An Analysis and Comparison of Database Cracking Kernels

Immanuel Haffner
Saarland University

Felix Martin Schuhknecht
Saarland University

Jens Dittrich
Saarland University

ABSTRACT

Database indexes are a core technique to speed up data retrieval in any kind of data processing system. However, in the presence of schemas with many attributes it becomes infeasible to create indexes for all columns, as maintenance costs and space requirements are simply too high. In these situations, a much more promising approach is to adaptively index the data, i.e. the database gradually partitions (or *cracks*) those columns that are frequently used in selections. In doing so, the “indexedness” of a table adapts to the requirements of the workload. A large body of work has investigated *database cracking*, which is a subset of *adaptive indexing*.

Irrespective of their algorithmic behavior, essentially all these works have in common, that the proposed methods use a simple *two-sided in-place cracking kernel* at the core, which performs a partitioning step. As this partitioning makes a large portion of the total indexing effort, the choice of the kernel can make a factor of two difference in the running time for a method sitting on top.

To approach the topic, we first perform an experimental evaluation of existing state-of-the-art kernels and study their respective downsides in detail. Based on the gained insights, we propose both an advanced version of the best existing kernel as well as a new and unconventional approach, which utilizes features of the operating system as well as data parallelism. In our final evaluation of all kernels, we vary entry size, index layout, selectivity, and number of threads, and provide a decision tree to select the best cracking kernel for the respective situation.

ACM Reference Format:
Immanuel Haffner, Felix Martin Schuhknecht, and Jens Dittrich. 2018. An Analysis and Comparison of Database Cracking Kernels. In *DaMoN'18: 14th International Workshop on Data Management on New Hardware, June 11, 2018, Houston, TX, USA.* ACM, Houston, Texas, USA, 10 pages. https://doi.org/10.1145/3211922.3211930

1 INTRODUCTION

Database indexing is a classical way of achieving fast query execution. However, effectively pre-building indices requires knowing the workload a priori and having enough time to build the indices. These conditions are not always satisfied, e.g. in schemas with many hundreds of frequently accessed attributes, in streaming environments, or with temporally changing workloads. As a consequence, *Database cracking* [5, 11, 14, 21, 24] incrementally builds an *adaptive index* as a side product of answering incoming queries.

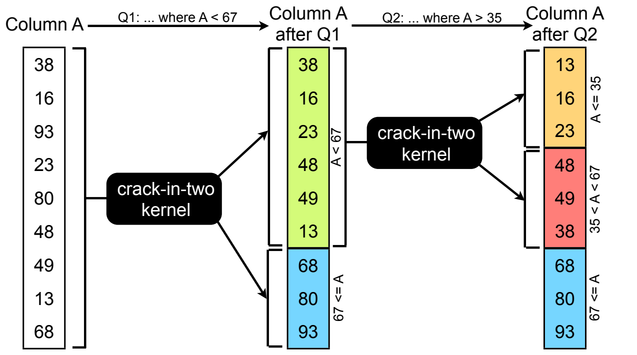


Figure 1: Column A is cracked by serving two range queries using a crack-in-two kernel.

1.1 Core Idea of Adaptive Indexing: Learn the Distribution of Search Predicates

The family of adaptive indexes is one of the many examples of query processing techniques that *learn* statistics on the data and/or queries, in order to speed-up query processing. The main idea of adaptive indexing is to learn the distribution of selection predicates in the queries and take that as a hint on how to index the data. Early proposals of adaptive indexing, like the original database cracking [14], gradually build a decision tree to learn the predicate distribution of queries for a particular column. Database cracking starts with an empty tree. Then, each incoming query is interpreted as an insert operation on that decision tree, i.e. a hint how to refine the decision tree, plus one partitioning step on the column, i.e. one input partition is broken into two subpartitions using a so-called *crack-in-two kernel.*

Figure 1 visualizes the concept of *standard cracking* [14], the most basic form of database cracking. Initially, the so called *cracker column A* has no partitions. When the first query **Q1** arrives, the cracker column is partitioned into two partitions according to the range predicate `A < 67` using a *crack-in-two kernel.* This introduces a partition boundary or *crack*[1]. To process the next range query **Q2**, we first identify the partition we have to refine to answer `A > 35`. We see that the predicate requires a repartitioning of the first partition, so we split it in two using the crack-in-two kernel again. The column is incrementally indexed by repeatedly serving range queries and query execution time improves as the adaptive index becomes more accurate.

In this paper, we focus on the partitioning part: how to split one partition into two subpartitions efficiently using a *cracking kernel.* These cracking kernels are the work horses in almost all adaptive indexing proposals (and even very mundane techniques like quicksort). Obviously, the choice of the cracking kernel has a huge impact on the overall running time of the cracking method

[1]The cracks are tracked in an auxiliary decision tree called the *cracker index.* That index also reflects the history of query predicates.

sitting on top, as the entire physical reorganization of the cracker column is performed by the kernel.

2 A BRIEF HISTORY OF ADAPTIVITY AND LEARNING IN INDEXING

Before we look at how the state-of-the-art cracking kernels work and perform, let us briefly discuss the history and current positioning of adaptive indexing, to understand the impact of this work.

A cracker index is a model of the distribution of query predicates. That model becomes more accurate over time. A similar observation is obviously also true for any other tree-index like B-trees where the inner nodes and/or each level of the tree represent a model of the key column distribution at various levels. The literature on drawing samples and modeling data distributions through histograms or *probability density functions* (pdf) is immense and the effects of exploiting data distributions for query processing (including indexing) have been known for long. Prominent examples include: buffer trees [7], bulkloading [6], approximate query processing [8], database compression [13], or even entropy encoding like Golomb-Rice Codes [10] which are constructed by first scanning the data to create a pdf, i.e. a model of the data distribution, and then using that pdf to construct the codes. A modern incarnation of this idea, focusing on deep learning rather than traditional statistical methods, is [18].

In a way, the initially proposed adaptive index [14], was a greedy approach to sampling (or learning) the predicate distribution: each predicate is directly translated into a refinement of the recursive partitioning process. This leads to a certain fragility w.r.t. the order of incoming queries which was targeted by follow-up approaches like stochastic cracking [11]. A comprehensive experimental evaluation of the different approaches can be found in [24, 25]. More recent work [22] investigates how to adapt the adaptivity (meta-adaptivity), i.e. the index learns how to improve its learning strategy (this is a form of meta-learning).

All these approaches have in common, that they perform simple data partitioning at the core. This data partitioning is executed by a cracking kernel. Thus, if we can improve the kernel, we can improve a variety of adaptive indexing methods. Let us start by analyzing the behavior of the state-of-the-art kernels.

3 REVISITING STATE-OF-THE-ART KERNELS

Branching Crack-in-Two. This kernel is used in the first work on database cracking [14] and is an implementation of Hoare's *partition* algorithm [12]. Figure 2 shows the hot loop of the kernel.

```
while (begin < end) {
    if (*begin < pivot) ++begin;
    else if (*end >= pivot) --end;
    else swap(*begin, *end);
}
```

Figure 2: Branching crack-in-two.

The algorithm maintains two cursors, called `begin` and `end`. The cursor `begin` marks the end of the partition of elements smaller than the pivot and the cursor `end` marks the beginning of the partition of elements greater or equal to the pivot; elements in between `begin` and `end` have yet to be partitioned. Initially, `begin` points to the first element of the input and `end` points to the last element of the input. In every iteration of the loop, the elements pointed to by the two cursors are compared to the pivot. If an element belongs inside the partition of its cursor, this cursor is advanced to the next element and the partition grows. If no cursor has been advanced in an iteration, both elements pointed to are swapped. Eventually, all elements are partitioned and both cursors point to the same element. The kernel terminates and the position of the crack is returned.

