我想要期末及格就好

磁盤儲存、基礎資料結構

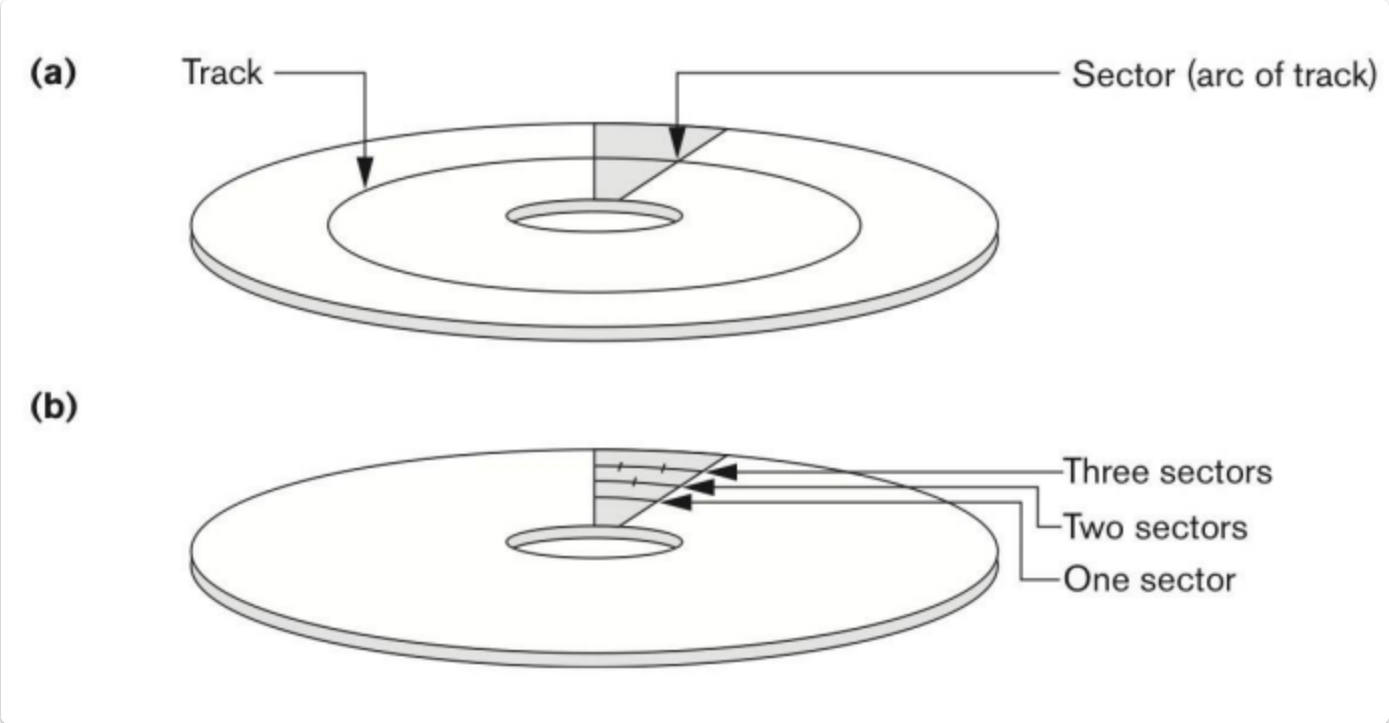
儲存媒體的分類

一級：快取、緩存
二級：硬碟、CD
三級：磁帶、黑膠唱片

HDD

- 軌道會被分成成更小的塊或扇片
  - 因為一個軌道常包含很多的資料
- 分配的方法被寫在磁碟上，無法被更改

兩種扇區的方法



將磁軌（track）劃分磁盤塊（block）的操作是在格式化它的時候
每一個塊是以固定尺寸大小 interblock gaps 分隔的

- 其中也包含初始化的編碼控制編碼

- 整個塊會和主記憶體傳輸

讀寫頭會在要傳輸的塊的磁軌**移動**

- 磁盤也會轉動在讀寫頭下的塊進行讀寫
- 兩種磁盤
  - 固定磁頭的磁盤 (有跟磁軌一樣多的磁頭)
  - 可移動磁頭的磁盤

硬碟控制器會控制硬碟驅動與電腦連接

磁碟區塊位址為以下組成

- 圓盤編號 (紀錄表面相同半徑軌道的集合)
- 軌道編號
- 塊編號

讀寫都是有時間花費，搜尋時間 (s)、旋轉延遲 (rd)。

- **Double buffering** 可以加快連續磁碟塊的傳輸

總時間 $\rightarrow$ 定位時間+傳輸任意塊時間

- 尋找時間
- 旋轉時間或淺在因素
- 塊傳輸時間 (上兩個因素佔的時間會比傳輸時間多上更多)

批量傳輸速率時間，為傳輸連續塊的時間

在磁碟定位資料是一個資料應用主要的瓶頸

- 將所需數據從磁盤定位和傳輸到主內存所需的塊數量最小化非常重要
- 將**有關聯的資訊**放在連續的塊是磁盤上主要的基本目標
  所以有了以下有效率訪問的目標
- 資料緩衝：雙緩衝策略
- 洽當的資料組織：有關連的資料放在連續塊
- 提前讀取要求

