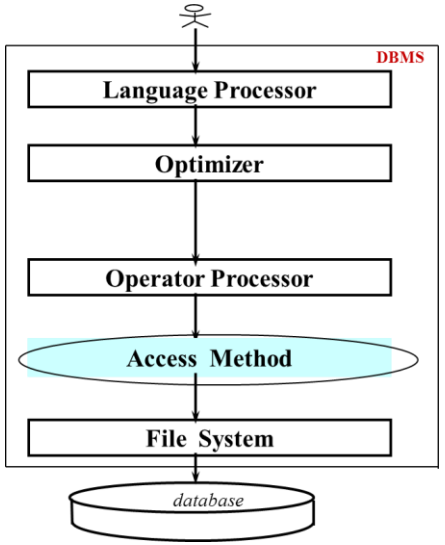
Unit 11
File Organization and Access Methods



本課程講授內容

- **PART I: 入門與導論**
  - Overview
  - DB2系統及SQL語言
  - 闡述關連式資料模型(The Relational Model)
  - 階層式資料模型(The Hierarchical Model)簡介
  - 網狀式資料模型(The Network Model)簡介
- **PART II: 資料庫設計 (Database Design)**
  - 資料庫問題分析與 E-R Model
  - 資料庫的表格正規化
  - 設計介面增刪查改資料庫
- **PART III: 進階探討**
  - 快速存取方法(Access Methods)
  - 資料庫回復(Database Recovery)
  - 協同控制(Concurrency Control)
  - 資料安全與資料正確(Security and Integrity)
  - 查詢最佳化(Query Optimization)
  - 分散式資料庫系統(Distributed Database)

PART III: 進階探討

- 快速存取方法(Access Methods):介紹大量資料庫之所以可以快速存取的技巧及方法，包括最有名的索引系統 B-tree index.
- 資料庫回復(Database Recovery):介紹萬一系統出問題，導致資料不正確時，如何回復到某一正確點的相關議題。
- 協同控制(Concurrency Control):通常同一資料庫是允許多人同時共用的，本單元探討DBMS是如做控制使得多人共同而不互相干擾。
- 資料安全與資料正確(Security and Integrity): 資料安全是指保護資料不讓未授權的人偷竊、破壞;資料正確是指保護資料不讓已有授權的合法使用者誤用，例如把存款餘額變成負數。
- 查詢最佳化(Query Optimization):查詢資料庫某一問題可以有不同的查詢指令。不同的指令查詢時間可能相差好幾倍，本單元介紹撰寫最佳化的查詢語言指令的方法。
- 分散式資料庫系統(Distributed Database): 分散式資料庫是透過電腦網路將多個資料庫連線起來組成的資料庫。現今我們用的系統有太多的分散式資料庫系統，本單元將介紹之，探討它的優缺點。
- 擴充E-R Model: 一般不太複雜的問題，前面介紹的E-R Model基本上就夠用。但大系統或比較特殊的系統就必須有更多功能的擴充式E-R Model(Extended E-R Model,簡稱EER-model)可處理包括Specialization、Generalization、Aggregation等問題。
- 進階表格正規化: 關連式資料庫是由許多表格(tables)組成的，表格要正規化。正規化分六級，一般也許到第三級正規化即可。理論上還有BCNF級、第四級、第五級的正規化。本單元要介紹這些特例。當年這些是博士論文級的問題探討。

Contents of PART III: 進階探討

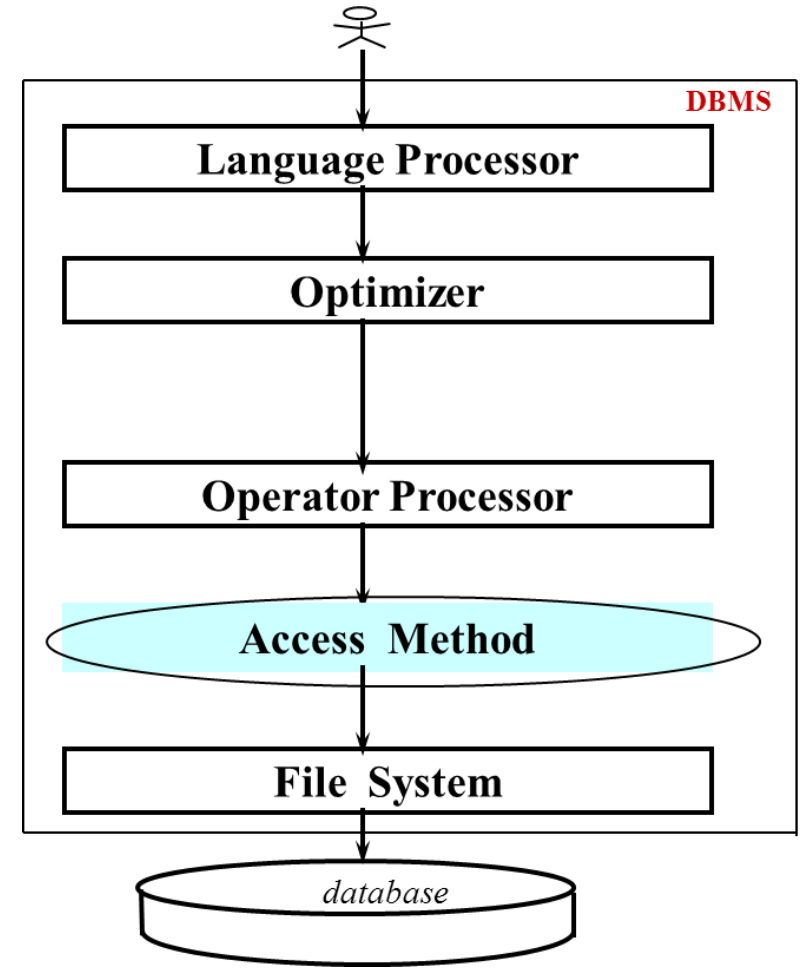
- **Unit 11 Access Methods**
- **Unit 12 Database Recovery**
- **Unit 13 Concurrency Control**
- **Unit 14 Security and Integrity**
- **Unit 15 Query Optimization**
- **Unit 16 Distributed Database**
- **Unit 17 More on E-R Model**
- **Unit 18 More on Normalization**
- **Unit 19 More on User Interfaces**
- **Unit 20 More on X?**

- **References:**

1. C. J. Date, An Introduction to Database Systems, 8th edition, Addison-Wesley, 2004.
2. A. Silberschatz, etc., Database System Concepts, 5th edition, McGraw Hill, 2006.
3. J. D. Ullman and J. Widom, A First Course in Database Systems, 3rd edition, Prentice Hall, 2007.
4. Cited papers (講義中提到之參考文獻)

Contents

- 11.1 Introduction
- 11.2 Indexing
- 11.3 Hashing
- 11.4 Pointer Chains
- 11.5 Compression Techniques
- 11.6 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

11.1 Introduction to Access Methods

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:

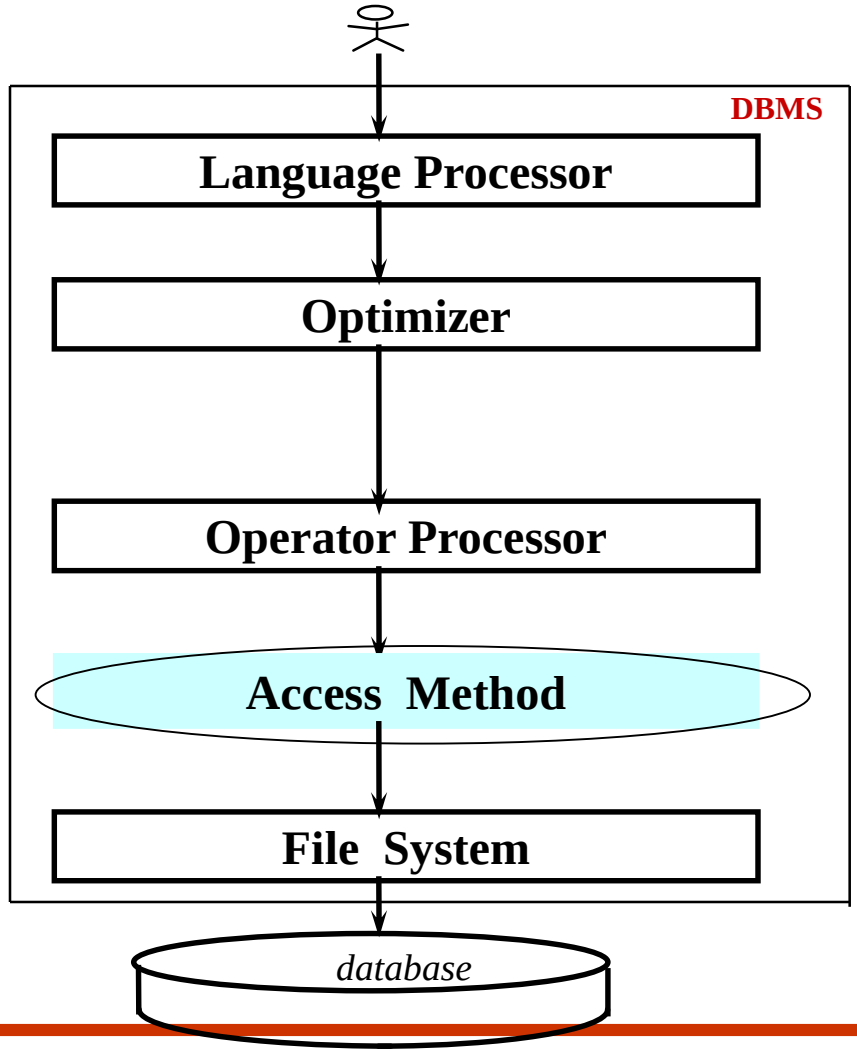
```
SCAN C using region index, create C
SCAN I using amount index, create I
SORT C?and I?on C#
JOIN C?and I?on C#
EXTRACT name field
```

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

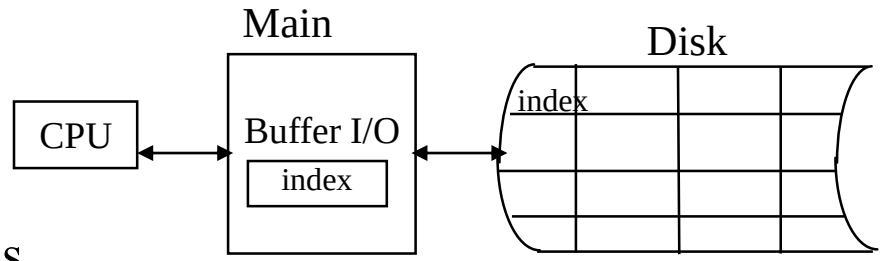
- **Objectives:**
  - concern the way the data is actually stored.
  - store data on direct access media. e.g. disk.
  - minimize the number of disk access (disk I/O).
  - disk access is much slower than main storage access time.

- **Storage Structure/File Structure:**
  - many different storage structures:

    <e.g> indexing, hashing, pointer chains, …
  - different structures have different performance

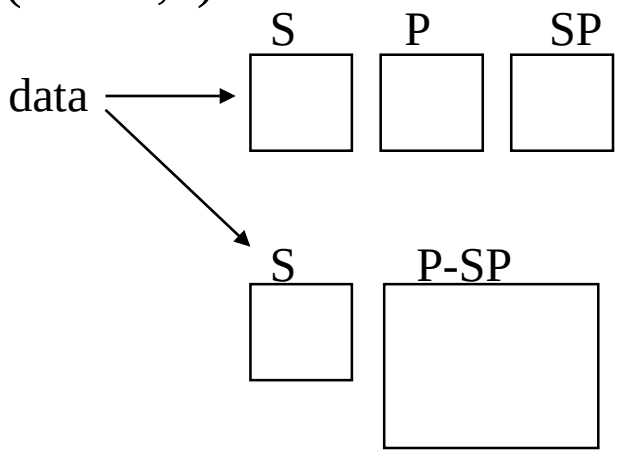
    => no single structure is optimal for all applications.

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



The Internal Level (cont.)

