Closing the Circle of Algorithmic and System-centric Database Optimization:

A Comprehensive Survey on Adaptive Indexing, Data Partitioning, and the Rewiring of Virtual Memory

Felix Martin Schuhknecht

A dissertation submitted towards the degree
Doctor of Engineering (Dr.-Ing.)
of the Faculty of Mathematics and Computer Science
of Saarland University

Saarbrücken, Germany
August 2016

To my family

Acknowledgements

First and foremost, I would like to thank my advisor Jens Dittrich for his guidance and friendship throughout all the years. Under his supervision, I had the chance to do research on highly interesting up-to-date topics. He allowed me the freedom to follow my own ideas and gave me the time to realize them, which is something I highly appreciate and that can not be taken for granted. At the same time, he has the talent to guide me into the right direction every time I was struggling or unsure how to proceed and what to focus on. I always appreciated his strict attitude on correctness and his style of explicit writing, that I tried to apply in the following thesis. Besides, he was a great teacher in how to present research in the right way — informative and precise, but entertaining and exiting at the same time. Apart from the research supervision, I would also like to thank him for the support and for finding the right words during the harder times of my life.

I would also like to thank the professors Wolfgang Lehner and Anastasia Ailamaki for agreeing to review this thesis. It is a pleasure and an honor for me to have a board of such outstanding researchers.

Further I would like to thank my co-authors, with whom I worked on the papers that contribute to this thesis. Thanks a lot to Alekh Jindal, who was a great advisor during my Bachelor's thesis back in 2010 and a colleague to look up to during my early PhD studies. I greatly enjoyed our work on my very first paper "How Achaeans Would Construct Columns in Troy" as well as the research on our awarded paper "The Uncracked Pieces in Database Cracking", that forms a large part of this thesis. I also would like to thank Victor Alvarez, who worked with me on "Main Memory Adaptive Indexing for Multi-core Systems", and showed me a more theoretical perspective on the things. Further, thanks to Pankaj Khanchandani for the great work on the mini-paper "On the Surprising Difficulty of Simple Things: the Case of Radix Partitioning", while he was still in his regular studies. He also participated in the early phase of the work on "RUMA has it: Rewired User-space Memory Access is Possible!". In this regard, I would like to thank Ankur Sharma for the work on that paper as well.

Besides, I would like to thank Endre Palatinus, Stefan Richter, Stefan Schuh, Ankur Sharma, Daniel Gnad, Alvaro Torralba, Marcel Steinmetz, and Laurent Linden for becoming great friends. Thanks to Stefan Richter for the nice conference travels we did together. Thanks for thrilling table tennis matches, table soccer plays, and gaming sessions. Further, I would like to thank our secretary Angelika Scholl-Danopoulos for handling all the administrative paperwork, that I simply don't like to do.

Finally, I would like to thank my mother for supporting me at any time. Without her this thesis would not exist.

The research work presented in this thesis was partially supported by the funding from the German Ministry of Education and Science (BMBI).

Abstract

Over the decades, with the increase of computing resources, the amount of data to manage also increased tremendously. Besides of the sheer quantity of information, the quality of it highly varies today.

Indexing all this data with equal effort is cumbersome and wasteful. Thus, *adaptive indexing* algorithms refine parts of interest more carefully. Unfortunately, the adaptivity also introduces a set of new problems. High variance in response times and low robustness against certain workloads are just two issues to mention. A vast amount of methods have been proposed to deal with these problems. Thus, in the first part of this thesis, we will *reinvestigate*, *analyze*, and *enhance* the class of adaptive indexing methods in a comprehensive evaluation on the *algorithmic level*. In total, we discuss 18 cracking methods, 6 sorting algorithms, and 3 full index structures, including our own proposed methods.

Consequently, we identify *data partitioning* as the common component. Thus, in the second part, we analyze the surprising amount of optimizations possible to enhance partitioning. Interestingly, they mostly originate from a more sophisticated mapping of the method to the system properties, thus shifting our perspective to a *system-centric view*.

Subsequently, in the third part, we dig down to the ground level by exploiting a core feature of any modern operating system, the *virtual memory system*. We investigate how virtual and physical memory can be separated in user space and how the mappings between the two memory types can be rewired freely at runtime. Using *rewiring*, we are able to significantly enhance core applications of data management systems.

Finally, we apply the techniques identified in this thesis to the initial adaptive indexing algorithm to significantly improve it — and close the circle.

Zusammenfassung

Im Laufe der Jahrzehnte kam es neben dem Anstieg der zur Verfügung stehenden Berechnungsressourcen auch zu einem heftigen Anstieg der zu verwaltenden Datenmengen. Abgesehen von der schieren Größe der Informationsmenge variiert heutzutage auch die Qualität dieser in großem Maße.

Die Datenmenge mit gleichverteiltem Aufwand zu indizieren ist ein mühseliges und verschwenderisches Unterfangen. Daher investiert die Klasse der *adaptiven Indizes* mehr Indizierungsaufwand auf Bereiche von Interesse. Leider bringt adaptive Indizierung auch einige neue Probleme mit sich, die es zu handhaben gilt. Große Varianz in der Laufzeit und schwache Robustheit gegenüber gewissen Anfragemustern sind nur zwei der zu benennenden Schwierigkeiten. Um diesen Problemen entgegenzuwirken kam es in der Vergangenheit zur Entwicklung einer großen Anzahl verschiedenster Algorithmen. Im ersten Teil dieser Dissertation werden wir daher die Klasse der adaptiven Indizes in einer allumfassenden Evaluierung auf *algorithmischer Ebene* neu *untersuchen*, *analysieren* und *erweitern*. Insgesamt behandelt diese Auswertung 18 verschiedene Cracking-Methoden, 6 Sortieralgorithmen und 3 traditionelle Indexstrukturen, inklusive unserer eigenen Algorithmen.

Auf Basis dieser Untersuchungen identifizieren wir *Datenpartitionierung* als gemeinsame Komponente. Daher analysieren wir im zweiten Teil dieser Dissertation die überraschend große Anzahl an möglichen Optimierungen zur Verbesserung der Partitionierungsphase. Interessanterweise entspringen diese hauptsächlich einer ausgeklügelteren Abbildung des Problems auf die Gegebenheiten des zu Grunde liegenden Systems. Daher verlagern wir unsere Perspektive auf eine *system-zentrische Ebene*.

Darauffolgend gelangen wir im dritten Teil dieser Dissertation auf die unterste Ebene der Betrachtung, indem wir uns der Ausnutzung eines elementaren Bestandteiles eines jeden modernen Betriebssystems widmen, der *virtuellen Speicherverwaltung*. Wir untersuchen wie virtueller und physischer Speicher in der Benutzerumgebung voneinander getrennt werden können und wie die Abbildungen zwischen den beiden Speichertypen zur Laufzeit manipuliert und neu verdrahtet werden können. Mit Hilfe dieser Technik sind wir in der Lage, Kernapplikationen von Datenbanksystemen nachhaltig zu verbessern.

Abschließend wenden wir die im Laufe dieser Arbeit identifizierten Methoden auf den initialen adaptiven Indizierungsalgorithmus an und verbessern diesen signifikant — um den Kreis zu schließen.

Contents

Chapter 1

Introduction

1.1 Motivation

Managing data in an efficient way in terms of maintenance and access time is one of the core challenges in computer science.

In the early days of information processing, where computing power and memory space were extremely scarce, algorithms and data structures were constructed very carefully to utilize the precious resources as cautious as possible. Nowadays, even a general purpose desktop computer offers a vast amount of computing power in form of one or multiple CPUs, each consisting of several computing cores, and has gigabytes of main-memory installed. One could think that these nice circumstances solve most of the problems that software engineers faced in the early days. However, this is clearly not the case.

With the increase of computing capacities, the amount of data to manage and to process also increased tremendously. Systems have to handle tables with billions of records that keep changing continuously at a high rate. At the same time, an increasing amount of users is accessing the data concurrently, demanding the results of their queries within milliseconds. Thus, a careful engineering of the applied algorithms and data structures is still as crucial as ever and actually demands for new types of data management techniques.

But what does careful engineering actually mean? We can basically differentiate between two different perspectives, when optimizing methods: the algorithmic view and the system-centric view. The former one treats the algorithm as a mathematical construct with the aim to optimize the method with respect to some complexity metric. For instance, the optimization goal could be to minimize the amount of elements the algorithm has to copy during processing. The latter view treats the algorithm as a physical instance that is executed on a real system. The main optimization goal is to map the algorithm more efficiently to the properties of the underlying hardware and the operating system. As we will see, a combination of both views is necessary to get the most out of a method. Throughout this thesis, we will perform a shift of perspective. We will start with the algorithmic perspective at the case of adaptive indexing.

1.1.1 The Lazy Approach — Adaptive Indexing

The vast amount of data that is produced all day introduces another phenomenon of modern data management: the omnipresence of useless, uninteresting, or even faulty data. For instance, via social media, users constantly produce posts and upload images or videos, that may be queried a million times or not at all — in any case, the system underneath has to store, to manage, and to provide the content. Another example are ubiquitous sensors, with their presence ranging all over from cars to phones, that constantly log various kinds of measurements of their surrounding. They do so not necessarily because there is a need to store all this information, the main driving force is to gather all the data because it is *possible*.

This attitude leads to the demand of new forms of data management software. To index and search this kind of mixture of important, unimportant, and garbaged information, new types of data structures are needed. The classical index structure, that treats all entries with equal priority, seems too wasteful in this scenario. It is not very efficient to index a billion entries element-wise, as for instance done by a classical B-Tree, if 99% of the data will never be queried or even thrown away tomorrow because it is already outdated. Instead, we want to focus our indexing effort on the 1% that is of interest and ignore the rest. This is the purpose of *adaptive indexing*, the first major topic of this thesis.

This mentality is the main driving force for the class of adaptive indexing algorithms, that came up in the last decade. Instead of building the entire index upfront, it is incrementally refined as a side product of query processing. Instead of indexing each key individually, only key ranges are indexed in a coarse-granular fashion.

As only the individual queries fired by the users can tell the system which parts of the data are important respectively interesting, adaptive index maintenance can only happen during query answering. Besides of focusing on interesting parts of the data, adaptive indexing also manages to divide the work of building up the index into many small parts, instead of a single huge one. This reduces the pressure on the system during index maintenance. Figure 1.1 visualizes the difference between traditional and adaptive indexing. Traditional indexing techniques put a high load on the system during the long phase of initial index creation. In comparison to that, adaptive indexing has a very short initialization phase and slowly improves the index query by query.

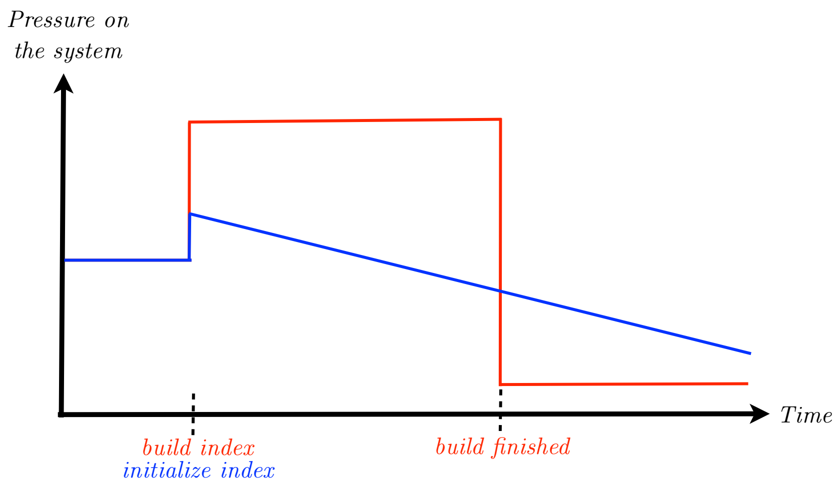


Figure 1.1: **Traditional Indexing** (build the index at once) vs **Adaptive Indexing** (build the index incrementally during query processing).

Within the last decade, a large amount of work has been published on adaptive indexing, showing the need and interest for this class of algorithms. The largest part of the research in this field originates from the group of original inventors of *database cracking*, the main representative of adaptive indexing, CWI Amsterdam. As different workloads and situations require different algorithms, they proposed a vast amount of methods that try to deal with the classical adaptive indexing issues *convergence speed*, *response time variance*, and *tuple reconstruction performance*.

After studying the literature [15, 39, 14, 17, 23, 22, 20, 21], that tackles these issues mostly one by one in an independent fashion, the question remains how these algorithms compare against each other in a unified environment under a common setup. Which algorithms are really the best to use in practice? Is there a need for such a large number of different algorithms — and how different are they actually?

Thus, a major part of this thesis we will spent on the study, improvement, and extension of adaptive indexing algorithms. We will reinvestigate the large set of existing algorithms and try to reproduce the results obtained by the original authors. Further, we will try to improve the methods and show new directions on how to approach existing issues of the field. Additionally, we will test the techniques under setups that have not been tried and discussed so far, to get new insights into the usability of adaptive indexing and how to avoid the pitfalls.

In Section 1.2.1 and 1.2.2, we outline the detailed contributions of our work in this field. To ease the discussion, we clearly separate the single-threaded and multi-threaded algorithms and look at them individually.

1.1.2 Invest It All — Sorting Data

Discussing adaptive indexing is impossible without the discussion of classical *sorting* algorithms. Sorting can be considered as the most unbalanced form of adaptive indexing — all the reorganization is done in a single query. It can also be seen as the most basic form of traditional indexing. If the keys are perfectly sorted, a queried key can be located via binary search in logarithmic complexity.

This is why throughout this thesis, sorting algorithms will not only serve as the baseline to compete with for any adaptive indexing method, but also as a possible alternative. Having the index in a sorted state offers pleasant advantages in query optimization, such as enabling cheap sort merge joins, apart from fast and stable index lookup times.

Of course, sorting algorithms are basically as old as computer science itself and thus, the area has been very well researched both from the theoretical and the practical perspective. Interestingly, as we will see throughout this thesis, there is still room for improvements. These do no longer result from deep algorithmic modifications, but mainly from a more sophisticated mapping to the underlying system and hardware. For instance, to fully utilize all computing cores across multiple sockets, the sorting algorithm must be made aware of the structure of the system and schedule the work accordingly.

In Section 1.2.1 (single-threaded) and Section 1.2.2 (multi-threaded), we outline the contributions we make in this thesis regarding sorting methods.

1.1.3 The Core of Indexing — Data Partitioning

If we analyze both adaptive indexing, with its main representative of database cracking, as well as sorting methods, then we realize that all of them are built upon a single common component — they basically apply a form of *data partitioning*. While adaptive indexing algorithms partition the data only with respect to the seen queries, sorting methods divide the data into a partition per unique key.

With that omnipresence of partitioning in mind, a reasonable decision is to improve the raw algorithm as much as possible. This seems hard to achieve due to the simplicity of the procedure, which does not really offer many options to pick from. Especially the out-of-place version is extremely primitive as it solely consists of a first pass counting the partition sizes and a second pass copying the entries to the designated partitions.

Again, algorithmically not much can be done to improve the performance. Nevertheless, it turns out that surprisingly many low-level optimizations can be applied to significantly enhance the process of partitioning, without changing its algorithmic nature. With an awareness of certain operating system features and the underlying hardware characteristics, the raw out-of-place algorithm can be mapped significantly more efficient to the system.

Our detailed investigation of these techniques clearly shows the path this thesis is trailing: the transition from an algorithm-centric to a system-centric optimization.

In Section 1.2.3 we discuss the contributions we make in this thesis in the field of data partitioning.

1.1.4 The Basis of Data Management — Virtual Memory

Our work on the improvement of data partitioning clearly demonstrates the need for close-to-the-system programming when it comes to processing speed. We want to push this paradigm of system centric optimization further by an investigation of the memory management system, that is utilized by basically every algorithm and data structure out there.

A very interesting system part that is present in state-of-the-art operating systems is the virtual memory management. Instead of directly delivering physical memory to the user, the operating system grants only access to virtual memory, that is internally mapped to physical one in a completely transparent fashion. While this has many convenient advantages like process separation and over-allocation[1], it adds a level of indirection between virtual and physical memory pages. Further, by default, the mapping between the layers is not modifiable by the user.

Thus, we will propose a way how to reintroduce physical memory into user space again — without any modifications of the operating system kernel. With both virtual and physical memory objects at hand, the user is able to manipulate the mapping between these types of memory during runtime. We coin this technique

[1]We refer to over-allocation as the process of virtually allocating more memory than the system physically has.

rewiring memory. As we will see, this method can be used in various data management algorithms.

In Section 1.2.4 we will outline the detailed contributions we make regarding the rewiring of memory.

1.2 Contributions

In this thesis, we analyze, evaluate, and extend fundamental components of database systems.

We start with a thorough analysis of the adaptive indexing paradigm by reevaluating, questioning, and extending the recent work in the field. In Chapter 2, we focus on *single-threaded adaptive indexing* entirely. Consequently, in Chapter 3, we present how adaptive indexing can be realized in a *multi-threaded* environment. In both Chapters 2 and 3, *sorting methods*, as a core building block of classical indexing, are always present as well and are discussed as counterparts to adaptive indexing.

Based on the obtained results, in Chapter 4 we dig deeper by investigating a core building block of any adaptive indexing and sorting algorithm, namely *data partitioning*, and investigate how it can be mapped efficiently to the capabilities of modern hardware.

In Chapter 5, we completely focus on the system-centric perspective by digging even deeper to the memory management and present *rewired memory*, a fundamental technique that enables the improvement of algorithms and data structures solely by exchanging the memory management system.

Finally, in Chapter 6 we wrap up the results and close the circle of optimization by applying the most valuable techniques discussed in this thesis onto the initial adaptive indexing algorithm to improve it significantly.

1.2.1 Single-threaded Adaptive Indexing and Sorting

Within the last decade, a large amount of publications appeared that cover the topic of adaptive indexing. They introduced the core technique of adaptive indexing in the form of *database cracking* respectively *standard cracking* [20, 26] as well as refining the concept in several ways to address certain problems that appeared along the way. To improve the convergence speed, a set of so called *hybrid cracking* methods was introduced [23]. To reduce the variance of query response times, *stochastic cracking* [17] was proposed that introduces random decisions into the index evolution. Efficient tuple reconstruction in the context of database cracking was addresses by *sideways cracking* [22]. *Predication cracking* [39] tried to reduce the penalty caused by branch misprediction by proposing a branch-free version of standard cracking.

While all of these works significantly contributed to the overall understanding of adaptive indexing algorithms and their applicability to the individual use-cases, we clearly felt the need for a reevaluation and analysis of all these techniques in a single environment.

Thus, in Chapter 2, we combine the work of 5 major publications in the field of adaptive indexing and reevaluate them in a single test framework. We first revisit the methods of the original authors and try to reproduce and confirm their results. This covers the reimplementation and reevaluation of standard cracking [20, 26], hybrid cracking [23] (in three variants), stochastic cracking [17] (MDD1R), predication and vectorized cracking [39], and sideways cracking [22].

Based on these insights, we propose new directions on how to approach the issues of convergence speed, variance, and tuple reconstruction. For each category, we propose one new technique that improves upon the existing leading method in that field. In terms of convergence speed, we propose *buffered swapping*, that buffers entries to swap in a heap before actually reordering them, to converge more per query. In terms of variance, we introduce a pre-partitioning step, coined *coarse-granular index*, that bulk loads the cracker index before the actual query answering. To improve tuple reconstruction, we propose *covered cracking*, that extends sideways cracking by the possibility to keep non-key attributes inside the cracker column. Accessing to these attributes through the index is therefore possible without expensive random jumps.

After extending the algorithms, we extend the experimental setup itself. We compare adaptive indexing against different sorting baselines, including quick sort and radix sort. We then test the impact of the indexing method above, including a B+ tree, an AVL tree, and the adaptive radix tree (ART) [31]. Further, we vary the selectivity and breakdown the cost of the individual parts of cracking in detail. We vary the query access pattern and the cracking depth. Finally, we classify all tested algorithms and present *signatures* of them that characterize their behavior.

In Chapter 2, we focus solely on single-threaded algorithms. The multi-threaded counterpart will be the topic of Chapter 3.

1.2.2 Multi-threaded Adaptive Indexing and Sorting

With the advent of the single-threaded adaptive indexing algorithms, multi-threaded versions were requested by the community as a logical consequence. However, until our investigation of the topic, only a single parallel version of standard cracking was proposed in [14, 15].

Based on the methods, results, and insights gained from the previous chapter, we present in Chapter 3 a set of new parallel versions of database cracking algorithms. We propose another parallel version of standard cracking, coined parallel-chunked standard cracking, that avoids the high amount of induced serialization in the first queries, as it is the case in the version in the literature. Connected to that, we test a parallel version using chunking of vectorized cracking. Additionally, we propose two parallel versions exploiting the coarse-granular index of the previous chapter. Further, we implement two parallel versions of sideways cracking using standard cracking and coarse-granular index underneath.

We evaluate these methods alongside with state-of-the-art parallel sorting-based methods. We show that full indexing is still a valid alternative — especially on highly parallel hardware.

We analyze the parallelization problems of parallel standard cracking and show why chunk-based parallelization is the only way to map nicely to the parallel hardware. Especially of interest is the scaling behavior of the individual algorithms — an ideal algorithm should be k-times faster on a machine with k physical computing cores compared to the single-threaded version. As the reality is different for almost all tested methods, we deeply analyze the reasons for that. We investigate the relationship of computing power to main memory bandwidth, the most important factor in understanding the scaling capabilities. Further, we compare the absolute runtimes and realize that for a high number of computing cores, the difference between fully sorting and adaptive indexing becomes almost negligible.

To complete this chapter, we also test the capabilities of parallel sideways cracking in comparison with a perfectly and lazily clustered full index. Finally, as in the single-threaded case, we close the investigation with a test of different query access patterns and data distributions.

Overall, in Chapter 3, we transport the concepts of adaptive indexing discussed in Chapter 2 into the world of state-of-the-art parallel hardware.

1.2.3 Understanding and Optimizing Data Partitioning

During the detailed investigation and evaluation of the numerous adaptive indexing and sorting algorithms, we were faced repeatedly with a common component — the data partitioning procedure, the basis of all indexing algorithms.

Therefore, in Chapter 4, we want to dig deeper and focus solely on analyzing and optimizing this very procedure. We focus here on the very standard two-pass out-of-place radix partitioning, as it is widely used in numerous situations in the database context. Interestingly, despite the simplicity of the original partitioning algorithm, surprisingly many optimizations can be applied to fit the process to the needs of modern hardware. These range from purely software-based optimizations to exploiting certain hardware properaties and operating system features.

We start with analyzing the impact of *software-managed buffers* [40, 45, 5, 52], that collect entries going to the same partition in a small buffer, that is flushed when it is full. We show the reduction on the TLB pressure, leading to a significantly improved runtime. Further, we investigate *non-temporal streaming stores* [11, 52, 40, 5], that can be used to bypass the caches when copying memory, further improving the performance in certain situations. We also test prefetching in multiple forms to further reduce the amount of cache and TLB misses. In the context of software-managed buffers, we investigate different possible memory layouts [40, 5] and the effect of storing working variables directly in the buffers. Finally, we test the impact of the underlying page size on the different versions of the algorithm. In all experiments, we vary the number of requested partitions as it drastically influences all versions. To complete the chapter, we scale the data and entry size.

As an overall conclusion, we show which *optimization path* is the most promising for a certain number of requested partitions — thus providing a strong guideline for any user of partitioning algorithms.

1.2.4 Rewiring Memory

In Chapter 4, a trend of modern computer science becomes visible: to improve an algorithm or a data structure in terms of performance, improving its algorithmic behavior becomes less important than mapping it carefully to the needs of the system. This becomes visible during the optimization of the partitioning algorithm and raises the question which parts of the system offer opportunities to be exploited by current algorithms and data structures.

Thus, in Chapter 5, we introduce the concept of rewiring memory, that manages to reintroduce physical memory into user-space. This is possible without any changes of the operating system kernel. We exploit the existing toolset of the standard linux and describe all components that are necessary to realize rewiring memory. This encloses the core components of rewiring: main-memory file systems and the system call `mmap`. We show how pooling can be supported efficiently.

After presenting the technique itself, we run a set of micro benchmarks to understand the behavior of rewired memory. We inspect different allocation types and analyze their pros and cons. Further, we investigate the cost of page faults and the overhead of remapping virtual to physical pages. Additionally, we analyze the effect of the allocation type on different access patterns.

With that knowledge, we then include rewiring into two applications, that are not only widely used in database management systems, but also used by developers of other communities. We propose a rewired vector, that resizes itself without any unnecessary physical copies. Further, we present rewired partitioning, that manages to generate a consecutive result array while avoiding a separate pass for histogram generation.

1.3 Publications and Awards

Subsets of the work presented in this thesis have been published in international conferences and journals and were presented therein. The chapters of this thesis are organized based on the following publications:

- **Chapter 2 — Single-threaded Adaptive Indexing and Sorting**

  Publications and Award:

  [48] Felix Martin Schuhknecht, Alekh Jindal, Jens Dittrich.
  The Uncracked Pieces in Database Cracking.
  VLDB 2014/PVLDB 2013, Hangzhou, China.
  VLDB 2014 Best Paper Award.

  [49] Felix Martin Schuhknecht, Alekh Jindal, Jens Dittrich.
  An Experimental Evaluation and Analysis of Database Cracking.
  The VLDB Journal.
  Special issue on best papers of VLDB 2014.
  The Sections 1 to 4 as well as Section 6 of the publication are covered in this chapter.

- **Chapter 3 — Multi-threaded Adaptive Indexing and Sorting**

  Publications:

  [49] Felix Martin Schuhknecht, Alekh Jindal, Jens Dittrich.
  An Experimental Evaluation and Analysis of Database Cracking.
  The VLDB Journal.
  Special issue on best papers of VLDB 2014.
  The Sections 5 and 6 of the publication are covered in this chapter.

  [3] Victor Alvarez, Felix Martin Schuhknecht, Jens Dittrich, Stefan Richter.
  Main Memory Adaptive Indexing for Multi-core Systems.
  SIGMOD DaMoN 2014, Snowbird, USA.

- **Chapter 4 — Data Partitioning**

  Publication:

  [50] Felix Martin Schuhknecht, Pankaj Khanchandani, Jens Dittrich.
  On the Surprising Difficulty of Simple Things: the Case of Radix Partitioning.
  VLDB 2015/PVLDB 2015, Hawaii, USA.

- **Chapter 5 — Rewiring Memory**

  Publication:

  [47] Felix Martin Schuhknecht, Jens Dittrich, Ankur Sharma.
  RUMA has it: Rewired User-space Memory Access is Possible!
  VLDB 2016/PVLDB 2016, New Delhi, India.

Chapter 2

Adaptive Indexing and Sorting in the Single-threaded Context

Database cracking has been an area of active research in recent years. The core idea of database cracking is to create indexes adaptively and incrementally as a side-product of query processing. Several works have proposed different cracking techniques for different aspects including updates, tuple-reconstruction, convergence, concurrency control, and robustness. This is the first comparative study of these different methods by an independent group. Our goal is to critically review several aspects, identify the potential, and propose promising directions in database cracking. With this study, we hope to expand the scope of database cracking and possibly leverage cracking in database engines other than MonetDB.

We repeat several prior database cracking works including the core cracking algorithms as well as three other works on convergence (hybrid cracking), tuple-reconstruction (sideways cracking), and robustness (stochastic cracking) respectively. Additionally, we also look at a recently published study about CPU efficiency (predication cracking). We evaluate these works and show possible directions to do even better. Altogether, in this chapter we revisit 5 papers on database cracking and evaluate in total 11 cracking methods, 4 sorting algorithms, and 3 full index structures.

Additionally, we test cracking under a variety of experimental settings, including high selectivity[1] queries, very low selectivity queries, varying selectivity, and multiple query access patterns. Finally, we compare cracking against different sorting algorithms as well as against different main-memory optimized indexes, including

[1] *Low* selectivity means, that *many* entries qualify. Consequently, a *high* selectivity means, that only *few* entries qualify.

the recently proposed Adaptive Radix Tree (ART).

Our results show that:

(i) the previously proposed cracking algorithms are repeatable,

(ii) there is still enough room to significantly improve the previously proposed cracking algorithms,

(iii) cracking depends heavily on query selectivity,

(iv) cracking needs to catch up with modern indexing trends, and

(v) different indexing algorithms have different indexing signatures.

2.1 Introduction

2.1.1 Background

Traditional database indexing relies on two core assumptions: (1) the query workload is available, and (2) there is sufficient idle time to create the indexes. Unfortunately, these assumptions are not valid anymore in modern applications, where the workload is not known or constantly changing and the data is queried as soon as it arrives. Thus, several researchers have proposed adaptive indexing techniques to cope with these requirements. In particular, *Database Cracking* has emerged as an attractive approach for adaptive indexing in recent years [26, 20, 21, 22, 23, 14, 17, 39]. Database cracking proposes to create indexes adaptively and as a side-product of query processing. The core idea is to consider each incoming query as a hint for data reorganization which eventually, over several queries, leads to a full index. Figure 2.1 recaps and visualizes the concept.

2.1.2 Our Focus

Database Cracking has been an area of active research in recent years, mainly led by researchers from CWI Amsterdam. This research group has proposed several different indexing techniques to address different dimensions of database cracking, including updates [21], tuple-reconstruction [22], convergence [23], and robustness [17]. They also worked on concurrency-control [14, 15], as we will see in Chapter 3 about parallel adaptive indexing.

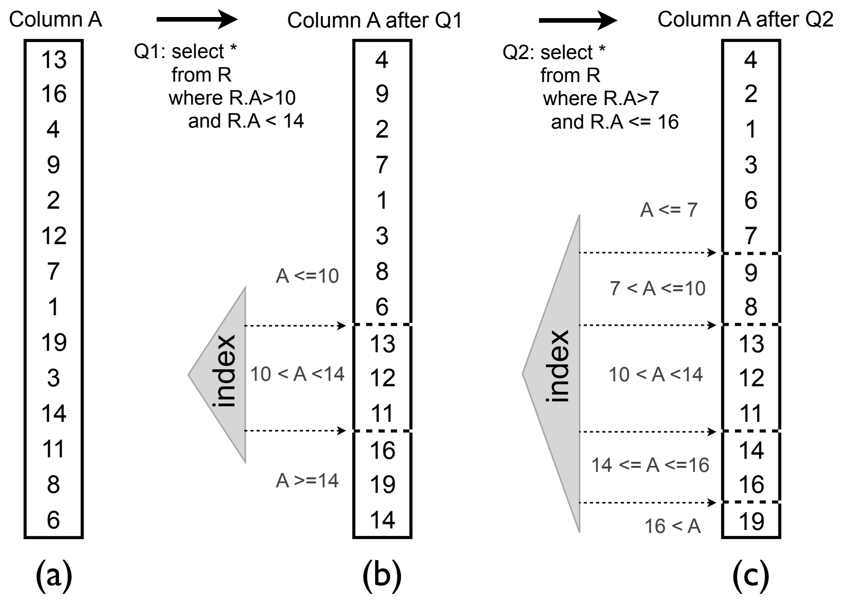


Figure 2.1: Database Cracking Example

In this chapter, we critically review single-threaded database cracking in several aspects. We repeat the core cracking algorithms, i.e. crack-in-two and crack-in-three [20], as well as three advanced cracking algorithms [22, 23, 17]. We identify the weak spots in these algorithms and discuss extensions to fix them. Additionally, we inspect a recently published work [39], which identifies CPU efficiency problems in the standard cracking algorithm and proposes alternatives. Finally, we also extend the experimental parameters previously used in database cracking, e.g. by varying the query selectivities and by comparing against more recent, main-memory optimized indexing techniques, including ART [31].

Our goal is to put database cracking in perspective by repeating several prior cracking works, giving new insights into cracking, and offering promising directions for future work. We believe that this will help the database community to understand database cracking better and to possibly leverage cracking for database systems other than MonetDB as well.

2.1.3 Contributions

Our core contributions in this chapter are as follows:

1. **Revisiting Cracking.** We revisit the core cracking algorithms, i.e. crack-in-two and crack-in-three [20], and compare them for different positions of the pivot elements. We do a cost breakdown analysis of the cracking algorithm into index lookup, data shuffle, index update, and data access costs. Further, we identify four major concerns, namely CPU efficiency, convergence, tuple reconstruction, and robustness. In addition, we evaluate advanced cracking algorithms, namely predication cracking [39], hybrid cracking [23], sideways cracking [22], and stochastic cracking [17] respectively, which were proposed to address these concerns. Additionally, in order to put together the differences and similarities between different cracking algorithms, we classify the cracking algorithms based on the strategy to pick the pivot, the creation time, and the number of partitions (Section 2.2).

2. **Extending Cracking Algorithms.** In order to better understand the cracking behavior, we modify three advanced cracking algorithms, namely hybrid cracking [23], sideways cracking [22], and stochastic cracking [17]. We show that buffering the swap elements in a heap before actually swapping them (*buffered swapping*) can lead to better convergence than hybrid cracking. Next, we show that covering the projection attributes with the cracker column (*covered cracking*) scales better than sideways cracking in the number of projected attributes. Finally, we show that creating more balanced partitions upfront (*coarse-granular indexing*) achieves better robustness in query performance than stochastic cracking. We also map these extensions to our cracking classification (Section 2.3).

3. **Extending Cracking Experiments.** As a next step, we extend the cracking experiments in order to test cracking under different settings. First, we compare database cracking against full indexing using different sorting algorithms and index structures. In previous works on database cracking quick sort is used to order the data indexed by the traditional methods that are used for comparison. Further, the cracker index is realized by an AVL-Tree [2] to store the index keys. In this chapter, we do a reality check with recent developments in sorting and indexing for main-memory systems. We show that full index creation with radix sort is twice as fast as with quick sort. We also show that ART [31] is up to 3.6 times faster than the AVL-Tree in terms of lookup time. We also vary the query selectivity from very high selectivity to medium selectivity down to very low selectivity and compare the effects. We conclude two key observations: (i) the choice of the index

structure has an impact only for very high selectivities, i.e. higher than 10^{-6} (one in a million), otherwise the data access costs dominate the index lookup costs; and (ii) cracking creates more balanced partitions and hence converges faster for medium selectivities, i.e. around 10%. We also look at the effect of stopping the cracking process at a certain partition size. Furthermore, we apply a sequential and a skewed query access pattern and analyze how the different adaptive indexing methods cope with them. Our results show that sequential workloads are the weak spots of query driven methods while skewed patterns increase the overall variance (Section 2.4).

4. **Conclusion.** Finally, we conclude by putting together the key lessons learned. Additionally, we also introduce *signatures* to characterize the indexing behavior of different indexing methods and to understand as well as differentiate them visually (Section 2.5).

2.1.4 Experimental Setup

We use a common experimental setup throughout this chapter and try to keep our setup as close as possible to the earlier cracking works. Similar to previous cracking works, we use an integer array with 10^8 uniformly distributed values with a key range of $[0; 100,000]$. Unless mentioned otherwise, we run 1000 random queries, each with selectivity 1%. The queries are of the form:

```
SELECT SUM(A) FROM R WHERE A >= low AND A < high
```

We repeat the entire query sequence three times and take the average runtime of each query in the sequence. We consider two baselines: (i) *scan* which reads the entire column and post-filters the qualifying tuples, and (ii) *full index* which fully sorts the data using quick sort and performs binary search for query processing. If not stated otherwise, all indexes are unclustered and uncovered.

We implemented all algorithms in a stand-alone program written in C/C++ and compile with G++ version 4.7 using optimization level 3. Our test bed consists of a single machine with two Intel Xeon X5690 processors running at a clock speed of 3.47 GHz and supports Intel Turbo Mode. Each CPU has 6 cores and supports 12 threads via Intel Hyper Threading. The L1 and L2 cache sizes are 64 KB and 256 KB respectively for each core. The shared L3 cache has a size of 12 MB. Our machine has 200 GB of main memory and runs a 64-bit linux with kernel 3.1.

2.2 Revisiting Cracking

Let us start by revisiting the original cracking algorithm [20]. Our goal in this section is to first compare crack-in-two with crack-in-three, then to repeat the standard cracking algorithm under similar settings as in previous works, then to break down the costs of cracking into individual components, and finally to identify the major concerns in the original cracking algorithm.

2.2.1 Crack-in-two vs Crack-in-three

crack-in-two: *partition the index column into two pieces using one end of a range query as the boundary.*

crack-in-three: *partition the index column into three pieces using the two ends of a range query as the two boundaries.*

The original cracking paper [20] introduces two algorithms: *crack-in-two* and *crack-in-three* to partition (or *split*) a column into two and three partitions respectively. Conceptually crack-in-two is suitable for one-sided range queries, e.g. $A < 10$, whereas crack-in-three for two-sided range queries, e.g. $7 < A < 10$. However, we could also apply two crack-in-twos for a two-sided range query. Let us now compare the performance of crack-in-two and crack-in-three on two-sided range queries. We consider the cracking operations from a single query and vary the position of the split line in the cracker column from bottom (low relative position) to top (high relative position). A relative position of the low key split line of p% denotes that the data is partitioned into two parts with size p% and $(100-p)$%. We expect the cracking costs to be the maximum around the centre of the column (since maximum swapping will occur) and symmetrical on either ends of the column.

Figure 2.2 shows the results. Though both 2×crack-in-two and crack-in-three have maximum costs around the center, surprisingly crack-in-three is not symmetrical on either ends. Crack-in-three is much more expensive in the lower part of the column than in the upper part. This is because crack-in-three always starts considering elements from the top. Another interesting observation from Figure 2.2 is that even though 2×crack-in-two performs two cracking operations, it is cheaper than crack-in-three when the split position is in the lower 70% of the column. Thus, we see that crack-in-two and crack-in-three are very different algorithms in terms of performance and future works should consider this when designing newer algorithms.

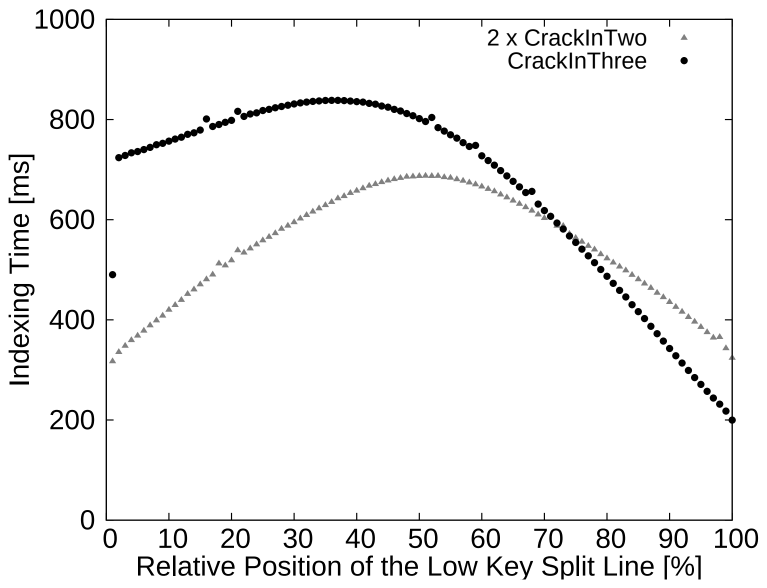


Figure 2.2: Comparing Single Query Indexing Time

2.2.2 Standard Cracking Algorithm

Standard Cracking

Incrementally and adaptively sort the index column using crack-in-three when both ends of a range query fall in the same partition and crack-in-two otherwise.

We implemented the standard cracking algorithm which uses crack-in-three wherever two split lines lie in the same partition, and tested it under the same settings as in previous works. As in the original papers, we use an AVL-Tree as cracker index to be able to compare the results.

Figure 2.3 shows the results. We can see that standard cracking starts with similar performance as full scan and gradually approaches the performance of full index. Moreover, the first query takes just 0.3 seconds compared to 0.24 seconds of full scan[2], even though standard cracking invests some indexing effort. In contrast, full index takes 10.2 seconds to fully sort the data before it can process the first query. This shows that standard cracking is lightweight and it puts little penalty on the first query. Overall, we are able to reproduce the cracking behavior of previous works.

[2]Note that the query time of full scan varies by as much as 4 times. This is because of lazy evaluation in the filtering depending on the position of low key and high key in the value domain.

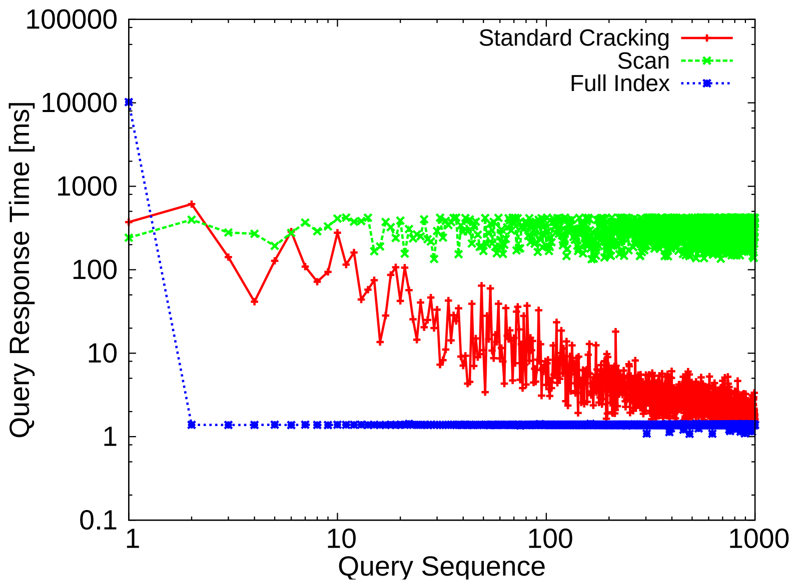


Figure 2.3: Reproducing Cracking Behavior

2.2.3 Cost Breakdown

Next let us see the cost breakdown of the original cracking algorithm. The goal here is to see where the cracking query engine spends most of the time and how that changes over time. Figure 2.4 shows the cost breakdown of the query response time into four components: (i) *index lookup* costs to identify the partitions for cracking, (ii) *data shuffle* costs of swapping the data items in the column, (iii) *index update* costs for updating the index structure with the new partitions, and (iv) *data access* costs to actually access the qualifying tuples.

We can see that the data shuffle costs dominate the total costs initially. However, the data shuffle costs decrease gradually over time and match the data access costs after $1,000$ queries. This shows that standard cracking does well to distribute the indexing effort over several queries. We can also see that index lookup and update costs are orders of magnitude less than the data shuffle costs. For instance, after 10 queries, the index lookup and update costs are about 1μs whereas the shuffle costs are more than 100 ms. This shows that standard cracking is indeed lightweight and has very little index maintenance overheads. However, as the number of queries increases, the data shuffle costs decrease while the index maintenance costs increase.

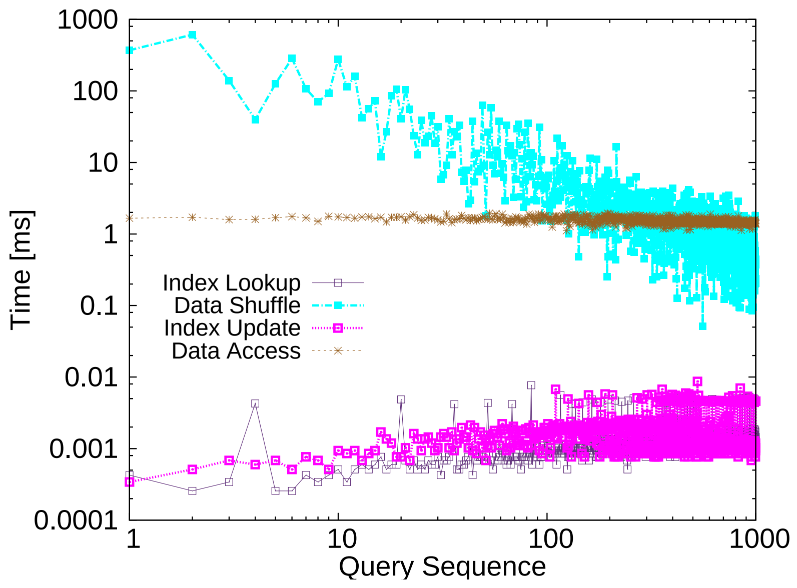


Figure 2.4: Cost Breakdown of Standard Cracking

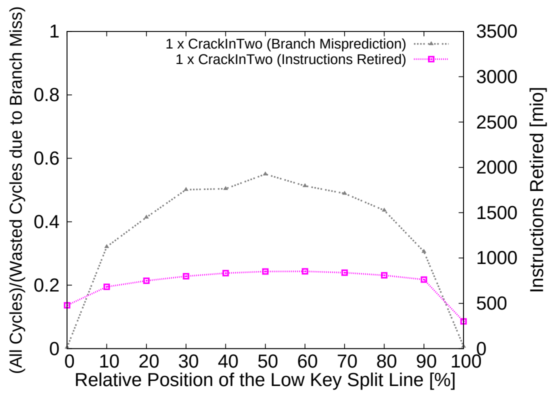
2.2.4 Key Concerns in Standard Cracking

Let us now take a closer look at the standard cracking algorithm from four different perspectives, namely (1) CPU efficiency on modern hardware, (2) convergence to a full index, (3) scaling the number of projected attributes, and (4) variance in query performance.

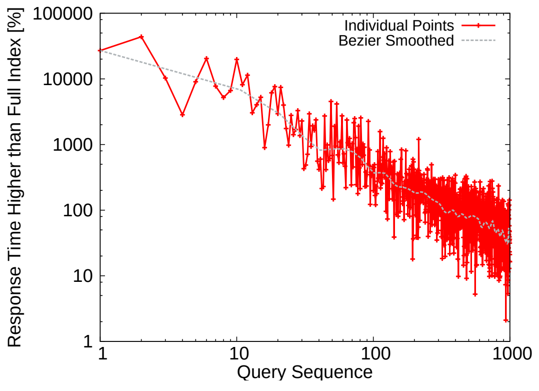
(1) **CPU Efficiency.** How an algorithm is mapped to the underlying hardware is crucial in memory resident data processing. Figure 2.5(a) shows the branch misprediction[3] as *the* weak spot of the crack-in-two algorithm with respect to the relative position of the split line, making this method clearly CPU bound. At a worst-case position of the split line dividing the partition in the middle, more than 50% of the branches are predicted incorrectly.

