Holistic Indexing

Eleni Petraki

Holistic Indexing

Academisch Proefschrift

ter verkrijging van de graad van doctor
aan de Universiteit van Amsterdam,
op gezag van de Rector Magnificus
prof. ...
ten overstaan van een door het
college voor promoties ingestelde commissie
in het openbaar te verdedigen
in ...
op ...

door
Eleni Petraki
geboren te Karditsa, Griekenland

Promotiecommissie

Promotor:	Prof. dr. M.L. Kersten
Copromotor:	Prof. dr. S. Manegold
	Asst. Prof. dr. S. Idreos
Overige Leden:	Prof. dr. ...
	Prof. dr. ...
	Prof. dr. ...
	Prof. dr. ...
	Prof. dr. ...

Faculteit der Natuurwetenschapen, Wiskunde en Informatica

The research reported in this thesis has been partially carried out at CWI, the Dutch National Research Laboratory for Mathematics and Computer Science, within the theme Database Architectures and Information Access, a subdivision of the research cluster Information Systems.

The research reported in this thesis has been partially carried out as part of the continuous research and development of the MonetDB open-source database management system.

SIKS Dissertation Series No-nnnn-nn.
The research reported in this thesis has been carried out under the auspices of SIKS, the Dutch Research School for Information and Knowledge Systems.

ISBN nn nnnn nnn n

Dedicated to my parents, Sofia and Theodoros,
and my husband, Evangelos.

"The time will come when diligent research over long periods will bring to light things which now lie hidden. A single lifetime, even though entirely devoted to the sky, would not be enough for the investigation of so vast a subject... And so this knowledge will be unfolded only through long successive ages. There will come a time when our descendants will be amazed that we did not know things that are so plain to them... Many discoveries are reserved for ages still to come, when memory of us will have been effaced."

Seneca, Natural Questions

Contents

Chapter 1

Introduction

Chapter 2

Related Work and Background

Intro

2.1 Row-oriented Storage and Data Access

2.2 Column-stores

2.3 Indices

2.4 Offline Indexing

Offline indexing is the earliest approach on self-tuning database systems. Nowadays, all major database products offer auto-tuning tools [**?** **?** **?**] to automate the database physical design. Auto-tuning tools mainly rely on a "what-if analysis" [**?**] and close interaction with the optimizer [**?**] to decide which indices are potentially more useful for a given workload.

Offline indexing requires heavy involvement of a database administrator (DBA). Specifically, a DBA invokes the tool and provides its input, i.e., a representative workload. The tool analyzes the given workload and recommends an appropriate physical design. However, the DBA is the one that decides which of the changes in the physical design should be applied. The main limitation of offline indexing appears when the workload cannot be predicted and/or there is not enough idle time to invest in the offline analysis and the physical design implementation.

2.5 Online Indexing

Online indexing addresses the limitation of offline indexing. Instead of making all decisions a priori, the system continuously monitors the workload and the physical design is periodically reevaluated. System COLT [**?**] was one of the first online indexing approaches. COLT continuously monitors the workload and periodically in specific epochs, i.e., every N queries, it reconsiders the physical design. The recommended physical design might demand creation of new indices or dropping of old ones. COLT requires many calls to the optimizer to obtain cost estimations. A "lighter" approach, i.e., requiring less calls to the optimizer, was proposed later [**?**]. Soft indices [**?**] extended the previous online approaches by building full indices on-the-fly concurrently with queries on the same data, sharing the scan operator.

The main limitation of online indexing is that reorganization of the physical design can be a costly action that a) requires a significant amount of time to complete and b) requires a lot of resources. This means that online indexing is appropriate mainly for moderately dynamic workloads where the query patterns do not change very frequently. Otherwise, it may be that by the time we finish adapting the physical design, the workload has changed again leading to a suboptimal performance.

2.6 Adaptive Indexing

Adaptive indexing is the latest and the most lightweight approach in self-tuning databases. Adaptive indexing addresses the limitations of offline and online indexing for dynamic workloads; it instantly adjusts to workload changes by building or refining indices partially and incrementally as part of query processing. By reacting to every single query with lightweight actions, adaptive indexing manages to instantly adapt to a changing workload. As more queries arrive, the more the indices are refined and the more performance improves. Adaptive indexing has been studied in the context of main-memory column-stores [**?** **?**], Hadoop [**?**] as well as for improving more traditional disk-based settings [**?**]. It has been shown to work for many core database architecture issues such as updates [**?**], multi-attribute queries [**?**], concurrency control [**?** **?** **?**], partition-merge-like logic [**?** **?**]. In addition, [**?**] shows how to benchmark adaptive indexing techniques, while stochastic database cracking [**?**] shows how to be robust on various workloads and [**?**] shows how adaptive indexing can apply to key columns. Finally, recent work on parallel adaptive indexing studies CPU-efficient implementations and proposes algorithms to exploit multi-cores [**?** **?**].

The main limitation of adaptive indexing is that it works only during query processing. In this way, the only opportunity to improve the physical design is only when queries arrive.

Recently, adaptive indexing concepts have been extended to provide adaptive indexes for time series data [**?**] as well as using incoming queries for more broad storage layout decisions, i.e., reorganizing base data (columns/rows) according to incoming query requests [**?**], or even about which data should be loaded [**?**]. In addition, adaptive indexing ideas have been used to design new generation data exploration tools such as touch-based data systems [**? ?**].

2.7 Database Systems for the Multi-core Era

Modern hardware offers opportunities for high parallelism; a single machine may be equipped with chip multiprocessors, which contain multiple cores with support for multiple context threads. Recent research focuses on exploiting parallelism opportunities by a) processing multiple queries concurrently, and b) by parallelizing tasks in the critical path during query processing [**? ? ? ?**]. Sorting is one of the most important database tasks (and a core component of adaptive indexing in column-stores) that can be highly-parallelized using modern hardware advances [**? ? ? ?**].

2.8 The MonetDB System

2.9 Summary

Chapter 3

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

Database Cracking is an appealingly simple 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 $10,000) servers.

3.1 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.

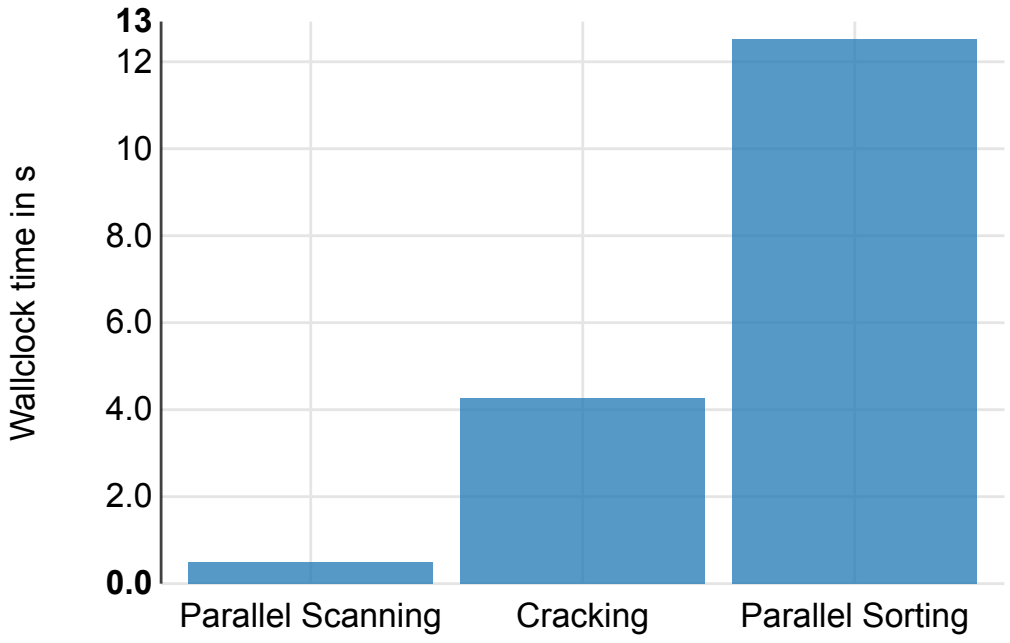


FIGURE 3.1: Costs of Database Operations

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.

