Universidade Federal do Paraná

Matheus Agio Nerone

Cracking KD-Tree: O Primeiro Índice Adaptativo Multidimensional

Cracking KD-Tree: The First Multidimensional Adaptive Indexing

Curitiba PR
2018

Matheus Agio Nerone

Cracking KD-Tree: O Primeiro Índice Adaptativo Multidimensional

Cracking KD-Tree: The First Multidimensional Adaptive Indexing

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre em Informática, no Programa de Pós-Graduação em Informática, setor de Ciências Exatas, da Universidade Federal do Paraná.

Área de concentração: *Ciência da Computação*.

Orientador: Prof. Dr. Eduardo Cunha de Almeida.

Curitiba PR
2018

Catalogação na Fonte: Sistema de Bibliotecas, UFPR
Biblioteca de Ciência e Tecnologia

N452c Nerone , Matheus Agio
Cracking KD-Tree: o primeiro índice adaptativo multidimensional / Matheus Agio Nerone . – Curitiba, 2018.

Dissertação - Universidade Federal do Paraná, Setor de Ciências Exatas, Programa de Pós-Graduação em Informática, 2018.

Orientador: Eduardo Cunha de Almeida . -

1. Banco de dados. 2. Índice multidimensional. 3. Indexação adaptativas. 4. Particionamento de banco de dados. I. Universidade Federal do Paraná. II. Almeida , Eduardo Cunha de. III. Título.

CDD: 005.74

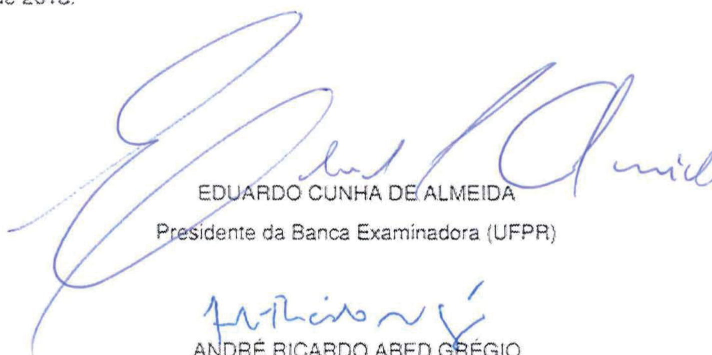
Bibliotecária: Vanusa Maciel - CRB - 9/1928

TERMO DE APROVAÇÃO

Os membros da Banca Examinadora designada pelo Colegiado do Programa de Pós-Graduação em INFORMÁTICA da Universidade Federal do Paraná foram convocados para realizar a arguição da Dissertação de Mestrado de **MATHEUS AGIO NERONE** intitulada: **Cracking KD-Tree: The First Multidimensional Adaptive Indexing**, após terem inquirido o aluno e realizado a avaliação do trabalho, são de parecer pela sua aprovação no rito de defesa.

A outorga do título de mestre está sujeita à homologação pelo colegiado, ao atendimento de todas as indicações e correções solicitadas pela banca e ao pleno atendimento das demandas regimentais do Programa de Pós-Graduação.

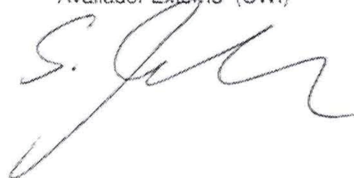
Curitiba, 03 de Outubro de 2018.



EDUARDO CUNHA DE ALMEIDA
Presidente da Banca Examinadora (UFPR)

ANDRÉ RICARDO ABED GRÉGIO
Avaliador Interno (UFPR)

STEFAN MANEGOLD
Avaliador Externo (CWI)



"We don't need bigger cars or fancier clothes. We need self-respect, identity, community, love, variety, beauty, challenge and a purpose in living that is greater than material accumulation." - Donella Meadows

Agradecimentos

Firstly, I would like to thank my family for giving me support throughout my whole life, without them nothing that I made would be possible. I also would like to thank my friends for helping me with questions relevant to my work, and for all the laughs that make for a better life.

Special thanks to my advisor, Prof. Dr. Eduardo Cunha de Almeida, for accepting me as a student and for guiding me with patience and motivation. My Master's Degree would not be possible without his valuable help.

I also would like to thank my dissertation committee, Prof. Dr. André Ricardo Abed Grégio, Prof. Dr. Stefan Manegold and Prof. Dr. Marcos Didonet Del Fabro, for the words of encouragement and insightful comments, and also for the valuable questions that can help me improve on my research.

Finally, I would like to thank anyone that somehow had contact with me, in a way or another they helped me be who I am today.

Resumo

A criação de índices é um das decisões mais difíceis no processo de criação de esquemas em bancos de dados. Dada uma carga de trabalho, o administrador do banco de dados precisa decidir quais índices criar levando em consideração os custos para construção e manutenção deles. Esse problema se torna ainda mais difícil quando é necessário lidar buscas em múltiplas dimensões em sistemas exploratórios, onde não se tem uma carga de trabalho disponível e o número de possíveis índices é ainda maior. Técnicas de indexação adaptativas, como *Sideways Cracking* e *Quasii*, são capazes de responder buscas de intervalo em múltiplas dimensões. Nessa dissertação nós propomos uma alternativa, a *Cracking KD-Tree*, que é uma estrutura de dados adaptativa usada para buscas em múltiplas dimensões. Comparando-a com outras técnicas adaptativas de indexação, nossa estrutura de dados teve eficiência melhor ou comparável, com respeito a tempo total de resposta para executar a carga de trabalho. Com 2 atributos nós fomos 6.7x mais rápidos que o *Sideways Cracking* e 1.4x que o *Quasii*. Com 16 atributos, a *Cracking KD-Tree* foi 19x mais rápida que o *Sideways Cracking* e 1.7x mais rápida que o *Quasii*.

Palavras-chave: Particionamento de Banco de Dados. Índice Multidimensional. Banco de Dados.

Abstract

Index creation is one of the main difficult decisions in database schema design. Given a workload, the database administrator has to decide which indexes to create taking into consideration the costs to build and maintain them. This problem becomes even more difficult when dealing with multidimensional queries in exploratory systems, where there is no workload available and the number of possible indexes is bigger. State of the art adaptive indexing techniques, such as Sideways Cracking and Quasii, are capable of answering multidimensional range queries. In this dissertation we propose an alternative, the *Cracking KD-Tree*, which is an adaptive data structure used for multidimensional queries. Comparing it with other adaptive indexing techniques, our data structure had more or comparable efficiency with respect to total workload response time. With 2 attributes we were 6.7x faster than Sideways Cracking and 1.4x than Quasii. With 16 attributes, the Cracking KD-Tree was 19x faster than Sideways Cracking and 1.7x faster than Quasii.

Keywords: Database Cracking. Multidimensional Index. Database Systems.

Lista de Figuras

1.1 Example of how a MDRQ would be executed by multiple one-dimensional indexes. 14

1.2 Example of how the Standard Cracking deals with multiple columns. 14

1.3 Comparison between a Full Scan, a KD-Tree and Database Cracking to answer a workload with only 2 columns. 15

2.1 Differences in data storage between NSM and DSM. 16

2.2 Example of the cracking process. 17

2.3 Left side: Table with Bitmap indexes on all attributes. Right side: Bitmaps used to answer a MDRQ. 19

2.4 Example of Grid File in a 2*d* space. 20

2.5 Example of R-Tree in a 2*d* space. 20

2.6 Example of Ball Tree in a 2*d* space. 21

2.7 Example of QuadTree in a 2*d* space. 22

2.8 Example of Multidimensional Range Tree using 8 tuples with 2 attributes. 23

2.9 Example of false positives when answering a query using Space-filling Curves in a 2*d* space.. 24

2.10 Example of how hash-like methods could work using database cracking. 25

2.11 Process of cracking in the Grid File. 25

2.12 Example of *cracker map* on attributes A and B. 27

2.13 Example of query on sideways cracking. 27

2.14 Example of multidimensional query on sideways cracking, adapted from [21]. . . 28

2.15 On the left side: points on a 2*d* space. On the right side: objects with area on a 2*d* space. 29

2.16 Abstract image of Quasii index structure. The white triangles represent the same structure the other nodes have, but were used because the lack of space. 29

2.17 Quasii cracking process. In the last picture, the min-max slices were simplified for lack of space. 31

2.18 Quasii search process.. 31

3.1 Example of KD-Tree. 33

3.2 Example of KD-Tree construction using medians. 34

3.3 Possible different outcomes of comparing a value to a range. 35

3.4 Two examples of the KD-Tree searching process demonstrating the difference between having to search one and multiple partitions. 36

3.5 Example of query transformation process on disjunctive searches. Note that the partitions in green still need to be scanned for the correct tuples. 37
3.6 Process of cracking the data using the query $4 \leq X < 7$ AND $1 \leq Y$. 38

4.1 Full Scan example. 41
4.2 Workload response time when changing the number of attributes.. 42
4.3 Time per query. 42
4.4 Total response time breakdown. 43
4.5 Response time per query. 44
4.6 Accumulated response time. 45

Lista de Tabelas

2.1 Brief explanations of why each multidimensional data structure was discarded as an adaptive index. 26

2.2 Advantages and disadvantages of each state of the art technique. 32

Lista de Acrônimos

DINF	Departamento de Informática
PPGINF	Programa de Pós-Graduação em Informática
UFPR	Universidade Federal do Paraná
MDRQ	Multidimensional Range Query
BST	Binary Search Tree
NSM	N-ary Storage Model
DSM	Decomposed Storage Model

Lista de Símbolos

τ	Quasii maximum number of objects in a slice in the last dimension.

Sumário

1	**Introduction**	**13**
2	**Related Work**	**16**
2.1	Adaptive Indexing	17
2.2	Multidimensional Index Structures	18
2.3	Sideways Cracking	26
2.3.1	Basic Concepts	26
2.3.2	Multidimensional Range Queries	27
2.4	Quasii	28
2.4.1	Basic Concepts	29
2.4.2	Cracking and Searching Process	30
2.5	Conclusion	31
3	**Proposed Work**	**33**
3.1	Cracking KD-Tree	33
3.1.1	Construction	34
3.1.2	Search	34
3.1.3	Disjunctive Searches	36
3.1.4	Adaptive Indexing	37
3.2	Search Complexity	38
3.3	Drawbacks	39
4	**Experiments**	**40**
4.1	Setup and Algorithms	40
4.2	Experiment Discussion	41
5	**Conclusion and Future Work**	**46**
	Referências	**48**

1 Introduction

An index is an access method used to retrieve data efficiently. Index creation is one of the main difficult decisions in database schema design. Based on a given workload, the database administrator must decide which indexes should be created, taking into consideration the costs of fully building and maintaining the index.

Indexes are useful to point out data qualified by a filter, for example, using a B+-Tree to find every person with 25 years instead of scanning the data, optimizes the number of comparisons from $O(N)$ to $O(\log N + K)$ operations, where N is the data size and K is the number of tuples that correctly answer the filter. Indexes can also be used for searches with inequalities preserving the same optimization, for example, finding every company with *more than* 50 employees would take $O(\log N + K)$ operations using a B+-Tree.

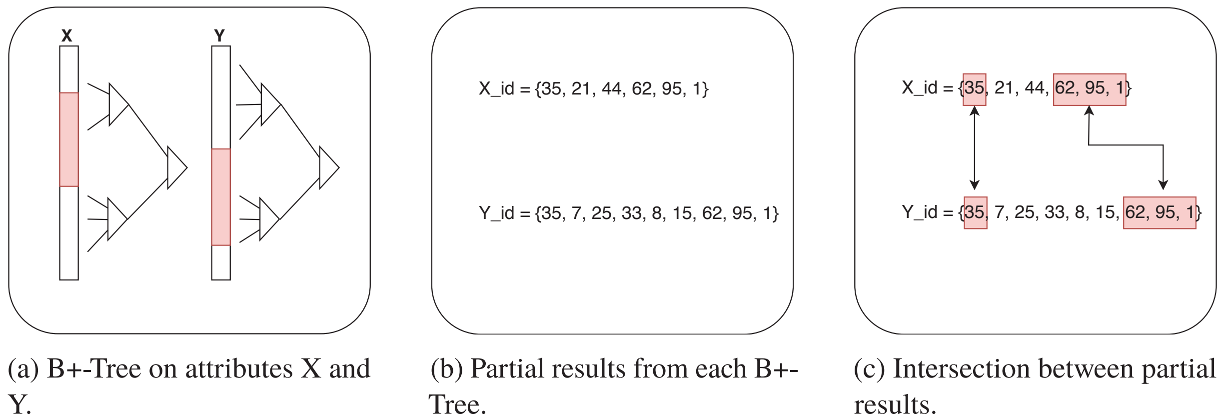
Filters can have one or two inequalities per attribute, e.g., find every person with height *between* 170 cm and 200 cm. This kind of search is called range query. A range query is an operation which retrieves every record that lies between two values. It has four possible operators: <, <=, >=, >.

A range query has filters on only one attribute, however, another class of searches, called multidimensional range queries (MDRQ), has range filters on multiple attributes, for example, find every person with age *between* 25 and 30 years *and* salary *between* \$15.000 and \$20.000. Listing 1.1 shows the difference between a regular range query and a multidimensional one. The problem is that indexes regularly used inside database systems (e.g., B+-Tree) have low efficiency when answering multidimensional range queries.

```
1  SELECT COUNT(R.C1) FROM R WHERE
2  10 ≤ R.C1 < 70   \% Range query
3
4  SELECT COUNT(R.C1) FROM R WHERE
5  10 ≤ R.C1 < 70 AND
6  40 ≤ R.C2 < 500 AND ... \% Multidimensional Range query
7  50 ≤ R.CN < 80
```

Listing 1.1: Range query vs Multidimensional Range query.

Figure 1.1 depicts an example of multiple one-dimensional indexes answering a MDRQ. Figure 1.1(a) represents an index on each attribute, and the corresponding portions of the columns that answer a generic multidimensional range query. Each index finds a different set of IDs, as represented in figure 1.1(b), where X_{ID} and Y_{ID} are sets for columns X and Y, respectively. Then, a costly intersection between all sets is necessary to find the IDs that answer the query, as shown in Figure 1.1(c). This intersection would not be a problem if each set of IDs was small, however, multidimensional range queries have a tendency to have high selectivity per column, creating huge intermediate results, and a small query selectivity, which renders one-dimensional indexes worse than simply scanning the data.



(a) B+-Tree on attributes X and Y. (b) Partial results from each B+-Tree. (c) Intersection between partial results.

Figura 1.1: Example of how a MDRQ would be executed by multiple one-dimensional indexes.

In order to alleviate the problem of searching on multiple attributes without losing efficiency, multidimensional data structures, like the KD-Tree, R-Tree, QuadTree, Gridfile and others, have been proposed. These structures can search on more than one key without having to intersect the points.

When used as an index in a database system, they differ from one-dimensional data structures, in that they allow searches on any of the indexed columns without an intersection phase, while the AVL-Tree, B+Tree and Encoded Bitmap, for example, only allows searches on the indexed columns using a costly intersection.

The problem of index creation becomes even harder on exploratory systems, where there is no time to create the indexes beforehand and there is no workload information whatsoever.

Adaptive indexing techniques, such as Database Cracking [20], attempt to solve the problem of up-front index creation by reorganizing relational databases as a byproduct of query execution. They work by physically ordering each column using query predicates as hints, resulting in a binary search tree (BST) to keep track of the cracked pieces. However, database cracking is not suited to manage MDRQs because each column is cracked separately from the others, creating different BSTs, and to obtain the final result, it is necessary to intersect the partial result from each index. Figure 1.2 depicts an example of a range query in a cracked database, that selects attributes of two different columns. As we can see, both columns are copied and cracked separately, resulting in two different BST. After the cracking step, it is still necessary to intersect the results of each index, i.e., the intersection between sets X_{ID}=(10, 9, 8) and Y_{ID}=(8, 9, 10, 1, 2).

