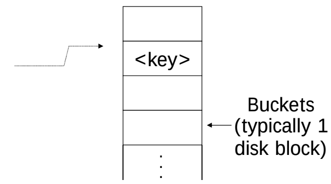
CS 245: Database System Principles

Notes 5: Hashing and More

Hector Garcia-Molina

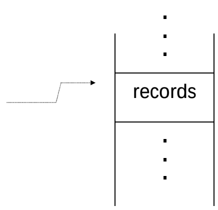
Hashing

key → h(key)



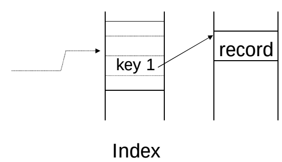
Two alternatives

(1) key → h(key)



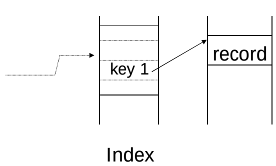
Two alternatives

(2) key → h(key)



Two alternatives

(2) key → h(key)



- Alt (2) for “secondary” search key

Example hash function

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

☒ This may not be best function …

☒ Read Knuth Vol. 3 if you really need to select a good function.

☒ This may not be best function …

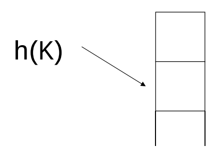
☒ Read Knuth Vol. 3 if you really need to select a good function.

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

<u>Within a bucket:</u>

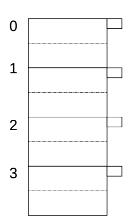
- Do we keep keys sorted?
- Yes, if CPU time critical & Inserts/Deletes not too frequent

<u>Next:</u> example to illustrate inserts, overflows, deletes



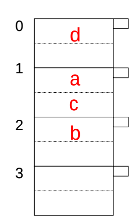
<u>EXAMPLE</u> 2 records/bucket

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



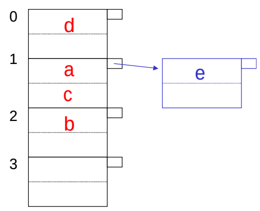
<u>EXAMPLE</u> 2 records/bucket

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



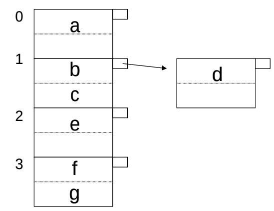
EXAMPLE 2 records/bucket

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



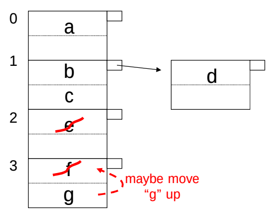
EXAMPLE: deletion

Delete:
e
f



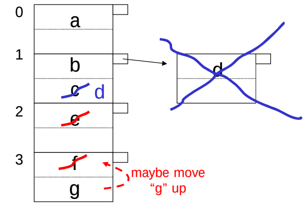
EXAMPLE: deletion

Delete:
e
f
c



EXAMPLE: deletion

Delete:
e
f
c



Rule of thumb:

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

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

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

How do we cope with growth?

- Overflows and reorganizations
- Dynamic hashing

How do we cope with growth?

- Overflows and reorganizations
- Dynamic hashing
  - Extensible
  - Linear

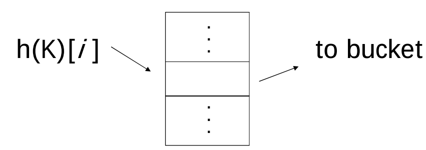
Extensible hashing: two ideas

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

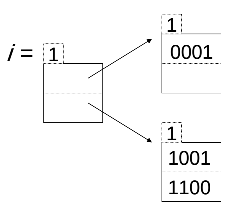
$$h(K) \rightarrow \overbrace{\boxed{\underbrace{00}_{\text{use } i}110101}}^{b}$$

use *i* → grows over time....