(2) **Cracking Convergence.** Convergence is a key concern and major criticism for database cracking. Figure 2.5(b) shows the number of queries after which the query response time of standard cracking is within a given percentage of full index. The figure also shows a bezier smoothened curve of the data points. From the figure we can see that after 1,000 queries, on average, the query response time of standard cracking is still 40% higher than that of full index.

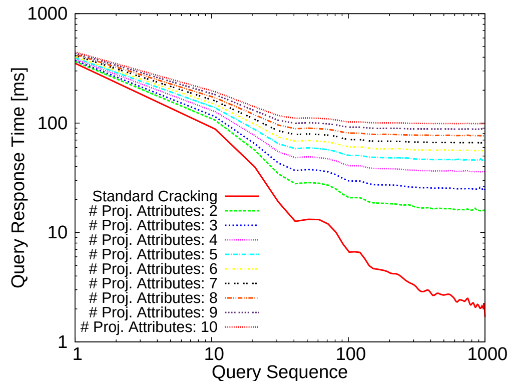
[3]Measured with Intel VTune Amplifier 2015.



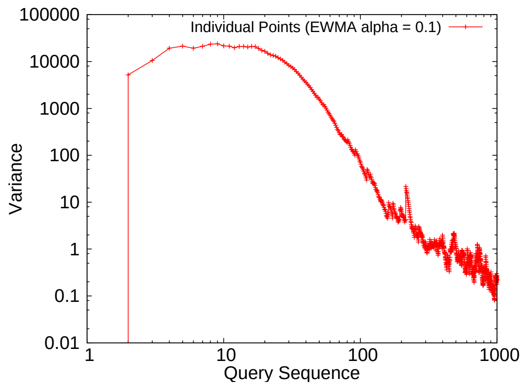
(a) CPU Efficiency



(b) Cracking Convergence



(c) Scaling Projected Attributes



(d) Cracking Variance

Figure 2.5: Key Concerns in Standard Cracking

(3) **Scaling Projected Attributes.** By default, database cracking leads to an unclustered index, i.e. extra lookups are needed to fetch the projected attributes. Figure 2.5(c) shows the query response time with tuple reconstruction, when varying the number of projected attributes from 1 to 10. For the ease of presentation, we show only the bezier smoothened curves. We can see that standard cracking does not scale well with the number of attributes. In fact, after $1,000$ queries, querying 10 attribute tuples is almost 2 orders of magnitude slower than querying 1 attribute tuples.

(4) **Cracking Variance.** Standard cracking partitions the index column based on the query ranges of the selection predicate. As a result, skewed query range predicates can lead to skewed partitions and thus unpredictable query performance. Figure 2.5(d) shows the variance of standard cracking query response times using the exponentially weighted moving average (EWMA). The variance is calculated as described in [13]. The degree of weighting decrease is $\alpha = 0.1$. We can see that unlike full index (see Figure 2.3), cracking does not exhibit stable query performance. Furthermore, we also see that the amount of variance for standard cracking decreases by five orders of magnitude.

2.2.5 Advanced Cracking Algorithms

Several follow-up works on cracking focussed on the key concerns in standard cracking. In this section, we revisit these advanced cracking techniques.

Predication & Vectorized Cracking

Decouple pivot comparison and physical reorganization by moving elements speculatively and correcting wrong decisions afterwards.

The technique of predication cracking [39] directly attacks a major problem in standard cracking — excessive branch misprediction leading to large amounts of unnecessarily executed code. In contrast to standard cracking, where based on the outcome of the comparison of the element with the pivot, pointers are moved and elements are swapped, predication cracking speculatively performs these reorganizations and interleaves them with the comparison evaluations of pivot and elements. When the result of the comparison is available, the incorrectly applied reorganizations are corrected. To ensure that the speculative writing does not cause data loss, the overwritten elements are backed up separately. This concept makes the algorithm completely branch-free and thus, the misprediction penalties do not longer exist. On the downside, the speculative writing adds an overhead compared to standard cracking. The question is now whether this trade-off can improve the runtime. Algorithm 1 shows the pseudo-code for two-sided predication cracking.

Algorithm 1: Predication Cracking

```
//  Input: "data" array,
//         "start" and "end" of working area,
//         "pivot"
// Output: position of split line
l = start;
u = end;
b[0] = data[l];
b[1] = data[u];
for(i = start; i <= end; ++i) {
  // write speculatively to both sides
  data[l] = b[i%2];
  data[u] = b[i%2];
  // compare current element with pivot
  aL = b[i%2] < pivot;
  aU = 1 - aL;
  // backup next element
  b[i%2] = aL * data[l+1] + aU * data[u-1];
  // advance pointer of correct side
  l += aL;
  u -= aU;
}
return l - 1;
```

In predication cracking, the granularity at which data is backed up is fixed to a single element. Thus the authors propose a generalization of the concept in form of *vectorized cracking*, where data is backed up and partitioned in larger blocks of adjustable size. This further decouples the costly backing up of data from the actual partitioning.

In Figure 2.6 we add both predication cracking as well as vectorized cracking with a vector size of 128B, which showed the best results in our evaluation, to the plot of Figure 2.5(a). Compared to standard cracking, the problem of branch misprediction vanishes almost entirely. However, we can also observe that the number of retired instructions drastically increases over the standard version. Thus, the concept of predication basically trades in a higher reorganization effort for less

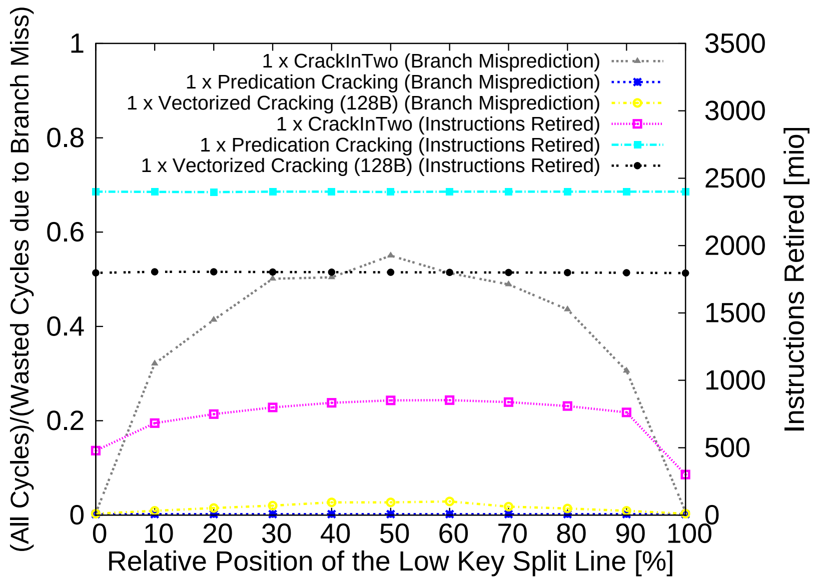
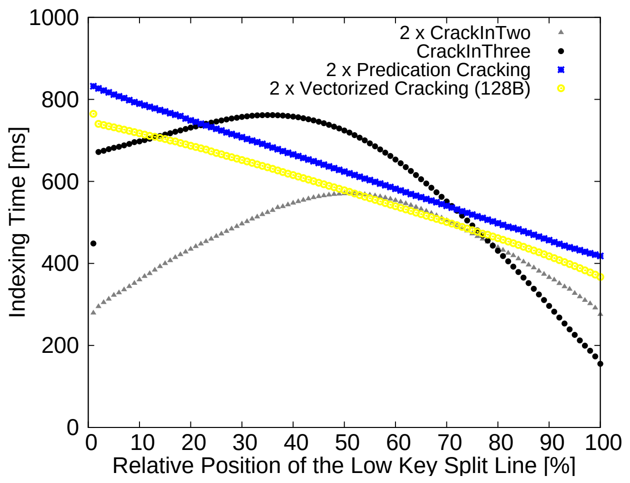
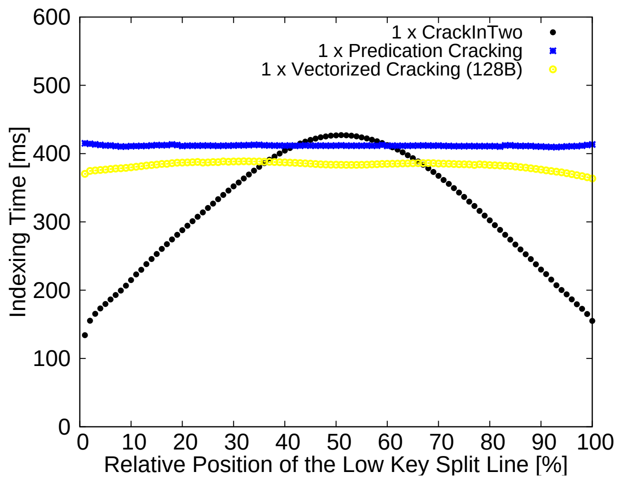


Figure 2.6: CPU Efficiency in Terms of Branch Misprediction and Instructions Retired.

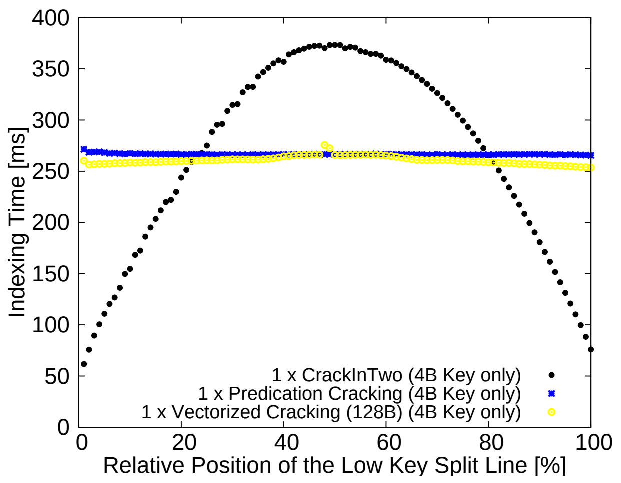
branching penalties. Vectorized cracking reduces this overhead by using larger blocks, resulting in a lower number of retired instructions while maintaining a negligible branch misprediction. This observed tradeoff already indicates that the actual runtimes between standard and predication/vectorized cracking might be close to each other. Figure 2.7(a) shows the result when extending the study of Figure 2.2 with predication and vectorized cracking. Unfortunately, although vectorized cracking performs slightly better than predication cracking in all cases, both methods do not significantly pay off over applying crack-in-two twice. On first sight, these results look contrary to the ones presented in [39]. However, comparing the experimental setups, two differences become clear. Firstly, in [39], the authors work on pure 4B keys in contrast to our 16B (key, rowID) pairs, that are in our opinion more realistic to represent an index column. As predication/vectorized cracking is write intensive, a larger element size puts more pressure on these methods than on the standard ones. Secondly, we perform one query consisting of two cracks here, instead of only a single crack in [39]. As our second crack is 1% of the data size to the right of the first one, only few swaps must be performed and the branch prediction works already very well for standard cracking. To confirm the original results of [39], we rerun the experiment with 4B keys and only a single crack in Figure 2.7(c). Now, we see a clear benefit of both predication and vectorized cracking over standard crack-in-two, if the split line falls between 20% and 80% of the key range. However, when using our standard 16B pairs (Figure 2.7(b))



(a) Single Query Indexing Time



(b) Single Crack Indexing Time on 16 B Elements



(c) Single Crack Indexing Time on 4 B Elements

Figure 2.7: Standard Cracking in Comparison with Predication Cracking and Vectorized Cracking in Terms of Indexing Time with Respect to the Split Line Position.

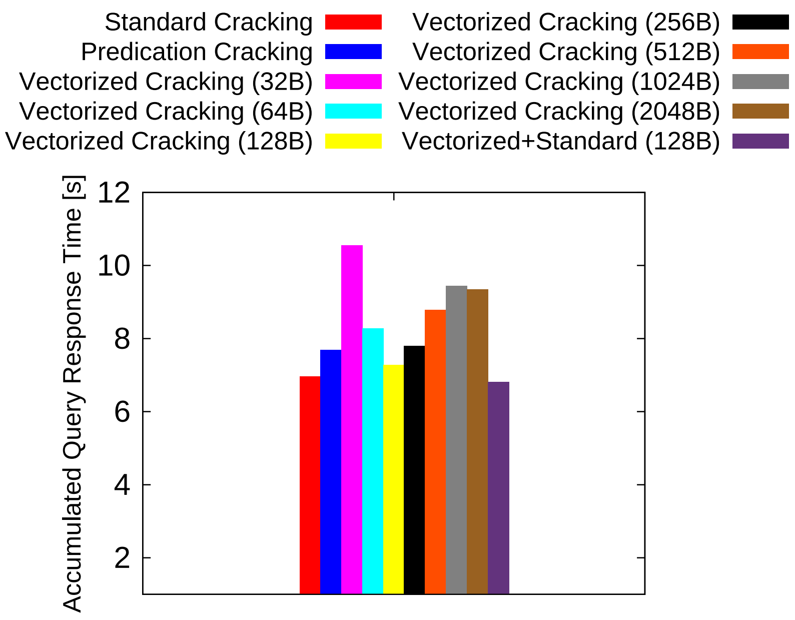


Figure 2.8: Predication and Vectorized Cracking over Query Sequence.

the single-crack runtime increases heavily for predication and vectorized cracking, but only slightly for standard crack-in-two.

Let us finally look at how the predicated methods perform under a sequence of 1000 queries. Figure 2.8 shows the results. In accordance to the results of Figure 2.7(a), neither predication nor vectorized cracking can beat standard cracking with respect to accumulated query response time. The best vector size is clearly 128B on our machine, where other sizes perform significantly worse. However, our previous insights enable us to use the best of two worlds. By using vectorized cracking for the first crack and standard cracking for the nearby second crack of a query, we are able to improve slightly over the standard version. Overall, whether predication and vectorized cracking pay off highly depends on element size, split line position, and machine properties.

Hybrid cracking

Create unsorted initial runs which are physically reorganized and then adaptively merged for faster convergence.

Hybrid cracking [23] aims at improving the poor convergence of standard cracking to a full index, as shown in Figure 2.5(b). Hybrid cracking combines ideas from adaptive merging [16] with database cracking in order to achieve fast convergence to a full index, while still providing low initialization costs. The key problem in standard cracking is that it creates at most two new partition boundaries per query, and thus requires several queries to converge to a full index. On the other hand, adaptive merging creates initial sorted runs, and thus pays a high cost for the first query.

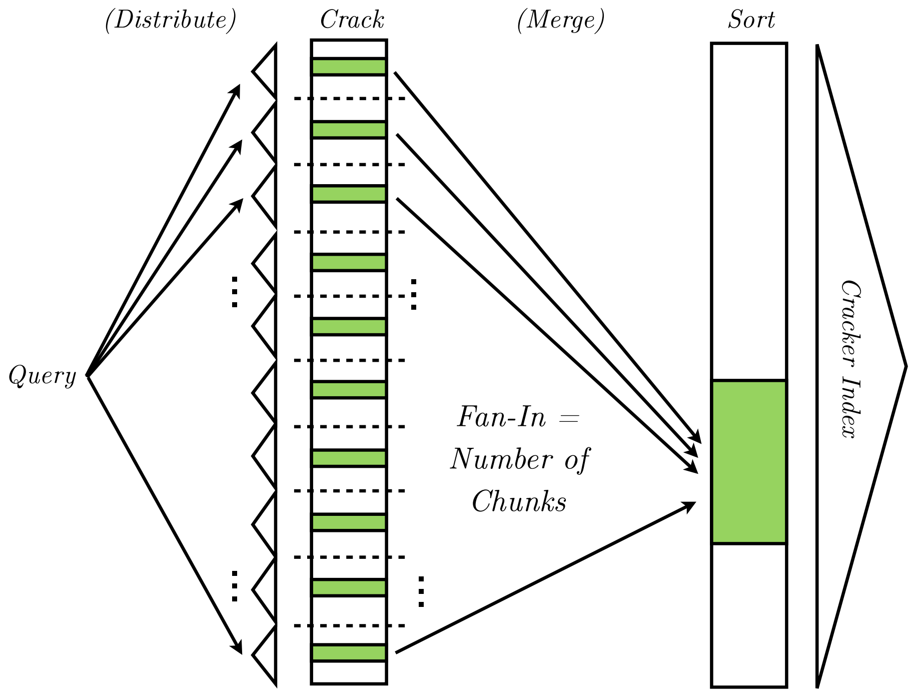


Figure 2.9: Visualization of the concept of Hybrid Crack Sort.

Hybrid cracking overcomes these problems by creating initial unsorted partitions and later adaptively refining them with lightweight reorganization. In addition to reorganizing the initial partitions, hybrid cracking also moves the qualifying tuples from each initial partition into a final partition. The authors explore different strategies for reorganizing the initial and final partitions, including sorting, standard cracking, and radix clustering, and conclude standard cracking to be the best for initial partitions and sorting to be the best for final partition. By creating initial partitions in a lightweight manner and introducing several partition boundaries, hybrid cracking converges better. We implemented *hybrid crack sort*, which showed the best performance in [23], as close to the original description as possible. Figure 2.9 visualizes the concept.

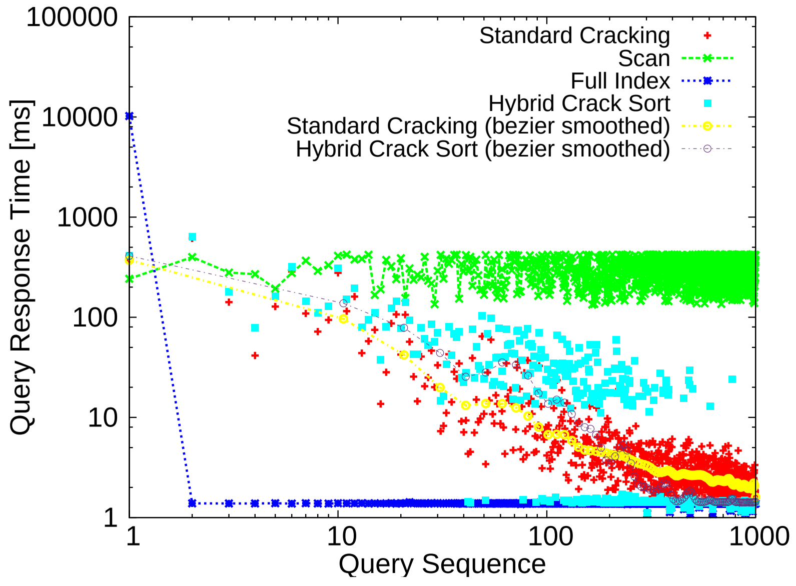


Figure 2.10: Revisiting Hybrid Cracking.

Figure 2.10 shows hybrid crack sort in comparison to standard cracking, full index, and scan. We can see that hybrid crack sort converges faster as compared to standard cracking.

Sideways Cracking

Adaptively create, align, and crack every accessed selection-projection attribute pair for efficient tuple reconstruction.

Sideways Cracking [22] uses *cracker maps* to address the issue of inefficient tuple reconstruction in standard cracking, as shown in Figure 2.5(c). A cracker map consists of two logical columns, the cracked column and a projected column, and it is used to keep the projection attributes aligned with the selection attributes. When a query comes in, sideways cracking creates and cracks only those crackers maps that contain any of the accessed attributes. As a result, each accessed column is always aligned with the cracked column of its cracker map. If the attribute access pattern changes, then the cracker maps may reflect different progressions with respect to the applied cracks. Sideways cracking uses a log to record the state of each cracker map and to synchronize them when needed. Thus, sideways cracking works without any workload knowledge and adapts cracker maps to the attribute access patterns. Further, it improves its adaptivity and reduces the amount of overhead by only materializing those parts of the projected columns in the cracker maps which are actually queried (*partial sideways cracking*).

We reimplemented sideways cracking similar to as described above, except that we store cracker maps in row layout instead of column layout. We do so because the two columns in a cracker map are always accessed together and a row layout offers better tuple reconstruction. In addition to the cracked column and the projected column, each cracker map contains the rowIDs that map the entries into the base table as well as a status column denoting which entries of the projected column are materialized. Figure 2.11 visualizes the concept for answering two queries.

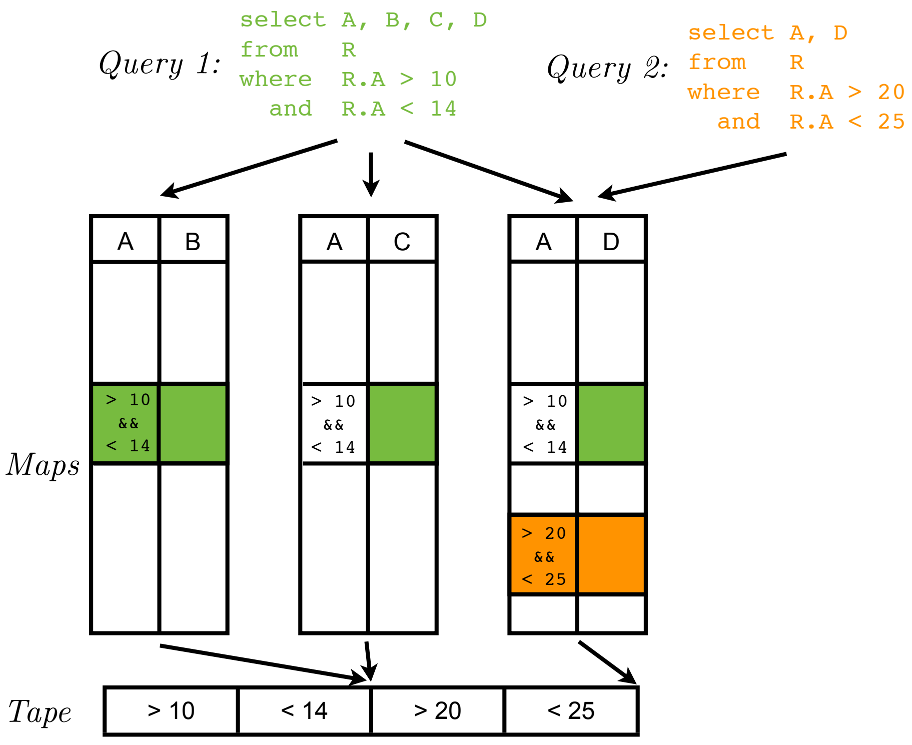


Figure 2.11: Visualization of the concept of query answering for sideways cracking. The first query cracks all cracker maps $[A, B]$, $[A, C]$, and $[A, D]$ with respect to $A > 10$ and $A < 14$ and logs the cracks in the tape. The second query cracks only the involved cracker map $[A, D]$ with respect to $A > 20$ and $A < 25$.

Figure 2.12 shows the performance of sideways cracking in comparison. In this ex-

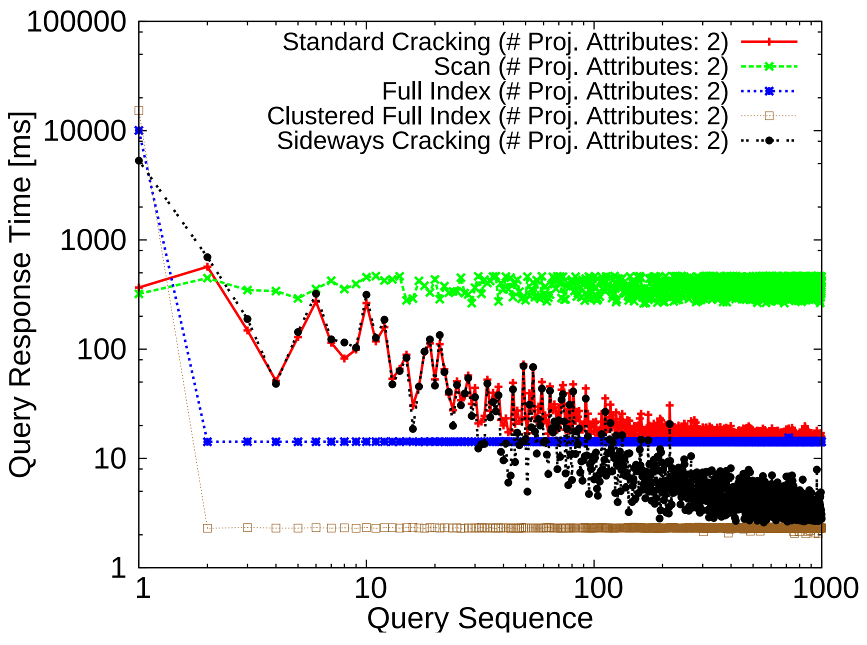


Figure 2.12: Revisiting Sideways Cracking.

periment the methods have to project one attribute while selecting on another. In addition to the unclustered version of full index, we also show the clustered version (clustered full index). We can see that sideways cracking outperforms all unclustered methods after about 100 queries and approaches the query response time of clustered full index. Thus, sideways cracking offers efficient tuple reconstruction.

Stochastic Cracking

Create more balanced partitions using auxiliary random pivot elements for more robust query performance.

Stochastic cracking [17] addresses the issue of performance unpredictability in database cracking, as seen in Figure 2.5(d). A key problem in standard cracking is that the partition boundaries depend heavily on the incoming query ranges. As a result, skewed query ranges can lead to unbalanced partition sizes and successive queries may still end up rescanning large parts of the data. To reduce this problem, stochastic cracking introduces additional cracks apart from the query-driven cracks at query time. These additional cracks help to evolve the cracker index in a more uniform manner. Stochastic cracking proposes several variants to introduce these additional cracks, including data driven and probabilistic decisions. By varying the amount of auxiliary work and the crack positions, stochastic cracking manages to introduce a trade-off situation between variance on one side and cracking overhead on the other side.

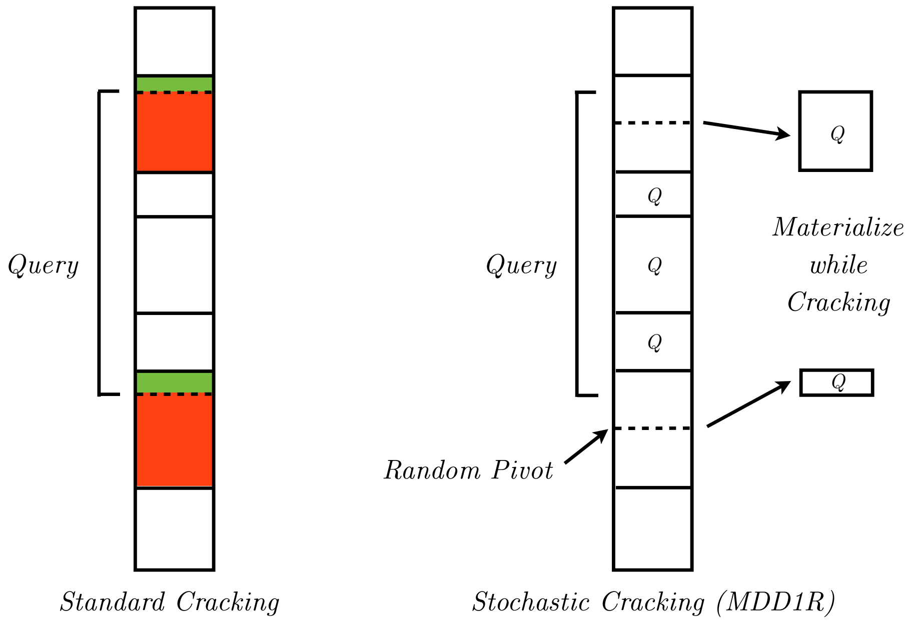


Figure 2.13: Visualization of the concept of query answering for Stochastic Cracking (MDD1R) in comparison with Standard Cracking.

We reimplemented the *MDD1R* variant of stochastic cracking, which showed the best overall performance in [17]. Figure 2.13 visualizes the concept. In this variant, the partitions in which the query boundaries fall are cracked by performing exactly one random split. Additionally, while performing the random split, the result of each partition at the boundary of the queried range is materialized in a separate view. At query time the result is built by reading the data of the boundary partitions from the views and the data of the inner part from the index.

Figure 2.14 shows the MDD1R variant of stochastic cracking. We can see that stochastic cracking (MDD1R) behaves very similar to standard cracking, although the query response times are overall slower than those of standard cracking. As the uniform random access pattern creates balanced partitions by default, the additional random splits introduced by stochastic cracking (MDD1R) do not have a positive effect. We will come back to stochastic cracking (MDD1R) with other access patterns in Section 2.4.7.

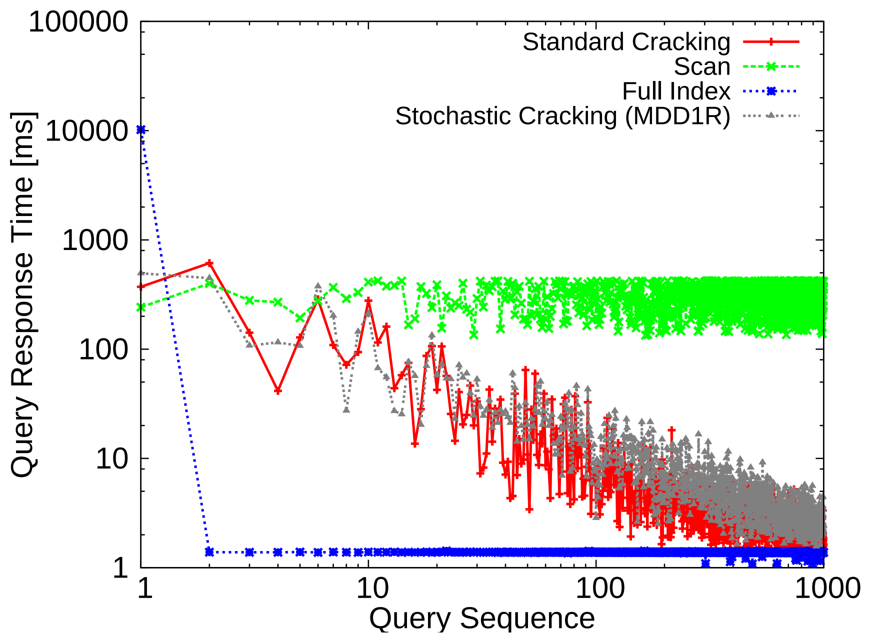


Figure 2.14: Revisiting Stochastic Cracking.

2.2.6 Cracking Classification

Let us now compare and contrast the different cracking algorithms discussed so far with each other. The goal is to understand what are the key differences (or similarities) between these algorithms. This will possibly help us in identifying the potential for newer cracking algorithms. Note that all cracking algorithms essentially *split* or *partition* the data incrementally. Different algorithms split the data differently. Thus, we categorize the cracking algorithms along three dimensions: (i) the number of split lines they introduce, (ii) the split strategy, and (iii) the timing of the split. Table 2.1 shows the classification of different cracking algorithms along these three dimensions. Let us discuss these below.

Number of Split Lines. The core cracking philosophy mandates all cracking algorithms to do some indexing effort, i.e. introduce at least one split line, when a query arrives. However, several algorithms introduce other split lines as well. We classify the cracking algorithms into the following four categories based on the number of split lines they introduce.

1. *Zero:* The trivial case is when a method introduces no split line and each query performs a full scan.

2. *Few:* Most cracking algorithms introduce a few split lines at a time. For

DIMENSIONS	CATEGORY	NO INDEX	STANDARD CRACKING / PREDICATION CRACKING	HYBRID CRACKING (CRACK SORT)	SIDEWAYS CRACKING	STOCHASTIC CRACKING (MDD1R)	FULL INDEX
NUMBER OF SPLIT LINES	ZERO	▨					
	FEW		▨		▨	▨	
	SEVERAL			▨			
	ALL			▨			▨
SPLIT STRATEGY	NONE	▨					
	QUERY BASED		▨	▨	▨	▨	
	RANDOM					▨	
	DATA BASED			▨			▨
SPLIT TIMING	NEVER	▨					
	PER QUERY		▨	▨	▨	▨	
	UPFRONT						▨

Table 2.1: Classification of Cracking Algorithms.

instance, standard cracking introduces either one or two splits lines for each incoming query. Similarly, sideways cracking introduces split lines in all accessed cracker maps.

3. *Several:* Cracking algorithms can also introduce several split lines at a time. For example, hybrid crack sort may introduce several thousand initial partitions and introduce either one or two split lines in each of them. Thus, generating several split lines in total.

4. *All:* The extreme case is to introduce all possible split lines, i.e. fully sort the data. For example, hybrid crack sort fully sorts the final partition, i.e. introduces all split lines in the final partition.

Split Strategy. Standard cracking chooses the split lines based on the incoming query. However, several advanced cracking algorithms employ other strategies. Below, we classify the cracking algorithms along four splitting strategies.

1. *Query Based:* The standard case is to pick the split lines based on the selection predicates in the query, i.e. the low and high keys in the query range.

2. *Data Based:* We can also split data without looking at a query. For example, full sorting creates split lines based only on the data.

3. *Random:* Another strategy is to pick the split lines randomly as in stochastic cracking.

4. *None:* Finally, the trivial case is to not have any split strategy, i.e. do not

split the data at all and perform full scan for all queries.

Split Timing. Lastly, we consider the timing of the split to classify the cracking algorithms. We show three time points below.

1. *Upfront:* A cracking algorithm could perform the splits before answering any queries. Full indexing falls in this category.

2. *Per Query:* All cracking algorithms we discussed so far perform splits when seeing a query.

3. *Never:* The trivial case is to never perform the splits, i.e. fully scanning the data for each query.

2.3 Extending Cracking Algorithms

In this section, we discuss the weaknesses in the advanced cracking algorithms and evaluate possible directions on how to improve them.

2.3.1 Improving Cracking Convergence

Let us see how well hybrid cracking [23] addresses the convergence issue and whether we can improve upon it. First, let us compare hybrid crack sort from Figure 2.10 with two other variants of hybrid cracking: *hybrid radix sort*, and *hybrid sort sort*. Figure 2.15 shows how quickly the hybrid algorithms approach to a full index. We can see that hybrid radix sort converges similar to hybrid crack sort and hybrid sort sort converges faster than both of them. This suggests that the convergence property in hybrid algorithms comes from the sort operations. However, keeping the final partition fully sorted is expensive. Indeed, we can see several spikes in hybrid crack sort in Figure 2.10. If a query range is not contained in the final partition, all qualifying entries from all initial partitions must be merged and sorted into the final partition. Can we do better? Can we move data elements to their final position (as in full sorting) in a fewer number of swaps, and thus improve the cracking convergence?

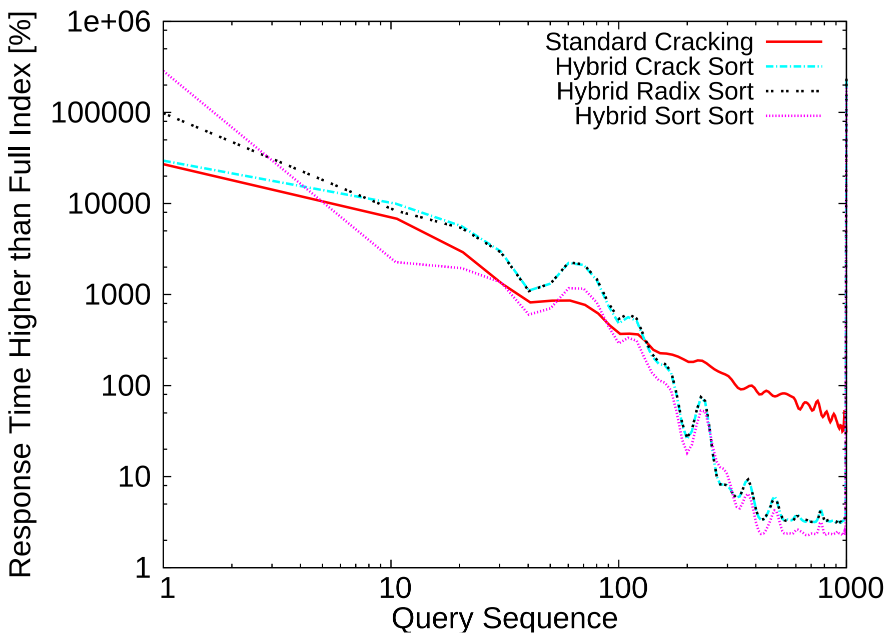


Figure 2.15: Convergence Speed towards Full Index.

Buffered-swapping

Instead of swapping elements immediately after identification by the cracking algorithm, insert them into heaps and flush them back into the index as sorted runs.

Let us look at the crack-in-two operation[4] in hybrid cracking. Recall that the crack-in-two operation scans the dataset from both ends until we find a pair of entries which need to be swapped (i.e. they are in the wrong partitions). This pair is then swapped and the algorithm continues its search until the pointers meet. Note that there is no relative ordering between the swapped elements and they may end up getting swapped again in future queries, thus penalizing them over and over again. We can improve this by extending the crack-in-two operation to buffer the elements identified as swap pairs, i.e. *buffered crack-in-two*.

Buffered crack-in-two collects the swap pairs in two heaps: a max-heap for values that should go to the upper partition and a min-heap for values that should go to the lower partition. In addition to the heap structures, we maintain two queues to store the empty positions in the two partitions. The two heaps keep the elements

[4]After the first few queries, cracking mostly performs a pair of crack-in-two operations as the likelihood of two splits falling in two different partitions increases with the number of applied queries.

in order and when the heaps are full we swap the top-elements in the two heaps to the next available empty position. This process is repeated until no more swap pairs can be identified and the heaps are empty. As a result of heap ordering, the swapped elements retain a relative ordering in the index after each cracking step. This order is even valid for entries that were not in the heap at the same time, but shared presence with a third element and hence a transitive relationship is established. Every pair element that is ordered in this process is never swapped in future queries and thus, the number of swaps is reduced. The above approach of buffered crack-in-two is similar to [32], where two heaps are used to improve the stability of the replacement selection algorithm. By adjusting the maximal heap size in buffered crack-in-two, we can tune the convergence speed of the cracked index. Larger heap size results in larger sorted runs. However, larger heaps incur high overhead to keep its data sorted. In the extreme case, a heap size equal to the number of (swapped) elements results in full sorting while a heap size of 1 falls back to standard crack-in-two.

Of course buffered crack-in-two can be embedded in any method that uses the standard crack-in-two algorithm. To separate it from the remaining methods we integrate it into a new technique called *buffered swapping* that is a mixture of buffered and standard crack-in-two. For the first n queries buffered swapping uses buffered crack-in-two. After that buffered swapping switches to standard cracking-in-two. Figure 2.16 visualizes the concept of the technique.

Figure 2.17(a) shows the number of swaps in standard cracking, hybrid crack sort, and buffered swapping over 1000 queries. In order to make them comparable, we force all methods to use only crack-in-two operations. For buffered swapping we vary the number buffered queries n_b along the x-axis, i.e. the first n_b queries perform buffered swapping while the remaining queries still perform the standard crack-in-two operation. We vary the maximal heap size from 100K to 10M entries.

From Figure 2.17(a), we can see that the number of swaps decrease significantly as n_b varies from 1 to 1000. Compared to standard cracking, buffered swapping saves about 4.5 million swaps for 1 buffered query and 73 million swaps for 1000 buffered queries and a heap size of 1M. The maximal size of the heap is proportional to the reduction in swaps. Furthermore, we can observe that the swap reduction for 1000 buffered queries improves only marginally over that of 100 buffered queries. This indicates that after 100 buffered queries the cracked column is already close to being fully sorted. Hybrid cracking performs even more swaps than standard cracking (including moving the qualifying entries from the initial partitions to the final partition).

Next let us see the actual runtimes of buffered swapping in comparison to stan-

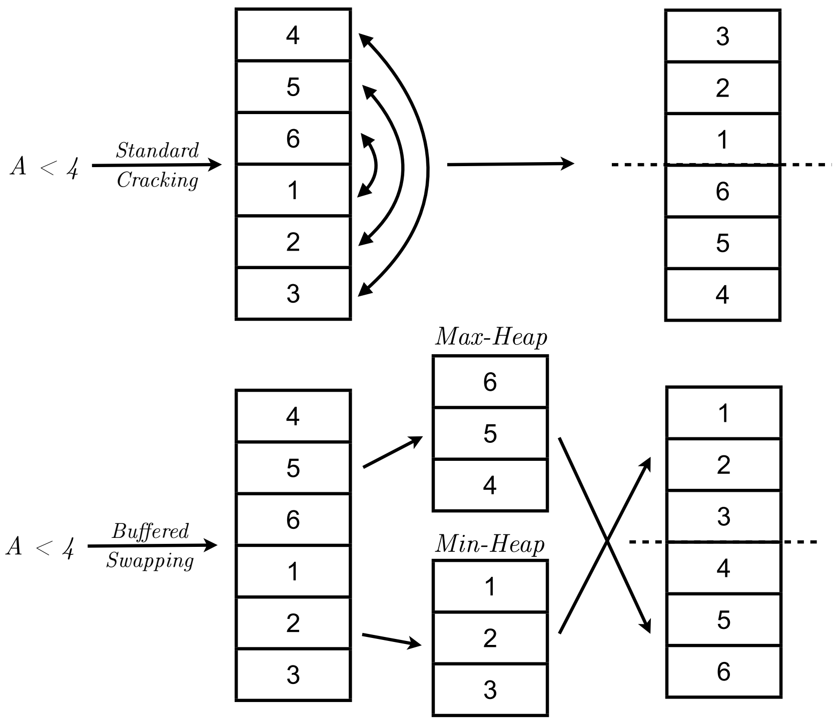
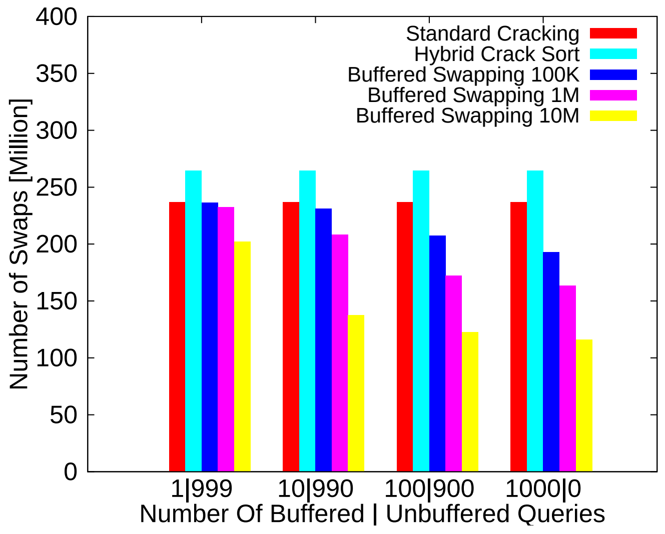
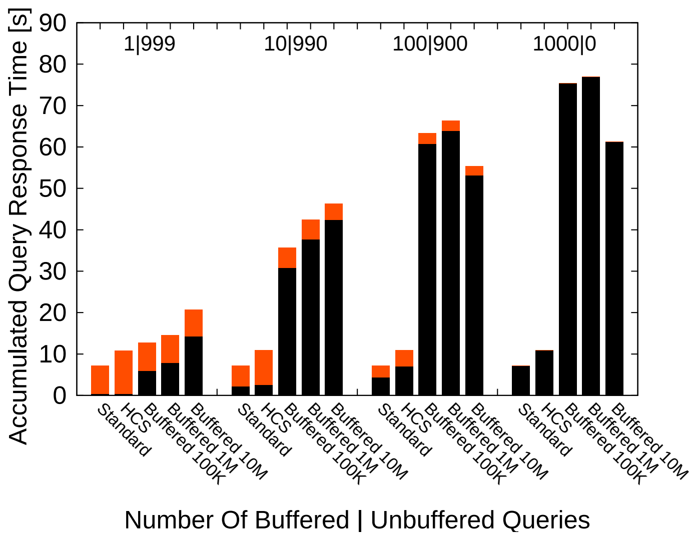


Figure 2.16: Visualization of the concept of buffered swapping in comparison with standard cracking. Buffered swapping uses a heap size of 3 elements in this example.

dard cracking and hybrid crack sort. Figure 2.17(b) shows the result. We see that the total runtime grows rapidly as the number of buffered queries (n_b) increases. However, we also see that the query time after performing buffered swapping improves. For example, after performing buffered swapping with a maximal heap size of 1M for just 10 queries, the remaining 990 queries are 1.8 times faster than hybrid crack sort and even 5.5% faster than standard cracking. This shows that buffered swapping helps to converge better by reducing the number of swaps in subsequent queries. Interestingly, a larger buffer size does not necessarily imply a higher runtime. For 100 and $1,000$ buffered queries the buffered part is faster for a maximum heap size of 10M entries than for smaller heaps. This is because such a large heap size leads to an earlier convergence towards the full sorting. Nevertheless, the high runtime of initial buffer swapped queries is a concern. In our experiments we implemented buffered swapping using the gheap implementation [1] with a fan-out of 4. Each element that is inserted into a gheap has to sink down inside of the heap tree to get to its position. This involves pairwise swaps and triggers many cache-misses. Exploring more efficient buffering mechanisms in detail opens up avenues for future work.



