracking updates.

piece. That is because the second piece created by $Q1$ already qualifies for $Q2$ as well. Generalizing the above analysis, we infer that, assuming such range queries, a query will analyze as most two *end* pieces, i.e., the ones intersecting with the query's requested value range boundaries. As more pieces are created by every query that does not find an exact match, pieces become smaller.

Basic Cracking Performance. Figure 4(a) shows a performance example where cracking is compared against a full indexing approach (Sort), in which we completely sort an array with the first query. In an in-memory environment, sorting a column creates the perfect index which allows for very fast data access using binary search over fixed-width and dense arrays. The data for this experiment consists of 10^8 tuples of unique integers in $[0, 2^{31})$. The query workload is random – the bounds are random while the ranges requested have a fixed selectivity of 1% per query. This scenario assumes a dynamic environment where there is no workload knowledge or idle time in order to pre-sort the data.

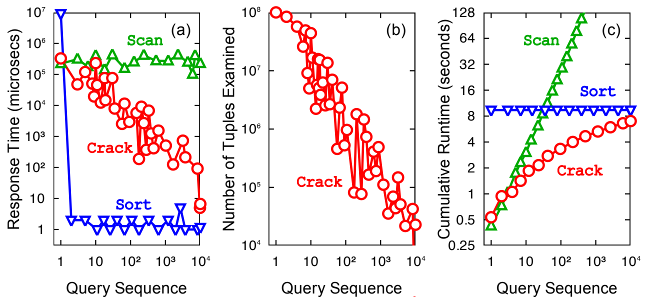

Figure 4: Adaptive behavior benefits of adaptive indexing.

As Figure 4(a) shows, once the data is sorted with the first query, from then on performance is extremely fast as we only need to perform a binary search to satisfy each request. Nevertheless, we overload the first query. On the other hand, cracking continuously improves performance without penalizing individual queries, eventually reaching the levels of Sort. Figure 4(b) shows the number of tuples each cracking query touches; as we process more queries, more pieces are created and each subsequent query touches less data by exploiting a more fine-grained index. We also compare against a plain Scan approach where data is always completely scanned. We observe that cracking does not significantly penalize any query more than Scan. Note that while Crack and Sort simply return a view of the (contiguous) qualifying tuples, Scan materializes a new array with the result.

Figure 4(c) shows the same results as in Figure 4(a) on the cumulative time as the query sequence evolves. Remarkably, by the time cracking has already answered all 10^4 queries, the full indexing approach still has not finished preparing the index. Thus, in dynamic environments with little idle time to invest in index preparation and with little knowledge about which columns are actually interesting for the queries, adaptive indexing has a huge advantage.

3.2 Problem 1: Long Query Sequences

Now we expose the non-resilience problem with adaptive indexing when it comes to long exploratory query sequences. We use a similar experiment as before with the same set-up, but now we fire up to 1 billion queries as opposed to only 10^4 queries.

Figure 2 shows the results, comparing cracking to full sorting. Figure 2(a) depicts the cumulative response time. It reveals that as the query sequence evolves, cracking loses its advantage. The total cost for cracking at the end of the query sequence is more than 4000 seconds, while full sorting needed less than half of that. Figure 2(b) shows the same results with a log scale for both axes. Up to roughly 1 million queries cracking enjoys an advantage; thereafter we reach a threshold where it would have been better to not use adaptive indexing considering cumulative costs.

We analyze this behavior further by breaking down the cracking costs in Figures 2(c) and (d). Figure 2(c) shows the total cracking costs (TotalT) separated into the costs of searching the tree (TreeT) and the cost of physical reorganization (CrackT). We observe that, up to the threshold of 1 million queries, total cost is dominated by the costs of reorganization. However, thereafter the cost is dominated by the cost of search. Figure 2(d) plots the number of pieces in the cracking index. As we pose more queries, more pieces are created (at most two per query), hence the tree grows and becomes more expensive to search and maintain.

3.3 Problem 2: Continuous Data Updates

We now show an even more significant source of non-resilience with adaptive indexing: long sequences where updates interleave with queries.

