

## Unit 6

# File Organization and Access Methods

# Contents

---

- ❑ 6.1 Introduction
- ❑ 6.2 Indexing
- ❑ 6.3 Hashing
- ❑ 6.4 Pointer Chains
- ❑ 6.5 Compression Techniques
- ❑ 6.6 Differential File Organization

# 大量資料存取方法之研究

## **Approaches to Access/Store Large Data**

---

楊維邦 博士

國立交通大學  
資訊科學系所教授

# **6.1 Introduction**

---

# The Role of Access Method in DBMS

## Query in SQL:

```
SELECT CUSTOMER. NAME  
FROM CUSTOMER, INVOICE  
WHERE REGION = 'N.Y.' AND  
AMOUNT > 10000 AND  
CUSTOMER.C#=INVOICE.C#
```

## Internal Form:

$$\Pi(\sigma(S \bowtie SP))$$

## Operator:

```
SCAN C using region index, create C  
SCAN I using amount index, create I  
SORT C?and I?on C#  
JOIN C?and I?on C#  
EXTRACT name field
```

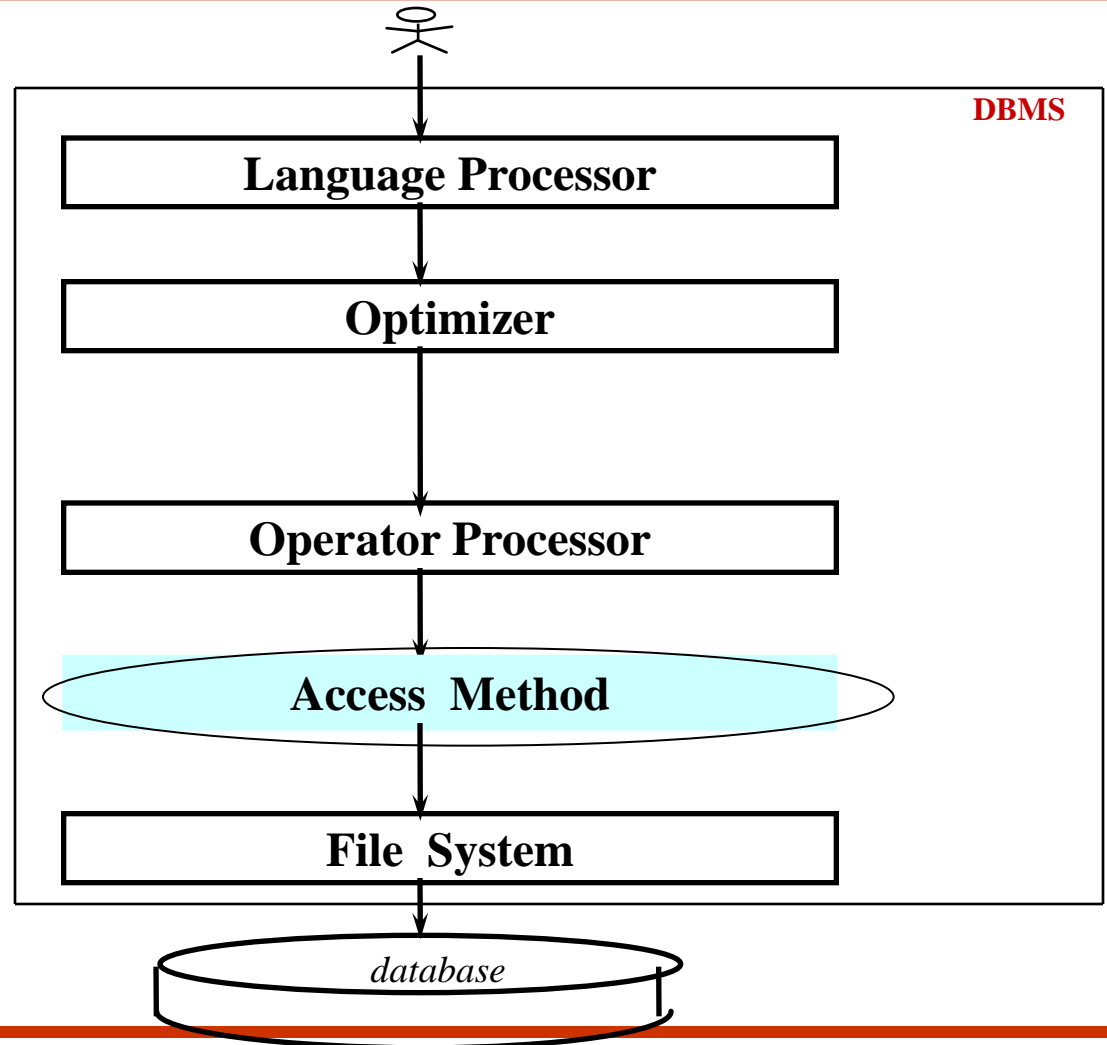
## Calls to Access Method:

```
OPEN SCAN on C with region index  
GET next tuple
```

⋮

## Calls to file system:

```
GET10th to 25th bytes from  
block #6 of file #5
```



# The Internal Level

---

## ■ Objectives:

- concern the way the data is actually stored.
- store data on direct access media. e.g. disk.
- minimize the number of disk access (disk I/O).
- disk access is much slower than main storage access time.

## ■ Storage Structure/File Structure:

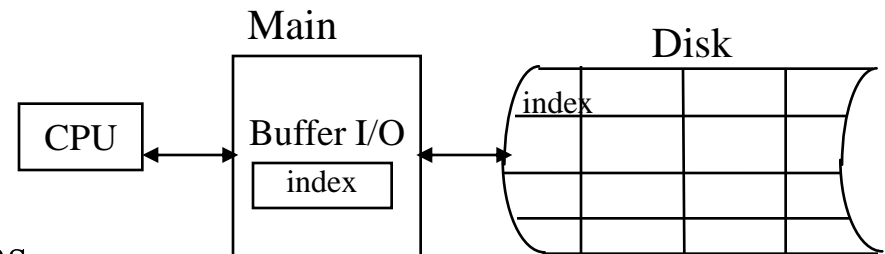
- many different storage structures:

<e.g> indexing, hashing, pointer chains, ...

- different structures have different performance

=> no single structure is optimal for all applications.

=> a good DBMS should support a variety of different structures (Access Methods)



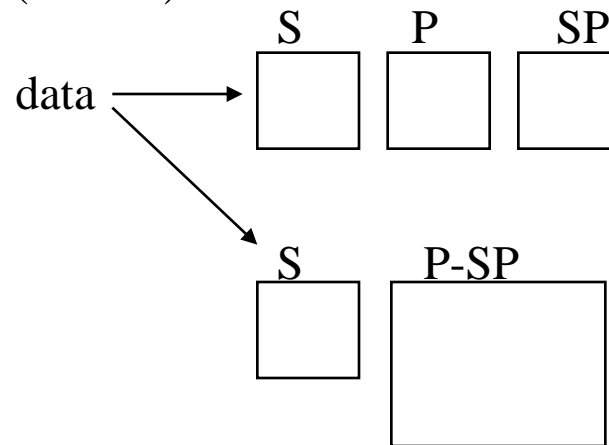
# The Internal Level (cont.)

---

- **Physical database design:**

- Process of choosing an appropriate storage representation for a given database (by DBA). E.g. designing B-tree or hashing
- Nontrivial task
  - require a good understanding of how the database will be used.

- **Logical database design:** (Unit 7)



## **6.2 Indexing**

---



# Indexing: Introduction

- Consider the Supplier table, **S**.
- Suppose "*Find all suppliers in city xxx*" is an important query. i.e. it is frequently executed.  
=> DBA might choose the stored representation as Fig. 6.2.

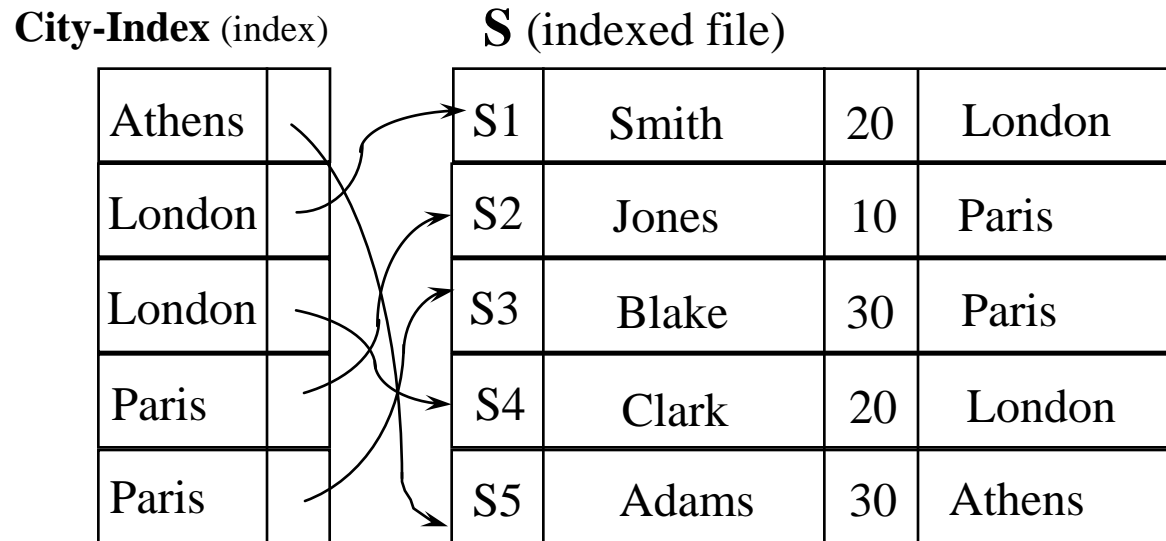


Fig. 6.2: Indexing the supplier file on CITY.

# Indexing: Introduction (cont.)

---

- Now the DBMS has two possible strategies:
  - <1> Search **S**, looking for all records with city = 'xxx'.
  - <2> Search **City-Index** for the desired entry.
  
- **Advantage:**
  - speed up retrieval.
  - index file is sorted.
  - fewer I/O's because index file is smaller.
  
- **Disadvantages:**
  - slow down updates.
  - both index and indexed file should be updated.

# Indexing: Multiple Fields

- Primary index : index on primary key. <e.g> s#
- Secondary index: index on other field. <e.g> city
- A given table may have any number of indexes.

