# CS232: Database System Principles

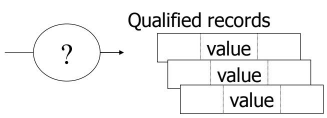
## INDEXING

1

## Indexing

<u>Given condition on attribute find qualified records</u>

Attr = value




Condition may also be

- Attr>value
- Attr>=value

2

## Indexing

- Data Stuctures used for quickly locating tuples that meet a specific type of condition
  - *Equality* condition: find Movie tuples where Director=*X*
  - Other conditions possible, eg, *range* conditions: find Employee tuples where Salary>40 AND Salary<50
- Many types of indexes. Evaluate them on
  - *Access* time
  - *Insertion* time
  - *Deletion* time
  - Disk *Space* needed (esp. as it effects access time)

## Topics

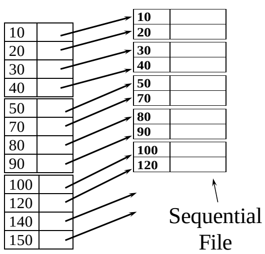
- Conventional indexes
- B-trees
- Hashing schemes

4

# Terms and Distinctions

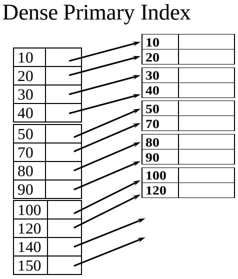
- *Primary* index
  - the index on the attribute (a.k.a. search key) that determines the sequencing of the table
- *Secondary* index
  - index on any other attribute
- *Dense* index
  - every value of the indexed attribute appears in the index
- *Sparse* index
  - many values do not appear

A Dense Primary Index

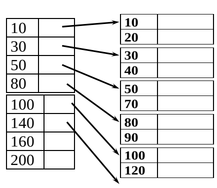



# Dense and Sparse Primary Indexes

Dense Primary Index




Sparse Primary Index




Find the index record with largest value that is less or equal to the value we are looking.

**+** can tell if a value exists without accessing file (consider projection)
+ better access to overflow records

**+** less index space

*more + and - in a while*

## Sparse vs. Dense Tradeoff

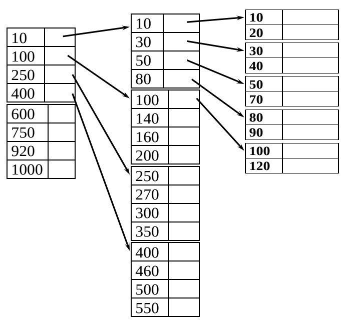
- Sparse: Less index space per record
  can keep more of index in memory
- Dense: Can tell if any record exists
  without accessing file

(Later:
- sparse better for insertions
- dense needed for secondary indexes)

7

## Multi-Level Indexes

- Treat the index as a file and build an index on it
- "Two levels are usually sufficient. More than three levels are rare."
- Q: Can we build a dense second level index for a dense index ?

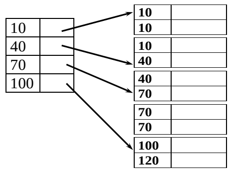



## A Note on Pointers

- *Record pointers* consist of *block pointer* and position of record in the block
- Using the block pointer only, saves space at no extra accesses cost
- But a block pointer cannot serve as record identifier

## Representation of Duplicate Values in Primary Indexes

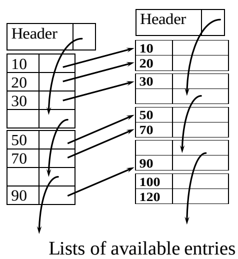
- Index may point to first instance of each value only




## Deletion from Dense Index

- Deletion from dense primary index file with no duplicate values is handled in the same way with deletion from a sequential file
- Q: What about deletion from dense primary index with duplicates

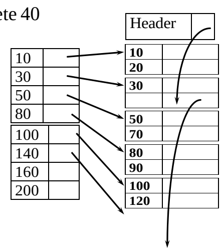
Delete 40, 80




## Deletion from Sparse Index

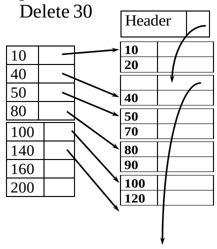
- if the deleted entry does not appear in the index do nothing

