Database cracking: towards auto-tunning database kernels

Ydraios, E.

Citation for published version (APA):
Chapter 7

Adaptive Indexing Hybrids*

7.1 Introduction

Inspired by database cracking, researchers from HP Labs, Goetz Graefe and Harumi Kuno, have recently invented *adaptive merging* (Graefe and Kuno, 2010a; Graefe and Kuno, 2010b). They adopt the continuous low-level reorganization and incremental index building ideas from cracking and the motivation is mainly adaptation with block-access devices such as disks, in addition to main memory. The principal goal for designing adaptive merging is to reduce the number of queries required to converge to a fully-optimized index, and the principal mechanism is to employ variable memory and CPU effort in each query. It is designed to fully exploit the time spent in I/O, and thus performs an early partial sort optimized for block-access devices. However, this comes at the cost of additional CPU overhead per query that becomes significant when input fits within memory. Partial results are then incrementally and adaptively merged in the target index as part of query processing. Contrary to cracking, adaptive merging focuses on providing a very fast adaptation process reaching quickly the final optimized stage.

*The material in this chapter has been the basis for a paper submitted for publication entitled “Adaptive Indexing” (Idreos et al., 2010c).
7.1.1 Contributions

In this chapter, we provide the first detailed comparison between adaptive merging and database cracking based on a complete implementation in a full kernel. We study the various trends and tradeoffs that occur and we propose a new hybrid adaptive indexing approach designed for lower overhead per query. By adopting from database cracking the design goal of minimizing per query overhead while at the same time exploiting the concept of runs and merging from adaptive merging, it combines benefits of both approaches. The net effect is that the new algorithm achieves such a light-weight adaptation that reflects a zero overhead over a full scan approach, essentially opening the road towards tuning-free systems that always adapt as the default action.

7.1.2 Outline

The rest of the chapter is organized as follows. Section 7.2 provides the necessary background describing in more detail adaptive merging. Section 7.3 then presents the hybrid algorithm and Section 7.4 provides a detailed experimental analysis. Finally, Section 7.5 concludes the paper.

7.2 Adaptive Merging

While database cracking functions as an incremental quicksort, with each query resulting in at most one sort step, adaptive merging functions as an incremental external merge sort. Under adaptive merging, the first query to use a given column in a predicate produces sorted runs and each subsequent query upon that same column performs at most one additional merge step. Each merge step only affects those key ranges that are relevant to actual queries, leaving records in all other key ranges in their initial places. This merge logic takes place as a side effect of query execution performed purely within query operators as in database cracking.

Adaptive merging enables control over the amount of CPU and memory invested into index refinement; the more resources invested, the fewer queries needed for index conversion. Judicious memory allocation can thus control run size, comparison count per record within each query, and thus overall CPU effort. However, the CPU effort required in order to achieve initial runs of substantial size is significantly higher than that required by database cracking for processing any query. That is to say, run generation imposes a substantial penalty in terms of CPU effort on this first query, although given today’s CPUs,
the principal cost may be in movement in the memory hierarchy, e.g., disk I/O or cache faults.

7.2.1 Motivation for Hybrid Designs

One concern about database cracking is that at most two new partition boundaries per query means that the technique requires thousands of queries to converge on an index for the focus range. One concern about adaptive merging is that the technique requires the first query to pay a significant cost for generating initial runs. The difference in reorganization performance, i.e., the number of queries required to have a key range fully optimized, is due to merging with a high fan-in rather than partitioning with a low fan-out of two or three and to merging a query’s entire key range rather than only dividing the two partitions with the query’s boundary keys. The difference in the cost of the first query is primarily due to the cost of sorting the initial runs.

7.3 The Hybrid Algorithm

Our goal in creating a hybrid algorithm is to “merge” the best qualities of both adaptive merging and database cracking. In particular, we strive to maintain the lightweight footprint of cracking, which imposes a minimal overhead to queries, and to combine it with adaptive merging’s approach of creating and then merging runs, which exploits large memory and ample CPU for fast convergence and enables continuous control of memory and CPU investments.