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 [**?**], however, *Cracking* appears increasingly unattractive. To illustrate this, Figure 3.1 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* [**?**]. 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.

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

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.

The rest of the paper is structured as follows: In Section **??**, we provide an overview of related work as well as necessary background knowledge on the optimization techniques we applied. In Section 3.3 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 3.4, and our parallel implementations in Section 3.5. We evaluate these algorithms on a range of different hardware platforms in Section 3.6 and conclude in Section **??**.

3.2 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 [**?**]. 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) [**?**]. Thus, care has to be taken to ensure that there are sufficiently many independent instructions [**? ?**].

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 [**?**], while [**? ?**] 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 [**?**] and join algorithms [**? ?**].

Indexing Techniques

In the majority of automated index tuning approaches, index tuning is clearly distinct from query processing. Offline indexing approaches [**? ? ?**] analyze a given workload and select/create the necessary indexes before the workload enters the system, whereas

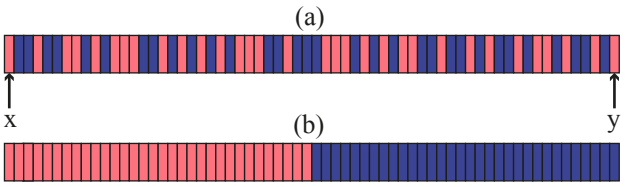


FIGURE 3.2: Original Cracking (single-threaded)

online indexing approaches [**?** **?** **?**] 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.

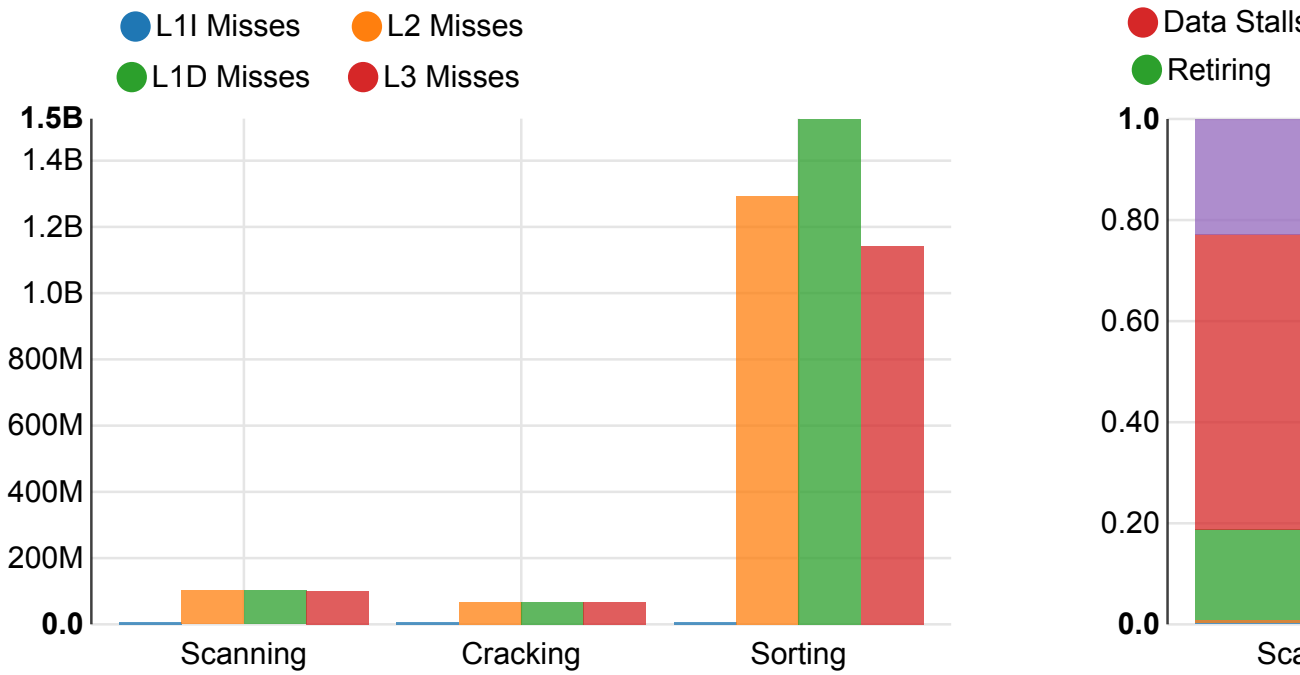
Adaptive indexing [**?** **?** **?**] is a recent, lightweight approach to self-tuning databases: data reorganization is integrated with query processing. *Database Cracking* [**?**] is an implementation of the adaptive indexing concept. Database Cracking initializes a partial index for an attribute 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 [**?**]) 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 [**?**], 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.

3.3 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 3.2. Figure 3.2(a) depicts an uncracked piece. Red indicates 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



(A) Cache Misses

FIGURE 3.3: Cost Breakdown of Database Operations

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 3.2(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 3.1).

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 3.3a, 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 [**?**]. The breakdown in Figure 3.3b 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*.

3.4 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 [**?**].

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.

[2] In this normalized plot, equal height bars indicate an absolute difference of almost factor 10, Figure 3.1 providing the scale

[3] or, more accurately microop execution slots

To achieve this, we developed a branch-free cracking implementation based on predication (illustrated in Figure 3.4). 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 3.4a), 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 3.4b), 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 3.4c) to advance the output cursors. One cursor is advanced by the value of *cmp*, the other by $1 - cmp$. This ensures that only one of the cursors is advanced. In the last phase (see Figure 3.4), 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 3.6).

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 3.5). 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, 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 4.4). This has the advantage of producing tight, CPU-efficient loops in the (expensive) partitioning phase while allowing bulk-backups of input values.

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

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 4.4, three backup slots are sufficient to maintain enough "slack space" for safe writing.

SIMD

Figure 3.5 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 3.6), we found that this algorithm generally performs worse than the previously discussed implementations. We include the description primarily for completeness sake.

3.5 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 *Cracking* algorithm. Finally, during the merge phase, a single

thread swaps wrongly located blocks of values into their final position. Figure 3.7 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 3.7(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 3.8(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 3.8(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 3.8 only two values are located in wrong positions, while in Figure 3.7, 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 3.4 are orthogonal to the thread-level parallelism presented above. Consequently, we can combine both techniques, hoping to accumulate their benefits. We focus on vectorization

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 3.1: Hardware Setup

as this proved to yield better single-threaded CPU efficiency than predication or SIMD (cf., Sections 3.4 and 3.6).

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.

3.6 Evaluation

Setup

We evaluated the presented implementations[5] on four different machines (see Table 3.1): 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 5GB 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 3.9): 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

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

[6] Including virtual cores (Hyperthreading)

(large intermediates lead to expensive swapping on 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 3.11). 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.

