

Adaptive Indexing: Cracking Facts and Merging Knowledge

Pavlo Shevchenko
Otto-von-Guericke-University, Magdeburg
pavlo.shevchenko@st.ovgu.de

Therefore, even as an index to a book
So to his mind was young Leander's look

Christopher Marlowe
Hero and Leander, 1593

Abstract—What did I do in a nutshell?

Keywords—Database indexing, adaptive indexing, database cracking, adaptive merging, hybrid approaches.

I. INTRODUCTION

The concept of indexes was widely used in the library science throughout the history of mankind and remains the main tool to help the reader find information quickly. Indexes easily made their way into database technologies. The introduction of such intuitive concept helped to improve the speed of data retrieval. In the early stages of database technology, data and indexes were stored on disk in a form of B-Trees. These datastructures, optimized for these purposes, enable efficient usage of disk pages and management of data stored in disk.

However, over the years, as in-memory databases (IMDB) found their way to the market, the need for more lightweight index structures has become evident. At first, a precalculated index in a form of a sorted column with pointers to the original records in the table has become a concept that helped to decrease the query response time. However, continuously growing amount of data, more sophisticated queries and unpredictable behavior of a user have turned advantages of indexes into disadvantages[2]. Storing large columns in a main memory has become a problem, the need to store indexes to numerous columns has even increased that problem. As it is hard to predict which column is especially interesting for the user, new indexes have to be calculated online as queries referring to new columns arrive.

All these issues motivated the development of advanced indexing techniques such as *partial* [1] and *soft* [3] indexes. We discuss both these concepts in detail in Section II. Creators of the latter realised that an index cannot remain static after being created, it requires to be continuously tuned as some parameters of the database system are changing. Such parameters include workload, size and distribution of data, schema and infrastructure changes [3]. Currently, indexes are tuned by DBA with help of some management tools that provide this information to system administrators. However, it is desired that database tuning is performed by DBMS alone (like it is already done for storage layout, access optimization, etc.).

Statistical data mentioned before is collected during query processing. Therefore, as new queries arrive, DBMS can adapt existing indexes or create new ones on-the-fly, depending on how these parameters have changed [3].

This concept known as adaptive indexing will be presented in this short paper. We revisit some fundamental approaches (e.g. *database cracking* [4] and *adaptive merging* [6]) in Sections III and IV respectively. Further approaches that try to combine the advantages of both methods will be introduced in Section V. We evaluate the presented approaches in Section VI. We conclude this paper by reviewing related work in Section VII and proposing directions for further research in Section VIII.

Our goal is to provide a complete overview of adaptive indexing techniques to help everyone interested in them understand the core concepts of the approaches. Furthermore, we aim to draw attention to these techniques and motivate further studies in the area of adaptive indexing.

II. BACKGROUND

As mentioned before, two indexing methods: partial indexes [1] and soft indexes [3] were developed to tackle some problems caused by traditional indexes. In this section, both of these approaches and their influence on adaptive indexing will be presented.

A. Partial Indexes

As its name suggests, a partial index, also known as a filtered index, is an index which contains only part of the indexed column that satisfies some condition. Such condition is usually expressed in the form of an interval known as *inclusion interval*. Depending on selectivity of a condition, the amount of rows in the indexed column can be decreased. This allows to reduce the amount of memory needed to store the index. The creation of such index may be specified as follows:

```
CREATE INDEX ON  
relation(column) name  
WHERE condition
```

In Figure 1, we present an example of a partial index with a condition ' $D' \leq A \leq O'$ '. Next to it, we give an example of a full index to a given table.