Data Structures

Before presenting this hybrid algorithm, we first describe the underlying data structures used in the implementation. Each logical column in our model is represented by a pair of parallel arrays that contain row identifiers and key values. Two types of data structures physically organize these pairs of arrays: one partitions them by row identifier then cracks each partition by value and one represents merged ranges of key values. The former represents initial “unsorted” data, and the latter, a partial index over queried ranges of values. These are reminiscent of adaptive merging’s runs and final merge partition except that both the ID and value partitions are initially not sorted on keys. As queries are processed, both types of partitions are “cracked” upon the query’s keys. Each ID partition uses a table of contents (shown as a small triangle in Figure 7.1)
to keep track of the boundary keys it contains. Finally, a single master table of contents — the adaptive index itself — keeps track of the both id and value partitions (shown as a large triangle in Figure 7.1). It is this adaptive index that is updated as a result of processing queries from a workload.

7.3.1 Algorithm

Assume an input column $C$ and a sequence of range selections over $C$. We distinguish between the first query and the rest of the queries in the sequence. The first query creates and populates initial data structures. Every query thereafter only performs merging and cracking actions, increasingly refining the adaptive index.

The First Query

The upper third of Figure 7.1 sketches, from left to right, the processing of a first query requesting values between a low bound $L_1$ and a high bound $H_1$ from $C$. This is as follows.

- Column values extracted from the main table are the input to the algorithm; at this point, row identifiers are implicit rather than explicitly represented in data structures.

- Column values are physically partitioned by copying values of $C$ into ID partitions of size $p$, and an explicit row identifier is associated with each value. At this point, these row ID/value pairs are partitioned only by row ID.

- Each ID partition is physically reorganized via cracking, i.e., (sub-partitioned by value). Tuples with values lower than the queried range are moved to the beginning of the ID partition; tuples that fall into the range of interest are moved in the middle; while those with bigger values are moved to the end of the partition.

- Qualifying values are sequentially moved (along with the respective IDs) from each ID partition to a new value partition created for this query. The relevant ID partitions’ table of contents, and also the adaptive index are both updated if necessary.

In this way, each initial ID partition is cracked in place, building a table of contents over each partition to guide future accesses. In Figure 7.1, the dotted
lines in each partition reflect the information stored in the respective index regarding value positions. The adaptive index maintains information about the relevant value ranges, e.g., for each value range whether these values are already merged into the adaptive index and, if so, which value partition holds these values in which positions.

**Rest of the Query Sequence**

Every query thereafter is required only to leave the adaptive index with all qualifying values merged and in a “logically contiguous” area.

For example, the middle third of Figure 7.1 shows, from left to right, the processing of a second query requesting values between a low value bound \( L_2 \) and a high value bound \( H_2 \) from \( C \): The algorithm starts with two searches on the adaptive index to locate the areas where low and high bounds fall. We can distinguish between a number of cases.

One scenario is that both bounds fall within a key range that has already been fully merged into a value partition. This happens, for example, when a previous query has already requested an overlapping value range, as shown in Query 2 of Figure 7.1. This query needs all values between \( L_2 \) and \( H_2 \) (marked with bold lines in the value range area). In this case, all requested values are already in the adaptive index and thus no merging actions are required. The algorithm needs to ensure that all qualifying values are in a contiguous area. To do so, it cracks the value partition that holds these values, reorganizing it in place such that values between \( L_2 \) and \( H_2 \) are stored continuously. The hybrid algorithm uses the adaptive index to determine where in the chunk it should crack and then registers any cracking actions back to the index such that future accesses to this chunk can exploit the refined knowledge.