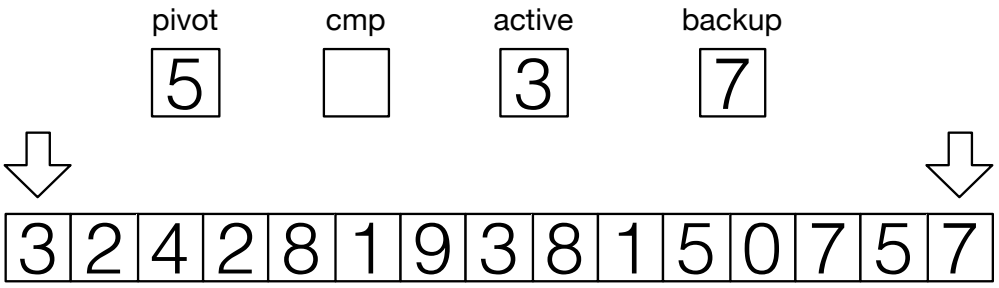
Multi-threaded

The results of our multi-threaded experiments are displayed in Figure 3.10. 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 3.1).

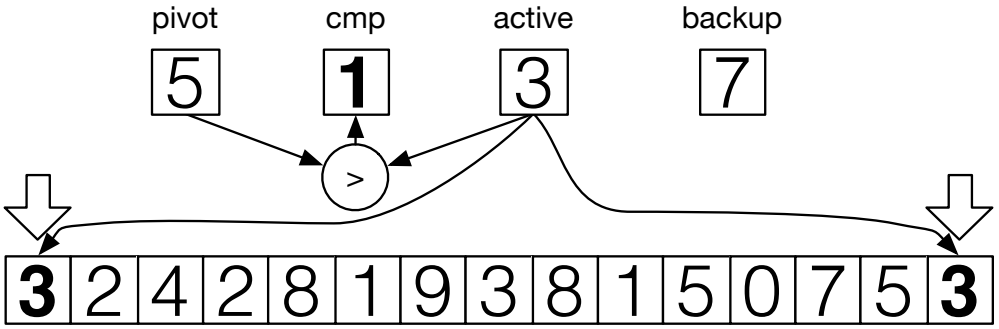
For reference, we include the best single-threaded implementation (Vectorized) in the chart. We observe a significant speedup in almost all cases. Naturally, the *Refined Partition & Merge* implementation performs better than its plain counterpart. In addition, both implementations achieve a performance 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.

3.7 Summary

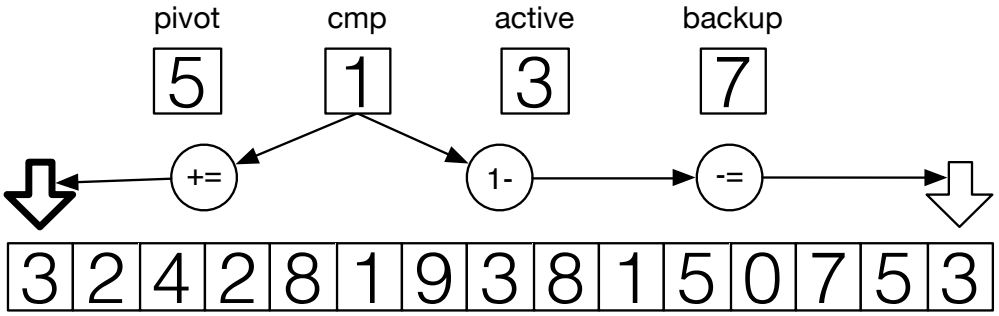
CPU-efficient implementation of even simple algorithms is hard: while common knowledge in many fields of computer science, this insight is still not properly appreciated in the field of data management. In this paper, we conducted an in-depth study of such a supposedly simple algorithm: pivoted partitioning. 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 in 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