Predicated Crack-in-Two. Pirk et al. [21] analyze branching crack-in-two and observe that the conditional branches in the loop body are frequently mispredicted, causing pipeline flushes and impairing performance. The effect of branch misprediction maximizes as the selectivity of the query approaches 50%, where branches are equally likely. The authors propose to replace conditional branching by predicated execution. The idea is to speculatively append an element to both partitions, and only advance the cursor of the partition the element belongs to. The goal of their work is to improve crack-in-two from a CPU-bound to a memory-bound operation.

```
 1 /* code to initialize the buffer (omitted) */
 2 i = 0;
 3 while (begin <= end) {
 4     value = buffer[i].values[buffer[i].which];
 5     *begin = *end = value;
 6     advance_lower  = value <  pivot;
 7     advance_higher = value >= pivot;
 8     begin += advance_lower;
 9     end   -= advance_higher;
10     buffer[i].values[0] = *begin;
11     buffer[i].values[1] = *end;
12     buffer[i].which = advance_higher;
13     i = 1 - i;
14 }
```

Figure 3: Predicated crack-in-two [21].

Figure 3 shows an excerpt of their implementation. Let us first focus on how predicated execution is implemented. In lines 6 and 7, the element `value` is compared to the pivot, and the results are saved in two variables. In lines 8 and 9, these variables are used to update the cursors `begin` and `end` without using conditional branches. The algorithm speculatively stores one element at the positions pointed to by the two cursors. In line 5, the element `value` is written to positions `begin` and `end`. Since the element is appended to both partitions, exactly one of the two speculative stores is correct while the other store falsely overwrites an element. As we do not know in advance which of the two stores is correct, both elements under the cursors need to be backed up in a buffer beforehand. In lines 10 and 11 and sketched in line 1, the elements under the cursors are copied to the buffer before they are speculatively overwritten in the next iteration in line 5. In line 12, the flag `which` in the buffer is set to indicate which of the two backed up elements has been stored correctly. Note that the buffer has two "levels", addressed by the index `i`. These two levels are necessary as the backup is also speculative: although only one element is falsely overwritten, both elements need to be backed up as we do not know beforehand which store is incorrect. At the end of the loop body, the index `i` is switched.

Vectorized Crack-in-Two. In the lines 1, 10, and 11 of Figure 3, predicated crack-in-two backups the two elements to overwrite in two buffers, where each buffer can store a single element. To get more predictable code, in vectorized crack-in-two [21] they allow larger buffers, that backup entire chunks of data. In this sense, vectorized crack-in-two can be seen as a generalization of predicated crack-in-two. In their evaluation, vectorized crack-in-two is the best performing variant throughout all experiments.

As we can see, the three previously described kernels partition the input in two parts. We want to mention here that there exist works on *crack-in-k* kernels as well, which produce *k* partitions in one reorganization step. For instance, in the first work on database cracking [14], the authors propose a branching crack-in-three kernel that they use to answer two-sided range queries. However, it turned out that two consecutive calls to crack-in-two perform better than a single crack-in-three call [24]. Thus, we focus on two-sided kernels for the remainder of this work.

3.1 System Setup

We run our experiments on a compute server with two Intel® Xeon® E5-2620v4 CPUs at 2.10 GHz clock frequency and eight physical cores each. Intel® Hyper-Threading and Intel® Turbo Boost are disabled. The system has 32 GiB main memory, divided into two NUMA regions of 16 GiB each. The kernel of our OS is Linux 4.15. We use CPUSET [1] to isolate cores and exclusively run our benchmarks on them. To ensure that all data resides on the same NUMA node where the program is running, we bind our benchmark program to the first node with `numactl` [2]. The benchmarks are compiled with `clang 5.0` and the options `-O3 -march=native`. All data points reported are the median of five runs. We measure running time by reading the TSC register; other events such as branch misses and stalled cycles are counted (not sampled) with PAPI [3]. All experiments are performed on a 4 GiB data set of `(key, payload)` tuples; keys are unique and uniformly distributed over their respective integer domain.

3.2 Evaluation

With the description of the state-of-the-art kernels and the system setup at hand, let us now evaluate how well the kernels perform. We compare the throughput of the kernels with the single core memory bandwidth, as measured with the `bandwidth` benchmark [26]. This baseline is the theoretical maximum, and allows us to judge how far an algorithm is from being memory bound.

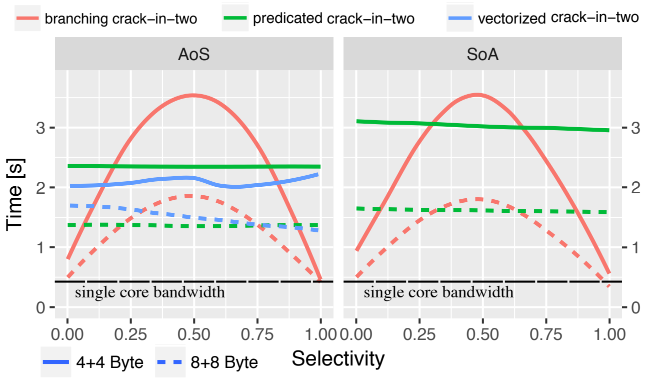


Figure 4: Relative comparison of branching and predicated crack-in-two to the single core memory bandwidth on array-of-struct (AoS) and struct-of-arrays (SoA) layout.

Figure 4 shows a comparison of the running times of branching crack-in-two, predicated crack-in-two, and vectorized crack-in-two in relation to the system's single core memory bandwidth. The x-axis shows the selectivity, i.e. how many of the elements are less than the pivot. The y-axis shows the running time of a single crack-in-two invocation in seconds. The elements subject to partitioning are 2-tuples of 4 or 8 Byte elements, abbreviated 4+4 and 8+8, respectively. We consider both *array-of-struct* (AoS) and *struct-of-arrays* (SoA) layouts. Pirk et al.'s vectorized crack-in-two was evaluated with a vector size of 1024 Byte using the source code provided on BitBucket [20]. The implementation supports only AoS layout, hence we are not able to provide an evaluation of vectorized crack-in-two for SoA layout. We make three major observations:

(1) Branching crack-in-two is up to 7 times slower and predicated crack-in-two is up to 6 times slower than the theoretical optimum. Branching crack-in-two takes the most time at 50% selectivity, and improves drastically as the selectivity approaches the extreme of 0% or 100%. Roughly below 20% and above 80% selectivity, branching crack-in-two becomes faster than predicated crack-in-two. The performance of predicated crack-in-two is independent of the selectivity and the throughput stays constant.

(2) Vectorized crack-in-two is faster than predicated crack-in-two for smaller elements. The smaller the elements, the more predicated crack-in-two suffers from backing them individually. As vectorized crack-in-two backs up entire chunks, the element size is less important for the running time.

(3) Smaller elements require more time than larger elements. Since there are twice as many 4+4 tuples as 8+8 tuples, twice as many instructions and comparison are executed.

Based on these observations, we conclude that the discussed kernels are CPU bound, spending a majority of their running time on evaluating comparisons, manipulating auxiliary data, and waiting for instructions to retire. The impact of the element size gives us an impression of how much overhead remains from being memory bound and that there is still room for improvement.