Cracking Updates. Let us first give a short description of how cracking performs updates [8]. Updates (inserts or deletes) are marked as pending upon arrival, while read queries on-the-fly merge in the actual array any pending updates that fall within the requested value range. A range that is never queried is never updated. However, as cracking works on top of fixed-width dense columns, it needs to ripple/shuffle values so that it can place new values in the proper pieces. Figure 3 shows an example of how tuples are moved across pieces upon insertions and deletions. Given that within each cracking piece values are not ordered, to move a full piece one position down, it suffices to move its first value to its end and change its borders, and let this action propagate to other pieces until it exits the query range. In reverse, deletes leave back holes, which are rippled at the borders of the piece where they belong. Holes are exploited to place new tuples in when possible.

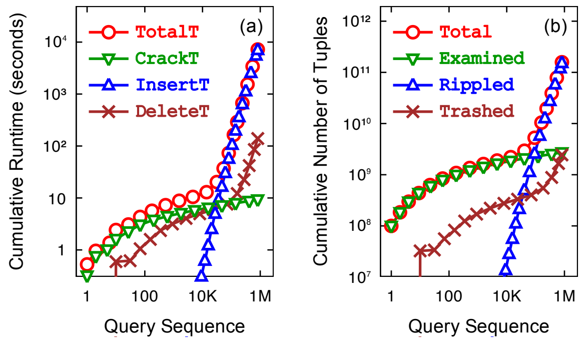

Figure 5: Non-resilience when updates interleave with queries.

The Problem. To demonstrate the non-resilience problem, we perform the same experiment as before with dynamic data: now data arrival events interleave with queries. This scenario matches modern applications where data arrive quite often, while we still want to explore the data for interesting patterns in online manner. The set-up is the same as before, with the addition that, after every 10 read queries, 10 random inserts and 10 random deletes arrive.

Figure 5 shows the results as the query sequence evolves. As Figure 5(a) shows, after about 10^4 queries the cumulative total cost (TotalT) grows significantly. Figure 5(a) also breaks this cost down to the cost for merging updates (InsertT), merging deletes (DeleteT), and reorganizing (CrackT). After 10^4 queries the cost to merge insertions becomes dominant, overshadowing the cracking costs by two orders of magnitude. Figure 5(b) depicts the amount of tuples examined as the query sequence evolves and the kind of actions performed. As we process more queries, significantly more tuples need to be touched. In particular, significantly more tuples need to be rippled, i.e., moved into a new position. After 10^4 queries, this action that dominates the total cost. As more pieces are created, more data movement is needed to maintain the index. Thus, as the sequence evolves, performance degrades; this degradation occurs much earlier than in the read-only scenario, i.e., after only 10^4 queries versus after 10^6 queries.

3.4 Patchwork Solutions

We showed in the previous section that while current adaptive indexing brings some significant advantages, when it comes to long exploratory query sequences, especially when those are interleaved with updates, then adaptive indexing loses all its performance advantages. To deal with modern exploratory applications, we need adaptive indexing techniques which are resilient to such problems and can maintain their adaptive properties in the long term. In this section, we study the space of possible solutions and we propose Comb (Cracking Over Malleable Buckets), a new adaptive indexing technique that maintains all the desirable properties of current approaches, while it is also resilient to cope with the data deluge challenges.

3.5 Target Performance

Our goal is to maintain the main properties of existing adaptive indexing techniques both *early* in a query sequence and as the query sequence *evolves*. In this way, our target performance includes the following characteristics.

- **Lightweight**. The per query cost should be kept low, i.e., individual queries should not be penalized.
- **Adaptive**. The system should be able to rapidly adjust to workload patterns.
- **Resilient**. Performance should not degrade in the long run and all adaptive properties should be maintained.

3.6 The Source of the Problem

We can attribute the non-resilience problem of existing adaptive indexing techniques to the maintenance and handling costs of the growing index structure and of the dense array data structures. As we have shown in Section 3, as query sequences evolve, the information stored in the index grows and it becomes much harder to maintain and traverse. The tree structure holding information regarding value ranges and piece boundaries grows with every query that refines the index. Thus, as the query sequence grows, the tree grows as well and its traversal leads to random memory access. In addition, in order to maintain the dense structure of columns and the partitioning information, merging of updates results in tuples being shuffled around. The more the pieces in a cracking column, i.e., the more the information in the index, the more tuple movements we have to do in order to put new values in or in order to move holes caused by deletions outside a given range.

