CS 525: Advanced Database Organization

06: Even more index structures

Boris Glavic

Slides: adapted from a course taught by Hector Garcia-Molina, Stanford InfoLab

COMPUTER SCIENCE

IIT College of Science and Letters

ILLINOIS INSTITUTE OF TECHNOLOGY

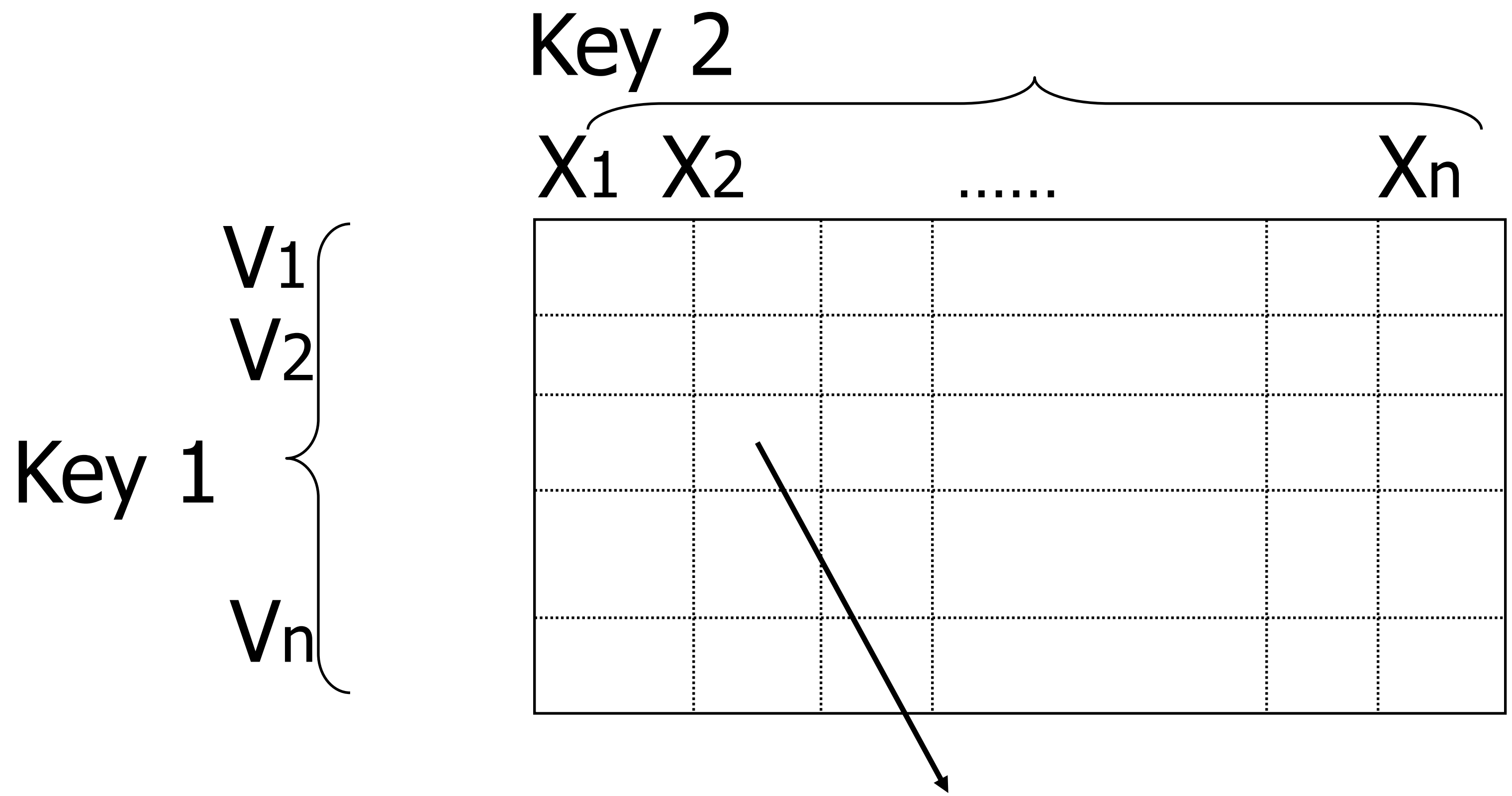
Recap

- We have discussed
  - Conventional Indices
  - B-trees
  - Hashing
  - Trade-offs
  - Multi-key indices
  - Multi-dimensional indices
    - … but no example

Today

- **Multi-dimensional index structures**
  - kd-Trees (very similar to example before)
  - **Grid File (Grid Index)**
  - Quad Trees
  - **R Trees**
  - **Partitioned Hash**
  - ...
- **Bitmap-indices**
- **Tries**

Grid Index



To records with key1=V_3, key2=X_2

CLAIM

- Can quickly find records with
  - key 1 = V_i $\wedge$ Key 2 = X_j
  - key 1 = V_i
  - key 2 = X_j

CLAIM

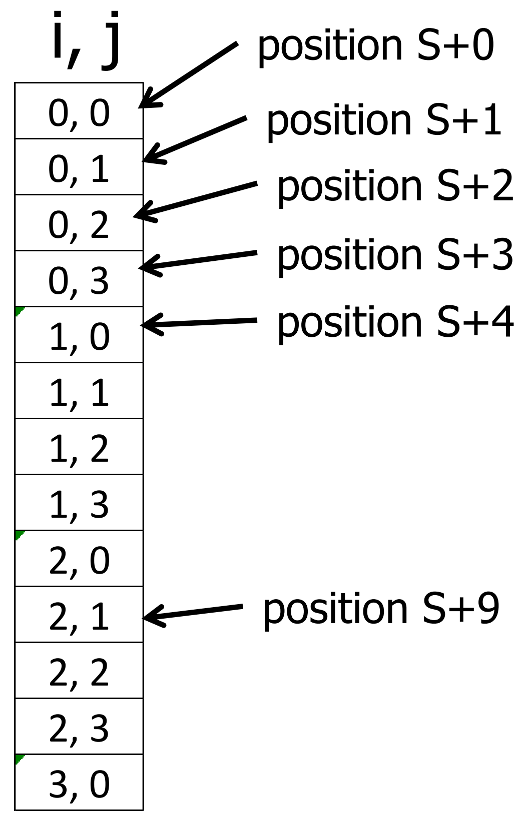
- Can quickly find records with
  - key 1 = V_i $\wedge$ Key 2 = X_j
  - key 1 = V_i
  - key 2 = X_j

- And also ranges….
  - E.g., key 1 $\geq V_i$ $\wedge$ key 2 $< X_j$

- How do we find entry i,j in linear structure?

max number of i values N=4

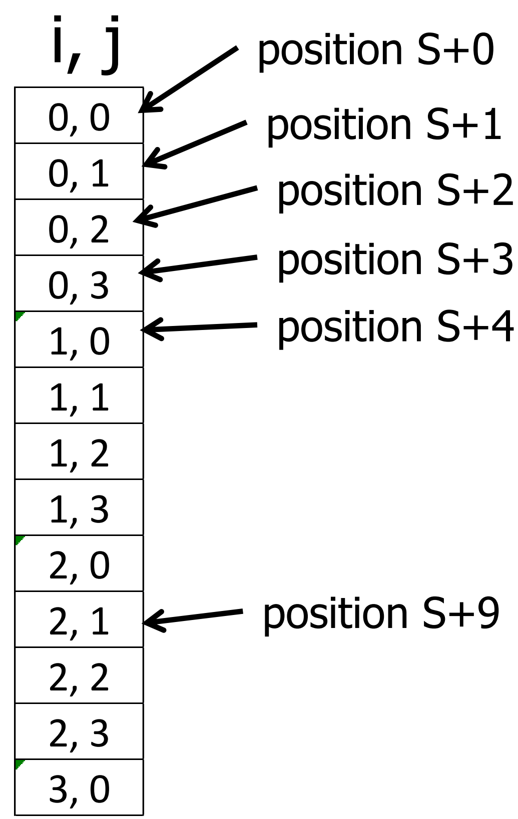
pos(i, j) =



- How do we find entry i,j in linear structure?

