Main Memory Adaptive Indexing for Multi-core Systems

Victor Alvarez Felix Martin Schuhknecht Jens Dittrich Stefan Richter

Information Systems Group
Saarland University
http://infosys.cs.uni-saarland.de

ABSTRACT

Adaptive indexing is a concept that considers index creation in databases as a by-product of query processing; as opposed to traditional full index creation where the indexing effort is performed up front before answering any queries. Adaptive indexing has received a considerable amount of attention, and several algorithms have been proposed over the past few years; including a recent experimental study comparing a large number of existing methods. Until now, however, most adaptive indexing algorithms have been designed single-threaded, yet with multi-core systems already well established, the idea of designing parallel algorithms for adaptive indexing is very natural. In this regard, and to the best of our knowledge, only one parallel algorithm for adaptive indexing has recently appeared in the literature: The parallel version of standard cracking. In this paper we describe three alternative parallel algorithms for adaptive indexing, including a second variant of a parallel standard cracking algorithm. Additionally, we describe a hybrid parallel sorting algorithm, and a NUMA-aware method based on sorting. We then thoroughly compare all these algorithms experimentally. Parallel sorting algorithms serve as a realistic baseline for multi-threaded adaptive indexing techniques. In total we experimentally compare seven parallel algorithms. The initial set of experiments considered in this paper indicates that our parallel algorithms significantly improve over previously known ones. Our results also *suggest* that, although adaptive indexing algorithms are a good design choice in single-threaded environments, the rules change considerably in the parallel case. That is, in future highly-parallel environments, sorting algorithms could be serious alternatives to adaptive indexing.

1. INTRODUCTION

Traditionally, retrieving data from a table in a database is improved by the use of indexes when highly selective queries are performed. Among the most popular data structures that are used as indexes we can find self-balancing trees, B-trees, hash maps, and bitmaps. However, on the one hand, building an index requires extra space, but perhaps most importantly, it requires time. Without any knowledge about the workload, the best hint a database has to build an index, is to simply build it up front the first time data is accessed. This, ironically, could slow down the whole workload of a database. Furthermore, it could be the case that only few queries are performed over different attributes, or it could simply be the case that response time per query, including the very first one, is the most important measure. In these cases, the up-front effort in building indexes does not really pay off. On the other hand, no index at all is in general not a solution either, as full table scans incur very quickly in high total execution times. Hence, ideally, we would like to have a method that allows us to answer *all* queries as fast as if there were an index, but with initial response time as if there were no index. This is the spirit of adaptive indexing.

In adaptive indexing, index creation is thought as a by-product of query execution; thus an index is built in a lazy manner as more queries are executed, see Figure 3. The very first adaptive indexing algorithm, called *standard cracking*, was presented in [10], and since then, adaptive indexing has received a considerable amount of research, see [5, 6, 7, 8, 11, 12, 13, 14, 21].

DaMoN '14, June 22-27 2014, Snowbird, UT, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM. ACM 978-1-4503-2971-2/14/06 ...$15.00.
http://dx.doi.org/10.1145/2619228.2619231.

1.1 Single-threaded adaptive indexing

In our recent paper [21] we presented a thorough experimental study of all major single-threaded adaptive indexing algorithms. Experiments shown therein support the claim that (single-threaded) standard cracking [10] still keeps being the algorithm any other new (single-threaded) algorithm has to improve upon, due to its simplicity and good accumulated runtime. For example, in the comparison of standard cracking [10] with stochastic cracking [8], as seen from the experiments shown in [21], the latter is more robust, but the former is in general faster — see Figure 4.c of [21]. With respect to hybrid cracking algorithms [13], the experiments of [21] indicate that, although convergence towards full index improves in hybrid cracking algorithms, this improvement can be seen only after roughly 100–200 queries, before that, standard cracking performs clearly better — see Figure 4.a of [21]. Moreover, as also seen in Figure 8.a of [21], w.r.t. the total accumulated query time, standard cracking is faster up to around 8000 queries. In all cases, standard cracking is an algorithm that is much easier to implement and maintain.

In [21] we also observed that pre-processing the input before standard cracking [10] is used, significantly improves the convergence towards full index, robustness, and total execution time of standard cracking. This pre-processing step is just a range-partitioning over the attribute to be (adaptively) indexed. For simplicity we will refer from now on to this adaptive indexing technique simply as the *coarse-granular index*, just as it is referred to in [21]. From the experiments of [21], the improvement of the coarse-granular index over all other adaptive indexing techniques considered therein can clearly be seen, see Figures 7.a and 7.b of [21]. However, the coarse-granular index also incurs in a higher initialization time, which, as discussed previously, is in general not desirable.

1.2 Representative single-threaded experiment

Taking the experiments of [21] as a reference, we have run a small, but very representative, set of experiments considering what we believe are the three best adaptive indexing algorithms to date — standard cracking [10], hybrid crack sort [13], and the coarse-granular index [21]. We additionally compare these methods with sorting algorithms.

The purpose of these experiments is *not* to reevaluate our own material of [21], but rather to make this paper from this point on self-contained. Also, these experiments represent the baselines for the multi-threaded algorithms shown later on — these experiments are the entry point of all other experiments herein presented.

The source code used for this representative set of experiments is a tuned version of the code we used in [21]. The workload under which we test the algorithms is defined in Section 1.3 below, and the experimental set-up is given in Section 4.

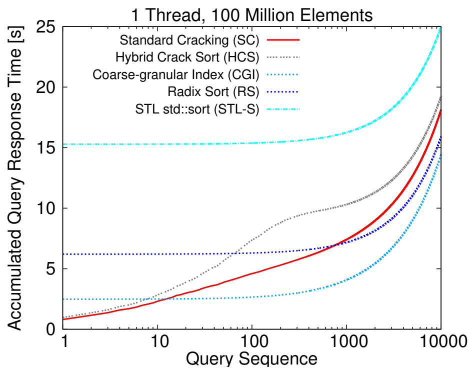

Figure 1: Total time of the single-threaded algorithms over 10,000 random queries.

The results of this first set of experiments can be seen in Figure 1. The most important message, at this point, is that the reader can rest assured that these experimental results are fully consistent with the ones already found in the literature, in particular with our own comprehensive comparison of single-threaded adaptive indexing algorithms presented in [21]. In particular, for a moderately large number of queries, we see that adaptive indexing algorithms achieve a high speedup over sorting algorithms, typically used for building full indexes. Thus, in single-threaded environments, adaptive indexing is a viable option in practice.

1.3 Workload

In line with previous work [5, 6, 8, 10, 13, 14, 21], we assume a main-memory column-store with 100 million tuples (which we also increase to 1 billion tuples later on), filled with 4-byte positive integers generated at random, w.r.t. the perfect uniform distribution over the interval $\left[0, 2^{32}\right)$. Each entry in the cracker column is then represented as a pair $(key, rowID)$ of 4-byte positive integers each. We are uniquely interested in analytical queries, thus we assume that no query performs updates on the database. To be more precise, we are interested in the following type of queries:

$$\texttt{SELECT SUM}(R.A)\ \texttt{FROM}\ R\ \texttt{WHERE}\ q_l \leq R.A < q_h$$

That is, each query has to filter the data from A and perform a simple aggregation over the set of results. The cracker column will always be clear from the context, thus, a query q can then be represented simply as a pair (q_l, q_h). We perform $10,000$ queries generated uniformly at random (which we change to skewed distribution later on), each having its selectivity set at 1%. As in previous work, we assume that queries are independent from each other and are performed sequencially as they arrive. Also, we assume no idle time in the system, *i.e.*, the cracker column is allocated immediately after the very first query arrives. Every measurement was run ten times. The input of the i-th run is the same for every algorithm, but any two different runs were generated completely independent from each other. The times reported for each algorithm are the average times over those ten runs. In all experiments we project only the index attribute. This means that no base table access is performed. In Section 1.5 we elaborate more on the considered workload.