(a) Influence on Swap Count



(b) Influence on Query Response Time

Figure 2.17: Comparing Convergence of Standard Cracking, Hybrid Cracking and Buffered Swapping

2.3.2 Improving Tuple Reconstruction

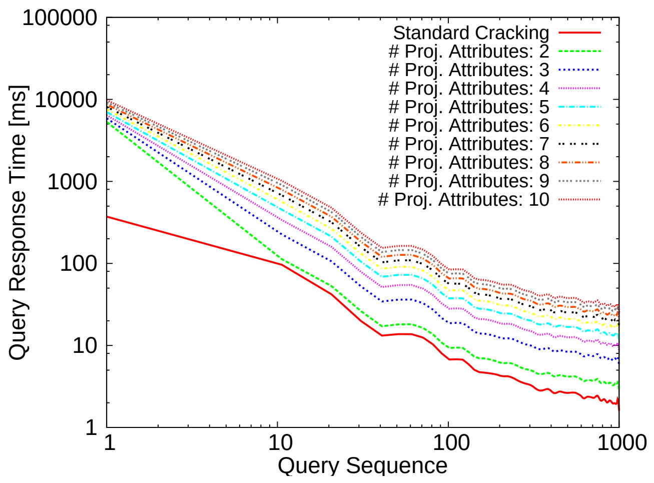
Our goal in this section is to see how well sideways cracking [22] addresses the issue of tuple reconstruction and whether we can improve upon it. Let us first see how the sideways cracking from Figure 2.12 scales with the number of attributes. Figure 2.18(a) shows the performance of sideways cracking for the number of projected attributes varying from 1 to 10. We see that in contrast to standard cracking (see Figure 2.5(c)), sideways cracking scales more gracefully with the number of projected attributes. However, still the performance varies by up to one order of magnitude. Furthermore, sideways cracking duplicates the index key in all cracker maps. So the question now is, can we have a cracking approach which is less sensitive to the number of projected attributes?

Covered Cracking

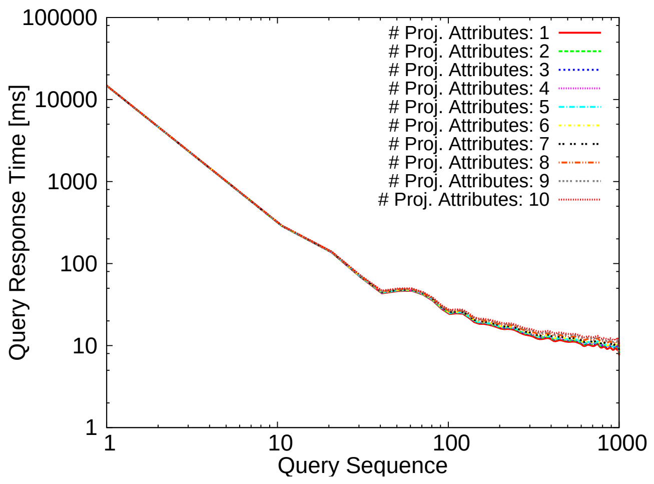
Group multiple non-key attributes with the cracked column in a cracker map. At query time, crack all covered non-key attributes along with the key column for even more efficient tuple reconstruction.

Note that with sideways cracking all projected columns are aligned with each other. However, the query engine still needs to fetch the projected attribute values from different columns in different cracker maps. These lookup costs turn out to be very expensive in addition to the overhead of cracking n replicas of the indexed column for n projected attributes. To solve this problem, we generalize sideways cracking to cover the n projected attributes in a single cracker map. In the following we term this approach *covered cracking*. While cracking, all covered attributes of a cracker map are reorganized with the cracked column. As a result, all covered attributes are aligned and stored in a consecutive memory region, i.e. no additional fetches are involved if the accessed attribute is covered. However, the drawback of this approach is that we need to specify which attributes to cover. To be on a safer side, we may cover all table attributes. However, this means that we will need to copy the entire table for indexing. We can think of fixing this by adaptively covering the cracked column, i.e. not copying the covered attributes upfront but rather on-demand when they are accessed. An option is to copy the covered attribute columns when they are accessed for the first time. An even more fine granular approach is to copy only the accessed values of covered attributes and thus reflecting the query access pattern in the covering status.

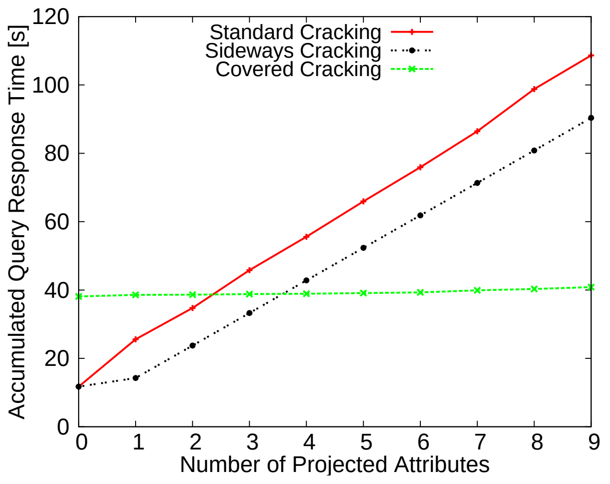
Figure 2.18(b) shows the performance of covered cracking over different numbers of projected attributes. Here we show the results from covered cracking which



(a) Varying Number of Projected Attributes for Sideways Cracking



(b) Varying Number of Projected Attributes for Covered Cracking



(c) Covering Tradeoff for Tuple Reconstruction

Figure 2.18: Comparing Tuple Reconstruction Cost of Standard, Sideways, and Covered Cracking

copies the data of all covered attributes in the beginning. We can see that covered cracking remains stable when varying the number of projected attributes from 1 to 10. Thus, covered cracking scales well with the number of attributes. Figure 2.18(c) compares the accumulated costs of standard, sideways, and covered cracking. We can see that while the accumulated costs of standard and sideways cracking grow linearly with the number of attributes, the accumulated costs of covered cracking remain pegged at under 40 seconds. We also see that sideways cracking outperforms covered cracking for only up to 4 projected attributes. For more than 4 projected attributes, sideways cracking becomes increasingly expensive whereas covered cracking remains stable. Thus, we see that covering offers huge benefits.

2.3.3 Improving Cracking Robustness

In this section we look at how well stochastic cracking [17] addresses the issue of query robustness and whether we can improve upon it. In Figure 2.14 we can observe that stochastic cracking is more expensive (for first as well as subsequent queries) than standard cracking. On the other hand, the random splits in stochastic cracking (MDD1R) create uniformly sized partitions. Thus, stochastic cracking trades performance for robustness. So the key question now is: can we achieve robustness without sacrificing performance? Can we have high robustness and efficient performance at the same time?

Coarse-granular Index

Create balanced partitions using range partitioning upfront for more robust query performance. Apply standard cracking later on.

Stochastic cracking successively refines the accessed data regions into smaller equal sized partitions while the non-accessed data regions remain as large partitions. As a result, when a query touches a non-accessed data region it still ends up shuffling large portions of the data. To solve this problem, we extend stochastic cracking to create several equal-sized[5] partitions upfront, i.e. we pre-partition the data into smaller range partitions. With such a *coarse-granular index* we shuffle data only inside a range partition and thus the shuffling costs are within a threshold.

[5]Please note that our current implementation relies on a uniform key distribution to create equal-sized partitions. Handling skewed distributions would require the generation of equi-depth partitions.

Note that in standard cracking, the initial queries have to anyways read huge amounts of data, without gathering any useful knowledge. In contrast, the coarse granular index moves some of that effort to a prepare step to create meaningful initial partitions. As a result, the cost of the first query is slightly higher than standard cracking but still significantly less than full indexing. With such a coarse-granular index users can choose to allow the first query to be a bit slower and witness stable performance thereafter. Also, note that the first query in standard cracking is anyways slower than a full scan since it partitions the data into three parts. Coarse-granular index differs from standard cracking in that it allows for creating *any* number of initial partitions, not necessarily three. Furthermore, by varying the number of initial partitions, we can trade the initialization time for more robust query performance. This means that, depending upon their application, users can adjust the initialization time in order to achieve a corresponding robustness level. This is important in several scenarios in order to achieve customer SLAs. In the extreme case, users can create as many partitions as the number of distinct data items. This results in a full index, has a very high initialization time, and offers the most robust query performance. The other extreme is to create only a single initial partition. This is equivalent to the standard cracking scenario, i.e. very low initialization time and least robust query performance. Thus, coarse-granular index covers the entire robustness spectrum between standard cracking and full indexing. Figure 2.19 visualizes the concept of the technique.

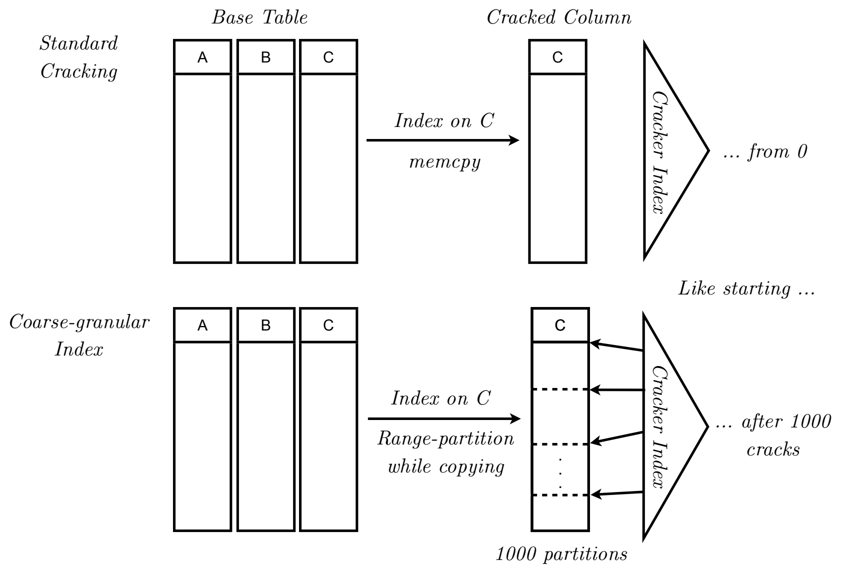


Figure 2.19: Visualization of the concept of coarse-granular index. While for standard cracking, the cracking starts with an unindexed cracker column, coarse-granular index pre-partitions the cracker column directly into e.g. 1000 partitions.

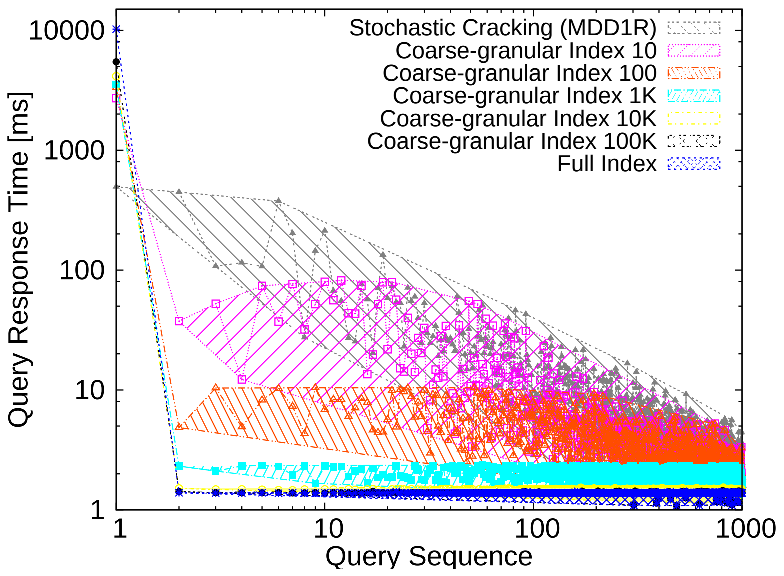


Figure 2.20: Variance in Response Time.

In Figure 2.20, we show the query response time region (convex hull) of different indexing methods, including stochastic cracking (MDD1R), coarse-granular index, and full index (quick sort + binary search). We vary the number of initial partitions, which are created in the first query by the coarse-granular index from 10 to 100, 000. While stochastic cracking (MDD1R) shows a variance similar to that of standard cracking, as observed in Figure 2.5(d), coarse-granular index reduces the performance variance significantly. In fact, for different number of partitions, coarse-granular index covers the entire space between the high-variance standard cracking and low-variance full index.

Figure 2.21 shows the results. We can see that the initialization time of stochastic cracking (MDD1R) is very similar to that of standard cracking. This means that stochastic cracking (like standard cracking) shifts most of the indexing effort to the query time. On the other extreme, full sort does the entire indexing effort upfront, and thus has the highest initialization time. Coarse-granular index fills the gap between these two extremes, i.e. by adjusting the number of initial partitions we can trade the indexing effort at the initialization time and the query time. For instance, for 1, 000 initial partitions, the initialization time of coarse-granular index is 65% less than full index, while still providing more robust as well as more efficient query performance than stochastic cracking (MDD1R). In fact, the total query time of coarse-granular index with 1, 000 initial partitions is 41% less than stochastic cracking (MDD1R) and even 26% less than standard cracking. Thus, coarse-granular index allows us to combine the best of both worlds.

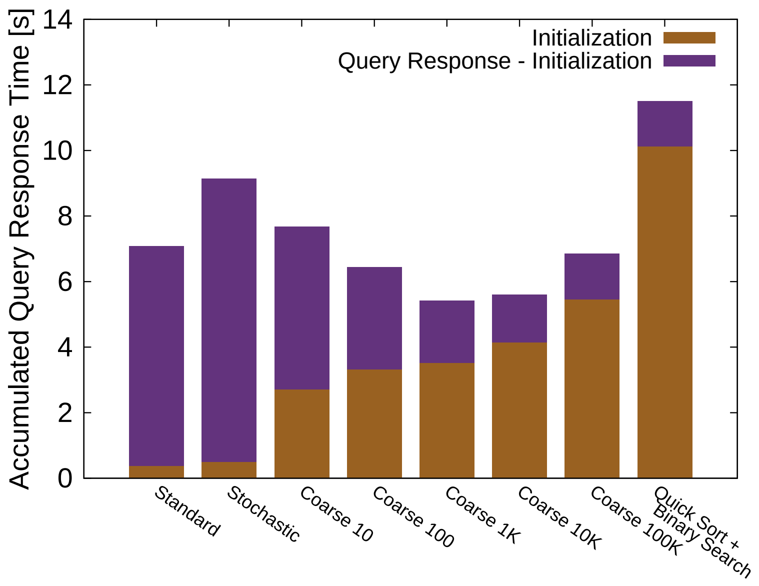


Figure 2.21: Initialization Time Tradeoff.

We can also extend the coarse-granular index and pre-partition the base table along with the cracker column. This means that we range partition the source table in exactly the same way as the adaptive index during the initialization step. Though, we still refine only the cracked column for each incoming query. The source table is left untouched. If the partition is small enough to fit into the cache, then the tuple reconstruction costs are negligible because of no cache misses. Essentially, we decrease the physical distance between external random accesses, i.e. the index entry and the corresponding tuple are *nearby clustered*. This increases the likelihood that tuple reconstruction does not incur cache misses. Thus, as a result of pre-partitioning the source table, we can achieve more robust tuple reconstruction without covering the adaptive index itself, as in covered cracking in Section 2.3.2. However, we need to pick the partition size judiciously. Larger partitions do not fit into the cache, while smaller partitions result in high initialization time. Note that if the data is stored in row layout, then the source table is anyways scanned completely during index initialization and so pre-partition is not too expensive. Furthermore, efficient tuple reconstruction using nearby clustering is limited to one index per table, same as for all primary indexes.

Figure 2.22 shows the effect of pre-partitioning the source table. We create both 100 and 1,000 partitions. The cost of pre-partitioning the source table is included in the accumulated query response time of coarse-granular index. Both standard cracking and coarse-granular index in Figure 2.22 start with perfectly aligned tuples. However, in standard cracking, the locality between index entry

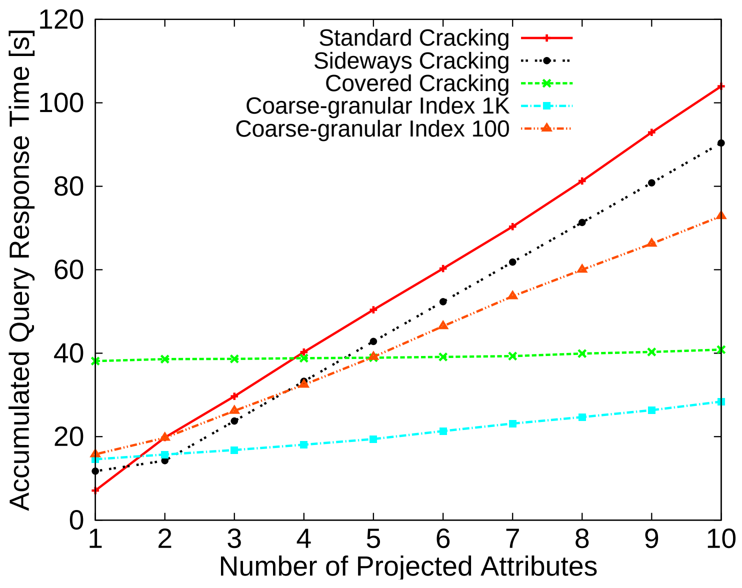


Figure 2.22: Extending 2.18(c) by Nearby Clustering.

and corresponding tuple decreases gradually and soon the cache misses caused by random accesses destroy the performance. Coarse-granular index, on the other hand, exploits the nearby clustering between the index entry and the corresponding tuple. Since tuples are never swapped across partitions, the maximum distance between an index entry and the corresponding tuple is at most the size of a partition. Thus, we can see from Figure 2.22 that coarse-granular index has a much more stable performance when scaling the number of projected attributes, without reorganizing the base table at query time. In fact, coarse-granular index 1K even outperforms covered cracking for any number of projected attributes.

For example, when projecting all 10 attributes, coarse-granular index 1K is 1.7 times faster than covered cracking, 3.7 times faster than sideways cracking, and 4.3 times faster than standard cracking. However, for 1,000 table partitions, each partition has a size of 8MB and thus fits completely in the CPU cache. For 100 partitions the partition size increases to 80MB and thus, it is over 6.5 times larger than the cache. The results show that the concept still works. Although coarse-granular index 100 is slower than covered cracking for more than 4 attributes, it is still faster than sideways and standard cracking for more than 3 attributes. It holds: the fewer partitions that we create the closer is the performance to that of standard cracking. To strengthen the robustness evaluation, we scale all experiments from Figure 2.20, 2.21 and 2.22 to a dataset containing 1 billion entries. As we want to inspect how well the methods scale with the data size, Table 2.2 shows the factor of increase in time when switching from 100 million to 1 billion

FACTOR SLOWER (FROM 100M to 1B)	INITIALIZATION	REMAINING	TOTAL
STANDARD CRACKING	10.01	9.92	9.93
STOCHASTIC CRACKING (MDD1R)	12.92	9.57	9.75
COARSE GRANULAR INDEX 10	11.73	9.92	10.56
COARSE GRANULAR INDEX 100	11.72	9.81	10.79
COARSE GRANULAR INDEX 1K	11.69	9.96	11.09
COARSE GRANULAR INDEX 10K	11.31	9.94	10.95
COARSE GRANULAR INDEX 100K	10.90	10.02	10.73
FULL INDEX	11.48	9.97	11.29
SIDEWAYS CRACKING	-	-	11.92
COVERED CRACKING	-	-	9.98
COARSE GRANULAR INDEX 100 (NEARBY CLUSTERED)	-	-	11.64
COARSE GRANULAR INDEX 1K (NEARBY CLUSTERED)	-	-	13.33

Table 2.2: Scalability under Datasize Increase by Factor 10

entries. For an increase in data size by factor 10, an algorithm that scales linearly is 10 times slower. Obviously, all tested methods scale very well. As expected, only nearby clustering suffers from larger partitions which exceed the cache size by far. Overall, we see that coarse-granular index offers more robust query performance both over arbitrary selection predicates as well as over arbitrary projection attributes.

Finally, Table 2.3 classifies the three cracking extensions discussed above — buffered swapping, covered cracking, and coarse-granular index — along the same dimensions as discussed in Section 2.2.6. Please note that the entry of coarse-granular index classifies only the initial partitioning step as it can be combined with various other cracking methods as well.

DIMENSIONS	CATEGORY	NO INDEX	BUFFERED SWAPPING	COVERED CRACKING	COARSE GRANULAR INDEX	FULL INDEX
NUMBER OF SPLIT LINES	ZERO	■				
	FEW		■	■		
	SEVERAL				■	
	ALL		■			■
SPLIT STRATEGY	NONE	■				
	QUERY BASED		■	■		
	RANDOM					
	DATA BASED		■	■	■	■
SPLIT TIMING	NEVER	■				
	PER QUERY		■	■		
	UPFRONT				■	■

Table 2.3: Classification of Extended Cracking Algorithms.

2.4 Extending Cracking Experiments

In this section, we compare cracking with different sort and index baselines in detail. Our goal here is to understand how good or bad cracking is in comparison to different full indexing techniques. In the following, we first consider different sort algorithms, then different index structures, and finally the effect of query selectivity.

2.4.1 Extending Sorting Baselines

The typical baseline used in previous cracking works was a full index wherein the data is fully sorted using quick sort and queries are processed using binary search to find the qualifying tuples. Sorting is an expensive operation and as a result the first fully sorted query is up to 30 times slower than the first cracking query (see Figure 2.3). So let us consider different sort algorithms.

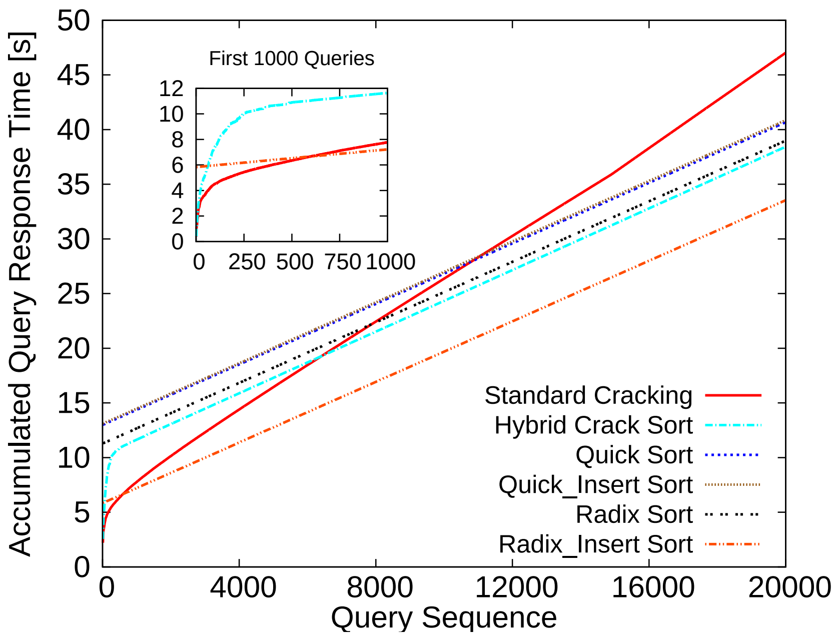


Figure 2.23: Comparing Different Sort Algorithms.

Quick sort is a reasonably good (and cache-friendly) algorithm, better than other value-based sort algorithms such as insertion sort and merge sort. But what about radix-based sort algorithms [18]? We compared quick sort with an in-place radix sort implementation [7]. This radix sort implementation switches to insertion sort (lets call this radix-insert) when the run length becomes smaller than 64. This algorithm is a recursive (in place) **M**ost **S**ignificant **D**igit radix sort, called left radix sort in [33].

What the algorithm of [33] does to work in place is the following: The input gets (symbolically) divided into $r = 2^m$ buckets, where m is the number of bits of the sorting digits. Thus, r represents the number of different values of the sorting digits. Now, instead of exchanging the keys between two arrays according to the value of the keys in the sorting digit, as in a traditional radix sort, the algorithm of [33] works in permutation cycles à la Cuckoo [37]. That is, it places an element in its correct bucket, with respect to the value of its sorting digit. In doing so, it evicts another element which is then placed in its correct bucket, as so on and so forth. Eventually, an element gets placed in the bucket of the very first element that initiated this permutation cycle. Then, the next element that has not been moved yet starts another permutation cycle, and so on. Eventually, all elements get moved to their corresponding buckets. At that point, the algorithm recurs in *each* of the r bucket that the input was (symbolically) divided into. This ensures that all the work done previously is not destroyed. In our implementation we treat the keys as eight-digit numbers, each digit consisting of 8 bits. Thus, $r = 2^8 = 256$.

We applied a similar switching to quick sort as well (we call it quick-insert). Figure 2.23 shows the accumulated query response times for binary search in combination with several sorting algorithms. We compare these with standard cracking and hybrid crack sort. The initialization times (i.e. the time to sort) for quick sort, quick-insert sort, and pure radix sort around 10 seconds are included in the first query. However, the initialization time for radix-insert sort drops by half to around 5 seconds. As a result, the first query with radix-insert is only 14 times slower, compared to 30 times slower with quick sort, than the first standard cracking query. Furthermore, we can clearly identify the number of queries at which one methods pays off over another. Already after 600 queries radix-insert sort shows the smaller accumulated query response times than standard cracking. For the two quick sort variants it takes 12,000 queries to beat standard cracking.

2.4.2 Extending Index Baselines

Let us now consider different index structures and contrast them with simple binary search on sorted data. The goal is to see whether or not it makes sense to use a sophisticated index structure as a baseline for cracking. We consider three index structures: (i) AVL-Tree [2], (ii) B+-Tree [6], and (iii) the very recently proposed ART [31]. We choose ART since it outperforms other main-memory optimized search trees such as CSB+-Tree [42] and FAST [28].

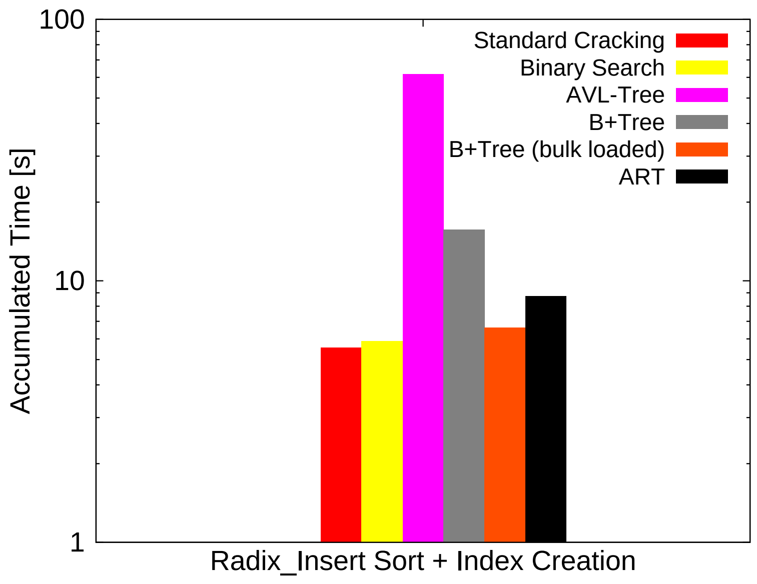


Figure 2.24: Indexing Effort of Diff. Indexes.

Let us first see the total indexing effort of different indexing methods over 1000 queries. For binary search, we simply sort the data (radix_insert sort) while for other full indexing methods (i.e. AVL-Tree, B+-Tree, and ART) we load the data into an index structure in addition to sorting (radix_insert sort). Standard cracking self-distributes the indexing effort over the $1,000$ queries while the remaining methods perform their sorting and indexing work in the first query. For the B+-Tree we present two different variants: one that is bulk loaded and one that is tuple-wise loaded. Figure 2.24 shows the results.

We can see that AVL-Tree is the most expensive while standard cracking is the least expensive in terms of indexing effort. The indexing efforts of binary search and B+-Tree (bulk loaded) are very close to standard cracking. However, the other B+-Tree as well as ART do more indexing effort, since both of them load the index tuple-by-tuple[6]. The key thing to note here is that bulk loading an index structure adds only a small overhead to the pure sorting. Let us now see the query performance of the different index structures. Figure 2.25 shows the per-query response times of different indexing methods. Surprisingly, we see that using a different index structure has barely an impact on query performance. This is contrary to what we expected and in the following let us understand this in more detail.

[6]The available ART implementation does not support bulk loading.

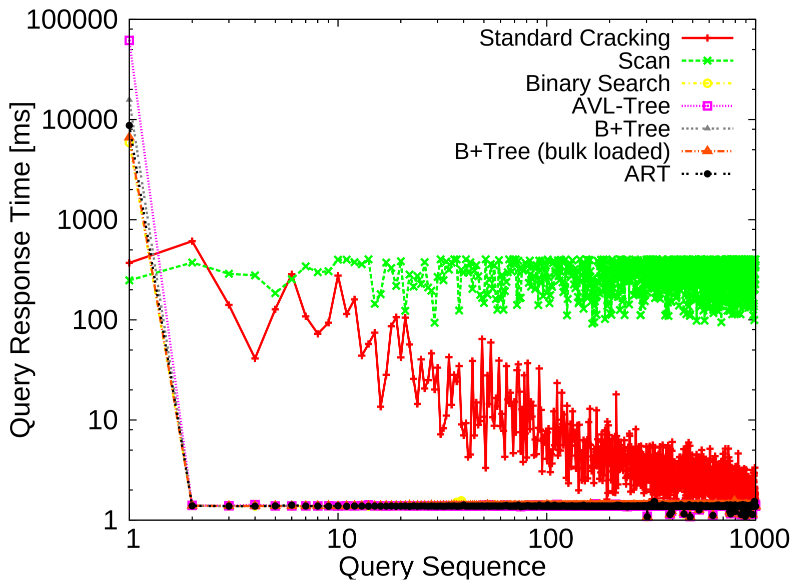
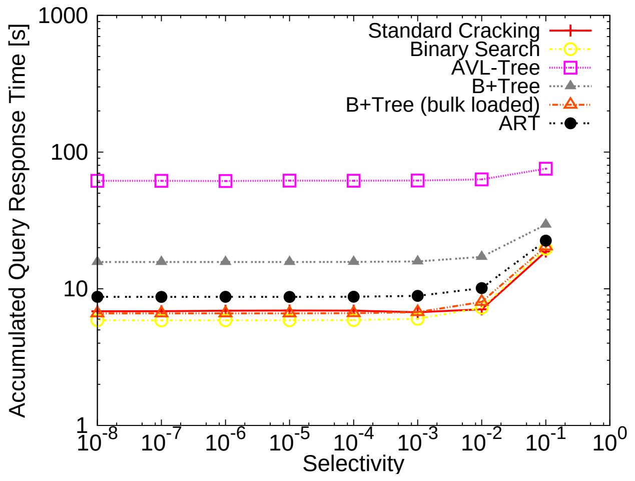


Figure 2.25: Per-Query Response Time of Diff. Indexes.

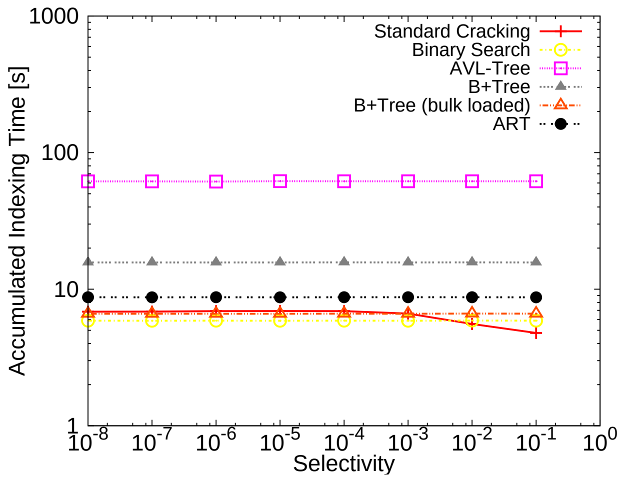
2.4.3 Effect of Varying Selectivity

To better understand this effect let us now vary the tuple selectivity of queries. Recall that we used a selectivity of 1% in all previous experiments. Selectivity is given as fraction of all entries. Figure 2.26(a) shows the accumulated query response times of different methods when varying the selectivity. We can see that the accumulated query response times change over varying selectivity for standard cracking, binary search, B+-Tree (bulk loaded), and ART. However, there is little relative difference between these methods over different selectivities. To dig deeper, let us split the query response time into two components: (i) the indexing costs to sort the data and to build the structure, and (ii) the index lookup and data access costs to retrieve the result.

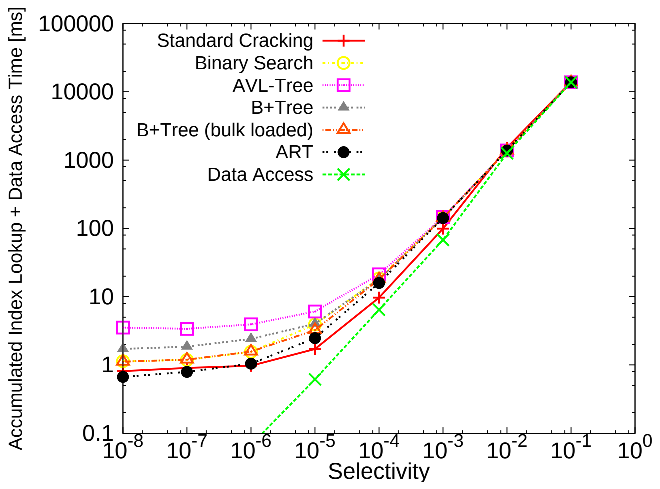
Figure 2.26(b) shows the accumulated indexing time for different methods when varying selectivity. Obviously, the indexing time is constant for all full indexing methods. However, the total indexing time of standard cracking changes over varying query selectivity. In fact, the indexing effort of standard cracking decreases by 45% when the selectivity changes from 10^{-5} to 10^{-1}. As a result, the indexing effort by standard cracking surpasses even the effort of binary search (more than 18%) and B+-Tree (bulk loaded) (more than 5%), both based on radix_insert sort for as little as 1,000 queries. The reason standard cracking depends on selectivity is that with high selectivity the two partition boundaries of a range query are located close together and the index refinement per query is small. As a result



(a) Accumulated Query Response Time



(b) Accumulated Indexing Time



(c) Acc. Index Lookup + Data Access Time

Figure 2.26: Comparing Standard Cracking with Index Baselines while Varying Selectivity (Note that: (a) = (b) + (c))

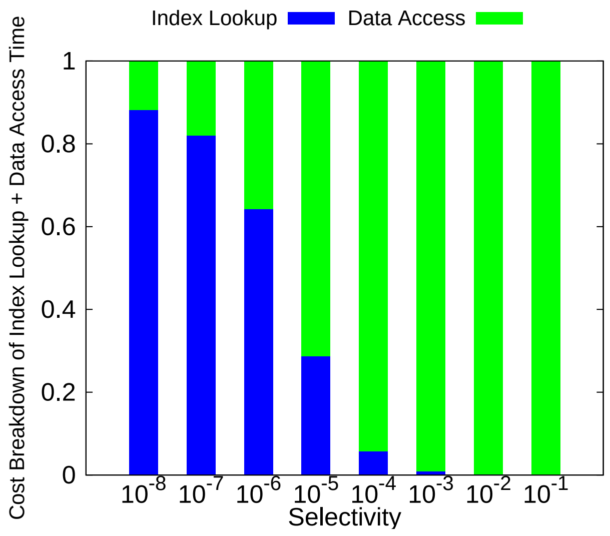
several data items are shuffled repeatedly over and over again. This increases the overall indexing effort as well as the time to converge to a full index.

Figure 2.26(c) shows the accumulated index lookup and data access costs of different methods over varying selectivity. We can see that the difference in the querying costs of different methods grows for higher selectivity. For instance, AVL-Tree is more than 5 times slower than ART for a selectivity of 10^{-8}. We also see that standard cracking is the most lightweight method in terms of the index lookup and data access costs and is closely followed by ART. However, for high selectivities, the index lookup and data access costs are small compared to the indexing costs. As a result, the difference in the index lookup and data access costs of different methods is not reflected in the total costs in Figure 2.26(a).

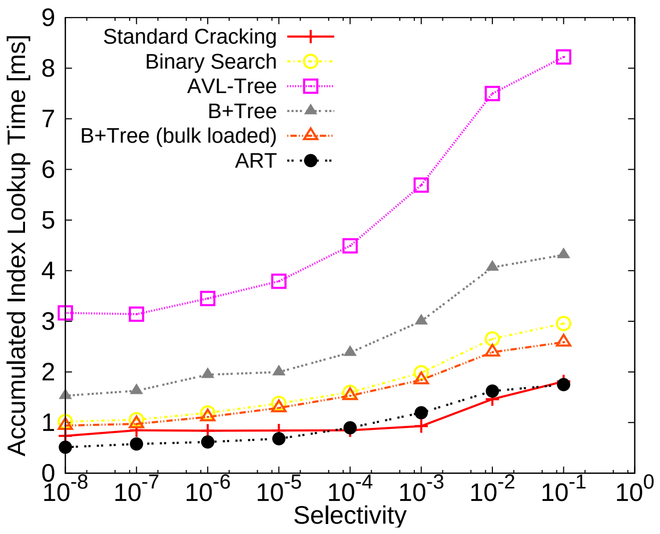
Let us now investigate the index lookup costs and the data access costs in comparison. Figure 2.27(a) shows the index lookup and data access costs as a fraction of the costs of Figure 2.26(c) for ART. We can see that the data access costs dominate the total time for a selectivity lower than 10^{-6}. This means that using better optimized index structures make sense only for very high query selectivities. Figure 2.27(b) shows only the index lookup costs without the data access costs of different methods when varying selectivity. We can see that the index lookup costs vary with selectivity, indicating different cache behavior for different query selectivities. Furthermore, we see that ART has the best index lookup times. For a selectivity of 10^{-8}, ART performs 30% faster index lookups than standard cracking. This is even though cracking has a much smaller index (created over just $1,000$ queries) whereas ART creates a full index over 100M data entries.

Finally, we also vary the number of queries to see how the index lookup times of standard cracking change in comparison to other indexing methods. Figure 2.27(c) shows the average per-query index lookup times for different methods when increasing the number of queries in the query sequence from 10 to 1M. We fix the query selectivity to 10^{-8}. Furthermore, we show stochastic cracking (MDD1R), coarse granular index 1K, and hybrid crack sort as they introduce additional split lines or handle them differently. We can see that the average index lookup time of standard cracking increases by almost one order of magnitude when the number of queries increase from 10 to 1M. This is because the cracker index grows with the number of queries[7]. Coarse-granular index and stochastic cracking (MDD1R) differ from standard cracking by showing a higher average index lookup time in the beginning as the additional splits weigh in that early phase. Hybrid crack

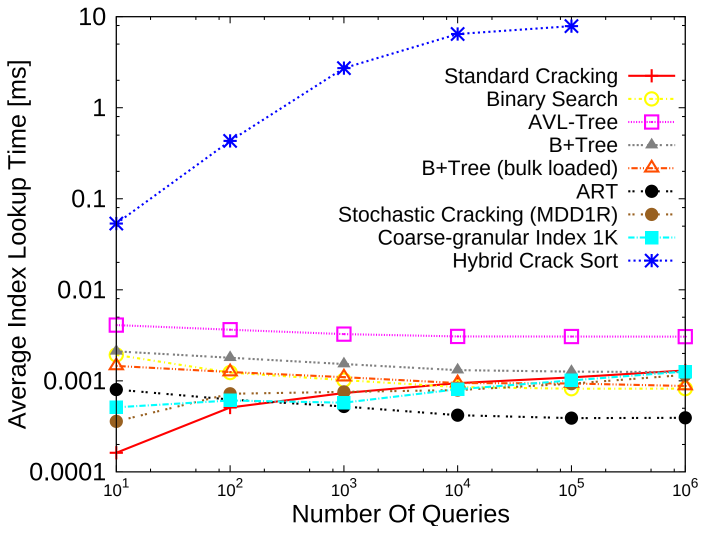
[7][20] proposed to stop cracking if a sufficiently small partition size is reached. However, this pays off only for a very large number of queries. As we apply $1,000$ queries in nearly all experiments, we use unbounded algorithms throughout this chapter. Nevertheless, we test the effect of stopping cracking at a certain threshold in Section 2.4.6.



(a) Cost Breakdown of Index Lookup and Data Access Time of ART



(b) Accumulated Index Lookup Time in Isolation



(c) Average Index Lookup Time (Sel. 10^{-8})

Figure 2.27: Lookup and Data Access of Standard Cracking and Index Baselines under Variation of Selectivity

sort shows the overall highest average lookup time which even increases with the number of queries[8]. The high selectivity leads to slow convergence and triggers repeatedly expensive lookups into the $10,000$ initial partitions. In contrast to that, the average per-query index lookup time of other indexing methods remains stable (or even improves slightly due to better cache performance) with the number of queries. Consequently, for 1M queries, the average index lookup time of ART is 3.6 times smaller than the average index lookup time of standard cracking.

To conclude, the take-away message from this section is three-fold: (i) using a better index structure makes sense only for very high selectivities, e.g. one in a million, (ii) cracking depends on query selectivity in terms of indexing effort, and (iii) although cracking creates the indexes adaptively, it still needs to catch up with full indexing in terms of the *quality* of the index.

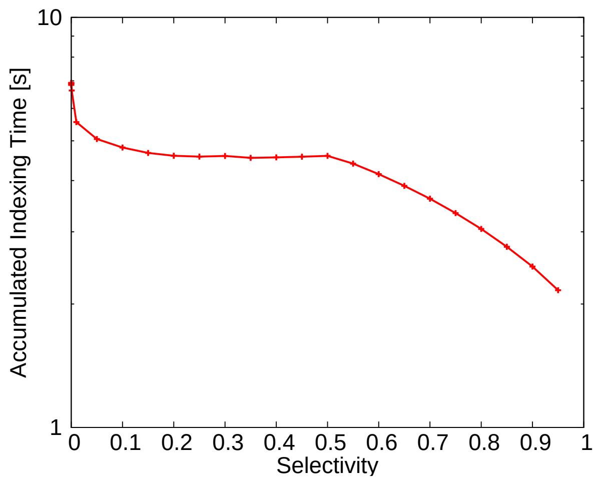
2.4.4 Effect of Very Low Selectivity

In the previous section we tested the methods under selectivities ranging from 10% down to point queries. Under a workload of such selectivities, standard cracking applies cracks over the entire column. However, when facing lower selectivities, cracking can show off its strength of indexing only the parts that are actually relevant for query answering. Thus, let us now see how low selectivities influence the indexing effort that standard cracking has to invest.

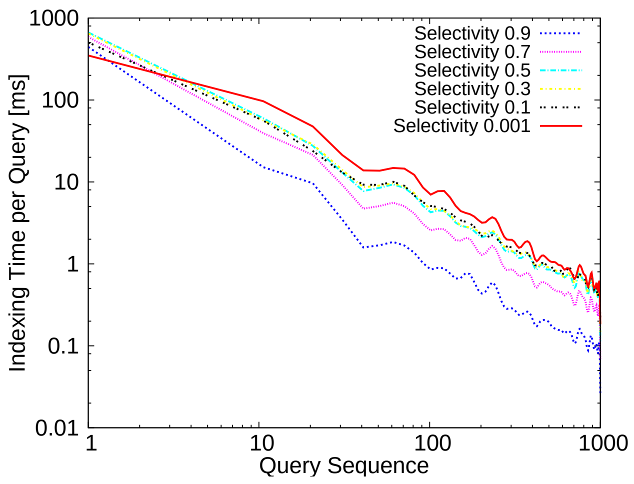
Figure 2.28(a) shows the accumulated indexing time when varying selectivity from 10^{-8} to 0.95, i.e. we now extend Figure 2.26(b) to very low selectivities. As intended by the cracking paradigm, the indexing effort decreases continually as we decrease the selectivity. In particular, the indexing effort drops significantly (almost 52%) after 0.50 selectivity. To understand this behaviour, let us see the actual split lines in the cracker index. Figure 2.29(a) shows the split lines with 0.001 selectivity and (b) shows the split lines with 0.9 selectivity for standard cracking. We can see that while high selectivity results in split lines in the entire column, very low selectivity results in split lines only at the two ends of the column, basically resembling a partial index [51].

Thus, very low selectivities (50% or lower) do less indexing effort and yet converge better since they only index the cracker column partially, as we show in Figure 2.28(b). We also tried very low selectivities on stochastic cracking (MDD1R) to see whether it indexes the cracker column more uniformly. Figure 2.29(c) shows the split lines created by stochastic cracking (MDD1R) with 0.9 selectivity queries.

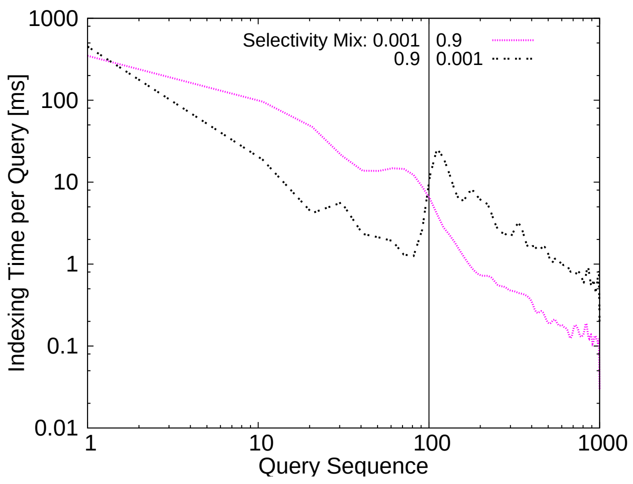
[8]The point for 10^6 queries is missing as the space consumption of 10K AVL-Trees each storing up to 1M entries exceeds capacity.



(a) Influence of Selectivity on Indexing Effort



(b) Influence of Selectivity on Convergence



(c) Effect of Mixed Selectivities

Figure 2.28: Cracking over Very Low Selectivities.

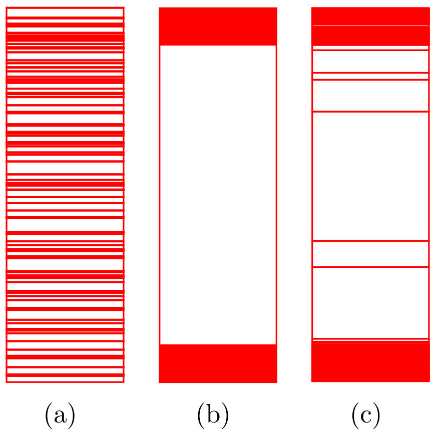


Figure 2.29: Split Lines for (a) Standard Cracking with Selectivity 0.001, (b) Standard Cracking with Selectivity 0.9, and (c) Stochastic Cracking (MDD1R) with Selectivity 0.9.