- 恰當的安排 I/O 的要求
- 可以記錄磁盤的讀寫：可以分配單個磁盤來記錄 (可以消除尋找時間)
- 用 SSD 或 flash 記憶體

SSD

最近的趨勢會是使用 flash memories (SSDs)，去當主內存跟 HDDs 的中間層

SSD 上**沒有放置的限制**，因為任何位址都可以直接尋找

- 因此資料不可能碎片化，所以資料不需要重組

通常 HDD 在寫入時，同一塊會被新數據覆蓋
而 SSD 會用不同的 NAND 實現磨損均衡，延長 SSD 的壽命

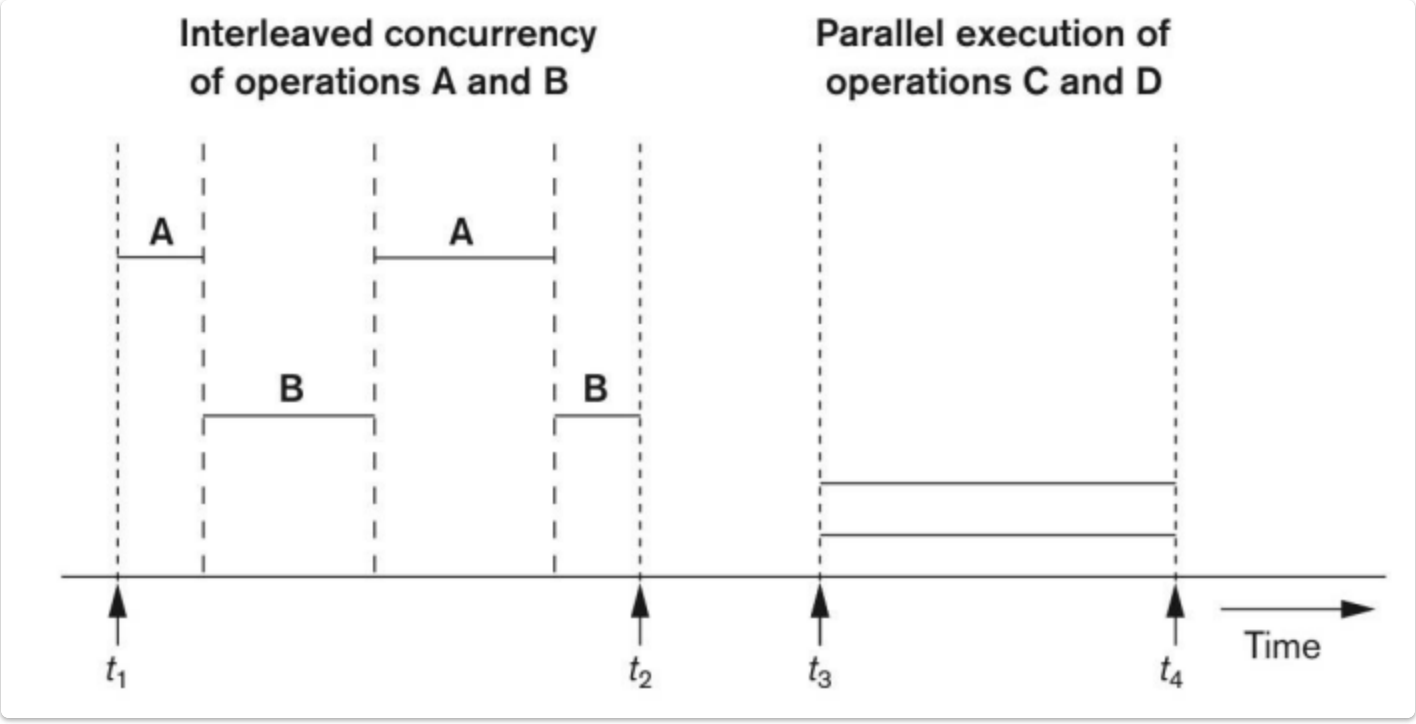
Magnetic Tape Storage Device 磁帶儲存裝置

- 磁碟是隨機訪問，**磁帶是連續的訪問**
- 磁帶儲存一樣是**用塊來儲存**，但塊會比磁碟的大小還大
- 磁帶提供一個很重要的功能，就是 backing up
- 磁帶通常儲存超大的資料
- 磁帶也可以儲存**圖片**跟**系統庫**

Buffering of Blocks

傳輸已知的塊，可以用主內存的 buffer 加快傳輸速度

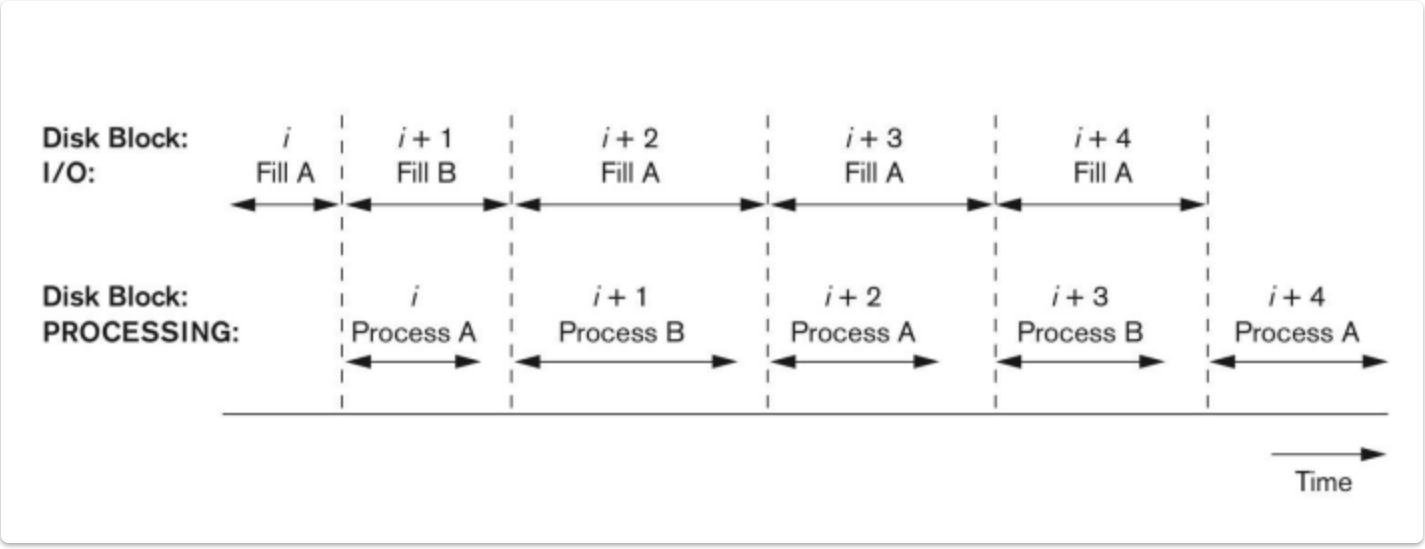
- 兩種平行傳輸方式
  + Interleaved 交錯
  + parallel 同時



當傳輸要以平行方式處理的時候，緩衝就非常有用

Double buffering

- CPU 在傳輸完後，主內存可以開始處理一個區塊，同時，磁碟的 I/O 可以讀取並傳輸另一個 buffer
- 可以用來將記憶體寫入磁碟的連續區塊
- 允許連續讀寫連續的塊上的資料，這樣可以消除第一個塊的尋找時間跟旋轉延遲
- 可以減少程式中的等待時間



Buffer manager

- 大多大型資料庫，都有很多塊的檔案，所以無法同時將所有資料放進主記憶體
- Buffer manager 是 DBMS 的軟件，用於回應資料請求並決定使用哪個 Buffer