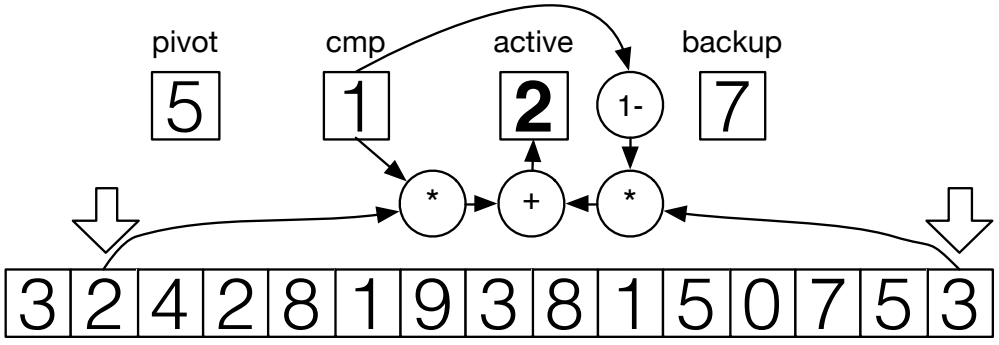
(A) Consistent State



(B) Compare & Write Phase



(C) Advance Cursor Phase



(D) Backup Phase

FIGURE 3.4: Predicated Cracking

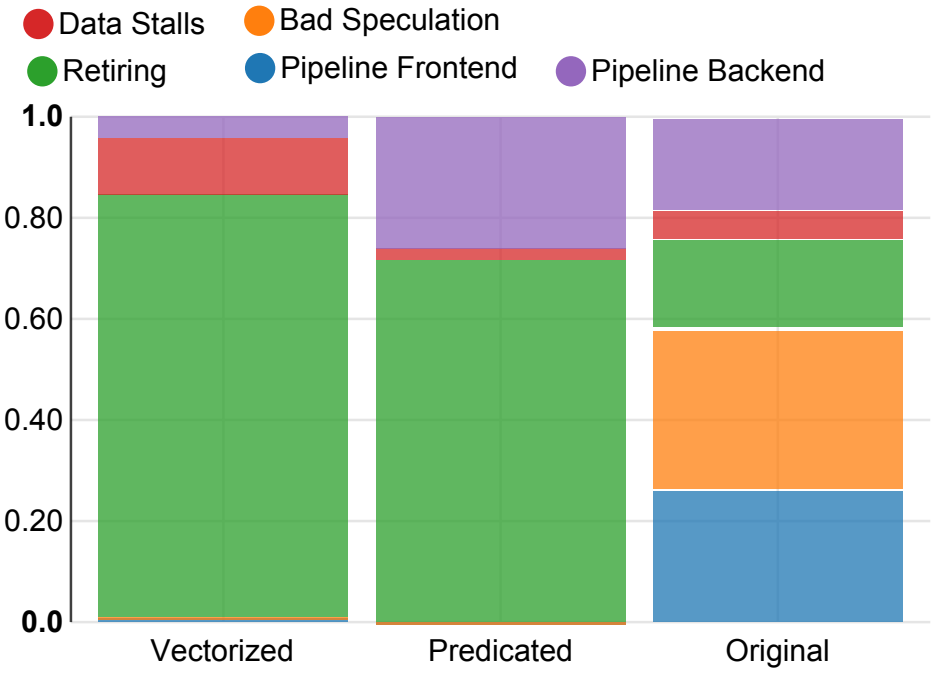


FIGURE 3.5: Cost breakdown of single-threaded implementations

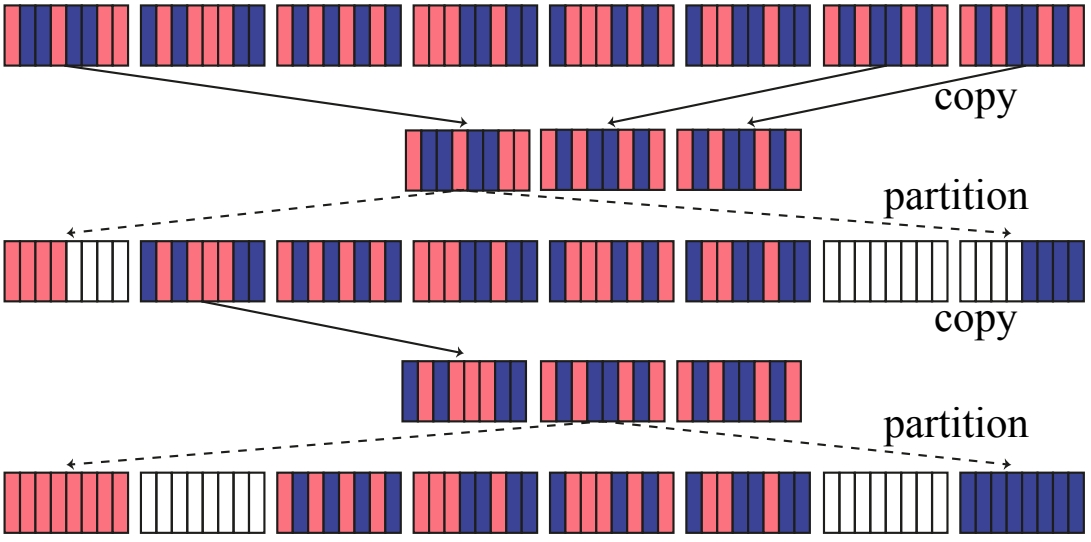


FIGURE 3.6: Vectorized Cracking

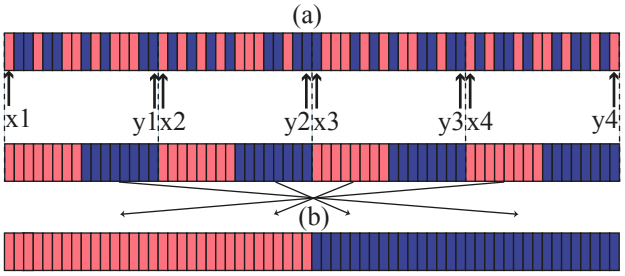


FIGURE 3.7: Simple Partition & Merge (multi-threaded)

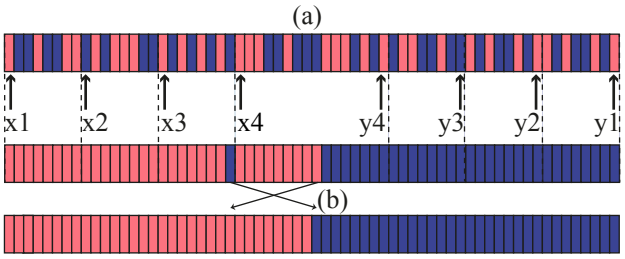
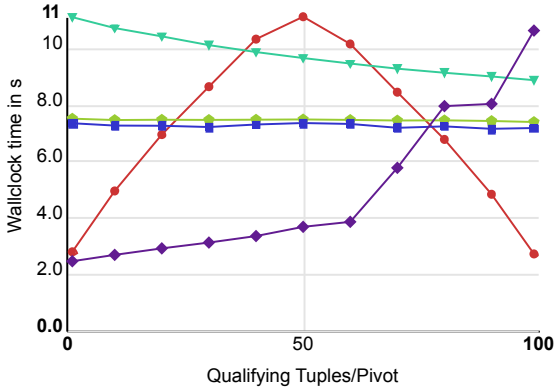
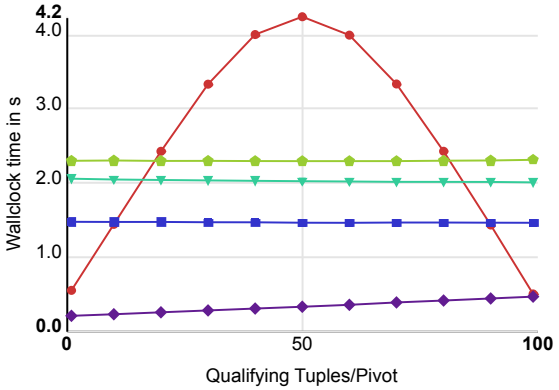


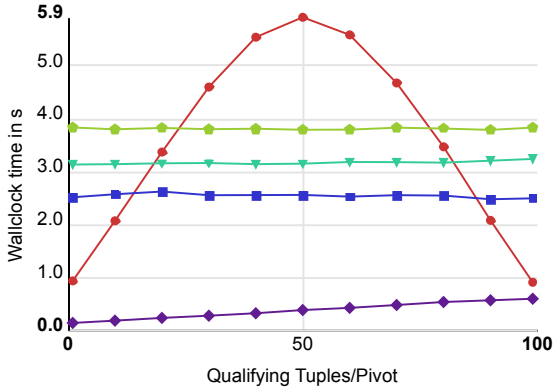
FIGURE 3.8: Refined Partition & Merge (multi-threaded)



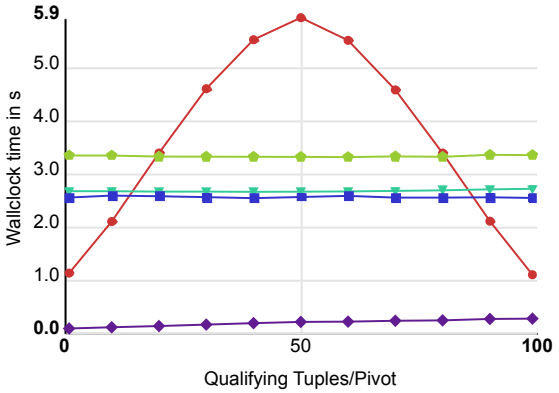
(A) Desktop



(B) Workstation

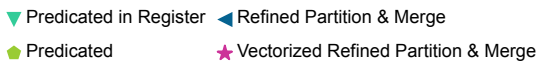


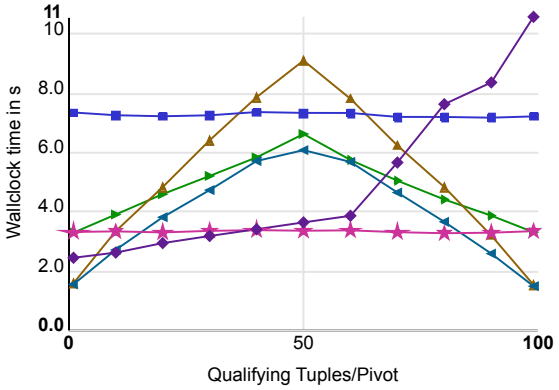
(C) Server



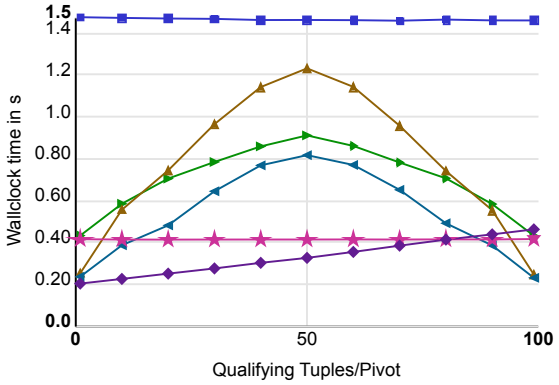
(D) High-End Server

FIGURE 3.9: Single Threaded Performance

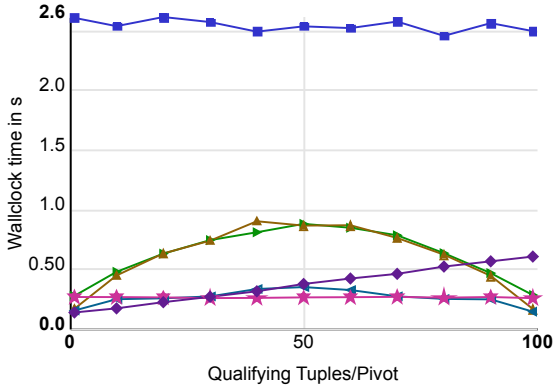

Predicated in Register
Refined Partition & Merge
Predicated
Vectorized Refined Partition & Merge