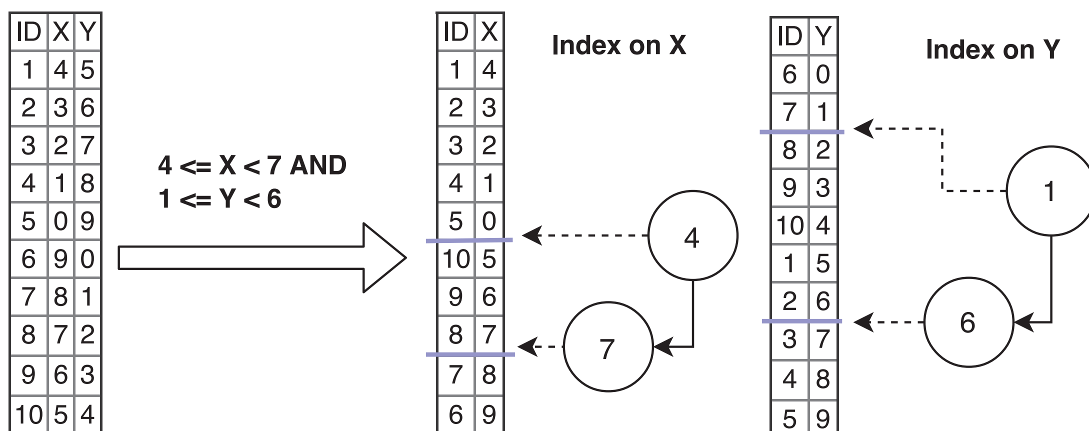


Figura 1.2: Example of how the Standard Cracking deals with multiple columns.

Figure 1.3 demonstrates the issue, using a workload with only 2 attributes and 1000 queries with 20% selectivity per column. Database Cracking was twice less efficient than a Full Scan, however, using a regular KD-Tree, instead of multiple AVL Trees, is more than twice as fast as the Full Scan.

In order to deal with MDRQs, a cracking variation called Sideways Cracking [21] was proposed. However, sideways cracking seems to be a lot more useful when dealing with tuple reconstruction problems than when dealing with MDRQ. We discuss their data structures in Section 2.

The problem of dealing with filters on multiple columns becomes more costly as the number of columns involved gets higher.

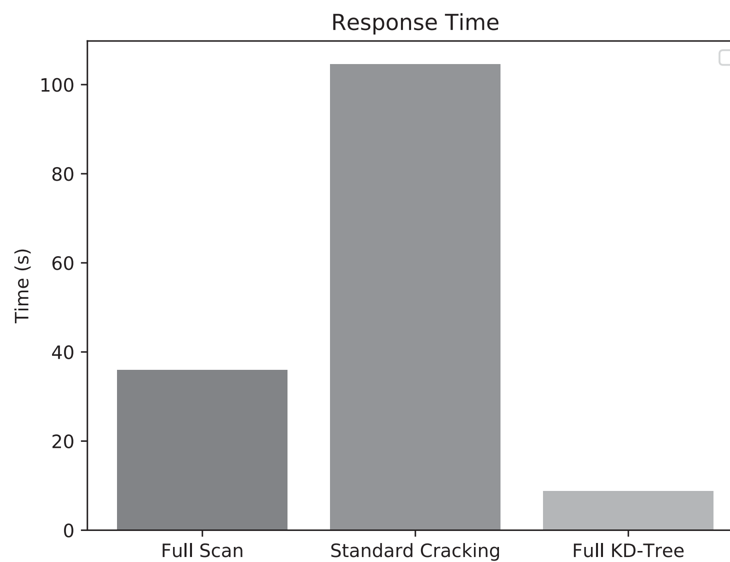


Figura 1.3: Comparison between a Full Scan, a KD-Tree and Database Cracking to answer a workload with only 2 columns.

To address this issue we propose the *Cracking KD-Tree*, a data structure to index multiple columns using database cracking. The Cracking KD-Tree is a binary search tree, where each node contains a column, a key, a position and two children. Our data structure works in the same way as a regular KD-Tree when searching and the major difference lies in the approach to create the index. The Cracking KD-Tree is built as a byproduct of query processing, i.e. given a query with ranges on multiple columns on the same table, we insert into the structure every search key of every range, instead of creating a separated index for each column.

The main contributions of this dissertation are:

- We introduce a novel adaptive indexing technique that is able to handle MDRQs.
- We investigate its performance by evaluating our work against other existing indexing techniques.

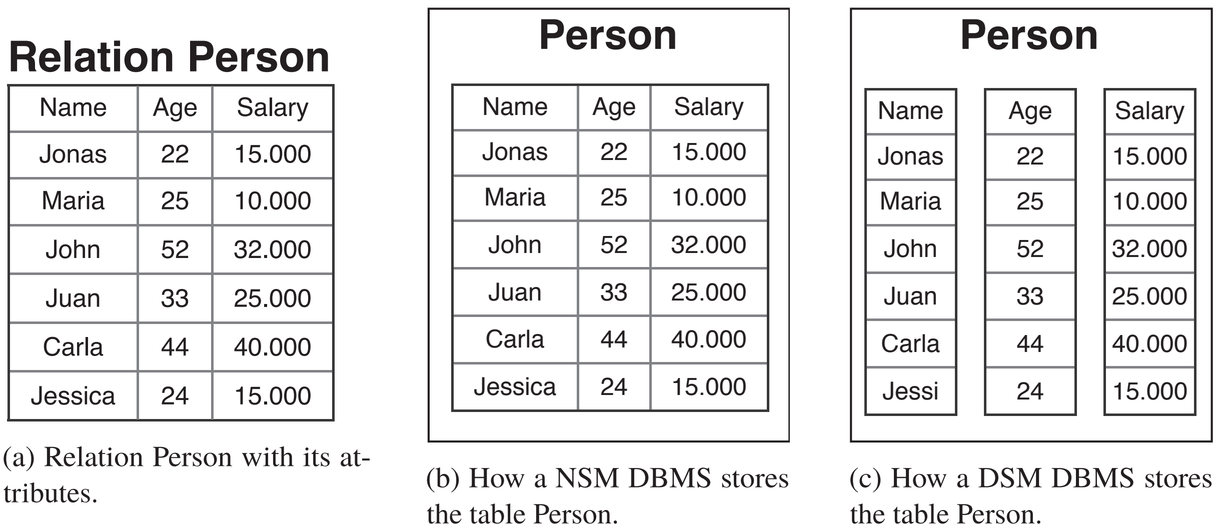
The remainder of this dissertation is organized as follow. In Chapter 2 the related work is presented. Chapter 3 introduces our proposed solution. In Chapter 4 the results obtained are discussed and analyzed. Lastly, in Chapter 5, we make our final conclusions and present future work.

2 Related Work

In this chapter, we discuss different automatic physical tuning methods proposed in the literature, analyzing how they behave in a multidimensional level in a exploratory system. We also study multidimensional index structures found in the literature, such as, R-Tree, Quad Tree, Octree and others. Finally, we analyze two state of the art methods that are very similar to ours.

In this dissertation we assume the Data Warehouse environment, where there is almost zero modifications on the data, and lots of data analysis. A database on such environment usually have a start schema, where there is one big fact table, with roughly 100 attributes, and dimension tables with lot more attributes, but less tuples, that are linked to the fact table through the usage of foreign keys. Queries on Data Warehouses are usually big read operations on four or five attributes.

We make use of range queries to implement the adaptive indexing techniques, these queries are common in Data Warehouse environments and read-intensive workloads alike, and reproduced in many benchmarks: TPC-H [38], TPC-DS [38], SetQuery Benchmark [35] and YCSB [11]. For instance, of the 22 queries in the TPC-H benchmark 12 have at least one part that is a range query. Going even further, of the 12 queries, 3 are range queries on more than one attribute.



Relation Person

Name	Age	Salary
Jonas	22	15.000
Maria	25	10.000
John	52	32.000
Juan	33	25.000
Carla	44	40.000
Jessica	24	15.000

(a) Relation Person with its attributes.

Person

Name	Age	Salary
Jonas	22	15.000
Maria	25	10.000
John	52	32.000
Juan	33	25.000
Carla	44	40.000
Jessica	24	15.000

(b) How a NSM DBMS stores the table Person.

Person

Name
Jonas
Maria
John
Juan
Carla
Jessi

Age
22
25
52
33
44
24

Salary
15.000
10.000
32.000
25.000
40.000
15.000

(c) How a DSM DBMS stores the table Person.

Figura 2.1: Differences in data storage between NSM and DSM.

Since we are in a Data Warehouse environment, we also assume columnar databases, or DSM (Decomposition Storage Model) [12], which are the ones used in this case. Also, row databases, or NSM (N-ary Storage Model), which are the traditional storage layout in database systems, cannot have more than one physical index, so it is hard to implement adaptive indexing techniques discussed here on them.

Figure 2.1 shows the difference between NSM and DSM. Figure 2.1(a) depicts relation Person with three attributes: Name, Age and Salary. In a NSM database (Figure 2.1(b)) each tuple is stored continuously, when one attribute is accessed the entire tuple comes to memory. On the other hand, (Figure 2.1(c)) DSM databases are vertically partitioned, having each attribute stored separately, differently from NSM, only the necessary attributes comes to memory when requested. Since attributes in columnar databases are stored separately, each one of them can have a different physical organization, enabling multiple physical indexes. On contrary, row databases can only have one physical index because all attributes need to be contiguous.

2.1 Adaptive Indexing

Traditional database indexes are build under two base assumptions: the workload is known, and there is sufficient idle time to create and update the indexes. Nowadays, these assumptions are not valid anymore, environments have continuous and sudden workload shifts and updates, and the data is queried as soon as it arrives [28]. In order to mitigate these problems, Database Cracking [20] implements the idea of index maintenance as a byproduct of query processing.

In Database Cracking, the first time an attribute is touched by a query, a copy of its column is created, called *cracker column*. Then this *cracker column* is refined by every subsequent query that touches it, by a process called *cracking*, hence the name Database Cracking. To keep track of every partition created by the *cracking* process, a *cracker index* is created.

Since the column is refined throughout the query workload, and only on relevant tuples, the cost of creating the index is spread in the stream of queries, and the overhead on each query is minimal.

Figure 2.2 depicts an example of the cracking process, when the query first touches attribute X, the column is copied and cracked based on the predicates on the query, i.e., predicate $4 \leq X < 7$. When the next query arrives, $1 \leq X$, the *cracker column* is cracked again based on predicate $1 \leq X$. As one may notice, the whole process of Database Cracking can be seen as a lazy Quicksort [18], where predicates act as pivots, and in the sense that ordering steps are only executed when needed.

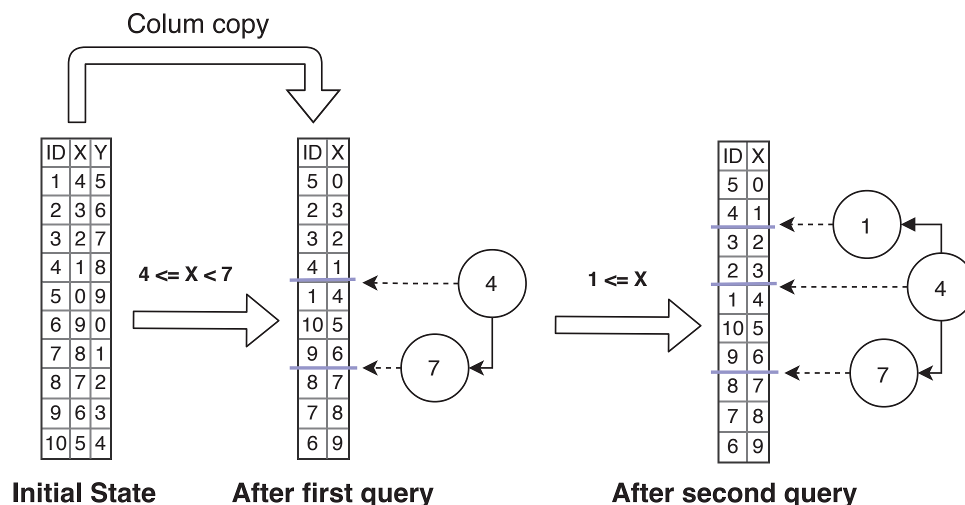


Figura 2.2: Example of the cracking process.

As explored in [40] Database Cracking method has three serious drawbacks: its time to converge to a full index may be high (convergence), variance of response time (robustness) and projections or selections on other columns (tuple-reconstruction). Three different advanced cracking algorithms were proposed with these problems in mind, namely: Hybrid Cracking [22], Stochastic Cracking [17] and Sideways Cracking [21].

While aiming to improve the convergence time concern, Hybrid Cracking [22] combines adaptive merging [15], which has a high initialization cost but converge rapidly to a full index, with database cracking, which has a low initialization cost but has a low convergence rate, in a way that the advantages of both algorithms complement themselves.

Cracking by its own nature depends on boundaries of the queries, for example on Figure 1.2, boundaries are the values 4, 7, 1 and 6. In case of a skewed workload only parts of the cracker column will go through the process of cracking. Stochastic Cracking [17] introduces additional arbitrary crack operations besides the ones given by the query, hence the name stochastic, in a way that cracking operations will be more distributed, and so making the algorithm more robust by diminishing variance of response time.

Standard Cracking [20], Hybrid Cracking [22] and Stochastic Cracking [17] suffer negative performance hits when dealing with MDRQ. Since all three methods make use of one dimensional indexes, when answering a range query on more than one attribute, they can end up creating huge intermediate results that have a high cost to intersect.

Imagine the following scenario, a relation with 1 billion tuples, and 8 attributes. Also a query that selects on all attributes and has selectivity per column equals 20%, which results in a query selectivity roughly of 0.2^8. The size of all 8 intermediate results would be 200.000.000 tuples, and the size of the final result would be 2560 tuples.

Sideways Cracking [21] comes in hand when dealing with multiple projections and/or selections on the same relation. It will be discussed further in section 2.3.

2.2 Multidimensional Index Structures

As stated in [39], multidimensional data is used in a variety of fields, e.g. computer graphics, geometric information systems, robotics, spatial databases and multimedia databases, to name a few. A wide range of data structures were proposed for dealing with the problem of records identified by one single key, most of them fall into one of these categories: tree like, hash like, sequentially allocated arrays. The same happens with multidimensional data structures, where more than one attribute can be used for search.

Bitmap indexes [34, 32, 33, 8, 9, 45, 2] are useful when processing complex queries, providing fast read operations for range and equality queries in one or more dimensions [42]. Bitmaps have 1 on positions that have the attribute equal to its key, and 0 otherwise. In Figure 2.3, on the left side, the age's bitmap with key equal to 20 has only the first two positions set to 1, because only the first and second tuples have age attribute equal to 20.

Figure 2.3 shows an example of how bitmaps can be used to answer MDRQs. On the left side, we have a table with bitmaps on every attribute (Age and Salary). Given a MDRQ that selects every tuple with $17 \leq age < 27$ and $2000 \leq Salary < 3500$. The method first finds which age's bitmaps are between $17 \leq age < 27$ (i.e., bitmaps 20 and 25). Since any position with 1 correctly answers the filter, the found bitmaps are intersected using an OR operation, creating the partial result *R1*, as depicted in the top box on the right side of the figure. Now, the method selects which salary's bitmaps are between $2000 \leq Salary < 3500$ (i.e., only bitmap

3000) and proceeds to intersect them with an OR operation creating the partial result *R2*[1], as depicted in the middle box. Finally, all partial results, *R1* and *R2*, are intersected using an AND operation (an AND operation is used because a tuple's position has to answer correctly every filter), creating the *final* result with 1's on the positions that answer the query, as depicted in the bottom box of the figure.