(b) Use directory

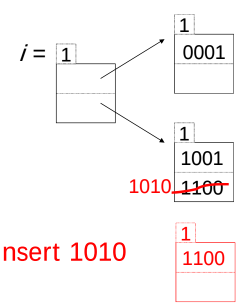


Example: h(k) is 4 bits; 2 keys/bucket



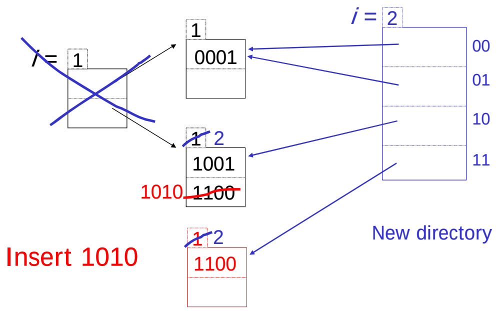
Insert 1010

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

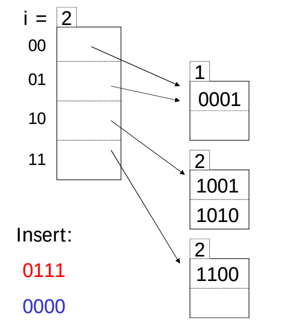


Insert 1010

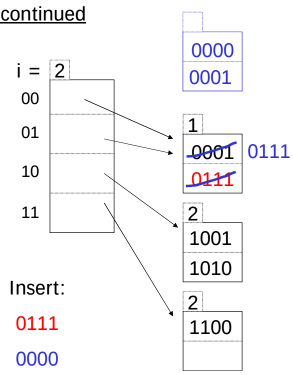
Example: h(k) is 4 bits; 2 keys/bucket



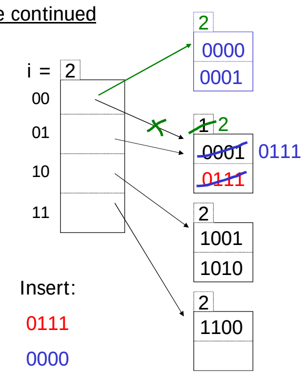
Example continued



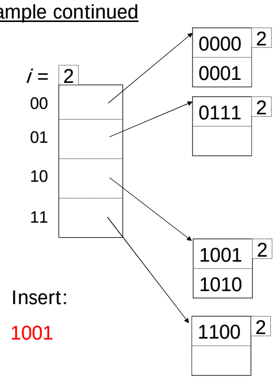
Example continued



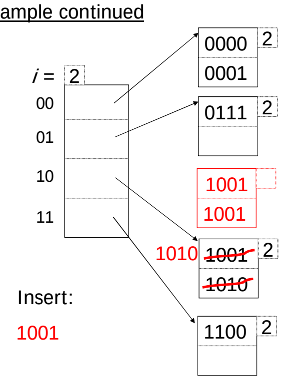
Example continued



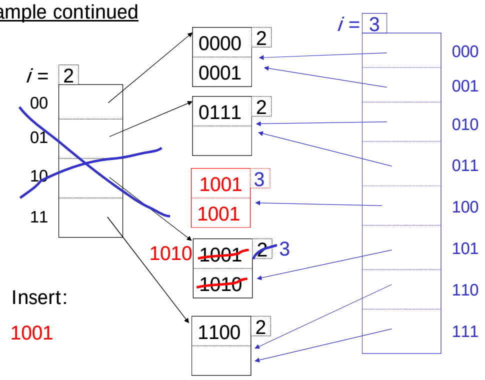
Example continued



Example continued



Example continued



Extensible hashing: <u>deletion</u>

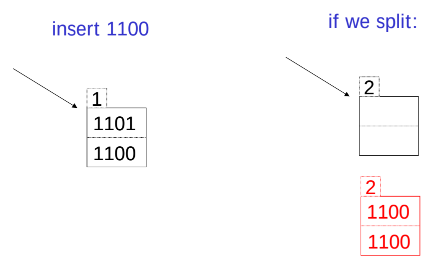
- No merging of blocks
- Merge blocks
  and cut directory if possible
  (Reverse insert procedure)

<u>Deletion example:</u>

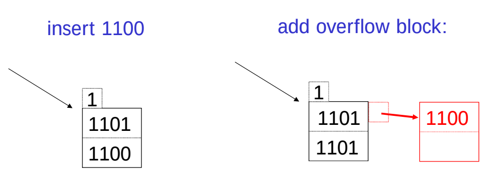
- Run thru insert example in reverse!