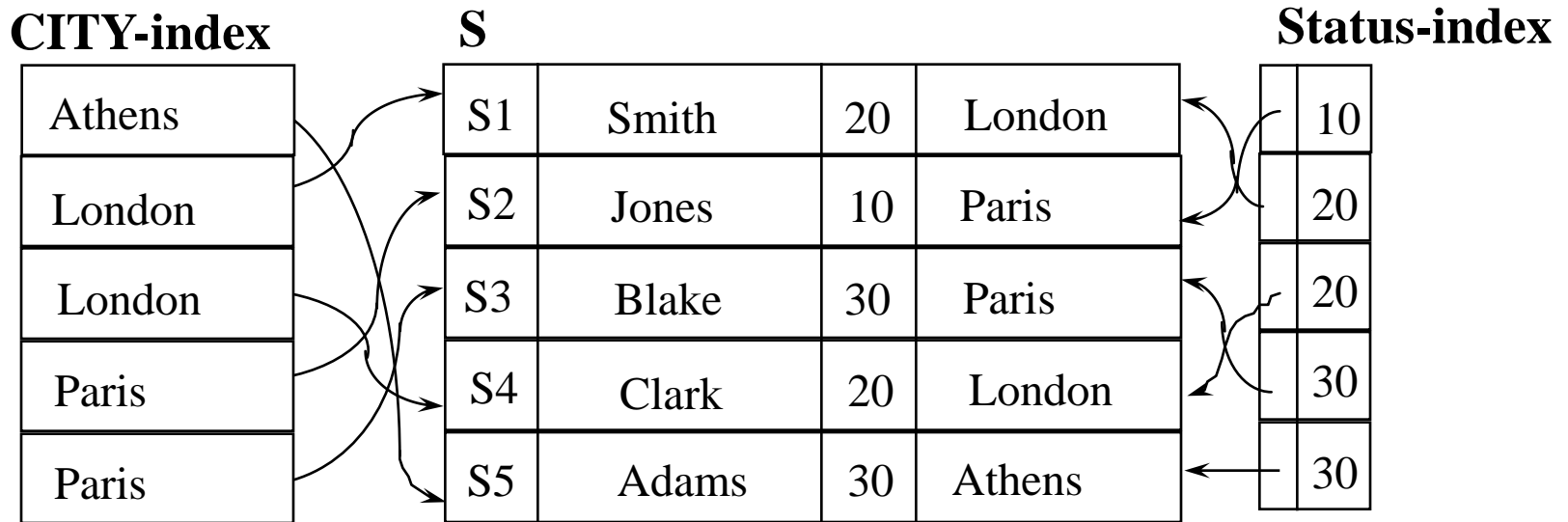
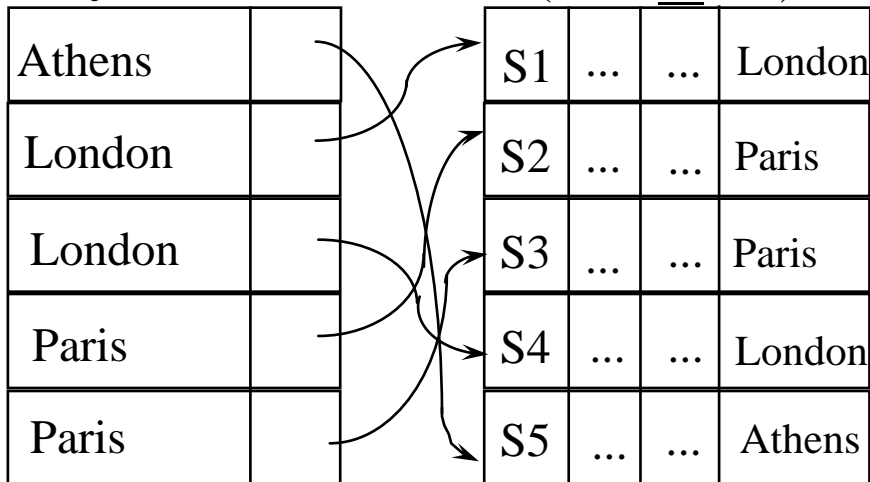


Fig. 6.3: Indexing the supplier file on both CITY and STATUS.

# How Index are used?

Consider:

## City-Index



### <1> **Sequential access :**

accessed in the sequence defined by values of the indexed field.

<e.g> Range query : "*Find the suppliers whose city begins with a letter in the range L-R.*"

### <2> **Direct Access :**

<e.g> "Find suppliers in London."

<e.g> list query: "*Find suppliers whose city is in London, Paris, and N.Y.*"

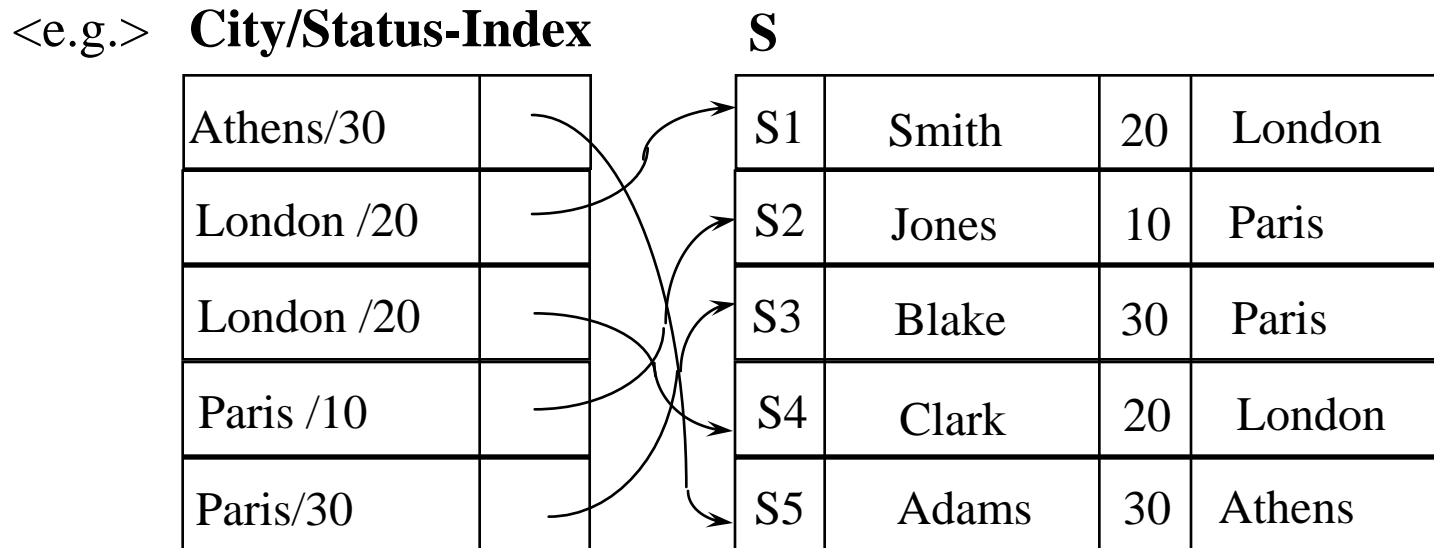
### <3> **Existence test :**

<e.g> "Is there any supplier in London ?"

Note: It can be done from the index alone.

# Indexing on Field Combinations

- To construct an index on the basis of values of two or more fields.



Query: “Find suppliers in Paris with status 30.”

- on city/status index: a single scan of a single index.
- on two separate indexes: two index scan => still difficult. (Fig. 6.3)

# Indexing on Field Combinations (cont.)

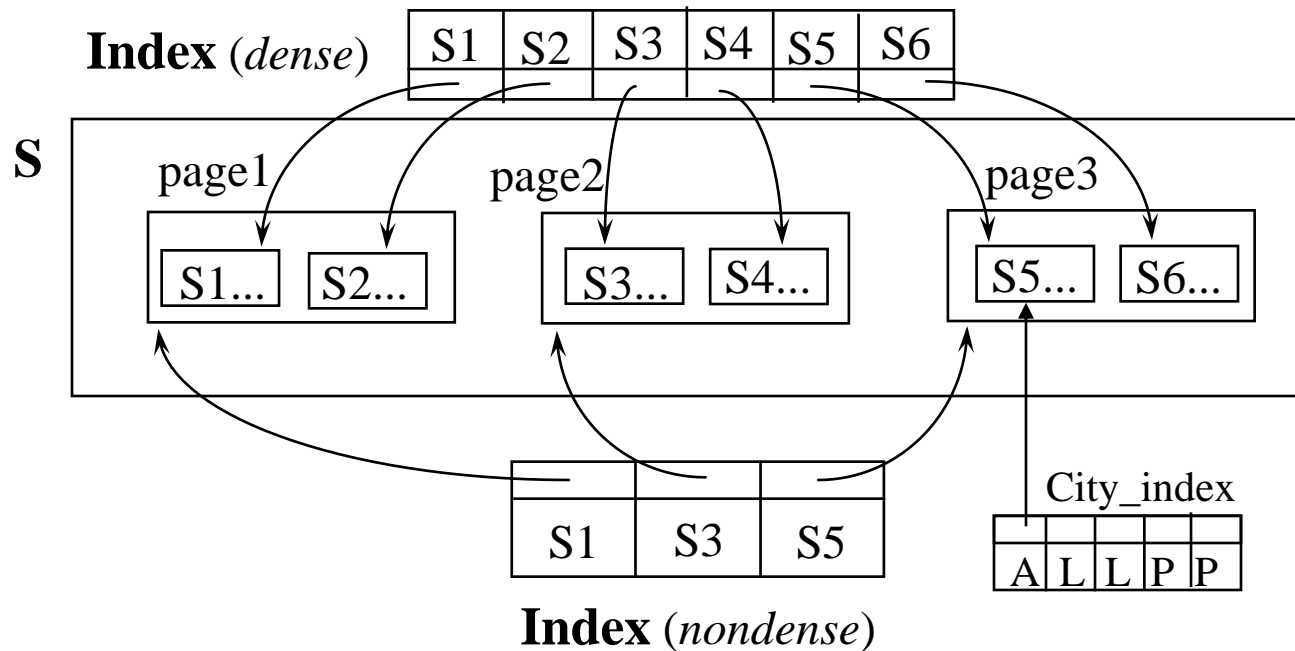
---

- <Note>
  1. Combined city/status index can serve as an index on the city field alone.
  2. In general, an index on the combination of fields F1 F2 ...Fn can serve as indexes on
    - F1 alone
    - F1 F2 or F2 F1
    - F1 F2 F3 or any combinations
    - ⋮

<Think>: How many indexes can F1...Fn serve as ?

# Dense v.s. Nondense Indexing

- Assume the **Supplier file (S)** is clustered on **S#**.



# Dense v.s. Nondense Indexing (cont.)

---

- **Nondense index:** not contain an entry for every record in the indexed file.
  - retrieval steps:
    - <1> scan the index (nondense) to get page # , say p.
    - <2> retrieve page p and scan it in main storage.
  - advantages:
    - <1> occupy less storage than a corresponding dense index
    - <2> quicker to scan.
  - disadvantages: can not perform existence test via index alone.
- **Note:** At most only one nondense index can be constructed. (why?)
- **Clustering:** logical sequence = physical sequence



# B-tree

---

- **Introduction:**

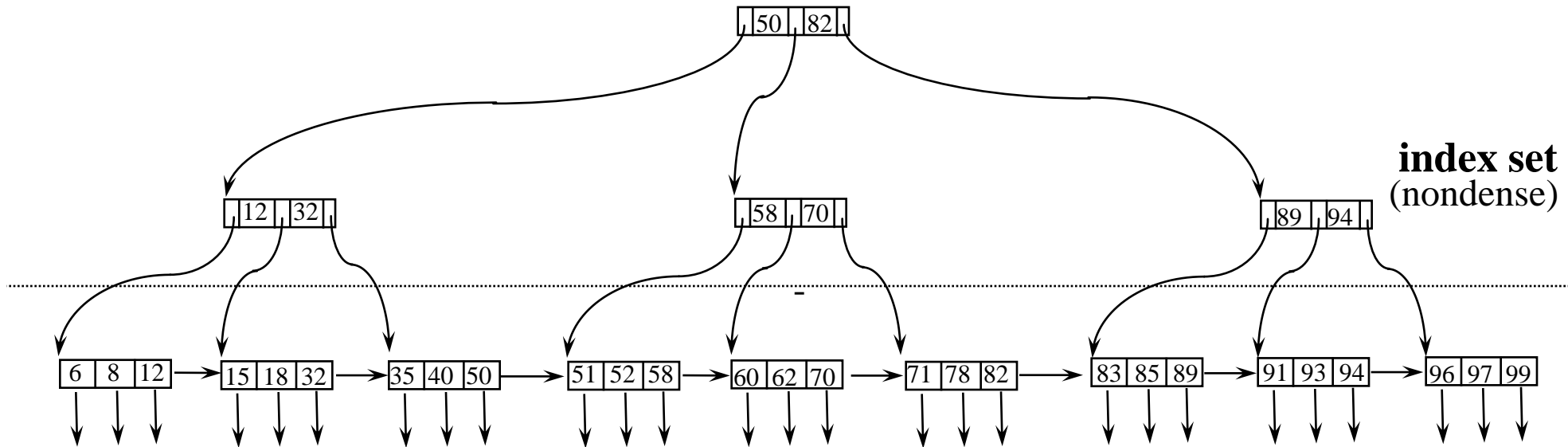
- is a particular type of multi-level (or tree structured) index.
- proposed by Bayer and McCreight in 1972.
- the commonest storage structure of all in modern DBMS.