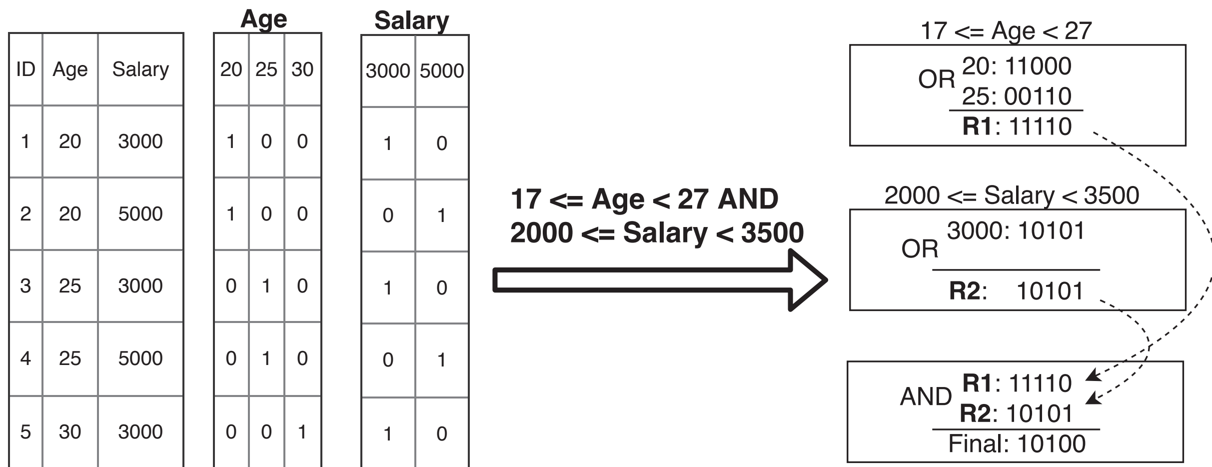


Figura 2.3: Left side: Table with Bitmap indexes on all attributes. Right side: Bitmaps used to answer a MDRQ.

The problem with bitmaps comes from the fact that they are specialized into some situations, i.e., attributes with high cardinality would create a large quantity of bitmaps. Another problem arises with workloads that demand management of the index, which are costly to do. Finally, bitmaps are full indexes, and need to be created upfront, before starting answering queries.

Hash like structures, like Grid File [30], partition the space in *grid cells*, where each of these *cells* contains the data points. EXCELL method [43] works in the same way as the Grid File, the major difference is how the partitions are created. In EXCELL, all cells have the same size, while in the Grid File this is not a necessity. Figure 2.4 presents a view of a Grid File.

[1]Since there is only one bitmap, the result of the intersection is the bitmap itself.

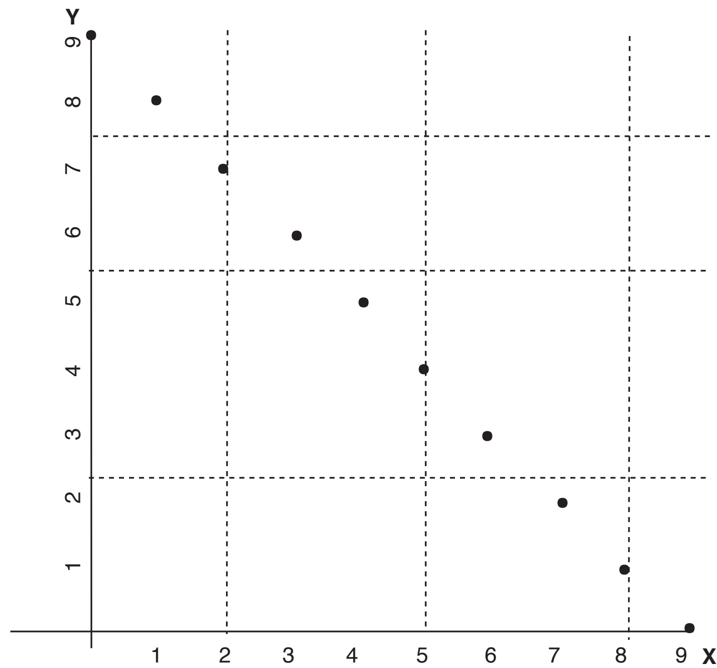


Figura 2.4: Example of Grid File in a 2*d* space.

Into tree like structures, R-Tree [16] was proposed for dealing with the problem of searching and updating an index with data objects with size greater than zero within a multidimensional space, e.g. the representation of a country in a map has an area. Figure 2.5 depicts an example of the R-Tree. There are other variations of the R-Tree, namely, R+-Tree [41], R*-Tree [3] and Priority R-Tree [1]. The X-Tree [6] works in pretty much the same fashion as the R-Tree, but it emphasizes prevention of overlapping bounding boxes.

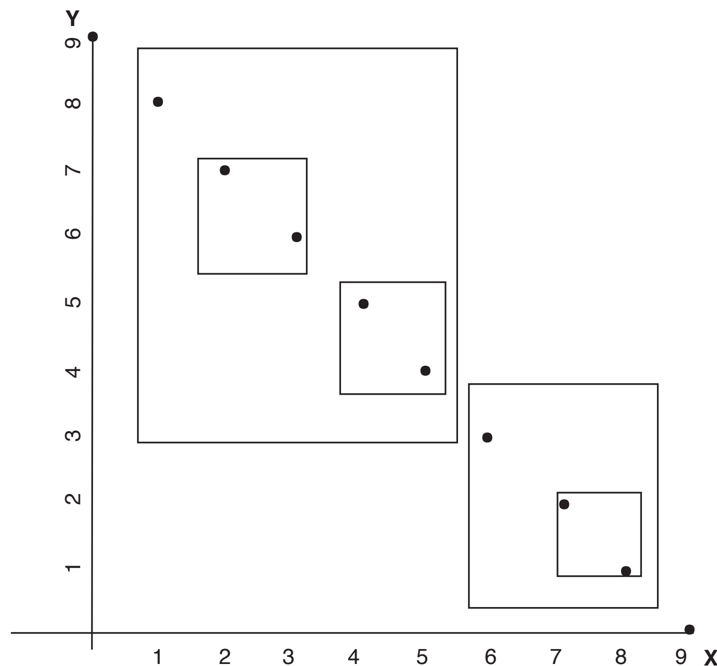


Figura 2.5: Example of R-Tree in a 2*d* space.

Vantage-point Tree [44, 46], Ball Tree [31, 25] and M-Tree [10] are all examples of trees that use hyperspheres to partition space instead of using hyperplanes or bounding boxes. Figure 2.6 shows an example.

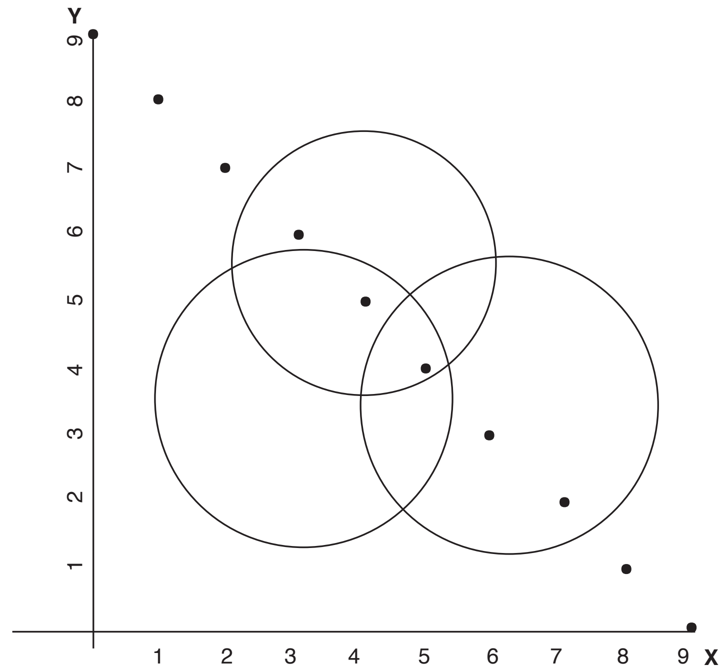


Figura 2.6: Example of Ball Tree in a 2d space.

QuadTree [14] is a data structure specialized for two dimensions, but can be generalized for any number. In a Quadtree each node represents a data point and has four children, that divide the space in four quadrants, i.e. NE, NW, SW and SE (using a geographic analogy), as we can see in Figure 2.7. The Octree [27] works in the same way as the Quadtree but is specialized for 3 dimensional spaces.

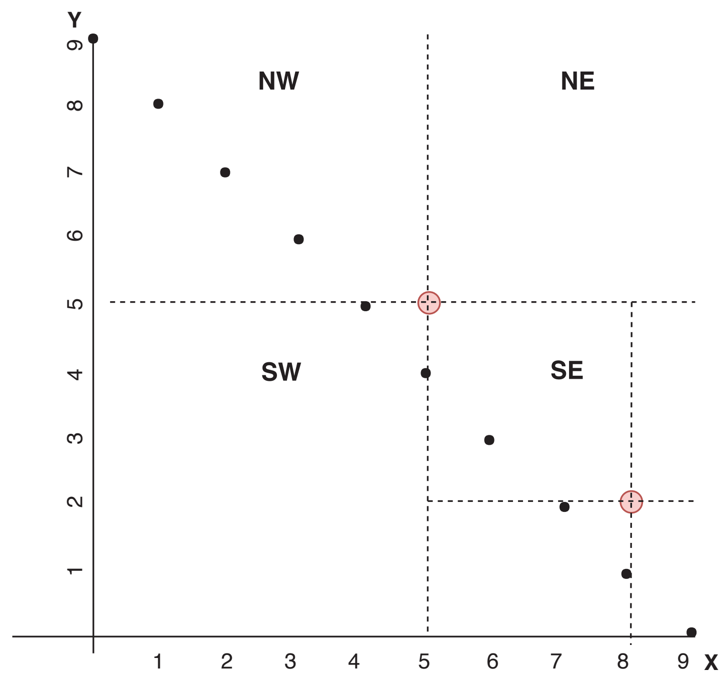
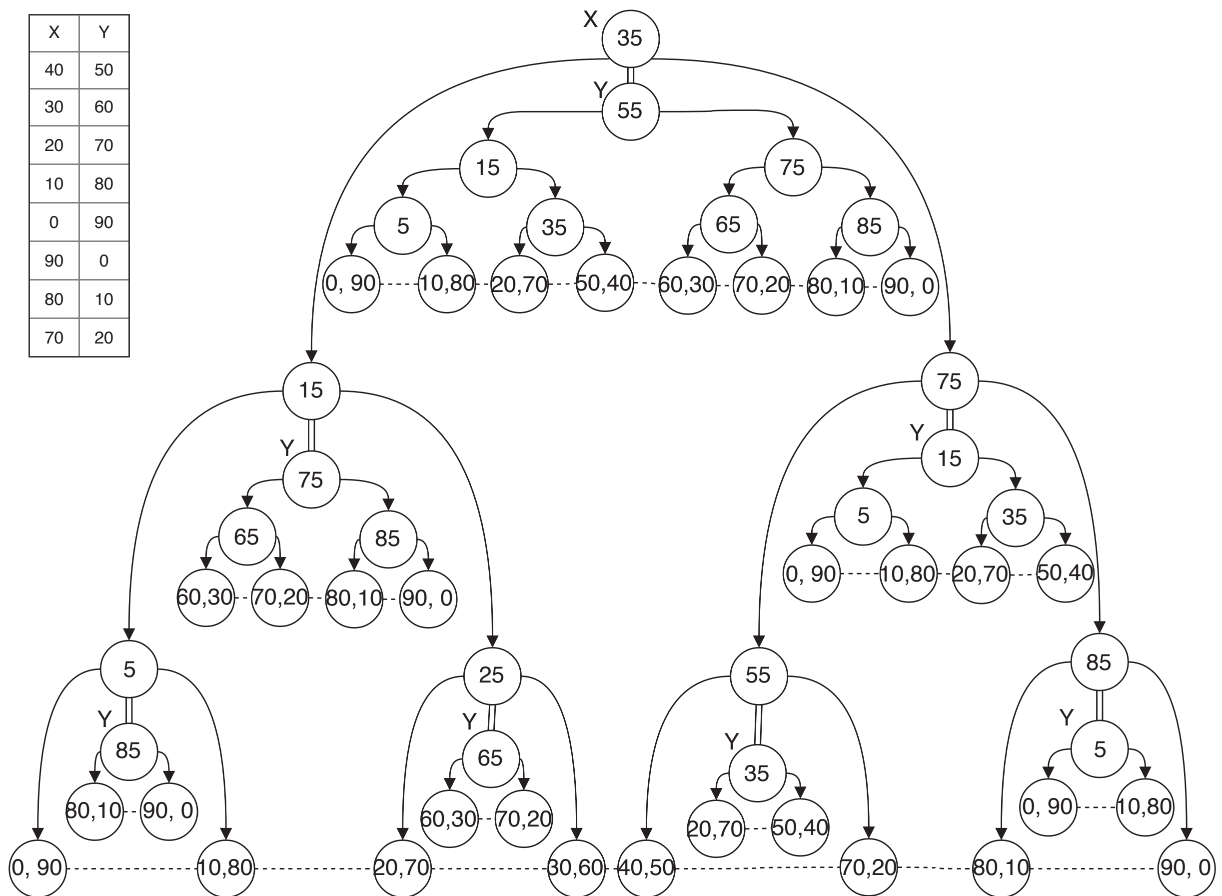


Figura 2.7: Example of QuadTree in a $2d$ space.

The Multidimensional Range Tree [26, 4] is a data structure asymptotically faster than the Quadtree and the KD-Tree for searching, although it has a significant higher space demand. Figure 2.8 depicts an example, the tree is formed by first ordering all points along one of the attributes, and then storing them in the leafs like a balanced binary search tree, but every node also contains another tree ordering points based on other attribute. Also, for every tree, the leafs are connected like in a B-Tree.



X	Y
40	50
30	60
20	70
10	80
0	90
90	0
80	10
70	20

Figura 2.8: Example of Multidimensional Range Tree using 8 tuples with 2 attributes.

It is also possible to use *space-filling curves* to order points. The ordering is basically a mapping from a D dimensional space to a one dimensional value. It is important to choose a transformation that preserves the proximity of the points, such as the Z-order [36] and the Hilbert curve [23], so that points that are close in the multidimensional level, are close in the one dimensional space [13, 29].

The biggest concern about space-filling curves is that, depending on which transformation was chosen, there can be a high number of false positives that need to be checked on every query. As we can see in Figure 2.9, the black line represents the transformation from the $2d$ space to a $1d$ space, the red box is the query, and the yellow lines are the false positives that need to be checked. Besides, there is also an overhead of having to do the transformation, which can be costly and goes against the idea of lightweight indexes.

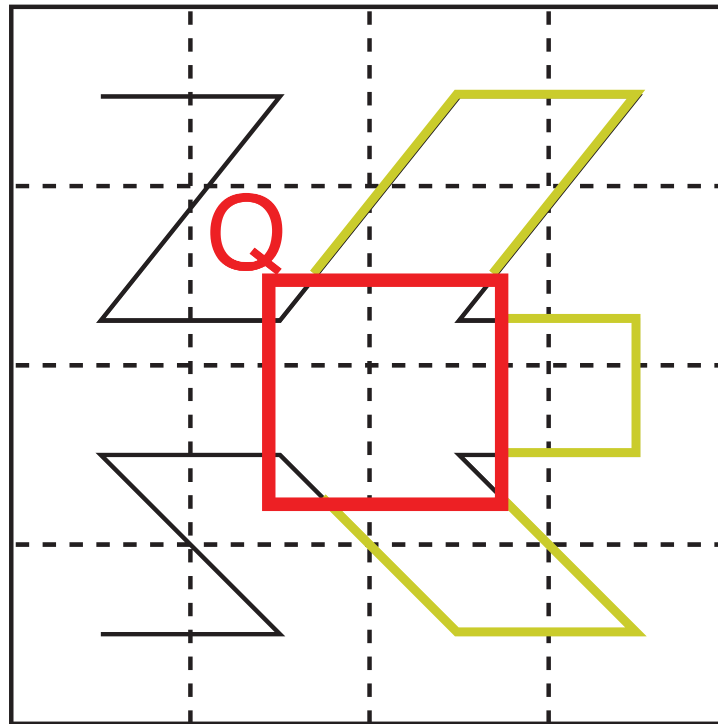


Figura 2.9: Example of false positives when answering a query using Space-filling Curves in a 2d space.

