Database Cracking

David Werner

23. Januar 2018

Zusammenfassung

This paper talks about the self-oraganized index maintainance approach of database cracking. It provides fundamental knowledge as well as the description of a basic implementation of database cracking. Alongside multiple cracking algorithms, the implementation also contains the code of a cracking index. By combining these two features the implementation allows the execution of range queries on a single column. These range queries are compared to the range queries of a B^+-Tree. The evaluation of the comparison shows that cracking based range queries perform more efficiently.

1 Introduction

In database management systems, index structures are used to speed up queries. An index can be built on an attribute and stores all existing attribute values and their corresponding physical position within the memory space in which the column resides. Typically only attributes with distinct values are indexed. By looking up the required attribute value in the index, a query can find their corresponding position faster than by using a table scan as long as there are not to many required values. When the database is updated by inserting or removing records, the index needs to be adapted by either inserting a new entry or removing an existing one for the affected attribute values. In this seminar paper I present another approach to indexing and index maintainance called database cracking. Database cracking relies on reordering the pyhsical database according to the needs of incoming queries. Queries on attributes which are indexed using database cracking trigger such a reordering according to the query conditions. This procedure is based on the idea that index maintaing should be part of the query processing instead of database updates.

2 Database Cracking

This section provides fundamental knowledge about database cracking. Different cracking algorithms are presented and a notion of how and what to safe in a cracking index is given. Furthermore some interesting properties of database cracking are discussed.

2.1 Basics

Database cracking is an approach for data indexing and index maintainance in a self-organized way. In a database system where database cracking is used, an incoming query requesting all elements which satisfy a certain condition c does not only return a result but it also causes a reodering of the physical database so that all elements satisfying c are stored in a contiguous memery space. Therefore the physical database is devided into multiple parts. This procedure is called cracking. The different parts of the database which result from cracking are called pieces. As the cracking process is caused by queries, a query on the database can be seen as a hint for the database on how the data has to be ordered. By using this mechanism the database reorganizes itself in the most favourable way according to the workload which is put on it. Alongside the cracking procedures a structure called cracking index is maintained. This index stores entries containing information about already existing pieces. By looking up matching entries for required pieces a query is able to use crackings from preceding queries to shorten its execution time.[1]

An example scenario in which queries cause crackings and use the index to speed up their execution can be seen in figure 1 where three consecutive queries use cracking to find their results. The first query requests all elements which are smaller than five. Therefore the database is cracked in two pieces. The second query wants to find all elements between five and eight. As the first query already created a piece which contains all elements greater or qual to five, the second query can use this information from the cracking index so that only one of the two existing pieces needs to be cracked. Consequently the second piece is cracked in two new pieces where one contains all elements from five to eight and the other contains all elements greater than eight. A third query is looking for all values greater than eight. Because the second query already induced the creation of a piece containing all values which are greater than eight, the third query is able to find a result without cracking but only with information from the index.

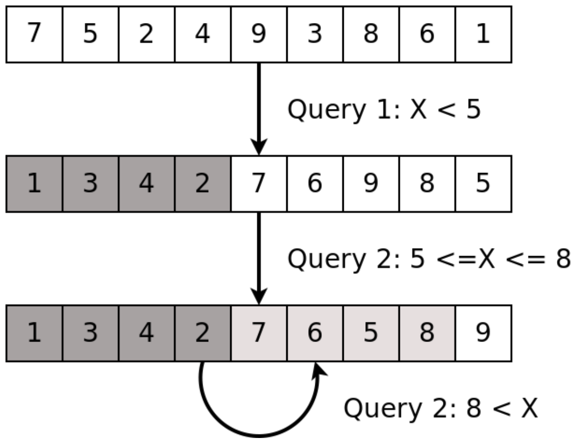


Abbildung 1: Example cracking scenario

2.2 Cracking Algorithms

When using database cracking, the cracking procedure which means the physical reordering of a column is not performed on the original column but on a copy of it, the so called cracking column. This cracking column is needed because the originial column has to stay in the insertion order. If it would be reorganized the insertion order would be destroyed. A column store database would then not be able to reconstruct complete records properly. Hence, before any query is able to perform crackings, a cracking column must be created on which the cracking algorithms can operate.

In this paper, I would like to present three different cracking algorithms. All of them crack a *column* by using by using two position variables *left* and *right* and one or two bound *values*. The *values* are the seperators which are used to split the column. On top of that *inclusive* flags are used to determine whether the bound *values* itselves are in the lower or the upper piece. The algorithms use comparison operators which are depicted as Δ_1 and Δ_2. The former is either $<$ or $\leq$ while the latter is either $>$ or $\geq$. Which operator is used depends on the *inclusive* flags.