<u>Note: Still need overflow chains</u>

- Example: many records with duplicate keys



<u>Solution: overflow chains</u>



Summary Extensible hashing

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

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

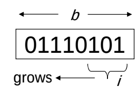
(Now it fits, now it does not)

Linear hashing

- Another dynamic hashing scheme

Two ideas:

(a) Use i low order bits of hash

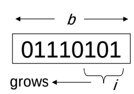


Linear hashing

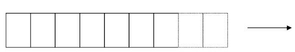
- Another dynamic hashing scheme

Two ideas:

(a) Use i low order bits of hash



(b) File grows linearly



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

00	01	10	11
0000	0101		
1010	1111		

Future growth buckets: 10, 11

m = 01 (max used block)

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

00	01	10	11
0000	0101		
1010	1111		

Future growth buckets: 10, 11

m = 01 (max used block)

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

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

- insert 0101

00	01	10	11
0000	0101		
1010	1111		

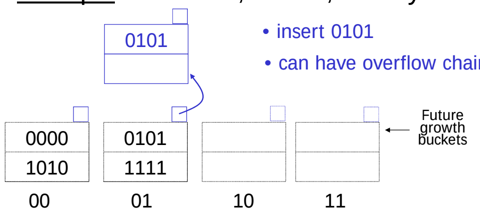
Future growth buckets: 10, 11

m = 01 (max used block)

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

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

- insert 0101
- can have overflow chains!

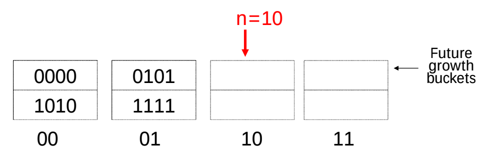


m = 01 (max used block)

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

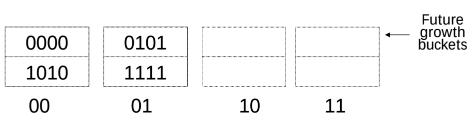
Note

- In textbook, n is used instead of m
- n=m+1



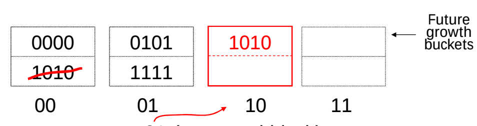
m = 01 (max used block)

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



m = 01 (max used block)

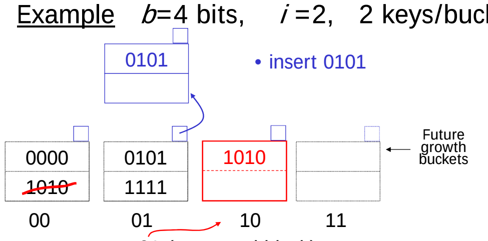
Example b=4 bits, i=2, 2 keys/bucket



m = ~~01~~ (max used block)
10

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

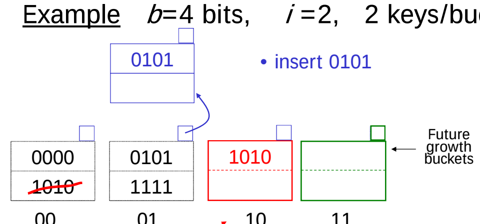
- insert 0101



m = ~~01~~ (max used block)
10

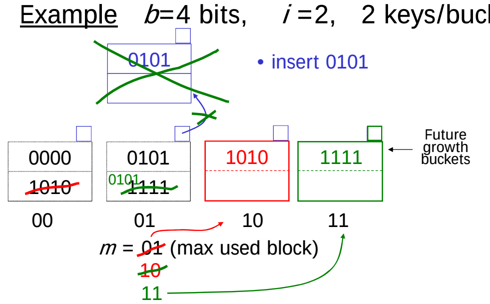
Example b=4 bits, i=2, 2 keys/bucket

- insert 0101

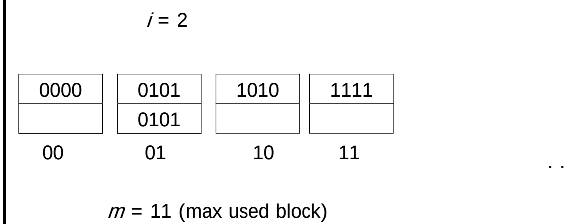


m = ~~01~~ (max used block)
~~10~~
11

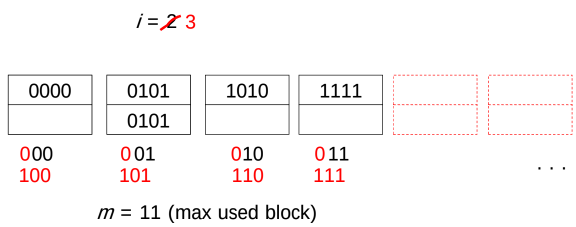
Example b=4 bits, i=2, 2 keys/bucket



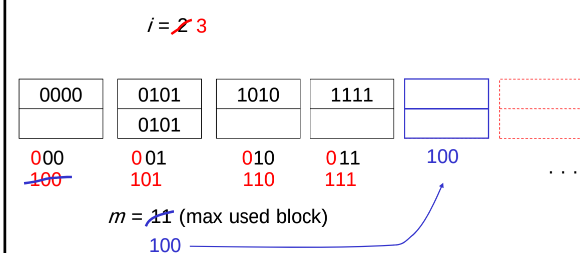
Example Continued: How to grow beyond this?



