Recipient of a best paper award at VLDB 2014

The Uncracked Pieces in Database Cracking

Felix Martin Schuhknecht[†] Alekh Jindal[‡*] Jens Dittrich[†]

[†]Information Systems Group, Saarland University
http://infosys.cs.uni-saarland.de

[‡]CSAIL, MIT
people.csail.mit.edu/alekh

ABSTRACT

Database cracking has been an area of active research in recent years. The core idea of database cracking is to create indexes adaptively and incrementally as a side-product of query processing. Several works have proposed different cracking techniques for different aspects including updates, tuple-reconstruction, convergence, concurrency-control, and robustness. However, there is a lack of any comparative study of these different methods by an independent group. In this paper, we conduct an experimental study on database cracking. Our goal is to critically review several aspects, identify the potential, and propose promising directions in database cracking. With this study, we hope to expand the scope of database cracking and possibly leverage cracking in database engines other than MonetDB.

We repeat several prior database cracking works including the core cracking algorithms as well as three other works on convergence (hybrid cracking), tuple-reconstruction (sideways cracking), and robustness (stochastic cracking) respectively. We evaluate these works and show possible directions to do even better. We further test cracking under a variety of experimental settings, including high selectivity queries, low selectivity queries, and multiple query access patterns. Finally, we compare cracking against different sorting algorithms as well as against different main-memory optimised indexes, including the recently proposed Adaptive Radix Tree (ART). Our results show that: (i) the previously proposed cracking algorithms are repeatable, (ii) there is still enough room to significantly improve the previously proposed cracking algorithms, (iii) cracking depends heavily on query selectivity, (iv) cracking needs to catch up with modern indexing trends, and (v) different indexing algorithms have different indexing signatures.

1. INTRODUCTION

1.1 Background

Traditional database indexing relies on two core assumptions: (1) the query workload is available, and (2) there is sufficient idle time to create the indexes. Unfortunately, these assumptions are not valid anymore in modern applications, where the workload is not known or constantly changing and the data is queried as soon as it arrives. Thus, several researchers have proposed adaptive indexing techniques to cope with these requirements. In particular, *Database Cracking* has emerged as an attractive approach for adaptive indexing in recent years [13, 8, 10, 9, 11, 4, 6]. Database Cracking proposes to create indexes adaptively and as a side-product of query processing. The core idea is to consider each incoming query as a hint for data reorganisation which eventually, over several queries, leads to a full index. Figure 1 recaps and visualizes the concept.

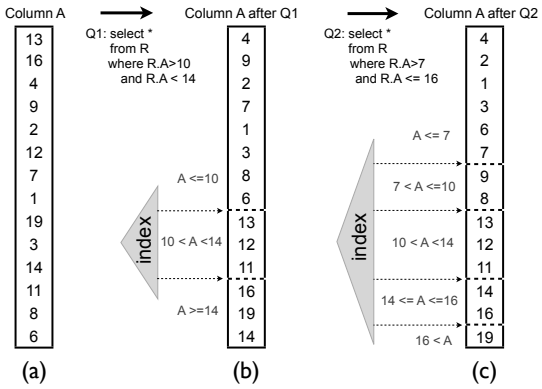


Figure 1: Database Cracking Example

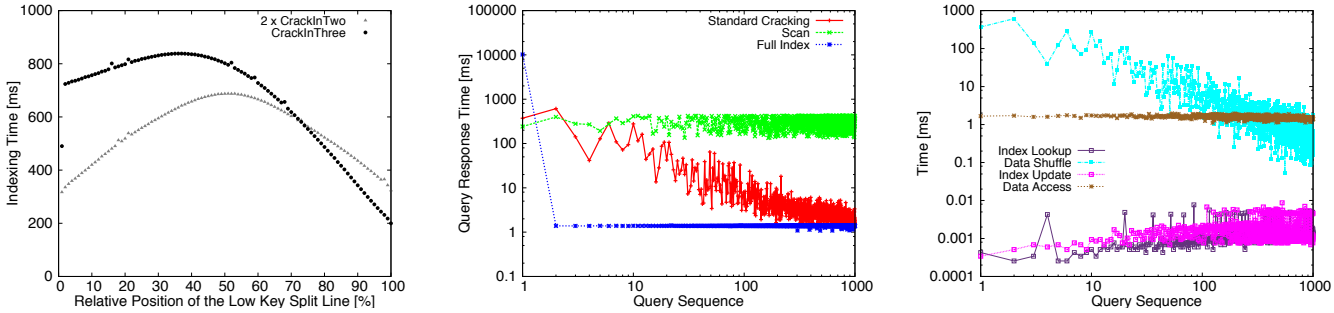
1.2 Our Focus

Database Cracking has been an area of active research in recent years, led by researchers from CWI Amsterdam. This research group has proposed several different indexing techniques to address different dimensions of database cracking, including updates [10], tuple-reconstruction [9], convergence [11], concurrency-control [4], and robustness [6]. In this paper, we critically review database cracking. We repeat the core cracking algorithms, i.e. crack-in-two and crack-in-three [8], as well as three advanced cracking algorithms [9, 11, 6]. We identify the weak spots in these algorithms and discuss extensions to fix them. Finally, we also extend the experimental parameters previously used in database cracking by varying the query selectivities and by comparing against more recent, main-memory optimised indexing techniques, including ART [15].

To the best of our knowledge, this is the first study by an independent group on database cracking. Our goal is to put database cracking in perspective by repeating several prior cracking works, giving new insights into cracking, and offering promising directions for future work. We believe that this will help the database community to understand database cracking better and to possi-

[*]Work done while at Saarland University.

Proceedings of the VLDB Endowment, Vol. 7, No. 2



(a) Comparing Single Query Indexing Time (b) Reproducing Cracking Behaviour (c) Cost Breakdown

Figure 2: Revisiting Standard Cracking

bly leverage cracking for database systems other than MonetDB as well. Our core contributions in this paper are as follows:

(1.) Revisiting Cracking. We revisit the core cracking algorithms, i.e. crack-in-two and crack-in-three [8], and compare them for different positions of the pivot elements. We do a cost breakdown analysis of the cracking algorithm into index lookup, data shuffle, index update, and data access costs. We identify three major concerns, namely convergence, tuple-reconstruction, and robustness. In addition, we evaluate advanced cracking algorithms, namely hybrid cracking [11], sideways cracking [9], and stochastic cracking [6] respectively, which were proposed to address these concerns. Additionally, in order to put together the differences and similarities between different cracking algorithms, we classify the cracking algorithms based on the strategy to pick the pivot, the creation time, and the number of partitions (Section 2).

(2.) Extending Cracking Algorithms. In order to better understand the cracking behaviour, we modify three advanced cracking algorithms, namely hybrid cracking [11], sideways cracking [9], and stochastic cracking [6]. We show that buffering the swap elements in a heap before actually swapping them (*buffered swapping*) can lead to better convergence than hybrid cracking. Next, we show that covering the projection attributes with the cracker column (*covered cracking*) scales better than sideways cracking in the number of projected attributes. Finally, we show that creating more balanced partitions upfront (*coarse-granular indexing*) achieves better robustness in query performance than stochastic cracking. We also map these extensions to our cracking classification (Section 3).

(3.) Extending Cracking Experiments. Finally, we extend the cracking experiments in order to test cracking under different settings. First, we compare database cracking against full indexing using different sorting algorithms and index structures. In previous works on database cracking quick sort is used to order the data indexed by the traditional methods that are used for comparison. Further, the cracker index is realized by an AVL-Tree to store the index keys. In this paper, we do a reality check with recent developments in sorting and indexing for main-memory systems. We show that full index creation with radix sort is twice as fast as with quick sort. We also show that ART [15] is up to 3.6 times faster than the AVL-Tree in terms of lookup time. We also vary the query selectivity from very high selectivity to medium selectivity and compare the effects. We conclude two key observations: (i) the choice of the index structure has an impact only for very high selectivities, i.e. higher than 10^{-6} (one in a million), otherwise the data access costs dominate the index lookup costs; and (ii) cracking creates more balanced partitions and hence converges faster for medium selectivities, i.e. around 10%. Furthermore, we apply a sequential and a skewed query access pattern and analyze how the different adaptive indexing methods cope with them. Our results show that sequential workloads are the weak spots of query driven methods while skewed patterns increase the overall variance (Section 4). Finally, we conclude by putting together the key lessons learned. Additionally, we also introduce *signatures* to characterise the indexing behaviour of different indexing methods and to understand as well as differentiate them visually (Section 5).

Experimental Setup. We use a common experimental setup throughout the paper. We try to keep our setup as close as possible to the earlier cracking works. Similar to previous cracking works, we use an integer array with 10^8 uniformly distributed values. Unless mentioned otherwise, we run 1000 random queries, each with selectivity 1%. The queries are of the form: `SELECT A FROM R WHERE A>=low AND A<high`. We repeat the entire query sequence three times and take the average runtime of each query in the sequence. We consider two baselines: (i) *scan* which reads the entire column and post-filters the qualifying tuples, and (ii) *full index* which fully sorts the data using quick sort and performs binary search for query processing. If not stated otherwise, all indexes are unclustered and uncovered. We implemented all algorithms in a stand-alone program written in C/C++ and compile with G++ version 4.7 using optimization level 3. Our test bed consists of a single machine with two Intel Xeon X5690 processors running at a clock speed of 3.47 GHz. Each CPU has 6 cores and supports 12 threads via Intel Hyper Threading. The L1 and L2 cache sizes are 64 KB and 256 KB respectively for each core. The shared L3 cache has a size of 12 MB. Our machine has 200 GB of main memory and runs openSUSE 12.2 (Mantis) linux in the 64-bit version with kernel 3.1.