max number of i values N=4

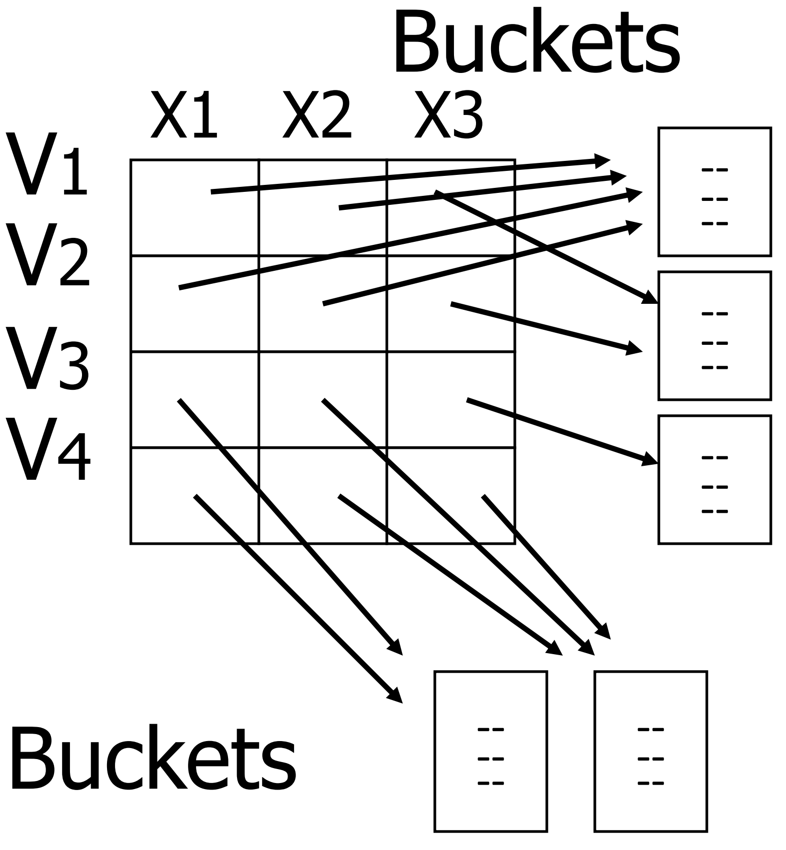
$pos(i, j) = S + iN + j$



Issue: Cells must be same size, and N must be constant!

Issue: Some cells may overflow, some may be sparse...

Solution: Use Indirection



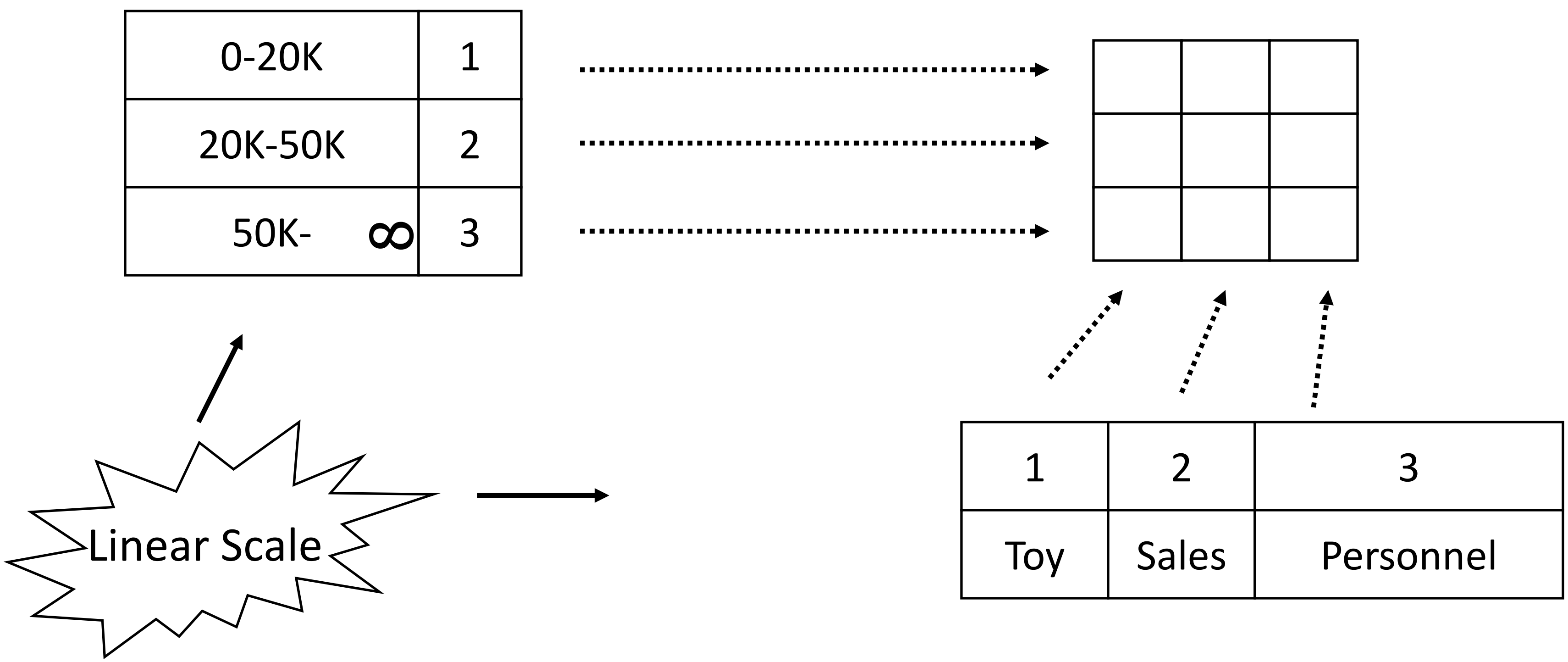
*Grid only contains pointers to buckets

With indirection:

- Grid can be regular without wasting space
- We do have price of indirection

Can also index grid on value ranges

Salary Grid



Grid files

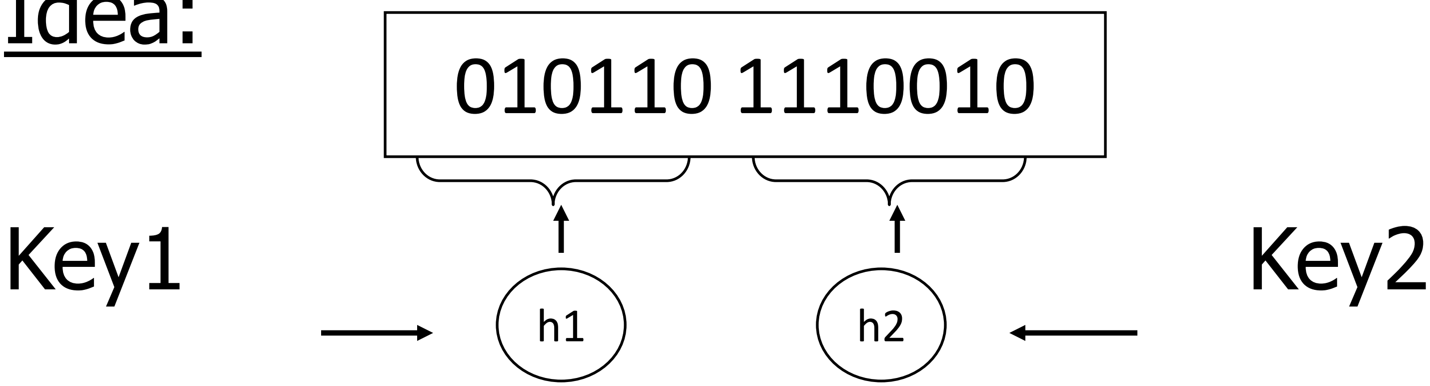
- ⊕ Good for multiple-key search
- ⊖ Space, management overhead

  (nothing is free)

- ⊖ Need partitioning ranges that evenly split keys

Partitioned hash function

<u>Idea:</u>



EX:

h1(toy) =0

h1(sales) =1

h1(art) =1

.

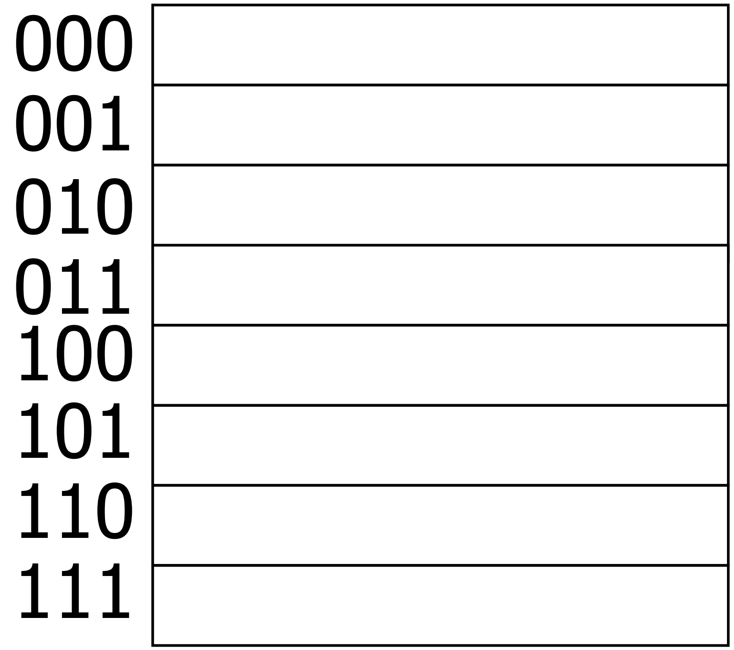
h2(10k) =01

h2(20k) =11

h2(30k) =01

h2(40k) =00

.



Insert <Fred,toy,10k>,<Joe,sales,10k>
<Sally,art,30k>

EX:

h1(toy) =0
h1(sales) =1
h1(art) =1
.
h2(10k) =01
h2(20k) =11
h2(30k) =01
h2(40k) =00
.