In order to deal with the non-resilience problem, we need to contain these extra costs, i.e., allow for less expensive traversal of the tree part of the index and more efficient merging of updates.

3.7 Optimizing Existing Adaptive Indexing

One way to deal with the non-resilience problem of adaptive indexing is to patch existing adaptive indexing techniques so that they are less prone to performance degradation in long query sequences. One intuition is to restrict the growth of the index. By allowing the index to grow up to a certain size, and thus the column to be cracked up to a maximum number of pieces, then there is a maximum traversal cost as there is a maximum index depth for the tree part of the cracking index. In addition, there is a maximum number of movements we have to do in order to update a cracking piece, as there is a maximum number of pieces (there are fewer pieces and thus bigger in size). Below we introduce four directions on how to restrict the index size.

Crack-NoIndex. This first approach allows cracking to continue working as normal with the exception that if a piece in the index has reached a minimum size of P_{min} tuples, then future queries continue to crack this piece but do not inject this information in the tree part of the index. The current query is answered as normal but its refinements are not marked in the index and thus cannot be exploited by future queries nor create excessive administration overheads.

Crack-Scan. The next variation of cracking does not crack any more pieces in the column which reach the threshold of P_{min} tuples. Instead, it scans a piece that it would normally crack. As discussed in Section 3, a cracking range-select operator needs to touch at most two pieces at the edges of its target range. Thus, these extra scan actions add only a limited cost. Contrary to the Crack-NoIndex approach, Crack-Scan cannot return a view of the whole result and needs to materialize the qualifying values of the edge pieces.

Crack-Sort. The third alternative aims towards maximum read performance. When the piece size limit is reached, then a future query sorts completely a piece which should have been normally cracked. The information that this piece is sorted is maintained in the tree index (an extra bit) and future queries may simply binary search when their boundaries fall within this piece.

Crack-Forget. Finally, another interesting direction is to continue cracking as normal with the addition that we "forget" part of the indexing information in an adaptive way. We can drop some of the partitioning knowledge only when necessary; this results in less pieces and thus less administrative overhead and update costs. In particular, in the spirit of adaptive indexing, we can drop pieces which are about to be updated and thus create extra costs. This ac-

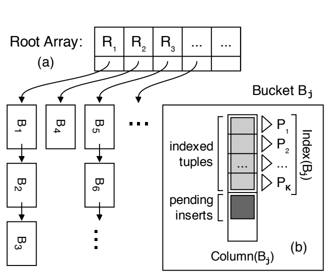

Figure 6: The Comb structure.

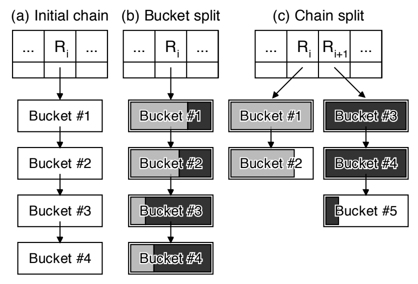

Figure 7: Bucket chains and splits.

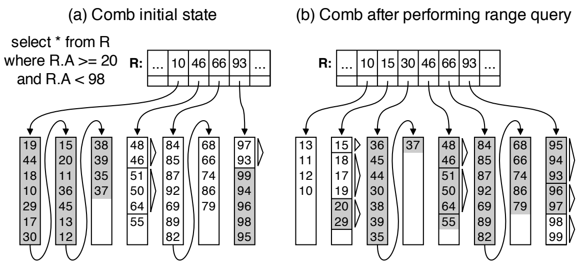

Figure 8: Query example.

tion is performed directly in the select operator during processing of a query that needs those pieces and only if we exceed a maximum number of pieces in the index. With this strategy the index continuously adapts via cracking and still is able to avoid expensive updates in very well refined areas of the index.

4. TRANSITIVE INDEXING

The problems discussed in the previous section convince us that far more radical solutions are needed. We propose such a radical solution, moving to the next step in the sequence from static to adaptive indexing. A static index has a stable design that can answer queries and accommodate updates (data insertions and deletions) as well. An adaptive index starts out with an unrefined structure and responds to queries by refining it in several ways; yet the design of the structure is fixed in advance. Our proposal takes this refinement one step further, applying it to the structure itself; then there is not only a progressive adaptation from an unindexed state to an indexed one, but also a progression through indexed states at different levels of refinement.