2. REVISITING CRACKING

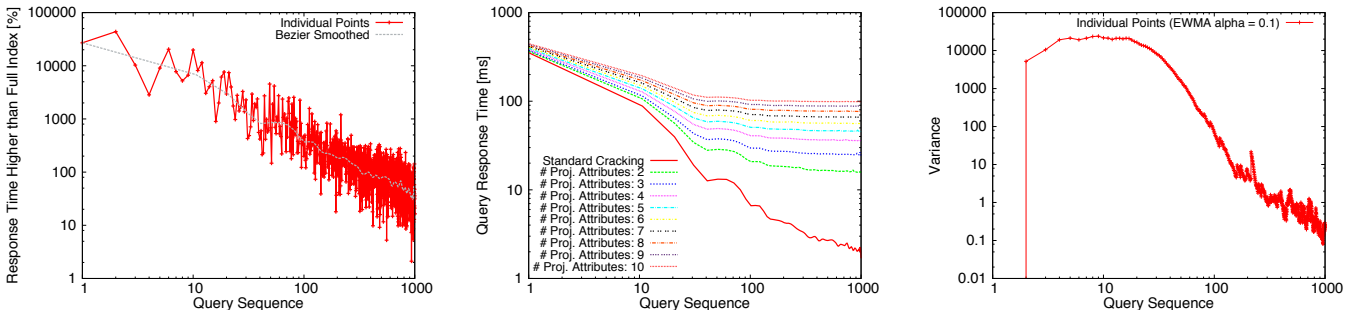
Let us start by revisiting the original cracking algorithm [8]. Our goal in this section is to first compare crack-in-two with crack-in-three, then to repeat the standard cracking algorithm under similar settings as in previous works, then to break down the costs of cracking into individual components, and finally to identify the major concerns in the original cracking algorithm.

2.1 Crack-in-two Vs Crack-in-three

crack-in-two: *partition the index column into two pieces using one end of a range query as the boundary.*

crack-in-three: *partition the index column into three pieces using the two ends of a range query as the two boundaries.*

The original cracking paper [8] introduces two algorithms: *crack-in-two* and *crack-in-three* to partition (or *split*) a column into two and three partitions respectively. Conceptually crack-in-two is suitable for one-sided range queries, e.g. $A < 10$, whereas crack-in-three for two-sided range queries, e.g. $7 < A < 10$. However, we could also apply two crack-in-twos for a two-sided range query.



(a) Cracking Convergence (b) Scaling Projected Attributes (c) Cracking Variance

Figure 3: Key Concerns in Standard Cracking

Let us now compare the performance of crack-in-two and crack-in-three on two-sided range queries. We consider the cracking operations from a single query and vary the position of the split line in the cracker column from bottom (low relative position) to top (high relative position). A relative position of the low key split line of $p\%$ denotes that the data is partitioned into two parts with size $p\%$ and $(100-p)\%$. We expect the cracking costs to be the maximum around the center of the column (since maximum swapping will occur) and symmetrical on either ends of the column. Figure 2(a) shows the results. Though both $2\times$crack-in-two and crack-in-three have maximum costs around the center, surprisingly crack-in-three is not symmetrical on either ends. Crack-in-three is much more expensive in the lower part of the column than in the upper part. This is because crack-in-three always starts considering elements from the top. Another interesting observation from Figure 2(a) is that even though $2\times$crack-in-two performs two cracking operations, it is cheaper than crack-in-three when the split position is in the lower 70% of the column. Thus, we see that crack-in-two and crack-in-three are very different algorithms in terms of performance and future works should consider this when designing newer algorithms.

2.2 Standard Cracking Algorithm

standard cracking: *incrementally and adaptively sort the index column using crack-in-three when both ends of a range query fall in the same partition and crack-in-two otherwise.*

We implemented the standard cracking algorithm which uses crack-in-three wherever two split lines lie in the same partition, and tested it under the same settings as in previous works. As in the original papers, we use an AVL-Tree as cracker index to be able to compare the results. Figure 2(b) shows the results. We can see that standard cracking starts with similar performance as full scan and gradually approaches the performance of full index. Moreover, the first query takes just 0.3 seconds compared to 0.24 seconds of full scan[1], even though standard cracking invests some indexing effort. In contrast, full index takes 10.2 seconds to fully sort the data before it can process the first query. This shows that standard cracking is lightweight and it puts little penalty on the first query. Overall, we are able to reproduce the cracking behaviour of previous works.

2.3 Cost Breakdown

Next let us see the cost breakdown of original cracking algorithm. The goal here is to see where the cracking query engine spends most of the time and how that changes over time. Figure 2(c) shows the cost breakdown of the query response time into four components: (i) *index lookup* costs to identify the partitions for cracking, (ii) *data shuffle* costs of swapping the data items in the column, (iii) *index update* costs for updating the index structure with the new partitions, and (iv) *data access* costs to actually access the qualifying tuples. We can see that the data shuffle costs dominate the total costs initially. However, the data shuffle costs decrease gradually over time and match the data access costs after $1,000$ queries. This shows that standard cracking does well to distribute the indexing effort over several queries. We can also see that index lookup and update costs are orders of magnitude less than the data shuffle costs. For instance, after 10 queries, the index lookup and update costs are about 1μs whereas the shuffle costs are more than 100 ms. This shows that standard cracking is indeed lightweight and has very little index maintenance overheads. However, as the number of queries increases, the data shuffle costs decrease while the index maintenance costs increase.

[1]Note that the query time of full scan varies by as much as 4 times. This is because of lazy evaluation in the filtering depending on the position of low key and high key in the value domain.

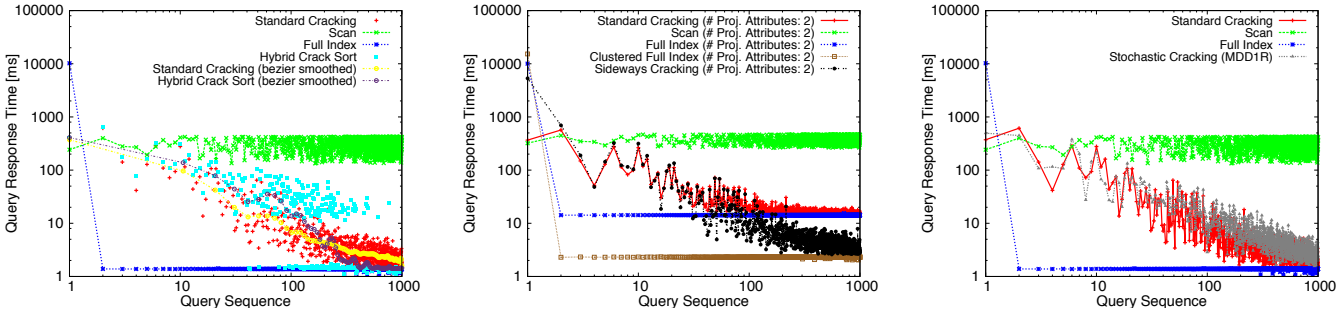
2.4 Key Concerns in Standard Cracking

Let us now take a closer look at the standard cracking algorithm from three different perspectives, namely convergence to a full index, scaling the number of projected attributes, and variance in query performance.

Cracking Convergence. Convergence is a key concern and major criticism for database cracking. Figure 3(a) shows the number of queries after which the query response time of standard cracking is within a given percentage of full index. The figure also shows a bezier smoothened curve of the data points. From the figure we can see that after $1,000$ queries, on average, the query response time of standard cracking is still 40% higher than that of full index.

Scaling Projected Attributes. By default, database cracking leads to an unclustered index, i.e. extra lookups are needed to fetch the projected attributes. Figure 3(b) shows the query response time with tuple reconstruction, when varying the number of projected attributes from 1 to 10. For the ease of presentation, we show only the bezier smoothened curves. We can see that standard cracking does not scale well with the number of attributes. In fact, after $1,000$ queries, querying 10 attribute tuples is almost 2 orders of magnitude slower than querying 1 attribute tuples.

Cracking Variance. Standard cracking partitions the index column based on the query ranges of the selection predicate. As a result, skewed query range predicates can lead to skewed partitions and thus unpredictable query performance. Figure 3(c) shows the variance of standard cracking query response times using the exponentially weighted moving average (EWMA). The variance is calculated as described in [3]. The degree of weighting decrease is $\alpha = 0.1$. We can see that unlike full index (see Figure 2(b)), cracking does not exhibit stable query performance. Furthermore,



(a) Hybrid Cracking (b) Sideways Cracking (c) Stochastic Cracking

Figure 4: Revisiting Three Advanced Cracking Algorithms

we also see that the amount of variance for standard cracking decreases by five orders of magnitude.