000	
001	<Fred>
010	
011	
100	
101	<Joe><Sally>
110	
111	

Insert <Fred,toy,10k>,<Joe,sales,10k>
<Sally,art,30k>

EX:

h1(toy) =0

h1(sales) =1

h1(art) =1

.

h2(10k) =01

h2(20k) =11

h2(30k) =01

h2(40k) =00

.

000	<Fred>
001	<Joe><Jan>
010	<Mary>
011	
100	<Sally>
101	
110	<Tom><Bill>
111	<Andy>

Find Emp. with Dept. = Sales $\wedge$ Sal=40k

EX:

h1(toy) =0
h1(sales) =1
h1(art) =1
.
h2(10k) =01
h2(20k) =11
h2(30k) =01
h2(40k) =00
.

000	<Fred>
001	<Joe><Jan>
010	<Mary>
011	
100	<Sally>
101	
110	<Tom><Bill>
111	<Andy>

Find Emp. with Dept. = Sales $\wedge$ Sal=40k

EX:

h1(toy) =0
h1(sales) =1
h1(art) =1

.

h2(10k) =01
h2(20k) =11
h2(30k) =01
h2(40k) =00

.

000	<Fred>
001	<Joe><Jan>
010	<Mary>
011	
100	<Sally>
101	
110	<Tom><Bill>
111	<Andy>

Find Emp. with Sal=30k

EX:

h1(toy) =0
h1(sales) =1
h1(art) =1
.
h2(10k) =01
h2(20k) =11
h2(30k) =01
h2(40k) =00
.

000	<Fred>
001	<Joe><Jan>
010	<Mary>
011	
100	<Sally>
101	
110	<Tom><Bill>
111	<Andy>

Find Emp. with Sal=30k

EX:

h1(toy) =0
h1(sales) =1
h1(art) =1
.
h2(10k) =01
h2(20k) =11
h2(30k) =01
h2(40k) =00
.

000	<Fred>
001	<Joe><Jan>
010	<Mary>
011	
100	<Sally>
101	
110	<Tom><Bill>
111	<Andy>

Find Emp. with Dept. = Sales

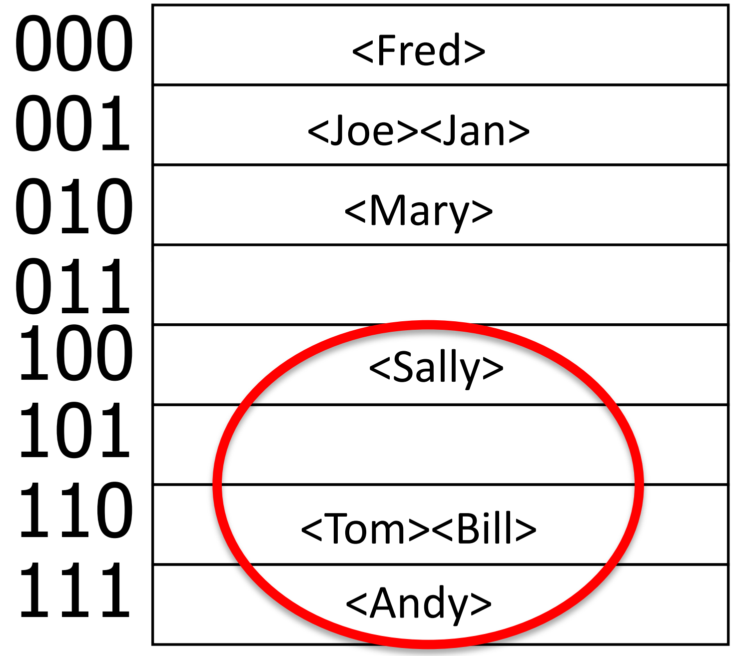
EX:

h1(toy) =0
h1(sales) =1
h1(art) =1

.

h2(10k) =01
h2(20k) =11
h2(30k) =01
h2(40k) =00

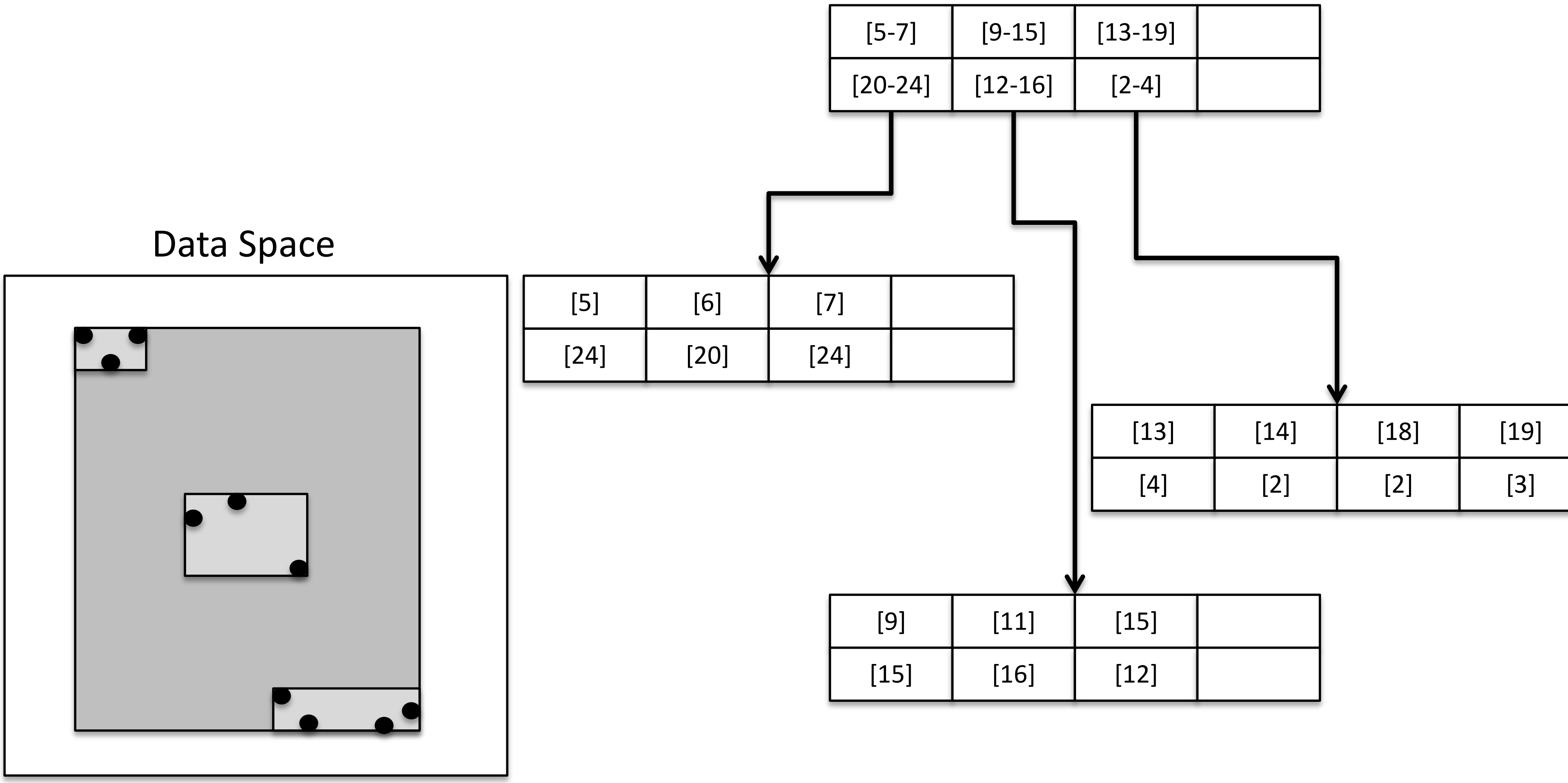
.



