The VLDB Journal (2016) 25:27–52
DOI 10.1007/s00778-015-0397-y

SPECIAL ISSUE PAPER

An experimental evaluation and analysis of database cracking

Felix Martin Schuhknecht[1] · Alekh Jindal[2] · Jens Dittrich[1]

Received: 19 December 2014 / Revised: 18 May 2015 / Accepted: 13 July 2015 / Published online: 22 August 2015

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. Our 2014 VLDB paper "The Uncracked Pieces in Database Cracking" (PVLDB 7:97–108, 2013/VLDB 2014) was the first comparative study of these different methods by an independent group. In this article, we extend our published experimental study on database cracking and bring it to an up-to-date state. 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. Additionally to our conference paper, we now also look at a recently published study about CPU efficiency (predication cracking). We evaluate these works and show possible directions to do even better. As a further extension, we evaluate the whole class of parallel cracking algorithms that were proposed in three recent works. Altogether, in this work we revisit 8 papers on database cracking and evaluate in total 18 cracking methods, 6 sorting algorithms, and 3 full index structures. Additionally, we test cracking under a variety of experimental settings, including high selectivity (*Low* selectivity means that *many* entries qualify. Consequently, a *high* selectivity means, that only *few* entries qualify) queries, low selectivity queries, varying selectivity, and multiple query access patterns. Finally, we compare cracking against different sorting algorithms as well as against different main memory optimized indexes, including the recently proposed adaptive radix tree (ART). Our results show that: (1) the previously proposed cracking algorithms are repeatable, (2) there is still enough room to significantly improve the previously proposed cracking algorithms, (3) parallelizing cracking algorithms efficiently is a hard task, (4) cracking depends heavily on query selectivity, (5) cracking needs to catch up with modern indexing trends, and (6) different indexing algorithms have different indexing signatures.

Keywords Adaptive indexing · Database cracking · Sorting · Multi-threaded algorithms

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 [8,11,14–18]. Since the release

✉ Felix Martin Schuhknecht
felix.schuhknecht@infosys.uni-saarland.de

[1] Information Systems Group, Saarland University, Saarbrücken, Germany

[2] CSAIL, MIT, Cambridge, MA, USA

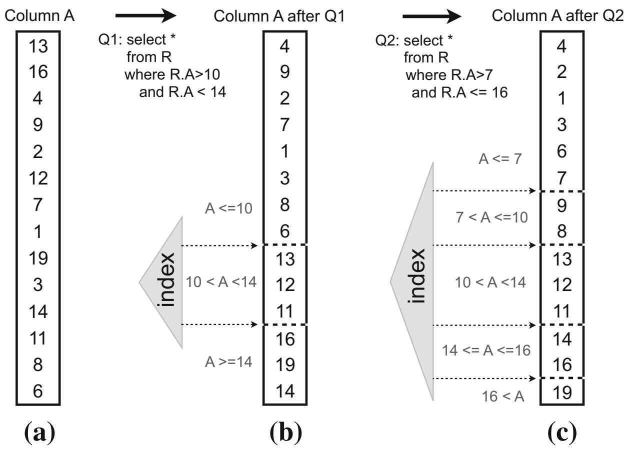


Fig. 1 Database cracking example

of our conference paper [25] on which this work builds upon, three more studies have been published [2,9,23]. 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 reorganization which eventually, over several queries, leads to a full index. Figure 1 recaps and visualizes the concept.

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 [14], tuple reconstruction [15], convergence [16], concurrency control [8,9], and robustness [11]. In this paper, we critically review database cracking in several aspects. We repeat the core cracking algorithms, i.e. crack-in-two and crack-in-three [17], as well as three advanced cracking algorithms [11,15,16]. We identify the weak spots in these algorithms and discuss extensions to fix them. Additionally, we inspect a recently published work [23], which identifies CPU efficiency problems in the standard cracking algorithm and proposes alternatives. Furthermore, we investigate the current state-of-the-art in parallel cracking algorithms [2,8,9,23] and compare them against each other. Finally, we also extend the experimental parameters previously used in database cracking, e.g. by varying the query selectivities and by comparing against more recent, main memory optimized indexing techniques, including ART [20].

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 possibly 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 [17], 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. Further, we identify four major concerns, namely CPU efficiency, convergence, tuple reconstruction, and robustness. In addition, we evaluate advanced cracking algorithms, namely predication cracking [23], hybrid cracking [16], sideways cracking [15], and stochastic cracking [11], 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 (Sect. 2).
2. *Extending cracking algorithms* In order to better understand the cracking behaviour, we modify three advanced cracking algorithms, namely hybrid cracking [16], sideways cracking [15], and stochastic cracking [11]. 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 (Sect. 3).
3. *Extending cracking experiments* As a next step, 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 [1] 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 [20] 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: (1) 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 (2) cracking creates more balanced partitions and hence converges faster for medium selectivities, i.e. around 10 %. We also look

at the effect of stopping the cracking process at a certain partition size. Furthermore, we apply a sequential and a skewed query access pattern and analyse 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 (Sect. 4).

4. *Parallelizing cracking algorithms* As exploiting modern hardware implies using the multi-threading capabilities of the system, we investigate in this section how the cracking algorithms can be parallelized. To do so, we first re-evaluate a lock-based parallel standard cracking algorithm [8,9] that serializes the crack-in-two operation at the granularity of partitions of data for inter-query parallelism. Additionally, we add the algorithms we proposed in our study on parallel adaptive indexing algorithms [2] and put the methods under a new set-up to the test. We include our parallel coarse-granular index that builds upon parallel standard cracking. We compare these methods to our intra-query parallel versions of standard cracking and coarse-granular index [2] that realize concurrency by dividing the column into chunks. We compare the parallel cracking algorithms with our two competitive parallel radix sort implementations [2] to evaluate the relation between cracking and sorting in a multi-threaded environment. Furthermore, we propose two realizations of parallel sideways cracking to put them to the test. Last but not least, we evaluate all parallel algorithms under skewed queries, skewed input, and clustered input (Sect. 5).
5. *Conclusion* Finally, we conclude by putting together the key lessons learned. Additionally, we also introduce *signatures* to characterize the indexing behaviour of different indexing methods and to understand as well as differentiate them visually (Sect. 6).

Experimental set-up We use a common experimental set-up throughout the paper. We try to keep our set-up 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 with a key range of $[0; 100,000]$. 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: (1) *scan* which reads the entire column and post-filters the qualifying tuples, and (2) *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 and supports Intel Turbo Mode. 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 a 64-bit linux with kernel 3.1.

2 Revisiting cracking

Let us start by revisiting the original cracking algorithm [17]. 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 versus 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 [17] 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. 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 centre of the column (since maximum swapping will occur) and symmetrical on either ends of the column. Figure 2a shows the results. Though both 2× crack-in-two and crack-in-three have maximum costs around the centre, 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 Fig. 2a is that even though 2× 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

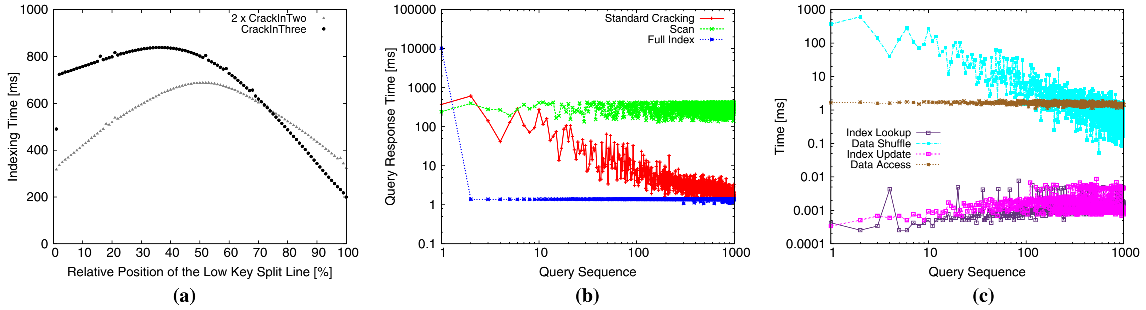


Fig. 2 Revisiting standard cracking. **a** Comparing single query indexing time. **b** Reproducing cracking behaviour. **c** Cost breakdown

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 2b 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 s compared to 0.24 s of full scan,[1] even though standard cracking invests some indexing effort. In contrast, full index takes 10.2 s 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 the 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 2c shows the cost breakdown of the query response time into four components: (1) *index lookup* costs to identify the partitions for cracking, (2) *data shuffle* costs of swapping the data items in the column, (3) *index update* costs for updating the index structure with the new partitions, and (4) *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 1000 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 1 μs, 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.

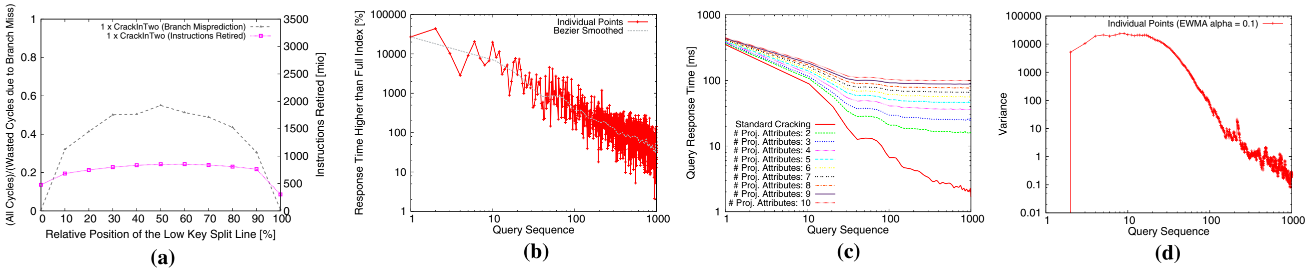
2.4 Key concerns in standard cracking

Let us now take a closer look at the standard cracking algorithm from four different perspectives, namely (1) CPU efficiency on modern hardware, (2) convergence to a full index, (3) scaling the number of projected attributes, (4) variance in query performance. Additionally, mapping cracking algorithms to parallel hardware is a challenging task. Therefore, we will spend an entire chapter on this.

1. *CPU efficiency* How an algorithm is mapped to the underlying hardware is crucial in memory resident data processing. Figure 3a shows the branch misprediction[2] as *the* weak spot of the crack-in-two algorithm with respect to the relative position of the split line, making this method clearly CPU bound. At a worst-case position of the split line dividing the partition in the middle, more than 50 % of the branches are predicted incorrectly.
2. *Cracking convergence* Convergence is a key concern and major criticism for database cracking. Figure 3b 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 1000 queries, on average, the query response time

[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] Measured with Intel VTune Amplifier 2015.



(a) (b) (c) (d)

Fig. 3 Key concerns in standard cracking. **a** CPU efficiency, **b** cracking convergence, **c** scaling projected attributes, **d** cracking variance

of standard cracking is still 40 % higher than that of full index.