2.5 Advanced Cracking Algorithms

Several follow-up works on cracking focussed on the key concerns in standard cracking. In this section, we revisit these advanced cracking techniques.

hybrid cracking: *create unsorted initial runs which are physically reorganised and then adaptively merged for faster convergence.*

Hybrid cracking [11] aims at improving the poor convergence of standard cracking to a full index, as shown in Figure 3(a). Hybrid cracking combines ideas from adaptive merging [5] with database cracking in order to achieve fast convergence to a full index, while still keeping low initialisation costs. The key problem in standard cracking is that it creates at most two new partition boundaries per query, and thus requires several queries to converge to a full index. On the other hand, adaptive merging creates initial sorted runs, and thus pays a high cost for the first query. Hybrid cracking overcomes these problems by creating initial unsorted partitions and later adaptively refining them with lightweight reorganisation. In addition to reorganising the initial partitions, hybrid cracking also moves the qualifying tuples from each initial partition into a final partition. The authors explore different strategies for reorganising the initial and final partitions, including sorting, standard cracking, and radix clustering, and conclude standard cracking to be the best for initial partitions and sorting to be the best for final partition. By creating initial partitions in a lightweight manner and introducing several partition boundaries, hybrid cracking converges better.

We implemented *hybrid crack sort*, which showed the best performance in [11], as close to the original description as possible. Figure 4(a) shows hybrid crack sort in comparison to standard cracking, full index, and scan. We can see that hybrid crack sort converges faster as compared to standard cracking.

sideways cracking: *adaptively create, align, and crack every accessed selection-projection attribute pair for efficient tuple reconstruction.*

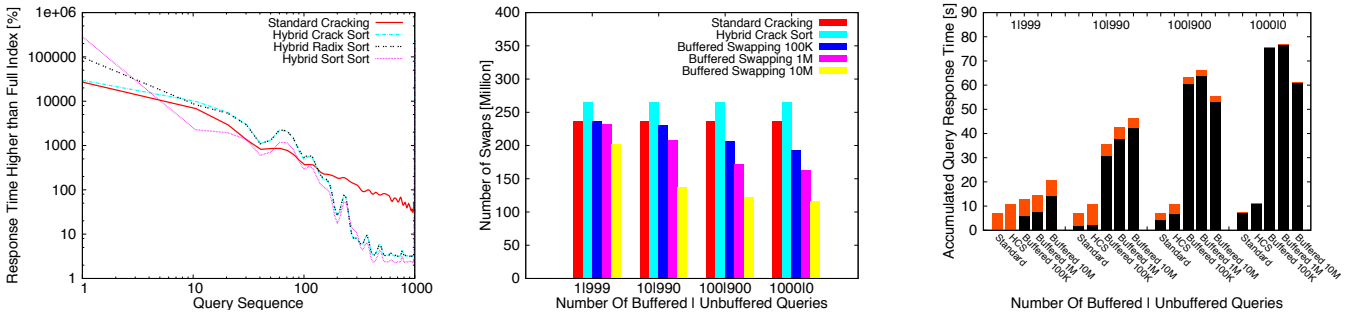
Sideways Cracking [9] uses *cracker maps* to address the issue of inefficient tuple reconstruction in standard cracking, as shown in Figure 3(b). A cracker map consists of two logical columns, the cracked column and a projected column, and it is used to keep the projection attributes aligned with the selection attributes. When a query comes in, sideways cracking creates and cracks only those crackers maps that contain any of the accessed attributes. As a result, each accessed column is always aligned with the cracked column of its cracker map. If the attribute access pattern changes, then the cracker maps may reflect different progressions with respect to the applied cracks. Sideways cracking uses a log to record the state of each cracker map and to synchronize them when needed. Thus, sideways cracking works without any workload knowledge and adapts cracker maps to the attribute access patterns. Further, it improves its adaptivity and reduces the amount of overhead by only materializing those parts of the projected columns in the cracker maps which are actually queried (*partial sideways cracking*).

We reimplemented sideways cracking similar to as described above, except that we store cracker maps in row layout instead of column layout. We do so because the two columns in a cracker map are always accessed together and a row layout offers better tuple reconstruction. In addition to the cracked column and the projected column, each cracker map contains the rowIDs that map the entries into the base table as well as a status column denoting which entries of the projected column are materialized. Figure 4(b) shows the performance of sideways cracking in comparison. In this experiment the methods have to project one attribute while selecting on another. In addition to the unclustered version of full index, we also show the clustered version (clustered full index). We can see that sideways cracking outperforms all unclustered methods after about 100 queries and approaches the query response time of clustered full index. Thus, sideways cracking offers efficient tuple reconstruction.

stochastic cracking: *create more balanced partitions using auxiliary random pivot elements for more robust query performance.*

Stochastic cracking [6] addresses the issue of performance unpredictability in database cracking, as seen in Figure 3(c). A key problem in standard cracking is that the partition boundaries depend heavily on the incoming query ranges. As a result, skewed query ranges can lead to unbalanced partition sizes and successive queries may still end up rescanning large parts of the data. To reduce this problem, stochastic cracking introduces additional cracks apart from the query-driven cracks at query time. These additional cracks help to evolve the cracker index in a more uniform manner. Stochastic cracking proposes several variants to introduce these additional cracks, including data driven and probabilistic decisions. By varying the amount of auxiliary work and the crack positions, stochastic cracking manages to introduce a trade-off situation between variance on one side and cracking overhead on the other side.

We reimplemented the *MDD1R* variant of stochastic cracking, which showed the best overall performance in [6]. In this variant, the partitions in which the query boundaries fall are cracked by performing exactly one random split. Additionally, while performing the random split, the result of each partition at the boundary of the queried range is materialised in a separate view. At query time the result is built by reading the data of the boundary partitions from the views and the data of the inner part from the index.



(a) Convergence Speed towards Full Index (b) Influence on Swap Count (c) Influence on Query Response Time

Figure 5: Comparing Convergence of Standard Cracking, Hybrid Cracking and Buffered Swapping

Figure 4(c) shows the MDD1R variant of stochastic cracking. We can see that stochastic cracking (MDD1R) behaves very similar to standard cracking, although the query response times are overall slower than those of standard cracking. As the uniform random access pattern creates balanced partitions by default, the additional random splits introduced by stochastic cracking (MDD1R) do not have an effect. We will come back to stochastic cracking (MDD1R) with other access patterns in Section 4.4.

2.6 Cracking Classification

Let us now compare and contrast the different cracking algorithms discussed so far with each other. The goal is to understand what are the key differences (or similarities) between these algorithms. This will possibly help us in identifying the potential for newer cracking algorithms. Note that all cracking algorithms essentially *split* the data incrementally. Different algorithms split the data differently. Thus, we categorise the cracking algorithms along three dimensions: (i) the number of split lines they introduce, (ii) the split strategy, and (iii) the timing of the split. Table 1 shows the classification of different cracking algorithms along these three dimensions. Let us discuss these below.

DIMENSIONS	CATEGORY	NO INDEX	STANDARD CRACKING	HYBRID CRACKING (CRACK SORT)	SIDEWAYS CRACKING	STOCHASTIC CRACKING (MDD1R)	FULL INDEX
NUMBER OF SPLIT LINES	ZERO	■					
	FEW		■		■	■	
	SEVERAL			■			
	ALL			■			■
SPLIT STRATEGY	NONE	■					
	QUERY BASED		■	■	■	■	
	RANDOM					■	
	DATA BASED			■			■
SPLIT TIMING	NEVER	■					
	PER QUERY		■	■	■	■	
	UPFRONT						■

Table 1: Classification of Cracking Algorithms.

Number of Split Lines. The core cracking philosophy mandates all cracking algorithms to do some indexing effort, i.e. introduce at least one split line, when a query arrives. However, several algorithms introduce other split lines as well. We classify the cracking algorithms into the following four categories based on the number of split lines they introduce.

(1.) *Zero:* The trivial case is when a method introduces no split line and each query performs a full scan.

(2.) *Few:* Most cracking algorithms introduce a few split lines at a time. For instance, standard cracking introduces either one or two splits lines for each incoming query. Similarly, sideways cracking introduces split lines in all accessed cracker maps.

(3.) *Several:* Cracking algorithms can also introduce several split lines at a time. For example, hybrid crack sort may introduce several thousand initial partitions and introduce either one or two split lines in each of them. Thus, generating several split lines in total.

(4.) *All:* The extreme case is to introduce all possible split lines, i.e. fully sort the data. For example, hybrid crack sort fully sorts the final partition, i.e. introduces all split lines in the final partition.

Split Strategy. Standard cracking chooses the split lines based on the incoming query. However, several advanced cracking algorithms employ other strategies. Below, we classify the cracking algorithms along four splitting strategies.

(1.) *Query Based:* The standard case is to pick the split lines based on the selection predicates in the query, i.e. the low and high keys in the query range.

(2.) *Data Based:* We can also split data without looking at a query. For example, full sorting creates split lines based only on the data.

(3.) *Random:* Another strategy is to pick the split lines randomly as in stochastic cracking.

(4.) *None:* Finally, the trivial case is to not have any split strategy, i.e. do not split the data at all and perform full scan for all queries.

Split Timing. Lastly, we consider the timing of the split to classify the cracking algorithms. We show three time points below.

(1.) *Upfront:* A cracking algorithm could perform the splits before answering any queries. Full indexing falls in this category.