1.4 The contributions of this paper

All algorithms tested in [21] are single-threaded, and actually, almost *all* adaptive indexing algorithms in the literature are designed for single-core systems. With multi-core systems not only on the rise, but actually well established by now, the idea of parallel adaptive indexing algorithms is very natural.

To the best of our knowledge, the work presented in [5, 6] is the only (published) work, so far, that deals with adaptive indexing in multi-core systems. Therein, a parallel version of the standard cracking algorithm of [10] is presented.

We are now ready to list the contributions of this paper:

(1) First and foremost, the most important contribution of this paper is the (critically) experimental evaluation of many parallel algorithms (six to be precise), for adaptive indexing as well as for creating full indexes — including the algorithm of [5, 6]. This work can be considered the very first experimental study of parallel adaptive indexing techniques.

(2) We describe three (natural) alternative parallel algorithms for adaptive indexing, one based on standard cracking [10], while the other two are based on the coarse-granular index presented in [21].

(3) We describe a hybrid parallel sorting algorithm. This algorithm greatly resembles the parallel radix sorting algorithms presented in [15, 18, 20]. Additionally, we present another method that further improves upon this hybrid parallel sorting algorithm by adding NUMA-awareness.

(4) The initial set of experiments considered in this paper *suggests* that our algorithms significantly improve over the methods presented in [5, 6]. Moreover, these experiments also indicate that, as opposed to the story with single-threaded algorithms where sorting algorithms are no match in practice against adaptive indexing techniques, parallel sorting algorithms could become serious alternatives to parallel adaptive indexing techniques.

Table 1 gives an overview of all algorithms considered in this paper. Throughout the paper, due to lack of space, we mostly refer to the algorithms by their short names. Thus, for an easier distinction between single- and multi-threaded algorithms, the short name of *each* multi-threaded algorithm starts with a **P**.

1.5 How is this paper not to be understood

(1) We are fully aware of *all* prior work in adaptive indexing, including the very recent paper of Graefe *et al.* [6]. In this

Algorithm	Reference	Description	Short name	Implementation used
Standard cracking	[10]	B.1	**SC**	[21]
Hybrid crack sort	[13]	[13]	**HCS**	[21]
Coarse granular index	[21]	B.2	**CGI**	[21]
Radix sort	[17]	B.3	**RS**	[21]
STL sort (C++)	[22]	[22]	**STL-S**	C++ STL
Parallel standard cracking	[5, 6]	§ 2.1	**P-SC**	[5]
Parallel-chunked standard cracking	This paper	§ 2.1	**P-CSC**	Ours
Parallel coarse-granular index	This paper	§ 2.2	**P-CGI**	Ours
Parallel-chunked coarse-granular index	This paper	§ 2.2	**P-CCGI**	Ours
Parallel range-partitioned radix sort	This paper	§ 2.3	**P-RPRS**	Ours
Parallel-chunked radix sort	This paper	§ 2.3	**P-CRS**	Ours

Table 1: All algorithms considered in this paper along their original references, their short names, and the implementations used in this paper.

regard, it is *not* the purpose of this paper to present parallel versions of *every* single adaptive indexing algorithm in the literature. Our previous experimental study [21] helped us to narrow down our options to the adaptive indexing techniques that show real potential for parallelization.

(2) The workload considered in this paper, given in Section 1.3, is *fully* equivalent to what has been considered in the literature so far — most of the work has used this sort of workload (integer-filled single-column) as an entry case of study, and actually, only the work presented in [12, 21] has gone to a more realistic workload. Thus, this paper should be considered as the *entry* point towards a thorough study of multi-threaded adaptive indexing techniques. Moreover, the multi-threaded standard cracking algorithm presented in [5, 6] — which happens to be the only one so far — has only been compared against single-threaded methods. We feel that once multi-threaded environments enter the picture, many natural choices for multi-threaded adaptive indexing enter the picture as well. It is also our purpose to compare the algorithm of [5, 6] against these natural choices.

The implementations used in this paper are described in Table 1 on an algorithm basis. These implementations will be made available upon publication of the paper.

1.6 The structure of this paper

The remainder of the paper is structured as follows: In Section 2 we briefly describe the algorithms to be tested. In Section 3 we give our experimental setup. In Section 4 we give a precise definition of the workload used to test the algorithms, and show the results of our experiments. In Section 5 we close the paper with our conclusions. For completeness, the reader can find in the appendix a discussion about the observed speedups (Appendix C) as well as additional experiments (Appendix D) showing the effect of varying different parameters of the workload, such as input size, tuple configuration, query access pattern, and query selectivity.

2. ALGORITHMS

In this section we give a short description of all parallel algorithms to be tested. The main part of this work contains no description of the single-threaded algorithms considered due to lack of space. These descriptions can be found in the Appendix B.

Throughout the paper we will denote by A the original column (attribute) we want to perform queries on, by B the corresponding cracker column, see Figure 3, by n the number of entries in A, and by k the number of available threads. To make the explanation simpler, we will assume that n is perfectly divisible by k. In the experiments, however, we do not make this assumption.

2.1 Parallel standard cracking

In [5, 6] a multi-threaded version of standard cracking was shown, which we will be denoted by **P-SC** from now on. To describe this multi-threaded version it suffices to observe that, in standard cracking (**SC**), as more queries arrive, they potentially partition independent parts of B, and thus, they can be performed in parallel.

When a query comes, it has to acquire two write locks on the border partitions, while *all* partitions in between are protected using read locks. When two or more queries have to partition, or aggregate over the same part of B, read and write locks are used over the relevant parts. That is, whenever two or more queries $q_1, \ldots, q_r$, $r \geq 2$, want to partition the same part of B, a write lock is used to protect that part; say q_i, $1 \leq i \leq r$, obtains the lock and partitions while the other queries wait for it to finish. After q_i has finished, the next query q_j, $i \neq j$, acquiring the lock has to reevaluate what part it will exactly crack, as q_i has modified the part all queries $q_1, \ldots, q_r$ were originally interested in. Clearly, as more queries are performed, the number of partitions in B increases, and thus also the probability that more queries can be performed in parallel. This is where the speedup of this multi-threaded version over the single-threaded version stems from. If two or more queries want to aggregate over the same part of B, then they all can be performed in parallel, as they are not physically reorganizing any data. However, if one query wants to aggregate over a part of B that is currently being partitioned by another query, then the former has to wait until the latter finishes, as otherwise the result of the aggregation might be incorrect. Also, all queries work with the same cracker index, thus a write lock occurs each time the cracker index is updated.

As the initialization time of **P-SC** we consider the time it takes to copy A onto B in parallel — in contrast to **SC**, where this initialization is done single-threaded. That is, for k available threads, we divide A and B into k parts and assign exactly one part to each thread. Every thread copies its corresponding part from A to B.

