An Experimental Evaluation and Analysis of Database Cracking Coverletter

Organization of this Document

First of all, we want to thank the reviewers for their helpful comments. In this cover letter, we document all the changes that we did. In addition, we attach a revised version of our paper where all changes over our previous submission are marked in blue. This will help you to track down the changes. Besides discussing / answering / documenting each of the comments / questions / revision requests (**"Answer"** part), we also explicitly indicate the changes we did in the paper to help you track how the revised version improved over our first submission (**"Changes in the Paper"** part).

1 Reviewer #1

Reviewer #1: C1) This paper is a well-executed extension of a PVLDB experimental track paper of high quality. The extensions in the journal submission are, in a big picture, as follows:
- It considers the issue of CPU efficiency in single-threaded as well as multi-threaded parallel cracking algorithms.
- It provides new experiments on the effect of selectivity, cracking granularity, and column cardinality.
The authors make good and fair use of their own previous work that has appeared since the original PVLDB paper, while they also investigate other works that have appeared since then.
Answer. Thank you.

Reviewer #1: C2) Footnote 2 is a bit surprising and does not match the flow of the paper up to that point. It was not hinted before that "the core purpose of database cracking is to provide a lightweight secondary index on non-key columns", and it is not clear if that is a "core" purpose.
Answer. The footnote was indeed out of place and cracking could indeed be used for both primary as well as secondary indexing. We have removed the footnote to avoid the confusion.
Changes in the Paper. We removed the footnote.

Reviewer #1: C3) The use of the term "selectivity", as it appears in this extension and the original paper, can be confusing. It seems that this problem exists all over the database literature, yet it would not harm to offer a clarification, in this extended version, that "very low selectivity" implies *larger* selectivity value, more matching results, large ranges, or any other equivalent expression.
Answer. Thanks for pointing out this problem, which indeed causes confusion in the literature again and again. We have added a clarification on the term "selectivity" the first time it appears.
Changes in the Paper. We have added footnote 2 to the paper.

Reviewer #1: C4) The experiment on varying the number of duplicates, and that on varying selectivity, do not really offer distinct lessons. It is the same lesson that they offer, as the two concepts are related to each other. This matter could be highlighted by putting these two experiments closer together, preferably in the same section, and thereby also address point (2) above in the process.
Answer. We have focussed on selectivity in Section 4 and decided to remove the discussion on duplicates entirely due to space constraints. Further, we shortened the Section 4.3, which now focuses on indexing time, index lookup time, and data access time.
Changes in the Paper. We removed Section 4.6 (old version) and updated the contributions accordingly.

Reviewer #1: C5) The conclusions at the end could also include lessons learned regarding selectivity & cardinality (which is one lesson, see above), and lessons learned regarding depth of cracking.

Answer. We have added these additional lessons learned.
Changes in the Paper. We updated lesson 5 in Section 6.

Reviewer #1: C6) The term ”chunk” appears to be overloaded in the paper. It is used throughout the discussion on parallel cracking algorithms, referring to ”independent chunks” on which cracking is applied in parallel, yet it is also used in the context of vectorized cracking, referring to ”large chunks of adjustable size” that data is partitioned into. Then, it would seem that ”Parallel-chunked vectorized cracking” employs ”chunks” in both of these senses. It may clarify the field if an alternative term is used for the second notion of ”chunk” in Page 16. It seems that ”vector” is used for that concept elsewhere. Then it would be good to say ”vectors of adjustable size” as well.
Answer. Thank you for pointing this out, this was indeed an overloaded term. We renamed the term “chunk” in the context of vectorized cracking to “block” and sticked to “chunk” in the context of parallelization, as we believe it is more widely known in that topic.
Changes in the Paper. We renamed “chunk” to “block“in the context of vectorized cracking.

2 Reviewer #2

Reviewer #2: C1.1) This paper extends the cracking the uncracked pieces work in various directions. Some of these, in particular sec4.4 (low selectivities) and sec 5.4 (result merging) I find not very relevant and my request is to remove these sections entirely.
Answer. Thank you for pointing this out. Our initial intention was to experimentally cover as many aspects of adaptive indexing as possible. This included the investigation of low selectivities which lead to the creation of a partially indexed cracker column as well as the analysis of different merging techniques, as the discussion whether to merge or not arises constantly in the context of parallel-chunked algorithms. However, after a detailed reevaluation of our parallel section, we fully agree with you that it makes sense to sacrifice these chapters in favor of a more detailed and in-depth analysis of the parallel algorithms themselves.
Changes in the Paper. We removed Section 4.4 (old version) and Section 5.4 (old version) entirely and also updated the contributions accordingly.

Reviewer #2: C1.2) What is very relevant is the extension towards multi-core parallelization (nicely combining with [3], tying all cracking work together in a single journal paper). This, however, shows a dismal picture which would lead to the conclusion that cracking is dead. Better to sort in parallel and binary search the data.
Answer. We also believe that parallelization of indexing algorithms is crucial nowadays and will become more and more important in the future. That is why both cracking and sorting methods have to scale well with the available resources. Our reinvestigation of the parallel algorithms indeed confirms, that the tested sorting algorithms scale better on highly parallel hardware than the adaptive counterparts that currently exist. Our completely rewritten in-depth analysis and profiling of the individual methods presents the reasons.