If both bounds fall in a single area not yet merged, the hybrid algorithm brings all necessary values from the initial partitions. Such an example is shown in Query 3 of Figure 7.1. It searches the table of contents of each ID partition \( P_i \) to find which pieces of \( P_i \) should be cracked for \( L_3 \) and \( H_3 \). After this is done, all values between \( L_3 \) and \( H_3 \) are in a contiguous area in \( P_i \). These values are then copied into a new value partition. After all qualifying ID partitions have been cracked and all qualifying values copied to the value partition, the value partition is registered in the adaptive index. Notice that each time the algorithm searches an ID partition for a new range of values, the ID partition is sub-partitioned by value (e.g., see the extra dotted lines in Figure 7.1), which further speeds up future searches.

Finally, there is the more general case that the two query bounds are con-
CHAPTER 7. ADAPTIVE INDEXING HYBRIDS

Figure 7.1: Hybrid Example Steps
tained in different value partitions. E.g., the low bound of Query 4 of Figure 7.1 falls in sub-partition $a_i$ of one value partition and the high bound in sub-partition $a_n$ of another. If $C$ possessed the ordering properties of a cracker column, this would mean that all areas in between, i.e., that $a_{i+1}$ to $a_{n-1}$ qualify for the result and we only need to inspect $a_1$ and $a_n$. However, in the case of the hybrid algorithm we also need to make sure that all areas in between are actually merged; if not, we must merge the needed key ranges into the value partitions in the adaptive index.

Let us go through the process step by step. The algorithm handles the qualifying areas one by one, starting with $a_i$. If $a_i$ is a non-merged area, we need to merge all values between the low bound of the query ($L_4$) and the low bound of $a_{i+1}$ from the initial ID partitions. This is done by cracking the ID partitions and copying qualifying values into a new chunk as discussed above. Otherwise, $a_i$ is already merged, and we can inspect the adaptive index to determine whether values larger than $L_4$ already form a contiguous area at the end of $a_i$. If not, we crack $a_i$ accordingly. This is the case for Query 4 of Figure 7.1.

Then, the algorithm checks areas $a_{i+1}$ to $a_{n-1}$. If an area is not merged, it again uses cracking to merge all qualifying values from the initial partitions into a new value partition and add it to the adaptive index. In our simple example (Query 4), just a single middle area needs to be merged.

Finally, when $a_n$ is reached it is treated as $a_i$, i.e., if $a_n$ is not merged, then we merge all values between the high bound of $a_{n-1}$ and the high bound of the query ($H_4$). Otherwise, if $a_n$ is merged, then we crack it based on $H_4$ to move all values smaller than $H_4$ to the beginning of $a_n$.

**Hybrid First Query Cracking**

In the above description, the first query performs a number of tasks as a side-effect. First, it slices the input column and copies its values into ID partitions. Then, for each ID partition, it materializes tuple identifiers, cracks on query boundaries, and copies qualifying values into a merge value partition.

Here we show how to avoid this last step to make the first query even more lightweight. The observation is that for each partition, the cracking action already writes the qualifying values once. This is to place them in the proper area of the partition. Then, the algorithm copies these values into the value partition, which leads to a second write action. The idea is to have a new cracking algorithm that immediately writes the qualifying values directly into the proper area of the value partition to avoid the second write. Notice, that
we do not need to write these values into the ID partition as well. The reason is that once a value range has been merged into a value partition, the algorithm will never look for it in any ID partition in the future.

In this way, for each ID partition the hybrid performs in one go the creation of the partition (including ID materialization) and the writing of the qualifying values into the appropriate value partitions. This boils down to a filtering action while reading the next \( p \) tuples from the input. It creates a new empty ID partition and places every value lower than the requested range in the beginning of the partition, while each value higher than the requested range is placed in the end. Qualifying values are written immediately in the adaptive index’s value partitions. The area corresponding to the qualifying values in the ID partition is left unwritten. However, its existence is required to align the positions of the rest of the pieces of each ID partition, allowing the tables of contents to work correctly and guide future accesses.