The first algorithm cracks a column or a column piece in two new pieces. The second algorithm works similar to the first one but cracks into three pieces. They operate according to the following pseudocode:

Algorithm 1 Crack in two pieces

```
procedure CRACK_IN_2($column, left, right, value, inclusive$)
    while $left < right$ do
        if $column[left]\ \Delta_1\ value$ then
            $left \leftarrow left + 1$
        else
            while $column[right]\ \Delta_2\ value$ and $left < right$ do
                $right \leftarrow right - 1$
            end while
            $swap(column[left], column[right])$
            $left \leftarrow left + 1$
            $right \leftarrow right - 1$
        end if
    end while
end procedure
```

Algorithm 2 Crack in three pieces

```
procedure CRACK_IN_3($column, left, right, value1, value2, inclusive1, inclusive2$)
    $tmp \leftarrow left$
    while $left < right$ do
        while $left < right$ and $column[left]\ \Delta_3\ value2$ do
            if $column[left]\ \Delta_4\ value1$ then
                $swap(column[left], column[tmp])$
                $tmp \leftarrow tmp + 1$
            end if
            $left \leftarrow left + 1$
        end while
        while $left < right$ and $column[right]\ \Delta_5\ value2$ do
            $right \leftarrow right - 1$
        end while
        if $left < right$ then
            $swap(column[left], column[right])$
        end if
    end while
end procedure
```

Algorithm 1 as well as Algorithm 2 are single-pass algorithms and were introduced in [1]. I modified the second algorithm to require less code than the original one. Both algotihms crack by walking through the column using two (left, right) or 3 (left,right and tmp) pointers reading the values. Values which fullfill the corresponding condition(greater or smaller) remain where they are. If both pointers identify values which do not satisfy their condtition the values are swapped. Thereby the algorithms touch as little elements of the cracking column as possible. Cracking in three pieces could also be realized by cracking two times in two pieces but since Algorithm two is a single-pass algorithm it performs slightly better.[1]

The third algorihtm also cracks a column or a piece of a column in two new pieces. It is therefore an alternative to Algorithm 1. The difference is that this algorithm does not have any branches. This is achieved by interpreting the results of condition evaluations as 1 (true) or 0 false. The algorithm cracks by taking a look at the elements pointed to by *left* and *right*. One of them is stored in an active variable, the other one is stored in a backup variable and then overwritten with the active value. Based on the evaluation of *active* Δ_1 *value* either *left* or *right* is advanced. The element at the advanced pointer becomes the new backup value while the old backup value becomes the new active element. This procedure is continued until *left* and *right* point to the same position. When this step is reached the element in the active slot is stored at this position and the algorithm is finished. It is depicted as pseudocode in the following figure:

Algorithm 3 Crack in two pieces (branch free)

```
1:  procedure CRACK_IN_2_BF($column, left, right, value, inclusive$)
2:      $cmp$
3:      $active \leftarrow column[left]$
4:      $backup \leftarrow column[right]$
5:      while $left < right$ do
6:          $cmp \leftarrow active\ \Delta_1\ value$
7:          $column[left] \leftarrow active$
8:          $column[right] \leftarrow active$
9:          $left \leftarrow left + cmp$
10:         $right \leftarrow right - (1 - cmp)$
11:         $active \leftarrow (column[left] * cmp) + (column[right] * (1 - cmp))$
12:         $swap(active, backup)$
13:     end while
14:     $column[left] \leftarrow active$
15: end procedure
```

2.3 Cracking Index

In order to provide information about crackings a cracking index is maintened. For each crack the index stores the value v by which the column was reordered during the crack, the corresponding inclusive flag i and a postition p. The position p points to a position of the cracking column. All values x in the cracking column which reside before or at this position satisfy the cracking condition $x < v$ or $x \leq v$ depending on i. Other values which reside at positions greater than p do not satisfy that condition.[1]

The index serves two purposes: When a query tries to find a result, the index is checked if there already exits a crack so that the column is ordered in a way If a index entry which matches the query parameters (value and inclusive flag) is found, no crack needs to be done and the query just needs to return the position from the entry. If no matching entry is found the query needs to take a look at the entries which contain the next smaller and the next larger element for which a crack exists the position stored in these entries mark the boundaries of the piece which needs to be cracked by the query in order to find its result.

