Database Cracking: Fancy Scan, not Poor Man's Sort!

Holger Pirk
CWI, Amsterdam
holger@cwi.nl

Eleni Petraki
CWI, Amsterdam
petraki@cwi.nl

Stratos Idreos
Harvard University
stratos@seas.harvard.edu

Stefan Manegold
CWI, Amsterdam
manegold@cwi.nl

Martin Kersten
CWI, Amsterdam
mk@cwi.nl

ABSTRACT

Database Cracking is an appealing approach to adaptive indexing: on every range-selection query, the data is partitioned using the supplied predicates as pivots. The core of database cracking is, thus, pivoted partitioning. While pivoted partitioning, like scanning, requires a single pass through the data it tends to have much higher costs due to lower CPU efficiency. In this paper, we conduct an in-depth study of the reasons for the low CPU efficiency of pivoted partitioning. Based on the findings, we develop an optimized version with significantly higher (single-threaded) CPU efficiency. We also develop a number of multi-threaded implementations that are effectively bound by memory bandwidth. Combining all of these optimizations we achieve an implementation that has costs close to or better than an ordinary scan on a variety of systems ranging from low-end (cheaper than $300) desktop machines to high-end (above $60,000) servers.

1. INTRODUCTION

One of the litanies about data management systems is that they are I/O bound, i.e., limited in performance by the bandwidth to the primary storage medium (be it disk or RAM). Indeed, many operations like scans or aggregations are relatively easy to implement at sufficiently high CPU-efficiency to make I/O bandwidth the dominating cost factor. However, other operations like joins or index-building are mostly bound by the computation speed of the CPU. When exploring alternative algorithms for data management operations, it is crucial to understand the contributing cost factors for the existing as well as the new implementation.

Database Cracking was introduced as an alternative to scanning to evaluate range-predicates on relational data. Rather than copying the matching tuples into a result buffer, *Cracking* physically partitions the data in-place using the specified range as pivot(s). Since one of the resulting partitions contains only the qualifying tuples, *Cracking* effectively answers the query. Additionally, the reordered data can be combined with an appropriate secondary data structure (usually a tree or a hash) to form a partial clustered index. Assuming that the next query can benefit from such a clustered index, the extra costs for the physical reordering will pay off over time.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
DaMoN '14, June 22 - 27 2014, Snowbird, UT, USA
Copyright 2014 ACM 978-1-4503-2971-2/14/06 ...$15.00.

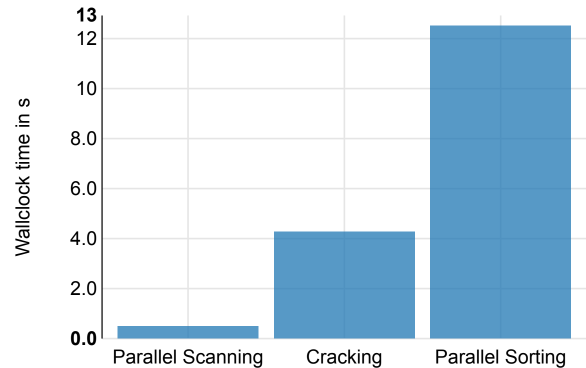


Figure 1: Costs of Database Operations

Since the fix-point of *Cracking* is fully sorted data, its costs are usually compared to those of fully sorting the data. With recent advancements in data (parallel) sorting algorithms [7], however, *Cracking* appears increasingly unattractive. To illustrate this, Figure 1 shows a quick comparison of the respective operations on 512 Million 32-bit integer values on a 4-Core Sandy Bridge CPU. It shows that while an off-the-shelf *(Parallel) Mergesort* implementation[1] is about 30 times more expensive than a (quasi I/O bound) *(Parallel) Scan*, it is only three times as expensive as MonetDB's implementation of *Cracking* [19]. Even though both *Scanning* and *Cracking*, (sequentially) read and write the same amount of data, they have vastly different costs. The performance difference must, thus, be due to their computational costs: *Cracking* is, unlike *Scanning*, not I/O bound. However,

> *we believe* that, if implemented with the underlying hardware in mind, *Cracking* can be (roughly) I/O bound.

To validate this hypothesis, we make the following contributions:

- We conduct an in-depth study of the contributing performance factors of the "classic" *Cracking* implementation.
- Based on the findings, we develop a number of optimizations based on "standard" techniques like predication, vectorization and manually implemented data parallelism using SIMD instructions.
- We develop two different parallel algorithms that exploit thread level parallelism to make use of multiple CPU cores.
- We rigorously evaluate all developed algorithms on a number of different systems ranging from low-end desktop machines to high-end servers.

[1] Part of the GNU libstdc++ Version 4.8.2

The rest of the paper is structured as follows: In Section 2, we provide an overview of related work as well as necessary background knowledge on the optimization techniques we applied. In Section 3 we present our analysis of the *Cracking* implementation in MonetDB discussing its problems with regard to CPU efficiency. We present our CPU-optimized sequential *Cracking* algorithms in Section 4, and our parallel implementations in Section 5. We evaluate these algorithms on a range of different hardware platforms in Section 6 and conclude in Section 7.

2. BACKGROUND & RELATED WORK

Before discussing the efficient implementation of *Database Cracking*, let us briefly establish the background knowledge regarding a) some architectural traits of modern CPUs that are relevant with respect to implementation efficiency, and b) partial and adaptive indexing techniques that are related to our approach.