Clearly, selecting the right condition is a key element for the creation of a partial index. This condition can be found in several ways:

- **User Input [1]:**
User can specify the insertion interval while possessing some additional knowledge about the data. However, one of the major principles of modern DBMS is removing any low-level data management aspects from the user of the system. As a result, DBMS should be able to derive such conditions itself.
- **Index Creation as Side-Effect of Query Processing [1]:**
Even though the user has been removed from tuning process, she can still indirectly participate in finding a suitable condition. The fact that user queries some relation (column) while applying some selection condition, leads to an assumption that this part of data is of some interest for the user and may be queried again in the future. So, taking query condition from the user as a qualification for partial index is not a bad idea after all. Continuous refinement of a condition and ad-hoc indexing may provide an optimal partial index for given relation. This concept, known as *online index creation* was later used in database cracking [4].
- **Statistical Approach [2]:**
Since index creation in a way described above may increase query processing time, statistical information, like the percentage of queries that access each column or distribution of values in each column, may be used to perform index creation after data loading and before queries are processed. In contrast to previous concept, this approach is known as *offline index creation*, as the index is not created during query processing, but before.

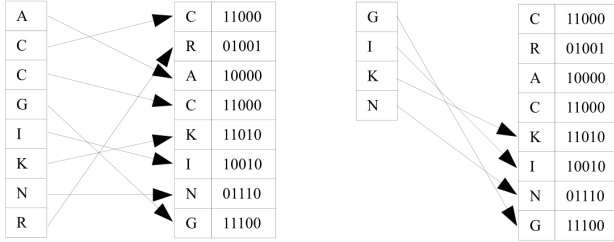


Figure 1. Full Index (left) and Partial Index (right)

B. Soft Indexes

In contrast to *hard indexes* that are created by database assistant and stored as long as he wants to, soft indexes proposed in [3] are created, modified and dropped by DBMS depending on the current configuration of the system. Index management is carried out in three major steps:

- Observation
- Prediction
- Reaction

At first, statistical information about current state of the system (workload, data, schema, infrastructure) is collected

in the Observation phase. In the Prediction phase, a list of index candidates sorted by some criteria is analyzed and top k candidates are chosen for later materialization. During the Reaction phase, depending on results of Prediction phase, some indexes will be created or deleted [3].

Value. Combination of advantages of soft and partial indexes resulted in further approaches known as adaptive indexing. This concept promotes the idea of completely automatic tuning process of a database system. Adaptive indexing achieves it by continuously reorganising physical design of a database system during incremental and partial online indexing [11]. On-the-fly index creation based on the query workload and storing only subsets of rows in an index motivated the first method for adaptive indexing, called *database cracking* [4], which we present in the next section.

III. DATABASE CRACKING

After presenting the necessary background about partial and soft indexes, we revisit one of the main approaches of adaptive indexing, database cracking. In Subsection III-A we discuss main idea behind database cracking, in Subsection III-B we revisit the standard cracking algorithm. In Figure 2, we give an example of how the described algorithm can be used. To conclude this section, we analyze database cracking and compare it to full index and full scan.

A. Motivation and Basic Idea

Motivation. Database cracking is pursuing the goal to provide fast access to the data and the self-organized behavior of database system. It is based on idea that index maintenance is a byproduct of query processing, not updates [4]. In contrast to soft indexes, discussed earlier, database cracking achieves its goal not by tuning system's configuration and choosing the best fitting indexes based on numerous parameters, but by using continuous physical reorganization (cracking the database into manageable pieces).

B. Standard Cracking

Main Idea: *incrementally perform quicksort on a copy of a column using crack-in-three when range query fully falls in the same partition and crack-in-two otherwise.*

Crack-In-Two: *partition the index column into two pieces using range as a split line.*

Crack-In-Three: *partition the index column into three pieces using ends of a range as two split lines.*

Steps of Standard Cracking (given a column-oriented database with column A ; range query q in a form of either $c \leq R.A$ or $c_1 \leq R.A \leq c_2$):

1) Initial Index Creation:

The first time query on attribute A arrives, copy of the column is created and cracked using *Crack-In-Two* or

Crack-In-Three (notation: A_{crk}). This allows to leave original column intact and to have a fast reconstruction of records.

2) Refinement:

As further queries on attribute A arrive, partition A_{crk} with regards to earlier cracking. E.g. $q_1 = 5 \leq R.A \leq 10$. After cracking, A_{crk} will contain three partitions as described earlier. When next query arrives $q_2 = 7 \leq R.A$, then cracking should be performed only on part of A_{crk} between positions p_1 and p_2 , as it is known that all elements before position p_1 satisfy the query and all elements after p_2 don't.

