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 [1–3] to automate the database physical design. Auto-tuning tools mainly rely on a "what-if analysis" [4] and close interaction with the optimizer [5] 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 [6] 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 [7]. Soft indices [8] 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 [9, 10], Hadoop [11] as well as for improving more traditional disk-based settings [12]. It has been shown to work for many core database architecture issues such as updates [13], multi-attribute queries [14], concurrency control [15–17], partition-merge-like logic [12, 18]. In addition, [19] shows how to benchmark adaptive indexing techniques, while stochastic database cracking [20] shows how to be robust on various workloads and [21] 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 [15, 22].

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 [23] as well as using incoming queries for more broad storage layout decisions, i.e., reorganizing base data (columns/rows) according to incoming query requests [24], or even about which data should be loaded [25]. In addition, adaptive indexing ideas have been used to design new generation data exploration tools such as touch-based data systems [26, 27].

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 [28–31]. 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 [32–35].

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.

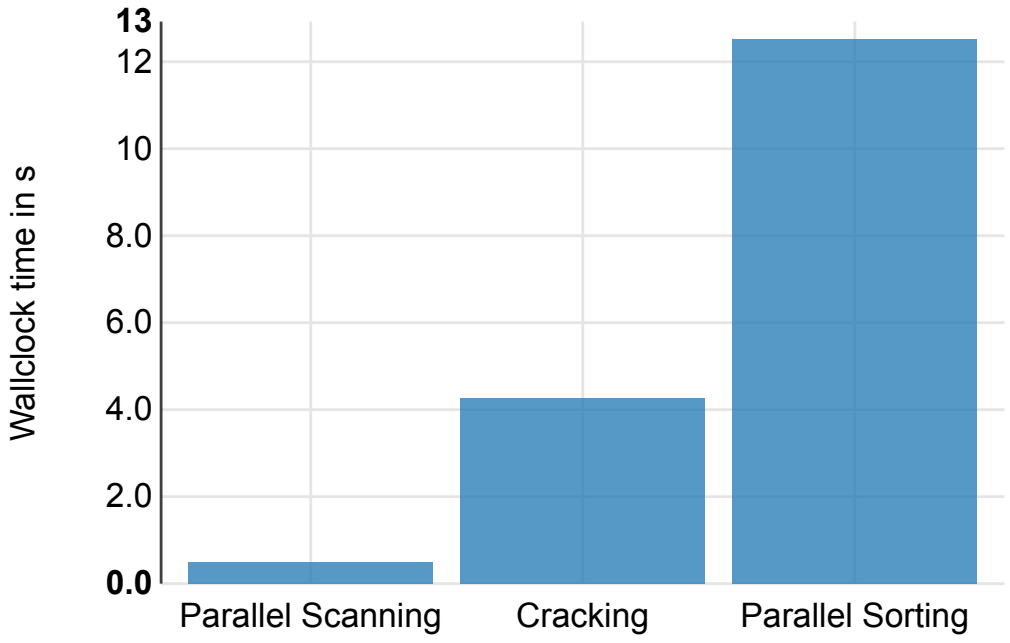


FIGURE 3.1: Costs of Database Operations

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 [36], 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* [9]. 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 [37]. 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) [38]. Thus, care has to be taken to ensure that there are sufficiently many independent instructions [39, 40].

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 [35], while [32, 33] 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 [41] and join algorithms [32, 33].

Indexing Techniques

In the majority of automated index tuning approaches, index tuning is clearly distinct from query processing. Offline indexing approaches [1–3] analyze a given workload and

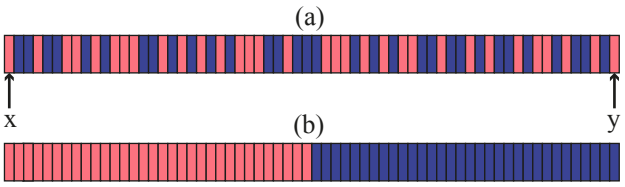


FIGURE 3.2: Original Cracking (single-threaded)

select/create the necessary indexes before the workload enters the system, whereas online indexing approaches [6–8] 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.

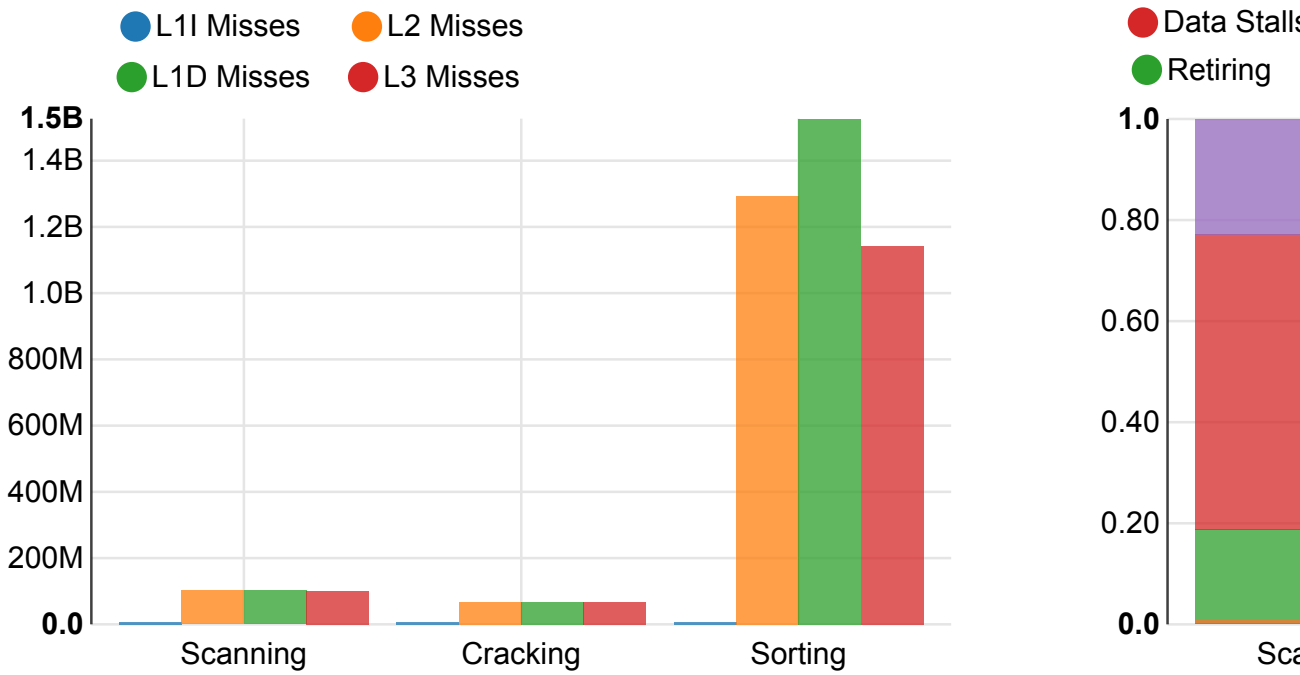
Adaptive indexing [9, 12, 18] is a recent, lightweight approach to self-tuning databases: data reorganization is integrated with query processing. *Database Cracking* [9] is an implementation of the adaptive indexing concept. Database Cracking initializes a partial index for an attribute the first time 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 [42]) 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 [10], 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 [43]. 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 [44].

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.

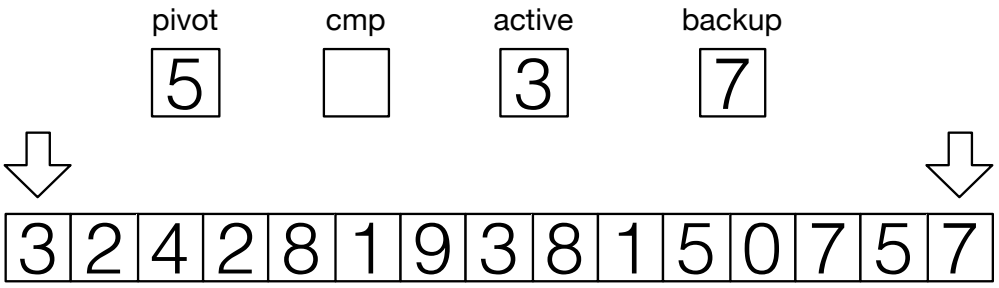
Multi-threaded

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).