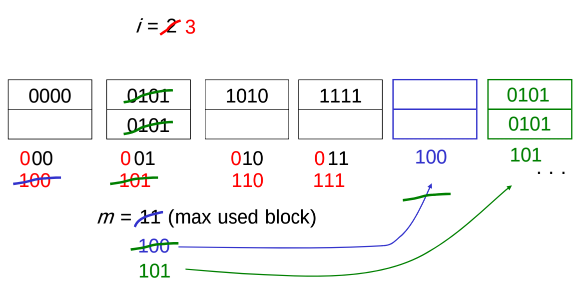
Example Continued: How to grow beyond this?



Example Continued: How to grow beyond this?



Example Continued: How to grow beyond this?



✉ When do we expand file?

- Keep track of: $\frac{\text{\# used slots}}{\text{total \# of slots}} = U$

When do we expand file?

- Keep track of: $\frac{\#\text{ used slots}}{\text{total }\#\text{ of slots}} = U$

- If U > threshold then increase *m*
  (and maybe *i*)

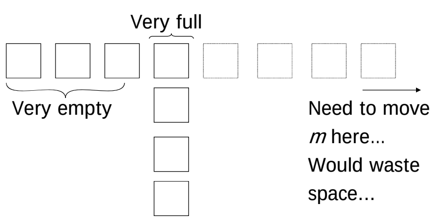
Summary Linear Hashing

⊕ 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

<u>Hashing</u>
- How it works
- Dynamic hashing
  - Extensible
  - Linear

<u>Next:</u>

- 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

Index definition in SQL

- Create index name on rel (attr)
- Create unique index name on rel (attr)
  → defines candidate key

- Drop INDEX name

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

... at least in SQL...

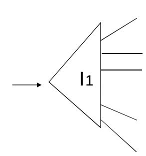
Note ATTRIBUTE LIST ⇒ MULTIKEY INDEX
(next)
e.g., CREATE INDEX foo ON R(A,B,C)

Multi-key Index

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

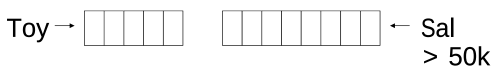
Strategy I:

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



Strategy II:

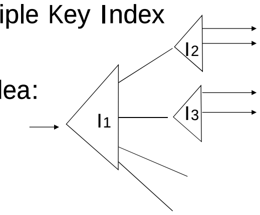
- Use 2 Indexes; Manipulate Pointers



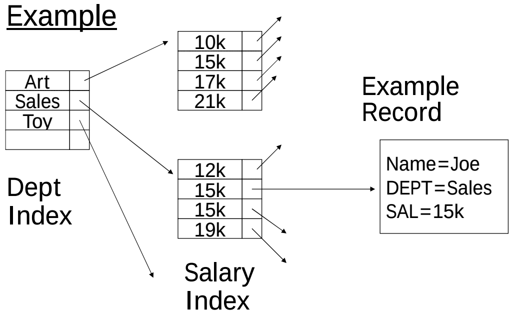
Strategy III:

- Multiple Key Index

One idea:



Example

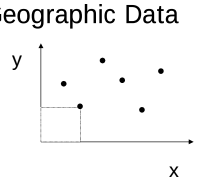


For which queries is this index good?