Nevertheless, we would not say that cracking is dead. Indeed, if a high number of threads are available for sorting, a vast amount of data can be ordered surprisingly quick nowadays. However, we believe that in a real system, often only a portion of the hardware resources are actually available for a single query/column. In this case, adaptive indexing still shows a significantly smaller initialization time. A similar behavior we see in our new investigation of tuple reconstruction in the context of parallel algorithms. As we can see in Figure 21 in the paper, if only a small number of threads is available, cracking can still pay off. This holds especially for tables with many columns where the access pattern is unknown.
Changes in the Paper. We discussed the relationship of cracking and sorting in the text more carefully and stated the situations explicitly in Section 5.5 and Section 5.6, in which we believe cracking has a clear advantage over upfront sorting.

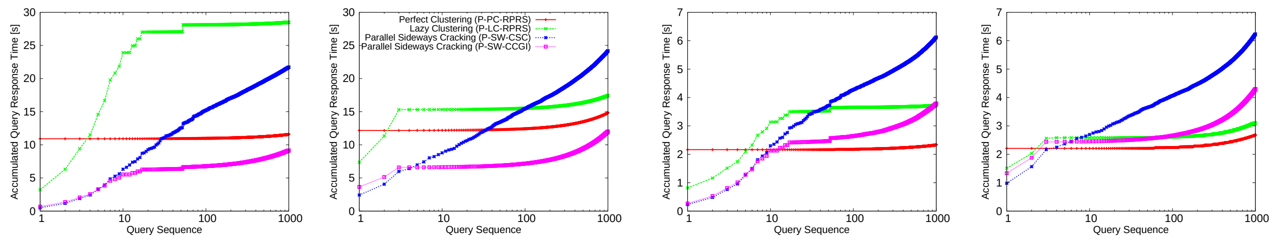
Reviewer #2: C1.3) However, the evaluation that leads to this conclusion is regrettably ill-defined (goals, setup??), ...
Answer. We agree with you that the experimental setup of the evaluation of parallel algorithms was underspecified. We extended the write-up by two sections, that explicitly state the testing configurations. Section 5.2 (Hardware Setup) gives a detailed explanation of the test machine while Section 5.3 (Experimental Setup) shows how exactly the queries are fired to the individual methods. Additionally to the explanations in the write-up, let us summarize the configurations in the following.

First of all, let us discuss our (updated) hardware setup. For the revised version, we decided to switch to a state-of-the-art high-end server consisting of 4 sockets equipped with Intel Xeon E7 4870 v2 processors and 60 physical cores (120 logical cores) in total. The motivation of this was two-folded: Firstly, on a machine with such a large number of computing cores, any scaling problem will be even more visible and thus easier to analyze. Secondly, we want to use Intel VTune Amplifier 2015 to profile the methods, where certain analysis types are only supported on up-to-date architectures.

Additionally, let us explain the way in which queries are fired to the methods. We have to distinct between intra- and inter-query parallel algorithms. For algorithms that perform inter-query-parallelism, like P-SC for instance, we divide the set of queries into k parts, if we want to parallelize the processing using k threads and each thread has to work $1000/k$ queries sequentially. This setup equals to the one used in [8] which introduced P-SC. In contrast to that, for the 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 setup, we introduced a barrier in the query execution loop: the answering of the next query starts only after all threads completed the current one.
Changes in the Paper. We extended Section 5.2 (Hardware Setup) and Section 5.3 (Experimental Setup).

Reviewer #2: C1.4) ... superficial in its investigation (memory bandwidth, turbo mode, numa effects are blamed but not analyzed in detail), and incomplete.
Answer. Passing through the parallelism section again indeed showed us what our earlier version was lacking in terms of depth and analysis. We heavily changed the discussion of parallel algorithms by basically redoing the entire chapter from scratch. First of all, as mentioned already, we switched to a more interesting hardware configuration. This also enabled us to profile the algorithms in a deeper way, as the old architecture did not support many measurements (e.g. bandwidth utilization). We will discuss all additional evaluation aspects such as bandwidth, turbo mode and NUMA effects each in the separate comments below.
Changes in the Paper. We heavily increased the analysis depth and profiled the algorithms using Intel VTune Amplifier 2015. Please see our answers to your detailed comments below.

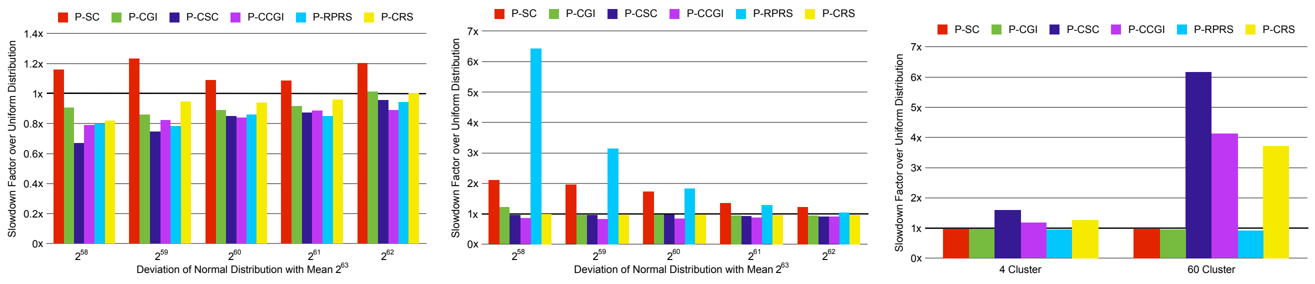
Reviewer #2: C1.5) The latter issue is that in terms of practicality, one should look at sideways cracking (in parallel) because that is what queries in the real world need, ...
Answer. Thank you for this point. Indeed, in reality, tuple reconstruction has to be performed in some way to answer the queries and sideways cracking is the way to do it for database cracking. Although we already analyzed its behavior in the single-threaded context, we believe as well that it is important to evaluate its parallel behavior. Since, to the best of our knowledge, a parallel sideways cracking algorithm has not been described or tested so far, we introduced our own concept in this work.