7.3.2 Insights

The hybrid algorithm replaces the expensive sorting actions during the first query in adaptive merging with cracking actions. In addition, compared to pure cracking, instead of having to operate over a complete array of values in one go, it operates over one small partition at a time, increasing locality and avoiding the delay of first loading the entire column. This makes the first query much more lightweight, resulting in a more fluid adaptation to workload changes without penalizing the query that happened to be the first.

The “overhead” is that if future queries need to merge more values, they cannot exploit fast binary search actions such as adaptive merging can employ to search its initially-sorted runs. Instead, every trip back to the ID partitions is achieved with cracking actions to retrieve the needed values and leave the ID partitions in a state where future accesses can exploit even more knowledge to improve performance. In short, every time we need to merge more values, merging becomes faster and faster as cracking the partitions becomes faster given the accumulated knowledge from previous cracks. This brings a more dynamic and self-organizing behavior with data structures dynamically adapting to the workload when needed.

Similarly, value partitions are not initially sorted. Unless a given query needs its result sorted, there is no benefit in investing in sorting a result area on-the-fly. However, insofar as value partitions represent a partial-ordering and the adaptive index maintains information on the value partitions and the data they contain, the adaptive index permits fast future accesses to data merged
into value partitions. If a future query overlaps with past ones and we need to search within a merged area, we cannot perform binary search actions and we have to perform cracking instead. Again, this transfers the cost dynamically to those queries that actually need to perform the extra work allowing the system to more gradually and incrementally adapt to the workload.

### 7.3.3 Updates and Multi-column Indexes

#### Updates

Update algorithms for cracking have been studied in Chapter 4. Given that the final adaptive index produced by the hybrid algorithm is essentially a crack array, the algorithms of Chapter 4 apply without any change in techniques or behavior. The main idea is that updates are modeled as pending insertions and pending deletions. Conceptually, they can be thought of as yet another partition needing to be merged. When an operator processes a given area of the crack array, it also checks whether there are any pending updates for this area. If there are, it directly applies the updates on-the-fly. The trick is to do this in a lightweight fashion while at the same time maintaining knowledge about how values are organized, i.e., without invalidating index information. The algorithms of Chapter 4 exploit the fact that a crack array is not fully sorted and swap/move/insert values only at the edges of the affected areas in an array. Similarly, deletes leave empty spaces at the edges of cracked areas, which are then filled in with future updates on this or neighboring areas.

#### Multi-column Indexes

Multi-column indexes have been studied in Chapter 5. Those techniques apply directly here, given that the end result of the hybrid algorithm is essentially an augmented crack column. The main idea in Chapter 5 is that the knowledge gained for one column over a sequence of queries is adaptively passed to other columns when the workload demands it, i.e., when multi-column queries appear. The whole design is geared towards improving access to one column given a filtering action in another one, leading to efficient tuple reconstruction in a column-store setting. The net result is that the multi-column index is essentially built and augmented incrementally and adaptively with new pieces from columns and value ranges as queries arrive.
7.4 Experimental Analysis

In this section, we continue with a detailed experimental evaluation. We use a 2.4 GHz Intel Core2 Quad CPU equipped with one 32 KB L1 cache per core, two 4 MB L2 caches, each shared by 2 cores, and 8 GB RAM. The operating system is Fedora 12.

Implementation Details

We implemented adaptive merging and the hybrid algorithm in MonetDB and fully integrated the implementation within the cracking module of MonetDB. This is the first such analysis of these two prior adaptive approaches relying on a complete implementation. In the same way, as with original cracking, adaptive merging and the hybrid algorithm resulted in new select operators that perform, register and maintain any necessary reorganization actions over the data on-the-fly. The complete code base is part of the latest release of MonetDB available via http://monetdb.cwi.nl/.