We immediately see two drawbacks with this multi-threaded version of standard cracking, which will also become apparent in the experiments: (1) The effect of having multiple threads will be visible only after the very first executed query has partitioned B. Before that, B consists of only one partition, and all other queries will have to wait for this very first query to finish. That is, the very first crack locks the whole column. (2) Locking incurs in unwanted time overheads.

To address these concerns we describe in this paper *another* version of parallel standard cracking, which as we will see, seems to perform quite good in practice, in particular, better than **P-SC**.

Parallel-chunked standard cracking (P-CSC). After copying A onto B in parallel, as in **P-SC**, we (symbolically) divide B into k parts, each having $\frac{n}{k}$ elements, and every thread will be responsible for exactly one of these parts. Now, *every* query will be executed by *every* thread on its corresponding part, and *every* thread will aggregate its results to a local variable assigned to it. At the end a single thread aggregates over all these local variables.

It is crucial for the performance of **P-CSC** to ensure complete independence between the individual parts. Any data that is unnecessarily shared among them can lead to *false sharing effects* (propagation of cache line update to a core although the update did not affect its part of the shared cache line) and *remote accesses* to memory attached to another socket. Thus, each part maintains its own structure of objects, containing its local data, cracker index, histograms, and result aggregation variables. Furthermore, by aligning all objects to cache lines, we ensure to avoid any shared resources, and each thread can process its part in complete independence from the remaining ones. Finally, we also pin threads (during its lifetime) to physical cores. This gives the strong hint that, when a thread instantiates its part — and all variables around it — it should do so in its NUMA region.

2.2 Parallel coarse-granular index

In this paper we present two parallel versions of the coarse-granular index (**CGI**) presented in [21]. For the first one it suffices to observe that the coarse-granular index is nothing but a range partitioning as a pre-processing step to standard cracking (**SC**). Thus, for the first parallel version of **CGI**, which will be denoted by **P-CGI** from now on, we show how to do a range partitioning in parallel. Afterwards we simply run the parallel standard cracking algorithm (**P-SC**) [5, 6] to answer the queries, taking into consideration that B is now range-partitioned. Our method to build a range partition in parallel requires no synchronization among threads, which of course helps to improve its performance.

Parallel coarse-granular index (P-CGI). The main idea behind the construction is very simple. Column A is (symbolically) divided into k parts, of $\frac{n}{k}$ elements each. Thread t_i, $1 \leq i \leq k$, gets assigned the i-th part of A and it writes its elements to their corresponding buckets[1] in the range partition on B, using $r \geq 2$ buckets. In order to do so, and not to incur in any synchronization overhead, *every* bucket of the target range partition on B is (symbolically) divided into k parts as well, so that thread t_i writes its elements in the i-th part of *every* bucket. Thus, clearly, any two threads read their elements from independent parts of A and write also to independent parts on B, see Figure 4. All this can be implemented in a way that all but one step are done in parallel[2], and every thread gets roughly the same amount of work. This, as we will see, helps to improve performance as the number of threads increases.

The second parallel version of **CGI** does not range-partition B. Instead, it works in the same spirit as **P-CSC**, thus its name.

Parallel-chunked coarse-granular index (P-CCGI). Symbolically divide A again into k parts, of $\frac{n}{k}$ elements each, and assign the i-th part to the i-th thread. Each thread t_i, $1 \leq i \leq k$, range partitions its part using **CGI**, materializing it onto B. Thus, B is also (symbolically) divided into k parts. Having done this chunked range partition of B, thread t_i keeps being responsible for the i-th range partition of B. When a query arrives, each thread executes **SC** on its part and aggregates its result in a global variable, for which we again use a write lock to avoid any conflicts that might occur during aggregation. Again, as for **P-CSC**, we ensure that no resources are shared among the parts. All objects are cache-line-aligned and no trips to the remote memory are necessary at any place. The initialization time for each one of these algorithms, **CGI**, **P-CGI**, and **P-CCGI**, is the time it takes to materialize the necessary range partitions on B.

[1]Our implementation of range partition is radix-based.

[2]This step is the aggregation of an histogram used by all threads.

2.3 Parallel full indexing

So far we have only described algorithms for adaptive indexing. However, in order to see how effective those algorithms really are, we have to compare them against full indexes. In [21] it was observed that when the selectivity of range queries is not extremely high, as in our case, more sophisticated indexing data structures such as AVL-trees, B$^+$-trees, ART [16], among others, have no significant benefit over full sort + binary search + scan for answering queries, as the scan cost (aggregation) of the result dominates the overall query time. Therefore, for this study, we regard sorting algorithms as a direct equivalent of full indexing algorithms. With this in mind, and also due to (1) the good performance of building a range partition in parallel, and (2) the good performance of the radix sort algorithm (**RS**) of [17], the following hybrid sorting algorithm suggests itself:

Parallel range-partitioned radix sort (P-RPRS). Build a range partitioning in parallel on B, as in the parallel coarse-granular index **P-CGI**. If the number of buckets in the range partitioning is $r = 2^m$, then it is not hard to see that the elements of B are now sorted w.r.t. the m most significant bits. Now, split the buckets of the range partitioning evenly among all k threads. Each thread then sorts the elements assigned to it on a bucket basis using the radix sort **RS**, but starting from the $(m+1)$-th most significant bit; remember that **RS** is a MSD radix sort. Since B is range-partitioned, and sorted w.r.t. the m most significant bits, calling **RS** on each bucket clearly fully sorts B in-place.

As we will see, **P-RPRS** performs quite good in practice. However, this algorithm suffers from a large amount of NUMA effects as it is discussed in Appendix C.3. To alleviate this we present the following algorithm:

Parallel-chunked radix sort (P-CRS). Symbolically divide A into k chunks, of $\frac{n}{k}$ elements each, and assign the i-th part to the i-th thread. Each thread t_i, $1 \leq i \leq k$, range partitions its part using **CGI**, materializing it onto the corresponding chunk of B. Afterwards, each thread t_i reuses the histogram of its chunk to sort these partitions using **RS** starting from the $(m+1)$-th most significant bit (the range-partitioning already sorts w.r.t. the m most significant bits). Thus, **P-CRS** basically applies the concept of **P-RPRS** to k chunks. As for all other chunked methods, we ensure that the chunks do not share any data structures and that all objects are again cache-line-aligned. Thus, the threads work completely independent from each other.

The initialization time for the parallel sorting algorithms **P-RPRS** and **P-CRS** is clearly the time it takes them to sort. After that, for **P-RPRS** the queries can be answered in parallel using binary search; every thread will answer a different query (inter-query parallelism). In contrast to that, the chunked **P-CRS** answers the individual queries in parallel (intra-query parallelism) by querying the chunks concurrently.

Table 1 in Section 1.4 constitutes a summary of the algorithms (experimentally) considered in this paper. We have decided to leave linear scan and hybrid cracking algorithms out of the presentation due to their high execution time, even in parallel.

3. EXPERIMENTAL SETUP

Our test system consists of a single machine having two Intel Xeon E5-2407 running at 2.20 GHz. Each processor has four cores, and thus the machine has eight (hardware) threads. Hyper-threading and turbo-boost is not supported by the processors. The L1 and L2 cache sizes are 32 KB and 256 KB respectively per core. The L3 cache is shared by the four cores in the same socket and has size 10 MB. The machine has a total of 48 GB of shared RAM.