- **Definition:** (from Horowitz "Data Structure")

- A **B-tree T** of order **m** is an **m-way** search tree, such that
- <1> the root node has at least 2 children.
  - <2> non-leaf nodes have at least  $\lceil m/2 \rceil$  children.
  - <3> all leaf nodes are at the same level.

- **Goal:** maintain balance of index tree by dynamically restructuring the tree as updates proceed.

# B<sup>+</sup>-tree (Knuth's variation)



**index set**  
(nondense)

**Sequence set**  
(with pointers  
to data records)  
(dense or nondense)

- index set: provides fast direct access to the sequential set and thus to the data too.
- sequence set: provides fast sequential access to the indexed data.

- Other variations: B\*-tree, B'-tree,...

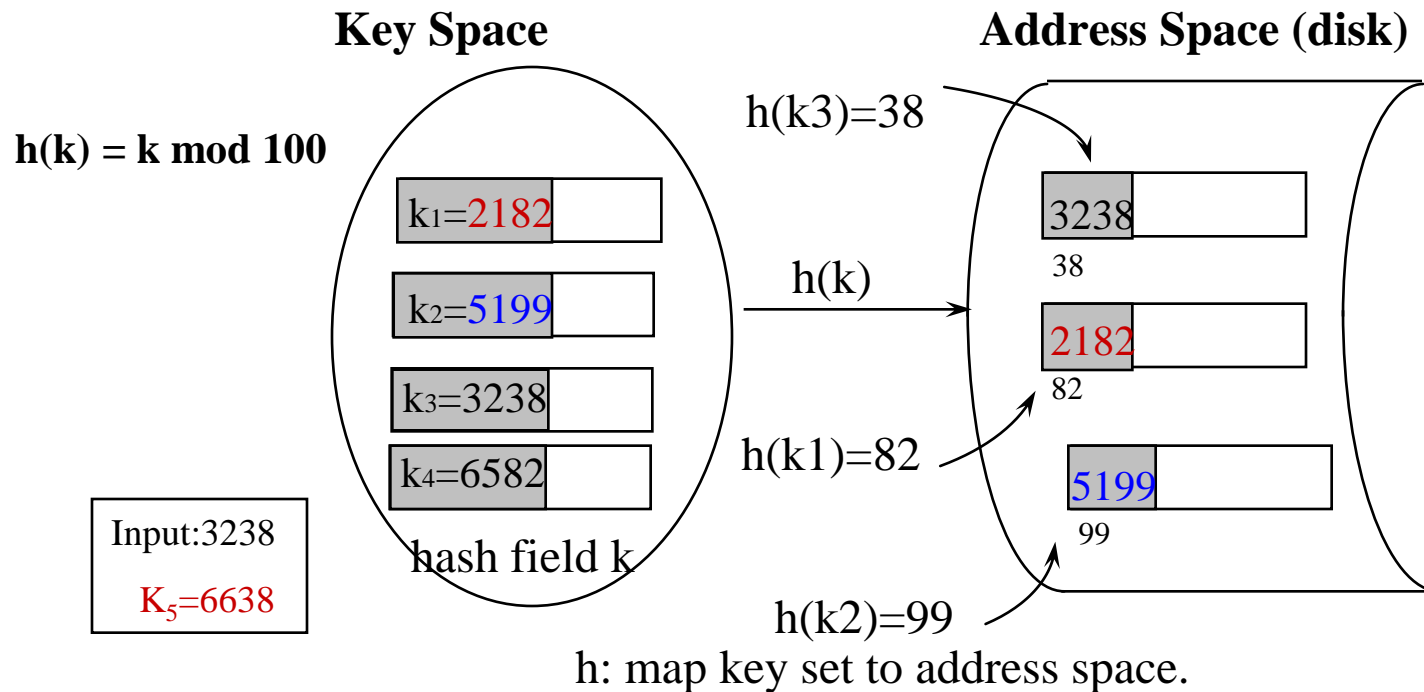
# 6.3 Hashing

---

# Hashing: Introduction

## ■ Hashing (or Hash Addressing )

- is a technique for providing fast direct access to a specific stored record on the basis of a given value for some fields.
- The field is usually but not necessarily the primary key



# Hashing: Introduction (cont.)

---

## ■ Basic Idea:

- Apply key-to-address transformation to determine in which bucket a record should be placed.
- partition storage space into buckets, each holds one or more records.
- handle bucket overflow

## ■ to store:

- DBMS computes the hash address (RID or page #) for the new record.
- DBMS instructs the file manager to place the record at that position.

## ■ to retrieve:

- given a key, the DBMS computes the hash address as before
- Using the computed address, DBMS instructs the file manager to fetch the record.

## ■ Advantages:

- fast.
- no space overhead as index method

## ■ Disadvantages:

- physical sequence  $\neq$  primary key sequence.
- Collisions:  $f(k_1) = f(k_2)$ ,  $k_1 \neq k_2$

# Address Transformation Algorithms

- **Convert key value into value of appropriate magnitude.**

<e.g> 'Smith' => Asc('s') + Asc('m') + Asc('i') + Asc('t') + Asc('h')

- **Common algorithms:**

- Division Method:

$$H(k) = \text{modulo } (k / n)$$

e.g.  $H(k) = k \bmod 100$

- Mid-square method:

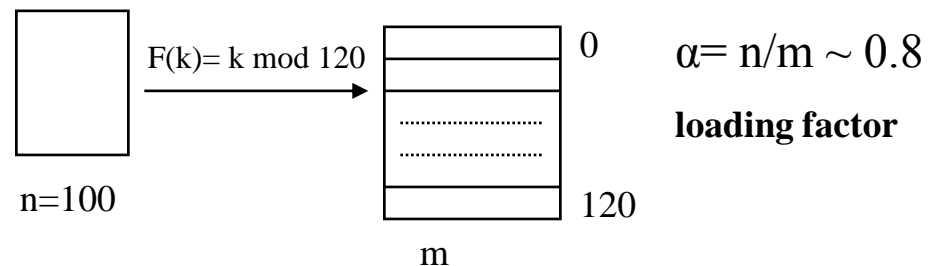
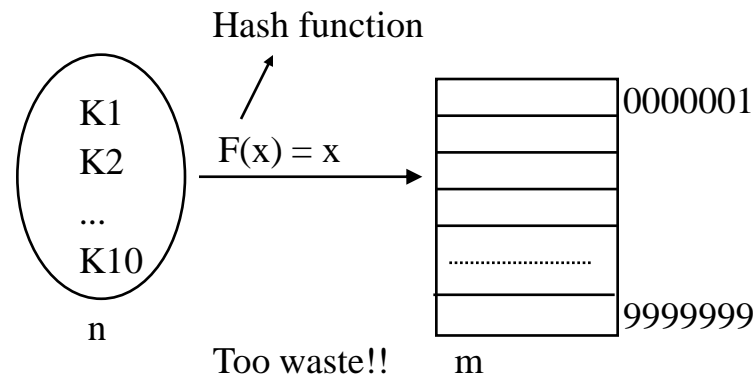
$$H(k) = \text{central digits of } K^2$$

e.g.  $k = 525 \quad K^2 = 275\underline{6}25$

- *Others: .....*

- **Observation:**

- Division method is good enough.

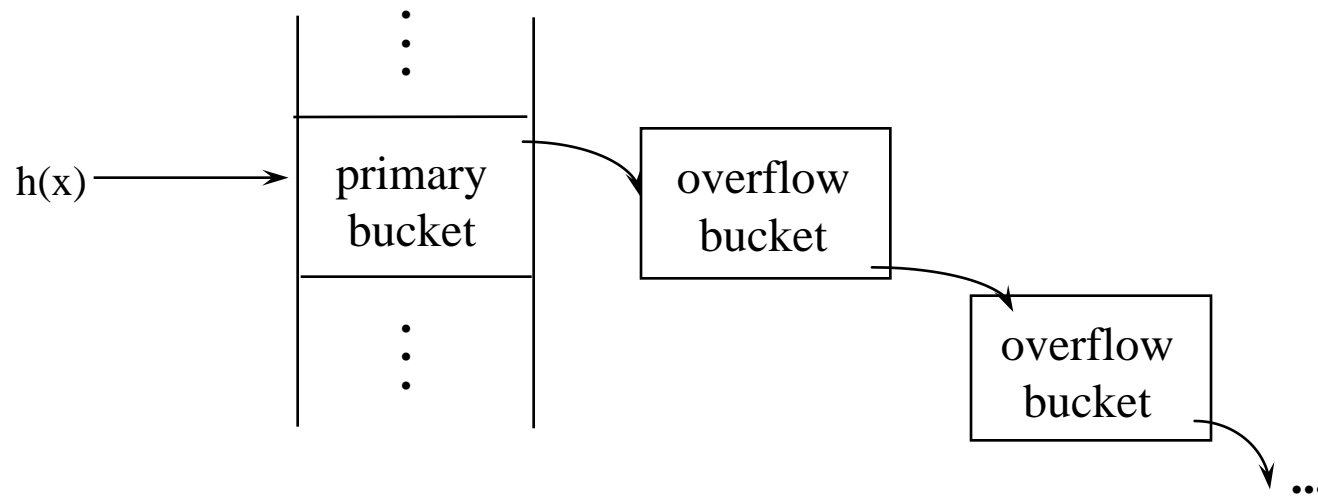


# Overflow Handling in Hashing

---

- **Overflow chaining:**

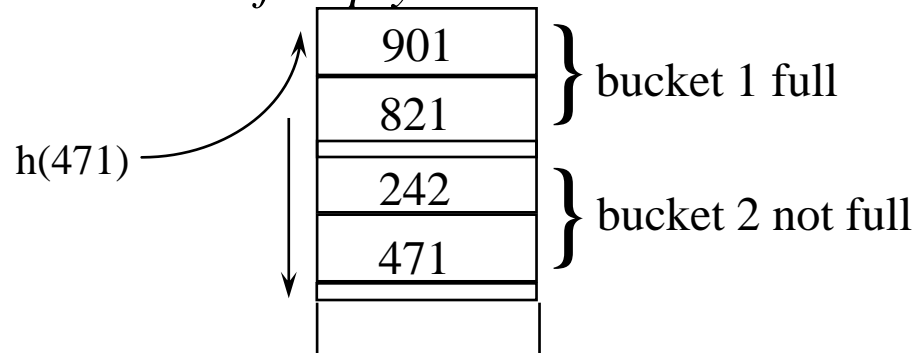
- allocate new bucket and chain to overflow bucket.