資料的索引結構、物理資料庫設計

Primary Index

建立在有排序的檔案
ordering key 一個欄位

Clustering Index

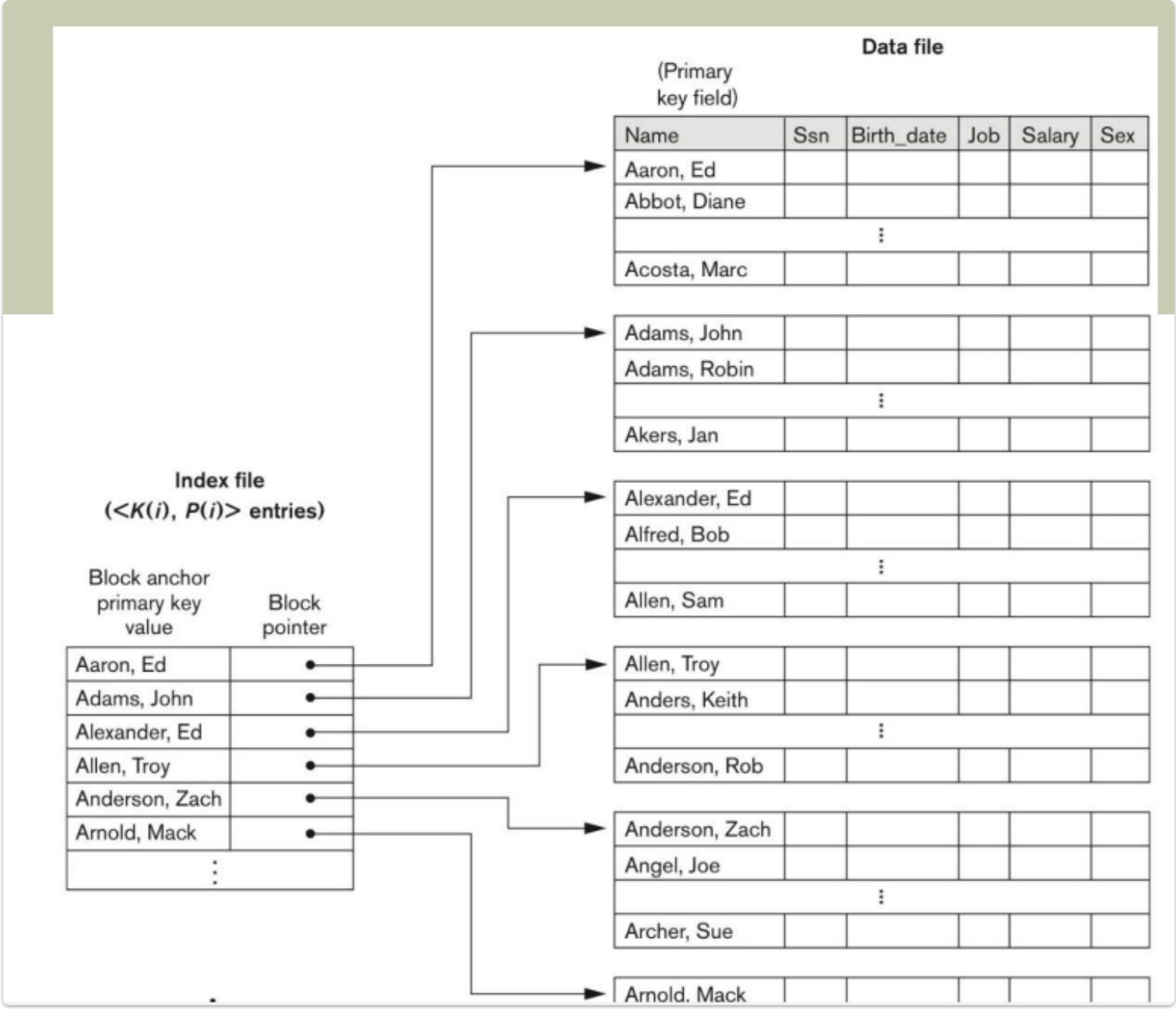
建立在有排序的檔案
ordering nonkey 一個欄位 可能是多值

Secondary Index

任何資料都可以，無須排序
欄位可以建立在沒有排序的欄位，且有可能是 key 或 nonkey
所以他可以建立多個欄位的
通常用於加速特定的查詢操作，但是可能會降低插入和更新操作的性能，因為需要在索引中插入或更新條目。

Primary Index

定義在檔案有排序的情況下
才在是 key 的欄位下
nondense
這樣每一個 block 就都可以用 Index 去尋找
搜尋到的資料會在該表的第一個 row
called block anchor
取上值
ex 搜尋 Abbot
他在前兩項中間，所以他會去搜尋上一個 row，會在她裡面



會比較少 index entries (索引數量)

bfr 一個 block 可以放 file 的數量 = 下高斯(B/R) blocksize / record length
b 多少 block = 上高斯(r/bfr) all of file / bfr
bfri 一個 block 可以放的 index 數量 = 下高斯(B/R_i) blocksize / (key + block pointer)
bi 總共放index的block的數量 = 上高斯(r_i/bfri)