Find Emp. with Dept. = Sales

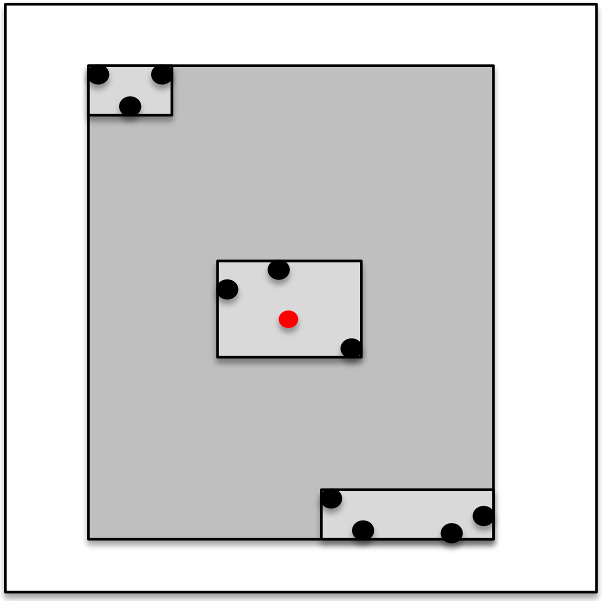
R-tree

- Nodes can store up to **M** entries
  - Minimum fill requirement (depends on variant)
- Each node rectangle in **n**-dimensional space
  - Minimum Bounding Rectangle (MBR) of its children
- MBRs of siblings are allowed to overlap
  - Different from B-trees
- balanced


Data Space
[5-7] [9-15] [13-19]
[20-24] [12-16] [2-4]
[5] [6] [7]
[24] [20] [24]
[13] [14] [18] [19]
[4] [2] [2] [3]
[9] [11] [15]
[15] [16] [12]

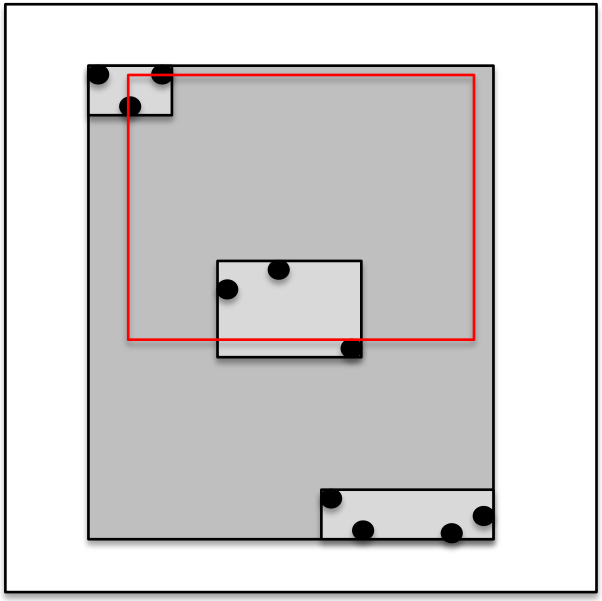
R-tree - Search

- Point Search
  - Search for p = <x_i, y_i>
  - Keep list of potential nodes
    - Needed because of overlap
  - Traverse to child if MBR of child contains p

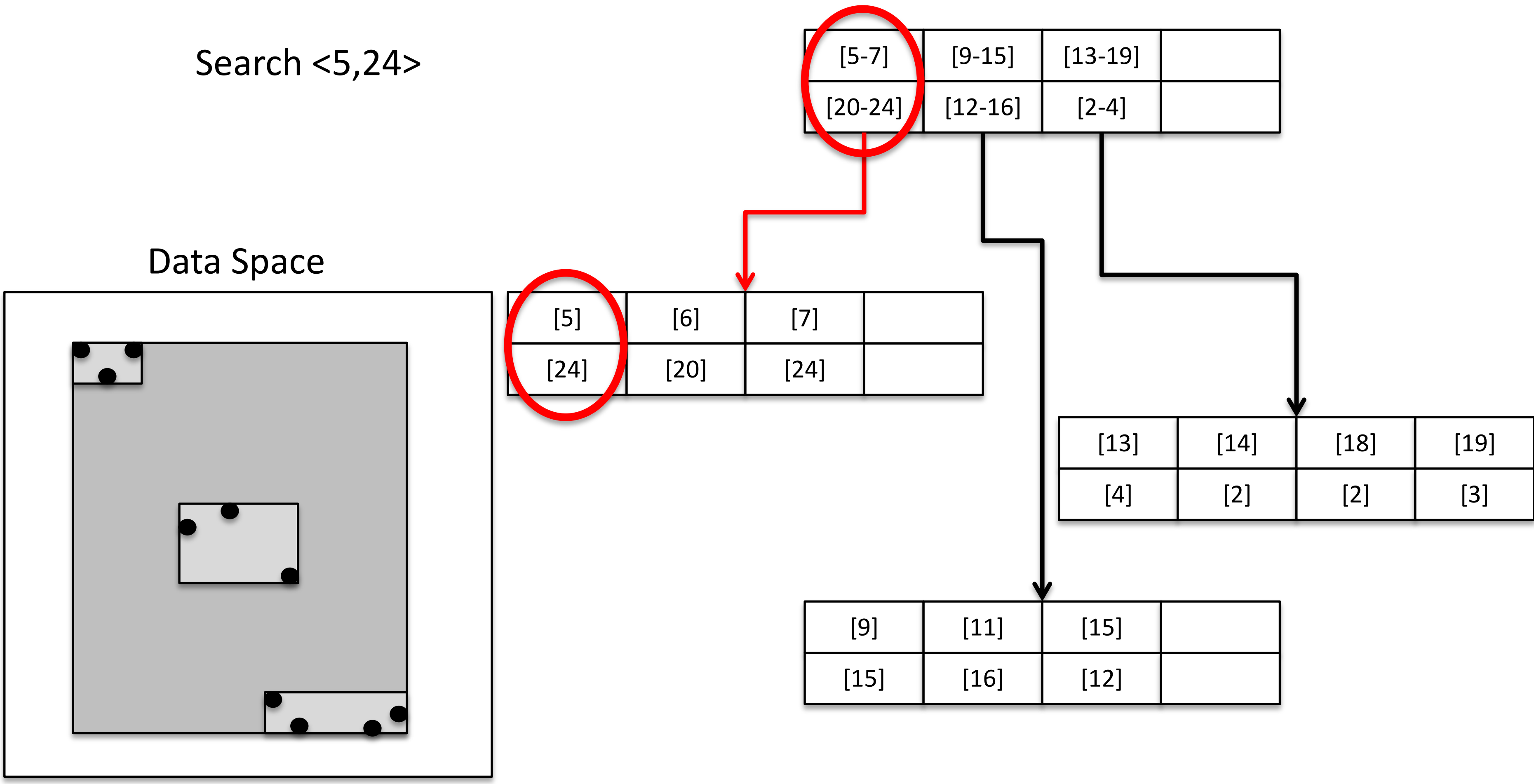


R-tree - Search

- Point Search
  - Search for points in region = <$[x_{min} - x_{max}], [y_{min} - y_{max}]$>
  - Keep list of potential nodes
  - Traverse to child if MBR of child overlaps with query region



Search <5,24>


Data Space
[5-7]
[9-15]
[13-19]
[20-24]
[12-16]
[2-4]
[5]
[6]
[7]
[24]
[20]
[24]
[13]
[14]
[18]
[19]
[4]
[2]
[2]
[3]
[9]
[11]
[15]
[15]
[16]
[12]

R-tree - Insert

- Similar to B-tree, but more complex
  - Overlap -> multiple choices where to add entry
  - Split harder because more choice how to split node (compare B-tree = 1 choice)
- 1) Find potential subtrees for current node
  - Choose one for insert (heuristic, e.g., the one the would grow the least)
  - Continue until leaf is found

R-tree - Insert

- 2) Insert into leaf
- 3) Leaf is full? -> split
  - Find best split (minimum overlap between new nodes) is hard ($O(2^M)$)
  - Use linear or quadratic heuristics (original paper)
- 4) Adapt parents if necessary

R-tree - Delete

- 1) Find leaf node that contains entry
- 2) Delete entry
- 3) Leaf node underflow?
  - Remove leaf node and cache entries
  - Adapt parents
  - Reinsert deleted entries

Bitmap Index

- Domain of values $D = \{d_1, \ldots, d_n\}$
  - Gender {male, female}
  - Age {1, …, 120?}
- Use one vector of bits for each value
  - One bit for each record
    - 0: record has different value in this attribute
    - 1: record has this value

Bitmap Index Example

Age

1	2	3
1	0	0
0	1	0
1	0	0
0	0	1

Todlers

Name	Age	Gender
Peter	1	male
Gertrud	2	female
Joe	1	male
Marry	3	female

Gender

male	female
1	0
0	1
1	0
0	1

Bitmap Index Example

Age

1	2	3
1	0	0
0	1	0
1	0	0
0	0	1

Todlers

Name	Age	Gender
Peter	1	male
Gertrud	2	female
Joe	1	male
Marry	3	female

Gender

male	female
1	0
0	1
1	0
0	1

Find all todlers with age **2 and** sex **female**:

Bitwise-and between vectors

0
1
0
0

Bitmap Index Example

Age

1	2	3
1	0	0
0	1	0
1	0	0
0	0	1

Todlers

Name	Age	Gender
Peter	1	male
Gertrud	2	female
Joe	1	male
Marry	3	female

Gender

male	female
1	0
0	1
1	0
0	1

0
1
0
1

Find all todlers with age **2 or** sex **female**:

Bitwise-or between vectors

Compression

- Observation:
  - Each record has one value in indexed attribute
  - For N records and domain of size |D|
    - Only 1/|D| bits are 1
  - -> waste of space
- Solution
  - Compress data
  - Need to make sure that **and** and **or** is still fast

Run length encoding (RLE)

- Instead of actual 0-1 sequence encode length of 0 or 1 runs
- One bit to indicate whether 0/1 run + several bits to encode run length
- But how many bits to use to encode a run length?
  - Gamma codes or similar to have variable number of bits

RLE Example

- 0001 0000 1110 1111 **(2 bytes)**
- 3, 1,4, 3, 1,4 **(6 bytes)**
- -> if we use one byte to encode a run we have 7 bits for length = max run length is 128(127)