CPU Efficiency Techniques

Advances in processor architectures and semiconductors have improved the performance of computer systems steadily over the years. However, the stagnation of clock frequency prompted the necessity for parallelization. Thus, modern CPUs provide several forms of parallelism, such as instruction level parallelism, data level parallelism and thread level parallelism.

Processors achieve *Instruction Level Parallelism (ILP)* by overlapping the execution of multiple instructions in a single clock cycle [23]. Independent instructions are executed in parallel if there are sufficient resources for all of them. ILP can be exploited by using multiple execution units to execute multiple instructions simultaneously (superscalar execution), or by executing instructions in any order that does not violate data dependencies (out-of-order execution) or even predicting the execution of instructions (speculative execution) [16]. Thus, care has to be taken to ensure that there are sufficiently many independent instructions [3, 5].

Performance improvement can also be achieved by exploiting *Data Level Parallelism (DLP)*. In its extreme, vector processors operate on the input arrays using one instruction per vector operation. In practice, most modern CPUs provide *Single Instruction Multiple Data* (SIMD) instructions that operate on a limited number of values (vector lengths ranging from 128 to 512 bit).

Thus, fewer instructions are fetched and executed. However, vector instructions usually have longer latencies and lower throughput than their scalar counterparts. They also rely on their inputs being stored in a contiguous (often even SIMD-word-aligned) memory region. In the most modern instruction sets (AVX2 and AVX-512), there is support for gather (AVX2 & AVX-512) and scatter (only AVX-512) instructions that fetch data from, respectively save data to, multiple, non-contiguous memory locations. Recent papers study the implementation of various database operations, e.g., scans, aggregations, index operations and joins, using SIMD instructions [30], while [4, 24] provide a thorough analysis of hash join and sort-merge join using SIMD. These operations significantly benefit from the SIMD technology by exploiting DLP and by eliminating branch mispredictions.

Thread Level Parallelism (TLP) allows multiple threads to work simultaneously. This allows an application to take advantage of TLP by splitting into independent parts that run in parallel. The advantage of multithreading is even more significant in systems that are equipped with multiple CPUs or multicore CPUs (chip multiprocessors). In addition, many chip multiprocessors incorporate the hyper threading technology which increases parallelism by allowing each physical core to appear as two logical cores in the operating system. Heavy load components such as instruction pipelines, registers or the execution units are usually replicated while others, such as caches, are shared among the logical cores. Basic database operations have been reexamined exploiting TLP, e.g., aggregations [8] and join algorithms [4, 24].

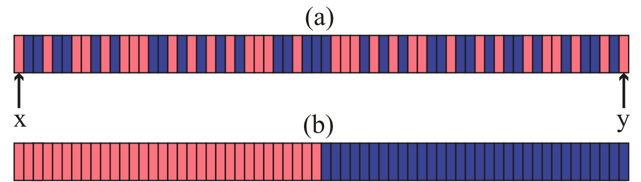


Figure 2: Original Cracking (single-threaded)

Indexing Techniques

In the majority of automated index tuning approaches, index tuning is clearly distinct from query processing. Offline indexing approaches [2, 9, 31] analyze a given workload and select/create the necessary indexes before the workload enters the system, whereas online indexing approaches [6, 25, 28] continuously monitor the workload and periodically reevaluate the index selection. In both cases, indexes equally cover all data items, even if some of them are not heavily queried. Thus, both index tuning and index creation may negatively affect the workload performance if there is not enough idle time to build the indexes or/and if the workload arbitrarily changes.

Adaptive indexing is a recent, lightweight approach to self-tuning databases: data reorganization is integrated with query processing. Adaptive indexing has been studied in the context of main-memory column-stores [19], Hadoop [26] as well as for improving more traditional disk-based settings [14]. It has been shown to work for many core database architecture issues such as updates [20], multi-attribute queries [21], concurrency control [10, 11], partition-merge-like logic [14, 22]. In addition, [12] shows how to benchmark adaptive indexing techniques, while stochastic database cracking [15] shows how to be robust on various query workloads. Finally, [13] shows how adaptive indexing can apply to key columns.

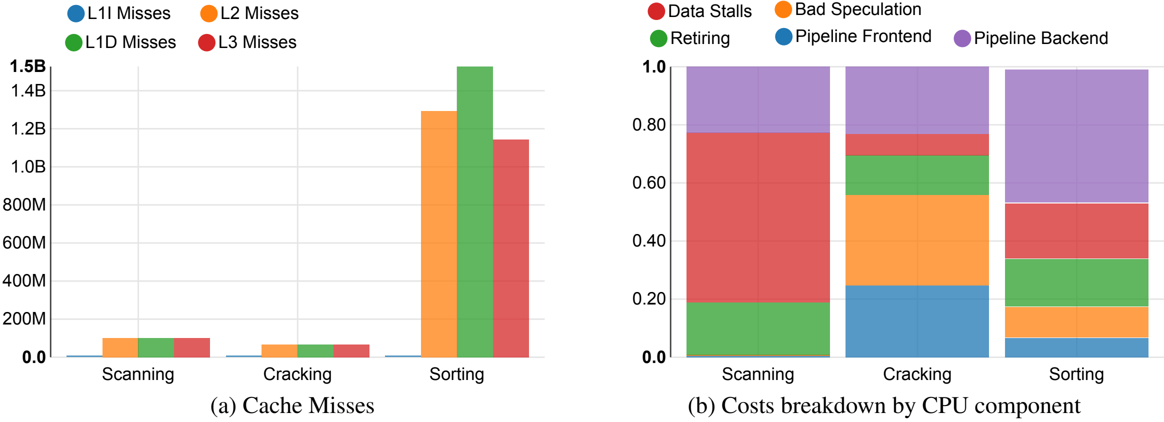
Database Cracking [19, 18] is the initial implementation of the adaptive indexing concept. Database Cracking initializes a partial index for an attribute the first time it is queried. Future queries on the same attribute further refine the index by partitioning the data using the supplied query predicates as pivots (similar to quicksort [17]) and updating the secondary dictionary structure. Since the reorganization of the index is part of the select operator, *Database Cracking* can be seen as an alternative implementation of scanning. While dictionary maintenance becomes the dominant cost factor as the average partition size decreases [29], the pivoted partitioning is the most important factor in the beginning. In this paper we focus purely on this step of the process, disregarding dictionary maintenance or order propagation to other columns. We believe that this makes our work orthogonal, i.e., easy to combine with other work in the field of *Database Cracking*.