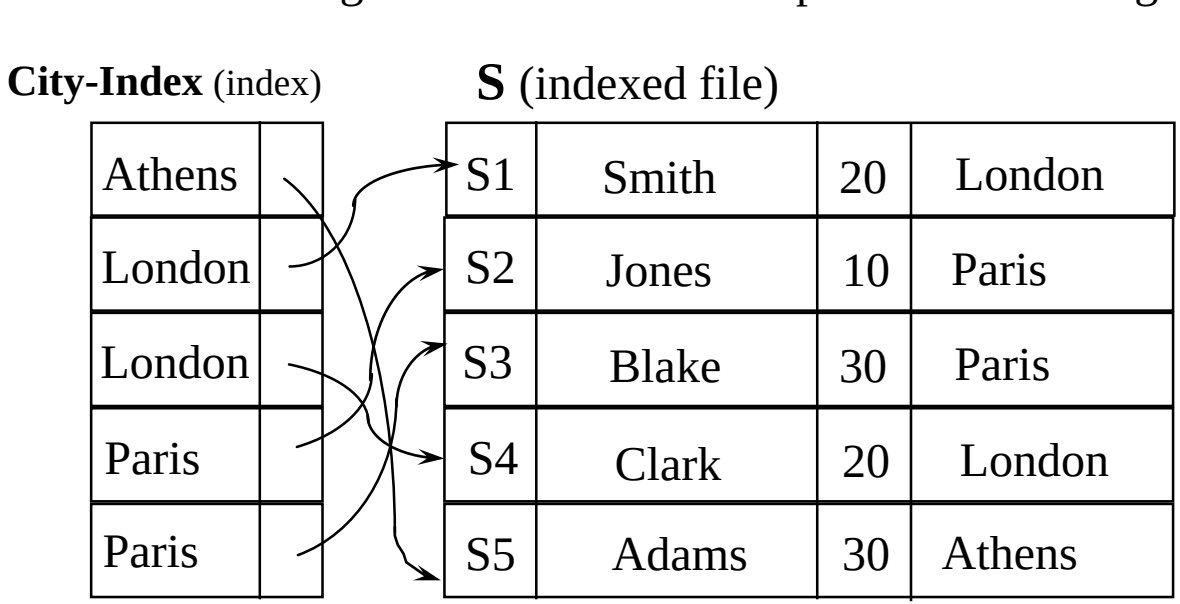
- **Physical database design:**
  - Process of choosing an appropriate storage representation for a given database (by DBA). E.g. designing B-tree or hashing
  - Nontrivial task
    - require a good understanding of how the database will be used.
- **Logical database design:** (Unit 6,7)



11.2 Indexing (1)

Indexing: Introduction

- Consider the Supplier table, **S**.
- Suppose "*Find all suppliers in city xxx*" is an important query.
  i.e. it is frequency executed.
  => DBA might choose the stored representation as Fig. 11.2.



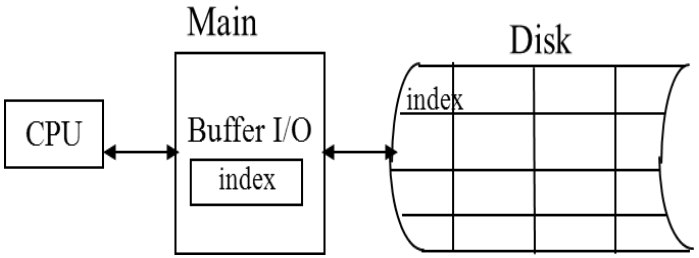


Fig. 11.2: Indexing the supplier file on CITY.

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.

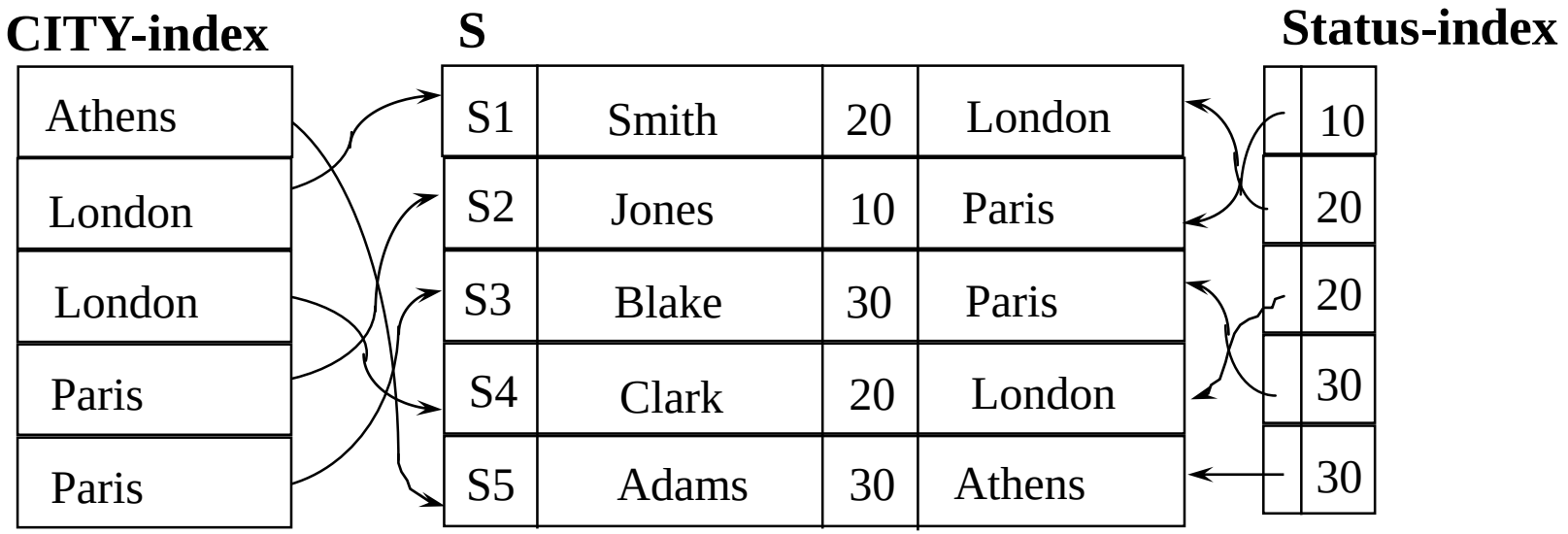
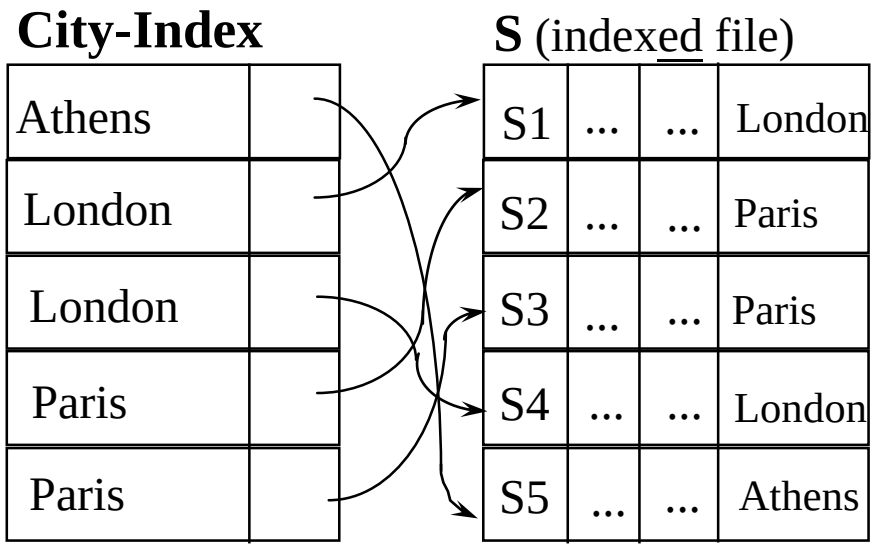


Fig. 11.3: Indexing the supplier file on both CITY and STATUS.

How Index are used?

Consider:



<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** :

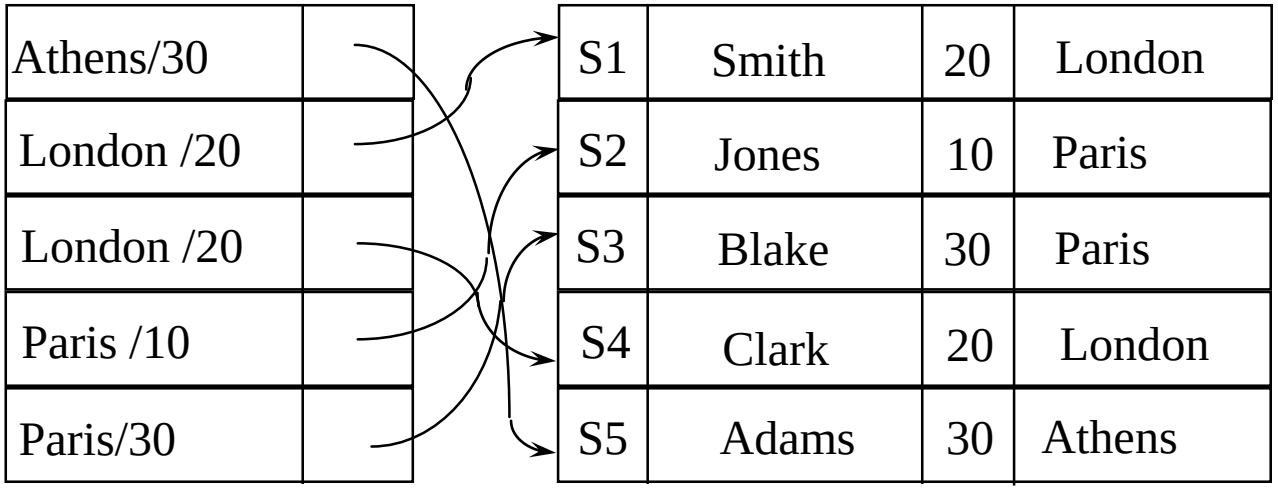
<e.g> "Is there any supplier in London ?"

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.

<e.g> **City/Status-Index** **S**



City/Status-Index	
Athens/30	
London /20	
London /20	
Paris /10	
Paris/30	

S			
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

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. 11.3)

Query1: "Find suppliers in Paris with status 30"
Query2: "Find suppliers with status 30"
Query3: "Find suppliers in Paris"

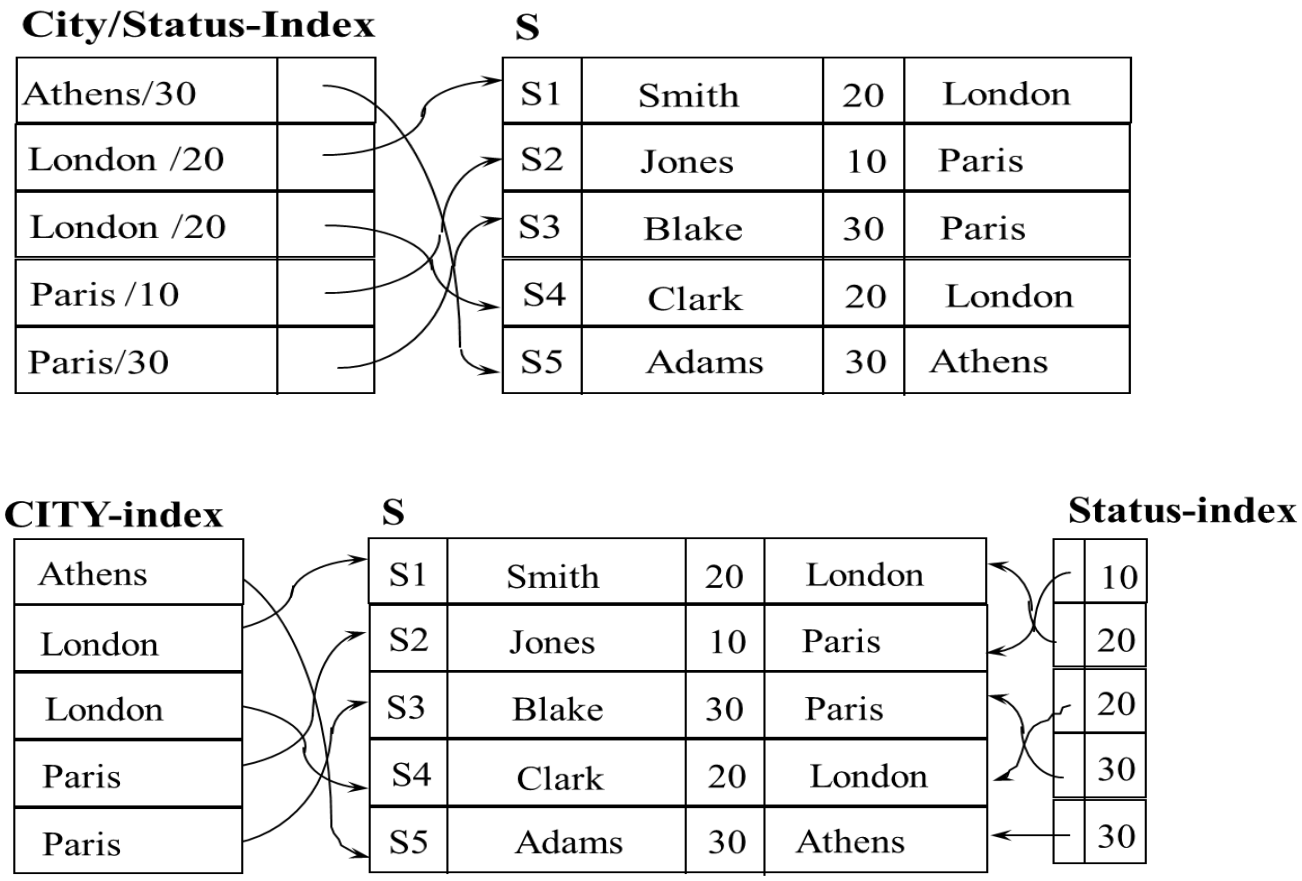


Fig. 11.3: Indexing the supplier file on both CITY and STATUS.