# Overflow Handling in Hashing (cont.)

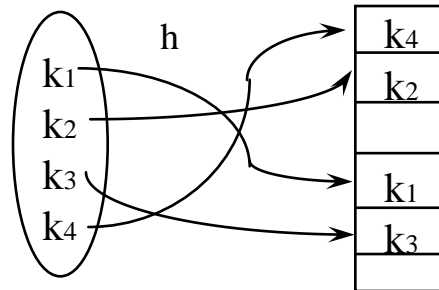
- **Open addressing:**

- *make use of empty slots in the next bucket.*



- *many variations*

- **Perfect hashing: one-to-one mapping.**





# Perfect Hash Function

---

- Rank Methods

[Ghosh 77]  $h(b_1, b_2, \dots, b_n) = m + b_1 t(\beta_1; \bar{b}_1) + b_2 * (\beta_1, \beta_2; b_1, \bar{b}_2) + \dots$

- Reduction Methods

[Sprugnoli 77]  $h(k_i) = \lfloor (k_i + S) / N \rfloor$

[Sprugndi 77]  $h(k) = \lfloor ((d + kg) \bmod m) / N \rfloor$

- Value Assignment Methods

[Cichelli 80] Hash value  $\leftarrow$  key length + f(1st char) + f(last c)

[Jaeschke 80] Counter-example

[Cook 82] improve Cichelli's method

- Reprocal Methods

[Jaeschke 81]  $h(k) = \lfloor C / (Dk + E) \rfloor \bmod n$

[Chang 84]  $h(k_i) = C \bmod P(k_i)$

- Hash Indicator Table [HIT] Methods

[Du 80, Du 83]  $h(k_i) = hj(k_i) = xi$  if  $HIT[ht(k_i)] \neq t$   $t < j$  and  $HIT[hj(k_i)] = j$

[Yang 83, Yang 85]

# Perfect Hash Function (cont.)

---

BIT 25(1985), 148-161

## A BACKTRACKING METHOD FOR CONSTRUCTING PERFECT HASH FUNCTIONS FROM A SET OF MAPPING FUNCTIONS

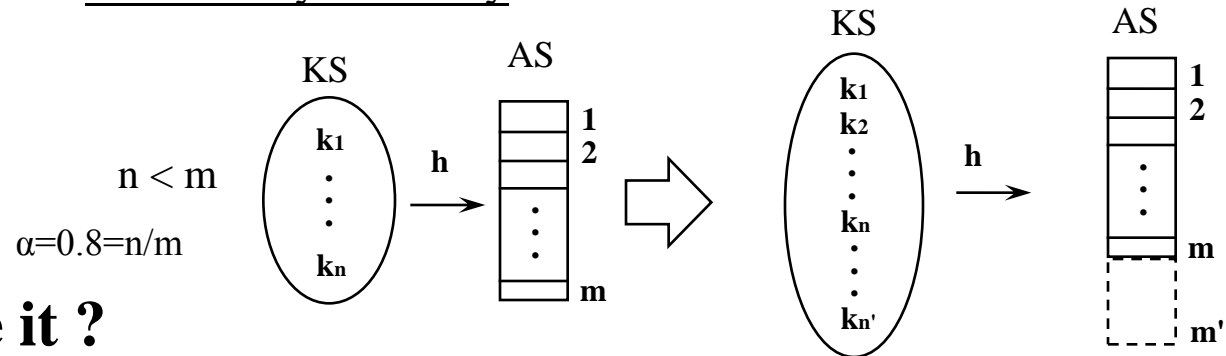
W. P. YANG and M. W. DU

*Institute of Computer Engineering, National Chiao Tung University, 45 Po Ai Street, HsinChu,  
Taiwan, Republic of China*

# Dynamic Hashing

## ■ Definition

- Dynamic Hashing: in the hashing scheme the set of keys can be varied, and the address space is allocated dynamically.

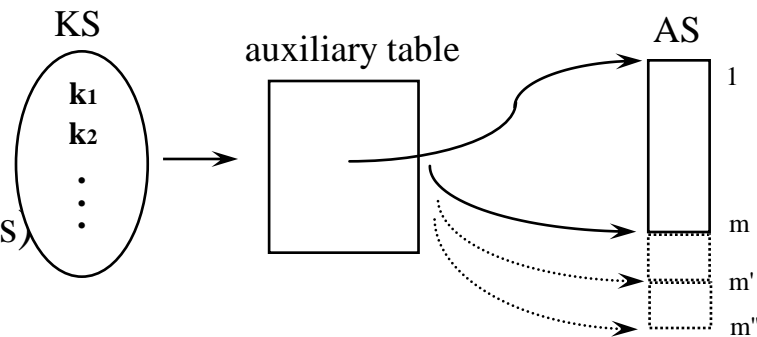


## ■ How to achieve it ?

- using a auxiliary table (e.g. index tree, bit-map table, prefix tree, directory, ...)

## ■ Problems?

- size (utilization)
- retrieval time (disk access times)
- algorithms



# Dynamic Hashing: Schemes

---

## (1) Expandable Hashing

- Knott, G. D. Expandable Open Addressing Hash Table Storage and Retrieval. Proc. ACM SIGFIDET Workshop on Data Description, Access, and Control, 186-206, 1971.

## (2) Dynamic Hashing

- Larson, P. A. Dynamic Hashing. BIT 18(1978) ,184-201.
- Scholl, M. New File Organization Based on Dynamic Hashing. ACM Trans. on Database Systems, 6, 1(March 1981), 194-211.

## (3) Virtual Hashing

- Litwin, W. Virtual Hashing: A Dynamically Changing Hashing. Proc. 4th Conf. on Very Large Data Bases, West Berlin, Sept. 1978, 517-523.

## (4) Linear Hashing

- Litwin, W. Linear Hashing: A New Tool for File and Table Addressing. Proc. 6th Conf. on Very Large Data Bases, 212-223, Montreal, Oct. 1980.
- Larson, P. A. Linear Hashing with Partial Expansions. Proc. 6th Conf. on Very Large Data Bases, Montreal, Oct. 1980, 224-232
- Larson, P. A. Performance Analysis of Linear Hashing with Partial Expansions. ACM Trans. on Database Systems, 7, 4(Dec. 1982), 566-587.

# Dynamic Hashing: Schemes (cont.)

---

## (5) Trie Hashing

- Litwin, W. Trie Hashing. Res. Rep. MAP-I-014, I.R.I.A. Le Chesnay, France, 1981. (also in Proc. 1981 ACM SIGMOD International Conference on Management of Data)

## (6) Extendible Hashing

- Fagin, R., Nievergelt, J., Pippenger, N., and Strong, H. R. Extendible Hashing - A Fast Access Method for Dynamic Files. ACM Trans. Database System 4, 3(Sept. 1979), 315-344.
- Tamminen, M. Extendible Hashing with Overflow. Information Processing Lett. 15, 5(Dec. 1982), 227-232.
- Mendelson, H. Analysis of Extendible Hashing. IEEE Trans. on Software Engineering, SE-8, 6(Nov. 1982), 611-619.
- Yao, A. C. A Note on the Analysis of Extendible Hashing. Information Processing Letter 11, 2(1980), 84-86.

## (7) **HIT (Hash Indicator Table) Method**

- Du, M. W., Hsieh, T. M., Jea, K. F., and Shieh, D. W. The Study of a New Perfect Hash Scheme. IEEE Trans. On Software Engineering, SE-9, 3(May 1983), 305-313.
- Yang, W. P., and Du, M. W. Expandable Single-Pass Perfect Hashing. Proc. of National Computer Symposium, Taiwan, Dec. 1983, 210-217.
- Yang, W. P., and Du, M. W. A Dynamic Perfect Hash Function Defined by an Extended Hash Indicator Table. Proc. 10th Conf. on Very Large Data Bases, Singapore, Aug. 1984.
- Yang, W. P. Methods for Constructing Perfect Hash Functions and its Application to the Design of Dynamic Hash Files. Doctor Thesis, National Chiao Tung University, Hsinchu, Taiwan, ROC, June 1984.

## (8) . . .

---

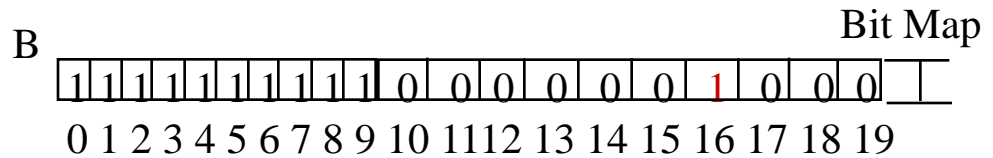
# Virtual Hashing

(Ref: "Virtual Hashing: A dynamically changing hashing", conf. VLDB 1978)

- Basic Idea: If a bucket overflows, split it into 2 buckets, and set a bit to remember it.

Example: key set:

{1366, 1256, 2519, 3546, ..., 2916, ...}

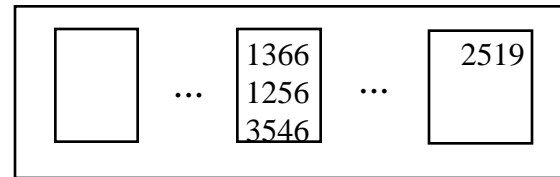


level  $l=0$

$$h_0(\text{key}) = R(\text{key}, 10)$$

Buckets

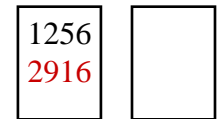
Bucket Size = 3  
N = 10



When insert: 2916  
Retrieval: 1256, 1366, 2519

level  $l=1$

$$h_1(\text{key}) = R(\text{key}, 20)$$



16 19

# Virtual Hashing (cont.)

---

- General hash function:  $h_l = R(\text{key}, 2^l \cdot N)$
- Algorithm Addressing (key)

```
1.  $j \leftarrow$  Level of  $h_j$  (Max j used)
2.  $m \leftarrow R(\text{key}, 2^l \cdot N)$ 
3. while  $B(m) = 0$ 
   |    $l \leftarrow l - 1$ 
   |    $m \leftarrow R(\text{key}, 2^l \cdot N)$ 
4. Return (m)
```