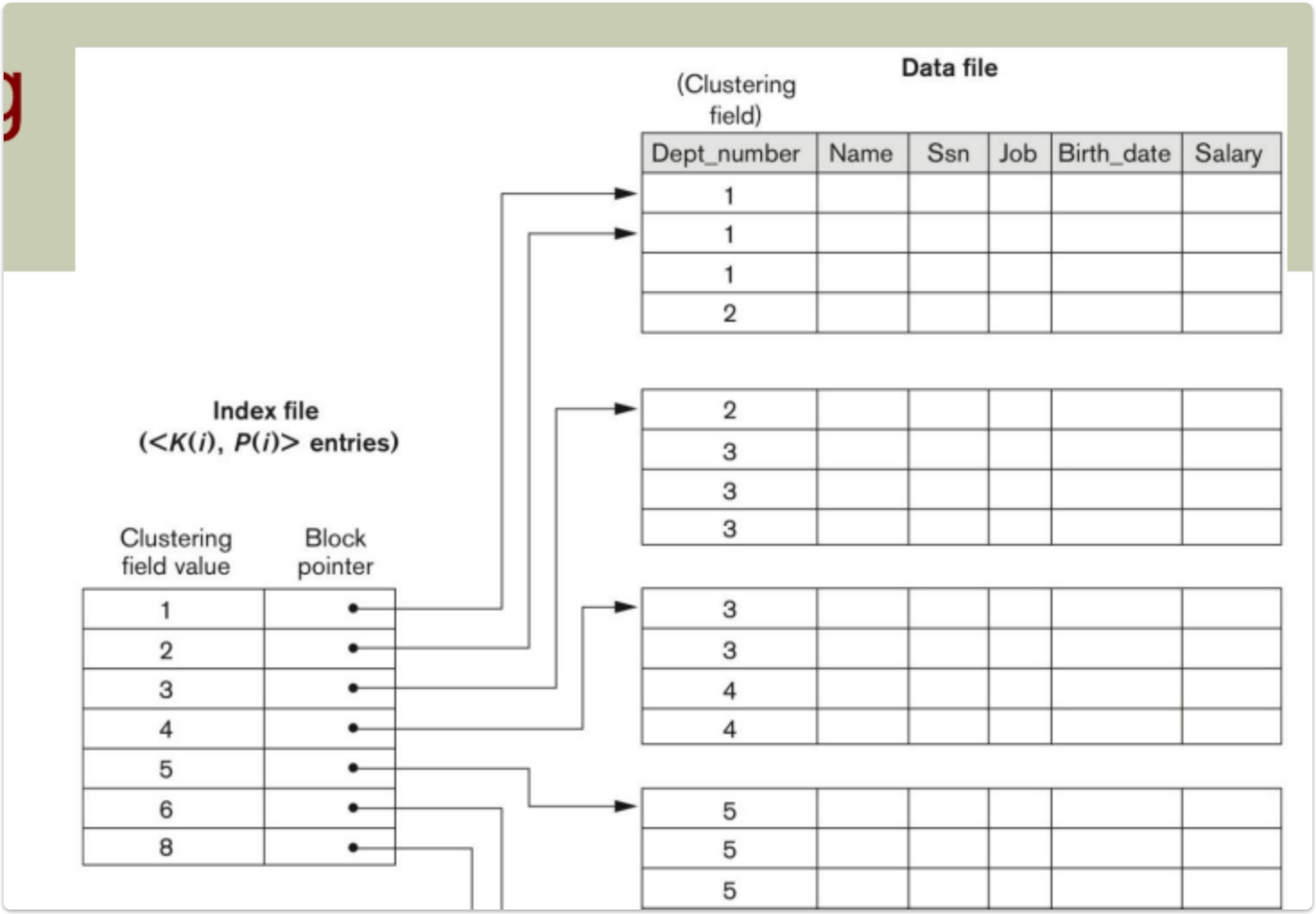
Problem

- 插入資料不夠時
  - 給定多一點的欄位數量，overflow file給他
  - 用 linked list

- 若要刪除就標記就好，不用直接刪

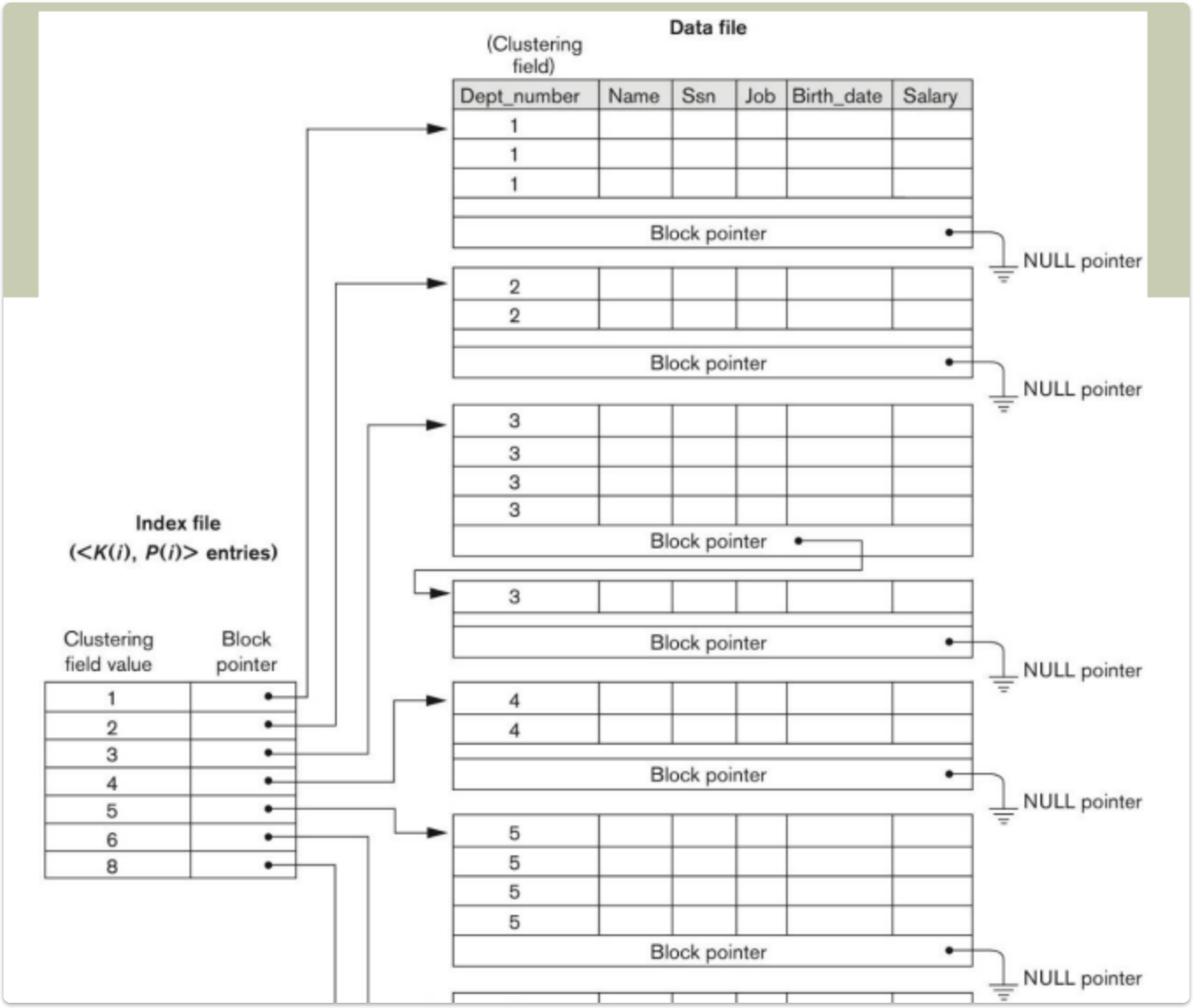
Clustering Index

檔案也需要排序過
使用 non-key，會有重複的值
nondense
用重複的東西來做分群
找到的index會是分群的第一個資料的那群，但她不一定是那個分群的資料，下圖舉例