The R-Trees variants and X-Tree are not suitable for our problem because they are specialized in dealing with non-zero size objects, whereas our research is specifically dealing with points (which have zero size).

The Quadtree, Octree, Vantage-point Tree, Ball Tree and M-Tree all use data points to partition the space, which means it is not possible to use parts of the query to crack the space, hence it is not possible to use them as a structure for cracking.

The Multidimensional Range Tree has a high space demand, as we can see in Figure 2.8, for only 8 tuples with 2 attributes, the tree has 56 nodes, which makes it infeasible for large quantities of tuples and/or for relations with lots of attributes.

Hash-like methods also could be expanded to an adaptive index, Figure 2.10 depicts an example, instead of creating *cells* during insertion of new points, they simply would be constructed based on the predicates of the incoming queries.

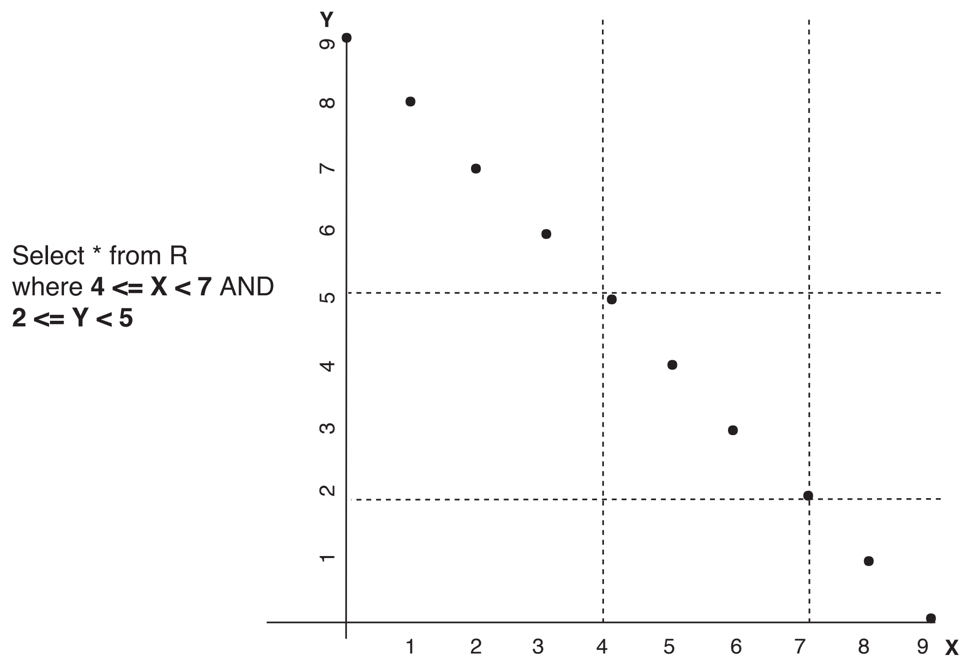


Figura 2.10: Example of how hash-like methods could work using database cracking.

Still, the EXCELL method would not be able to be used because it needs all *cells* to be the same size, which cannot happen only by using the predicates of queries.

The Grid File on the other hand, could be used, but it still has problems. Firstly, the more cracking operations are done more *cells* are created, and so, the next cracking operation will have a higher cost, instead of lower as expected in other structures. Figure 2.11 depicts an example, on the last step to create another division, red line, it would need to split 6 cells, which also leads to another problem. Which data structure to use? It is not an easy task to define a data structure for the *grid*. If a simple k-dimensional matrix is used, the split operations would have extremely high costs. Otherwise, if linked lists were used, maybe split operations would have low costs, but scan and read operations would not be efficient.

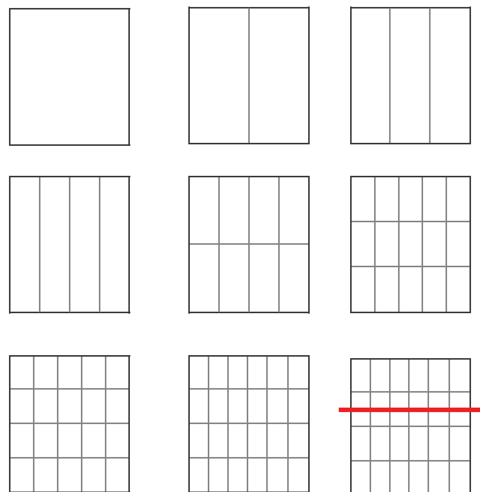


Figura 2.11: Process of cracking in the Grid File.

Notice that all multidimensional data structures cited above probably can be expanded to adaptive indexes, we only gave the reasons why they were not the first choice. Table 2.1 contains a brief explanation of each reason.

Algorithms	Reasons
Bitmaps	Specialized into some situations. Needs to be created upfront.
Space-filling Curves	High number of false positives. Mapping costs can be high.
R-Tree/X-Tree	Created for objects with size greater than zero, instead of points.
QuadTree/Octree Ball Tree Vantage Point Tree M-Tree	All make use of data points to partition the space, which means it is not possible to use parts of the query to crack the space.
Multidimensional Range Tree	Has an extremely high memory cost.
Hash-like Methods	The grid structure is not easy to implement in an efficient way.

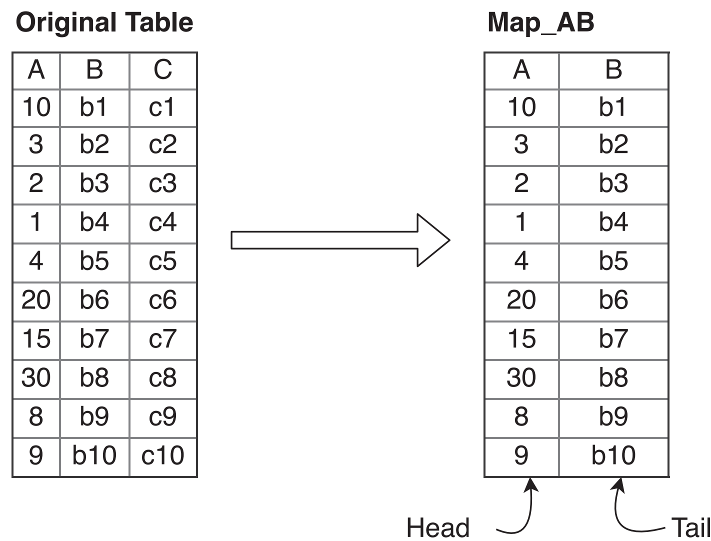
Tabela 2.1: Brief explanations of why each multidimensional data structure was discarded as an adaptive index.

2.3 Sideways Cracking

In this section we are going to study the Sideways Cracking method, which is one of closest methods to what our proposal tries to solve, i.e., how to perform adaptive indexing on multidimensional data.

2.3.1 Basic Concepts

Sideways Cracking [21] is a cracking technique capable of efficiently dealing with multiple projections and/or selections on the same relation. It introduces a new data structure called *cracker map*, where each *cracker map*, M_{AB}, consists of a fully materialized two-column table over two attributes, A and B, of the relation R. The left attribute, A, is called the *head*, and the right attribute, B, is called *tail*, Figure 2.12 depicts an example. All maps created using the same header belong to the same *map set*, for example, all maps using the header A belong to the set S_A. For each map there is a *cracker index* (AVL-Tree) that maintains information about the ordering of the map.

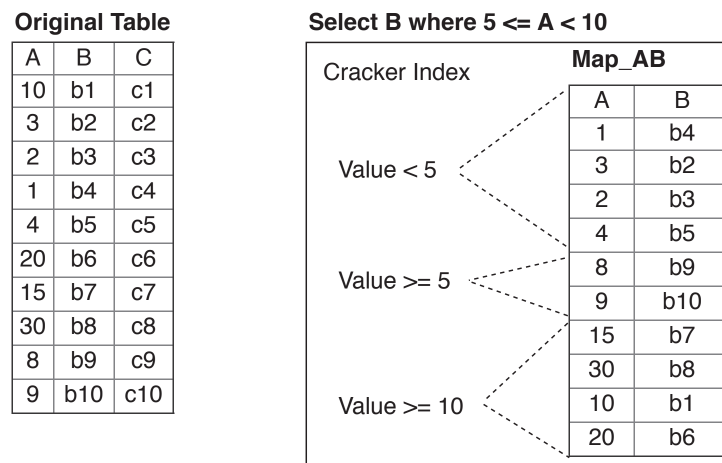


A	B	C
10	b1	c1
3	b2	c2
2	b3	c3
1	b4	c4
4	b5	c5
20	b6	c6
15	b7	c7
30	b8	c8
8	b9	c9
9	b10	c10

A	B
10	b1
3	b2
2	b3
1	b4
4	b5
20	b6
15	b7
30	b8
8	b9
9	b10

Figura 2.12: Example of *cracker map* on attributes A and B.

The maps are created only when requested, i.e., when a query needs to access attribute B based on a restriction on A, it first checks if the map M_{AB} exists, if not then it is created. It then cracks that map, based on the query predicates, Figure 2.13 shows an example, first the *cracker map* does not exist, so a copy of the two attributes is made. Then, the map proceeds to be cracked based on predicates of the query. And then the answer, $\{b9, b10\}$, is returned.



A	B	C
10	b1	c1
3	b2	c2
2	b3	c3
1	b4	c4
4	b5	c5
20	b6	c6
15	b7	c7
30	b8	c8
8	b9	c9
9	b10	c10

A	B
1	b4
3	b2
2	b3
4	b5
8	b9
9	b10
15	b7
30	b8
10	b1
20	b6

Figura 2.13: Example of query on sideways cracking.

All the maps of a set are aligned, which means they all share the same ordering. This is achieved by keeping a *cracker tape* on the set structure. Before doing the cracking step, there is an *alignment step* which applies all the physical reorganizations that happened to maps of that same set. This step is necessary since only the cracker maps accessed in the query are cracked, and so, the others will stay "behind" in the tape.

2.3.2 Multidimensional Range Queries

To answer MDRQs the method needs to find the best, usually the most selective, *set* and its *cracker maps* so that their alignment can be exploited. Figure 2.14 depicts an example. Given a table R with four attributes, A, B, C and D, and a query, **select** D **from** R **where** $3 < A < 10$

and $4 < B < 8$ **and** $1 < C < 7$, that access all of them, as depicted in Figure 2.14 (a). First, the algorithm finds the most selective attribute, for simplicity we defined it as attribute A. Then, it selects the set S_A and maps M_{AB}, M_{AC}, M_{AD}. Next, it aligns the maps to the latest crack operation, including the predicate $3 < A < 10$ in the query (notice the difference in order between the initial state and the maps in Figure 2.14 (b), (c) and (d)). After, the query is executed on each map, resulting in bit-vectors with same size, Figure 2.14 (b) for query on attributes A and B, and Figure 2.14 (c) for the query on attributes A and C. Finally, an AND operation is done between all bit-vectors and the resulting bit-vector is checked against the *cracker map* which contains the projection attribute, map M_{AD}, as depicted Figure 2.14 (d).

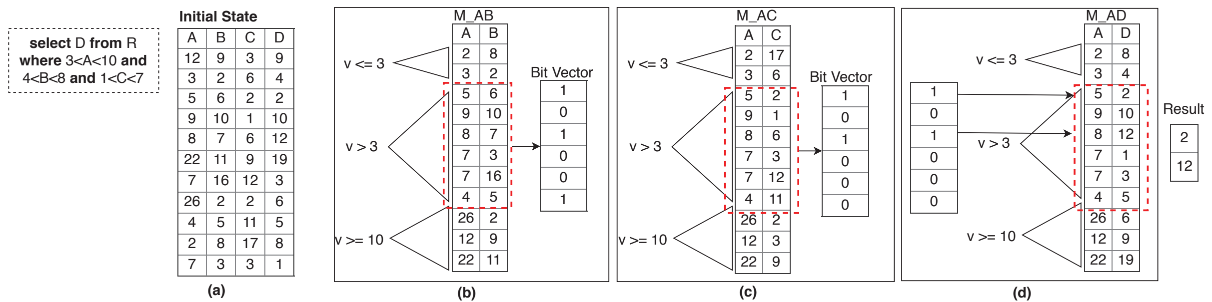


Figura 2.14: Example of multidimensional query on sideways cracking, adapted from [21].

Sideways Cracking improves a lot on Database Cracking by having good performance when projecting on attributes that are not the same as the cracked one.

However, when dealing with MDRQ, the maps may need to go through lots of cracking steps, which can result in a performance hit. Imagine that a map is far behind compared to the others in the set (i.e., it still has to go through lots of cracking steps until it is aligned with the others in the set), then it would need go through all cracking steps until it can be used.

Also, Sideways Cracking still suffers the same problem of huge intermediate results that Database Cracking suffers, however they mitigate some of it by using bit-vectors to do the intersection.

2.4 Quasii

Quasii [37] is another method that deals with multidimensional queries and data, but they differ from our proposal on which type of data they specialize. Figure 2.15 provides an example, Quasii deals specifically with objects in a space, for example, countries in a map. While our work deals only with points in a space. Still, with very few modifications Quasii can work with point data.

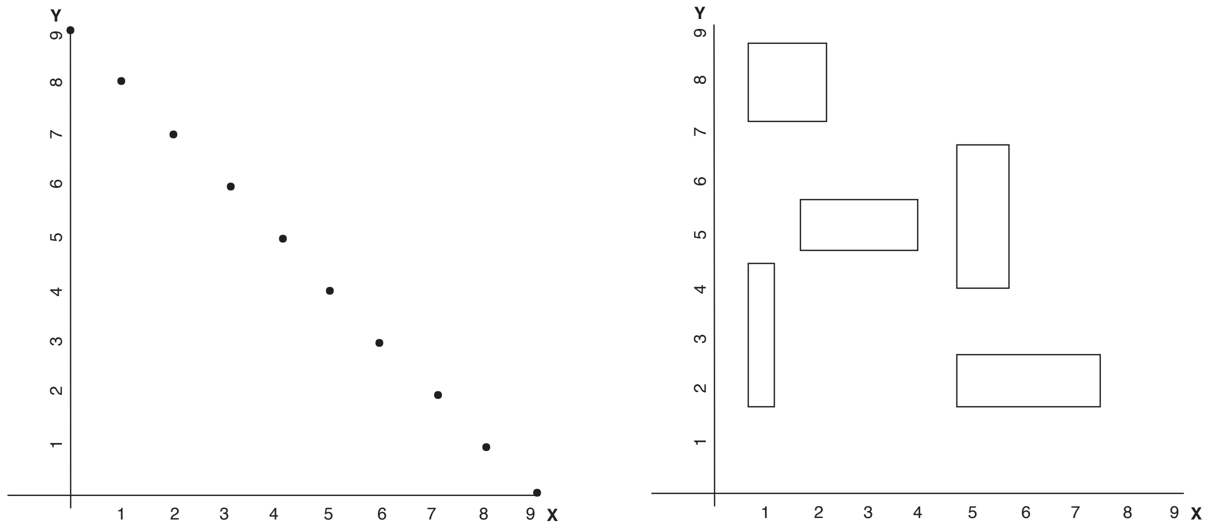


Figura 2.15: On the left side: points on a $2d$ space. On the right side: objects with area on a $2d$ space.

2.4.1 Basic Concepts

Although Quasii can be seen as an extension of the R-Tree to behave in an adaptive manner, it is not a tree. Quasii has a d-level hierarchical structure, where each level corresponds to one dimension. Each level has an array of slices, each slice contains: its level, a bounding box, pointers to first and last elements corresponding to the slice in the data array, pointers to sub-slices that refine the slice in the next dimension.

Figure 2.16 depicts an example using $3d$ data. Every slice segments the space on an attribute, for example, the first yellow slice is a segment that captures every tuple who has an X value between 20 and 50. It also contains an array of slices on the next attribute, for example, the first red slice is a segment on attribute Y that captures all the points that have Y value between 15 and 20 and X value between 51 and 60. It is important to notice that the arrays of segments in each slice is independent from the others in other slices.