$2^0 \cdot 10 = 10$   
 $h_2(\text{key}) = R(\text{key}, 40)$

# Extendible Hashing

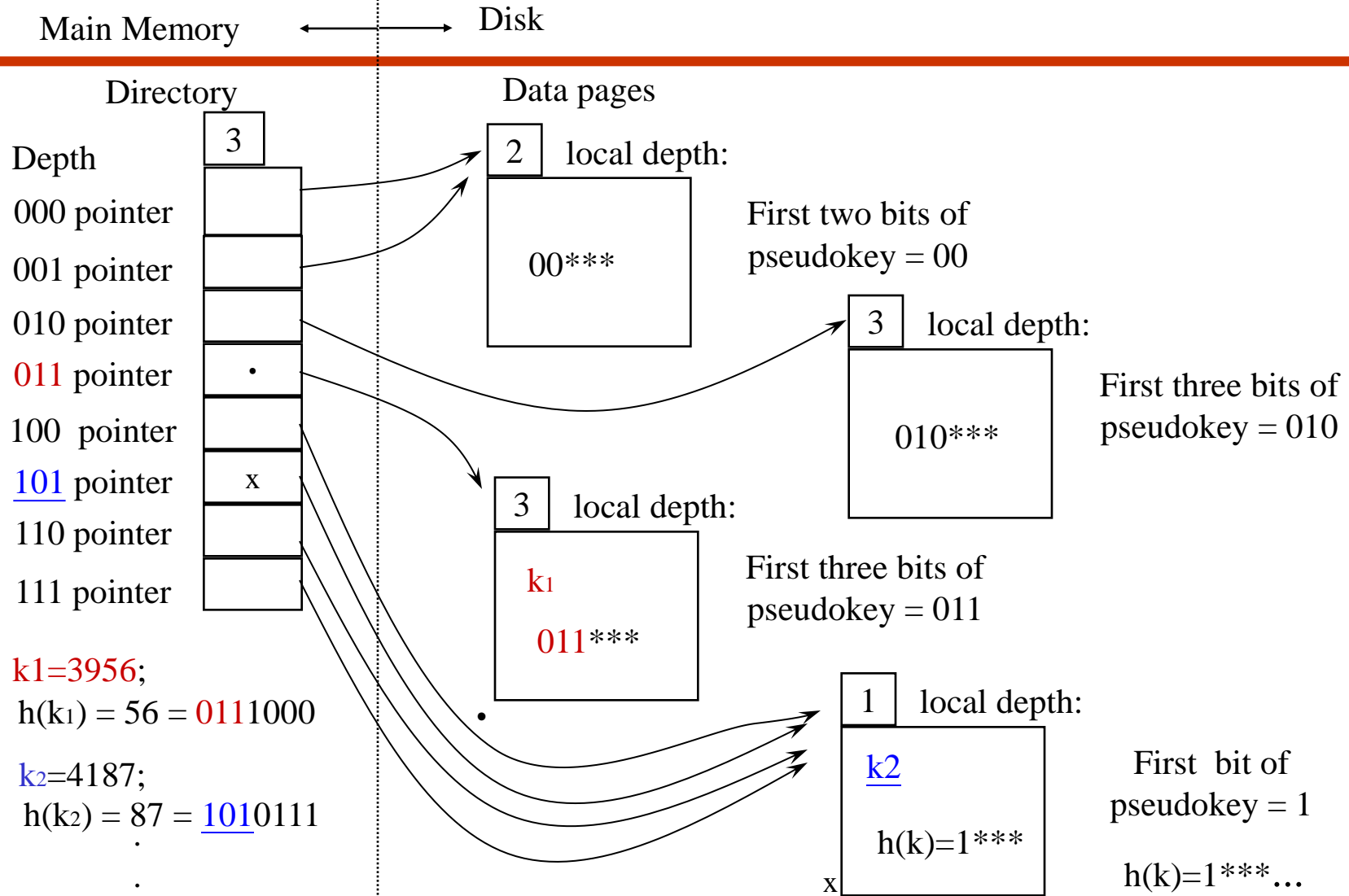
---

( Ref: Fagin, R. et. al. "Extendible Hashing-A fast access method for dynamic files", ACM TODS, Vol.4, #3 Sept. 79 )

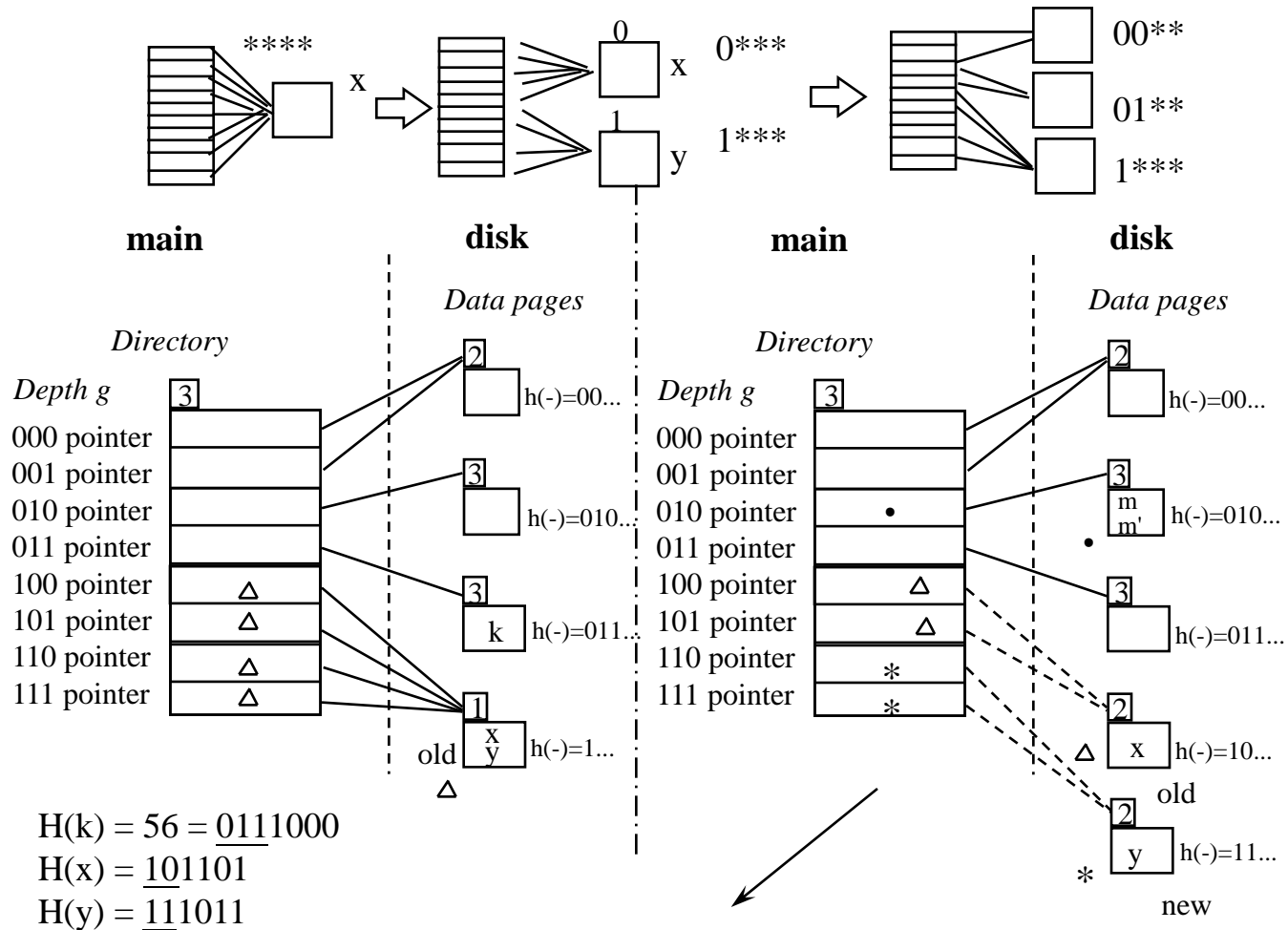
- **Basic idea** : Allow number of buckets in a certain key range to vary dynamically based on actual demand.
  - Depth(d): the number of the most significant bits in  $f(k)$  that will be taken to determine a directory entry.
    - > total number of entries in directory =  $2^d$
  - Each entry in directory points to a bucket.
  - Each bucket  $x$  has a local depth  $l_x \leq d$
  - When a bucket  $x$  overflows --> increasing  $l_x$ 
    - if  $l_x > d$  --> double directory (i.e. increasing  $d$ ).

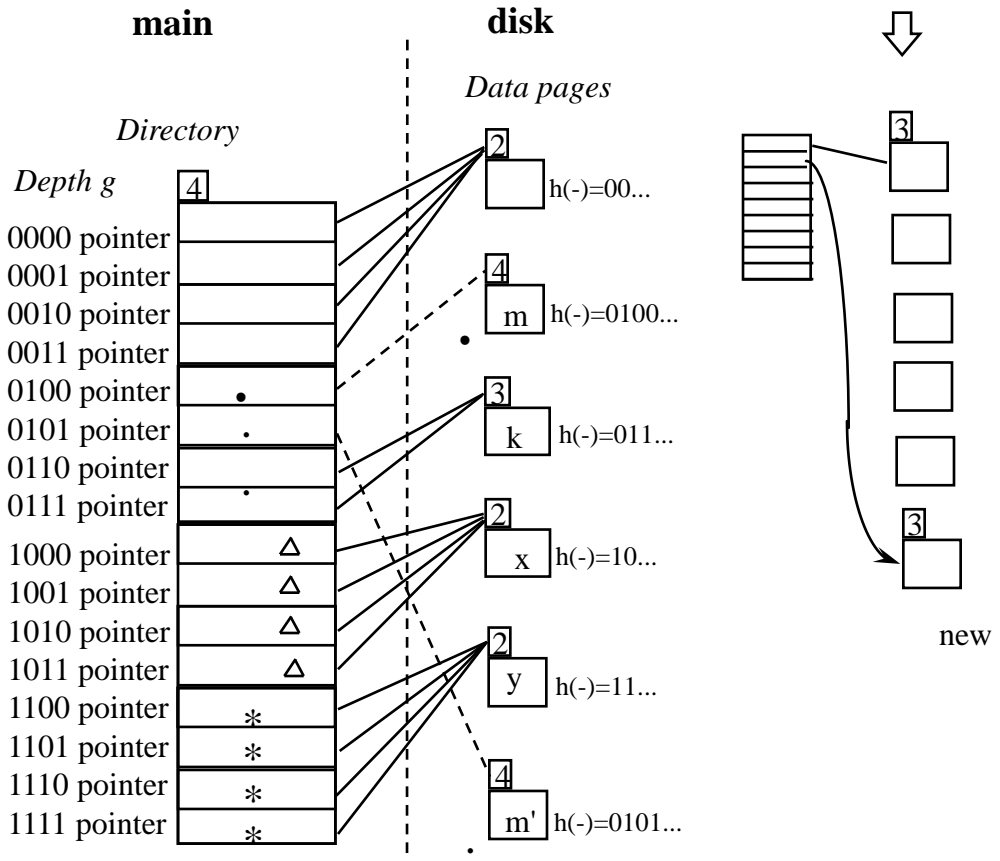
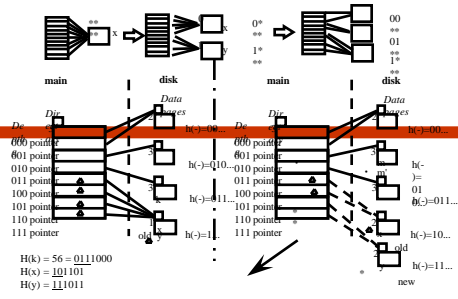


# Extendible Hashing: Example



# Extendible Hashing: Example (cont.)





# Our Research Results

---

## Concurrent Operations in Extendible Hashing

[VLDB86]

Meichun Hsu  
Wei-Pang Yang

*Harvard University  
Cambridge MA 02138*

### *Abstract*

*An algorithm for synchronizing concurrent operations on extendible hash files is presented. The algorithm is deadlock free and allows the search operations to proceed concurrently with insertion operations without having to acquire locks on the directory entries or the data pages. It also allows concurrent insertion/deletion operations to proceed without having to acquire locks on the directory entries. The algorithm is also unique in that it combines the notion of verification, fundamental to the optimistic concurrency control algorithm, and the special and known semantics of the operations in extendible hash files. A proof of correctness for the proposed algorithm is also presented.*

# Linear Hashing

---

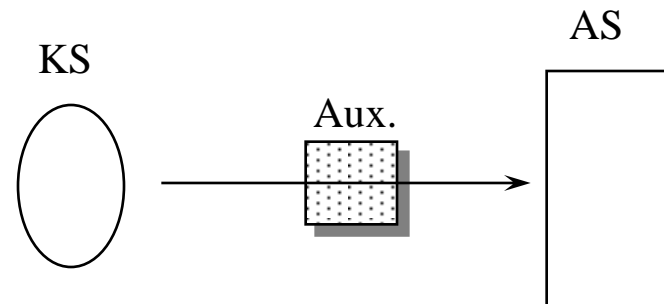
( Ref. "Linear Hashing: A new tool for file and database addressing", VLDB 1980. by W. Litwin )

## ■ Basic Idea:

- keep load factor flat by adjusting number of buckets.
- start with some number of bucket  $N$ , when loading factor exceeds a threshold  $t$  split the first bucket.
- when  $t$  is reached again, split the second bucket, and so on.
- maintain an index indicating the next bucket to be split.