Elias Gamma Codes

- $X = 2^N + (x \bmod 2^N)$
  - Write N as N zeros followed by one 1
  - Write ($x \bmod 2^N$) as N bit number
- $18 = 2^4 + 2 = 000010010$

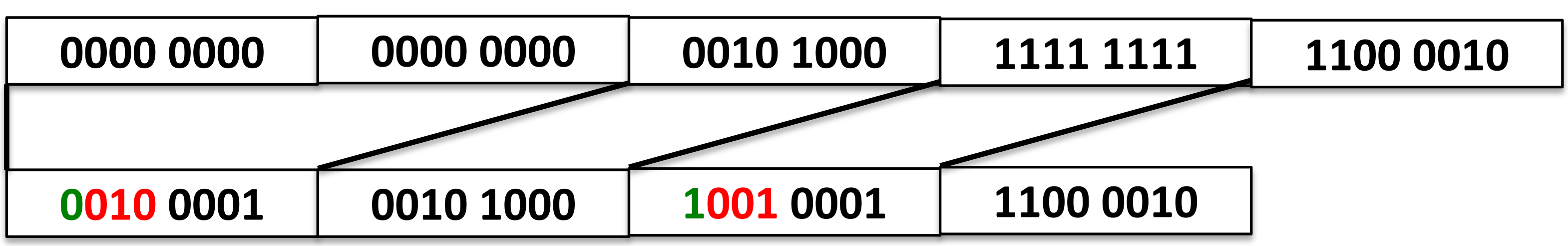
- 0001 0000 1110 1111 **(2 bytes)**
- 3, 1,4, 3, 1,4 **(6 bytes)**
- 0111 0010 0011 1001 00 **(3 bytes)**

Hybrid Encoding

- Run length encoding
  - Can waste space
  - And/or run length not aligned to byte/word boundaries
- Encode some bytes of sequence as is and only store long runs as run length
  - EWAH
  - BBC (that's what Oracle uses)

Extended Word aligned Hybrid (EWAH)

- Segment sequence in machine words (64bit)
- Use two types of words to encode
  - Literal words, taken directly from input sequence
  - Run words
    - ½ word is used to encode a run
    - ½ word is used to encode how many literals follow



Bitmap Indices

- Fast for read intensive workloads
  - Used a lot in datawarehousing
- Often build on the fly during query processing
  - As we will see later in class

Trie

- From Retrieval
- Tree index structure
- Keys are sequences of values from a domain D
  - D = {0,1}
  - D = {a,b,c,....,z}
- Key size may or may not be fixed
  - Store 4-byte integers using D = {0,1} (32 elements)
  - Strings using D={a,...,z} (arbitrary length)

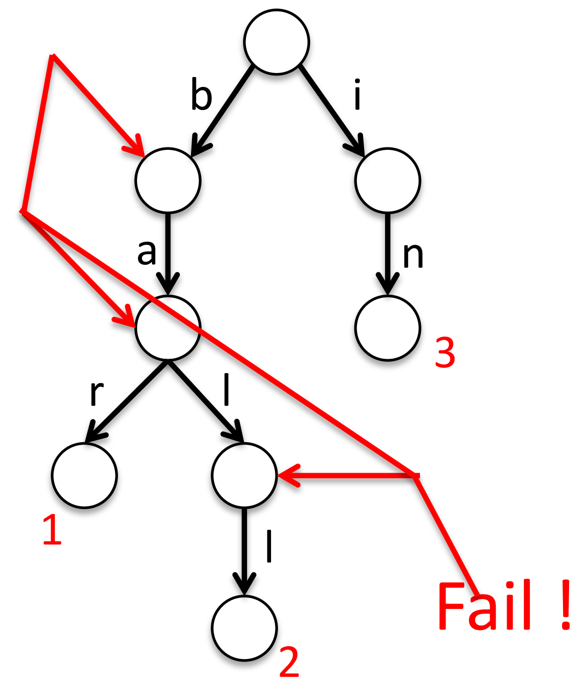
Trie

- Each node has pointers to |D| child nodes
  - One for each value of D
- Searching for a key $k = [d_1, \ldots, d_n]$
  - Start at the root
  - Follow child for value d_i

Trie Example

Words: bar, ball, in

Search for **bald**



Tries Implementation

- 1) Each node has an array of child pointers
- 2) Each node has a list or hash table of child pointers
- 3) array compression schemes derived from compressed DFA representations

Index structures in the Main Memory DBMS era

- Larger and large portions of the data fit into main memory
  - Disk I/O no longer the (only) bottleneck
  - Highly optimized and specialized operator code
    - Difference of the constant factor for full scan versus index increase
  - Increasing amounts of parallelism
    - Traditional methods for parallel access to indexes no longer effective enough
- => Do not use indexes anymore?

Index structures in the Main Memory DBMS era

- Solutions
  - More Light-weight and coarse-grained data structures
  - Use data-structures that have less parallelization bottle-necks

Index structures in the Main Memory DBMS era

- **Solutions**
  - More Light-weight and coarse-grained data structures, e.g.:
    - Data skipping (small materialized aggregates)
    - Database cracking
  - Use data-structures that have less parallelization bottle-necks, e.g.,
    - Skip lists
    - B^w-trees

Data skipping

- Consider a relation stored in an unsorted page file
  - Regular DBMS
  - HDFS parquet file
  - …
- Main idea of data skipping
  - For each page store min/max values of each attribute
- To evaluate a selection predicate on attribute A say c1 <= A <= c2
  - if for page P: A_{max} < c1 or A_{min} > c2 then none of the tuples on that page will qualify and we can skip reading this page

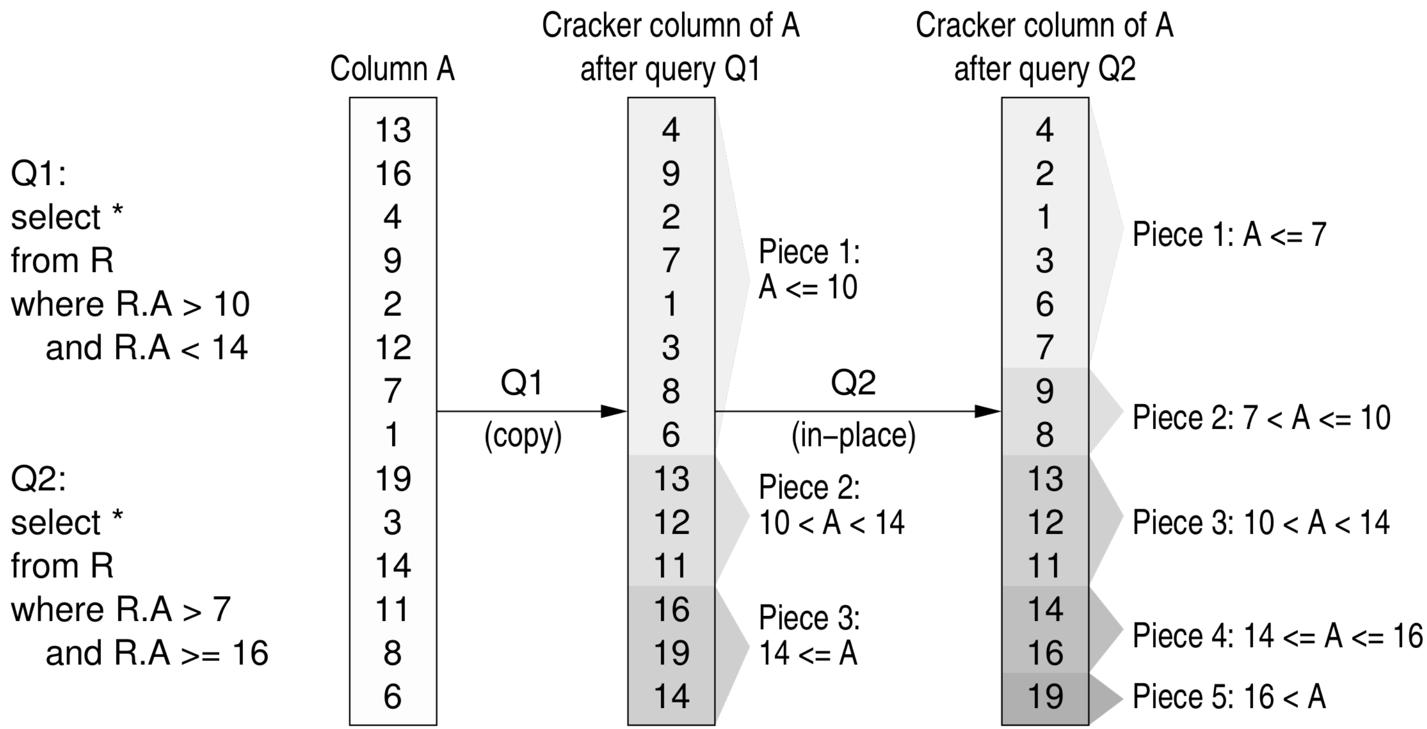
R

A	B	C
a	1	10
b	5	20
c	2	10
d	2	35
e	3	45
f	4	40

Database cracking

- Main rationale
  - Originally designed for columnar databases
  - The amount of indexing effort we spend for a part of the key space should be based on how frequently this part of the keyspace is accessed
- Basic idea
  - Start with an unsorted file
  - Whenever a query applies a selection condition on a column A, say A< 50, then split the current partition containing 50 into two fragments one with data < 50 and one with the remaining data (partial sort)
  - Keep a small in-memory tree index for these fragments