3. CLASSIC CRACKING

Database Cracking is a pleasantly simple approach to adaptive indexing. However, it is not trivial to implement efficiently. In this section, we recapitulate the original *Cracking* algorithm and we examine the problems with the current implementation regarding CPU efficiency.

The Algorithm

The original, single-threaded *Cracking* algorithm is illustrated in Figure 2. Figure 2(a) depicts an uncracked piece. Red indicates



(a) Cache Misses

(b) Costs breakdown by CPU component

Figure 3: Cost Breakdown of Database Operations

values that are lower than the pivot, while blue indicates values that are greater than the pivot. Two cursors, x and y, point at the first and at the last position of the piece respectively. The cursors move towards each other, scanning the column, skipping values that are in the correct position while swapping wrongly located values. The result of this process is the cracked piece shown in Figure 2(b). Values that are less/greater than the pivot finally lie in a contiguous space. To crack a piece that consists of n values, the two cursors read all n values while moving towards each other resulting in $O(n)$ complexity in terms of computation as well as memory access. Thus, *Cracking* and *Scanning* are in the same complexity class but have significantly different costs (recall Figure 1).

Analysis

The classic way of analyzing in-memory data management system performance is to count the number of cache misses at different levels. This stems from the assumption that data management performance is dominated by data access costs. However, as displayed in Figure 3a, the number of cache misses do not provide an explanation for the performance difference of *Cracking* and scanning. In fact, scanning induces more cache misses because it produces the result set out of place. This indicates that merely looking at the number of cache misses is not sufficient - we have to determine the costs induced in other components of the CPU.

To do so, we conducted a systematic analysis of the costs component according to the Intel optimization manual for our (Ivy Bridge) CPU [1]. The breakdown in Figure 3b shows that *Cracking* merely spends 7% of the cycles stalling because of data access latencies. This explains why the number of cache misses alone is a poor predictor for the overall performance. The other cost factors, however give a much better explanation of the performance difference between *Cracking* and scanning [2]: The breakdown shows that 14% of the cycles [3] are spent retiring (useful) instructions at the end of the execution pipeline. Assuming that all instructions are necessary, this indicates that *Cracking* spends almost 10 times as much CPU cycles as scanning doing actual work. It also gives us an upper bound on the performance that can be achieved using a single CPU core: 14% of the current runtime, i.e., a speedup factor of about 7. Most importantly, however, this breakdown indicates where there is most potential for performance improvement: in eliminating branch mispredictions which 1. cause a significant amount of wasted cycles due to *bad speculation* and 2. prevent instructions from entering the *pipeline* at the *frontend*.

[2]In this normalized plot, equal height bars indicate an absolute difference of almost factor 10, Figure 1 providing the scale

[3]or, more accurately microop execution slots

4. CPU EFFICIENT CRACKING

Based on the outcome of our analysis in the previous section, we can direct our efforts to the performance painpoints of the original *Cracking* implementation, starting with branch mispredictions.

Predication

A common technique to address costs for branch mispredictions is "predication". The idea is to unconditionally write output but only advance one of the output cursors by the value of the evaluation of the predicate. This decouples the writing operation from the predicate evaluation and, therefore, effectively eliminates the conditional branch instructions at the costs of more write instructions. Since these write instructions generally only operate in L1 cache, the performance benefit for, e.g., selections, can be significant [27].

Unfortunately, not all algorithms are equally amenable to optimization through predication: implementations of out-of-place algorithms like selections can speculatively write to the output buffer as long as they write to empty slots. In-place algorithms, however, have to ensure that they do not overwrite any of the data values. They, therefore have to create backup copies of values that are speculatively overwritten. Naturally, deciding which value to backup has to be branch-free as well.