(2.) *Per Query:* All cracking algorithms we discussed so far perform splits when seeing a query.

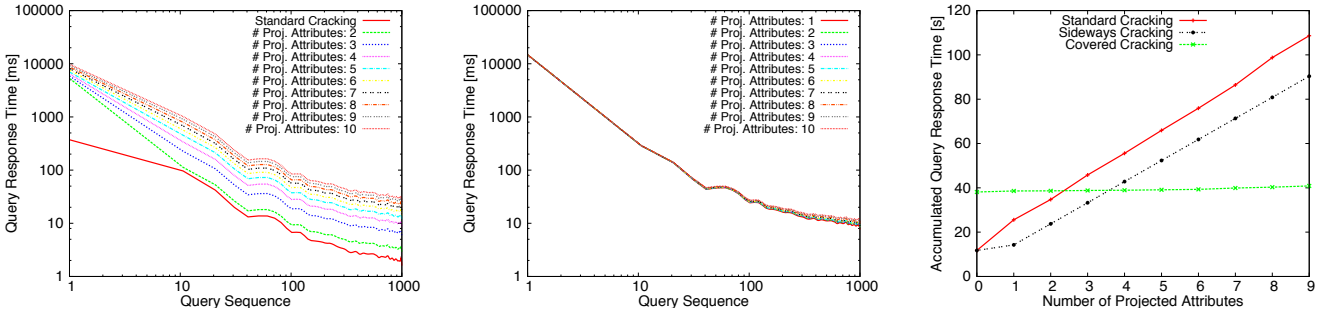
(3.) *Never:* The trivial case is to never perform the splits, i.e. fully scanning the data for each query.

3. EXTENDING CRACKING ALGORITHMS

In this section, we discuss the weaknesses in the advanced cracking algorithms and evaluate possible directions on how to improve them.

3.1 Improving Cracking Convergence

Let us see how well hybrid cracking [11] addresses the convergence issue and whether we can improve upon it. First, let us compare hybrid crack sort from Figure 4(a) with two other variants of hybrid cracking: *hybrid radix sort*, and *hybrid sort sort*. Figure 5(a) shows how quickly the hybrid algorithms approach to a full index. We can see that hybrid radix sort converges similar to hybrid crack sort and hybrid sort sort converges faster than both of them. This suggests that the convergence property in hybrid algorithms comes from the sort operations. However, keeping the final partition fully sorted is expensive. Indeed, we can see several spikes in hybrid crack sort in Figure 4(a). If a query range is not contained in the final partition, all qualifying entries from all initial partitions must be merged and sorted into the final partition. Can we do better?



(a) Varying Number of Projected Attributes for Sideways Cracking

(b) Varying Number of Projected Attributes for Covered Cracking

(c) Covering Tradeoff for Tuple Reconstruction

Figure 6: Comparing Tuple Reconstruction Cost of Standard, Sideways, and Covered Cracking

Can we move data elements to their final position (as in full sorting) in a fewer number of swaps, and thus improve the cracking convergence?

buffered-swapping: *Instead of swapping elements immediately after identification by the cracking algorithm, insert them into heaps and flush them back into the index as sorted runs.*

Let us look at the crack-in-two operation[2] in hybrid cracking. Recall that the crack-in-two operation scans the dataset from both ends until we find a pair of entries which need to be swapped (i.e. they are in the wrong partitions). This pair is then swapped and the algorithm continues its search until the pointers meet. Note that there is no relative ordering between the swapped elements and they may end up getting swapped again in future queries, thus penalising them over and over again. We can improve this by extending the crack-in-two operation to buffer the elements identified as swap pairs, i.e. *buffered crack-in-two*. Buffered crack-in-two collects the swap pairs in two heaps: a max-heap for values that should go to the upper partition and a min-heap for values that should go to the lower partition. In addition to the heap structures, we maintain two queues to store the empty positions in the two partitions. The two heaps keep the elements in order and when the heaps are full we swap the top-elements in the two heaps to the next available empty position. This process is repeated until no more swap pairs can be identified and the heaps are empty. As a result of heap ordering, the swapped elements retain a relative ordering in the index after each cracking step. This order is even valid for entries that were not in the heap at the same time, but shared presence with a third element and hence a transitive relationship is established. Every pair element that is ordered in this process is never swapped in future queries and thus, the number of swaps is reduced. The above approach of buffered crack-in-two is similar to [16], where two heaps are used to improve the stability of the replacement selection algorithm. By adjusting the maximal heap size in buffered crack-in-two, we can tune the convergence speed of the cracked index. Larger heap size results in larger sorted runs. However, larger heaps incur high overhead to keep its data sorted. In the extreme case, a heap size equal to the number of (swapped) elements results in full sorting while a heap size of 1 falls back to standard crack-in-two. Of course buffered crack-in-two can be embedded in any method that uses the standard crack-in-two algorithm. To separate it from the remaining methods we integrate it into a new technique called *buffered swapping* that is a mixture of buffered and standard crack-in-two. For the first n queries buffered swapping uses buffered crack-in-two. After that buffered swapping switches to standard cracking-in-two.

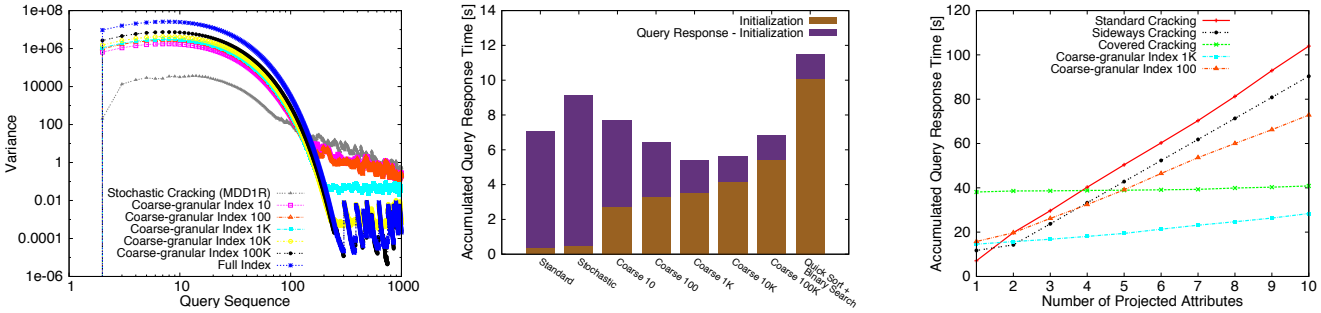
Figure 5(b) shows the number of swaps in standard cracking, hybrid crack sort, and buffered swapping over 1000 queries. In order to make them comparable, we force all methods to use only crack-in-two operations. For buffered swapping we vary the number buffered queries n_b along the X-axis, i.e. the first n_b queries perform buffered swapping while the remaining queries still perform the standard crack-in-two operation. We vary the maximal heap size from $100K$ to $10M$ entries. From Figure 5(b), we can see that the number of swaps decrease significantly as n_b varies from 1 to 1000. Compared to standard cracking, buffered swapping saves about 4.5 million swaps for 1 buffered query and 73 million swaps for 1000 buffered queries and a heap size of $1M$. The maximal size of the heap is proportional to the reduction in swaps. Furthermore, we can observe that the swap reduction for 1000 buffered queries improves only marginally over that of 100 buffered queries. This indicates that after 100 buffered queries the cracked column is already close to being fully sorted. Hybrid cracking performs even more swaps than standard cracking (including moving the qualifying entries from the initial partitions to the final partition).

Next let us see the actual runtimes of buffered swapping in comparison to standard cracking and hybrid crack sort. Figure 5(c) shows the result. We see that the total runtime grows rapidly as the number of buffered queries (n_b) increases. However, we also see that the query time after performing buffered swapping improves. For example, after performing buffered swapping with a maximal heap size of $1M$ for just 10 queries, the remaining 990 queries are 1.8 times faster than hybrid crack sort and even 5.5% faster than standard cracking. This shows that buffered swapping helps to converge better by reducing the number of swaps in subsequent queries. Interestingly, a larger buffer size does not necessarily imply a higher runtime. For 100 and $1,000$ buffered queries the buffered part is faster for a maximum heap size of $10M$ entries than for smaller heaps. This is because such a large heap size leads to an earlier convergence towards the full sorting. Nevertheless, the high runtime of initial buffer swapped queries is a concern. In our experiments we implemented buffered swapping using the gheap implementation [1] with a fan-out of 4. Each element that is inserted into a gheap has to sink down inside of the heap tree to get to its position. This involves pairwise swaps and triggers many cache-misses. Exploring more efficient buffering mechanisms in detail opens up avenues for future work.

3.2 Improving Tuple Reconstruction

Our goal in this section is to see how well sideways cracking [9] addresses the issue of tuple reconstruction and whether we can im-

[2]After the first few queries, cracking mostly performs a pair of crack-in-two operations as the likelihood of two splits falling in two different partitions increases with the number of applied queries.