The operating system is a 64-bit version of Linux. All programs are implemented in `C++` and compiled with the Intel compiler `icpc 14.0.1` [1] with optimization `-O3`.

We are aware that our test server (up to eight cores) might be somewhat dated in comparison to what is nowadays available (say 64 cores or more). However, we would like to point out that running experiments on (very) large servers makes understanding the behavior of the algorithms unnecessarily complicated — in the end we want to obtain reasonable explanations as of why the algorithms behave the way they do. Also, and more importantly, splitting computing resources becomes important in large servers executing different kinds of workloads simultaneously. Thus, if certain kinds of workloads show reasonable speedups when only using a "small" number of threads (say four or eight), we see no reason why they should be assigned more computing resources.

4. CORE EXPERIMENTS

To better observe the effect of the number of threads in all algorithms, we ran the experiments with 2, 4, and 8 threads. For completeness, the reader can find in Appendix D an additional set of experiments showing the effect of varying different parameters of the workload, such as input size, tuple configuration, query access pattern, and query selectivity.

4.1 Benchmarks considered

The experiments for one thread correspond to our small representative experiment discussed in Section 1.2, and whose results are shown in Figure 1. The two benchmarks considered in this evaluation are: Initial response time, and total execution time.

From the initial response time we get an idea on how long it takes the algorithms to start answering queries once they have been asked to. That is, we consider the initialization time, as explained in Section 2 for each algorithm, plus the time taken by the very first query. For single-threaded algorithms, initial response time is the strongest point in favor of standard cracking, and against sorting algorithms and other adaptive indexing techniques. From Figure 1 we can observe that the single-threaded version of standard cracking (**SC**) can perform around 750 queries while single-threaded version of radix sort (**RS**) catches up. After that threshold **RS** becomes faster than **SC**. The comparison against **STL-S** (std::sort) is even worse. The story looks very different when comparing the single-threaded version of the coarse-granular index (**CGI**) against **SC**. From the same Figure 1 we observe that **SC** can perform only about 10 queries while **CGI** is building its range partitioning. After that threshold the coarse-granular index is already faster.

From the total execution time we clearly obtain the overall speedup of the multi-threaded algorithms over their single-threaded counterparts. In this regard we would like to point out that we are *mainly* interested in the time it takes to perform X number of queries (query throughput) independent of the algorithm.

Due to lack of space we show here only the figure corresponding to the 8-threaded experiment, Figure 2.

4.2 Single-threaded vs. multi-threaded

Based on the results we obtained in the single-threaded experiment, Figure 1, we now show how the parallel algorithms perform w.r.t. their single-threaded counterparts. In this section we only present the results of the experiments, the speedups observed are discussed in details in Appendix C.

In Table 2 we compare all standard cracking algorithms (multi-threaded versions were described in Section 2.1). In Table 3 we compare all coarse-granular algorithms (multi-threaded versions were described in Section 2.2). In Table 4 we compare all sorting

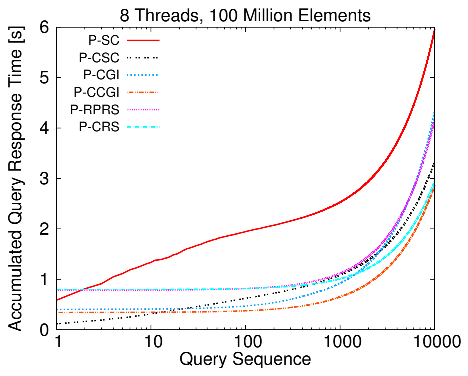

Figure 2: Total time of the 8-threaded algorithms over 10,000 random queries.

algorithms (multi-threaded versions were described in Section 2.3). It is worth noting that we also test the parallel radix sort algorithm presented in [18]; which was therein reported to be very fast in practice. However, the two parallel radix-based algorithms (**P-RPRS** and **P-CRS**) described here turned out to be faster, so we decided to leave the algorithm of [18] out of the discussion. The complete set of experiments (including the algorithm of [18]) can be found in the extended version [3] of this work.

In all tables, times are reported in seconds. The speedup reported is w.r.t. the single-threaded version of the respective algorithm. The absolute best times among all algorithms, w.r.t. the number of threads and benchmarks, are highlighted in blue.

It comes as no surprise that multi-threaded versions of the algorithms performed better than the single-threaded ones. With respect to absolute initial response time, we can see that **P-CSC** is the fastest algorithm, followed by **P-SC** and **P-CCGI**. Yet, it is quite interesting to see that sorting algorithms profited from parallelism the most — the highest speedups are observed there, up to 7.903× speedup for eight threads. That is, we are now able to build a full index over the main column A eight times faster.

With respect to total execution time, the fastest algorithm turned out to be **P-CCGI**, followed rather close by **P-CRS** and **P-CSC** — the chunked methods turned out to be the fastest w.r.t. total execution time, where we can see a five-fold increase in speed for eight threads. See Appendix C for a discussion on the observed speedups (for both benchmarks considered).

Algorithm	Threads	**Initial time**	Speedup	**Total time**	Speedup
SC	1	0.8076	1×	18.14	1×
P-SC	2	0.6617	1.22×	13.82	1.313×
P-CSC	2	0.4067	1.986×	9.051	2.004×
P-SC	4	0.5744	1.406×	8.225	2.205×
P-CSC	4	0.2093	3.859×	4.95	3.664×
P-SC	8	0.5863	1.377×	5.957	3.044×
P-CSC	8	0.1178	6.859×	3.344	5.423×

Table 2: Comparison of standard cracking algorithms. Times are shown in seconds.

5. CONCLUSIONS

The most important lessons learned from the initial set of experiments presented in this work are the following:

1. Let us assume for one moment that a system has to decide whether to use the standard cracking algorithm of [5, 6, 10]

Algorithm	Threads	**Initial** time	Speedup	**Total** time	Speedup
CGI	1	2.483	1×	14.43	1×
P-CGI	2	1.488	1.668×	10.9	1.324×
P-CCGI	2	1.243	1.997×	7.212	2.001×
P-CGI	4	0.7456	3.33×	6.086	2.371×
P-CCGI	4	0.6293	3.946×	4	3.608×
P-CGI	8	0.4032	6.158×	4.345	3.321×
P-CCGI	8	0.3436	7.226×	2.867	5.033×

Table 3: Comparison of coarse-granular index algorithms. Times are shown in seconds.

Algorithm	Threads	**Initial** time	Speedup	**Total** time	Speedup
RS	1	6.201	1×	15.91	1×
P-RPRS	2	2.894	2.143×	9.833	1.618×
P-CRS	2	2.792	2.221×	8.038	1.98×
P-RPRS	4	1.467	4.227×	5.306	2.999×
P-CRS	4	1.485	4.176×	4.453	3.574×
P-RPRS	8	0.7846	7.903×	4.163	3.823×
P-CRS	8	0.797	7.78×	2.94	5.413×

Table 4: Comparison of sorting algorithms. Times are shown in seconds.

or build a full index. For this, the system schedules $X \geq 1$ number of threads. From our experiments we can see that a sorting algorithm pays off earlier as X increases. For instance, the crossing point between the single-threaded algorithms — standard cracking algorithm (**SC**) and radix sort (**RS**) — is around 750 queries, see Figure 1. When X = 2,4,8, the crossing point moves to around 100, 10, 2 queries, respectively, between **P-SC** [5, 6] and **P-RPRS**. Figure 2 represents the case $X = 8$. This is a strong argument in favor of parallel sorting algorithms.