To achieve this, we developed a branch-free cracking implementation based on predication (illustrated in Figure 4). The fundamental idea is to create a backup copy of the value that is speculatively overwritten in a "backup" slot (we term the slot containing the value that is currently processed "active"). Based on this idea, each iteration goes through multiple phases with all (significant) operations within a phase being independent. At the beginning of each iteration, the to-be-cracked array is in a "consistent" state (see Figure 4a), i.e., each input value is stored exactly once in the array[4] and the "active" and "backup" slots contain the values at both cursors. In the *Compare & Write Phase* (see Figure 4b), the "active" value is written to both cursors and (independently) compared to the pivot. The result of the comparison (cmp) is used in the next phase (see Figure 4c) to advance the output cursors. One cursor is advanced by the value of cmp, the other by $1 - cmp$. This ensures

[4]Note that this does not imply that there cannot be duplicate values in the input

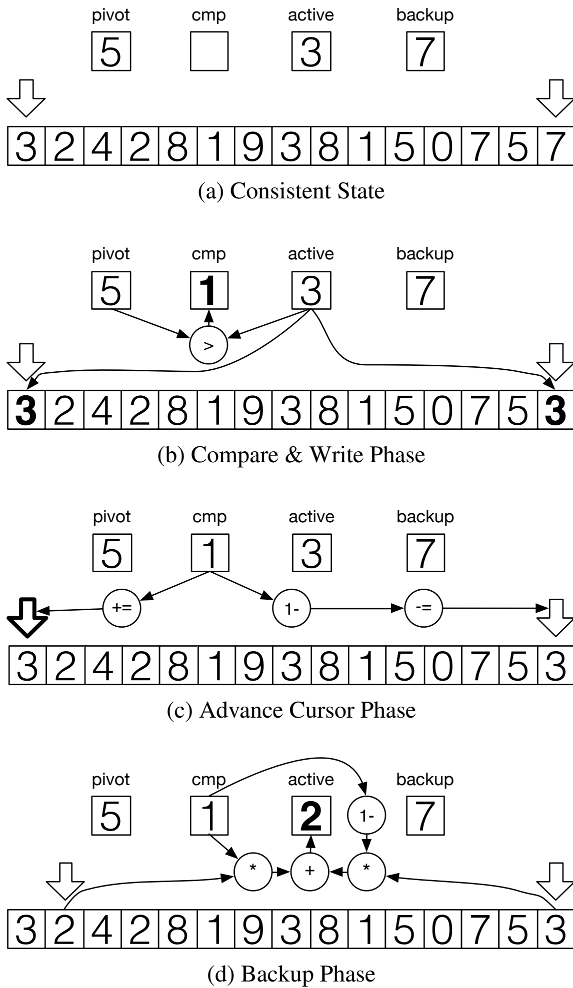


Figure 4: Predicated Cracking

that only one of the cursors is advanced. In the last phase (see Figure 4), the value at the advanced cursor is backed up. To ensure a branch-free implementation, we, again, use arithmetic calculations rather than branching to select the right value to store. At the end of each iteration, the *backup* and *active* slots switch roles (not shown in figure).

We implemented this idea in two variants that vary in the way they create the necessary backup copies of input values. The first implementation creates the backup copies to a small (cache-resident) buffer. This implementation has a slight disadvantage: the compiler can either use multiple registers to hold the two slots of the local buffer or flush the registers to L1-cache after each phase. To alleviate this problem, we developed a variant that uses one 64-bit register to hold the "active" as well as the "backup" value. This yields a slight performance benefit (see Section 6).

Vectorization

The main problem with the predicated implementation is the effort spent on backing up data (indicated by the *Pipeline Backend* bar which includes costs for writing data in Figure 5). The main tuning parameter for this operation is the granularity at which data is copied. Naturally, copying larger chunks results in more predictable code (at compile-time as well as run-time). The extreme case for this optimization would be copying the entire input-array,

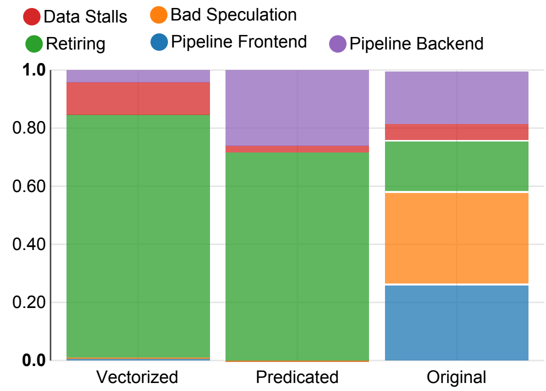


Figure 5: Cost breakdown of single-threaded implementations

making it an out-of-place implementation. This is not only memory intensive but also cache-inefficient since it requires two scans of the data. The natural solution to this problem is vectorization: small, cache-resident chunks of the input data are copied and, subsequently, partitioned out-of-place (see Figure 6). This has the advantage of producing tight, CPU-efficient loops in the (expensive) partitioning phase while allowing bulk-backups of input values.

However, the lack of control in the partitioning phase slightly complicates things in the backup: we have to deal with overflowing output buffers. It is, therefore, not enough to back up one vector per side since a half-full buffer may overflow into the adjacent one. This requires additional backup slots to ensure that the distance between each read-cursor and the trailing write-cursor is greater than the size of a vector. As visualized in Figure 6, three backup slots are sufficient to maintain enough "slack space" for safe writing.

SIMD

Figure 5 indicates that more than 80% of the cycles of the cracking implementation are now spent retiring (useful) instructions. This indicates that, to further improve single-threaded performance, we have to perform more work per instruction. This can be achieved by the use of SIMD instructions. The AVX-2 instruction set of current Intel CPUs provides instructions to gather values from multiple addresses into an SIMD word in a single instruction. The opposite, i.e. scatter instructions, are, however, only available in AVX-512 which is, currently, only supported by the Intel Xeon Phi extension cards. We, therefore, implemented *Cracking* using AVX-2 instructions to gather the input values. The main idea is to have one cursor per SIMD lane, gathering values that satisfy the partitioning predicate until the word is filled and can be flushed. We implemented all necessary operations (comparison, cursor advancing, ...) using 256-bit SIMD instructions and predication.