We can see that stochastic cracking does a bit better by creating a few split lines in the middle. However, most split lines are still concentrated towards either end. Finally, let us see how standard cracking behaves if query selectivity changes. Figure 2.28(c) shows two curves with changing selectivities. The selectivity changes from 0.001 to 0.9 in one curve and the other way around in the other curve. We can see that there is a jump of more than an order of magnitude when changing the selectivity from 0.9 to 0.001, demonstrating the downside of partial indexing. On the other hand, the runtimes drop sharply when varying the selectivity from 0.001 to 0.9 since the convergence is much faster in the partial index region.

Overall, we see that with very low selectivities cracking can display its ability to invest the minimal amount of necessary effort. On the flip side, we can also observe a certain sensitiveness to changes in query selectivities. This claim is also confirmed by the accumulated indexing times over all 1000 queries. A selectivity change from 0.9 to 0.001 leads to an accumulated indexing time of 14.98 seconds, whereas a change from 0.001 to 0.9 results in an accumulated indexing time of 9.96 seconds. Let us investigate this further below by measuring the per-query indexing times for different query selectivities. Figure 2.28(b) shows the result. We show smoothened curves for the ease of presentation. From Figure 2.28(b), we can see that standard cracking converges faster with very low selectivities than with high selectivities. For instance, after 1000 queries, standard cracking with 0.9 selectivity is almost 8 times faster than standard cracking with 0.001 selectivity. This is even though standard cracking with 0.9 selectivity puts 2.25 times less indexing effort than standard cracking with 0.001 selectivity.

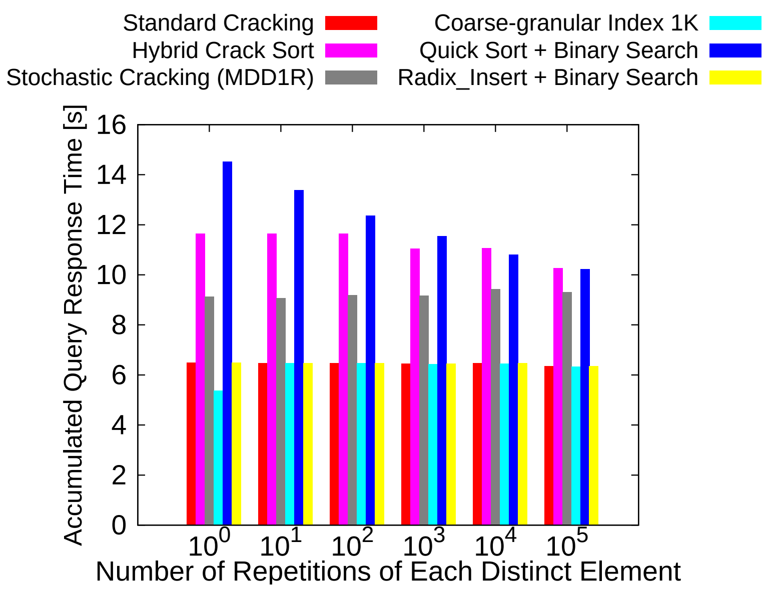


Figure 2.30: Varying the number of repeated elements in the dataset.

2.4.5 Effect of Duplicates in the Dataset

In all previous experiments, we used a dataset consisting of repeated keys. For 100 million elements, each key is replicated 1000 times, leading to a key range of $[0; 100,000]$. As database cracking is a method to provide lightweight indexing especially for non-key columns, we find it more realistic to test the methods under a setup of duplicated keys. Let us now see the impact of the column cardinality on the individual methods. In this experiment, we consider the main representatives of cracking, namely standard cracking, hybrid crack sort, stochastic cracking (MDD1R), and coarse-granular index with 1000 partitions. Furthermore, we test quick sort and radix_insert sort in combination with binary search. The number of duplicates is varied from 10^0, representing unique keys, to 10^5, where each key is repeated $100,000$ times. Figure 2.30 shows the accumulated query response time. Interestingly, the repetition factor has no significant impact on the cracking methods. The accumulated query response times of standard and stochastic cracking remain basically unchanged under variation of the repetition factor, only hybrid crack sort worsens slightly with the cardinality of the column. As quick sort basically applies the concept of cracking till the sorted state, we expect to see a similar behavior there. However, quick sort drastically worsens from 10^5 to 10^0 repetitions by a factor of 1.42x. This is caused by the fact that the additional effort of reordering unique elements comes into play at deeper recursion levels, which are never reached by the 1000 queries (2000 cracks) that cracking has to apply. In contrast to that, radix sort behaves independently of the repetition factor.

2.4.6 Effect of Varying Cracking Depth

The cracking algorithms tested so far reorganize the cracker column till the fully sorted state is reached. However, the authors of [20] suggested in their original work, that it might make sense to stop further reorganization at a certain partition size and filter the partitions instead. Thus, in the following experiment, we vary the threshold at which we stop applying standard cracking and inspect the effect on the runtime. Figure 2.31 shows the results. We present the accumulated query response time over our query sequence of 1000 queries and split the bars into indexing time, representing the time to reorganize the partition(s), and index lookup with data access time, representing the query result computation. This result computation corresponds to a simple scan if the column has been cracked by this query or a scan with a filtering, if no cracking has been performed previously. The threshold at which we stop cracking is varied from 16,000 (250KB) to 256,000 (4000KB) entries.

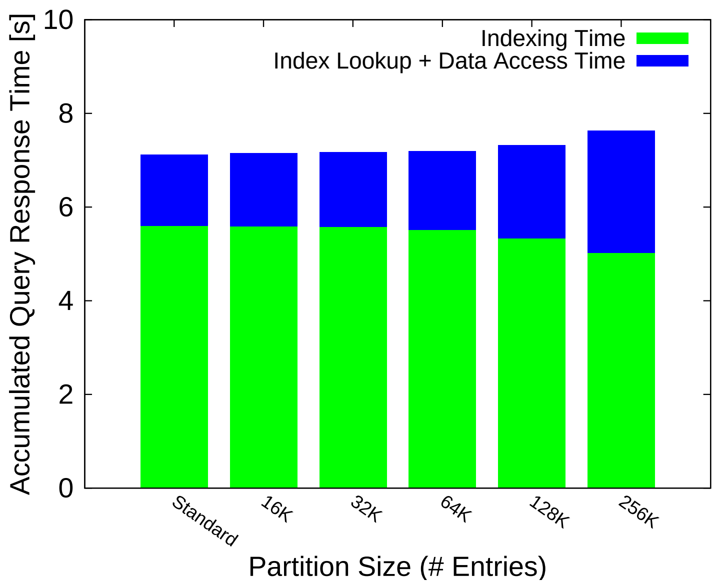


Figure 2.31: Stopping cracking at a certain partition size.

In Figure 2.31 we can observe that a threshold larger than 64,000 entries has a clear impact on the accumulated query response time. The larger the threshold, the smaller is the indexing time as less cracking effort is needed. As a consequence of the reduced indexing effort, the querying time increases due to the additional filtering. Unfortunately, the savings in indexing effort are eclipsed by the larger increase in querying time. As a result, the overall runtime increases. Therefore, stopping cracking at a certain partition size and applying filtering does not improve performance.

2.4.7 Effect of Query Access Pattern

So far, all experiments applied a uniform random access pattern to test the methods. However, in reality, queries are often logically connected with each other and follow certain non-random and non-uniform patterns. To evaluate the methods under such patterns, we pick two representatives: the *sequential access pattern* and the *skewed access pattern*.

We create the sequential access pattern as follows: starting from the beginning of the value domain, the queried range is moved for each query by half of its size towards the end of the domain to guarantee overlapping query predicates. When the end is reached, the query range restarts from the beginning. The position to begin is randomly set in the first 0.01% of the domain to avoid repetition of the same sequence in subsequent rounds. Figure 2.32 visualizes the generated predicates.

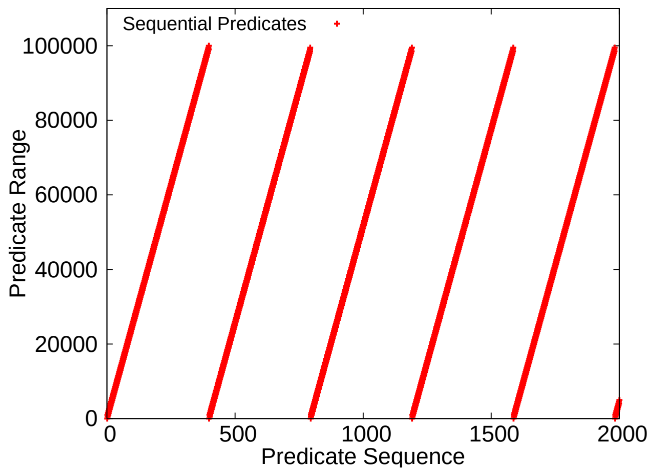
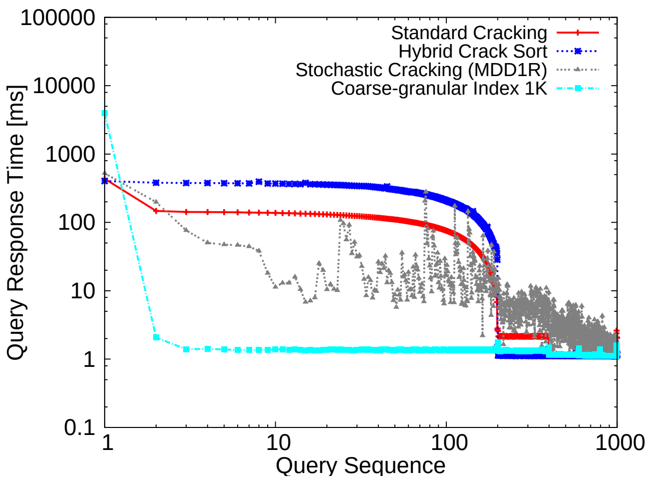
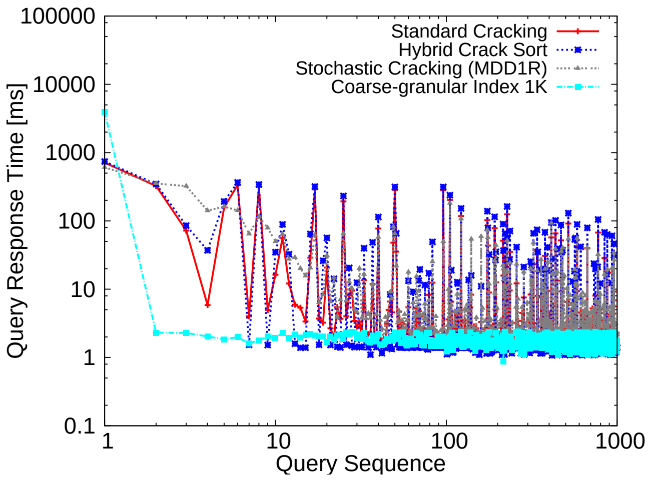


Figure 2.32: Generated Predicates for Sequential Pattern.

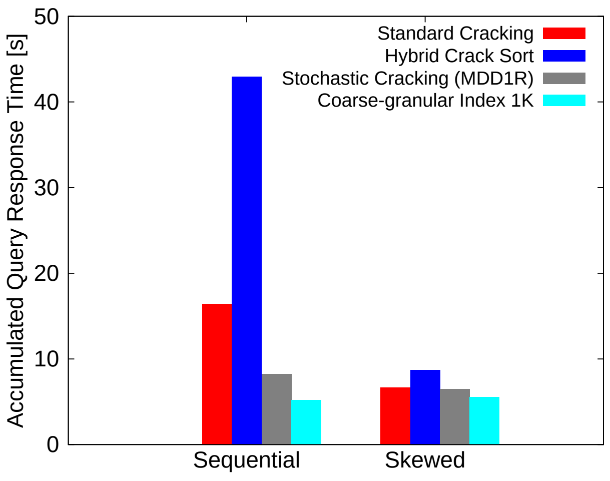
In Figure 2.33(a) we show the query response time under the sequential access pattern for standard cracking, stochastic cracking, coarse-granular index with 1,000 partitions, and hybrid crack sort. We can clearly separate the figure into the first 200 queries and the remaining 800 queries. As the selectivity is 1% and the query



(a) Sequential Access Pattern



(b) Skewed Access Pattern



(c) Accumulated Query Response Times

Figure 2.33: Effect of Query Access Pattern on Adaptive Methods

range moves by half of its size per query, it takes 200 queries until the entire data set has been accessed. Within that period the query response time of standard cracking and hybrid crack sort decreases only gradually. Large parts of the data are scanned repeatedly and the unindexed upper part decreases only slightly per query. Furthermore, hybrid crack sort is considerably slower than standard cracking in this phase. Stochastic cracking reduces this problem significantly by applying additional splits to the unindexed upper area. Coarse-granular index shows the most stable performance. After the initial partitioning in the first query, the query response time does not significantly vary. Additionally, the query response time is the lowest of all methods (except for the first query). For the remaining 800 queries the performance differences between all methods decrease as the entire data set has been queried and is therefore cracked more or less. Now, stochastic cracking is slower than standard cracking as the additional effort of random cracking and materializing the result is no more necessary to provide a decent performance.

Finally, let us investigate how the methods perform under a skewed access pattern. We create the skewed access pattern in the following way: first, a zipfian distribution is generated over n values, where n corresponds to the number of queries. Based on that distribution the domain around the hotspot, which is the middle of the domain in our case, is divided into n parts.

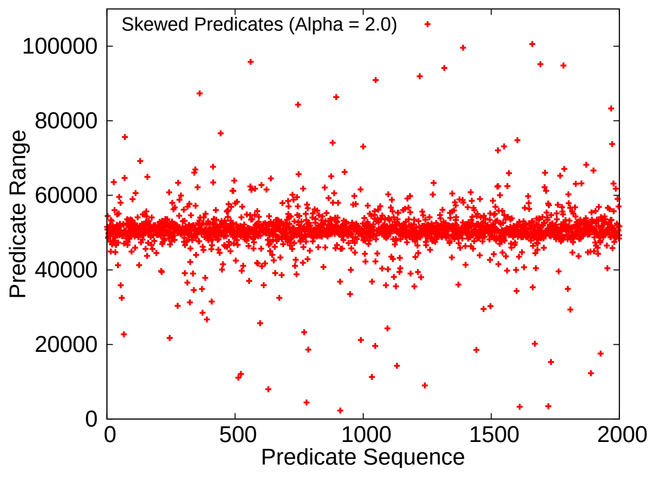


Figure 2.34: Generated Predicates for Skewed Pattern.

After that the query predicates are set according to the frequencies in the distribution. The k values with the l highest frequency in the distribution lead to k query predicates lying in the l-th nearest area around the hotspot. Figure 2.34 shows the generated predicates for $\alpha = 2.0$. These predicates are randomly shuffled before they are used in the query sequence. Figure 2.33(b) shows the query response time for the skewed pattern. We can observe a high variance in all methods except coarse-granular index.

Between accessing the hotspot area and regions that are far off, the query response time varies by almost 3 orders of magnitude. Early on, all methods index the hotspot area heavily as most query predicates fall into that region. Stochastic cracking manages to reduce the negative impact of predicates that are far off the hotspot area. However, it is slower than standard cracking if the hotspot area is hit. Hybrid crack sort copies the hotspot area early on to its final partition and exhibits the fastest query response times in the best case. However, if a predicate requests a region that has not been queried before, copying from the initial partitions to the final partition is expensive.

Finally, Figure 2.33(c) shows the accumulated query response time for both sequential and skewed access patterns. Obviously handling sequential patterns is challenging for all adaptive methods. Especially hybrid crack sort suffers from large repetitive scans in all initial partitions and is therefore by far the slowest method in this scenario. Stochastic cracking (MDD1R) manages to reduce the problems of standard cracking significantly and fulfills its purpose by providing a workload robust query answering. In total, coarse-granular index is the fastest method under this pattern. Overall, for the skewed access pattern the difference between the methods is significantly smaller than for the sequential pattern.

2.5 Conclusion

Let us now put together the major lessons learned in this chapter.

1. **Database cracking is a mature field of research.** Database cracking is a simple yet effective technique for adaptive indexing. In contrast to full indexing, database cracking is lightweight, i.e. it does not penalize the first query heavily. Rather, it incrementally performs at most one quick sort step for each query and nicely distributes the indexing effort over several queries. Moreover, database cracking indexes only those data regions which are actually touched by incoming queries. As a result, database cracking fits perfectly to the modern needs of adaptive data management. Furthermore, apart from the incremental index creation in standard cracking, several other follow-up works have looked into other aspects of adaptive indexing as well. These include updating a cracked database, convergence of database cracking to a full index, efficient tuple reconstruction, and robustness over unpredictable changes in query workload. Thus, we can say that database cracking has

come a long way and is a mature field of research.

2. **Database cracking is repeatable.** In this chapter, we repeated five previous database cracking works, including standard cracking using crack-in-two and crack-in-three [20], predication cracking [39], hybrid cracking [23], sideways cracking [22], and stochastic cracking [17]. We reimplemented the cracking algorithms from each of these works and tested them under similar settings as in the previous works. Our results match very closely to the ones presented in the previous works and we can confirm the findings of those works, i.e. hybrid cracking indeed improves in terms of convergence to full index, sideways cracking allows for more efficient tuple reconstruction, and stochastic cracking offers more predictable query performance than standard cracking. We can say that cracking is repeatable in any ad-hoc query engine, other than MonetDB as well.

3. **Still, lot of potential to improve database cracking.** There is still a lot of potential to do better in several aspects of database cracking, including faster convergence to full index, more efficient tuple reconstruction, and more robust query performance. For example, by buffering the elements to be swapped in a heap, we can reduce the number of swaps and thus have better convergence. Similarly, by covering the cracked index we can achieve better scalability in the number of projected attributes. Likewise, we can trade the initialization time to create a coarse-granular index which improves query robustness. All these are promising directions in the database cracking landscape. Thus, we believe that even though cracking has come a long way, it still has a lot more to go.

4. **Database cracking depends heavily on the query access pattern.** As the presented techniques are adaptive due to their query driven character, each of them is more or less sensitive to the applied query access pattern. A uniform random access pattern can be considered the best case for all methods as it leads to uniform partition sizes across the data. In contrast to that sequential patterns crack the index in small steps and the algorithms have to rescan large parts of the data. Skewed access patterns lead to a high variance in runtime depending on whether the query predicate hits the hotspot area or not. Overall, stochastic cracking (MDD1R) and coarse-granular index, which extend their query driven character by data driven influences, are less sensitive to the query access pattern than the methods that take only the seen queries into account.

5. **Workload selectivities affect the amount of indexing effort in database cracking.** Since cracking reorganizes only the accessed portions

of the data, the total indexing effort varies with the query selectivities. In fact, the total indexing effort in standard cracking drops by 45% when the selectivity changes from 10^{-5} to 10^{-1}. Although high selectivity queries reorganize smaller portions of the data, the reorganization happens much more often before reaching the final state. Additionally, earlier cracking works suggested to stop data reorganization at a certain partition size, in order to reduce the indexing effort. However, we saw that the overhead of additional filtering eclipses the savings from indexing effort.

6. **Database cracking needs to catch up with modern indexing trends.** We saw that for sorting radix sort is twice as fast as quick sort. After 600 queries the total query response time of binary search based on radix sorted data is even faster than standard cracking. This means that a full sorting pays-off over standard cracking in less than 1000 queries. Thus, we need to explore more lightweight techniques for database cracking to be competitive with radix sort. Furthermore, several recent works have proposed main-memory optimized index structures. The recently proposed ART has 1.8 times faster lookups than standard cracking after 1000 queries and 3.6 times faster lookups than standard cracking after $1M$ queries. We note two things here: (i) the cracker index offers much slower lookups than modern main-memory indexes, and (ii) the cracker index gets even worse as the number of queries increase. Thus, we need to look into the index structures used in database cracking and catch up with modern indexing trends.

7. **Different indexing methods have different signatures.** We looked at several indexing techniques in this chapter. Let us now contrast the indexing behavior of different indexing methods in a nutshell. To do so, we introduce a way to fingerprint different indexing methods. We measure the *progress of index creation* over the *progress of query processing*, i.e. how different indexing methods index the data over time as the queries are being processed (Figure 2.35). This measure essentially acts as a signature of different indexing methods. The x-axis shows the normalized accumulated lookup and data access time (querying progress) and the y-axis shows the normalized accumulated data shuffle and index update time (indexing progress). We can see that different indexing methods have different curves. For example, standard cracking gradually builds the index as the queries are processed whereas full index builds the entire index before processing any queries. Hybrid crack sort and hybrid sort sort have steeper curves than standard cracking, indicating that they build the index more quickly. On the other hand, stochastic cracking has a much smoother curve. Sideways and covered cracking perform large parts of their querying process already

in the first query by copying table columns into the index to speed up tuple reconstruction. It is interesting to see that each method has a unique curve which characterizes its indexing behavior. Furthermore, there is still lot more room to design adaptive indexing algorithms with even more different indexing signatures.

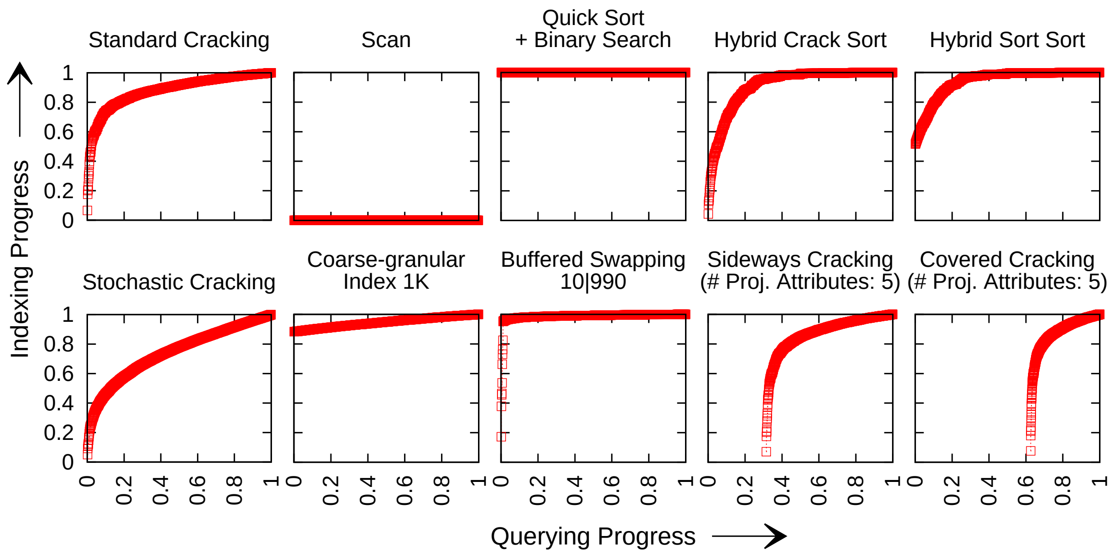


Figure 2.35: Signatures of Indexing Methods.

In the past chapter, we have discussed a vast amount of adaptive indexing algorithms, extended them in various ways, and tested them under different setups. To gather a deeper understanding of the methods themselves, we focused so far entirely on the single-threaded execution of them. Multi-threading introduces a new layer of challenges that must be tackled. This will be the focus of the upcoming chapter.

Chapter 3

Adaptive Indexing and Sorting in the Multi-threaded Context

As we have seen in the previous chapter, adaptive indexing is a concept that considers index creation in databases as a by-product of query processing, as opposed to traditional full index creation where the indexing effort is performed up front before answering any queries. As demonstrated, adaptive indexing has received a considerable amount of attention, and a large number of algorithms have been proposed over the past few years.

Until now, however, most adaptive indexing algorithms have been designed single-threaded, yet with multi-core systems already well established, the idea of designing parallel algorithms for adaptive indexing is very natural. In this regard, and to the best of our knowledge, only one parallel algorithm for adaptive indexing has recently appeared in the literature: A parallel version of standard cracking. Therefore, in this chapter we describe four alternative parallel algorithms for adaptive indexing, including a second variant of a parallel standard cracking algorithm. Additionally, we describe a hybrid parallel sorting algorithm, and a NUMA-aware method based on sorting. We then thoroughly compare all these algorithms experimentally in terms of single-column query answering performance.

After that, we extend the two most promising parallel adaptive indexing algorithms to support tuple reconstruction efficiently. We bring the concept of sideways cracking to the multithreaded level and experimentally evaluate it against the aforementioned hybrid parallel sorting algorithm, that we also extended to support multi-column queries.

The initial set of experiments considered in this chapter indicates that our parallel algorithms significantly improve over previously known ones. Our results also suggest that, although adaptive indexing algorithms are a good design choice in single-threaded environments, the rules change considerably in the parallel case. That is, in future highly-parallel environments, sorting algorithms could be serious alternatives to adaptive indexing.

3.1 Introduction

In Chapter 2, we presented a thorough experimental study of all major single-threaded adaptive indexing algorithms. The experiments we showed there support the claim that (single-threaded) standard cracking [20] still keeps being the algorithm any other new (single-threaded) algorithm has to improve upon, due to its simplicity and good accumulated runtime.

For example, in the comparison of standard cracking [20] with stochastic cracking [17], as seen from the experiments shown in Section 2.2.5, Figure 2.14, the latter is more robust, but the former is in general faster. With respect to the hybrid cracking algorithms [23], the experiments of the previous chapter indicate that, although convergence towards full index improves in hybrid cracking algorithms, this improvement can be seen only after roughly 100 to 200 queries, before that, standard cracking performs clearly better (Section 2.2.5, Figure 2.10). Moreover, as discussed in Section 2.4.1, Figure 2.23, with respect to the total accumulated query time, standard cracking is faster than hybrid cracking up to around 8000 queries. In all cases, standard cracking is the algorithm that is much easier to implement and to maintain.

In Chapter 2 we also observed that pre-processing the input before applying standard cracking significantly improves the convergence towards full index, robustness, and total execution time of standard cracking. This pre-processing step is simply a range-partitioning over the attribute to be (adaptively) indexed, coined *coarse-granular index* in the previous chapter. From the experiments we showed in Figures 2.20 and 2.21 in Section 2.3.3, the improvement of the coarse-granular index over all other adaptive indexing techniques considered can clearly be seen. However, the coarse-granular index also incurs in a higher initialization time, which, as discussed previously, is often not desirable.

3.1.1 Contributions

All algorithms tested in Chapter 2 were single-threaded, and actually, almost *all* adaptive indexing algorithms in the literature are designed for single-core systems. With multi-core systems not only on the rise, but actually well established by now, the idea of parallel adaptive indexing algorithms is very natural.

To the best of our knowledge, the work presented in [14, 15] as well as the work of [39] are the only ones so far that deal with adaptive indexing in multi-core systems. In [14, 15], a parallel version of the standard cracking algorithm is presented, while in [39] predication/vectorized cracking is extended to support multi-threading. In this chapter, we will improve the situation by making the following contributions:

1. First and foremost, the most important contribution of this chapter is the critical experimental evaluation of eleven parallel algorithms for adaptive indexing as well as for creating full indexes (seven methods for single-column query answering and four methods for multi-attribute queries) — including the algorithm of [14, 15, 39]. This work can be considered the very first experimental study of parallel adaptive indexing techniques.

2. We describe four (natural) alternative parallel algorithms for adaptive indexing, one based on standard cracking [20], one based on predication/vectorization cracking [39], while the other two are based on the coarse-granular index presented in Chapter 2.

3. We describe a hybrid parallel sorting algorithm. This algorithm greatly resembles the parallel radix sorting algorithms presented in [30, 34, 43]. Additionally, we present another method that further improves upon this hybrid parallel sorting algorithm by adding NUMA-awareness.

4. The set of experiments considered in this chapter suggests that our algorithms significantly improve over the methods presented in [14, 15]. Moreover, these experiments also indicate that, as opposed to the story with single-threaded algorithms where sorting algorithms are no match in practice against adaptive indexing techniques, parallel sorting algorithms could become serious alternatives to parallel adaptive indexing techniques.

Throughout the chapter, we mostly refer to the algorithms by their short names for simplicity. Additionally, for an easier distinction between single- and multi-threaded algorithms, the short name of each multi-threaded algorithm starts with a **P**.

Algorithm	**Reference**
Standard cracking	[20]
Predication/Vectorized cracking	[39]
Hybrid crack/radix/sort sort	[23]
Buffered swapping	this work
Stochastic cracking (MDD1R)	[17]
Coarse-granular Index	this work
Sideways cracking, Covered cracking	[22], this work
Sorting: Quick(_insert) sort, Radix(_insert)	[19],[18]
Full Index: AVL-tree, B+-tree, ART	[2],[6],[31]

Table 3.1: All single-threaded algorithms that were evaluated in Chapter 2.

Algorithm	**Abbreviation**	**Reference**
Parallel standard cracking	P-SC	[14, 15]
Parallel coarse-granular index	P-CGI	[3]
Parallel-chunked standard cracking	P-CSC	[3]
Parallel-chunked vectorized cracking	P-CVC	variant of [39]
Parallel-chunked coarse-granular index	P-CCGI	[3]
Parallel range-partitioned radix sort	P-RPRS	[3]
Parallel-chunked radix sort	P-CRS	[3]
Parallel sideways cracking (with P-CSC)	P-SW-CSC	this work
Parallel sideways cracking (with P-CCGI)	P-SW-CCGI	this work
Parallel range-partitioned radix sort (cluster complete)	P-PC-RPRS	this work
Parallel range-partitioned radix sort (cluster lazy)	P-LC-RPRS	this work

Table 3.2: All multi-threaded algorithms evaluated in Chapter 3.

3.2 Multi-threaded Algorithms

In this section we give a short description of all parallel algorithms to be tested. But before we can discuss the methods, we have to distinct the *types of parallelism*.

3.2.1 Types of Parallelism

In general, parallel query processing can be realized in two ways: (1) *inter-query parallelism*, which interleaves the execution of multiple queries while isolating them semantically, and (2) *intra-query parallelism*, which serializes the answering of the query sequence while each individual query is evaluated in parallel. In the following, let us look at the main representatives of these two classes of parallelism.

Table 3.2 gives an overview of the methods alongside with their initial sources and used abbreviations. Additionally, Table 3.1 recaps the single-threaded algorithms, that were evaluated in Chapter 2. To the best of our knowledge, these algorithms form the complete set of parallel cracking algorithms known to date. Let us now look at the different parallel algorithms in detail.

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

3.2.2 Parallel Standard Cracking (P-SC)

Interleave answering of multiple queries in isolation by serializing crack-in-two on the granularity of partitions.

In [14, 15] a multi-threaded version of standard cracking was shown, which we will denote by P-SC from now on. To describe this multi-threaded version it suffices to observe that, in standard cracking, as more queries arrive, they potentially partition independent parts of B, and thus, they can be performed in parallel.

When a query comes, it has to acquire two write locks on the border partitions, while *all* partitions in between are protected using read locks. When two or more queries have to partition or aggregate over the same part of B, read and write locks are used over the relevant parts. That is, whenever two or more queries $q_1, \ldots, q_r$, $r \geq 2$, want to partition the same part of B, a write lock is used to protect that part; say q_i, $1 \leq i \leq r$, obtains the lock and partitions while the other queries wait for it to finish. After q_i has finished, the next query q_j, $i \neq j$, acquiring the lock has to reevaluate what part it will exactly crack, as q_i has modified the part all queries $q_1, \ldots, q_r$ were originally interested in. Clearly, as more queries are performed, the number of partitions in B increases, and thus also the probability that more queries can be performed in parallel. This is where the speedup of this multi-threaded version over the single-threaded version stems from. If two or more queries want to aggregate over the same part of B, then they all can be performed in parallel, as they are not physically reorganizing any data. However, if one query wants to aggregate over a part of B that is currently being partitioned by another query, then the former has to wait until the latter finishes, as otherwise the result of the aggregation might be incorrect. Also, all queries work with the same cracker index, thus a write lock is acquired each time the cracker index is updated. Figure 3.1 visualizes the concept for two overlapping queries $Q1$ and $Q2$, that are executed by two threads $T1$ respectively $T2$.

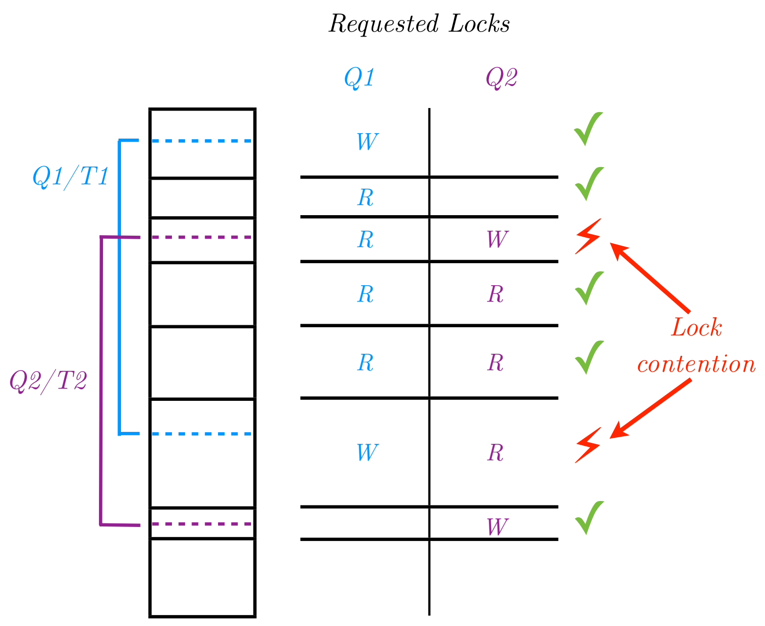


Figure 3.1: Visualization of the concept of parallel standard cracking (P-SC).

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

A to B. This maximizes the memory bandwidth utilization of the machine.

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

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

3.2.3 Parallel-chunked Standard Cracking (P-CSC)

Divide the cracker column non-semantically into independent chunks and apply standard cracking on each chunk in parallel.

After copying A onto B in parallel, as in P-SC, we (symbolically) divide B into k parts, each having n/k elements, and every thread will be responsible for exactly one of these parts. Now, *every* query will be executed by *every* thread on its corresponding part, and *every* thread will aggregate its results to a local variable assigned to it. At the end a single thread aggregates over all these local variables. Figure 3.2 visualizes the concept.

It is crucial for the performance of P-CSC to ensure complete independence between the individual parts. Any data that is unnecessarily shared among them can lead to *false sharing effects* (propagation of cache line update to a core although the update did not affect its part of the shared cache line) and *remote accesses* to memory attached to another socket. Thus, each part maintains its own structure of objects, containing its local data, cracker index, histograms, and result aggregation variables. Furthermore, by aligning all objects to cache lines, we ensure to avoid any shared resources, and each thread can process its part in complete independence from the remaining ones.

A similar concept as in P-CSC has been used in [39] in combination with vectorized cracking. Thus, we also test a vectorized version, denoted as P-CVC[1] from here on. Of course, the concept of work division can be applied to more advanced cracking algorithms as well.

[1]In contrast to [39], we do not merge the chunks after each query as this results in overhead.

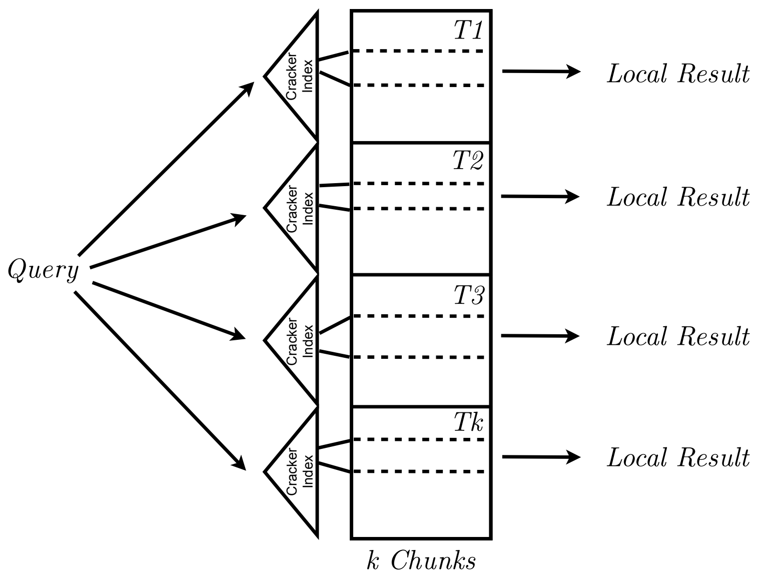


Figure 3.2: Visualization of the concept of parallel standard cracking (P-CSC).

3.2.4 Parallel Coarse-granular Index (P-CGI)

Apply a parallel range partitioning to bulk-load the cracker index before starting the query answering using P-SC.

In this chapter we present two parallel versions of the coarse-granular index discussed in Chapter 2. For the first one it suffices to observe that the coarse-granular index is nothing but a range partitioning as a pre-processing step to standard cracking. Thus, for the first parallel version of coarse-granular index, which will be denoted by P-CGI from now on, we show how to do a range partitioning in parallel. Afterwards we simply run the parallel standard cracking algorithm (P-SC) [14, 15] to answer the queries, taking into consideration that B is now range-partitioned. Our method to build a range partition in parallel requires no synchronization among threads, which of course helps to improve its performance.

The main idea behind the construction is very simple. Column A is (symbolically) divided into k parts, of $\frac{n}{k}$ elements each. Thread T_i, $1 \leq i \leq k$, gets assigned the i-th part of A and it writes its elements to their corresponding buckets[2] in the range partition on B, using $r \geq 2$ buckets. In order to do so, and not to incur any synchronization overhead, *every* bucket of the target range partition on B is (symbolically) divided into k parts as well, so that thread T_i writes its elements

[2]Our implementation of range partitioning is radix-based.

in the *i*-th part of *every* bucket. Thus, any two threads read their elements from independent parts of A and write also to independent parts on B. All this can be implemented in a way that all but one step are done in parallel[3], and every thread gets roughly the same amount of work. This, as we will see, helps to improve performance as the number of threads increases. Figure 3.3 visualizes the concept.

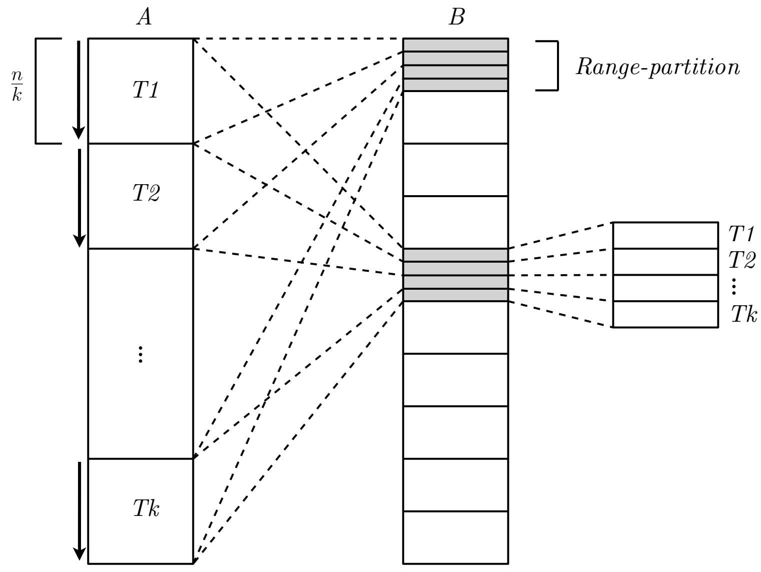


Figure 3.3: Visualization of the partitioning phase of parallel coarse-granular index (P-CGI).

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

3.2.5 Parallel-chunked coarse-granular index (P-CCGI)

Divide the cracker column non-semantically into multiple independent chunks and apply coarse-granular index on each chunk in parallel. Then, apply standard cracking locally for the query answering.

Symbolically divide A again into k parts, of $\frac{n}{k}$ elements each, and assign the i-th part to the i-th thread. Each thread t_i, $1 \leq i \leq k$, range partitions its part using single-threaded coarse-granular index, materializing it onto B. Thus, B is also (symbolically) divided into k parts. Having done this chunked range partition

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

of B, thread t_i keeps being responsible for the i-th range partition of B. When a query arrives, each thread executes standard cracking on its part and aggregates its result in a global variable, for which we again use a write lock to avoid any conflicts that might occur during aggregation. Again, as for P-CSC, we ensure that no resources are shared among the parts. All objects are cache-line-aligned and no trips to the remote memory are necessary at any place.

3.2.6 Parallel Full Indexing

So far we have only described algorithms for adaptive indexing. However, in order to see how effective those algorithms really are, we have to compare them against full indexes. In Chapter 2 it was observed that when the selectivity of range queries is not extremely high, as in our case, more sophisticated indexing data structures such as AVL-trees, B^+-trees, ART [31], among others, have no significant benefit over full sort + binary search + scan for answering queries, as the scan cost (aggregation) of the result dominates the overall query time. Therefore, in this chapter, we regard sorting algorithms as a direct equivalent of full indexing algorithms. With this in mind, and also due to (1) the good performance of building a range partition in parallel, and (2) the good performance of the radix sort algorithm (RS) of [33], the following hybrid sorting algorithm suggests itself:

3.2.7 Parallel Range-partitioned Radix Sort (P-RPRS)

Perform P-CGI to create range partitions in parallel. Then, apply radix sort RS on each partition in parallel.

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

As we will see, P-RPRS performs quite good in practice. However, this algorithm suffers from a large amount of negative NUMA effects as we will see. To alleviate this we present the following algorithm.

3.2.8 Parallel-chunked Radix Sort (P-CRS)

Divide the column non-semantically into multiple independent chunks and apply coarse-granular index on each chunk in parallel. Then, apply radix sort locally on all partitions of all chunks in parallel.

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

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

Table 3.2 in Section 3.1.1 constitutes a summary of the algorithms (experimentally) considered in this chapter. We have decided to leave the full scan as well as the hybrid cracking algorithms out of the presentation due to their high execution time, even in parallel.

3.3 Experimental Evaluation

Before we can start with the actual experimental evaluation, we have to discuss the used hardware setup as well as the experimental settings.

3.3.1 Hardware Setup

Our test system is a high-end server consisting of 4 sockets, each equipped with an Intel Xeon E7-4870 v2 processor with 15 physical and 30 logical cores, running at 2.3 GHz. Therefore, the machine has 60 physical and 120 logical cores available. The overclocking capabilities of the processors (Intel Turbo Mode) are disabled for all experiments, as they unnecessarily complicate the analysis. The private L1 and L2 caches of each core have a size of 32KB and 256KB respectively, while the shared L3 cache of each processor has a size of 30MB. Each processor has 3 QPI[4] links such that remote memory access is equally expensive for all neighboring processors. Each socket is attached to 128GB of 1600 MHz DDR3 RAM running in Intel Performance Mode, resulting in 512GB of available main memory. The operating system is a 64-bit Debian with kernel version 3.2. As the memory bandwidth plays an important role in the following discussion, we measured the throughput of the machine using the STREAM benchmark [35], that runs a set of simple read/write vector kernels. Instead of relying on the calculated bandwidth of the benchmark, we measured the throughput directly at the memory controller using Intel VTune Amplifier 2015. Figure 3.4 shows the aggregated bandwidth for all 4 sockets. As we can see, we reach 65 GB/s per socket and thus achieve a total machine bandwidth of 260 GB/s.

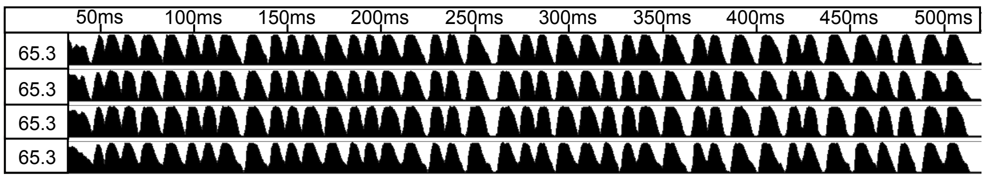


Figure 3.4: Stream Benchmark with 60 threads. We can reach 65 GB/s per socket for the aggregated read/write bandwidth per socket.

[4]Intel QuickPath Interconnect.

3.3.2 Experimental Setup

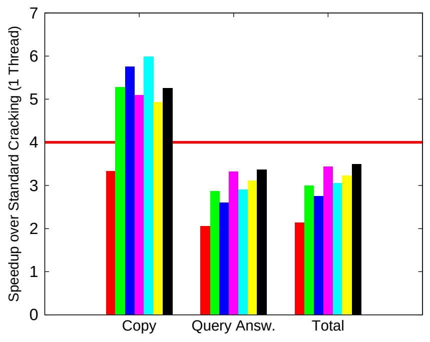
Let us now define the way in which the queries are fired and executed. As in previous experiments, we have a set of 1000 queries that should be answered as fast as possible. All queries are directly ready to be processed and there is no artificial idle time introduced between queries. Depending on the type of the algorithms, this query batch is processed differently. For algorithms that perform inter-query-parallelism, like P-SC for instance, we divide the set of queries into k parts, which are processed using k threads with each thread working $\frac{1000}{k}$ queries sequentially. This resembles the way of firing queries in [14]. In contrast, for algorithms that perform intra-query parallelism, like P-CSC, the 1000 queries will be answered sequentially one after another. However, each individual query is answered by k threads in parallel on a portion of the data. Please note that to get a more realistic setup, we introduced a barrier in the query execution loop: the answering of the next query starts only after all threads completed the current one.

3.3.3 Scaling of Parallel Cracking Algorithms

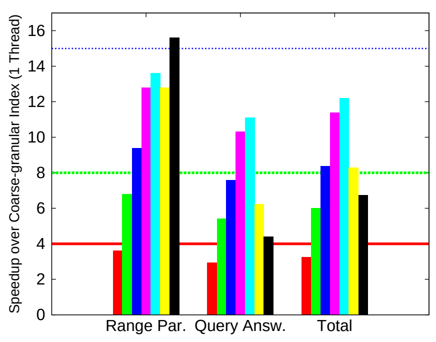
In Section 3.2, we described the set of algorithms for parallelizing database cracking. Besides the raw query processing times, these parallel methods offer another important dimension to analyze: the capabilities to scale with the multi-threading resources of the hardware. An algorithm, which scales poorly might be the winner in terms of runtime on a small machine, but completely looses the pace on a large server.