2. The parallel adaptive indexing techniques presented in this paper (**P-CSC**, **P-CGI**, **P-CCGI**) improve upon the existing adaptive indexing algorithm [5, 6] in terms of query throughput. This improvement is achieved by minimizing synchronization among threads. There is however a fundamental difference between the algorithm **P-SC** of [5, 6, 10] and our algorithms. While the former assigns one thread per query (inter-query parallelism), the latter three assign many threads to a query (intra-query parallelism). In realistic environments, servers have several sockets, each consisting of many cores. Therefore, we do not see the assumption of assigning multiple threads to a query unrealistic — speedups seem to be fairly good already when assigning only four threads per query.

3. This work considers the basic workload already used multiple times in the literature [5, 6, 8, 10, 13, 14, 21]. This workload, if not too realistic, serves as a good initial case of study and helps to (easily) identify strengths and weaknesses of the tested algorithms. In this regard, only the work of [12, 21] has gone to a more realistic workload, which we are currently exploring: multi-selection-, multi-projections queries working on an entire table instead of only on a single index column. This in particular includes tuple-reconstruction, and is being considered on a larger multi-core machine.

Acknowledgement

We would like to thank Felix Halim and Stratos Idreos for kindly providing the source code for **P-SC** used in [5, 6]. Research partially supported by BMBF.

References

[1] Intel® Composer XE Suites. `http://software.intel.com/en-us/intel-composer-xe`.

[2] Intel® VTune™ Amplifier XE 2013. `http://software.intel.com/en-us/intel-vtune-amplifier-xe`.

[3] V. Alvarez, F. M. Schuhknecht, J. Dittrich, and S. Richter. Main memory adaptive indexing for multi-core systems. *Computing Research Repository (CoRR)*, April 2014. Available at `http://arxiv.org/abs/1404.2034`.

[4] C. Balkesen, G. Alonso, and M. Ozsu. Multi-core, main-memory joins: Sort vs. hash revisited. *Proceedings of the VLDB Endowment*, 7(1), 2013.

[5] G. Graefe, F. Halim, S. Idreos, H. Kuno, and S. Manegold. Concurrency control for adaptive indexing. *Proceedings of the VLDB Endowment*, 5(7):656–667, 2012.

[6] G. Graefe, F. Halim, S. Idreos, H. Kuno, S. Manegold, and B. Seeger. Transactional support for adaptive indexing. *The VLDB Journal*, pages 1–26, 2014.

[7] G. Graefe, S. Idreos, H. Kuno, and S. Manegold. Benchmarking adaptive indexing. In *Performance Evaluation, Measurement and Characterization of Complex Systems*, pages 169–184. Springer, 2011.

[8] F. Halim, S. Idreos, P. Karras, and R. H. Yap. Stochastic database cracking: Towards robust adaptive indexing in main-memory column-stores. *Proceedings of the VLDB Endowment*, 5(6):502–513, 2012.

[9] C. A. Hoare. Quicksort. *The Computer Journal*, 5(1):10–16, 1962.

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

[11] S. Idreos, M. L. Kersten, and S. Manegold. Updating a cracked database. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 413–424. ACM, 2007.

[12] S. Idreos, M. L. Kersten, and S. Manegold. Self-organizing tuple reconstruction in column-stores. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 297–308. ACM, 2009.

[13] S. Idreos, S. Manegold, H. Kuno, and G. Graefe. Merging what's cracked, cracking what's merged: adaptive indexing in main-memory column-stores. *Proceedings of the VLDB Endowment*, 4(9):586–597, 2011.

[14] M. L. Kersten and S. Manegold. Cracking the database store. In *CIDR*, pages 213–224, 2005.

[15] S. Lee, M. Jeon, D. Kim, and A. Sohn. Partitioned parallel radix sort. *Journal of Parallel and Distributed Computing*, 62(4):656–668, 2002.

[16] V. Leis, A. Kemper, and T. Neumann. The adaptive radix tree: Artful indexing for main-memory databases. In *ICDE*, 2013.

[17] A. Maus. Arl, a faster in-place, cache friendly sorting algorithm. *Norsk Informatik konferranse NIK*, 2002:85–95, November 2002.

[18] A. Maus. A full parallel radix sorting algorithm for multicore processors. *Norsk Informatik konferranse NIK*, 2011:37–48, November 2011.

[19] R. Pagh and F. Rodler. Cuckoo hashing. In *ESA 2001*, volume 2161, pages 121–133. Springer Berlin Heidelberg, 2001.

[20] L. Rashid, W. M. Hassanein, and M. A. Hammad. Analyzing and enhancing the parallel sort operation on multithreaded architectures. *The Journal of Supercomputing*, 53(2):293–312, 2010.

[21] F. M. Schuhknecht, A. Jindal, and J. Dittrich. The uncracked pieces in database cracking. *Proceedings of the VLDB Endowment*, 7(2), 2013.

[22] A. Stepanov and M. Lee. *The standard template library*, volume 1501. Hewlett Packard Laboratories 1501 Page Mill Road, Palo Alto, CA 94304, 1995.

APPENDIX

A. COMPLEMENTARY FIGURES

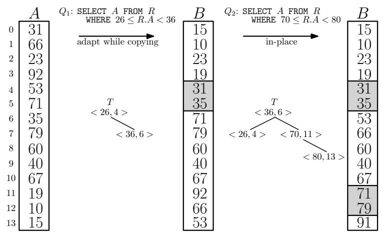

Figure 3: Adaptively indexing attribute A using two queries (Q_1, Q_2), and using standard cracking [10]. Column B is a copy of A on which future queries are performed. B is called the cracker column *in the literature. T is called the* cracker index, *and it tells future queries how B is currently partitioned. These two structures together, B and T, replace a traditional index. A node $\langle x, y \rangle$ in T tells that* all *elements of B strictly larger than x start at position y.*

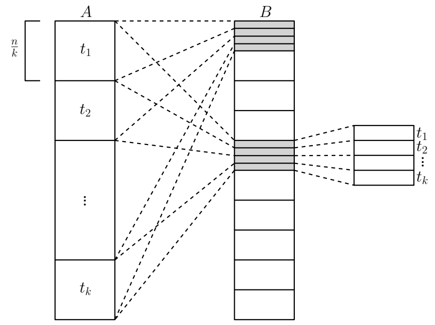

*Figure 4: In parallel coarse-granular index (**P-CGI**), column A is (symbolically) divided into k parts of $\frac{n}{k}$ elements each. Cracker column B is range-partitioned using r buckets. Every bucket of the range partition of B is also (symbolically) divided into k parts. Thread t_i, $1 \leq i \leq k$, writes to the i-th part of every bucket.*

B. SINGLE-THREADED ALGORITHMS

B.1 Standard cracking (SC)

This is the first adaptive indexing algorithm that appeared [10], and also the most popular one due to its simplicity and good performance. This algorithm is by now well-known, and it can be described as follows: Let $q = (q_l, q_h)$ be a query, with $q_l < q_h$. In its simplest version, in order to filter out the results, standard cracking performs two partition steps of quicksort [9] over B, each one taking q_l and q_h as their pivots respectively. The split location of each partition step is then inserted into the cracker index, using q_l and q_h as the keys, so that future queries can profit from the work previous queries have already done, see Figure 3. This incrementally improves query response times of standard cracking, as partitions become smaller over time.