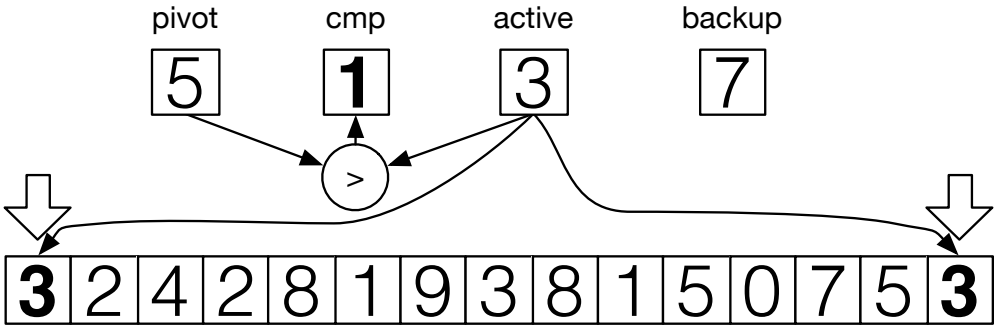
For reference, we include the best single-threaded implementation (Vectorized) in the chart. We observe a significant speedup in almost all cases. Naturally, the *Refined Partition & Merge* implementation performs better than its plain counterpart. In addition, both implementations achieve a performance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

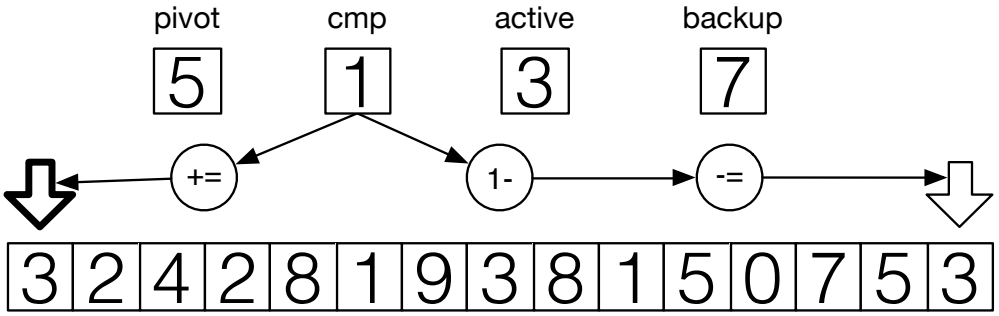
CPU-efficient implementation of even simple algorithms is hard: while common knowledge in many fields of computer science, this insight is still not properly appreciated in the field of data management. In this paper, we conducted an in-depth study of such a supposedly simple algorithm: pivoted partitioning.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



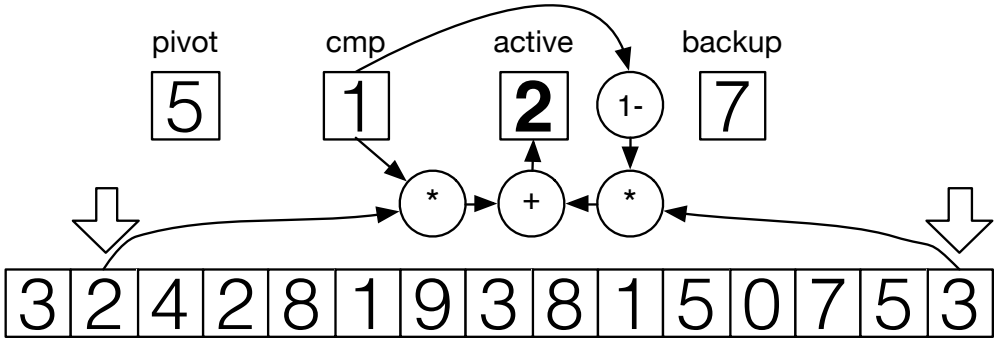
(A) Consistent State



(B) Compare & Write Phase



(C) Advance Cursor Phase



(D) Backup Phase

FIGURE 3.4: Predicated Cracking

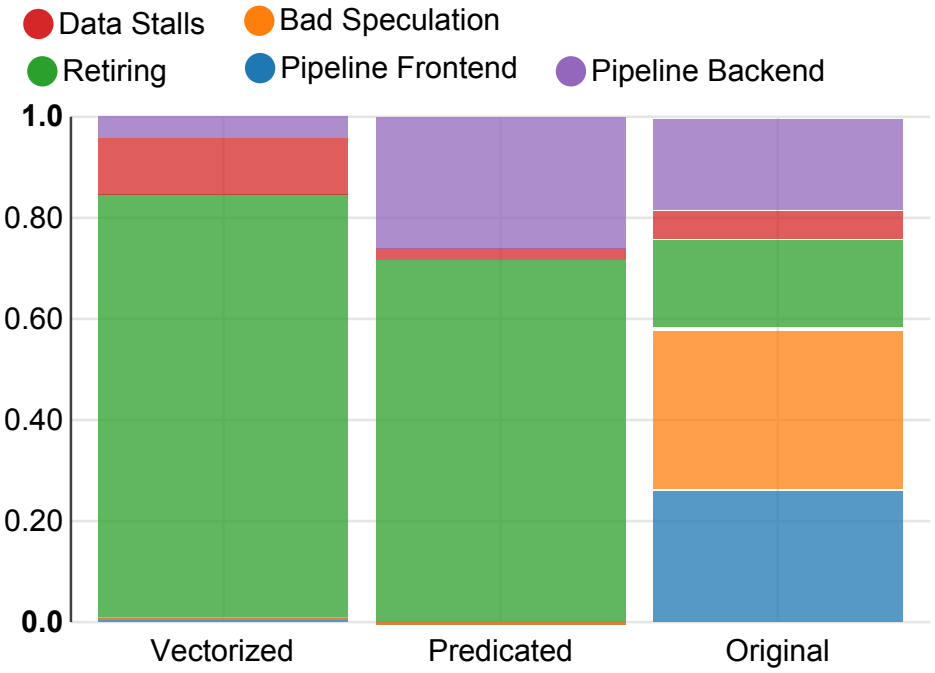


FIGURE 3.5: Cost breakdown of single-threaded implementations

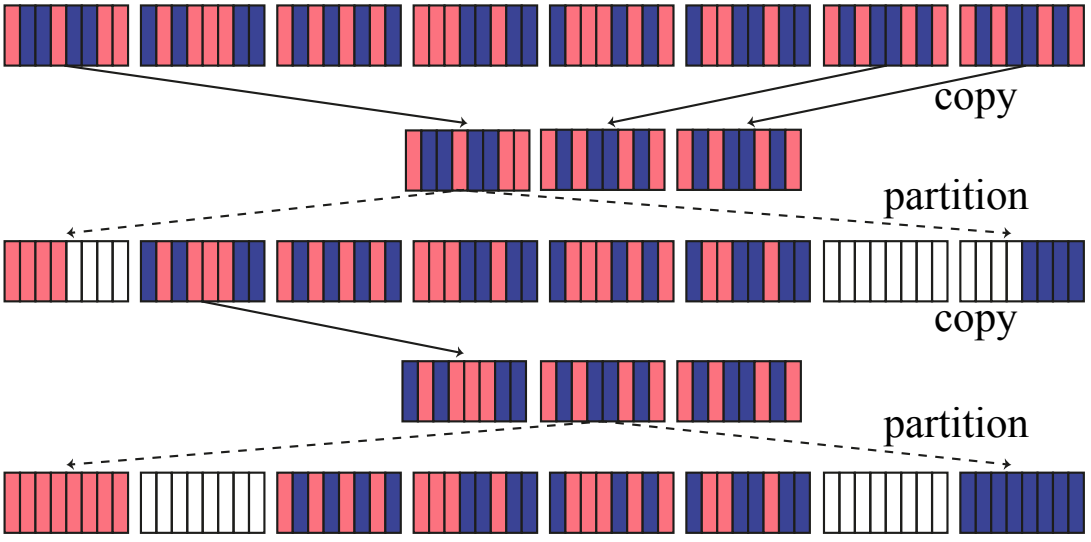


FIGURE 3.6: Vectorized Cracking

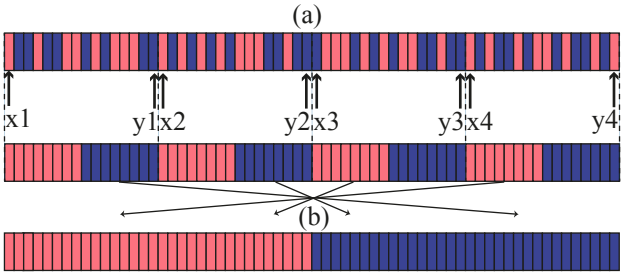


FIGURE 3.7: Simple Partition & Merge (multi-threaded)

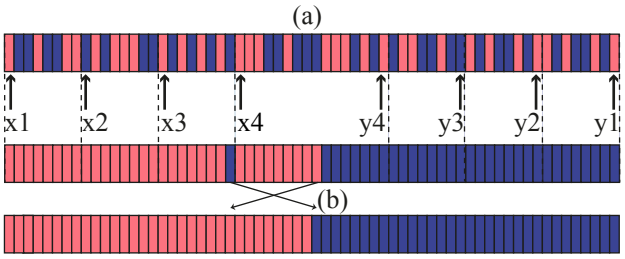
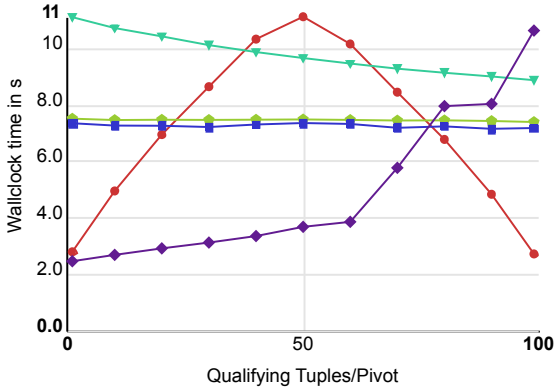
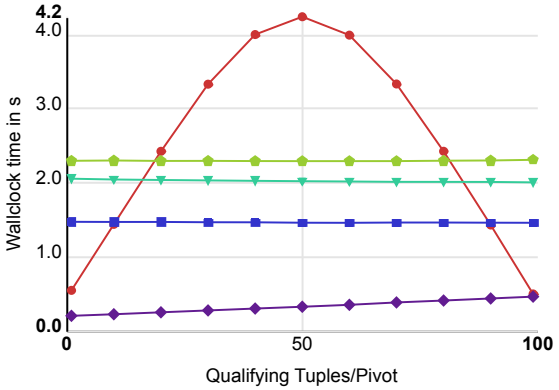