Therefore, in the following we will inspect for each method individually how it scales with an increase of the number of threads. We run each method using 4, 8, and 15 threads to utilize the computing cores up to $\frac{1}{4}$-th of the machine. Additionally, we test 30, 45, and 60 threads to investigate the scaling over the sockets. Finally, we run 120 threads to utilize all logical cores of the machine as well. We do not apply any form of thread pinning and let the operating system decide.

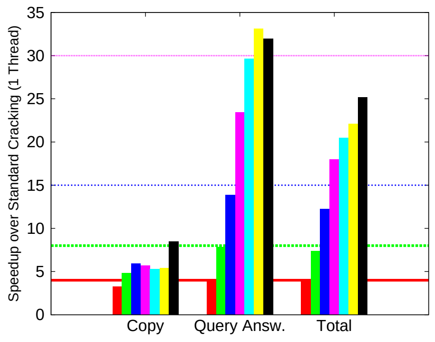
Figure 3.5 presents the accumulated speedups of the algorithms relative to their single-threaded counterparts. We inspect the individual parts (copying, range-partitioning, sorting, query answering) of the methods to analyze them separately as well as the total speedup. Let us go through the methods one by one and analyze their scalability.



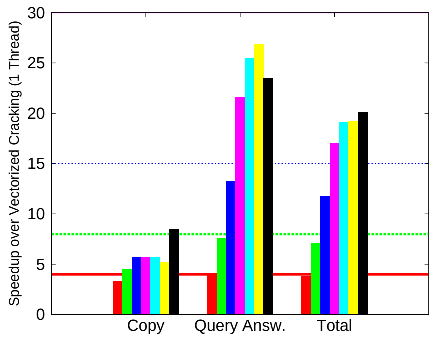
(a) Parallel Standard Cracking



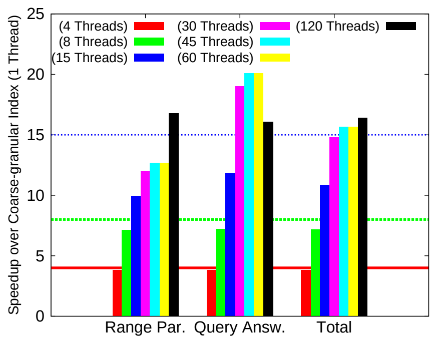
(b) Parallel Coarse-granular Index



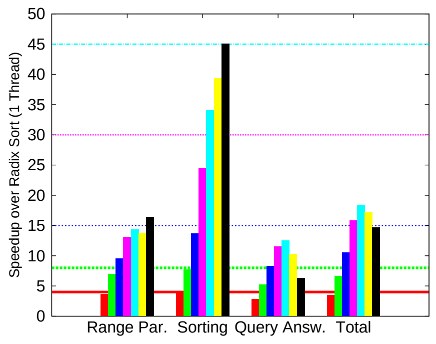
(c) Parallel-chunked Standard Cracking



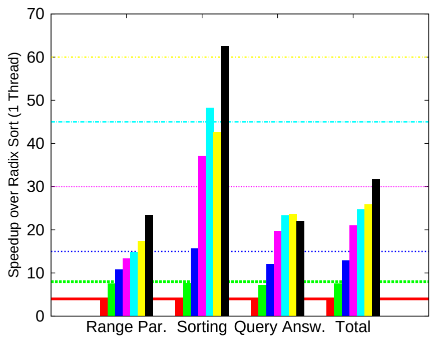
(d) Parallel-chunked Vectorized Cracking



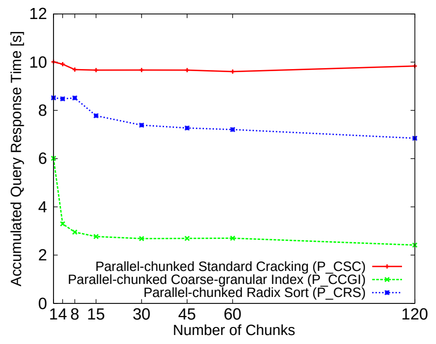
(e) Parallel-chunked Coarse-granular Index



(f) Parallel Range-partitioned Radix Sort



(g) Parallel-chunked Radix Sort



(h) Serial execution of chunked algorithms

Figure 3.5: Speedup of parallel cracking and sorting algorithms over their single-threaded counterparts while varying the number of threads. We show both the speedups of the characteristic phases as well as the overall achieved speedups. Colored horizontal lines show the expected perfect linear speedup. In Figure 3.5(h), we show for the chunked algorithms how the chunking itself influences the methods by serially working the chunks.

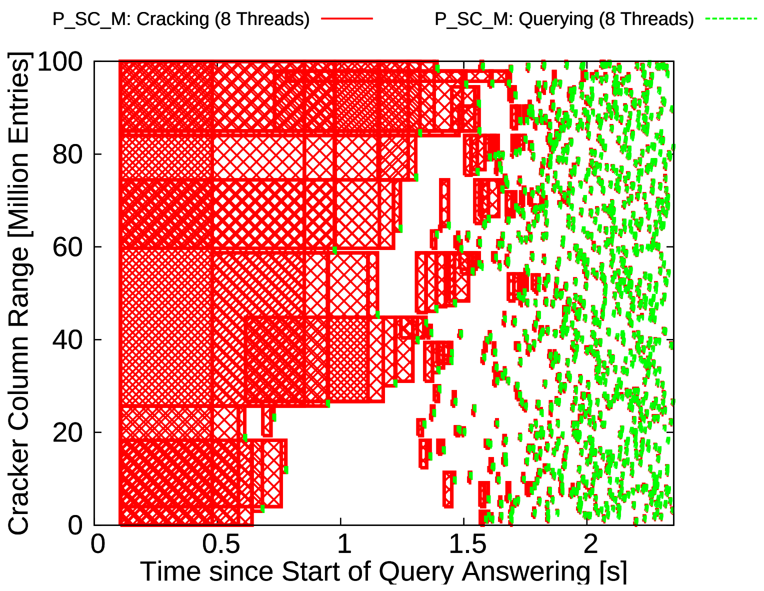
Parallel Standard Cracking (P-SC)

Figure 3.5(a) presents the scaling capabilities of the well known, lock-based P-SC. Unfortunately, it scales poorly with the increasing number of threads. The highest total speedup we observe is around 3.7x for 120 threads. The situation is particularly bad in the early phase of the query answering as the measured speedups are only between 1.5x and 2x. To understand this scaling problem, let us visualize the processing behavior of the algorithm. To do so, each time a thread is processing a partition, we log the time it takes as well as the processed area in the cracker column. This time includes possible waiting to acquire locks as well as the actual data processing. Figure 3.6 shows the plotted result. A rectangle $[x_1, y_2, x_2, y_2]$ means that within the time from x_1 to x_2, a thread was processing the cracker column at the range y_1 to y_2. The colors indicate the processing type, where red is modifying access (cracking) and green reading access (querying). Figure 3.6(a) presents the results for P-SC for 8 threads. We can observe a severe access contention in the first half of the run. The early queries lock huge parts of the cracker column as there exist only large partitions and thus serialize each other heavily. Therefore, the algorithm has no chance to scale linearly when starting from an unpartitioned state. Thus, let us see how the problem decreases when prepending a range-partitioning step in the next algorithm.

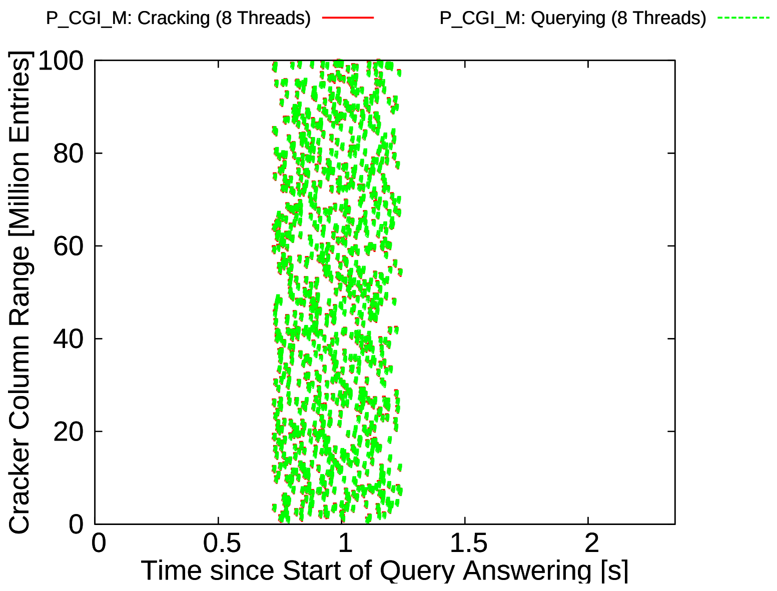
Parallel Coarse-granular Index (P-CGI)

As described before, this algorithm extends P-SC by applying an initial parallelized range-partitioning step, that creates 1024 partitions right away before any query answering starts. This should have a positive effect on P-SC and significantly reduce the contention that we have measured before. Figure 3.6(b) presents again the partition processing contention, this time for P-CGI. The blank space between time 0s and 0.7s is the range-partitioning phase. Afterwards, we see from 0.7s till 1.2s the actual query answering, which indeed parallelizes nicely now. No heavy contention is visible anymore and the algorithm behaves as intended, as the partitions are already small and the chance of two threads accessing the same partition is small. This is confirmed by the scaling factors of the P-CGI query answering phase in Figure 3.5(b), which now reaches a factor of 11x for 45 threads. For more threads, the performance significantly drops again, as access contention (both on the column as well as on the protected cracker index) throttles the throughput again.

Figure 3.7(a) presents a query-wise view on the answering phase. We can see that directly in the first query, the scaling is still very limited. This is caused by the



(a) Parallel Standard Cracking (P-SC)



(b) Parallel Coarse-granular Index (P-CGI)

Figure 3.6: Visualization of the partition processing contention for 8 threads. A rectangle $[x_1, y_1, x_2, y_2]$ means that within the time from x_1 to x_2, a thread was processing the cracker column at the range y_1 to y_2. Processing also includes wait times to acquire a lock. A red square indicates a writing process (cracking a partition) while a green square visualizes a reading process (querying a partition). Overlapping squares indicate that multiple threads intent to work on the same area of the cracker column at the same time.

setup and assignment of the threads to the tasks, which is expensive in comparison to the short running queries. Additionally, NUMA remote accesses are throttling the query answering phase. As the parallel range-partitioning algorithms creates partitions that are scattered across regions, a thread that answers a query has consequently a large number of remote accesses. Table 3.3 shows that based on hardware counters 2 out of 3 accesses are remote for P-CGI.

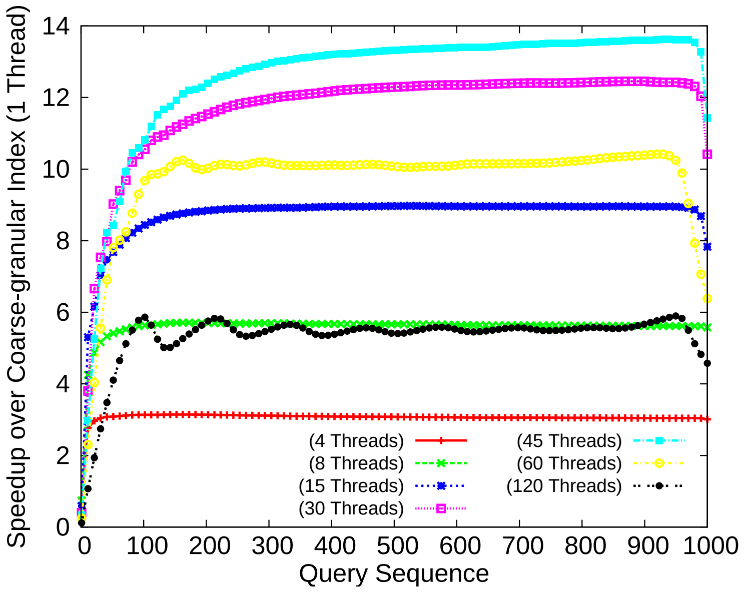
Let us now look at the range-partitioning itself. For 120 threads, we achieve the best speedup of factor 15x. Memory bandwidth is clearly not the problem, as it can be seen in the early phase of Figure 3.7(b), where only the histogram generation maximizes the bandwidth utilization. Our VTune analysis indicates that the range-partitioning algorithm is heavily back-end bound by the random-nature of the partitioning. Advanced partitioning techniques like software-managed buffers and non-temporal streaming stores might improve upon this problem, as we will investigate in the next chapter on partitioning.

Method	**Local Accesses** [Mio]	**Remote Accesses** [Mio]
P-SC	107	418
P-CGI	99	202
P-CSC	442	0
P-CVC	357	0.2
P-CCGI	44	0.2
P-RPRS	115	230
P-CRS	365	0.6

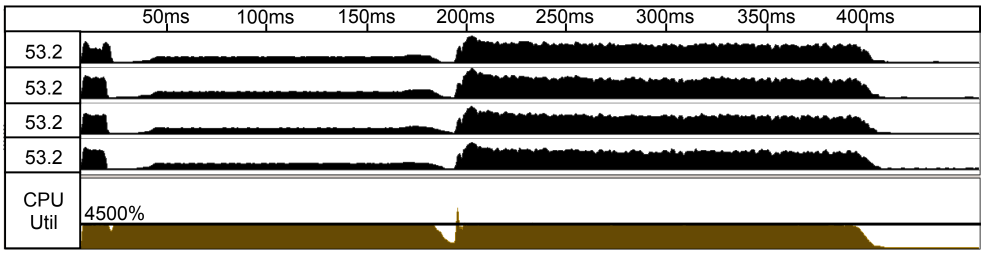
Table 3.3: Number of LLC cache misses that are served with local respectively remote DRAM access presented in millions of measured events. The measured counters are `OFFCORE_RESPONSE.DEMAND_DATA_RD.LLC_MISS.LOCAL_DRAM` and `OFFCORE_RESPONSE.DEMAND_DATA_RD.LLC_MISS.REMOTE_DRAM`.

Parallel-chunked Standard Cracking (P-CSC) / Parallel-chunked Vectorized Cracking (P-CVC)

After looking at the inter-parallel version of standard cracking, let us now inspect the scaling behavior of the intra-parallel version named P-CSC. The results are shown in Figure 3.5(c). We can see that this algorithms scales considerably better than the previous ones, which is what we expect from a method that parallelizes by chunking. However, we can also observe that the scaling is not linear with the number of threads. The highest total speedup that we achieve for 120 threads is only around 25x. To understand this behavior, let us inspect the individual parts.



(a) Scaling of Parallel-chunked Standard Cracking (P-CGI) of the query answering phase without the range-partitioning phase.



(b) Parallel-chunked Standard Cracking (P-CGI) with 45 threads. Highest bandwidth observed: 53.25GB/s

Figure 3.7: Bandwidth of P-CGI measured at 4 sockets in GB/s. The bottom line shows the CPU utilization in percentage.

Interestingly, the copying phase, which simply duplicates the input column into a separate array, scales particularly bad with a maximum speedup of 8x. As we can see from the bandwidth plot of Figure 3.8(a) for 60 threads, the memory bus is not the limiting factor, which is poorly utilized within the first 150ms. We identified page faults, which are surprisingly expensive to resolve when touching the cracker column for the first time during the copying phase as the cause of this behavior.

Let us now see how the query answering part alone scales in P-CSC. In Figure 3.5(c), we see a maximal speedup of the query answering phase of 33x for 60 threads, which is still not linear. NUMA effects are not a problem here as we can see in Table 3.3, all accesses are local. Apparently, the scaling is limited from

45 threads on, so let us inspect the utilized bandwidth of the query answering phase for 30 threads (Figure 3.8(b)), 45 threads (Figure 3.8(c)), and 60 threads (Figure 3.8(d)). We can see that in the early phase the bandwidth for 30 threads is with almost 59 GB/s already close to the cap of 65 GB/s, so we can not expect a linear scaling when increasing the number of threads by a factor of 1.5x (45 threads) respectively 2x (60 threads).

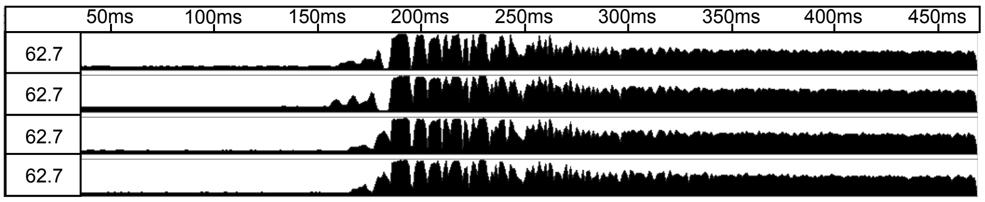
From Figure 3.5(d), we can see that **Parallel-chunked vectorized cracking (P-CVC)** shows a very similar scaling behavior as P-CSC. It scales slightly worse than P-CSC due to its nature of being even more bandwidth bound.

Parallel-chunked Coarse-granular Index (P-CCGI)

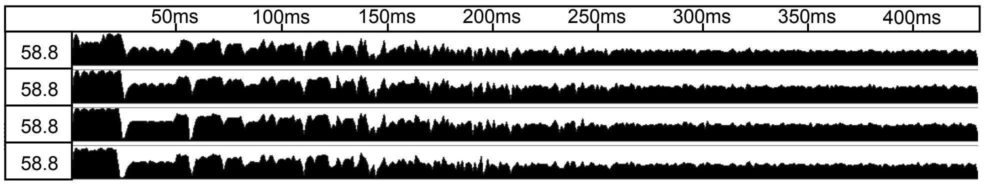
Let us now inspect the intra-parallel version of coarse-granular index in Figure 3.5(e). Interestingly, the range-partitioning phase scales almost exactly the same as the one of P-CGI in Figure 3.5(b), although the former uses a parallel range-partitioning while the latter one chunks a single-threaded implementation. This shows again, that the partitioning is heavily back-end bound and that stalls throttle the algorithm. The query answering phase scales with 20x for 60 threads much better than that of P-CGI. One reason is that each chunk can be processed individually without any concurrency control except the barrier at the end of each query. Another reason is the almost perfect NUMA locality, that we can observe in Table 3.3.

Parallel Range-partitioned Radix Sort (P-RPRS)

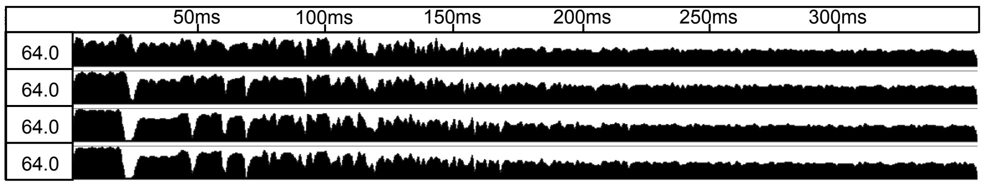
Finally, we want to analyze the scaling capabilities of the sorting baselines. Let us start with P-RPRS presented in Figure 3.5(f). The initial range-partitioning phase resembles the one of P-CGI which is why we see exactly the same scaling behavior. Afterwards, each created partition is sorted individually in parallel. Obviously, this phase scales much better with 45x for 120 threads at best. The reason lies in the great cache-locality created by the previous range-partitioning. By dividing the dataset into 1024 pieces, each partition has a size of 1.49MB. Since each processor has a L3 cache size of 30MB and 15 physical cores, each core has basically 2MB of cache available (1MB per logical core). This is obviously enough to keep all currently worked partitions completely inside the caches in the case of 60 threads. The scaling of the query answering phase is at best only 13x for 45 threads. This is again due to the high number of remote accesses in Table 3.3 caused by the initial parallel range-partitioning. They also have a negative impact on the sorting phase, although the worked partition is loaded once into the cache and then worked locally.



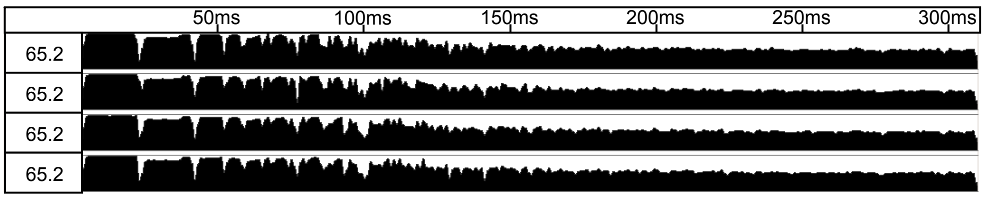
(a) Parallel-chunked Standard Cracking (P-CSC) with 60 threads. The initialization phase (copying the data into the cracker column in parallel) utilizes the bandwidth only partially (around 8GB/s).



(b) Parallel-chunked Standard Cracking (P-CSC) with 30 threads without initialization phase. Highest bandwidth observed: 58.77GB/s



(c) Parallel-chunked Standard Cracking (P-CSC) with 45 threads without initialization phase. Highest bandwidth observed: 64.02GB/s



(d) Parallel-chunked Standard Cracking (P-CSC) with 60 threads without initialization phase. Highest bandwidth observed: 65.24GB/s

Figure 3.8: Bandwidth measured at 4 sockets with Intel VTune Amplifier 2015 in GB/s.

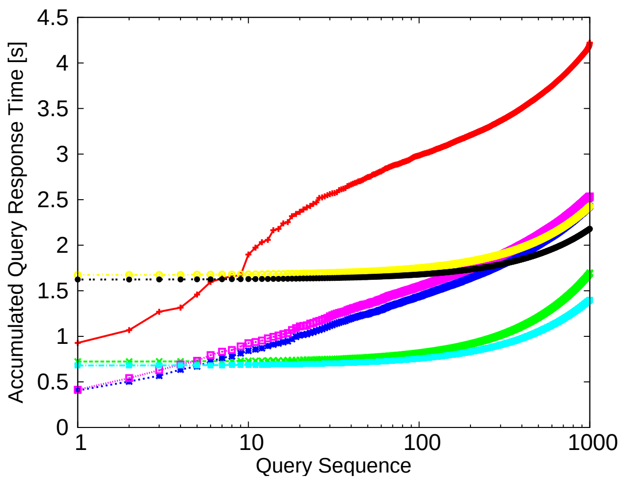
Parallel-chunked Radix Sort (P-CRS)

The second sorting algorithm, which does not create a global sorting, range-partitions and sorts all chunks locally in parallel. In the scaling result of Figure 3.5(g), we can see that the sorting phase scales even better than in P-RPRS. One reason lies in the NUMA local accesses, as we can see in Table 3.3. Another reason is presented in Figure 3.5(h). As sorting multiple smaller chunks is by default cheaper than sorting a large one, a part of the speedup also originates from that. This also causes the super-linear sorting speedup for 30 and 45 threads. The query answering phase of P-CRS also scales better than the one of P-RPRS due to the NUMA local processing nature.

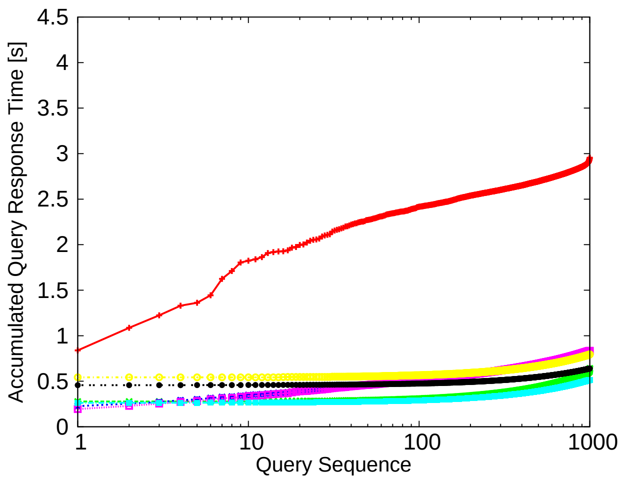
3.3.4 Runtime of Parallel Cracking Algorithms

After investigating the scaling capabilities of the algorithms on an individual basis, let us see how they compete against each other. To do so, we measure and compare the accumulated query response time over 1000 queries and present the results using different thread configurations in Figure 3.9. For 4 threads in Figure 3.9(a), there is a clear difference in runtime between the individual algorithms visible. Obviously, P-CSC has the lowest initialization time with almost 0.5s, while the sorting methods need with around 1.7s considerably more time for their first query. Over 1000 queries, the cracking methods P-CCGI and P-CGI clearly win in terms of accumulated runtime, while P-SC is far behind the remaining methods due to its serialization behavior in the early querying phase. When increasing the number of threads to 15 in Figure 3.9(b) and to 60 in Figure 3.9(c), we see a clear trend: the different between the sorting and cracking methods significantly decreases. For 60 threads, the time of the first query for P-CSC is with 191ms only 46ms shorter than that of P-CRS, which fully sorts the chunks and answers the first query in 237ms, caused by the superior scaling of the sort-based algorithms.

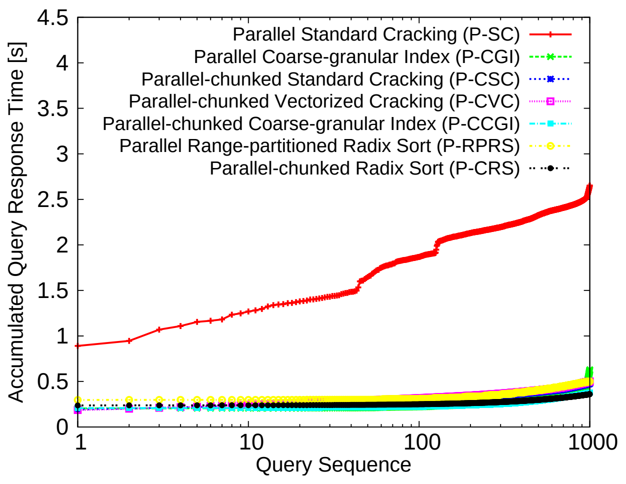
This analysis indicates that for a large number of threads, the sorting algorithms are a clear alternative over the adaptive methods, especially since they are easier to integrate into the system stack and offer interesting orders. Nevertheless, we believe that in a real system with many queries processing several columns at the same time, only a portion of the physical resources are available to initialize a column. Under such circumstances, cracking remains its advantage of offering the significantly cheapest option of enabling indexing.



(a) 4 Threads



(b) 15 Threads



(c) 60 Threads

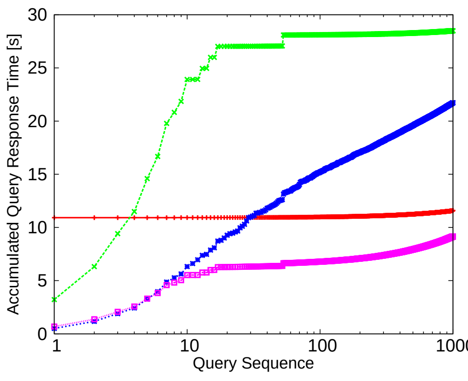
Figure 3.9: Accumulated query response time of parallel cracking algorithms in comparison with parallel radix-based sorting methods.

3.3.5 Tuple Reconstruction in the Context of Parallelism

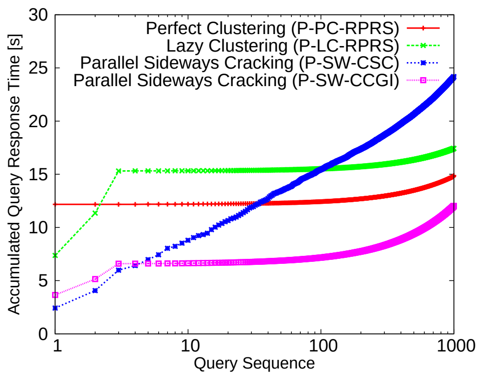
So far, we have looked at the parallel indexing methods without considering tuple reconstruction in order to focus solely on the cracking and sorting algorithms. Now, let us see how the tuple reconstruction concepts we have seen already in the single-threaded case, like sideways cracking [22], can be applied on top of multi-threaded algorithms. Precisely, we will investigate how sideways cracking can be combined with parallel cracking algorithms. To the best of our knowledge, this is the first work to approach this question. Then, we will compare the tuple reconstruction performance of the parallel cracking algorithms with a clustered table that has been ordered with respect to the sorted index column using our parallel range-partitioned radix sort.

As the basis for parallel sideways cracking, we pick the two cracking algorithms that performed the best in the previous evaluation — P-CSC and P-CCGI. This allows us to apply the concept of chunking to sideways cracking as well. For each chunk, we keep separate cracker maps and a separate tape and thus, the chunks can be worked independently by the individual threads. We name these two methods P-SW-CSC respectively P-SW-CCGI in the following. The baseline for parallel sideways cracking is formed by a clustered table, that can be created in two ways. The first version, coined P-PC-RPRS, clusters the entire table (stored in column layout) directly in the first query with respect to the selection column. The sorting is performed using our parallel range-partitioned radix sort. The second version, called P-LC-RPRS, establishes the clustering in a lazy manner, by copying and clustering only the columns that are actually touched by a query. In this case, the clustering of a column is created by applying a fresh sort on the selection column.

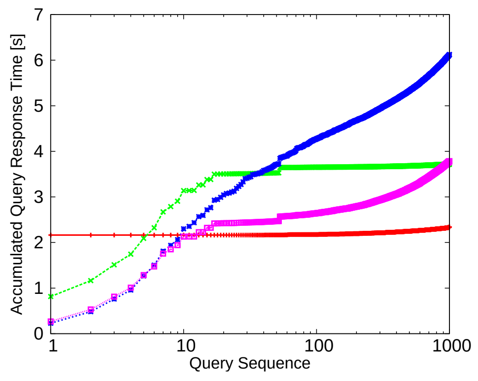
To put these methods to the test, we apply different workloads. All share the property that the selection is performed on a single, fixed attribute of a table composed of 10 columns following a uniform random distribution. We perform separate runs projecting 1 and 5 attributes respectively, that are randomly selected for each query. Figure 3.10 shows the accumulated query response times for 4 and 60 threads. Let us focus on the 4 threaded case first in Figures 3.10(a) and 3.10(b). For all numbers of projected attributes, P-PC-RPRS behaves in the most predictable way. Clustering the entire table of 10 columns takes around 11 seconds and the following query answering takes only a small amount of additional time, even if 5 attributes are projected. P-LC-RPRS, which clusters a column when it is touched for the first time is heavily affected by the number of projected attributes. Interestingly, the larger the number of projected attributes, the smaller is the accumulated query response time. This makes sense as a query projecting multiple attributes can cluster multiple columns in a single sorting run. We can also observe, that the lazy clustering pays off only for the first few queries, at least for a table consisting of only 10 columns.



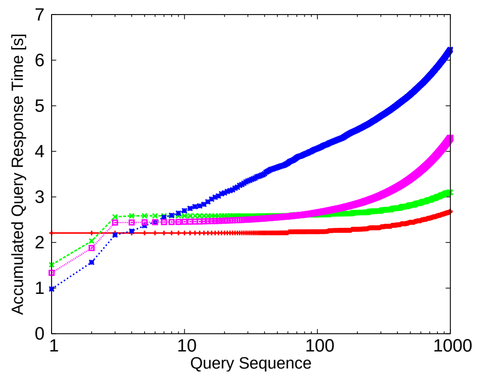
(a) 4 Threads: one σ, one π



(b) 4 Threads: one σ, five π



(c) 60 Threads: one σ, one π



(d) 60 Threads: one σ, five π

Figure 3.10: Accumulated tuple reconstruction cost for 1000 queries and a table consisting of 10 columns, shown for 4 and 60 threads. We select on a single fixed attribute. In Figures 3.10(a) and 3.10(c), each query projects a single randomly chosen attribute. In Figures 3.10(b) and 3.10(d), each query projects five randomly selected attributes.

In comparison to that, parallel sideways cracking offers in both implementations a significantly smaller initialization time. The first query of P-SW-CCGI is slightly more expensive than that of P-SW-CSC, as it range partitions the dataset during the initialization of a cracker map. In the long run, it always clearly pays off to prepend a range-partitioning step. Overall, for 4 threads and 1000 queries, P-SW-CCGI shows the best accumulated runtime in all tested cases. This picture changes if we switch to 60 threads in Figures 3.10(c) and 3.10(d). Obviously, all methods benefit from the increased number of threads, however, the sort based methods win at a higher degree.

Obviously, the better scaling capabilities of the sort based methods that we saw in the previous analysis pay off in the tuple reconstruction case as well. P-PC-RPRS needs less than 10 queries to beat both Parallel Sideway Cracking implementations. The difference between the lazy P-LC-RPRS and P-PC-RPRS has also significantly decreased and even P-LC-RPRS outperforms P-SW-CSC around 40 queries for 1 projected attribute and 8 queries for 5 projections.

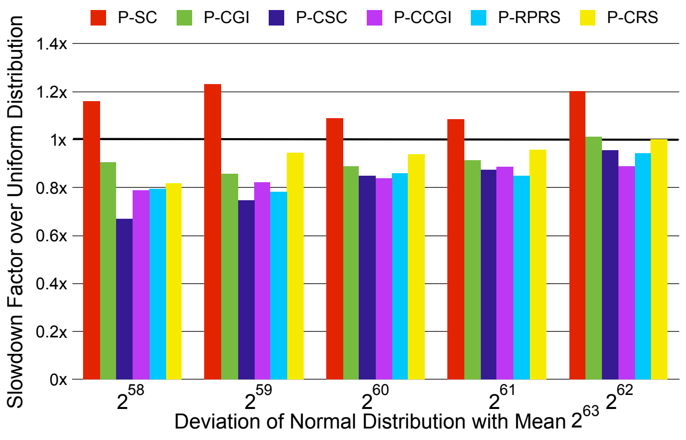
Overall, we see the same trend as before: the more threads available, the more the advantage shifts to the sorting side. Still, if only few threads are available for the initialization step, parallel sideways cracking shows a significantly smaller preparation time. Further, for tables consisting of multiple hundreds of attributes, only on-demand initialization of columns is a viable option, as offered by parallel sideways cracking.

3.3.6 Skew in the Context of Parallelism

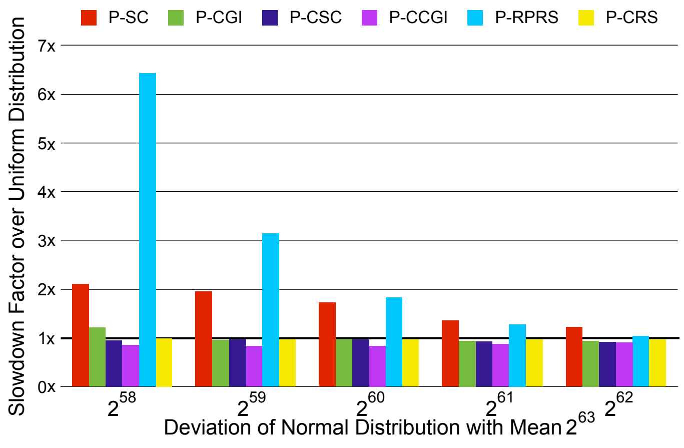
Up to this point, we evaluated the parallel methods under a uniformly distributed random workload on top of uniformly distributed data. In the following, we will investigate how different kinds of skewness affect the parallelism. We will test both skewed query predicates as well as skewed input data. Furthermore, we cluster the input data into range-partitions and inspect the impact on the methods. Precisely, we run the following configurations independently:

1. The query predicates follow a normal distribution with mean $\mu = 2^{63}$ (middle of the domain). The deviation is varied from $\sigma = 2^{58}$ (high skew) to $\sigma = 2^{62}$ (low skew). This pattern simulates a high interest in certain keys.
2. The keys of the input data follow a normal distribution with mean $\mu = 2^{63}$ (middle of the domain). The deviation is varied from $\sigma = 2^{58}$ (high skew) to $\sigma = 2^{62}$ (low skew). This pattern simulates a higher appearance frequency of certain keys.
3. The keys of the input data follow a uniform distribution. However, the input is physically clustered into k uniform range-partitions. We test a low clustering using $k = 4$ and a high one using $k = 60$. This pattern simulates data where the key locality resembles physical locality, typically the case for sensor or financial data.

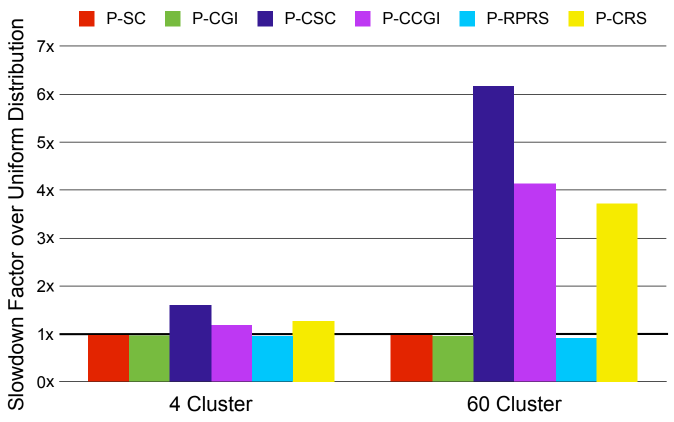
Figure 3.11 shows the results in form of speedup factors, that different methods achieve when switching from the uniformly random distributed data and queries seen so far to the respective form of skewness. A factor below 1 indicates a speedup. We compare the runtimes of the entire query sequence of 1000 queries.



(a) Skewed Queries



(b) Skewed Input



(c) Clustered Input

Figure 3.11: Impact of skewness variants on the methods for 60 threads. The shown numbers present the speedup over the uniform random dataset using uniformly distributed query predicates. A number smaller than 1 represents a speedup of the version under skew.

Figure 3.11(a) shows the influence of skewed query predicates on the methods. We can observe that only P-SC is affected negatively by the skew with a slowdown of up to 1.2x, interestingly even the low skew triggers it. All remaining methods improve with a higher selectivity by factors between 0.67x (P-CSC) and 0.91x (P-CGI) for a deviation of 2^{58}. P-SC suffers from the focus on a certain data region due to a higher lock contention, P-CGI can outweigh this problem with the initial range-partitioning. The remaining algorithms exploit the denser access locality that result in more fine granular cracks and a better cache utilization. The parallelism of the chunked algorithms is not affected at all by the skewed predicates as the work balance remains the same.

Figure 3.11(b) presents the impact of skewed input data. As we can see, this has a more severe influence on some of the methods. P-SC and especially P-RPRS are heavily slowed down by a factor of 2.10x and 6.42x respectively for the highest skewness. P-SC suffers from the fact that queries falling into the skewed region work on a larger part of the column and thus limit the amount of possible parallelism. P-RPRS has the problem that the range-partitioning phase creates partitions of unbalanced size and thus, the following sorting work is unequally divided among the threads. The creation of equi-depth partitions could help here, however, we leave this to future work. Again, the chunked methods are completely unaffected by this type of skew.

However, the picture changes when data clustering is introduced in Figure 3.11(c). We test a lower clustering of 4 partitions and a heavy clustering of 60 partitions. Under these circumstances, the chunked algorithms experience a severe slowdown. The pre-clustered input leads to an unbalanced work division, as only some of the chunks contain data that is relevant for the query. P-CSC suffers the most, as its entire behavior is query driven and thus influenced by the clustering. For P-CCGI and P-CRS, at least the range-partitioning and sorting is query independent and thus balances well. To resolve this problem, a cluster-aware chunk division would be necessary, e.g. as proposed in [9]. However, this is left for future work.

Overall, we learned that the chunked methods are completely resilient to both skewed queries and input. However, in their current state, they have severe problems in handling clustered input. P-RPRS suffers from skewed input as the range-partitioning phase creates equi-width partitions that do not balance the sorting work. P-SC reacts negatively to both skewed input and queries due to the higher contention.

3.4 Conclusion

Database cracking needs to improve mapping to parallel hardware. In this chapter, we inspected several different parallel cracking algorithms that use either inter- or intra-query parallelism and compared them in terms of scaling with available hardware resources and absolute runtimes with sort-based approaches. We identified lock contention and the shared memory bus as main limitations for parallel cracking algorithms.

In terms of absolute query response times, the sorting methods are a hard match for their cracking based competitors and offer nice additional properties like interesting orders — however, only if a large number of threads is available. This picture is confirmed in the tuple reconstruction case, where parallel sideways cracking is the winner over parallel clustering only under limited computing resources. Skew affects the parallel algorithms at different degrees depending on its type: a higher skewness is preferred by most algorithms although e.g. clustered input heavily throttles certain methods in their current realizations.

Overall, in the past two chapters, we have discussed both single- and multi-threaded adaptive indexing and sorting algorithms to the utmost. We have seen various algorithmically different techniques, that optimize towards convergence speed, low variance, high tuple-reconstruction performance, and linear scalability with computing cores.

Despite their differences, they all share a single common characteristic — they *partition* the data. This is why in the upcoming chapter, we want to focus solely on the procedure of partitioning. The question is: what gains are possible for such a simple and reduced technique on modern hardware and operating systems?

Chapter 4

Understanding and Optimizing Data Partitioning

Partitioning a dataset into ranges is a task that is common in various applications such as the previously discussed adaptive indexing and sorting [40, 52, 45] as well as hashing [5] which are in turn building blocks for almost any type of query processing. Especially radix-based partitioning is very popular due to its simplicity and high performance over comparison-based versions [40].

While the two-sided in-place partitioning algorithm that was used by standard cracking does not offer much room for improvement, the out-of-place radix-based partitioning algorithm that we used to create our coarse-granular index has much potential as we will see.

Thus, in the following chapter, we will analyze the nature of partitioning and discuss several optimization techniques, that can be applied to enhance the core algorithm. In total, we manage to improve the runtime by a factor of 2.5x over the original method. Let us see step by step how such an improvement can be achieved.

4.1 Introduction

Algorithm 2: Original version of radix partitioning.

```
 1 // build histogram
 2 for(i = 0; i < num_elems; ++i) {
 3   ++histogram[input[i] >> (32 - R)];
 4 }
 5 // build partition index
 6 offset = 0;
 7 for(i = 0; i < num_partitions; ++i) {
 8   dest[i] = offset;
 9   offset += histogram[i];
10 }
11 // partition the data
12 for(i = 0; i < num_elems; ++i) {
13   bucket_num = input[i] >> (32 - R);
14   output[dest[bucket_num]] = input[i];
15   ++dest[bucket_num];
16 }
```

In its most primitive form, coined *original version* from here on, radix partitioning partitions a dataset into 2^R (where $R \leq 32$) partitions as shown in Algorithm 2: in the first pass over the data, we count for each partition the number of entries that will be sent to it (lines 2-4). From this generated histogram, we calculate the start index of each partition (lines 6-10). The second pass over the data finally copies the entries to their designated partitions (lines 12-16).

Despite of its simple nature, several interesting techniques can be applied to enhance this algorithm such as *software-managed buffers* [40, 45, 5, 52], *non-temporal streaming operations* [11, 52, 40, 5], *prefetching*, and different *memory layouts* [40, 5] with many variables having an influence on the performance like *buffer sizes*, *number of partitions*, and *page sizes*.

Although being heavily used in the database literature, it is unclear how these techniques individually contribute to the performance of partitioning. Therefore, in this work we will incrementally extend the original version by the mentioned optimizations to carefully analyze the individual impact on the partitioning process. As a result this chapter provides a strong guideline on when to use which optimization for partitioning.

4.1.1 Experimental Setup

The dataset used in the evaluation of this chapter consists of $N = 100$ million entries, where each entry is composed of a 4B key (unsigned int) and a 4B payload[1], which is typically a rowID. The keys follow a uniform and random distribution and cover the full unsigned 4B space. We repeat each experiment five times and report the average runtime. All algorithms evaluated are purely single-threaded.

We run all the experiments on a two-socket machine consisting of two quad-core Intel Xeon E5-2407 running at 2.2 GHz. The CPU neither supports hyper-threading nor turbo mode. The sizes of the L1, L2, and L3 caches are 32KB, 256KB, and 10MB respectively. The TLB can cache 64 (L1 dTLB) and 512 (L2 TLB) address translations for 4KB pages and 32 (L1 dTLB) translations for 2MB pages. The system is equipped with 48GB of main memory in total and runs a 64-bit openSuse 12. All programs are written in C++ and compiled using g++ 4.7.1 with optimization level O3.

4.2 Experimental Evaluation

In the following evaluation, we incrementally extend and modify the *original version* of radix partitioning by applying both known techniques from the literature as well as new approaches to analyze the impact of the optimizations on the total runtime of partitioning. Figure 4.1 shows the paths we follow when incrementally optimizing the original version alongside with their appearances in the literature.

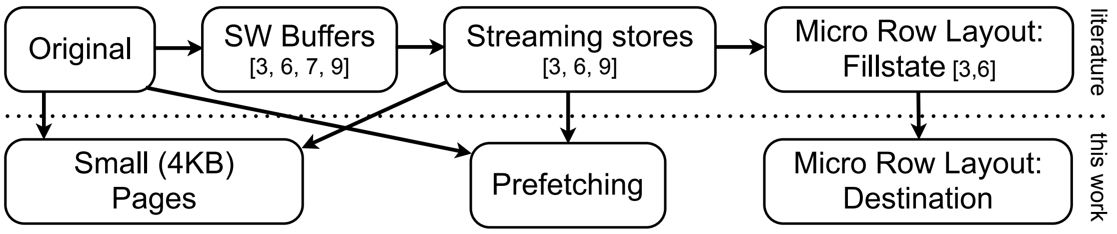

Original
SW Buffers [3, 6, 7, 9]
Streaming stores [3, 6, 9]
Micro Row Layout: Fillstate [3,6]
literature
Small (4KB) Pages
Prefetching
Micro Row Layout: Destination
this work

Figure 4.1: Overview of the applied optimizations.

[1]We choose these sizes as typical numbers for main-memory indexes. In Section 4.2.6 we vary the number of entries and entry size.