4 IMPROVING STATE-OF-THE-ART

In the previous section, we have seen that the state-of-the-art kernels do not fully utilize the capabilities of the system. To identify the nature of the problems, let us investigate branching crack-in-two and predicated crack-in-two in the following in detail.

In Figure 5, we compare branching crack-in-two and predicated crack-in-two by branch misses, stalled cycles, and running time. For branching crack-in-two, we can see that the amount of branch misses strongly correlates with the algorithm's running time. Further, we see that stalls only occur for extreme selectivities. The reason for that is that branch prediction succeeds very often and the kernel processes data faster than it can be served. Hence, the processor has to stall instructions until the data arrives in the fill buffers. For predicated crack-in-two, we observe no branch misses at all, which corresponds to the algorithm's use of predicated execution. However, we observe an unexpected high amount of stalled cycles. The same argument as for branching crack-in-two, that data cannot be served fast enough, does not apply here; the running time of predicated crack-in-two is far away from exceeding the memory's bandwidth limits.

A major performance bottleneck of the algorithm only becomes apparent when analyzing the compiled assembly code. The compiler is not able to promote the buffer to registers, although it only contains four elements and two flags. We credit this behavior to the complex address computation in line 4 of Figure 3. Communicating

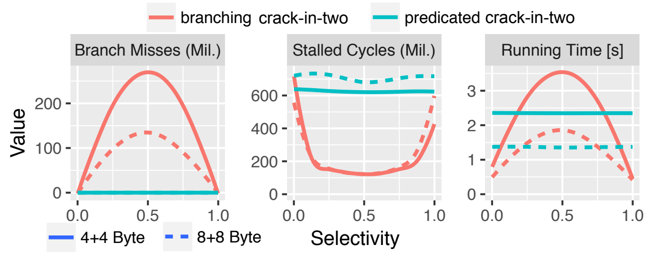


Figure 5: Low-level comparison of branching crack-in-two and predicated crack-in-two.

the backups through memory rather than registers has two disadvantages. First, additional loads and stores are issued. Although the buffer remains in the L1 data cache and loads/stores are served within few cycles, this significantly increases the overall latency per iteration. Second, using the `which` flag to select one of the two elements in the buffer through address computation prevents using the conditional move instruction `CMOVcc` of the x86 architecture.

```
 1 first = *begin;
 2 second = *end;
 3 while (begin <= end) {
 4     /* first level, i=0 */
 5     *begin = *end = first;
 6     left = begin[1];
 7     right = end[-1];
 8     advance_lower = first < pivot;
 9     begin += advance_lower;
10     end   += advance_lower - 1;
11     first = advance_lower? left:right;
12     /* second level, i=1 */
13     *begin = *end = second;
14     left = begin[1];
15     right = end[-1];
16     advance_lower = second < pivot;
17     begin += advance_lower;
18     end   += advance_lower - 1;
19     second = advance_lower? left : right;
20 }
```

Figure 6: Predicated++ crack-in-two.

4.1 Predicated++ Crack-in-Two

To promote the backups to registers, we must get rid of the complex address computation. First we eliminate the indirection introduced by having two levels, which are addressed by index `i`. To do so, we manually unroll the loop by factor 2. This allows us to specialize the two instances of the loop body for the two values of `i`, i.e. 0 and 1. The buffer is split in two, where either instance of the loop body is assigned one of the levels of the buffer. Explicitly addressing the level with index `i` is now superfluous. Loading the next element from the buffer still requires address computation involving the `which` flag. Let us examine Figure 3 again. Using the `which` flag to determine which element to load from the buffer in line 4 is only necessary because in lines 10 and 11 both elements are backed up unconditionally. Alternatively, we can only backup the element that will be selected by the `which` flag. Thereby, the `which` flag becomes redundant and need not be stored. Further, as only the selected element is stored, the buffer shrinks in size to store just two elements, one for each level.

Figure 6 shows our optimized predicated crack-in-two algorithm, called *predicated++ crack-in-two*. Lines 1 and 2 correspond to the initialization of the buffer; the elements under the cursors are backed up. Lines 4 to 11 show the loop body specialized for `i = 0`, lines 12 to 19 show the specialization for `i = 1`. In line 5, the selected element is speculatively appended to both partitions. In lines 6 and 7 the next elements are loaded unconditionally into temporaries called `left` and `right`. Lines 8 to 10 perform the comparison and advance the cursors. In line 11, the next element is selected and stored in the buffer `first`. Lines 12 to 19 are equivalent.

4.2 Rewired Crack-in-Two

Before we evaluate predicated++ crack-in-two, let us discuss another approach for kernel improvement. As we have seen before, the performance of in-place partitioning algorithms is limited by inherent data dependencies. In traditional crack-in-two kernels, in each iteration an element is loaded, compared to the pivot, stored, and one of the two pointers delimiting the partitions is advanced. In the next iteration, again an element is loaded. The CPU must delay execution of this load until the store operation of the previous iteration retires [9, 16]. To overcome this limitation, we must find a way to decouple successive iterations. When we look at out-of-place partitioning, we see that the structure of the main loop is roughly the same as for in-place partitioning: an element is loaded from the input, compared to the pivot, stored at the output location, and the pointer associated to that output location is updated. The subtle difference here is that loading an element in one iteration need not be delayed until the store of the previous iteration retires.

We conclude that, in order to break the aforementioned long-latency data dependency, an output location different from the input location in form of a buffer is required. This buffer must be "small" and of fixed size. But how can this algorithm be in-place if input data is written to a buffer? The answer is that we have to move the partitioned data inside the buffer back into the memory region of the input. A simple way to do so is copying the data inside the buffer back into the memory region of the input. For example, consider Pirk et al.'s vectorized crack-in-two, which uses small, cache-resident buffers and a tuned AVX2 copy routine. Despite the effort, it achieves only small improvement over predicated crack-in-two for 4+4 tuples and performs worse for 8+8 tuples. As we can see, copying back the partitioned data adds considerable overhead and nullifies the reason for using a buffer in the first place. So how to make use of the buffer and avoid copying back partitioned data?

The answer lies in the *rewiring of memory*, that grants us the required flexibility. Traditionally, the programmer works solely on virtual memory. This virtual memory is transparently mapped to physical memory by the operating system. To get a handle on the mapping, in [23] we reintroduce physical memory to user space in form of so-called main-memory files. Using the `mmap` system call, it is possible to freely map (or rewire) a virtual page to an offset in a main-memory file. As that file is internally backed by physical memory, we establish a transitive mapping from virtual to physical pages. This mapping can be freely manipulated at runtime, enabling interesting opportunities in algorithm design. In the following, we explain how rewiring is used to move the partitioned data inside the buffer back into the input's virtual memory region.

4.2.1 Core Idea and Example. We explain rewired crack-in-two by performing an example run step by step. The example is given in Figure 7 and pseudocode is provided in Appendix A. In the first row, there are three virtual memory regions, separated by a small gap.