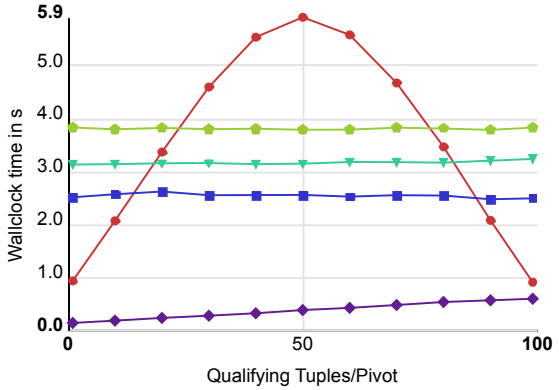
FIGURE 3.8: Refined Partition & Merge (multi-threaded)



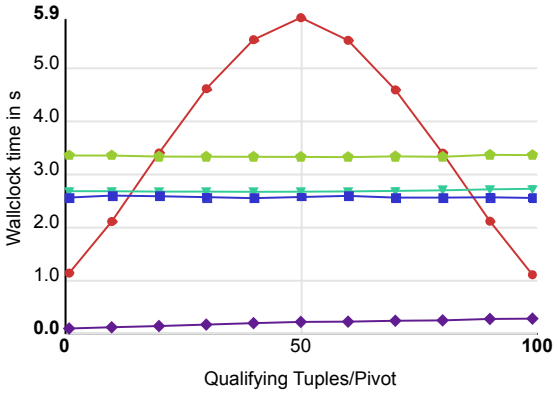
(A) Desktop



(B) Workstation

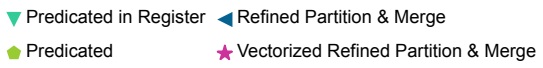


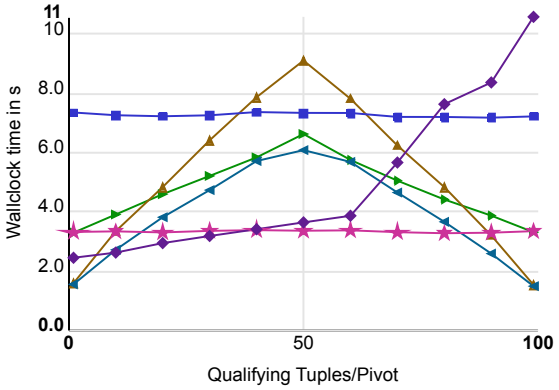
(C) Server



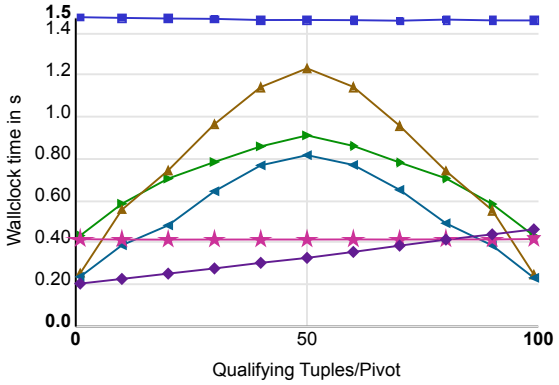
(D) High-End Server

FIGURE 3.9: Single Threaded Performance

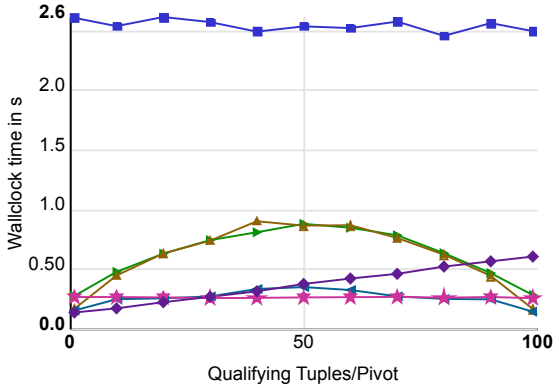

Predicated in Register
Refined Partition & Merge
Predicated
Vectorized Refined Partition & Merge



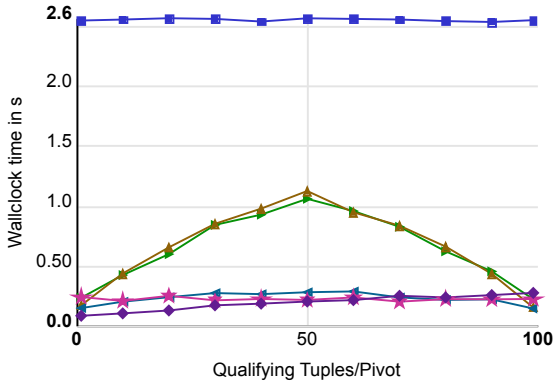
(A) Desktop



(B) Workstation



(C) Server



(D) High-End Server

FIGURE 3.10: Multi Threaded Performance

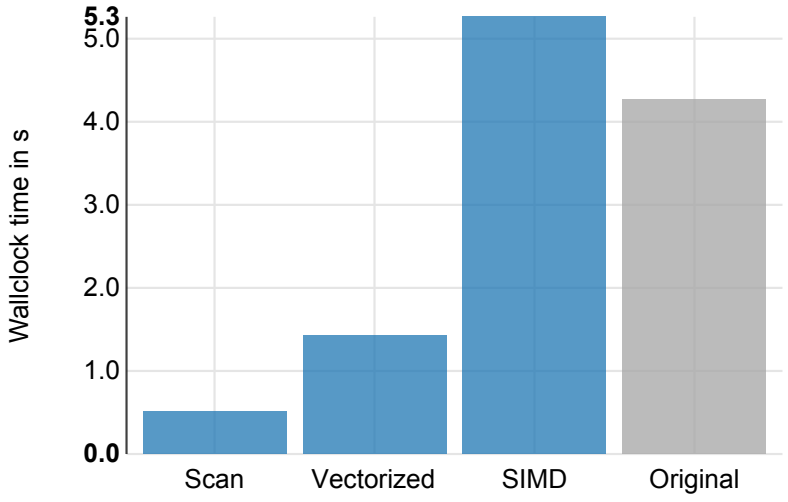


FIGURE 3.11: SIMD Processing Performance at 50% Selectivity

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 [45] that can access data better and faster [46–49], that can better exploit modern hardware and opportunities for massive parallelization [50] that can support efficient processing of OLTP and/or OLAP queries [51–54].

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 [1–3]. 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 [6, 7] and adaptive indexing [9] 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 [55, 56]. 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 4.5(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 4.3.5 (Figure 4.14) 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

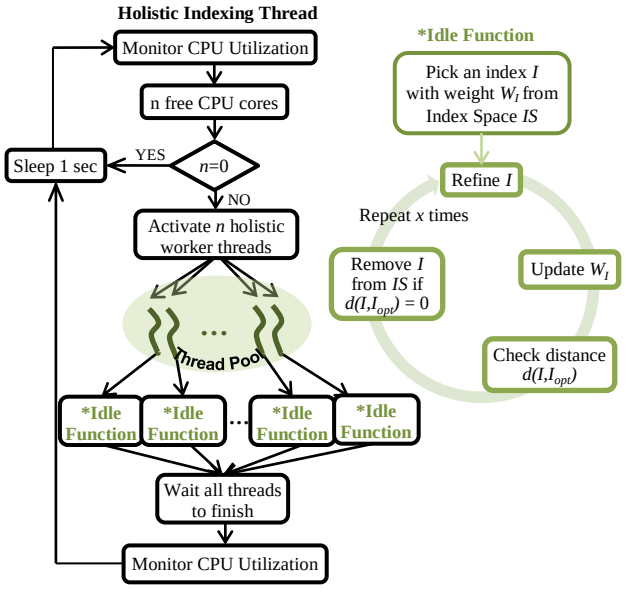


FIGURE 4.1: Tuning actions.

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.

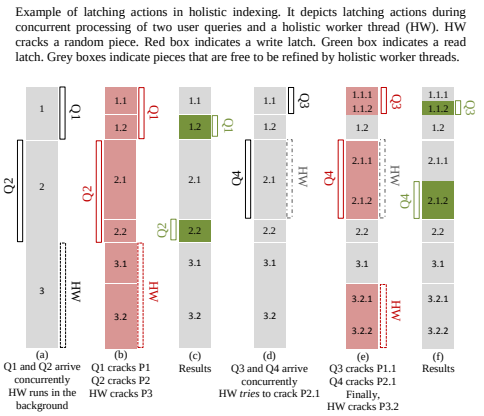


FIGURE 4.2: Concurrency control in holistic indexing.