In particular, the index configuration starts out as a collection of unordered vectors. Each of these vectors gets broken into smaller, partially ordered pieces as queries touch it, while hot data flows from one vector to another. Eventually, the contents of hot vectors get absorbed into a trie structure that indexes those vectors themselves.

4.1 Vector Management

The content of what used to be COMB main body goes here, in its essentials.

4.2 Comb: Cracking Over Malleable Buckets

Optimizations and fine-grain variants of existing adaptive indexing techniques are limited by fundamental choices in the original approaches. In Section **??**, we show that even though our optimized adaptive indexing designs help with the resilience problem, the improvement is not enough.

Comb. In this section, we introduce Comb (Cracking Over Malleable Buckets), a novel adaptive indexing technique designed and tailored to deal with the non-resilience problem of adaptive indexing when it comes to long strings of read and write queries, while still maintaining all the good properties of previous adaptive indexing approaches.

A high level view of the Comb structure is shown in Figure 6. There are two fundamental components; (a) the root array and (b) the buckets. A single Comb structure $Comb(C)$ maintains the contents of a single column C in a column-store.

Root Array. The buckets in a Comb structure contain the actual tuples, while the root array serves the role of guiding queries to the proper bucket. As Figure 6 shows, each entry R_i in the root array corresponds to a collection of one or more chained buckets CB_i. Each root entry contains two variables. The first one is a physical pointer $Pointer(R_i)$ to the first bucket B_i in chain CB_i. The second one is the key $Key(R_i)$, which defines the starting point of the value range corresponding to bucket collection CB_i. The root array is maintained sorted on its key values.

Buckets. Each bucket in an index $Comb(C)$ is responsible for holding part of the tuples of the base column C. Each chain of buckets CB_i, corresponding to root element R_i, holds all values in the range $[Key(R_i)$,$Key(R_{i+1}))$. The value $Key(R_i)$ is the smallest value which is currently stored inside the buckets of CB_i. The bottom part of Figure 6 shows how chains of malleable buckets connect to the root array and an instance of the internals of a bucket.

Inside each bucket B_j, there is a column, $Column(B_j)$, which holds the local tuples. This column is continuously cracked and reorganized as more queries arrive. The last part of $Column(B_j)$ contains values which have been inserted but not yet merged with the rest of the partitioned values in $Column(B_j)$.

In addition, there is another array, $Index(B_j)$, which maintains the partitioning information for $Column(B_j)$; it contains the pivots used for cracking on $Column(B_j)$ and the corresponding physical position where each piece starts in $Column(B_j)$; it is equivalent to the root array with the difference that it is used for guiding through the local column. Array $Index(B_j)$ is maintained sorted.

Furthermore, there is a bit vector, $Sorted(B_j)$, which is aligned to $Index(B_j)$ and maintains information regarding whether a piece in $Column(B_j)$ is sorted or not (as in Crack-Sort).

Slicing. Comb carefully slices the data and the indexing information across buckets and across the pieces in the local columns within each bucket. The main target is to minimize the administrative costs as well as the access and update costs.

In a column of N total tuples, each bucket may contain up to a maximum of M tuples. In our experimental evaluation we found that performance is best when M is such that each bucket fits in L1 cache. Inside each bucket, each local column may be partitioned up to a maximum number of pieces P. By default, this is purposely kept small in the order of $P = 128$ pieces per bucket to guarantee low update costs. We will discuss the effect of those parameters later on. In this way, the root array may contain up to N/M entries, while each cracking piece in the local column inside each bucket contains on average M/P tuples.

As we focus on main-memory environments, we design Comb to have one root array for the following reason. For example, consider a typical 2MB L2 cache; if the root array fits in L2, our Comb implementation can index a raw column of 256GB integer values (including auxiliary row IDs).[1] This is already more than conventional memory capacity and also more than the typical size of a single column in a database. Multi-level root arrays are certainly a

[1]When N is even smaller, a good choice is that of $M = sqrt(N)$, balancing the size of the root array and buckets.

possible extension. However, this is beyond the scope of this paper.