(a) Variance in Response Time ($\alpha = 0.1$) (b) Initialization Time Tradeoff (c) Extending 6(c) by Nearby Clustering

Figure 7: Comparing Robustness of Standard Cracking, Stochastic Cracking, Coarse-granular Index, and Full Index

prove upon it. Let us first see how the sideways cracking from Figure 4(b) scales with the number of attributes. Figure 6(a) shows the performance of sideways cracking for the number of projected attributes varying from 1 to 10. We see that in contrast to standard cracking (see Figure 3(b)), sideways cracking scales more gracefully with the number of projected attributes. However, still the performance varies by up to one order of magnitude. Furthermore, sideways cracking duplicates the index key in all cracker maps. So the question now is, can we have a cracking approach which is less sensitive to the number of projected attributes?

covered-cracking: *group multiple non-key attributes with the cracked column in a cracker map. At query time, crack all covered non-key attributes along with the key column for even more efficient tuple reconstruction.*

Note that with sideways cracking all projected columns are aligned with each other. However, the query engine still needs to fetch the projected attribute values from different columns in different cracker maps. These lookup costs turn out to be very expensive in addition to the overhead of cracking n replicas of the indexed column for n projected attributes. To solve this problem, we generalize sideways cracking to cover the n projected attributes in a single cracker map. In the following we term this approach *covered cracking*. While cracking, all covered attributes of a cracker map are reorganized with the cracked column. As a result, all covered attributes are aligned and stored in a consecutive memory region, i.e. no additional fetches are involved if the accessed attribute is covered. However, the drawback of this approach is that we need to specify which attributes to cover. If we want to be on the safer side, we may cover all table attributes. However, this means that we will need to copy the entire table for indexing and we might cover unrequested columns. On the other hand, if the set of covered attributes is too small or poorly chosen, external lookups might be triggered. Thus, choosing a suitable set of covered attributes based on the observed workload is crucial. The decision which attributes to cover is similar to computing the optimal vertical partitioning for a table. Various applicable algorithms are presented in [12].

Figure 6(b) shows the performance of covered cracking over different numbers of projected attributes. Here we show the results from covered cracking which copies the data of all covered attributes in the beginning. We can see that covered cracking remains stable when varying the number of projected attributes from 1 to 10. Thus, covered cracking scales well with the number of attributes. Figure 6(c) compares the accumulated costs of standard, sideways, and covered cracking. We can see that while the accumulated costs of standard and sideways cracking grow linearly with the number of attributes, the accumulated costs of covered cracking remain pegged at under 40 seconds. We also see that sideways cracking outperforms covered cracking for only up to 4 projected attributes. For more than 4 projected attributes, sideways cracking becomes increasingly expensive whereas covered cracking remains stable. Thus, we see that covering offers huge benefits.

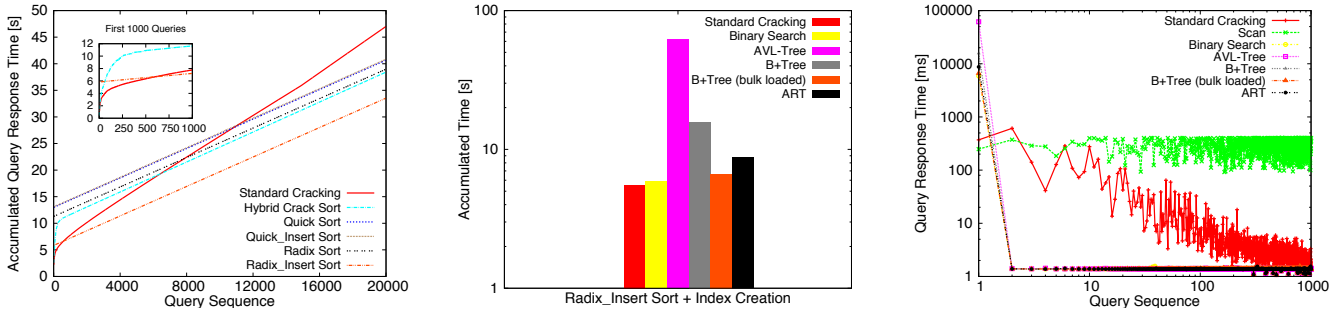
3.3 Improving Cracking Robustness

In this section we look at how well stochastic cracking [6] addresses the issue of query robustness and whether we can improve upon it. In Figure 4(c) we can observe that stochastic cracking is more expensive (for first as well as subsequent queries) than standard cracking. On the other hand, the random splits in stochastic cracking (MDD1R) create uniformly sized partitions. Thus, stochastic cracking trades performance for robustness. So the key question now is: can we achieve robustness without sacrificing performance? Can we have high robustness and efficient performance at the same time?

coarse-granular index: *create balanced partitions using range partitioning upfront for more robust query performance. Apply standard cracking later on.*

Stochastic cracking successively refines the accessed data regions into smaller equal sized partitions while the non-accessed data regions remain as large partitions. As a result, when a query touches a non-accessed data region it still ends up shuffling large portions of the data. To solve this problem, we extend stochastic cracking to create several equal-sized[3] partitions upfront, i.e. we pre-partition the data into smaller range partitions. With such a *coarse-granular index* we shuffle data only inside a range partition and thus the shuffling costs are within a threshold. Note that in standard cracking, the initial queries have to anyways read huge amounts of data, without gathering any useful knowledge. In contrast, the coarse granular index moves some of that effort to a prepare step to create meaningful initial partitions. As a result, the cost of the first query is slightly higher than standard cracking but still significantly less than full indexing. With such a coarse-granular index users can choose to allow the first query to be a bit slower and witness stable performance thereafter. Also, note that the first query in standard cracking is anyways slower than a full scan since it partitions the data into three parts. Coarse-granular index differs from standard cracking in that it allows for creating *any* number of initial partitions, not necessarily three. Furthermore, by varying the number of initial partitions, we can trade the initialization time for more robust query performance. This means that, depending upon their application, users can adjust the initialisation time in order to achieve a

[3]Please note that our current implementation relies on a uniform key distribution to create equal-sized partitions. Handling skewed distributions would require the generation of equi-depth partitions.



(a) Comparing Different Sort Algorithms (b) Indexing Effort of Diff. Indexes (c) Per-Query Response Time of Diff. Indexes

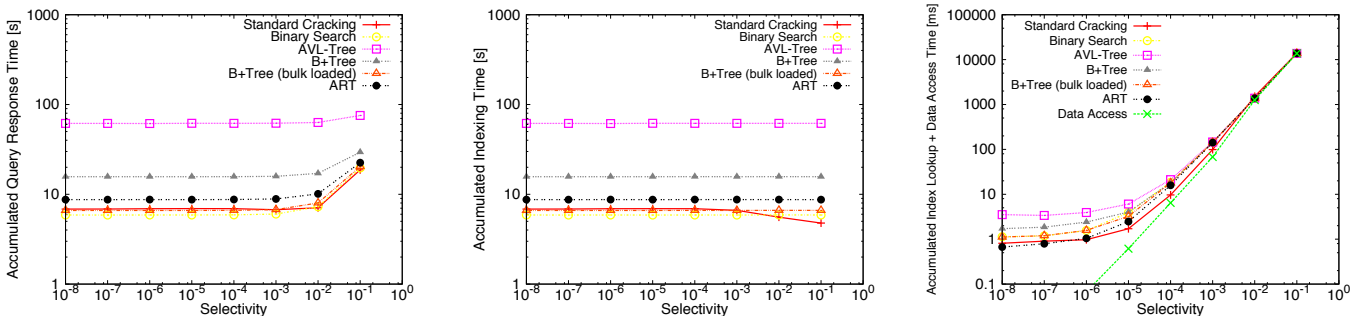
Figure 8: Comparing Standard Cracking with Different Sort and Index Baselines

corresponding robustness level. This is important in several scenarios in order to achieve customer SLAs. In the extreme case, users can create as many partitions as the number of distinct data items. This results in a full index, has a very high initialisation time, and offers the most robust query performance. The other extreme is to create only a single initial partition. This is equivalent to standard cracking scenario, i.e. very low initialisation time and least robust query performance. Thus, coarse-granular index covers the entire robustness spectrum between standard cracking and full indexing.

Figure 7(a) shows the variance in query response time of different indexing methods, including stochastic cracking (MDD1R), coarse-granular index, and full index (quick sort + binary search), computed in the same way as in Figure 3(c). We vary the number of initial partitions, which are created in the first query by the coarse-granular index from 10 to $100,000$. While stochastic cracking (MDD1R) shows a variance similar to that of standard cracking, as observed in Figure 3(c), coarse-granular index reduces the performance variance significantly. In fact, for different number of partitions, coarse-granular index covers the entire space between the high-variance standard cracking and low-variance full index. Figure 7(b) shows the results. We can see that the initialisation time of stochastic cracking (MDD1R) is very similar to that of standard cracking. This means that stochastic cracking (like standard cracking) shifts most of the indexing effort to the query time. On the other extreme, full sort does the entire indexing effort upfront, and thus has the highest initialisation time. Coarse-granular index fills the gap between these two extremes, i.e. by adjusting the number of initial partitions we can trade the indexing effort at the initialisation time and the query time. For instance, for $1,000$ initial partitions, the initialization time of coarse-granular index is 65% less than full index, while still providing more robust as well as more efficient query performance than stochastic cracking (MDD1R). In fact, the total query time of coarse-granular index with $1,000$ initial partitions is 41% less than stochastic cracking (MDD1R) and even 26% less than standard cracking. Thus, coarse-granular index allows us to combine the best of both worlds.