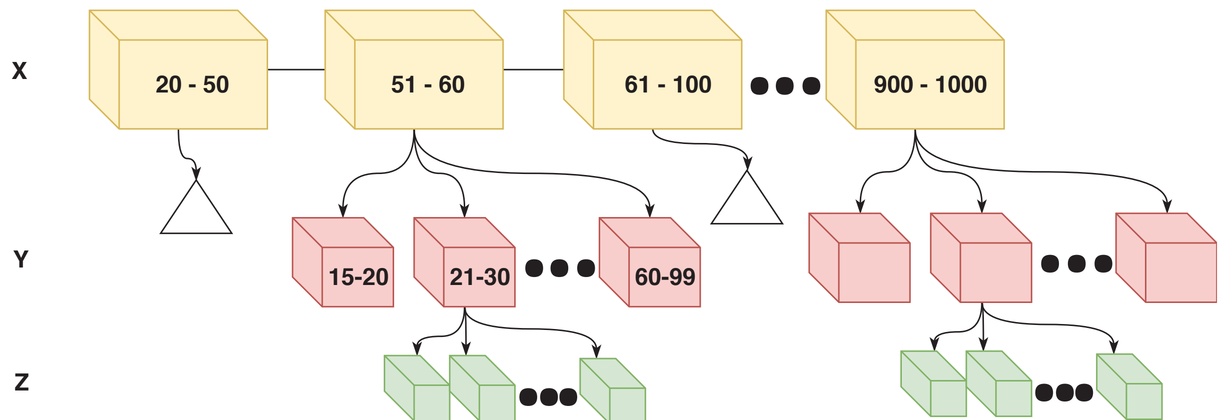


Figura 2.16: Abstract image of Quasii index structure. The white triangles represent the same structure the other nodes have, but were used because the lack of space.

2.4.2 Cracking and Searching Process

As Quasii is an adaptive index, it also has a cracking process that builds the index as a byproduct of query processing, utilizing the predicates. Quasii has only one configuration parameter, a size threshold τ, that determines the maximum number of objects in a slice on the last dimension, i.e., if a slice has more objects than τ it can still be refined. The maximum number of slices in the last dimension is defined by $\lceil n/\tau \rceil$, where n is the size of the dataset. The number of times Quasii has to slice each dimension to produce $\lceil n/\tau \rceil$ partitions at the bottom level is:

$$r = \left\lceil \sqrt[d]{n/\tau} \right\rceil \tag{2.1}$$

To calculate the maximum number of objects per slice on each level the following recursive expression can be used, note that $\tau_d = \tau$ (the slice threshold at the bottom level):

$$\tau_{d-1} = r * \tau_d \tag{2.2}$$

Quasii has three refinement methods. The first, called *Slice two way* happens when or the lower or the higher range of a query intersects a slice, then the slice is simply cracked into two pieces. *Slice three way* happens when both lower and higher ranges of a query are inside a slice, then the slice is cracked into three pieces. Finally, *Slice Artificial* happens when a slice intersects with a query but neither the lower nor the higher parts are inside the slice.

Slice Artificial is different from the other techniques because the predicates are unable to help. In order to crack, the method uses as pivot the middle value of the bounding box in its dimension, e.g., if it has values from 30 to 50, it uses 40 as pivot.

Notice that, when a slice has been refined in the next dimensions,i.e. it has children, it cannot be split again in its original dimension, since it would destroy all of its children. Because of that, during the cracking process every new slice created that intersects with the query has to be re-refined, using one of the techniques described above.

Figure 2.17 depicts an example using 2 dimensions. The initial state of the index has one slice that covers the entire data. When the first query arrives, SELECT * FROM R WHERE $25 \leq X < 50$ AND $60 \leq Y < 80$, it first creates three new segments, one covering from the minimum value until 25, one that from 25 until 50 and one from 50 until the maximum value. Then it proceeds to refine the middle segment, as it is the one that intersects with the query, creating three new segments on the Y attribute.

When a second query arrives, SELECT * FROM R WHERE $70 \leq X < 90$ AND $20 \leq Y < 30$, first it finds segments that intersect with the query on attribute X, only 50 - max, and then refines them. creating three new slices, 50 - 70, 70 - 90 and 90 - max, and again, it proceeds to refine the children of the ones that intersect with the query.

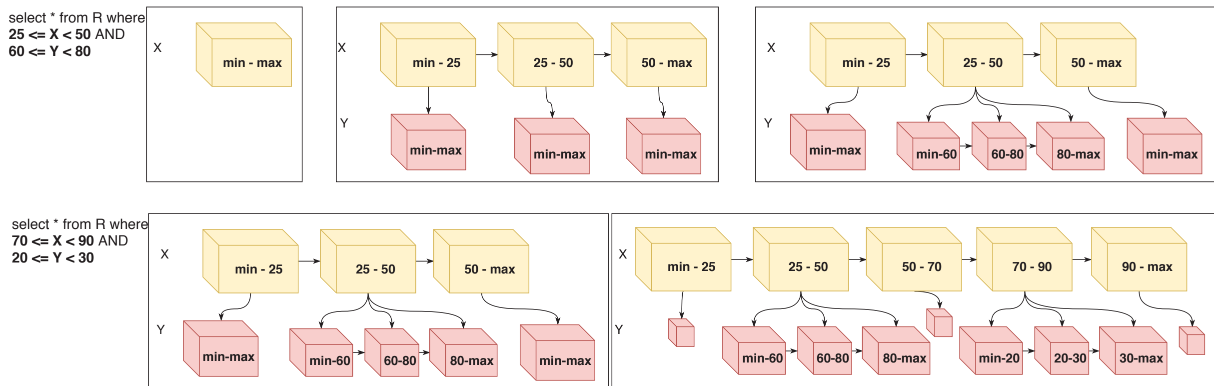


Figura 2.17: Quasii cracking process. In the last picture, the min-max slices were simplified for lack of space.

One great disadvantage of this cracking method is that if a query has 100% selectivity on every attribute it would create a full index, because of the refinement step that every new slice, that still intersects with the query, goes through.

The process of searching is straight forward, starting on the first dimension, it first needs to find the first slice that intersects with the query on that attribute, in our example (Figure 2.18) the slices 50 - 70 and 70 - 90 intersects with the query. Then for each slice the process is repeated on its children, in our example, for 50 - 70, there is only one child available, and for 70 - 90, the last two children intersect the query. When a leaf is found, then the data can be searched.

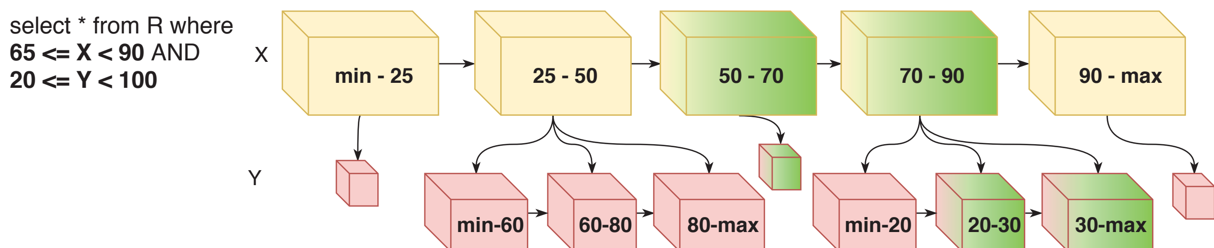


Figura 2.18: Quasii search process.

2.5 Conclusion

After analyzing the literature, we can see that there were efforts on creating data structures that do not discriminate between which attribute is being accessed, but they do not work on exploratory environments, where there is not time and reason to create an index ahead of time. On the other hand, there has been recent efforts to create adaptive indexes capable of dealing with exploratory environments, but the majority of them only deals with one dimensional range queries. There has been efforts to create adaptive indexes capable of dealing with MDRQ, Table 2.2 presents a brief explanation of each state of the art algorithm advantages and disadvantages. With that in mind, in the next chapter we present our proposal, an adaptive indexing technique that uses a multidimensional data structure capable of dealing with exploratory environments.

Algorithms	Advantages	Disadvantages
Sideways Cracking	Efficient tuple reconstruction. Capable of using already established research on one-dimensional adaptive indexes	Still need to use intersection techniques to answer MDRQs, which makes the technique less efficient
Quasii	Good efficiency on MDRQs Can be used with object data with size greater than zero	Cracking method can end up creating almost a full index, if the query has high selectivity per column.

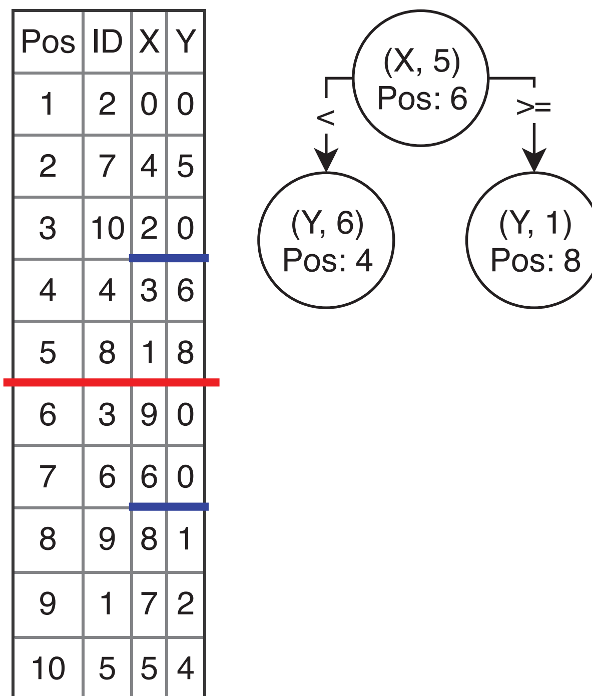
Tabela 2.2: Advantages and disadvantages of each state of the art technique.

3 Proposed Work

In this Section we study how to perform adaptive indexing on multidimensional data. We start by analyzing the KD-Tree [5], demonstrating its construction and search algorithms. Afterwards, we expand the KD-Tree to be able to perform adaptive indexing, which we call Cracking KD-Tree.

3.1 Cracking KD-Tree

We start by analyzing the traditional KD-Tree [5]. The KD-Tree is a multidimensional binary search tree used for searches on one or more attributes. Each of its nodes contains: a key, a discriminator column, two pointers for its children and a integer that represents the starting position on the data. By our own definition, every left path leads to data strictly less than the key, and every right path leads to data greater or equal than the key.



Pos	ID	X	Y
1	2	0	0
2	7	4	5
3	10	2	0
4	4	3	6
5	8	1	8
6	3	9	0
7	6	6	0
8	9	8	1
9	1	7	2
10	5	5	4

Figura 3.1: Example of KD-Tree.

Figure 3.1 depicts an example, the root of the tree has: *key* equals 5, *discriminator column* equals *X*, *position* equals 6, which means that every tuple after the sixth position has an *X* values greater or equal to 5, and two pointers for its children.

One should also notice that the node (Y,6) does not split the entirety of the dataset, but only the partition it is inserted. In fact, what the node (Y, 6) shows is: from position 3 to 1, we

have all tuples with $X < 5$ and $Y < 6$, and from position 4 to 5, we have all tuples with $X < 5$ and $Y \geq 6$.

3.1.1 Construction

There are different ways to construct a KD-Tree, the traditional method is using the median of each column to split the data horizontally. The algorithm splits each partition by finding the its median on a determined attribute, chosen in a round robin fashion.

Figure 3.2 shows an example. Firstly, the median, on attribute X, of the entire dataset is found, which is 5. Then, the data is split into two different partitions, with X strictly less than the median and greater or equal to it, as shown in Figure 3.2 (B). On part (C), the algorithm finds the median of attribute Y, but only considering data from positions 1 to 5, and then proceeds to split that same data. Finally on part (D), the algorithm finds the median on attribute Y but now on the partition that goes from positions 6 until 10, and splits it creating two new partitions, then the algorithm stops because every partition has a size less or equal to a predefined threshold, in this case 3.

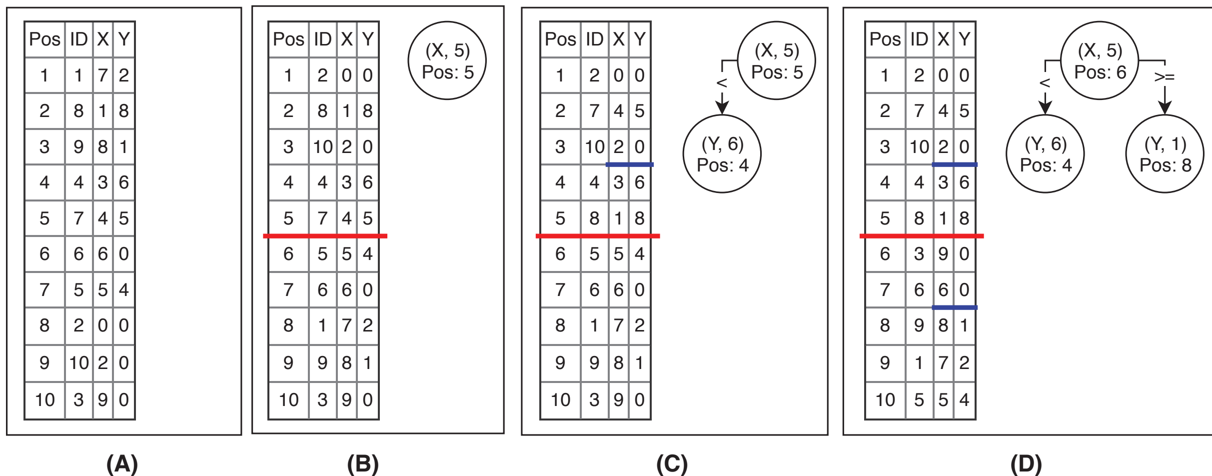


Figura 3.2: Example of KD-Tree construction using medians.

Since, finding the median of any set of elements by ordering is very costly, we decided to use the algorithm of Quickselect [19]. It works by finding the nth smallest element of an array without having to order it, to find the median we simple need to find the $\frac{N}{2}th$ smallest element.

3.1.2 Search

Searching on a KD-Tree starts in the same way as any tree-like data structure, by setting the root as the current node, but differently from other structures, the search can end in one or more non neighbor leafs. The process of searching consists of comparing the current node's key with its respective part in the query, i.e., if a node has a discriminator column of X then it should compare its key only with the part of the query that has a filter on X.

It is extremely important to notice that: the search process will yield the partitions that have the results, but this does not mean that every tuple in said partitions answer the query, it is still necessary to do a scan.

Comparing a node with a query can yield three different outcomes, Figure 3.3 depicts a visual representation:

1. The node's key is less than the range, Figure 3.3 (A), so the left path should be examined.
2. The node's key is greater than the range, Figure 3.3 (B), so the right path should be examined.
3. The node's key is inside the range, Figure 3.3 (C), both paths should be examined.

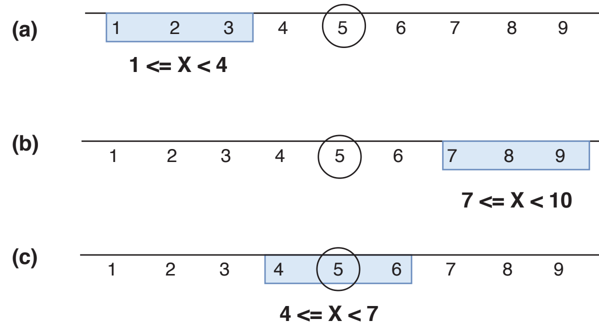


Figura 3.3: Possible different outcomes of comparing a value to a range.

If a partial multidimensional range query is to be searched, some nodes may not have predicates that select on their discriminator column, in this case both of the paths should be followed.

Figure 3.4 shows two examples of the search process. We start by analyzing example 1, the left most. On figure (1.A), the search compares the root node with the given query, since the query is greater than the root, the right path should be followed. On figure (1.B), the node (Y, 1) is compared, again, the right path should be followed, since the query is greater than the node. On figure (1.C), the leaf node (X, 7) is analyzed, which leads us to the left path, which is the partition with only position 8.

