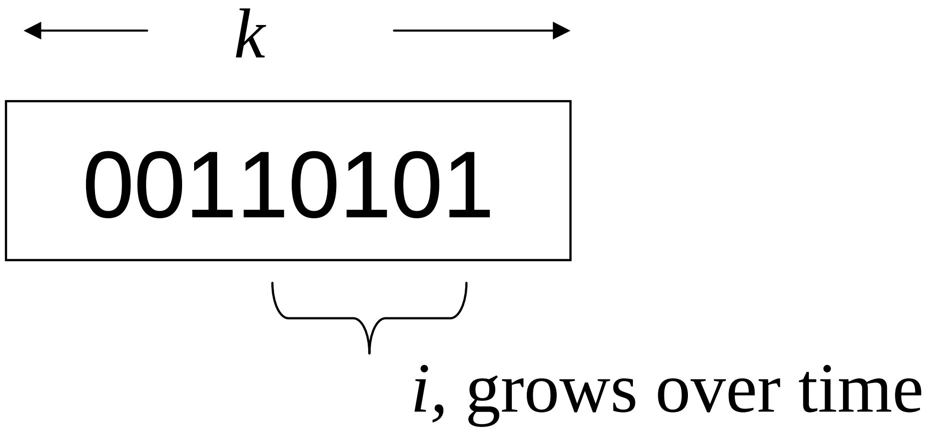
Data Storage and Query Answering

Indexing and Hashing (5)

Linear Hash Tables

Introduction

- No directory.
- Hash function computes sequences of k bits. Take only the i last of these bits and interpret them as bucket number m.



- n: *number of last bucket*, first number is 0.

Linear Hash Tables

Insertions

- If m <= n, store record in bucket m. Otherwise, store it in bucket number $m - 2^{i-1}$
- If bucket overflows, add overflow block.
- If *space utilization* becomes too high, add one bucket at the end and increment n by 1.

$$space\ utilization = \frac{r}{(n+1) \cdot c}, \text{ where } r = \text{total number of records}$$