3. *Scaling projected attributes* By default, database cracking leads to an unclustered index, i.e. extra lookups are needed to fetch the projected attributes. Figure 3c 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 1000 queries, querying 10 attribute tuples is almost 2 orders of magnitude slower than querying 1 attribute tuples.
4. *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 3d shows the variance of standard cracking query response times using the exponentially weighted moving average (EWMA). The variance is calculated as described in [6]. The degree of weighting decrease is $\alpha = 0.1$. We can see that unlike full index (see Fig. 2b), cracking does not exhibit stable query performance. Furthermore, we also see that the amount of variance for standard cracking decreases by five orders of magnitude.
5. *Cracking parallelization* The support of concurrency is crucial for performance on modern multi-core hardware. Therefore, the cracking algorithms must be extended to scale well with the available computing cores. As this is a challenging task, we will investigate this separately in Sect. 5.

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.

Predication and vectorized cracking *Decouple pivot comparison and physical reorganization by moving elements speculatively and correcting wrong decisions afterwards.*

The technique of predication cracking [23] directly attacks a major problem in standard cracking—excessive branch misprediction leading to large amounts of unnecessarily executed code. In contrast to standard cracking, where based on the outcome of the comparison of the element with the pivot, pointers are moved and elements are swapped, and predication cracking speculatively performs these reorganizations and interleaves them with the comparison evaluations of pivot and elements. When the result of the comparison is available, the incorrectly applied reorganizations are corrected. To ensure that the speculative writing does not cause data loss, the overwritten elements are backed up separately. This concept makes the algorithm completely branch free, and thus, the misprediction penalties do not longer exist. On the downside, the speculative writing adds an overhead compared to standard cracking. The question is now whether this trade-off can improve the runtime.

In predication cracking, the granularity at which data is backed up is fixed to a single element. Thus, the authors propose a generalization of the concept in form of *vectorized cracking*, where data is backed up and partitioned in larger blocks of adjustable size. This further decouples the costly backing up of data from the actual partitioning.

In Fig. 5, we add both predication cracking as well as vectorized cracking with a vector size of 128 B, which showed the best results in our evaluation, to the plot of Fig. 3a. Compared to standard cracking, the problem of branch misprediction vanishes almost entirely. However, we can also observe that the number of retired instructions drastically increases over the standard version. Thus, the concept of predication basically trades in a higher reorganization effort for less branching penalties. Vectorized cracking reduces this overhead by using larger blocks, resulting in a lower number of retired instructions while maintaining a negligible branch misprediction. This observed trade-off already indicates that the actual runtimes between standard and predication/vectorized cracking might be close to each other. Figure 4a shows the result when extending the study of Fig. 2a with predication and vectorized cracking. Unfortunately, although vectorized cracking performs slightly better than predication cracking in all cases, both methods do not

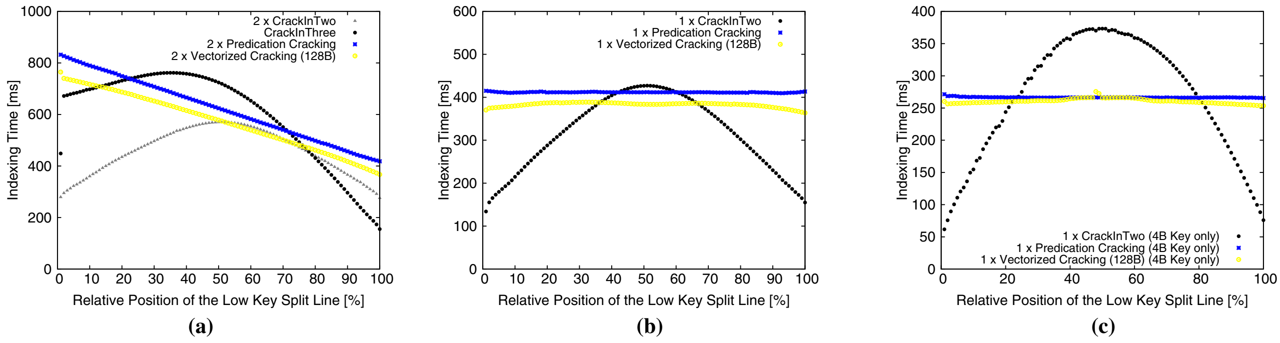


Fig. 4 Standard cracking in comparison with predication cracking and vectorized cracking. **a** Single query indexing time, **b** single crack indexing time on 16 B elements, **c** single crack indexing time on 4 B elements

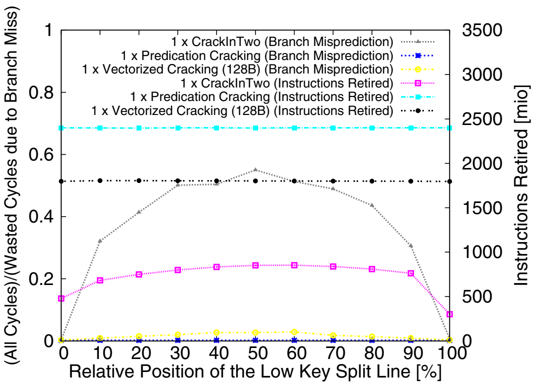


Fig. 5 CPU efficiency

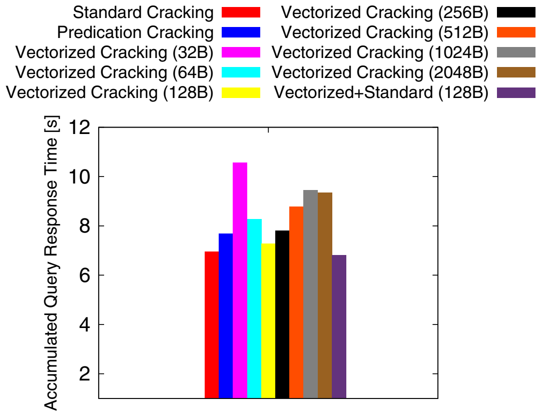


Fig. 6 Predication and vectorized cracking over query sequence

significantly pay off over applying crack-in-two twice. On first sight, these results look contrary to the ones presented in [23]. However, comparing the experimental set-ups, two differences become clear. Firstly, in [23], the authors work on pure 4 B keys in contrast to our 16 B (key, rowID) pairs that are in our opinion more realistic to represent an index column. As predication/vectorized cracking is write intensive, a larger element size puts more pressure on these methods than on the standard ones. Secondly, we perform one query consisting of two cracks here, instead of only a single crack in [23]. As our second crack is 1 % of the data size to the right of the first one, only few swaps must be performed and the branch prediction for standard cracking works already very well. To confirm the original results of [23], we rerun the experiment with 4 B keys and only a single crack in Fig. 4c. Now, we see a clear benefit of both predication and vectorized cracking over standard crack-in-two, if the split line falls between 20 and 80 % of the key range. However, when using our standard 16 B pairs (Fig. 4b), the single crack runtime increases heavily for predication and vectorized cracking, but only slightly for standard crack-in-two.

Let us finally look at how the predicated methods perform under a sequence of 1000 queries. Figure 6 shows the results. In accordance with the results of Fig. 4a, neither predication nor vectorized cracking can beat standard cracking with respect to accumulated query response time. The best vector size is clearly 128 B on our machine, where other sizes perform significantly worse. However, our previous insights enable us to use the best of two worlds. By using vectorized cracking for the first crack and standard cracking for the nearby second crack of a query, we are able to improve slightly over the standard version. Overall, whether predication and vectorized cracking pay off highly depends on element size, split line position, and machine properties.

Hybrid cracking *Create unsorted initial runs which are physically reorganized and then adaptively merged for faster convergence.*

Hybrid cracking [16] aims at improving the poor convergence of standard cracking to a full index, as shown in Fig. 3b. Hybrid cracking combines ideas from adaptive merging [10] with database cracking in order to achieve fast convergence to a full index, while still keeping low initialization 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 reorganization. In

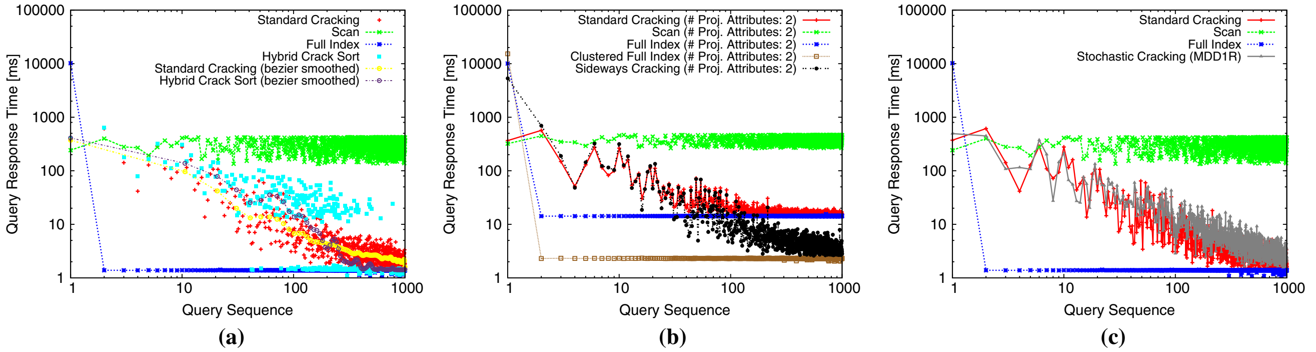


Fig. 7 Revisiting three advanced cracking algorithms. **a** Hybrid cracking, **b** sideways cracking, **c** stochastic cracking

addition to reorganizing the initial partitions, hybrid cracking also moves the qualifying tuples from each initial partition into a final partition. The authors explore different strategies for reorganizing 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 [16], as close to the original description as possible. Figure 7a shows hybrid crack sort in comparison with 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 [15] uses *cracker maps* to address the issue of inefficient tuple reconstruction in standard cracking, as shown in Fig. 3c. 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 7b 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 [11] addresses the issue of performance unpredictability in database cracking, as seen in Fig. 3d. 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 [11]. 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 materialized 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. Figure 7c 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 Sect. 4.5.

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 categorize the cracking algorithms along three dimensions: (1) the number of split lines they introduce, (2) the split strategy, and (3) the timing of the split. Table 1 shows the classification of different cracking algorithms along these three dimensions. Let us discuss these below.

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.

Table 1 Classification of cracking algorithms

DIMENSIONS	CATEGORY	NO INDEX	STANDARD CRACKING / PREDICATION 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						▨

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.