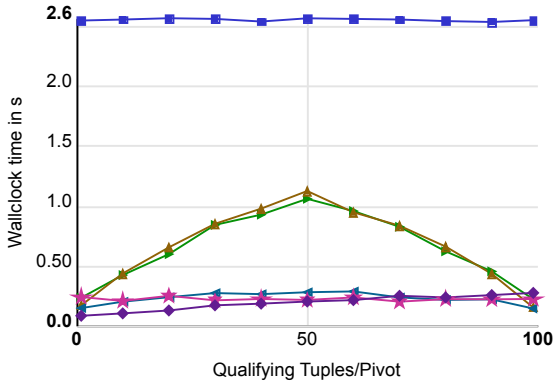
(A) Desktop



(B) Workstation



(C) Server



(D) High-End Server

FIGURE 3.10: Multi Threaded Performance

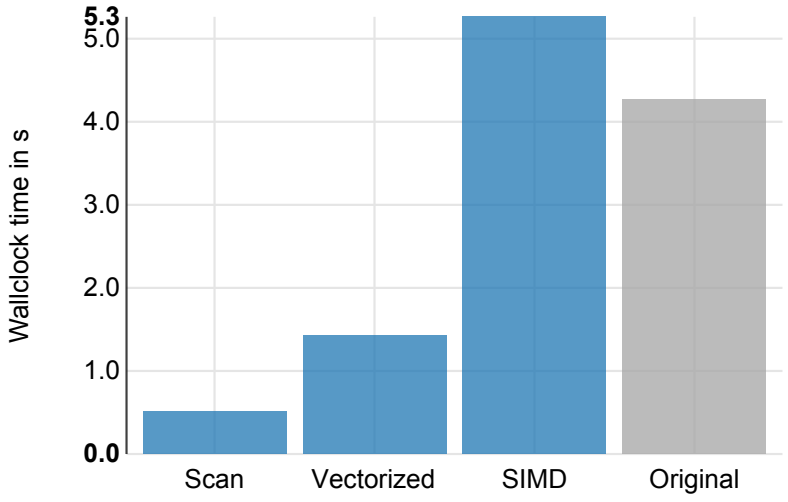


FIGURE 3.11: SIMD Processing Performance at 50% Selectivity

Chapter 4

Holistic Indexing in Main-memory Column-stores

Great database systems performance relies heavily on index tuning, i.e., creating and utilizing the best indices depending on the workload. However, the complexity of the index tuning process has dramatically increased in recent years due to ad-hoc workloads and shortage of time and system resources to invest in tuning.

This paper introduces *holistic indexing*, a new approach to automated index tuning in dynamic environments. Holistic indexing requires zero set-up and tuning effort, relying on adaptive index creation as a side-effect of query processing. Indices are created incrementally and partially; they are continuously refined as we process more and more queries. Holistic indexing takes the state-of-the-art adaptive indexing ideas a big step further by introducing the notion of a system which never stops refining the index space, taking educated decisions about which index we should incrementally refine next based on continuous knowledge acquisition about the running workload and resource utilization. When the system detects idle CPU cycles, it utilizes those extra cycles by refining the adaptive indices which are most likely to bring a benefit for future queries. Such idle CPU cycles occur when the system cannot exploit all available cores up to 100%, i.e., either because the workload is not enough to saturate the CPUs or because the current tasks performed for query processing are not easy to parallelize to the point where all available CPU power is exploited.

In this paper, we present the design of holistic indexing for column-oriented database architectures and we discuss a detailed analysis against parallel versions of state-of-the-art indexing and adaptive indexing approaches. Holistic indexing is implemented in an open-source column-store DBMS. Our detailed experiments on both synthetic and standard benchmarks (TPC-H) and workloads (SkyServer) demonstrate that holistic

Indexing	Statistical analysis	Exploitation of idle resources *before* query processing	Exploitation of idle resources *during* query processing	Index materializatio
Offline	√	√	×	full
Online	√	×	√	full
Adaptive	×	×	×	partial
Holistic	√	√	√	partial

TABLE 4.1: Qualitative difference among offline, online, adaptive and holistic indexing.

indexing brings significant performance gains by being able to continuously refine the physical design in parallel to query processing, exploiting any idle CPU resources.

4.1 Introduction

The big data era is causing the research community to rethink fundamental issues in the design of database systems towards more usable systems [**?**] that can access data better and faster [**? ? ? ?**], that can better exploit modern hardware and opportunities for massive parallelization [**?**] that can support efficient processing of OLTP and/or OLAP queries [**? ? ? ?**].

The Physical Design Problem. Physical design, and in particular proper index selection, is a predominant factor for the performance of database systems and has only become more crucial in the big data era. With new dynamic and exploratory environments, physical design becomes especially hard given the instability of workloads and the continuous stream of big data; a single physical design choice is not necessarily correct or useful for long stretches of time, while at the same time workload knowledge is scarce given the exploratory user behavior.

State-of-the-Art. In database applications, where "the future is known", physical design is assigned to database administrators who may also be assisted by auto-tuning tools [**? ? ?**]. Still though, a significant amount of human intervention is necessary and everything needs to happen offline. Thus, offline indexing can be applied with good results only on applications where there is enough workload knowledge and idle time to prepare the physical design appropriately before queries arrive.

Unfortunately, for many modern applications "the future is unknown", e.g., in scientific databases, social networks, web logs, etc. In particular, the query processing patterns follow an exploratory behavior, which changes so arbitrarily that it cannot be predicted. Such environments cannot be handled by offline indexing. Online indexing [**? ?**] and adaptive indexing [**?**] are two approaches to automatic physical design in such dynamic environments, but none of them in isolation handles the problem sufficiently. Online indexing periodically refines the physical design but it may negatively affect running

queries every time it needs to use resources for reconsidering the physical design and it may not be quick to follow the workload changes as it reacts only periodically. Adaptive indexing does not have this problem as it introduces continuous, incremental and partial index refinement but it adjusts the physical design only during query processing based on queries.

Always Indexing. In this paper, we make the observation that in real systems there are plenty of resources that remain under-utilized and we propose to exploit those resources to be able to better address dynamic and ad-hoc environments. In particular, we focus on exploiting CPU cycles to the maximum by continuously detecting idle CPU cycles and using them to refine the physical design (in parallel with query processing). Such idle CPU cycles occur when the system does not exploit all available cores up to 100%. We distinguish between "*idle time*" as in "*there is no user-driven workload at all and the entire CPU (all its hardware contexts) is idle (except possible occasional operating system background activity)*" and "*idle CPU resources*" as in "*the active user-driven workload does / can not use all physically available CPU hardware contexts.*" Intuitively, there are two options when resources are under-utilized but still there are active queries in the system. The first option is to introduce more parallel query processing algorithms to maximize utilization for the existing workload. The second one is to exploit the extra resources towards a different goal (extra indexing actions in our case). We investigate and compare both directions.

