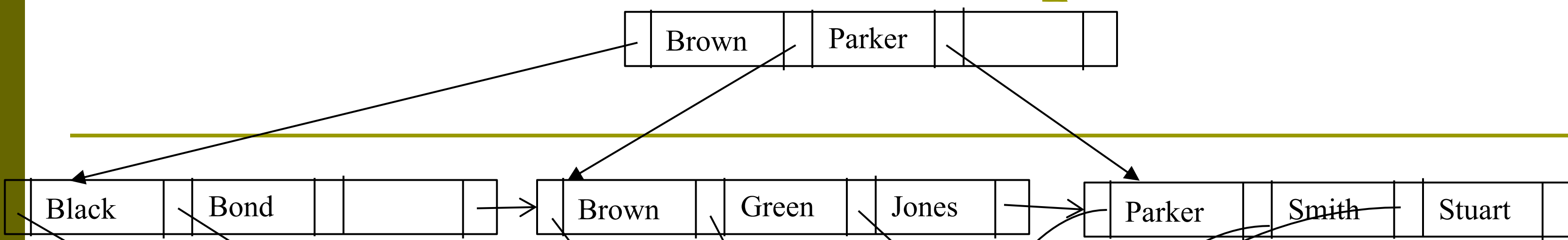
Adaptive Indexing

- Why adaptive indexing?
- Standard database cracking
- Adaptive merging
- Stochastic cracking
- Coarse granular index
- Multidimensional adaptive indexing

Background: Database Indexing

- Indexing is a fundamental approach in database systems that facilitates search
- Objective: given a search-key value q for attribute A of a DB table T, find fast the record(s) r in T for which r.A = q
  - Range query: find records s.t. $q_1 \le r.A \le q_2$
- An index is a table of <search-key, rid> pairs, organized in a data structure
  - search-key is a value of A in table T
  - rid is an identifier or pointer to a record (or set of records) in T for which A = search-key

B+-Tree Index: Example



Employees file

12-824	Jones	24	13000	22
13-322	Smith	36	25000	45
13-324	Parker	52	20000	125
17-307	Stuart	55	56000	12
19-564	Bond	37	31000	34
20-111	Black	29	17000	22
21-397	Brown	21	12000	45
31-936	Green	30	15000	10

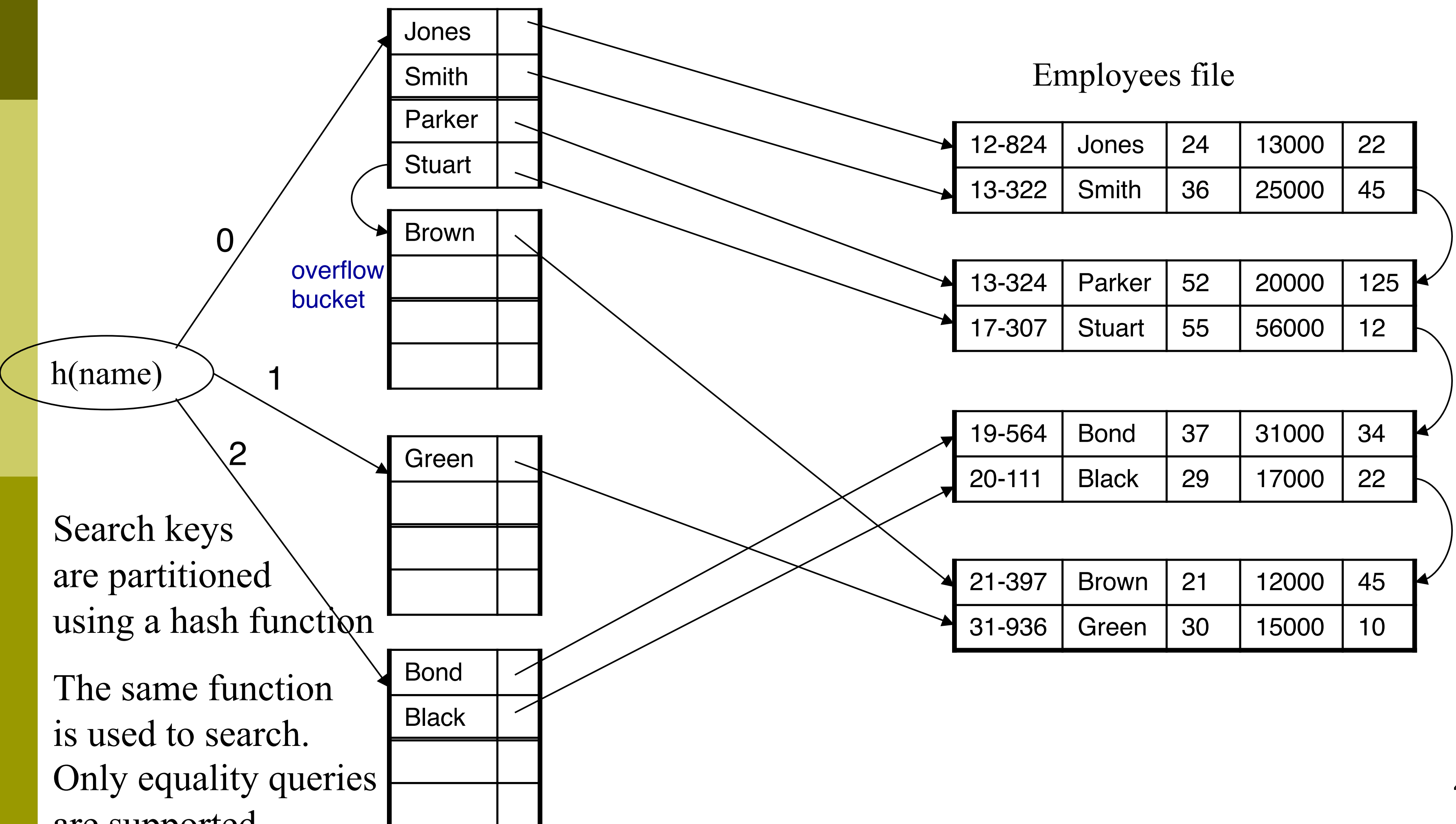
The leaf level of the B^+-tree contains <rid, search-key> values sorted by search-key

The non-leaf nodes of the tree are used to guide search

B^+-trees are appropriate for

- equality queries
- range queries

Example of Hash Index



Search keys are partitioned using a hash function

The same function is used to search. Only equality queries are supported

Indexing in main memory

- B^+-trees and hash tables can be constructed and used in main memory
- Motivation: In most applications nowadays data can fit in memory
- Sorting the index table by search-key and using binary search can be an effective index
  - Simple solution that can be used both for equality and range queries
- Binary search trees (AVL, red-black) can also be used if we want to support updates

Sorted index table

Employees file

index table

Black	
Bond	
Brown	
Green	
Jones	
Parker	
Smith	
Stuart	

12-824	Jones	24	13000	22
13-322	Smith	36	25000	45

13-324	Parker	52	20000	125
17-307	Stuart	55	56000	12

19-564	Bond	37	31000	34
20-111	Black	29	17000	22

21-397	Brown	21	12000	45
31-936	Green	30	15000	10

When do we create a database index?