3) Cracking Index:

In order to quickly find the partition to be refined, each cracker column is provided with a *cracker index* which stores how values are currently distributed in the cracker column. Each node of a tree stores key value, start position of a partition with this key, and whether the key itself is included in the range or not.

4) Query for new column B arrives

Perform steps 1 through 3 for new column B and store previous results.

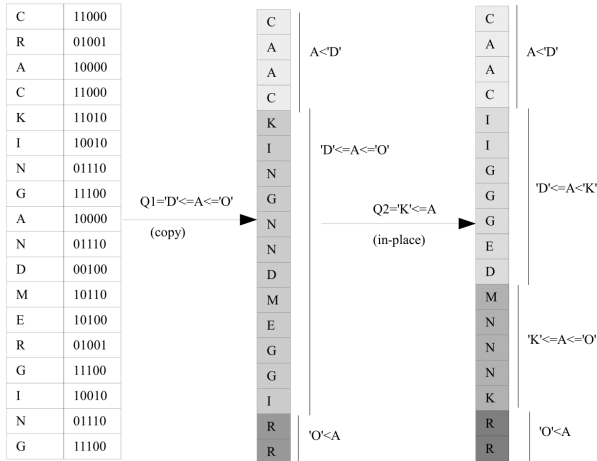


Figure 2. Cracking a column (adapted from [4])

C. Analysis and Criticism

Analysis. In order to analyze database cracking, we compare this approach to a full index and a full scan on a respective column.

- Costs of Initial Run:

High creation costs are the main concern of a full index. While using full index, query processing can start only after initial index creation. Database cracking offers significant improvements to avoid full sort of

an index column. Initial run happens as the first query arrives. In general, first crack takes longer than full scan, as cracking invests some time in indexing. However, cracking still needs less time for initial run than full index to sort the data [5].

- Costs of Query Processing:

After being created, full index processes new queries fast, utilizing advantages of index structures. Full scan takes on average same time to process each next query as it scans the whole table to find relevant tuples. Apart from first query, database cracking performs better than full scan and incrementally reaches the performance of a full index [5].

- Convergence Towards Full Index:

As mentioned in previous point, database cracking constantly narrows its performance towards the full index. It is achieved by incrementally sorting the cracked column when new queries arrive. Consequently, cracked column converges towards full index after number of queries. Clearly, the quality of a cracking step highly depends on the split line defined by query's range. Thus, it is hard to predict how soon query response time of cracking will be reasonably close to one of the full index. On average, after 1,000 queries, query response time of standard cracking is still about 40% higher than of full index [10].

Criticism. Several weaknesses of database cracking were outlined in the past. Apart from slow convergence towards full index, they include unpredictable query response time, worse performance with growing number of projected attributes, and not suitable implementation for block-access storage. In [5], Schuhknecht reviewed several approaches such as Vectorized [9], Hybrid [8], Sideways [12] and Stochastic [13] cracking, that try to solve these problems. However, one of the key problems of database cracking: slow convergence, remains unsolved. This concern motivated the development of a further adaptive indexing method: *adaptive merging*, which we present in the next section.

IV. ADAPTIVE MERGING

In Section III, we presented a pioneering approach for adaptive indexing, known as database cracking. We outlined some issues related to this method. In following subsections, we present another adaptive technique for index creation, known as adaptive merging, that tackles unsolved issues of database cracking [6]. In Subsection IV-A, we present motivation and main idea behind adaptive merging. In Subection IV-B, we describe the approach, followed by an example in Figure 3 and in Subsection IV-C, we conclude the section with some analysis of this method.

A. Motivation and Basic Idea

Motivation. Creators of adaptive merging, Graefe and Harumi, outlined four main weaknesses of database cracking that motivated their research [6]:

- 1) Slow convergence towards full index.
- 2) Efficiency of cracking depends on query pattern.
- 3) Query response time never reaches that of full index, when cracked column has minimal number of unsorted partitions.
- 4) Database cracking is well-suited for in-memory databases but not for block-access storage.