Delete 40



## Deletion from Sparse Index (cont'd)

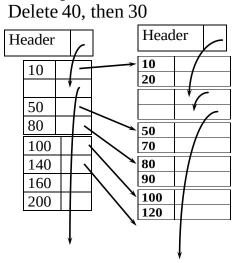
Delete 30




- if the deleted entry does not appear in the index do nothing
- if the deleted entry appears in the index replace it with the next search-key value
  - comment: we could leave the deleted value in the index assuming that no part of the system may assume it still exists without checking the block

## Deletion from Sparse Index (cont'd)

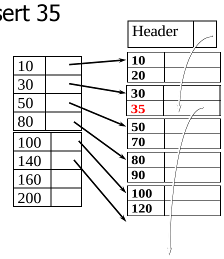
Delete 40, then 30




- if the deleted entry does not appear in the index do nothing
- if the deleted entry appears in the index replace it with the next search-key value
- unless the next search key value has its own index entry. In this case delete the entry

## Insertion in Sparse Index

Insert 35

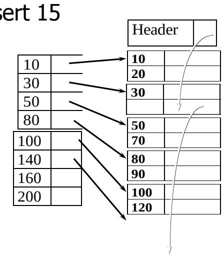



- if no new block is created then do nothing

# Insertion in Sparse Index

- if no new block is created then do nothing
- else create overflow record
  - Reorganize periodically
  - Could we claim space of next block?
  - How often do we reorganize and how much expensive it is?
  - B-trees offer convincing answers

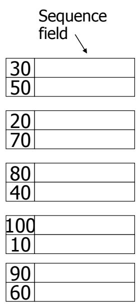
Insert 15




16

## Secondary indexes

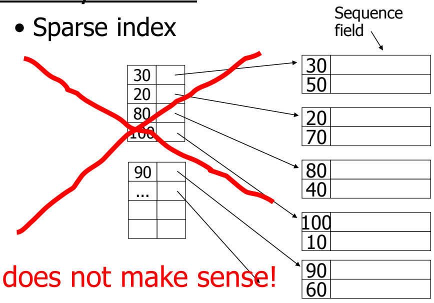
File not sorted on secondary search key




17

## Secondary indexes

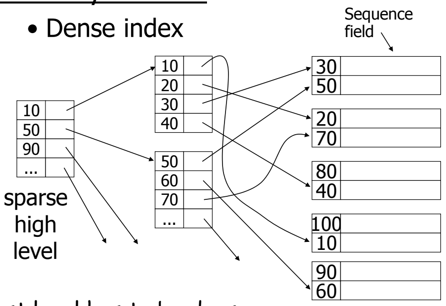
- Sparse index




18

## Secondary indexes

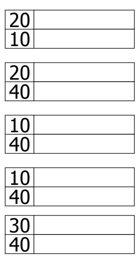
- Dense index




First level has to be dense,
next levels are sparse (as usual)

19

## Duplicate values & secondary indexes

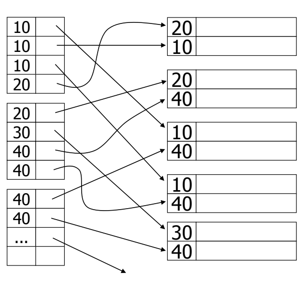



20

## Duplicate values & secondary indexes

one option...

Problem:
excess overhead!
- disk space
- search time

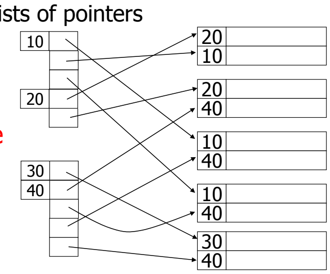



21

## Duplicate values & secondary indexes

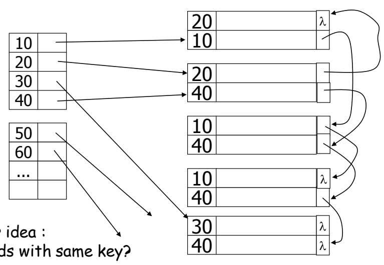
another option: lists of pointers

Problem: variable size records in index!