- Indexes are expensive to build and maintain, so they should be used only when necessary
- Offline analysis
  - Most commercial DBMS include auto-tuning tools
  - Monitor a running workload for a while and then decide what indexes to create or drop based on query patterns
  - Not appropriate if there is no query history or query workload is dynamic
- Online analysis
  - Monitor workload and performance while processing queries
  - Trigger (dynamically), the creation of new indexes or dropping unnecessary indexes
  - Monitoring and creating/dropping full indexes is quite expensive

Database Cracking

Reading: S. Idreos, M. L. Kersten, and S. Manegold, "Database Cracking," in Proceedings of the 3rd International Conference on Innovative Data Systems Research (CIDR), Asilomar, California, 2007, pp. 68-78

Adaptive indexing: motivation

- Constructing a complete index is expensive compared to a simple data scan
- If the index is not used frequently its construction does not pay off
- Some parts of the data may be searched more frequently than others
  - It may be worth to index the data only partially
- Solution: database cracking
  - Evaluate queries using linear scan
  - Opportunistically update index based on queries, then progressively guide search

Database cracking: background

- DB cracking is based on sorting algorithms such as quicksort
- Quicksort places a given value (the pivot) in the correct position in the array
  - Performs in-place partitioning of the array into two pieces, based on the pivot (smaller than, greater than)
- Then, it recursively sorts the two partitions
- Example:
  - array: {52, 37, 63, 14, 17, 8, 6, 25}, pivot: 25
  - array becomes {$\underbrace{6, 8, 17, 14}_{\text{smaller than pivot}}$, 25, $\underbrace{63, 37, 52}_{\text{greater than pivot}}$}

Quicksort algorithm

```
algorithm quicksort(A, lo, hi)
 if 0<=lo && lo < hi && hi <= |A|-1 then // indices in correct order
     p := partition(A, lo, hi) // partition A from lo to hi and get pivot position
     quicksort(A, lo, p) // recursively sort first partition
     quicksort(A, p + 1, hi) // recursively sort second partition

algorithm partition(A, lo, hi)
 pivot := A[floor((hi + lo) / 2)] // choose as pivot value the middle element of array
 i := lo - 1 // left index
 j := hi + 1 // right index
 loop forever
    do i := i + 1 while A[i] < pivot
    do j := j - 1 while A[j] > pivot
    if i ≥ j then return j // if indexes cross then pivot is in position j and partitions are correct
    swap A[i] with A[j] // because i<j, A[i] and A[j] are in wrong partitions, so swap them
```

Hoare's quicksort algorithm adapted from https://en.wikipedia.org/wiki/Quicksort

Pivot-based partitioning example

pivot := 14
lo = 0
hi = 7

i j
{52, 37, 63, 14, 17, 8, 6, 25}

i j
{52, 37, 63, 14, 17, 8, 6, 25} swap!

i j
{6, 37, 63, 14, 17, 8, 52, 25}

i j
{6, 37, 63, 14, 17, 8, 52, 25} swap!

i j
{6, 8, 63, 14, 17, 37, 52, 25}

i j
{6, 8, 63, 14, 17, 37, 52, 25} swap!

Hoare's quicksort algorithm adapted from https://en.wikipedia.org/wiki/Quicksort

Pivot-based partitioning example

returned value (j=2)

j i

{6, 8, **14**, 63, 17, 37, 52, 25} **i≥j ⇒ finished!**

smaller than pivot

greater than pivot

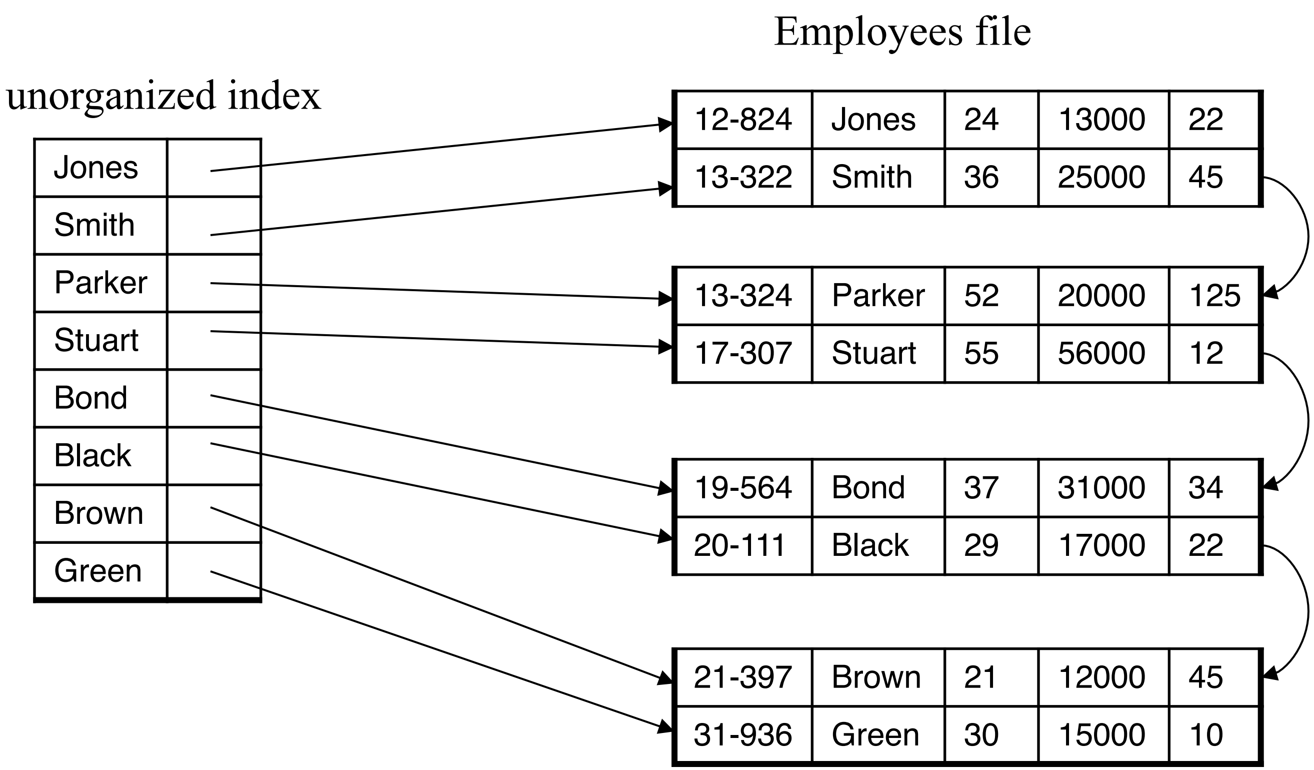
pivot := 14
lo = 0
hi = 7

- With one pass over the data, we crack the array into two pieces based on pivot value

Hoare's quicksort algorithm adapted from https://en.wikipedia.org/wiki/Quicksort

Standard database cracking

- Initially, we have an unorganized index



- For simplicity, assume that the index is an unordered array

Standard database cracking (cont'd)

- Example initial index (keys only):
  {52, 37, 63, 14, 17, 8, 6, 25}
- Example first query: 10<key<=20
- First, crack using 10<key

  i → ← j

  - {52, 37, 63, 14, 17, 8, 6, 25}
  - {6, 37, 63, 14, 17, 8, 52, 25}
  - {6, 37, 63, 14, 17, 8, 52, 25}
  - {6, 8, 63, 14, 17, 37, 52, 25}

  j | i

  - {6, **8**, | **63**, 14, 17, 37, 52, 25}

  10<key

Standard database cracking (cont'd)

