Database Cracking

David Werner

January 23, 2018

Technische Universität München

Table Of Contents

- Database cracking
  - Basics
  - Cracking index
  - Algorithms
  - Advantages
- Implementation
  - Cracking Algorihtms
  - Cracking index struct
  - $B^+ - Tree$
- Evaluation

Database cracking

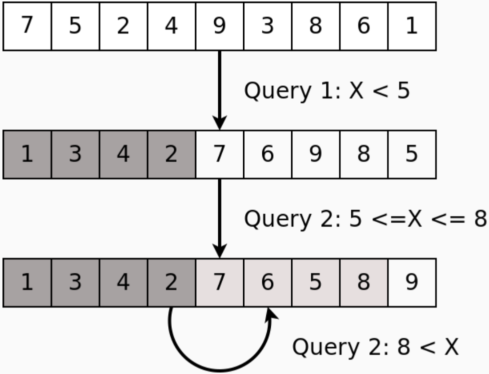
What is database cracking? - 1

- self organized indexing and index maintenance
- queries are used as advice to crack the database in pieces
- cracking means physically reordering the database
- sequential access for range queries is guaranteed

What is database cracking? - 2

- original column stays in insertion order
- cracking column is used for reordering
- this allows fast reconstruction of records

Cracking example



7	5	2	4	9	3	8	6	1

Query 1: X < 5

1	3	4	2	7	6	9	8	5

Query 2: 5 <=X <= 8

1	3	4	2	7	6	5	8	9

Query 2: 8 < X

Cracking index

- index on cracking column
- stores information about every crack
    - bound value
    - end position of piece
    - inclusive flag

Cracking in two pieces - basic

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
```

Δ_1 is $<$ or $\leq$, Δ_2 is $>$ or $\geq$ depedending on the inclusive flag

Cracking in two pieces - branch free

Algorithm 2 Crack in two pieces (branch free)

```
procedure CRACK_IN_2_BF(column, left, right, value, inclusive)
    cmp
    $active \leftarrow column[left]$
    $backup \leftarrow column[right]$
    while $left < right$ do
        $cmp \leftarrow active\ \Delta_1\ value$
        $column[left] \leftarrow active$
        $column[right] \leftarrow active$
        $left \leftarrow left + cmp$
        $right \leftarrow right - (1 - cmp)$
        $active \leftarrow (column[left] * cmp) + (column[right] * (1 - cmp))$
        $swap(active, backup)$
    end while
    $column[left] \leftarrow active$
end procedure
```

Cracking in three pieces

Algorithm 3 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 $column[left]\ \Delta_1\ value2$ do
            if $column[left]\ \Delta_1\ value1$ then
                $swap(column[left], column[tmp])$
                $tmp \leftarrow tmp + 1$
            end if
            $left \leftarrow left + 1$
        end while
        while $left < right$ and $column[right]\ \Delta_2\ value2$ do
            $right \leftarrow right - 1$
        end while
        if $left < right$ then
            $swap(column[left], column[right])$
        end if
    end while
end procedure
```

Advantages

Database cracking has some interesting properties:

- no copying of query results
- no updfront knowledge about workload required
- physcial reordering can be supported by index
- consecutive cracks receive speed from index

Implementation

Cracking Algorihtms

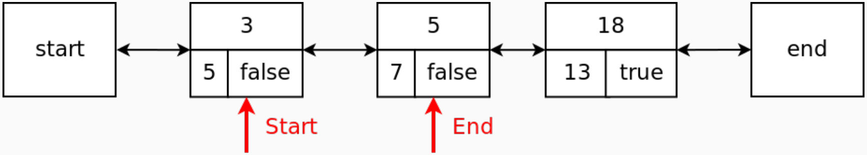
- All three cracking algorithms
- Return last position of piece in cracking column
- $<$ and $\leq$ cracks only
- $>$ and $\geq$ queries can use these results

Cracking index struct

- Combines cracking algorithms with cracking index
- Comprises:
  - Pointer to original column
  - Pointer to cracking column
  - Column size
  - Map as index
- Main functionality:
  - Find pieces
  - Query (single bound, double bound)

Find piece - 1

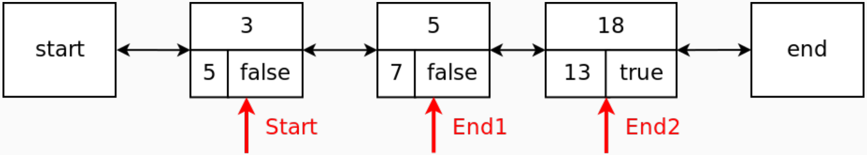
exact match:



returns: true

Find piece - 2

no match at all[1] or inclusive flag does not match[2] :



returns: false

Query

Two different types of queries

- single bound (e.g. $X < a$)
- double bound (e.g. $a < X < b$)

Query method interface:

- Require bound value(s) and inclusive flag(s)
- Return start/end position of result piece(s)

Query - single bound

simple control flow:

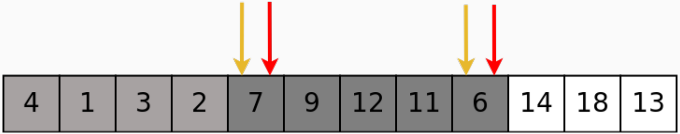
1. Find piece for value
2. If exact match: return
3. Otherwise: crack
4. Add crack to index
5. Return

Query - double bound

- Find piece for both bounds
- Depending on results different cases need to be handled
- Four easy cases:
  - None of both bounds needs a crack
  - Both bounds need crack in different pieces
  - Upper/lower bound needs crack
- Two involved cases

Special case 1

example query: $9 \leq X < 12$

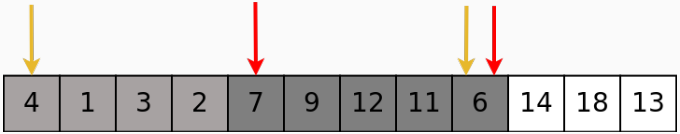


4	1	3	2	7	9	12	11	6	14	18	13

solution: crack in three pieces

Special case 2

example query: $4 < X \leq 13$

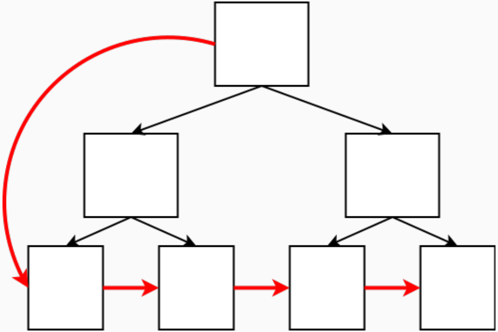


4	1	3	2	7	9	12	11	6	14	18	13

solution: crack yellow first, use result to crack red

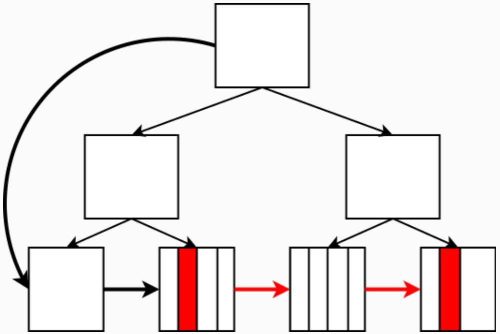
Extensions and Usage

- Extensions:
  - Leaves have sibling pointers
  - Pointer to leftmost leaf
- Tree stores:
  - bound values as keys
  - position and inclusive flag as payload



Query operation

1. Find start position
2. Find end position
3. Traverse leaves
4. Lookup column positions
5. Copy column values to output
6. Stop at end position



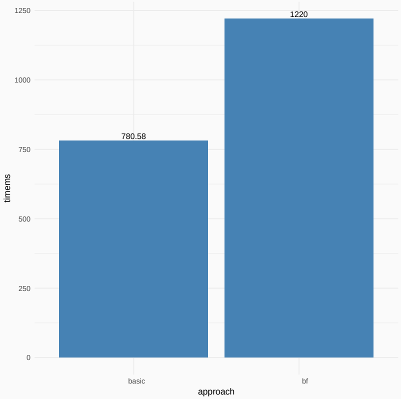
Evaluation

Test cases

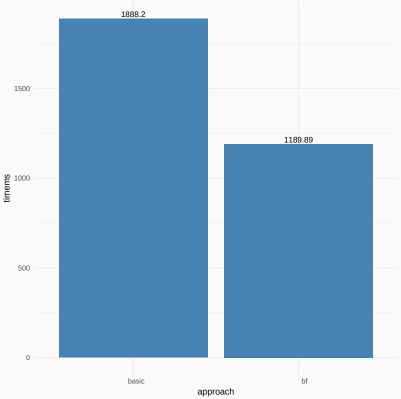
- Comparison of „Crack in two“ algorithms
  - 500’000’000
  - single crack
- Cracking vs. Indexing
  - 50’000’000 values in column
  - 100 consecutive cracks

Comparison of cracking algorithms

small result piece:

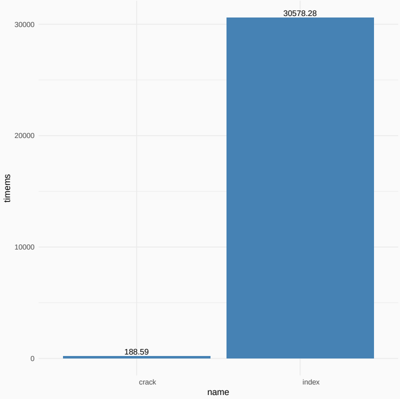


big result piece:

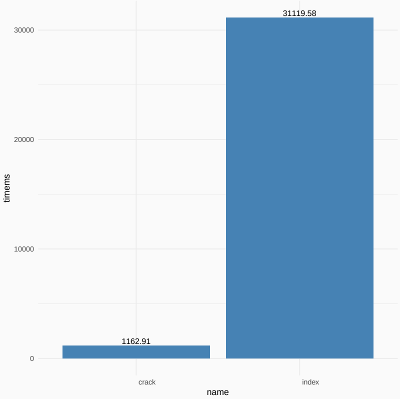


Cracking vs Indexing

single crack workload:



only cracks workload:


timems
crack
index
name