22

## Duplicate values & secondary indexes

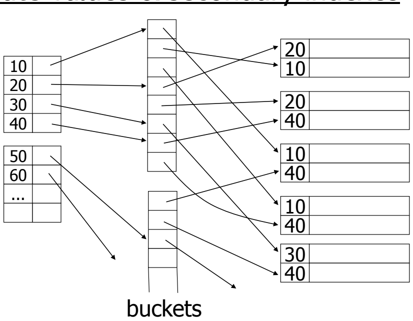



Yet another idea :
Chain records with same key?

Problems:
- Need to add fields to records, messes up maintenance
- Need to follow chain to know records

23

## Duplicate values & secondary indexes




24

## Why "bucket" + record pointers is useful

- Enables the processing of queries working with pointers only.
- Very common technique in Information Retrieval

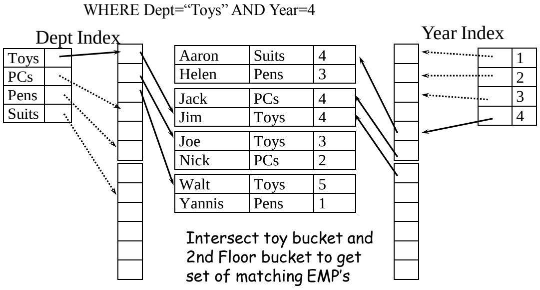
| Indexes | Records |
|---|---|
| Name: primary | EMP (name,dept,year,...) |
| Dept: secondary | |
| Year: secondary | |

25

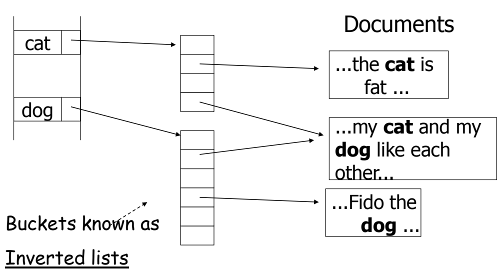
## Advantage of Buckets: Process Queries Using Pointers Only

Find employees of the Toys dept with 4 years in the company

SELECT Name FROM Employee
WHERE Dept="Toys" AND Year=4




## This idea used in text information retrieval



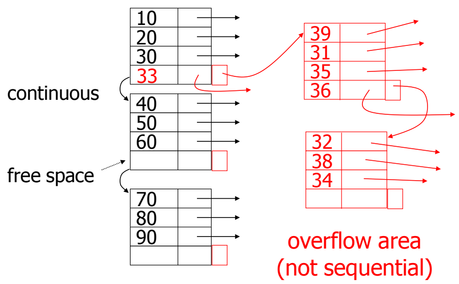

27

## Summary of Indexing So Far

- Basic topics in conventional indexes
  - multiple levels
  - sparse/dense
  - duplicate keys and buckets
  - deletion/insertion similar to sequential files
- Advantages
  - simple algorithms
  - index is sequential file
- Disadvantages
  - eventually sequentiality is lost because of overflows, reorganizations are needed



<u>Example</u> Index (sequential)



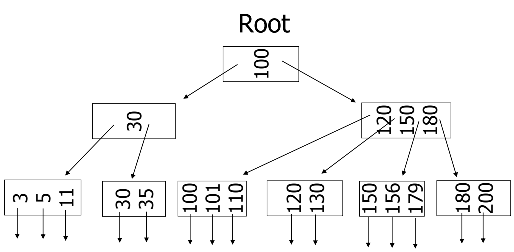

## Outline:

- Conventional indexes
- B-Trees $\Rightarrow$ NEXT
- Hashing schemes

- NEXT: Another type of index
  - Give up on sequentiality of index
  - Try to get "balance"

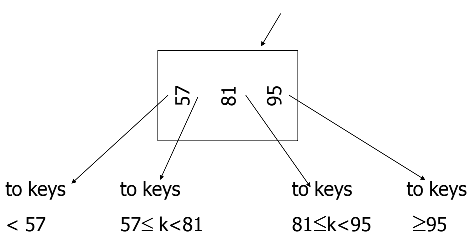
31

## B+Tree Example n=3



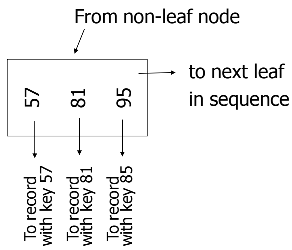

32

## Sample non-leaf




33

## Sample leaf node:

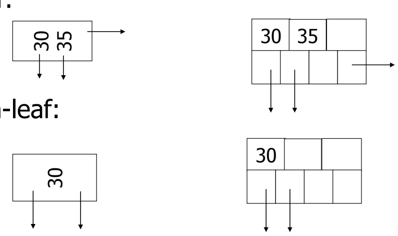





## In textbook's notation n=3

Leaf:

Non-leaf:






**Size of nodes:**

n+1 pointers
n keys (fixed)

## Non-root nodes have to be at least half-full

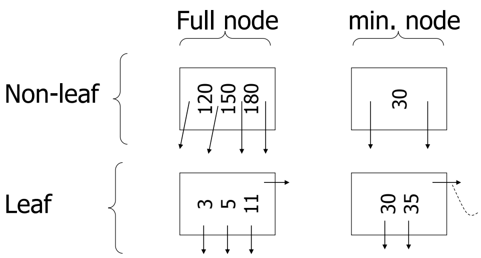
- Use at least

Non-leaf: $\lceil (n+1)/2 \rceil$ pointers

Leaf: $\lfloor (n+1)/2 \rfloor$ pointers to data

37

n=3




38

## B+tree rules    tree of order *n*

(1) All leaves at same lowest level
(balanced tree)

(2) Pointers in leaves point to records
except for "sequence pointer"

39

(3) Number of pointers/keys for B+tree

| | Max ptrs | Max keys | Min ptrs→data | Min keys |
|---|---|---|---|---|
| Non-leaf (non-root) | n+1 | n | $\lceil (n+1)/2 \rceil$ | $\lceil (n+1)/2 \rceil - 1$ |
| Leaf (non-root) | n+1 | n | $\lfloor (n+1)/2 \rfloor$ | $\lfloor (n+1)/2 \rfloor$ |
| Root | n+1 | n | 1 | 1 |



## Insert into B+tree

(a) simple case
- space available in leaf

(b) leaf overflow

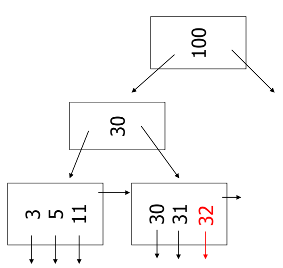
(c) non-leaf overflow

(d) new root



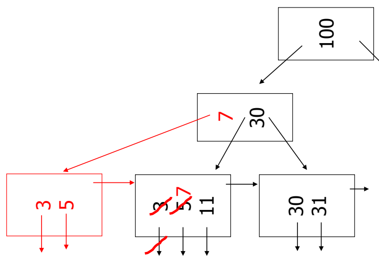
(a) Insert key = 32

n=3



(a) Insert key = 7

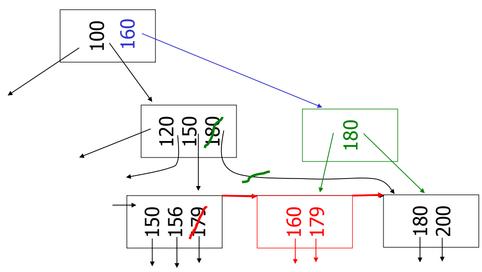
n=3





(c) Insert key = 160

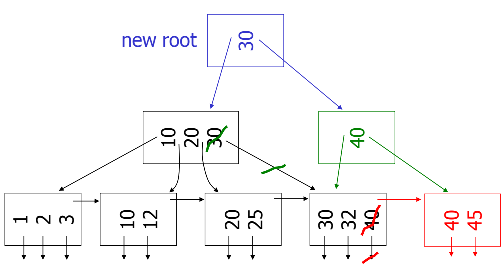
n=3





(d) New root, insert 45

n=3



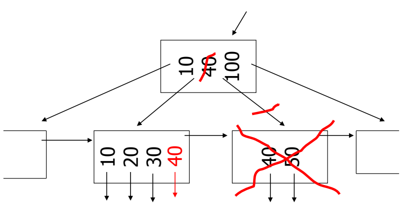
## Deletion from B+tree

(a) Simple case - no example
(b) Coalesce with neighbor (sibling)
(c) Re-distribute keys
(d) Cases (b) or (c) at non-leaf