- Example initial index (keys only):
  {52, 37, 63, 14, 17, 8, 6, 25}
- Example first query: 10<key<=20
- Second, crack using key<=20

i → ← j

  - {6, 8, 63, 14, 17, 37, 52, 25}
  - {6, 8, 63, 14, 17, 37, 52, 25}
  - {6, 8, 17, 14, 63, 37, 52, 25}

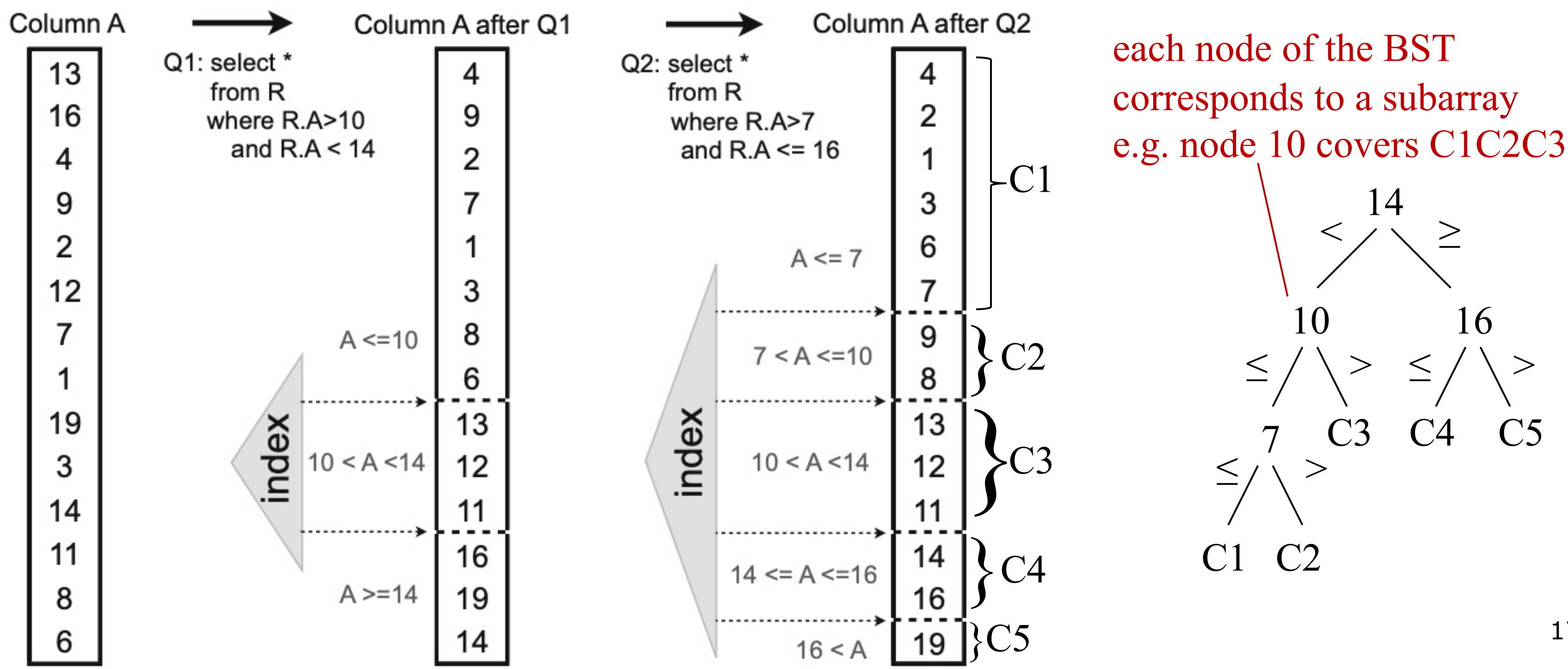
j i

  - {6, 8, 17, **14**, **63**, 37, 52, 25}

10<key key<=20

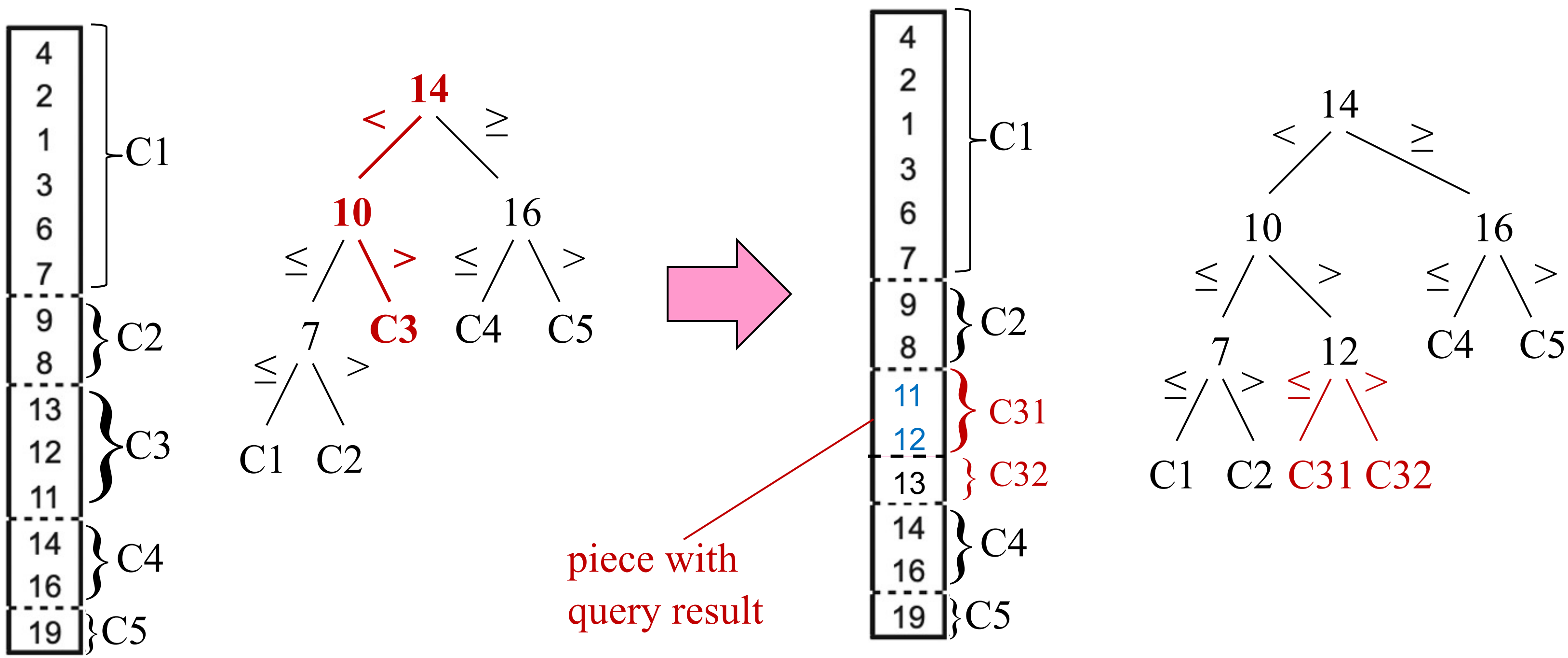
Standard database cracking (cont'd)

- After each query, the partition algorithm of quicksort is used to progressively crack the array
  - Query boundaries define the pivots
- A binary search tree organizes the cracks



Standard database cracking (cont'd)

- Example: new query (10,12]
  - Search tree (depth-first search)
  - Perform cracks at leaves whose range overlaps with query range; collect query results from new pieces