Database cracking

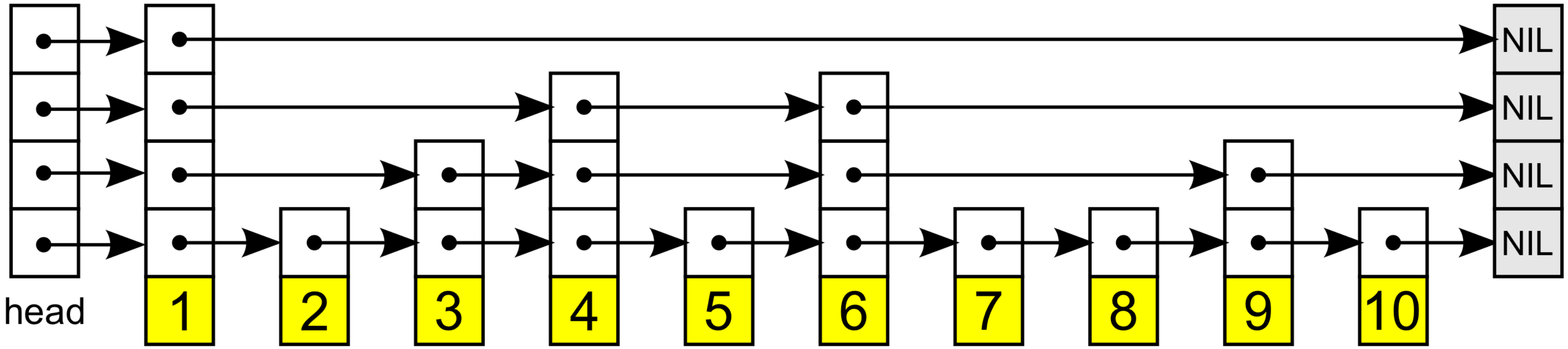


From **Database Cracking – CIDR 2007**

Skip lists

- Probabilistic datastructure
  - Behavior depends on randomization
  - Gives only probabilistic guarantees
    - => with high probability will guarantee good performance
  - Approximates a search tree using the much simpler (and easier to parallelize linked list datastructure)

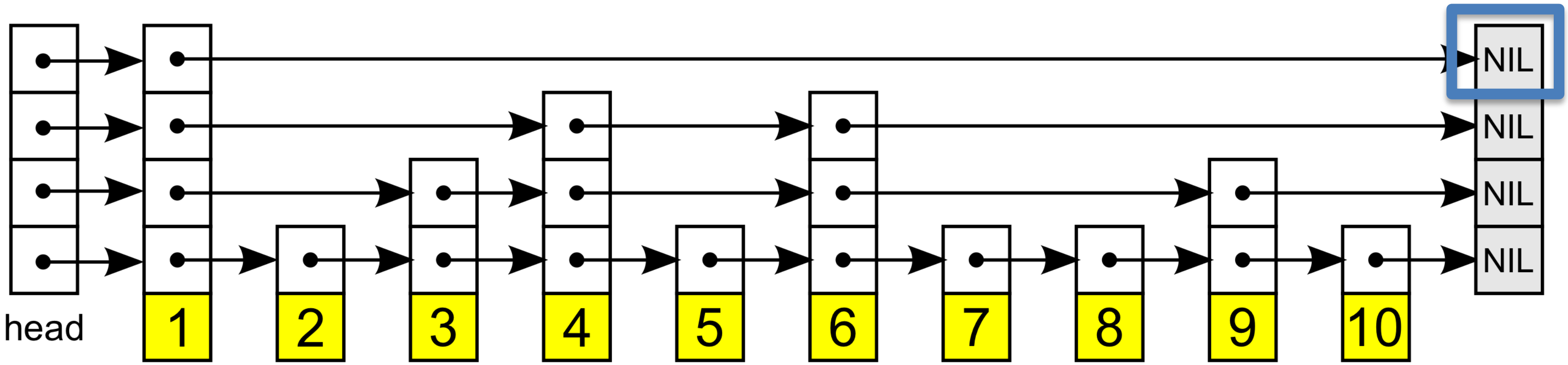
Skip lists



- **Search**:
  - Start from the top list
  - 1) Move through list until element is found or we are at a larger element/end of the list
  - 2) move to previous element (smaller than search key) and follow a down pointer to the next deeper level
  - 3) Goto 1)

Search 5

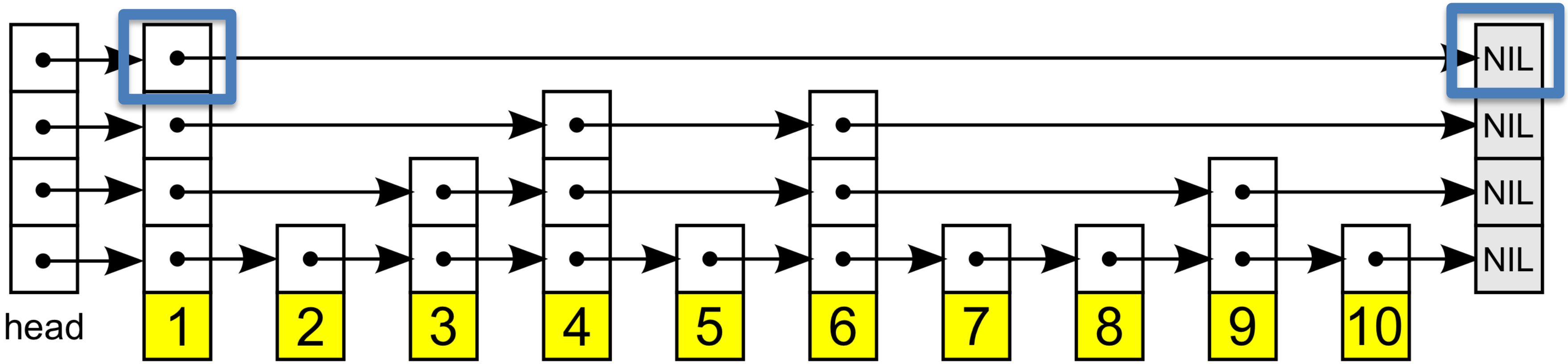
Skip lists



- **Search**:
  - Start from the top list
  - 1) Move through list until element is found or we are at a larger element/end of the list
  - 2) move to previous element (smaller than search key) and follow a down pointer to the next deeper level
  - 3) Goto 1)

Search 5

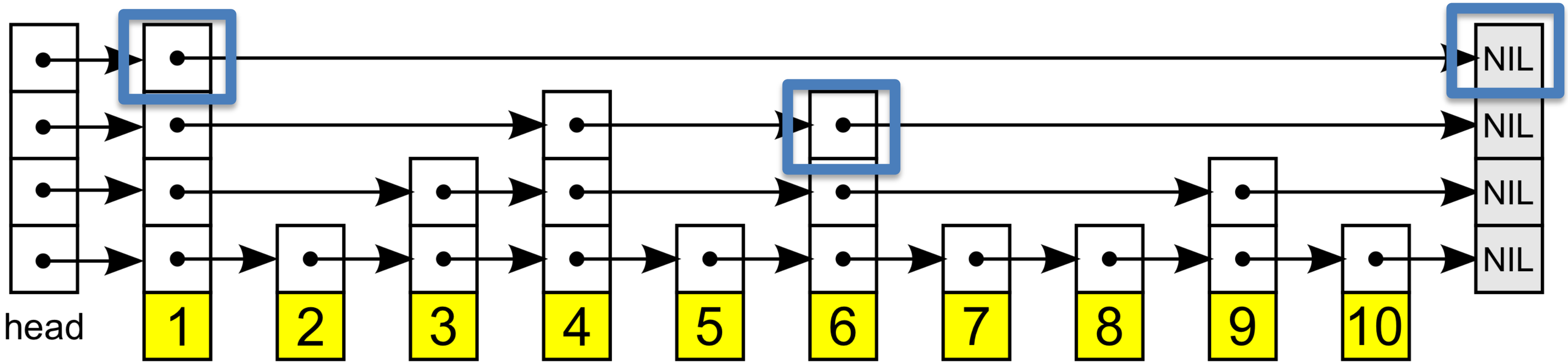
Skip lists



- **Search**:
  - Start from the top list
  - 1) Move through list until element is found or we are at a larger element/end of the list
  - 2) move to previous element (smaller than search key) and follow a down pointer to the next deeper level
  - 3) Goto 1)