Analyzing example 2, on figure (2.A), the root is compared with the query, since the node's key is inside the query range, both paths should be followed. On figure (2.B), the node (Y, 6) is greater than the query, so its left path is followed, and node (Y, 1) is smaller than the query, so its right path is followed. Finally, on figure (2.C), the leaf (X, 2) is inside the query range, so both partitions need to be scanned, and the leaf (X, 7) is greater than the query, so only its left partition needs to be scanned.

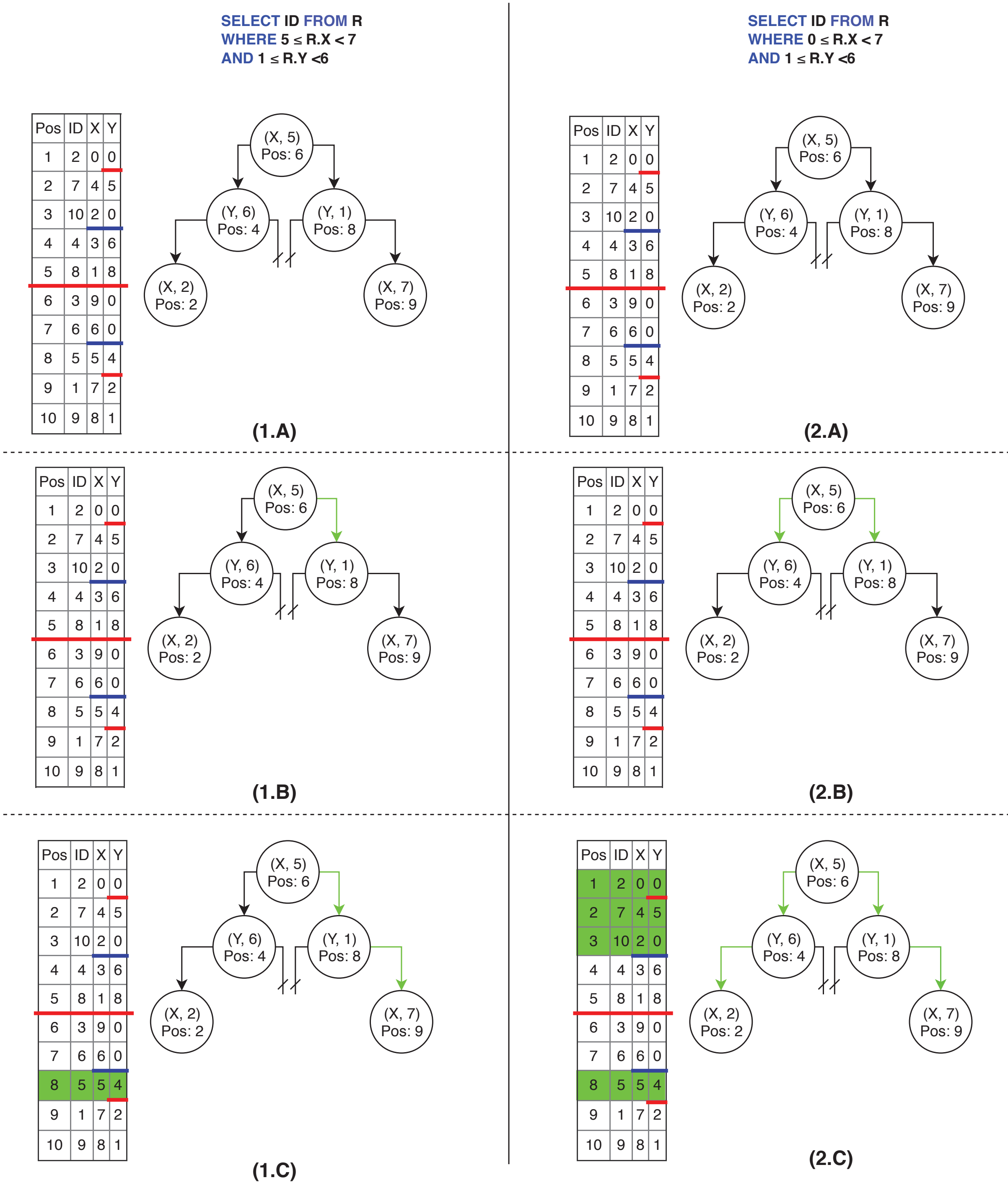


Figura 3.4: Two examples of the KD-Tree searching process demonstrating the difference between having to search one and multiple partitions.

3.1.3 Disjunctive Searches

Up until now we only talked about conjunctive searches, i.e., searches that have an AND operation between the filters. However, we also need to deal with disjunctive searchers, i.e., searches that have an OR operation between the filters.

Our proposed solution for disjunctive queries is: splitting the query into different conjunctive queries.

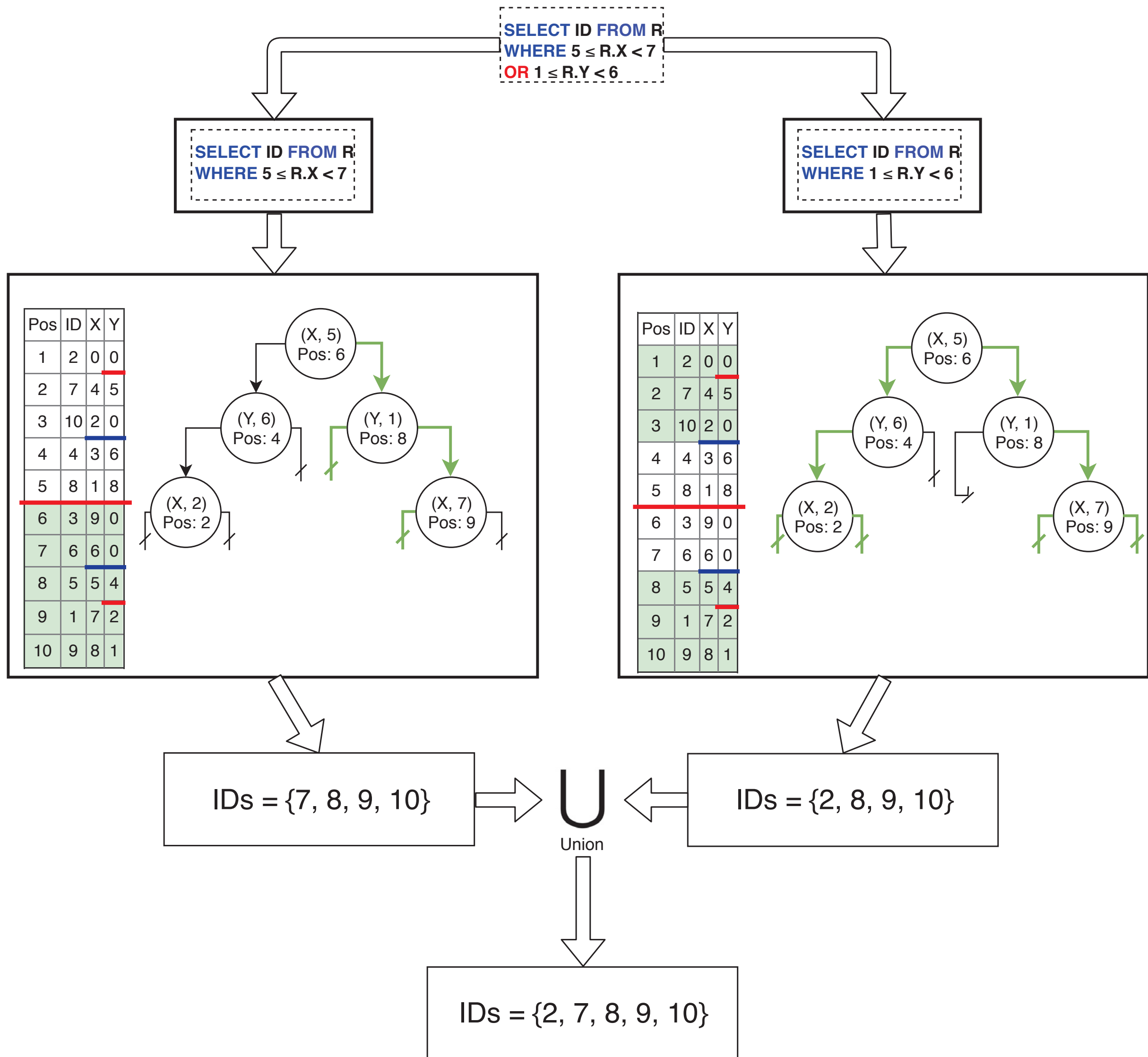


Figura 3.5: Example of query transformation process on disjunctive searches. Note that the partitions in green still need to be scanned for the correct tuples.

Figure 3.5 depicts an example. Given the query SELECT id FROM R WHERE $5 \leq R.X < 7$ OR $1 \leq R.Y < 6$. We can split it into two different queries:

SELECT id FROM R WHERE $5 \leq R.X < 7$

SELECT id FROM R WHERE $1 \leq R.Y < 6$.

Then proceed to search them separately, which will yield two sets of IDs, $ID_1 = \{7, 8, 9, 10\}$ and $ID_2 = \{2, 8, 9, 10\}$, and simple calculate the union of the two results, which is the set of $IDs = \{2, 7, 8, 9, 10\}$.

3.1.4 Adaptive Indexing

The major difference between the Cracking KD-Tree and a regular KD-Tree is how they are built. As mentioned above, the regular KD-Tree is constructed based on the medians of each column, whereas the Cracking KD-Tree is constructed based on the incoming range queries. Given any range query, for example, $x_1 <= X < x_2$ AND $y_1 <= Y < y_2$, we first split the query into pairs of columns and keys, from left to right, e.g., $(X, x_1), (X, x_2), (Y, y_1), (Y, y_2)$, then each pair is added to the index. It is important to notice that one pair can be inserted in multiple

locations, different from one dimensional search trees where one insertion meant one new leaf, here one insertion can mean one or more new leafs.

Notice that the cracking process does not happen on the original table, when the first query arrives, the table is copied to a data structure called *cracker table*, and then this data structure is cracked. By making the copy we are able to keep all attributes aligned, which eases the process of tuple reconstruction afterwards. Also, since the tuples are aligned, the method works with both explicit and implicit IDs on the columns.

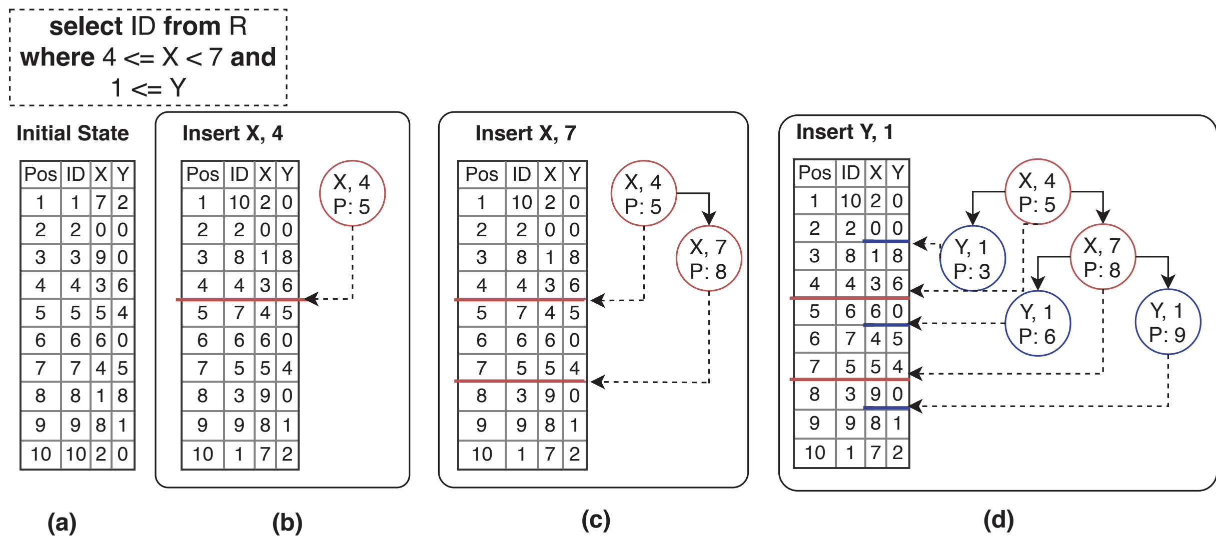


Figura 3.6: Process of cracking the data using the query $4 \leq X < 7$ AND $1 \leq Y$.

Figure 3.6 provides an example. In step (a), we have the initial state of the data, and the incoming query, $4 \leq X < 7$ AND $1 \leq Y$. In step (b), first, the table is copied to the *cracker table*, and since there is no index yet, the table is cracked and the node (X ,4) is inserted as the root. In step (c), first it is necessary to find what partitions should be cracked, starting from the root, the new node (X, 7) is greater than the root, so the right path should be followed, which leads to a partition, which is cracked and the node is inserted as the right child of the root. In step (d), the node (Y, 1) is to be inserted, first the partitions to be cracked need to be found, since the root and the node to be inserted do not share the same column, both paths should be followed. When following the left one, we end up in a partition, which is then cracked and the new node is inserted as the left child of the root. The right path leads to node (X, 7), since this node and the node to be inserted do not have the same column, both paths should be followed, since both paths lead to partitions, both partitions are cracked, separately, and both new nodes are inserted as the children of (X, 7).

3.2 Search Complexity

Since the construction process of the Cracking KD-Tree depends on the incoming searches and has no restrictions whatsoever, the resulting KD-Tree may or may not be balanced, i.e., Cracking KD-Tree‘s complexity will have as upper and lower bounds the best and worst case scenarios, respectively, discussed below.

In the best case scenario, a balanced KD-Tree, Lee and Wong [24] demonstrated that, given N points, in the worst case the cost of a range search in a complete KD-Tree is

$O(d * N^{1-1/d} + F)$, where d is the number of dimensions queried, and F is the number of points found in the range.

In the worst case scenario, the resulting KD-Tree has only one branch, behaving in the same way as a linked list, and thus having a search complexity of $O(N)$ where N is the number of nodes.

3.3 Drawbacks

There are two possible scenarios that may decrease the performance of the Cracking KD-Tree. However, none of them were tested.

The first scenario is when there is a huge difference between the number of attributes in a query, and the number of attributes indexed by the Cracking KD-Tree. For example, imagine that the we have 20 different attributes indexed, and we need to answer a query that filters on only one of them. When the search process arrives at a node with the column equal to one of the other 19 attributes, both children will need to be searched. In other words, the node did not help with the search. Imagining that the distribution of nodes per attribute in the index is approximately equal, only $\frac{1}{20}$ nodes would be helpful.

The second scenario, since the Cracking KD-Tree is not balanced, some workloads may create an unbalanced index resulting in high performance variation, which is not ideal. One workload of this kind is one with ever increasing values on filters. For example, the first query is in the likes of $0 \leq X < 5\,(...)$, then the second is $6 \leq X < 10\,(...)$, the third is $11 \leq X < 15\,(...)$ and so on.

4 Experiments

In this Section we describe and analyze all experiments performed. We start with the setup used, then we briefly explain the algorithms compared, and finally we study the results obtained.

4.1 Setup and Algorithms

To implement and compare the *Cracking KD-Tree* with other algorithms, we extended the core of the database cracking simulator[1] used in [40] to be used with multidimensional data and queries. The simulator is a single-threaded stand-alone program written in C++ and compiled with GNU g++ version 7.3.1 using optimization level `-O3`[2]. All experiments were conducted on a machine equipped with 256 GB of main memory and two 2.6 GHz Intel Xeon E5-2650 v2 CPUs, each with 20 MB L3 cache, 8 cores and hyper-threading enabled, running Fedora 26.

The experiment is a multidimensional extension of the experimentation used in [40], the major differences are the selectivity and number of tuples. Our dataset consisted of a table with 8-bit integers attributes holding 10^7 tuples. The values per attribute were independent and uniformly distributed. We varied the number of attributes between 2, 4, 8 and 16.

```
8  SELECT COUNT(R.C1) FROM R
9  WHERE LowC_1 ≤ R.C1 < HighC_1 AND ... AND LowC_n ≤ R.Cn < HighC_n
```

Listing 4.1: Query form used on experiments.