The method of performing two partition steps is called 2x-crack-in-two in [10], but the authors also observed that both partition steps can be combined into a single one, which they call crack-in-three. Experiments shown in [21] suggest that 2x-crack-in-two performs better than crack-in-three most of the time, and when it does not, the difference is small. Thus, the used implementation of standard cracking performs only 2x-crack-in-two steps. Finally, it is important to know that the initialization cost of standard cracking is only that of creating the cracker column B, *i.e.*, copying column A onto B before performing the very first query.

B.2 Coarse-granular index (CGI)

In [21] a new adaptive indexing technique, therein called the *coarse-granular index*, was presented. This technique range-partitions B as a pre-processing step[3]. That is, the range of values of the keys of B is divided into $r \geq 2$ buckets[4], such that the first bucket contains the first $\frac{n}{r}$ largest values in B, the second bucket contains the second $\frac{n}{r}$ largest values, and so on. For the sake of explanation, we assume that n is divisible by r. In the experiments, however, we do not make this assumption. The keys inside each bucket are in any arbitrary order. Once B has been range-partitioned, the position where each bucket ends is inserted in a cracker index T, and standard cracking **SC** is run on B to answer the queries; taking T into consideration.

Range partitioning B gives standard cracking a huge speed-up over standard cracking alone, see Figure 1. As it turns out, this range partitioning can be done very fast; the elements of A can be range-partitioned while being copied to B. This requires only two passes over A. Also, as we pointed out in [21], **CGI** converges faster towards full index than **SC**, and is also more robust w.r.t. skewed queries. However, **CGI** incurs in relatively high initialization time w.r.t. **SC**, see Figure 1. There, it can be seen that **SC** can perform around 10 queries while **CGI** is still range-partitioning, after that, **CGI** pays off already.

B.3 Radix sort (RS)

The single-threaded sorting algorithm used in our experiments is the one presented in [17]. This algorithm was also used in [21], where it was reported to be very fast. This algorithm is a recursive (in place) **M**ost **S**ignificant **D**igit radix sort, called left radix sort in [17].

What the algorithm of [17] does to work in place is the following: The input gets (symbolically) divided into $r = 2^m$ buckets, where m is the number of bits of the sorting digits. Thus, r represents the number of different values of the sorting digits. Now, instead of exchanging the keys between two arrays according to the value of the keys in the sorting digit, as in a traditional radix sort, the algorithm of [17] works in permutation cycles à la Cuckoo [19]. That is, it places an element in its correct bucket, w.r.t. the value of its sorting digit. In doing so, it evicts another element which is then placed in its correct bucket, as so on and so forth. Eventually, an element gets placed in the bucket of the very first element that initiated this permutation cycle. Then, the next element that has not been moved yet starts another permutation cycle, and so on. Eventually, all elements get moved to their corresponding buckets. At that point, the algorithm recurs in *each* of the r bucket that the input was (symbolically) divided into. This ensures that all the work done previously is not destroyed. In our implementation we treat the numbers as four-digit numbers — radix-2^8 numbers. That is, $r = 256$. Thus, only four passes are necessary for sorting.

[3]If there is a bias in the distribution of the keys, an equi-depth partition could be used instead of a range partition; at the expense of more pre-processing time.

[4]In the core part of our experiments, as in the ones shown in [21], we set $r = 1024$, since that was the value for which the best performance was observed. When increasing the size of the input by a factor of ten we set $r = 8192$.

C. DISCUSSING THE SPEEDUPS

As we explained in Section 2, most of our algorithms are lock-free, and are highly parallelizable. Still, we can observe that for k threads, the speedups obtained are not always k-folded. To understand this phenomenon, we have profiled our algorithms with Intel Vtune Amplifier XE 2013 [2]. This tool allows us to gather all sorts of information about a running program, such as consumed memory bandwidth (GB/s), cpu utilization, cache misses, lock contention, among many others. We have carefully analyzed *all* the considered algorithms for *all* considered number of threads. Here, however, we do not discuss *each* single algorithm in each one of the considered configurations, we will rather pick a representative subset of all configurations and give the explanations for them. Finally, we would like to point out that profiling runs of the algorithms are slower than the regular runs, as profiling code is added for this purpose.

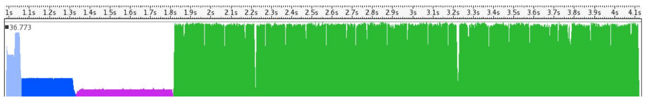

*Figure 5: The highest rate at which memory is consumed in **P-CRS** is 36.773 GB/s. The colors indicate the different stages of the algorithm (from left to right): (1) Histogram creation, (2) Materializing the range-partitioning, (3) Radix sorting, and (4) Query answering.*

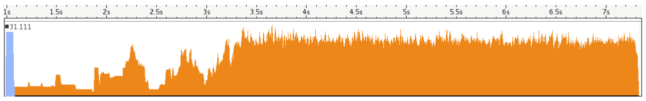

*Figure 6: The highest rate at which memory is consumed in **P-SC** is 31.111 GB/s. The colors indicate the different stages of the algorithm (from left to right): (1) Copying the input to the cracker column, (2) Cracking and query answering. In the very beginning, up to around the 3-seconds mark, waiting times hinder scalability. After that threshold there are enough partitions in the cracker column so that all threads can work mostly concurrently.*

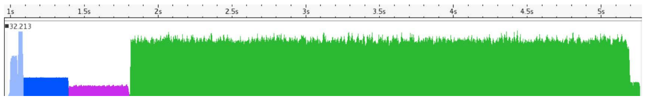

*Figure 7: The highest rate at which memory is consumed in **P-RPRS** is 32.213 GB/s. The colors indicate the different stages of the algorithm (from left to right): (1) Histogram creation, (2) Materializing the range-partitioning, (3) Radix sorting, and (4) Query answering.*

C.1 Chunked algorithms

All chunked algorithms, *i.e.*, **P-CSC**, **P-CCGI**, and **P-CRS**, share a large portion of code, and their implementations, as we already pointed out, are NUMA-aware. That is, we give the system the strong hint that each thread should work on its chunk in its own NUMA region by pinning the working threads to physical cores of the system. For example, for four threads, the first two threads (1, 2) will be pinned in one socket and the other two (3, 4) will be pinned in the other socket. For eight threads, the first four threads (1, 2, 3, 4) will be pinned in a socket, and the other four (5, 6, 7, 8) will be pinned in the other socket. We also avoid false-sharing among threads by aligning the working data of a thread with the boundary of the cache lines, so that for any two threads, the working data is found on different cache lines, and thus one thread does not invalidate the cache line of any other thread.