Experimental set-up

The experiments are based on running a sequence of queries with a specific workload pattern over a given data set. The metrics used are (a) the response time for each individual query, (b) the cumulative time to answer the complete query sequence and (c) how fast performance reaches the optimal level, i.e., retrieval performance similar to a complete index. We purposely keep the queries simple to focus on the adaptive behavior of adaptive indexing and how the hybrid algorithm can improve it. Behavior for more complex queries follows the same patterns as in Chapter 5. Here, we use queries of the following form.

\[
\text{select } a1 \text{ from } R \text{ where } a1 > \text{low and } a1 < \text{high}
\]

We always compare our adaptive indexing methods against two classic approaches. First, we compare against the simple scan of a table without indexes. This represents performance in the absence of investment in index optimization, the typical alternative of a system that has not been prepared for a given workload. Second, we compare against a table with an appropriate sorted index created prior to query processing. In our implementation, the first query performs a complete sort of the data such that all subsequent queries may employ binary search. This case assumes sufficient knowledge and idle time prior to query processing that appropriate indexes can be prepared.
7.4. EXPERIMENTAL ANALYSIS

To meet the motivation for adaptive indexing, we will test all methods using a completely dynamic scenario, i.e., a scenario where we do not have the luxury of a-priori knowledge or idle time to prepare indexes ahead of time.

7.4.1 Random Workloads

In our first set of experiments, we use a column of $4 \times 10^8$ tuples with randomly distributed integer values in $[0, 10^8)$. We fire $10^4$ queries, where each query asks for a randomly located range with 10% selectivity. In general, such random scenarios can be considered the “worst cases” for adaptive indexing as a widely spread access pattern requires many queries to build-up dense index information. We will see in the next section how more focused workloads result in a much faster optimization of the relevant key ranges.

In principle, all strategies (other than scans) eventually result in a fully sorted storage. They differ in how they implement the required $n \log n$ comparisons and when they schedule them across the workload. The major trade-off is low overhead over a simple scan for the first queries vs. fast convergence to a complete index.

Figure 7.2 depicts the cumulative average response time $\bar{T}(i) = \sum_{j=1}^{i} t(q_j)/i$ for $i \in \{1, \ldots, 10^4\}$. Part c) shows the whole range of $10^4$ queries, while parts b) and a) magnify the results for the first 200 and first 10 queries, respectively.

Scan and Sort

Figure 7.2a highlights that sorting makes the first query almost ten times as expensive as a simple scan. Figure 7.2b reveals that, in terms of cumulative costs, sorting in this case only pays off once more than 24 queries have to be answered. Figure 7.2c shows how with long query sequences the cumulative average response time with scan stabilizes at a high level, while with sort it decreases linearly.

Scan has a relative stable performance for each individual query as it always does the same job. Only the first query is more expensive as it needs to also load the data from disk. On the contrary, the complete sort method has a high start-up cost to fully sort the column, but as of the second query it enjoys the optimal performance, which is multiple orders of magnitude better than that of the simple scan (the 100% curves in Figure 7.4 demonstrate the per query costs).
Figure 7.2: Improving Adaptation with the Hybrid
Adaptive Merging

We prepared two implementations of adaptive merging. They differ in how they merge tuples from the initial partitions to the final column. The first one, AMpq, uses a traditional priority queue mechanism that has proved to work very well in the context of disk-based paged row-stores. However, in our column-store engine using arrays for in-memory processing, this implementation appeared to be sub-optimal. In short, a large number of runs is required to limit the runs to CPU cache size for efficient sorting. However, merging this large number of runs requires a large priority queue that exceeds the CPU cache size. Perhaps more importantly, a large fan-in interacts poorly with virtual memory and address translation. We plan on a deeper analysis in the future, including multi-level merging.

Consequently, performance suffered from too many cache misses due to the inherently random accesses, both inside the priority queue itself and when accessing the original runs. To overcome this problem, and using the lessons learned from the MonetDB code base, we exploited a “brute-force” mechanism of copying the qualifying key ranges from all runs successively to the final merge column, and then sorting the result there in place, AMcs. Featuring purely sequential access during the copy and random access beyond CPU cache size only in the first steps of the final quick-sort, this turned out to be a significantly more efficient implementation for an array-based column-store.