- **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 4.3.4 (Figure 4.12), 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 [16, 17] 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 [16, 17]. 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 [14]. The design in [14] 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 [16, 17]. 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.

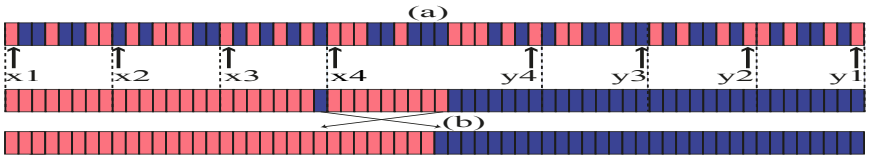


FIGURE 4.3: Refined Partition & Merge (multi-threaded) [22].

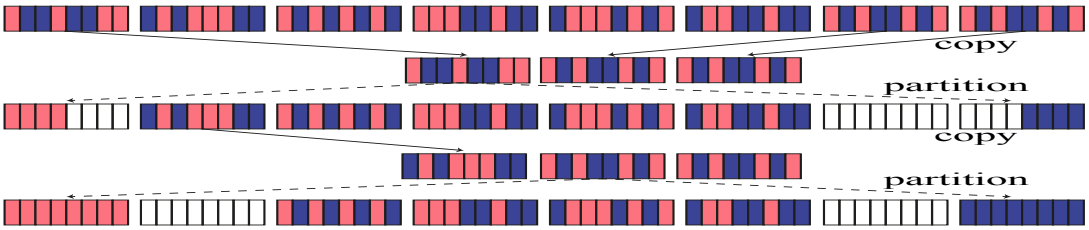
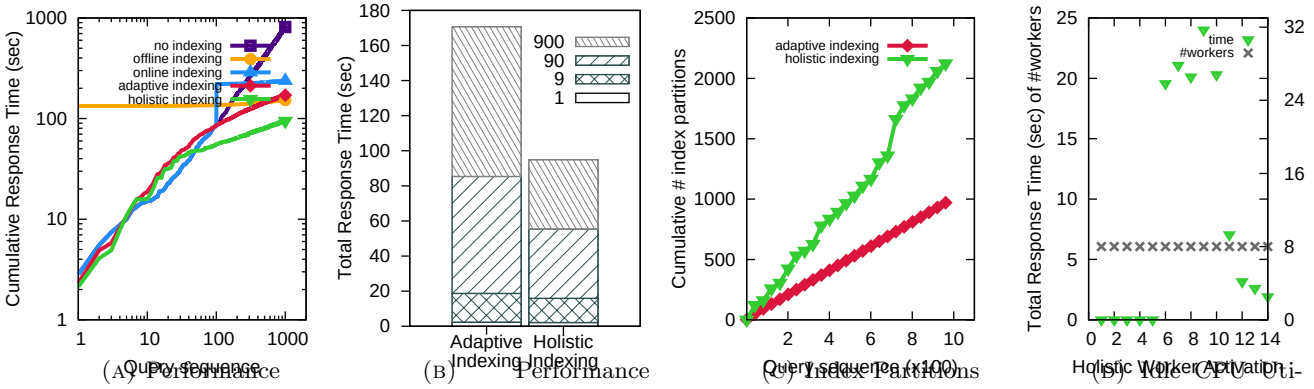


FIGURE 4.4: Vectorized Cracking [22].



(A) Performance (B) Performance Breakdown (C) Index Partitions (D) Idle CPU Utilization

FIGURE 4.5: Improving performance with holistic indexing.

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 [14].

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 [22], 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 [22] 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 [32] (m-way, 16-byte keys) and is publicly available in [57]. 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 [22] (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.

$$\textit{select } A \textit{ from } R \textit{ where } A \text{ ¡ } v$$

The tested scenario assumes a dynamic and ad-hoc environment with zero workload knowledge and zero idle time to pre-index the data. Figure 4.5(a) shows the results. On the x-axis queries are ranked in execution order. The y-axis represents the cumulative response time as the query sequence evolves, i.e., each point (x, y) represents the sum of the execution time y for the first x queries. In this way, the graph shows how the response times evolve as we process more and more queries.

If there is no indexing support (plain scans), the entire column/ attribute is scanned in parallel by 32 threads for every query. Because of this stable access pattern, the cumulative response time of the query sequence grows linearly as every query has similar cost. With offline indexing, on the other hand, it takes 12 seconds to completely sort each column, assuming a priori workload knowledge. This leads to a 120 seconds initialization overhead to sort all 10 columns. Since there is no idle time before the first query, the sorting cost is added to the execution time of the very first query in Figure 4.5(a). After the first query, all queries are answered with fast binary search actions which results in a rather flat cumulative curve. With online indexing, the first 100 queries are answered without any index support and thus the cost grows linearly. After the 100th query, assuming enough workload knowledge has been obtained via monitoring, we proceed to sort all 10 columns. The sorting cost is added to the execution cost of the 101st query, since there is no idle time between the 100th and the 101st query. As with offline indexing, all subsequent queries are answered extremely fast with binary search over the sorted column. Nevertheless, both in offline and in online indexing, the whole query sequence is affected by the sorting costs. On the other hand, adaptive indexing continuously improves performance requiring no workload knowledge and without penalizing individual queries. This improvement comes from the fact that adaptive indexing builds only partial indices which it incrementally refines as queries arrive. However, there is still room for big improvements.

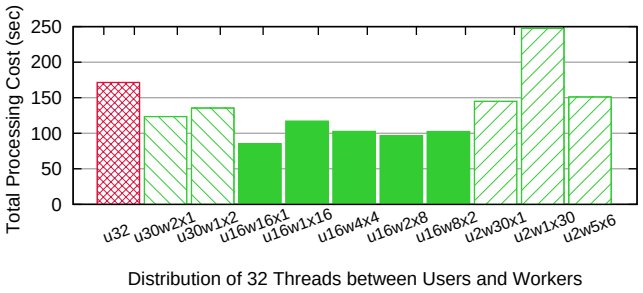


FIGURE 4.6: Performance improves if we distribute the threads equally between user queries and holistic workers.

Holistic indexing manages to further improve the performance of the workload by about 50%. Contrary to the other indexing approaches, MonetDB with holistic indexing enabled monitors the CPU utilization and constantly tries to maximize it. When holistic indexing detects idle CPU resources, it triggers index refinement actions on existing adaptive indices. In this experiment, an index is inserted in the index set, and specifically in C_{actual}, when a user query creates it. For holistic indexing the actual user queries behave in exactly the same way as in adaptive indexing, i.e., the first query will create an adaptive index and subsequent queries refine it using the very efficient and almost linearly scaling parallel vectorized cracking implementation from [22]. The difference is that with holistic indexing enabled idle CPU resources are exploited towards further refining the adaptive indices in a way which does not hurt running queries. Since parallel vectorized cracking [22] is designed to be CPU efficient, encountering only very little resource stalls, we generated kind of a worst-case scenario for holistic indexing: we limited the maximal number of hardware context assigned to user queries to 16 (equal to number of physical cores), leaving at least 16 (otherwise not effectively usable by the prime user query workload) hardware contexts ("hyper-threads") available for holistic indexing. Constantly, our load-checker usually detects 16 idle hardware contexts, and consequently starts 8 holistic indexing workers (each using two threads) as shown in Figure 4.5(d). Figure 4.6 shows that the combination of using maximal 16 (out of 32) hardware contexts for user queries (performing parallel vectorized adaptive indexing [22]), while devoting any remaining idle hardware context to holistic indexing, improved the overall performance by a factor 2 over using all 32 hardware contexts for user queries (and thus none for holistic indexing).

Figure 4.5(b) is a breakdown of the performance of holistic indexing and adaptive indexing. The y-axis represents the total response time of the first query, the next 9 queries, etc. The total height of each bar represents the total response time to run the entire workload of 10^3 queries. The first few queries do not see any improvement because holistic indexing cannot concurrently refine a column if there are user queries cracking it. This is because initially columns have not been cracked at all and thus the first few user queries will lock big pieces for cracking. However, as each column is cracked into

smaller pieces, holistic indexing may invoke actions to refine a column even if concurrent queries are cracking it. Essentially, each user query needs to lock at most one piece of an adaptive index at a time, i.e., the piece it is about to crack, and thus holistic indexing may choose any of the remaining pieces to perform further index refinements.

Holistic indexing outperforms adaptive indexing by a factor 2 by injecting additional index refinements on top of those that adaptive indexing does anyway. Figure 4.5(c) shows the amount of pieces which have been created in all 10 adaptive indices; holistic indexing creates more pieces than adaptive indexing. As a result, future user queries need to touch less data as they find a fine-grained index, and thus performance improves for holistic indexing. As Figure 4.7 shows, the first queries that access the same index run slower, because they reorganize big partitions. Thus, additional pieces have to be inserted in the index as early as possible in the query sequence.