4.2.1 Software-Managed Buffers

A well known optimization for accessing multiple data streams are software-managed buffers, that are used in a variety of works [40, 45, 5, 52]. As writing to p different streams means accessing p pages (if stream offset is larger than the page size), we have to cache p address translations in the TLB to avoid page walks. Unfortunately, as the TLB capacity is scarce, for a larger value of p, TLB-misses occur frequently [8]. Software-managed buffers try to reduce this problem by keeping a buffer of b entry slots for each partition, such that these buffers are filled first. Only if a buffer is full, its b entries are flushed to the final partition. Thus, the output array is now accessed at buffer granularity and no longer at entry granularity, reducing the amount of required address translations by a factor of b. Figure 4.2 visualizes the concept.

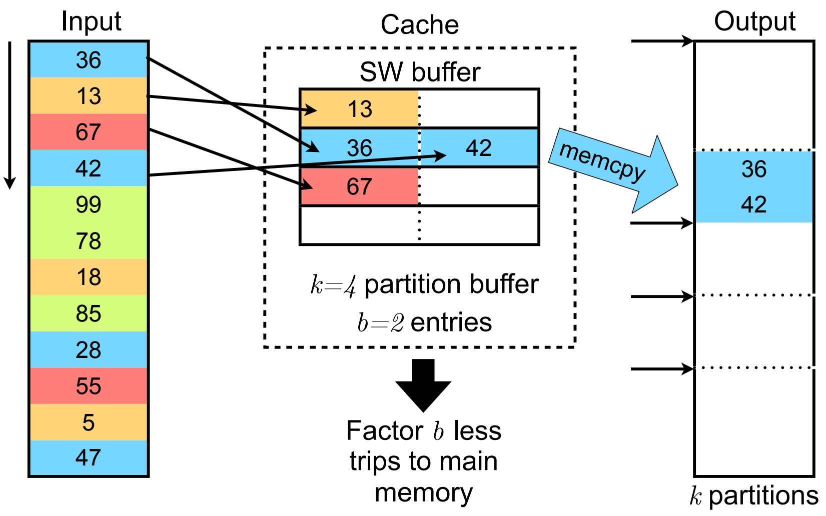


Figure 4.2: Visualization of radix partitioning using software-managed buffers.

In recent work [40, 5] a partition buffer is set to the size of a single cache-line to minimize cache-misses when filling the buffers. However, by choosing larger partition buffers, we might decrease the number of TLB-lookups even more (at the risk of more data cache misses as buffers are more likely to get evicted). Thus, we will test a variety of partition buffer sizes to analyze this tradeoff.

Let us start by looking at the partitioning time using software-managed buffers while varying the size of a partition buffer from one cache-line (64B, capacity for 8 entries) to 1248 cache-lines (78KB, capacity for 9984 entries) and the number of partitions[2] from 2^5 to 2^{14} in Figure 4.3. As expected, the runtime increases

[2]We focus on typical numbers of partitions that are used in practice like 256 partitions (byte-

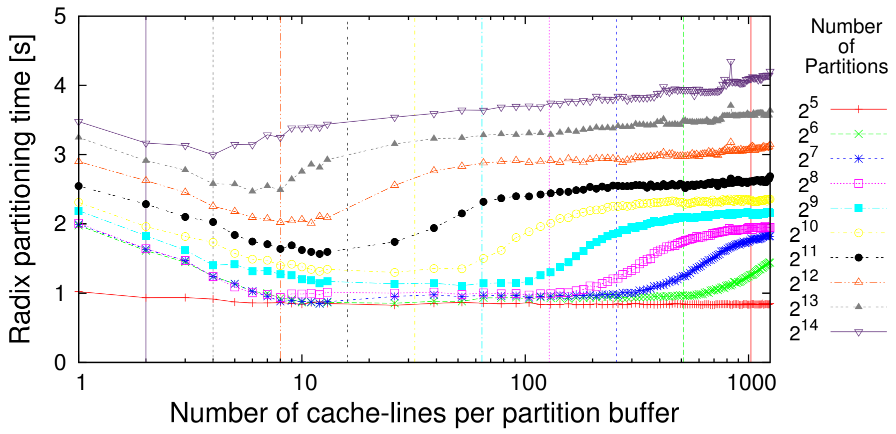


Figure 4.3: Partitioning time of the software-managed buffers version for varying partition buffer sizes.

with the requested number of partitions. However, we can also observe that for all tested numbers of partitions, the runtime improves with the buffer size until a certain optimal point is reached and then slowly degrades again. The more partitions we request, the smaller is this optimal partition buffer size. The point at which the performance degrades also heavily depends on the number of partitions (2048 elements per buffer for 128 partitions and around 512 elements per buffer for 1024 partitions). Thus, the performance degradation is related to the total space that the buffers occupy with a degradation point around 2MB (visualized by the vertical lines).

To get further insights into the behavior of the method, we measured the total amount of cache-misses for the software-managed buffers version in Figure 4.4 using *perf*. Compared with the runtimes of Figure 4.3, we observe a clear correlation. The increase in runtime for buffers larger than 2MB is related to the increase in cache-misses. Furthermore, the fastest runtime is measured around the partition buffer size that triggers the smallest amount of misses. The question is now how much we gained over the original version by using software-managed buffers. Table 4.1 shows a direct comparison.

wise radix sort) and 1024 partitions (initial range-partitioning step in database cracking algorithms, see Chapter 2).

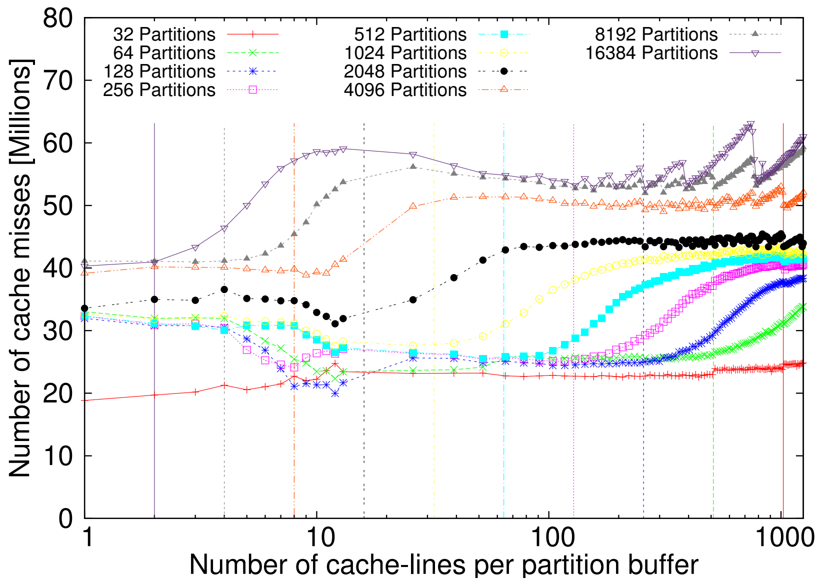


Figure 4.4: Total cache-misses (trips to main memory)

# Partitions	Original Version [s]	SW Buffers [s]	([cl])	Speedup
32	0.79	0.83	(26)	0.95x
64	1.91	0.85	(12)	2.25x
128	2.03	0.85	(12)	2.39x
256	2.12	0.94	(8)	2.25x
512	2.27	1.11	(52)	2.05x
1024	2.41	1.30	(26)	1.85x
2048	2.57	1.57	(12)	1.64x
4096	2.91	2.01	(11)	1.45x
8192	3.32	2.47	(6)	1.34x
16384	3.61	3.00	(4)	1.20x

Table 4.1: Partitioning time of the *original version* and the *software-managed buffers version* (as shown in Figure 4.3). For software-managed buffers, we show the runtime and the buffer size per partition in number of cache-lines (cl) in brackets.

In the case of software-managed buffers, we individually picked the best suitable partition buffer size (shown in the brackets) for the comparison. Obviously, the buffered version significantly improves the partitioning time for all numbers of partitions from 64 on, ranging from a speedup of 2.25x for 64 partitions to 1.20x for $16,384$ partitions. The improvement decreases with the number of partitions, as the buffers run out of the private caches. Only for 32 partitions, the original version performs significantly better, as it does not yet suffer from TLB- and cache-misses.

4.2.2 Non-temporal Streaming Stores

So far, we have flushed the buffers in the traditional way using `memcpy`. Internally, this triggers the fetching of the corresponding cache-lines from main memory to write the new data to them. While this is a valid strategy in general, where modified cache-lines are likely to be used subsequently, in the case of partitioning this behavior is wasteful. Instead, we want to directly write the data to main memory and bypass the caches entirely.

This can be achieved by using *non-temporal streaming stores*, that are used widely in write-intensive situations [11] and partitioning tasks [52, 40, 5]. Since we write entries of size 8B, we can use the following AVX intrinsic to write 4 buffered entries to the partition at once:

```
_mm256_stream_si256(__m256i* mem, __m256i a)
```

Furthermore, the processor tries to apply write-combining [11, 40, 52, 5] to fill a cache-line in its *write-combine buffer* before writing to main memory. As soon it is filled (after two subsequent calls to the stream intrinsic), it is flushed out without ever reading the corresponding cache-line from main memory. Figure 4.5 visualizes the concept.

However, streaming stores also introduce difficulties: the address we flush to must be a multiple of 32B. One solution to this problem is padding the partitions of the output array such that they are cache-line aligned, but this increases the size of the array unnecessarily. In contrast to that, our implementation simply offsets the start of the partitioning filling to align it and corrects the few wrongly written entries afterwards. As there is no runtime difference observable between the padded and unpadded version, we will not distinguish them in the further investigation.

We can see in comparison to Figure 4.3 that for a high number of partitions, using

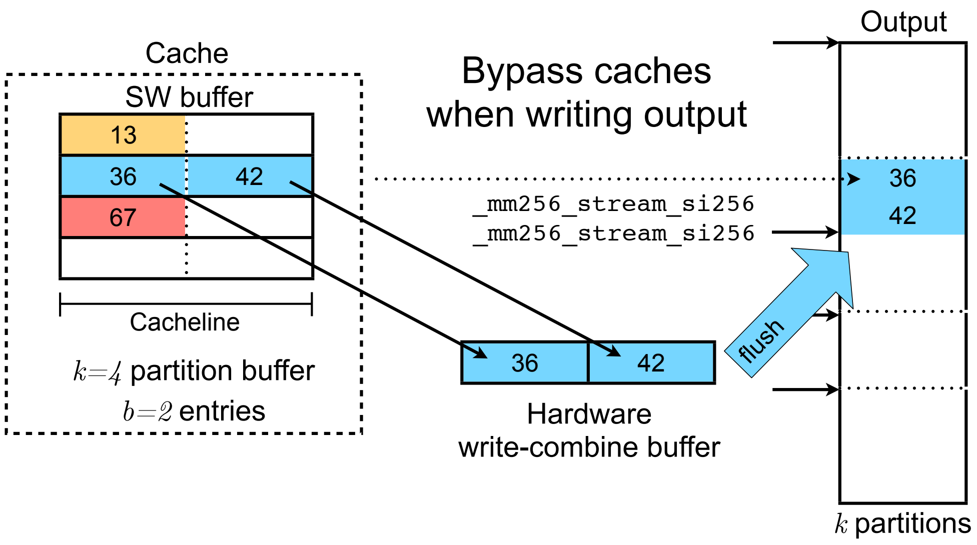


Figure 4.5: Visualization of radix partitioning using software-managed buffers and *non-temporal streaming stores.*

the non-temporal streaming store clearly improves the runtime of partitioning. The higher the total number of partitions, the more space the buffers occupy and the more precious cache memory becomes. In this situation, avoiding cache-pollution caused by bringing in output cache-lines is clearly beneficial. However, one should use small buffers (less than 8 cache-lines) to make this technique pay off or else the cache-misses triggered by buffer filling overshadow the streaming benefit.

In Table 4.2, we compare the streamed version with the plain software-managed buffers version of Figure 4.3 to see the impact of this technique. In contrast to the previous optimization (Table 4.1), the positive impact of streaming stores increases with the number of partitions, up to a speedup of 1.36x for 16384 partitions. The optimal partition buffer size decreases again with the number of partitions, with the best size of a single-cache line from 2048 partitions on. We can see that the streamed version prefers in all cases smaller buffers than the unstreamed version. This shows that the avoidance of cache-misses when flushing a buffer has a higher priority when using streaming stores. Nevertheless, we also see that streaming introduces overhead for a small number of partitions.

Furthermore, the optimal partition buffer size decreases with the number of partitions. From 2048 partitions on, a partition buffer size of a single cache-line shows the best results. Thus, reducing the pressure on the caches has a higher importance than reducing the TLB-misses by doing less buffer flushes. Nevertheless, we also see that streaming introduces overhead for a small number of partitions.

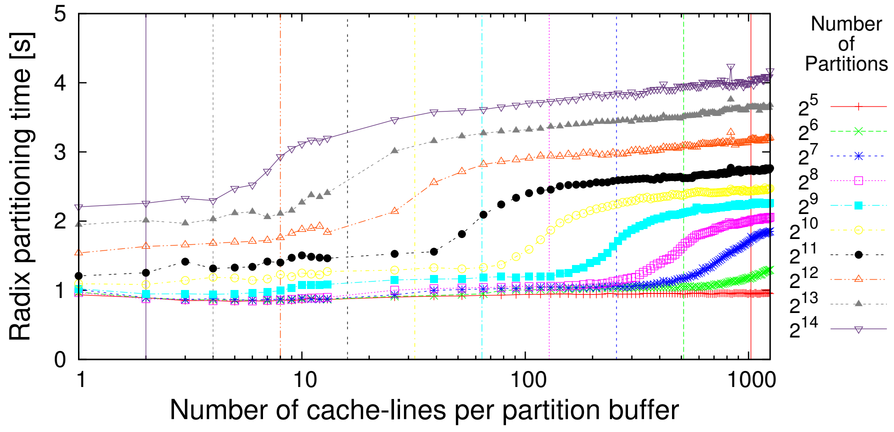


Figure 4.6: Partitioning time of the *buffered & streamed version* for varying number of partitions and partition buffer sizes.

# Partitions	SW Buffers [s]	([cl])	SW Buffers + Streaming [s]	([cl])	Speedup
32	0.83	(26)	0.84	(5)	0.98x
64	0.85	(12)	0.85	(7)	1x
128	0.85	(12)	0.86	(6)	0.99x
256	0.94	(8)	0.84	(6)	1.12x
512	1.11	(52)	0.94	(4)	1.18x
1024	1.30	(26)	1.08	(2)	1.20x
2048	1.57	(12)	1.21	(1)	1.30x
4096	2.01	(11)	1.54	(1)	1.31x
8192	2.47	(6)	1.95	(1)	1.27x
16384	3.00	(4)	2.21	(1)	1.36x

Table 4.2: Partitioning time of the *software-managed buffers version* (Figure 4.3) and the *software-managed buffers version using non-temporal streaming stores* (Figure 4.6). We show the best runtime and partition buffer size in number of cache-lines (cl).

4.2.3 Prefetching

Writing to individual partitions respectively buffers resembles random access that can cause cache-misses. Ideally, we want to hide these by introducing prefetching hints to ensure that requested data is cached in time. To do so, we use the following intrinsic:

```
__builtin_prefetch(void* mem, int rw, int l)
```

This triggers the generation of data prefetch instructions that will prefetch the memory at address `mem`, such that it is (hopefully) already available in cache at access time. We set `rw` to 1 since we write and `l` to 0, indicating that the temporal locality of the memory is low (high eviction chance after first access). Figure 4.7 visualizes the concept. To the best of our knowledge, we are the first group extending both the original version and the version using buffers and streaming stores in a way that when writing to an output partition respectively a buffer, the subsequent slot is prefetched.

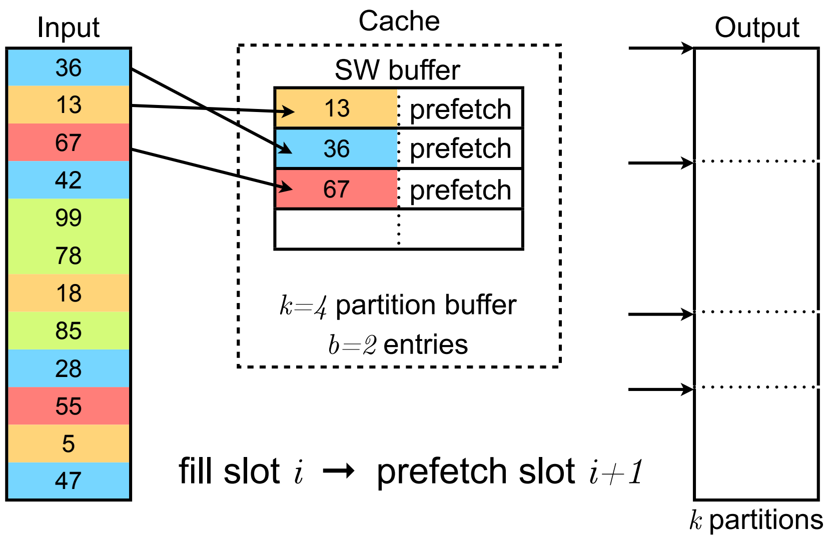
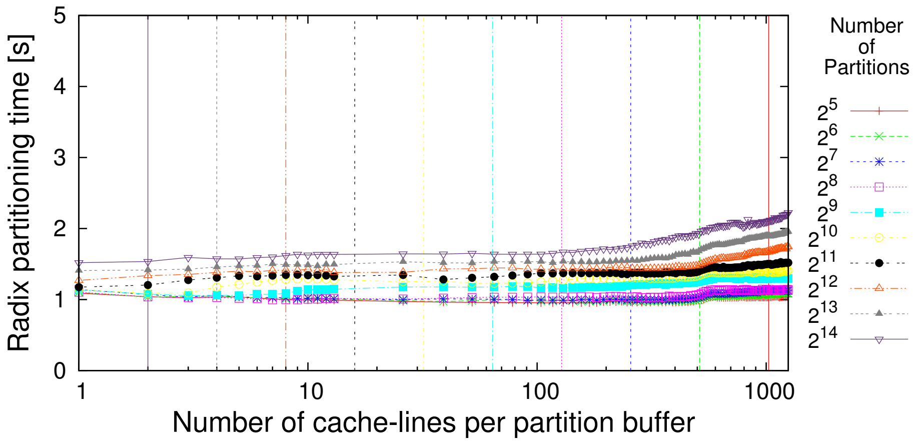
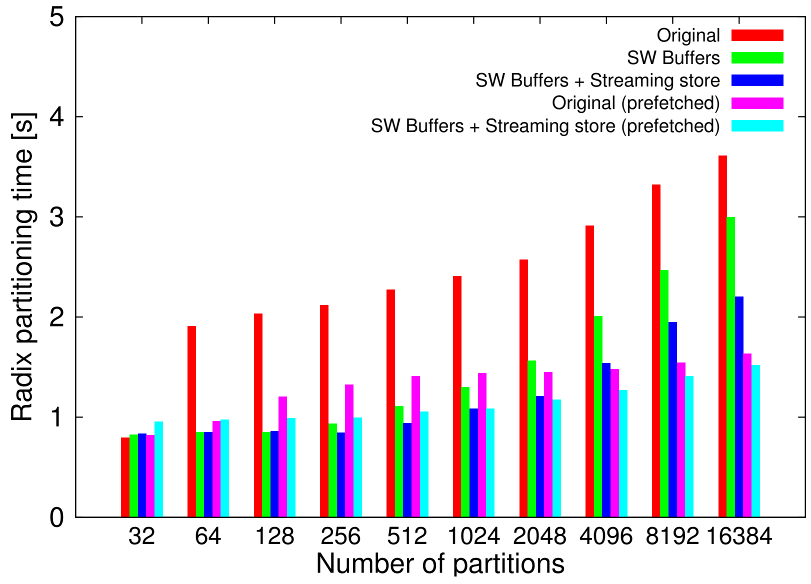


Figure 4.7: Visualization of radix partitioning using software-managed buffers with *prefetching*.

Figure 4.8(a) shows the runtime of the version that extends the buffered and streamed version with previously mentioned prefetching hints. Obviously, the performance is much more stable with respect to the buffer size, indicating that the prefetching indeed hides the cache-miss latency nicely. If we compare that with the versions without prefetching (see Figure 4.8(b)), we can observe a high impact of the prefetching hints for both the unbuffered and buffered version. From 4096



(a) Partitioning time of the *buffered & streamed & prefetched version.*



(b) Comparison of partitioning time of versions *with and without prefetching* (best buffer size shown).

Figure 4.8: Analyzing the effect of *prefetching hints* on the original version and the buffered & streamed version.

partitions on, both prefetching versions improve over all previous ones, showing that prefetching can indeed mask the cache misses from random writes. However, we can also observe that prefetching adds overhead and decreases the performance if misses are not a major problem, e.g. for less than 1024 partitions.

4.2.4 Micro Row Layouts

In the trivial implementation using software-managed buffers, each insertion of an entry into a buffer triggers the access of two cache-lines. First, the buffer fillstate is read from an array (first cache-line) and then, the entry is inserted (second cache-line). Additionally, when a buffer is flushed, the destination index is read from another array, leading to the access of a third cache-line. However, when limiting the buffer size to a single cache-line, it is possible to guarantee the access of only one line per entry. To do so, the last slot of each buffer is used to temporarily store the working variables. The aforementioned fillstate is stored in that way in [5, 40]. We additionally store the write destination in that slot. Figure 4.9 shows the layout of a single buffer.

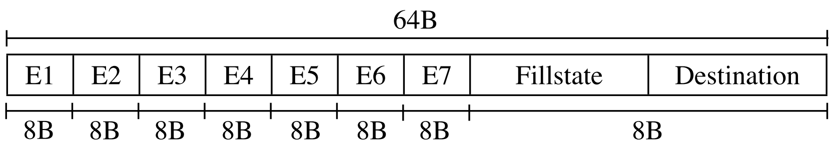


Figure 4.9: Storing both *fillstate and write destination* in the last slot of a buffer. The buffer has the size of a single cache-line.

Note that this technique does not sacrifice a slot in the partition buffer: the fillstate and the destination are overwritten by the last entry that is inserted and restored from a local variable after a flush. In Table 4.3 we present the impact of placing only the fillstate, or both fillstate and destination on the cache-line (basically a micro row layout) over the naive approach of storing them separately.

Obviously, these tiny changes have a surprising and significant impact on the runtime. For 16384 partitions, we see an improvement of factor 1.52x just due to this effect! Interestingly, we also observe that for less than 1024 partitions, storing that data in separate arrays (a micro column layout) shows the best runtime as these structures are small (e.g. 1KB for 128 partitions) and remain in L1 cache.

4.2.5 Small vs. Huge Pages

The initial motivation in using software-managed buffers was to reduce TLB-misses when writing to a large number of partitions. In this context, the size of the memory page is directly connected to the amount of required address translations when writing to the output array. In current linux kernels, pages of size 4KB and 2MB (*huge pages*) are supported. Huge pages reduce the amount of pages needed to allocate a consecutive memory area by a factor of 512, therefore decreasing the

# Partitions	Nothing [s]	Fillstate [s]	Fillstate & Destination [s]
32	0.84	0.91	0.87
64	0.85	0.93	0.89
128	0.86	0.90	0.88
256	0.84	0.88	0.87
512	0.94	0.94	0.92
1024	1.08	0.99	0.96
2048	1.21	1.02	0.99
4096	1.54	1.18	1.11
8192	1.95	1.42	1.31
16384	2.21	1.59	1.45

Table 4.3: Partitioning time of the *buffered & streamed version* with a *partition buffer size of one cache-line.* We distinguish on whether only the buffer fillstate, destination and fillstate, or nothing is stored on the partition buffer cache-line.

chance of TLB misses at access time. By default, our system was set up such that *transparent huge pages* were considered for all allocations, so we were actually using them in the previous experiments already. Let us now see how a smaller page size affects the runtime of the algorithms in Figure 4.10.

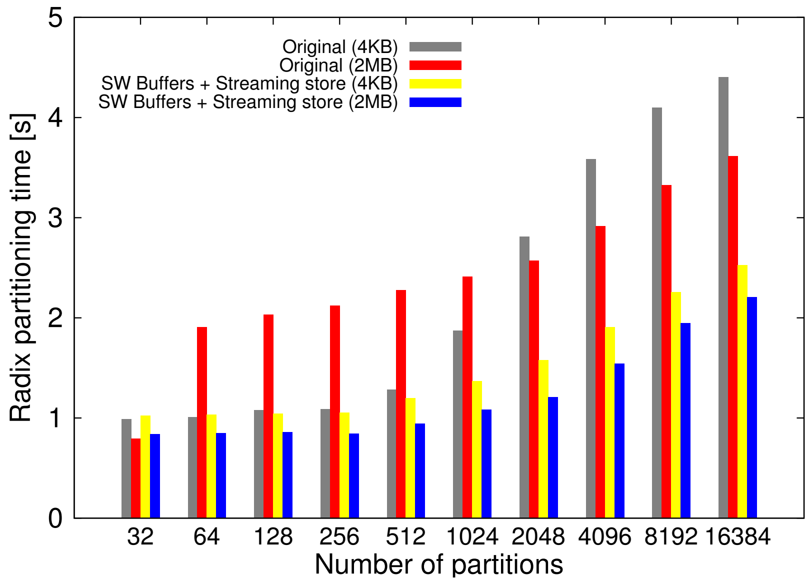


Figure 4.10: Partitioning time of the *original version* and the *buffered & streamed version* for 4KB and 2MB pages. 2MB resembles the setup of all previous experiments of this chapter.

For 32 partitions, the impact of the page size is negligible as the translations can be cached in any case. From that on till 1024 partitions, we see a clear advantage of 4KB over 2MB pages for the original version, since for a small number of partitions, a single partition is still larger than a huge page and thus, we benefit from the higher number of 4KB page TLB slots. This turns around as soon as multiple partitions fit on a single huge page. Interestingly, in combination with software-managed buffers and streaming stores, the 4KB pages do never pay off. By buffering, the potential TLB miss rate decreased already by factor b (see Section 4.2.1) and the higher allocation costs for 4KB pages render the advantage void.

4.2.6 Varying Experimental Setup

So far, we have evaluated all methods over an array of 100 million (key-payload) pairs of 8B each. Let us now see how the algorithms scale with a ten-fold increase of the number of entries to 1 billion and when doubling the entry size to 16B for a pair. Figure 4.11 shows the average slowdown factors over all tested numbers of partitions. Obviously, the tested methods scale almost linearly with the number of entries. A doubling of the entry size results in an average slowdown between only 1.31x for Original (prefetched) and 1.69x for SW Buffers + Streaming (prefetched), although the amount of data to move is twice as large as before. This indicates again, that not data transfer is the limiting factor but the processing of individual elements in terms of random access costs.

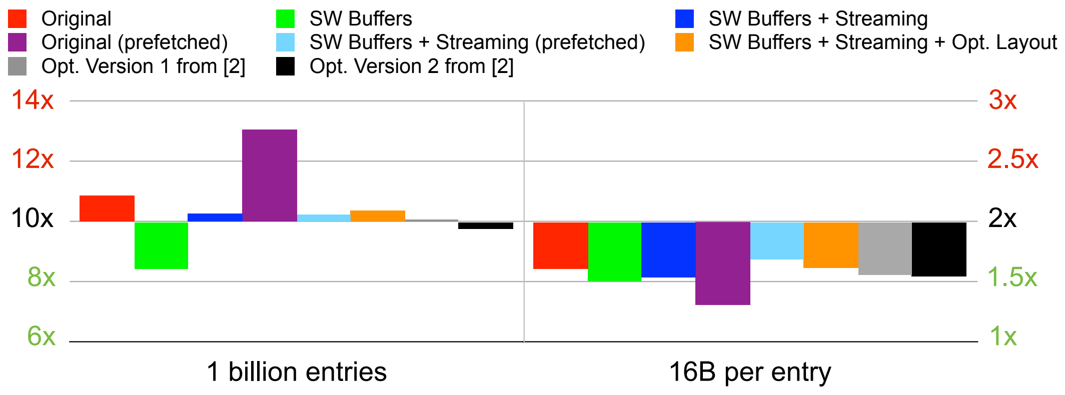


Figure 4.11: Slowdown when increasing the number of entries from 100 million to 1 billion and the entry size from 8B to 16B.

4.3 Conclusion

In this chapter, we saw that even a trivial algorithm such as radix partition can be significantly improved by making it aware of current hardware features, leading to an improvement over the original version of factor 2.5x.

Figure 4.12 gives a complete overview over all evaluated methods in comparison to two state-of-the-art implementations [4] used in [5]. As a side-effect of our evaluation, we were even able to improve over the existing methods in all tested configurations. For smaller number of partitions, using larger partition buffers without micro layouts shows the best performance. For 32 partitions, we recommend using the original version without any optimizations. For larger partition numbers, the layout-optimized single cache-line version pays off the most. Further, prefetching hints have a high impact on the runtime and are an option as soon as cache-space becomes the limiting factor.

Additionally to the identification of an absolute winner in terms of runtime, we carefully investigated the individual positive and negative impact of each optimization. Figure 4.13 summarizes for all tested optimization paths of Figure 4.1 those, that improved over the original version. This final overview showing the influenceability of the techniques by the partition count and their varying impact on the runtime clearly demonstrates a surprising difficulty of simple things.

Overall, in this chapter, we have seen that even simple algorithms with little algorithmic freedom can be drastically improved by mapping them appropriately to the hardware and operating system. In the upcoming chapter, we push this observation to the limit and turn around the approach: instead of doing an application-driven optimization, we investigate how to exploit existing features in the memory management system of the operating system to enhance algorithms and data structures from underneath.

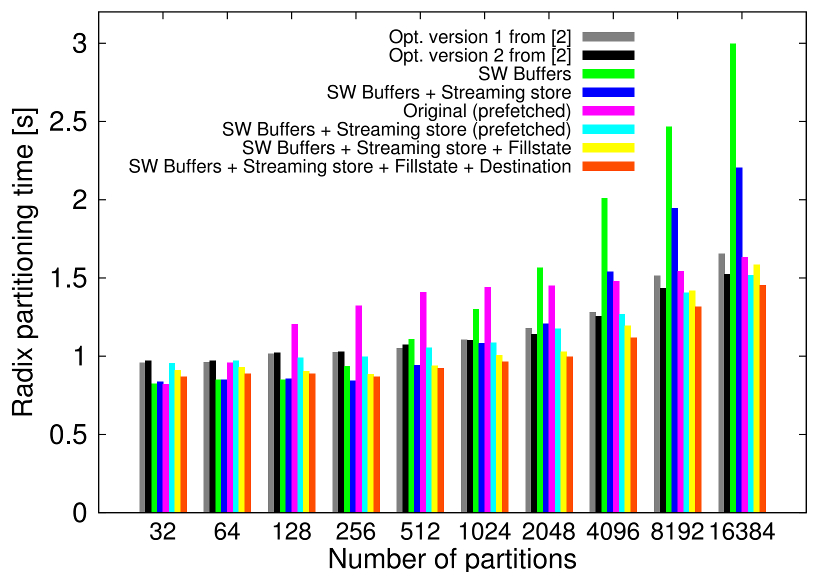


Figure 4.12: Overview of partitioning time of all optimized versions in comparison with a state-of-the-art partitioning from [5, 4].

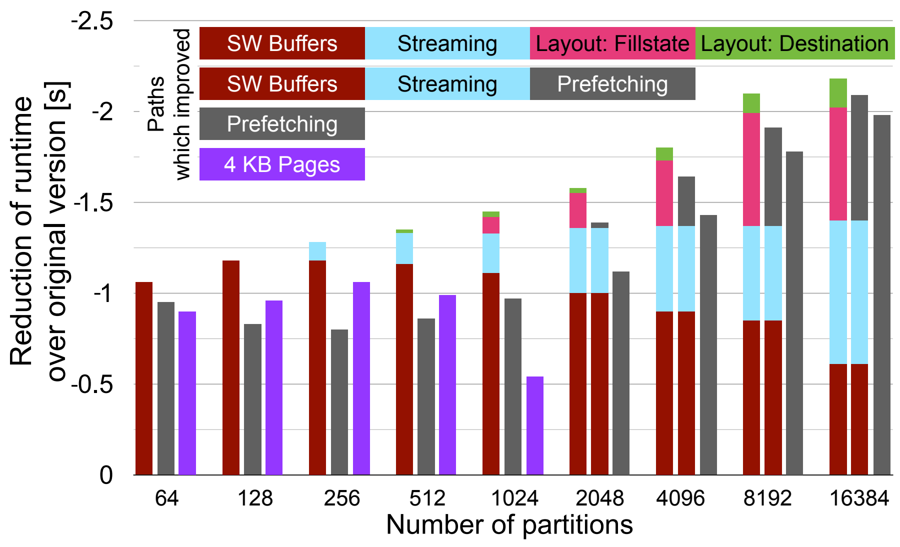


Figure 4.13: Reduction of runtime of the individual techniques over the original version. Each stacked bar corresponds to a path of optimizations in Figure 4.1.

Chapter 5

Rewiring Memory

Memory management is one of the most boring topics in database research. It plays a minor role in tasks like free-space management or efficient space usage. In the previous three chapters, it did not play a major role during the investigation of the algorithms. Here and there we realize its impact on database performance when worrying about NUMA-aware memory allocation (as seen for instance in Sections 3.2.7 and 3.2.8 for P-RPRS and P-CRS), or in data compacting, snapshotting, and defragmentation. But, overall, let's face it: the entire topic sounds as exciting as 'garbage collection' or 'debugging a program for memory leaks'.

What if there were a technique that would promote memory management from a third class helper technique to a first class citizen in algorithm and systems design? What if that technique turned the role of memory management in a database system (and any other data processing system) upside-down? What if that technique could be identified as a key for *re-designing various core algorithms* with the effect of outperforming existing state-of-the-art methods considerably?

We introduce RUMA: Rewired User-space Memory Access. It allows for *physiological* data management, i.e. we allow developers to freely rewire the mappings from virtual to physical memory (in user space) *while* at the same time exploiting the virtual memory support offered by hardware and operating system. We show that fundamental database building blocks such as array operations, partitioning, sorting, and snapshotting benefit strongly from RUMA.

5.1 Introduction

Database management systems handle memory at multiple layers in various forms. The allocations differ heavily in size, frequency, and lifetime. Many programmers treat memory management as a necessary evil that is completely decoupled from their algorithm and data structure design. They claim and release memory using standard `malloc` and `free` in a careless fashion, without considering the effects of their allocation patterns on the system.

This careless attitude can strike back heavily. A general example to counter this behavior is *manual pooling*. With classical allocators (like `malloc`) it is unclear whether an allocation is served from pooled memory or via the allocation of fresh pages, requested from the kernel. The difference, however, is significant. Requesting fresh pages from the system is extremely expensive as the program must be interrupted and the kernel has to initialize the new page with zeroes[1], before the program can continue the execution. Thus, careful engineers implement their own pooling system in order to gain control over the memory allocation and to reuse portions of it as effectively as possible.

However, manual pooling also complicates things. To write efficient programs, engineers rely on *consecutive memory regions*. Fast algorithms process data that is stored in large continuous arrays. Data structures store that data as compact as possible to maximize memory locality. This need is anchored deeply in state-of-the-art systems. For instance, the authors of [38] argue against storing the input to a relational operator at several memory locations for MonetDB: "*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.*"

Unfortunately, it is not always possible to gather large consecutive memory regions from the pool due to fragmentation. To work around this problem, memory can be claimed as chunks from the pool, using a simple software-based indirection. Allocations of memory are served by glueing together individual memory chunks via a directory. Thus, instead of accessing the entry at offset i by `a[i]`, the access is performed indirectly via `dir[i / chunkSize][i % chunkSize]`. Of course, this relaxes the definition of continuous memory, as every access has to go through the indirection now. As we will see, depending on the usage of the memory, this can incur significant overhead.

Obviously, demonstrated at the example of pooling, we face a general trade-off in memory management: *flexibility* vs *access performance*. Apparently, these prop-

[1]As the page that is served to resolve the page fault might contain freed data from another process, the page is zeroed by the kernel.

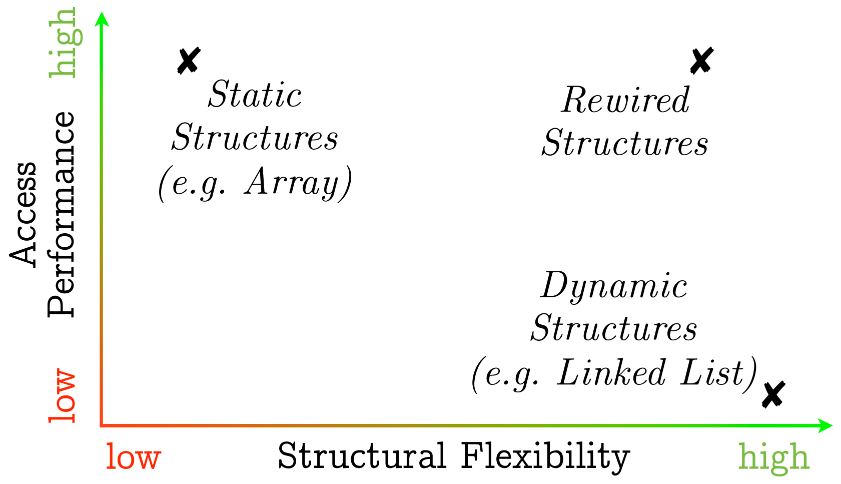


Figure 5.1: **Structural Flexibility** vs **Lookup Performance** — While static structures like the array provide fast and convenient access performance, their structure is hard to modify (extend, shrink). While dynamic structures like the linked list are easy to modify in their structure, the lookup of entries is indirect and slow. Rewired structures, as proposed in this chapter, offer direct access and high structural flexibility at the same time.

erties seem to be contradictory to each other. On the one hand, a static fixed-size array is extremely efficient to process in tight loops, but hard to extend, shrink, or modify structurally. On the other hand, a chunk-based structure as the linked-list is highly flexible, but suffers from an access indirection. Figure 5.1 visualizes this problem. The goal of RUMA is precisely to end this duality as rewired structures offer both the access performance of a consecutive memory region *while* providing the flexibility of a chunk-based structure.

To do so, we exploit a mechanism that is already present in todays operating systems: the virtual to physical page mapping, realized by a page table maintained by every process. Thus, instead of replicating this concept on the user level via a software-indirection, we make the existing virtual memory facility work for us. To do so, we introduce the concept of *rewiring memory*, which consists of two parts:

1. we reintegrate the concept of physical memory into user space without modifying the kernel in any way;
2. we actively manipulate the mapping from virtual to physical pages during runtime.

This allows us to both reclaim individual pages from a pool and still form actual continuous memory regions that can be accessed with little to no overhead. Apart from the benefits in memory allocation, flexible rewiring of individual pages can come very handy in numerous situations.

We believe that the design of efficient data management algorithms is only possible with the knowledge of how the internals of the memory system work — and consequently exploit them as much as possible.

This is exactly what we cover in this chapter, by making the following contributions:

(1) **RUMA.** We introduce **Rewired User-space Memory Access** (RUMA or 'rewiring' for short) allowing us to rewire the mapping of virtual memory addresses to physical addresses at runtime. This remapping happens at the granularity of pages (both small or huge pages). The technique does not need any modifications of the operating system and also does not interfere with process isolation or user security. In particular, we do not disable any virtual memory mechanisms, i.e. all of the benefits of virtual memory management are still available to us. The trick is to piggyback on the memory mappings that are maintained by the operating system anyways and exploit and manipulate the existing mappings for higher-level tasks. We can realize the entire technique in user space and purely in software, without modifying the linux kernel. We will investigate the toolset provided by the Linux kernel to separate virtual and physical memory in the user space and we will show how to freely manipulate the mapping from virtual to physical pages at runtime (see Section 5.2 and Section 5.3).

(2) **Micro-Benchmarking.** We benchmark the costs of rewiring in depth. First, we perform a set of *micro-benchmarks* to analyze both behavior and performance for memory allocation and access of rewiring in comparison to their traditional counterparts. Then, we inspect the costs and types of individual page faults to see their impact on the techniques and look at the expenses of shuffling items using rewiring respectively copying. Finally, we measure the impact of different access patterns on the techniques to understand the need for rewiring. We will learn that a major effect of rewiring is that it allows us to push one storage indirection down to the operating system. At the same time, rewiring drastically increases the flexibility of storage management without introducing considerable overhead. This has a dramatic runtime effect for all mass-operations reading or writing memory (see Section 5.4).

(3) **Applications.** Rewiring can be applied in many places in data management. It applies whenever data is copied or moved around, e.g. for resizable data structures, partitioning, merging, hiding of data fragmentation, and realizing multiple views on the same physical memory. Rewiring also applies in sit-

uations where data is read-only or very hard to update in place, e.g. when applying a set of changes collected in a differential file or when adaptively refining a part of an index. Investigating all of the possible applications of RUMA is beyond the scope of this work. From the long list of possible applications of rewiring we will focus on two applications that are central to many data managing tasks (see Section 5.5) and leave other promising applications to future work (see Section 6.5).

(4) **Rewired Vector**. We demonstrate the concept of rewiring at one of the most fundamental structures in data management: the array. It serves as a building block for fundamental main-memory structures like columns, hash tables, and indexes. One array operation that is particularly painful is resizing. We cannot enlarge an array without triggering costly physical copy operations, fresh page allocations, or wrapping the array in software into a list of arrays (leading to an additional storage indirection that is painful to handle for many processing tasks). This resizing problem is also at the heart of several read-optimized structures like column layouts and read-optimized indexes including differential indexes. Using STL's resizing array implementation (vector) as a pivot for this problem we will introduce a rewired vector. This data structure grows and shrinks page-wise without any copying of old entries, thus significantly improving over STL vector. Further, we are still able to use pooled pages underneath, in contrast to our second baseline using the system call `mremap` for the resizing. Finally, we compare the rewired vector with our third baseline using a software-indirection to realize the resizing. We show that only our structure manages to integrate comfortable flexibility into an array-based data structure (see Section 5.5.1).

(5) **Rewired Partitioning & Sorting**. We now pick an important building block in database systems: radix-based partitioning. It is a fundamental technique for indexing, join processing, and sorting. We re-investigate two state-of-the-art out-of-place partitioning algorithms, including the one we discussed in Chapter 4, which either perform a histogram generation pass before partitioning or maintain a linked list of chunks inside the partitions to handle the key distribution. We also test a version enlarging the partitions adaptively using `mremap`. We propose a rewired partitioning algorithm that manages to avoid the histogram generation pass altogether while still producing a contiguous output array by embedding our rewired vector. We show that partitioning and accessing the partitions in a separate sorting phase is significantly faster for rewired partitioning than for all the baselines (see Section 5.5.2).

5.2 A Brief Recap of Virtual Memory Management

Let us first briefly recap virtual memory management as supported by the *Linux* kernel. We focus on Linux as we believe this is by far the most common platform for data management applications and particularly servers.

Virtual memory has many advantages: processes may allocate more memory than physically available, the memory belonging to different processes may easily be protected, and all binaries may be compiled using static virtual addresses. Traditionally, the programmer works solely on the level of virtual memory. Physical memory is simply not accessible to him.

In most situations, the view solely on virtual memory is convenient and handy for the user. However, in recent years, programmers repeatedly identified situations in which an awareness of the underlying physical memory can be beneficial. For instance, the KISS-tree [29], a radix trie, implements an extremely wide root-level node of 256MB with the argue, that only a portion of it is actually physically allocated. Another example is the system HyPer [25], that exploits implicit copy-on-write performed between processes related using `fork`. The authors of [52] engineer a virtual-memory aware counting sort, that avoids the initial counting pass by exploiting massive over-allocation of the output array. Finally, in [10] the need for a user mode page allocation system is identified, that intends to separate virtual and physical memory in user space. The author requests the operating system developers to rewrite the system calls such that physical pages can be allocated manually by the programmer and mapped freely.

In the following, we will demonstrate that such a deep change of the system is not necessary at all: we show how *existing features* in Linux can be exploited to achieve exactly that: separate physical and virtual memory in user space and freely manipulate the mappings between them. How could we get access to those mappings?

5.2.1 Physical Memory — Main-Memory Files

There is a way to work around this limitation in form of *main memory files*. Linux provides main memory filesystems that are mounted and used like traditional disk-based filesystems, but backed by volatile main memory. Widely used representatives are `tmpfs` and `hugetlbfs` which are typically used for shared memory objects that can be backed both by small and huge pages. To evaluate the impact of the page size, we perform the micro benchmarks in Section 5.4 both with small and huge pages backed files where appropriate. Let us assume we have mounted a `tmpfs` under `/mnt/mmfs`. We can now create a fresh file using a simple call to the standard `open` system call as follows:

```
int fd = open("/mnt/mmfs/f", FILE_FLAGS, MODE);
```

This call tries to open the file that is named `f`. If that file does not exist (which will be our standard case), `open` creates the file and sets the permissions to the provided `MODE`. Furthermore, the call makes the file ready for read and write access by setting the `FILE_FLAGS` accordingly. Finally, it returns the file descriptor in form of an integer, which will serve as a handle to identify the file later on.

Alternatively, we can use the following system call to create an anonymous file, that does not need a mounted main memory files system:

```
int fd = memfd_create("f", FILE_FLAGS);
```

An anonymous file offers the nice property that it is automatically released if all references to it are dropped.

As a newly created file has a length of zero bytes by definition, we also have to set its size in the next step to s bytes. This can be done using a system call to the function `ftruncate(fd, s)`.

From this point on, we have a programmatic representation of physical memory. Although we do not directly handle physical memory pages, we know that file `f` is backed by physical memory. Specifically, for a page size of p, this means that in file `f`, the memory at offsets $[0; p-1]$ is backed by *some* physical page, say physical page $ppage_{42}$ and the memory at offsets $[p; 2 \cdot p - 1]$ is backed by another physical page, say $ppage_7$ and so on. Figure 5.2 visualizes the concept. Notice, that the mapping from offsets to physical files is handled by the file system and hence we do not have to reimplement that mapping.

