According to midterm 2018:

MT1

1、 Disk 计算
  → space
  → time

2、 Record store
  → char
  → varchar

3、 Page Replacement Algorithm

4、 Schedule

5、 SQL

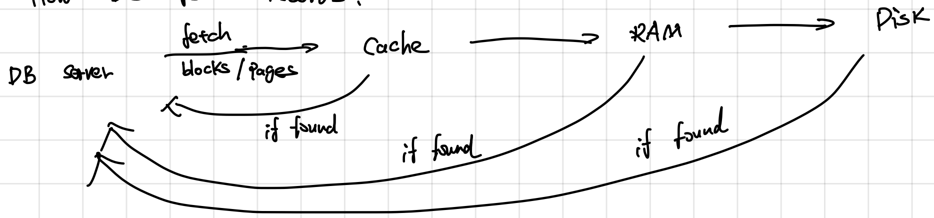
6、 补充 : min → s → ms → μs

Disk Storage

- Learning Goals

1. Explain the impack that disk activity on DBMS query performance

↳ How DB fetch records?



所以DBMS Query有时确实需要从 Disk中找数据，所以 Disk 这还一个请求的快慢就和 DBMS Query 的快慢正相关了

且不仅仅是 Read ↑，还有 write ↓

Write: Transfer data from RAM to disk

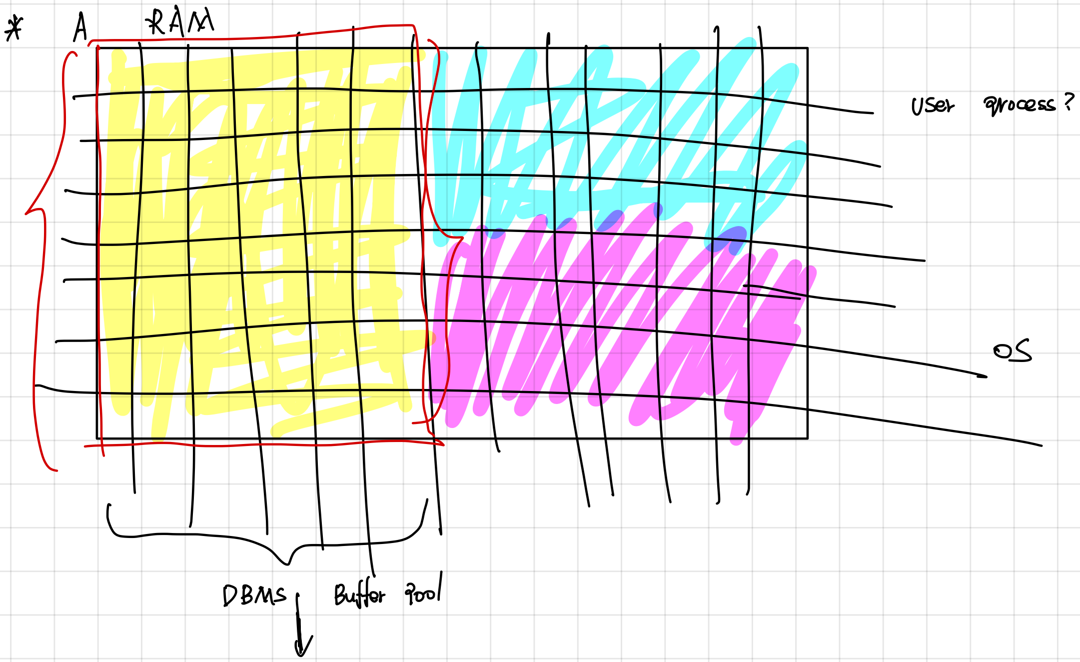
并且 ↳ 这个区别令 Server 在读取 disk 的数据时需要格外注意.

Random-access memory: A RAM device allows items to be read/written in "almost" the same amount of time irrespective of the physical location of data.

physical location X Access time

Disk: Access time of a disk pages varies depending upon its location on disk.

physical location ⟶ Access time



Buffer pool : 1. independently of the OS

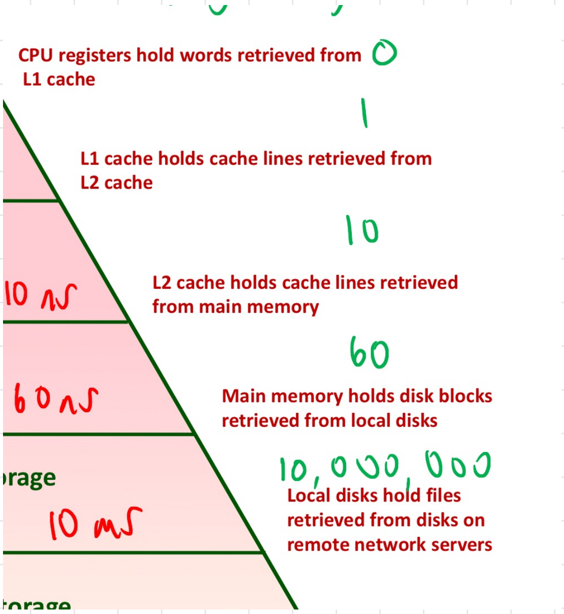
2. be allocated by the database manager for caching table and index data as it is read from disk

