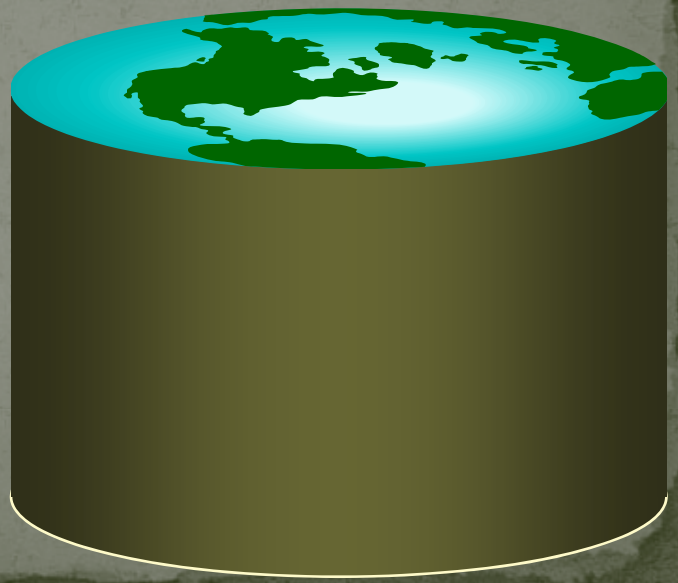
CS 505: Intermediate Topics to Database Systems

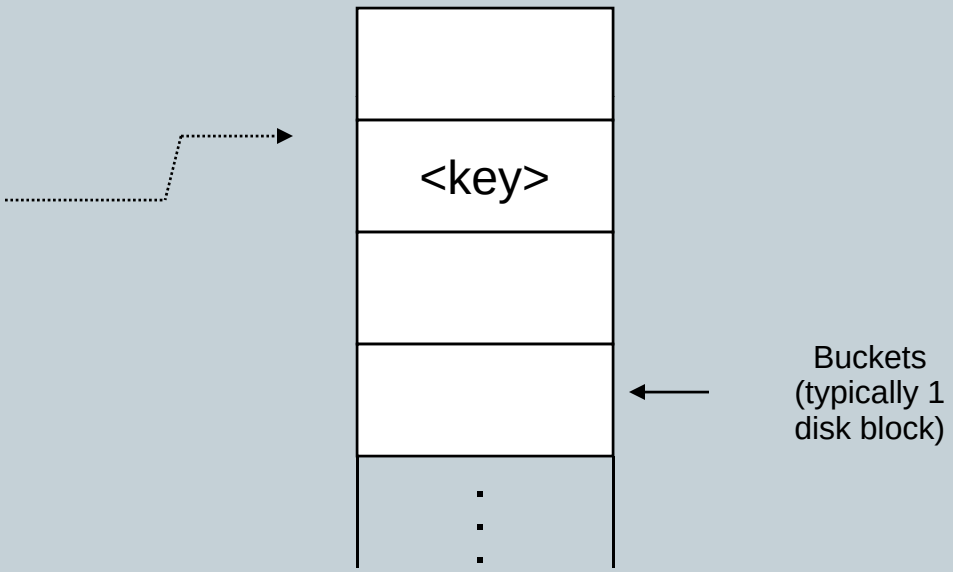
Instructor: Jinze Liu

Fall 2008



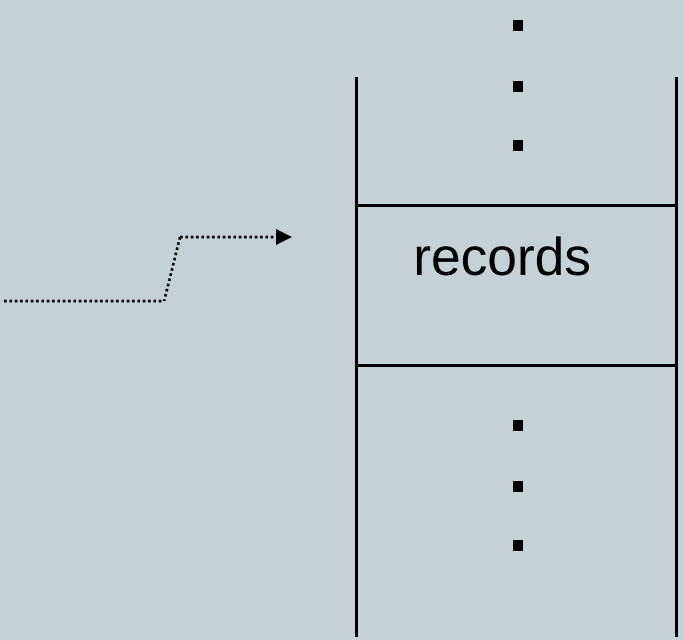
Hashing

$key \rightarrow h(key)$



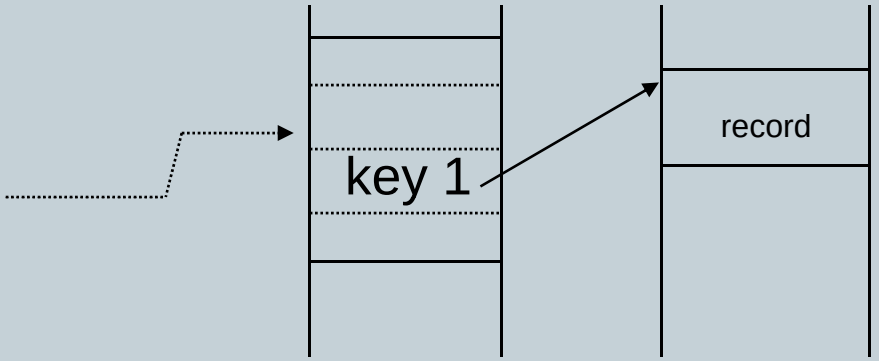
Two Alternatives

(1) key $\rightarrow$ h(key)



Two Alternatives

(2) key $\rightarrow$ h(key)



Index

- 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.

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

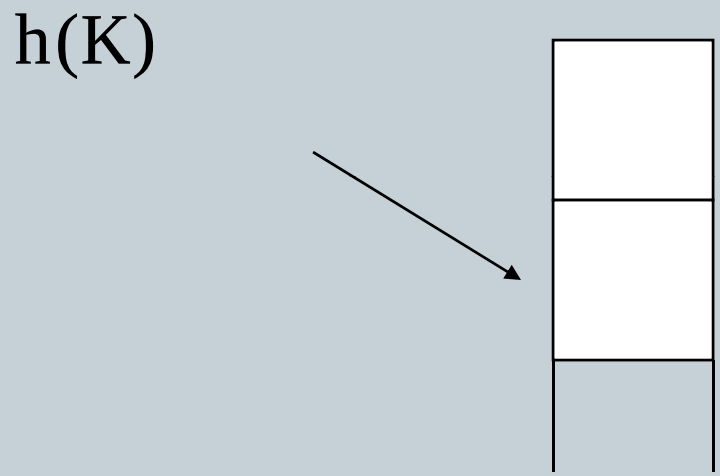
Within a bucket:

- Do we keep keys sorted?

  Yes, if CPU time critical

  & Inserts/Deletes not too frequent

Inserts, overflows, deletes



EXAMPLE 2 records/bucket

INSERT:

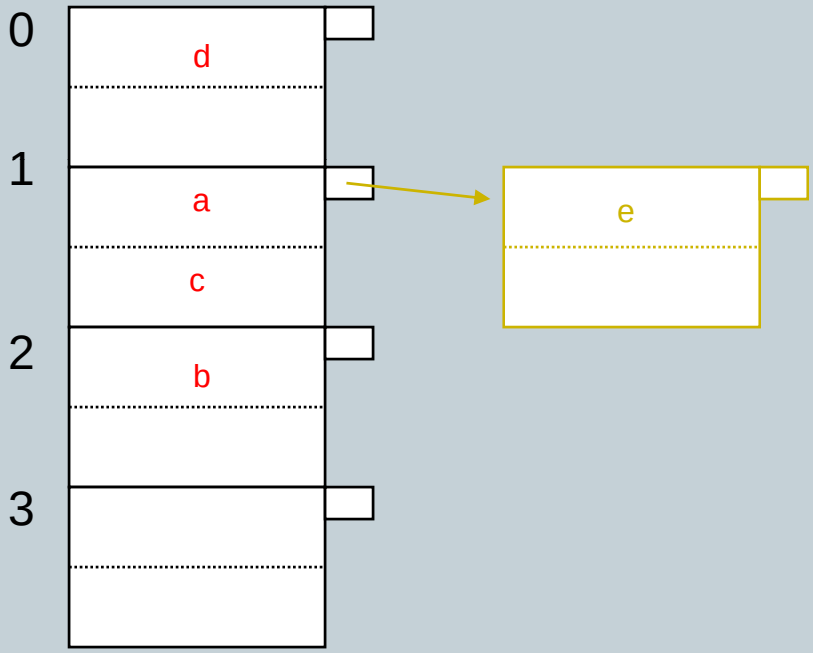
h(a) = 1

h(b) = 2

h(c) = 1

h(d) = 0

h(e) = 1



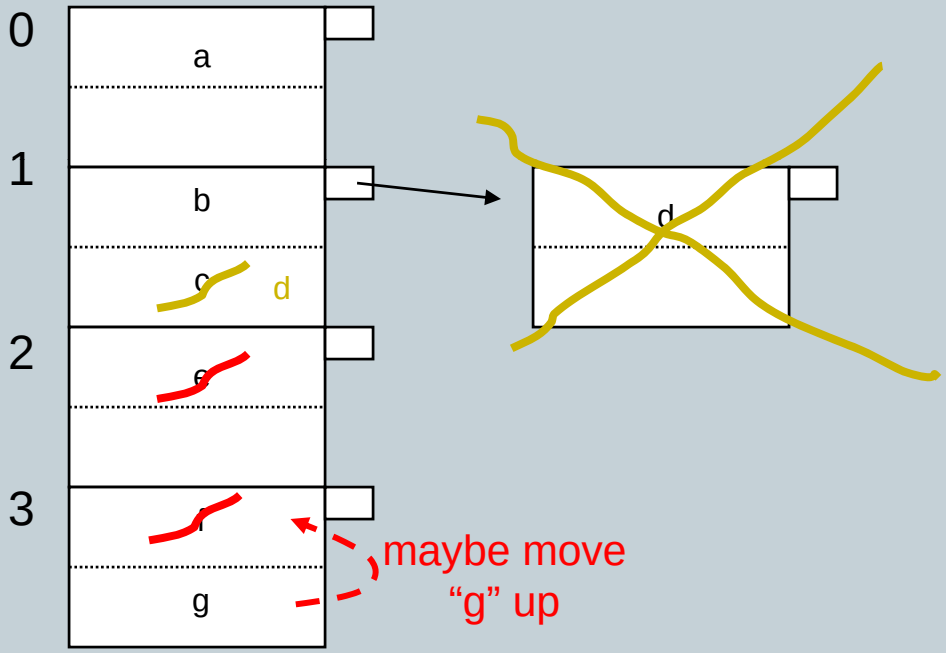
Example: Deletion

Delete:

e

f

c



Rule of thumb:

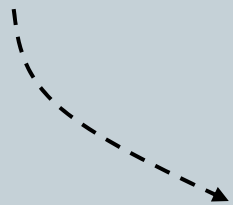
- Try to keep space utilization between 50% and 80%

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

- If < 50%, wasting space
- If > 80%, overflows significant
  - depends on how good hash function is & on # keys/bucket

How do we cope with growth?

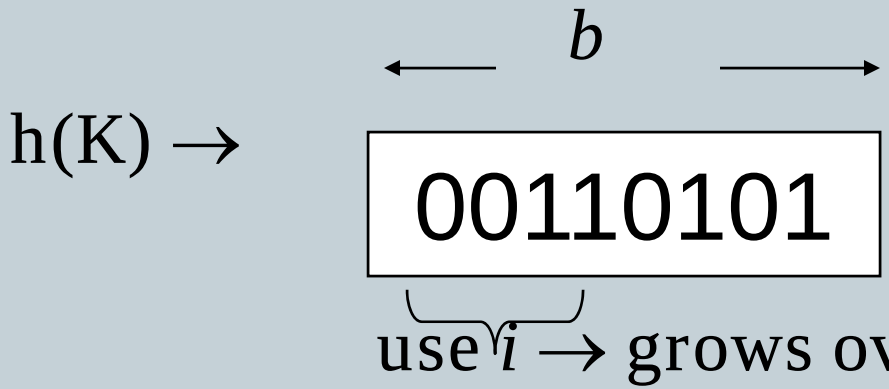
- Overflows and reorganizations
- Dynamic hashing