(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

Figure 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 Figures 22(a) and 22(c), each query projects a single randomly chosen attribute. In Figures 22(b) and 22(d), each query projects five randomly selected attributes.

As the basis, we picked the two best performing cracking algorithms P-CSC and P-CCGI, which both rely on chunking for parallelization. Thus, we can apply the concept of chunks as well on sideways cracking by replicating the cracker maps and tapes for each chunk and processing them individually. As baselines, we implemented two sort-based versions, that cluster the table with respect to the sorted index column using our parallel range-partitioned radix sort algorithm. The first version clusters the entire table directly in the first query, while the latter version does it lazily when a column is touched, similar to the behavior of sideways cracking. Figure 22 shows the new results. We decided to present 4 and 60 threads as two extremes and varied the number of randomly projected attributes. We can observe the low initialization time of parallel sideways cracking, that especially pays off for 4 threads. Sorting the entire table shows of course the most stable performance. The lazy clustering is affected by the number of projected attributes and shows its strength again especially for a large number of threads. Overall, we see a similar trend as before: the larger the number of threads, the more the advantage shifts towards the sorting site because of the better scaling capabilities.
Changes in the Paper. Added Section 5.6 to analyze tuple reconstruction in the context of parallelization. This includes the new Figure 22, that shows the accumulated query response time of the parallel sideways cracking and parallel clustering methods. We decided to analyze parallel tuple reconstruction in a separate section and not the entire context as we want to analyze the parallel algorithms alone in depth. Mixing the two layers would have unnecessarily complicated the evaluation.



(a) Skewed Queries (b) Skewed Input (c) Clustered Input

Figure 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.

Reviewer #2: C1.6) ... as well as skewed workloads (which is were cracking shines)

Answer. We agree with you that skewness is indeed an interesting point that is worth looking at in the context of adaptive indexing. Therefore, we added a new section solely about the behavior of the methods under different types of skewed input data and queries. We believe three kinds of skewness are of special interest: (1) skewed queries to simulate a special interest in a certain region, (2) skewed input to simulate a higher appearance frequency of certain values, and (3) clustered input where the key locality fits the physical locality to simulate real-world data e.g. produced by sensors. For (1) and (2), we generate the queries respectively the input based on a normal distribution, where we vary the deviation from 2^{58} (high skew) to 2^{62} (low skew) around a mean of 2^{63} (middle of the domain). The clustered input is realized using a range-partitioning of 4 partitions (low clustering) and 60 partitions (high clustering). The new results are shown in Figure 23 for 60 threads. We can see that the individual methods are influenced differently by the skewness. The chunked methods for instance are completely unaffected by skewed queries and input, but suffer heavily from clustered input. P-RPRS can not balance the sort work evenly under skewed input while the access contention of P-SC in a certain area worsens the locking problem. Indeed, we can see that the remaining cracking algorithms gain from the focus on a certain area. Overall, we performed a detailed discussion of the results in the new Section 5.7.

Changes in the Paper. We added Section 5.7 that discusses the impact of different forms of skewness on the individual adaptive indexing and sorting methods. This includes the new Figure 23, testing the methods under skewed queries, skewed input and clustered input.

Reviewer #2: C1.7) The current (not so clearly articulated) conclusion would be that cracking is dead, because it does not scale (factor 3 on 12 cores). As said, the analysis on which this is based is sloppy and superficial. More evidence and indeed optimization attempts are needed before accepting this conclusion.

Answer. Please see our answer to your earlier comment C1.2.

Reviewer #2: C1.8) Even if the situation is bad, I would ask for experiments on a skewed workload, because in a skewed workload the hot part will fit in the CPU caches and this would at least be one pocket were there is no ground to think that it would not scale (i.e. memory bandwidth cannot be a bottleneck there).

Answer. From the experiments we can learn that indeed all tested cracking methods except of P-SC significantly improve from skewed queries, that focus on a certain region. The higher the skew, the higher is the improvement. For P-CSC, we observe a speedup over the uniform case of around 0.65x. This is due to the higher cracking density around this area and the improved cache-locality of the queried region. However, this is also exploited by the sort-based methods, where the hot part, that is queried again and again, resides in the caches.

Changes in the Paper. In Section 5.7, we describe how and why the methods exploit the skewed queries.

Reviewer #2: C1.9) As such, I think that this paper is a missed opportunity. Because the authors have lot's of the pieces readily at their disposal, I still have hope that they can turn the paper in my desired direction in a major revision, which is why I do not reject it (yet).

Answer. You are right that we have by now a huge codebase available with numerous cracking and sorting algorithms. Thank you for the second chance, we believe that we significantly improved our investigation.

Reviewer #2: C2) sec1: to make the paper a bit more self-contained, it would be good in the Experimental setup description to explain what the maximum Turbo Frequency of the CPU is.

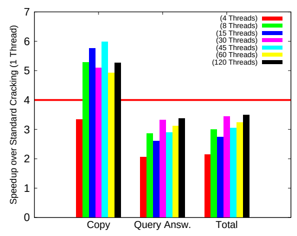
Answer. For the revised evaluation, we decided to disable Intel Turbo Mode entirely for all parallel experiments. By default, the processor can increase its base frequency of 2.3GHz to 2.9GHz. This automatic overclocking does not only happen if not all cores are utilized but also when certain temperature limits are not yet reached. Thus, the behavior of this feature is rather uncontrollable. Overall, we believe Turbo mode unnecessarily complicates any analysis on parallel algorithms, especially when it comes to the scaling of methods and thus, we disabled it this time entirely.
Changes in the Paper. We disabled Intel Turbo Mode in the BIOS.