We can also extend the coarse-granular index and pre-partition the base table along with the cracker column. This means that we range partition the source table in exactly the same way as the adaptive index during the initialisation step. Though, we still refine only the cracked column for each incoming query. The source table is left untouched. If the partition is small enough to fit into the cache, then the tuple reconstruction costs are negligible because of no cache misses. Essentially, we decrease the physical distance between external random accesses, i.e. the index entry and the corresponding tuple are *nearby clustered*. This increases the likelihood that tuple reconstruction does not incur any cache misses. Thus, as a result of pre-partitioning the source table, we can achieve more robust tuple reconstruction without covering the adaptive index itself, as in covered cracking in Section 3.2. However, we need to pick the partition size judiciously. Larger partitions do not fit into the cache, while smaller partitions result in high initialisation time. Note that if the data is stored in row layout, then the source table is anyways scanned completely during index initialisation and so pre-partition is not too expensive. Furthermore, efficient tuple reconstruction using nearby clustering is limited to one index per table, same as for all primary indexes.

Figure 7(c) shows the effect of pre-partitioning the source table. We create both 100 and $1,000$ partitions. The cost of pre-partitioning the source table is included in the accumulated query response time of coarse-granular index. Both standard cracking and coarse-granular index in Figure 7(c) start with perfectly aligned tuples. However, in standard cracking, the locality between index entry and corresponding tuple decreases gradually and soon the cache misses caused by random accesses destroy the performance. Coarse-granular index, on the other hand, exploits the nearby clustering between the index entry and the corresponding tuple. Since tuples are never swapped across partitions, the maximum distance between an index entry and the corresponding tuple is at most the size of a partition. Thus, we can see from Figure 7(c) that coarse-granular index has a much more stable performance when scaling the number of projected attributes, without reorganising the base table at query time. In fact, coarse-granular index 1K even outperforms covered cracking for any number of projected attributes. For example, when projecting all 10 attributes, coarse-granular index 1K is 1.7 times faster than covered cracking, 3.7 times faster than sideways cracking, and 4.3 times faster than standard cracking. However, for $1,000$ table partitions, each partition has a size of 8MB and thus fits completely in the CPU cache. For 100 partitions the partition size increases to 80MB and thus, it is over 6.5 times larger than the cache. The results show that the concept still works. Although coarse-granular index 100 is slower than covered cracking for more than 4 attributes, it is still faster than sideways and standard cracking for more than 3 attributes. It holds: the fewer partitions that we create the closer is the performance to that of standard cracking. To strengthen the robustness evaluation, we scale all experiments from Figure 7 to a dataset containing 1 billion entries. As we want to inspect how well the methods scale with the data size, Table 2 shows the factor of increase in time when switching from 100 million to 1 billion entries. For an increase in data size by factor 10, an algorithm that scales linearly is 10 times slower. Obviously, all tested methods scale very well. As expected, only nearby clustering suffers from larger partitions which exceed the cache size by far. Overall, we see that coarse-granular index offers more robust query performance both over arbitrary selection predicates as well as over arbitrary projection attributes.



(a) Accumulated Query Response Time (b) Accumulated Indexing Time (c) Acc. Index Lookup + Data Access Time

Figure 9: Comparing Standard Cracking with Index Baselines while Varying Selectivity (Note that: (a) = (b) + (c))

FACTOR SLOWER (FROM 100M to 1B)	INITIALIZATION	REMAINING	TOTAL
STANDARD CRACKING	10.01	9.92	9.93
STOCHASTIC CRACKING (MDD1R)	12.92	9.57	9.75
COARSE GRANULAR INDEX 10	11.73	9.92	10.56
COARSE GRANULAR INDEX 100	11.72	9.81	10.79
COARSE GRANULAR INDEX 1K	11.69	9.96	11.09
COARSE GRANULAR INDEX 10K	11.31	9.94	10.95
COARSE GRANULAR INDEX 100K	10.90	10.02	10.73
FULL INDEX	11.48	9.97	11.29
SIDEWAYS CRACKING	-	-	11.92
COVERED CRACKING	-	-	9.98
COARSE GRANULAR INDEX 100 (NEARBY CLUSTERED)	-	-	11.64
COARSE GRANULAR INDEX 1K (NEARBY CLUSTERED)	-	-	13.33

Table 2: Scalability under Datasize Increase by Factor 10

Finally, Table 3 classifies the three cracking extensions discussed above — buffered swapping, covered cracking, and coarse-granular index — along the same dimensions as discussed in Section 2.6. Please note that the entry of coarse-granular index classifies only the initial partitioning step as it can be combined with various other cracking methods as well.

DIMENSIONS	CATEGORY	NO INDEX	BUFFERED SWAPPING	COVERED CRACKING	COARSE GRANULAR INDEX	FULL INDEX
NUMBER OF SPLIT LINES	ZERO	■				
	FEW		■	■		
	SEVERAL				■	
	ALL		■			■
SPLIT STRATEGY	NONE	■				
	QUERY BASED		■	■		
	RANDOM					
	DATA BASED		■	■	■	■
SPLIT TIMING	NEVER	■				
	PER QUERY		■	■		
	UPFRONT				■	■

Table 3: Classification of Extended Cracking Algorithms.

4. EXTENDING CRACKING EXPERIMENTS

In this section, we compare cracking with different sort and index baselines in detail. Our goal here is to understand how good or bad cracking is in comparison to different full indexing techniques. In the following, we first consider different sort algorithms, then different index structures, and finally the effect of query selectivity.

4.1 Extending Sorting Baselines

The typical baseline used in previous cracking works was a full index wherein the data is fully sorted using quick sort and queries are processed using binary search to find the qualifying tuples. Sorting is an expensive operation and as a result the first fully sorted query is up to 30 times slower than the first cracking query (See Figure 2(b)). So let us consider different sort algorithms.

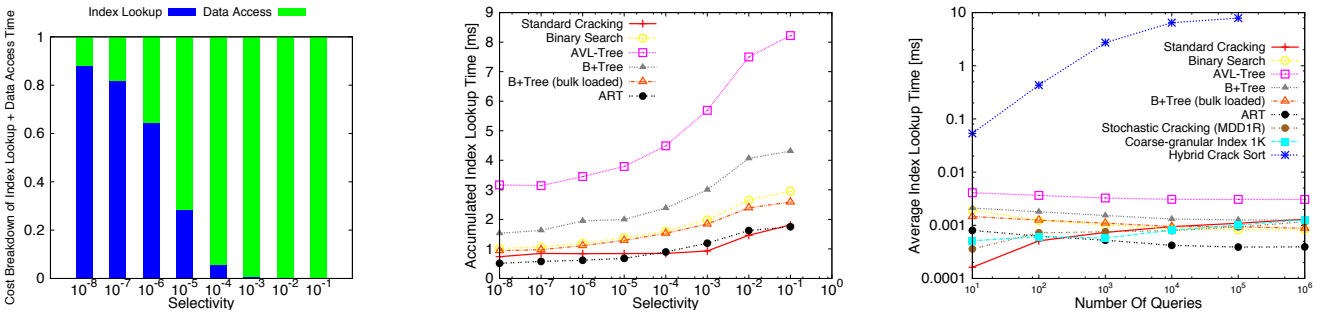
Quick sort is a reasonably good (and cache-friendly) algorithm, better than other value-based sort algorithms such as insertion sort and merge sort. But what about radix-based sort algorithms [7]? We compared quick sort with an in-place radix sort implementation [2]. This recursive radix sort implementation switches to insertion sort (lets call this radix_insert) when the run length becomes smaller than 64. We applied a similar switching to quick sort as well (lets call it quick_insert). Figure 8(a) shows the accumulated query response times for binary search in combination with several sorting algorithms. We compare these with standard cracking and hybrid crack sort. The initialization times (i.e. the time to sort) for quick sort, quick_insert sort, and pure radix sort around 10 seconds are included in the first query. However, the initialisation time for radix_insert sort drops by half to around 5 seconds. As a result, the first query with radix_insert is only 14 times slower, compared to 30 times slower with quick sort, than the first standard cracking query. Furthermore, we can clearly identify the number of queries at which one methods pays off over another. Already after 600 queries radix_insert sort shows the smaller accumulated query response times than standard cracking. For the two quick sort variants it takes 12,000 queries to beat standard cracking.

4.2 Extending Index Baselines

Let us now consider different index structures and contrast them with simple binary search on sorted data. The goal is to see whether or not it makes sense to use a sophisticated index structure as a baseline for cracking. We consider three index structures: (i) AVL-Tree, (ii) B+-Tree, and (iii) the very recently proposed ART [15]. We choose ART since it outperforms other main-memory optimised search trees such as CSB+-Tree [17] and FAST [14].

Let us first see the total indexing effort of different indexing methods over 1000 queries. For binary search, we simply sort the data (radix_insert sort) while for other full indexing methods (i.e. AVL-Tree, B+-Tree, and ART) we load the data into an index structure in addition to sorting (radix_insert sort). Standard cracking self-distributes the indexing effort over the $1,000$ queries while the remaining methods perform their sorting and indexing work in the first query. For the B+-Tree we present two different variants: one that is bulk loaded and one that is tuple-wise loaded. Figure 8(b) shows the results. We can see that AVL-Tree is the most expensive while standard cracking is the least expensive in terms of indexing effort. The indexing efforts of binary search and B+-Tree (bulk loaded) are very close to standard cracking. However, the other B+-Tree as well as ART do more indexing effort, since both of them load the index tuple-by-tuple[4]. The key thing to note here is that bulk loading an index structure adds only a small overhead to the pure sorting. Let us now see the query performance of the different index structures. Figure 8(c) shows the per-query response times of different indexing methods. Surprisingly, we see

[4]The available ART implementation does not support bulk loading.