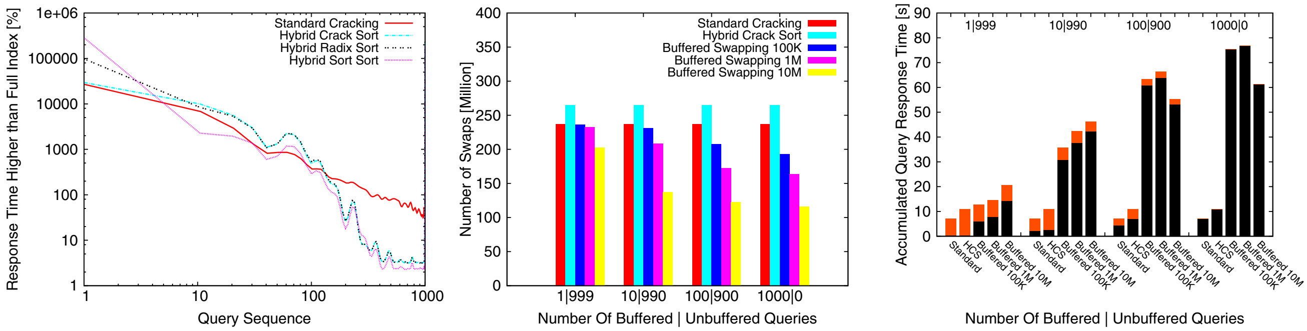
1. *Upfront:* A cracking algorithm could perform the splits before answering any queries. Full indexing falls in this category.
2. *Per Query:* All cracking algorithms we discussed so far perform splits when seeing a query.
3. *Never:* The trivial case is to never perform the splits, i.e. fully scanning 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 [16] addresses the convergence issue and whether we can improve upon it. First, let



(a) **(b)** **(c)**

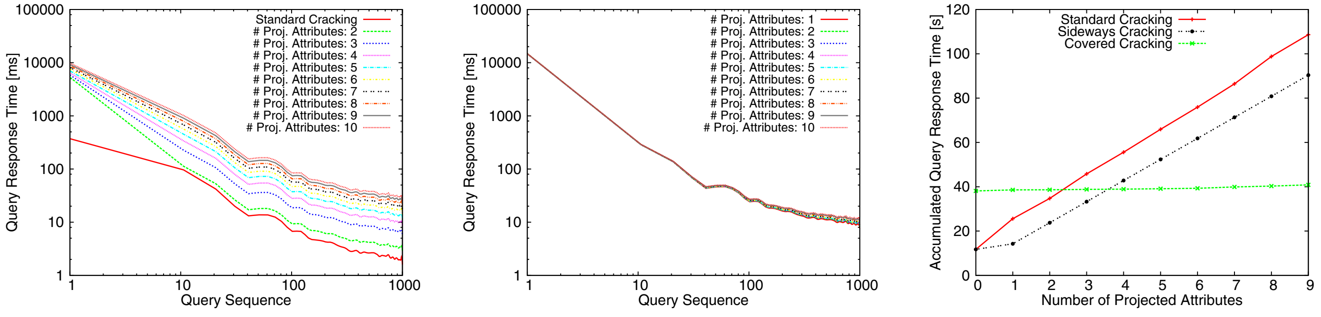
Fig. 8 Comparing convergence of standard cracking, hybrid cracking and buffered swapping. **a** Convergence speed towards full index, **b** influence on swap count, **c** influence on query response time

us compare hybrid crack sort from Fig. 7a with two other variants of hybrid cracking: *hybrid radix sort* and *hybrid sort* . Figure 8a 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 Fig. 7a. 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? 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[3] 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 penalizing 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 [21], 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 8b 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 100 K to 10 M entries. From Fig. 8b, we can see that the number of swaps decrease significantly as n_b varies from 1 to 1000. Compared to standard cracking, buffered swap-

[3] 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) **(b)** **(c)**

Fig. 9 Comparing tuple reconstruction cost of standard, sideways, and covered cracking. **a** Varying number of projected attributes for sideways cracking, **b** varying number of projected attributes for covered cracking, **c** covering trade-off for tuple reconstruction

ping saves about 4.5 million swaps for 1 buffered query and 73 million swaps for 1000 buffered queries and a heap size of 1 M. 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 with standard cracking and hybrid crack sort. Figure 8c 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 1 M 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 1000 buffered queries, the buffered part is faster for a maximum heap size of 10 M 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 [7] 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 [15] addresses the issue of tuple reconstruction and whether we can improve upon it. Let us first see how the sideways cracking from Fig. 7b scales with the number of attributes. Figure 9a 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 Fig. 3c), 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. To be on a safer side, we may cover all table attributes. However, this means that we will need to copy the entire table for indexing. We can think of fixing this by adaptively covering the cracked column, i.e. not copying the covered attributes upfront but rather on-demand when they are accessed. An option is to copy the covered attribute columns when they are accessed for the first time. An even more fine granular approach is to copy only the accessed values of covered attributes and thus reflecting the query access pattern in the covering status. Figure 9b 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 9c 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 s. 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 [11] addresses the issue of query robustness and whether we can improve upon it. In Fig. 7c, 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[4] 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 initialization time in order to achieve a 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 initialization 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 initialization time and least robust query performance. Thus, coarse-granular index covers the entire robustness spectrum between standard cracking and full indexing.

Figure 10a shows the query response time region (convex hull) of different indexing methods, including stochastic cracking (MDD1R), coarse-granular index, and full index (quick sort + binary search). 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 Fig. 3d, 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 10b shows the results. We can see that the initialization 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 initialization 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 initialization time and the query time. For instance, for 1000 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 1000 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.

[4] 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.

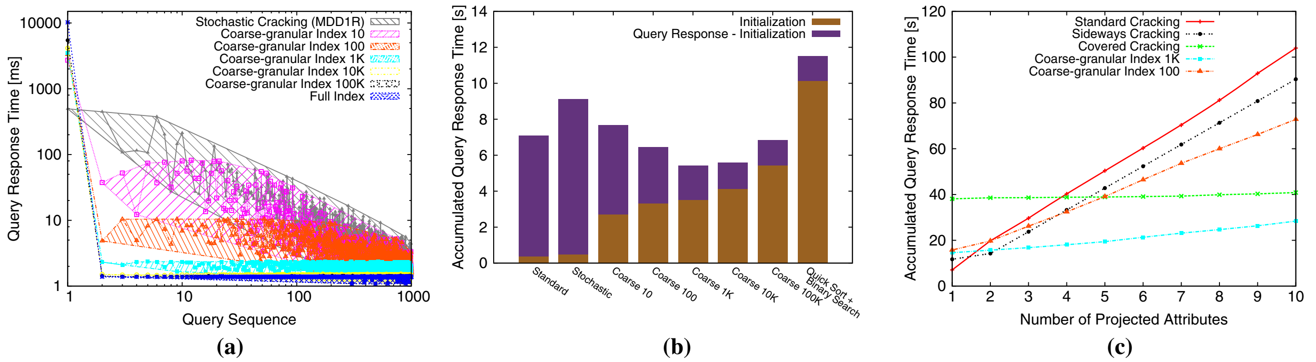


Fig. 10 Comparing robustness of standard cracking, stochastic cracking, coarse-granular index, and full index. **a** Variance in response time, **b** initialization time trade-off, **c** extending Fig. 9c by nearby clustering

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 initialization 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 Sect. 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 initialization time. Note that if the data is stored in row layout, then the source table is anyways scanned completely during index initialization 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 10c shows the effect of pre-partitioning the source table. We create both 100 and 1000 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 Fig. 10c 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 Fig. 10c that coarse-granular index has a much more stable performance when scaling the number of projected attributes, without reorganizing the base table at query time. In fact, coarse-granular index 1 K even outperforms covered cracking for any number of projected attributes. For example, when projecting all 10 attributes, coarse-granular index 1 K 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 1000 table partitions, each partition has a size of 8 MB and thus fits completely in the CPU cache. For 100 partitions, the partition size increases to 80 MB, 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 Fig. 10 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.

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 Sect. 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.

Table 2 Scalability under data size increase by factor 10

Factor slower (from 100 M to 1 B)	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 1 K	11.69	9.96	11.09
Coarse-granular index 10 K	11.31	9.94	10.95
Coarse-granular index 100 K	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 1 K (nearby clustered)	–	–	13.33

Table 3 Classification of extended cracking algorithms

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				■	■

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 with 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 Fig. 2b). 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 [12]? We compared quick sort with an in-place radix sort implementation [4]. 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 11a 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 s are included in the first query. However, the initialization time for radix-insert sort drops by half to around 5 s. 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: (1) AVL-tree [1], (2) B+-tree [3], and (3) the very recently proposed ART [20]. We choose ART since it outperforms other main memory optimized search trees such as CSB+-tree [24] and FAST [19].

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 1000 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 11b 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.[5] 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 11c shows the per-query

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

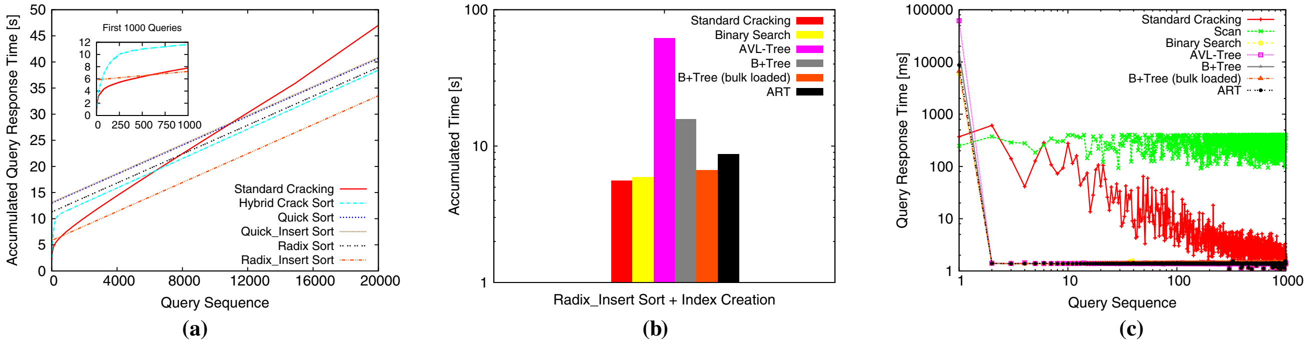


Fig. 11 Comparing standard cracking with different sort and index baselines. **a** Comparing different sort algorithms, **b** indexing effort of diff. indexes, **c** per-query response time of diff. indexes

response times of different indexing methods. Surprisingly, we see 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 12a 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: (1) the indexing costs to sort the data and to build the structure, and (2) the index lookup and data access costs to retrieve the result.