- Extensible
- Linear

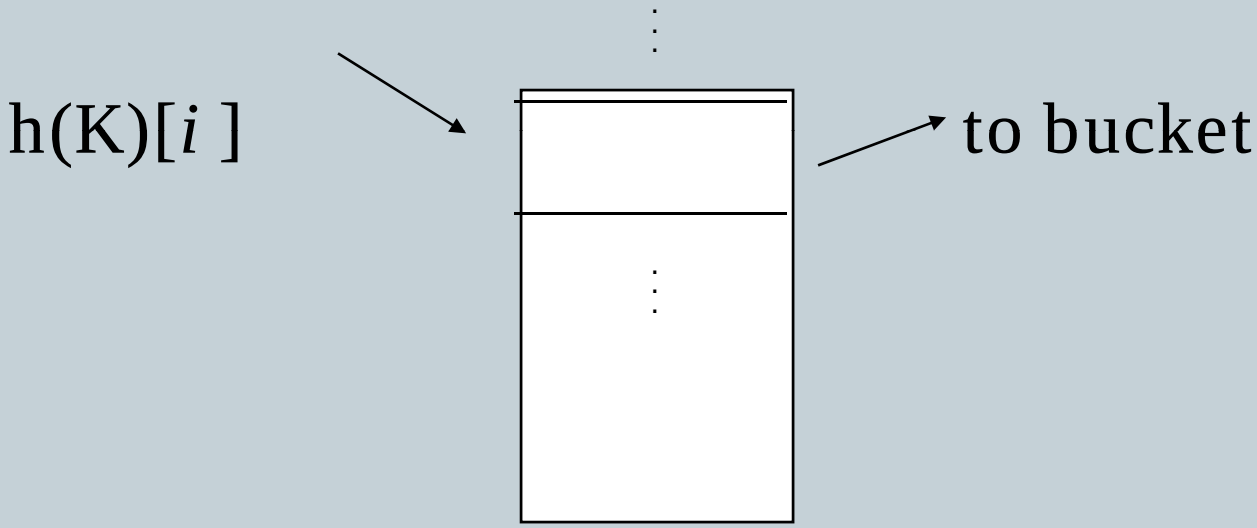
Extensible hashing: two ideas

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

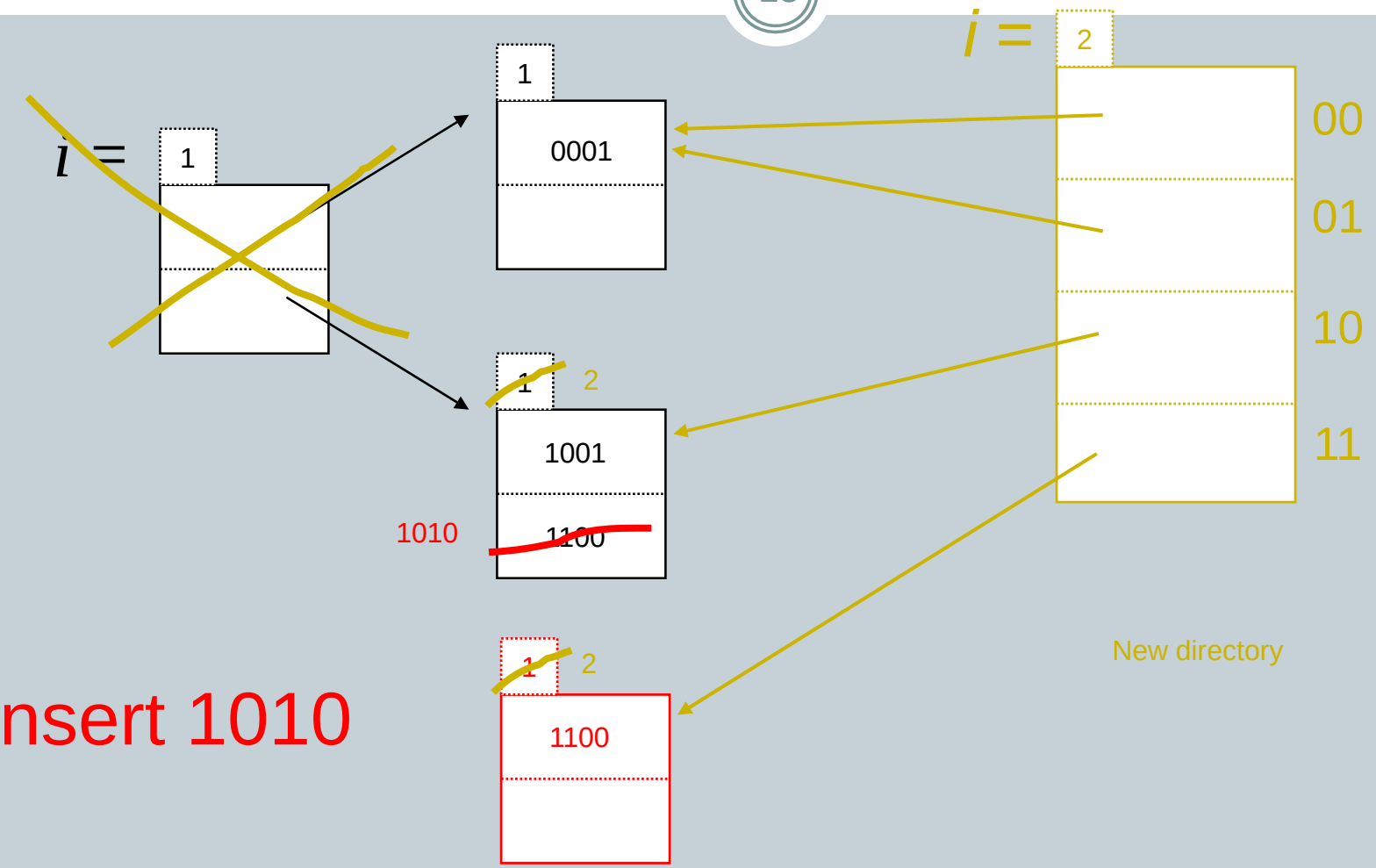


Extensible hashing: two ideas

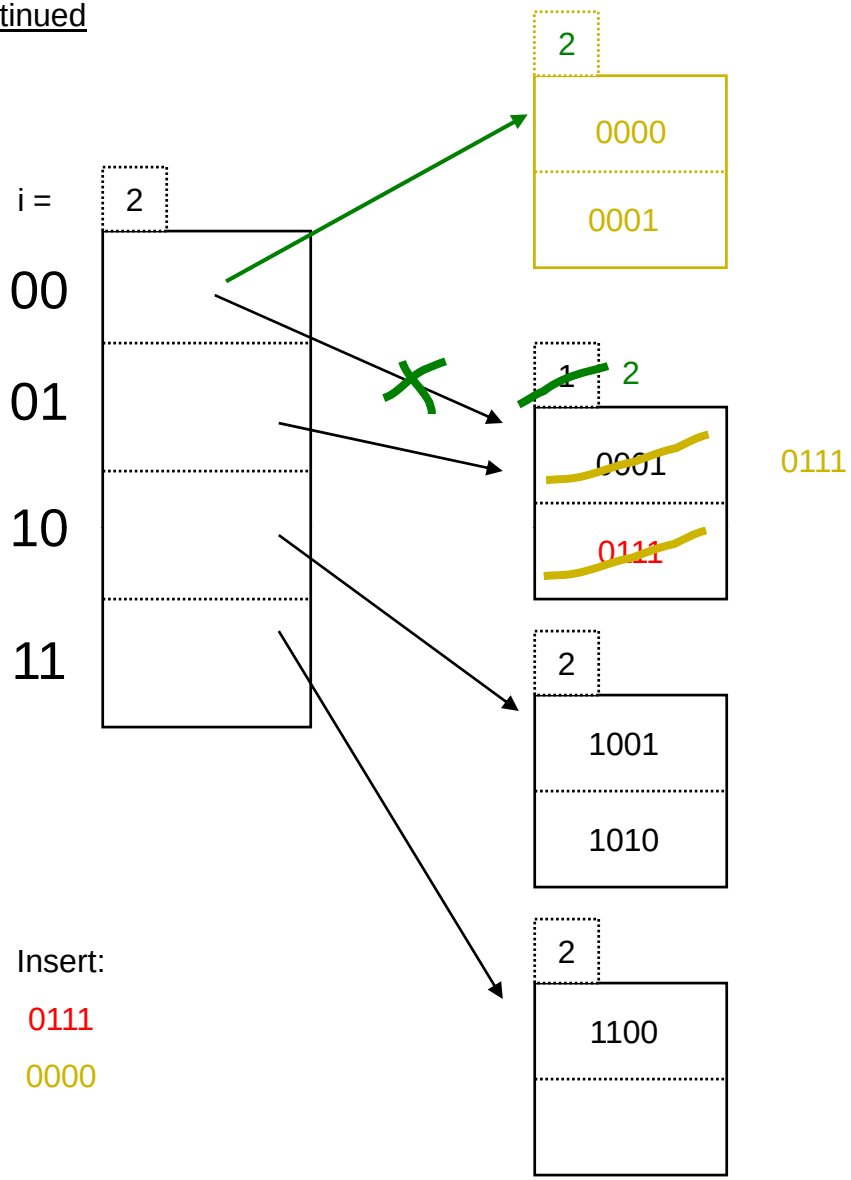
(b) Use directory



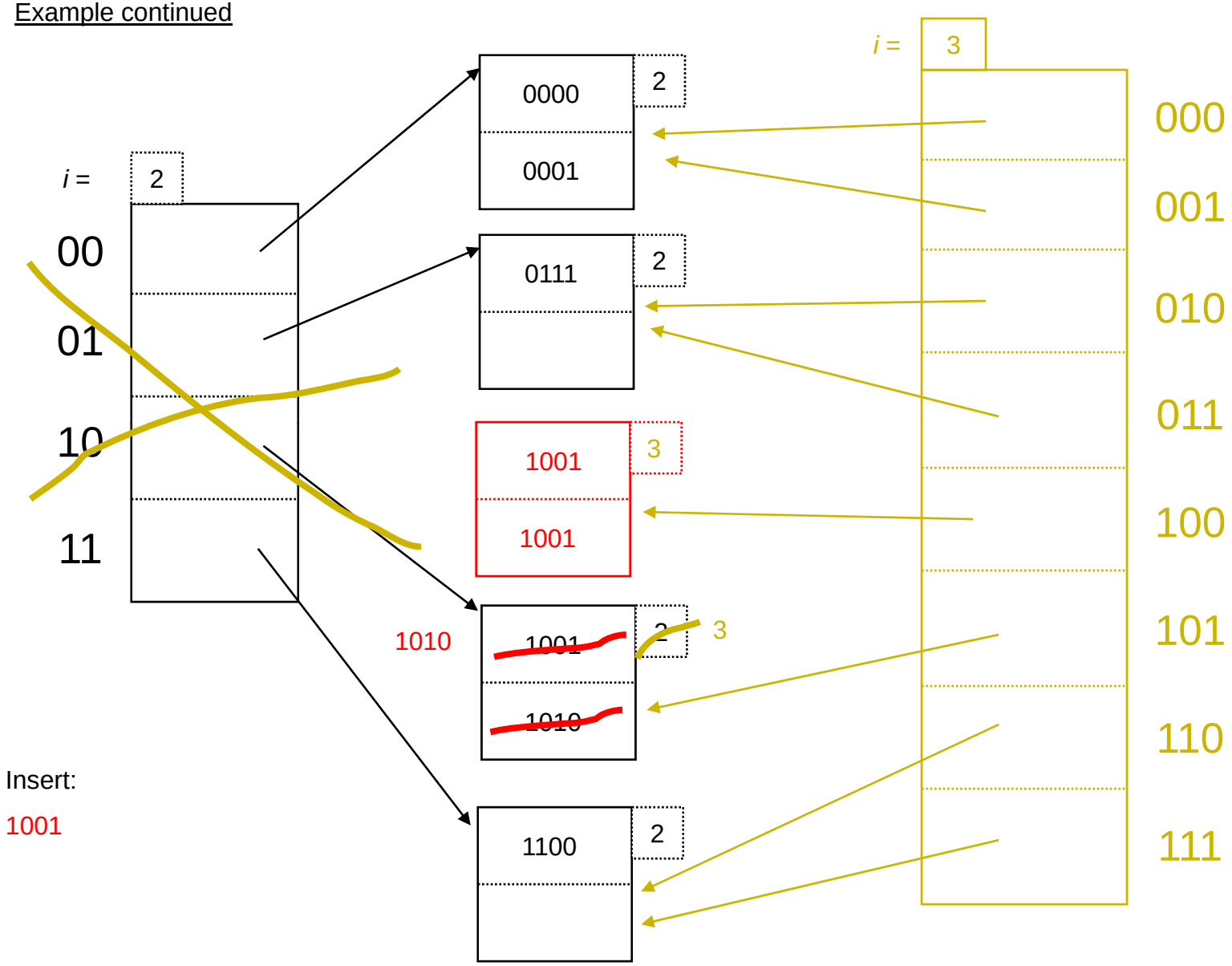
Example : h(k) is 4 bits; 2 keys/bucket



Example continued



Example continued



Insert:

1001

Extensible hashing: deletion

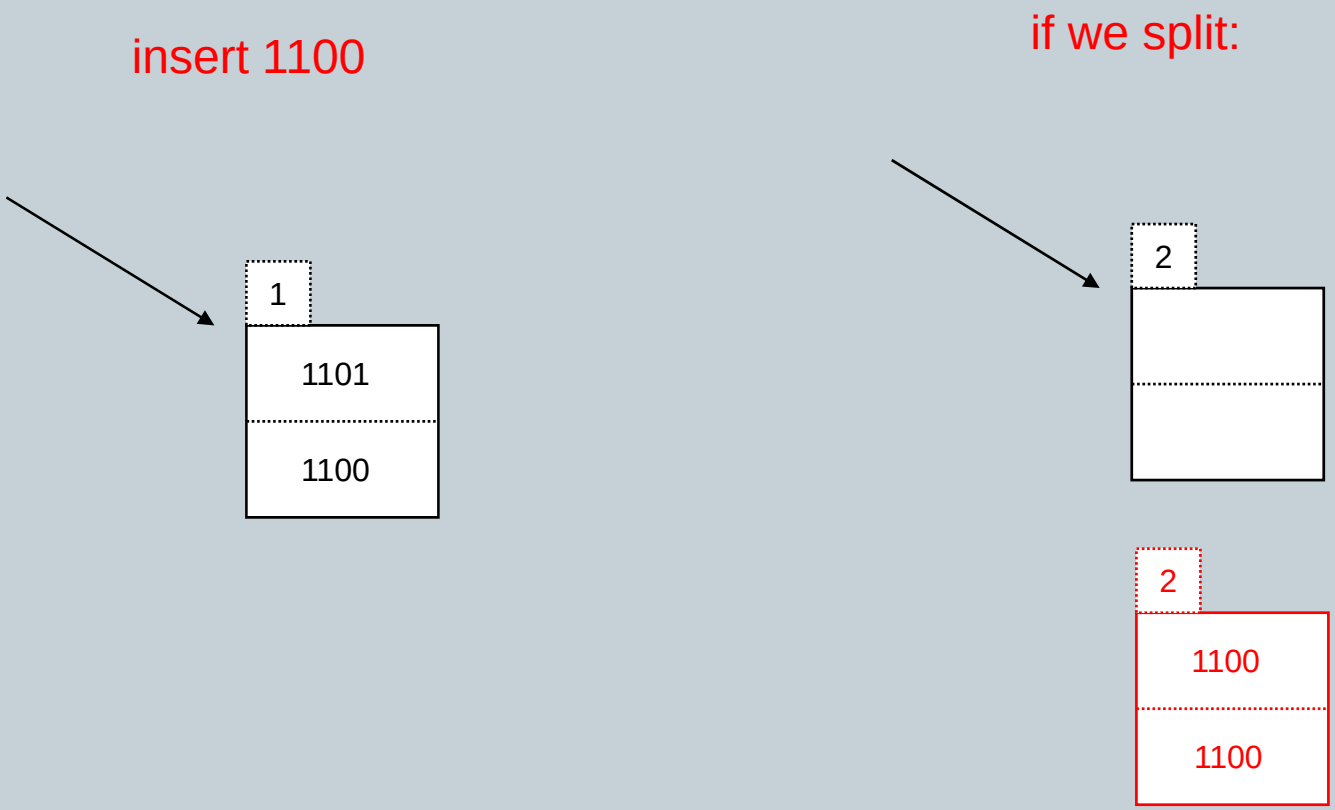
- No merging of blocks
- Merge blocks
  and cut directory if possible
  (Reverse insert procedure)

Deletion example:

- Run thru insert example in reverse!