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
     - ⋮

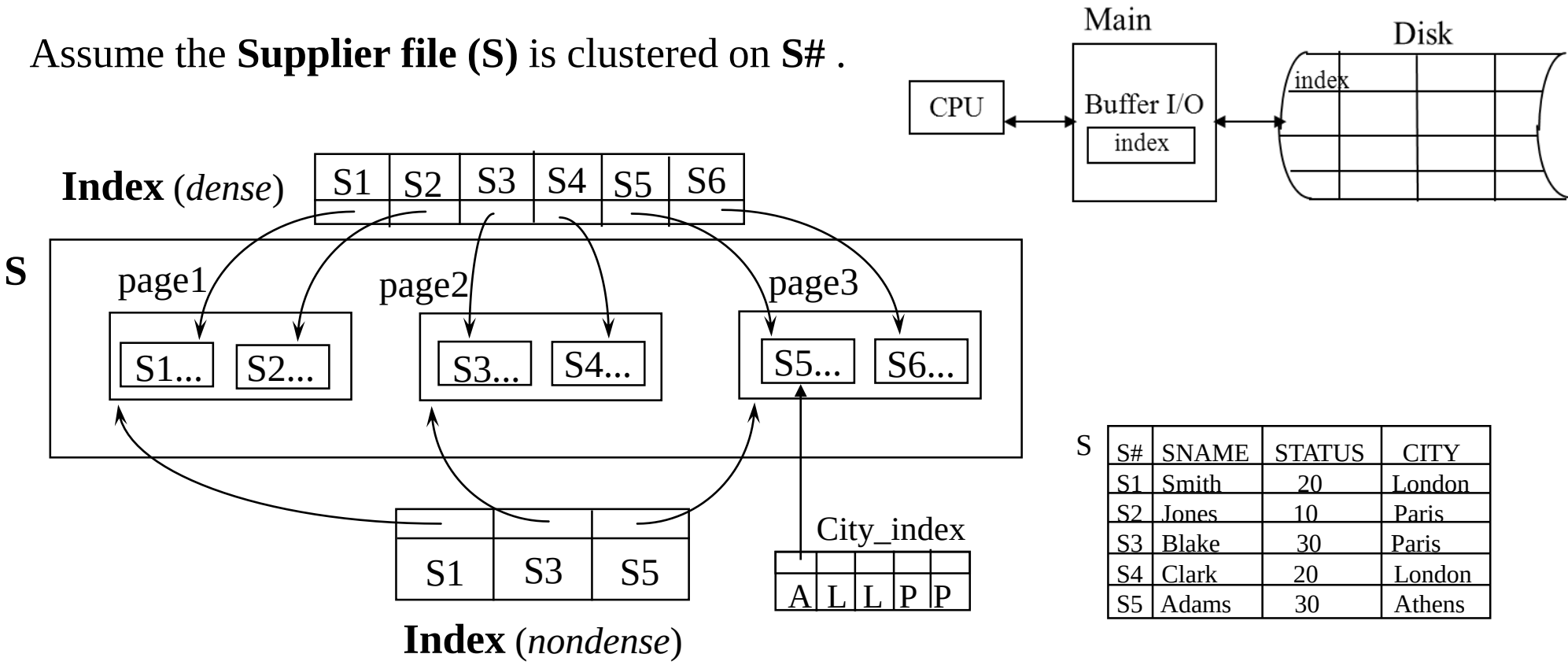
<Think>: How many indexes can F1....Fn serve as ?

11.2 Indexing (2)

- Dense vs. Nondense Indexing
- B-tree and B^+-tree

Dense v.s. Nondense Indexing

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



S

S#	SNAME	STATUS	CITY
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

Dense v.s. Nondense Indexing (cont.)

- **Nondense index:** not contain an entry for every record in the indexed file.
  - retrieval steps:

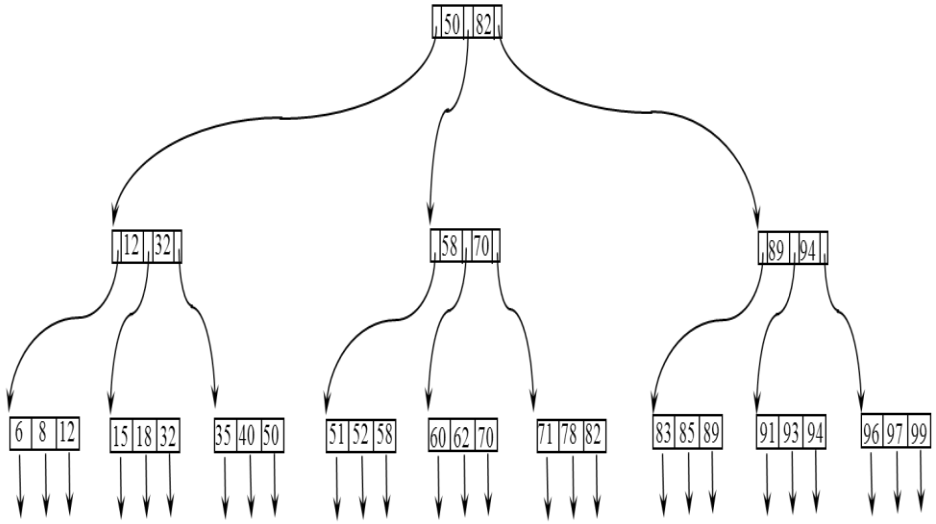
    <1> scan the index (nondense) to get page # , say p.
    <2> retrieve page p and scan it in main storage.
  - advantages:

    <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")**

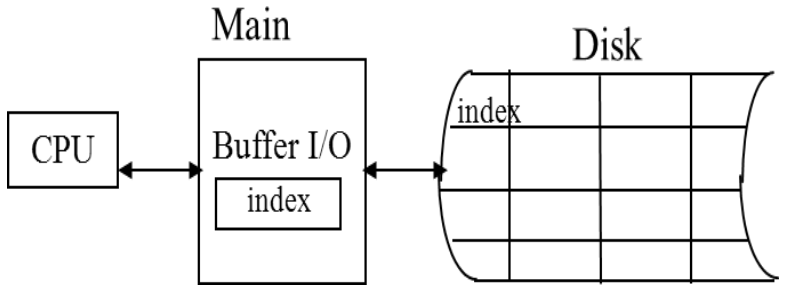
  A **B-tree T** of order **m** is an **m-way** search tree, such that
  <1> the root node has at least 2 children.
  <2> non-leaf nodes have at least [m/2] children.
  <3> all leave nodes are at the same level.

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

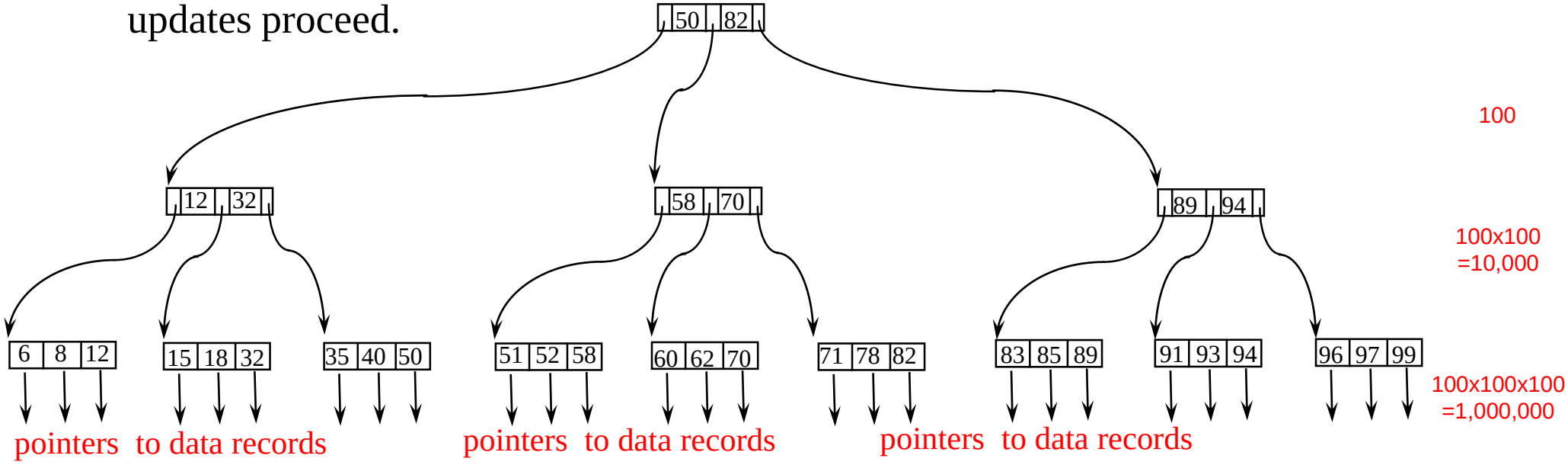
B-tree (cont.)

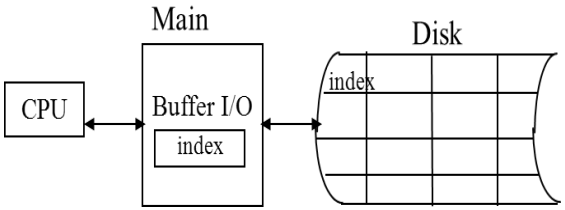
- **Definition: (from Horowitz "Data Structure")**

  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 [m/2] children.
  <3> all leave nodes are at the same level.

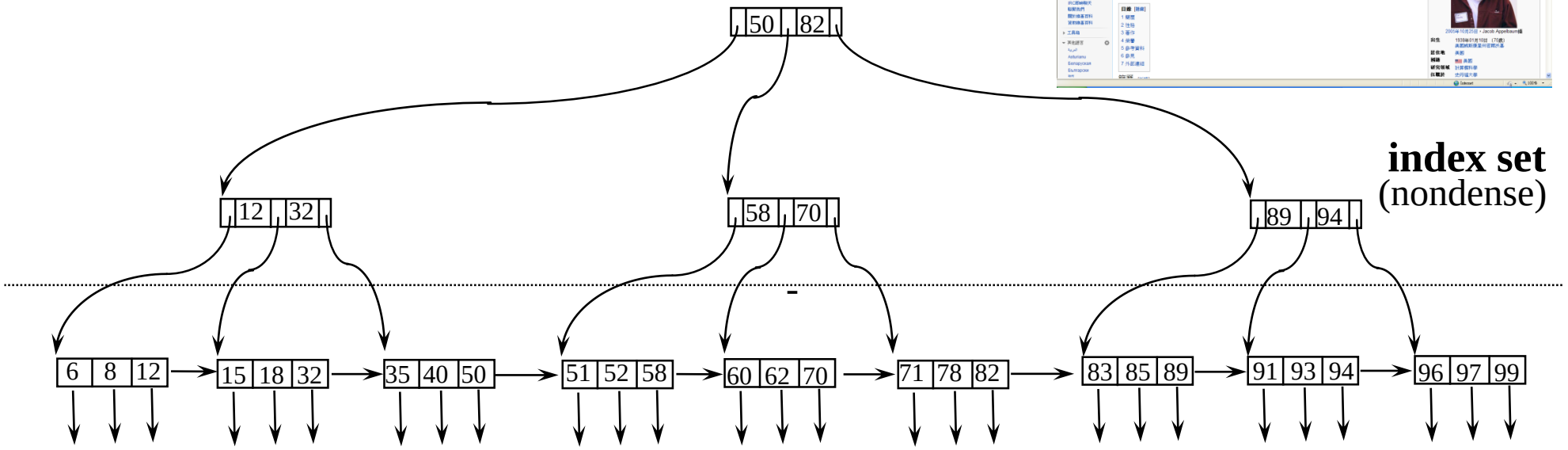


- **Goal:** maintain balance of index tree by dynamically restructuring the tree as updates proceed.





B^+-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.