(a) Cost Breakdown of Index Lookup and Data Access Time of ART

(b) Accumulated Index Lookup Time in Isolation

(c) Average Index Lookup Time (Sel. 10^{-8})

Figure 10: Lookup and Data Access of Standard Cracking and Index Baselines under Variation of Selectivity

that using a different index structure has barely an impact on query performance. This is contrary to what we expected and in the following let us understand this in more detail.

4.3 Effect of Varying Selectivity

To better understand this effect let us now vary the tuple selectivity of queries. Recall that we used a selectivity of 1% in all previous experiments. Selectivity is given as fraction of all entries. Figure 9(a) shows the accumulated query response times of different methods when varying the selectivity. We can see that the accumulated query response times change over varying selectivity for standard cracking, binary search, B+-Tree (bulk loaded), and ART. However, there is little relative difference between these methods over different selectivities. To dig deeper, let us split the query response time into two components: (i) the indexing costs to sort the data and to build the structure, and (ii) the index lookup and data access costs to retrieve the result.

Figure 9(b) shows the accumulated indexing time for different methods when varying selectivity. Obviously, the indexing time is constant for all full indexing methods. However, the total indexing time of standard cracking changes over varying query selectivity. In fact, the indexing effort of standard cracking decreases by 45% when the selectivity changes from 10^{-5} to 10^{-1}. As a result, the indexing effort by standard cracking surpasses even the effort of binary search (more than 18%) and B+-Tree (bulk loaded) (more than 5%), both based on radix_insert sort for as little as $1,000$ queries. The reason standard cracking depends on selectivity is that with high selectivity the two partition boundaries of a range query are located close together and the index refinement per query is small. As a result several data items are shuffled repeatedly over and over again. This increases the overall indexing effort as well as the time to converge to a full index.

Figure 9(c) shows the accumulated index lookup and data access costs of different methods over varying selectivity. We can see that the difference in the querying costs of different methods grows for higher selectivity. For instance, AVL-Tree is more than 5 times slower than ART for a selectivity of 10^{-8}. We also see that standard cracking is the most lightweight method in terms of the index lookup and data access costs and is closely followed by ART. However, for high selectivities, the index lookup and data access costs are small compared to the indexing costs. As a result, the difference in the index lookup and data access costs of different methods is not reflected in the total costs in Figure 9(a).

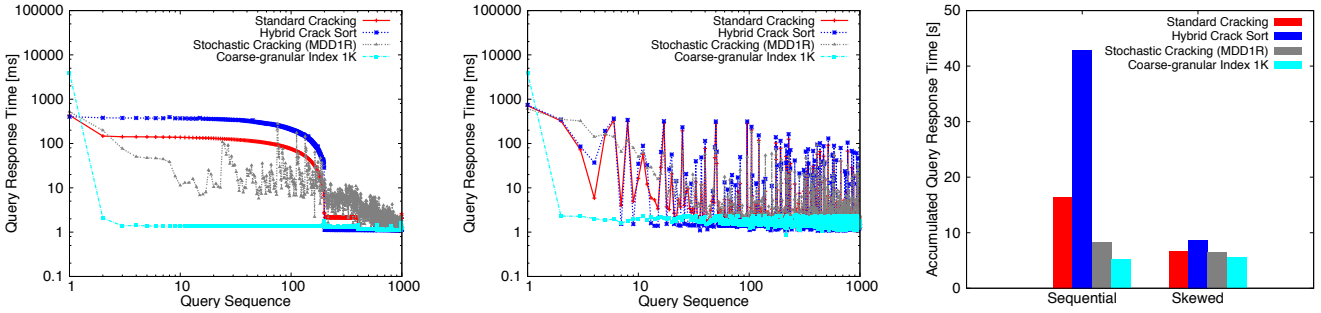
Let us now investigate the index lookup costs and the data access costs in comparison. Figure 10(a) shows the index lookup and data access costs as a fraction of the costs of Figure 9(c) for ART. We can see that the data access costs dominate the total time for a selectivity lower than 10^{-6}. This means that using better optimised index structures make sense only for very high query selectivities. Figure 10(b) shows only the index lookup costs without the data access costs of different methods when varying selectivity. We can see that the index lookup costs vary with selectivity, indicating different cache behaviour for different query selectivities. Furthermore, we see that ART has the best index lookup times. For a selectivity of 10^{-8}, ART performs 30% faster index lookups than standard cracking. This is even though cracking has a much smaller index (created over just $1,000$ queries) whereas ART creates a full index over $100M$ data entries.

Finally, we also vary the number of queries to see how the index lookup times of standard cracking change in comparison to other indexing methods. Figure 10(c) shows the average per-query index lookup times for different methods when increasing the number of queries in the query sequence from 10 to $1M$. We fix the query selectivity to 10^{-8}. Furthermore, we show stochastic cracking (MDD1R), coarse granular index 1K, and hybrid crack sort as they introduce additional split lines or handle them differently. We can see that the average index lookup time of standard cracking increases by almost one order of magnitude when the number of queries increase from 10 to $1M$. This is because the cracker index grows with the number of queries[5]. Coarse-granular index and stochastic cracking (MDD1R) differ from standard cracking by showing a higher average index lookup time in the beginning as the additional splits weigh in that early phase. Hybrid crack sort shows the overall highest average lookup time which even increases with the number of queries[6]. The high selectivity leads to slow convergence and triggers repeatedly expensive lookups into the $10,000$ initial partitions. In contrast to that, the average per-query index lookup time of other indexing methods remains stable (or even improves slightly due to better cache performance) with the number of queries. Consequently, for $1M$ queries, the average index lookup time of ART is 3.6 times smaller than the average index lookup time of standard cracking.

To conclude, the take-away message from this section is threefold: (i) using a better index structure makes sense only for very high selectivities, e.g. one in a million, (ii) cracking depends on query selectivity in terms of indexing effort, and (ii) although cracking creates the indexes adaptively, it still needs to catch up with full indexing in terms of the *quality* of the index.

[5][8] proposed to stop cracking if a sufficiently small partition size is reached. However, this pays off only for a very large number of queries. As we apply $1,000$ queries in nearly all experiments, we use unbounded algorithms throughout this paper.

[6]The point for 10^6 queries is missing as the space consumption of $10K$ AVL-Trees each storing up to $1M$ entries exceeds capacity.



(a) Sequential Access Pattern (b) Skewed Access Pattern (c) Accumulated Query Response Times

Figure 11: Effect of Query Access Pattern on Adaptive Methods

4.4 Effect of Query Access Pattern

So far, all experiments applied a uniform random access pattern to test the methods. However, in reality, queries are often logically connected with each other and follow certain non-random and non-uniform patterns. To evaluate the methods under such patterns, we pick two representatives: the *sequential access pattern* and the *skewed access pattern*.

We create the sequential access pattern as follows: starting from the beginning of the value domain, the queried range is moved for each query by half of its size towards the end of the domain to guarantee overlapping query predicates. When the end is reached, the query range restarts from the beginning. The position to begin is randomly set in the first 0.01% of the domain to avoid repetition of the same sequence in subsequent rounds. Figure 11(a) shows the query response time under the sequential access pattern for standard cracking, stochastic cracking, coarse-granular index with 1,000 partitions, and hybrid crack sort. We can clearly separate the figure into the first 200 queries and the remaining 800 queries. As the selectivity is 1% and the query range moves by half of its size per query, it takes 200 queries until the entire data set has been accessed. Within that period the query response time of standard cracking and hybrid crack sort decreases only gradually. Large parts of the data are scanned repeatedly and the unindexed upper part decreases only slightly per query. Furthermore, hybrid crack sort is considerably slower than standard cracking in this phase. Stochastic cracking reduces this problem significantly by applying additional splits to the unindexed upper area. Coarse-granular index shows the most stable performance. After the initial partitioning in the first query, the query response time does not significantly vary. Additionally, the query response time is the lowest of all methods (except for the first query). For the remaining 800 queries the performance differences between all methods decrease as the entire data set has been queried and is therefore cracked more or less. Now, stochastic cracking is slower than standard cracking as the additional effort of random cracking and materializing the result is no more necessary to provide a decent performance.

Finally, let us investigate how the methods perform under a skewed access pattern. We create the skewed access pattern in the following way: first, a zipfian distribution is generated over n values, where n corresponds to the number of queries. Based on that distribution the domain around the hotspot, which is the middle of the domain in our case, is divided into n parts. After that the query predicates are set according to the frequencies in the distribution. The k values with the l highest frequency in the distribution lead to k query predicates lying in the l-th nearest area around the hotspot. Figure 12 shows the generated predicates for $\alpha = 2.0$. These predicates are randomly shuffled before they are used in the query sequence. Figure 11(b) shows the query response time for the skewed pattern. We can observe a high variance in all methods except coarse-granular index.