46

(b) Coalesce with sibling
– Delete 50

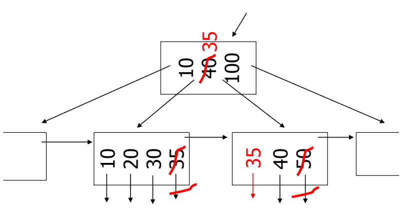
n=4




47

(c) Redistribute keys
– Delete 50

n=4

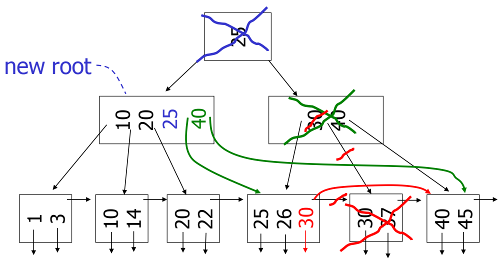



48

(d) Non-leaf coalese

– Delete 37

n=4






## B+tree deletions in practice

– Often, coalescing is not implemented
  – Too hard and not worth it!



Is LRU a good policy for B+tree buffers?

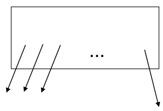
→ Of course not!

→ Should try to keep root in memory at all times

(and perhaps some nodes from second level)

Hardware+ indexing problem:

For B+tree, how large should *n* be?




*n* is number of keys / node

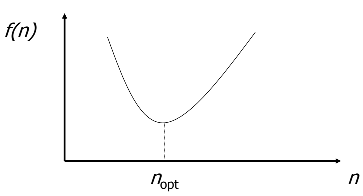


# Assumptions

- You have the right to set the block size for the disk where a B-tree will reside.
- Compute the optimum page size *n* assuming that
  - The items are 4 bytes long and the pointers are also 4 bytes long.
  - Time to read a node from disk is *12+.003n*
  - Time to process a block in memory is unimportant
  - B+tree is full (I.e., every page has the maximum number of items and pointers

⇘Can get:

$f(n)$ = time to find a record



- FIND $n_{opt}$ by $f'(n) = 0$

  Answer should be $n_{opt}$ = "few hundred"

- What happens to $n_{opt}$ as
  - Disk gets faster?
  - CPU get faster?

55

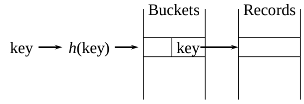
Outline/summary

- Conventional Indexes
  - Sparse vs. dense
  - Primary vs. secondary
- B+ trees
- Hashing schemes --> Next
- Bitmap indices

56

## Hashing

- hash function $h$(key) returns address of bucket
- if the keys for a specific hash value do not fit into one page the bucket is a linked list of pages



## Example hash function

- Key = '$x_1\ x_2 \dots x_n$' *n* byte character string
- Have *b* buckets
- h: add $x_1 + x_2 + \dots\dots x_n$
  - compute sum modulo *b*

58

☒ This may not be best function …
☒ Read Knuth Vol. 3 if you really need to select a good function.

Good hash function: ☞ Expected number of keys/bucket is the same for all buckets

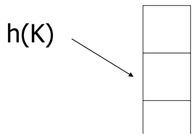
59

## Within a bucket:

- Do we keep keys sorted?
- Yes, if CPU time critical & Inserts/Deletes not too frequent

60

Next: example to illustrate
inserts, overflows, deletes

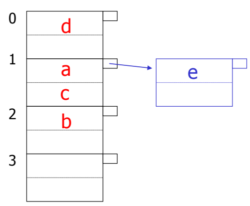



61

EXAMPLE 2 records/bucket

INSERT:
h(a) = 1
h(b) = 2
h(c) = 1
h(d) = 0

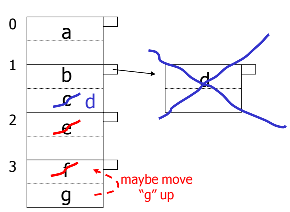
h(e) = 1




62

EXAMPLE: deletion

Delete:
e
f
c




63

Rule of thumb:

- Try to keep space utilization between 50% and 80%

Utilization = $\frac{\text{\# keys used}}{\text{total \# keys that fit}}$

- If < 50%, wasting space
- If > 80%, overflows significant

depends on how good hash function is & on # keys/bucket

64

How do we cope with growth?