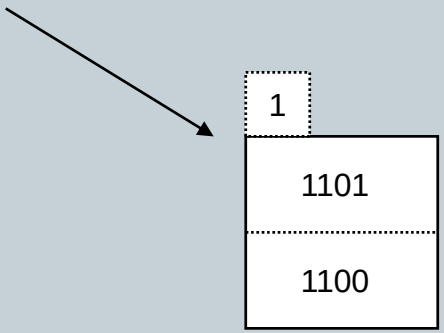
Note: Still need overflow chains

- Example: many records with duplicate keys

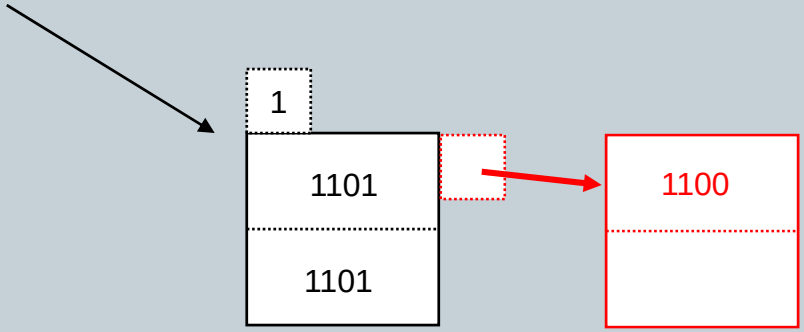


Solution: overflow chains

insert 1100



add overflow block:



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

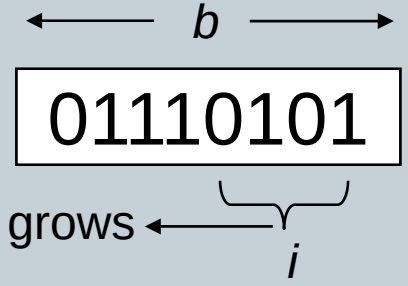
  (Now it fits, now it does not)

Linear hashing

- Another dynamic hashing scheme

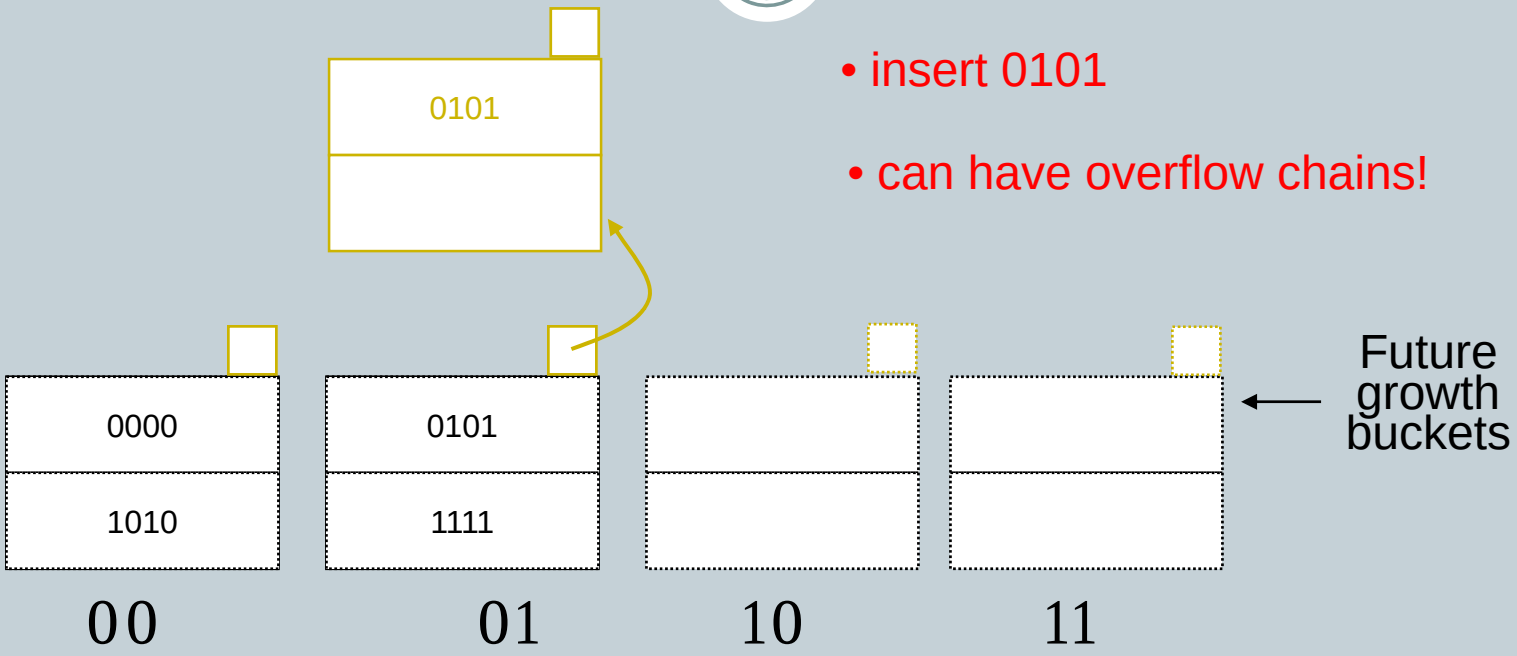
Two ideas:

(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!

m = 01 (max used block)

Rule

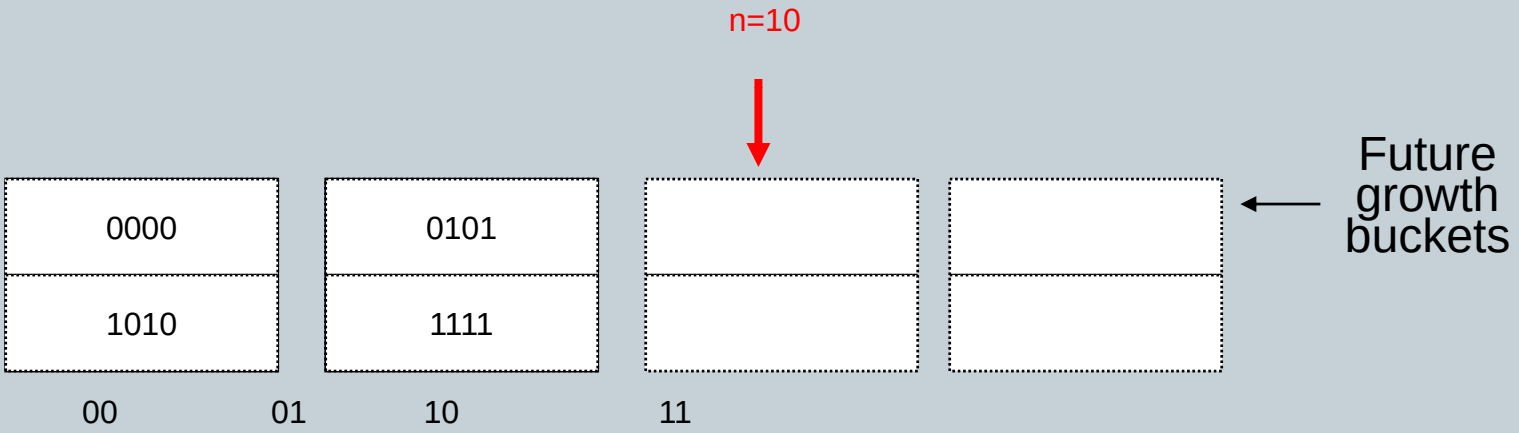
If h(k)[i] ≤ m, then

look at bucket h(k)[i]

else, look at bucket h(k)[i] - 2^{i-1}

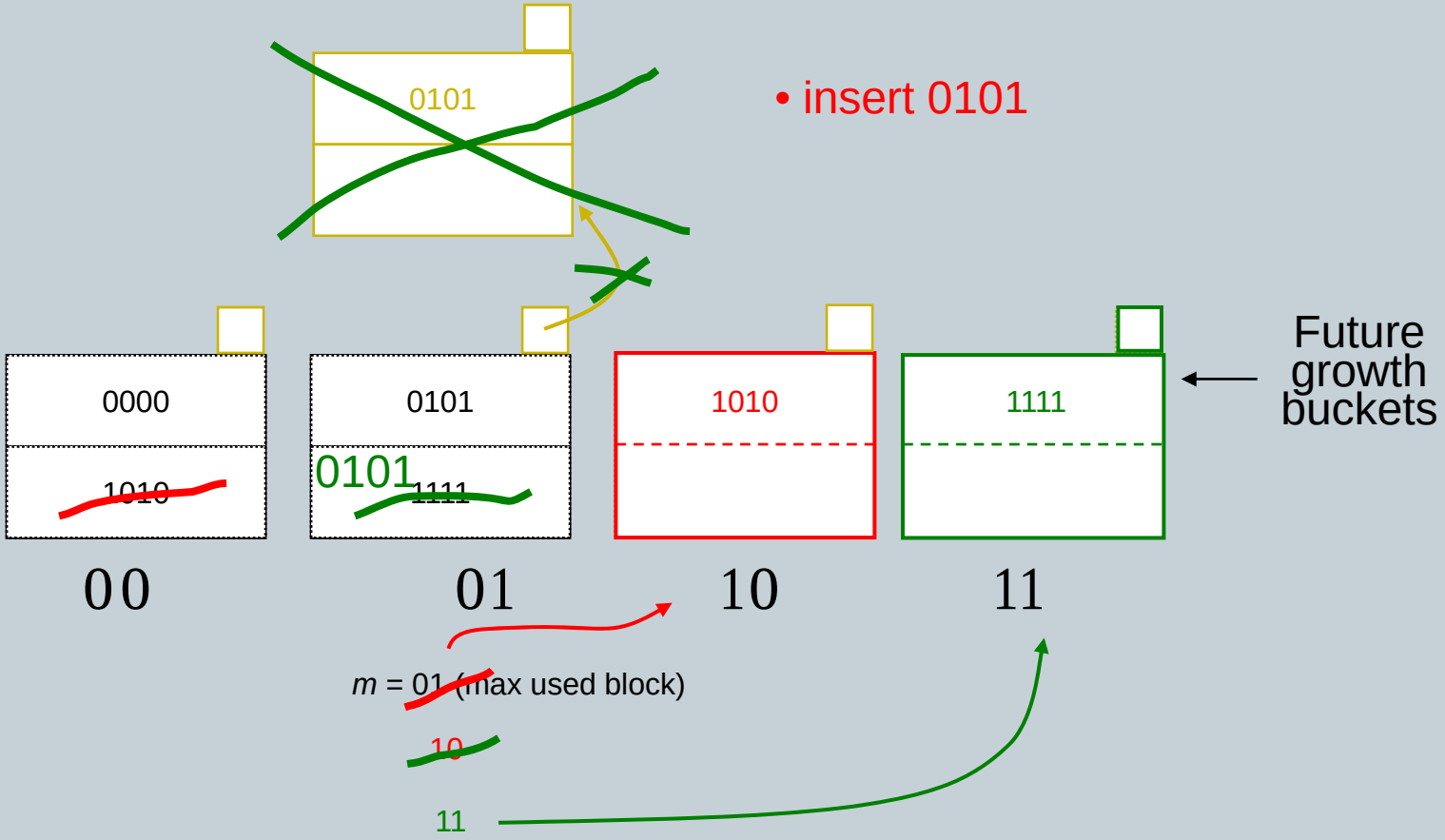
Note

- 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



Example Continued: How to grow beyond this?

i = 2 3

0000	0101	1010	1111		0101
	0101				0101
0 00	0 01	010	0 11	100	101
100 101	110 111				

. . .

m = 11 (max used block)

100

101

When do we expand file?