Using a main-memory files system, we could now reimplement our own virtual memory mechanism. However, like that we would lose the existing hardware and operating system support for virtual memory management including all their benefits. For instance, we would neither be able to exploit the translation lookaside buffer of the CPU that caches translations from virtual to physical addresses nor could we use the hardware page walker in case of a TLB miss. Hence, we need to perform one additional step.

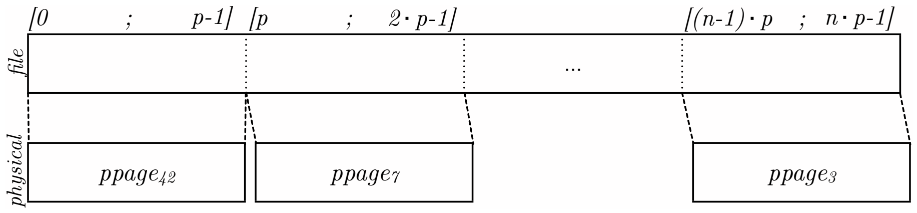


Figure 5.2: **Backing of a main memory file by physical pages when using a main-memory file system**. The file offsets in interval $[i \cdot p, (i+1) \cdot p - 1]$ are mapped to *some* physical page by the file system. This mapping is not accessible for the user.

5.2.2 Virtual Memory — mmap

We need to bring virtual memory into the game. To do so, we have to map a region of virtual memory to the file. It is important to note that in the following all accesses to memory will still be performed entirely through virtual memory — there will not be any direct data accesses to physical memory. On linux, the way to create a virtual memory mapping onto a main memory file is performed by a system call to `mmap` as follows:

```
void* vmem = mmap(b=NULL, s,
                  PROT_READ | PROT_WRITE,
                  MAP_SHARED, fd, 0);
```

In this example, the `mmap` call creates a virtual memory area of size s and maps it to the main memory file `fd` (as created in Section 5.2.1). Passing `b=NULL` as the first argument indicates, that the kernel should decide on the start address b of the new virtual memory area, which is guaranteed to be at a page boundary. Later on, we will pass an existing address b to remap existing mappings (see Section 5.3). `MAP_SHARED` will be explained in Section 5.2.5. The last argument, which is `0` in this example, specifies the start offset for the mapping into the file. By mapping a virtual memory area of size s into file `fd` at offset `0`, we basically map s/p many virtual pages to the physical pages backing the main-memory file (we consider s to be a multiple of the page size). Thus, we have created a handmade virtual memory to file offset mapping, which is then again indirected by the file system to physical pages. Figure 5.3 visualizes this mapping.

In total, we have created a two-level mapping of virtual memory to physical pages. Sounds expensive? Don't worry. In the following, the filesystem will only be called very rarely. Only when accessing a virtual page for the first time, the file system is involved. The second access to that virtual page does not differ from accessing

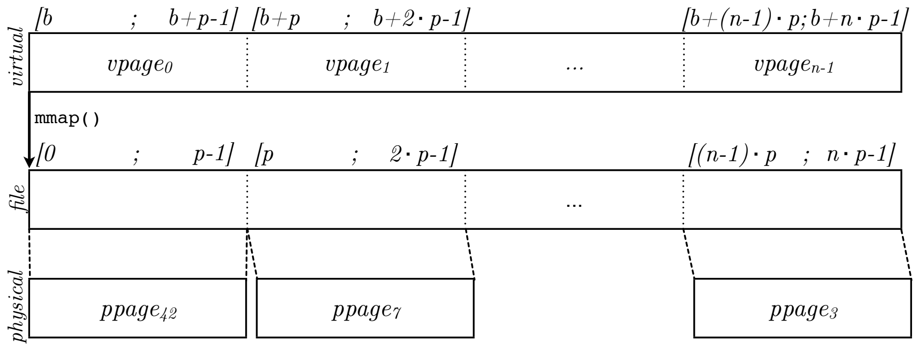


Figure 5.3: **Mapping of virtual memory to main memory file**. A single call to `mmap` maps multiple virtual memory pages to the main-memory file. The start address of the area is denoted as b. The virtual page $vpage_i$ starting at address $b+i\cdot p$ is mapped to the file at offset $i\cdot p$ which in turn is backed by some physical page.

normal virtual memory. Hence, all calls except the first one have to resolve a one-level mapping only.

5.2.3 Virtual Memory: Reserving vs Copy-on-Write

Let us go back to the normal memory allocation using `mmap` on anonymous memory, e.g. as used by `malloc` for large allocations:

```
char* data = mmap(NULL, 42 * p,
                  PROT_READ | PROT_WRITE,
                  MAP_ANONYMOUS | MAP_PRIVATE,
                  -1, 0);
```

The incautious programmer might think that this call allocates 42 physical main-memory pages of size p. What actually happens is the following: the operating system simply *reserves* a virtual address space of size $42\cdot p$. This is done by creating a *conceptual memory mapping* in form of a `vm_area_struct`. By maintaining these structures, the kernel keeps track, for each process independently, which virtual address ranges have been *reserved* for that process. Notice that the operating system makes only *one* entry for all 42 pages at this point; no entry is inserted into the page table. The process' page table will only be populated on demand whenever the data is accessed. This works as follows.

Let us assume we try to read from some virtual address, e.g. by accessing `data`$[7\cdot p+16]$. In that case, the operating system will try to retrieve the mapping of that virtual page to its physical page at the virtual address `data` $+ 7 \cdot p$. If that mapping is neither in the TLB cache, nor in the page table, this will trigger a page fault. The operating system will now inspect the `vm_area_struct` instances to determine how this page fault can be resolved. It finds the previously created `vm_area_struct` instance which determines that `data` $+ 7 \cdot p$ is a valid virtual address as it was reserved by this process.

Hence, the operating system will insert a new mapping into the page table from the virtual page starting at `data` $+ 7 \cdot p$ to the so called *zero-page*. The zero-page is a read-only page where all bytes are set to zero. Now, we have a valid entry for this virtual page, hence we can serve the read request to `data`$[7 \cdot p + 16]$. It will return 0. Any read request touching a virtual page that does not have an entry in the page table will be mapped like this.

Now let's assume we write to the same page `data` $+ 7 \cdot p$ for the first time say at address `data`$[7 \cdot p + 20]$. What happens? In this case the operating system will apply copy-on-write (COW). It will get a new physical page, copy the contents of the zero-page over that physical page, write the new value to that page, and update the entry in the page table: the virtual page starting at `data` $+ 7 \cdot p$ now maps to the new physical page. This happens only for the first write to any virtual memory page.

In summary, a call to `mmap` does neither allocate physical memory nor does it modify the page table.

5.2.4 Memory-backed File — Reserving vs Copy-on-Write

As we have discussed the behavior in the two-layered case in the previous Section 5.2.2, let us now inspect the differences in the three-layered memory-mapping, which we have in the case of a file-backing, as displayed in Figure 5.3.

Just like in the two-layered mapping the call to `mmap` simply creates a `vm_area_struct` without modifying the page table in any sense. What happens now if we access a virtual page for the first time, i.e. we trigger a page fault? In that case, the operating system will inspect the `vm_area_struct` and notice that this address space is backed by a file at a certain offset, not by anonymous memory. The operating system will then request the file system to return the address of the physical page it uses to back that offset. As the physical page does not yet exist, the file system allocates it (equivalent to the page fault handling for anonymous memory), and returns the physical address of it. With this address, the page table entry can be created. Of course, the page fault handling mechanism employed by the file system adds some overhead compared to a page fault on anonymous memory. But as the page fault through the file system happens only for the first access to that virtual page, the three-layered mapping is basically turned into a two-layered mapping after this. Subsequent access costs are equal to those of anonymous memory. We will micro-benchmark this in detail in Section 5.4.3.

5.2.5 Shared vs. Private Mappings

Especially in the context of file backed memory mappings, two `mmap` options should be discussed: `MAP_SHARED` and `MAP_PRIVATE`. These modes steer how changes are propagated through mappings and whether COW is performed. The option `MAP_SHARED` propagates[2] any changes made through it to the underlying file and hence to the underlying physical pages (COW is turned off).

The option `MAP_PRIVATE` does not propagate writes to the underlying file. Instead, the first write attempt to a virtual page through a private mapping triggers COW on that page. Important to note here is that the new physical page inserted by the operating system is an anonymous page, even if the mapping is file backed. This behavior can be very useful in implementing a lazy data copying mechanism.

5.2.6 Page Table Population Strategies

Apart from populating the page table lazily at access time, as described in Section 5.2.3, other options are possible. In the following, we list the four possibilities:

1. The first strategy is to populate the page table on demand, i.e. every page fault triggers an entry into the page table (this strategy was described in Section 5.2.3).
2. A second strategy is to bulk-load the page table. The `mmap` function supports an option to pre-fault the pages (option `MAP_POPULATE`) at mapping time by performing a read-ahead of the file. Population at mapping time can be useful when mapping a large number of small pages, in particular when the mappings are known a priori. However, as the option triggers the reading of the entire pages, which is wasteful and unnecessary on main-memory files, we do not recommend using it in this context.
3. A third strategy to populate the page table is by manually reading a byte of each page. Recall that our mapping is file-backed. Therefore, each of these

[2]Changes in a *disk-backed* file system may not be propagated immediately to the file, `msync` can be used to force changes to disk.

reads will insert the correct mapping into the page table rather than the mapping to the zero page. This has the advantage of avoiding to read the entire page which is unfortunately done by `MAP_POPULATE`.

4. A fourth strategy is to pre-fault the pages asynchronously in a separate thread. Thus, the `mmap` call is not blocked by the actual bulk-loading operation.

5.3 Rewiring Memory

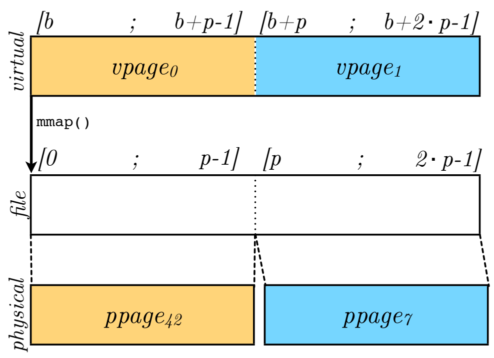
Up to this point, we have discussed virtual memory that is backed by main memory files instead of anonymous memory. We did this in order to reintroduce the concept of physical memory into the user-space, to get separate handles for virtual and physical pages. However, so far, we did not exploit the power of this new freedom and what we believe to be the strongest aspect of this approach: the ability to actively modify, i.e. *to rewire* the mappings from virtual to physical pages. Doing so will allow us to *use rewiring as a building block for designing algorithms and data structures*.

To get started, let us look at the following simple example. Assume we have an array `data` occupying two pages of size p each. Let's assume we want to swap the contents of the two pages backing the array. Traditionally, this is done with a three-way-swap:

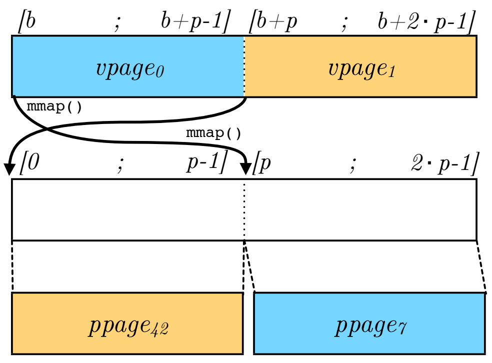
```
char* tmp = malloc(p);      // get temporary page
memcpy(tmp, data, p);       // three
memcpy(data, data + p, p); // way
memcpy(data + p, tmp, p);  // swap
free(tmp);                  // free tmp page
```

These operations trigger actual physical copy operations of the data. In total, three pages are physically copied while an additional physical page is allocated as helper memory. Using rewiring, we can perform this swap using neither physical data copying nor any allocation of helper memory.

Figure 5.4 shows the concept. The initial state is shown in Figure 5.4(a). A linear mapping of a two page sized virtual memory area starting at address b into the file exists. We created it as follows:



(a) Before: *vpage*$_0$ shows the contents of *ppage*$_{42}$, *vpage*$_1$ shows the contents of *ppage*$_7$.



(b) After: *vpage*$_0$ shows the contents of *ppage*$_7$, *vpage*$_1$ shows the contents of *ppage*$_{42}$.

Figure 5.4: Rewiring two pages. Notice that no data is physically copied across physical pages.

```
// (0) create a linear mapping:
char* b = (char*) mmap(NULL, 2 * p,
                       PROT_READ | PROT_WRITE,
                       MAP_SHARED, fd, 0);
```

Precisely, we map the virtual page *vpage*$_0$, starting at address b, to the file at offset 0, which is backed by physical page *ppage*$_{42}$. Consequently, we map the virtual page *vpage*$_1$, starting at address $b + p$, to the file at offset p, that is backed by physical page *ppage*$_7$.

We now perform the actual page swap solely by rewiring the mapping from virtual pages to the file respectively the physical pages. Figure 5.4(b) shows the state after rewiring. Note that no data was physically copied across pages. All we did is update two links using the following two calls to `mmap` in combination with the `MAP_FIXED` option:

```
// (1) Remap first virtual page to file offset p:
mmap(b, p,
     PROT_READ | PROT_WRITE,
     MAP_SHARED | MAP_FIXED, fd, p);
```

```
// (2) Remap second virtual page to file offset 0:
mmap(b + p, p,
     PROT_READ | PROT_WRITE,
     MAP_SHARED | MAP_FIXED, fd, 0);
```

The first `mmap` call (1) remaps the virtual page $vpage_0$, starting at address b, to the file at offset p, which is backed by the physical page $ppage_7$. The key component of this control is using `mmap` in combination with the argument `MAP_FIXED`. By providing this option, we can remap a virtual page to a different physical one by providing a new offset into the file. The second call (2) remaps the virtual page $vpage_1$, starting at address $b + p$, to the file at offset 0, which is backed by the physical page $ppage_{42}$. After this, the content visible through the virtual pages is swapped without copying any physical pages.

Obviously, we could have achieved a similar effect by implementing a software-directory by ourselves. Like that we could have swapped pointers to the two pages just like we swapped the two mappings through `mmap`. However, doing so *we would have introduced one additional indirection for every access to an array slot*. Traversing this indirection can be very costly under certain access patterns. The beauty of rewiring is that virtual memory maintains one indirection *anyways*. Instead of adding an auxiliary indirection level, we let the existing one work for us. Hence, the central question we will be exploring in the following is: when can we piggy back on the existing virtual to physical page mappings *in order to get rid of one level of indirection in our algorithms?* In other words, how could we change state-of-the-art data processing techniques to delegate some of their implementations to operating system and hardware?

Before approaching these questions, let us outline how rewiring could be wrapped inside of a lightweight library, that simplifies the integration of the technique into existing applications. At the heart of rewiring is the separation of physical and virtual memory and the mapping between these two memory types. Thus, the core of the library consists of the following three components: (1) The allocation of physical memory that wraps the creation, maintenance, and initialization of main memory files. (2) The allocation of virtual memory that wraps the calls to `mmap` and keeps track of the state of the current mappings. (3) The rewiring functionality, which internally calls `mmap` to establish and modify the mappings between virtual and physical pages. The library can further increase the usability by rewiring on page identifiers instead of raw memory addresses, while still exposing them to the user if requested.

Overall, these components already suffice to form the core of a rewiring library, that can be assembled into existing code. However, before we can actually integrate rewiring into applications, we need to understand the possible overheads of rewiring in depth.

5.4 Micro-Benchmarks

In this section, we will perform a set of micro-benchmarks to understand the impact of virtual memory and rewiring. Firstly, we inspect the costs of allocation and access. Secondly, we evaluate the page fault mechanism in detail. Thirdly, we dynamically rewire existing mappings and measure the cost. Finally, we look at the impact of different access patterns on the memory.

5.4.1 Experimental Setup

We run all experiments on a two-socket server consisting of two quad-core Intel Xeon E5-2407 with a clock speed of 2.2 GHz. The CPU does neither support hyper-threading nor turbo mode. The sizes of the L1 and L2 caches are 32KB, respectively 256KB; the shared L3 cache has a size of 10MB. The processor offers 64 slots in the fast first-level data-TLB, to cache translations of virtual to physical 4KB pages. In a slightly slower second-level TLB, 512 more translations can be stored. For 2MB huge pages, the TLB cache can store 32 translations in L1 dTLB. In total, the system is equipped with 48GB of main memory, divided into two NUMA regions. For all experiments, we make sure that all memory (both file-backed and anonymous) is allocated on a single NUMA region and that the thread is running on the socket attached to that region. The operating system is a 64-bit version of Debian 8.1 with Linux kernel version 3.16. The codebase is written in C99 and compiled using gcc 4.9.2 with optimization level 3.

Throughout the following micro benchmarks, we will use a dataset of 1 billion entries, where each entry is of type `uint64_t` and has a size of 8B leading to a total size of roughly 7.45GB, unless mentioned otherwise.

5.4.2 Allocation Types

Before we can start with the evaluation, we have to define what an allocation actually means in our context. In the following, we distinguish three different types of allocations, that will serve as the competitors in our evaluation:

(a) *Private Anonymous Memory* — Allocating a memory area of n pages means mapping a consecutive virtual memory area of n pages to n (unfaulted) anonymous physical pages. The operating systems resolves all page faults with fresh physical pages, we can not use a pool. We can access the memory directly.

(b) *Software-Indirected Memory* — Allocating a memory area of n pages using a software-indirection means creating a directory of n slots where each slot contains the virtual address of a page in the pool. We realize the pool using (faulted) virtual pages. We translate and redirect any access through the directory.

(c) *Rewired Memory* — Allocating a memory area of n pages using rewiring means mapping a consecutive virtual memory area of n pages to n (faulted) pages in the pool. We realize the pool using a main-memory file and we access the memory directly.

When we use a page pool in (b) and (c), the way in which we select the mapped pages can have an influence on the mapping and access performance. In the best case, where the pool is non-fragmented, the n requested pages can be gathered consecutively. In the worst case of a highly fragmented pool, we have to gather each page individually. Thus, we test both extremes in the following where meaningful. Note that we do not count the effort to find and maintain unused pages in the pool.

5.4.3 Allocation & Access

The first step in understanding the runtime costs of memory usage is measuring memory allocation and its implication on access performance. To analyze the total cost and impact of allocation using different techniques, we perform the following simple experiment: Firstly, we *allocate* a memory area. The allocation of a memory area highly differs depending on the used memory management, see Section 5.4.2. Secondly, we *write* random values to the area sequentially from start to end. Finally, we *repeat* the sequential write pass. We compare the total time of the three steps under the allocation types (a), (b), and (c), where for (b) and (c) using the page pool, we test both a sequential and random assignment of pages. Figure 5.5 shows the results. For the allocation type (a) using private anonymous memory, we focus solely on huge pages as the page fault costs for small pages are significant and render them inferior over huge pages in basically all scenarios. For the types (b) and (c) we test both small and huge pages, as the page size influences the flexibility of the memory and both sizes can come handy in certain situations.

We can see that the allocation phase is basically for free in all cases, except when

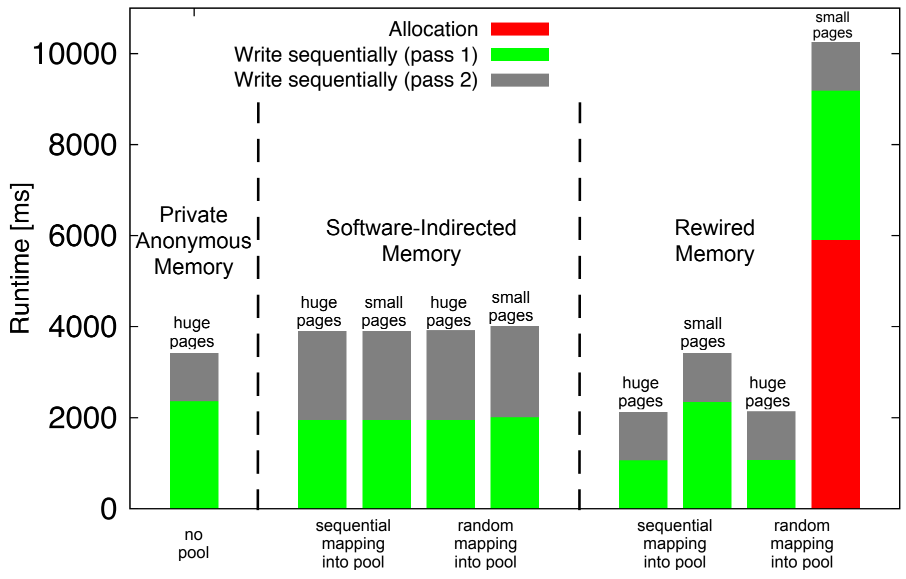


Figure 5.5: Allocation costs (red), sequential write costs for the first pass (green), and sequential write costs for the second pass (grey). The graph compares the allocation types of Section 5.4.2 using both small and huge pages where appropriate.

randomly rewiring individual small pages. When rewiring on a per huge page basis (random pool), the 3815 necessary calls to `mmap` are negligible. When mapping small pages, the almost 2 million calls (and the generated `vm_area_struct` instances) are painful. However, mapping each page individually simulates the extreme case of the most fragmented pool that is possible. With a good pool management scheme underneath, this is very unlikely to occur [24].

More interesting is the runtime of the first sequential write pass. Here, rewired memory clearly outperforms the remaining two methods as it manages to combine the reusing of pages as well as direct access. During the first writing to the rewired memory, soft page faults are triggered, that are only significant for small pages. We will inspect the different types of page faults (soft vs hard) in Section 5.4.4 in detail. The software-indirected memory suffers from the offset translation and the lookup into the directory. Overall, rewired memory is around 1.8x faster than the software-indirected memory for huge pages. For private anonymous memory, the first write performs even worse than in the software-indirected case and is 2.2x slower than rewired memory using huge pages. As the mapping does not rely on a pool, the first write to a virtual pages triggers a costly hard page fault, that is served by the operating system with a zeroed physical page.

The second sequential write pass shows for rewired memory the same runtime as the first pass in the case of huge pages, for small pages it is significantly cheaper. The difference between the passes is that the second one can already exhibit a populated page table. For huge pages, as the number of entries is small, this difference is negligible. For small pages however, is becomes visible, where the second pass is 2.2x (sequential pool) to 3.1x (random pool) faster than the first one. The software-indirected memory still suffers from the translation and directory access costs. The write runtime of the private anonymous memory is also significantly faster than in the previous pass and equals the one of rewired memory now. The reason for this is that the second pass, in contrast to the first one, does not suffer from page faults anymore.

Overall, we clearly see that rewiring combines the advantages of both base-lines while avoiding the drawbacks. Rewiring offers the flexibility of individual page re-usage while offering direct access without a slowdown.

5.4.4 Costs of an Individual Page Fault

In order to understand the high costs of the first write of private anonymous memory and the advantages of using a page pool in more detail, we perform an experiment where we trigger a number of page faults and compute the average costs. We use a dataset of 2GB that is either backed by 1024 huge pages or by $524,288$ small pages.

We simply loop over these pages and write to the first byte on each. Here, we can evaluate only two out of our three allocation types of Section 5.4.2. Firstly, we look at (a) (Private Anonymous Memory), which is the two-level memory mapping where the virtual pages are not yet backed by any anonymous physical pages. Thus, the first access will trigger a kernel request for a zeroed page. Again, we focus on huge pages due to the limited usability of small pages in this context.

Secondly, we look at type (c) (Rewired Memory) where the physical pages come from our pool of initialized pages. Again, we divide between sequentially and randomly mapped pages and test both small and huge pages. Additionally, we look at both `MAP_SHARED` and `MAP_PRIVATE` (recall Section 5.2.5), as it influences the costs and both can be useful in certain situations.

Note that we do not evaluate (b) (Software-Indirected Memory) in this context, as it is not connected to page faults in any way.

Table 5.1 shows the results. We repeated all measurements 100 times, each run performed in a fresh process. Let us focus on the "per page" results. We see that the costs of faulting a page of private anonymous memory are significant

Backing Type	**Pooled Pages?**	**Page Size**	**Mapping Type**	**Amortized Time for page fault [ns]**	
				per page	**per KB**
Private Anonymous Memory	no	huge	private	600 251	293
Rewired Memory	yes (sequential)	small	shared	686	172
			private	2 048	512
		huge	shared	710	< 1
			private	526 519	257
	yes (random)	small	shared	1 053	263
			private	2 727	682
		huge	shared	810	< 1
			private	528 910	258

Table 5.1: **Cost of a single page fault** for the allocation types (a) (Private Anonymous Memory) in comparison with (c) (Rewired Memory) from a page pool. We trigger page faults for a dataset of 2GB and report the average costs amortized per page and per KB.

with around $600 \mu s$. In comparison with its direct counterpart, rewired memory using huge pages with `MAP_SHARED` option that faults in $710 ns$, private anonymous memory is three orders of magnitude slower, i.e. this clearly shows that zeroing the page is the dominant part of the page fault, not setting up and inserting the page table entry, which is done by both methods. Therefore, we have to distinct between two types of page faults: *hard* and *soft* page faults. If the page is freshly claimed from the kernel, we have a hard page fault. If the page exists already, for instance in a file representing a pool, and only the page table entry must be inserted, we face a soft page fault. The expensive page cleaning is again confirmed when using the `MAP_PRIVATE` option, where a fresh anonymous page is used to resolve the COW. The difference between mapping the pooled pages sequentially and randomly is only visible for small pages, where in total $524,288$ pages are mapped. Here, a sequential mapping is up to 1.5x faster than mapping them individually, as only a single `vm_area_struct` is queried in contrast to hundreds of thousands.

The major takeaway message of this experiment is: the actual page fault costs are negligible if the physical page pointed to was already created before (soft page fault), which is the case when we use a user-managed page pool.

Type of Backing	Page Size	Mapping Type	Alloc. Time [ms]		Write Time [ms]		Total Time [ms]	
			MAP_POPULATE	async. population	MAP_POPULATE	async. population	MAP_POPULATE	async. population
untouched file	small	shared	353.76	0.10	158.26	782.33	512.02	782.43
		private	554.25	0.09	105.11	810.73	659.36	810.83
	huge	shared	180.24	0.17	105.93	209.95	286.17	210.12
		private	177.58	0.18	105.69	209.72	283.27	209.91
initialized file	small	shared	47.06	0.12	157.45	168.11	204.50	168.22
		private	327.41	0.12	106.67	500.49	434.07	500.60
	huge	shared	0.62	0.19	105.90	105.88	106.51	106.08
		private	206.40	0.20	105.94	257.33	312.34	257.53

Table 5.2: Effect of **page table population strategies** under variation of **backing type** (was the file initialized beforehand or not), **page size** (small or huge pages), and **mapping type** (shared or private). Allocation time encloses the times to create the main memory file (if requested) and to create the mapping. Write time presents the time to write to the mapping sequentially.

5.4.5 Page Table Population Strategies

As already described in Section 5.2.6, there are different population strategies possible. Besides from the lazy population, which is basically the default strategy, the two most interesting alternatives are using either the option `MAP_POPULATE` to delegate the page table population to the operating system during the `mmap` call or to pre-fault the pages manually in our own routine that writes to the first byte of each page asynchronously.

To investigate the usability of these two strategies, we create a mapping to a file, populate it using one of the mentioned methods and then write to it. Table 5.2 presents the allocation time, write time, and total time under a variation of the backing type (was the file initialized beforehand or not), page size (small or huge pages), and mapping type (shared or private). First of all, we can observe that the allocation time using `MAP_POPULATE` is drastically higher than that of the asynchronous approach, as `mmap` is blocking during the population in the former case. Especially interesting is the different semantic of `MAP_POPULATE` depending on the chosen mapping type. In combination with a private mapping, the population is significantly more expensive than in the shared case, as COW is triggered already during population. The only exception to this are private mappings to untouched files, that are backed by huge pages. Further, we also see again that we have to distinct between two types of page faults: the ones where only the page table entry is missing (soft page fault) in contrast to those, where also the file has to allocate a page for backing (hard page fault). The first one is considerably cheaper, as seen at the allocation time between the untouched and the initialized file. The times to write are very constant using `MAP_POPULATE`. This is not the case for the asynchronous population, which can not completely hide the page faults, as the sequential writing catches up quickly. Still, in total, it can pay off especially on initialized files to populate asynchronously, especially, if there is a time frame between mapping and accessing.

5.4.6 Costs of Rewiring Memory

So far, we have seen how memory allocation, access, and faulting behaves for rewired memory in comparison with the traditional approaches. Let us now see how expensive the actual rewiring of memory is. To answer this question, we perform the following experiment: we form chunks of size $2^x \cdot 4$KB and randomly shuffle the data at this granularity. The exponent x is varied[3] from 0 to 19.

For the traditional approach using private anonymous memory and `memcpy` shuffling implies physically copying the chunks into a separate, fresh array. For the rewired approach, we establish a new virtual memory area that is mapped. All shuffling can be done through rewiring virtual memory. Figure 5.6 shows the results for both small and huge pages for rewired memory. The rewired versions are additionally tested with manual population, i.e. directly after mapping a chunk of memory, we access the first byte of each page of the chunk to trigger the page fault.

We can see that the chunk size has a large impact on the runtime of the shuffling for rewired memory. For very high granularities (e.g. a chunk size of 4KB), the calls to `mmap` create a significant overhead ($1,953,125$ calls for 4KB chunks).

Recall that for a single mapping of a contiguous virtual memory area to *one* physical offset in the file we only require a single `vm_area_struct` which is kept in a separate tree structure maintained by the kernel. For the shuffling of chunks however, we need (in the worst case) one `vm_area_struct` per chunk. This overhead can be observed in Figure 5.6. We can see that for small chunk sizes of about 4KB the costs of rewiring are more expensive than the actual physical copy operation of private anonymous memory. However, as soon as we start increasing the size of the chunks, rewired memory clearly outperforms private anonymous memory. We can also observe that the page table population by triggering soft page faults is not for free and observable, at least for small pages.

In summary, when using reasonable large chunks sizes, rewiring is considerably cheaper than the actual copying of the data.

[3]Notice that $2^{19} \cdot 4$KB= 2GB is the largest chunk size fitting two times into the array.

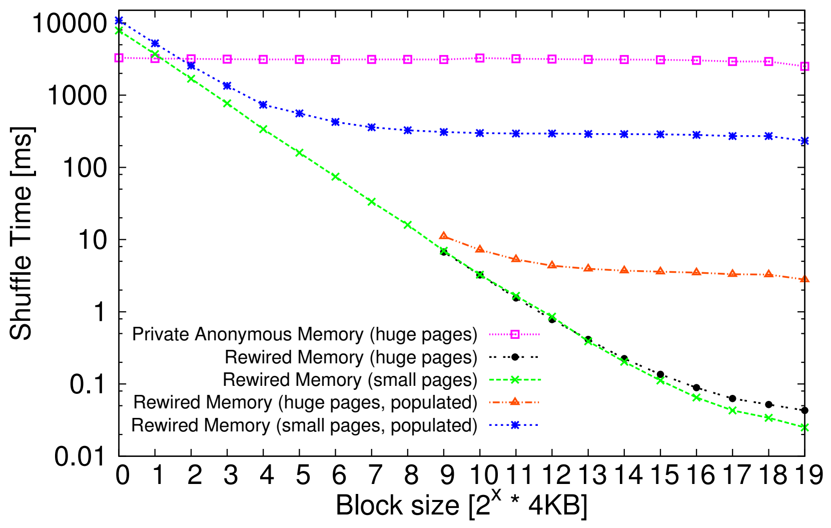


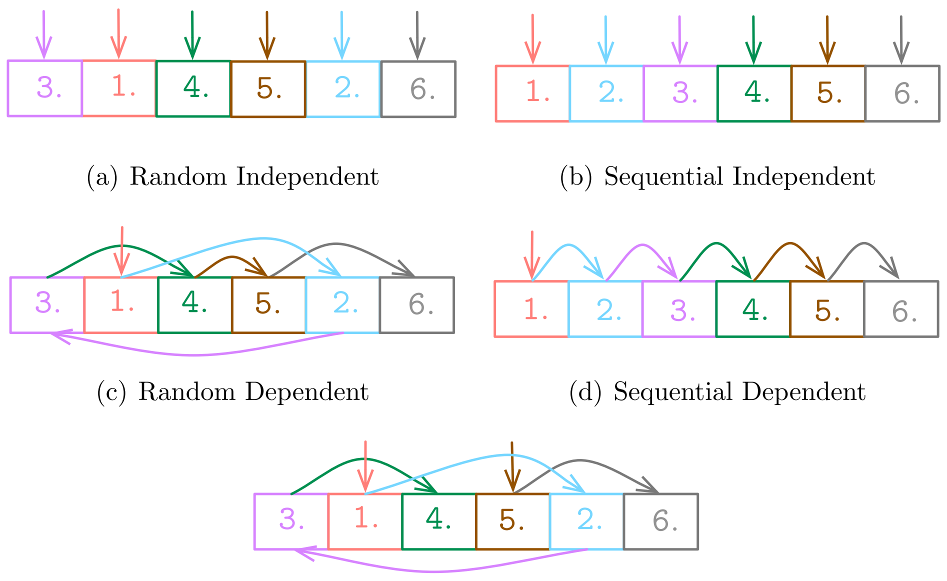
Figure 5.6: **Time to shuffle** chunks of $2^x \cdot 4$KB size. As the test array consists of $1,953,125$ chunks of 4KB, we can shuffle at most chunks of size $2^{19} \cdot 4$KB. We compare the traditional shuffling using private anonymous memory and `memcpy` with rewired memory (with and without explicit page table population). For huge pages, the highest granularity we can test is $2^9 \cdot 4\text{KB} = 2\text{MB}$.

5.4.7 Impact of Access Pattern

Previously, we have seen that a page-wise software-indirection is a possible alternative to rewiring when implementing a pool-based memory allocation system. However, when it comes to accessing the memory, overhead must be factored in due to additional index translation and access of the auxiliary directory. In Figure 5.5 of Section 5.4.3 we already saw that sequentially reading through an indirection is considerably more expensive than scanning a flat array. The question arises whether accessing memory through the software-indirection is generally more expensive than direct memory access. Thus, we now test the following access patterns on both rewired memory and a software-indirection. For all tests, page faults are already resolved.

We distinct the following five patterns, which we visualize in Figure 5.7:

- For *random, independent* pattern (Figure 5.7(a)), we access the memory at random places where the access index is generated individually.
- For the *sequential, independent* pattern (Figure 5.7(b)), we scan the memory from beginning to end where the access index is the iteration variable of the



(a) Random Independent (b) Sequential Independent

(c) Random Dependent (d) Sequential Dependent

(e) Random Mixed

Figure 5.7: **Visualization of Access Patterns**.

loop.

- For *random dependent* access (Figure 5.7(c)), we access the memory at random places where the access index is the result of the previous access. There is only one cycle in the dependency chain.
- For *sequential dependent* access (Figure 5.7(d)), we access the memory at sequential places where the access index is the result of the previous access. Again, there is only one cycle in the dependency chain.
- Finally, for *random, mixed* access (Figure 5.7(e)), we access the memory at random places. However, we interleave accesses based on a random number generator and accesses based on the result of the previous access.

For rewired memory, we work on a flat array `a` that can be accessed directly at index i via `a[i]`. In the case of software-indirected memory, we have a directory `a` where a chunk has the size of a huge page and translate each access at index i to to `a[i / pageSize][i % pageSize]`.

For the patters of Figures 5.7(a) to 5.7(d), the number of accesses equals 100% of the data. For the pattern of Figure 5.7(e), the number of independent accesses equals 20% of the data, where each independent access is followed by four dependent ones.

Table 5.3 shows the results for all five patterns. We can see that as expected, the direct access offered by rewired memory is in any case faster than going through the indirection. However, the amount of difference depends on the type of access pattern. The highest difference we observe in the case of sequential access with factors of 1.88x for independent and 2.12x for dependent access. In contrast to that, when performing any type of random access, the difference is overall less, ranging from 1.07x for independent access to 1.53x for mixed access. This is due to expensive cache-misses triggered by the random accesses both for the direct and the indirect cases, that overshadow the impact of the access pattern. The overall message is: software-indirection might be a valid alternative in terms of flexibility, however, it certainly has a negative impact on the performance, depending on the type of access pattern. Thus, a software-indirection should be used with caution and only, if the subsequent access patterns are known. This is in general not the case. In contrast to that, rewiring offers an equal flexibility without any negative impact of the access performance.

Access Pattern	**Software-Ind.** [s]	**Rewired** [s]	**Speedup**
5.7(a) random, independent	14.03	13.16	1.07x
5.7(b) sequential, independent	1.58	0.84	1.88x
5.7(c) random, dependent	113.40	106.99	1.05x
5.7(d) sequential, dependent	6.60	3.11	2.12x
5.7(e) random, mixed	58.33	38.19	1.53x

Table 5.3: **Comparison of access patterns** on both software-indirected memory and rewired memory when using huge pages. Both methods map sequentially into a pool of initialized pages.

5.5 Applications

In Section 5.2 we recapped virtual memory, in Section 5.3 we introduced rewiring, and in Section 5.4 we micro-benchmarked it. Now, we are in the position to demonstrate the concept on the basis of concrete applications.

Instead of implementing the technique in a full-fledged database management system, we want to use rewiring inside isolated applications, that represent core components of basically every DBMS. By this, we are able to analyze the individual impact of rewiring more carefully than plugging it in at several layers and places in a complete system at once. Of course, seeing rewiring integrated in a full-fledged system is the final goal of this research. However, it goes far beyond the scope of this chapter and we leave it open for future work, see Section 6.5 for more details.

To showcase rewiring, we start with a rewired vector, that is a natural candidate for our technique as it requires both high flexibility and access performance. We then present the benefit of this data structure exemplary by embedding it into our state-of-the-art partitioning algorithm, leading to a significant improvement.

5.5.1 Rewired Vector

Let us start simple and explore a data structure that is not only present at various layers in database systems, but widely used in basically any kind of software: the vector, a resizable array.

In a DBMS, a vector-like structure is the fundamental component of every storage layer, representing tables or columns that grow and shrink under inserts and deletes. Obviously, it is crucial for the storage-structure to provide high access performance and low update cost at the same time.

Traditionally, there is a tradeoff between the flexibility and adaptiveness of a data structure and its provided access times. For instance, a plain array grants us direct access via a static offset, but has a fixed size. In contrast to that, a linked list can grow and shrink dynamically, but suffers from slower access of individual entries. The widely used vector structure lies between these two extremes. It offers direct access into a consecutive virtual memory area but has the ability to grow and shrink dynamically. Let us see how state-of-the-art implementations offer these properties.

Classical Vectors

The vector is easy to handle for the user as it adjusts its size transparently based on the fillstate (similar to a linked list), but at the same time offers a high access performance as the underlying data is guaranteed to be stored in a consecutive memory region. However, a price needs to be paid if the data structure runs out of internal space and needs to grow. In that case, different *resizing policies* are possible: (1) Allocate a consecutive virtual memory area of twice the size, physically copy all data from the old to the new region, and free the old one. This is the standard policy of `std::vector` from STL of C++. (2) Increase the size of the virtual memory area using the `mremap` system call. (3) Realize the vector as a linked list of memory chunks that are hidden through a software-indirection.

From our current point of view, all these techniques seem suboptimal. Policy (1) leads to performing unnecessary physical copies, (2) is not compatible with a pool of pages, and (3) disallows storing all entries in a single continuous virtual memory area. In contrast to that, our rewired vector avoids all these problems entirely.

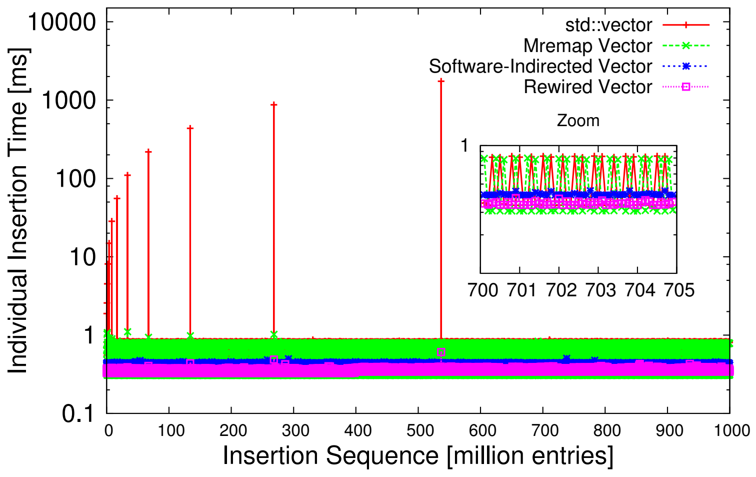
Rewired Vector

For the *rewired vector*, in order to double the capacity of the structure, we do not physically copy any of the data that is already stored in the vector, thus avoiding the issue of policy (1). Instead, we map the first half of a fresh virtual memory area of twice the size to the physical pages containing the old data and the second half to unused physical pages in the pool. The old memory area is unmapped. Of course, the same concept is applied in the opposite direction for shrinking the data structure. This solves both the problems of policy (2) and (3) as we are able to reuse pooled pages freely and provide a single consecutive virtual memory area storing all the data at any time. The simplicity shows how natural rewiring fits to the problem: we map and unmap physical pages on demand without giving up the conveniences of direct memory access. All this is possible with less than 40 lines of code to setup a fresh rewired vector and around 20 lines for the insert function.

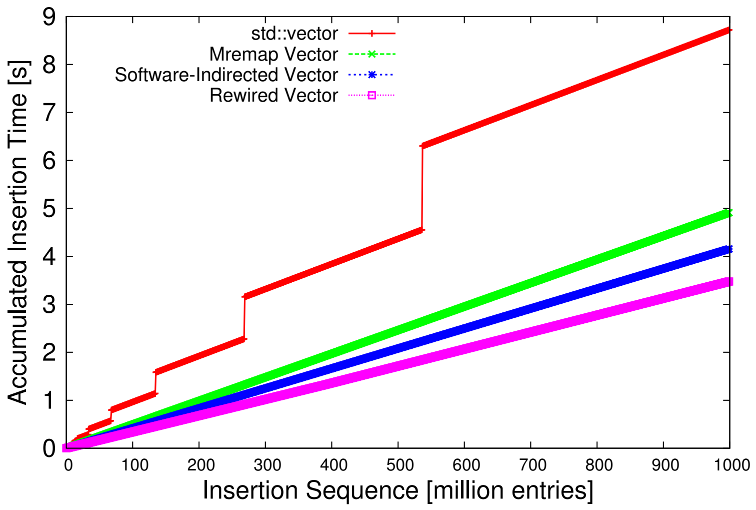
Experimental Evaluation

To evaluate the structure, we compare the rewired vector with representatives of the resizing policies (1), (2), and (3). The `std::vector` of STL[4] represents

[4]The `std::vector` test is written in C++ and compiled using `g++` version 4.9.2. The result is then linked against our C-codebase compiled using `gcc`. For the experiment inserting n



(a) Individual Insertion Times



(b) Accumulated Insertion Times

Figure 5.8: **Insertion** of 1 billion random elements into a vector with an initial size of 2MB. We compare the `std::vector` of the STL, that physically copies the content into an area of twice the size when the capacity is reached with the mremap vector, that uses `mremap` to grow a private anonymous memory area. Further, we test a software-indirected vector as well as the rewired vector, that both claim pages from a page pool. All methods back their memory with huge pages.

elements, we call the C++ library *exactly once* passing a pointer to an array with the elements to insert to avoid interface overhead.

policy (1), the *mremap vector* represents policy (2), and the *software-indirected vector* represents policy (3).

We create all vectors with an initial capacity of 2MB. All tested methods are backed by huge pages. We then insert 1 billion entries of 8B each into the vectors, leading to a total dataset size of 7.45GB. To see the detailed behavior of the vectors, we measure the time for every batch of $100,000$ consecutive inserts. Figure 5.8(a) shows the times of the individual batches over the entire insertion sequence, alongside with a zoom-in of 5 million inserts. Additionally, Figure 5.8(b) shows the accumulated insertion time over the entire insertion sequence. All structures double their capacity as soon as they run out of space. As expected, `std::vector` suffers from expensive physical copying steps every time it has to double its internal memory. Over the entire sequence, a significant amount of the runtime is spent purely on the reallocation process. The last doubling to 8GB after around 536 million insertions physically copies 4GB of data from the old to the new memory region. The remaining three techniques avoid physical copying and thus do not show any significant runtime spikes. Nevertheless, there are differences in the insertion times observable, when we look at the zoom-in of Figure 5.8(a): the mremap vector as well as `std::vector` suffer from hard page faults whereas the rewired vector only from soft ones. The software-indirected vector also exploits the pool and thus offers high insertion throughput. However, it is slightly throttled by checking if page boundaries are crossed.

Overall, rewired vector can insert the entries in the simplest possible way and thus shows the best accumulated runtime. It improves throughput by 150% over `std::vector` and by 40% over mremap vector. Even over the software-indirected vector, that plays in a lower league as it does not keep the data consecutively, the pure insertion improvement is still 20%.

Let us now see a concrete application of the vector in the database context: the partitioning of a dataset. This algorithm requires the flexibility of enlarging the partitions on the fly. Besides, the further efficient processing of the partitions requires the data to be stored consecutively. Both is offered by our rewired vector.

5.5.2 Rewired Partitioning & Sorting

Partitioning a dataset into k disjoint partitions based on a key is a fundamental task in data processing. It is widely used to divide work among threads or as an intermediate step in sorting [3], and join processing [5]. Despite of its simplicity, several optimizations can be applied to the base radix partitioning algorithm such as software-managed buffers [5, 40, 52], non-temporal streaming stores [5, 40, 52, 11], and clever placing of working variables [5, 40]. In Chapter 4, we thoroughly studied partitioning and the impact of the different techniques. In this section we will take this as a baseline and explore how to create a rewired version of state-of-the-art partitioning, that manages to push the end-to-end performance even further.