Our workload consisted of 1000 queries, all of them in the form depicted in Listing 4.1. Where n is the number of dimensions queried[3]. All queries had selectivity equal to 20% per column, one might notice that the total selectivity of the queries in the query stream will vary since the query predicates are selected in a random pattern. We repeat the entire workload 10 times and take the average run time of each query as the reported time.

We implemented five different algorithms to compared with the Cracking KD-Tree. Two of them, Quasii and Sideways Cracking, were already described in Chapter 2, now we briefly explain the remaining three.

Standard Cracking AVL. Each column goes through the process of database cracking separately. Afterwards, the results are intersected by the creation of bit-vectors, as explained in Chapter 2.

Full Index KD-Tree. All columns are indexed using a KD-Tree pivoting by median values and choosing the dimensions in a round robin fashion. The query result is then given by a look up in the KD-Tree.

[1]Available at: https://bigdata.uni-saarland.de/research/publications.php

[2]According to the GCC Optimize Options Documentation (see https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html) -O3 is the option for which the compiler applies the most optimization in the code.

[3]All queries searched on all available attributes.

Full Scan. We use a vectorized, predicated scan approach [7] that produces a candidate list per scanned vector of a column, Figure 4.1 depicts an example. Given a MDRQ and a table, with attributes X, Y and Z, in a columnar database. The algorithm starts by scanning attribute X creating a candidate list with tuple IDs, $\{1, 2, 3, 9, 10\}$. Then, it proceeds to refine the list by scanning on attribute Y, but only on tuples that have the ID in the candidate list, resulting in $\{1, 9, 10\}$. Finally, the same process happens on attribute Z and the final result is presented, $IDs = \{1, 10\}$. Also, the algorithm scans the data by blocks, instead of the entire table at once, each block size is in accordance to the L2 cache size.

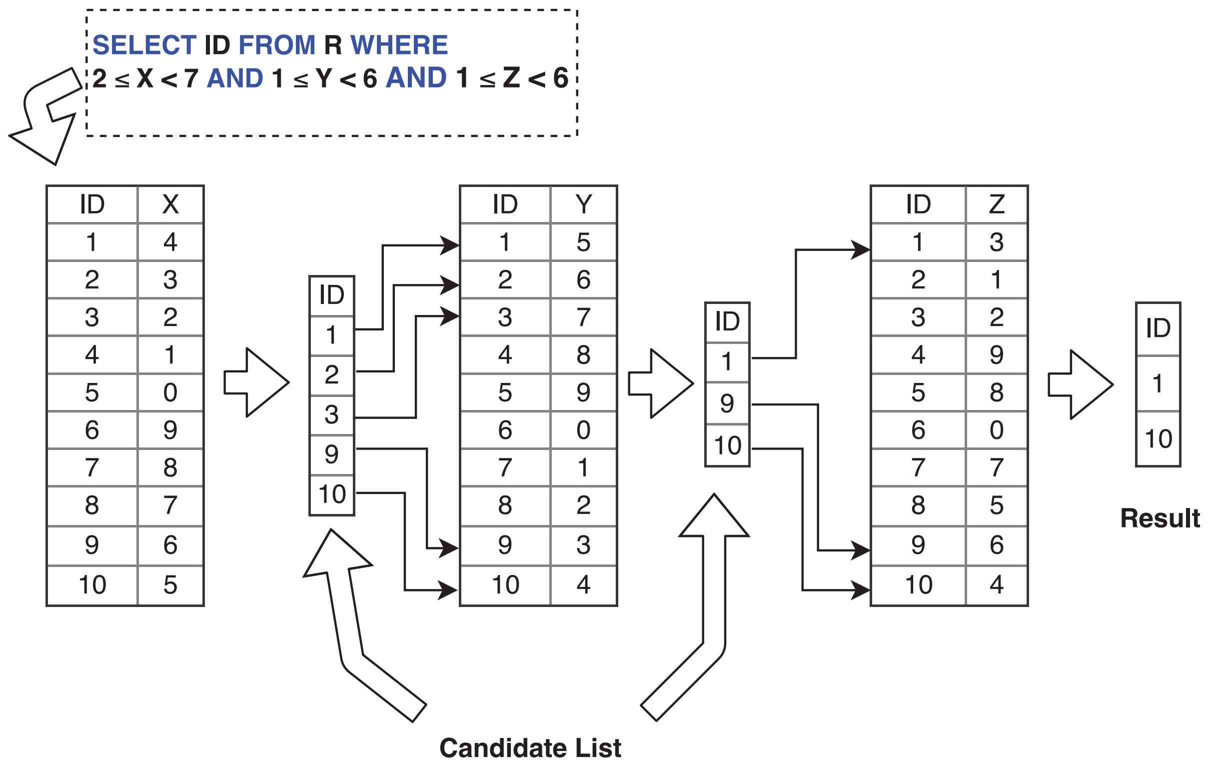


Figura 4.1: Full Scan example.

4.2 Experiment Discussion

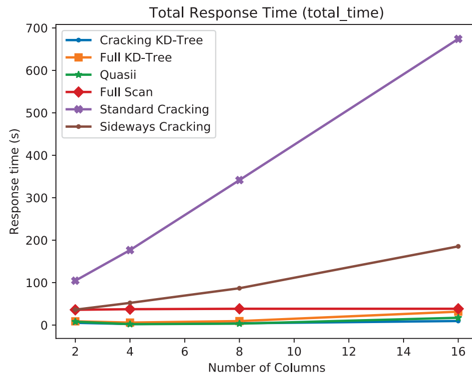
We start by analyzing how every algorithms responds to workloads with varying number of attributes, more specifically, the time every algorithm took to answer separated workloads with 2, 4, 8 and 16 attributes. And then we make our conclusions.

In figure 4.2(a) we can see the change in workload response time as we vary the number of columns. One might notice that with more attributes the query selectivity gets smaller, with four attributes it is less than 1%.

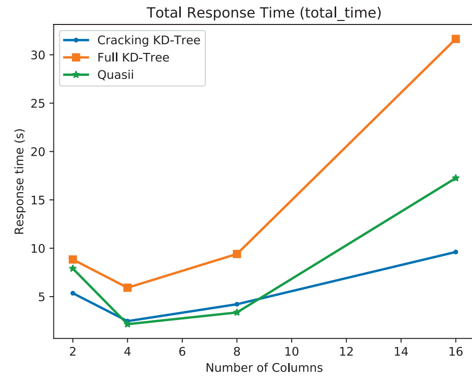
As expected Standard Cracking and Sideways Cracking have the worst performance, because both have to deal with huge partial results that have a high cost to intersect. It gets even worse with 16 attributes, where the result size is extremely small, but each partial result is 20% of each column. Both techniques cannot make good use of low query selectivity in MDRQs.

Next comes the Full Scan algorithm, that maintains its performance because the query selectivity diminishes with more attributes. This happens because it produces a candidate list per scanned column, since the number of intersections between each column is low the candidate list tends to get small quickly.

Finally, Quasii, Full KD-Tree and Cracking KD-Tree all have similar response times, when compared to the other three algorithms. Figure 4.2(b) presents a comparison between the three, and from now on will be our main contrast point.



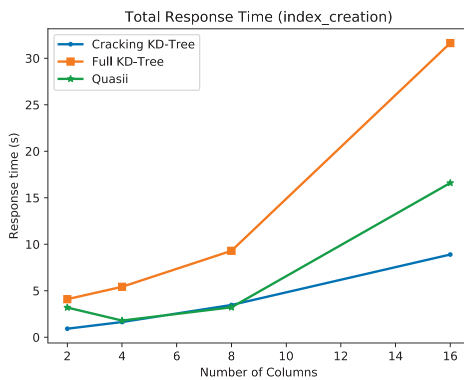
(a) All algorithms.



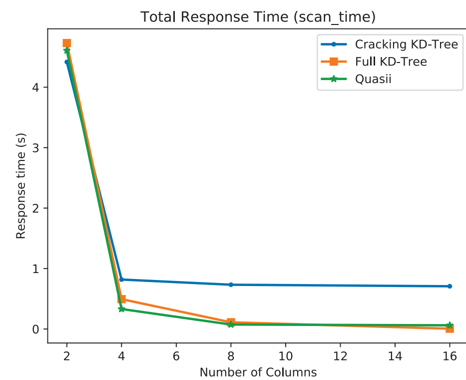
(b) Quasii, Full KD-Tree and Cracking KD-Tree.

Figura 4.2: Workload response time when changing the number of attributes.

In Figure 4.3 we can observe how the algorithms' *index creation* and *scan times* change when we increase the number of columns. The first thing to notice is that while the index creation increases in a linear fashion, the scan time resembles a negative exponential.



(a) Time to create the index.



(b) Time to scan the table.

Figura 4.3: Time per query.

One can see that the Cracking KD-Tree shows a linear scalability with respect to the number of attributes, however it stagnates the scan time when the selectivity gets too low.

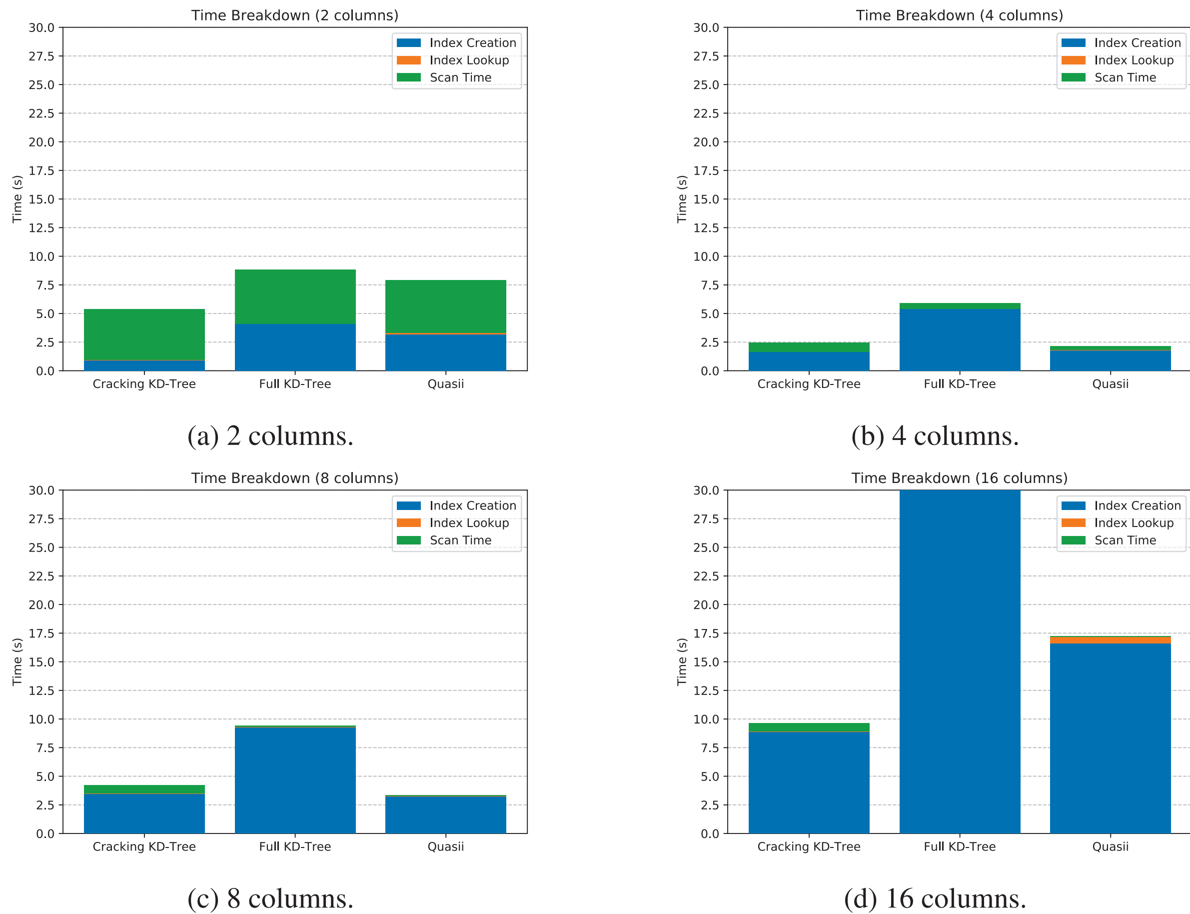
In Figure 4.4 we can see with more details what happened in each workload. The total response time is split into four categories:

Index Creation. Time spent constructing the index, includes the cracking time.

Index Lookup. Time spent traversing the index structure to find the partitions.

Scan Time. Time to scan the table for the correct ID's.

Comparing Figure 4.4(a) to Figure 4.4(b), there is a huge decrease in scan time, from 4 seconds to around 1 second. While the index creation time increases, it is nowhere near the rate of the scan time. That explains why a workload with two columns takes more time to process than one with four or eight columns.



(a) 2 columns. (b) 4 columns.

(c) 8 columns. (d) 16 columns.

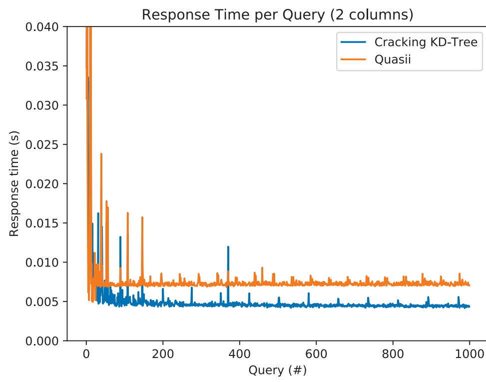
Figura 4.4: Total response time breakdown.

Figure 4.4 provides us some insights: Firstly, Cracking KD-Tree's index creation time is the fastest of the three. Seconly, both KD-Trees have negligible index lookup times, however, the Cracking KD-Tree's scan time is higher. This happens because the Cracking KD-Tree cannot create good partitions when the query selectivity gets too low, and it has to scan more parts of the data to get the final result.

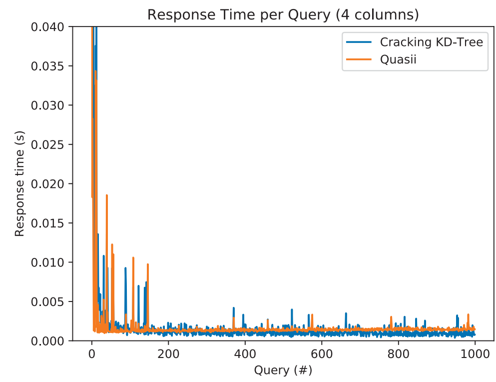
Now we need to analyze why Quasii has a better performance when compared to the Cracking KD-Tree on workloads with four and eight columns.

If we observe Figure 4.3, the variation in index creation and scan time, with four and eight columns, is the point where scan time has diminished a lot and index creation time is still slowly increasing, which is the best world for Quasii, low selectivity and not a lot of columns make its cracking operation fast, and the low selectivity makes the scan time be near zero because Quasii creates good partitions.

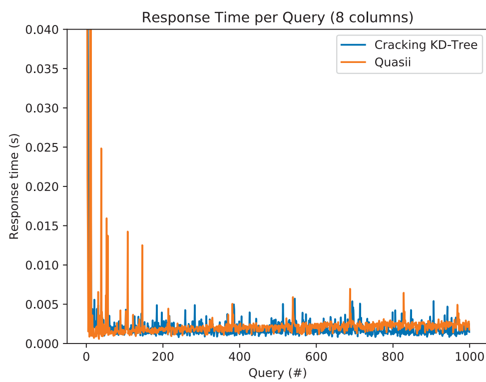
One might notice that, with two columns the cracking time is high because the selectivity is high and lots of slices will go through the artificial slicing process, and with 16 attributes the time is high because there is a lot of dimensions to crack. Figures 4.5(a) and 4.5(d) demonstrate this effect on the response time per query.