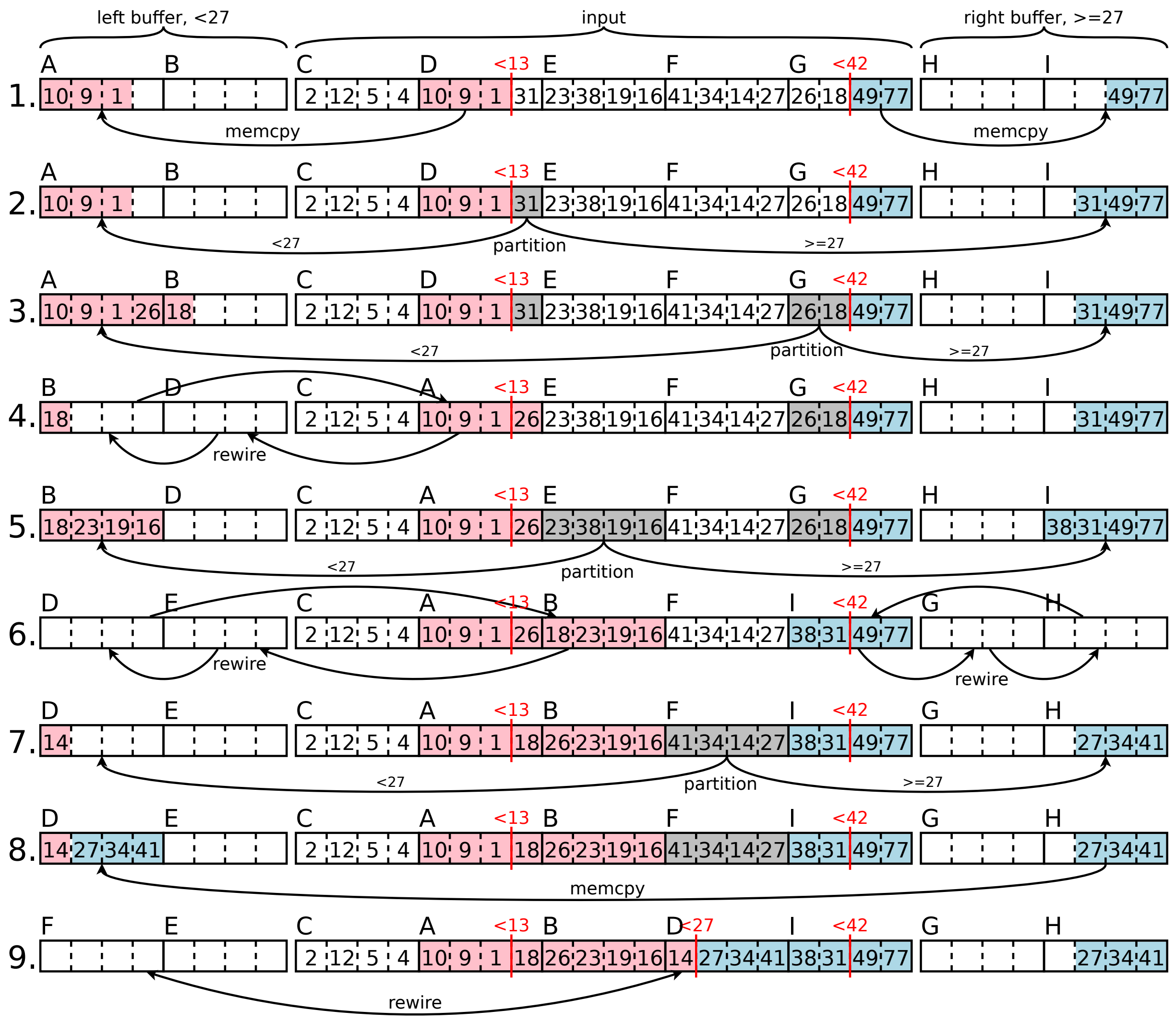


Figure 7: Rewired crack-in-two on an input of five pages. The partition to crack has the lower boundary 13 and upper boundary 42. The cracking predicate is 27. The physical pages are labeled A-I. The buffers consist of two pages each.

The left region is the buffer for elements less than the pivot, and is backed by the two physical pages A and B. The right region is the buffer for elements greater or equal to the pivot, and is backed by the two pages H and I. The region in between is the input, backed by pages C to G, which we want to partition with the pivot 27. Before our rewired crack-in-two is invoked, the cracking strategy finds the lower and upper bounds of the region that is cracked. In our scenario, the index already contains the cracks 13 and 42, which are the lower and upper bounds for our crack-in-two run. Elements highlighted in red evaluated less than the pivot, elements highlighted in blue evaluated greater equals to the pivot. Elements highlighted in gray have been partitioned.

1. We begin with backing up elements that lie outside the cracked region but inside pages that are part of the cracked region. The elements 10, 9, 1 in page D lie outside the cracked region, but page D is part of the cracked region, and hence these elements must be backed up. We know that all elements left of crack 13 are less than 13, and by transitivity less than our pivot 27. Therefore, we simply copy the three elements to the buffer of smaller elements. Similarly, for page G we copy the elements 49, 77 to the buffer of larger elements. Note that this buffer is filled from right to left.

2. After performing the backups, we must partition the rest of the pages D and G. We partition the remainder of page D, which is the element 31. The element is greater than our pivot, and is therefore appended to the right buffer.

3. The rest of page G is partitioned. Both elements 26, 18 are less than the pivot and are appended to the left buffer.

4. By now we partitioned exactly two pages into our two buffers of two pages each. We infer that at least one page of a buffer is filled, and at most two pages are filled in total. In step four we test the buffers to find out which buffer's "first" page is filled, i.e. we test pages A and I from the third row. We find that page A is filled and page I is not. Therefore, page A needs to be rewired back into the virtual memory region of the input. The arrows in the fourth row picture how the mapping from virtual to physical pages is permuted. For example, the virtual memory region that was backed by physical page D in the third row is now backed by physical page A. It is safe for us to rewire these pages, because all elements from the rewired input page D have been partitioned into the buffers; no element is lost. Page B is moved to the start of the buffer, and page D becomes the second page of that buffer. The elements in page D are discarded and page D becomes a fresh, empty buffer page. Page I need not be rewired, and we proceed to the next step.

5. As we rewired a page from the left buffer, we need to partition the next input page from the left, page E. Observe that it is guaranteed that both buffers have *at least* one whole page of free space left. This follows immediately from the previous step, where we rewired the first filled page of each buffer. Partitioning moves elements 23, 19, 16 to the left and element 38 to the right buffer.

6. After partitioning one whole page to the buffers, we can again infer that at least one page of a buffer is filled, and at most two pages are filled in total. We test both buffers and find that pages B and I in the fifth row are filled. Hence, both pages need to be rewired back into the virtual memory region of the input. The arrows in the sixth row depict the rewiring process.

7. The last page from the input, page F, is partitioned into the buffers and the algorithm finishes. The next two steps show how the elements remaining in the buffer are rewired back into the virtual memory region of the input.

8. Currently, in row seven, the remaining elements are dissected into two buffers. To merge them into one buffer we append the elements in the right buffer to the left buffer. It holds that this merge will always exactly fill the first page of the left buffer.

9. At last, we rewire page D into the virtual memory region of the input. The algorithm terminates by returning the position of the crack, which lies in the page that was rewired last.

A concerned reader might argue that rewiring introduces heavy fragmentation of the data, having a negative effect on sequential access performance. However, this concern is not justified: Virtual to physical address translation has to happen for every memory access, no matter whether accessed physical pages are consecutive.