Formally :

	RAM	VS	Disk
physical → Access time :	X		✓
Cost :	\$45 for 8 GB $\$ 3^2 \cdot 5 \cdot 2^{-33}$ /byte		\$60 for 2^{40} byte $\$ 15 \cdot 2^{-38}$ /byte
is Volatility :	✓		X

Position on ↓ :

register
↓
On chip L1 cache (SRAM)
↓
off chip L2 cache (SRAM)
↓
main memory
↓
Local secondary storage
↓
remote secondary storage



CPU registers hold words retrieved from L1 cache

L1 cache holds cache lines retrieved from L2 cache

L2 cache holds cache lines retrieved from main memory

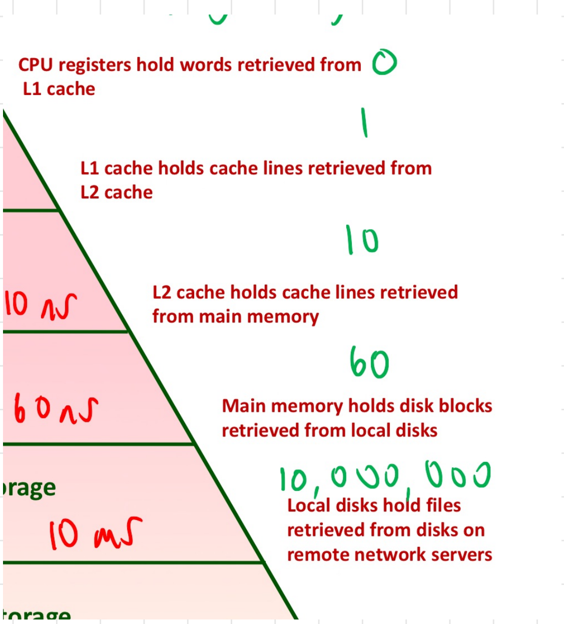
Main memory holds disk blocks retrieved from local disks

Local disks hold files retrieved from disks on remote network servers

2. Draw the memory hierachy. Show where database botelenects are mose likely to occur and where extensive caching take place.

Position on ↓ :

register
↓
On chip L1 cache (SRAM)
↓
off chip L2 cache (SRAM)
↓
main memory
↓
Local secondary storage
↓
remote secondary storage



CPU registers hold words retrieved from L1 cache

L1 cache holds cache lines retrieved from L2 cache

L2 cache holds cache lines retrieved from main memory

Main memory holds disk blocks retrieved from local disks

Local disks hold files retrieved from disks on remote network servers

Bottleneck: ?

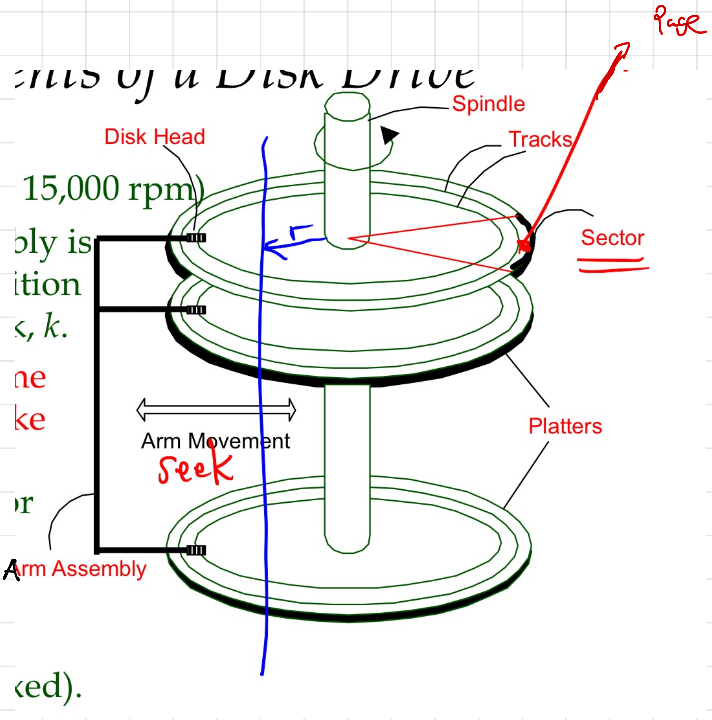
Extensive caching where : ?