Figure 7.2a shows that both AMpq and AMcs significantly reduce the costs for the first query compared to a full sort. This is due to sorting all small slices individually such that random data access only occurs inside the CPU caches, significantly reducing the number of cache misses compared to completely sorting the whole column. We use runs of size equal to the L1 cache, which proved to provide the best performance. Absolute per-query performance improves very fast for both implementations (much faster than cracking), helping to quickly reduce the average response times (more evident in Figure 7.4). During the initial part of the query sequence, when a completely new value range is requested we need to perform more merging which is why we see a few peaks in performance. Essentially, the cost of the full sort is amortized over multiple queries in a self-organizing way. (This is even more evident if one looks at the per query response time graphs at the end of this section in Figure 7.4).

Purely considering total sequence costs, with AMpq, the high merging costs eliminate the benefit over sort already after the second query. When running at most 8 queries, AMcs offers lower total costs than sort.
CHAPTER 7. ADAPTIVE INDEXING HYBRIDS

Cracking

Adaptive merging achieves its incremental behavior by improving over sorting for the first query while sacrificing a few queries after that to compensate for a non-complete index. Cracking on the other hand concentrates on minimizing the initialization overhead as much as possible at the expense of a longer compensation period targeting more dynamic workloads and bigger savings in case of focused workloads that do not touch the complete data set. For example, in Figure 7.2a we see that cracking needs only 18 seconds for the first query while sorting needs 90 and AMcs 56. After the first query, though, cracking needs significantly more queries than adaptive merging to refine its storage and index before it reaches (close to) minimal response times (cf., Figure 7.4). However, thanks to its low overhead on the first query, the cumulative average times stay below that of adaptive merging and sorting (cf., Figure 7.2c). When purely considering total sequence costs, already as of 5 queries, the initial investment pays off even compared to a simple scan (cf., Figure 7.2a).

The question that arises then is; How can the ideas of adaptive merging and database cracking be combined to bring the costs for the first queries down to no more than the simple scan, while still maintaining the adaptive behavior that quickly converges to an optimal status?

Hybrid

By sorting only small partitions, adaptive merging reduces the cost of the first query compared to sort. However, the cost breakdown shown in Figure 7.3 reveals that sorting the small partitions is still the largest cost component for the first query with AMcs. Hence, to reduce the overhead of the first query, we need to avoid sorting completely. Since also cracking the whole column with the first query comes at a significant cost, we combine the partitioned approach of adaptive merging with cracking to create our new hybrid approach.

Figure 7.2a shows that cracking only CPU cache-sized partitions comes at practically no overhead compared to a scan in the first query. Also for the next few queries, the hybrid algorithm shows no overhead compared to simple scan (see also the per query costs in Figure 7.4). By continuously reorganizing the ID partitions as well as the value partitions according to the requested key ranges, it successively refines its storage and index structures. With this, per query costs in Figure 7.4 successively improve at about the same rate and low merging costs as with cracking. Thus, just like cracking it converges slightly slower than adaptive merging, but with the advantage of generally lower and
more predictable merging costs. The major benefit of the hybrid algorithm both over adaptive merging and cracking is that it achieves this adaptive behavior at no extra cost over simple scan in the first queries.

Figures 7.2b and 7.2c show that in the long run, the hybrid sacrifices some performance in terms of cumulative and average response times compared to cracking but at the same time it is significantly faster than those of adaptive merging and of course scan. The major reason is the administrative costs to handle cracking on the large number of partitions required to have CPU cache-sized partitions. We are confident that this cost can be significantly reduced and eventually eliminated by a more sophisticated implementation that fuses partitions to reduce their number as they become smaller, with tuples being merged into the index.