Reviewer #2: C3) sec 2.4: should multi-core parallelization not be one of the key concerns of cracking? especially given the results shown later on..
Answer. Indeed, we identified parallelization as a key concern of cracking. However, at that early point of the paper, we want to focus on the cracking algorithms themselves, while we believe parallelization sits on a layer on top. Nevertheless, in Section 2.4, we added a short note about the challenge of parallelization referring to the latter multi-threading investigation.
Changes in the Paper. We added the parallelization efficiency to the list of key concerns of standard cracking, that is presented in Section 2.4.

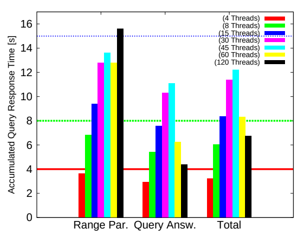
Reviewer #2: C4) sec 4.4: this new addition feels a bit like focusing on an artifact of cracking: indeed, in case of non-selective queries (more than 50% of the data gets selected), then necessarily the split lines are concentrated in the beginning and end and much of the data (in the middle) is left untouched, which affects the behavior of the algrithm. However, with more than 50% of the data selected, the performance of the cracking algorithm is irrelevant in the whole query performance picture, since the scan/filter that it implements is not the major cost of any reasonable query (excluding queries that perform only a straight COUNT(*) on it , as implemented in MonetDB, which happens no to touch any data, just measure the size of the region – any other query will touch all returned data, which is expensive on large results). The interesting behavior where most of the data is left untouched are those scenarios were people are not interested at all in certain regions of their database. Here cracking can shine, but this is already covered by experiments with the skewed workload (though arguably there could be regions with zero queries, the skewed workload just has regions with a low nonzero access probability). As such, I am not interested in 4.4 as an extension.
Answer. Thanks for comments and lower selectivity indeed leaves cracker indexes less useful anyways. Therefore, we have removed Section 4.4 (old version) from the extension.
Changes in the Paper. We removed Section 4.4 (old version) entirely.

Reviewer #2: C5) sec 5: the section starts without proper explanation of how parallelization is tested: how many client sessions are connected to the system. Oops, there are probably no client sessions at all – maybe better ask how many queries are taken into execution concurrently and what (queue?) mechanism is used for that. One could test concurrency at any N¿=1 parallel degree with special interest for both power (N=1, 1-at-a-time) and throughput (N=C, for C logical cores or N=2C for physical cores).
Answer. For the query firing setup, please see our answer to your comment of C1.3. For the parallel degree we picked 4, 8, and 15 threads to utilize up to $1/4$-th of the machine. Additionally, we tested 30, 45 and 60 threads to increase the number of utilized sockets up to all physical cores. Also, we tried 120 threads to utilize all hardware threads and logical cores of the machine. The single-threaded version of the algorithm serves always as the baseline. Please note that we use a special single-threaded implementation, not the parallel version executed with one thread. It is also worth mentioning, that we do not apply any thread pinning to sockets or cores. Therefore, it is possible that e.g. the 8 threaded case is distributed to all 4 processors to utilize the aggregated memory bandwidth of all channels. We believe that it makes sense to leave this distribution to the operating system. Regarding that, the input data is distributed evenly onto all 4 NUMA regions, with the k-th chunk of the array placed on NUMA region k.
Changes in the Paper. The query firing setup is described in detail in Section 5.3. The motivation for the tested thread numbers is explained in Section 5.4.

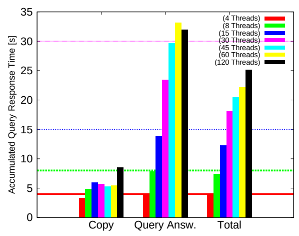
Reviewer #2: C6.1) fig 20/21: P-SC/P-CSC and P-CGI/P-CCGI have opposed behavior for scaling wrt progressing the query sequence. The reasons for this are not sufficiently explained, nor investigated in the paper. Given the dismal scaling results, this lack of depth (just stating "P-SC scales less well" is not enough) makes the addition of section 5 currently not acceptable. It needs a major revision that properly investigates the reasons for non-scaling or fixes the scaling. The authors at best allude to causes for the non-scaling.
Answer. As already mentioned, we basically redid the entire parallel evaluation from scratch to get a deeper insight into the behavior of the algorithms. First of all, we changed the way in which the scaling of the methods is presented. In the previous version, we showed the accumulated speedup up to the current query. The problem of this visualization is that methods with a long initialization phase overshadow all remaining parts. Consequently, for P-RPRS for instance, it is not possible to tell



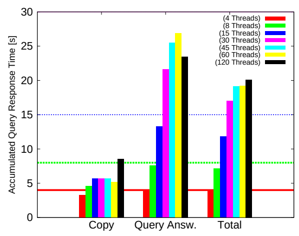
(a) Parallel Standard Cracking (P-SC)



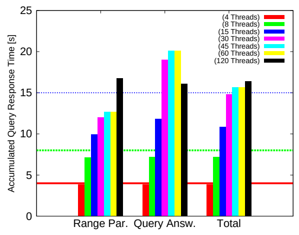
(b) Parallel Coarse-granular Index (P-CGI)



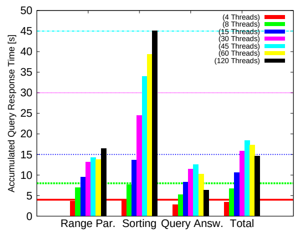
(c) Parallel-chunked Standard Cracking (P-CSC)