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

Interesting application:

- Geographic Data



DATA:

<X_1,Y_1, Attributes>

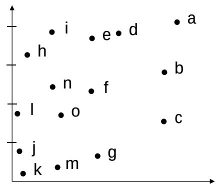
<X_2,Y_2, Attributes>

⋮

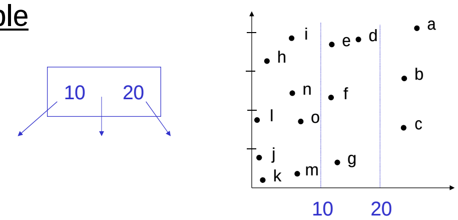
Queries:

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

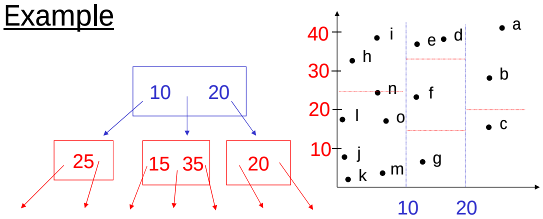
Example



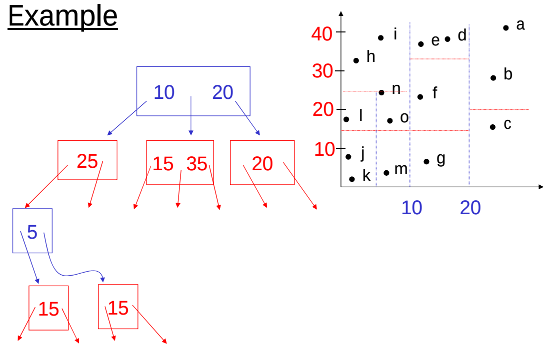
Example



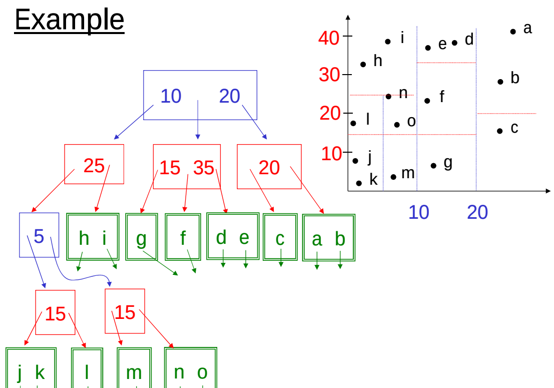
Example



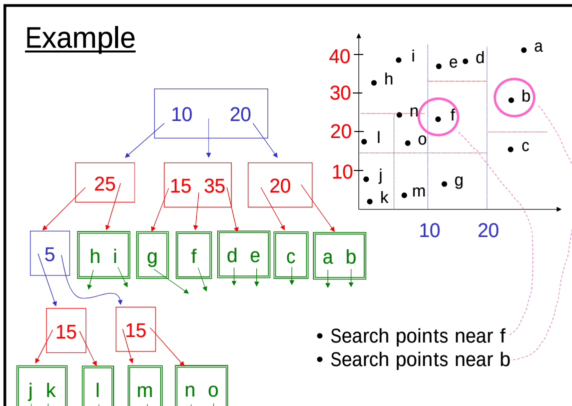
Example



Example



Example



Queries

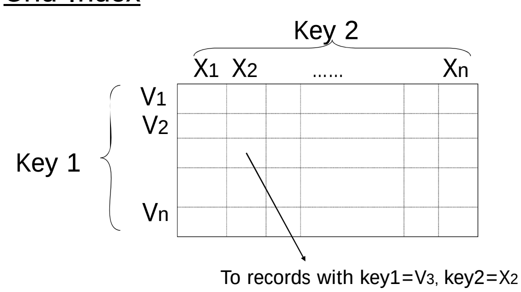
- Find points with Yi > 20
- Find points with Xi < 5
- Find points “close” to i = <12,38>
- Find points “close” to b = <7,24>

- Many types of geographic index structures have been suggested
  - kd-Trees (very similar to what we described here)
  - Quad Trees
  - R Trees
  - ...

Two more types of multi key indexes

- Grid
- Partitioned hash

Grid Index



CLAIM

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

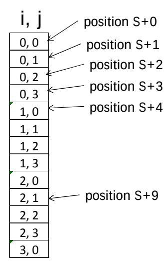
CLAIM

- 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) =



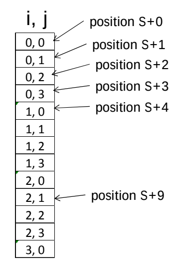
- How do we find entry i,j in linear structure?