- Overflows and reorganizations
- Dynamic hashing
  - Extensible
  - Linear

65

Extensible hashing: two ideas

(a) Use *i* of *b* bits output by hash function

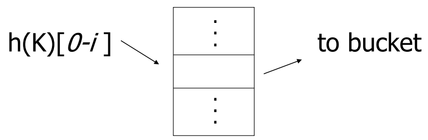
$\longleftarrow b \longrightarrow$

h(K) → $\boxed{00110101}$

use *i* → grows over time….

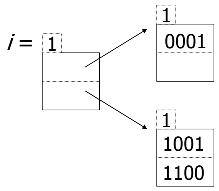
66

(b) Use directory



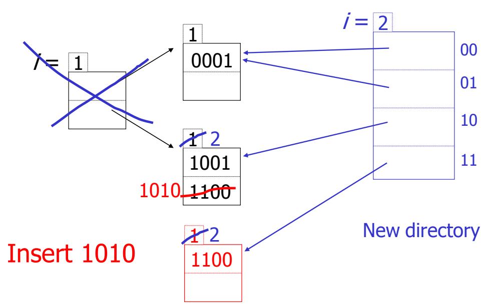



Example: h(k) is 4 bits; 2 keys/bucket

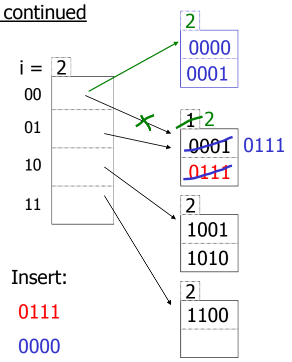



"slide" conventions:
- slide shows h(k), while actual directory has key+pointer

Example: h(k) is 4 bits; 2 keys/bucket

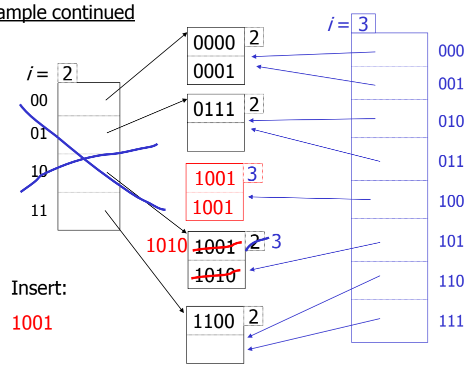


Example continued




70

Example continued




71

## Extensible hashing: deletion

- No merging of blocks
- Merge blocks
  and cut directory if possible
  (Reverse insert procedure)

72

## Deletion example:

- Run thru insert example in reverse!



## Summary Extensible hashing

⊕ Can handle growing files
- with less wasted space
- with no full reorganizations

⊖ Indirection
(Not bad if directory in memory)

⊖ Directory doubles in size
(Now it fits, now it does not)

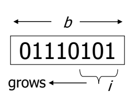


## Linear hashing

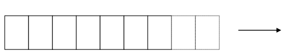
- Another dynamic hashing scheme

Two ideas:

(a) Use $i$ low order bits of hash

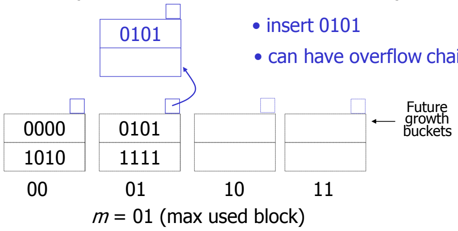



(b) File grows linearly



## Example *b*=4 bits, *i* =2, 2 keys/bucket

- insert 0101
- can have overflow chains!

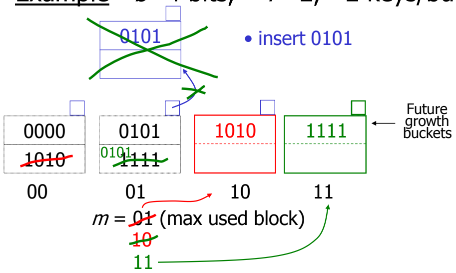



Rule: If $h(k)[i] \le m$, then
look at bucket h(k)[i ]
else, look at bucket $h(k)[i] - 2^{i-1}$