4.2.2 *SIMD.*

Rewired crack-in-two spends most of its running time in partitioning the data to the buffers. We can exploit data parallelism to speed up this task. We use the *advanced vector extensions 2* (AVX2) [15] to partition 256 bits of keys and 256 bits of payloads in a data parallel fashion. Figure 8 shows our SIMD pipeline for 4 Byte elements, which is greatly inspired by the work of Menon et al. [19]. First, eight keys are loaded into a SIMD register. Then, the keys are compared to the pivot, yielding a 256 bit mask. Using `movemask` we convert this mask to an index, an 8 bit integer composed of the most significant bits of the masks $m_7, \ldots, m_0$. The number of 1-bits in this integer corresponds to the number of keys that are less than the pivot. We can extract this number efficiently with the `popcount` instruction. Further, we use the index to access a pre-computed, constant *lookup table* (LUT) of permutation vectors. Such a permutation vector is then used to permute the keys and payloads ($p_7, \ldots, p_0$) such that the elements inside the SIMD register are properly partitioned. We append the entire SIMD register to both lo and hi buffer. Yet, we advance the boundaries of the buffers only by `num_less` and `8−num_less`, respectively. To optimize the SIMD pipeline as much as possible, we made extensive use of the *Intel Architecture Code Analyzer* (IACA) [17]. Of our many different

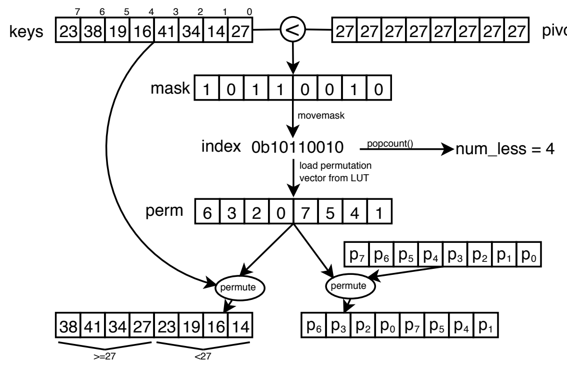


Figure 8: The SIMD pipeline of partitioning.

implementations, we found the one described above to achieve the highest throughput. For AoS layout, the innermost loop consists of only 29 μOps – which nicely fit into the μOp loop buffer – and takes on average 9.00 clock cycles per iteration for 4 Byte elements (30 μOps and 10.42 cycles for 8 Byte). For SoA layout, the innermost loop consists of 27 μOps and takes on average 7.90 clock cycles per iteration for 4 Byte elements (27 μOps and 8.84 cycles for 8 Byte).

4.2.3 *Multithreading.*

To make use of multi-core systems, which are omnipresent nowadays, we also want to parallelize cracking. We elaborate two approaches to do so: (1) Initially, split the data set into n equally sized chunks. For each incoming query, crack the chunks independently. Queries are then served by extracting the qualifying partitions of each chunk. (2) Alternatively, we can parallelize the crack-in-two phase. The partition to crack is split into n equally sized chunks, the chunks are cracked independently with *any* crack-in-two kernel, and then the cracked chunks are merged by a single thread to obtain exactly one crack. In the remainder of this section, we focus on the second approach and explain how rewiring can be used to significantly improve over state-of-the-art. In Section 5.3 we will evaluate both approaches.

The second approach was proposed by Pirk et al. [21] and termed *refined partition & merge*. The authors engineer a specialized version of predicated crack-in-two, that enables cheap merging. In Figure 9, we can see how refined partition & merge splits data into three chunks, where a chunk consists of two slices. The size of a slice is chosen depending on the expected selectivity of the predicate, such that the elements smaller than the predicate fit into the left slice of the chunk, and the elements greater or equal to the predicate fit into the right slice of the chunk. The slices are chosen such that if the prediction of the selectivity is correct, almost no data needs to be moved in the merge phase. It is generally difficult to predict the selectivity of a predicate precisely and hence we do not want to rely on this assumption.

We present *rewired partition & merge*, a merge algorithm that is completely independent of the selectivity of the predicate and the used crack-in-two algorithm. We explain our algorithm along the example in Figure 10 using three threads.

0. We divide the input data into three equally sized chunks, and crack each chunk independently and concurrently using any crack-in-two algorithm.

1. In the next step, we collect all pages entirely less than the pivot from right to left, giving $P_l = [I, F, E, A]$. Similarly, we collect all pages entirely greater or equal to the pivot from left to right, giving $P_g = [C, D, H, J, K, L]$. The cracks are collected from left to

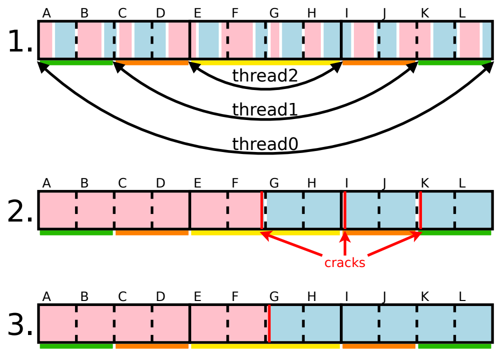


Figure 9: Refined partition & merge with a predicted selectivity of 50%. Thread 0 partitions pages A,B,K,L (green), thread 1 partitions pages C,D,I,J (orange), and thread 2 partitions pages E,F,G,H (yellow).

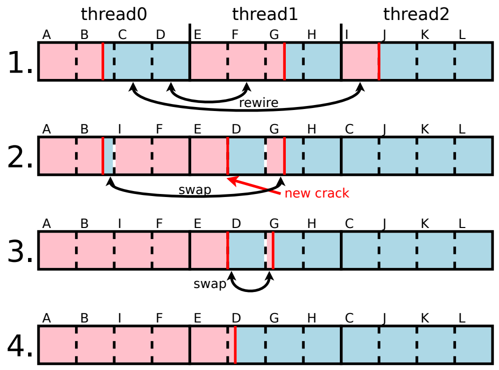


Figure 10: The merge phase of rewired partition & merge.

right and stored in a sorted set $S_{crack} = \{c_B, c_G, c_J\}$, sorted by their location in the input data.

After computing P_g, P_l, and S_{crack}, we compute the pages to rewire. Starting with index $i = 0$, we extract pages p_{l_i} from P_l and p_{g_i} from P_g. If p_{l_i} is right of p_{g_i}, we add the pair (p_{l_i}, p_{g_i}) to the list of pages to swap, and increment i. For $i = 0$, $p_{l_0} = I$ and $p_{g_0} = C$, and because I is right of C, we add (I, C) to the list of pages to swap. Similarly, for $i = 1$, we add (F, D) to the list. For pages E and H, however, E is not right of H, and we stop.

The list of pages to rewire is $[(I, C), (F, D)]$. We process this list from right to left, swapping the pages in a tuple via rewiring. First, pages F and D are rewired. Because of the particular order in which we rewire pages, we can immediately tell that rewiring pages F and D introduces a new crack between pages E and D (highlighted in row 2), and we add this crack to S_{crack}. Next, we rewire pages I and C. The crack c_J at the beginning of page J is thereby removed and hence erased from S_{crack}. After rewiring, we have $S_{crack} = \{c_B, c_D, c_G\}$.

2. Swapping pages in step 1 brought us close to the partitioned state. However, the pages containing the cracks S_{crack} still contain incorrectly placed data. To place data correctly, we swap the leftmost element greater or equal to the pivot with the rightmost element less than the pivot, similar to branching crack-in-two. However, whenever we reach a page boundary, we continue at the next crack. In step 2, we swap elements of page B with elements of page G, until we reach the end of page B and the crack in page B is removed.

3. We advance from page B to the next crack, which is at the beginning of page D. Then we swap elements between page D and page G until the crack in G is removed.

4. Eventually, all cracks but one have been removed. The last remaining crack partitions the data into elements less than the pivot and elements greater or equal to the pivot. Rewired partition & merge terminates and returns the position of that crack.