### 7.4.2 Focused Workloads

Up to now, we studied random workloads. Here, we show how adaptive indexing performs in more focused workloads.

Figure 7.4 shows per query response times $T(i) = t(q_i)$ with queries that
Figure 7.4: Focused workloads
focus on 100%, 60% and 20% of the data set. The first case is exactly the case we studied so far (same run) representing random queries over the entire data set, while the other two focus on smaller parts. Serving as the two extreme reference points, the scan and sort results are identical across all three graphs. Scan and sort are not affected by the workload pattern, whereas all adaptive indexing methods manage to more quickly adapt as the workload becomes more focused. What happens is that even without a completed index all adaptive methods have already gained enough knowledge for the workload’s hot set and can provide the optimal performance even though the rest of the index is essentially empty.

In general, all adaptive indexing strategies show excellent performance and behavior with a variety of focused workloads while the hybrid algorithm maintains its major benefit of essentially building an index for free.

We continue by studying another set of detailed experiments, focusing on individual query performance. In particular, we study various kinds of non-random workloads, i.e., cases that express a particular non-random query pattern. We show that in such cases adaptive indexing techniques achieve even better performance by being able to more quickly reach a fully optimized status, accentuating benefits over traditional techniques.

Up to now, we studied simple focused workloads where the pattern was that we simply restricted our interest to a specific part of the domain, i.e., to $x\%$ of the possible ranges we could query. Here, we study more complex and general patterns where the focus gradually shifts into target areas based on a specific pattern. These kind of scenarios represent more realistic workloads where the user explores and analyzes the data based on on-the-fly observations.
CHAPTER 7. ADAPTIVE INDEXING HYBRIDS

Jump Patterns
In this experiment, we analyze a workload with a jumping focus, i.e., the focus is initially on 20% of the data set, then after 1000 queries it jumps to a different 20%, then to a different 20% and so on. This represent a behavior where the user will eventually study the whole data set but we do not need the complete data set optimized in one go or at all times. The left part of Figure 7.5 shows such a workload; as the query sequence evolves the workload focuses on a small area of the domain and then the focus jumps to a different area.

Zoom Patterns
The zoom workload pattern reflects a zooming behavior where progressive understanding during query processing leads to the actual point of interest, i.e., the workload stepwise zooms into shrinking areas of interest. The middle part of Figure 7.5 shows such an example. Here, the first 2000 queries are randomly spread over the whole key range, the next 2000 queries focus on only the center 80% of the key range, the next 2000 queries focus on the center 60% and so on, until in the 5th step the target area has shrunk to the center 20% of the key range.

Exploration Patterns
Finally, we mimic an exploratory workload by combining the Jump and Zoom ideas as follows. The first 500 queries explore the whole key range. Then, in steps of 500 queries, the target is successively limited to the first 50%, 40%, 30% and finally 20% of the key range. Thereafter, 500 queries again consider the whole key range, before a next sequence of 500-query steps zooms into 50%, 40%, 30%, 20% at a different location in the key range. We repeat this 2 more times, i.e., 4 times in total. This workloads represents a quite realistic exploration of the data set where the user continuously zooms in and out of interesting areas trying to interpret the data. The right part of Figure 7.5 reflects such a behavior. Again, at any given time, we only need part of the data set optimized.

Discussion
Figures 7.6, 7.7 and 7.8 show the results. They are meant to provide a high level visualization to provide insights of how adaptive indexing works in a variety of
Figure 7.6: Adaptive Merging for Various Workload Patterns
Figure 7.7: Selection Cracking for Various Workload Patterns
Figure 7.8: Hybrid for Various Workload Patterns
scenarios. For each workload pattern we report all adaptive indexing techniques in comparison to the baselines of a priori sorting and on the fly scan.