Two-pass Partitioning

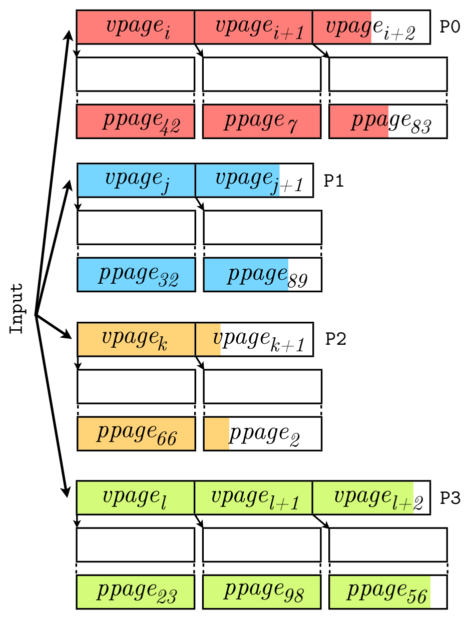
One of the most popular partitioning algorithms [5, 36], as already discussed in Chapter 4, coined *two-pass partitioning* in the following, aims at generating a consecutive and partitioned result area. Let us quickly recap how it works: Initially, we perform a first pass over the data to build a histogram. The histogram counts for each partition the number of entries that belong to it. Based on that, we can compute the start of each partition in the consecutive result area. Afterwards, we perform a second pass over the input data where we physically copy each entry to its destination partition. Many low-level optimizations may be applied to this base algorithm, see Chapter 4. The major drawback of two-pass partitioning is implied by its name already: we need two complete passes over the input to perform the partitioning — costs that can be significant if the input is large and/or the table is wide in row-layout.

Chunked Partitioning

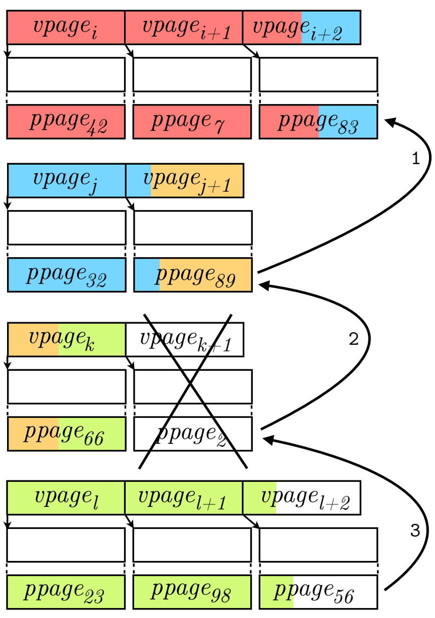
The second option, coined *chunked partitioning* in the following, partitions the dataset in a single pass without creating a histogram in the first place. Notice that chunked partitioning was also used in very recent work [40, 36]. Instead of computing a histogram upfront to determine the exact partition sizes, we organize each partition as a list of chunks (respectively pages) that we create on demand. Each write to a chunk is preceded by a check for sufficient space in the current chunk. While chunked partitioning indeed partitions the input data in a single pass, it has a severe drawback — it does not create a consecutive result area, but only a list of chunks belonging to that partition. That list may lead to additional costs in further processing, as we will see in the evaluation.

Mremap Partitioning

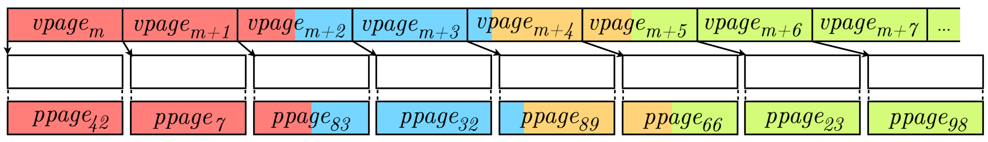
A third option that is positioned between the previously mentioned two approaches uses the `mremap` system call again, thus, we coin it *mremap partitioning*. Like


Input
P0
P1
P2
P3
$vpage_i$ $vpage_{i+1}$ $vpage_{i+2}$
$ppage_{42}$ $ppage_7$ $ppage_{83}$
$vpage_j$ $vpage_{j+1}$
$ppage_{32}$ $ppage_{89}$
$vpage_k$ $vpage_{k+1}$
$ppage_{66}$ $ppage_2$
$vpage_l$ $vpage_{l+1}$ $vpage_{l+2}$
$ppage_{23}$ $ppage_{98}$ $ppage_{56}$

(a) Partitioning.


1
2
3
$vpage_i$ $vpage_{i+1}$ $vpage_{i+2}$
$ppage_{42}$ $ppage_7$ $ppage_{83}$
$vpage_j$ $vpage_{j+1}$
$ppage_{32}$ $ppage_{89}$
$vpage_k$ $vpage_{k+1}$
$ppage_{66}$ $ppage_2$
$vpage_l$ $vpage_{l+1}$ $vpage_{l+2}$
$ppage_{23}$ $ppage_{98}$ $ppage_{56}$

(b) Complete pages via `memcpy`.


$vpage_m$ $vpage_{m+1}$ $vpage_{m+2}$ $vpage_{m+3}$ $vpage_{m+4}$ $vpage_{m+5}$ $vpage_{m+6}$ $vpage_{m+7}$ ...
$ppage_{42}$ $ppage_7$ $ppage_{83}$ $ppage_{32}$ $ppage_{89}$ $ppage_{66}$ $ppage_{23}$ $ppage_{98}$

(c) Rewire consecutively.

Figure 5.9: **Rewired partitioning using chunks** partitions a dataset into a consecutive array **without a histogram generation pass**.

chunked partitioning, it avoids the generation of a histogram pass. However, instead of forming the partitions out of manually linked chunks, we increase the size of the partitions adaptively page-wise by calling `mremap`. On one hand, this has the advantage that every partition is represented by a consecutive virtual memory area. On the other hand, it disables the ability of pooling. Furthermore, it is not possible to create a *single* consecutive virtual memory area over all partitions.

Rewired Partitioning

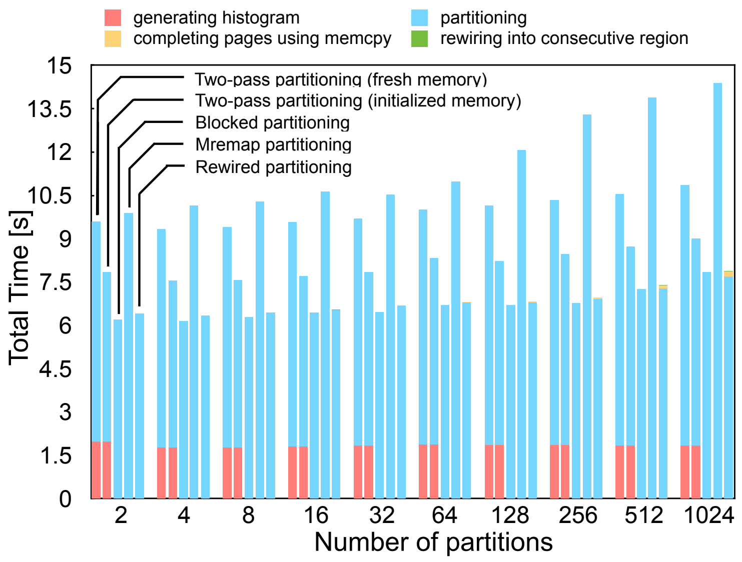
To overcome all these limitations, in the following, we propose an algorithm that combines the best properties of the previously mentioned techniques, while avoiding the disadvantages. We skip the generation of a histogram and thus of a complete pass while at the same time generating a perfectly consecutive result area.

We use chunked partitioning as the basis and modify it in a sense that each individual partition is represented using a rewired vector. In the partitioning phase, visualized in Figure 5.9(a), we add an entry to a partition by pushing it into the corresponding rewired vector. Each vector is configured with an initial capacity of a single page and increases its capacity not by doubling, as evaluated in Section 5.5.1, but by appending individual pages gathered from the pool. This ensures that the overhead in memory consumption remains small. Subsequent to the partitioning phase, we want to establish a single consecutive memory region containing the data of all partitions without any holes or gaps. This is done in two steps. First we have to fill up the last page of all partitions except the one of the very last partition. If the last page of the vector representing partition i has space left for k tuples, then we move k tuples from the end of the vector representing partition $i+1$ to vector i, if it contains a sufficient amount of entries. Figure 5.9(b) demonstrates the concept. The application of this algorithm might clear pages entirely, as it can be seen in the example for $vpage_{k+1}$. In the second step, we are now able to rewire the remaining physical pages backing the rewired vectors into a single consecutive virtual memory area, as we show in Figure 5.9(c).

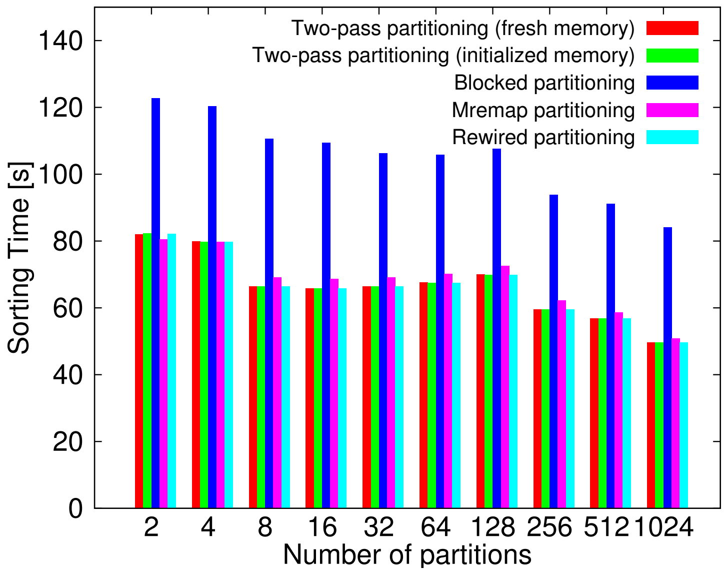
Experimental Evaluation

Let us now see how our new rewired partitioning performs in comparison to the traditional counterparts. We vary the number of requested partitions from 2 to 1024 in logarithmic steps and partition 1 billion entries of 8B each. The keys are uniformly and randomly distributed. We test two-pass partitioning generating a histogram both when partitioning into fresh and initialized memory. Furthermore, we evaluate classical chunked partitioning using a linked list as well as `mremap` partitioning, enlarging the partitions using the system call. We compare these baselines agains our rewired partitioning.

In Figure 5.10(a), we display the end-to-end partitioning times achieved by the different algorithms. To get a more detailed view of the behavior, we additionally show the individual parts of the processing. As we can see, rewired partitioning, that outputs the result in form of a single consecutive memory area, offers basically the same runtime as chunked partitioning, that creates only a linked list of memory chunks. Rewired partitioning is significantly faster than two-pass partitioning due to the avoidance of the additional histogram generation. In comparison with two-pass partitioning into fresh memory, the improvement in throughput ranges from 49% for 2 partitions to 37% for 1024 partitions. Even when making the assumption of partitioning into initialized memory, the improvement still ranges from 22% to 14% for 2 respectively 1024 partitions.



(a) **Time to partition** the input.



(b) **Time to sort** the partitions locally.

Figure 5.10: **Partitioning** an array of 1 billion entries out of place into 2 to 1024 partitions, which are then **sorted** in place. We divide the total partitioning time into generating the histogram, partitioning, completing the pages using `memcpy`, and rewiring into a consecutive region.

Interestingly, `mremap` partitioning is the slowest of all methods. In comparison, rewired partitioning offers an up to 83% higher throughput (1024 partitions). The relocation of virtual memory regions, that is performed in case an enlargement at the current place is not possible, turns out to be surprisingly expensive. For rewired partitioning, physically completing the last pages of the vectors and the subsequent rewiring into a consecutive area cause negligible costs. Even for 1024 partitions, where the last page of each partition is filled around 75%, this cost makes only 2.5% of the end-to-end time.

The alerted reader might argue now that chunked partitioning is still a valid alternative to the rewired version. As we will see now, this depends heavily on how the produced partitions are further processed and whether this processing can be made aware of the list of chunks. Imaging the very common use-case of locally sorting the individual partitions to establish a globally sorted state using your favorite sorting algorithm. It is straight-forward to apply it onto the result of rewired partition, but it certainly has to be modified to apply it onto the chunked partitions. This can be (1) impossible if the algorithm is black-boxed or (2) very hard, depending on the type of algorithm. Extending the algorithm with a chunk-wise indirection is at least always possible if the code is accessible. However, this can have a tremendously negative effect, as demonstrated in Figure 5.10(b). The sorting time of chunked partitioning, working through the indirection, is significantly higher in all cases than the direct approach.

Overall, only rewired partitioning manages to truly combine the best of both worlds: *flexibility* and *processing speed*.

5.6 Conclusion

We have presented the basic toolbox to bridge the gap between the duality of *flexibility* and *access performance*. We reinterpreted the usage of memory mapped files and shared memory to introduce a convenient handle for physical memory in user-space. We evaluated the properties of the technique in depth under micro-benchmarks to learn the strengths and pitfalls and integrate the technique in a set of real-world applications. We showcased how easily rewiring improves the insertion throughput of a consecutive vector by 40% to 150%. By integrating the technique into state-of-the-art partitioning algorithms, we managed to improve the end-to-end throughput by 37% to almost 50%, while still offering a consecutive result. Overall, we managed to significantly improve these applications solely by adapting their memory management, not their algorithmic nature.

Chapter 6

Final Discussion

6.1 Adaptive Indexing: A True Alternative?

In this thesis, we have discussed fundamental techniques of data processing. We started at the algorithmic level and performed an in-depth analysis of the various forms of adaptive indexing methods. We inspected the single-threaded as well as the multi-threaded perspective on the topic, learned pros and cons of the methods and positioned them with respect to traditional indexing structures. To the best of our knowledge, our survey on adaptive indexing covered basically all techniques that are known at this point of time. Thus, Chapters 2 and 3 can be considered as the most comprehensive work on adaptive indexing so far. Apart from evaluating and analyzing the algorithms, our aim was to enhance the state-of-the-art wherever possible. We achieved this by proposing techniques, that improve over existing methods in terms of convergence speed, variance, and tuple reconstruction performance. Further, we introduced a set of adaptive indexing methods supporting multi-threading, a topic that was mostly untouched so far.

An important aim of this work was also to *critically* review adaptive indexing in comparison to classical indexing approaches. As most of the previous analysis of adaptive indexing techniques was done by the researchers that originally proposed them, we felt the need for a neutral perspective on the topic. At any time, we avoided favoring adaptive over classical indexing and put the methods to test under circumstances, that were challenging for them.

After all these investigations, it is time to come to a final résumé. Should a database management systems designer consider using adaptive indexing methods over traditional ones? As we have seen, the answer highly depends on the requirements of the user as well as the workload and system circumstances. Obviously, adaptive indexing can not simply replace traditional indexing. At a first glance at the topic, one could think of adaptive indexing as a successor to classical up-front indexing, as it avoids the costly initialization phase and adapts to the workload requirements. However, our results showed that the situation is not that trivial. It all depends on the knowledge about the workload: if we know that we frequently select on a certain attribute, a traditional index is the way to go as it offers stable response times and besides enables interesting orders on the column. Without any information about whether a column is accessed frequently or not, it is worth considering an adaptive index. In this sense, instead of considering adaptive indexing as a replacement for traditional indexing, it should be seen as a replacement for full scans.

As we have seen, the available system resources also highly influence any decision between adaptive and traditional indexing. The more computing cores and memory bandwidth available, the less significant becomes the difference between both paradigms. If it is possible to fully sort hundreds of millions of index entries in less than a second using massive parallelism, the question arises whether there is a need for a more lightweight technique at all. In case a highly parallel machine is fully available for sorting, we clearly recommend building a traditional index. It scales better with the available cores besides from simply becoming negligibly cheap. If only a smaller portion of the resources are available, e.g. because of load caused by other queries running in parallel, adaptive indexing remains a lightweight alternative, that offer significantly lower initialization times.

Besides of that, the various different adaptive indexing methods that we have analyzed in this thesis highly focus on improving a single dimension at a time such as convergence speed, variance, robustness, and upfront initialization effort. In most cases, an improvement in one dimension results in a decrease in performance in the other dimensions. This extends the question of *whether* to use adaptive indexing to *which* adaptive index to integrate in a system.

As a result of our investigation, we can say that standard cracking still offers the most lightweight type of adaptive index and serves as the best allrounder. It does not over-emphasize on a single dimension and is simple to integrate in existing access paths. However, it should not be used on an uninitialized column due to its severe weaknesses against sequential access patterns. Instead, it should be combined with a coarse-granular index to highly improve the workload robustness as well as the variance resistance and the accumulated query response time.

In this sense, we come to the conclusion that the most adaptive (respectively laziest) approach, which does the least amount of reorganization to answer a query, is in general too fragile to be used. The first query, that initializes the index by copying the data from the base table to a separate array, should certainly invest time into pre-partitioning the keys. The coarse-granular index positions itself between the purely query-driven adaptive indexing and the data-driven traditional indexing. With the right partitioning implementation, its overhead stays small. But what is the right partitioning implementation?

6.2 Data Partitioning: The Core of Indexing

Surprisingly, there were a large amount of options to analyze in finding the right partitioning implementation. Mostly, these optimizations did not originate from algorithmic changes, but from mapping the algorithm more efficiently to the properties of modern hardware and operating systems. A great example for this were software-managed buffers: introducing them doubles the amount of copied data over the unbuffered version, however, at the same time it significantly improves the runtime for higher numbers of partitions.

Therefore, to create high-performance algorithms, a detailed knowledge of the system characteristics must be available in the design phase already and the algorithm must be tuned specifically to the machine. Even in the simple case of partitioning, numerous low-level components of the system were involved critically in the performance: the cache hierarchy, the virtual memory subsystem of the operating system (including the page table), the TLB caching address translations from virtual to physical pages (composed of multiple levels), the hardware page-walker of the CPU, the size of a page, the size of a cache-line, the prefetcher, and available SIMD instructions. Only with these individual components and their specific sizes and behaviors on a machine at hand, it is possible to push the performance to the limits.

This also means that a particular algorithm can only be judged in conjunction with the machine and operating system it is running on. It might be the case that algorithm A is faster than algorithm B on system X, but slower on system Y. To gather the most value out of our experimental evaluation, for all setups used in this thesis we picked state-of-the-art hardware and operating systems that are present in nowadays data management infrastructures.

6.3 Rewiring Memory

Based on our observations during the analysis of the low-level data partitioning optimizations, we wanted to push the art of system centric optimization to the limit of what is possible on a given operating system. Instead of starting with an algorithm to improve, we came from the opposite direction by investigating what is offered by the system in general. A critical component in terms of performance was the virtual memory subsystem, that is involved in any algorithm or data structure we have seen so far in one way or the other. It is present in all modern end-user operating systems and any memory access is directed through it.

While it is a very convenient concept in general, we often felt being restricted too much by it. Thus, we broke out by reintroducing the concept of physical memory into user-space. We showed how to actively manipulate the mappings from virtual to physical memory at runtime — without making any modifications to the mainstream linux kernel.

This served as the basis for optimizing various existing algorithms and data structures — in this thesis, we showcased the technique at the examples of a vector and data partitioning. In our corresponding conference publication, we also used the technique to improve virtual memory aware snapshotting, as used by HyPer [25] for instance. In neither case, we had to change the algorithm or data structure in its nature. All we did was replacing the way the memory management behaves.

Overall, this work showed another trend in performance optimization: the need to gain more *control* over the system. Of course, the operating system and hardware designers limit the amount of control the individual programmer has to a certain degree — to ease the use and to let the system decide on critical components. As a consequence, the operating system for instance handles transparently the swapping of pages or the scheduling of the running processes. However, to develop performance critical applications, we have to break these limits. In this case, we reintroduced physical memory, that was hidden to the user by default. Of course this slightly complicates the programming but it opens the possibility to map algorithms and data structures more efficiently to the system characteristics — the more aware they are of the underlying system, the faster they become.

6.4 Closing the Circle

On our way through this thesis, we started with adaptive indexing, moved down to data partitioning, and finally down to low-level virtual memory management.

Our view has changed during the discussion from the algorithmic level to a more system-centric perspective. Of course, this raises the question whether the insights gained from this change of perspective can actually flow back to the start of this journey, namely adaptive indexing, and have a positive impact on it. Thus, in the following we will propose an algorithm that closes the circle of our journey by combining the best techniques from all chapters into a single method. We will then compare this algorithm against the state-of-the-art adaptive indexing techniques under different workloads and discuss the improvements.

6.4.1 Rewired Radix Cracking

As the basis, we start with standard cracking as the most lightweight form of adaptive indexing. We have seen before that it works very well on uniform random and skewed key distributions and workloads. We will extend it with the most promising techniques from the previous chapters and see how far we get in comparison to the state-of-the-art techniques. Our aim is to design an algorithm that is both fast over a sequence of queries as well as per individual query. Additionally, it should converge quickly towards a full index and show low variances in response times.

To achieve this, we start with standard cracking and make the following extensions, that originate partially from an algorithmic perspective as well as from a system-centric view:

1. We create a *coarse-granular index in the very first query* to bulk-load the index, as done successfully in Chapter 2 and Chapter 3. This increases the robustness against sequential workloads, reduces the variance, and leads to short response times from the second query on.
2. This initial coarse-granular index should create a *large number of partitions*, but it should still *keep the response time of the first query low.* To achieve this, we realize the coarse-granular index using the *optimized out-of-place radix partitioning* algorithm as discussed in Chapter 4. We activate software-managed buffers, non-temporal streaming stores, and the optimized buffer layout.
3. Further, we reduce the pressure on the very first query by *avoiding the histogram generation pass* of the out-of-place radix partitioning using *page-wise rewiring* as discussed in Chapter 5.
4. We introduce the creation of *coarse-granular indexes also for the remaining queries* to increase the convergence speed. Traditionally, standard cracking locates the two partitions into which the predicates of the range query fall.

Then, it applies two times crack-in-two for the reorganization (assuming the predicates fall into two different partitions). We add one step in between by *radix partitioning the two located partitions before applying the two cracks.* Traditionally, this would require an in-place radix partitioning algorithm, as the reorganized data has to remain in the cracker column. Unfortunately, the in-place radix partitioning algorithm has two drawbacks: Firstly, it suffers from random-accesses due to the cuckoo-style processing nature. Secondly, it is not stable. This leads to incorrect results if we reorganize a partition that contains a split line created by a crack-in-two. Thus, we want to apply the out-of-place radix partitioning algorithm, that is stable and in general also faster than the in-place version. To be able to do this, we have to apply a further modification in the next step.

5. Instead of using in-place radix partitioning, we *use the out-of-place version and rewire the content back into the cracker column.* Thus, we basically hide the out-of-place nature of radix partitioning using rewiring and make it look like an in-place algorithm. Before partitioning a part of the cracker column, we identify the pages onto which the data resides. Then, we allocate the same amount of unused pages from the pool and perform out-of-place radix partitioning from the old to the new pages, respecting the given partition boundaries. Data that lies on the boundary pages but outside the given partition boundaries is manually copied. During the partitioning process, we can safely ignore any existing split lines that have been created by crack-in-two operations, as they remain valid after partitioning due to the stable nature of the algorithm. Subsequently, we rewire the new pages into the cracker column and release the old ones to the pool. Figure 6.1 visualizes the concept.

6. We stop applying the partitioning step if the overhead becomes too large. The overhead is the amount of data we have to copy additionally to fit to the page boundaries (see `memcpy` in Figure 6.1). If the overhead becomes larger than the amount of data to copy during partitioning, we do not apply the partitioning step.

Based on the components assembling our algorithm, we coin it *Rewired Radix Cracking* in the following.

6.4.2 Experimental Evaluation

Let us now see how rewired radix cracking performs in comparison to its ancestor standard cracking. Additionally, we compare it with stochastic cracking (MDD1R)

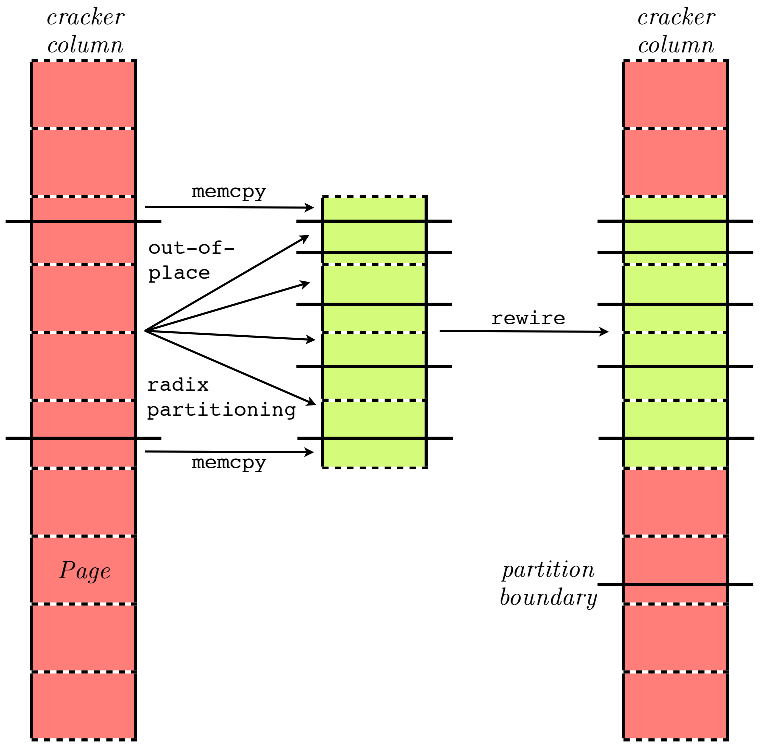
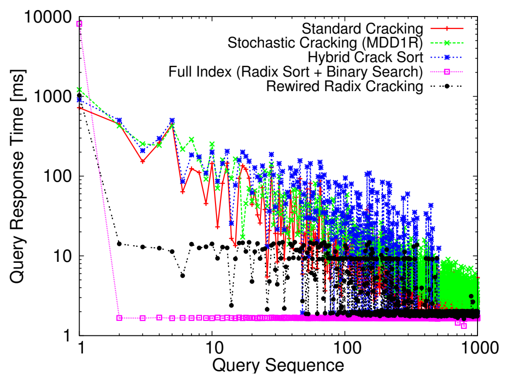


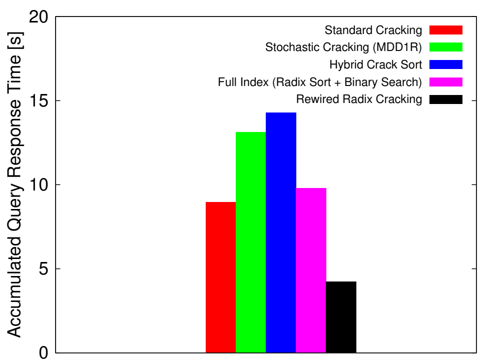
Figure 6.1: Hiding the out-of-place nature of radix partitioning using rewiring. This allows us to use out-of-place radix partitioning on the cracker column in-place.

as well as hybrid crack sort, which are both state-of-the-art techniques tackling the issues variance respectively convergence speed. Further, we compare it with a full index that is realized by sorting the array using radix sort and performing binary search for query answering. We use the usual setup of 100 million entries of 16B key-value pairs where the keys follow a uniform random distribution. The queries are generated following three different patterns: uniform random, sequential, and skewed, according to the patterns used in Chapter 2. We fire 1000 queries in total where each query selects 1% of the data. We perform the experiments on the machine that was used in Chapter 5 and which was described in Section 5.4.1. Rewired radix cracking is configured such that the first partitioning step creates 128 partitions, while the remaining partitioning steps build 32 partitions. This configuration was identified empirically as well performing with respect to a low initialization time as well as a high convergence speed.

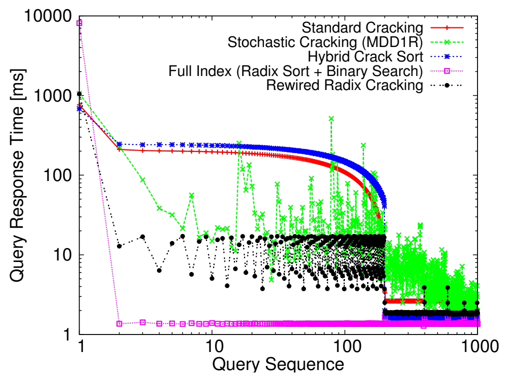
In Figures 6.2(a), 6.2(c), and 6.2(e) we present the individual query response times for all tested methods over the entire sequence of 1000 queries. We can observe that



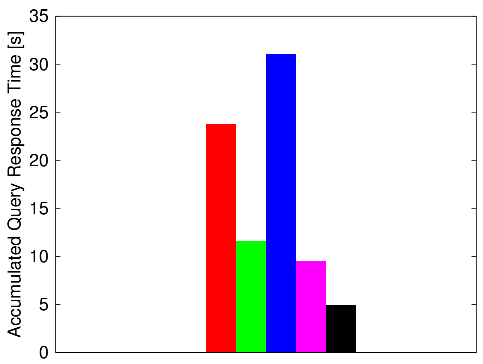
(a) Individual query response times on a random query workload



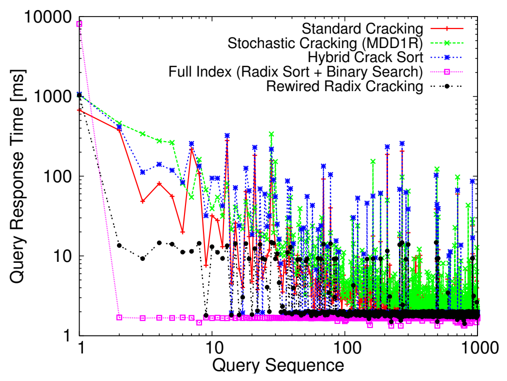
(b) Accumulated query response times on a random query workload.



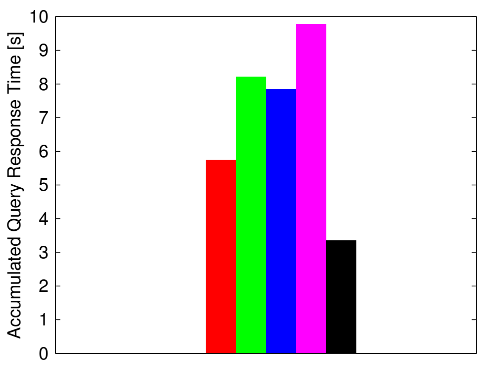
(c) Individual query response times on a sequential query workload



(d) Accumulated query response times on a sequential query workload.



(e) Individual query response times on a skewed query workload



(f) Accumulated query response times on a skewed query workload.

Figure 6.2: **Closing the Circle** — With the optimizations discussed throughout this thesis, we are able to go back to the start of the investigation and propose a significantly improved adaptive indexing technique. We exploit the pleasant properties offered by the coarse-granular indexing approach discussed in Chapter 2 respectively Chapter 3 and design an entire adaptive indexing algorithm around it, that is driven by radix partitioning. Then, we optimize the radix partitioning step to the limits by applying the system-centric optimizations of Chapter 4 and the rewiring capabilities of Chapter 5.

rewired radix cracking performs considerably better under all workloads than the baselines in several regards. For any workload, its response times drop to less than 20ms from the second query on, while standard, stochastic, and hybrid cracking show significantly higher response times even after more than 100 queries. Thus, we can clearly see the drastically faster convergence towards the performance of a full index. Further, the variance is highly reduced in the case of rewired radix cracking. Instead of seeing a mixture of low and high response times, we can clearly identify three levels depending on how deep the partitioning recursion went already. Level 1 around 1000ms is seen only for the very first query, level 2 is around 10ms to 20ms when hitting the initial partitions, and level 3 is around 1ms when hitting data that has undergone two rounds of partitioning already. Besides of that, we managed to keep the overhead of the expensive initial partitioning step fairly low in comparison. With all the partitioning optimizations activated, the first query is only 300ms slower than the most lightweight one, which is performed by standard cracking.

Figures 6.2(b), 6.2(d), and 6.2(f) show the accumulated query response times over the query sequence. As we can see, rewired radix cracking is able to answer the queries around two times faster than the best baseline for each workload. As we can see, each of the baselines is vulnerable to certain workloads: stochastic cracking performs poorly under a random pattern, standard cracking under a sequential pattern, full index under a skewed pattern, and hybrid cracking trades convergence speed with accumulated response time — only rewired radix cracking performs stable and fast under all patterns.

Overall, with the proposal of this algorithm, we managed to close the circle of optimization, as depicted in Figure 6.3. As we have seen, to successfully enhance an algorithm, it is necessary to move back and forth between the algorithmic and the system-centric view. After introducing the coarse-granular index as a pre-partitioning step on the algorithmic level and enhancing it using techniques close to the system, we came back to the algorithmic level by introducing the enhanced partitioning also to intermediate queries. Thus, as a final résumé we want to emphasize that both views must be seen together at any time and an improvement on one side might trigger an advancement on the other side as well.

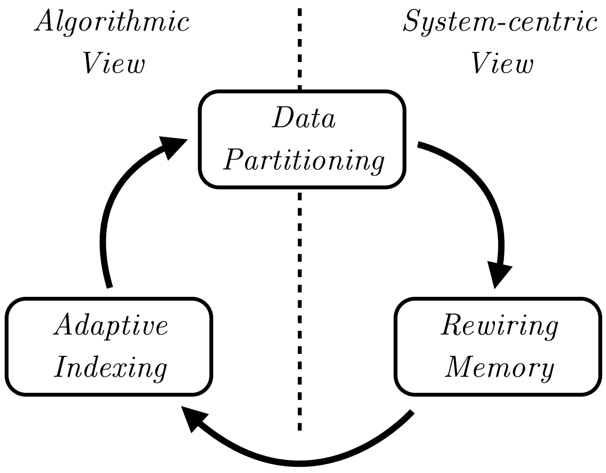


Figure 6.3: The Circle of Optimization.

6.5 Future Work

The work of this thesis opens the way for several research projects, that can be considered as follow-up work to topics previously discussed. In the following, we want to outline this future work.

1. **Simplifying adaptive indexing.** After investigating this vast amount of adaptive indexing algorithms that are present nowadays, the question arises whether all of them are actually necessary. Most of the methods where designed to tackle a specific problem, e.g. convergence speed. However, often, the solution to one problem introduced or enforced another one. Besides, a system that wants to apply adaptive indexing has to be equipped with various different implementations of the concept to handle the different incoming workloads. Based on the observation that all database cracking algorithms share the single same component, namely simple data partitioning, we are currently working on a *single* adaptive indexing algorithm, that can adapt to the shape of the existing algorithms, based on the workload and the circumstances. This results in an *adaptive adaptive indexing* algorithm. Rewired radix cracking, that was presented in the previous section, is a first step into this direction.

2. **Changing the perspective of adaptive indexing.** There is another interesting commonness among the seen adaptive indexing algorithms — they come to their reorganization decision solely based on the *current* query. However, if a stream of queries arrives at a high frequency, it can make sense to inspect a *window* of queries before actually making a decision. This allows to reorder, modify, or even drop reorganizations. By introducing a window, we could create a trade-off between individual query response time and knowledge available for reorganization decisions.

3. **Introducing RUMA into a full-fledged system.** We have seen some fundamental applications of rewiring. To demonstrate the impact of our technique, we picked the vector structure as well as the partitioning and sorting problem. All of these are fundamental use-cases both in database systems as well as generally in computer science. Further, we showed the impact on fork-based snapshotting, a naturally candidate to be enhanced by rewiring, in the corresponding publication [47]. Of course, there are several other promising applications where we believe that rewiring could be applied successfully. In future work, an entire DBMS can be build around or adapted to rewiring memory from the ground. Let us discuss these components and how they could be assembled in a single system.

On the storage layer, we can realize the managed tables and columns purely with rewired memory, as seen in the vector application. Thus, **inserts** and **deletes** are as cheap as possible to handle. Updates to a memory region can be collected on separate pages in a COW style and rewired into the base table from time to time (**differential updates**). The memory is claimed and released in a managed **pool**. Further, multiple different **virtual views** on the same physical data are possible. For instance, a view selecting rows on a table can directly map to only those physical pages, that contain qualifying rows. Connected to this, classical **data duplication** can be applied exploiting rewiring. If data is repeated, it is possible to store it physically only once and map to it from multiple places. Thus, it becomes possible to realize the inverse operation of COW: merge-on-write (MOW)[1]. With respect to multi-socket machines, we can use rewiring to control the distribution of the memory across the difference memory regions for **NUMA-awareness**. Rewiring could be exploited to transparently swap NUMA-remote with NUMA-local pages, while the thread is working on the corresponding virtual memory area.

On the index layer, rewiring can be exploited at two places: firstly, there is a tremendous amount of read-optimized or even **read-only** index structures as for instance the CSS-tree [41] or FAST-tree [27]. Often, these structures are very hard to update. Massive physical copying or even a rebuild of the entire structure is necessary. Rewiring offers the chance to cheaply swap in and out parts of the structure in a transparent fashion. Secondly, rewiring can be used to speed up lookups in tree-based index structures by removing one level of indirection. It can also be used to enhance increasingly popular hashing methods [44]. For instance, the classical **extendible hashing** [12] uses a directory to indirect lookups into its buckets; this directory may be realized directly in form of the page table which completely removes the costs for that lookup.

Besides of that, the **snapshotting** can realized using rewiring to enable fast concurrent processing of OLAP and OLTP queries. The seen rewired partitioning can be applied at many places, for instance to divide the data for **join-processing** [5, 46] or to split it into **horizontal partitions**. Obviously, rewiring can play a role at various places of a data management systems. Due to the enormous complexity, integrating these concepts in a full-fledged system will be a future project.

[1]`https://youtu.be/sLl-kyv-DCo`

List of Figures

List of Tables

Bibliography

[1] Generalized Heap Impl. https://github.com/valyala/gheap.

[2] Georgy Adelson-Velsky et al. An algorithm for the organization of information. In *USSR Academy of Sciences*, pages 263–266, 1962.

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

[4] Cagri Balkesen. `http://www.systems.ethz.ch/sites/default/files/file/sort-merge-joins-1_4_tar.gz`, January 2015.

[5] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M. Tamer Özsu. Multi-core, main-memory joins: Sort vs. hash revisited. *PVLDB*, 7(1):85–96, 2013.

[6] Rudolf Bayer and Edward M. McCreight. Organization and maintenance of large ordered indices. *Acta Inf.*, 1:173–189, 1972.

[7] Olaf René Birkeland. *Searching Large Data Volumes with MISD Processing.* PhD thesis, 2008.

[8] Peter A. Boncz, Stefan Manegold, and Martin L. Kersten. Database architecture optimized for the new bottleneck: Memory access. In *PVLDB*, pages 54–65, 1999.

[9] David J. DeWitt, Jeffrey F. Naughton, et al. Practical skew handling in parallel joins. In *VLDB, 1992, Proceedings.*, pages 27–40.

[10] Niall Douglas. User mode memory page allocation: A silver bullet for memory allocation? *CoRR*, abs/1105.1811, 2011.

[11] Ulrich Drepper. What every programmer should know about memory, 2007.

[12] Ronald Fagin et al. Extendible hashing - A fast access method for dynamic files. *ACM Trans. Database Syst.*, 4(3):315–344, 1979.

[13] Tony Finch. Incremental calculation of weighted mean and variance. University of Cambridge Computing Service, 2009.

[14] Goetz Graefe, Felix Halim, Stratos Idreos, et al. Concurrency Control for Adaptive Indexing. In *PVLDB*, volume 5, pages 656–667, 2012.

[15] Goetz Graefe, Felix Halim, Stratos Idreos, et al. Transactional support for adaptive indexing. *VLDB J.*, 23(2):303–328, 2014.

[16] Goetz Graefe and Harumi Kuno. Self-selecting, Self-tuning, Incrementally Optimized Indexes. In *EDBT*, pages 371–381, 2010.

[17] Felix Halim, Stratos Idreos, et al. Stochastic Database Cracking: Towards Robust Adaptive Indexing in Main-Memory Column-Stores. In *PVLDB*, volume 5, pages 502–513, 2012.

[18] Paul Hildebrandt and Harold Isbitz. Radix Exchange - An Internal Sorting Method for Digital Computers. *J. ACM*, 1959.

[19] C. A. R. Hoare. Quicksort. *Commun. ACM*, 4(7):321–, July 1961.

[20] Stratos Idreos et al. Database Cracking. In *CIDR*, pages 68–78, 2007.

[21] Stratos Idreos, Martin Kersten, and Stefan Manegold. Updating a Cracked Database. In *SIGMOD*, pages 413–424, 2007.

[22] Stratos Idreos, Martin Kersten, and Stefan Manegold. Self-organizing Tuple Reconstruction In Column-stores. In *SIGMOD*, pages 297–308, 2009.

[23] Stratos Idreos, Stefan Manegold, et al. Merging What's Cracked, Cracking What's Merged. In *PVLDB*, volume 4, pages 586–597, 2011.

[24] Mark S. Johnstone and Paul R. Wilson. The memory fragmentation problem: Solved? In Simon L. Peyton Jones and Richard E. Jones, editors, *International Symposium on Memory Management, ISMM '98, Vancouver, British Columbia, Canada, 17-19 October, 1998, Conference Proceedings*, pages 26–36. ACM, 1998.

[25] A. Kemper and T. Neumann. Hyper: A hybrid oltp & olap main memory database system based on virtual memory snapshots. In *ICDE*, pages 195–206, April 2011.

[26] Martin Kersten et al. Cracking the Database Store. In *CIDR*, pages 213–224, 2005.

[27] Changkyu Kim, Jatin Chhugani, Nadathur Satish, et al. FAST: fast architecture sensitive tree search on modern cpus and gpus. In *SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10*, pages 339–350, 2010.

[28] Changkyu Kim et al. FAST: Fast Architecture Sensitive Tree Search on Modern CPUs and GPUs. In *SIGMOD*, pages 339–350, 2010.

[29] Thomas Kissinger, Benjamin Schlegel, Dirk Habich, and Wolfgang Lehner. *KISS-Tree*: smart latch-free in-memory indexing on modern architectures. In *DaMoN, Scottsdale, AZ, USA, May 21, 2012*, pages 16–23, 2012.

[30] Shin-Jae Lee, Minsoo Jeon, Dongseung Kim, and Andrew Sohn. Partitioned parallel radix sort. *J. Parallel Distrib. Comput.*, 62(4):656–668, 2002.

[31] Viktor Leis et al. The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases. In *ICDE*, pages 38–49, 2013.

[32] Xavier Martinez-Palau, David Dominguez-Sal, et al. Two-way Replacement Selection. In *PVLDB*, volume 3, pages 871–881, 2010.

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

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

[35] John D. McCalpin. STREAM benchmark, version from January 17, 2013. https://www.cs.virginia.edu/stream/FTP/Code/stream.c.

[36] Ingo Müller, Peter Sanders, Arnaud Lacurie, Wolfgang Lehner, and Franz Färber. Cache-efficient aggregation: Hashing is sorting. In *SIGMOD*, pages 1123–1136, 2015.

[37] Rasmus Pagh and FlemmingFriche Rodler. Cuckoo hashing. In *ESA 2001*, volume 2161, pages 121–133. Springer Berlin Heidelberg, 2001.

[38] Eleni Petraki, Stratos Idreos, and Stefan Manegold. Holistic indexing in main-memory column-stores. In *SIGMOD 2015*, pages 1153–1166.

[39] Holger Pirk, Eleni Petraki, Stratos Idreos, Stefan Manegold, and Martin L. Kersten. Database cracking: fancy scan, not poor man's sort! In *DaMoN, Snowbird, UT, USA*, pages 4:1–4:8, 2014.

[40] Orestis Polychroniou and Kenneth A. Ross. A comprehensive study of main-memory partitioning and its application to large-scale comparison- and radix-sort. In Curtis E. Dyreson, Feifei Li, and M. Tamer Özsu, editors, *SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 755–766. ACM, 2014.

[41] Jun Rao and Kenneth A. Ross. Cache conscious indexing for decision-support in main memory. In *VLDB'99*, pages 78–89, 1999.

[42] Jun Rao and Kenneth A. Ross. Making B+-Trees Cache Conscious in Main Memory. In *SIGMOD*, pages 475–486, 2000.

[43] Layali K. Rashid, Wessam Hassanein, and Moustafa A. Hammad. Analyzing and enhancing the parallel sort operation on multithreaded architectures. *The Journal of Supercomputing*, 53(2):293–312, 2010.

[44] Stefan Richter, Victor Alvarez, and Jens Dittrich. A seven-dimensional analysis of hashing methods and its implications on query processing. *PVLDB*, 9(3):96–107, 2015.

[45] Nadathur Satish, Changkyu Kim, Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, Daehyun Kim, and Pradeep Dubey. Fast sort on cpus and gpus: a case for bandwidth oblivious SIMD sort. In *SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10*, pages 351–362, 2010.

[46] Stefan Schuh, Xiao Chen, and Jens Dittrich. An Experimental Comparison of Thirteen Relational Equi-Joins in Main Memory. In *SIGMOD 2016*.

[47] Felix Martin Schuhknecht, Jens Dittrich, and Ankur Sharma. RUMA has it: Rewired user-space memory access is possible! *PVLDB*, 9(10):768–779, 2016.

[48] Felix Martin Schuhknecht, Alekh Jindal, and Jens Dittrich. The Uncracked Pieces in Database Cracking. In *PVLDB*, volume 7, pages 97–108, 2013.

[49] Felix Martin Schuhknecht, Alekh Jindal, and Jens Dittrich. An experimental evaluation and analysis of database cracking. *VLDB J.*, 25(1):27–52, 2016.

[50] Felix Martin Schuhknecht, Pankaj Khanchandani, and Jens Dittrich. On the surprising difficulty of simple things: the case of radix partitioning. In *PVLDB*, volume 8, pages 934–937, 2015.

[51] Michael Stonebraker. The Case for Partial Indexes. *SIGMOD Record*, 18(4), 1989.

[52] Jan Wassenberg and Peter Sanders. Engineering a multi-core radix sort. In *Euro-Par, Bordeaux, France, August 29 - September 2*, pages 160–169, 2011.