變種的資料儲存方式，就可以讓bolck都是同一個index的值，插入時也會更好插，算

是用hash value來搜尋



Secondary Index

有無排序都可以
可以是 key 或 nonkey

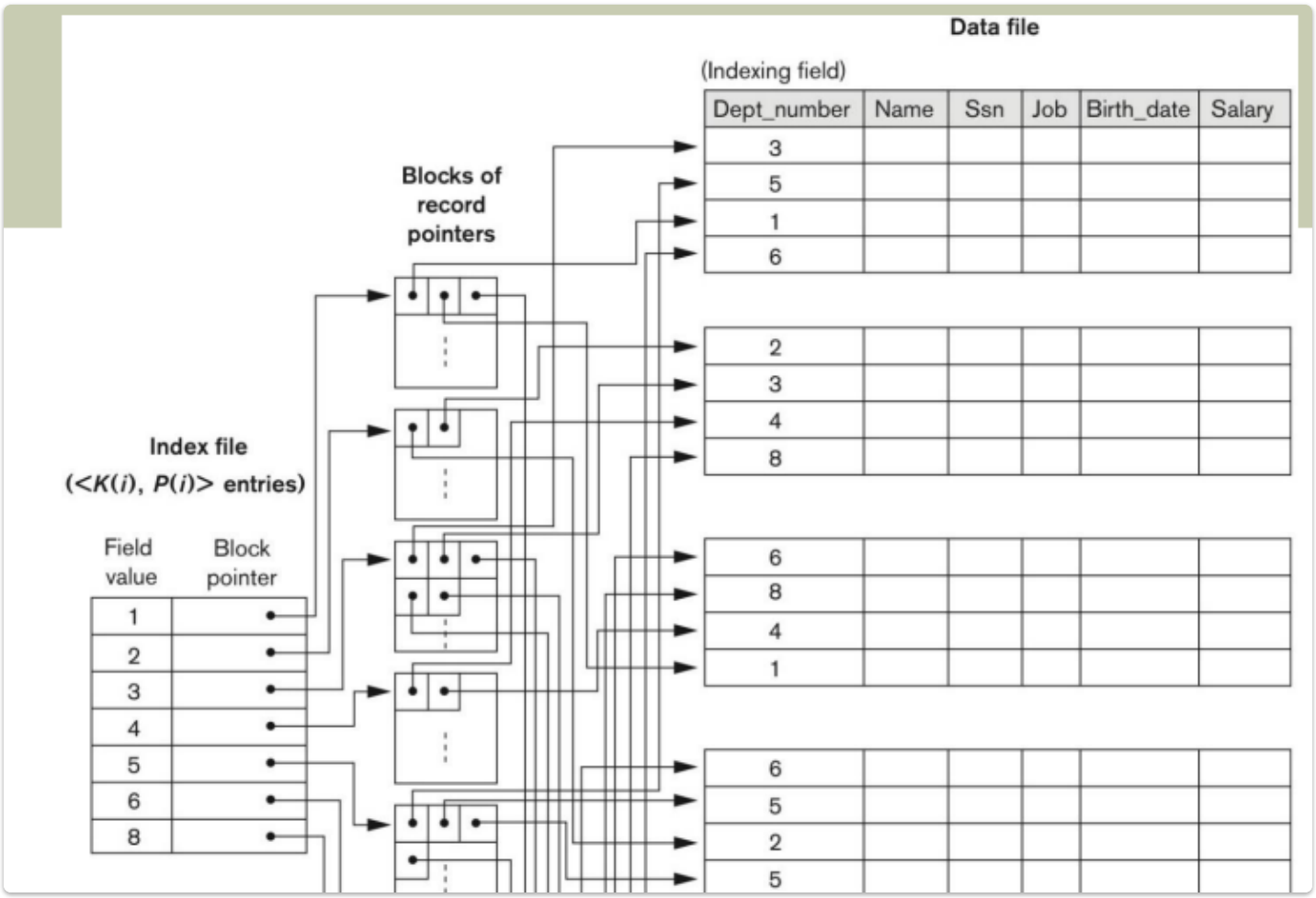
會把index 是排序
可以用 block pointer 或 record pointer
每一個key都有對應的，所以是dense

需要比較多的空間跟時間
但改進比較好，因為他不需要linear的搜尋，無須排序

當nonkey時

- dense

- nondense，同一個指標放在一起，每一個指標都有一block
- nondense，給一個額外的block放指標，然後再去對應真正的block，下圖舉例，會多讀一個block

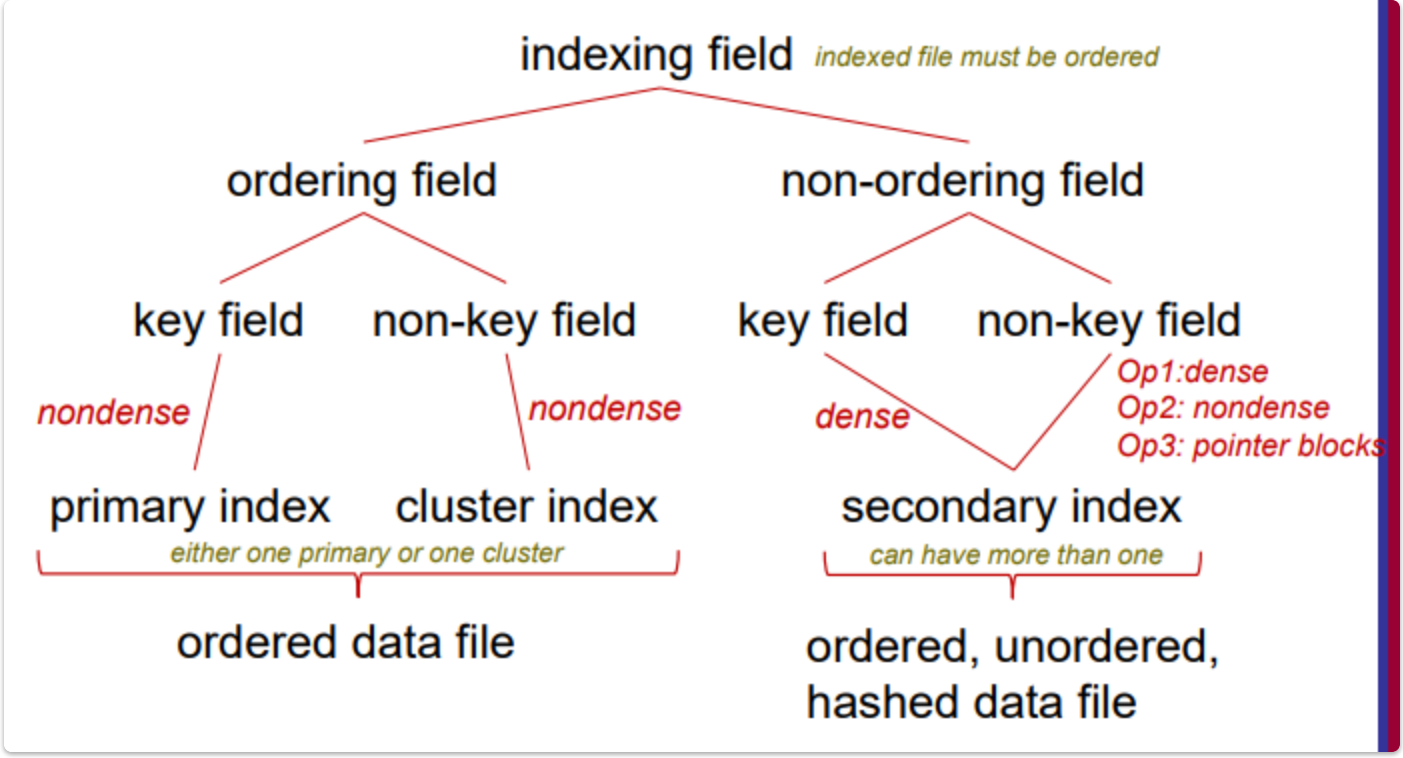


Secondary 以上提供邏輯上的排序基於index

Dense vs. NonDense

Dense 通常是用於，查詢頻繁、插入和更新較少的資料
NonDENSE 適用於查詢較少但插入和更新頻繁的資料

Summary



How to Create an Index

```
CREATE [ UNIQUE ] INDEX <index name>
ON     <table name> ( <colume name> [ <order> ]
{ , <colume name> [ <order> ]} )
[ CLUSTER ]
```