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

- If U > threshold then increase *m*

  (and maybe *i*)

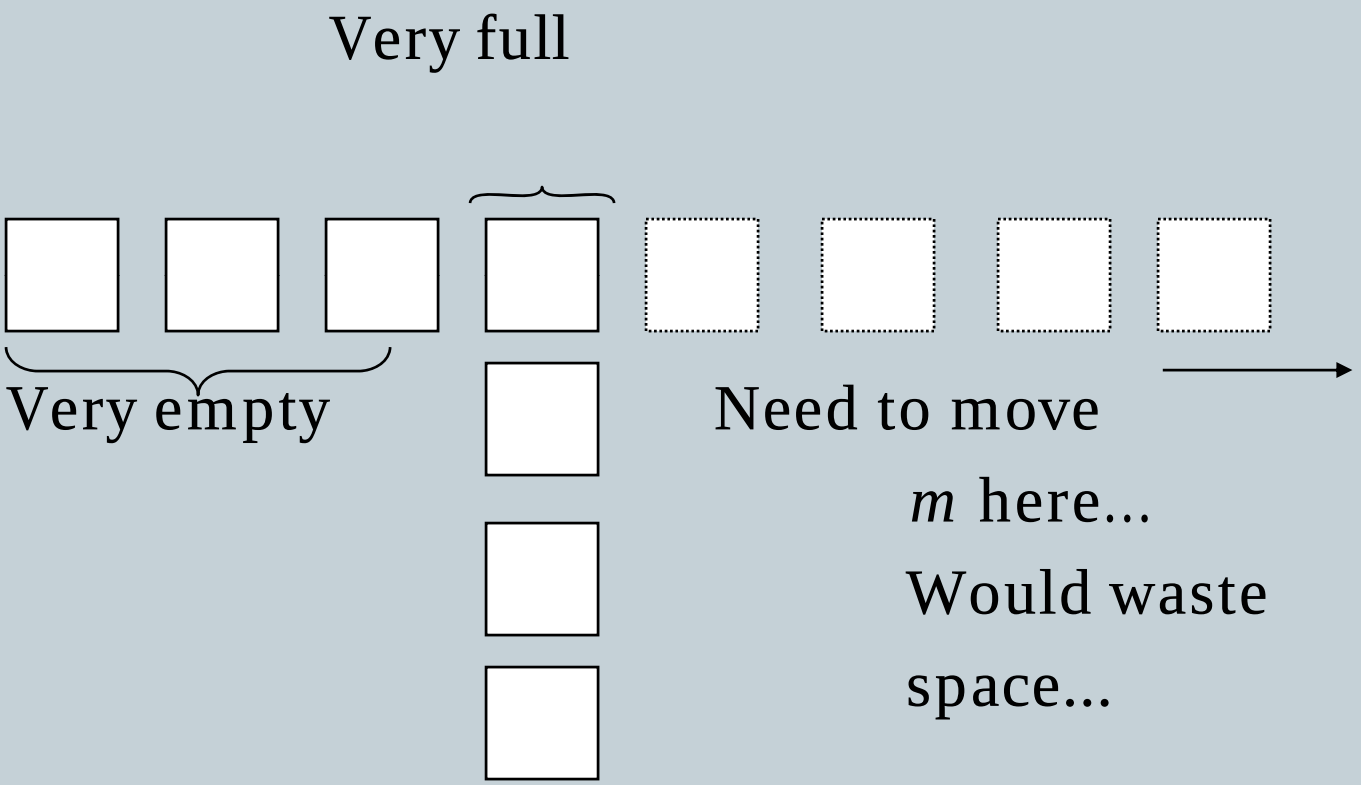
Linear Hashing

+ 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:

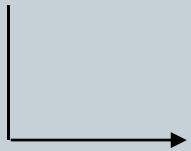
  e.g., SELECT

  FROM R

  WHERE R.A > 5

Index definition in SQL

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



defines candidate key

- Drop INDEX name

SQL

CANNOT SPECIFY TYPE OF INDEX

(e.g. B-tree, Hashing, …)

OR PARAMETERS

(e.g. Load Factor, Size of Hash,…)

ATTRIBUTE LIST ⇒ MULTIKEY INDEX

(next)

e.g., CREATE INDEX foo ON R(A,B,C)

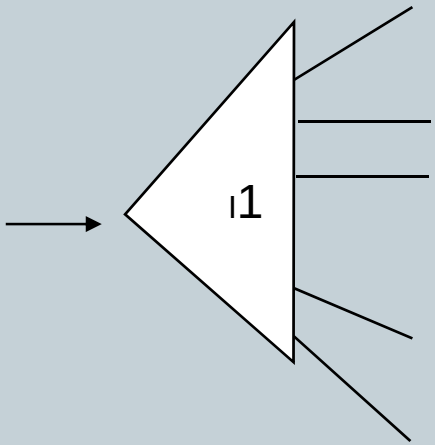
Multi-key Index

Motivation: Find records where

DEPT = “Toy” AND SAL > 50k

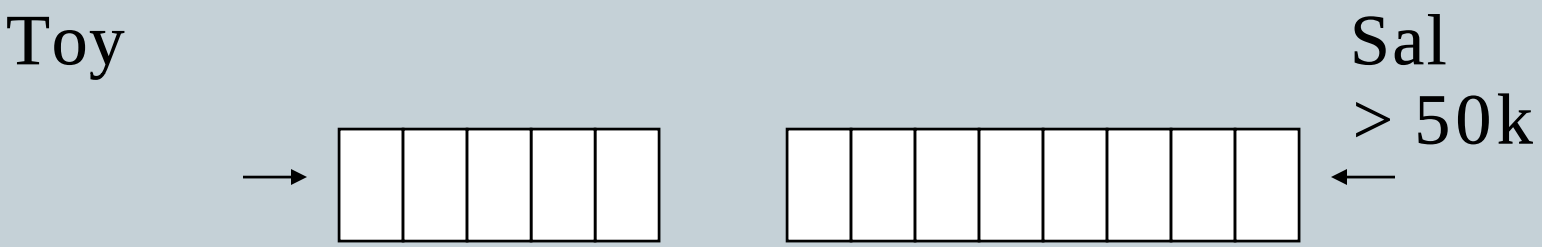
Strategy I:

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



Strategy II:

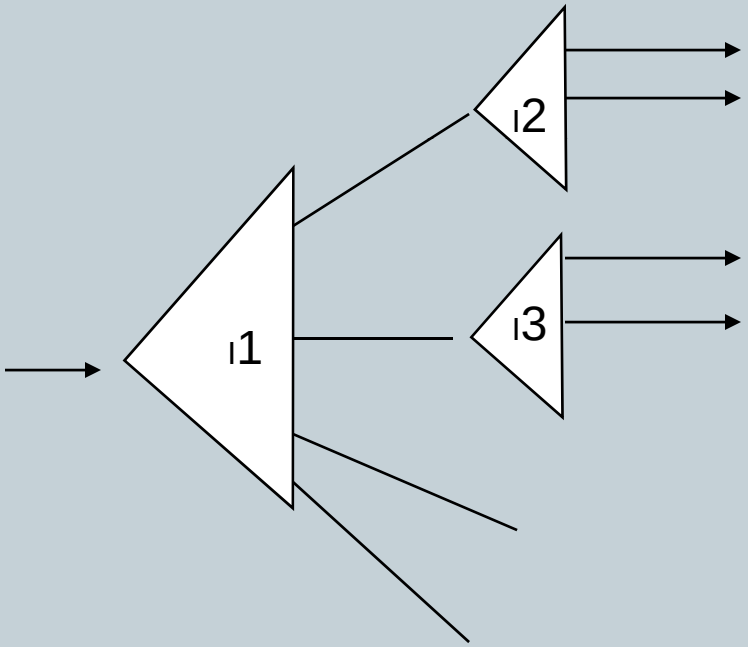
- Use 2 Indexes; Manipulate Pointers



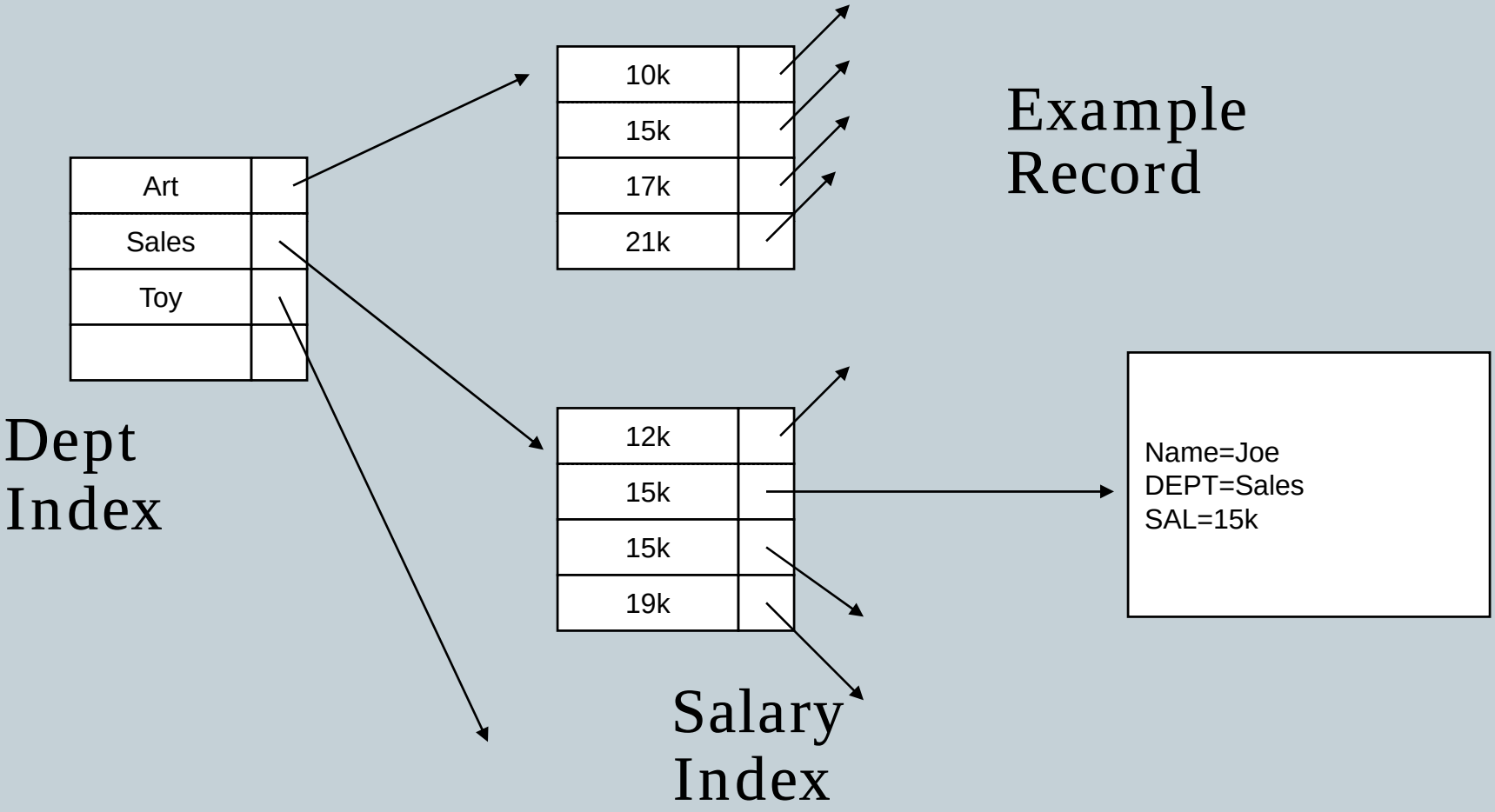
Strategy III:

- Multiple Key Index

One idea:



Example

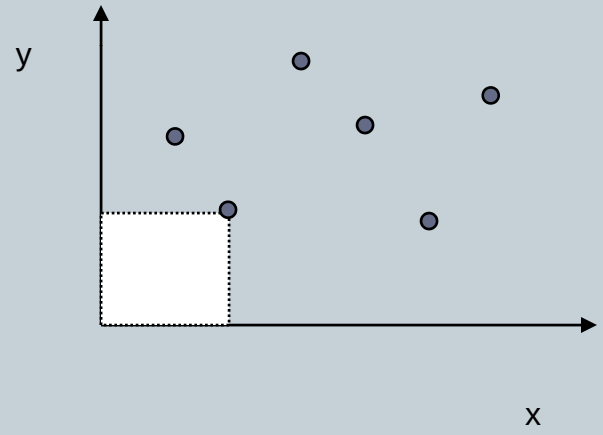

Art
Sales
Toy
Dept Index
10k
15k
17k
21k
12k
15k
15k
19k
Salary Index
Example Record
Name=Joe
DEPT=Sales
SAL=15k

For which queries is this index good?

- ☐ Find RECs Dept = “Sales” ∧ SAL=20k
- ☐ Find RECs Dept = “Sales” ∧ SAL ≥ 20k
- ☐ Find RECs Dept = “Sales”
- ☐ Find RECs SAL = 20k

Interesting application:

- Geographic Data



DATA:

<X1,Y1, Attributes>

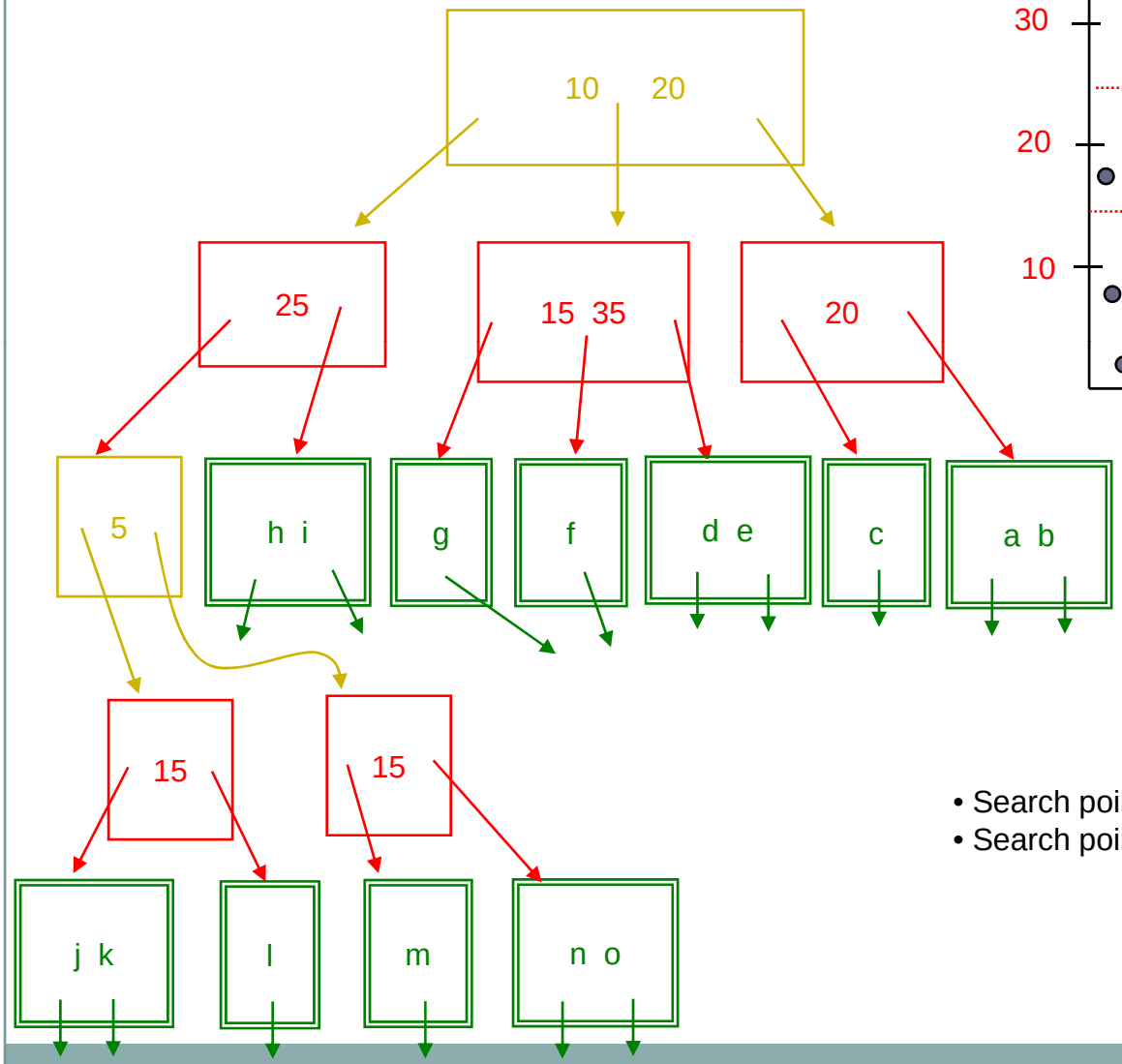
<X2,Y2, Attributes>

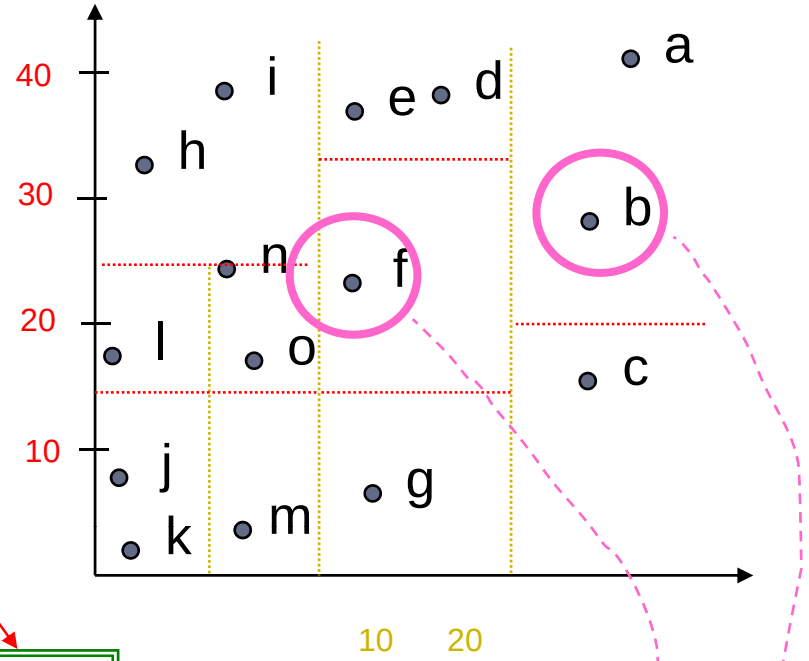
⋮

Queries:

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

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>

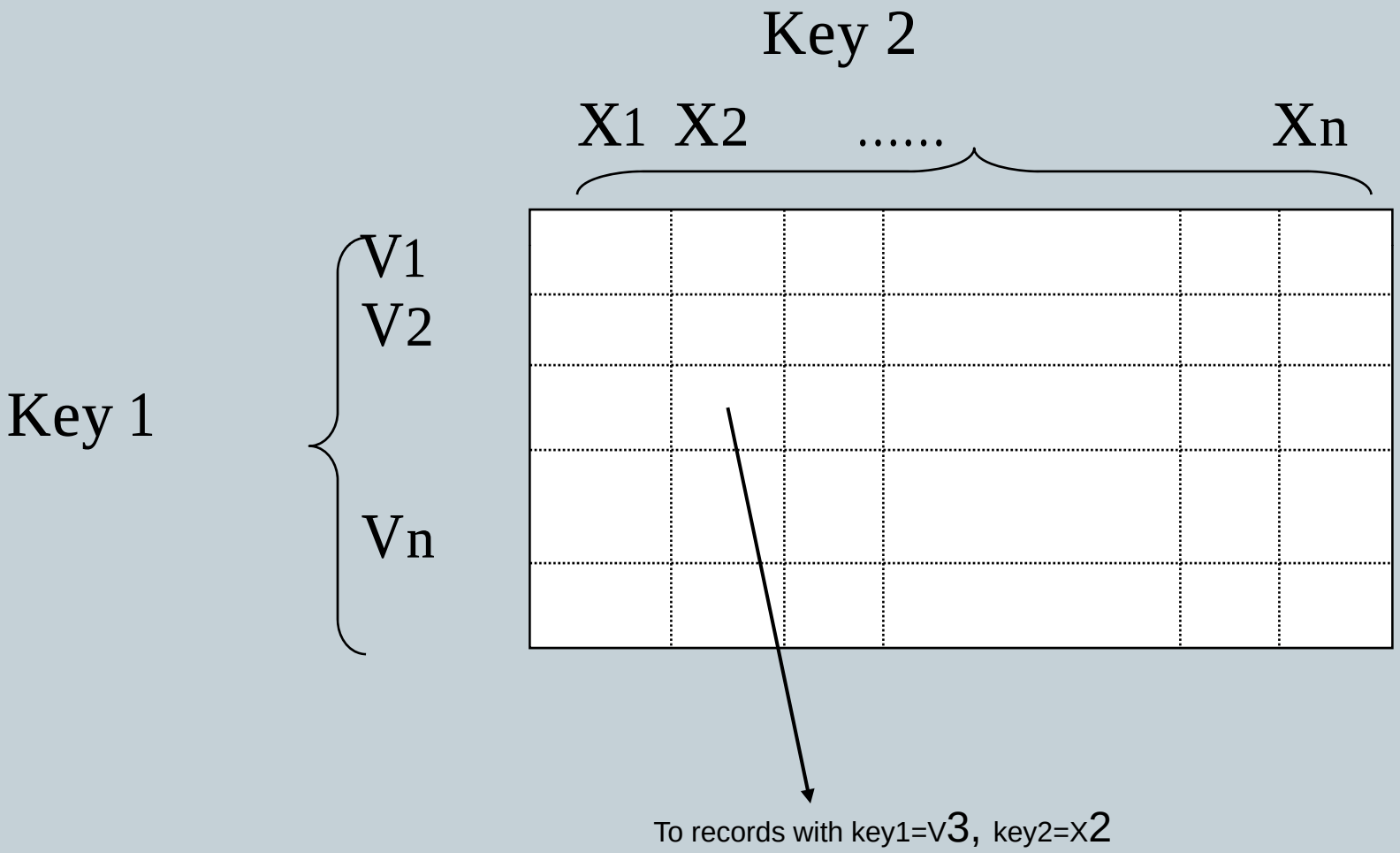
Geographic index

- 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

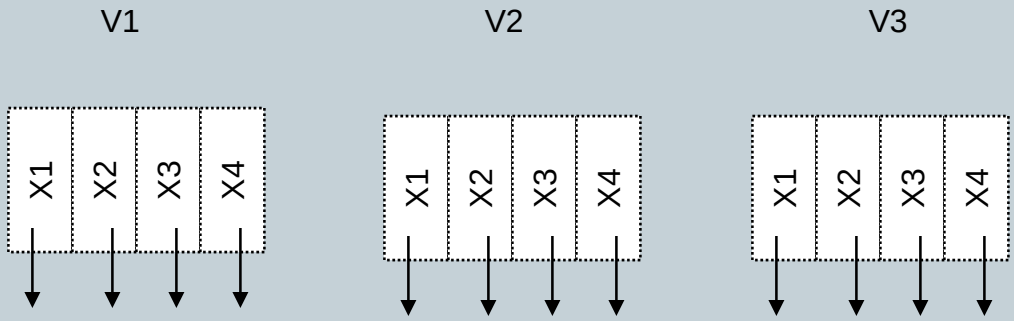
- Can quickly find records with
  - key 1 = V_i $\wedge$ Key 2 = X_j
  - key 1 = V_i
  - key 2 = X_j

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

But there is a catch with Grid Indexes!

- How is Grid Index stored on disk?

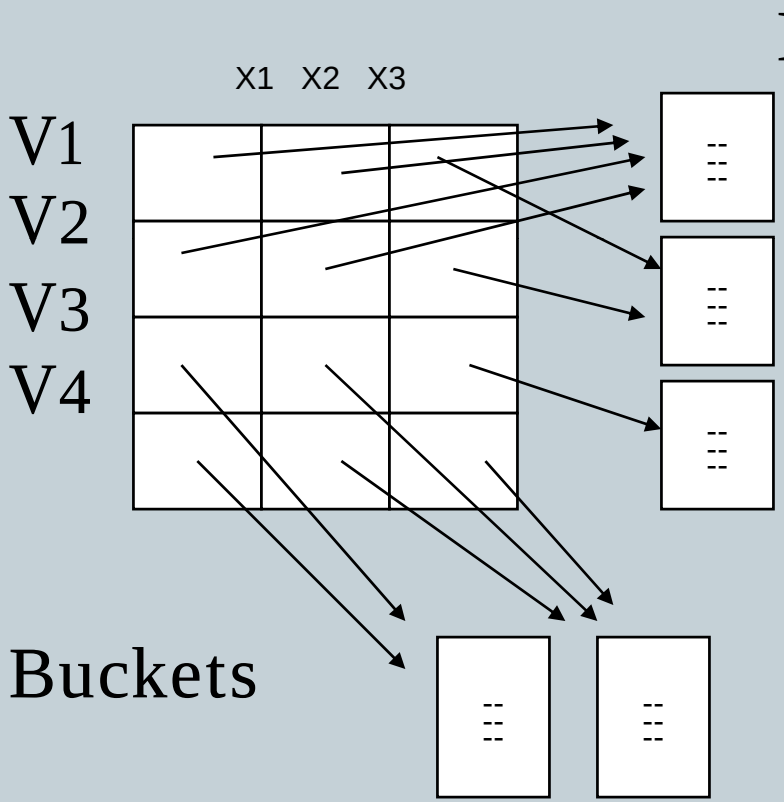
Like Array...



Problem:

- Need regularity so we can compute position of <Vi,Xj> entry

Solution: Use Indirection



*Grid only contains pointers to buckets

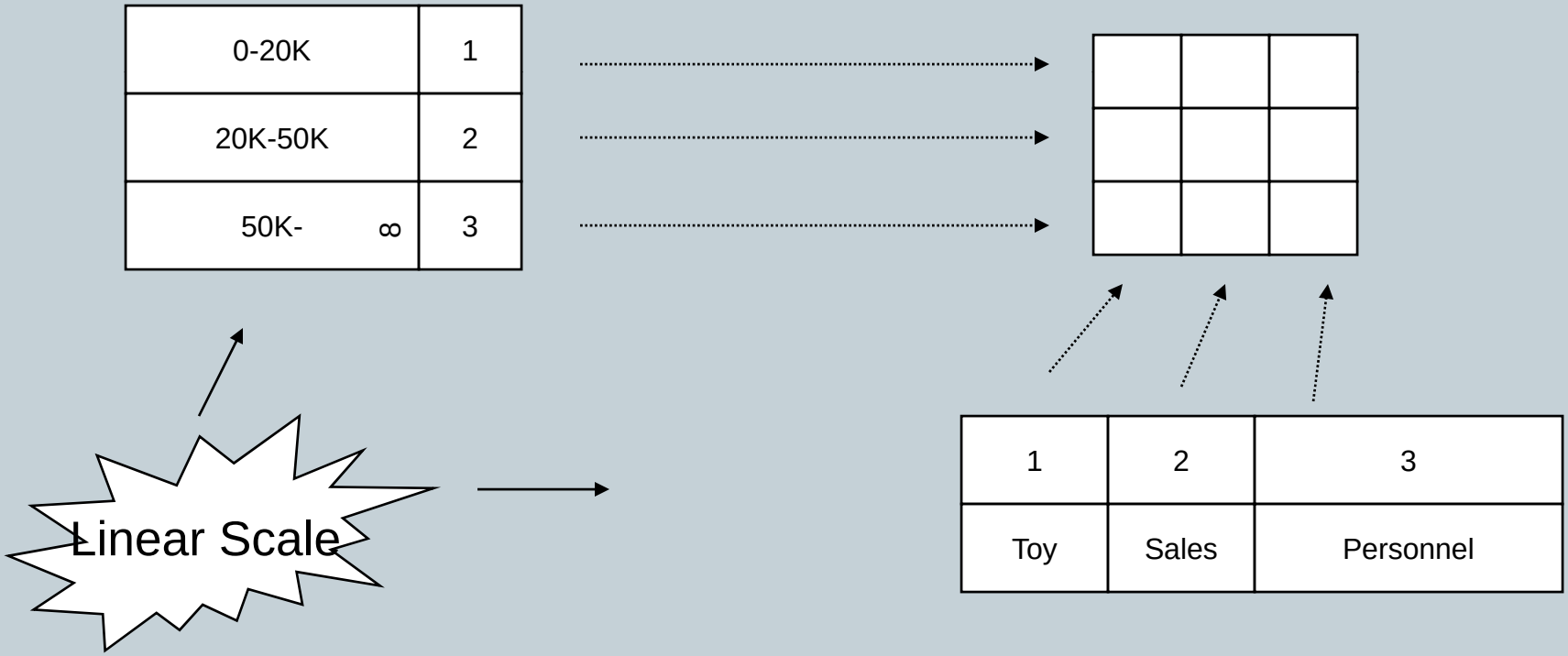
With indirection:

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

Can also index grid on value ranges

Salary

Grid

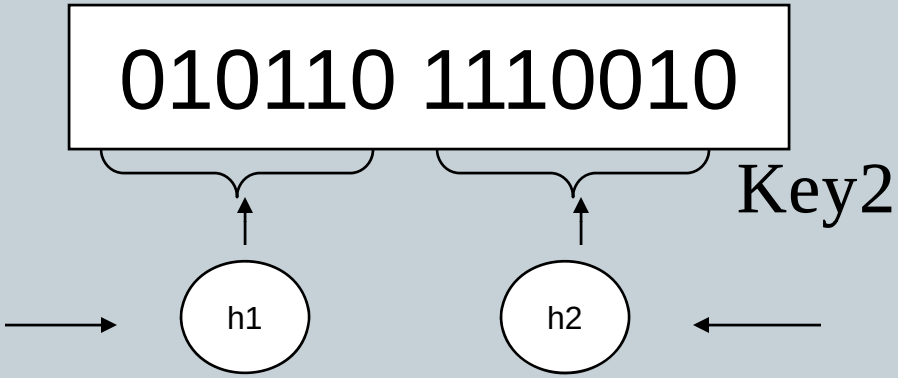


Grid files

- (+) Good for multiple-key search
- (-) Space, management overhead (nothing is free)
- (-) Need partitioning ranges that evenly split keys

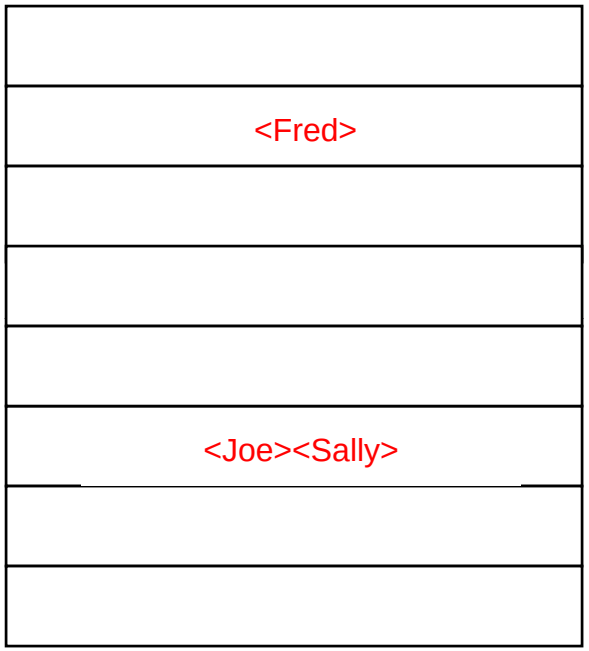
Partitioned hash function

Idea:



h1(toy) =0 000
h1(sales) =1 001
h1(art) =1 010
011
.
.

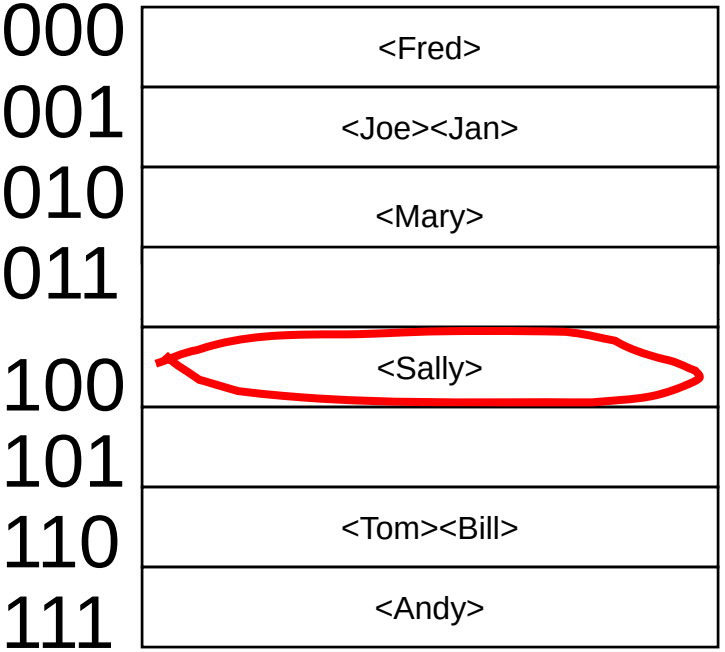
h2(10k) =01 100
h2(20k) =11 101
h2(30k) =01 110
h2(40k) =00 111
.
.



Insert

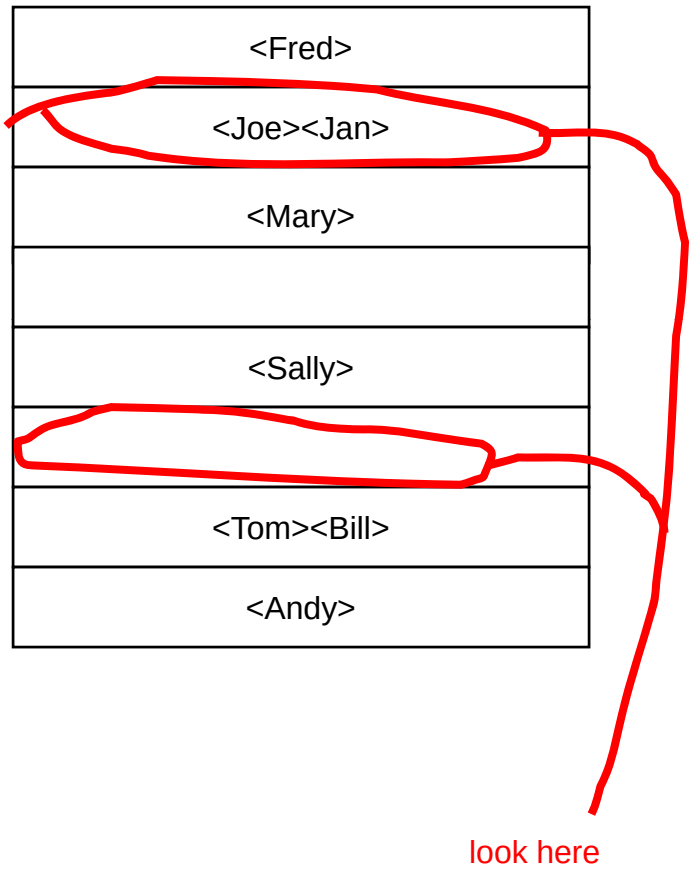
<Fred,toy,10k>,<Joe,sales,10k>
<Sally,art,30k>

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



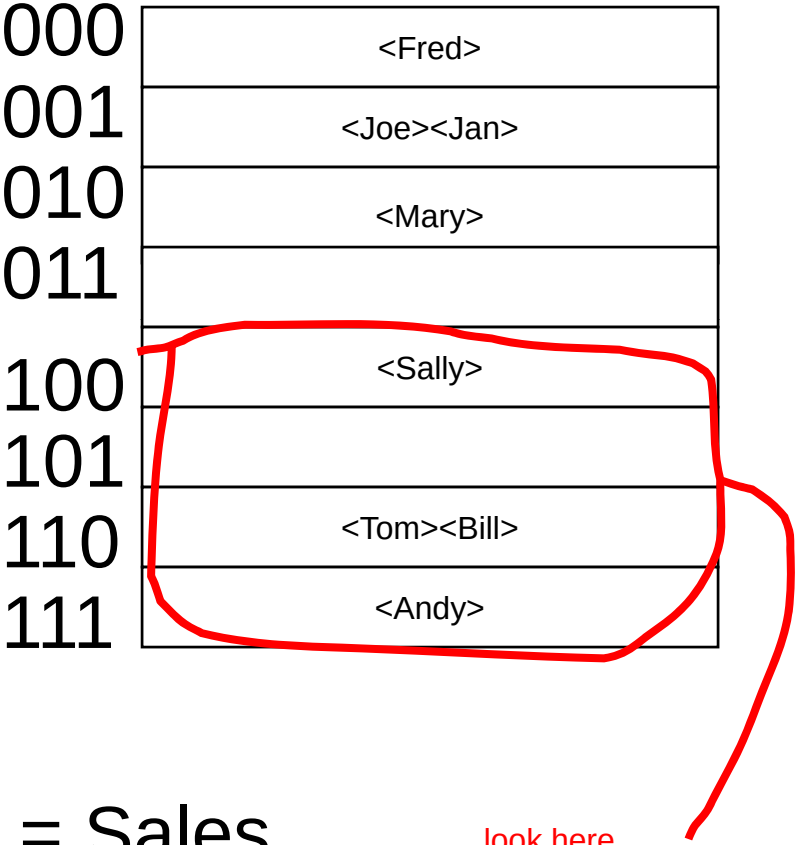
- Find Emp. with Dept. = Sales $\wedge$ Sal=40k

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



- Find Emp. with Sal=30k

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

Summary

Post hashing discussion:

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