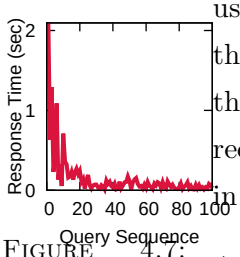


FIGURE 4.7: Adaptive Ind.

As discussed in Section 4.2.2, on some occasions the main holistic indexing thread waits for all workers to finish before assigning new tasks, leaving under-utilized CPU resources for some brief periods. Figure 4.5(d) shows how the total response time of all workers in every tuning cycle changes over time and as the query sequence evolves. The right y-axis shows the number of holistic worker threads the holistic indexing thread activates whenever it detects idle CPU resources. The maximum number of workers that holistic indexing can activate is 8. The left y-axis represents the total response time of all workers during a single tuning cycle. The x-axis represents the activations of holistic indexing. A single activation of holistic indexing triggers the activation of multiple holistic workers. The total response time of the workload is 90 seconds. Within this amount of time, holistic indexing is activated only 15 times, because of the waiting time (1 second) between two CPU load measurements and because of the waiting time until all workers finish in every tuning cycle. We observe that the response time of the workers is high only for the first few activations and reduces very fast as the index becomes fine-grained. In this way, the system adapts on its own and eventually no worker is a bottleneck.

Holistic indexing sees even bigger performance benefits when there is idle time before query processing. When there is idle time and no workload knowledge, holistic indexing chooses random indexes to insert in $C_{potential}$ and refines them until the first query arrives. Here using the same set-up as in the previous experiment, we manually induce idle time and holistic indexing adds 10 random indices in $C_{potential}$. Figure 4.8 shows the results. Compared to adaptive indexing, which does not exploit the a priori idle time e (22 seconds), holistic indexing exploits this time to spread tuning actions over 10 indices. Thus, when user queries are processed they

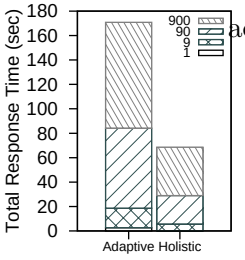
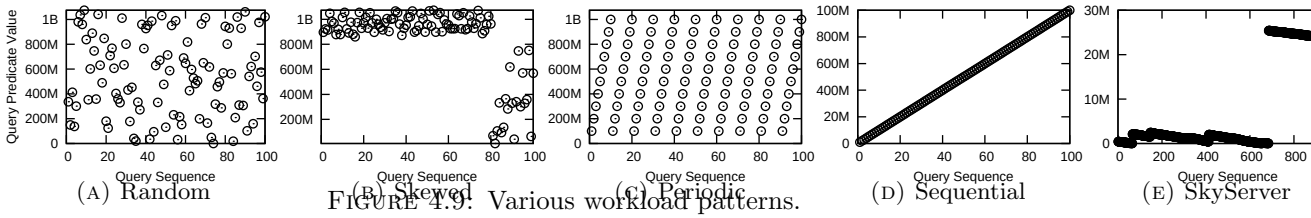


FIGURE 4.8:



(A) Random (B) Skewed (C) Periodic (D) Sequential (E) SkyServer

FIGURE 4.9: Various workload patterns.

reorganize smaller partitions and the benefit is already obvious in the beginning of the workload compared to Figure 4.5(b), where the benefit in the workload appears after the 10th query when all indices have been inserted in the index set automatically by the system.

By being able to completely automatically utilize available CPU resources and direct them towards lightweight actions that may improve future requests, holistic indexing can bring significant performance gains on top of existing indexing approaches. It outperforms adaptive indexing by a factor 2 in terms of individual query performance At the same time it outperforms offline and online indexing, especially in the beginning of the workload, when offline indexing penalizes the first queries with the index building cost and online indexing does not provide any indexing support until the 100th query. Besides the difference in performance, the qualitative difference described in Table **??**, makes holistic indexing a very appealing indexing approach in modern dynamic environments.

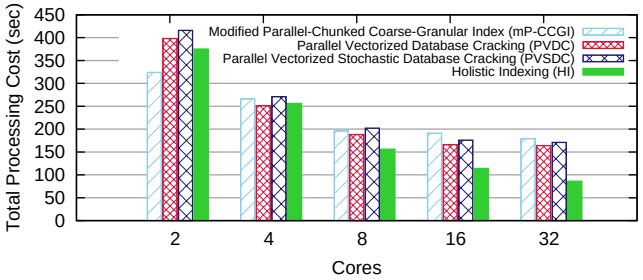


FIGURE 4.10: Holistic indexing utilizes resources more effectively than multi-core state-of-the-art adaptive indexing baselines.

4.3.2 Holistic Vs. Multi-core Adaptive Indexing

Assuming there are several CPU cores available in a modern system, in holistic indexing we utilize them by spreading auxiliary tuning actions across multiple indices. An alternative way to "keep the CPU busy", is to parallelize the existing adaptive indexing algorithms. In this experiment we study how state-of-the-art adaptive indexing baselines compare to holistic indexing. In particular, we study parallel vectorized database cracking (PVDC), parallel vectorized stochastic database cracking (PVSDC) [22] and a modified version of parallel chunked coarse granular index (P-CCGI) [15] that we name mP-CCGI.

Stochastic cracking aims to provide robustness by performing auxiliary stochastic indexing actions. Although stochastic cracking studied the option of collecting statistics to target the proper value ranges to index, it was shown in [20] that reacting immediately to workload changes by auxiliary stochastic cracking actions has a better effect (i.e., more robust). This is because in stochastic cracking a running query performs auxiliary random cracking only within the piece that is already about to be cracked within a given column and as a result any action brings a benefit as it imposes more order. Holistic indexing considers a much more broad space of statistics keeping track of column-statistics to decide which columns to fine tune.

Both stochastic cracking and plain database cracking in this experiment utilize multi-cores as described in the last paragraph of Section 4.2. The original P-CCGI algorithm partitions the data into as many chunks as the available number of threads and the first query cracks each chunk in parallel having a separate cracking index for each chunk. Subsequent queries crack the chunks in parallel, while they benefit from the initial range partitioning. However, this way, data that belongs in a single value range is physically stored in separate chunks/arrays and feeding from there other relational operators is not compatible with a column-store such as MonetDB that relies on bulk processing; it does not allow to exploit tight for loops without intermediate if statements to detect when we should skip from one chunk to the next during an operator. To address this we extended the original P-CCGI algorithm [15] with the ability to consolidate selection results in a single array using the same techniques that were used for hybrid adaptive indexing which also operates on multiple chunks (but not in parallel) [18]; each query consolidates only the qualifying value ranges and each value range needs to be written to the contiguous array by a single query only, i.e., subsequent queries will only have to do consolidation if they need a new value range never consolidated before. In a vectorized column-store this could be done without consolidation, potentially improving performance further as has been indicated by partial sideways cracking [14]; vectorized processing for adaptive indexing is an open topic, though, orthogonal to this work.

The workload in this experiment consists of 10^3 select-project queries (as in the previous experiment) on 10 integer attributes. Each attribute consists of 2^{30} uniformly distributed integers. The value range requested by each query is random while we vary the number of available CPU cores from 2 to 32, i.e., the maximum number of cores in our system. For holistic indexing we give half of the cores to user queries and the rest of the cores are used by the workers (after testing all possible configurations, we found that this is the one that performs best in all cases). The labels on top of the bars that represent the performance of holistic indexing indicate the distribution of the threads between user queries and workers in every case (similar to Figure 4.6).

Figure 4.10 shows the results. In all cases, the performance improves as we invoke more cores into query processing. For multi-core database cracking and stochastic cracking the performance improves because many threads crack in parallel for one query at a time while for holistic indexing performance improves because many threads work in the background in parallel with query processing to further refine the various indices with auxiliary indexing actions. Holistic indexing sees a bigger improvement, because it is active all the time, i.e., maximizing CPU usage.On the contrary, multi-core vectorized database cracking and multi-core stochastic cracking improve the performance only during user queries and only during the cracking actions. Another subtle difference but one with a major performance impact is that while stochastic cracking and database cracking target all their adaptive indexing actions on specific pieces as they are driven by individual queries, holistic indexing spreads its actions across the whole range of the domain and thus across the whole range of an adaptive index (stochastic cracking does random actions but only within a single piece). This creates a nicely balanced index which has more potential to benefit future queries. The modified version of P-CCGI initially range partitions the data, which can be seen as a pre-index step. However, this is a cost that penalizes the first set of queries.