$$\text{and } c = \text{bucket capacity (number records)}$$

→ file grows linearly

Linear Hash Tables

Insertions

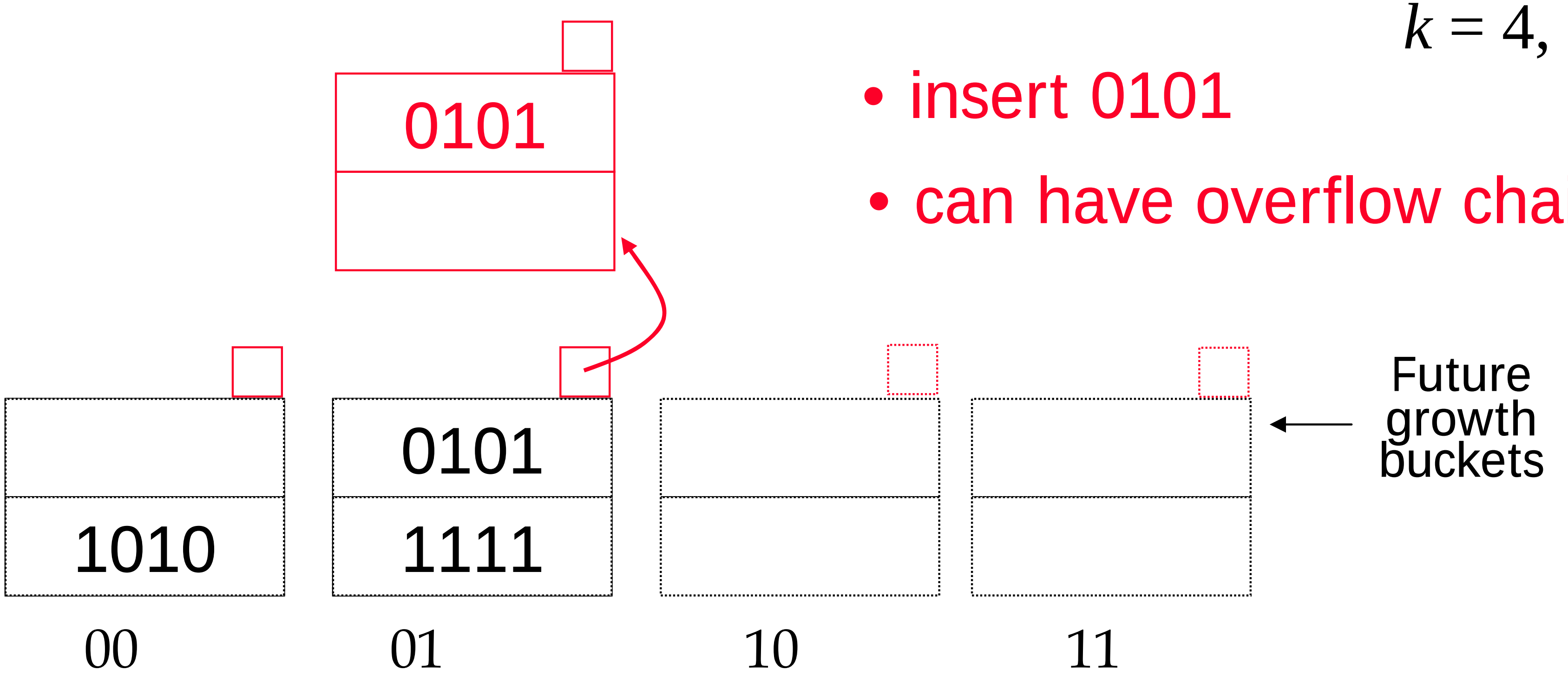
- Bucket we add is usually not in the range of hash keys where an overflow occurred.
- When $n > 2^i$, increment i by 1.
- i is the number of *rounds* of doubling the size of the Linear Hash table.
- No need to move entries.

Linear Hash Tables

Example

$k = 4,\ i = 2$

- insert 0101
- can have overflow chains!