max number of
i values N=4

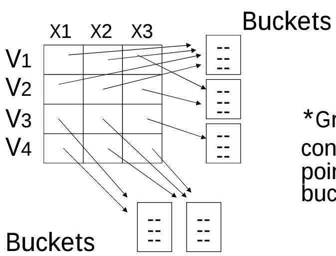
pos(i, j) = S + iN + j

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

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



Solution: Use Indirection

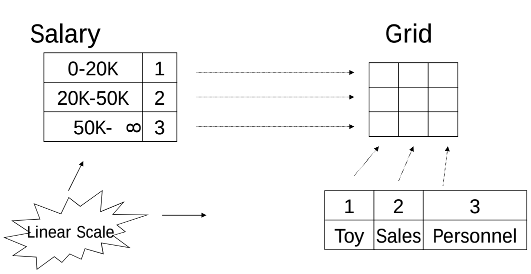


*Grid only contains pointers to buckets

With indirection:

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

Can also index grid on value ranges

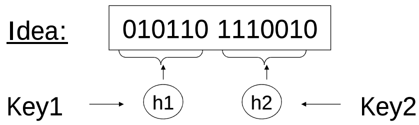


Grid files

⊕ 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	000	
h1(sales)	=1	001	
h1(art)	=1	010	
.		011	
h2(10k)	=01	100	
h2(20k)	=11	101	
h2(30k)	=01	110	
h2(40k)	=00	111	
:			

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

EX:

h1(toy)	=0	000	
h1(sales)	=1	001	<Fred>
h1(art)	=1	010	
.		011	
h2(10k)	=01	100	
h2(20k)	=11	101	<Joe><Sally>
h2(30k)	=01	110	
h2(40k)	=00	111	
:			

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

h1(toy)	=0	000	<Fred>
h1(sales)	=1	001	<Joe><Jan>
h1(art)	=1	010	<Mary>
.		011	
h2(10k)	=01	100	<Sally>
h2(20k)	=11	101	
h2(30k)	=01	110	<Tom><Bill>
h2(40k)	=00	111	<Andy>
:			

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

h1(toy)	=0	000	<Fred>
h1(sales)	=1	001	<Joe><Jan>
h1(art)	=1	010	<Mary>
.		011	
h2(10k)	=01	100	<Sally>
h2(20k)	=11	101	
h2(30k)	=01	110	<Tom><Bill>
h2(40k)	=00	111	<Andy>
:			

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

h1(toy)	=0	000	<Fred>
h1(sales)	=1	001	<Joe><Jan>
h1(art)	=1	010	<Mary>
.		011	
h2(10k)	=01	100	<Sally>
h2(20k)	=11	101	
h2(30k)	=01	110	<Tom><Bill>
h2(40k)	=00	111	<Andy>
:			

- Find Emp. with Sal=30k

h1(toy)	=0	000	<Fred>
h1(sales)	=1	001	<Joe><Jan>
h1(art)	=1	010	<Mary>
.		011	
h2(10k)	=01	100	<Sally>
h2(20k)	=11	101	
h2(30k)	=01	110	<Tom><Bill>
h2(40k)	=00	111	<Andy>
:			

- Find Emp. with Sal=30k

look here

h1(toy)	=0	000	<Fred>
h1(sales)	=1	001	<Joe><Jan>
h1(art)	=1	010	<Mary>
.		011	
h2(10k)	=01	100	<Sally>
h2(20k)	=11	101	
h2(30k)	=01	110	<Tom><Bill>
h2(40k)	=00	111	<Andy>
:			

- Find Emp. with Dept. = Sales

h1(toy)	=0	000	<Fred>
h1(sales)	=1	001	<Joe><Jan>
h1(art)	=1	010	<Mary>
.		011	
h2(10k)	=01	100	<Sally>
h2(20k)	=11	101	
h2(30k)	=01	110	<Tom><Bill>
h2(40k)	=00	111	<Andy>
:			

- Find Emp. with Dept. = Sales

look here

Summary

Post hashing discussion:

- Indexing vs. Hashing
- SQL Index Definition
- Multiple Key Access
    - Multi Key Index
        - Variations: Grid, Geo Data
    - Partitioned Hash

Reading Chapter 5

- Skim the following sections:
    - Sections 14.3.6, 14.3.7, 14.3.8 [Second Ed: 14.6.6, 14.6.7, 14.6.8]
    - Sections 14.4.2, 14.4.3, 14.4.4 [Second Ed: 14.7.2, 14.7.3, 14.7.4]
- Read the rest

The BIG picture....

- Chapters 11 & 12 [13]: Storage, records, blocks...
- Chapters 13 & 14 [14]: Access Mechanisms
    - Indexes
    - B trees
    - Hashing
    - Multi key
- Chapters 15 & 16 [15, 16]: Query Processing

NEXT