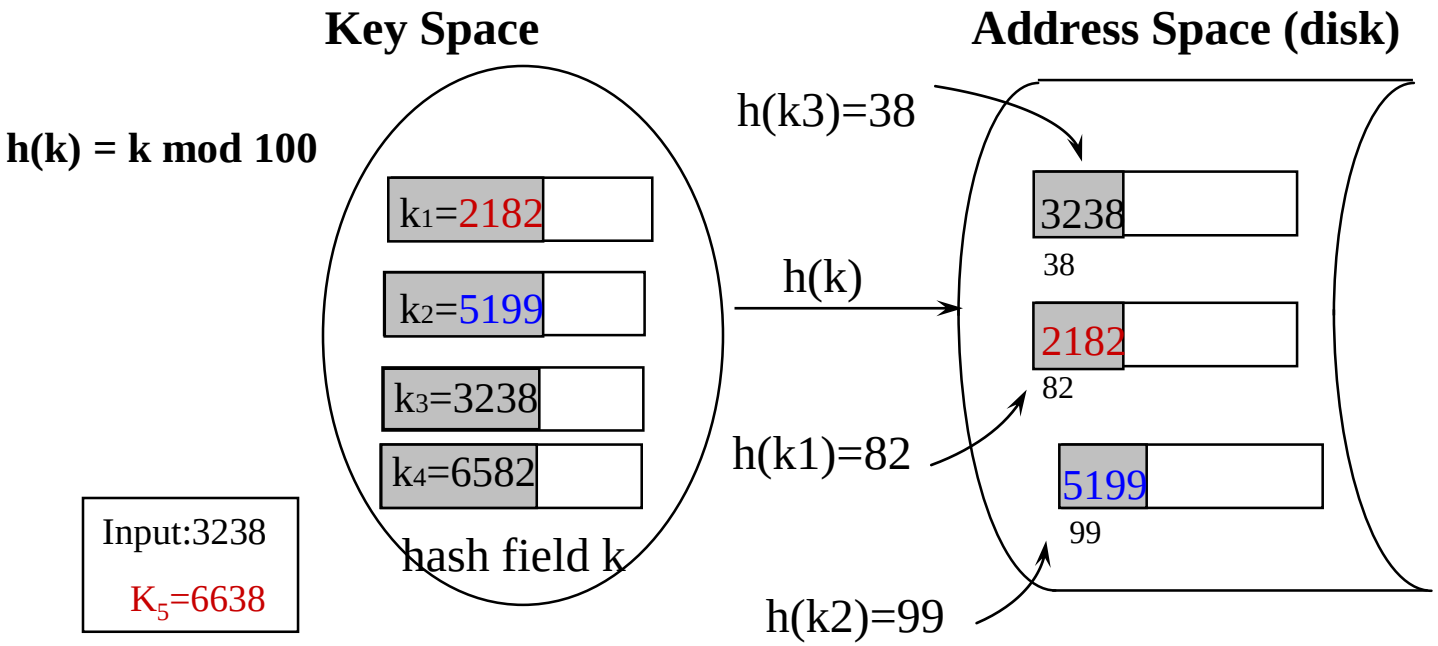
- Other variations: B*-tree, B'-tree,...

11.3 Hashing

- 11.3.1 Dynamic Hashing
- 11.3.2 Extendible Hashing
- 11.3.3 Linear 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



h: map key set to address space.

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')

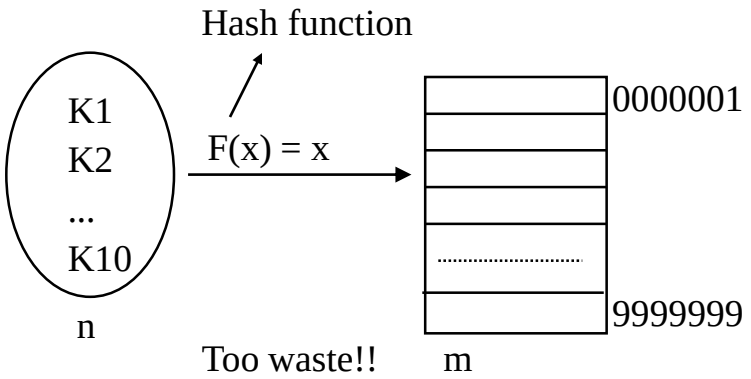
- **Common algorithms:**
  - Division Method*:*

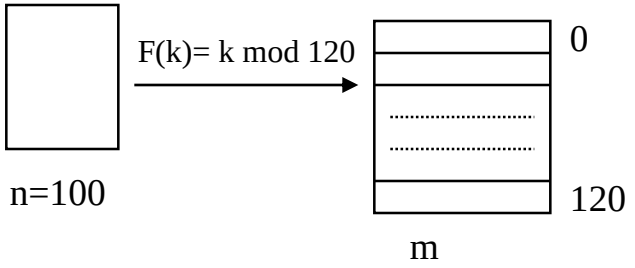
    H(k) = modulo (k /n)

    e.g. H(k) = k **mod** 100
  - Mid-square method*:*

    H(k) = central digits of K^2

    e.g. k = 525 K^2= 275625
  - *Others: .....*
- **Observation:**
  - Division method is good enough.



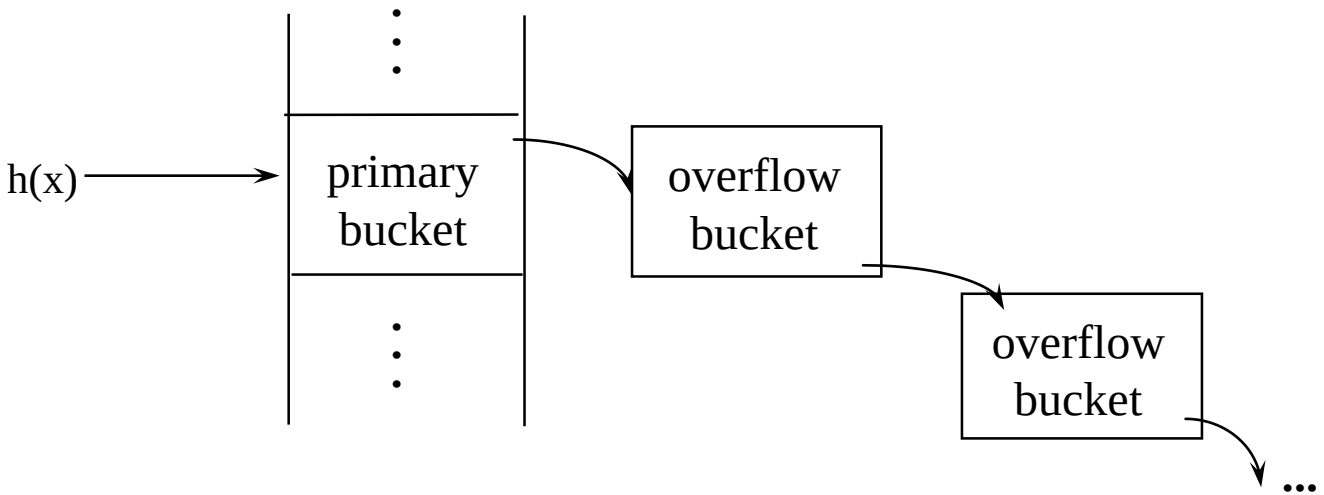


$\alpha = n/m \sim 0.8$

loading factor

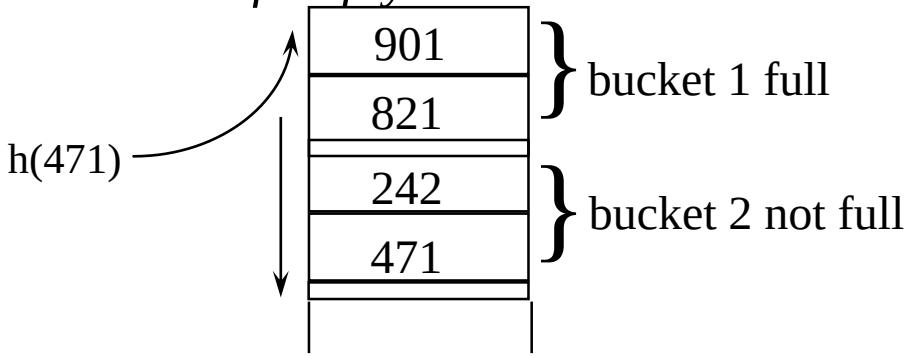
Overflow Handling in Hashing

- **Overflow chaining:**
  - allocate new bucket and chain to overflow bucket.



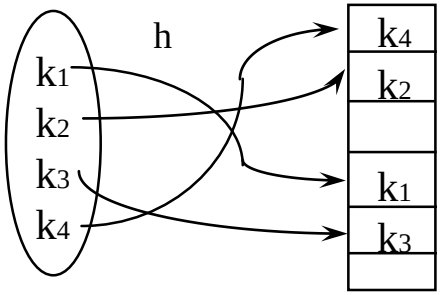
Overflow Handling in Hashing (cont.)

- **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($b_1, b_2, ..., b_n$) = m + b_1t(β_1 ; $\bar{b}_1$) + b_2 * (β_1, β_2; b_1 , $\bar{b}_2$) + ...
- 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) = $\lfloor C / (Dk + E) \rfloor$ mod n
  [Chang 84] h(k_i) = C mod 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

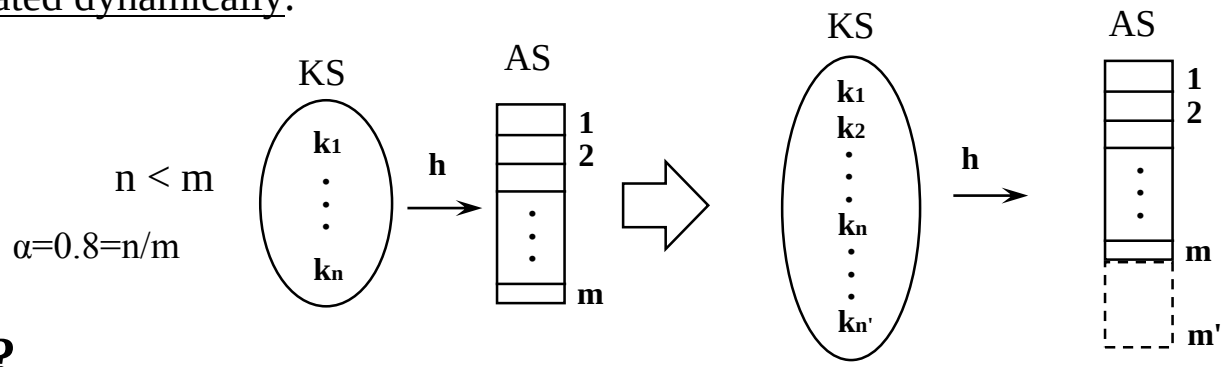
A BACKTRACKING METHOD FOR CONSTRUCTING PERFECT HASH FUNCTIONS FROM A SET OF MAPPING FUNCTIONS

W. P. YANG and M. W. DU

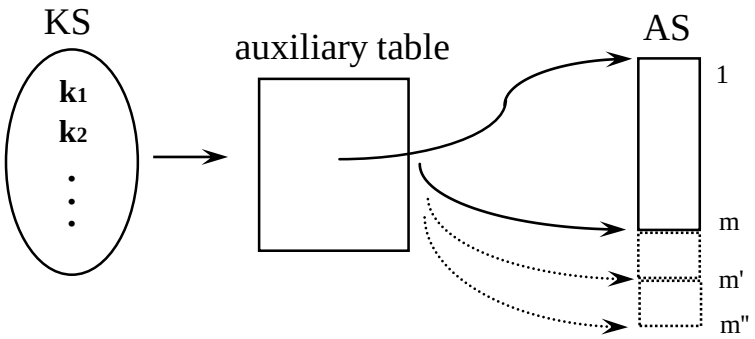
Institute of Computer Engineering, National Chiao Tung University, 45 Po Ai Street, HsinChu, Taiwan, Republic of China

11.3.1 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.

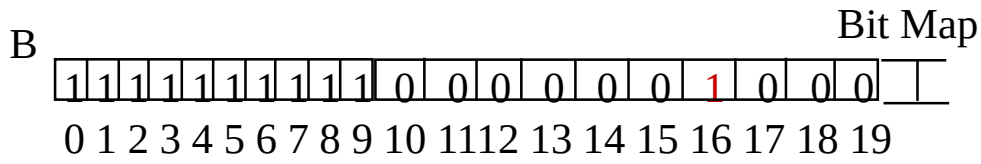
(8) • • •

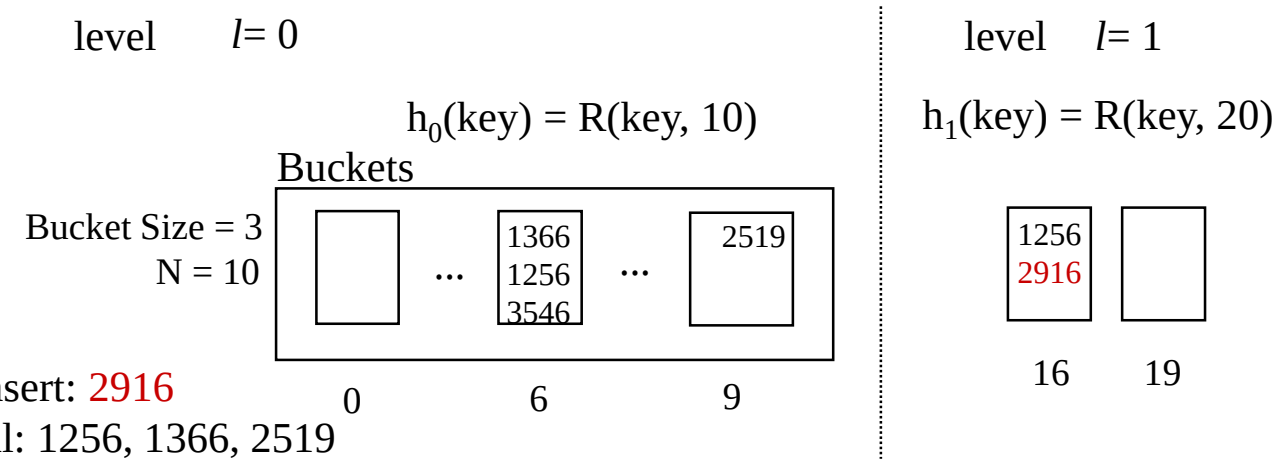
Virtual Hashing

(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,...}





Virtual Hashing (cont.)