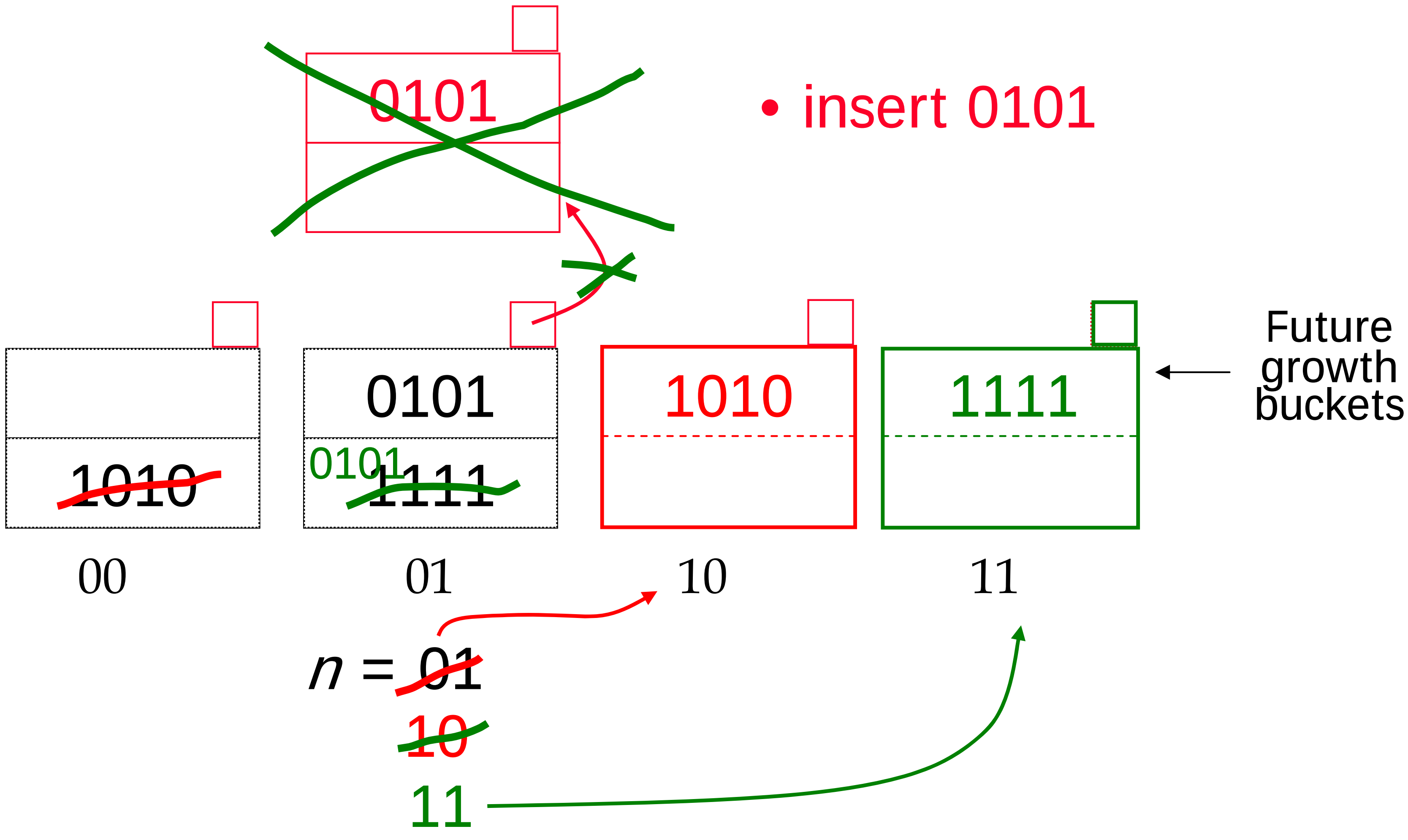
n = 01

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

Linear Hash Tables

Example

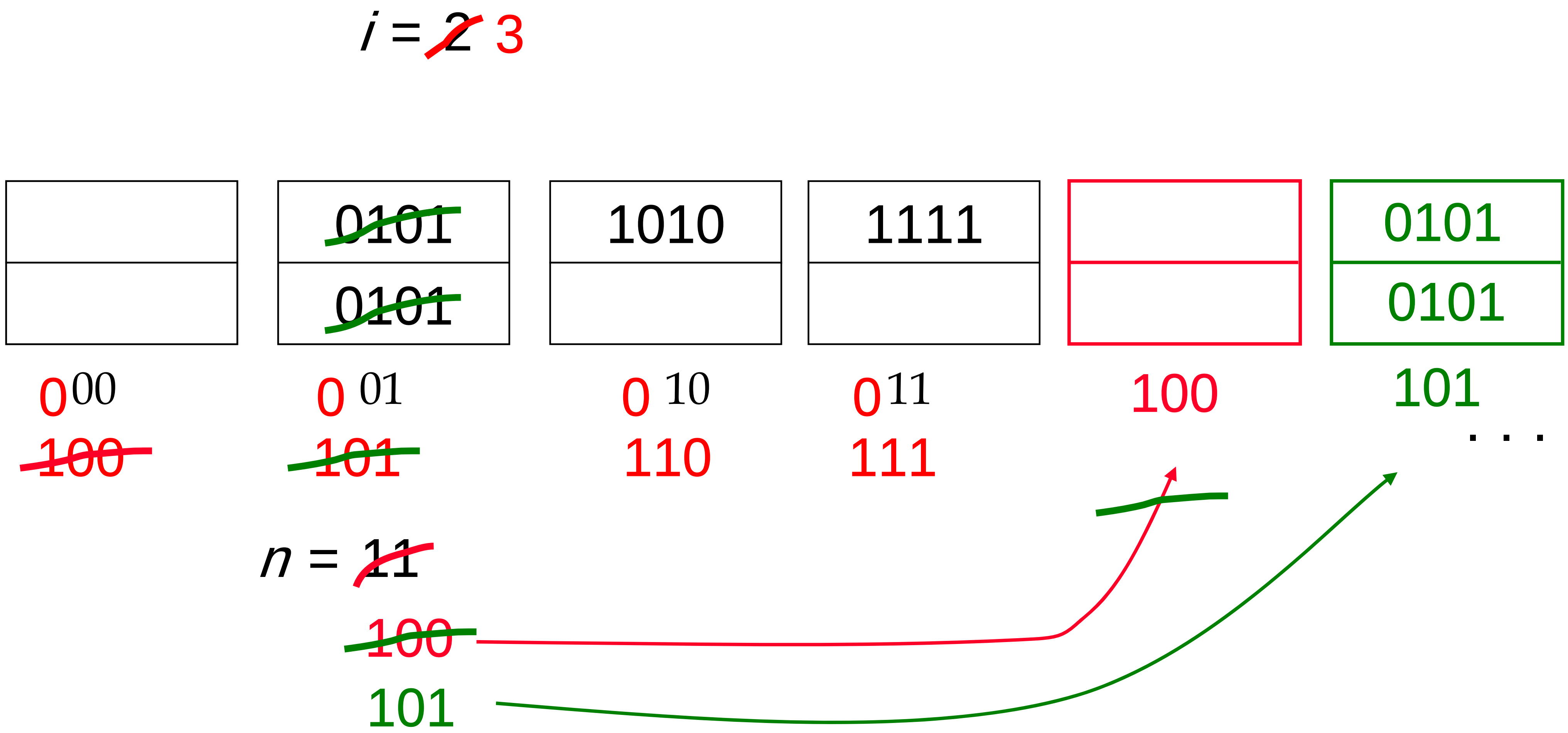
$k = 4, i = 2$



Linear Hash Tables

Example

$k = 4$