During evaluation (see Section 6), we found that this algorithm generally performs worse than the previously discussed implementations. We include the description primarily for completeness sake.

5. PARALLELIZATION

In this section we present two *Cracking* algorithms that exploit thread-level parallelism, i.e., first a simple partition & merge parallel algorithm, and then a refined variant of the simple algorithm.

Partition & Merge

The simple parallel *Cracking* algorithm divides an uncracked piece into T consecutive partitions that are concurrently cracked by T threads. Each thread cracks a partition by applying the original

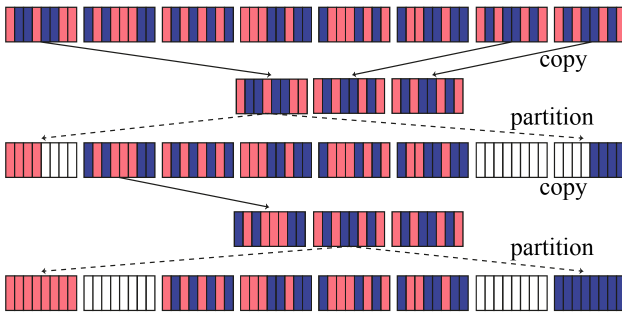


Figure 6: Vectorized Cracking

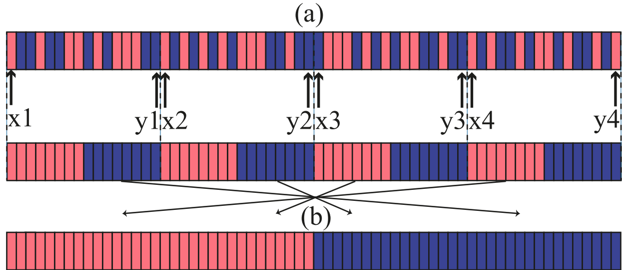


Figure 7: Simple Partition & Merge (multi-threaded)

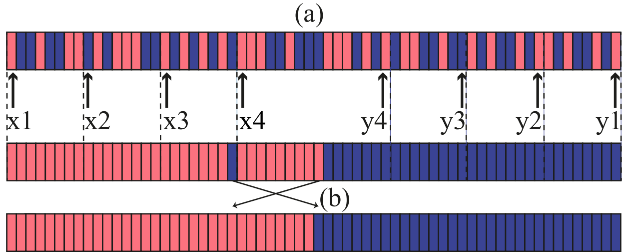


Figure 8: Refined Partition & Merge (multi-threaded)

Cracking algorithm. Finally, during the merge phase, a single thread swaps wrongly located blocks of values into their final position. Figure 7 shows an instance of the simple parallel *Cracking*. Four threads crack four partitions concurrently. Red indicates values that are less than the pivot, while blue indicates values that are greater than the pivot. After cracking all partitions, the merge phase takes place, i.e., a single thread relocates blocks of elements to the correct positions, resulting in the final cracked piece shown in Figure 7(b). During the merge phase the relocation of data causes many cache misses, which can be avoided with the refined partition & merge *Cracking* described in the following subsection.

Refined Partition & Merge

The refined partition & merge *Cracking* algorithm divides the uncracked piece into T partitions. The center partition is consecutive with size $S = \#elements/\#threads$, while the remaining $T-1$ partitions consist of two disjoint pieces that are arranged concentrically around the center partition. Assuming the selectivity is known and it is expressed as a fraction of 1, the size of the left piece equals to $S * selectivity$, while the size of the right piece equals to $S*(1-selectivity)$. For instance, in Figure 8(a), the size of the disjoint pieces is equal, since the selectivity is 0.5 (50%). As in the simple partition & merge *Cracking*, T threads crack the T partitions concurrently applying the original *Cracking* algorithm. The thread that cracks the center (consecutive) partition, swaps values within this partition. Each thread that cracks two disjoint pieces swaps wrongly located values between the two pieces. For example, in Figure 8(a) one thread exchanges values between the first and the last piece. Finally, a single thread (as in the simple parallel algorithm) locates wrongly-located blocks to the correct positions.

Although the refined algorithm swaps values that are in longer distance compared to the simple algorithm, it moves less data during the merge phase, because more data is already in the correct position. For instance, in Figure 8 only two values are located in wrong positions, while in Figure 7, we relocate 6 blocks of 8 values each. Both parallel algorithms make $O(n)$ comparisons/exchanges during the partitioning phase. However, the merging cost is significantly lower for the refined partition & merge *Cracking* algorithm.

CPU Efficiency & Parallelization

In principle, the single-threaded CPU efficiency improvements as presented in Section 4 are orthogonal to the thread-level parallelism presented above. Consequently, we can combine both techniques, hoping to accumulate their benefits. We focus on vectorization as this proved to yield better single-threaded CPU efficiency than predication or SIMD (cf., Sections 4 and 6).

Vectorization of the simple partition & merge *Cracking* algorithm is straight-forward. We simply have each thread perform vectorized *Cracking* instead of original *Cracking* on its contiguous partition. With the refined partition & merge *Cracking* algorithm, we need to additionally handle the case that, in case of skewed data, one of the two write cursors exceeds its partition half, and thus needs to "fast-forward" (or "jump") to the other half to continue.