Sort and scan are insensitive to the workload and thus they provide the same performance across the range of workloads tested. For example, scan will always scan the same amount of data while sort will always do a complete sort to exploit binary search from there on. Thus, scan has always the same high cost, while sort will always pay a heavy initial cost. Adaptive indexing though provides a very different behavior trying to always exploit the workload patterns in order to improve. This is true for all our adaptive indexing techniques with the hybrid having the edge due to the more lightweight adaptation as we discussed in the main paper. We do not provide figures that focus on the initial part of the sequence, comparing initial costs, as this is the same behavior seen before. The goal here is to present an overview of how the adaptive indexing algorithms adapt to the changing workload patterns. For example, notice how for the Zoom workload all adaptive indexing techniques improve faster (in terms of queries processed) to the optimal performance levels compared to a random workload as well as improving even further. The fastest query in the Zoom workload is one order of magnitude faster than the fastest in the Random one. By focusing the workload on specific key ranges, adaptive indexing can improve its knowledge over these key ranges more quickly. Similar observations stand for the Jump and Explore workload patterns as well. In these cases, the shift from one key range to another is visible by a few high peaks each time representing a workload change. Once the workload focuses again, performance improves very fast and in most cases performance stays way faster than the plain scan approach. Without any external knowledge and completely in an automatic and self-tune way performance improves purely based on the workload.

7.5 Summary

Adaptive merging extends the ideas of cracking in a very interesting way targeting mainly disk processing trying to minimize the adaptation period. The new techniques presented in this chapter make the first step in expanding the scope of adaptive indexing even further by trying to blend ideas and concepts from both adaptive merging and cracking. They offer an additional choice that combines advantages and avoids disadvantages: hardly any burden is added to a scan! This suggests an interesting new approach to physical database design. The initial database contains no indexes (or only indexes required in support of uniqueness integrity constraints). Query processing initially relies on large
scans, yet all scans contribute to index optimization in the key ranges of actual interest. Due to the fast index optimization demonstrated in our implementation and our experiments, query processing quickly transitions from relying on scans to exploiting indexes. Due to low burden required to create and optimize indexes, query processing using this approach to physical database design should offer the best of both traditional scan-based query processing and traditional index-based query processing, without the need for explicit tuning or for a workload for index creation.

Adaptive indexing opens a whole new research area. The work in this thesis has only scratched the surface of it, demonstrating the feasibility and the potential. Several issues are open for detailed studies. For example, the work in this chapter concentrates on the index initialization overhead, assuming mainly a memory resident DB or at least one where the hot set mostly fits in memory. There are several other parameters to consider when comparing all these adaptive indexing techniques, i.e., concurrency control, updates, recovery, scalability for large data, etc. In many of these cases, a more active method than pure cracking can potentially be more beneficial. For example, pure adaptive merging with its very active sorting steps in the beginning of a query sequence, significantly reduces the adaptation period resulting in a highly optimized index very soon in terms of numbers of queries needed. This helps with concurrent queries since the index needs no further reorganization and thus there are no conflicts. It also requires less effort in continuously rewriting a disk based index every time we reorganize it as there are less reorganization periods. In addition, low-level implementation details could potentially affect the performance picture, i.e., adaptive merging could gain a lot from a multi-level merging approach.

It is evident thus, that there is a plethora of paths to investigate towards the ultimate adaptive indexing method that can cope with any kind of workload and system set-up. The first hybrid approach presented in this chapter has all the hooks to become the vehicle towards this ultimate goal and properly balance the relevant tradeoffs, e.g., initialization investment, adaptation speed, scalability, etc. There is a more detailed discussion on these issues in the next chapter. In simple words, with the hybrid we can dynamically schedule how many reorganization actions we do. The trigger does not have to be a single query, but the general status of the system and the workload, resulting in more active or more lazy reorganization periods to fit the system and workload.

Having seen in the previous chapters a detailed discussion on the cracking architecture, the algorithms for updates and join processing, as well as techniques inspired by cracking, merging and hybrids in this chapter, we move on to discuss future research directions in the next chapter.