Main Idea. Graefe and Harumi proposed adaptive merging, new technique that combines the efficiency of traditional B-tree creation with adaptive nature of database cracking. As its name suggests, adaptive merging is based on merging (as used in merge sort) rather than on partitioning (as used in quicksort and database cracking) [6]. This new technique promises better query response time and quicker adaption to the new data and new query patterns that database cracking.

B. Adaptive Merging

Main Idea: *exploit partitioned B-trees [7] by merging key ranges relevant to actual queries, aiming to reach a desired state and have only a single partition.*

Steps of Adaptive Merging

- 1) Index Selection:
If a column is referred by query for the first time, a new index is created by copying values of a column.
- 2) Initial Index Creation:
These values are loaded into a number of sorted partitions of predefined size.
- 3) Index Refinement:
When a column is used for the second time, the query scans one or multiple partitions for a desired key range. It is done by efficiently finding the low end of the range and then scanning to the high end. Several sorted streams of values, resulting in this step, will be merged into a single sorted stream that will be written into a new partition. Moreover, this stream is returned as a result of given query.
- 4) Overlapping Key Ranges:
Same as in database cracking, ranges of further queries may be fully included in already existing partitions, lay outside of them, or only part of new range is included in existing partitions. Following strategy is applied in these cases:
 - New query range is a subset of previous query: Perform efficient search in a partitioned B-tree.
 - New and old query ranges do not overlap: Perform Step 3, create new partition and return it as a result.

- New and old query ranges partially overlap: Split new range into overlapping and non-overlapping sub-ranges. For overlapping subranges perform search in a partitioned B-tree. For non-overlapping subrange perform Step 3 and create new partition for further queries.

5) Storage of Reorganization Information:

Like in database cracking, some data structure is required to retain information about performed reorganization. In contrast to a cracker index used in database cracking, the data structure in adaptive merging contains a range of identifiers for the partitions in which given range of keys can be found. As numerous partitions can contain given key range, list of partitions with this key range has to be stored.

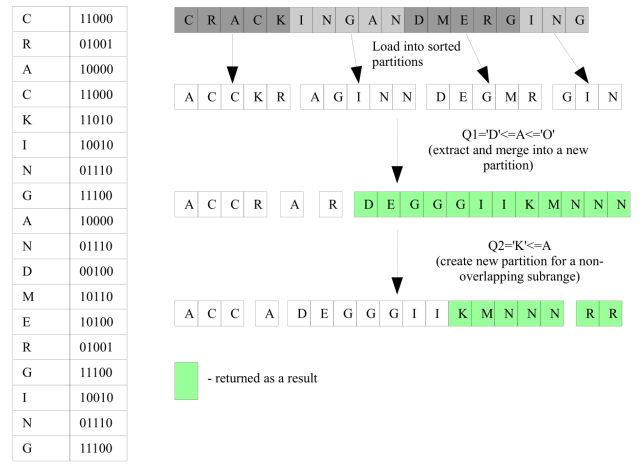


Figure 3. Example for adaptive merging (adapted from [6])

C. Analysis and Criticism

Analysis. In order to analyze adaptive merging, we compare this approach to database cracking.

- Cost of Query Processing:
Similarly to database cracking, the overhead of the first query is large, as the whole domain has to be scanned and partitioned. However, overhead of query processing by adaptive merging decreases faster than that of database cracking. This can be explained by following:
 - 1) Fast convergence: In order to divide 10,000,000 records into partitions with appr. 1,000 entries requires at least 9,999 partitioning keys. As each query provides at most two new split lines, to reach this aim, database cracking requires around 5,000 queries. On the other hand, adaptive merging creates a fully optimized B-tree after 40 queries, reaching the behavior of a full index [6].

- 2) Sorted partitions: Sorted partitions in adaptive merging enable lower query response time even if B-tree is not fully optimized. In database cracking, although new range query may completely lie in already existing partition, cracking must be performed. As partition is unsorted, distribution of relevant elements for new query in existing partition is unknown. On the other hand, sorted partitions in adaptive merging enable fast search for the position of a lower key in a partition and scan of entries till higher value.