A special case of an index search occurs when an entry is found which matches the query value but not the inclusive flag. In this case the cracking column needs to be recracked for the affected value since no piece exists which contains all values of the result. This can be demonstrated by a short example. Let us assume a query wants to find values x which satisfy $x \leq 5$. While searching the index an entry containing 5 and false is found which means a crack < 5 exists. The entry contains the position 8. The query could thus find all values smaller than 5 but not 5 itself if it exists in the column. Since the postition of 5 in the column cannot be determined a recrack needs to happen.

2.4 Advantages of Database Cracking

The Database Cracking approach has several advantages which make it an interesting approach for organizing and indexing data.

As already mentioned, database cracking enables the database to be self-orgarnized. The data is ordered in a favourable way for the queries without the need for upfront knowledge about the workload. The second advantage of cracking is that it is performed on the cracker column instead of the original one. Although this may seem like a drawback because the higher memory consumption, the cracking column removes the necessity of copying query results to a seperate output space. Since copying elements causes a lot of memory accesses this feature improves the performance of queries by a lot. Another positive aspect is that the cracking of the physical database can be supported by the cracking index. If queries find an enrty in the index which matches their search value as well as their search condition cracking can be omitted in the query procedure which speeds up the execution. Even if no matching entry is found, existing pieces reduce the range of the cracking column which needs to be considered when the query has to crack.

3 Implementation

In this section I will present my implementation of Database Cracking. It consists of multiple cracking algorithms as well as a structure which uses these algorithms combined with an index structure in order to perform queries. Furthermore I added a range scan on an existing B^+-Tree implemetation for comparison purposes.

The implementation relies on various simplifications which reduce otherwise necessary overhead and promote the comparability between cracking and regular indexing. Albeit these assumptions also reduce code complexity a complete cracking scenario is implmeneted which provides enough functionality to compare it to other indexing techniques. The simplifications are:

1. Distinct values in the column

2. Only one column is cracked

3. Only lookups are supported, no inserts and deletes

In theory cracking algorithms are able to handle non distinct values in a column. But as it needs a workaround to handle non-distinct values in B^+-Trees we only allow distinct values. Since I want to focus on the cracking aspect I only handle one column which is represented by a pointer to an array. Furthermore my implementation does not offer means to add or remove elements to the column. Implementing this would produces overhead and is not needed to show the concept of database cracking.

3.1 Cracking Algorithms

The current implementation supports all three cracking algorihtms presented in 2.2. On top of the execution of the algortihms, the corresponding C++ methods return the position in the cracking column which points to the last element of the left piece resulting from the cracking which means the last element that satisfies the cracking condition.This position corresponds to the position which needs to be stored in the cracking index (see 2.3). In order to use the incusive flag to determine which comparison operators should be used for Δ_1 and Δ_2 in the cracking algorithms the following conditions are evalutated:

$$X\Delta_1 Y : \ (X < Y) \mid\mid ((X == Y) \ \&\& \ \mathit{inclusive})$$

$$X\Delta_2 Y : \ (X > Y) \mid\mid ((X == Y) \ \&\& \ \mathit{inclusive})$$

3.2 Cracking index structure

The cracking index structures combines a cracking index with the cracking algorithms from the preceding chapter. It offers means to maintain the index and methods to perform range queries on the values of a column. The structure consists of a pointer to the column, the size of the column, a pointer to the cracking column and a map which is used as cracking index. The map stores pairs of values and entry structs. The values represent the bound values of pieces while entry structs store information about their position as well as the inclusive flag.

Additionally the index struct offers a method to find a matching piece for a query. A matching piece is either a piece which satisfies the range condition of the query or the smallest known piece which needs to be cracked in order to get a satisfiying one. The method searches the map for the bound values it requires. If it finds a map element containing the desired value the inclusive flag from the entry struct is checked. If the inclusive flag matches the flag from the query a matching piece was found and the position of the piece is returned. If the inclusive flag of the entry does not match or no map element for the bound value was found, the method returns a piece which contains the desired bound value but is not already cracked. On this piece a crack or a recrack needs to be performed.

In order to be able to insert new entries into the cracking index a method is provided which adds information about cracking pieces to the map. The method decides between two cases: If no entry for the bound of the query is found, a whole new value-entry pair is inserted in the map. When an entry for the bound value exists but the inclusive flag doesnt match only the entry struct is replaced in the existing map element with a new one containing the new position and new inclusive flag. The latter case is used to adapt the index after a recrack.