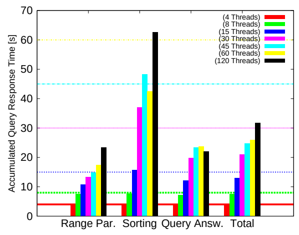
(d) Parallel-chunked Vectorized Cracking (P-CVC)



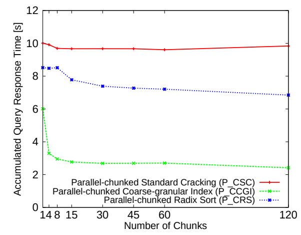
(e) Parallel-chunked Coarse -granular Index (P-CCGI)



(f) Parallel Range-partitioned Radix Sort (P-RPRS)



(g) Parallel-chunked Radix Sort (P-CRS)



(h) Serialization of Chunked Algorithms

Figure 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 of the algorithms as well as the overall achieved speedups. Horizontal lines show the expected linear speedup. In Figure 18(h), we show for the chunked algorithms how the chunking itself influences the methods by serially working the chunks.

whether the query answering phase scales in a good or bad manner. Especially hard are comparisons across methods with this plot style. In the revised version shown in Figure 18, we instead show the accumulated speedup of individual phases like *copying*, *range-partitioning*, *sorting*, and *query answering*. Of course, we also present the total scaling. This makes it possible to compare phases easily across methods. We also present the linear scaling in form of horizontal lines. In Figure 18(h), we further show the influence that might be induced by the chunking itself. For instance, it is cheaper to sort multiple small chunks over sorting a single large one, independently from the parallelization aspect.

The detailed analysis of the scaling of the individual algorithms is presented in Section 5.4. Nevertheless, let us present here some of the main points demonstrating how we approached the analysis in the case of P-SC and P-CGI shortly.

P-SC still scales particularly bad with the number of threads with a maximal speedup of around 3.5x. As the algorithm uses locking on the partition level and partitions are coarse-granular in the early phase, a natural consequence is an induced serialization at the beginning. To demonstrate that, we plotted Figure 17. It visualizes the access contention on the cracker column (y-axis) with respect to time (x-axis) for 8 threads. Overlapping regions indicate, that at least two threads want to work the same area. From that plot, we can clearly see that around half of the entire runtime is heavily serialized by access contention. For instance the first crack, that locks the entire column takes around 400ms to finish. During that time, all other threads have to wait. Thus, starting with a fresh, uncracked column must lead to a poor scaling behavior. This problem can be reduced by prepending a range-partitioning step as done by P-CGI, as shown in Figure 17(b).

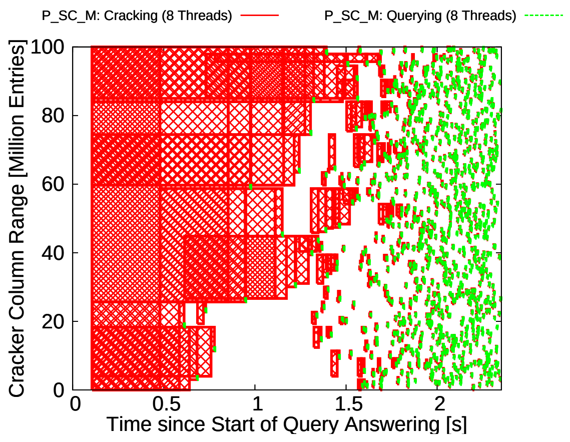
Changes in the Paper. We increased the analysis depth of the scaling discussion by adding various kinds of profiling evidences, that substantiate our claims in Section 5.4. We redid Figure 18 that show the scaling capabilities of the methods. Further, we plotted Figuren 17 to explicitly visualize the lock contention of P-SC and P-CGI.

Reviewer #2: C6.2) First there is Turbo Mode, although we would like to know what the highest observed CPU frequency was. Also, it would be beneficial for the sake of analysis to also have experiments with Turbo Mode disabled, just to see to which extent the limitation is in the data/algorithm (rather hardware), and hopefully arrive at linear scaling at some point.

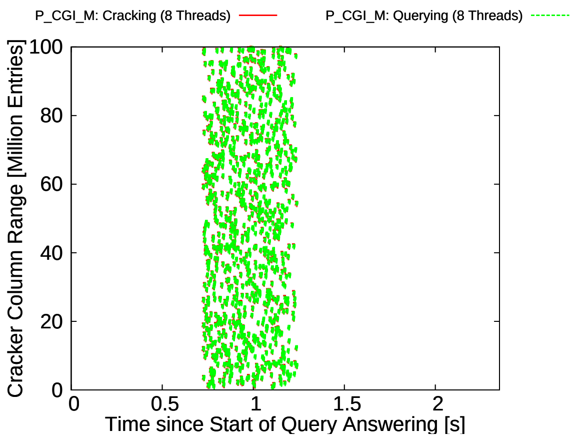
Answer. Please see our answer to comment C2.

Reviewer #2: C6.3) A second reason the authors name is "limited memory bandwidth". However, for such claims, minimally the authors should mention the memory bandwidth of their platform, both according to spec and according to their best microbenchmark (test it in parallel, e.g. stream benchmark). Further, it would be good to measure the actual bandwidth achieved by the cracking algorithms. If the latter is close to the maximum real micro-benchmark, the reader might get more convinced of the statement.

Answer. You are right, our bandwidth analysis was not profound enough. In the revised version, the measurement of bandwidth utilization is a central part of understanding the algorithms. First of all, we tested the STREAM benchmark on our



(a) Parallel Standard Cracking (P-SC)



(b) Parallel Coarse-granular Index (P-CGI)

Figure 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.