## ■ Advantage:

- No directory overflow.
- simple to be implemented.



# Linear Hashing: Example

<e.g>

key set = { 16, 20, 24, 13, 26, 30, 17, 38, 15, ... }

$$H_l(k) = \text{mod}(k, N * 2^l)$$

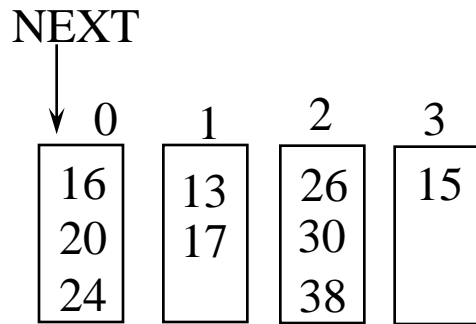
$$N * 2^0 = 4 * 1 = 4$$

$$N = 4 \quad l = 0 \quad \Rightarrow \quad H_0(k) = k \text{ mod } 4$$

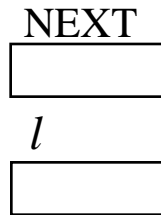
↓ insert '42'

bucket 2 overflow  
split bucket 0 (the first)

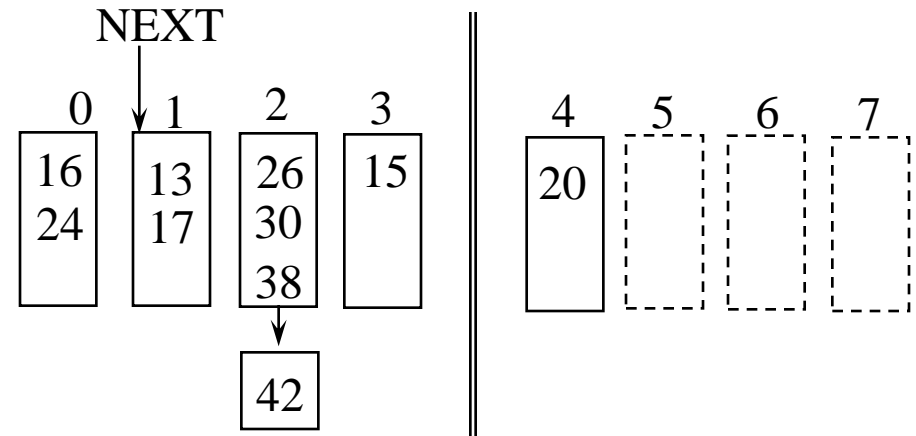
$$l = 1 \rightarrow H_1(k) = k \text{ mod } 8$$



Var:

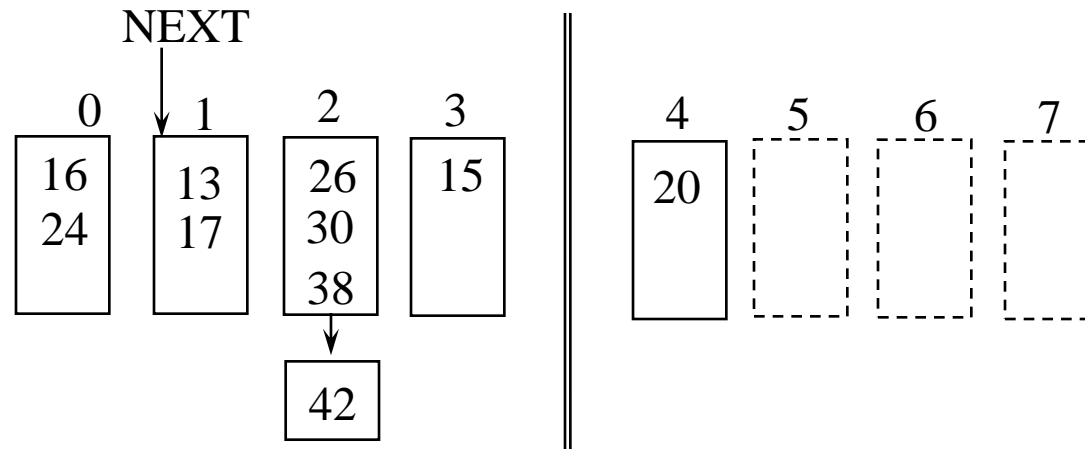


Function:  $H_i()$



# Linear Hashing: Example (cont.)

$$l = 1 \rightarrow H_1(k) = k \bmod 8$$



- Max. bucket # =  $N * 2^{l-1} + \text{NEXT} - 1 = 2^0 * 4 + 1 - 1 = 4$
- retrieve 15  $\rightarrow H_1(15) = 15 \bmod 8 = 7 > 4$   
 $\rightarrow H_0(15) = 7 - 4 = 3 \leq 4$
- Simulation:  $b = 20$  (bucket size) 20 records  
Disk I/O for retrieval  $\approx 1.6$

# An Amortized Analysis of Linear Hashing [NSC'89]

---

Been-Chien Chien and Wei-Pang Yang

*Institute of Computer Science and Information Engineering  
National Chiao Tung University  
Hsinchu, Taiwan, Republic of China*

## *Abstract*

*In this paper we analyze the amortized cost under a sequence of  $m$  split operations in linear hashing which is one of dynamic storage structures without any directory in database system. We prove that the split cost of linear hashing with uncontrolled split strategy will be bounded under  $6m + \frac{2m - 2(t+L)N}{c}$  bucket accesses even in a pessimistic situation in which split ration is zero, where  $N$  is the number of initial primary buckets,  $c$  is the capacity of overflow bucket, and  $t$  is the last file level of  $m$  split operations. And achieving almost  $5m$  bucket accesses with the assumption of split ratio in every one of split operations is  $b$ , where  $b$  is the capacity of primary bucket. Under the same assumption of split ratio, the result is close to the expansion cost of other dynamic hashing schemes with extra storage for directory such as extendible hashing. It shows the expansion strategy of linear hashing with uncontrolled split can provide the same expansion function efficiently but need not extra directory.*

*Keyword: algorithm, database, amortize, hashing.*



# Concurrent Operations in Linear Hashing

---

MEICHUN HSU

*Center for Research in Computing Technology, Harvard University, Cambridge, Massachusetts*

SHANG-SHENG TUNG

WEI-PANG YANG

*National Chiao-Tung University, Hsinchu, Taiwan, ROC*

Communicated by Ahmed K. Elmagarmid

## ABSTRACT

New concurrent operations for linear hashing are presented. The algorithm uses an optimistic concurrency control technique which leads an operation to "retry" when interaction among concurrent conflicting operations occurs. The method is unique in that it makes use of a strictly increasing counter to *filter out* a significant portion of unnecessary retries, thus allowing search, insert, and delete operations to proceed concurrently with split and merge operations. The search operation does not need to set any lock, and no operation needs to set lock on shared *directory* variables, such as the pointer to the next bucket to split, thus enabling a higher degree of interleaving in the system. An argument for correctness, in terms of a correctness criterion which incorporates external consistency, is presented, as well as a discussion of the filter performance.

# Concurrent Operations in Multi-Dimensional Extendible Hashing

[JISE89]

PAO-CHUNG HO AND WEI-PAN YANG

*Institute of Computer Science and Information Engineering  
National Chiao Tung University  
Hsinchu, Taiwan 30050, Republic of China*

MEICHUN HSU

*Center of Research in Computing Technology  
Aiken Computation Laboratory, Harvard University  
Cambridge, MA 02138, U.S.A.*

*An algorithm for synchronizing concurrent operations on multi-dimensional extendible hash files is presented. the algorithm is deadlock free and allows the search and partial-match operations to proceed concurrently with the insertion operations without having to acquire any locks. it also allows concurrent insertion/deletion operations to proceed without having to acquire locks on the directory entries. The algorithm combines the notion of verification, the principle of the optimistic concurrency control algorithm, and the special and known semantics of operations in multi-dimensional extendible hash files. A correctness argument for the proposed algorithm is also presented.*

*Keywords: Concurrency control, extendible hashing, algorithm, database.*

# 6.4 Pointer Chains

---

# Pointer Chains

- **Suppose the query:** "Find all suppliers in city xxx" is an important one.
- **parent / child organization:**

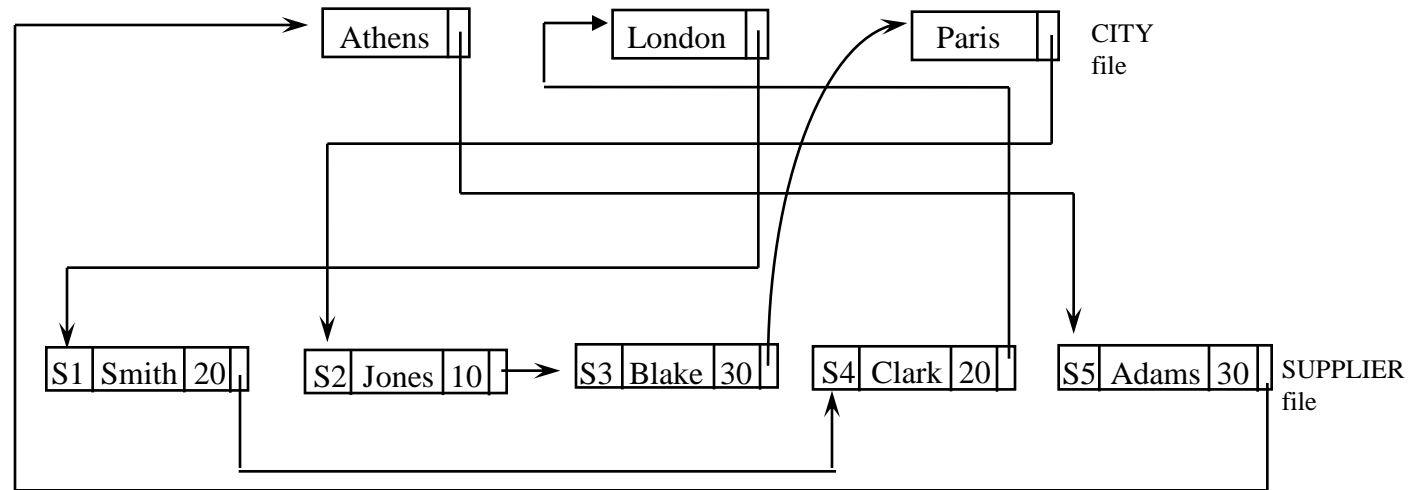


Fig. 6.6: Example of a parent/child structure

- **Advantages:**
  - 1. insert, delete are simpler and more efficient than index structure. (§6-2, Fig. 6-2)
  - 2. occupy less storage than index structure. (Fig. 6-2)
- **Disadvantages:**
  - 1. to access the nth supplier --> sequential access --> slow !
  - 2. create new pointer-chain requires database reorganization.