Search 5

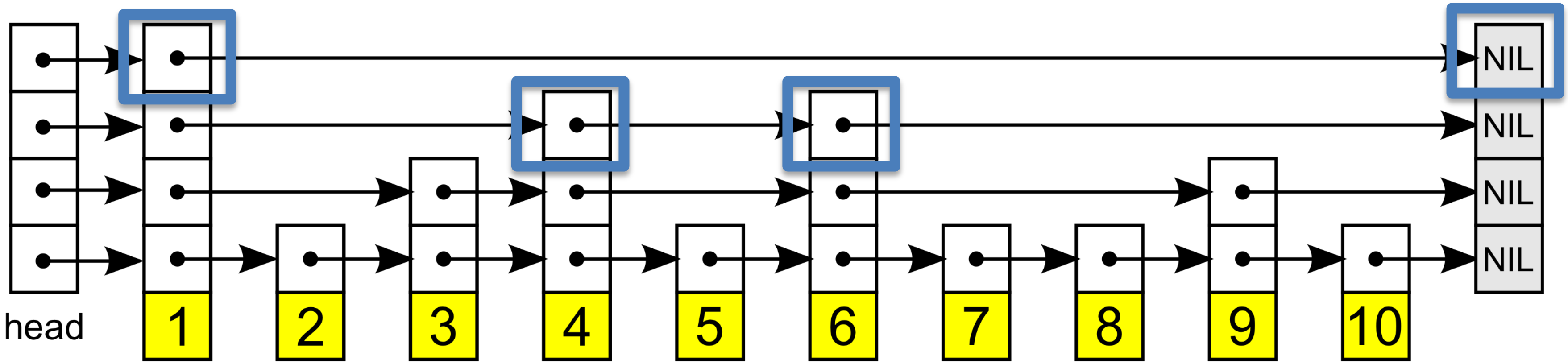
Skip lists



- **Search**:
  - Start from the top list
  - 1) Move through list until element is found or we are at a larger element/end of the list
  - 2) move to previous element (smaller than search key) and follow a down pointer to the next deeper level
  - 3) Goto 1)

Search 5

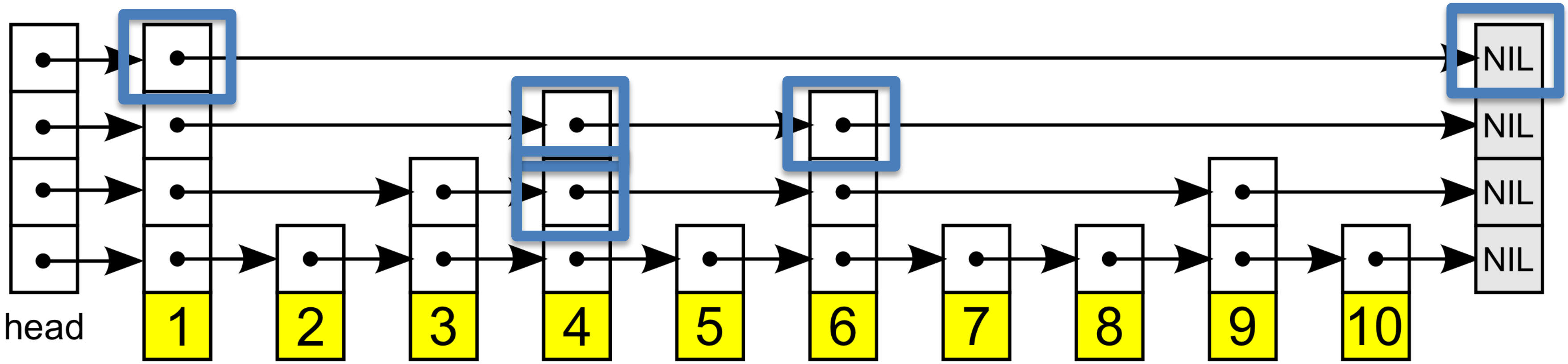
Skip lists



- **Search**:
  - Start from the top list
  - 1) Move through list until element is found or we are at a larger element/end of the list
  - 2) move to previous element (smaller than search key) and follow a down pointer to the next deeper level
  - 3) Goto 1)

Search 5

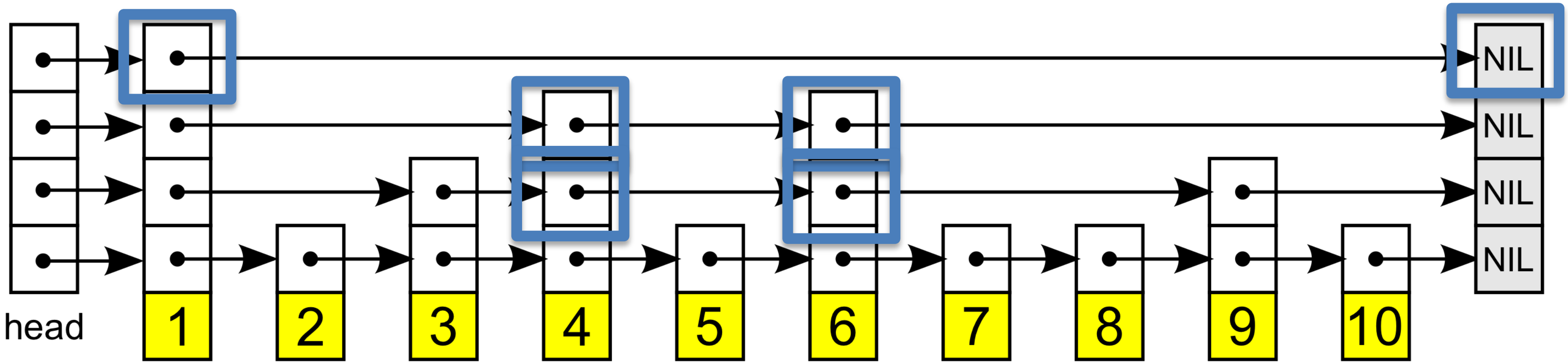
Skip lists



- **Search**:
  - Start from the top list
  - 1) Move through list until element is found or we are at a larger element/end of the list
  - 2) move to previous element (smaller than search key) and follow a down pointer to the next deeper level
  - 3) Goto 1)

Search 5

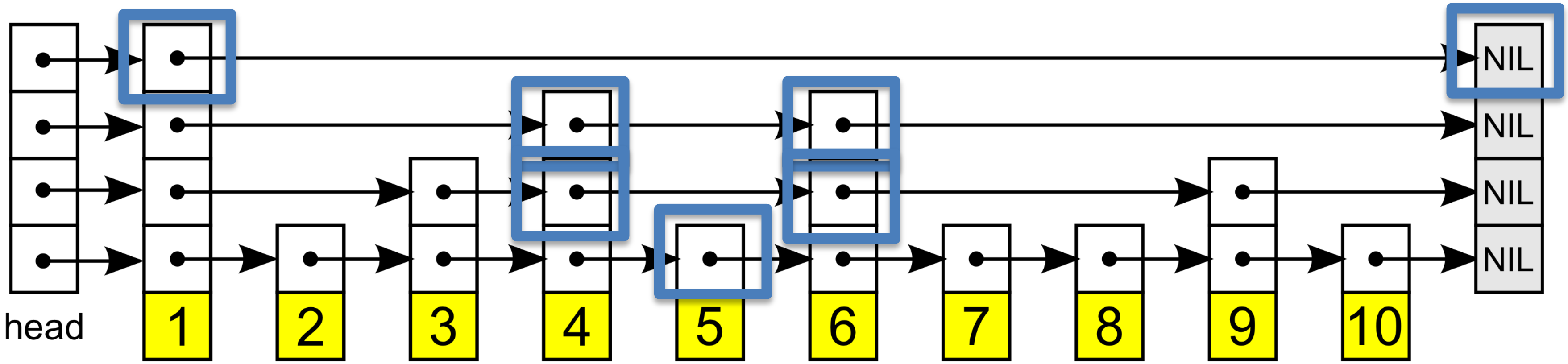
Skip lists



- **Search**:
  - Start from the top list
  - 1) Move through list until element is found or we are at a larger element/end of the list
  - 2) move to previous element (smaller than search key) and follow a down pointer to the next deeper level
  - 3) Goto 1)