To summarize, rewired crack-in-two can be dissected into two phases: In the first phase, we create n equally sized partitions and crack them independently and concurrently using n threads. We call this phase *parallel partitioning*. The second phase, which starts after parallel partitioning terminates, is called the *merge phase*. In the merge phase, a single thread collects and rewires incorrectly placed pages and swaps remaining incorrectly placed elements. At the very end of the merge phase, a single crack remains in the data set. Rewired partition & merge terminates by returning the position of this crack.

In contrast to refined partition & merge, our rewired partition & merge does not rely on a specialized crack-in-two kernel or the correct prediction of the predicate selectivity.

5 EVALUATION

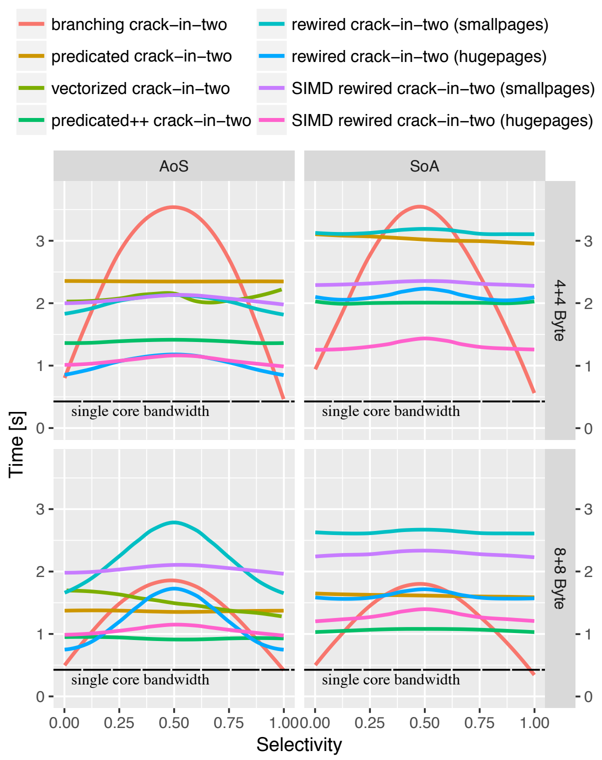
As we have introduced an improved version of predicated crack-in-two in form of predicated++ crack-in-two and our unconventional approach of rewired crack-in-two, let us now perform a comparison with branching crack-in-two, predicated crack-in-two, and vectorized crack-in-two to find out whether we are able to improve the weak spots of the baselines.

5.1 Running Time by Selectivity

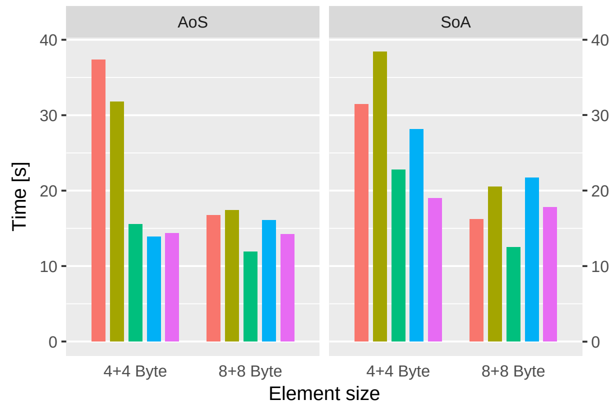
First, we evaluate the performance of the crack-in-two kernels for varying selectivity of the predicate. We run every algorithm on the same initial data set, where the pivot is chosen to achieve the desired selectivity. Figure 11a depicts our results. The black horizontal line shows the theoretical single core bandwidth.

We see that Pirk et al.'s predicated crack-in-two outperforms branching crack-in-two only for selectivities in the range of 20% – 80% for AoS and 40% – 60% for SoA layout. In the best case, predicated crack-in-two improves over branching crack-in-two by 40% less running time. Vectorized crack-in-two performs better on 4 Byte elements, but slightly worse on average for 8 Byte elements.

Still, the algorithms are 3-4 times slower than the theoretical optimum, and certainly not memory bound. Our predicated++ crack-in-two has at least 33% less running time than predicated respectively vectorized crack-in-two, and outperforms branching crack-in-two in the range of 10% – 90% with a best of 2.5x the throughput of branching crack-in-two. Our rewired crack-in-two on small pages always performs worse than predicated++ crack-in-two, even when using SIMD. Rewired crack-in-two on huge pages, however, shows significantly lower running times. The reason is soft page faults, that occur whenever a page is accessed for the first time after being rewired. These page faults happen 512 times more often for small pages. On AoS layout, the SIMD variants of our rewired crack-in-two algorithms perform equally to slightly better than their scalar counterparts. Rewired crack-in-two on huge pages improves over



(a) Crack-in-two running time under varying selectivity.



(b) Accumulated time of 1000 uniformly distributed cracking steps.

Figure 11: Comparison of crack-in-two kernels on both AoS and SoA layout for 4 and 8 Byte elements.

predicated++ crack-in-two by 30% less running time for 4 Byte elements on AoS layout. On SoA layout, the non-SIMD version of rewired crack-in-two on huge pages is slightly slower than predicated++ crack-in-two. However, the SoA layout makes SIMD data access particularly cheap, and the SIMD variant greatly improves over its scalar variant. For 4 Byte elements, the algorithm improves over predicated++ crack-in-two by 35% less running time. Overall, predicated++ crack-in-two and rewired crack-in-two are able to tighten the gap towards the memory's bandwidth limit. The improved variants outperform branching crack-in-two on selectivities of 5% – 95%, and improve the throughput by up to 3.5x.

5.2 End-to-End Running Times

In our next experiment, we compare the algorithms by performing 1000 uniformly distributed cracks on the key range and compare the end-to-end running time. The cracker index is implemented with a `std::map` [4]. Figure 11b shows our results. Analyzing the plot validates our findings from the first experiment. The fastest algorithms again have the shortest running times in this experiment. It is interesting to see that predicated crack-in-two performs just as good (AoS) or even worse (SoA) than branching crack-in-two. Although predicated crack-in-two improves over the worst case of branching crack-in-two, i.e. at a selectivity of 50%, a uniform distribution of selectivities frequently hits a point where branching crack-in-two is actually faster. Therefore, the accumulated running time of predicated crack-in-two is higher than branching crack-in-two. As faster algorithms outperform branching crack-in-two on a wider range of selectivities, less cracks fall outside this range, where branching crack-in-two would be faster. For instance, in the first experiment rewired crack-in-two (hugepages) outperforms branching crack-in-two by up to 3.5x but the difference in this experiment is just a factor of 2x. We suppose that rewired crack-in-two in particular has relatively high costs for corner case handling, i.e. proper initial partitioning of the boundary pages.

5.3 Multi-Threaded Execution

So far, we purely looked at single-threaded cracking on the entire column. However, following the first approach of Section 4.2.3, we can divide the column non-semantically into chunks and crack these chunks individually in parallel. Thus, in Figure 12, we show the scaling behavior of the kernels under multi-threaded execution in comparison with their single-threaded counterpart. We focus on AoS layout with 8 Byte elements and compare end-to-end times for 1000 queries. We vary the number of threads up to eight to fully utilize a single CPU.

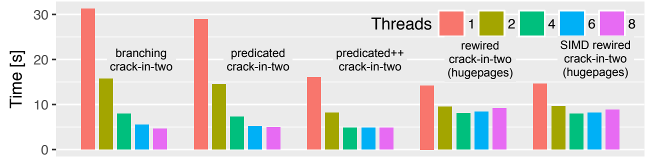


Figure 12: Scaling capabilities of the kernels when varying the threads from 1 to 8 under AoS layout and 4+4 Byte.