# 6.5 Compression Techniques

---

Objective : reducing storage space

--> reducing disk I/O

# Differential Compression

- consider a page of entries from an "employee name" index :

-Method 1

*front compression:*

6												
ROBERT	ON	\$\$\$										→
ROBERT	SON	\$\$\$										→
ROBERT	STONES											→
ROBINHOOD	\$\$\$											→

0 1 2 3 4 5 6 7 8 9 10 11

0 - ROBERTON\$\$\$\$

6 - SON\$\$\$

7 - TONES

3 - INHOOD\$\$\$



ROBERTON\$\$\$\$

ROBERTSON\$\$\$

ROBERTSTONES

ROBINHOOD\$\$\$

-Method 2

*rear compression:* eliminate blanks, replaced by a count.

0 - 7 - ROBERTO

▼ 6 - 2 - SO

7 - 1 - T

3 - 1 - I



ROBERTO?????

ROBERTSO????

ROBERTST????

ROBI????????

# Hierarchic Compression

- Suppose a file is clustered by same field F (e. g. CITY), and each distinct value of F occurs in several (consecutive) records of that file.

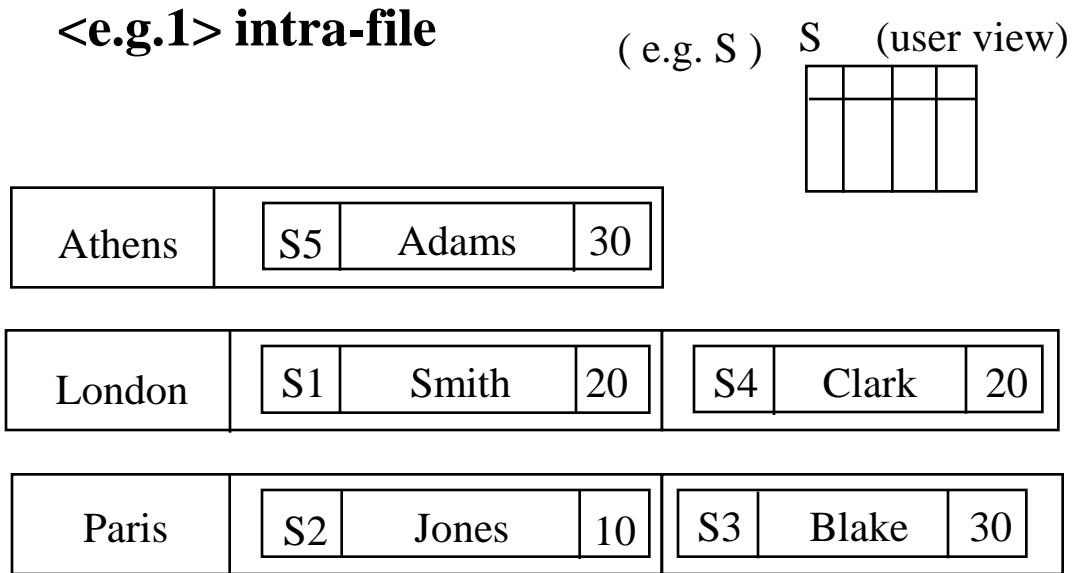
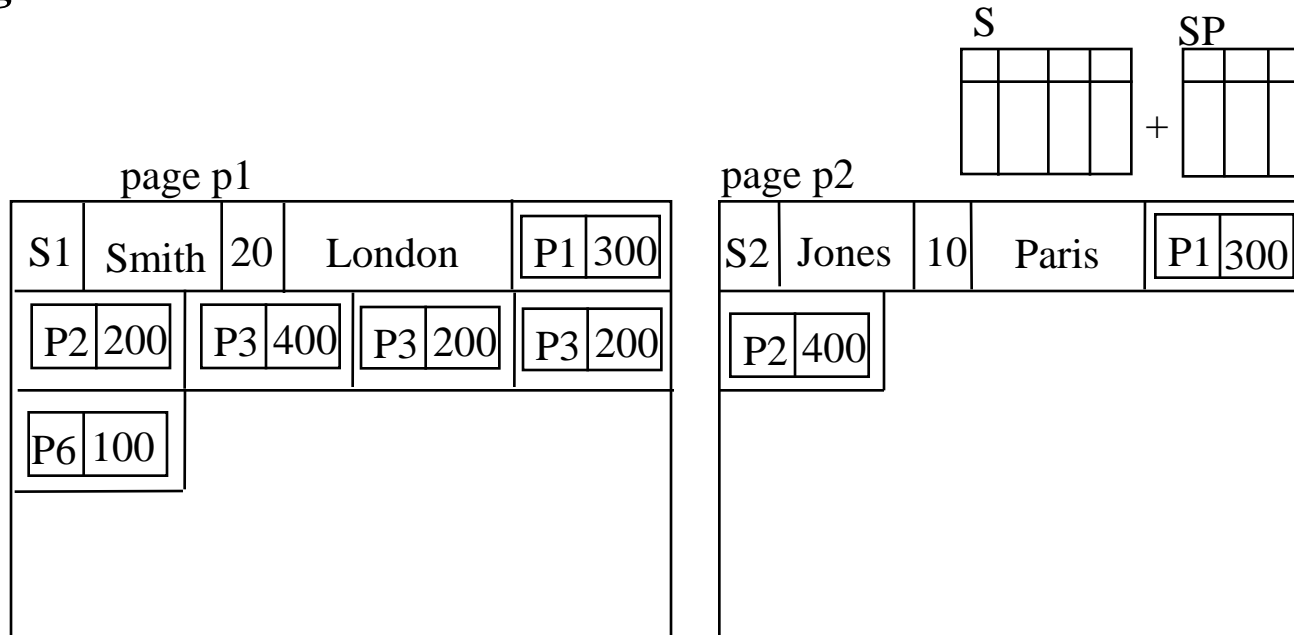


Fig. 6.7: Example of hierarchic compression (*intra-file*)

# Hierarchic Compression (cont.)

<e.g. 2> inter-file



( and similarly for pages p3, p4, p5 )

Fig. 6.8: Example of hierarchic compression (*inter-file*)



# Huffman Coding

## ■ Consider the coding schemes for {A,B,C,D,E}

### • Method 1:

A: 000  
 B: 001  
 C: 010  
 D: 011  
 E: 100

average length = **3** bits/char  
 encoding:  $\frac{010}{C} \frac{100}{E} \frac{011}{D}$

### • Method 2:

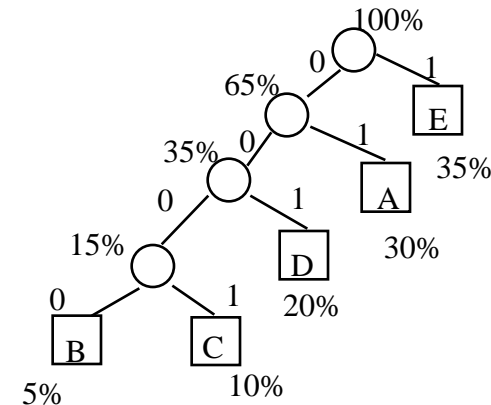
A: 1  
 B: 01  
 C: 001  
 D: 0001  
 E: 0000

average length =  $(1+2+3+4+4)/5$   
 = **2.8** bits/char  
 encoding:  $\frac{01}{B} \frac{0000}{E} \frac{001}{D}$

### • Method 3:: Huffman coding

the most commonly occurring characters are represented by the shortest strings.

Assume: A: 30% - 01  
 B: 5% - 0000  
 C: 10% - 0001  
 D: 20% - 001  
 E: 35% - 1



average length =  $2*30\%+4*5\%+4*10\%+3*20\%+1*35\%$   
 = **2.15** bits/char

encoding:  $\frac{01}{A} \frac{001}{D} \frac{0000}{B}$

# 6.6. Differential File Organization

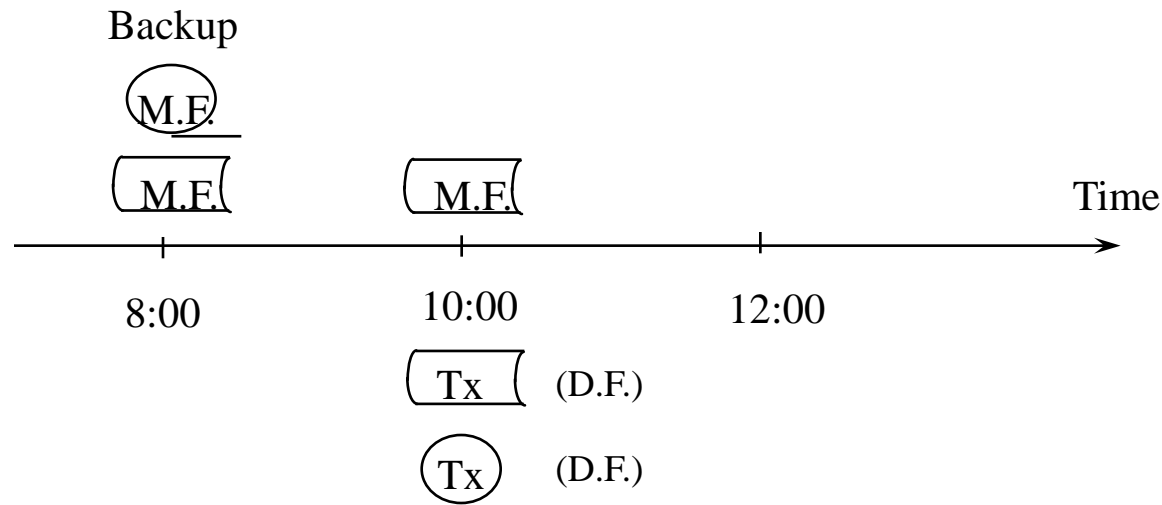
---

(Ref : 1. "Differential files : their application to the maintenance of large databases",  
by D.G. Severance and G.M Lohman, TODS, 1, 3, Sept, 1976.