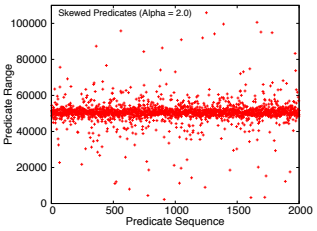


Figure 12: Skewed Predicates

Between accessing the hotspot area and regions that are far off, the query response time varies by almost 3 orders of magnitude. Early on, all methods index the hotspot area heavily as most query predicates fall into that region. Stochastic cracking manages to reduce the negative impact of predicates that are far off the hotspot area. However, it is slower than standard cracking if the hotspot area is hit. Hybrid crack sort copies the hotspot area early on to its final partition and exhibits the fastest query response times in the best case. However, if a predicate requests a region that has not been queried before, copying from the initial partitions to the final partition is expensive.

Finally, Figure 11(c) shows the accumulated query response time for both sequential and skewed access patterns. Obviously handling sequential patterns is challenging for all adaptive methods. Especially hybrid crack sort suffers from large repetitive scans in all initial partitions and is therefore by far the slowest method in this scenario. Stochastic cracking (MDD1R) manages to reduce the problems of standard cracking significantly and thus fulfills its purpose by providing a workload robust query answering. In total, coarse-granular index is the fastest method under this pattern. Overall, for the skewed access pattern the difference between the methods is significantly smaller than for the sequential pattern.

5. LESSONS LEARNED & CONCLUSION

Let us now put together the major lessons learned in this paper.

1. Database cracking is a mature field of research. Database cracking is a simple yet effective technique for adaptive indexing. In contrast to full indexing, database cracking is lightweight, i.e. it does not penalise the first query heavily. Rather, it incrementally performs at most one quick sort step for each query and nicely distributes the indexing effort over several queries. Moreover, database cracking indexes only those data regions which are actually touched by incoming queries. As a result, database cracking fits perfectly to the modern needs of adaptive data management. Furthermore, apart from the incremental index creation in standard cracking, several other follow-up works have looked into other aspects of adaptive indexing as well. These include updating a cracked database, convergence of database cracking to a full index, efficient tuple reconstruction, and robustness over unpredictable changes in query workload. Thus, we can say that database cracking has come a long way and is a mature field of research.

2. Database cracking is repeatable. In this paper, we repeated four previous database cracking works, including standard cracking using crack-in-two and crack-in-three [8], hybrid cracking [11], sideways cracking [9], and stochastic cracking [6]. We reimplemented the algorithms from each of these works and tested them under similar settings as in the previous works. Our results match very closely to the results presented in the previous works and we can confirm the findings of those works, i.e. hybrid cracking indeed improves in terms of convergence to full index, sideways cracking allows for more efficient tuple reconstruction, and stochastic cracking offers more predictable query performance than standard cracking. Thus, we can say that database cracking is repeatable in any ad-hoc query engine, other than MonetDB as well.

3. Still, lot of potential to improve database cracking. Several aspects of database cracking are still improvable, including faster convergence to full index, more efficient tuple reconstruction, and more robust query performance. For example, by buffering the elements to be swapped in a heap, we can reduce the number of swaps and thus have better convergence. Similarly, by covering the cracked index we can achieve better scalability in the number of projected attributes. Likewise, we can trade the initialisation time to create a coarse-granular index which improves the query robustness. Based on these promising directions, we believe that even though cracking has come a long way, it still has a lot more to go.

4. Database cracking depends heavily on the query access pattern. As the presented techniques are adaptive due to their query driven nature, each of them is more of less sensitive to the applied query access pattern. A uniform random access pattern can be considered the best case for all methods as it leads to uniform partition sizes. In contrast to that sequential patterns crack the index in small steps and the algorithms rescan large parts of the data. Skewed patterns lead to a high variance in runtime depending on whether the predicate hits the hotspot area or not. Overall, stochastic cracking and coarse-granular index, which extend their query driven character by data driven influences, are less sensitive to the access pattern than the methods that take only the seen queries into account.

5. Database cracking needs to catch up with modern indexing trends. We saw that for sorting radix sort is twice as fast as quick sort. After 600 queries the total query response time of binary search based on radix sorted data is even faster than standard cracking. This means that a full sorting pays-off over standard cracking in less than 1000 queries. Thus, we need to explore more lightweight techniques for database cracking to be competitive with radix sort. Furthermore, several recent works have proposed main-memory optimised index structures. The recently proposed ART has 1.8 times faster lookups than standard cracking after 1000 queries and 3.6 times faster lookups than standard cracking after $1M$ queries. We note two things here: (i) the cracker index offers much slower lookups than modern main-memory indexes, and (ii) the cracker index gets even worse as the number of queries increase. Thus, we need to look into the index structures used in database cracking and catch up with modern indexing trends.

6. Different indexing methods have different signatures. We looked at several indexing techniques in this paper. Let us now contrast the behaviour of different indexing methods in a nutshell. To do so, we introduce a way to fingerprint different indexing methods. We measure the *progress of index creation* over the *progress of query processing*, i.e. how different indexing methods index the data over time as the queries are being processed. This measure acts as a signature of different indexing methods. Figure 13 shows the indexing progress against the querying progress of different methods. The x-axis shows the normalised accumulated lookup and data

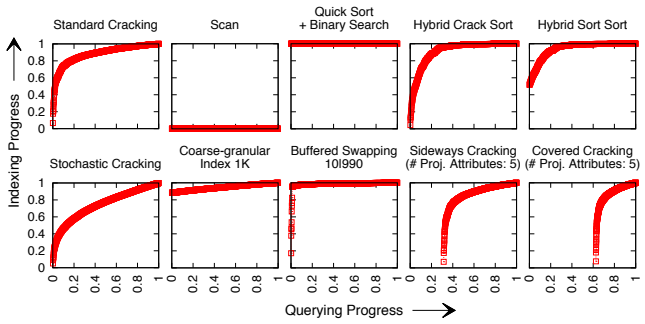


Figure 13: Signatures of Indexing Methods.

access time (the querying progress) and the y-axis shows the normalised accumulated data shuffle and index update time (the indexing progress). Obviously, different indexing methods have different curves. For example, standard cracking gradually builds the index as the queries are processed whereas full index (quick sort + binary search) builds the entire index before processing any queries. Hybrid crack sort and hybrid sort sort have steeper curves than standard cracking, indicating that they build the index more quickly. On the other hand, stochastic cracking has a smoother curve. Sideways and covered cracking perform large parts of their querying process already in the first query by copying table columns into the index to speed up tuple reconstruction. It is interesting to see that each method has a unique curve which characterises its indexing behaviour. Furthermore, there is still room to design adaptive indexing algorithms with even more different indexing signatures.

Acknowledgments. We would like to thank Stratos Idreos for helping us to understand the hybrid cracking algorithms. Research partially supported by BMBF.

6. REFERENCES

[1] Generalized Heap Implementation. https://github.com/valyala/gheap.
[2] O. R. Birkeland. Searching Large Data Volumes with MISD Processing. PhD thesis.
[3] T. Finch. Incremental calculation of weighted mean and variance. University of Cambridge Computing Service, 2009.
[4] G. Graefe, F. Halim, S. Idreos, et al. Concurrency Control for Adaptive Indexing. In *PVLDB*, pages 656–667, 2012.
[5] G. Graefe and H. Kuno. Self-selecting, Self-tuning, Incrementally Optimized Indexes. In *EDBT*, pages 371–381, 2010.
[6] F. Halim, S. Idreos, et al. Stochastic Database Cracking: Towards Robust Adaptive Indexing in Main-Memory Column-Stores. In *PVLDB*, pages 502–513, 2012.
[7] P. Hildebrandt and H. Isbitz. Radix Exchange - An Internal Sorting Method for Digital Computers. *J. ACM*, pages 156–163, 1959.
[8] S. Idreos et al. Database Cracking. In *CIDR*, pages 68–78, 2007.
[9] S. Idreos, M. Kersten, et al. Self-organizing Tuple Reconstruction In Column-stores. In *SIGMOD*, pages 297–308, 2009.
[10] S. Idreos, M. Kersten, and S. Manegold. Updating a Cracked Database. In *SIGMOD*, pages 413–424, 2007.
[11] S. Idreos, S. Manegold, H. Kuno, and G. Graefe. Merging What's Cracked, Cracking What's Merged: Adaptive Indexing in Main-Memory Column-Stores. In *PVLDB*, pages 585–597, 2011.
[12] A. Jindal, E. Palatinus, V. Pavlov, and J. Dittrich. A Comparison of Knives for Bread Slicing. In *PVLDB*, pages 361–372, 2013.
[13] M. Kersten et al. Cracking the Database Store. In *CIDR*, pages 213–224, 2005.
[14] C. Kim et al. FAST: Fast Architecture Sensitive Tree Search on Modern CPUs and GPUs. In *SIGMOD*, pages 339–350, 2010.
[15] V. Leis et al. The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases. In *ICDE*, pages 38–49, 2013.
[16] X. Martinez-Palau, D. Dominguez-Sal, and J. L. Larriba-Pey. Two-way Replacement Selection. In *PVLDB*, pages 871–881, 2010.
[17] J. Rao and K. A. Ross. Making B+-Trees Cache Conscious in Main Memory. In *SIGMOD*, pages 475–486, 2000.