Figure 12b 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 1000 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 12c 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

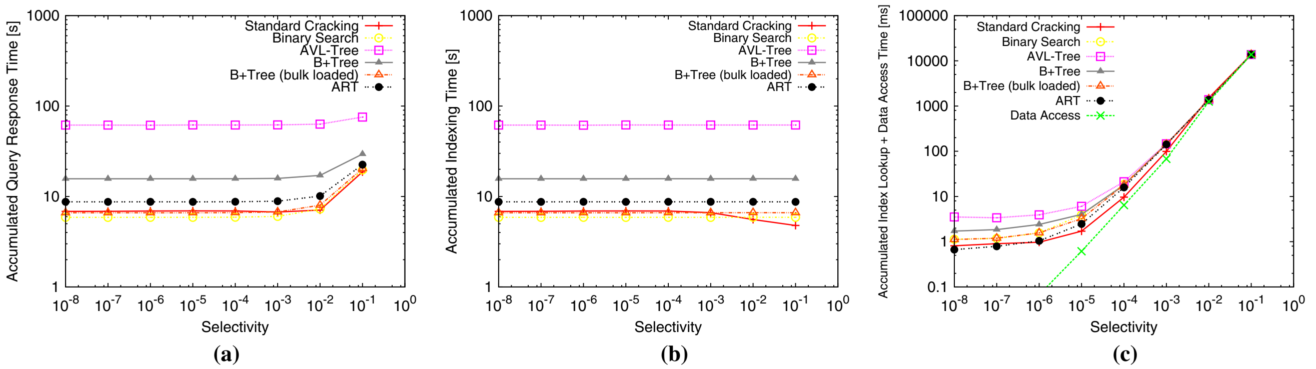


Fig. 12 Comparing standard cracking with index baselines while varying selectivity (note that: (**a**) = (**b**) + (**c**)). **a** Accumulated query response time, **b** accumulated indexing time, **c** acc. index lookup + data access time

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 Fig. 12a.

To conclude, the take-away message from this section is threefold: (1) using a better index structure makes sense only for very high selectivities, e.g. one in a million, (2) cracking depends on query selectivity in terms of indexing effort, and (3) although cracking creates the indexes adaptively, it still needs to catch up with full indexing in terms of the *quality* of the index.

4.4 Effect of varying cracking depth

The cracking algorithms tested so far reorganize the cracker column till the fully sorted state is reached. However, the authors of [17] suggested, in their original work, that it might make sense to stop further reorganization at a certain partition size and filter the partitions instead. Thus, in the following experiment, we vary the threshold at which we stop applying standard cracking and inspect the effect on the runtime. Figure 13 shows the results. We present the accumulated query response time over our query sequence of 1000 queries and split the bars into indexing time, representing the time to reorganize the partition(s), and index lookup with data access time, representing the query result computation. This result computation corresponds to a simple scan if the column has been cracked by this query or a scan with a filtering, if no cracking has been performed previously. The threshold at which we stop cracking is varied from 16,000 (250 KB) to 256,000 (4000 KB) entries. In Fig. 13, we can observe that a threshold larger than 64,000 entries has a clear impact on the accumulated query response time. The larger the threshold, the smaller is the indexing time as less cracking effort is needed. As a consequence of the reduced indexing effort, the querying time increases due to the additional filtering. Unfortunately, the savings in indexing effort are eclipsed by the larger increase in querying time. As a result, the overall runtime increases. Therefore, stopping cracking at a certain partition size and applying filtering does not improve performance.

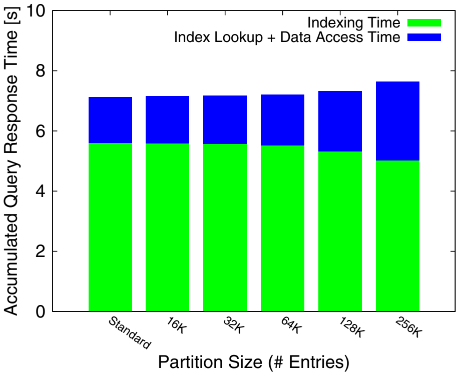


Fig. 13 Stopping cracking at a certain partition size

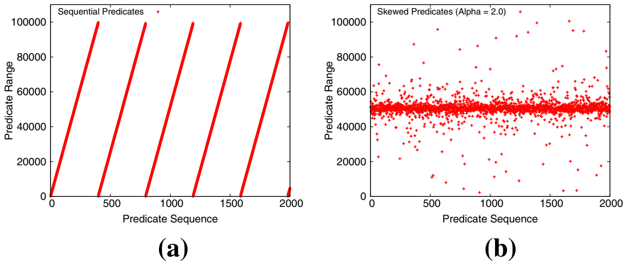


Fig. 14 Generated predicates for different access pattern. **a** Sequential pattern, **b** skewed pattern

4.5 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 14a visualizes the generated predicates. In Fig. 15a, we show the query response time under the sequential access pattern for standard cracking, stochastic cracking, coarse-granular index with 1000 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 there-

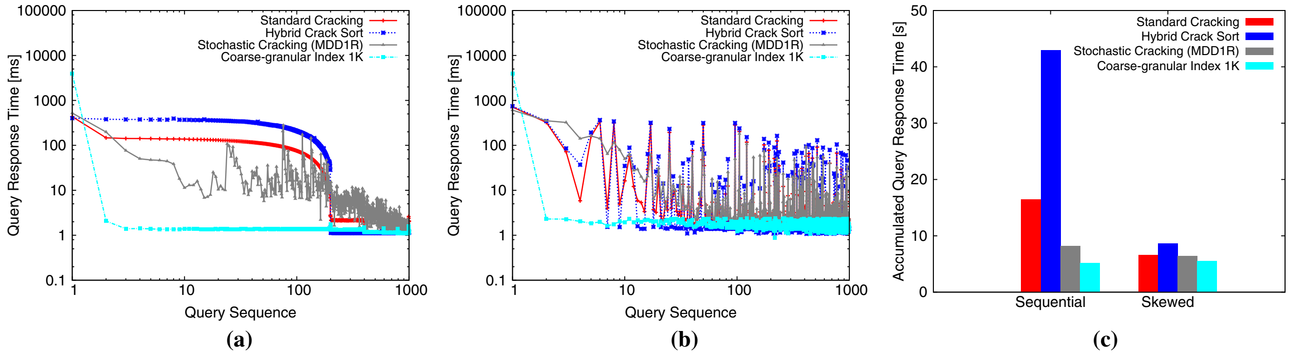


Fig. 15 Effect of query access pattern on adaptive methods. **a** Sequential access pattern, **b** skewed access pattern, **c** accumulated query response times

fore 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 14b shows the generated predicates for $\alpha = 2.0$. These predicates are randomly shuffled before they are used in the query sequence. Figure 15b shows the query response time for the skewed pattern. We can observe a high variance in all methods except coarse-granular index. 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, Fig. 15c 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 fulfils 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 Parallelising cracking algorithms

Of course, the concept of So far, we looked entirely at single-threaded implementations of cracking algorithms, sorting methods, and index structures (Table 4). However, nowadays, with commodity hardware offering multiple hardware threads and server machines easily consisting of several multi-core processors, parallelizing an algorithm is crucial for efficiency. Thus, in the following section, we investigate the current state-of-the-art parallel cracking algorithms and identify their strengths and weaknesses.

5.1 Parallel cracking methods

In general, parallel query processing can be realized in two ways: (1) *inter-query parallelism*, which interleaves the execution of multiple queries while isolating them semantically, and (2) *intra-query parallelism*, which serializes the answering of the query sequence, while each individual query is

Table 4 All single-threaded algorithms evaluated in this paper

Algorithm	References
Standard cracking	[17]
Predication/vectorized cracking	[23]
Hybrid crack/radix/sort sort	[16]
Buffered swapping	This paper resp. [25]
Stochastic cracking (MDD1R)	[11]
Coarse-granular index	This paper resp. [25]
Sideways cracking, Covered cracking	[15], this paper resp. [25]
Sorting: Quick(_insert) sort, Radix(_insert)	[12,13]
Full index: AVL-tree, B+-tree, ART	[1,3,20]

Table 5 All multi-threaded algorithms evaluated in this paper

Algorithm	Abbreviation	References
Parallel standard cracking	P-SC	[8,9]
Parallel coarse-granular index	P-CGI	[2]
Parallel-chunked standard cracking	P-CSC	[2]
Parallel-chunked vectorized cracking	P-CVC	Variant of [23]
Parallel-chunked coarse-granular index	P-CCGI	[2]
Parallel range-partitioned radix sort	P-RPRS	[2]
Parallel-chunked radix sort	P-CRS	[2]
Parallel sideways cracking (with P-CSC)	P-SW-CSC	This paper
Parallel sideways cracking (with P-CCGI)	P-SW-CCGI	This paper
Parallel range-partitioned radix sort (cluster complete)	P-PC-RPRS	This paper
Parallel range-partitioned radix sort (cluster lazy)	P-LC-RPRS	This paper

evaluated in parallel. In the following, let us look at the main representatives of these two classes of parallelism. Table 5 gives an overview alongside with their initial sources and used abbreviations. To the best of our knowledge, these algorithms form the complete set of parallel cracking algorithms known to date. Obviously, many of the methods originate from our own study on parallel adaptive indexing techniques [2] that was the follow-up work of the paper that this article extends. Thus, we use this chance to combine both works and re-evaluate the methods under a new set-up as well as in comparison with new methods. Let us now look at the different parallel algorithms in detail.

Parallel standard cracking (P-SC) *Interleave answering of multiple queries in isolation by serializing crack-in-two on the granularity of partitions.*

A very natural form of inter-query parallelism is realized in parallel standard cracking [8,9], denoted as *P-SC* from here on. It is based on the observation that a query modifies at most two partitions of the cracker column. Thus, if we want to execute multiple queries at the same time on the same cracker column, all we have to do is serializing the cracking of partitions. To do so, the authors of [8,9] introduce read and write locks on the partition level. An incoming query, running in its own thread, tries to acquire (at most two) write locks for the partitions at the border, that it has to crack, and read locks for the inner partitions. Since acquiring write locks is exclusive, only one query at a time can modify a certain partition. Similar to the single-threaded case, we can apply the concept of coarse-granular index to P-SC as well. To do so, the first query applies the lock-free parallel range-partitioning algorithm we used in our study on parallel cracking algorithms [2] before starting the actual query answering. We will refer to this method as *P-CGI*.

Parallel coarse-granular index (P-CGI) *Apply a parallel range partitioning to bulk-load the cracker index before starting the query answering using P-SC.*

Obviously, for P-SC and P-CGI, the degree of parallelism highly depends on the current cracking state of the cracker column and on the query access pattern. Thus, the following algorithm, that we introduced in [2], implements the parallelism inside the answering of a single query to create a more stable parallel execution over the query sequence.

Parallel-chunked standard cracking (P-CSC) *Divide the cracker column non-semantically into independent chunks and apply standard cracking on each chunk in parallel.*

The concept of parallel-chunked standard cracking [2], denoted *P-CSC* from here on, is as simple as effective. We divide the column logically into multiple chunks and treat each chunk as a separate cracker column with its own cracker index. The incoming queries are executed sequentially within the query burst, while each individual query is evaluated on all chunks in parallel. Thus, each chunk is cracked individually and produces a local query result. When all threads finish the evaluation of the query locally, the global result can be computed. As there is almost no communication or synchronization necessary during cracking and result computation, this method naturally offers a high degree of parallelism from the very first query on. The same concept has been used in [23] in combination with vectorized cracking. Thus, we also test a vectorized version, denoted as *P-CVC*[6] from here on. Of course, the concept of work division can be applied to more advanced cracking algorithms as well. Since our coarse-granular indexing method offers an interesting alternative to the standard version, we also test our chunked implementation of parallel coarse-granular index that we first introduced in [2].