Holistic Indexing. In this paper, we introduce a new indexing approach, called *holistic indexing*. Holistic indexing addresses the automatic physical design problem in modern applications with dynamic and exploratory workloads. It continuously monitors the workload and the CPU utilization; when idle CPU cycles are detected, holistic indexing exploits them in order to partially and incrementally adjust the physical design based on the collected statistical information. Each index refinement step may take only a few microseconds to complete and the system will typically perform several such steps in one go depending on available system resources. Everything happens in parallel to query processing but without disturbing running queries. The net effect is that holistic indexing refines the physical design, improving performance and robustness by enabling better data access patterns for future queries. Table **??** summarizes the qualitative difference between holistic indexing and current state-of-the-art indexing approaches. Compared to past approaches, holistic indexing manages to minimize both initialization and maintenance costs, as it relies on partial indexing, and to exploit all possible CPU resources in order to provide a more complete physical design.

Contributions. Our contributions are as follows:

- We introduce the idea of exploiting idle CPU resources towards continuously adapting the physical design to ad-hoc and dynamic workloads.
- We discuss in detail the design of holistic indexing on top of modern column-store architectures, i.e., how to detect and exploit idle CPU resources during query processing.
- We implemented holistic indexing in an open-source column-store, MonetDB [**? ?**]. Through a detailed experimental analysis both with microbenchmarks and with TPC-H we demonstrate that we can exploit idle CPU resources to prepare the physical design better, leading to significant improvements over past indexing approaches in dynamic environments.

Paper Structure. The rest of the paper is structured as follows: Section **??** provides an overview of related work. In Section **??**, we shortly recap the basics of column-store architectures and the basics of adaptive indexing. Then, Section 4.2 introduces holistic indexing, while Section 4.3 presents a detailed experimental analysis. Finally, Section **??** discusses future work and concludes the paper.

4.2 Holistic Indexing

In this section we discuss the fundamentals of holistic indexing. We designed holistic indexing on top of column-store architectures inspired by their flexibility on manipulating some attributes without affecting the rest. During query processing indices are built and optimized incrementally by adapting to query predicates, as in adaptive indexing. However, in contrast to adaptive indexing, index refinement actions are not triggered only as a side-effect of query processing; in holistic indexing incremental index optimization actions take place continuously in order to exploit under-utilized CPU cores. Thus, concurrently with user queries, system queries also refine the index space. Holistic indexing monitors the workload and CPU resources utilization and every time it detects that the system is under-utilized it exploits statistical information to decide which indices to refine and by how much.

Thus, with holistic indexing we achieve an always active self-organizing DBMS by continuously adjusting the physical design to workload demands.

Problem Definition: *Given a set of adaptive indices, statistical information about the past workload, storage constraints and the CPU utilization, continuously select indices from the index space and incrementally refine them, while the materialized index space size does not exceed the storage budget.*

In the rest of this section, we discuss in detail how we fit holistic indexing in a modern DBMS architecture.

4.2.1 Preliminary Definitions

First, we give a series of definitions.

Workload. A workload W consists of a sequence of user queries, inserts and deletes. Updates are translated into a deletion that is followed by an insertion.

CPU Utilization. CPU utilization in a time interval dt describes how much of the available CPU power is used in dt. Specifically, it expresses the percentage of total CPU time, i.e., the amount of time for which the CPU is used for processing user or kernel processes instead of being idle. CPU utilization is calculated using (operating system) kernel statistics.

Configuration. A configuration is defined as a set of adaptive indices that can be used in the physical design. There are three kinds of configurations. The *actual configuration*, C_{actual}, contains indices on attributes that have already been accessed by user queries in the workload. Indices are inserted in C_{actual} when they are created during query processing. For instance, assume a query Q enters the system and contains a selection on an attribute A. If the adaptive index on A does not exist, it is created on-the-fly and it is inserted in C_{actual}.

Besides C_{actual}, holistic indexing also maintains the *potential configuration*. $C_{potential}$, which contains indices on attributes that have not been queried yet. Indices are inserted in $C_{potential}$ either automatically by the system or manually by the user. Finally, the *optimal configuration*, $C_{optimal}$, contains indices that have reached the optimal status (the next paragraph describes when an index is considered optimal). The union of C_{actual} and $C_{potential}$ constitutes the index space *IS*, i.e., the indices which are candidates for incremental optimization when the system is under-utilized. Later, in Section 4.2.2, we describe how the system is educated to pick an index from *IS*. Indices from $C_{optimal}$ are not considered for further refinement during the physical design reorganization.

Optimal Index. Holistic indexing exploits adaptive indices. As seen in Section **??**, an adaptive index is refined during query processing by physically reorganizing pieces of the cracker column based on query predicates. As more queries arrive, more pieces are created, and thus, the pieces become smaller. We have found that when the size of the pieces becomes equal to L_1 cache size ($|L_1|$), further refinements are not necessary; a smaller size increases administration costs to maintain the extra pieces and it does not bring any significant extra benefit as scanning inside L_1 is fast anyway (no cache

misses). Pieces of size smaller than L_1 cache can either be sorted or queries simply need to scan them (a range select operator has to scan at most two L_1 pieces). An index I on an attribute A is considered optimal (I_{opt}), when the average size of pieces ($|p|$) in A_{CRK} is equal to the size of L_1 cache. Equation (4.1) describes the distance between I and I_{opt}.

$$d(I, I_{opt}) = |p| - |L_1| = \frac{N_A}{p_A} - |L_1| \tag{4.1}$$

N_A is the total number of tuples in A_{CRK} while p_A is the total number of pieces in A_{CRK}. This information is readily available and thus we can easily calculate the average piece size in a cracker column and in turn we can calculate the distance of the respective cracker index from its optimal status.

Statistical Information. During query processing holistic indexing continuously monitors the workload and the CPU utilization. For each column in the schema it collects information regarding how many times it has been accessed by user queries, how many pieces the relevant cracker column contains, how many queries did not need to further refine the index because there was an exact hit. Besides the statistical information about the workload, kernel statistics are used in order to monitor the CPU utilization.

4.2.2 System Design

Holistic indexing is always active. It continuously monitors the workload and the CPU utilization. When under-utilized CPU cores are detected, holistic indexing exploits them in order to adjust the physical design based on the collected statistical information. The system performs several index refinement steps simultaneously depending on available CPU resources. Everything happens in parallel to query processing, but without disturbing running queries.

We discuss in detail the continuous tuning process and how to exploit under-utilized CPU cycles. We also discuss how existing adaptive indexing solutions on core database architectures issuessuch as updates and concurrency control can be directly adapted to work with holistic indexing.

Statistics per Column/Index. Statistics per column are collected during query processing. This is the job of the select operator as it is within the select operator that all (user query) adaptive indexing actions take place. Every time an attribute is accessed for a selection of a user query, the select operator updates a data structure, which contains all statistics for the respective index. Given that the select operator performs adaptive indexing actions anyway, it already has access to critical information such as how many new pieces were created during new cracking actions for this query, whether the select

was an exact match, etc. All information is stored in a heap structure (one node per index) which allows us to easily put new indices in the configuration or drop old ones. The structure is protected with read/write latches as multiple queries or holistic workers (discussed later on) may be cracking in parallel.