Beside these helper methods the cracking index struct implements two query methods. One of them handles single bound queries while the other one handles double bound queries. The query methods return the start and end position of the piece/pieces containing the result values within the cracking column. The following steps are roughly performed during the execution of the query methods:

1. Search index for matching piece(s)
2. Return if exact matchings were found
3. Otherwise crack found piece(s)
4. Add newly created cracks to index
5. Return

While the method which handles single bound query is fairly simple, the double bound query has multiple cases which it needs to cover. The „find piece“ method needs to be called twice as one piece for each bound needs to be found. Depending on the result of these method calls, different cracking scenarios have to take place. In case both methods return an exact match the query is finished and able to return the piece containing the results. If only one of them finds an exact match, a crack for the bound value which did not correspond to an index entry needs to be done. The most interesting cases are the ones where no match is found for either of both bounds. If no exact matches are found by the „find piece“ method, it returns the start and end position of a piece which needs to be cracked. A comparison of start and end position for the lower bound piece and the upper bound piece is done. If for both bound values the same start and end positions are received, the bound values reside in the same piece. In this case the „crack in three“ algorithm needs to be executed. If the bound values share either the same start or the same end position two cracks need to happen. It is important that in this case, the crack for the lower bound is performed first as the crack for the upper bound needs its result to be performed correctly. In case lower and upper bound have neither start nor end position in common, they reside in two different pieces. Both pieces can be cracked independently from each other.

3.3 B-Tree implementation

For comparison purposes I extended the B^+-Tree implementation of [3] in order to be able to scan leaf nodes. The implementation already contained the code for inner nodes, leaf nodes and full trees which make use of the node implementations. Both node structs provide means to insert into and split them as well as finding an lower bound for a certain key. The tree code offers methods to perform inserts in the tree and lookups of keys and their respective pasyloads. These methods make use of the corresponding node methods.

As I want to use the B^+-Tree as index, it stores the values from a column as keys and their corresponding column position as payload. In order to realize a leaf node traversal, leaf nodes now have a pointer to their right sibling. Due to the implemented splitting mechanism the first leaf created in a B^+-Tree is always the leftmost leaf in the tree which means it always contains the entries with the smallest keys. To make use of this property I additionally added a pointer to the B^+-Tree code which references this „first“ leaf. This pointer simplifies scans starting from the smallest key in the tree.

In order to make use of the extended B^+-Tree implementaion in range scans on a column, I embedded it in an index struct similar to the cracking index struct described in [3.2]. It also offers query two methods for single bound and double bound queries. Both query methods operate logically in the same way:

1. Determine start position within start leaf
2. Determine end end position within end leaf
3. Start traversal of leaves
4. For each key lookup the position within column
5. Use positions to copy values from column to output
6. Stop traversal when end position is reached

In contrast to the query methods form the cracking index struct the single bound query needs an indicator whether the bound represents a lower or upper bound. According to this indicator the bound parameter is either set as the start position (lower bound) or the end position (upper bound). In the former case the end position is the last position in the rightmost leaf node, in the latter case the start position is the first position in the leftmost leaf. The double bound query does not need suc an indicator as it dermines it start position with the help of the lower bound and its end position with the help of the upper bound.

3.4 Test scenarios

My implementation features two main test cases. The first test case aims at determining which of the „crack in two“ algorithms performs better. Depending on the result of the evalutation of this test case I determined which of both algorithms in the final implementation. The test scenario has the following properties:

- column with 500’000’000 values
- single crack is
- performed twice with two different workloads:
  - crack with small piece as result
  - crack with large piece as result

The second test case compares the database cracking range queries with range queries using the B^+-Tree index. The properties of this test case are:

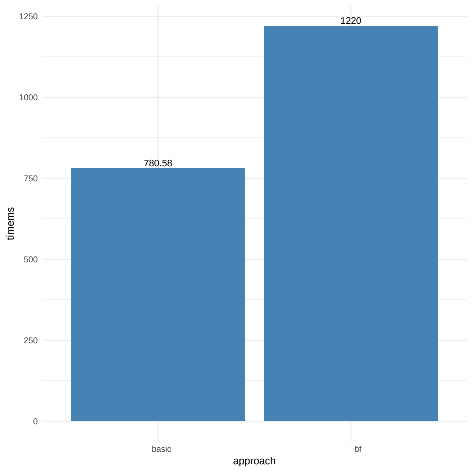
- column with 500’000’000 values
- 100 cack are performed consecutively
- performed twice with two different workloads:
  - workload causing a single crack
  - workload causing crack in every single query