Parallel-chunked coarse-granular index (P-CCGI) *Divide the cracker column non-semantically into multiple independent chunks and apply coarse-granular index on each chunk in parallel. Then, apply standard cracking locally for the query answering.*

The concept remains the same. The only difference to P-CSC is an initial step of range partitioning within each chunk as performed by the single-threaded coarse-granular index. After that, the single-threaded standard cracking is used in each chunk for the local result computation. We will call this method from here on *P-CCGI* [2].

[6] In contrast to [23], we do not merge the chunks after each query as this results in overhead.

In comparison with the different multi-threaded cracking versions, we test our two parallel radix-based sorting methods from [2]. The first version, called *P-RPRS*, applies first a parallel range partitioning and then sorts the partitions locally in parallel using a single- threaded radix sort. The second version, denoted *P-CRS* from here on, chunks the input non-semantically and then applies single-threaded range partitioning and radix sort on each chunk in parallel.

5.2 Hardware set-up

For the multithreading experiments, we use a high-end server of 4 sockets, each equipped with an Intel Xeon E7-4870 v2 processor with 15 physical and 30 logical cores, running at 2.3 GHz. Therefore, the machine has 60 physical and 120 logical cores available. The overclocking capabilities of the processors (Intel Turbo Mode) are disabled for all experiments, as they unnecessarily complicate the analysis. The private L1 and L2 caches of each core have a size of 32 and 256 KB, respectively, while the shared L3 cache of each processor has a size of 30 MB. Each processor has 3 QPI links such that remote memory access is equally expensive for all neighbouring processors. Each socket is attached to 128 GB of 1600 MHz DDR3 RAM running in Intel Performance Mode, resulting in 512 GB of available main memory. The operating system is a 64-bit Debian with kernel version 3.2. As the memory bandwidth plays an important role in the following discussion, we measured the throughput of the machine using the STREAM benchmark [22] that runs a set of simple read/write vector kernels. Instead of relying on the computed bandwidth of the benchmark, we measured the throughput directly at the memory controller using Intel VTune Amplifier 2015. Figure 16 shows the aggregated bandwidth for all 4 sockets. As we can see, we reach 65 GB/s per socket and thus achieve a total machine bandwidth of 260 GB/s.

5.3 Experimental set-up

Before we can start with any experimental evaluation, let us define the way in which the queries are fired and executed. As in previous experiments, we have a set of 1000 queries that should be answered as fast as possible. All queries are directly ready to be processed, and there is no artificial idle time introduced between queries. Depending on the type of the algorithms, this query batch is processed differently. For algorithms that perform inter-query-parallelism, like P-SC for instance, we divide the set of queries into k parts, which are processed using k threads with each thread working $1000/k$ queries sequentially. This resembles the way of firing queries in [8]. In contrast, for algorithms that perform intra-query parallelism, like P-CSC, the 1000 queries will be answered sequentially one after another. However, each individual query is answered by k threads in parallel on a portion of the data. Please note that to get a more realistic set-up, we introduced a barrier in the query execution loop: the answering of the next query starts only after all threads completed the current one.

5.4 Scaling of parallel cracking algorithms

In Sect. 5.1, we described the set of algorithms for parallelizing database cracking. As mentioned before, many of these algorithms originate from our earlier study on parallel adaptive indexing [2]. In that work, we studied both the absolute runtimes and the scalability of the parallel cracking algorithms. In this paper, we revisit the scalability of parallel cracking algorithms in depth. To do this, we extend the parallel cracking experiments in two ways. First, in contrast to our previous study that used a low-end server with only 8 cores, we now use a massively parallel high-end machine consisting of 4 sockets and 60 physical, respectively, 120 logical cores (see Sect. 5.2 for a detailed description of the hardware). We believe that it is valuable to re-evaluate these techniques under a vastly different set-up to get possibly new insights from them. Second, we dig into and analyse the performance of parallel cracking by looking at contention and bandwidth using Intel VTune Amplifier. Our goal is to understand and explain the scalability of parallel cracking algorithms in a massively parallel environment.

Besides the raw query processing times of different algorithms, parallel methods offer another important dimension to analyse: the capabilities to scale with the multi-threading resources of the hardware. An algorithm, which scales poorly, might be the winner in terms of runtime on a small machine, but completely looses the pace on a large server. Therefore, in the following, we will inspect for each method individually how it scales with an increase of the number of threads. We run each method using 4, 8, and 15 threads to utilize the computing cores up to $\frac{1}{4}$-th of the machine. Additionally, we test 30, 45, and 60 threads to investigate the scaling over the sockets. Finally, we run 120 threads to utilize all logical cores

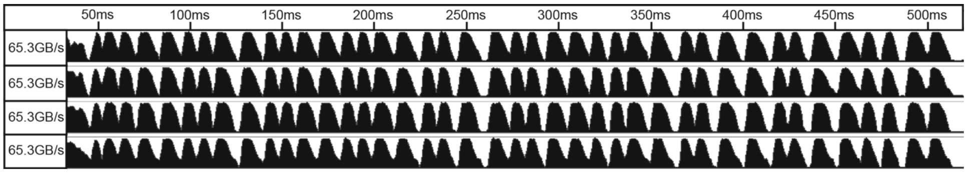


Fig. 16 Stream benchmark with 60 threads. We can reach 65 GB/s per socket for the aggregated read/write bandwidth per socket

of the machine as well. We do not apply any form of thread pinning and let the operating system decide.

Figure 18 presents the accumulated speedups of the algorithms relative to their single-threaded counterparts. We inspect the individual parts (copying, range partitioning, sorting, query answering) of the methods to analyse them separately as well as the total speedup. Let us go through the methods one by one and analyse their scalability.

Parallel standard cracking (P-SC) Figure 18a presents the scaling capabilities of the well- known, lock-based P-SC. Unfortunately, it scales poorly with the increasing number of threads. The highest total speedup we observe is around 3.7× for 120 threads. The situation is particularly bad in the early phase of the query answering as the measured speedups are only between 1.5× and 2×. To understand this scaling problem, let us visualize the processing behaviour of the algorithm. To do so, each time a thread is processing a partition, we log the time it takes as well as the processed area in the cracker column. This time includes possible waiting to acquire locks as well as the actual data processing. Figure 17 shows the plotted result. A rectangle $[x_1, y_2, x_2, y_2]$ means that within the time from x_1 to x_2, a thread was processing the cracker column at the range y_1 to y_2. The colours indicate the processing type, where red is modifying access (cracking) and green reading access (querying). Figure 17a presents the results for P-SC for 8 threads. We can observe a severe access contention in the first half of the run. The early queries lock huge parts of the cracker column as there exist only large partitions and thus serialize each other heavily. Therefore, the algorithm has no chance to scale linearly when starting from an unpartitioned state. Thus, let us see how the problem decreases when prepending a range-partitioning step in the next algorithm.

Parallel coarse-granular index (P-CGI) As described before, this algorithm extends P-SC by applying an initial parallelized range-partitioning step that creates 1024 partitions right away before any query answering starts. This should have a positive effect on P-SC and significantly reduce the contention that we have measured before. Figure 17b presents again the partition processing contention, this time for P-CGI. The blank space between time 0 and 0.7 s is the range-partitioning phase. Afterwards, we see from 0.7 s till 1.2 s the actual query answering, which indeed parallelizes nicely now. No heavy contention is visible anymore and the algorithm behaves as intended, as the partitions are already small and the chance of two threads accessing the same partition is small. This is confirmed by the scaling factors of the P-CGI query answering phase in Fig. 18b, which now reaches a factor of 11× for 45 threads. For more threads, the performance significantly drops again, as access contention (both on the column as well as on the protected cracker index) throttles the throughput again. Figure 19a presents a query-wise view on the answering phase. We can see that directly in the first query, the scaling is still very limited. This is caused by the set-up and assignment of the threads to the tasks, which is expensive in comparison with the short running queries. Additionally, NUMA remote accesses are throttling the query answering phase. As the parallel range-partitioning algorithms creates partitions that are scattered across regions, a thread that answers a query has consequently a large number of remote accesses. Table 6 shows that based on hardware counters 2 out of 3 accesses are remote for P-CGI. Let us now look at the range partitioning itself. For 120 threads, we achieve the best speedup of factor 15×. Memory bandwidth is clearly not the problem, as it can be seen in the early phase of Fig. 19b, where only the histogram generation maximizes the bandwidth utilization. Our VTune analysis indicates that the range-partitioning algo-

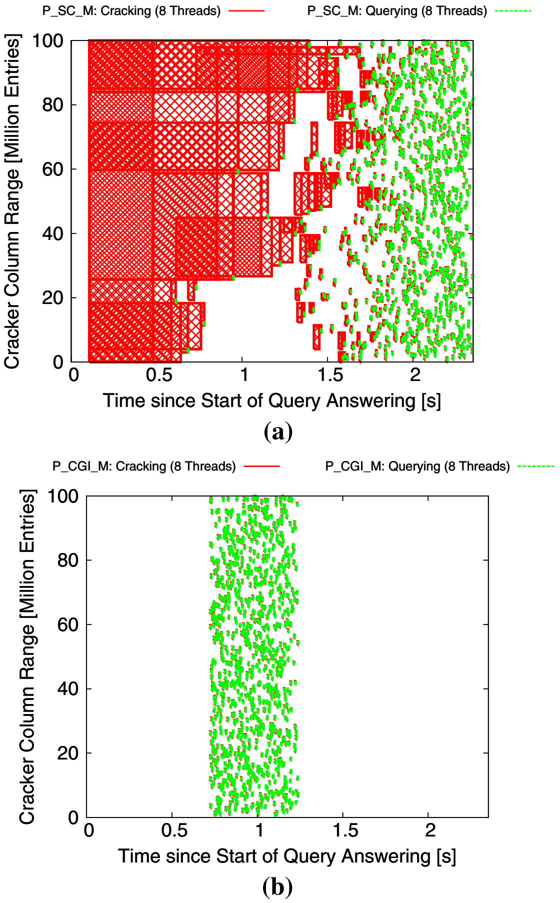


Fig. 17 Visualization of the partition processing contention for 8 threads. A *rectangle* $[x_1, y_1, x_2, y_2]$ means that within the time from x_1 to x_2, a thread was processing the cracker column at the range y_1 to y_2. Processing also includes wait times to acquire a lock. A *red square* indicates a writing process (cracking a partition), while a *green square* visualizes a reading process (querying a partition). *Overlapping squares* indicate that multiple threads intent to work on the same area of the cracker column at the same time. **a** Parallel standard cracking (P-SC), **b** parallel coarse-granular index (P-CGI) (colour figure online)