Class	CPU	Cores	ISA	RAM
Desktop	AMD E-350	2	SSE4a	8GB
Workstation	Intel i7-4770	8[6]	AVX-2	32GB
Server	2×Intel E5-2650	2×16[6]	AVX	256GB
HE Server	4×Intel E5-4657L	4×24[6]	AVX	1024GB

Table 1: Hardware Setup

6. EVALUATION

Setup

We evaluated the presented implementations[5] on four different machines (see Table 1): a $300-class desktop machine, a $1,000-class workstation, a $10,000-class server and a $60,000-class high-end server. All experiments were evaluated on an array with 4GB of 32-bit integer values with varying selectivity/pivot position. We used Fedora 20, a 3.13.5 Linux kernel and gcc version 4.8.2. Since we compare single- as well as multi-threaded algorithms, we measure the average unix wallclock time of seven (memory-resident) runs rather than spent CPU-cycles or microop execution slots.

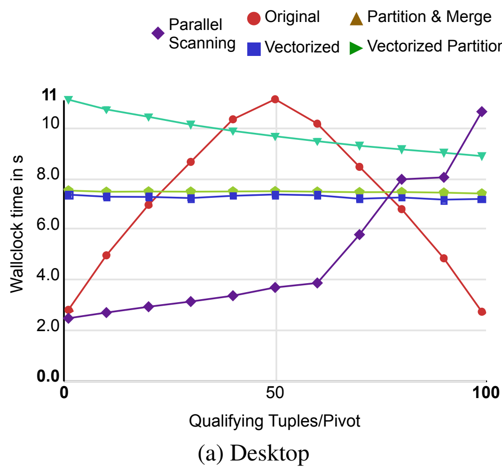
Results

Single-threaded Cracking

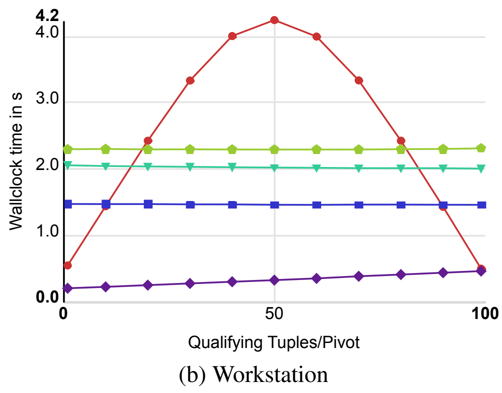
At first, let us look at single threaded performance (Figure 9): we are comparing the original cracking implementation to the single-threaded predicated (in register as well as cache) and the vectorized version. For reference, we also include the costs for the (parallel & predicated) scan which is (roughly) memory access bound in most cases (large intermediates lead to expensive swapping on

[5]Available for download at
`http://www.cwi.nl/~holger/cracking/sortvsscan`

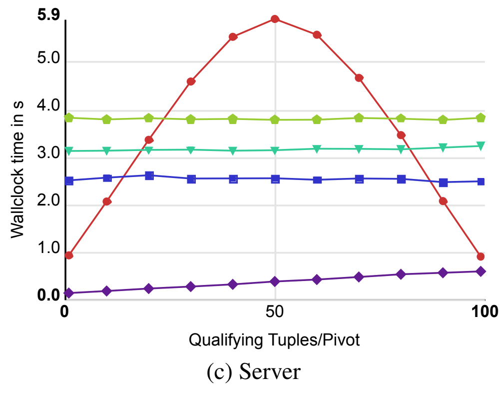
[6]Including virtual cores (Hyperthreading)