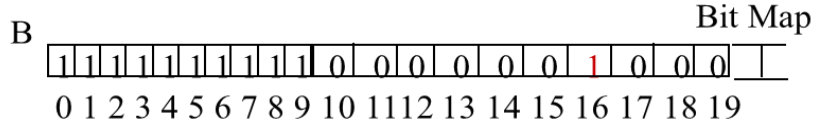
- General hash function: $h_l = R(key, 2^l \cdot N)$
- Algorithm Addressing (key)

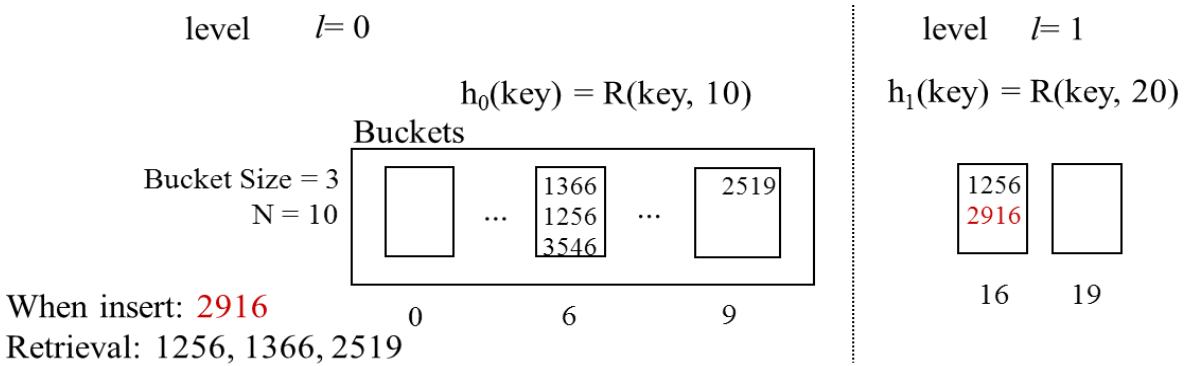
1. $j \leftarrow$ Level of h_j (Max j used)
2. m $\leftarrow$ R(key, $2^l \cdot$ N)
3. while B(m) = 0
   - $l \leftarrow l - 1$
   - m $\leftarrow$ R(key, $2^l \cdot$ N)
4. Return (m)

$2^0 \cdot 10 = 10$

h_2(key) = R(key, 40)

Example: key set:
{1366, 1256, 2519, 3546, ..., 2916,...}





11.3.2 Extendible Hashing

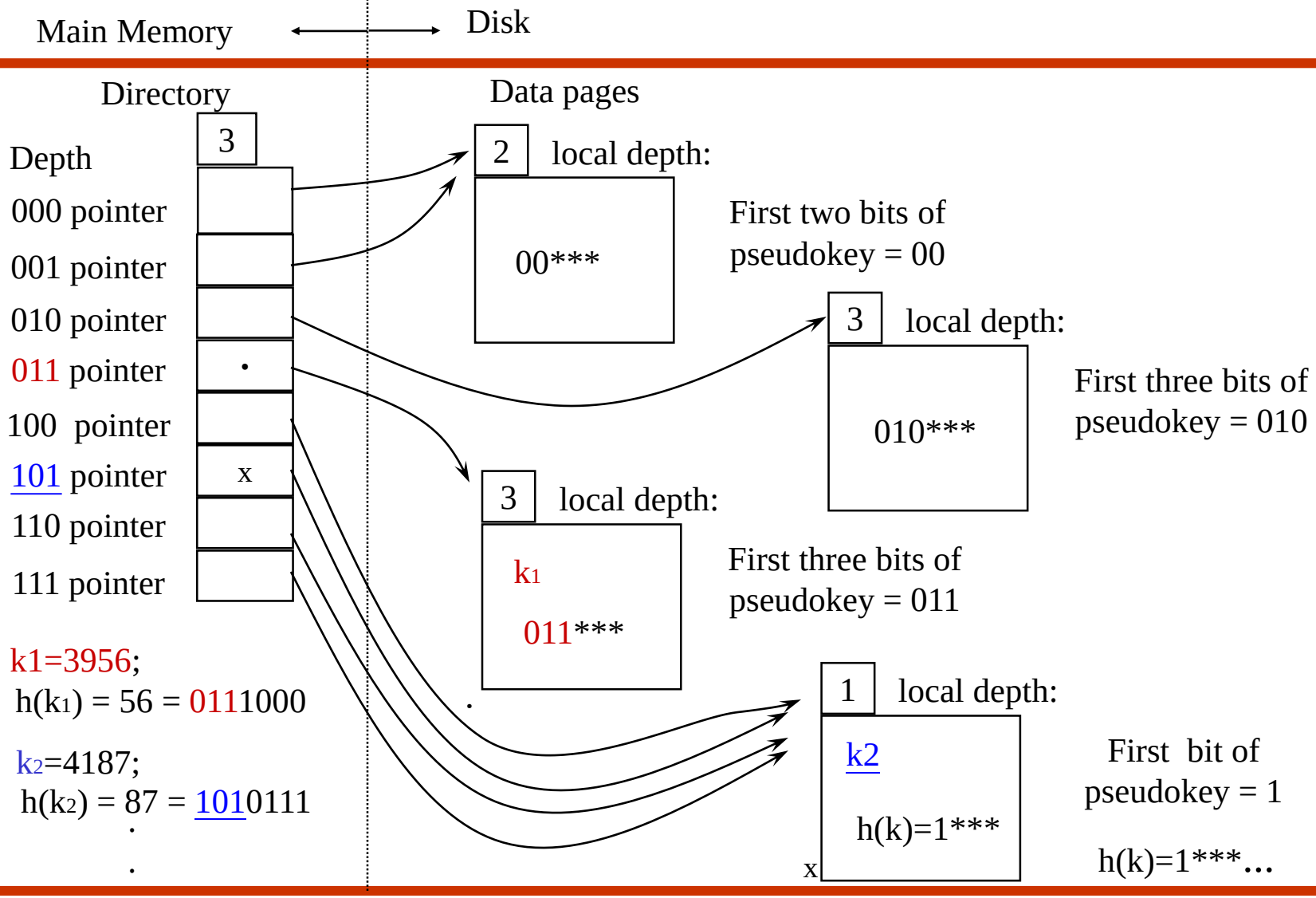
(Ref: Fagin, R. et. al. "Extendible Hashing-A fast access method for dynamic files", ACM TODS, Vol.4, #3 Sept. 79)

- **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.

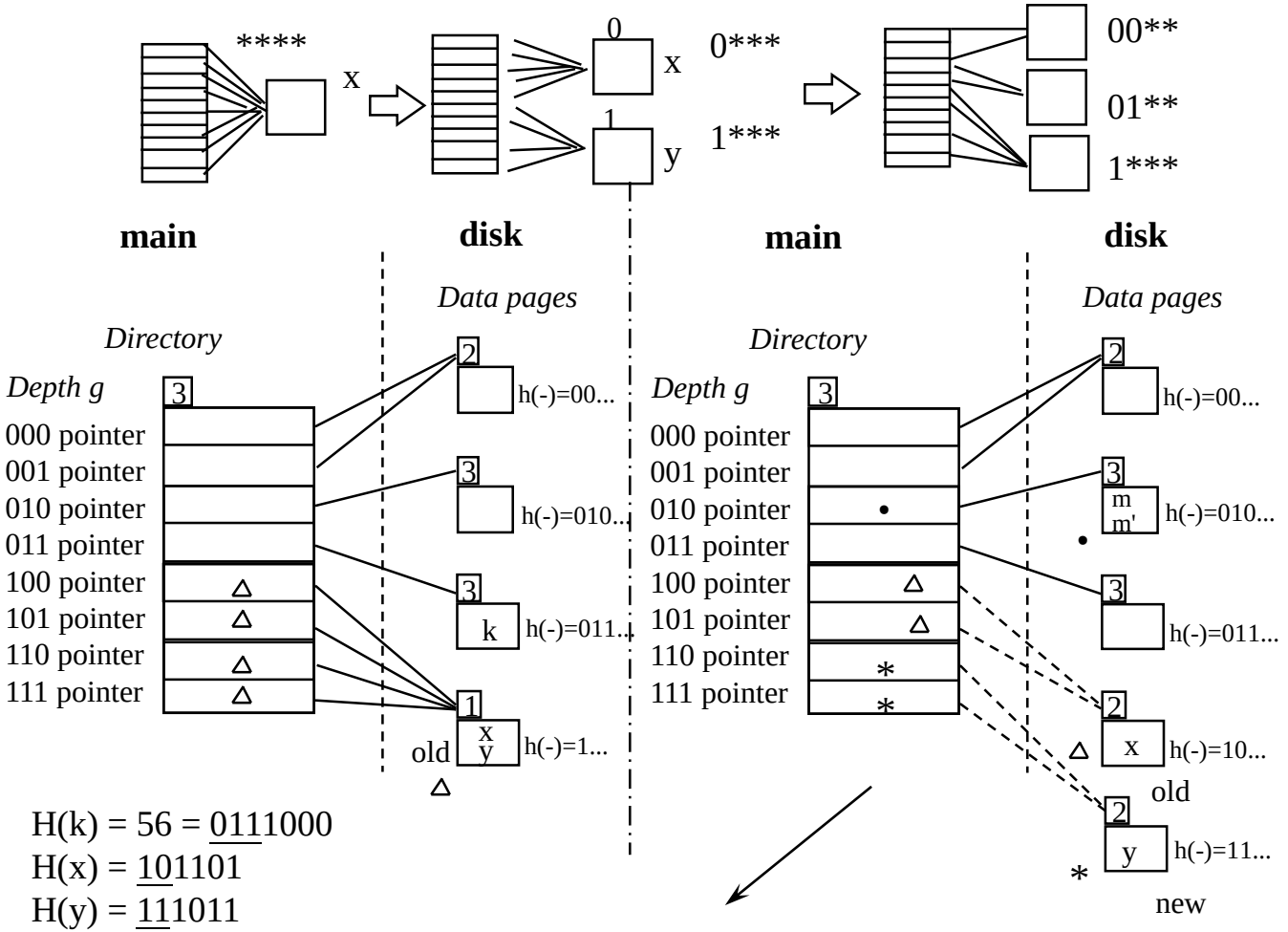
    -> total number of entries in directory = 2^d
  - Each entry in directory points to a bucket.
  - Each bucket x has a local depth $l_x \leqq$ d
  - When a bucket x overflows --> increasing l_x