Crack-in-two vs crack-in-three

- Range queries select values in an interval range
- If the interval range falls entirely in an unordered piece the piece should be cracked in three

{…, 10, 18, 8, 14, 12, 34, 9, …}

Cracked piece: $7 < x \leq 38$

new query: (15,20]

cracking

{…, 10, 9, 8, | 14, 12, 18, | 34, …}

$7 < x \leq 15$ $15 < x \leq 20$ $20 < x \leq 38$

- Otherwise, at most two cracks-in-two are applied

{…, 10, 9, 8, 14, 12, 18, 34, 25, …}

$7 < x \leq 15$ $15 < x \leq 20$ $20 < x \leq 38$

new query: (9,30]

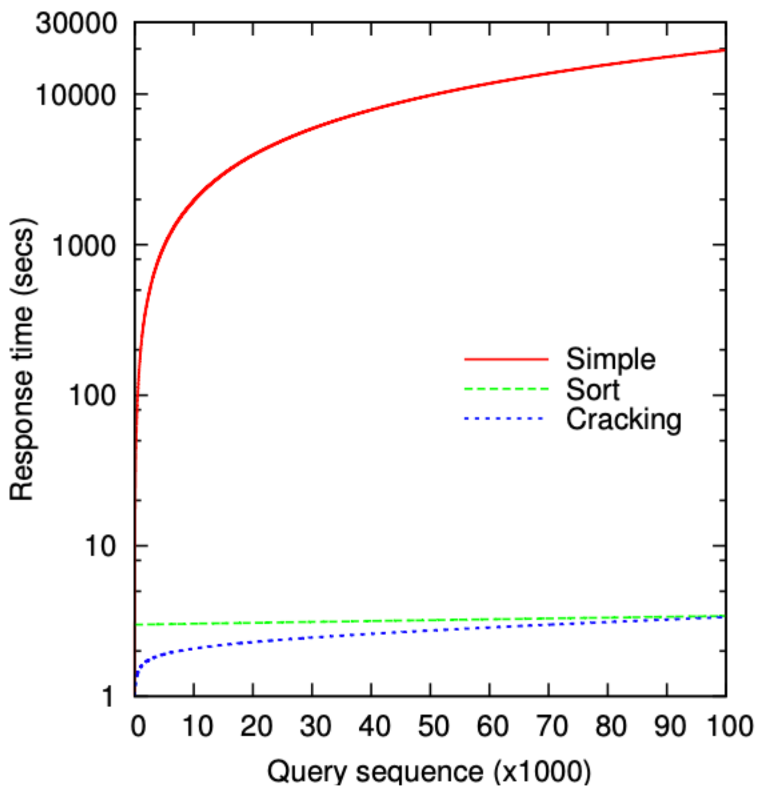
cracking

{…, 8, 9, | 10, 14, 12, 18, 25, | 34, …}

Cumulative query cost

Data: unordered array with 10 million distinct integers

Queries: $v_1 < x < v_2$, where v_1 and v_2 are random



Simple: answer each query by linear scan

Sort: sort data before 1st query, then binary search

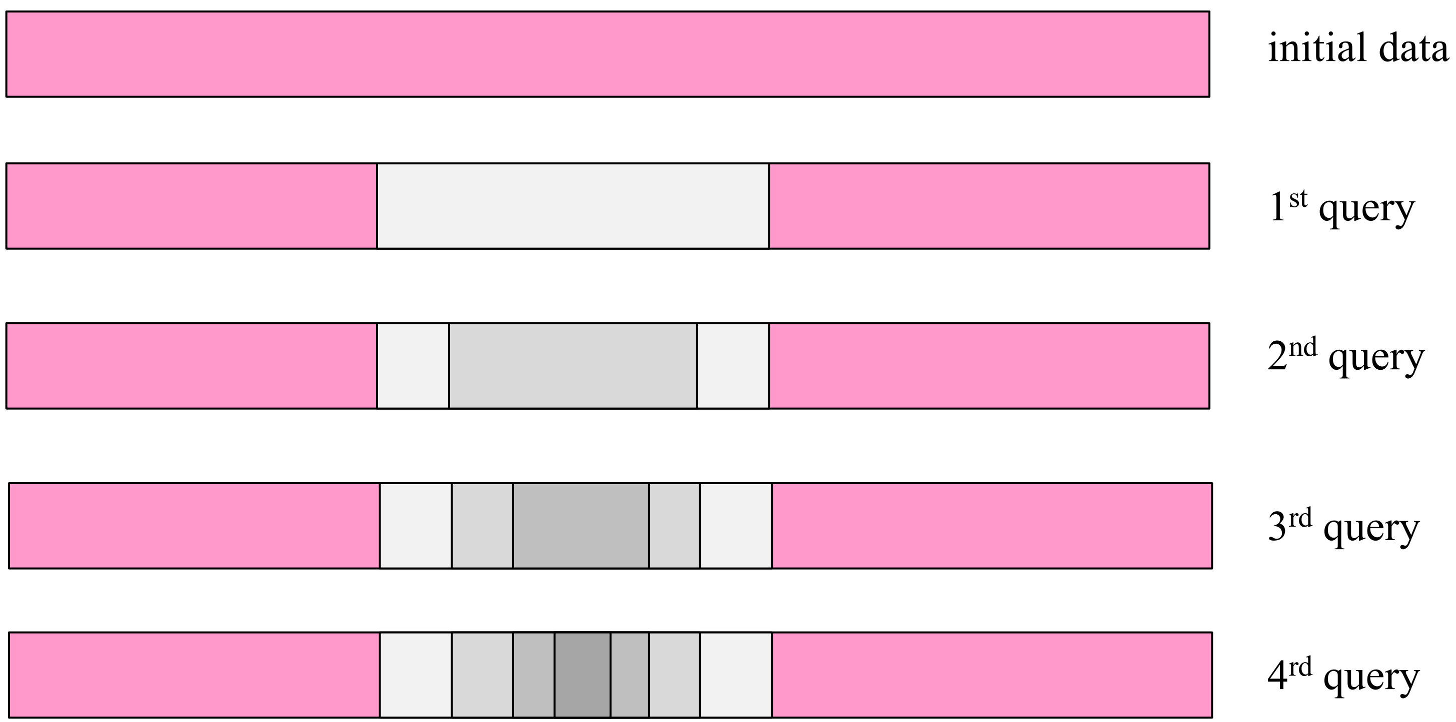
Cracking: perform cracking during queries

Notes on standard cracking

- After large number of random queries, the array becomes completely sorted
  - In practice, a piece is not cracked if it is very small
  - i.e. when sequentially scanning the piece is faster than BST-based searching of the corresponding subarray
- If the query workload is skewed and most queries fall within a limited range, we avoid accessing and/or sorting large parts of the array
- Cracking can be used to adaptively index columns in a column-store
- Cracking is not appropriate for disk-based data