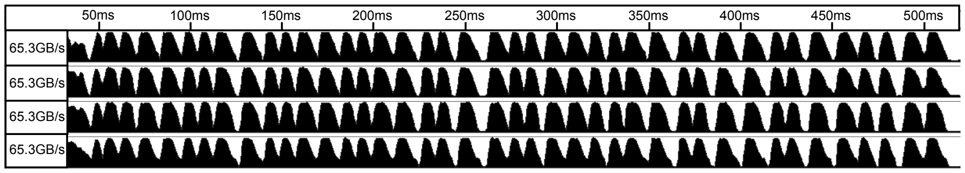


Figure 16: Stream Benchmark with 60 threads. We can reach 65 GB/s per socket for the aggregated read/write bandwidth per socket.

hardware and measured the bandwidth utilization in Intel VTune Amplifier 2015 directly at the memory controller. Figure 16 shows the results. Apparently, our machine shows an aggregated bandwidth of 65GB/s per socket and thus 260GB/s in total.

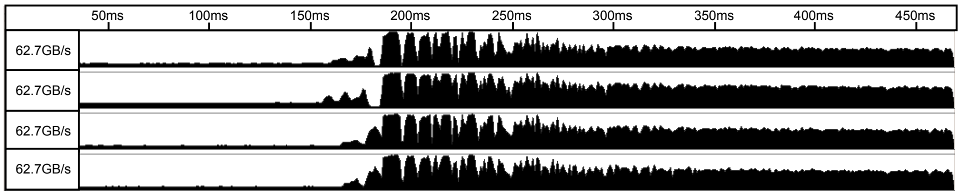
We now measured the bandwidth in the same way for the algorithms to understand their scaling behaviors. Consider for instance P-CSC, which is a clear candidate for being throttled by the memory transfer. In Figure 20(a), we present the bandwidth utlization for 60 threads of the entire run. We first see a very low utilization in the copying phase. Interestingly, this phase still scales poorly with the number of threads. We identified the page-fault handling when touching the array for the first time as the cause for this surprising behavior. Further, we see that the query answering phase utilizes the bandwidth for 60 threads in the early phase almost completely. When we compare the query answering phase for 30, 45, and 60 threads in Figures 20(b), 20(c), and 20(d) we see that already for 30 threads, when the poor scaling begins, the bandwidth limit is close to be reached. Therefore, we cannot expect a linear scaling from this point on anymore.

Similarly, we profiled the bandwidth for other algorithms and included it where is was help- and meanginful in the investigation. This was the case for P-CGI in Figure 19, which showed for instance that the range-partitioning is not bandwidth bound at all, so other causes for the bad scaling must be taken in consideration.

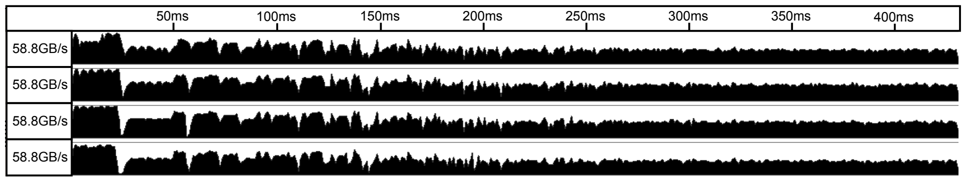
Changes in the Paper. We profiled the bandwidth utilization using Intel VTune Amplifier 2015 and included the results where they are helpful to explain the scaling behavior. This holds especially for the cracking algorithms. Therefore, we extended the paper with bandwidth plots in Figures 16 (STREAM benchmark), 20 (P-CSC analysis) and 19 (P-CGI analysis).

Reviewer #2: C6.4) Finally, the authors mention NUMA effects as a possible culprit without measuring inter-node memory traffic (there is an Intel tool for that) and without considering NUMA specific memory layout in cracking.

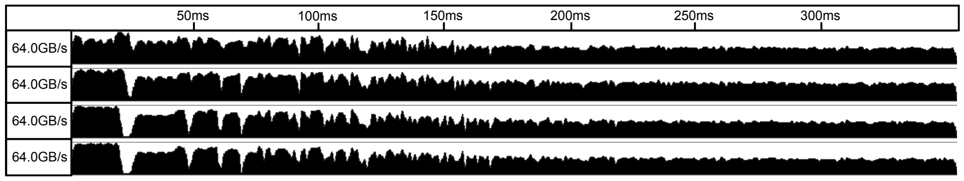
Answer. NUMA effects play an important role in parallelization — especially, when it comes to the discussion whether to chunk or not. To check the access type, we inspected two counters, that measure whether a miss in the last level cache is served via an access to local or remote DRAM. The results for our algorithms are shown in Table 6. We can clearly see that the chunked algorithm work as expected almost entirely on their local region, which is of course a clear advantage over the remaining methods. This holds of course also for the chunked cracking algorithms P-CSC and P-CCGI, which have in that sense a NUMA specific memory layout. In contrast to that, for inter-parallel algorithms, each thread can potentially work on any part of the data and thus access remotely stored data as well — the chance of local access is only at 25% on



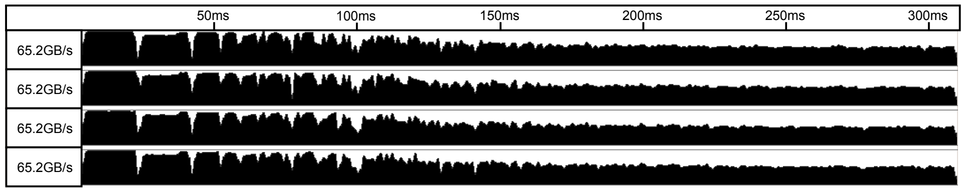
(a) Parallel-chunked Standard Cracking (P-CSC) with 60 threads. The initialization phase (copying the data into the cracker column in parallel) utilizes the bandwidth only partially (around 8GB/s).