As we can see, the slower methods branching crack-in-two and predicated crack-in-two show the best scaling capabilities and scale essentially linearly with the number of threads. In contrast to that, the methods that were faster in single-threaded mode reach their highest performance with four threads already. The rewired methods perform the worst under multi-threading. The reason for this is the serialized handling of `mmap` by the operating system.

To evaluate the second approach of Section 4.2.3, we compare our rewired partition & merge with Pirk et al.'s refined partition & merge and and vary the number of threads to up to eight. We measure the time for the parallel partitioning phase and the merge phase separately and present them in a stacked area plot in Figure 13a. As baseline, we again show the single core bandwidth limit. The time at the bottom of each blue area shows the time taken by the partitioning phase. We can see that for any number of threads our algorithm performs faster than refined partition & merge. Our algorithm spends significantly less time in the partitioning phase, and the merge phase of our algorithm takes so little time it is not even visible in the plot.

A slight modification of rewired crack-in-two was necessary to make the algorithm scale well with the number of threads. When filling the buffers using regular store operations, the cachelines of the buffers are first streamed to the CPU before they are modified. This works fine as long as the bandwidth of the memory bus is not exhausted. However, loading the cachelines before modifying them doubles the amount of bytes read from main memory, and jams the memory bus with unnecessary I/O. Since increasing the number of threads puts more pressure on the memory bus, we want to reduce memory I/O to a minimum to maximize scalability. Since we know that the buffer is written only, we can replace the regular stores by non-temporal stores that bypass the caches and do not require loading the corresponding cacheline first. This cuts the reads from main memory in half and makes rewired partition & merge scale well with an increasing number of threads.

Since the merge phase of our algorithm is barely visible in Figure 13a, we provide another plot where we focus on the merge phase only. Figure 13b shows the time spent on merging the partitioned chunks. The number above each area shows the maximum time spent on merging, aggregated over all selectivities. The interesting shape of refined partition & merge stems from the fact that the algorithm cuts partitions into slices depending on the predicted selectivity of the pivot. If prediction is accurate, the proportions of the slices make merging very cheap. We did not assume that the selectivity of the pivot can be predicted, and simply fixed the predicted selectivity to 50%. Therefore, the algorithm takes the least time for a selectivity of 50%. The extremes of 0% and 100% selectivity are also very cheap because only very few elements are incorrectly placed in the beginning and hence only very few elements must be moved during the merge phase.

Our merge algorithm rewires pages that are entirely less or entirely greater or equal to the pivot and hence avoids moving elements whenever possible. Our algorithm only needs to move incorrectly placed elements of pages that contain a crack. Since every thread creates exactly one crack, we expect the merge phase to process the elements of one page per thread.[2] Subsequently, the amount of work spent in the merge phase depends mostly on the number of threads, and the time is almost constant over different selectivities. However, the time taken by the merge phase slightly increases with the number of threads. In a direct comparison, our merge algorithm improves over Pirk et al.'s merge algorithm by a factor of up to 58x. If the selectivity of a predicate can be estimated, we expect refined partition & merge to perform roughly as fast as our rewired partition & merge. However, if the estimate is just 5% off, this could already mean a slow-down by an order of magnitude. Therefore, we want to stress that the advantage of our rewired merge phase is not just the very fast merging itself, but also that the algorithm does not rely on precise selectivity estimates.

[2]The merge phase may introduce new or remove existing cracks. However, for n threads the expected number of cracks and hence pages to process is n.

6 CONCLUSION

In this experimental study, we revisited the state-of-the-art database cracking kernels and identified their major weak spots. We exposed branch misprediction and stalling as obstacles making the kernels CPU bound. Based on these insights, we introduced advanced kernels in form of predicated++ crack-in-two and (SIMD) rewired crack-in-two, which reduce the aforementioned problems and utilize the memory bandwidth of the system more efficiently. Our evaluation shows that the choice of the kernel can make a difference of up to factor 2x. Finally, based on the parameters that we vary in our experimental evaluation, we conclude our work with the decision tree shown in Figure 14, that provides recommendations which kernel to use in which situation.

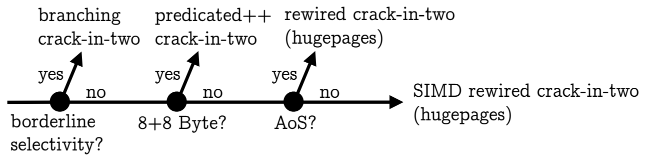

branching crack-in-two
predicated++ crack-in-two
rewired crack-in-two (hugepages)
yes
no
yes
no
yes
no
SIMD rewired crack-in-two (hugepages)
borderline selectivity?
8+8 Byte?
AoS?

Figure 14: Kernel decision tree. A selectivity is classified as borderline, if it is below 5% or above 95%.

REFERENCES

[1] 2017. cpuset(7) - Linux Programmer's Manual. (2017). Retrieved 2017-07-03 from http://man7.org/linux/man-pages/man7/cpuset.7.html

[2] 2017. numactl(8) - Linux Programmer's Manual. (2017). Retrieved 2017-07-03 from http://man7.org/linux/man-pages/man8/numactl.8.html

[3] 2017. Performance Application Programming Interface. (2017). Retrieved 2017-07-07 from http://icl.utk.edu/papi/

[4] 2017. std::map. (2017). Retrieved 2017-07-17 from http://en.cppreference.com/w/cpp/container/map

[5] Victor Alvarez, Felix Martin Schuhknecht, Jens Dittrich, and Stefan Richter. 2014. Main memory adaptive indexing for multi-core systems. In *DaMoN 2014*. 3:1–3:10.

[6] Peter M. G. Apers, Paolo Atzeni, Stefano Ceri, Stefano Paraboschi, Kotagiri Ramamohanarao, and Richard T. Snodgrass (Eds.). 2001. *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy.* Morgan Kaufmann.

[7] Lars Arge. 2003. The Buffer Tree: A Technique for Designing Batched External Data Structures. *Algorithmica* 37, 1 (01 Sep 2003), 1–24.

[8] Gautam Das. 2009. Sampling Methods in Approximate Query Answering Systems. In *Encyclopedia of Data Warehousing and Mining, Second Edition (4 Volumes)*, John Wang (Ed.). IGI Global, 1702–1707.

[9] Agner Fog. 2017. *An optimization guide for assembly programmers and compiler makers.* Technical University of Denmark. http://agner.org/optimize/microarchitecture.pdf

[10] Solomon W. Golomb. 1966. Run-length encodings (Corresp.). *IEEE Trans. Information Theory* 12, 3 (1966), 399–401.

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

[12] C. A. R. Hoare. 1962. Quicksort. *Comput. J.* 5, 1 (1962), 10–15.

[13] Allison L. Holloway, Vijayshankar Raman, Garret Swart, and David J. DeWitt. 2007. How to barter bits for chronons: compression and bandwidth trade offs for database scans. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007.* 389–400.

[14] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. 2007. Database Cracking. In *CIDR 2007*. 68–78.

[15] Intel 2012. *Intel Architecture Instruction Set Extensions Programming Reference.* Intel.