Best case for cracking

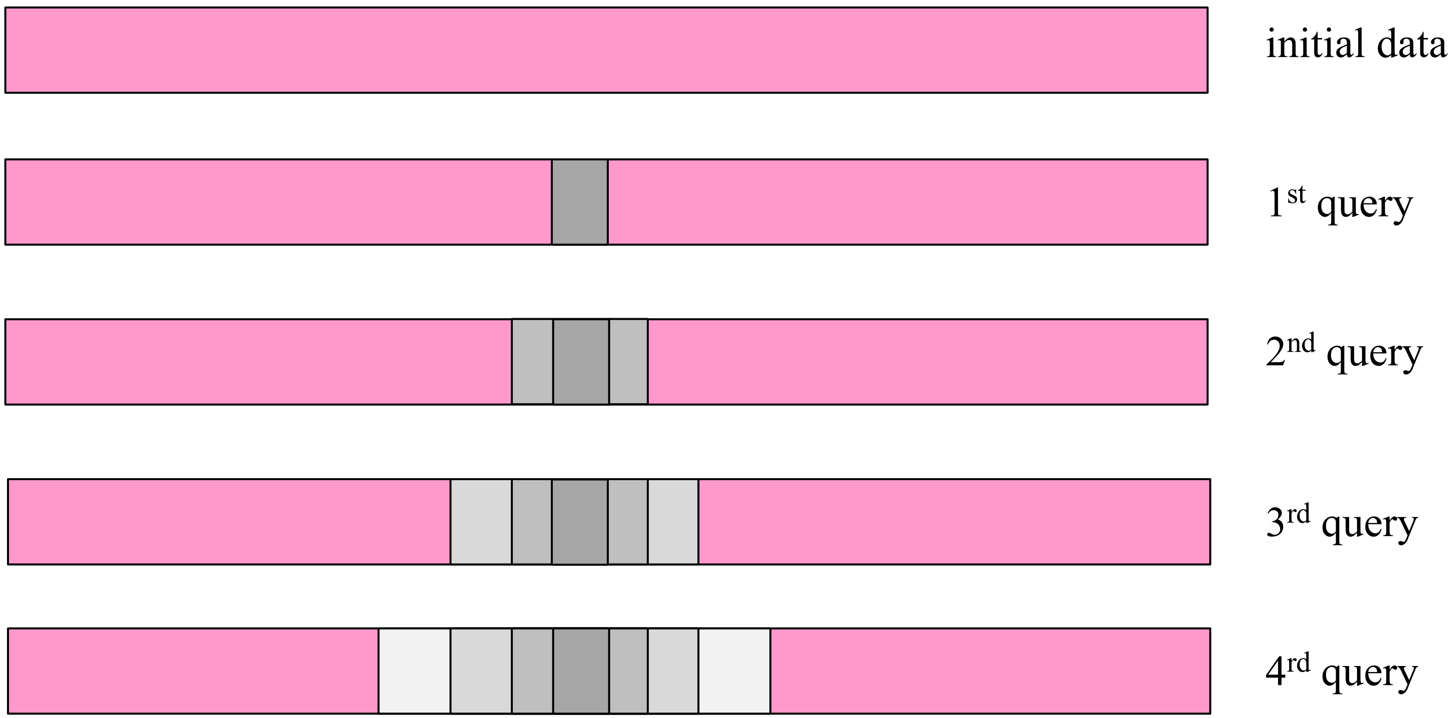
- Each query is a sub-range of the previous one



- Cracked pieces are always small

Worst case for cracking

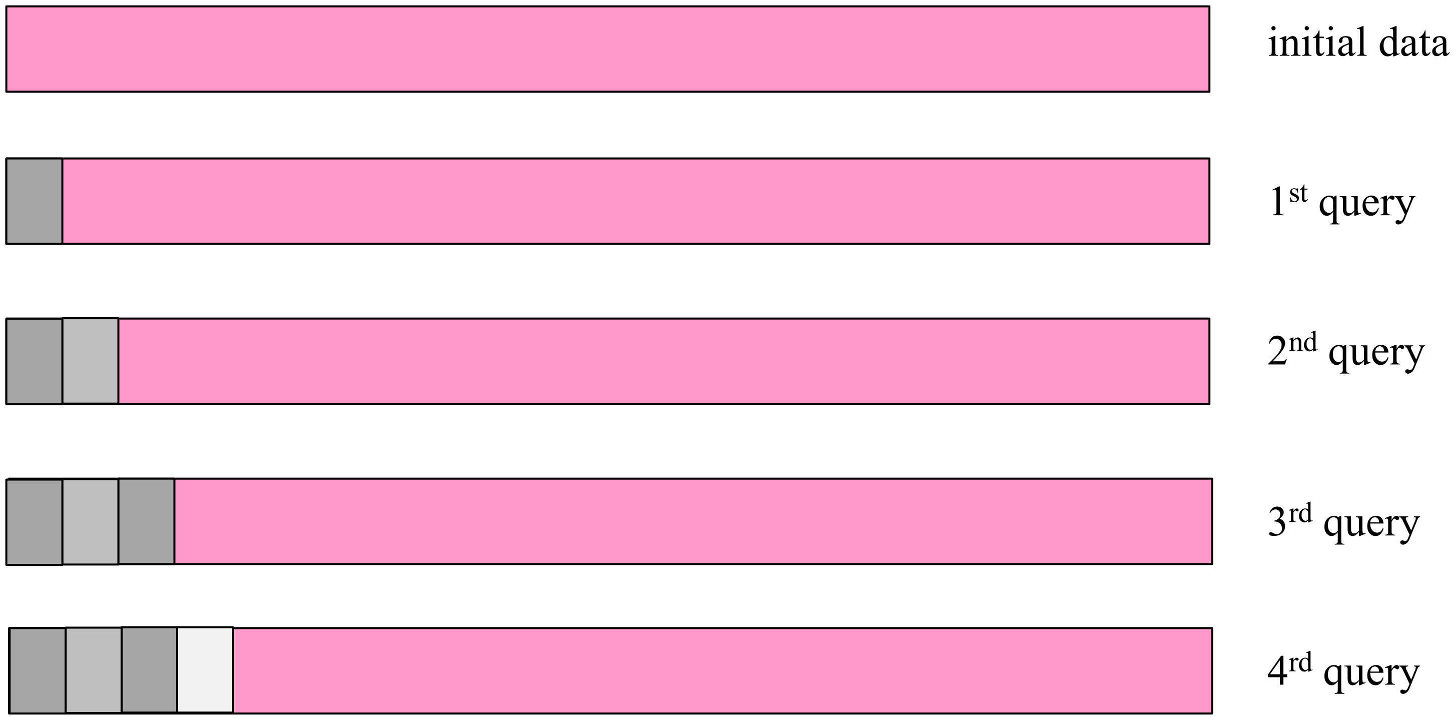
- Each query is a super-range of the previous one



- Big pieces are cracked multiple times!

Worst case for cracking

- First query asks for smallest element, second query for second smallest etc.



- Big pieces are cracked multiple times!