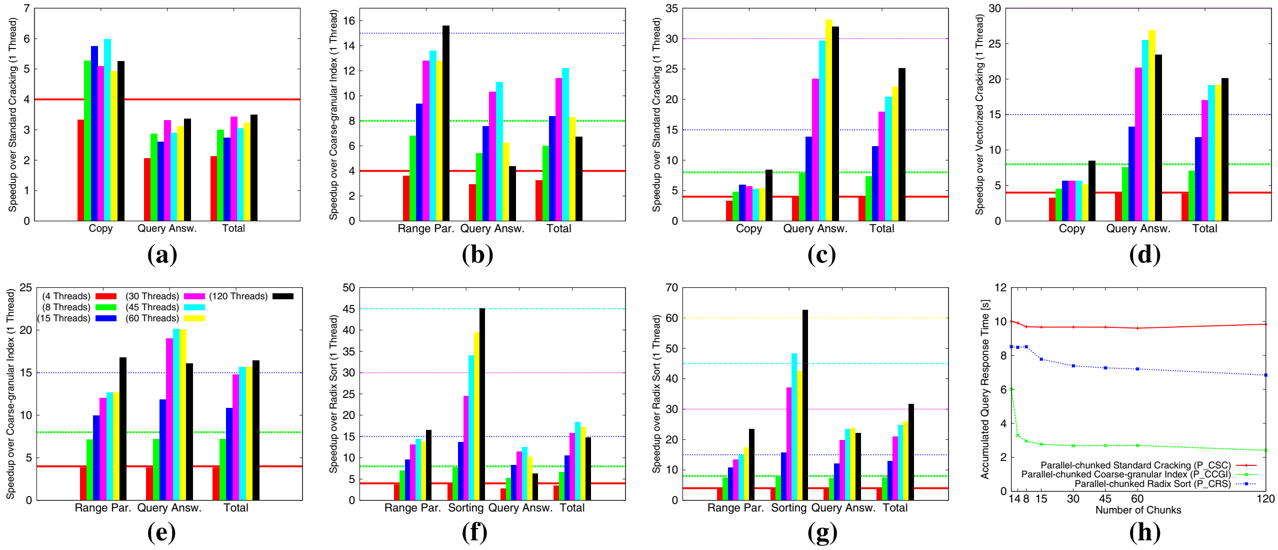


Fig. 18 Speedup of parallel cracking and sorting algorithms over their single-threaded counterparts while varying the number of threads. We show both the speedups of the characteristic phases as well as the overall achieved speedups. *Coloured horizontal lines* show the expected perfect linear speedup. In **h**, we show for the chunked algorithms how the chunking itself influences the methods by serially working the chunks. **a** Parallel standard cracking (P-SC), **b** parallel coarse-granular index (P-CGI), **c** parallel-chunked standard cracking (P-CSC), **d** parallel-chunked vectorized cracking (P-CVC), **e** parallel-chunked coarse-granular index (P-CCGI), **f** parallel range-partitioned radix sort (P-RPRS), **g** parallel-chunked radix sort (P-CRS), **h** serial execution of chunked algorithms (colour figure online)

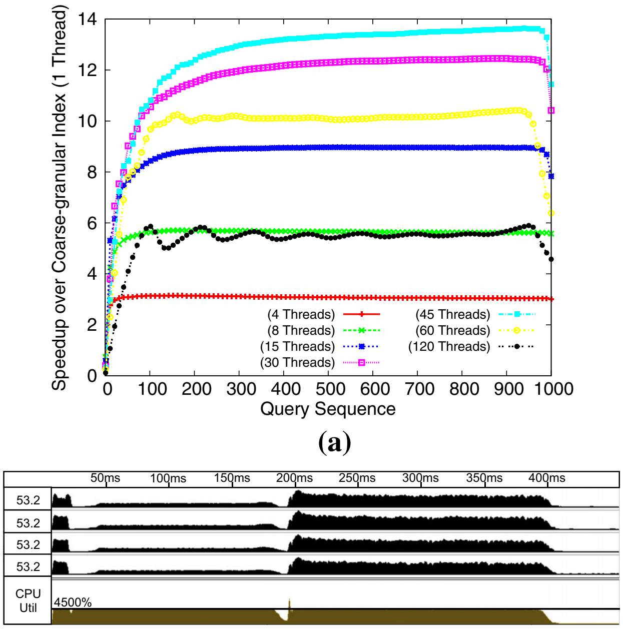
rithm is heavily back-end bound by the random nature of the partitioning. Advanced partitioning techniques like software-managed buffers and non-temporal streaming stores might improve upon this problem, as we investigate in a separate study on partitioning [26].

Parallel-chunked standard cracking (P-CSC) After looking at the inter-parallel version of standard cracking, let us now inspect the scaling behaviour of the intra-parallel version named P-CSC. The results are shown in Fig. 18c. We can see that this algorithms scales considerably better than the previous ones, which is what we expect from a method that parallelizes by chunking. However, we can also observe that the scaling is not linear with the number of threads. The highest total speedup that we achieve for 120 threads is only around 25×. To understand this behaviour, let us inspect the individual parts. Interestingly, the copying phase, which simply duplicates the input column into a separate array, scales particularly bad with a maximum speedup of 8×. As we can see from the bandwidth plot of Fig. 20a for 60 threads, the memory bus is not the limiting factor, which is poorly utilized within the first 150 ms. We identified page faults, which are surprisingly expensive to resolve when touching the cracker column for the first time during the copying phase as the cause of this behaviour. Let us now see how the query answering part alone scales in P-CSC. In Fig. 18c, we see a maximal speedup of the query answering phase of 33× for 60 threads, which is still not linear. NUMA effects are not a problem here as we can see in Table 6, all accesses are local. Apparently, the scaling is limited from 45 threads on, so let us inspect the utilized bandwidth of the query answering phase for 30 threads (Fig. 20b), 45 threads (Fig. 20c), and 60 threads (Fig. 20d). We can see that in the early phase the bandwidth for 30 threads is with almost 59 GB/s already close to the cap of 65 GB/s, so we cannot expect a linear scaling when increasing the number of threads by a factor of 1.5× (45 threads), respectively, 2× (60 threads).

From Fig. 18d, we can see that *Parallel-chunked vectorized cracking (P-CVC)* shows a very similar scaling behaviour as P-CSC. It scales slightly worse than P-CSC due to its nature of being even more bandwidth bound.

Parallel-chunked standard cracking (P-CCGI) Let us now inspect the intra-parallel version of coarse-granular index in Fig. 18e. Interestingly, the range-partitioning phase scales almost exactly the same as the one of P-CGI in Fig. 18b, although the former uses a parallel range partitioning while the latter one chunks a single-threaded implementation. This shows again that the partitioning is heavily back-end bound and that stalls throttle the algorithm. The query answering phase scales with 20× for 60 threads much better than that of P-CGI. One reason is that each chunk can be processed individually without any concurrency control except the barrier at the end of each query. Another reason is the almost perfect NUMA locality that we can observe in Table 6.

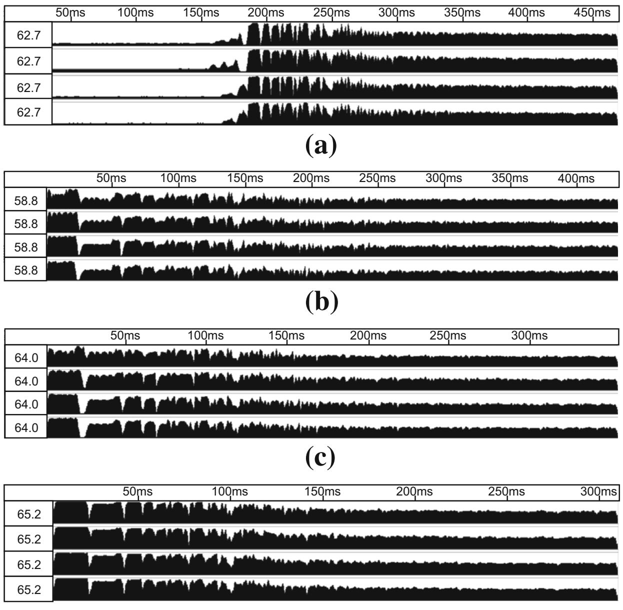
Parallel-chunked standard cracking (P-RPRS) Finally, we want to analyse the scaling capabilities of the sorting baselines. Let us start with P-RPRS presented in Fig. 18f. The initial range-partitioning phase resembles the one of P-CGI



(a)

(b)

Fig. 19 Bandwidth of P-CGI measured at 4 sockets in GB/s. The *bottom line* shows the CPU utilization in percentage. **a** Scaling of parallel-chunked standard cracking (P-CGI) of the query answering phase without the range-partitioning phase, **b** parallel-chunked standard cracking (P-CGI) with 45 threads. Highest bandwidth observed: 53.25 GB/s



(a)

(b)

(c)

(d)

Fig. 20 Bandwidth measured at 4 sockets with Intel VTune Amplifier 2015 in GB/s. **a** Parallel-chunked standard cracking (P-CSC) with 60 threads. The initialization phase (copying the data into the cracker column in parallel) utilizes the bandwidth only partially (around 8 GB/s), **b** parallel-chunked standard cracking (P-CSC) with 30 threads without initialization phase. Highest bandwidth observed: 58.77 GB/s. **c** parallel-chunked standard cracking (P-CSC) with 45 threads without initialization phase. Highest bandwidth observed: 64.02 GB/s, **d** parallel-chunked standard cracking (P-CSC) with 60 threads without initialization phase. Highest bandwidth observed: 65.24 GB/s

Table 6 Number of LLC cache misses that are served with local, respectively, remote DRAM access presented in millions of measured events

Method	Local accesses (Mio)	Remote accesses (Mio)
P-SC	107	418
P-CGI	99	202
P-CSC	442	0
P-CVC	357	0.2
P-CCGI	44	0.2
P-RPRS	115	230
P-CRS	365	0.6

The measured counters are `OFFCORE_RESPONSE.DEMAND_DATA_RD.LLC_MISS.LOCAL_DRAM` and `OFFCORE_RESPONSE.DEMAND_DATA_RD.LLC_MISS.REMOTE_DRAM`

which is why we see exactly the same scaling behaviour. Afterwards, each created partition is sorted individually in parallel. Obviously, this phase scales much better with 45× for 120 threads at best. The reason lies in the great cache locality created by the previous range partitioning. By dividing the dataset into 1024 pieces, each partition has a size of 1.49 MB. Since each processor has a L3 cache size of 30 MB and 15 physical cores, each core has basically 2 MB of cache available (1 MB per logical core). This is obviously enough to keep all currently worked partitions completely inside the caches in the case of 60 threads. The scaling of the query answering phase is at best only 13× for 45 threads. This is again due to the high number of remote accesses in Table 6 caused by the initial parallel range partitioning. They also have a negative impact on the sorting phase, although the worked partition is loaded once into the cache and then worked locally.

Parallel-chunked standard cracking (P-CRS) The second sorting algorithm, which does not create a global sorting, range partitions and sorts all chunks locally in parallel. In the scaling result of Fig. 18g, we can see that the sorting phase scales even better than in P-RPRS. One reason lies in the NUMA local accesses, as we can see in Table 6. Another reason is presented in Fig. 18h. As sorting multiple smaller chunks is by default cheaper than sorting a large one, a part of the speedup also originates from that. This also causes the super-linear sorting speedup for 30 and 45 threads. The query answering phase of P-CRS also scales better than the one of P-RPRS due to the NUMA local processing nature.