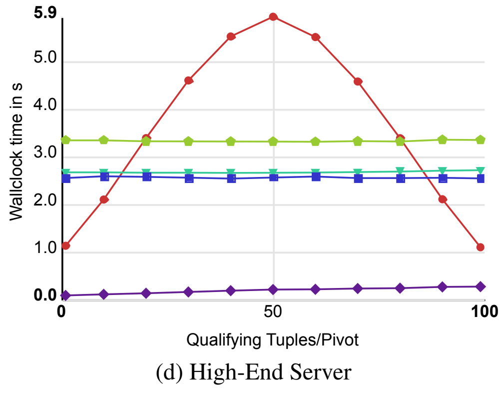
(a) Desktop



(b) Workstation

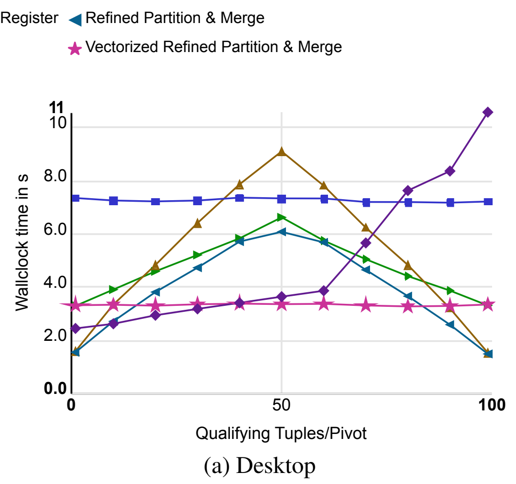


(c) Server

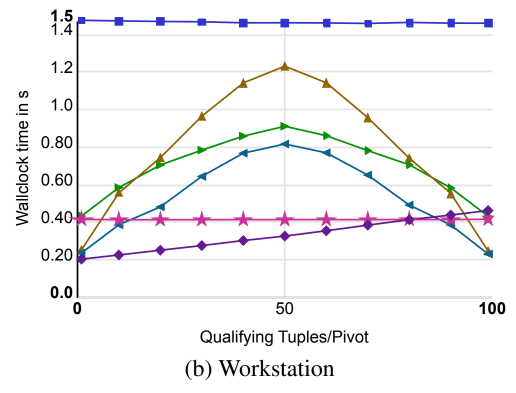


(d) High-End Server

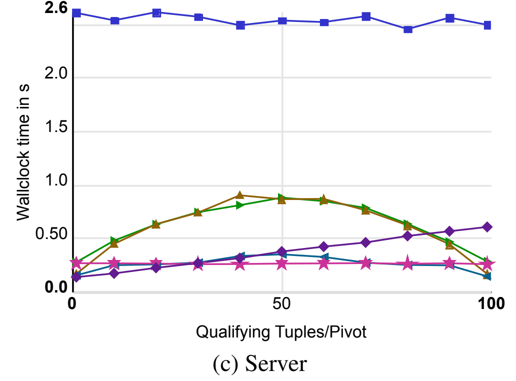
Figure 9: Single Threaded Performance



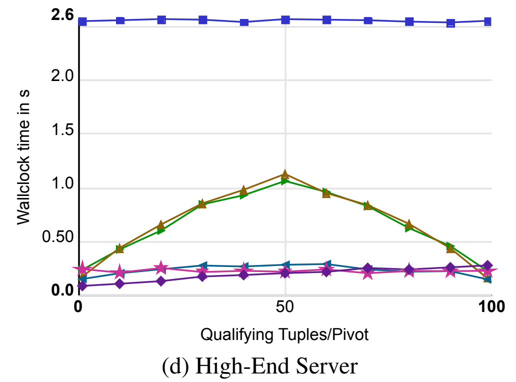
(a) Desktop



(b) Workstation



(c) Server



(d) High-End Server

Figure 10: Multi Threaded Performance

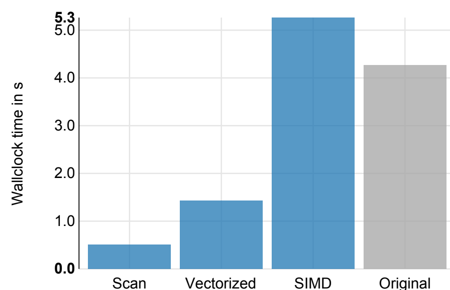


Figure 11: SIMD Processing Performance at 50% Selectivity

the desktop). The first observation is that (the original) *Cracking* is most expensive at 50% selectivity (incidentally the most useful case when considering the indexing aspect of *Cracking*). This is to be expected since this case yields the worst branch prediction performance. We observe that, at 50% selectivity, all systems benefit significantly from predication. Beyond that, things become more complicated. While the server and workstation systems achieve a benefit from keeping "active" and "backup" values in the same register, it even has a negative effect on the performance of the desktop system (that, surprisingly, decreases with increasing selectivity). While the branch-free algorithms perform better than the original *Cracking* for most of the selectivity spectrum, the better CPU performance does not outweigh the additional writes towards the ends of the spectrum. This is a common observation with predicated algorithms that stems from the better branch prediction at the ends of the spectrum.

SIMD

One of the most interesting (and disappointing) results of our experiments is the performance of the SIMD-based *Cracking* implementation (see Figure 11). The figure shows that the SIMD implementation performs significantly worse than the best single-threaded implementation (Vectorized) on our workstation system. It is even outperformed by the original *Cracking* implementation. While surprising at first, modeling the costs of this implementation provides a satisfying explanation: since an SIMD-word is only flushed to the output when it is completely filled with qualifying values, it (usually) takes multiple gathers to process one SIMD-word. Since every pointer has a certain probability to read a qualifying value, filling the SIMD-word can be modeled as a binomial process. The average number of gathers per flush can be derived from this model using stochastic analysis (we omit the details for lack of space). For a word-length of 8 values and a pivot in the middle of the range, it takes around 4.42 gathers to fill a word. Given that each gather costs 6 cycles (on Nehalem), it takes, on average, more than three cycles per value to only gather the values. Adding the costs for cursor advancing and predicate evaluation, the costs of the SIMD-implementation are prohibitively high.

Multi-threaded

The results of our multi-threaded experiments are displayed in Figure 10. To accommodate to the varying number of cores in our experimentation platforms, we set the degree of parallelism to the number of (virtual) cores in each machine (see Table 1). For reference, we include the best single-threaded implementation (Vectorized) in the chart. We observe a significant speedup in almost all cases. Naturally, the *Refined Partition & Merge* implementation performs better than its plain counterpart. In addition, both implementations achieve a performance improvement if combined with vectorization. This effect is, however less pronounced on the highly parallel server systems. On the 96-core High-End Server system it is virtually non-existent. In general, we found the *Vectorized Refined Partition & Merge* implementation the fastest of our implementation across all parameters. In fact, it even outperforms (parallel & predicated) scanning in some cases: the in-place nature of *Cracking* yields fewer cache-line fill misses than the out-of-place scan and gives it a (slight) performance edge.

7. CONCLUSION

CPU-efficient implementation of even simple algorithms is hard: while common knowledge in many fields of computer science, this insight is still not properly appreciated in the field of data management. In this paper, we conducted an in-depth study of such a supposedly simple algorithm: pivoted partitioning. We demonstrated that, in its naïve implementation, it is not an I/O bound algorithm. Starting from this understanding, we systematically analyzed and addressed the dominant cost factors using state-of-the-art techniques. The result is an implementation that rivals and sometimes even outperforms a parallelized scan on a variety of systems. In that, it is up to 25 times faster than the initial implementation.