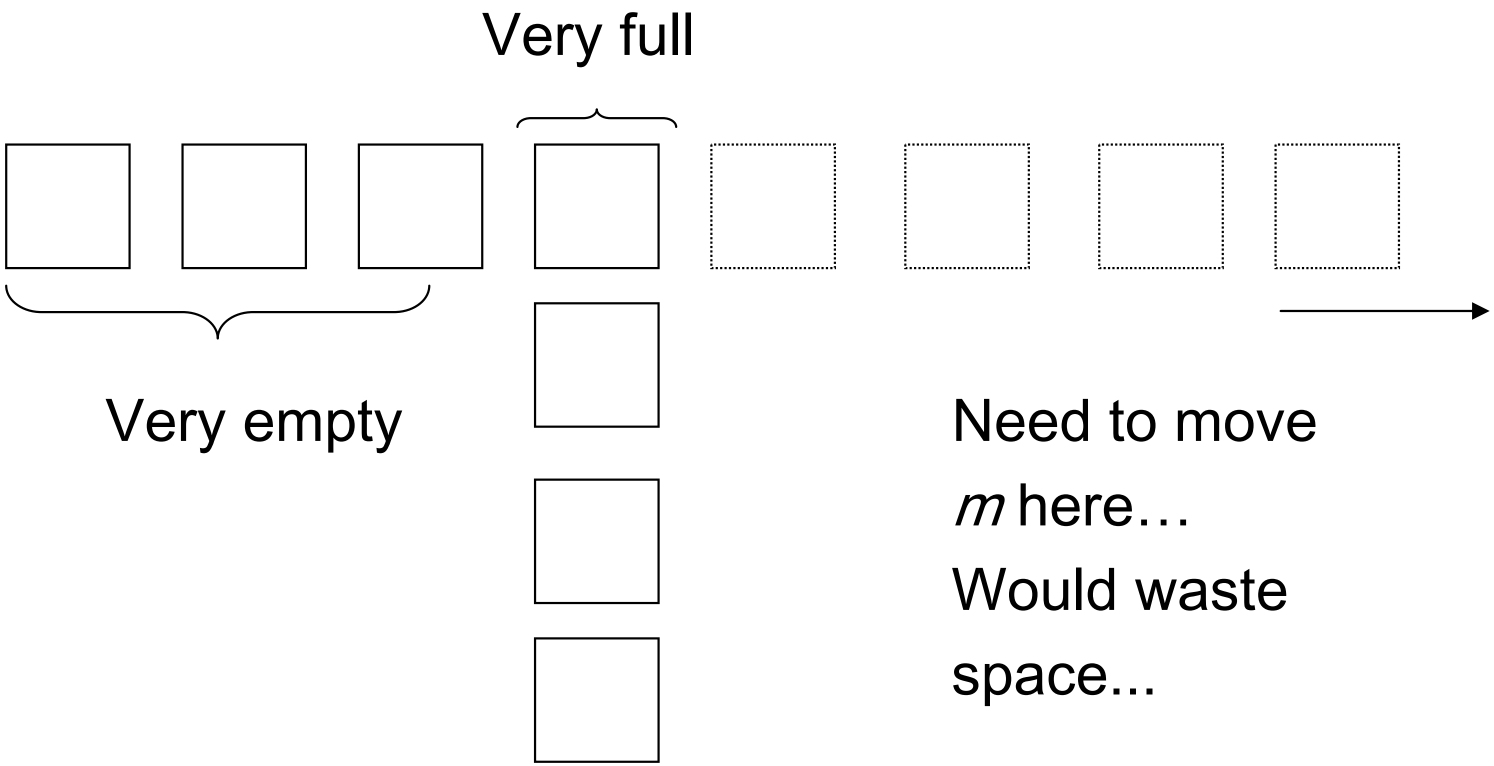


Linear Hash Tables

Discussion

- Can manage growing number of buckets without wasting too much space.
- No directory, i.e. no indirection in access and no expensive doubling operation.
- Significant need for overflow chains, even if no duplicates among last i bits of hash values.

Example: BAD CASE



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

Index Definition in SQL

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

... at least in SQL...