Tuning Cycle. At all times there is an active *holistic indexing thread* which runs in parallel to user queries. The responsibility of the holistic indexing thread is to monitor the CPU utilization and to activate *holistic worker threads* to perform auxiliary index refinement actions whenever idle CPU cycles are detected. The tuning process is shown in Figure 4.1. The holistic indexing thread continuously monitors the CPU load at intervals of 1 second at a time. In case holistic worker threads are activated, the holistic indexing thread waits for all worker threads to finish and measures the CPU utilization within the next 1 second. In our analysis, we found that 1 second is the time limit that gives proper kernel statistics. When n idle CPU cores are detected, n holistic worker threads are activated. Each worker thread executes an instance of the IdleFunction, which picks an index from the Index Space IS and performs x partial index refinement actions on it. Every time an index is refined, the respective statistics, e.g., distance from the optimal index, are updated. When an index reaches the optimal status, it is moved into the optimal configuration $C_{optimal}$.

A side-effect of the tuning process is that some of the holistic worker threads might remain idle while the holistic indexing thread waits for all workers to finish. However, as we show later in Section 4.3.1 (Figure **??**(d)), this happens only for very short periods of time and as the system adapts to the workload this phenomenon disappears (as the pieces queried in the adaptive indices become smaller and smaller the holistic indexing workers end up doing tasks of similar weight as none is going to touch a very big piece).

Index Refinement. Every time a worker thread wakes up, it performs x index refinements on a single column. x is a tuning parameter. In our analysis in Section **??** (Figure **??**) we found that a good value for our hardware set-up is $x = 16$. The index refinements are performed by picking x random values in the domain of the respective attribute and cracking the column based on these values. In this way, each time a worker thread cracks a single piece of a column it splits this piece into two new pieces based on the pivot.

There are numerous choices on how to choose a pivot. We found that picking a random pivot is the most cost efficient choice. Other options include to crack the biggest piece of the column, i.e., with the rationale that this takes more work out of future queries. Another option is to crack the smallest piece, i.e., with the rationale that this piece is small because it is hot (because many queries access it for cracking). However, such options are hard to achieve in a lightweight way as we need to maintain a structure such

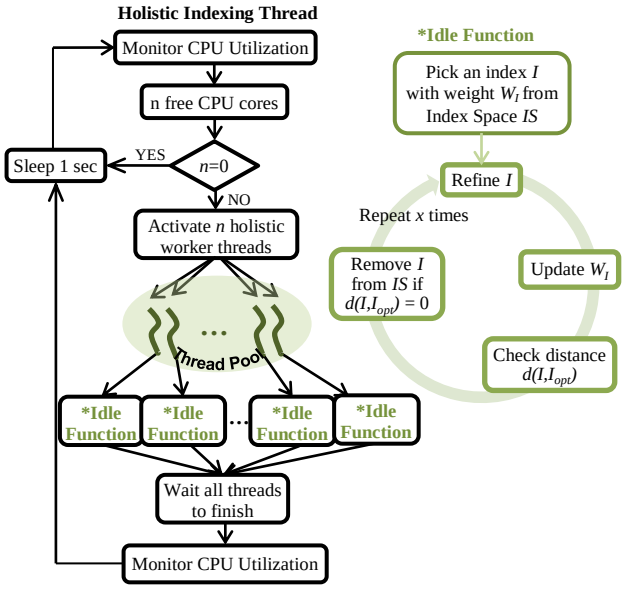


FIGURE 4.1: Tuning actions.

as a priority queue to know which piece is the biggest or smallest every time. Since every cracking action costs a few microseconds or milliseconds it is not worth the extra storage and CPU cost to maintain auxiliary structures. Random pivots converge quickly to cracking the whole domain, providing a column which is balanced in terms of which pieces are cracked and requiring no extra costs in deciding which pivot to choose.

Index Decision Strategies. Another decision we have to make is which index to refine out of the pool of candidate indices. Here, we describe four different strategies we can follow in order to pick an index from the index space. The notion behind the first three strategies is that, since the only information we have is the past workload, we can exploit this information in order to prepare the physical design for a similar future workload. On the contrary, the fourth strategy makes random choices.

For all strategies, a weight W_I is assigned to each index I in the index space. When an index I is added in the candidate indices, its weight is initialized to the distance between I and I_{opt}, which is given by Equation (4.1). For each index I, initially, there is only one partition ($p_I = 1$) in I, i.e., the entire column. Thus, the initial weight $W_{I_{init}}$ is equal to $N_I - L_{1_s}$, where N_I is the cardinality of the respective attribute (with type T) and L_{1_s} is the number of elements of type T that can fit into L_1 cache. The weight is used as a priority number in the first three strategies. The index with the highest priority, i.e., the maximum weight, in C_{actual} is refined first. When W_I becomes equal to zero, I is transferred from C_{actual} to $C_{optimal}$ and it is not considered for further refinement in the future. If C_{actual} is empty, an index is randomly picked from $C_{potential}$. The weight of each index is constantly updated after every index refinement regardless of whether it is caused by a user query or by holistic indexing. Below we describe the four strategies.

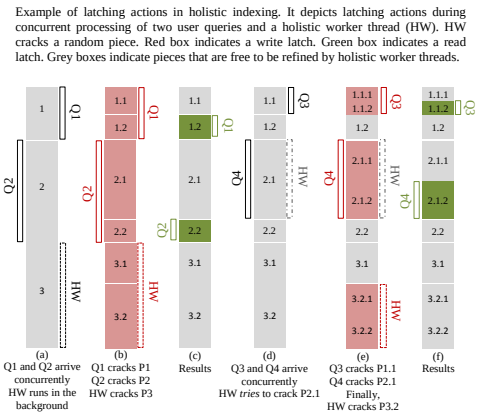


FIGURE 4.2: Concurrency control in holistic indexing.

- **W1:** $W_I = d_I = d(I,I_{opt})$. Using this strategy, we give a priority to indices with large partitions.
- **W2:** $W_I = f_I * d_I$. Priority is given to indices that have large partitions and at the same time are accessed frequently in the workload. f_I is the number of user queries that access I.
- **W3:** $W_I = (f_I - f_{I_h}) * d_I$. In this strategy we try to identify indices that are accessed frequently in the workload and at the same time have large partitions, because they have a high hit rate. These indices have a smaller priority compared to indices with large partitions that are accessed less frequently. f_I is the number of user queries that access I, while f_{I_h} is the number of user queries that do not trigger a refinement of I because the requested value bound already exists in I.
- **W4:** Make a random choice.

Overall, our analysis, which is described later in Section **??** (Figure **??**), with numerous workloads showed that even though small improvements can be achieved when picking the perfect strategy for each workload, the random strategy gives a good and robust overall solution that is always close to the best for all workloads.

Concurrency Control. An index refinement due to holistic indexing happens in parallel with user queries. Since user queries may also cause refinement of adaptive indices, we need to properly control these changes. In addition, as more than one holistic thread may be active at any time, they may be trying to refine the same index. The study of concurrency control for adaptive indexing [**?** **?**] showed that it is possible to allow multiple concurrent index refinements in adaptive indices via lightweight concurrency control, i.e., relying only on latches of individual pieces in an adaptive index. The point is that an index refinement only changes the structure of the index and not its contents (contrary to an actual update). In this case, an index refinement only rearranges values in a single piece of a column at a time. Thus it is sufficient to allow other queries to work on different pieces in parallel by taking read/write latches on individual pieces, called piece latches in [**?** **?**]. We exploit these techniques here in order to allow user queries

and holistic workers to work concurrently over a single column, but we also identify extra opportunities to increase parallelism for holistic workers.