Adaptive Merging

Reading: Goetz Graefe, Harumi A. Kuno: Self-selecting, self-tuning, incrementally optimized indexes. EDBT 2010: 371-381

Adaptive Merging

- Motivated by merge-sort, instead of quick-sort
- Initially, divides the data into small partitions and sorts each of them
  - Sorting small partitions is much faster than sorting the entire array
- Each query selects segments from sorted partitions and merges them into a new partition
  - Binary search is used to search in partitions
- Merging may cause some original partitions to disappear
  - Eventually everything is merged to a single sorted array

Adaptive Merging (example)

{52, 37, 63, 14, 17, 8, 6, 25, 7, 10, 38, 20} original array

first pass: create initial sorted partitions

{14, 37, 52, 63} {6, 8, 17, 25} {7, 10, 20, 38}

search each partition and merge

new query: (15,20]

{17, 20} {14, 37, 52, 63} {6, 8, -, 25} {7, 10, -, 38}

search each partition and merge

new query: (5,10]

{6, 8, 7, 10} **{17, 20}** {14, 37, 52, 63} {-, -, -, 25} {-, -, -, 38}

not touched (out of query bounds)

Notes on Adaptive Merging

- B-tree-like indexes can be used to accelerate search in each partition
- Multi-way merge can be used to merge multiple segments
- Results of overlapping queries are stored in single partitions
  - query results are also partitions, which are merged with next queries
  - partitions that are query results are never split/cracked
  - eventually, all data are merged to a single sorted array with the results of all queries

Adaptive Merging vs. DB cracking

- Slow startup, as it requires sorting each initial partition
- Creating new partitions allocates new memory space and moves data in memory
  - “gaps” are created in original partitions
  - not as efficient as swaps performed by cracking
- Harder to implement compared to DB cracking
- Partitions that are query results are not split
  - excessive fragmentation is avoided
- Avoids moving a key multiple times, which may happen often in DB cracking
- Converges to a single sorted array much faster compared to DB cracking

Hybrid Adaptive Indexing

- Main drawback of DB cracking: slow convergence
- Main drawback of adaptive merging: slow startup
- Objective: a hybrid approach that combines the best from both strategies
- Main idea: start with some initial partitions as in adaptive merging, but not necessarily sorted
  - Option 1: apply cracking to initial partitions, to obtain the segments that should be merged to form a query result
  - Option 2: apply radix-clustering to initial partitions
    - Convert values to integer codes, use k most significant bits of codes to divide partition to 2^k segments
- Option 1 performs well in an experimental analysis

Hybrid Adaptive Indexing(example)

{52, 37, 63, 14, 17, 8, 6, 25, 7, 10, 38, 20} original array

create initial partitions (not sorted)

{52, 37, 63, 14} {17, 8, 6, 25} {7, 10, 38, 20}

crack each partition and merge

new query: (15,20]

{17, 20} {14, 37, 63, 52} {8, 6, -, 25} {7, 10, -, 38}

crack each partition and merge

new query: (5,10]

{6, 8, 7, 10} **{17, 20}** {14, 37, 52, 63} {-, -, -, 25} {-, -, -, 38}

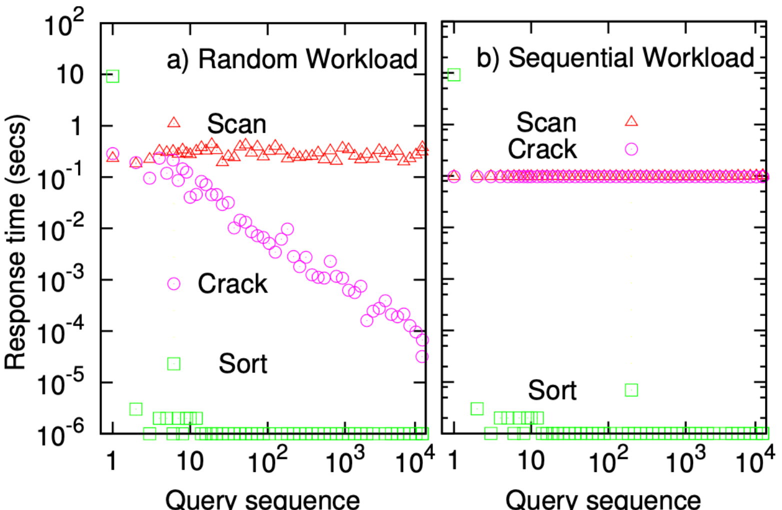
not touched (out of query bounds)

Stochastic Cracking

Reading: F. Halim, S. Idreos, P. Karras, and R. H. C. Yap, "Stochastic Database Cracking: Towards Robust Adaptive Indexing in Main-Memory Column-Stores," Proceedings of the Very Large Databases Endowment (PVLDB), vol. 5, no. 6, pp. 502-513, 2012

Stochastic Cracking

- Motivation: standard cracking assumes a random query workload
  - No prediction or care for future queries
  - Large pieces are left completely unindexed
  - Performs badly for specific workloads (sequential)



- However, access pattern is typically unknown in exploratory analysis
- Main idea: proactively perform additional cracks that can serve future queries
  - Extra crack points are determined to some degree by the queries

Stochastic Cracking: DDC

- Data Driven Center strategy
- Given a query Q=[a,b]
  - Recursively halve the piece(s) where Q falls, until it (they) become(s) sufficiently small
  - Then, crack the piece(s) based on [a,b]

{52, 37, 63, 14, 17, 8, 6, 25, 7, 10, 38, 20} initial array

first pass: crack based on median

new query: (6,7]

{14, 17, 8, 6, 7, 10} {52, 37, 63, 25, 38, 20}

second pass on 1^{st} piece: crack based on median

{8, 6, 7} {14, 17, 10} {52, 37, 63, 25, 38, 20}

third pass on 1^{st}/1^{st} piece: crack based on query

{6} {7} {8} {14, 17, 10} {52, 37, 63, 25, 38, 20}

Finding median is combined with cracking (in linear time)