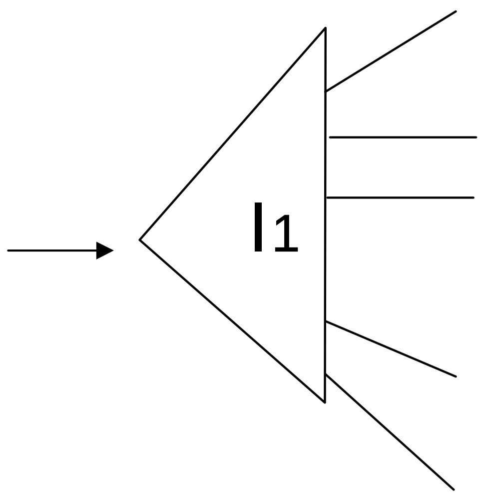
Multi-Key Index

Motivation: Find records where

DEPT = “Toy” AND SAL > 50k

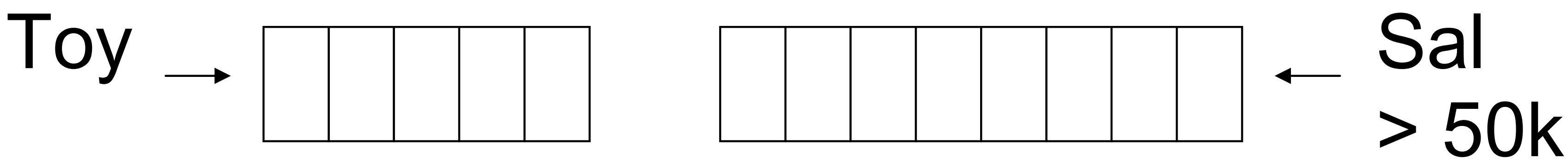
Strategy I

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



Strategy II

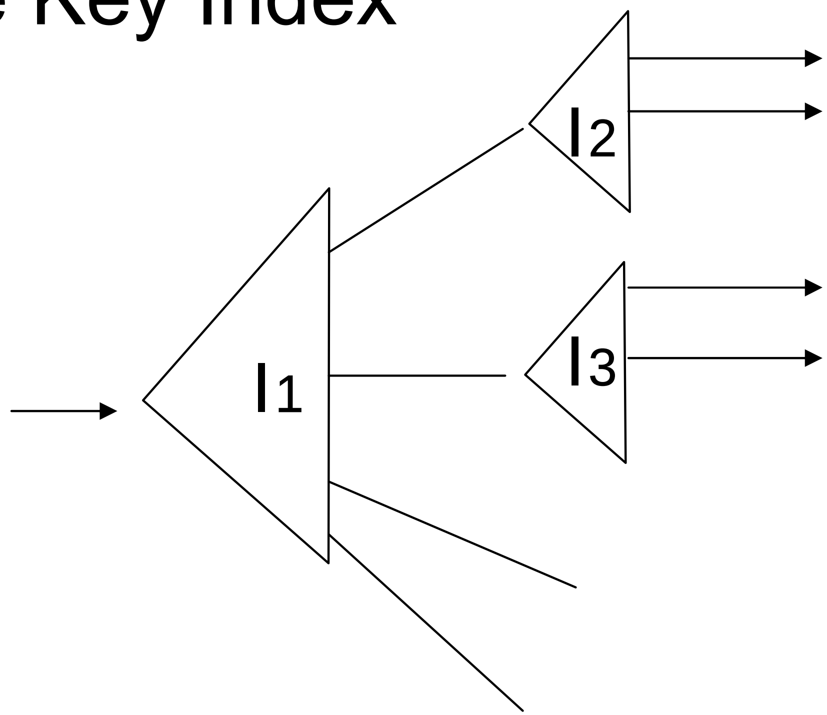
- Use 2 Indexes; Manipulate Pointers



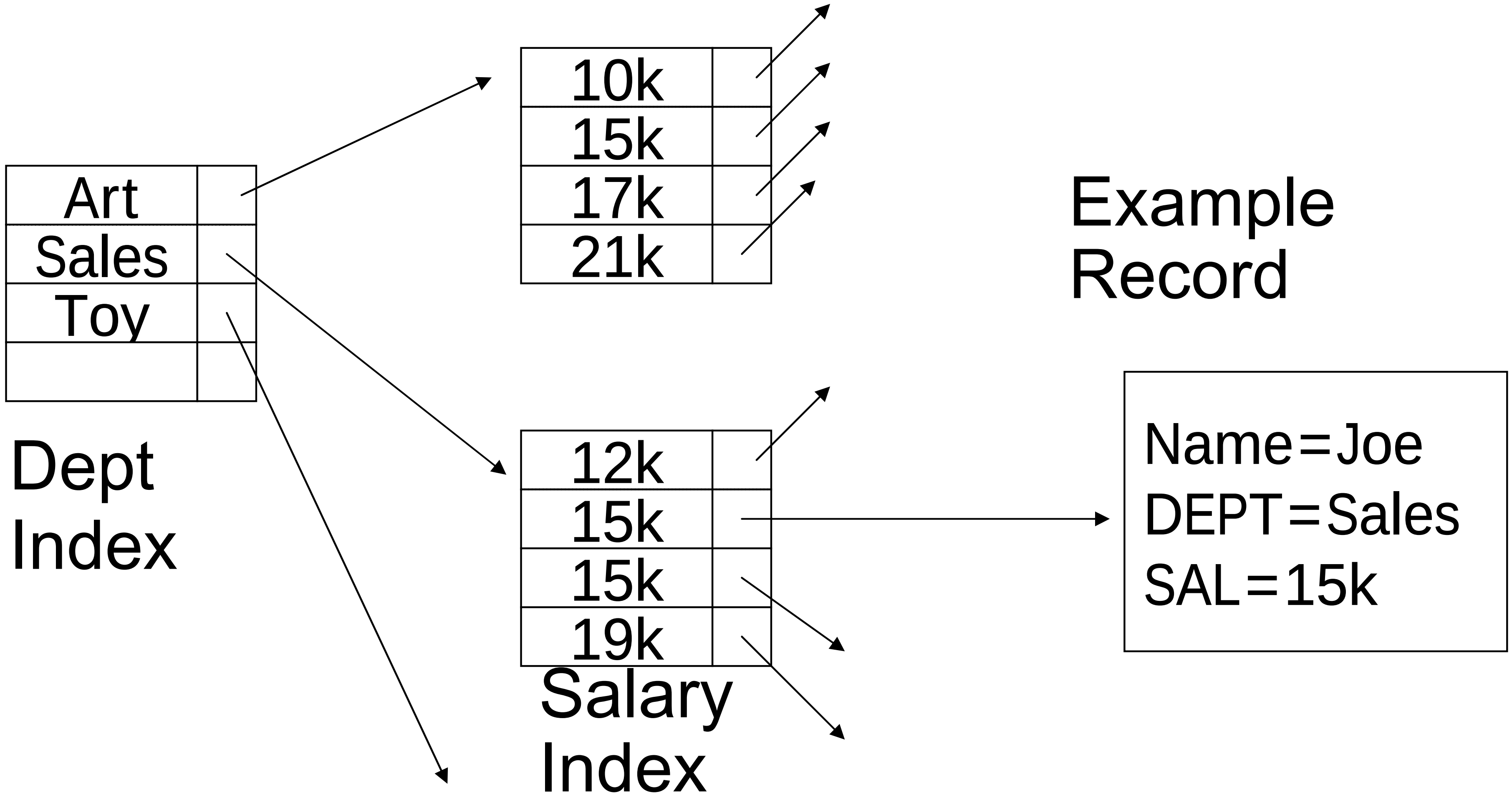
Strategy III

- Multiple Key Index

One idea:



Example



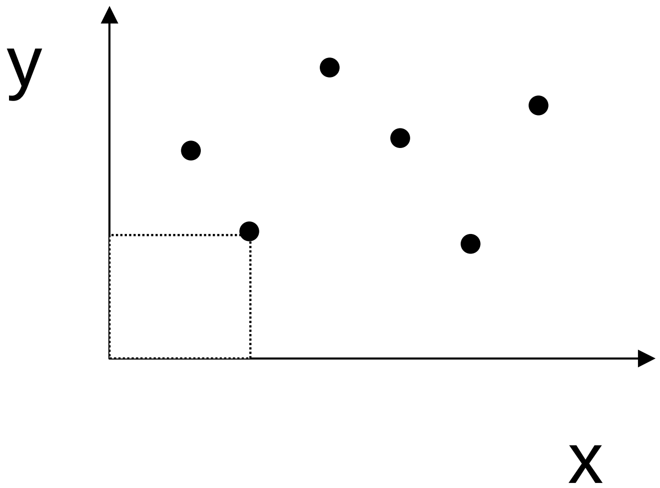
Art
Sales
Toy
Dept Index
10k
15k
17k
21k
12k
15k
15k
19k
Salary Index
Example Record
Name=Joe
DEPT=Sales
SAL=15k

For Which Queries is This Index Good?

- Find RECs Dept = “Sales” $\wedge$ SAL=20k
- Find RECs Dept = “Sales” $\wedge$ SAL > 20k
- Find RECs Dept = “Sales”
- Find RECs SAL = 20k

Interesting Applications

- Geographic Data



DATA:

<X_1,Y_1, Attributes>

<X_2,Y_2, Attributes>

⋮

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

Comments

- Many types of geographic index structures have been suggested
  - Kd-Trees (very similar to what we described here)
  - Quad Trees
  - R Trees
  - ...
- To be discussed in the topics of “advanced queries”.