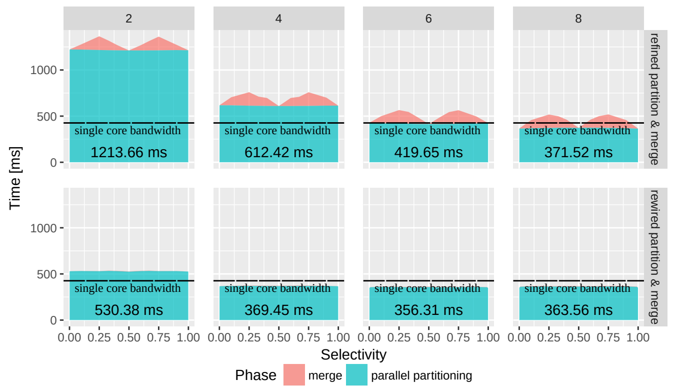
[16] Intel 2016. *Intel 64 and IA-32 Architectures Optimization Reference Manual.* Intel.

[17] Intel. 2017. Intel Architecture Code Analyzer. (2017). Retrieved 2018-01-30 from https://software.intel.com/en-us/articles/intel-architecture-code-analyzer/

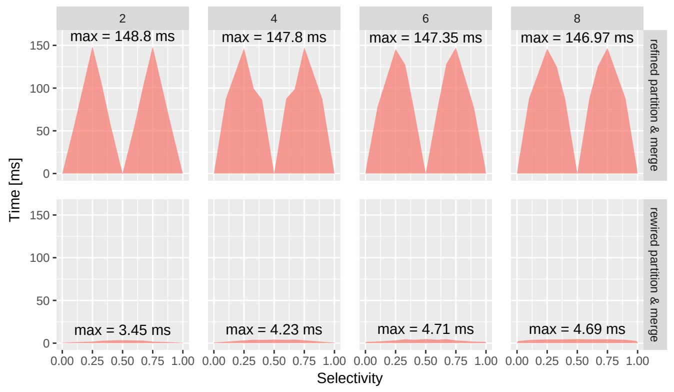
[18] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2017. The Case for Learned Index Structures. *CoRR* abs/1712.01208 (2017). arXiv:1712.01208 http://arxiv.org/abs/1712.01208

[19] Prashanth Menon, Todd C. Mowry, and Andrew Pavlo. 2017. Relaxed Operator Fusion for In-Memory Databases: Making Compilation, Vectorization, and Prefetching Work Together At Last. *PVLDB* 10, 11 (2017).

[20] Holger Pirk. 2017. Vectorized crack-in-two source code. (2017). Retrieved 2017-07-19 from https://bitbucket.org/holger/crackingscanvssort



(a) Comparison of one crack-in-two step.



(b) Comparison of the different merge algorithms for 2-8 threads.

Figure 13: Comparison of refined partition & merge and rewired partition & merge for 2-8 threads. The x-axis shows the selectivity of the predicate, the y-axis shows the time taken in milliseconds. The two colors show the time spent on the two phases *merge* and *parallel partitioning*. The time shown in the blue area is the time taken by the parallel partitioning phase.

[21] Holger Pirk, Eleni Petraki, Stratos Idreos, Stefan Manegold, and Martin L. Kersten. 2014. Database cracking: fancy scan, not poor man's sort!. In *DaMoN 2014*. 4:1–4:8.
[22] Felix Martin Schuhknecht, Jens Dittrich, and Laurent Linden. 2018. Adaptive Adaptive Indexing. *ICDE* (2018).
[23] Felix Martin Schuhknecht, Jens Dittrich, and Ankur Sharma. 2016. RUMA has it: Rewired User-space Memory Access is Possible! *PVLDB* 9, 10 (2016), 768–779.
[24] Felix Martin Schuhknecht, Alekh Jindal, and Jens Dittrich. 2013. The Uncracked Pieces in Database Cracking. *PVLDB* 7, 2 (2013), 97–108.
[25] Felix Martin Schuhknecht, Alekh Jindal, and Jens Dittrich. 2016. An experimental evaluation and analysis of database cracking. *The VLDB Journal* 25, 1 (2016), 27–52.
[26] Zack Smith. 2017. Bandwidth: a memory bandwidth benchmark. (2017). Retrieved 2018-01-29 from http://zsmith.co/bandwidth.html

A PSEUDOCODE OF REWIRED CRACK-IN-TWO

```
Data: lo buffer $buf_{lo}$, hi buffer $buf_{hi}$
Input  : left boundary lb, right boundary rb, the pivot
Output: position of the crack for the given pivot
// Initialize variables.
$page_{left} \leftarrow$ page of lb
$page_{right} \leftarrow$ page of rb
$page_{lo} \leftarrow page_{left}$
$page_{hi} \leftarrow page_{right}$
// Handle the two boundary pages.
if lb not a page boundary then
    copy elements left of lb in $page_{left}$ to $buf_{lo}$
if rb not a page boundary then
    copy elements right of rb in $page_{right}$ to $buf_{hi}$
if lb not a page boundary then
    partition elements right of lb in $page_{left}$ to buffers
    $page_{left} \leftarrow page_{left} + 1$
if rb not a page boundary then
    partition elements left of rb in $page_{right}$ to buffers
    $page_{right} \leftarrow page_{right} - 1$
// At least one buffer is half full; rewire.
fromLeft $\leftarrow buf_{lo}$ is half full?
isInit $\leftarrow buf_{lo}$ is half full? and $buf_{hi}$ is half full?
if $buf_{lo}$ is half full then
    rewire $buf_{lo}$ and $page_{lo}$
    $page_{lo} \leftarrow page_{lo} + 1$
if $buf_{hi}$ is half full then
    rewire $buf_{hi}$ and $page_{hi}$
    $page_{hi} \leftarrow page_{hi} - 1$
/* Partition one page at a time.                                  */
while $page_{left} \leq page_{right}$ do
    // Partition the next page.
    cur_page $\leftarrow$ fromLeft ? $page_{left}$ : $page_{right}$
    partition all elements in cur_page to buffers
    if fromLeft then $page_{left} \leftarrow page_{left} + 1$
    else $page_{right} \leftarrow page_{right} - 1$
    // Determine which page to partition next.
    $isFilled_{lo} \leftarrow buf_{lo}$ is half full?
    $isFilled_{hi} \leftarrow buf_{hi}$ is half full?
    if isInit and not $isFilled_{lo}$ and not $isFilled_{hi}$ then
        isInit $\leftarrow$ FALSE
        fromLeft $\leftarrow$ not fromLeft
    else if $isFilled_{lo}$ and $isFilled_{hi}$ then
        isInit $\leftarrow$ TRUE
        fromLeft $\leftarrow$ TRUE
    else if $isFilled_{lo}$ then
        fromLeft $\leftarrow$ TRUE
    else if $isFilled_{hi}$ then
        fromLeft $\leftarrow$ FALSE
    // Rewire half-full buffers.
    if $isFilled_{lo}$ then
        rewire $buf_{lo}$ and $page_{lo}$
        $page_{lo} \leftarrow page_{lo} + 1$
    if $isFilled_{hi}$ then
        rewire $buf_{hi}$ and $page_{hi}$
        $page_{hi} \leftarrow page_{hi} - 1$
end
// Compute position of the crack.
cur_page $\leftarrow$ fromLeft ? $page_{lo}$ : $page_{hi}$
offset $\leftarrow$ size($buf_{lo}$)
// If necessary, merge buffers and rewire.
if offset $\neq 0$ then
    append elements of $buf_{hi}$ to $buf_{lo}$
    rewire cur_page and $buf_{lo}$
return cur_page + offset                    // Return crack position.
```

Algorithm A.1: Pseudo-code for rewired crack-in-two.