Insertions. Let us now discuss how new values are inserted in Comb. Assume a new value v. Comb first searches its root array to find the bucket chain which contains v. Since the root array is always sorted, searching is done via binary search and costs $O(\log(N/M))$. The result bucket chain CB_i is the one where $Key(CB_i) <= v < Key(CB_{i+1})$. There are two cases when inserting a new value v.

Appending Inserts and Chaining Buckets. The first case is that the corresponding bucket chain CB_i where v belongs has enough space to accommodate v. This means that the last bucket B_j in CB_i has at least one empty position. In this case, the new value v is simply appended at the pending inserts part of $Column(B_j)$. This means that v is placed immediately after the last indexed value of $Column(B_j)$ or immediately after the last pending insert in the case that there are already pending inserts in $Column(B_j)$. Holding both indexed values and pending inserts in $Column(B_j)$ means that a query which needs all values in the value range stored in B_j can blindly retrieve $Column(B_j)$. Analysis and merging is needed only if a query needs to search within the range of $Column(B_j)$.

The second case when inserting a value v in a chain CB_i is when the last bucket B_j is full and no more values can be stored in CB_i. In this case, Comb creates a new empty bucket B_{new} and *chains* B_{new} to B_j. The new value is appended as a pending insert in B_{new}. Examples of several buckets chained together are shown in Figures 6, 7 and 8. All buckets under the same root entry R_i contain values in $[Key(R_i), Key(R_{i+1}))$.

The design option to chain buckets under the same root entry guarantees low-cost insertions. However, at the same time, the whole Comb structure should remain well adjusted to the workload to accommodate fast read access. This means that there should be no individual long bucket chains in hot ranges. Therefore, as we discuss later on, the buckets are *malleable*: queries can adaptively and on demand split chained buckets to balance and optimize the value ranges which are relevant for the workload. An alternative design would be to eagerly split buckets during insertions. However, we show in Section **??** that this is not a favorable design as it hurts the adaptive behavior of Comb.

Merging Inserts Inside a Bucket. Locally within each bucket B_j, insertions remain as pending insertions, stored at the last part of $Column(B_j)$ as shown in Figure 6. Insertions are merged with the indexed part of $Column(B_j)$ only when a query needs to refine the indexing information in B_j or when deletes arrive in B_j. Comb merges local insertions in its cracked bucket columns using the cracking update algorithms as described in Section 3 and in [8]. However, with Comb, the size of the columns as well as the partitioning depth of cracking in its local bucket columns are all kept under certain limits. This enables Comb to significantly restrict the update costs by doing only small local actions within a single bucket when merging an update. In addition, during query processing only the boundary pieces need to be updated; we elaborate more on this important detail later in this section when we discuss about range queries.

Initialization. Initializing a Comb index for an existing base column follows the procedure described for inserts above. Starting with a single bucket, it continuously inserts new values and chains new buckets as they become full. After loading a full column, and before any query has arrived, the result is a multi-bucket structure as in Figure 7(a) with a single chain of buckets.