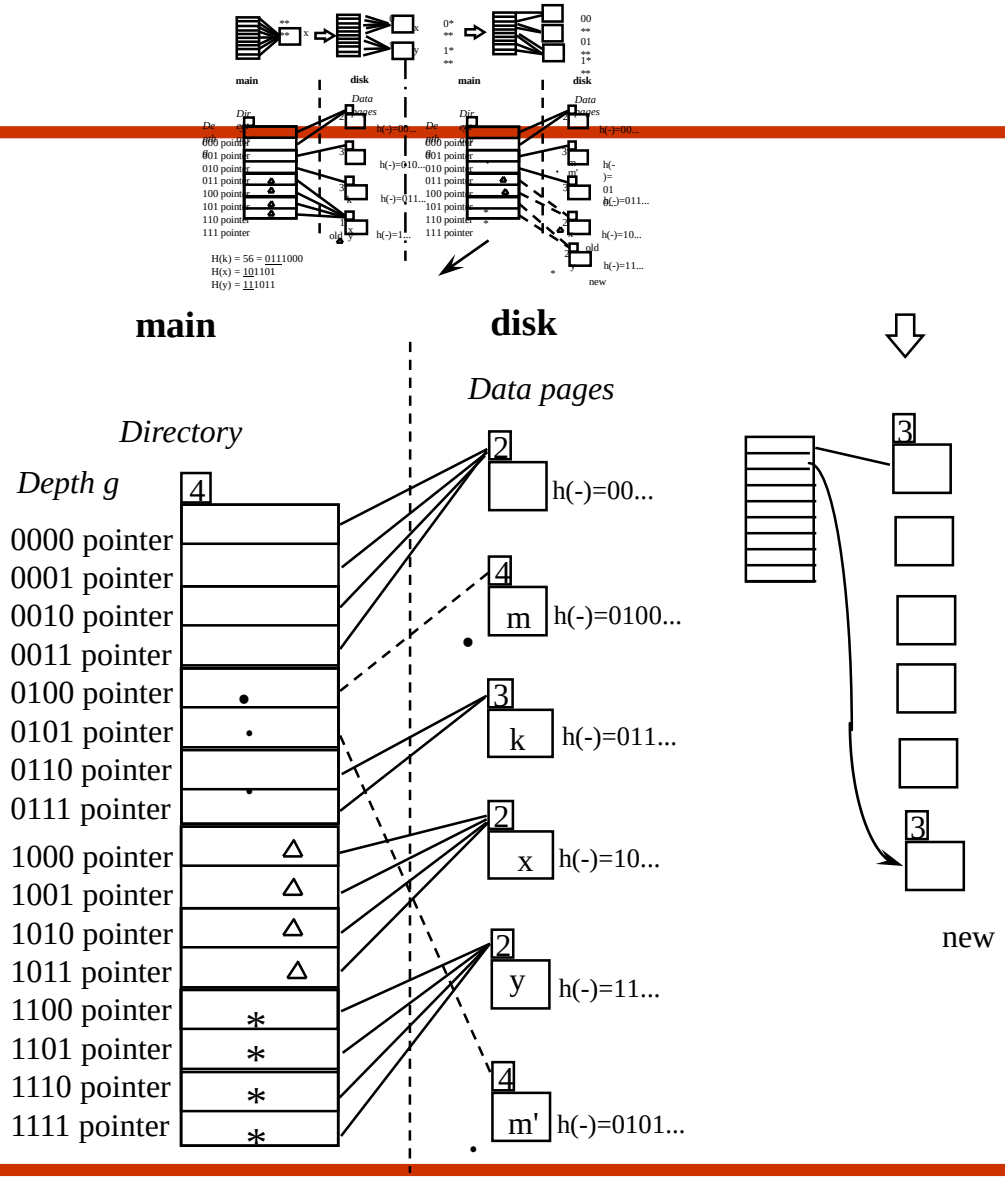
    if $l_x >$ d -->double directory (i.e. increasing d).

Extendible Hashing: Example



Extendible Hashing: Example (cont.)




main
disk
Data pages
Directory
Depth g
4
0000 pointer
0001 pointer
0010 pointer
0011 pointer
0100 pointer
0101 pointer
0110 pointer
0111 pointer
1000 pointer
1001 pointer
1010 pointer
1011 pointer
1100 pointer
1101 pointer
1110 pointer
1111 pointer
2
h(-)=00...
4
m
h(-)=0100...
3
k
h(-)=011...
2
x
h(-)=10...
2
y
h(-)=11...
4
m'
h(-)=0101...
3
new

Our Research Results

Concurrent Operations in Extendible Hashing [VLDB86]

Meichun Hsu
Wei-Pang Yang

Harvard University
Cambridge MA 02138

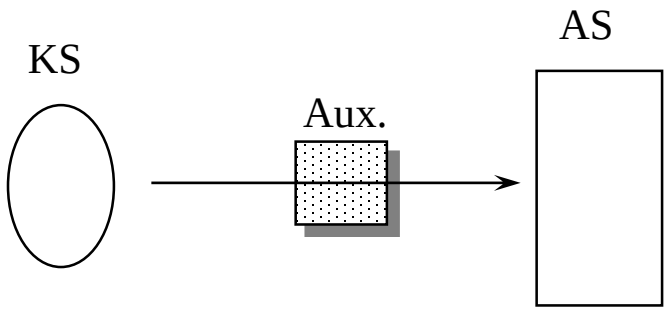
Abstract

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.

11.3.3 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, ...}

$H_l(k) = mod(k, N*2^l)$

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

N = 4

l = 0

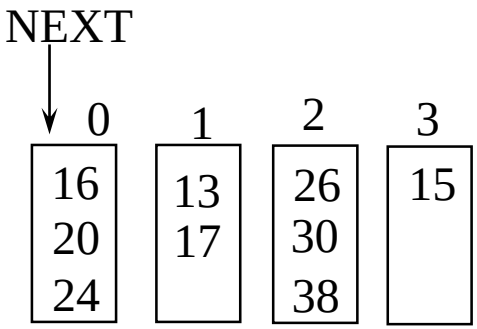
⇨ $H_0(k) = k \bmod 4$

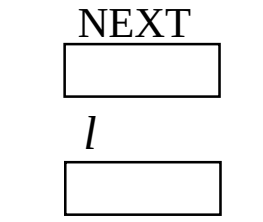
⇩ insert '42'

bucket 2 overflow

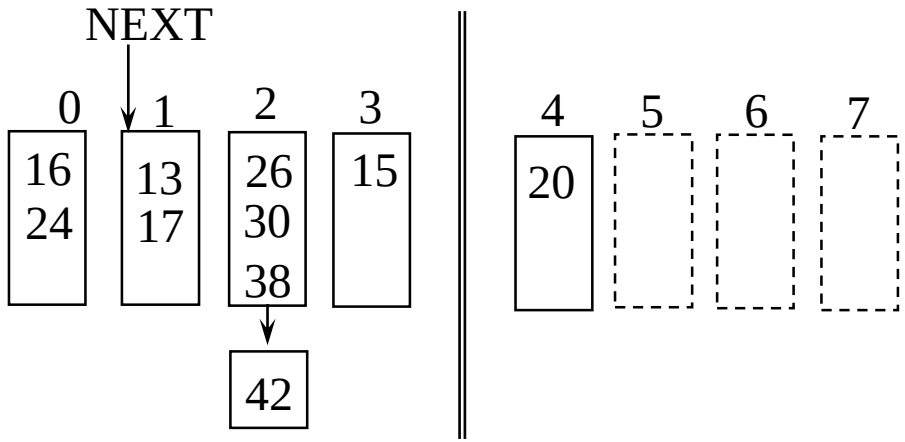
split bucket 0 (the first)

l = 1 --> $H_1(k) = k \bmod 8$



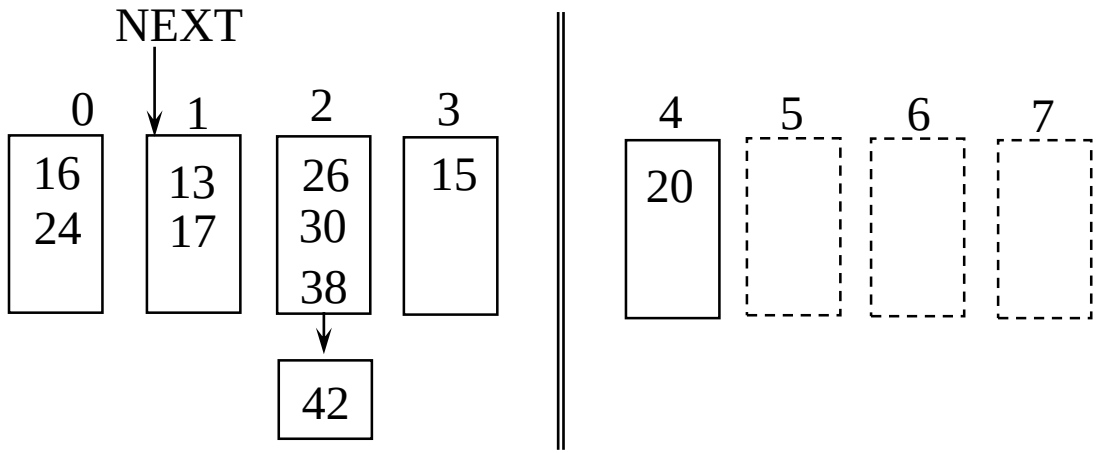


Function: $H_i( )$



Linear Hashing: Example (cont.)

l = 1 --> $H_1(k) = k \bmod 8$



- Max. bucket # = $N*2^{l-1}$ + NEXT-1 = $2^0*4+1-1 = 4$

- retrieve 15 --> $H_1(15) = 15 \bmod 8 = 7 > 4$
  --> $H_0(15) = 7 - 4 = 3 \leq 4$
- 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.

11.4 Pointer Chains

S

S#	SNAME	STATUS	CITY
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

Pointer Chains

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

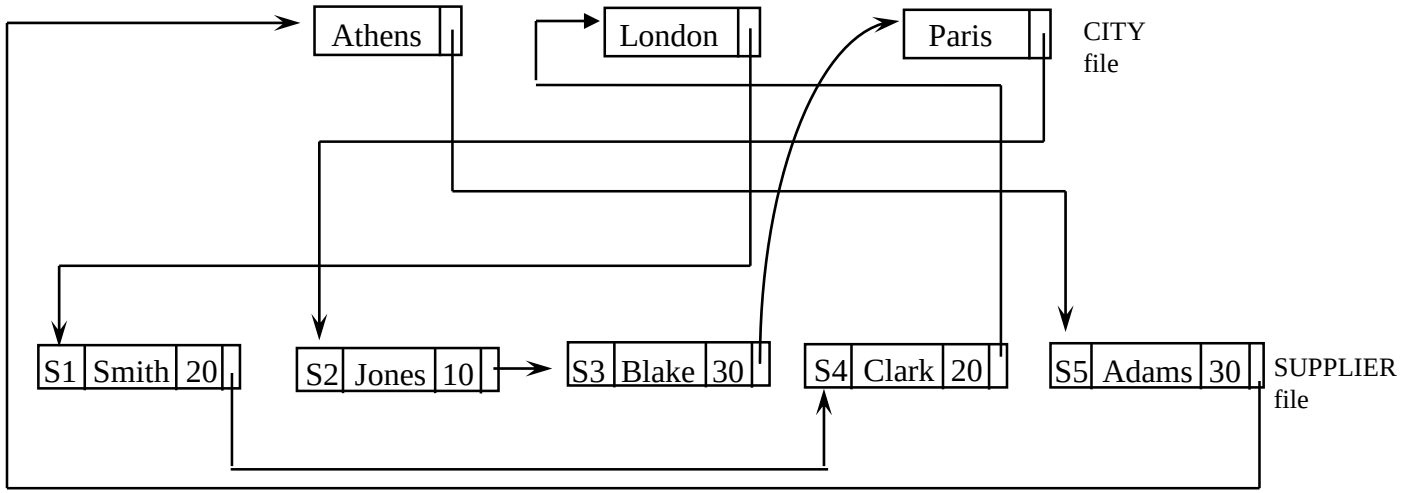


Fig. 11.6: Example of a parent/child structure

- **Advantages:**
  - 1. insert, delete are simpler and more efficient than index structure. (§11-2, Fig. 11-2)
  - 2. occupy less storage than index structure. (Fig. 11-2)
- **Disadvantages:**
  - to access the nth supplier --> sequential access --> slow !.

Pointer Chain vs. Indexing

Compare:

- insert, delete
- Storage
- access the *n*th record
- sequential access

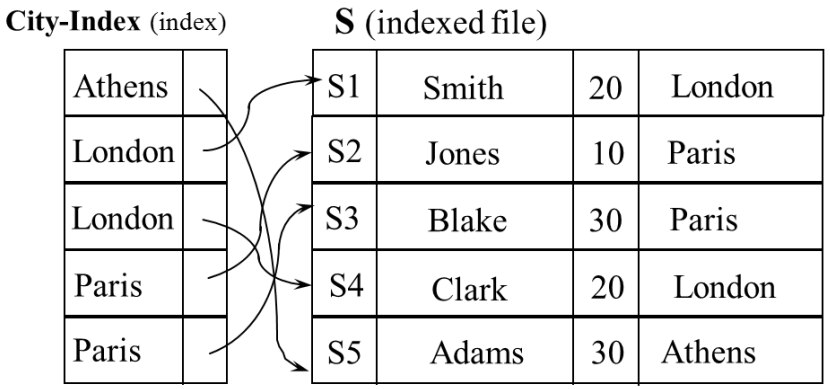


Fig. 11.2: Indexing the supplier file on CITY.