建立一個分群 index

```
CREATE INDEX DnoIndex
ON EMPLOYEE (Dno)
CLUSTER ;
```

會根據Dno去做分群

Multi-Level Indexes

以上描述是 single-level index，所以 index 對到的東西一定是 Primary index，所以後面都會是 Primary index 分類

且整體搜尋會是 log_fo，因為每一層都剪掉一個 fo，fo = bfri

r1 = 會是所有個block的數量
r2 = r1會對到幾個block的數量
.
.
.
r_n = r_n-1會對到幾個block的數量
且去上高斯

然後總數最後要加上1，因為要多一個存資料的

所帶來的代價是
當資料有更新或是新增的時候，DBMS需要去更新很多東西

RAID

為一個 disk striping
降低冗餘的資訊
有很多個level

Storage Area Networks

管資料的成本會比儲存的設備的成本低

Possible Question

Indexing 算法

題目會給：

1. 資料筆數 [r]
2. Block Size [b]
3. Spanned / Unspaned
4. Record Length [R] (有時候要自己算)
   要算：
5. Blocking factor [bfr]
6. Block Access

正規化

資料庫設計目標

- 保留訊息
- 最小化多餘的東西

品質的測量

- 確保屬性**語意清晰**
- **減少多餘**的資料
- **減少 NULL**
- **禁止有虛假**的資料

正規化

- 將不理想的"壞"關係分解為更小的關係，確保其具有無損拼接屬性和依賴性保留屬性
- 提高資料的一致性和完整性
- 過程部會改變資料庫的結構，只會改變關係的物理結構

功能性相依（FD）

如果屬性 X 的值可以確定 Y 的唯一值，則屬性集 X 功能性的確定屬性集 Y

1NF

- 禁止使用複合屬性、多值屬性、嵌套關係
- 只允許單個值作為屬性值
- 所以須拆複合屬性、多值屬性

2NF

- 要求每個關係 R 中的每個非主見 A 都完全依賴於 R 的主鍵
- 就是不能有部分功能性相依

- 簡單來說，要分解成多個 table，且那些 table 的每一個非主鍵都必須被所有主鍵給定義

3NF

- 不允許存在遞移功能性相依關係
- 白話一點，就是表裡面不能有某一個屬性可以被非主鍵屬性組成

BCNF (Boyce-Codd)

- 必須滿足 3NF
- 要求每個屬性都必須依賴於主鍵或與主鍵相關的屬性
- 簡單來說
  - 在表為單一主鍵時，必定滿足 BCNF
  - 在表為複合主鍵時，且有候選鍵時，屬性不能有共有的

Advanced SQL

Null 定義

1. 不知道的值
2. 不可用或保留價值
3. 無法用的屬性
   不能將一個值 = NULL

IN

可以檢查是否在表內

```
SELECT *
FROM   table1
WHERE  (c1, c2)
IN     (
        SELECT c1, c2
        FROM   table2
)
```

UNION

聯集
SELECT 的屬性必須一樣

EXISTS

檢查是否存在

```
SELECT Name
FROM Table1
WHERE EXISTS (
        SELECT *
        FROM Table2
        WHERE Table2.sn = Table1.sn AND Name = 'Wheels')
```

JOIN (+ ON)

- Inner JOIN：回傳兩個表共同屬性的表格，不會有空值
- LEFT JOIN：回傳左表所有的屬性，若右表沒有得對應就會是 NULL
- RIGHT JOIN：回傳右表所有的屬性，若左表沒有得對應就會是NULL
- FULL JOIN：回傳兩表所有的屬性，不管有沒有空值

整合函數

- sum( )：總和
- max( )：最大值
- min( )：最小值
- avg( )：平均值
- count( )：總數量

GROUP BY

將同樣的條件群組化，且取第一條

HAVING

- 通常加在 GROUP BY 後面，給予條件

```
SELECT *
FROM student
LEFT JOIN grade_report ON student.StudentNumber =
grade_report.StudentNumber
GROUP BY student.StudentNumber
HAVING student.StudentNumber < 120000000
```

WITH AS

像是寫一個 function 一樣

```
WITH function (column) AS
(
SELECT *
)
SELECT *
FROM function;
```

CASE

```
UPDATE EMPLOYEE
SET Salary =
CASE WHEN Dno = 5 THEN Salary + 2000
                  WHEN Dno = 4 THEN Salary + 1500
                  WHEN Dno = 1 THEN Salary + 3000
                  ELSE Salary + 0 ;
```

RECURSIVE

沒錯，就是遞迴

```
WITH RECURSIVE SUP_EMP (SupSsn, EmpSsn) AS
          ( SELECT SupervisorSsn, Ssn
                  FROM EMPLOYEE
                  UNION
                  SELECT E.Ssn, S.SupSsn
```

```
                FROM EMPLOYEE AS E, SUP_EMP AS S
                WHERE E.SupervisorSsn = S.EmpSsn )
SELECT*
FROM SUP_EMP;
```

CREATE VIEW AS

```
CREATE VIEW DEPT5EMP AS
SELECT *
FROM EMPLOYEE
WHERE Dno = 5;
```

且不能有重複的屬性定義名

ALTER

可以直接改變表裡面的屬性型態

```
ALTER TABLE COMPANY.EMPLOYEE ADD COLUMN Job VARCHAR(12);
ALTER TABLE COMPANY.EMPLOYEE DROP COLUMN Address CASCADE;
ALTER TABLE COMPANY.DEPARTMENT ALTER COLUMN Mgr_ssn DROP DEFAULT;
```

Hashing

What is Hashing?