Deletes. For robustness reasons deletes follow an eager strategy and are applied in one go in each bucket. If we want to delete a value v, we first locate the corresponding bucket B_i. Then, first all pending, if any, local inserts are merged and subsequently we locate and delete v by shuffling the tuple at the end of its piece as described in Section 3. In case there are several chained buckets under the same root entry, then all chained buckets are updated.

```
Algorithm RangeQuery(low bound $v_1$, high bound $v_2$)
1.  $lo$ = point_query($v_1$)
2.  $hi$ = point_query($v_2$)
3.  return create_view($lo$, $hi$) // position of $v$ in Comb

function point_query(boundary value $v$)
4.  if (isEmpty($R$)) return {$|R|$, 0, 0} // past-the-end position
5.  $i$ = lower_bound($R$, $v$) // the first $i$ such that $Key(R_i) \geq v$
6.  $i$ = make_standalone($i$, $v$)
7.  $j$ = $Pointer(R_i)$
8.  $k$ = crack($j$, $v$)
9.  return {$i$, $j$, $k$}

function make_standalone(root index $i$, boundary value $v$)
10. while (true) do
11.     $j$ = the first bucket pointed by $Pointer(R_i)$
12.     if ($B_j$ does not have a next bucket) break
13.     stochastic_split_chain($i$)
14.     if ($v \geq Key(R_{i+1})$) $i = i + 1$ // adjust root index to where $v$ is
15. return $i$

function stochastic_split_chain(root index $i$)
16. $pv$ = pick_random_pivot(i) // stochastic crack pivot value
17. for ($j$ = $|R| - 1$; $j > i$; $j$--) do // shift elements at $j > i$ to the right
18.     $Pointer(R_{i+1}) = Pointer(R_i)$
19.     $Key(R_{i+1}) = Key(R_i)$
20. $|R| = |R| + 1$ // increase the number of stored tuples |R|
21. $Key(R_{i+1})$ = $pv$ // set the smallest value in $R_{i+1}$ bucket chain
22. perform split_chain($i$, $pv$) which split a bucket chain pointed by
        $Pointer(R_i)$, transferring tuples with values $\geq pv$ to buckets
        in chain $R_{i+1}$ (implementation details are in Section 4.3)

function crack(bucket number $j$, boundary value $v$)
23. flush_pending_inserts($j$)
24. let $P_v$ = the (local) cracker piece of $B_j$ where $v$ is in
25. while ($|P_v| > CRACK\_AT$) stochastic_split($P_v$) // perform DD1R
26. if ($P_v$'s sorted flag is not set)
27.     if (this is a read query) sort the piece $P_v$ and set its sorted flag
28.     else return the position of $v$ in $P_v$ via linear scan
29. return the position of $v$ in $P_v$ via binary search

function flush_pending_inserts(bucket number $j$)
30. if (no local cracker index) consider all tuples are merged and return
31. do merge insert completely to all the (local) pending tuples in $B_j$
32. unset the sorted flags of the pieces that are touched during merging
```

Figure 9: Range query in Comb.

An alternative strategy would be to allow deletes to remain pending and only merge them later on as we do with inserts. However, as we show later on in Section **??**, this is not a robust solution as it can severely hurt read queries.

Continuous Adaptation. Comb is an adaptive index. As such queries are the driving force which shapes the index. As more queries arrive, the index is continuously refined to adjust to the workload patterns. The more a value range is queried, the more resolution the index adopts in this range. At the same time, value ranges not queried are seen in a much more coarse resolution until relevant queries arrive.

Adaptation During Queries. We now proceed to describe how select operations work on top of Comb. Figures 7 and 8 show how Comb is adaptively refined after a read query. Figure 7 shows how a single chain is split into two new ones, while Figure 8 shows the end result of a full range query after all its split and cracking actions. The chains at the borders of the requested range are adaptively split, while the buckets at the very borders are cracked using the query bounds. This way, Comb adapts both at a global level by splitting chains at hot workload areas and at a local level by refining the index information within buckets in hot areas. The work performed is minimized as only the 2 boundary chain buckets need to be touched, i.e., the chains where the query bounds fall; all chains and buckets in between are known to qualify so there is no need to be refined (this is similar to the discussion we did for cracking in Figure **??** and Section 3). Given that we do not have to refine all chains in between the query boundaries, this also means that we do not have to merge local updates in their respective buckets as we need all qualifying values anyway (both indexed and non-indexed values). This is why a more lazy approach to inserts is beneficial. With deletes, however, we need to guarantee there are no deletes leftover across all qualifying buckets both in the borders of the queried range and in between. Being lazy creates excessive costs at query time, hence, we choose eager deletes locally within each bucket. We revisit the last two points in Section **??**.

Querying Comb. The algorithm for a Comb range select is described in detail Algorithm 9. Assume a select operator requesting all values in $[low, high]$ from column C. Comb first searches its root array to find the bucket chain CB_{low} which contains low and the chain CB_{high} which contains $high$. Since the root array is always sorted, searching is done via binary search and costs $O(\log(N/M)$.