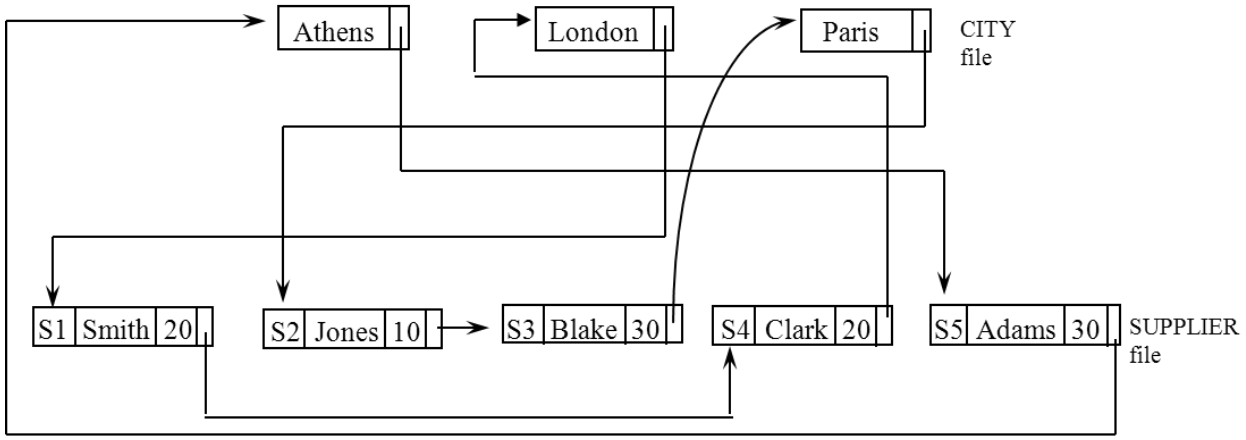
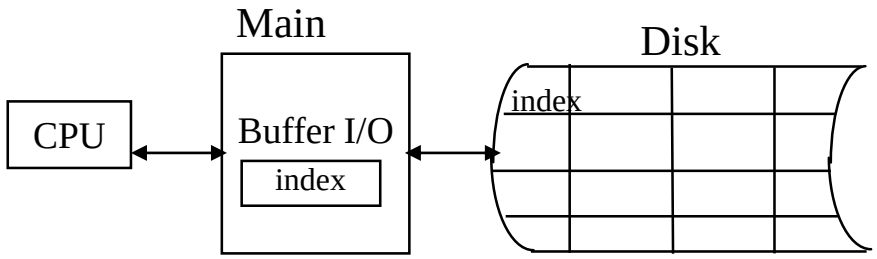


Fig. 11.6: Example of a parent/child structure

11.5 Compression Techniques

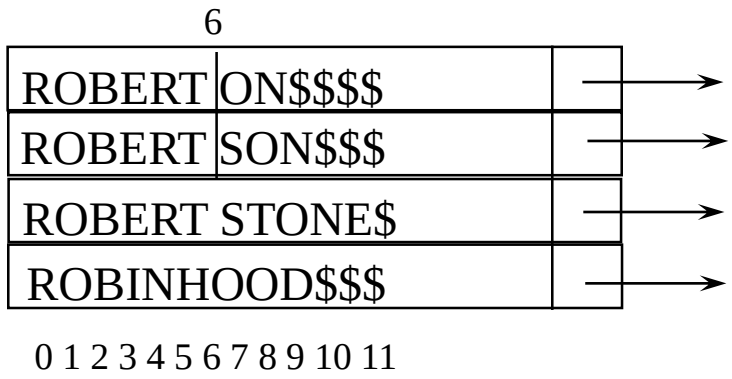
Objective : reducing storage space

--> reducing disk I/O



Differential Compression

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



-Method 1

front compression:

0 - ROBERTON$$$$	ROBERTON$$$$
6 - SON$$$	ROBERTSON$$$
7 - TONE$	ROBERTSTONE$
3 - INHOOD$$$	ROBINHOOD$$$

-Method 2

rear compression: eliminate blanks, replaced by a count.

0 - 7 - ROBERTO	ROBERTO????
6 - 2 - SO	ROBERTSO????
7 - 1 - T	ROBERTST????
3 - 1 - I	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.

<e.g.1> intra-file

S

S#	SNAME	STATUS	CITY
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

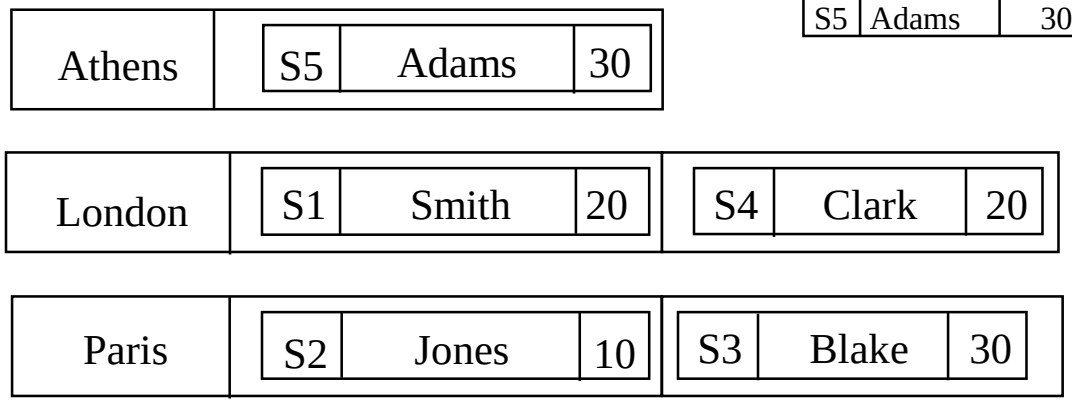
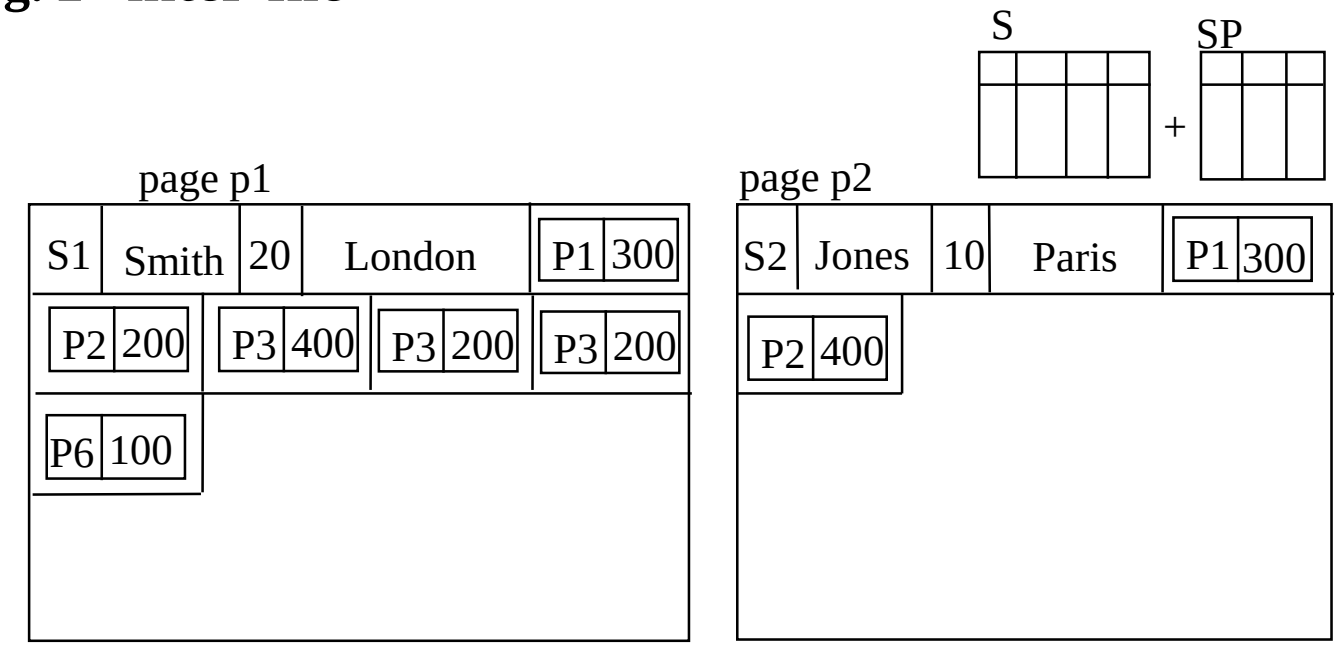


Fig. 11.7: Example of hierarchic compression (*intra-file*)

Hierarchic Compression (cont.)

<e.g. 2> inter-file



S

S#	SNAME	STATUS	CITY
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

SP

S#	P#	QTY
S1	P1	300
S1	P2	200
S1	P3	400
S1	P4	200
S1	P5	100
S1	P6	100
S2	P1	300
S2	P2	400
S3	P2	200
S4	P2	200
S4	P4	300
S4	P5	400

(and similarly for pages p3, p4, p5)

Fig. 11.8: 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: 010 100 011
    C E D

  - ***Method 2:***

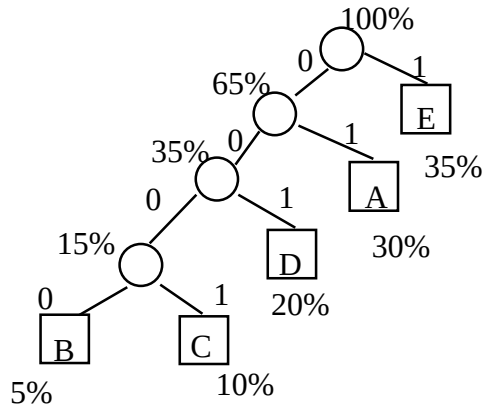
    A: 1
    B: 01
    C: 001
    D: 0001
    E: 0000

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

11.6. 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.

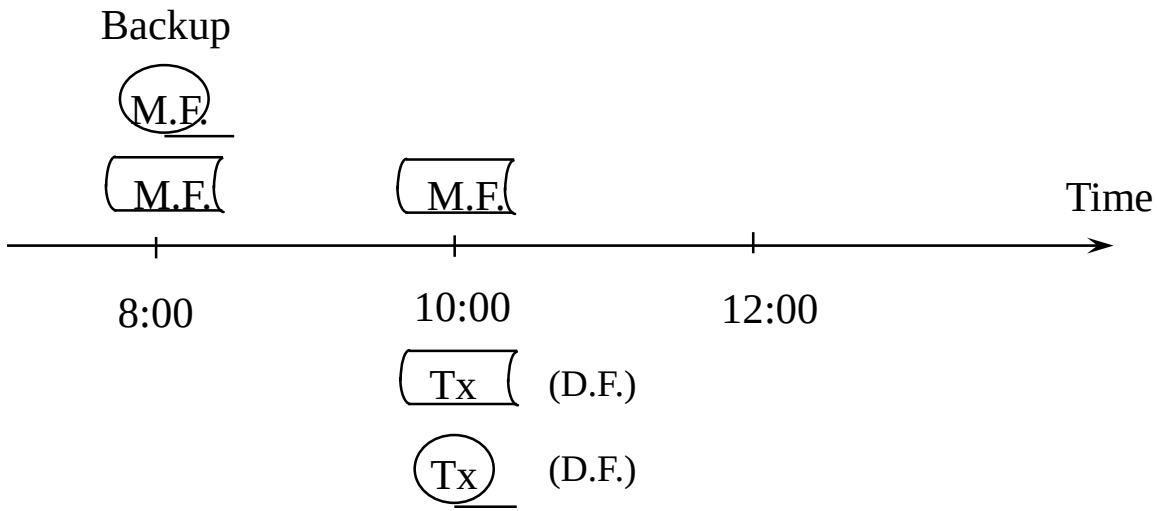
2. "A practical guide to the design of differential files for recovery on-line database", by H. Aghili and D.G. Severance, TODS, 7, 4, Dec, 1982)

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:

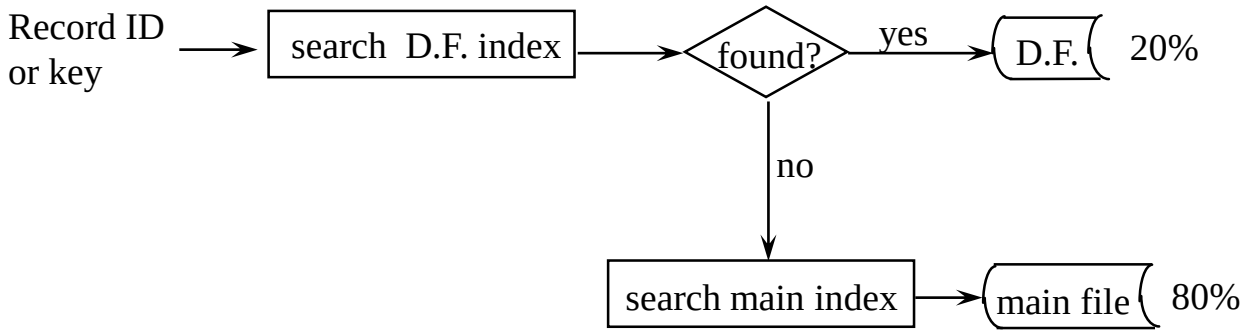
  <1> search in the differential file first.