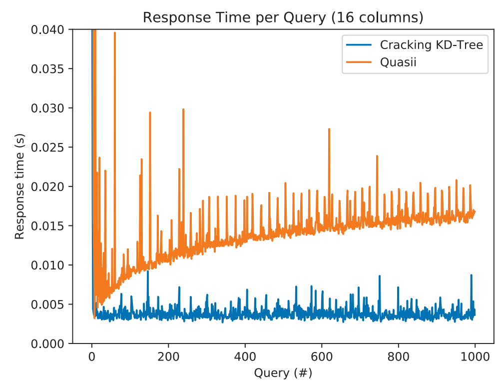
(a) 2 columns.



(b) 4 columns.



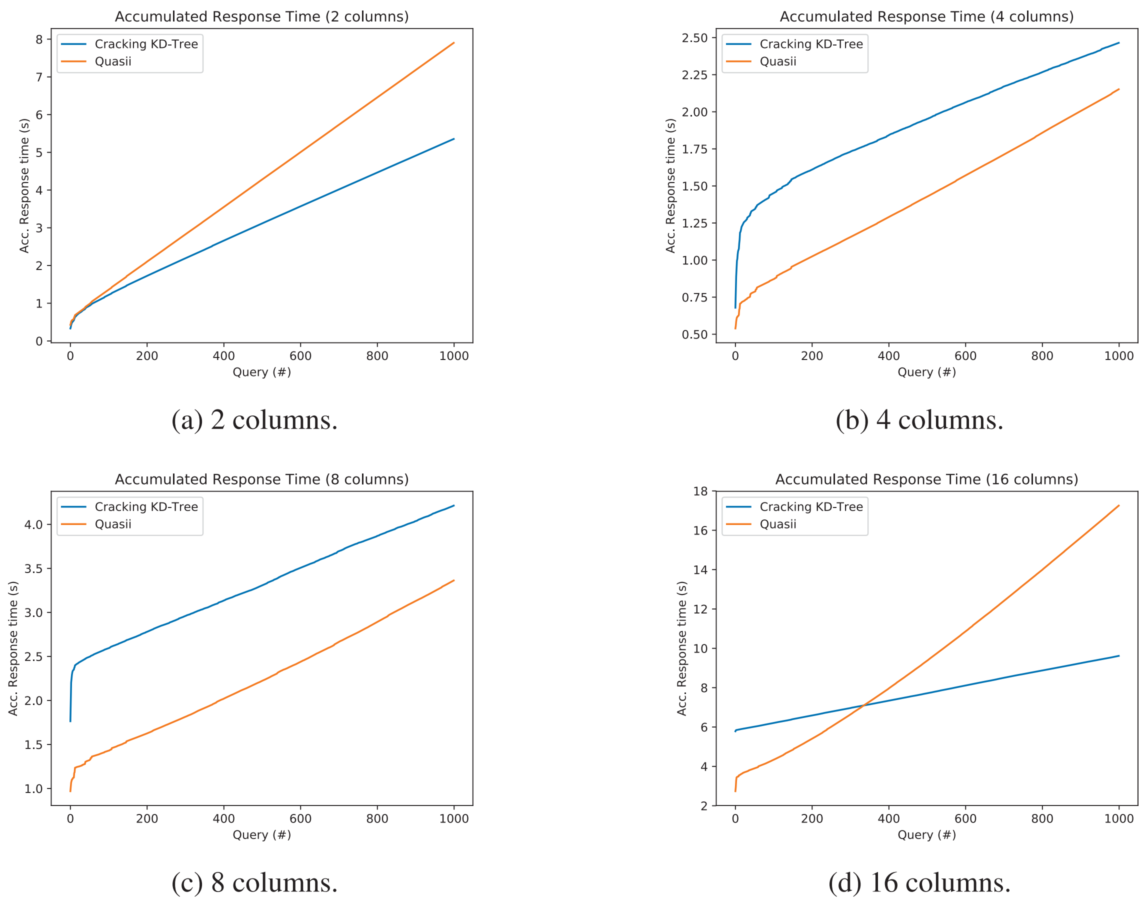
(c) 8 columns.



(d) 16 columns.

Figura 4.5: Response time per query.

In Figures 4.5(b) and 4.5(c), the Cracking KD-Tree starts high a higher response time per query, however at the end of the workload the cost is slightly smaller. If more queries existed in the workload, Cracking KD-Tree would surpass Quasii. As one can see in Figures 4.6(b) and 4.6(c), which represent the accumulated query response time, by analyzing the growth of each algorithm.



(a) 2 columns.

(b) 4 columns.

(c) 8 columns.

(d) 16 columns.

Figura 4.6: Accumulated response time.

From all this analysis, we can make some conclusions regarding the behavior of the Cracking KD-Tree and Quasii: With respect to **index creation**, the Cracking KD-Tree demonstrated a good scalability, with a higher number of attributes, and is independent from the selectivity. In contrast, Quasii depends heavily on both, high selectivity or lots of attributes makes Quasii's adaptive process to be slow.

Cracking KD-Tree demonstrated a good performance on **index lookup** time, since in every experiment it was almost negligible.

With respect to **scan time**. Since the Cracking KD-tree utilizes the predicates of the queries as hints, queries with low selectivity makes it create bad partitions during the index creation process, and so, during the scan more partitions have to searched. Quasii, on the other hand, is efficient on low selectivity, but is slower otherwise.

To conclude, we proposed a novel adaptive indexing technique for multidimensional data, utilizing a multidimensional index to store the partitions created during the adaptive process. Our technique, Cracking KD-Tree, demonstrated to be more efficient (Sideways Cracking) or at least equivalent (Quasii) with respect to answering multidimensional range queries.

5 Conclusion and Future Work

Database Cracking was proposed in the last decade, implementing the idea of adaptively create indexes as a byproduct of query execution. Since then, multiple researches were made to improve it in other directions. To name a few: Stochastic Cracking improved its robustness, Hybrid Cracking its time to converge to a full index, Sideways Cracking improved tuple reconstruction, Quasii extended it to multidimensional object data and queries. However, no work had been done yet on multidimensional point data and MDRQs.

We proposed a novel adaptive indexing technique, called Cracking KD-Tree, capable of efficiently processing multidimensional range queries. Our technique makes use of a *cracker table* during the cracking process, which moves the entire tuple instead of only the attributes filtered in the query, keeping the data aligned. We also make use of a multidimensional data structure, KD-Tree, to keep track of the partitions created. We experimented on workloads with 20% per column selectivity and independent uniform random data and query predicates distribution. The following results were obtained:

- **2 Attributes.** 6.7x faster than Sideways Cracking, and 1.4x faster than Quasii.
- **4 Attributes.** 21x faster than Sideways Cracking, and 0.87x slower than Quasii, however given a bigger workload the Cracking KD-Tree will be faster than Quasii.
- **8 Attributes.** 20x faster than Sideways Cracking, and 0.79x slower than Quasii, however given a bigger workload the Cracking KD-Tree will be faster than Quasii.
- **16 Attributes.** 19x faster than Sideways Cracking, and 1.7x faster than Quasii.

Still, there are more areas that can be explored for future work, to name them:

- **New benchmarks.** Until now we only tested the work using a synthetic benchmark, the next steps are to find new benchmarks that reflect real world applications.
- **Data structures.** Not only the KD-Tree can be used as an adaptive index with multidimensional data. Other structures like Grid-File, R-Tree and Multidimensional Search Tree can also be adapted.
- **Concurrency.** Until this point we have not touched the case of multiple queries accessing the index structure.
- **Updates.** In this work we only studied the problem of adaptive indexing on multidimensional data on a read-only environment, but it is an interesting challenge to export these concepts to environments with updates.
- **KD-Tree cracking.** The cracking process we described is more spread through the index, however creating one with a more narrow approach could be useful, since it would decrease the index creation time.

- **Machine Learning.** KD-Trees are used to speed up the K-Nearest Neighbors machine learning algorithm, perhaps the Cracking KD-Tree can be used in the same sense.
- **Insertion order.** Analyze what would be the impact of inserting using different orders in the cracking process.
- **Keys per node.** Implement the Cracking KD-Tree using multiple keys per node and using the same discriminator column, and analyze the impacts.
- **Sorting.** Analyze the impacts of sorting the partitions, on some attribute, when it hits the threshold.
- **One dimensional range queries.** Analyze the behavior of each algorithm when dealing with one dimensional range queries.

Referências

[1] Lars Arge, Mark De Berg, Herman Haverkort, and Ke Yi. The priority r-tree: A practically efficient and worst-case optimal r-tree. *ACM Trans. Algorithms*, 4(1):9:1–9:30, March 2008.

[2] Manos Athanassoulis, Zheng Yan, and Stratos Idreos. Upbit: Scalable in-memory updatable bitmap indexing. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1319–1332, New York, NY, USA, 2016. ACM.

[3] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The r*-tree: An efficient and robust access method for points and rectangles. *SIGMOD Rec.*, 19(2):322–331, May 1990.

[4] J. L. Bentley and H. A. Maurer. Efficient worst-case data structures for range searching. *Acta Inf.*, 13(2):155–168, February 1980.

[5] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, September 1975.

[6] Stefan Berchtold, Daniel A. Keim, and Hans-Peter Kriegel. The x-tree: An index structure for high-dimensional data. In *Proceedings of the 22th International Conference on Very Large Data Bases*, VLDB '96, pages 28–39, San Francisco, CA, USA, 1996. Morgan Kaufmann Publishers Inc.

[7] Peter A Boncz, Marcin Zukowski, and Niels Nes. Monetdb/x100: Hyper-pipelining query execution. In *CIDR*, volume 5, pages 225–237, 2005.

[8] Chee-Yong Chan and Yannis E. Ioannidis. Bitmap index design and evaluation. *SIGMOD Rec.*, 27(2):355–366, June 1998.

[9] Chee-Yong Chan and Yannis E. Ioannidis. An efficient bitmap encoding scheme for selection queries. *SIGMOD Rec.*, 28(2):215–226, June 1999.

[10] Paolo Ciaccia, Marco Patella, and Pavel Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, VLDB '97, pages 426–435, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

[11] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[12] George P. Copeland and Setrag N. Khoshafian. A decomposition storage model. *SIGMOD Rec.*, 14(4):268–279, May 1985.

[13] C. Faloutsos and S. Roseman. Fractals for secondary key retrieval. In *Proceedings of the Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '89, pages 247–252, New York, NY, USA, 1989. ACM.

[14] R. A. Finkel and J. L. Bentley. Quad trees a data structure for retrieval on composite keys. *Acta Inf.*, 4(1):1–9, March 1974.

[15] Goetz Graefe and Harumi Kuno. Self-selecting, self-tuning, incrementally optimized indexes. In *Proceedings of the 13th International Conference on Extending Database Technology*, EDBT '10, pages 371–381, New York, NY, USA, 2010. ACM.

[16] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, SIGMOD '84, pages 47–57, New York, NY, USA, 1984. ACM.

[17] Felix Halim, Stratos Idreos, Panagiotis Karras, and Roland H. C. Yap. Stochastic database cracking: Towards robust adaptive indexing in main-memory column-stores. *Proc. VLDB Endow.*, 5(6):502–513, February 2012.

[18] C. A. R. Hoare. Algorithm 64: Quicksort. *Commun. ACM*, 4(7):321–, July 1961.

[19] C. A. R. Hoare. Algorithm 65: Find. *Commun. ACM*, 4(7):321–322, July 1961.

[20] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. Database cracking. In *CIDR 2007, Third Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 7-10, 2007, Online Proceedings*, pages 68–78, 2007.

[21] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. Self-organizing tuple reconstruction in column-stores. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 297–308, New York, NY, USA, 2009. ACM.

[22] Stratos Idreos, Stefan Manegold, Harumi Kuno, and Goetz Graefe. Merging what's cracked, cracking what's merged: Adaptive indexing in main-memory column-stores. *Proc. VLDB Endow.*, 4(9):586–597, June 2011.

[23] H. V. Jagadish. Linear clustering of objects with multiple attributes. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, SIGMOD '90, pages 332–342, New York, NY, USA, 1990. ACM.

[24] D. T. Lee and C. K. Wong. Worst-case analysis for region and partial region searches in multidimensional binary search trees and balanced quad trees. *Acta Inf.*, 9(1):23–29, March 1977.

[25] Ting Liu, Andrew W. Moore, and Alexander Gray. New algorithms for efficient high-dimensional nonparametric classification. *J. Mach. Learn. Res.*, 7:1135–1158, December 2006.

[26] Jon Louis Bentley. Decomposable searching problems. *Journal of Algorithms*, 8:244–251, 06 1979.

[27] Donald Meagher. Geometric modeling using octree encoding. *Computer Graphics and Image Processing*, 19(2):129 – 147, 1982.

[28] Jorge Augusto Meira, Eduardo Cunha Almeida, Dongsun Kim, Edson Ramiro Filho, and Yves Traon. "overloaded!-- a model-based approach to database stress testing. In *Proceedings, Part I, 27th International Conference on Database and Expert Systems Applications - Volume 9827*, DEXA 2016, pages 207–222, New York, NY, USA, 2016. Springer-Verlag New York, Inc.

[29] Bongki Moon, H. v. Jagadish, Christos Faloutsos, and Joel H. Saltz. Analysis of the clustering properties of the hilbert space-filling curve. *IEEE Trans. on Knowl. and Data Eng.*, 13(1):124–141, January 2001.

[30] J. Nievergelt, Hans Hinterberger, and Kenneth C. Sevcik. The grid file: An adaptable, symmetric multikey file structure. *ACM Trans. Database Syst.*, 9(1):38–71, March 1984.

[31] Stephen M. Omohundro. Five balltree construction algorithms. Technical report, 1989.

[32] Patrick O'Neil and Goetz Graefe. Multi-table joins through bitmapped join indices. *SIGMOD Rec.*, 24(3):8–11, September 1995.

[33] Patrick O'Neil and Dallan Quass. Improved query performance with variant indexes. *SIGMOD Rec.*, 26(2):38–49, June 1997.

[34] Patrick E. O'Neil. Model 204 architecture and performance. In Dieter Gawlick, Mark Haynie, and Andreas Reuter, editors, *High Performance Transaction Systems*, pages 39–59, Berlin, Heidelberg, 1989. Springer Berlin Heidelberg.

[35] Patrick E O'Neil. The set query benchmark, 1993.

[36] J. A. Orenstein and T. H. Merrett. A class of data structures for associative searching. In *Proceedings of the 3rd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, PODS '84, pages 181–190, New York, NY, USA, 1984. ACM.

[37] Mirjana Pavlovic, Darius Sidlauskas, Thomas Heinis, and Anastasia Ailamaki. Quasii: Query-aware spatial incremental index. 2018.

[38] Meikel Poess and John M. Stephens, Jr. Generating thousand benchmark queries in seconds. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, VLDB '04, pages 1045–1053. VLDB Endowment, 2004.

[39] Hanan Samet. *Foundations of Multidimensional and Metric Data Structures (The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[40] Felix Martin Schuhknecht, Alekh Jindal, and Jens Dittrich. The uncracked pieces in database cracking. *Proc. VLDB Endow.*, 7(2):97–108, October 2013.

[41] Timos Sellis, Nick Roussopoulos, and Christos Faloutsos. The r+-tree: A dynamic index for multi-dimensional objects. pages 507–518, 1987.

[42] Kurt Stockinger and Kesheng Wu. Bitmap indices for data warehouses, 01 2006.

[43] Markku Tamminen. The extendible cell method for closest point problems. *BIT Numerical Mathematics*, 22(1):27–41, Mar 1982.

[44] Jeffrey K. Uhlmann. Satisfying general proximity/similarity queries with metric trees. *Inf. Process. Lett.*, 40:175–179, 1991.

[45] Harry K. T. Wong, Hsiu-Fen Liu, Frank Olken, Doron Rotem, and Linda Wong. Bit transposed files. In *Proceedings of the 11th International Conference on Very Large Data Bases - Volume 11*, VLDB '85, pages 448–457. VLDB Endowment, 1985.

[46] Peter N. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '93, pages 311–321, Philadelphia, PA, USA, 1993. Society for Industrial and Applied Mathematics.