4 Evaluation

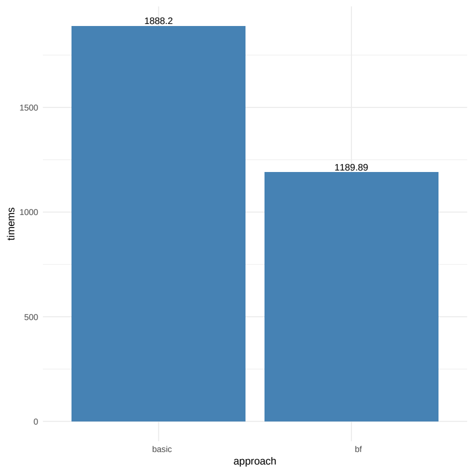
This section shows the results of the test scenarios described in section 3.4. The results are briefly discussed.

4.1 Comparison of cracking algorithms

The result of the comparison of the „Crack in two“ algorithms can be seen in the following figure:



(a) small resulting piece

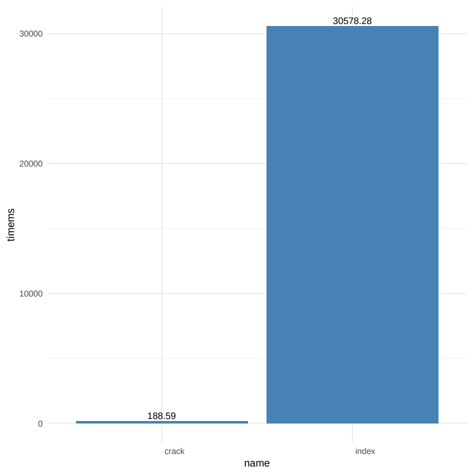


(b) big resulting piece

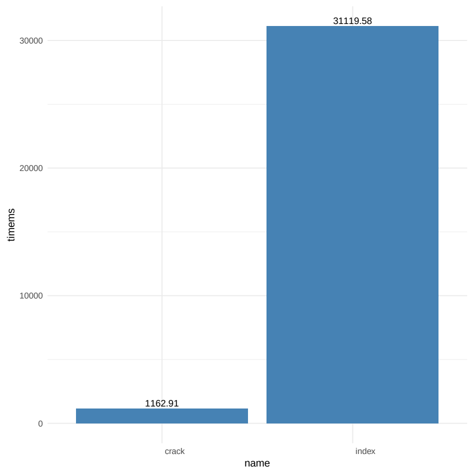
We can see that the two approaches performs differently well depending on the size of the resulting cracking piece. Based on this result I chose to use the basic algortihm within the cracking index struct as over time the size of the resulting cracks within a database cracking scenario reduces.

4.2 Database Cracking versus Indexing

The following fogure show the results of the cracking versus indexing test:



(c) single crack workload



(d) only cracks workload

The graphs indicate that cracking performs considerably better than B^+-Tree indexing. This is no surprise for the optimal workload since it only a single crack is triggered while the rest of the queries are answered with cracking index lookups only. More interesting are the results or the workload which produces cracks in every query. Although this is the wirst case for database cracking as cracking takes more time than just cracking index lookups, the query performance is still better than the performance of the B^+-Tree based queries. The extreme performance differences between both approaches can be explained with memeory accesses. The indexing approach requires to copy the results to an output array which takes takes a fairly large amount of time. This confirms the statement from the introduction that a regular index structure should not be used when a lot of records are touched by a query. As we can see this is not the case for database cracking.

5 Conclusion

In this paper I provided fundamental knowledge about Database Cracking. I presented different cracking algorihtms as well as ideas on how to maintain a cracking index. Furthermore I introduced an implemenetation using database cracking to perform range queries on single columns and compared them to range queries using a B^+-Tree as index. As seen in the evaluation section the cracking based queries performed significantly better than the index based queries. The use of an cracking index storing information about cracking pieces combined with the lack of the necessity to copy elements to an output space is able to speed up the query execution so that even a query workload which causes a lot of cracks can be handled efficiently. Thus i would consider database cracking as an interesting approach to organize and index data within a database system.

Literatur

[1] Stratos Idreos. Martin L. Kersten. Stefan Manegold. Database Cracking. CIDR, 2007.

[2] Holger Pirk. Eleni Petraki. Stratos Idreos. Stefan Manegold. Martin Kersten. Database Cracking: Fancy Scan not Poor Man's Sort. DaMoN Workshop, 2014.

[3] Viktor Leis. Lecture: Data Processing on modern Hardware. TUM, winter term 2017. https://db.in.tum.de/teaching/ws1718/dataprocessing/?lang=de