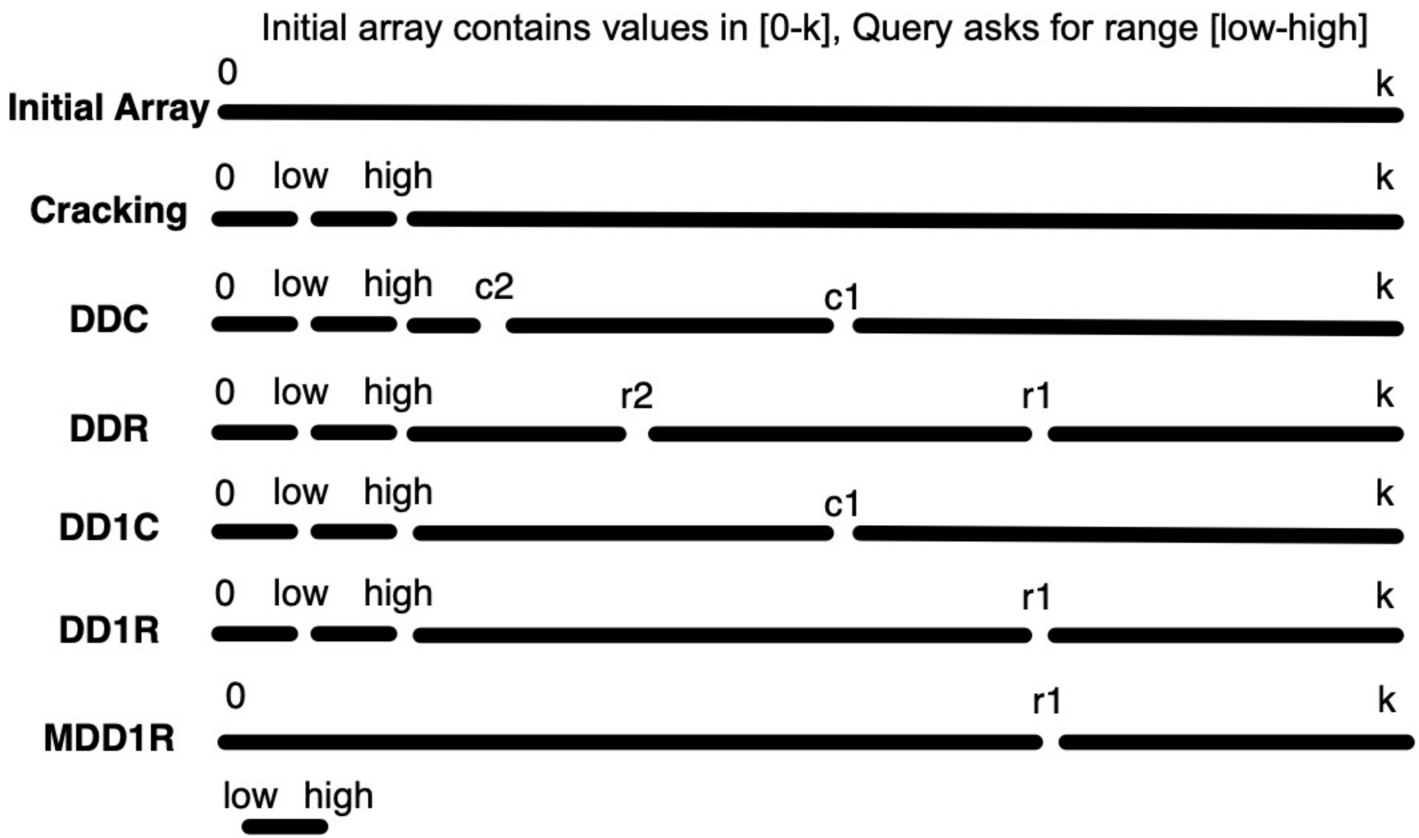
Stochastic Cracking: DDR

- Data Driven Random strategy
- Given a query Q=[a,b]
  - Recursively randomly crack piece(s) where Q falls, until it (they) become(s) sufficiently small
  - Then, crack the piece(s) based on [a,b]
- Uses randomly picked pivots for cracking recursively
- Avoids the overhead of finding medians
- Remains query-driven, since it recursively cracks only the piece that overlaps with query range

Stochastic Cracking: Other heuristics

- DD1C same as DDC, but performs at most one center crack before final query crack
- DD1R same as DDR, but performs at most one random crack before final query crack
- MDD1R same as DD1R, but does not perform final query crack; instead it copies query results to a separate array
  - Result identification and copying is done at the same time as the random crack
- Progressive MDD1R: perform only a small percentage of swaps at each crack
  - Lowers burden of first few queries
- Selective Stochastic Cracking:
  - selectively eschew stochastic cracking for some queries
  - such queries are answered using original cracking

Stochastic Cracking: all approaches



MDD1R is experimentally shown to be the best option in terms of overall cost for different query workloads

Coarse Granular Index

Reading: Felix Martin Schuhknecht, Alekh Jindal, Jens Dittrich: An experimental evaluation and analysis of database cracking. VLDB J. 25(1): 27-52 (2016)

Coarse Granular Index

- Motivation: First queries in DB cracking are slow because large parts of the data are read and possibly swapped
- Idea: build a coarse granular index
  - Create balanced partitions using range partitioning upfront
    - All data in one partition are smaller than all data in the next partition
  - Apply standard cracking later on
  - The cost of the first query is higher than standard cracking but significantly lower than full indexing

Coarse Granular Index

{52, 37, 63, 14, 17, 8, 6, 25, 7, 10, 38, 20} initial array

first query: partition data + crack

new query: (6,7]

{14, 17, 8, 6, 7, 10, 20} {37, 25, 38} {52, 63}

{6} {7} {14, 17, 8, 10, 20} {37, 25, 38} {52, 63}

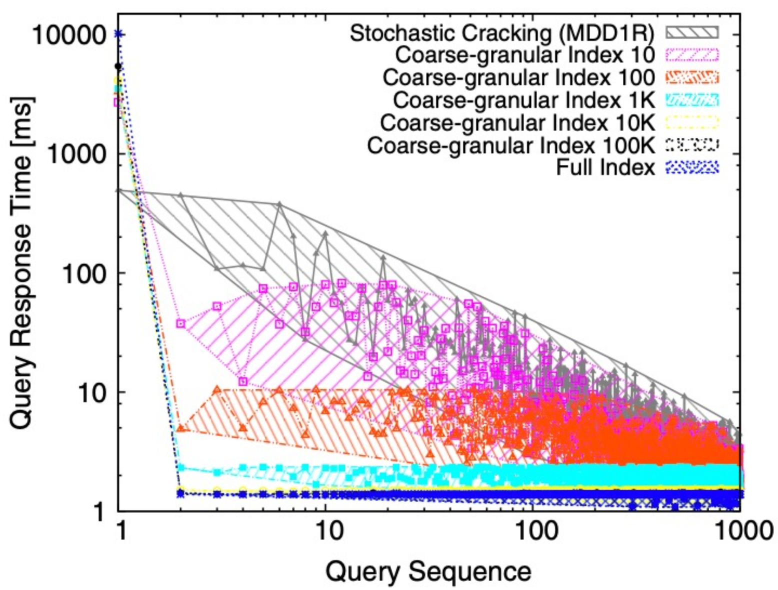
next queries: identify partition(s) + crack

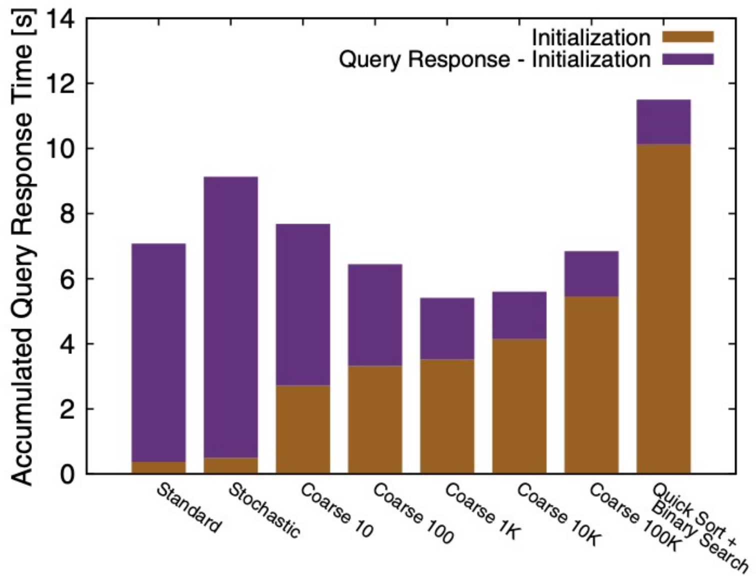
new query: (15,20]

{6} {7} {8, 10, 14} {17, 20} {37, 25, 38} {52, 63}

Coarse Granular Index

- Experiment:
  - 10^8 uniformly distributed values with a key range of [0; 100,000]
  - 1000 random queries of the form low ≤ x < high, each with selectivity 1%