5.5 Runtime of parallel cracking algorithms

After investigating the scaling capabilities of the algorithms on an individual basis, let us see how they compete against

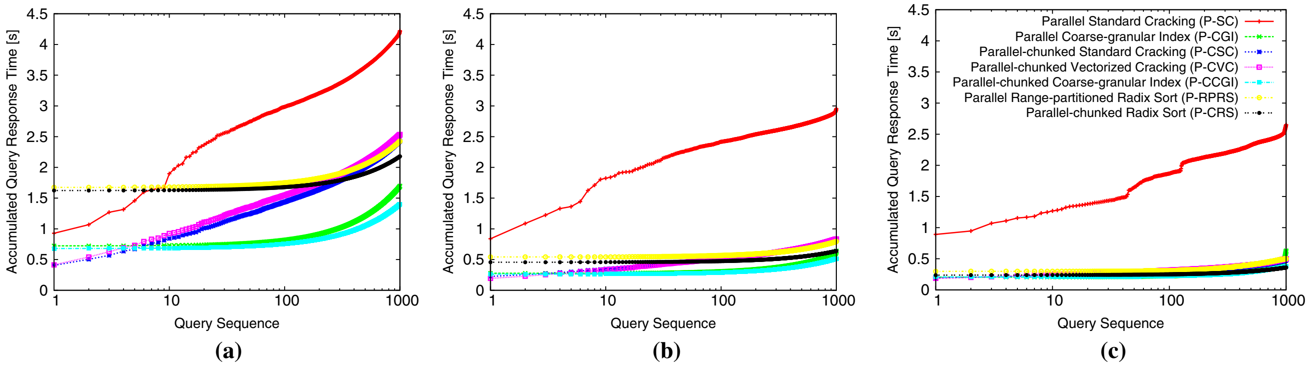


Fig. 21 Accumulated query response time of parallel cracking algorithms in comparison with parallel radix-based sorting methods. **a** 4 threads, **b** 15 threads, **c** 60 threads

each other. To do so, we measure and compare the accumulated query response time over 1000 queries and present the results using different thread configurations in Fig. 21. For 4 threads in Fig. 21a, there is a clear difference in runtime between the individual algorithms visible. Obviously, P-CSC has the lowest initialization time with almost 0.5 s, while the sorting methods need with around 1.7 s considerably more time for their first query. Over 1000 queries, the cracking methods P-CCGI and P-CGI clearly win in terms of accumulated runtime, while P-SC is far behind the remaining methods due to its serialization behaviour in the early querying phase. When increasing the number of threads to 15 in Fig. 21b and to 60 in Fig. 21c, we see a clear trend: the difference between the sorting and cracking methods significantly decreases. For 60 threads, the time of the first query for P-CSC is with 191 ms only 46 ms shorter than that of P-CRS, which fully sorts the chunks and answers the first query in 237 ms, caused by the superior scaling of the sort-based algorithms. This analysis indicates, that for a large number of threads, the sorting algorithms are a clear alternative over the adaptive methods, especially since they are easier to integrate into the system stack and offer interesting orders. Nevertheless, we believe that in a real system with many queries processing several columns at the same time, only a portion of the physical resources are available to initialize a column. Under such circumstances, cracking remains its advantage of offering the significantly cheapest option of enabling indexing.

5.6 Tuple reconstruction in the context of parallelism

So far, we have looked at the parallel indexing methods without considering tuple reconstruction in order to focus solely on the cracking and sorting algorithms. Now, let us see how the tuple reconstruction concepts we have seen already in the single-threaded case, like sideways cracking [15], can be applied on top of multi-threaded algorithms. Precisely, we will investigate how sideways cracking can be combined with parallel cracking algorithms. To the best of our knowledge, this is the first work to approach this question. Then, we will compare the tuple reconstruction performance of the parallel cracking algorithms with a clustered table that has been ordered with respect to the sorted index column using our parallel range-partitioned radix sort. As the basis for parallel sideways cracking, we pick the two cracking algorithms that performed the best in the previous evaluation—P-CSC and P-CCGI. This allows us to apply the concept of chunking to sideways cracking as well. For each chunk, we keep separate cracker maps and a separate tape, and thus, the chunks can be worked independently by the individual threads. We name these two methods P-SW-CSC, respectively, P-SW-CCGI in the following. The baseline for parallel sideways cracking is formed by a clustered table that can be created in two ways. The first version, coined P-PC-RPRS, clusters the entire table (stored in column layout) directly in the first query with respect to the selection column. The sorting is performed using our parallel range-partitioned radix sort. The second version, called P-LC-RPRS, establishes the clustering in a lazy manner, by copying and clustering only the columns that are actually touched by a query. In this case, the clustering of a column is created by applying a fresh sort on the selection column. To put these methods to the test, we apply different workloads. All share the property that the selection is performed on a single, fixed attribute of a table composed of 10 columns following a uniform random distribution. We perform separate runs projecting 1 and 5 attributes, respectively, that are randomly selected for each query. Figure 22 shows the accumulated query response times for 4 and 60 threads. Let us focus on the 4 threaded case first in Fig. 22a, b. For all numbers of projected attributes, P-PC-RPRS behaves in the most predictable way. Clustering the entire table of 10 columns takes around 11 seconds and the following query answering takes only a small amount of additional time, even if 5 attributes are projected. P-LC-RPRS, which clusters a column when it is touched for the first time is heavily affected by the number of projected attributes. Interestingly, the larger

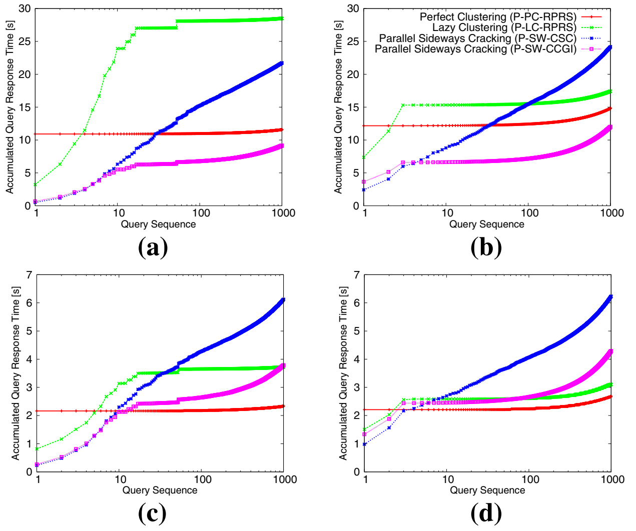


Fig. 22 Accumulated tuple reconstruction cost for 1000 queries and a table consisting of 10 columns, shown for 4 and 60 threads. We select on a single fixed attribute. In **a** and **c**, each query projects a single randomly chosen attribute. In **b** and **d**, each query projects five randomly selected attributes. **a** 4 threads: one σ, one π, **b** 4 Threads: one σ, five π, **c** 60 Threads: one σ, one π, **d** 60 threads: one σ, five π

the number of projected attributes, the smaller is the accumulated query response time. This makes sense as a query projecting multiple attributes can cluster multiple columns in a single sorting run. We can also observe that the lazy clustering pays off only for the first few queries, at least for a table consisting of only 10 columns. In comparison with that, parallel sideways cracking offers in both implementations a significantly smaller initialization time. The first query of P-SW-CCGI is slightly more expensive than that of P-SW-CSC, as it range partitions the dataset during the initialization of a cracker map. In the long run, it always clearly pays off to prepend a range-partitioning step. Overall, for 4 threads and 1000 queries, P-SW-CCGI shows the best accumulated runtime in all tested cases. This picture changes if we switch to 60 threads in Fig. 22c, d. Obviously, all methods benefit from the increased number of threads; however, the sort-based methods win at a higher degree. Obviously, the better scaling capabilities of the sort-based methods that we saw in the previous analysis pay off in the tuple reconstruction case as well. P-PC-RPRS needs less than 10 queries to beat both parallel sideway cracking implementations. The difference between the lazy P-LC-RPRS and P-PC-RPRS has also significantly decreased, and even P-LC-RPRS outperforms P-SW-CSC around 40 queries for 1 projected attribute and 8 queries for 5 projections. Overall, we see the same trend as before: the more threads available, the more the advantage shifts to the sorting side. Still, if only few threads are available for the initialization step, parallel sideways cracking shows a significantly smaller preparation time. Further, for tables consisting of multiple hundreds of attributes, only on-demand initialization of columns is a viable option, as offered by parallel sideways cracking.

5.7 Skew in the context of parallelism

Up to this point, we evaluated the parallel methods under a uniformly distributed random workload on top of uniformly distributed data. In the following, we will investigate how different kinds of skewness affect the parallelism. We will test both skewed query predicates as well as skewed input data. Furthermore, we cluster the input data into range partitions and inspect the impact on the methods. Precisely, we run the following configurations independently:

1. The query predicates follow a normal distribution with mean $\mu = 2^{63}$ (middle of the domain). The deviation is varied from $\sigma = 2^{58}$ (high skew) to $\sigma = 2^{62}$ (low skew). This pattern simulates a high interest in certain keys.
2. The keys of the input data follow a normal distribution with mean $\mu = 2^{63}$ (middle of the domain). The deviation is varied from $\sigma = 2^{58}$ (high skew) to $\sigma = 2^{62}$ (low skew). This pattern simulates a higher appearance frequency of certain keys.
3. The keys of the input data follow a uniform distribution. However, the input is physically clustered into k uniform range partitions. We test a low clustering using $k = 4$ and a high one using $k = 60$. This pattern simulates data where the key locality resembles physical locality, typically the case for sensor or financial data.