- **Block-Access Storage:**

In contrast to database cracking, designed for in-memory databases, adaptive merging enables adaptive index creation in large data warehouses on external storages. Similarly to traditional B-trees and external merge sort, adaptive merging can be applied to partitioned B-trees and used in block-access storage [6].

Issues. Although adaptive merging matches and may even exceed query response time of full index after a small number of queries, its performance over first queries is concerning. While database cracking matches the scan performance with the second query and exceeds it after the third, adaptive merging needs around 7 queries to exceed the scan time. The first query may be nearly 5 times slower than a scan [8].

V. HYBRID APPROACHES

In the previous sections, we revisited two fundamental approaches for adaptive indexing: database cracking and adaptive merging. In this section we present further approaches that try to combine strength of both database cracking and adaptive merging. In Subsection V-A we discuss the main idea behind hybrid methods. Later, in Subsection V-B, we review main components of an algorithm for adaptive indexing and discuss how these components can be combined to develop a hybrid algorithm for adaptive indexing. To conclude this section, we present existing hybrid approaches followed by examples to provide better understanding of the concept.

A. Motivation

In previous sections, we revealed two key criteria that adaptive indexing algorithms are judged by: (1) initialization costs for the first query and (2) the number of queries needed to reach the performance of a perfect (full) index. As for (1), it is desired that the initialization does not require more costs than a full scan. And as for (2), one wants to converge towards full index as fast as possible, while performing lightweight operations during query processing. Based on analysis of database cracking and adaptive merging, we find the strengths of cracking and merging complementary. Database cracking performs lightweight partitioning to keep initialization costs low, but requires a lot of time to reach the performance of a full index. On the other hand, adaptive merging aims to achieve the behavior of a full index as fast as possible while

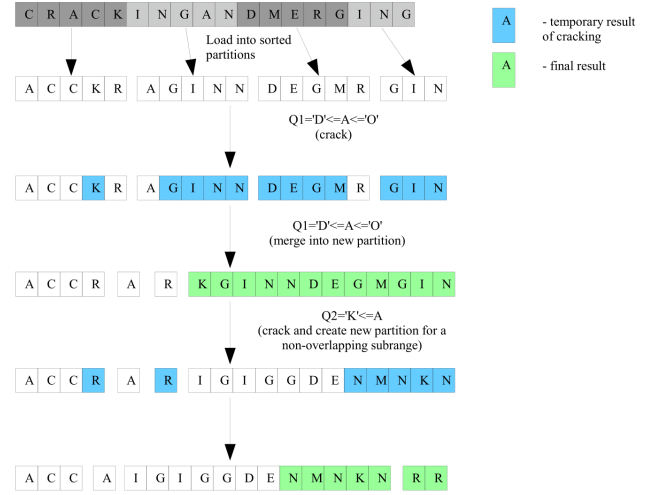


Figure 4. Hybrid Sort-Sort Algorithm

performing costly operation of sorting both while loading the data into initial and final partitions [8]. In the next subsection, we reveal the sources of their strength and how they can be combined.

B. Strategy

The main difference between database cracking and adaptive merging is in the way how initial and final partitions are organized. In [8], three key methods were presented:

- **Sorting:**
Adaptive merging uses sorting for both the initial and the final partitions. Sorting the initial partition leads to huge costs of initial run that adaptive merging has been criticised for. On the other hand, sorting a final partition is less costly, as it is only performed on a subset of data defined by a query.
- **Cracking:**
As used in database cracking, this organization step requires minimal investments, as only two or three partitions are being created. However, because of that and the fact, that new partitions remain unsorted, database cracking is slow in reaching the performance of a full index.
- **Radix-Clustering:**
In contrast to sorting, this way of partition organization is more lightweight which helps to reduce organization costs. Although, radix-clustering doesn't produce fully sorted partitions, it still brings more order to the partitions than cracking.