Now, let us consider the eight-threaded version of **P-CRS**, see Table 4, and let us focus on total running time. There, we see that **P-CRS** achieves a speedup of only $5.413\times$, when ideally it should achieve a speedup of $8\times$. How this algorithm utilizes the memory bandwidth of the system can be seen in Figure 5, as reported by Vtune. The highest rate at which memory was consumed by **P-CRS** is 36.773 GB/s. We measured the combined memory bandwidth of our system to be around 40 GB/s, that is, 20 GB/s per socket. So we can conclude that memory bandwidth is not the bottleneck of the algorithm. The real bottleneck is the following. In order to obtain $8\times$ speedup, we should obtain $4\times$ speedup from each socket (four threads per socket). The algorithm is highly parallelizable. However, we measured the speedup of each socket to be only $2.9\times$. We measured this by scheduling a four-threaded version of **P-CRS** in one socket only. The discrepancy between the expected $4\times$ speedup and the obtained $2.9\times$ speedup comes from the fact that the four threads are now sharing one L3 cache, the one corresponding to the socket. If this L3 cache was four times bigger, the configuration would be equivalent to the ideal configuration of having four sockets, each one having its own large L3 cache, and thus the processors would benefit from the larger cache sizes. We simulate this experiment with a two-threaded version of **P-CRS**, as we do not have a four-socket system. When this version is scheduled on one socket only, the number of cache misses[5] is on average (over ten runs of the algorithm) $231,060,000$ for this socket, as reported by Vtune. When this version is scheduled on two different sockets, the number of cache misses per socket is on average $102,500,000$. Overall there are $26,060,000$ ($\approx 12\%$) more cache misses on the former configuration than in the latter, and this of course affects the scalability, going from $2\times$ speedup in the latter configuration, to $1.8\times$ speedup in the former.

This explains the $2.9\times$ speedup of a single socket executing four threads. This also means that the achievable speedup by fully using both sockets (4 threads per socket) is about $2 \cdot 2.9\times = 5.8\times$. Yet, we are obtaining $5.4\times$, as reported in Table 4. This last discrepancy is a NUMA effect. Even though our implementations are fully NUMA-aware, at certain times during the execution of the algorithm, the system allocates temporary variables in different NUMA regions — even though all threads are pinned to physical cores, and they should instantiate their data in their NUMA region. That is, every once in a while there are local cache misses that are served by data from a different NUMA region — we noticed that roughly 1% of the total number of caches misses come from a different NUMA region, this all can be seen in Vtune. It is well-known that bringing data from a different NUMA region is more expensive than accessing the same NUMA region. Thus this slightly slows down the algorithm.

We performed the very same analysis on **P-CSC** and **P-CCGI** and we noticed the same effect — which was to be expected since all those methods share a large portion of code. The overall numbers are different nevertheless, and for brevity we will not show them here.

With these arguments we are able to explain the speedups of *all* chunked methods: **P-CSC**, **P-CCGI**, and **P-CRS**. We will now

[5]The counters quantified by Vtune are `OFFCORE_RESPONSE.DEMAND_DATA_RD.LLC_MISS.LOCAL_DRAM`, `OFFCORE_RESPONSE.DEMAND_DATA_RD.LLC_MISS.REMOTE_DRAM`

proceed to analyze **P-SC** and **P-CGI**, which use locks to ensure consistency in the data.

C.2 Algorithms that are not lock-free

Among all algorithms, two are fundamentally different to the rest. These two algorithms are **P-SC** [5, 6] and **P-CGI**, and the difference lies in that they both use locks. As a first reference, Figure 6 shows how the eight-threaded version of **P-SC** utilizes the memory bandwidth of the system. The conclusion is that memory bandwidth is again not hindering the scalability of the algorithms. The real bottleneck is the time *each* thread has to wait while another thread is blocking the resources. Using Vtune we can obtain actual numbers about this. We again take the eight-threaded version of **P-SC** as a reference. Table 5 shows the total accumulated waiting time of **P-SC** among all threads. Adding up those numbers and dividing by the number of working threads (eight in this case) we obtain that the real waiting time **P-SC** incurs in is 2.105 seconds on average. Observe in Figure 6 that **P-SC** starts achieving its full potential after the 3-seconds mark. That is, from the reported total execution time (7.3 secs) of the eight-threaded version of **P-SC** shown in Figure 6, more than one-third is just waiting time.

Mutex	Wait time (s)
Piece lock	11.671
Cracker index lock	5.169
Total	16.84
Average (Total by 8)	2.105

*Table 5: Waiting times incurred by the eight-threaded version of **P-SC**. Adding up and dividing by the number of threads we obtain that* 2.105 *seconds out of the total execution time of **P-SC** are waiting time.*

As a quick reference, **P-CGI** builds a range-partitioning before **P-SC** kicks in. This range partitioning alleviates the waiting time, as now a thread does not have to block the whole column when performing the very first crack. As given by Vtune, the total waiting time of **P-CGI** drops to about 1 second, from the 2.1 secs of **P-SC**. Thus, the range-partitioning cuts in half the waiting times.

The correctness of **P-SC** and **P-CGI** depends on the use of locks. Therefore, waiting times will always hinder the scalability of these algorithms.

C.3 Sorting algorithms

Let us again for the sake of brevity focus only on the eight-threaded version of **P-RPRS**. How this algorithm utilizes the memory bandwidth of the system is shown in Figure 7. From that figure we can clearly see that the sorting stage of **P-RPRS** scales essentially linearly w.r.t. the number of working threads. So we will pass onto discussing the speedups achieved by the query part of the algorithms.

As we already mentioned, the query part of *all* sorting algorithms performs a binary search to filter the elements that belong to the result, and then it (sequentially) aggregates over all those elements. That is, there are no hidden overheads in the code executing the queries. The poor scaling of the query part is due to NUMA effects. In contrast to the chunked algorithms, the sorted column in **P-RPRS** is shared by both NUMA regions. Queries are served as they come, and to answer a query, this query is assigned to a free thread. This thread then jumps to the NUMA region the query belongs to and filters and aggregates over the corresponding elements. Using Vtune we can quantify the cache misses that are served by the same NUMA region, and the cache misses that are served by a different NUMA region — and thus also being served slower. For the former, as reported by Vtune, we obtain $175,520,000$ cache misses on average (over ten runs of **P-RPRS**). For the latter we obtain $197,700,000$ on average. So there are 12% more remote cache misses as local cache misses, and overall 53% of the total number of caches misses are remote. This strongly contrast against the chunked algorithms for example, where the number of remote cache misses is negligible, roughly 1% of the total.

The only way we could get rid of such a high number of NUMA effects is by designing **P-CRS**. There, we sacrifice the fully sorted column — although, if later needed, the sorted chunks can be merged using NUMA-aware merge procedures like the ones discussed in [4]. This sacrifice, nevertheless, comes with the speedups our system allows, as we already discussed.

D. ADDITIONAL EXPERIMENTS

D.1 Scaling input size by a factor of ten

We tested the algorithms with an input size of one billion, *i.e.*, ten times larger than the input size of the previous experiments. The scalability of the algorithms is shown in Table 6. Times are again given in seconds. The shown factors are w.r.t. the times shown in Tables 2 to 4 for the 8-threaded algorithms. This time nonetheless, all shown factors represent slowdowns due to scaling of the input size. Other than input size, the workload is as described in Section 1.3 — in particular, selectivity stays at 1%.

Algorithm	**Initial** time	Slowdown	**Total** time	Slowdown
P-SC	5.328	9.087×	59.69	10.02×
P-CSC	1.156	9.816×	31.71	9.482×
P-CGI	5.088	12.62×	40.4	9.298×
P-CCGI	3.414	9.934×	26.64	9.29×
P-RPRS	9.093	11.59×	42.75	10.27×
P-CRS	7.418	9.307×	27.43	9.329×

Table 6: Scale factors of the algorithms when increasing input size by a factor of ten.

We observe that all algorithms scale gracefully as the input size increases. The slowdown is essentially linear, *i.e.*, we observe times that are essentially ten times slower, although sorting algorithms suffer the most.