(b) Parallel-chunked Standard Cracking (P-CSC) with 30 threads without initialization phase. Highest bandwidth observed: 58.77GB/s

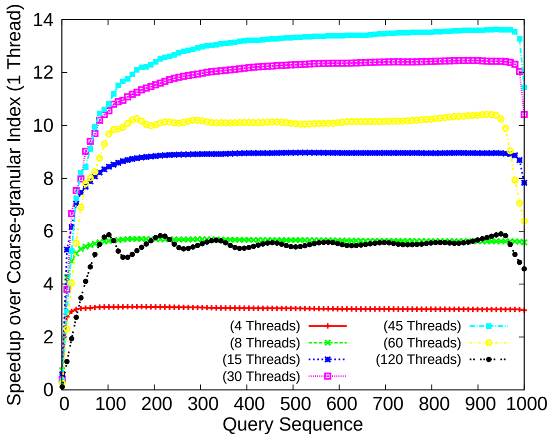


(c) Parallel-chunked Standard Cracking (P-CSC) with 45 threads without initialization phase. Highest bandwidth observed: 64.02GB/s

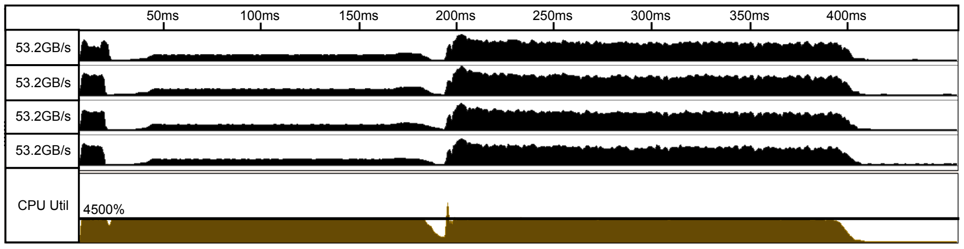


(d) Parallel-chunked Standard Cracking (P-CSC) with 60 threads without initialization phase. Highest bandwidth observed: 65.24GB/s

Figure 20: Bandwidth measured at 4 sockets with Intel VTune Amplifier 2015 in GB/s.



(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: 53.25GB/s

Figure 19: Bandwidth of P-CGI measured at 4 sockets with Intel VTune Amplifier 2015 in GB/s. The bottom line shows the CPU utilization in percentage.

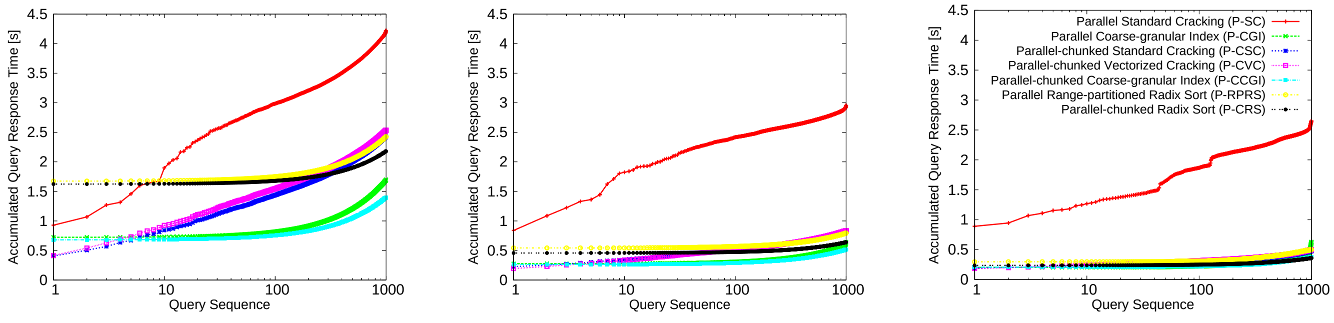
our four socket machine. This effect is especially visible when comparing the same phases across algorithms. For instance, although the query answering phase of P-RPRS and P-CRS is exactly the same except of the chunking, the latter one scales significantly better. This is due to the fact that it accesses its data locally.
Changes in the Paper. We added Table 6 and showed the effect of local and remote access on the tested algorithms.

Reviewer #2: C7) sec 5.3: "what matters most in the end is the raw runtime": I disagree. When on a 12-core system the scaling converges to a factor less than four even in the best alternative, this is the major concern and not absolute runtime. The reason is that the problem is already very bad (less than 30% system utilization), and likely to get worse with higher core counts.
Answer. You are right, this statement was misplaced in the presence of the scaling results we had. Nevertheless, we believe that we also have to look at the absolute runtimes in order to be able to compare the algorithms agains each other. The new results are shown in Figure 21 for 4, 15, and 60 threads. For 4 threads, we still see a clear difference between the methods: the cracking algorithms show a significantly smaller initialization time than the sort-based ones. This changes the more threads we use. For 60 threads, there is almost no difference visible anymore as the sorting scales significantly better than any cracking phase. Please see our answer to your next comment for the reasons.
Changes in the Paper. We updated the runtime analysis in Section 5.5.