Multidimensional Adaptive Indexing

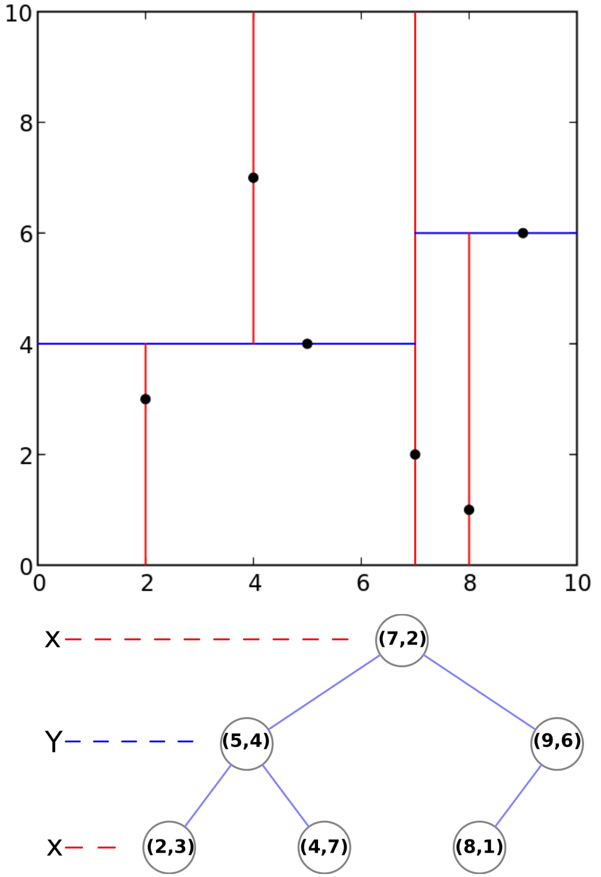
Reading: Matheus Agio Nerone, Pedro Holanda, Eduardo Cunha de Almeida, Stefan Manegold: Multidimensional Adaptive & Progressive Indexes. ICDE 2021: 624-635

Adaptive KD-tree

- Target: Exploratory range query workloads in multidimensional spaces
- Idea: Apply cracking in a multidimensional space using a KD-tree to index the cracked pieces
  - Cracking is performed at one dimension at a time
  - Each node of the KD-tree holds one attribute (dimension) and a value (split point)
  - Leaf nodes point to cracked pieces

Background: the KD-tree

- Reading: https://en.wikipedia.org/wiki/K-d_tree
- A binary tree for points in a k-dimensional space
- Root uses one point (median) to divide the space into two subspaces using one dimension
  - sort a sample to find median
- Each node uses a point to divide its subspace in two parts
- Dimensions alternate per level



Adaptive KD-tree example

Offset in array

Initialized Table

A	B	Off
6	5	0
3	9	1
16	4	2
13	2	3
2	8	4
1	11	5
8	7	6
19	19	7
7	12	8
12	20	9
11	3	10
4	6	11
9	16	12
14	2	13

Q: 6 < A ≤ 13 AND 5 < B ≤ 8

A ≤ 6

6 < A

Adapt (A, 6)

A	B	Off
6	5	0
3	9	1
4	6	2
1	11	3
2	8	4
13	2	5
8	7	6
19	19	7
7	12	8
12	20	9
11	3	10
16	4	11
9	16	12
14	2	13

A ≤ 6

6 < A

B ≤ 5

5 < B

Adapt (B, 5)

A	B	Off
6	5	0
3	9	1
4	6	2
1	11	3
2	8	4
13	2	5
14	2	6
16	4	7
11	3	8
12	20	9
7	12	10
19	19	11
9	16	12
8	7	13

tel:7%201210%2019191

A ≤ 6

6 < A

B ≤ 5

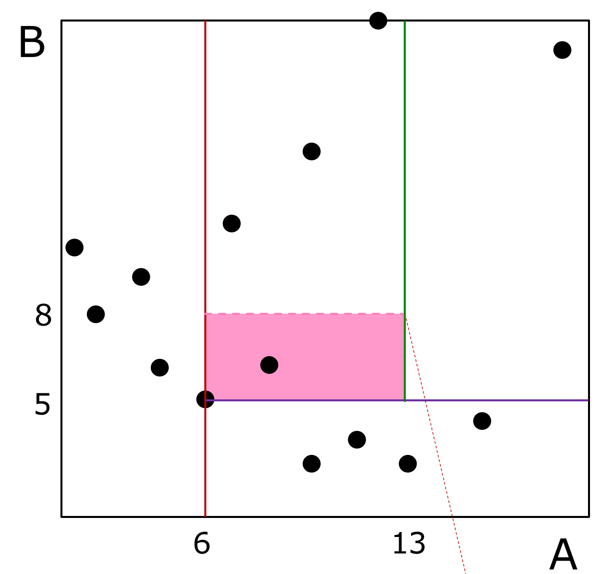
5 < B

A ≤ 13

13 < A

Adapt (A, 13)

A	B	Off
6	5	0
3	9	1
4	6	2
1	11	3
2	8	4
13	2	5
14	2	6
16	4	7
11	3	8
12	20	9
7	12	10
9	16	11
8	7	12
19	19	13



Cracking not done because size is 4 or less

first crack using lower Q bounds in each dim, then upper Q bounds

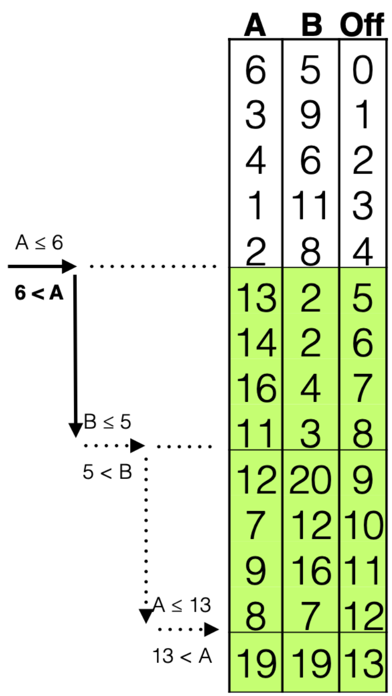
size threshold for cracking = 4

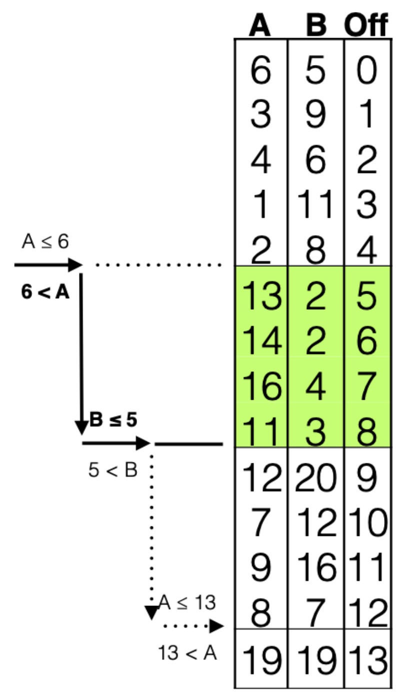
Notes on Adaptive KD-tree

- Progressively cracks each piece that overlap with query range
  - Crack based on lower bound in dimension 1, then based on l.b. in dim. 2, and so on
  - Repeat the above using upper query bounds in each dimension
  - Stop cracking when the piece becomes smaller than or equal to size threshold
  - At each crack a new KD-tree node is defined

Index Lookup

- When a new query arrives, the current KD-tree is used to guide search
  - When a leaf node is reached, it is cracked if necessary to introduce new nodes





Leaf node reached; since size is 4 or less, it is not cracked

new query: Q: 6 < A ≤ 15 AND 0 < B ≤ 5

Summary

- Adaptive, self-organizing indexes are dynamically constructed as a result of query evaluation
- Database cracking was the first idea toward this direction
- Another approach is adaptive merging
- Stochastic cracking improves standard cracking to handle non-random query workloads
- A cheap and coarse data partitioning before cracking is beneficial
- Cracking also applied to multidimensional data
  - KD-tree can be used to index cracked pieces