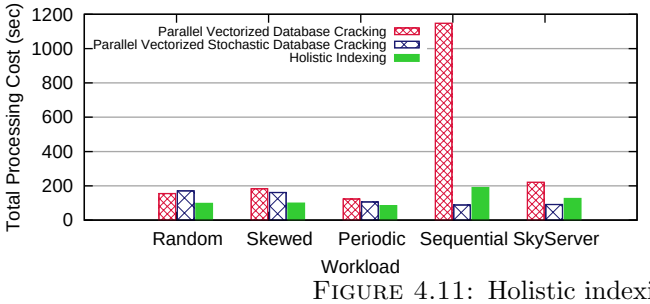
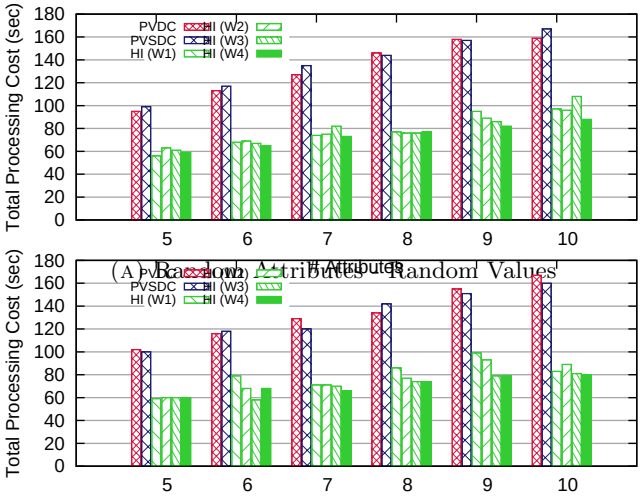
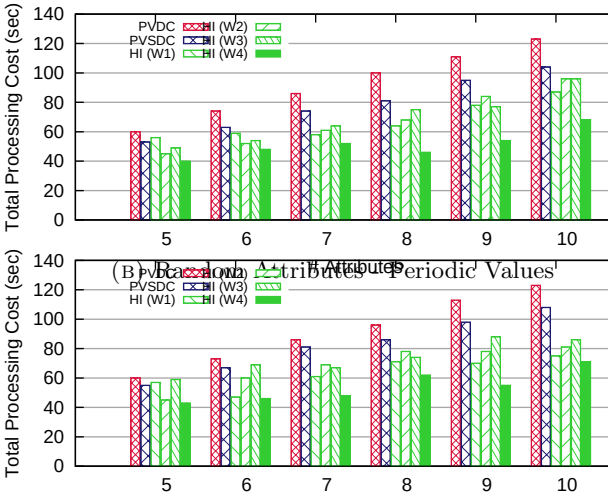


FIGURE 4.11: Holistic indexing is robust.



(A) Random Attributes - Random Values



(B) Random Attributes - Periodic Values

(C) Skewed Attributes - Random Values

(D) Skewed Attributes - Periodic Values

FIGURE 4.12: More attributes mean more room for holistic indexing as more skewed attributes a schema. All strategies have similar performance.

4.3.3 Robustness

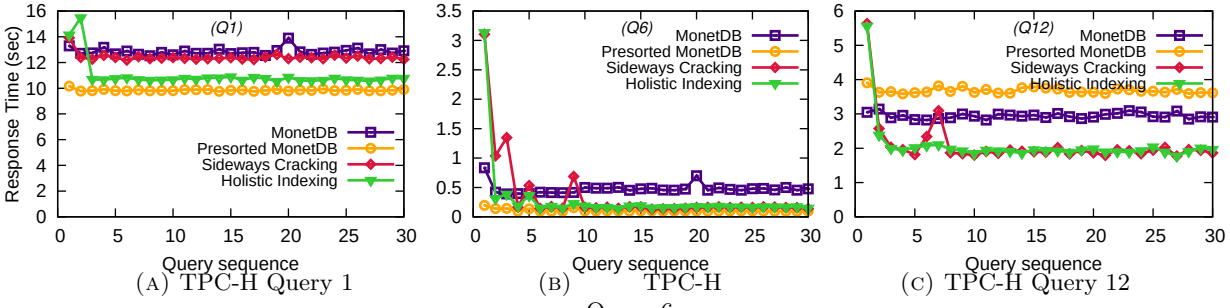
In our next experiment we study how holistic indexing compares to parallel database cracking and parallel stochastic cracking in terms of robustness. We show that holistic indexing maintains the good properties of adaptive indexing by utilizing the available CPU resources more effectively. Both holistic indexing and the parallel variants of adaptive indexing utilize all the available CPU cores.

We test four synthetic workloads. Each workload consists of 10^3 queries on 10 attributes ($\sim$100 queries/attribute). Each attribute consists of 2^{30} uniformly distributed integers. The queries follow a different pattern in each workload. The first four subfigures in Figure 4.9 depict those workload patterns. For each workload, the respective figure illustrates graphically how a sequence of queries touches the value domain of a single attribute.

Furthermore, we test holistic indexing in a real-life workload using data and queries from SkyServer [58]. SkyServer collects astronomical data and the database can be accessed publicly by individual users and institutions. We pose 10^4 real user queries that have been logged by the project servers on the "Photoobjall" table. The "Photoobjall" table consists of 1.2 Billion tuples. All queries access the "Ascension" attribute and are posed in exactly the same chronological order they were logged. The pattern the SkyServer queries follow is shown in Figure 4.9(e).

Figure 4.11 shows the results. For each indexing method we report the total time needed to process all queries for each workload.

Synthetic Workloads. In all synthetic workloads holistic indexing outperforms multi-core database cracking by a factor 2-10 depending on the workload. Multi-core database cracking is strictly driven by query predicates, and thus, can leave large unindexed pieces to be reorganized by future queries. For instance, in the sequential workload in Figure 4.9(d), each query cracks a column in a small piece and in a big piece, and then, a future query needs to crack the big piece again, resulting in a high cost. Multi-core stochastic cracking solves these robustness issues by injecting one extra random cracking action for each user query in order to distribute cracking more evenly. However, holistic indexing can materialize an even bigger advantage. This is because it is not restricted to perform auxiliary index refinement actions only during user queries but it can exploit all possible CPU cycles to refine the indices, resulting in many more actions taking place in parallel with user queries. Moreover, holistic indexing spreads the auxiliary index refinements across the entire value domain (by choosing random pivots) without leaving big unindexed pieces. For example, in the skewed workload in Figure 4.9(b), both multi-core database cracking and multi-core stochastic cracking show a similar performance,



(A) TPC-H Query 1 (B) TPC-H Query 6 (C) TPC-H Query 12

FIGURE 4.13: TPC-H results (Scale Factor 10, "pre-sorted" times exclude pre-sorting costs; Q1,6,12: 8 sec).

because they restrict the index refinements to a small region of the domain according to user query predicates, i.e, from 800 million to 2^{30}. Future queries have to reorganize a big unindexed area, i.e., from 0 to 800 million; this area is already indexed in holistic indexing before the 800th query arrives. Thus, holistic indexing prepares the physical design better for (ad-hoc) future queries.

SkyServer. The SkyServer workload in Figure 4.9(e) shows the pattern logged in SkyServer for queries using the "right ascension" attribute of the "Photoobjall" table. We observe that the queries follow non-random patterns, i.e., they focus on a specific part of the sky before moving to a different part. Figure 4.11 shows that holistic indexing manages to significantly outperform multi-core database cracking by inducing auxiliary index refinement actions in parallel with query processing without penalizing individual user queries.

Overall, in all workloads tested, holistic indexing not only maintains the nice properties of the parallel variants of database cracking and stochastic cracking, but it also enhances the behavior further by being able to exploit all available CPU resources effectively for a better prepared physical design.

4.3.4 More Benefits with Complex Schemas

In this experiment, we show that as the database schema becomes more complex by containing more attributes, this brings more opportunities for holistic indexing to gain in performance; more attributes make the indexing space bigger and thus any a priori decisions are even more prone to be wrong. In addition, we test the various strategies for choosing among the candidate indices and we show that indeed making random decisions is a robust approach.

Here, we assume a gradually bigger database table which consists of 5-10 attributes. Each attribute consists of 2^{30} uniformly distributed integers. We fire select-project queries as in the previous experiments but this time we may query up to X attributes in

every run. Each query touches a single attribute and we vary the frequency with which each attribute is accessed, i.e., we run both a random workload where every attribute is evenly queried as well as a skewed workload where some attributes are queried more than others. For each workload we vary also the workload pattern followed by the queries. Here, we present the results for random and periodic workload patterns. For each case, we perform 10^3 queries.

We compare holistic indexing using one of the four strategies described in Section 4.2.2 against multi-core variants of database cracking and stochastic cracking. Figure 4.12 shows the results; for all cases, holistic indexing materializes a big benefit. As the number of attributes in the database table grows, the performance benefit for holistic indexing increases. What happens is that holistic indexing makes sure to evenly spread auxiliary index refinement actions across all attributes in parallel with user queries whenever free CPU cycles are available. Then, future queries on those attributes can exploit this refined indexing. Compared to the case where we have a few attributes, having more attributes means that more heavy indexing actions have to be performed overall in order to crack the columns into small pieces. This allows holistic indexing to materialize a bigger benefit as it performs those actions in the background as opposed to only during user queries as in multi-core database cracking and multi-core stochastic cracking.

In addition, all index choosing strategies have similar performance on workloads where attributes are queried on random values (Figures 4.12(a) and (c)), because indices are already fine-grained in such cases (even when some indices are refined more than others in a skewed workload). However, in case of queries on periodic values (Figures 4.12(b) and (d)) the random choice ($W4$) shows a clear performance benefit compared to the rest of the strategies, because it refines indices with big unindexed partitions, and proves to be a robust design decision.