  <2> if not found, then search in the main file.
- Motivation
  - reduces dumping cost for backup and recovery



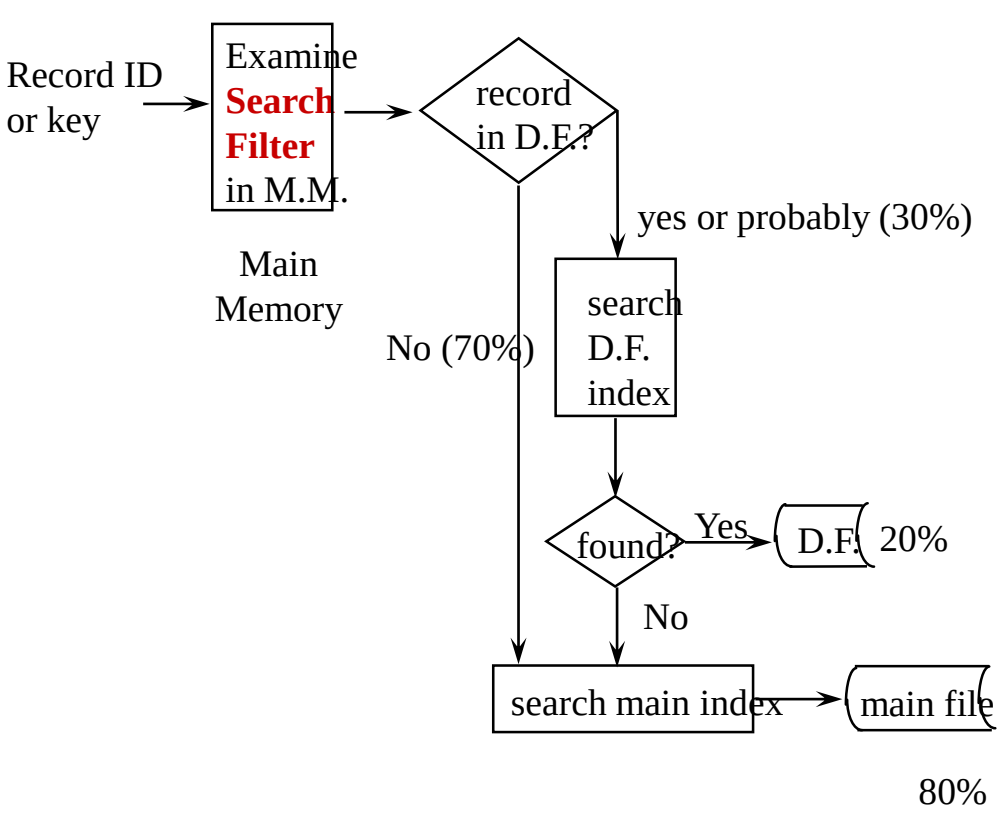
Differential File (cont.)

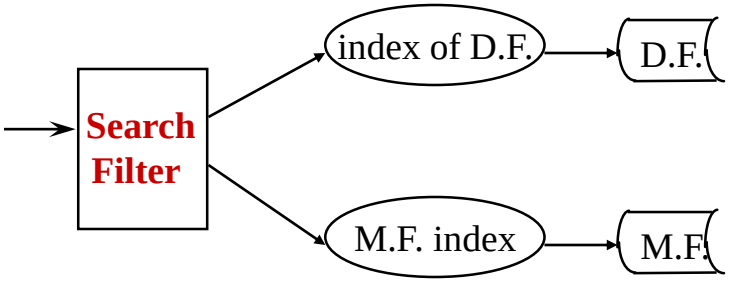
- How to access when using differential file?



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

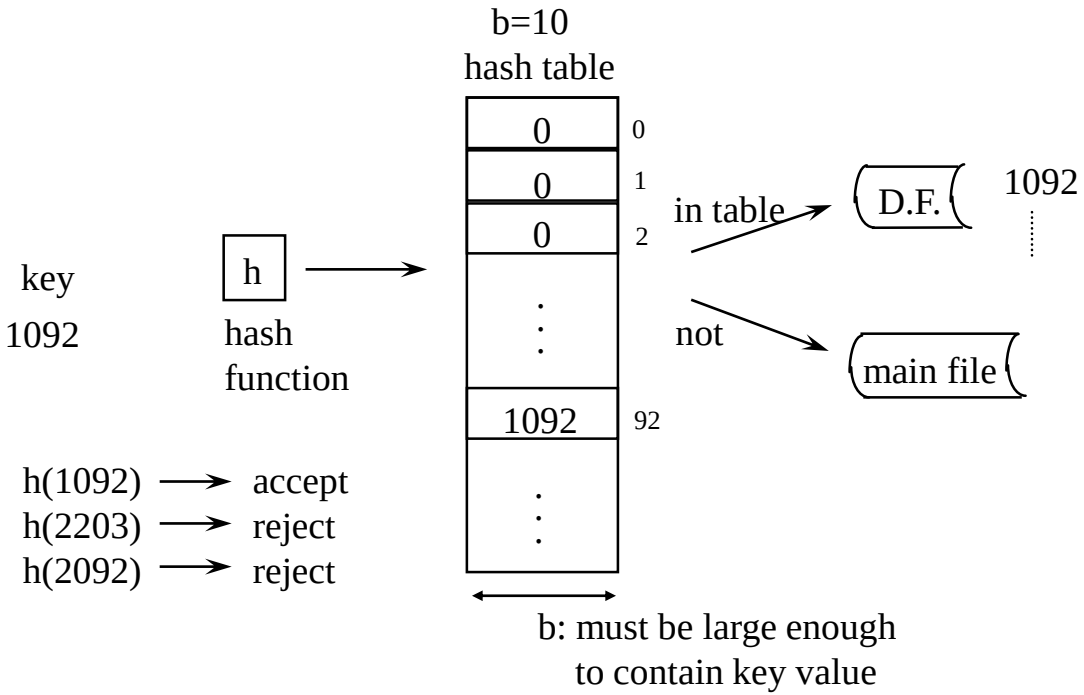
Search Filter





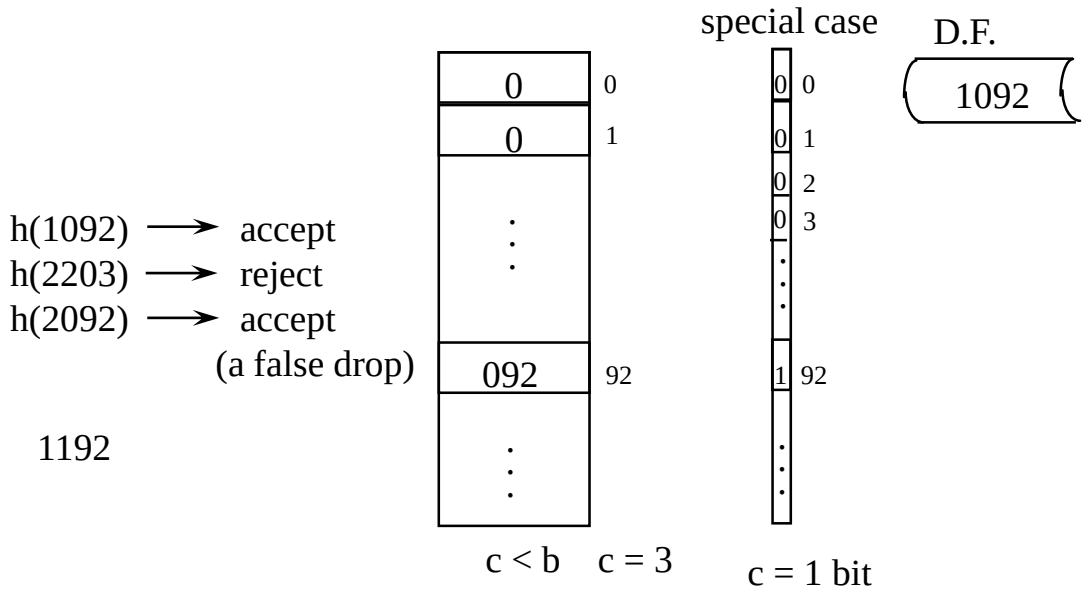
Search Filter: Method 1

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



Search Filter: Method 2

- **Method 2:**



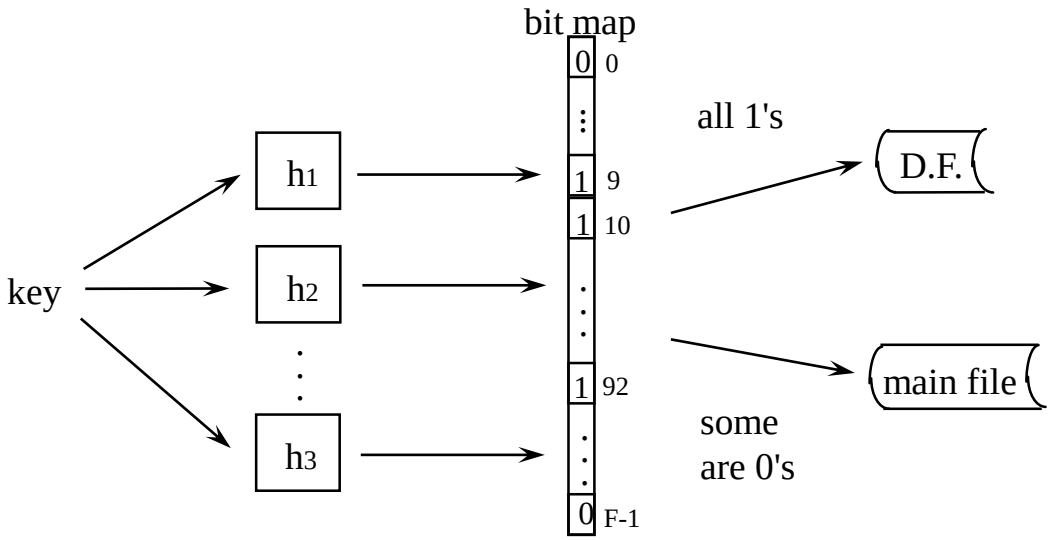
-Advantage: smaller space overhead

-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> $h_1(1092) = 92$
$h_2(1092) = 09$
$h_3(1092) = 10$

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

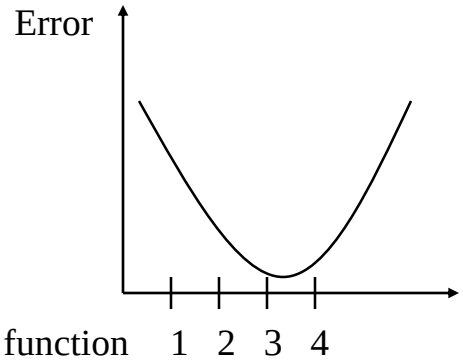
–fails drop probability depend on

-k: # of hash function used

-F: size of hash space

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

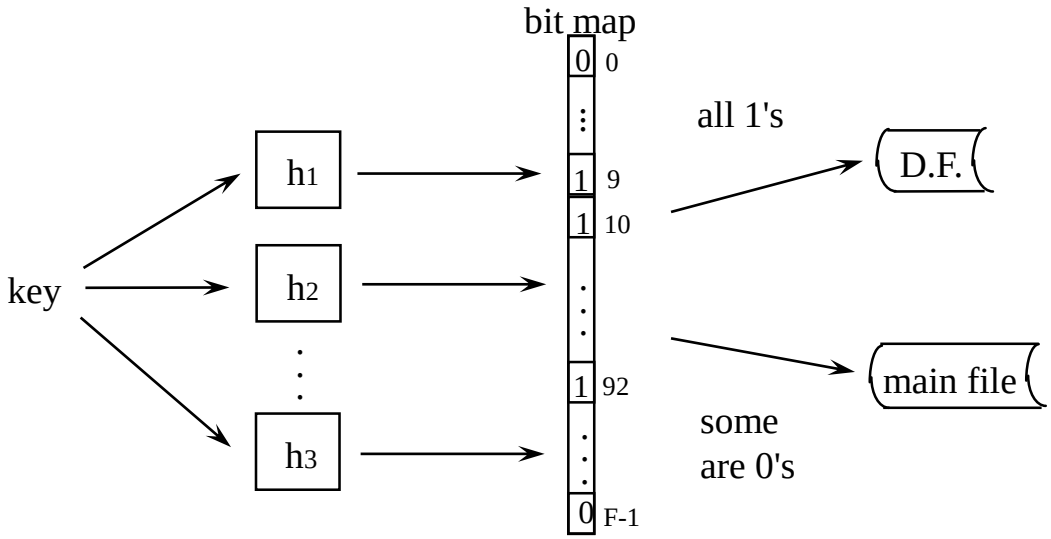
<e.g> : k=3
N = 1000
F = 5000 → $P_{fd} = 0.0919$
F = 10000 → $P_{fd} = 0.0174$



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.



end of unit 11