Hybrid Algorithm Design. Each of the mentioned organization techniques can be applied both to the initial and final partitions. Consequently, 9 potential combinations can

be created: starting with Sort-Sort (which is adaptive merging) and ending by Crack-Crack (which is a slight modification of database cracking) [8].

C. Hybrid Algorithms

After presenting the design of hybrid algorithms, we will review 9 existing approaches and present some of them in Figures 4, 5, and 6. Furthermore, we will try to narrow the research field to a couple of algorithms that promise improvements on the current adaptive indexing algorithms and discuss their key properties.

1) Sort-* Algorithms:

Fully sorting the initial partition is the reason why overhead for initial run is so high by adaptive merging. In fact, sorting takes about 80% of the time of the first query processing [8]. As a result, none of 3 algorithms (Sort-Sort, Sort-Radix, Sort-Crack) will provide the desired behavior, specifically low initialization costs.

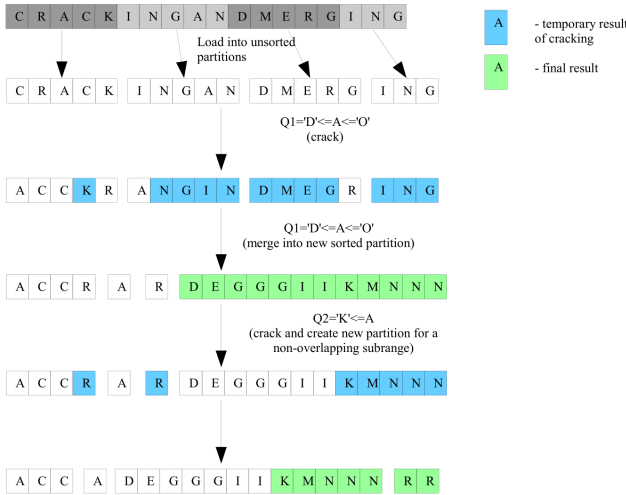


Figure 5. Hybrid Crack-Sort Algorithm

2) Crack-* Algorithms:

• Crack-Crack:

This modification of database cracking improves over standard cracking in a way that initialization costs are never higher than the ones of the full scan. It is achieved by performing cracking on parts of the column at a time and by creating and cracking initial partitions in one go, instead of copying and then cracking as in plain database cracking. Although, Crack-Crack algorithm significantly improves over standard database cracking, it does not solve the main issue: slow convergence, as it maintains the smooth behavior of cracking.

• Crack-Sort:

Crack-Sort solves this problem by sorting the

final partitions as done in adaptive merging. This allows the algorithm to achieve same costs for the first query as original cracking and fast convergence of adaptive merging. Although this behavior matches the desired one, Crack Sort sacrifices the fast initial run of Crack-Crack and is slower than the scan for the first couple of queries

• Crack-Radix:

This hybrid approach achieves balanced behavior between the previous hybrids. It achieves similar performance as Crack-Crack at the beginning of the workload and similarly to Crack-Sort it achieves a significant boost in performance. Although it does not achieve the performance of adaptive merging of Crack-Sort, Crack-Radix still significantly improves over plain cracking. Thus, Crack-Radix achieves a nice balance between database cracking and adaptive merging.

3) Radix-* Algorithms:

Although costs for first couple of queries for this group of hybrid approaches are slightly higher than the ones of Crack-* approaches, Radix-* approaches significantly improve on adaptive merging in this regard. What is more, organization techniques of final partitions can exploit the extra information provided by clustering to quickly place relevant values. This helps Radix-* approaches to come closer to the convergence behavior of adaptive merging.