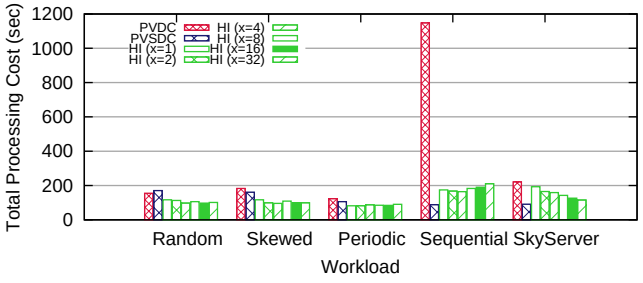


FIGURE 4.14: Performance of holistic indexing improves while the number of index refinements x increases.

4.3.5 Design Decisions

Holistic Worker Thread Refinements. In this experiment we demonstrate how the tuning parameter x, i.e., the number of index refinements per worker (Figure 4.1), affects the workload performance. We test five workloads that consist of 10^3 queries on a relation with 10 attributes as in Section 4.3.3 (Figure 4.11). We vary the number of index refinements each holistic worker thread does from 1 to 32 and we compare holistic indexing with multi-core variants of database cracking and stochastic cracking. Figure 4.14 shows the results. The more index refinements each thread does, the bigger the benefit for holistic indexing because more pieces are created and thus future queries need to refine smaller pieces, touching less data. However, when we increase the number of index refinements from 16 to 32, performance does not improve significantly, because in both cases indices converge very fast to optimal ones. Thus, we use 16 as the number of index refinements that each holistic worker thread does in all our experiments.

4.3.6 TPC-H

In our next experiment, we evaluate holistic indexing on the standard database benchmark, TPC-H [59]. We compare against offline indexing and relying on plain scans. We use scale factor 10 and we test with Queries 1, 6, and 12. For each query type, we created a sequence of 30 variations using the random query generator distributed with the benchmark. For offline indexing, we created the proper column-store projections by pre-sorting the data depending on each query individually, i.e., we created the perfect projection for each query. Specifically, for Queries 1 and 6 we created a copy of the Lineitem table sorted on the l_shipdate attribute. For Query 12 we created a copy of the Lineitem table sorted on the l_receiptdate attribute.

Figure 4.13 depicts the results. For all cases, holistic indexing brings a significant advantage, resulting in a robust and stable performance across all queries. The first query is slower as it creates the first adaptive indices which implies extra data copying but after that all queries perform significantly better than plain MonetDB. Holistic indexing matches offline indexing without having to incur the high offline indexing cost and without requiring any workload knowledge. The pre-sorting cost for all queries is 8 seconds. For Query 12, it turns out that pre-sorting does not help. This happens because even though we may improve the selection by pre-sorting the Lineitem table, it turns out we hurt the join between the Lineitem and the Orders table. This is because in the base data, the Lineitem table contains the order date ordered and this can be exploited during the join. With holistic indexing we do not face this problem, because the initial order changes only partially.

4.3.7 Updates

So far we tested read-only workloads. In this experiment we demonstrate that holistic indexing maintains its nice properties in workloads where read-only queries interleave with write queries. We test two scenarios. In the first scenario (High Frequency Low Volume - HFLV), 10 inserts arrive every 10 queries. In the second scenario (Low Frequency High Volume - LFHV), 100 inserts arrive every 100 queries. In both scenarios the workload consists of 500 select project range queries on a single attribute A and 500 insertions in total on a single attribute A. While all queries are processed sequentially one after the other, the 11th query arrives 20 seconds after the 10th query resulting in idle time of 20 seconds in the system. Attribute A consists of 10^9 uniformly distributed integers.

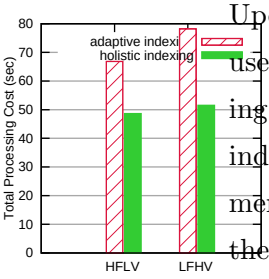


FIGURE 4.15: Updates.

Updates are temporarily stored in a pending insertions column. We use the Ripple algorithm [13] in order to apply the updates; a pending inserted value is merged with the original data if and only if the index partition, where the specific value belongs, is refined. Thus, the merging process happens on-the-fly and maintains the information of the index. In this experiment we test single-threaded adaptive indexing against holistic indexing with a single worker that refines the index only during idle time. In holistic indexing, auxiliary index refinement actions also cause insertions to be merged. In this way, holistic indexing not only refines the indices but also consumes pending insertions, which speeds up future queries even more and all that by exploiting idle CPU resources in parallel with query processing. Figure 4.15 shows the results. In both scenarios, holistic indexing maintains its advantage over adaptive indexing; it is not affected by updates and still provides roughly a 50% improvement.

4.3.8 Varying Number of Clients

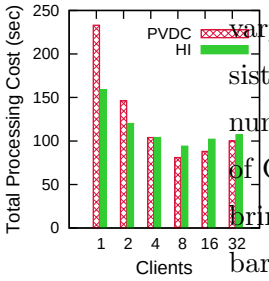


FIGURE 4.16: Varying number of clients.

In this experiment we test the performance of holistic indexing with varying number of concurrent clients in the system. Our workload consists of 1024 queries on a relation with 10 attributes. We vary the number of concurrent clients between 1 and 32, where 32 is the number of CPU cores in our machine. Figure 4.16 shows that holistic indexing brings a big benefit in case of a few clients. The labels on top of the bars indicate the distribution of the available threads across user queries and holistic workers (similar to Figure 4.6). When the number of clients increases, holistic indexing does not bring significant benefits, because

it easily detects these cases as it monitors the CPU load continuously and so it is triggered only if the load is below a threshold.

Bibliography

[1] Sanjay Agrawal, Surajit Chaudhuri, Lubor Kollár, Arunprasad P. Marathe, Vivek R. Narasayya, and Manoj Syamala. Database Tuning Advisor for Microsoft SQL Server 2005. In *Proceedings of the 30th International Conference on Very Large Data Bases (VLDB)*, pages 1110–1121, 2004.

[2] Benoît Dageville, Dinesh Das, Karl Dias, Khaled Yagoub, Mohamed Zaït, and Mohamed Ziauddin. Automatic SQL Tuning in Oracle 10g. In *Proceedings of the 30th International Conference on Very Large Data Bases (VLDB)*, pages 1098–1109, 2004.

[3] Daniel C. Zilio, Jun Rao, Sam Lightstone, Guy M. Lohman, Adam J. Storm, Christian Garcia-Arellano, and Scott Fadden. DB2 Design Advisor: Integrated Automatic Physical Database Design. In *Proceedings of the 30th International Conference on Very Large Data Bases (VLDB)*, pages 1087–1097, 2004.

[4] Surajit Chaudhuri and Vivek R. Narasayya. AutoAdmin "What-if" Index Analysis Utility. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 367–378, 1998.

[5] Surajit Chaudhuri and Vivek R. Narasayya. An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server. In *Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB)*, pages 146–155, 1997.

[6] Karl Schnaitter, Serge Abiteboul, Tova Milo, and Neoklis Polyzotis. COLT: Continuous On-Line Tuning. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 793–795, 2006.

[7] Nicolas Bruno and Surajit Chaudhuri. An Online Approach to Physical Design Tuning. In *Proceedings of the IEEE 23rd International Conference on Data Engineering (ICDE)*, pages 826–835, 2007.

[8] Martin Lühring, Kai-Uwe Sattler, Karsten Schmidt, and Eike Schallehn. Autonomous Management of Soft Indexes. In *Proceedings of the 2nd International Workshop on Self-Managing Data Bases (SMDB)*, pages 450–458, 2007.

[9] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. Database Cracking. In *Proceedings of the 3rd Biennial Conference on Innovative Data Systems Research (CIDR)*, pages 68–78, 2007.

[10] Felix Martin Schuhknecht, Alekh Jindal, and Jens Dittrich. The Uncracked Pieces in Database Cracking. *Proceedings of the Very Large Data Bases Endowment (PVLDB)*, 7(2):97–108, 2013.

[11] Stefan Richter, Jorge-Arnulfo Quiané-Ruiz, Stefan Schuh, and Jens Dittrich. Towards Zero-Overhead Static and Adaptive Indexing in Hadoop. *The Very Large Data Bases Journal (VLDB J.)*, 23(3):469–494, 2013.

[12] Goetz Graefe and Harumi A. Kuno. Self-Selecting, Self-Tuning, Incrementally Optimized Indexes. In *Proceedings of the 13th International Conference on Extending Database Technology (EDBT)*, pages 371–381, 2010.

[13] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. Updating a Cracked Database. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 413–424, 2007.

[14] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. Self-Organizing Tuple Reconstruction in Column-Stores. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 297–308, 2009.

[15] Victor Alvarez, Felix Martin Schuhknecht, Jens Dittrich, and Stefan Richter. Main Memory Adaptive Indexing for Multi-core Systems. In *Proceedings of the 10th International Workshop on Data Management on New Hardware (DaMoN)*, 2014.