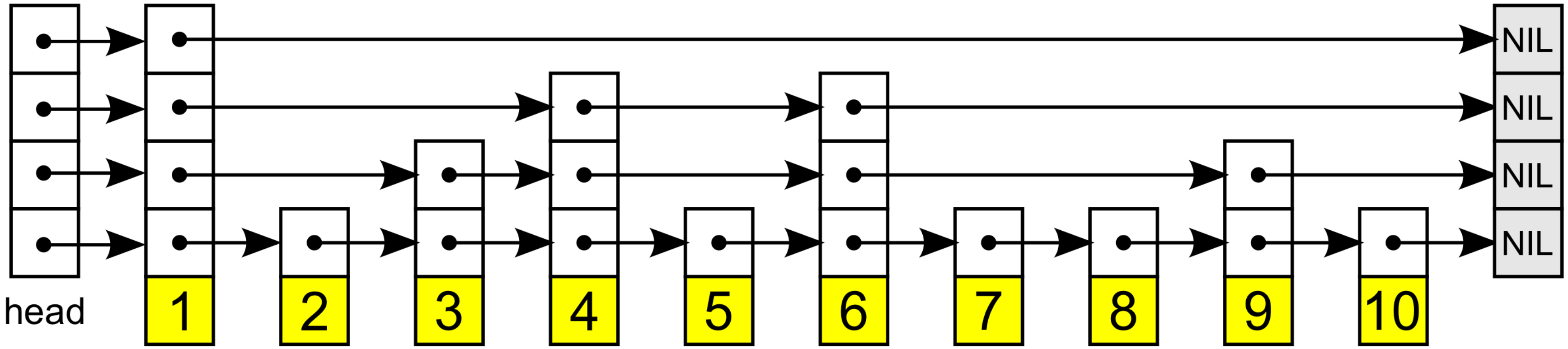
Search 5

Skip lists



- **Search**:
  - Start from the top list
  - 1) Move through list until element is found or we are at a larger element/end of the list
  - 2) move to previous element (smaller than search key) and follow a down pointer to the next deeper level
  - 3) Goto 1)

Skip lists



- **Insert**:
  - Use search to find insertion position at the lowest level (keep pointers at the higher levels)
  - Insert element in the lowest list
  - Then for every level throw a dice and insert key with probability p (typically ½)

Observation: in expectation each level has p as many nodes as the next lower level

Skip lists

- **Characteristics**
  - O(log(n)) expected performance (insert, delete, search)
  - Easy to parallelize (linked lists)
  - Simpler to implement (also less CPU ops) than B-trees
- **Example implementations**
  - MemSQL (main memory database system)
  - Lucene
  - leveldb

Improving insert/update performance

- B-tree
  - O(log(n)) I/O
- Hash-index
  - O(1) I/O, but potential reorg cost
- Consider Key-value store (e.g., Cassandra) application
  - Need fast write-throughput
  - Need fast point-lookup

One Solution: LSM-trees

- **Log-structured merge** (**LSM**) trees
  - Have small index that is memory resident (**memtable**)
  - When memtable exceeds a size threshold write it as one sorted run to disk (will explain algorithm when talking about query execution)
    - Sequential I/O!
    - Runs are immutable after being written (exception compaction)
    - Runs may contain outdated values for keys that exist in newer runs of the memtable
    - Over time me we have multiple sorted runs
  - **Inserts/Updates**
    - Always applied to memtable
  - **Lookup**
    - If we find a key in the memtable then return it
    - Otherwise lookup keys in the sorted runs in reverse chronological order

LSM-trees

- **Performance**
  - **Inserts/Updates**
    - O(1)!
  - **Lookup**
    - O(#runs)
    - => want to make sure the number of runs does not grow indefinitely
- **Compaction**
  - Merge sorted runs on disks to reduce #runs => improve lookup performance

Basic Compaction

- Have levels
  - Once there are more then x runs on a level these are merged into one larger run
  - Run sizes increase exponentially per level
- E.g., threshold is 4 runs
  - first level: runs are of same size as memtable
  - 2^{nd} level: 4 * size of memtable
  - 3^{rd} level: 4 * 4 * size of memtable
  - …

LSM-trees

- **Other lookup improvements**
  - Block index in memory (similar to sparse index)
  - Bloomfilters
    - A bloom filter is a small over-approximation of set
      - Can be used to test if a key K is contained in a set S
        - » Returns yes, then the key **may** be in the set
        - » Returns no, then the key is guaranteed to not be in the set
    - => fast way to avoid looking a runs that are guaranteed to not contain a key

Bw-trees

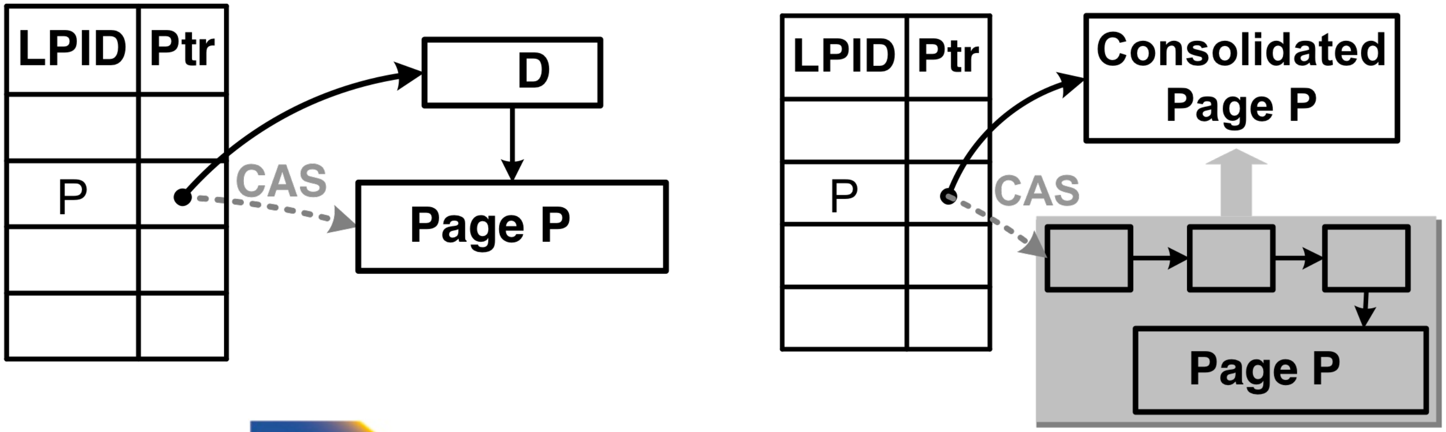
- **Motivation**
  - Improve concurrency properties of B-trees
  - Improve cache effectiveness of B-trees
  - Designed for flash-storage
    - Fast random/sequential reads
    - Fast sequential writes
    - Comparably slower random writes (albeit smaller factor

Bw-trees

- **Overview**
  - Updateable B-tree without latches
    - Threads almost never block
      - => Improved instruction cache performance
  - Backed up by log-structured storage
  - Updates never modify pages but append deltas to a page
    - Deltas are “installed” using CAS (atomic compare and swap)

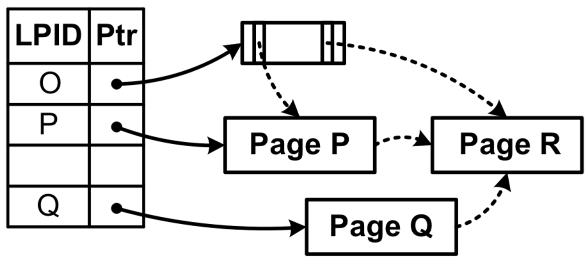
Bw-trees

- **Mapping table**
  - Pages are logical identified by a LPID which is stable
  - Locations and size of pages can change over time
  - Updates create a delta record that points to the previous address of the page
  - The delta record's address is swapped for the current address in the mapping table using CAS

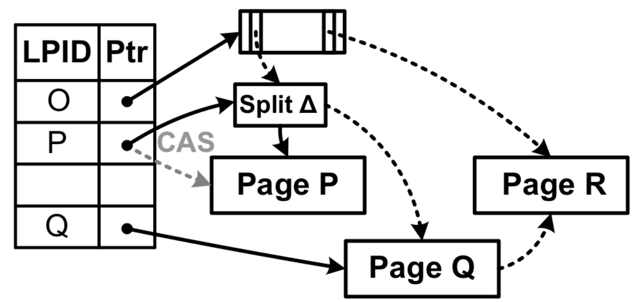


Bw-trees

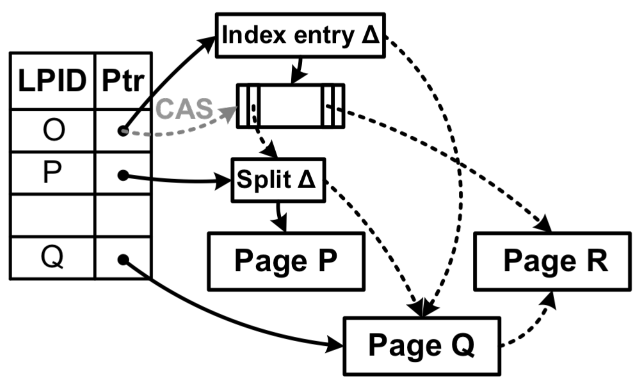
- **Making page splits atomic**



(a) Creating sibling page Q



(b) Installing split delta



(c) Installing index entry delta

Summary

Discussion:

- Conventional Indices
- B-trees
- Hashing (extensible, linear)
- SQL Index Definition
- Index vs. Hash
- Multiple Key Access
- Multi Dimensional Indices
  Variations: Grid, R-tree,
- Partitioned Hash
- Bitmap indices and compression
- Tries
- Database cracking
- Data skipping (small materialized aggregates/zone maps)
- Skip-lists
- Log-structured merge trees (LSM)