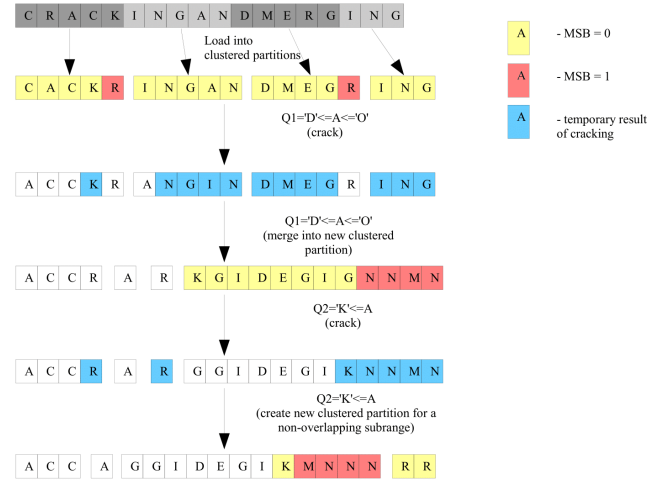


Figure 6. Hybrid Radix-Radix Algorithm

Summary. Crack-* and Radix-* algorithms show promising results. Crack-* methods approach desired behavior by minimizing the costs of initial run by using lightweight operations of cracking and keeping fast convergence of adaptive merging. Radix-* approaches achieve this goal from other angle: although they use a more expensive operation

of clustering to create initial partitions, the information about created clusters help to minimize the costs for organizing final partitions.

VI. EVALUATION

Point out complementary nature of cracking and merging. Compare to other hybrid approaches. Speculate on future and usage of the methods

VII. RELATED WORK

Some research on related work has to be done.

VIII. CONCLUSIONS

What did I find out?

ACKNOWLEDGEMENT

I thank Gabriel Campero Durand of Otto-von-Guericke-University, Magdeburg for providing insight and expertise to start this research and for his enormous support throughout the whole process of research, writing and evaluation of this scientific work. I would also like to show my gratitude to the DBSE Research Group of Otto-von-Guericke-University, Magdeburg for making this work possible and organising the “Seminar on Modern Software Engineering and Database Concepts”, during which this research took place.

REFERENCES

- [1] Stonebraker, Michael. "The case for partial indexes." *Sigmod Record* 18.4, 1989.
- [2] Seshadri, Praveen, and Arun Swami. "Generalized partial indexes." *Data Engineering*, 1995. *Proceedings of the Eleventh International Conference on*. IEEE, 1995.
- [3] Lühring, Martin, et al. "Autonomous management of soft indexes." *Data Engineering Workshop*, 2007 *IEEE 23rd International Conference on*. IEEE, 2007.
- [4] Idreos, Stratos, Martin L. Kersten, and Stefan Manegold. "Database Cracking." *CIDR*. Vol. 7. 2007.
- [5] Schuhknecht, Felix Martin. "Closing the circle of algorithmic and system-centric database optimization: a comprehensive survey on adaptive indexing, data partitioning, and the rewiring of virtual memory.", 2016.
- [6] Graefe, Goetz, and Harumi Kuno. "Self-selecting, self-tuning, incrementally optimized indexes." *Proceedings of the 13th International Conference on Extending Database Technology*. ACM, 2010.
- [7] Graefe, Goetz. "Sorting and indexing with partitioned B-trees". *CIDR*, 2003.
- [8] Idreos, Stratos, et al. "Merging what's cracked, cracking what's merged: adaptive indexing in main-memory column-stores." *Proceedings of the VLDB Endowment* 4.9, 2011.
- [9] Pirk, Holger, et al. "Database cracking: fancy scan, not poor man's sort!" *Proceedings of the Tenth International Workshop on Data Management on New Hardware*. ACM, 2014.
- [10] Schuhknecht, Felix Martin, Alekh Jindal, and Jens Dittrich. "The uncracked pieces in database cracking." *Proceedings of the VLDB Endowment* 7.2 (2013): 97-108.
- [11] Idreos, Stratos, Stefan Manegold, and Goetz Graefe. "Adaptive indexing in modern database kernels." *Proceedings of the 15th International Conference on Extending Database Technology*. ACM, 2012.
- [12] Idreos, Stratos, Martin L. Kersten, and Stefan Manegold. "Self-organizing tuple reconstruction in column-stores." *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. ACM, 2009.
- [13] Halim, Felix, et al. "Stochastic database cracking: Towards robust adaptive indexing in main-memory column-stores." *Proceedings of the VLDB Endowment* 5.6 (2012): 502-513.