[16] Goetz Graefe, Felix Halim, Stratos Idreos, Harumi A. Kuno, and Stefan Manegold. Concurrency Control for Adaptive Indexing. *Proceedings of the Very Large Data Bases Endowment (PVLDB)*, 5(7):656–667, 2012.

[17] Goetz Graefe, Felix Halim, Stratos Idreos, Harumi A. Kuno, Stefan Manegold, and Bernhard Seeger. Transactional Support for Adaptive Indexing. *The Very Large Data Bases Journal (VLDB J.)*, 23(2):303–328, 2014.

[18] Stratos Idreos, Stefan Manegold, Harumi A. Kuno, and Goetz Graefe. Merging What's Cracked, Cracking What's Merged: Adaptive Indexing in Main-Memory Column-Stores. *Proceedings of the Very Large Data Bases Endowment (PVLDB)*, 4(9):585–597, 2011.

[19] Goetz Graefe, Stratos Idreos, Harumi A. Kuno, and Stefan Manegold. Benchmarking Adaptive Indexing. In *Proceedings of the 2nd Technology Conference on Performance Evaluation and Benchmarking (TPCTC)*, pages 169–184, 2010.

[20] Felix Halim, Stratos Idreos, Panagiotis Karras, and Roland H. C. Yap. Stochastic Database Cracking: Towards Robust Adaptive Indexing in Main-Memory Column-Stores. *Proceedings of the Very Large Data Bases Endowment (PVLDB)*, 5(6): 502–513, 2012.

[21] Goetz Graefe and Harumi Kuno. Adaptive Indexing for Relational Keys. In *Workshops Proceedings of the IEEE 26th International Conference on Data Engineering (ICDE)*, pages 69–74, 2010.

[22] Holger Pirk, Eleni Petraki, Stratos Idreos, Stefan Manegold, and Martin L. Kersten. Database Cracking: Fancy Scan, not Poor Man's Sort! In *Proceedings of the 10th International Workshop on Data Management on New Hardware (DaMoN)*, 2014.

[23] Kostas Zoumpatianos, Stratos Idreos, and Themis Palpanas. Indexing for Interactive Exploration of Big Data Series. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1555–1566, 2014.

[24] Ioannis Alagiannis, Stratos Idreos, and Anastasia Ailamaki. H2O: A Hands-free Adaptive Store. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1103–1114, 2014.

[25] Stratos Idreos, Ioannis Alagiannis, Ryan Johnson, and Anastasia Ailamaki. Here are my Data Files. Here are my Queries. Where are my Results? In *Proceedings of the 5th Biennial Conference on Innovative Data Systems Research (CIDR)*, pages 57–68, 2011.

[26] Stratos Idreos and Erietta Liarou. dbTouch: Analytics at your Fingertips. In *Proceedings of the 6th Biennial Conference on Innovative Data Systems Research (CIDR)*, 2013.

[27] Erietta Liarou and Stratos Idreos. dbTouch in Action Database Kernels for Touch-based Data Exploration. In *Proceedings of the IEEE 30th International Conference on Data Engineering (ICDE)*, pages 1262–1265, 2014.

[28] Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. Near-Optimal Cache Block Placement with Reactive Nonuniform Cache Architectures. In *IEEE Micro*, page 29, 2010.

[29] Stavros Harizopoulos and Anastassia Ailamaki. A Case for Staged Database Systems. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, 2003.

[30] Stavros Harizopoulos, Vladislav Shkapenyuk, and Anastassia Ailamaki. QPipe: A Simultaneously Pipelined Relational Query Engine. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 383–394, 2005.

[31] Ryan Johnson, Ippokratis Pandis, Nikos Hardavellas, Anastasia Ailamaki, and Babak Falsafi. Shore-MT: A Scalable Storage Manager for the Multicore Era. In *Proceedings of the 12th International Conference on Extending Database Technology (EDBT)*, pages 24–35, 2009.

[32] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M. Tamer Özsu. Multi-Core, Main-Memory Joins: Sort vs. Hash Revisited. *Proceedings of the Very Large Data Bases Endowment (PVLDB)*, 7(1):85–96, 2013.

[33] Changkyu Kim, Eric Sedlar, Jatin Chhugani, Tim Kaldewey, Anthony D. Nguyen, Andrea Di Blas, Victor W. Lee, Nadathur Satish, and Pradeep Dubey. Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-Core CPUs. *Proceedings of the Very Large Data Bases Endowment (PVLDB)*, 2(2):1378–1389, 2009.

[34] Orestis Polychroniou and Kenneth A. Ross. A Comprehensive Study of Main-memory Partitioning and Its Application to Large-scale Comparison- and Radix-sort. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 755–766, 2014.

[35] Jingren Zhou and Kenneth A. Ross. Implementing Database Operations Using SIMD Instructions. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 145–156, 2002.

[36] Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, William Macy, Mostafa Hagog, Yen-Kuang Chen, Akram Baransi, Sanjeev Kumar, and Pradeep Dubey. Efficient Implementation of Sorting on Multi-Core SIMD CPU Architecture. In *PVLDB*, 2008.

[37] Mike Johnson. *Superscalar Microprocessor Design.* Prentice Hall, 1991.

[38] John L. Hennessy and David A. Patterson. *Computer Architecture - A Quantitative Approach (5. ed.).* Morgan Kaufmann, 2012.

[39] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and David A. Wood. DBMSs on a Modern Processor: Where Does Time Go? In *VLDB*, 1999.

[40] Peter A. Boncz, Marcin Zukowski, and Niels Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR*, 2005.

[41] John Cieslewicz and Kenneth A. Ross. Adaptive Aggregation on Chip Multiprocessors. In *VLDB*, 2007.

[42] C. A. R. Hoare. Algorithm 64: Quicksort. *Commun. ACM*, 4(7), 1961.

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

[44] Kenneth A Ross. Selection Conditions in Main Memory. In *ACM Transactions on Database Systems (TODS)*, 2004.

[45] H. V. Jagadish, Adriane Chapman, Aaron Elkiss, Magesh Jayapandian, Yunyao Li, Arnab Nandi, and Cong Yu. Making Database Systems Usable. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 13–24, 2007.

[46] Martin L. Kersten, Stratos Idreos, Stefan Manegold, and Erietta Liarou. The Researcher's Guide to the Data Deluge: Querying a Scientific Database in Just a Few Seconds. *Proceedings of the Very Large Data Bases Endowment (PVLDB)*, 4(12): 1474–1477, 2011.

[47] Udayan Khurana and Amol Deshpande. HiNGE: Enabling Temporal Network Analytics at Scale. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1089–1092, 2013.

[48] Arun Kumar, Feng Niu, and Christopher Ré. Hazy: Making it Easier to Build and Maintain Big-Data Analytics. *Communications of the ACM*, 56(3):40–49, 2013.

[49] Nikolay Laptev, Kai Zeng, and Carlo Zaniolo. Very Fast Estimation for Result and Accuracy of Big Data Analytics: The EARL System. In *Proceedings of the IEEE 29th International Conference on Data Engineering (ICDE)*, pages 1296–1299, 2013.

[50] Jens Dittrich, Jorge-Arnulfo Quiané-Ruiz, Alekh Jindal, Yagiz Kargin, Vinay Setty, and Jörg Schad. Hadoop++: Making a Yellow Elephant Run Like a Cheetah (Without It Even Noticing). *Proceedings of the Very Large Data Bases Endowment (PVLDB)*, 3(1):518–529, 2010.

[51] Ioannis Alagiannis, Renata Borovica, Miguel Branco, Stratos Idreos, and Anastasia Ailamaki. NoDB: Efficient Query Execution on Raw Data Files. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 241–252, 2012.

[52] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alex Rasin, Stanley B. Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. H-store: A High-Performance, Distributed Main Memory Transaction Processing System. *Proceedings of the Very Large Data Bases Endowment (PVLDB)*, 1(2):1496–1499, 2008.

[53] Alfons Kemper and Thomas Neumann. HyPer: A Hybrid OLTP&OLAP Main Memory Database System Based on Virtual Memory Snapshots. In *Proceedings of the IEEE 27th International Conference on Data Engineering (ICDE)*, pages 195–206, 2011.

[54] Yinan Li and Jignesh M. Patel. BitWeaving: Fast Scans for Main Memory Data Processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 289–300, 2013.

[55] Stratos Idreos, Fabian Groffen, Niels Nes, Stefan Manegold, K. Sjoerd Mullender, and Martin L. Kersten. MonetDB: Two Decades of Research in Column-oriented Database Architectures. *IEEE Data Engineering Bulletin*, 35(1):40–45, 2012.

[56] ZZZ. *MonetDB*. `www.monetdb.org`.

[57] Sort-Merge Joins. http://www.systems.ethz.ch/projects/paralleljoins.

[58] Sloan Digital Sky Survey (SkyServer). http://cas.sdss.org/.

[59] TPC Benchmark H. http://www.tpc.org/tpch/.

List of Figures

List of Tables

Abbreviations

LAH **L**ist **A**bbreviations **H**ere

Summary

Samenvatting

Acknowledgements

SIKS Dissertation Series