2. "A practical guide to the design of differential files for recovery on-line  
database", by H. Aghili and D.G. Severance, TODS, 7, 4, Dec, 1982 )

# Differential File: Basic Idea

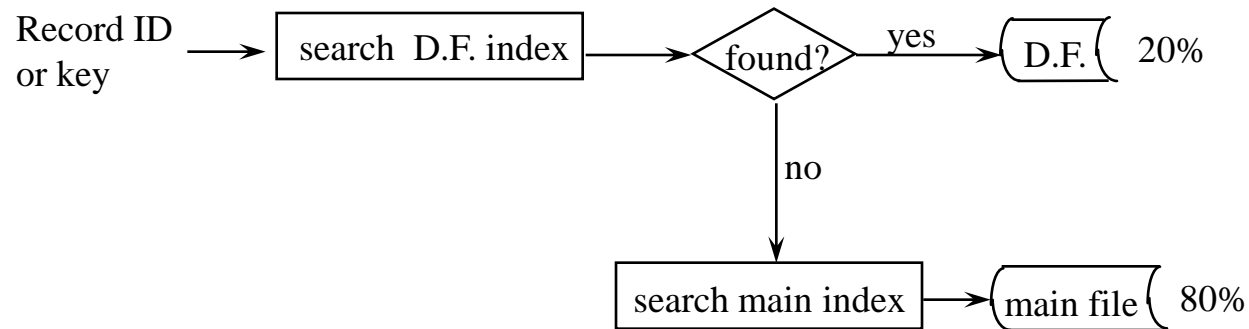
- Main file (Primary file) remains static until reorganization.
- Updated records get inserted into differential file.
- To search for a record requires:
  - <1> search in the differential file first.
  - <2> if not found, then search in the main file.
- Motivation
  - reduces dumping cost for backup and recovery



# Differential File (cont.)

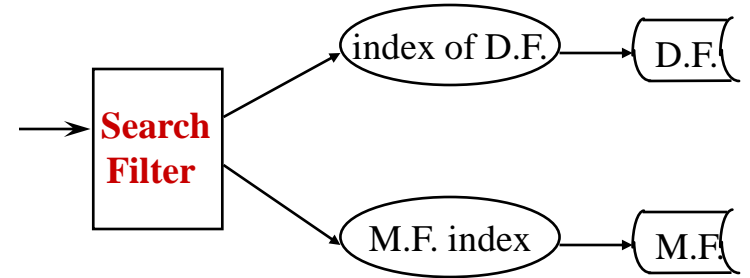
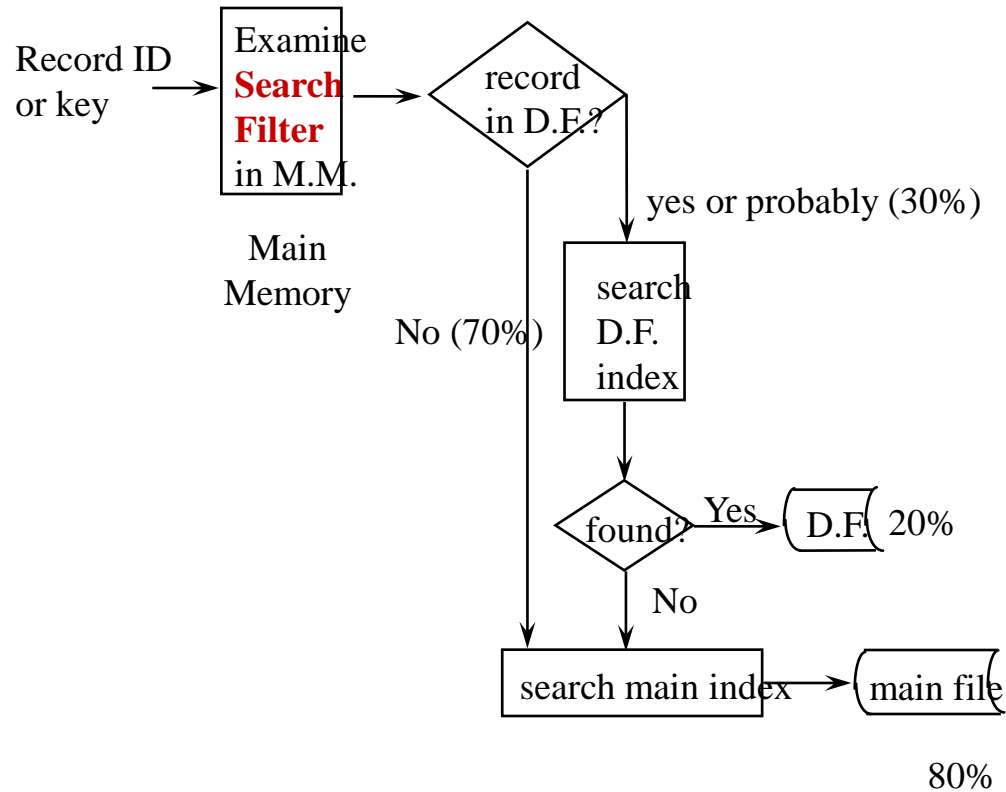
---

- How to access when using differential file?



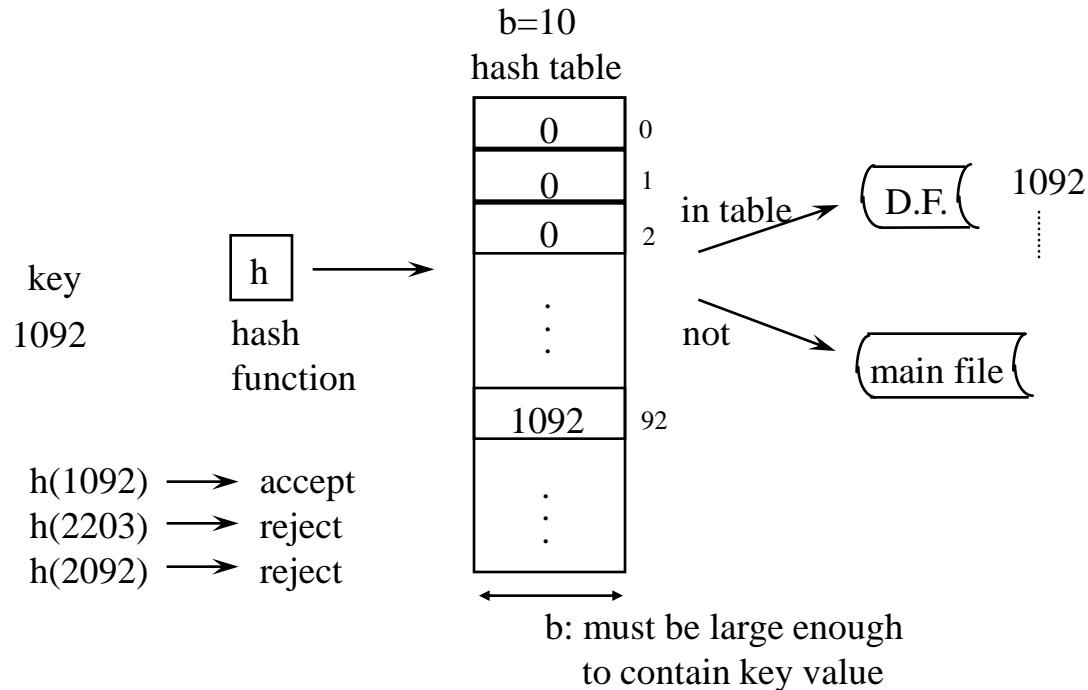
- Disadvantage: requires double index access if data is in main file

# Search Filter



# Search Filter: Method 1

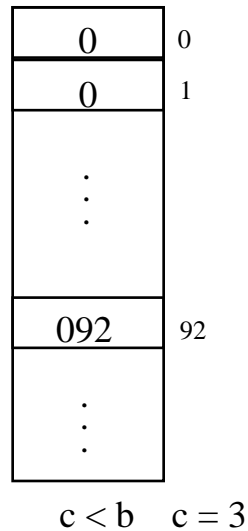
- *Generally employ hash coding technique*
  - **Method 1 : (error free)**



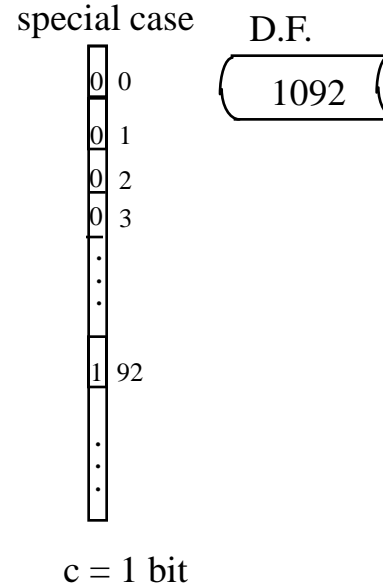
# Search Filter: Method 2

- Method 2:

$h(1092) \rightarrow$  accept  
 $h(2203) \rightarrow$  reject  
 $h(2092) \rightarrow$  accept  
 (a false drop)  
 1192



- Method 2'



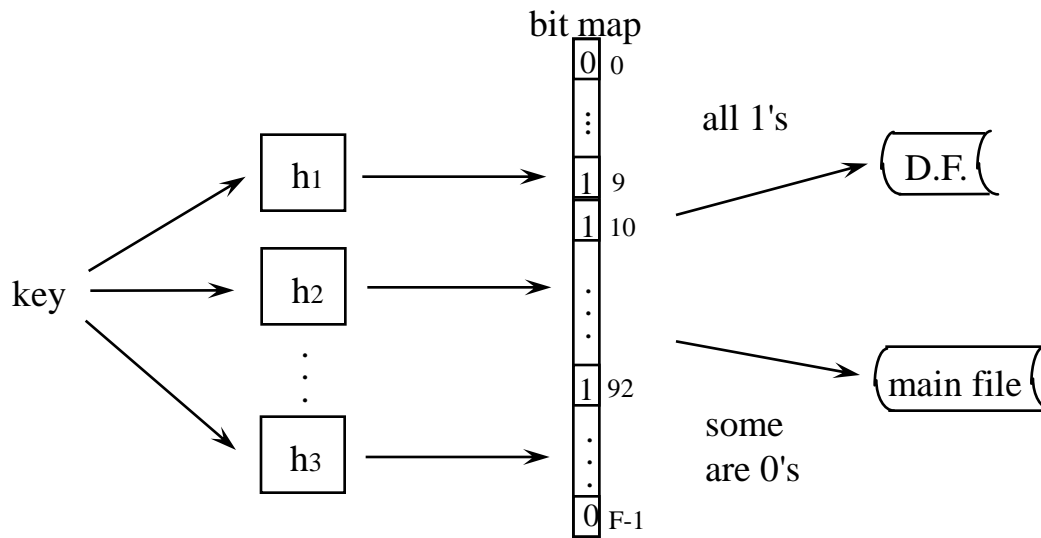
-Advantage: smaller space overhead

-Disadvantage: *false drops* occasionally occur

<Note>: the larger the hash space is, the smaller the false drop probability

# Search Filter: Method 3 - Bloom Filter

- **Method 3: Bloom Filter** (ref : B.H. Bloom, "space/time trade offs in hash coding with allowable errors", CACM,1970)



<e.g.>  $h_1(1092) = 92$   
 $h_2(1092) = 09$   
 $h_3(1092) = 10$

$$h_i \neq h_j, i \neq j$$

-fails drop probability depend on

-k: # of hash function used

-F: size of hash space

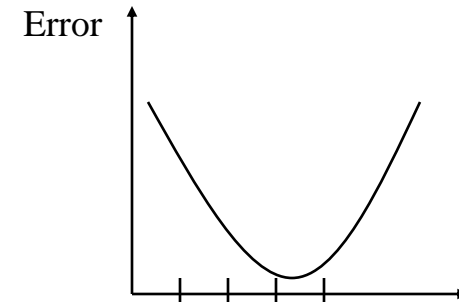
-N: # of keys to be stored in hash space.

<e.g.> :  $k=3$

$N = 1000$

$F = 5000 \longrightarrow P_{fd} = 0.0919$

$F = 10000 \longrightarrow P_{fd} = 0.0174$



#-of hash function 1 2 3 4



# Our Research Results: Method 4 - Random Filter

## Method 4: Random Filter

C. R. Tseng and W. P. Yang,  
“2D Random Filter and Analysis,”  
*International Journal of Computer Mathematics*,  
vol. 42, pp. 33-45, 1992.