- 他就是一個 function，透過這個 function 計算出所要的 index
- 在不同的 key 卻會得到相同的 index，此時會造成 collision (碰撞)
- 一個 index 會對上一個一個的 bucket，但一個 bucket 不一定只有一個資料，可能裡面有很多的 slot
  - 當資料量超過了 slot 容量，就會發生 overflow
- 可以提供非常快速的訪問紀錄，且只需要一個 block 就可以記錄檢索的紀錄

Keywords

- key：要經由 hashing 過後，可以只到 index
- T：index 的數量

- n：資料數量
- Key density：n/T，資料數量在所有可能的比例
- Loading density (load factor)：n/(bucket * slot) = α
  - α 越大 $\rightarrow$ bucket 利用度越高，但碰撞的機率也會提高
- m： bucket 數量

Purpose

- 計算要簡單
  - 複雜度希望是 O(1)
- 碰撞要少
  - 希望每個 bucket 接觸的機率一樣
- 減少分群的現象 clustering
  - Load 平均

Rehashing

- 當 load factor 太大的時候，就應該考慮重新建構 function
  - 太大的時候會 bucket 使用率變高，使 slot 快滿

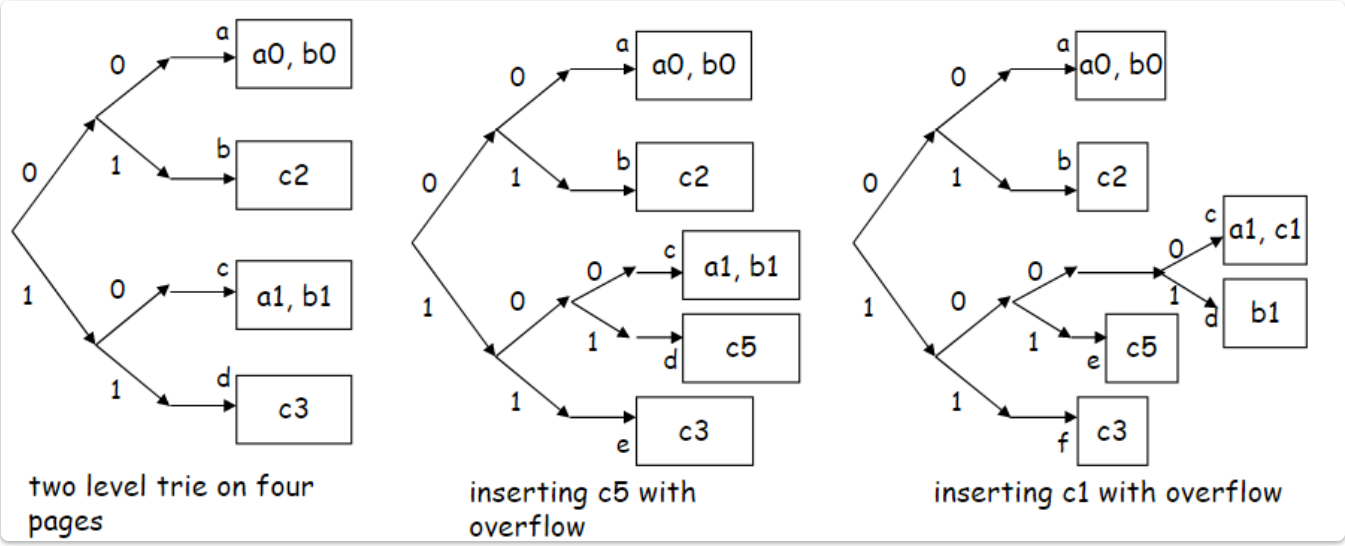
k	h(k)
A0	100 000
A1	100 001
B0	101 000
B1	101 001
C1	110 001
C2	110 010
C3	110 011
C5	110 101

h(k,i)=bits 0-i of h(k)
Example:
h(A0,1)=0
h(A1,3)=001=1
h(B1,4)=1001=9

- 所以此時會開始 Rehashing，而且通常會將 array 增大兩倍去存取，會導致你要插入載滿的表格的時候就要重新設計整個 Hashing function

Hashing type

分為兩種：

- 靜態雜湊：
  - 使用固定大小的雜湊表，可能會發生碰撞
  - 溢出表對查詢性能不佳
- 動態雜湊 (又稱 extendible hashing)：
  - 使用目錄指向多個雜湊表，表可以輕鬆增加
  - 這樣就可以減少碰撞次數，並且在雜湊表增長時可以更有效的利用空間
  - 對有問題的表增加表就可以解決
  - 

  - 這樣可以不會碰到舊的東西，也可以解決 overflow 的問題

linear hashing

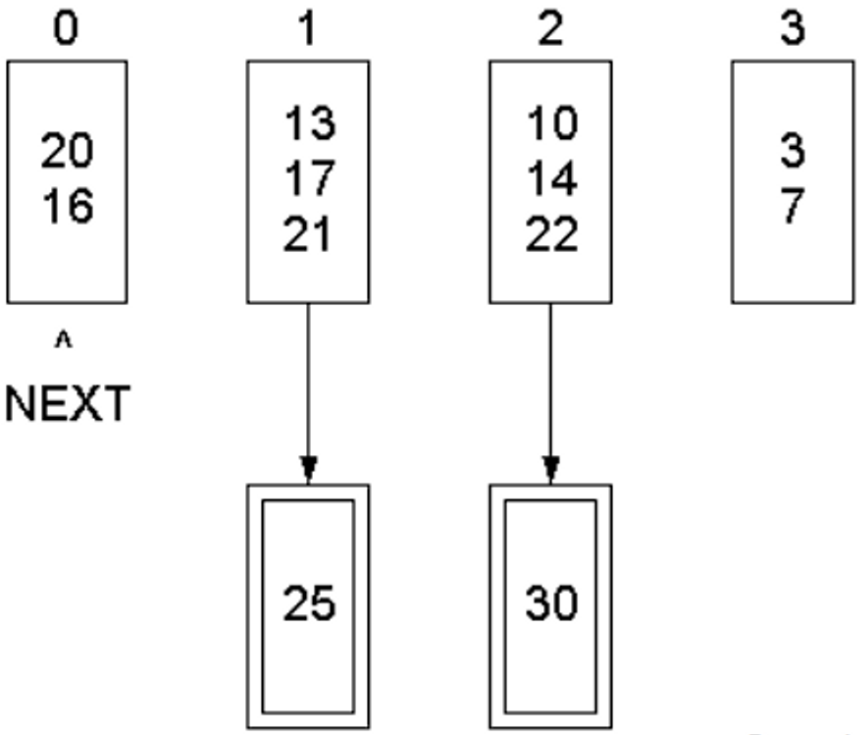
- 是一種 dynamic hashing
- 會將 overflow 記錄起來，然後會將每個 bucket 的 overflow 紀錄連接起來，
- when collison happening
  - h(K) = K mod M
  - hi + 1(K) = 2M