References

[1] *Intel 64 and IA-32 Architectures Optimization Reference Manual*. Appendix B: Using Performance Monitoring Events. Intel Corporation, June 2013.

[2] S. Agrawal, S. Chaudhuri, L. Kollár, A. P. Marathe, V. R. Narasayya, and M. Syamala. Database Tuning Advisor for Microsoft SQL Server 2005. In *VLDB*, pages 1110–1121, 2004.

[3] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. DBMSs on a Modern Processor: Where Does Time Go? In *VLDB*, pages 266–277, 1999.

[4] C. Balkesen, G. Alonso, J. Teubner, and M. T. Özsu. Multi-Core, Main-Memory Joins: Sort vs. Hash Revisited. *PVLDB*, 7(1):85–96, 2013.

[5] P. A. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR*, pages 225–237, 2005.

[6] N. Bruno and S. Chaudhuri. An Online Approach to Physical Design Tuning. In *ICDE*, pages 826–835, 2007.

[7] J. Chhugani, A. D. Nguyen, V. W. Lee, W. Macy, M. Hagog, Y.-K. Chen, A. Baransi, S. Kumar, and P. Dubey. Efficient Implementation of Sorting on Multi-Core SIMD CPU Architecture. *PVLDB*, 1(2):1313–1324, 2008.

[8] J. Cieslewicz and K. A. Ross. Adaptive Aggregation on Chip Multiprocessors. In *VLDB*, pages 339–350, 2007.

[9] B. Dageville, D. Das, K. Dias, K. Yagoub, M. Zaït, and M. Ziauddin. Automatic SQL Tuning in Oracle 10g. In *VLDB*, pages 1098–1109, 2004.

[10] G. Graefe, F. Halim, S. Idreos, H. A. Kuno, and S. Manegold. Concurrency Control for Adaptive Indexing. *PVLDB*, 5(7):656–667, 2012.

[11] G. Graefe, F. Halim, S. Idreos, H. A. Kuno, S. Manegold, and B. Seeger. Transactional Support for Adaptive Indexing. *VLDBJ*, 23(2):303–328, 2014.

[12] G. Graefe, S. Idreos, H. A. Kuno, and S. Manegold. Benchmarking Adaptive Indexing. In *TPCTC*, pages 169–184, 2010.

[13] G. Graefe and H. Kuno. Adaptive Indexing for Relational Keys. In *ICDE Workshops*, pages 69–74, 2010.

[14] G. Graefe and H. A. Kuno. Self-Selecting, Self-Tuning, Incrementally Optimized Indexes. In *EDBT*, pages 371–381, 2010.

[15] F. Halim, S. Idreos, P. Karras, and R. H. C. Yap. Stochastic Database Cracking: Towards Robust Adaptive Indexing in Main-Memory Column-Stores. *PVLDB*, 5(6):502–513, 2012.

[16] J. L. Hennessy and D. A. Patterson. *Computer Architecture - A Quantitative Approach*. Morgan Kaufmann, 5th edition, 2012.

[17] C. A. R. Hoare. Algorithm 64: Quicksort. *CACM*, 4(7), 1961.

[18] S. Idreos. *Database Cracking: Towards Auto-tuning Database Kernels*. PhD thesis, CWI, June 2010.

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

[20] S. Idreos, M. L. Kersten, and S. Manegold. Updating a Cracked Database. In *SIGMOD*, pages 413–424, 2007.

[21] S. Idreos, M. L. Kersten, and S. Manegold. Self-Organizing Tuple Reconstruction in Column-Stores. In *SIGMOD*, pages 297–308, 2009.

[22] S. Idreos, S. Manegold, H. A. Kuno, and G. Graefe. Merging What's Cracked, Cracking What's Merged: Adaptive Indexing in Main-Memory Column-Stores. *PVLDB*, 4(9):585–597, 2011.

[23] M. Johnson. *Superscalar Microprocessor Design*. Prentice Hall, 1991.

[24] C. Kim, E. Sedlar, J. Chhugani, T. Kaldewey, A. D. Nguyen, A. D. Blas, V. W. Lee, N. Satish, and P. Dubey. Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-Core CPUs. *PVLDB*, 2(2):1378–1389, 2009.

[25] M. Lühring, K.-U. Sattler, K. Schmidt, and E. Schallehn. Autonomous Management of Soft Indexes. In *SMDB*, pages 450–458, 2007.

[26] S. Richter, J.-A. Quiané-Ruiz, S. Schuh, and J. Dittrich. Towards zero-overhead static and adaptive indexing in hadoop. *VLDBJ*, 23(3):469–494, 2014.

[27] K. A. Ross. Selection Conditions in Main Memory. *TODS*, pages 132–161, 2004.

[28] K. Schnaitter, S. Abiteboul, T. Milo, and N. Polyzotis. COLT: Continuous On-Line Tuning. In *SIGMOD*, pages 793–795, 2006.

[29] F. M. Schuhknecht, A. Jindal, and J. Dittrich. The Uncracked Pieces in Database Cracking. *PVLDB*, 7(2):97–108, 2013.

[30] J. Zhou and K. A. Ross. Implementing Database Operations Using SIMD Instructions. In *SIGMOD*, pages 145–156, 2002.

[31] D. C. Zilio, J. Rao, S. Lightstone, G. M. Lohman, A. J. Storm, C. Garcia-Arellano, and S. Fadden. DB2 Design Advisor: Integrated Automatic Physical Database Design. In *VLDB*, pages 1087–1097, 2004.