Reviewer #2: C8) It would finally be very interesting to see a proper analysis of the parallel performance of the sort algorithms. From what I know of the sort benchmark, good parallel speedup can be achieved on that task. Is that also the case in these measurements? I would say that "good" in this case would be scaleup of a factor around 10. The big question is why cracking is so far behind.
Answer. You are right, as the sort-based algorithms show better scaling capabilities, it is important to understand why. To do so, let us visualize how the sort-based methods parallelize their initialization. First, there is the range-partitioning phase which scales poorly as it is heavily back-end bound. This phase is shared with the cracking algorithms P-CGI and P-CCGI. Afterwards, the 1024 partitions are divided among the threads and each one sorts them one by one. As each partition has a size of roughly 1.5MB and each core has a 2MB share of the 30MB L3 cache, a partition can be sorted entirely in cache. Therefore, the data to process is transferred exactly once to the processors, completely sorted locally and written back. This is basically the perfect case for parallelization, as the cores can operate independently from the (shared) memory system during

Table 6: Number of LLC cache misses that are served with local respectively remote DRAM access presented in millions of measured events. The measured counters are `OFFCORE_RESPONSE.DEMAND_DATA_RD.LLC_MISS.LOCAL_DRAM` and `OFFCORE_RESPONSE.DEMAND_DATA_RD.LLC_MISS.REMOTE_DRAM`.

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



(a) 4 Threads (b) 15 Threads (c) 60 Threads

Figure 21: Accumulated query response time of parallel cracking algorithms in comparison with parallel radix-based sorting methods.

their complete reorganization process. In comparison to that, a cracking algorithm that has to transfer the data again and again through the shared memory bus for small reorganization steps, consequently must scale worse.
Changes in the Paper. We added the explanation of the cracking and sorting relation to the text.

Reviewer #2: C9) sec 5.4: merging: I have a basic motivational problem with reading about merging which is similar to my objections to the low-selectivity experiments. That is, cracking (and indexing in general) is only really relevant for selection queries that are at least reasonably selective (say $\leq 1\%$) because otherwise the query would not be dominated by scan (which is the win of indexing/cracking), but rather by the further processing. The other problem withs sec5.4 is that honestly I do not understand it, because in my opinion this section is badly written. It is said that in chunk-based approaches in the average two of the three parts of a chunk need to be moved. It is unclear what three parts we are talking about, and why we need to move only two of them . A picture would help? It seems that trying to create a contiguous area will in the end lead to creating a single range-partitioned set of chunks (they start randomly partitioned), which is *not* desirable in the long run. Rather, in order to let all threads work together on one query it needs to remain partitioned uniformly. This, obviously, ties in to the problem already described: the authors have not properly described their parallelization goals (power or throughput? I would say both are needed, and power does not jive well with a range-based chunking). But given my non-interest to merging in the first place, I'd rather just leave out the whole section then try to fix it.
Answer. The section is verbose and loosely connected, therefore we have removed it.
Changes in the Paper. We removed Section 5.4.

Reviewer #2: C10) fig 23: it is unclear what the selectivity is, if it is only 1% then it is too big; it would be interesting to see smaller selectivities as well. Currently, the data in (b) shows that indeed merging is very expensive for the still relatively large 1% result (showing that cracking is in effect only marginally of influence in the overall query execution picture). Better go for the 0.1% selectivity, where it could be more decisive.
Answer. As we removed Section 5.4 entirely, we also removed Figure 23.
Changes in the Paper. We removed Figure 23.

Reviewer #2: C11) we distinct whether we interleave => distinguish? determine?

Answer. Thank you, this was indeed a typo.
Changes in the Paper. We removed the Section.

Reviewer #2: C12) Final words, please do not be discouraged by my at points sometimes string wording (this is just passion for the subject). I like the work and see the opportunity for a great journal paper. However, in this form I cannot accept it.
Answer. Thanks for your elaborate comments and feedback. We have tried to fix most of them. Finally, we want to point out that in a work which reevaluates eight adaptive indexing papers on 25 pages, a selection of experiments and analysis has to be made. We believe that in the revised write-up, we managed to balance topics, depth and content to create a comprehensive study of the state-of-the-art in database cracking.
Changes in the Paper. See above.

3 Reviewer #3

Reviewer #3: C1) The paper is an extended version of the authors VLDB 2014 paper ”The Uncracked Pieces in Database Cracking”. Fully in line with the good quality of the original paper, the authors considerably extended their experimental study and analysis of database cracking compare to the original paper. The paper now also covers new cracking approaches and variants that have been published since the publication of the authors original paper. Most notably, this includes the predication, vectorization and parallelization techniques proposed in [23] as well as the parallelization techniques proposed by the authors in [3]. While none of these techniques is new, the major contribution of this paper (as well as of the original paper) is an exhaustive experimental evaluation, analysis and comparison of most (if not all) variations of database cracking published so far. The authors clearly indicate where they are able to repeat and confirm published results, but also enter yet unexplored terrain and explain why their results differ in those cases. Overall, the paper is well structured, well written and rather educative.
Answer. Thank you!

References

[3] V. Alvarez, FM. Schuhknecht, J. Dittrich, S. Richter. Main Memory Adaptive Indexing for Multi-core Systems. In *DaMoN*, pages 3:1-3:10, 2014.

[8] G. Graefe, F. Halim, S. Idreos, et al. Concurrency Control for Adaptive Indexing. In *PVLDB*, volume 5, pages 656-667, 2012.

[23] H. Pirk, E. Petraki, S. Idreos, S. Manegold, ML. Kersten. Database Cracking: Fancy Scan, not Poor Man's Sort! In *DaMoN*, pages 4:1-4:18, 2014.