- hi + 2(K) = 4M

Example 1:

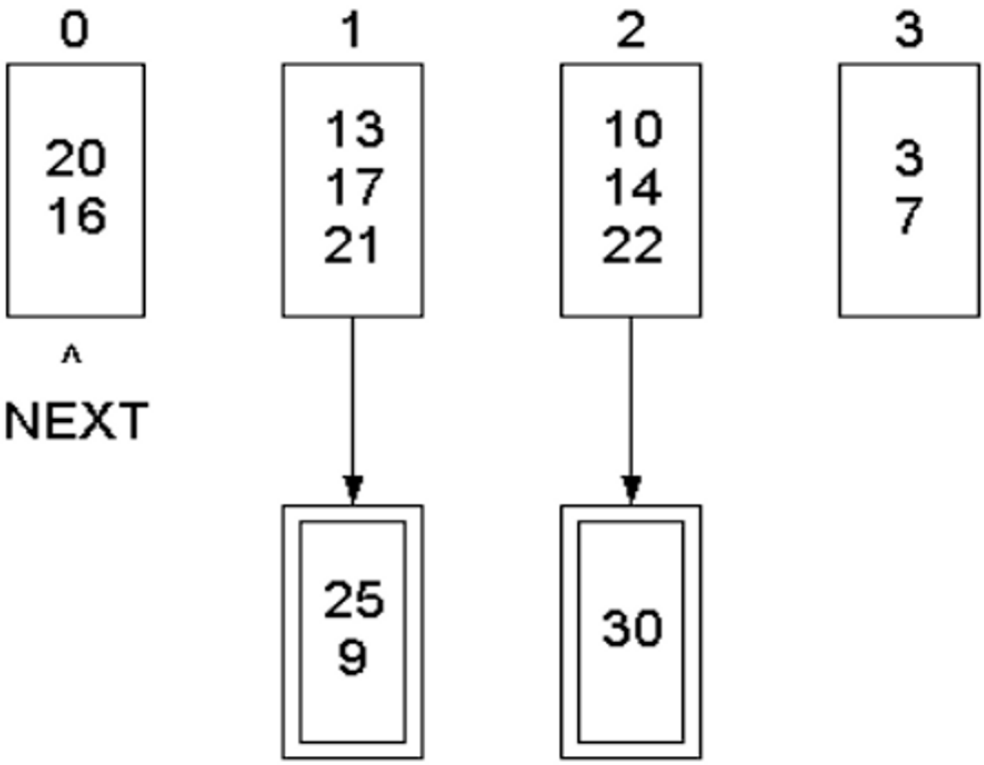
(40% <-> 80%)

h_0(key) = key mod 4

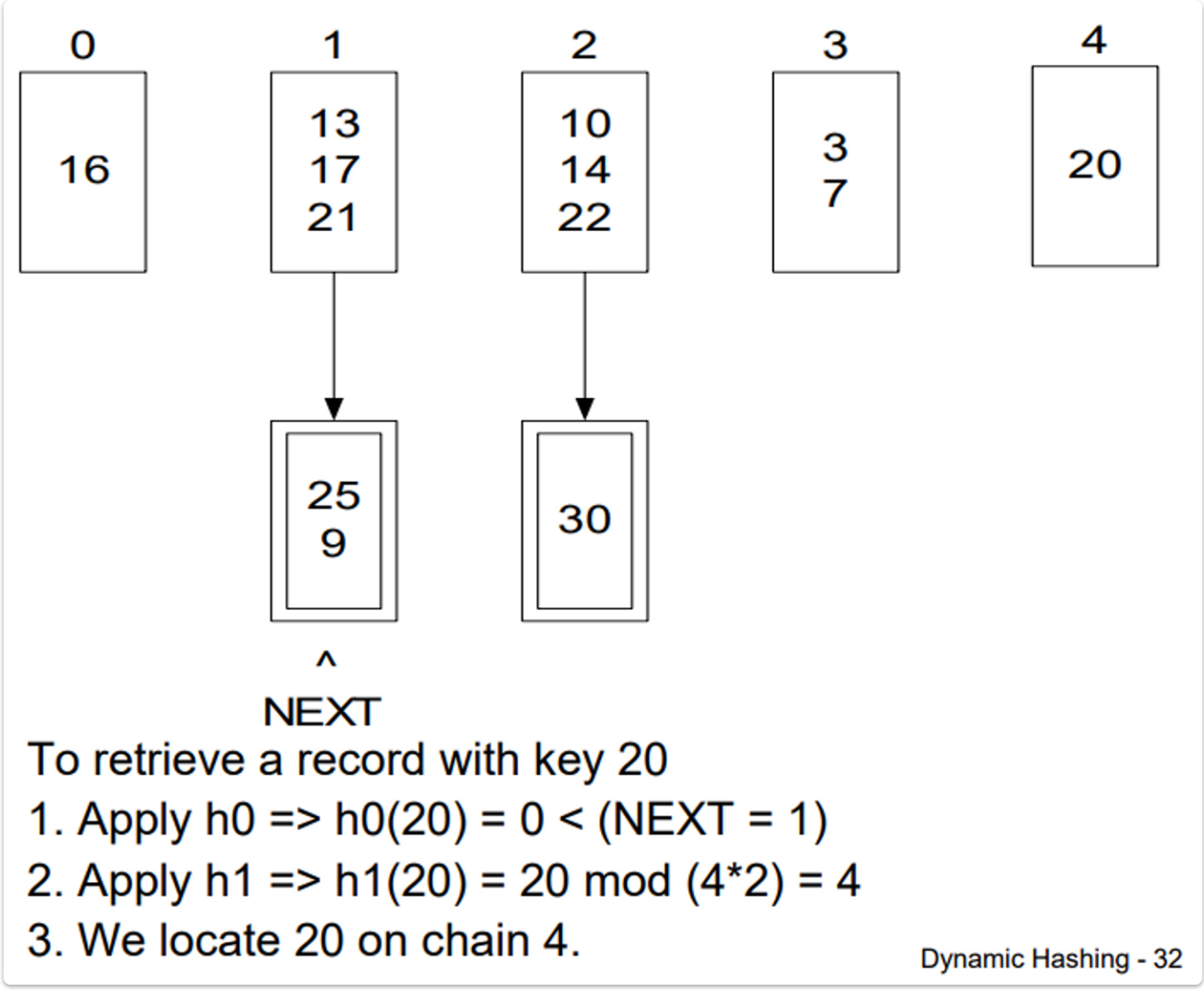


- Insert a record with a key of 9

$h_0(9) = 1$



- Storage utilization = 81%
- To split the NEXT chain



參考資料

[Hashing 基礎介紹 | MeteorV's Blog]

[OpenAI]

[上課講義]

[聯合大學大寶典]