Figure 4.2 shows an example of an adaptive index where two queries are actively cracking it. Each query is interested in its own value range and needs to crack one piece, i.e., at the value of its selection. The idea is that *all* other pieces of the column are available for index refinement by holistic worker threads. One direction would be that each holistic worker decides which piece of an index to refine by picking from a list of pieces that currently have no locks. However, such information is expensive to maintain similarly to our discussion in the "Index Refinement" paragraph. Thus, holistic workers make random choices regarding which value to use as pivot and thus which piece to crack. However, when a holistic worker requests a write latch to crack a piece and it happens that the piece is locked at the moment, then if the latch is not given immediately, the worker picks another random pivot and repeats the procedure until it finds a free piece to crack. In contrast, user queries need to always block in such cases and wait for the piece to be unlocked. For instance, in Figure 4.2(d) the holistic worker thread tries to lock piece 2.1, which is already locked by Q4. Instead of waiting for the lock to be released, the worker chooses another pivot. The new pivot falls in piece 3.2, which is not locked and it is reorganized finally by the worker (Figure 4.2(e)). As we process more queries and as we perform more holistic indexing, the number of pieces in an index grows; as a side-effect the waiting time for taking a latch decreases as there are more candidate pieces to pick from.

Updates. Updates for adaptive indexing have been studied in [**?**]. The design in [**?**] is that updates remain as pending updates and are merged during query processing, i.e., if a query requests a value range that contains one or more pending updates, then only those updates are merged on-the-fly and without destroying any of the information on the adaptive index. Each query needs to lock at most one column piece at a time for cracking and can update this piece at the same time if pending updates for this piece exist [**?** **?**]. Multiple queries may work in parallel updating and cracking separate pieces (value ranges) of the same column.

The difference here is that with holistic indexing, holistic workers not only perform auxiliary index refinement actions but also merge pending updates. That is, if a holistic worker picks a pivot which falls within a piece of the respective column and the value range, for which this piece holds values, has pending updates, then all those pending updates are merged by the holistic worker. In this way, holistic worker threads not only refine the adaptive indices in the background but also bring them more up to date which removes further load from future queries.

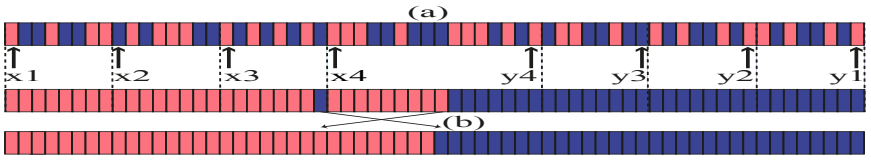


FIGURE 4.3: Refined Partition & Merge (multi-threaded) [**?**].

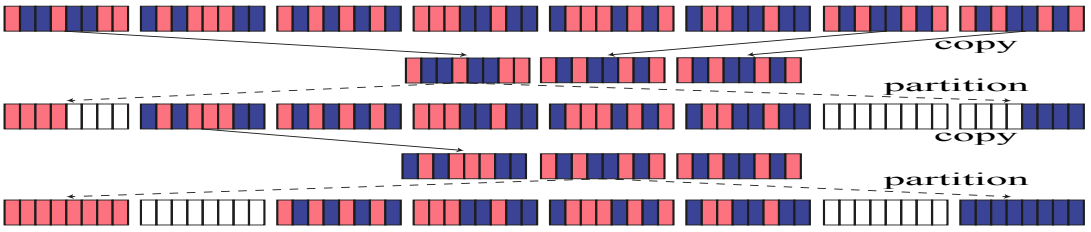


FIGURE 4.4: Vectorized Cracking [**?**].

Storage Constraints. Holistic indexing works within a limited storage budget. Adaptive indices may be dropped or recreated at any time. They consist auxiliary information and thus dropping an index does not lead to any loss of data. In case the storage budget does not allow adding a new index triggered by a user query, then indices are removed with a least frequently used (LFU) policy from the index space at an index-level granularity or at a fine-grained granularity that allows for creating and dropping individual ranges dynamically, as partial cracking suggested in [**?**].

Multi-core Adaptive Indexing. The goal of holistic indexing is to improve the physical design by fully utilizing the available CPU resources. An alternative approach to achieve maximum CPU utilization is to parallelize the index refinement actions triggered by user queries. This problem was studied in [**?**], which introduced a multi-core, CPU efficient cracking algorithm shown in Figure 4.3. In this algorithm, the to-be-cracked piece is partitioned initially into as many slices as the number of threads, e.g., n (Figure 4.3(a)). The center slice is contiguous, while the remaining $n-1$ slices consist of two disjoint halves, each, that are arranged concentrically around the center slice (x_i and y_i indicate the first and the last element of piece i respectively). n threads crack the n slices independently applying a vectorized, out-of-place cracking algorithm (Figure 4.4), which was proven in [**?**] to be the most CPU efficient single-threaded cracking implementation reported so far. Finally, the local data are merged into a big cracked piece (Figure 4.3(b)). We found that devoting all resources to perform adaptive indexing for user queries in parallel does not lead to the absolute best performance. Specifically, we found that we can improve performance even more by assigning part of the resources to holistic indexing. In this way, some of the CPU resources are assigned to parallel cracking for user queries but the rest of the CPU resources are distributed across several holistic workers for additional index refinements. In the experimental section we show why this approach is better than assigning all available resources to user queries.

4.3 Experimental Analysis

In this section, we demonstrate that holistic indexing leads to a self-organizing always-on DBMS with substantial benefits in terms of response time; with zero administration or set-up effort holistic indexing improves performance adaptively by exploiting all available CPU resources to the maximum. We present a detailed experimental analysis using both standard benchmarks such as TPC-H and real-life workloads such as SkyServer as well as synthetic microbenchmarks for a fine-grained analysis.

We use a dual-socket machine equipped with two 2.00 GHz Intel(R) Xeon(R) CPU E5-2650 processors and with 256 GB RAM. Each processor has 8 hyper-threading cores resulting in 32 hardware threads in total. The operating system is Fedora 20 (kernel version 3.12.10). All experiments we report are based on an implementation of holistic indexing in MonetDB and assume a main-memory environment.

4.3.1 Improving over State-of-the-Art Indexing

In our first experiment we demonstrate that holistic indexing has the potential to bring substantial performance improvements over existing state-of-the-art indexing approaches. We test holistic indexing against parallel versions of adaptive indexing (database cracking), offline indexing, online indexing and plain scans.

For plain scans (no indexing), we use a parallel select operator implemented in MonetDB. For offline and online indexing we sort the columns using a highly parallel NUMA-aware sorting algorithm that was introduced in [**?**] (m-way, 16-byte keys) and is publicly available in [**?**]. Specifically, in offline indexing we pre-sort all the columns before query processing, while in online indexing we assume that after processing a few queries we understand the workload patterns and then we sort the relevant columns. MonetDB automatically detects that a column is sorted and can use efficient binary search actions during select operations. For adaptive indexing we use the parallel vectorized database cracking algorithm that was introduced in [**?**] (see Section 4.2.2).

Here we use a synthetic benchmark. The query workload consists of 10^3 range select queries over a table of 10 attributes; each query touches a single attribute (we will see more complex queries later on). Each attribute consists of 2^{30} uniformly distributed integers, while the value range requested by each query (and thus the selectivity) is random. All queries are of the following form.

select A from R where A ¡ v