D.2 Different tuple configuration

Until now we have assumed that the entries in the cracker column B are pairs $(key, rowID)$ of 4-byte positive integers each. While this assumption seems in general reasonable, it could also be limiting in some cases. Thus, here we show how the algorithms perform when used with pairs $(key, rowID)$ of 8-byte positive integers generated uniformly at random over the interval $\left[0, 2^{64}\right)$. The results of these experiments can be seen in Table 7. Times are shown in seconds, and the shown (slowdown) factors are w.r.t. the times shown in Tables 2 to 4 for the 8-threaded algorithms.

Other than tuple configuration, the workload is as described in Section 1.3. That is, there are 100 million entries, and $10,000$ queries with selectivity 1% are performed. Queries are this time, of course, 8-byte pairs of positive integers generated uniformly at random (respecting selectivity). For sorting, we considered the numbers still as radix-2^8 numbers, *i.e.*, radix sort requires now twice as many passes to fully sort the numbers, *i.e.*, eight passes.

As we can see, there is a generalized two-fold slowdown in all algorithms w.r.t. total execution time; as expected for a two-fold increment in entry size. In this regard, **P-CSC** is being affected

Algorithm	Initial time	Slowdown	Total time	Slowdown
P-SC	0.7122	1.215×	9.887	1.66×
P-CSC	0.2032	1.726×	5.426	1.622×
P-CGI	0.6675	1.656×	7.958	1.831×
P-CCGI	0.5841	1.7×	4.914	1.714×
P-RPRS	1.188	1.514×	7.859	1.888×
P-CRS	1.078	1.352×	5.117	1.74×

Table 7: Scale factors of the algorithms when increasing entry size from 4*-byte pairs to* 8*-byte pairs.*

the least, and **P-RPRS** being affected the most. Initial response time seems to scale more gracefully for all algorithms. With respect to the shown slowdowns of the sorting algorithms we would like to point out that they perform much more work than all other algorithms. Sorting algorithms have to not only shuffle data that is twice as big, which is where the slowdown of all other algorithms stems from, but also they have to do twice as many passes to sort the numbers. Thus, we think that the scale factors of sorting algorithms should be considered exceptionally good.

D.3 Skewed query access pattern

So far we have seen how the algorithms scale w.r.t. to input size and different tuple configurations. Now, we show how the algorithms perform when the queries are skewed, *i.e.*, they are no longer uniformly distributed over the range of values the input keys fall into. For these experiments we have generated the queries from a normal distribution with mean 2^{31} and standard deviation 2^{28}. That is, queries are now tightly concentrated around 2^{31}.

As originally stated, we generated $10,000$ queries with selectivity 1%, and run the algorithms over 100 million entries of 4-byte positive integers generated uniformly at random. The results of the experiments can be seen in Table 8. Times are shown in seconds, and the shown factors are w.r.t. the times shown in Tables 2 to 4 for the 8-threaded algorithms. A factor larger than one represents a slowdown and a factor smaller than one represents a speedup.

Algorithm	Initial time	Factor	Total time	Factor
P-SC	0.7489	1.277×	6.928	1.163×
P-CSC	0.1286	1.092×	2.81	0.8402×
P-CGI	0.4317	1.071×	6.7	1.542×
P-CCGI	0.342	0.9953×	2.609	0.9101×
P-RPRS	0.7852	1.001×	4.161	0.9996×
P-CRS	0.7989	1.002×	2.85	0.9694×

Table 8: Scale factors of the algorithms when queries are skewed.

We observe that, w.r.t. initial response time, **P-SC** was affected the most. The explanation for that is the following. As now the queries are tightly concentrated around a certain area, most threads try to access the same area, but this incurs into a great deal of locking. One thread locks the region it partitions while the others are forced to wait. Once done, this thread now has to aggregate over a continuous region, but this region is most probably being locked by another thread partitioning it. So, as it seems, the very first query takes longer to finish just because more threads are accessing the same region. This is the effect of skewness in the query access pattern. **P-CGI** is also being hit due to the same argument, since it uses **P-SC** after range-partitioning the input. Moreover, as *all* queries are tightly concentrated around the same value, there is essentially no difference between **P-SC** and **P-CCGI**, as they lock the same areas. This of course affects **P-CCGI** more than **P-SC**. All other algorithms are completely oblivious to query access patterns. Thus, the shown factors for **P-CSC**, **P-CCGI**, and **P-RPRS** are simply variations obtained from different measurements.

D.4 Varying selectivity

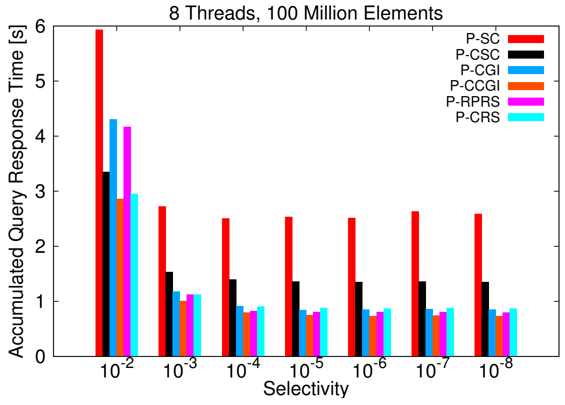

Figure 8: Effect of varying selectivity from 10^{-2} *(*1% *of the data) to* 10^{-8} *(point query). Ten thousand queries are performed each time.*

So far, all experiments considered a selectivity of 1%, as this is what is usually used in the literature. Nevertheless, a selectivity of 1% can be deemed to be too low for actually using an index structure at all. Also, the querying part of a query with a low selectivity might overshadow the actual index access. Thus, in this section, we present the results of an experiment where we vary the selectivity from the usual 1% down to highly selective point queries in logarithmic steps. The total number of queries stays at $10,000$ each time. In Figure 8 we show the accumulated query response times for the individual methods under the variation of the selectivity.

Several observations can be made in Figure 8. First of all, a selectivity higher than 10^{-4} (0.01%) does not affect the overall runtime anymore, as the querying part becomes negligible. At that point, the runtime of all methods is determined only by the index creation/maintenance. We can also see that for higher selectivities, the relative order of the methods in terms of performance changes. For a selectivity of 10^{-2}, **P-CRS** performs much better than **P-RPRS**. This is no longer the case for higher selectivities, in fact, **P-RPRS** suddenly performs at least as good as **P-CRS** (or even better) from 10^{-3} on. The reason for this is that the advantage of **P-CRS** lies in the querying part, which can be performed locally in the chunks due to NUMA-awareness, while **P-RPRS** suffers from remote accesses. When selectivity increases, index access overshadows aggregation costs. In **P-CRS** all threads work towards answering *every* query, while in the other methods *every* thread answers a different query. That is, the former performs 8 times more index accesses than the latter. Furthermore, we can also observe that **P-CGI** benefits from high selectivities. From 10^{-3} on, its performance is very close to that of the best remaining methods. This improvement results from the fact that from 10^{-3} on, a query fits into a partition of the range-partitioning. That is, a thread must lock, in the beginning, at most two partitions of the range-partitioning. The likelihood that any two threads require the same partition is very small. Thus, **P-CGI** becomes mostly lock-free. Overall, we can see that **P-CCGI** shows the best accumulated query response time for all tested selectivities, albeit negligible differences for high-selectivity queries.