For ease of presentation, assume first that both boundary chains contain one bucket each. For chain CB_{low}, the select operator searches within the single bucket B_{low} for low. In this way, for the single bucket B_{low} in CB_{low}, it does a lookup in $Index(B_{low})$ to find the piece in $Column(B_{low})$ where low is contained. Given that $Index(B_{low})$ is sorted, the search is done via a binary search action (similar to searching the root array). This costs $O(\log(P))$. Once we know the corresponding piece P_{low}, then the next action depends on whether the size of the piece has reached the minimum allowed size (M/P) (denoted as $CRACK\_AT$ in Algorithm 9). In the case that we have already reached the piece size limit, then we check (in $Sort(B_{low})$) whether P_{low} is already sorted. If yes, then we simply binary search in P_{low} to find low. Otherwise, we first sort P_{low} in-place and then binary search for low. The respective piece is marked in bit vector $Sort(B_{low})$. If the size of the piece is still above the size threshold, then P_{low} is recursively cracked using random pivots until it reaches a small enough size where it can be sorted with a small cost. For this step, Comb uses the stochastic cracking algorithm (DDR) which is shown to be robust [5]. $Index(B_{low})$ is updated with a new entry for the new pieces created. The most expensive case is when this is the very first query in B_{low} and B_{low} is full. Then, we need to reorganize $Column(B_{low})$ touching all M tuples.

The above procedure is repeated by analogy for CB_{high} and $high$. Once we have reorganized the boundary chains, then all chains and buckets in between form the result as shown in the example of Figure 8 (qualifying tuples are marked with grey color). The total cost of a query is $O(\log(N/M) + \log(P) + M)$.

Adaptive Splits During Queries. In the case that a boundary chain where a query bound b falls in contains more than one buckets, then a query adaptively and recursively splits this chain until the boundary chain where b falls in remains with a single bucket. In order to split a chain CB_i as the one in Figure 7(a), the first step is to choose a random pivot rv. The pivot rv is used to crack, i.e., to physically reorganize all chained buckets in CB_i, such as to split the values in two partitions per chained bucket; one partition contains all values $x < rv$ and the second partition contains all values $x >= rv$. The result of this step is shown in Figure 7(b). Now we can create two new sets of chained buckets as shown in Figure 7(c) where the right set of buckets is under a new root entry on the random pivot which was chosen for the split. Bound b falls in one of those chained buckets. The process of adaptive splitting continues recursively until the chain where b falls in is a single bucket and thus no more splits are required.

We could split directly on b as a pivot, but then Comb would be vulnerable to unfavorable workload patterns, as it has been shown for adaptive indexing in [5]. Thus, we opt to choose random pivots and keep splitting until there is only one bucket in the chain that contains b so as to achieve robustness. The extra splitting overhead we pay affords more robust performance in subsequent queries.

Example. Figure 8 shows an example of a range query in Comb. It shows the initial state and the final state after the index adapts to the query. The query asks for all values in $[20, 98)$. Initially, there are 4 bucket chains linked in the root array as shown in Figure 8(a). The first chain under $key = 10$ contains 3 chained buckets. The second chain under $key = 46$ contains a single bucket which is already cracked in two pieces (indicated by triangles) and also contains 1 pending insert value $val = 55$. The third chain contains 2 chained buckets, while the fourth one is again a stand-alone bucket. In order to answer the range query, Comb first searches for low bound, $low = 20$ in the initial state in Figure 8(a). This lies in the first chain of buckets (between root entries with $key = 10$ and $key = 46$). Comb adaptively splits this chain in a stochastic fashion into 3 new chains as shown in Figure 8(b) with keys 10, 15 and 30. The bound $low = 20$ falls in [15,30) so it falls in the second chain in Figure 8(b) under $key = 15$. Then, this chain is cracked using $val = 20$ as pivot. The grey region in the buckets in Figure 8(a) shows which tuples are touched from the initial state during the query process. An important feature of Comb is that the chain under $key = 46$ from Figure 8(a) is untouched – it is not accessed, cracked and its pending inserts are not merged although it qualifies for the result. To answer the query, it suffices to fetch all values in the bucket regardless of the order or the partitioning information. The same holds for the chain under $key = 66$ which remains as a chain of two unindexed chained buckets. Finally, using the high bound of the query $high = 98$, Comb adapts the chain under $key = 93$ and cracks it using $val = 98$ as pivot (the 98 is not inclusive). The tuples which qualify for the query result are shown by the grey area in Figure 8(b).