76

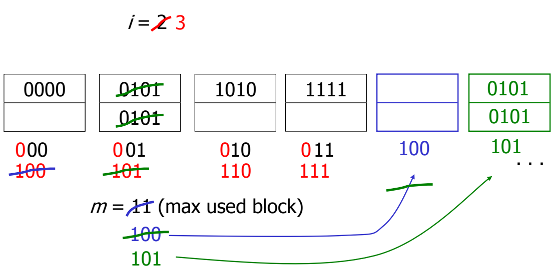
## Example *b*=4 bits, *i* =2, 2 keys/bucket

- insert 0101




77

## Example Continued: How to grow beyond this?




78

## When do we expand file?

- Keep track of: $\frac{\#\text{used slots (incl. overflow)}}{\#\text{total slots in primary buckets}} = U$

equiv, $\frac{\#\text{(indexed key ptr pairs)}}{\#\text{total slots in primary buckets}}$

  - If U > threshold then increase $m$
    (and $i$, when $m$ reaches $2^i$)



## Summary Linear Hashing

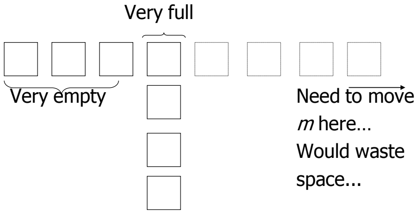
⊕ Can handle growing files
- with less wasted space
- with no full reorganizations

⊕ No indirection like extensible hashing

⊖ Can still have overflow chains



## Example: BAD CASE



## Summary

Hashing
- How it works
- Dynamic hashing
  - Extensible
  - Linear



## Next:

- Indexing vs Hashing
- Index definition in SQL
- Multiple key access



## Indexing vs Hashing

- Hashing good for probes given key

  e.g., SELECT …
  FROM R
  WHERE R.A = 5

## Indexing vs Hashing

- INDEXING (Including B Trees) good for
  Range Searches:
  e.g., SELECT
  FROM R
  WHERE R.A > 5

85

## Index definition in SQL

- Create index name on rel (attr)
- Create unique index name on rel (attr)

  defines candidate key

- Drop INDEX name

86

Note CANNOT SPECIFY TYPE OF INDEX
(e.g. B-tree, Hashing, …)
OR PARAMETERS
(e.g. Load Factor, Size of Hash,...)

... at least in SQL...

87

$\boxed{\text{Note}}$ ATTRIBUTE <u>LIST</u> ⇒ MULTIKEY INDEX
(next)
e.g., <u>CREATE</u> <u>INDEX</u> foo <u>ON</u> R(A,B,C)

88

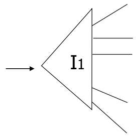
$\boxed{\text{Multi-key Index}}$

Motivation: Find records where
DEPT = "Toy" AND SAL > 50k

89

<u>Strategy I:</u>

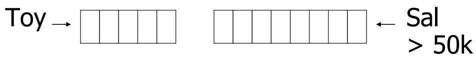
- Use one index, say Dept.
- Get all Dept = "Toy" records and check their salary




90

## Strategy II:

- Use 2 Indexes; Manipulate Pointers

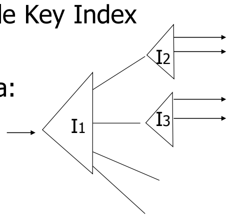



91

## Strategy III:

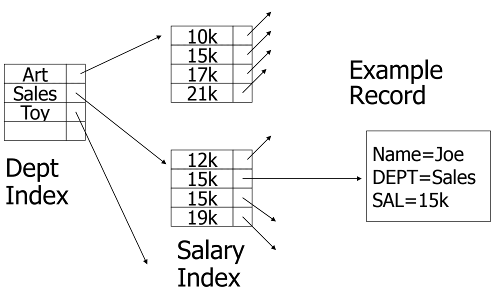
- Multiple Key Index

One idea:




92

## Example




93

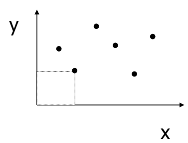
For which queries is this index good?