Figure 23 shows the results in form of speedup factors that different methods achieve when switching from the uniformly random distributed data and queries seen so far to the respective form of skewness. A factor below 1 indicates a speedup. We compare the runtimes of the entire query sequence of 1000 queries. Figure 23a shows the influence of skewed query predicates on the methods. We can observe that only P-SC is affected negatively by the skew with a slowdown of up to 1.2×, interestingly even the low skew triggers it. All remaining methods improve with a higher selectivity by factors between 0.67× (P-CSC) and 0.91× (P-CGI) for a deviation of 2^{58}. P-SC suffers from the focus on a certain data region due to a higher lock contention, P-CGI can outweigh this problem with the initial range partitioning. The remaining algorithms exploit the denser access locality that result in more fine granular cracks and a better cache utilization. The parallelism of the chunked algorithms is not affected at all by the skewed predicates as the work balance remains the same. Figure 23b presents the impact of skewed input data. As we can see, this has a more severe influence on some of the methods. P-SC and especially P-RPRS are heavily slowed down by a factor of 2.10× and 6.42×, respectively, for the highest skewness. P-SC suffers from the fact that queries falling into

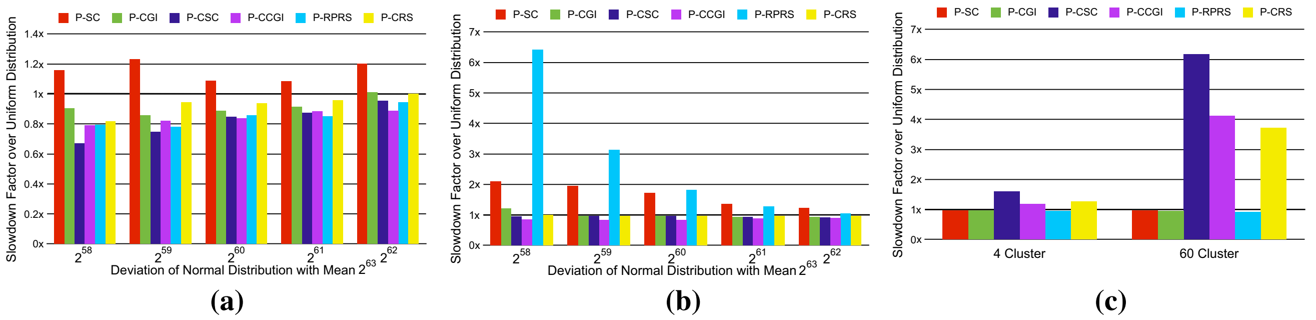


Fig. 23 Impact of skewness variants on the methods for 60 threads. The shown numbers present the speedup over the uniform random dataset using uniformly distributed query predicates. A number smaller than 1 represents a speedup of the version under skew. **a** Skewed queries, **b** skewed input, **c** clustered input

the skewed region work on a larger part of the column and thus limit the amount of possible parallelism. P-RPRS has the problem that the range-partitioning phase creates partitions of unbalanced size, and thus, the following sorting work is unequally divided among the threads. The creation of equi-depth partitions could help here; however, we leave this to future work. Again, the chunked methods are completely unaffected by this type of skew. However, the picture changes when data clustering is introduced in Fig. 23c. We test a lower clustering of 4 partitions and a heavy clustering of 60 partitions. Under these circumstances, the chunked algorithms experience a severe slowdown. The pre-clustered input leads to an unbalanced work division, as only some of the chunks contain data that is relevant for the query. P-CSC suffers the most, as its entire behaviour is query driven and thus influenced by the clustering. For P-CCGI and P-CRS, at least the range partitioning and sorting is query independent and thus balances well. To resolve this problem, a cluster-aware chunk division would be necessary, e.g. as proposed in [5]. However, this is left for future work.

Overall, we learned that the chunked methods are completely resilient to both skewed queries and input. However, in their current state, they have severe problems in handling clustered input. P-RPRS suffers from skewed input as the range-partitioning phase creates equi-width partitions that do not balance the sorting work. P-SC reacts negatively to both skewed input and queries due to the higher contention.

6 Lessons learned and conclusion

Let us now put together the major lessons learned.

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 penalize 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 eight previous database cracking works, including standard cracking using crack-in-two and crack-in-three [17], predication cracking [23], hybrid cracking [16], sideways cracking [15], and stochastic cracking [11] as well as the whole line of parallel cracking works [2,8,9]. We reimplemented the cracking algorithms from each of these works and tested them under similar settings as in the previous works. Our results match very closely to the ones 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. We can say that cracking is repeatable in any ad hoc query engine, other than MonetDB as well.
3. *Still, lot of potential to improve database cracking* There is still a lot of potential to do better in several aspects of database cracking, 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 initialization time to create a coarse-granular index which improves query robustness. All these are promising directions in the database cracking landscape. Thus, 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 character, 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 across the data. In contrast to that sequential patterns crack the index in small steps and the algorithms have to rescan large parts of the data. Skewed access patterns lead to a high variance in runtime depending on whether the query predicate hits the hotspot area or not. Overall, stochastic cracking (MDD1R) and coarse-granular index, which extend their query- driven character by data driven influences, are less sensitive to the query access pattern than the methods that take only the seen queries into account.
5. *Workload selectivities affect the amount of indexing effort in database cracking* Since cracking reorganizes only the accessed portions of the data, the total indexing effort varies with the query selectivities. In fact, the total indexing effort in standard cracking drops by 45 % when the selectivity changes from 10^{-5} to 10^{-1}. Although high selectivity queries reorganize smaller portions of the data, the reorganization happens much more often before reaching the final state. Additionally, earlier cracking works suggested to stop data reorganization at a certain partition size, in order to reduce the indexing effort. However, we saw that the overhead of additional filtering eclipses the savings from indexing effort.
6. *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 optimized 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 1 M queries. We note two things here: (1) the cracker index offers much slower lookups than modern main memory indexes, and (2) 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.
7. *Database cracking needs to improve mapping to parallel hardware* We inspected several different parallel cracking algorithms that use either inter- or intra-query parallelism and compared them in terms of scaling with available hardware resources and absolute runtimes with sort-based approaches. We identified lock contention and the shared memory bus as main limitations for parallel cracking algorithms. In terms of absolute query response times, the sorting methods are a hard match for their cracking-based competitors and offer nice additional properties like interesting orders—however, only if a large number of threads is available. This picture is confirmed in the tuple reconstruction case, where parallel sideways cracking is the winner over parallel clustering only under limited computing resources. Skew affects the parallel algorithms at different degrees depending on its type: a higher skewness is preferred by most algorithms although, e.g. clustered input heavily throttles certain methods in their current realizations.
8. *Different indexing methods have different signatures* We looked at several indexing techniques in this paper. Let us now contrast the indexing 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 (Fig. 24). This measure essentially acts as a signature of different indexing methods. The x-axis shows the normalized accumulated lookup and data access time (querying progress) and the y-axis shows the normalized accumulated data shuffle and index update time (indexing progress). We can see that different indexing methods have different curves. For example, standard cracking gradually builds the index as the queries are processed, whereas full index builds the entire index before processing any queries. Hybrid crack sort and hybrid sort have steeper curves than standard cracking, indicating that they build the index more quickly. On the other hand, stochastic cracking has a much smoother curve. Sideways

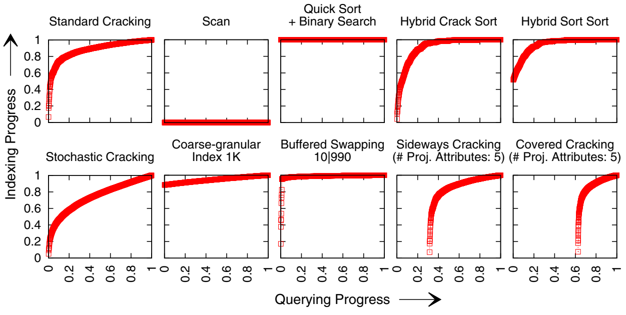


Fig. 24 Signatures of indexing methods

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 characterizes its indexing behaviour. Furthermore, there is still lot more room to design adaptive indexing algorithms with even more different indexing signatures.

Acknowledgments Special thanks to Stratos Idreos for helping us in understanding the hybrid methods. Work partially supported by BMBF.

Compliance with ethical standards

Competing interests As we re-evaluate research, there are potential competing interests with CWI Amsterdam and the authors of [8].

References

1. Adelson-Velsky, G., et al.: An algorithm for the organization of information. In: USSR Academy of Sciences, pp. 263–266 (1962)
2. Alvarez, V., Schuhknecht, F.M., Dittrich, J., Richter, S.: Main memory adaptive indexing for multi-core systems. In: DaMoN, Snowbird, UT, USA, pp. 3:1–3:10 (2014)
3. Bayer, R., McCreight, E.M.: Organization and maintenance of large ordered indices. Acta Inf. **1**, 173–189 (1972)
4. Birkeland, O.R.: Searching large data volumes with MISD processing. Ph.D. Thesis (2008)
5. DeWitt, D.J., Naughton, J.F., et al.: Practical skew handling in parallel joins. In: VLDB, Proceedings, pp. 27–40 (1992)
6. Finch, T.: Incremental Calculation of Weighted Mean and Variance. University of Cambridge Computing Service, Cambridge (2009)
7. Generalized Heap Impl. https://github.com/valyala/gheap
8. Graefe, G., Halim, F., Idreos, S., et al.: Concurrency control for adaptive indexing. PVLDB **5**, 656–667 (2012)
9. Graefe, G., Halim, F., Idreos, S., et al.: Transactional support for adaptive indexing. VLDB J. **23**(2), 303–328 (2014)
10. Graefe, G., Kuno, H.: Self-selecting, self-tuning, incrementally optimized indexes. In: EDBT, pp. 371–381 (2010)
11. Halim, F., Idreos, S., et al.: Stochastic database cracking: towards robust adaptive indexing in main-memory column-stores. PVLDB **5**, 502–513 (2012)
12. Hildebrandt, P., Isbitz, H.: Radix exchange: an internal sorting method for digital computers. J. ACM **6**(2), 156–163 (1959)
13. Hoare, C.A.R.: Quicksort. Commun. ACM **4**(7), 321 (1961)
14. Idreos, S., Kersten, M., Manegold, S.: Updating a cracked database. In: SIGMOD, pp. 413–424 (2007)
15. Idreos, S., Kersten, M., Manegold, S.: Self-organizing tuple reconstruction in column-stores. In: SIGMOD, pp. 297–308 (2009)
16. Idreos, S., Manegold, S., et al.: Merging what's cracked, cracking what's merged. PVLDB **4**, 586–597 (2011)
17. Idreos, S., et al.: Database cracking. In: CIDR, pp. 68–78 (2007)
18. Kersten, M., et al.: Cracking the database store. In: CIDR, pp. 213–224 (2005)
19. Kim, C., et al.: FAST: Fast architecture sensitive tree search on modern CPUs and GPUs. In: SIGMOD, pp. 339–350 (2010)
20. Leis, V., et al.: The adaptive radix tree: ARTful indexing for main-memory databases. In: ICDE, pp. 38–49 (2013)
21. Martinez-Palau, X., Dominguez-Sal, D., et al.: Two-way replacement selection. PVLDB **3**, 871–881 (2010)
22. McCalpin, J.D.: STREAM benchmark, version from January 17. https://www.cs.virginia.edu/stream/FTP/Code/stream.c (2013)
23. Pirk, H., Petraki, E., Idreos, S., Manegold, S., Kersten, M.L.: Database cracking: fancy scan, not poor man's sort! In: DaMoN, Snowbird, UT, USA, pp. 4:1–4:8 (2014)
24. Rao, J., Ross, K.A.: Making B+-trees cache conscious in main memory. In: SIGMOD, pp. 475–486 (2000)
25. Schuhknecht, F.M., Jindal, A., Dittrich, J.: The uncracked pieces in database cracking. PVLDB **7**, 97–108 (2013)
26. Schuhknecht, F.M., Khanchandani, P., Dittrich, J.: On the surprising difficulty of simple things: the case of radix partitioning. PVLDB **8**, 934–937 (2015)