4.3 Bucket Splitting

The *malleability* of buckets, i.e., their capacity to be split and recombined, is one of the most crucial properties in Comb. Thanks to this malleability, we can maintain a well balanced structure which is easy to access and update. The key action is that of partitioning a collection of chained buckets into roughly two equal pieces each, i.e., the step shown in Figure 7(b).

The partitioning of each bucket can be done using a standard partitioning algorithm for partitioning columns (similar to what cracking is using). The resulting algorithm, $Crack :: split\_chain$, which splits a chain of buckets is shown in Figure **??**. It first splits all buckets in a chain in two pieces based on the same (random) pivot and then creates two new chains to place the resulting pieces, trying to minimize data movement as much as possible.

Fission. We discuss two new alternative designs which are optimized to split chains and buckets in Comb. The first algorithm, called *Fission*, is shown in Figure **??**. Its main characteristic is that it splits buckets without using any branches; this removes the hazard of branch mispredictions which, as we show later on, is excessive when splitting buckets in two half pieces. Splitting buckets in half is preferable in order to minimize the number of splits required (since adaptive splitting works recursively).

Fusion. Still though there are more opportunities to improve this crucial Comb operation further. We propose a new algorithm, called *Fusion*, to split Comb chains. The goal is to minimize both data movement and branch misprediction cases. The main idea is that it works in two passes over the data which turns out to be faster in modern architectures compared with a single pass. The first pass marks which tuples need to be moved in a branch-free way, while the second pass implements these data movements in-place. The details are in Algorithm **??**. Since the all to be moved tuples have been marked in the first pass, Fusion can minimize data movements as in the second pass it knows exactly which tuples in each bucket need to be moved and where they should go.

4.4 Transitions and Merges

A separate discussion about each transitive step goes here.

5. REFERENCES

[1] S. Agarwal, A. Panda, B. Mozafari, S. Madden, and I. Stoica. Blink and it's done: Interactive queries on very large data. In *PVLDB*, 2012.

[2] S. Chaudhuri. What next? a half-dozen data management research goals for big data and cloud. In *PODS*, 2012.

[3] F. Färber, N. May, W. Lehner, P. Große, I. Müller, H. Rauhe, and J. Dees. The SAP HANA database – an architecture overview. *IEEE Data Eng. Bull.*, 35(1):28–33, 2012.

[4] G. Graefe and H. Kuno. Self-selecting, self-tuning, incrementally optimized indexes. In *EDBT*, pages 371–381, 2010.

[5] F. Halim, S. Idreos, P. Karras, and R. H. C. Yap. Stochastic database cracking: Towards robust adaptive indexing in main-memory column-stores. *PVLDB*, 5(6):502–513, 2012.

[6] S. Idreos, I. Alagiannis, R. Johnson, and A. Ailamaki. Here are my data files. here are my queries. where are my results? In *CIDR*, pages 57–68, 2011.

[7] S. Idreos, M. L. Kersten, and S. Manegold. Database cracking. In *CIDR*, pages 68–78, 2007.

[8] S. Idreos, M. L. Kersten, and S. Manegold. Updating a cracked database. In *SIGMOD*, pages 413–424, 2007.

[9] 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*, 4(9):585–597, 2011.

[10] A. Kemper and T. Neumann. HyPer: A hybrid OLTP & OLAP main memory database system based on virtual memory snapshots. In *ICDE*, 2011.

[11] M. Kersten, S. Idreos, S. Manegold, and E. Liarou. The researcher's guide to the data deluge: Querying a scientific database in just a few seconds. *PVLDB*, 4(12):1474–1477, 2011.

[12] V. Leis, A. Kemper, and T. Neumann. The adaptive radix tree: Artful indexing for main-memory databases. In *ICDE*, pages 38–49, 2013.

[13] J. Rao and K. A. Ross. Making B^{+}-trees cache conscious in main memory. In *SIGMOD*, pages 475–486, 2000.

[14] S. Richter, J.-A. Quiané-Ruiz, S. Schuh, and J. Dittrich. Towards zero-overhead static and adaptive indexing in Hadoop. *VLDBJ*, 2013.

[15] L. Sidirourgos, M. L. Kersten, and P. A. Boncz. Sciborq: Scientific data management with bounds on runtime and quality. In *CIDR*, 2011.