☐ Find RECs Dept = "Sales" $\wedge$ SAL=20k
☐ Find RECs Dept = "Sales" $\wedge$ SAL $\geq$ 20k
☐ Find RECs Dept = "Sales"
☐ Find RECs SAL = 20k

94

Interesting application:

- Geographic Data




DATA:
<$X_1$,$Y_1$, Attributes>
<$X_2$,$Y_2$, Attributes>
⋮

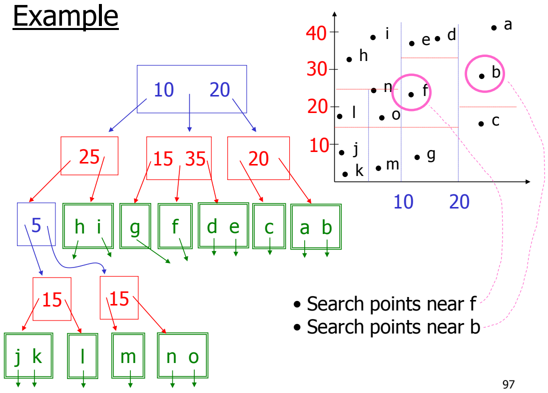
95

Queries:

- What city is at <Xi,Yi>?
- What is within 5 miles from <Xi,Yi>?
- Which is closest point to <Xi,Yi>?

96

## Example




- Search points near f
- Search points near b



## Queries

- Find points with Yi > 20
- Find points with Xi < 5
- Find points "close" to i = <12,38>
- Find points "close" to b = <7,24>



- Many types of geographic index structures have been suggested
  - Quad Trees
  - R Trees

## Outline/summary

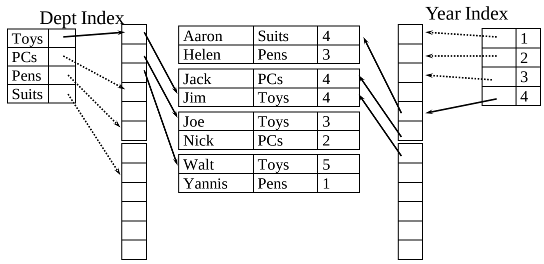
- Conventional Indexes
  - Sparse vs. dense
  - Primary vs. secondary
- B+ trees
- Hashing schemes
- Bitmap indices --> Next

100

## Revisit: Processing queries without accessing records until last step

Find employees of the Toys dept with 4 years in the company

SELECT Name FROM Employee
WHERE Dept="Toys" AND Year=4




## Bitmap indices: Alternate structure, heavily used in OLAP

Assume the tuples of the Employees table are ordered.

Conceptually only!

Dept Index

| | |
|---|---|
| Toys | 00011010 |
| PCs | 00100100 |
| Pens | 01000001 |
| Suits | 10000000 |

| | | |
|---|---|---|
| Aaron | Suits | 4 |
| Helen | Pens | 3 |
| Jack | PCs | 4 |
| Jim | Toys | 4 |
| Joe | Toys | 3 |
| Nick | PCs | 2 |
| Walt | Toys | 1 |
| Yannis | Pens | 1 |

Year Index

| | |
|---|---|
| 00000011 | 1 |
| 00000100 | 2 |
| 01001000 | 3 |
| 10110000 | 4 |

**+** Find even more quickly intersections and unions (e.g., Dept="Toys" AND Year=4)
**?** Seems it needs too much space -> We'll do compression
**?** How do we deal with insertions and deletions -> Easier than you think

102

## 2nlog*m* Compression

- Naive solution needs *mn* bits, where *m* is #distinct values and *n* is #tuples
- But there is just *n* 1's=> let's utilize this
- Bit encoding of sequence of **runs** (e.g. [3,0,1])

Toys: 00011010

3 0 1

Third run says: The 3rd ace appears after 1 zero after the 2nd

First run says: The first ace appears after 3 zeros

Second run says: The 2nd ace appears immediately after the 1st

*10***11** *0****0*** *0* **1**

*10* says: The binary encoding of the first number needs 1+1 digits.
**11** says: The first number is 3

103

## 2nlog m compression

- Example
- Pens: 01000001
- Sequence [1,5]
- Encoding: *0***1***110***101**

104

## Insertions and deletions & miscellaneous engineering

- Assume tuples are inserted in order
- Deletions: Do nothing
- Insertions: If tuple *t* with value *v* is inserted, add one more run in *v*'s sequence (compact bitmap)

105

## The BIG picture….

- Chapters 2 & 3: Storage, records, blocks...
- Chapter 4 & 5: Access Mechanisms
  - Indexes
  - B trees
  - Hashing
  - Multi key
- Chapter 6 & 7: Query Processing NEXT

106