3. Components and contrast : cost, capacity and speed of access , in the levels of memory hierachy

	Access time (seconds)	Cost
register	0	↓ cheaper
↓		
On chip L1 cache (SRAM)	10^{-9}	
↓		
off chip L2 cache (SRAM)	10^{-8}	
↓		
main memory	10^{-8}	
↓		
Local secondary storage	10^{-2}	
↓		
remote secondary storage		

4. Identify the components of a disk drive

- * Spindle
- * Disk Heads (Read/write head)
- * Tracks
- * Sector
- * platters
- * Arm Assembly



Other views :

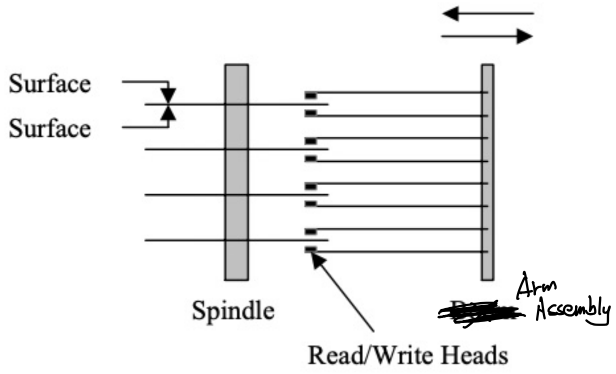
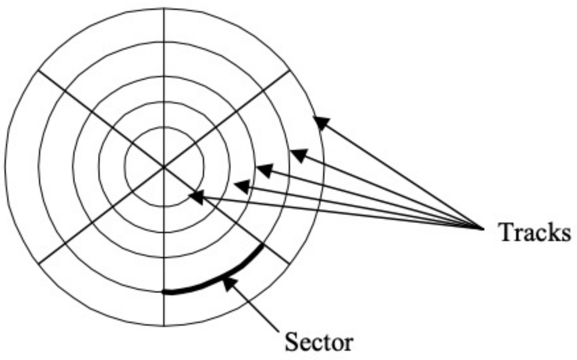


Figure 1: Disk drive with 4 platters and 8 surfaces

Looking at a surface:



* 所以一个 platter 的两个 surface 上都有 head

* 一个 sector 单纯指 $\frac{\theta}{360}$ · track (圆) 的那部分

* cylinder : All tracks with the same track # (就和一个圆柱体一样)

* 读取顺序 : 1. Arm assembly 把 head 全部移到一个 track 上 (其实是整体移动到一个 cylinder
2. Only one head read / write at a time (虽然整体移动到了一个 cylinder, 但只有一个 track 在读/写)

Key components :
- Head
- Spindle
- Tracks
- Sectors
- Platters
- Arm assembly
- Cylinder

Key calculation components :
- Tracks
- Platters <=> surface
- Cylinder
- Sector

* 一个 Sector 是 smallest addressable unit in a disk

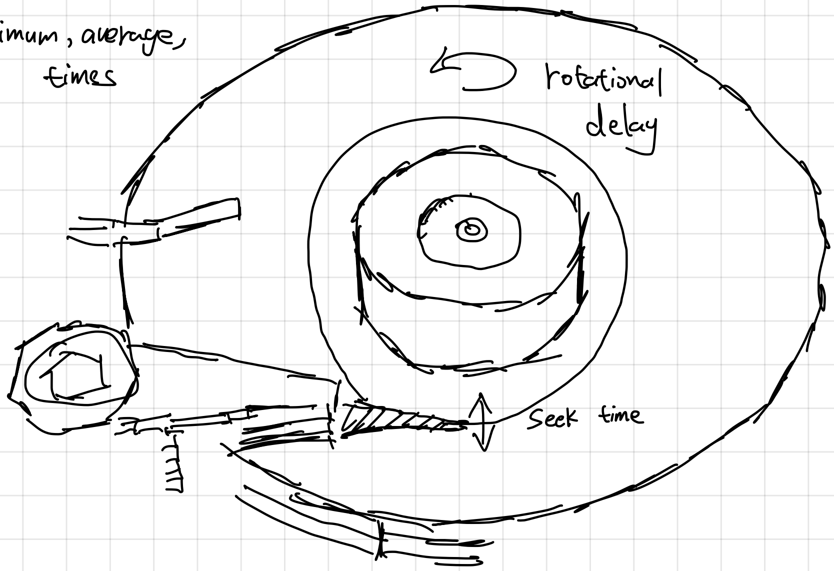
↳ Block size = multiple of sector size

5. Given disk geometry figures, calculate the amount of time it takes to read / write with a number of blocks / pages, tracks, or cylinders of data to / from a disk

* Access time: The time to read / write a disk page / block

= Seek time (move the arm to position the disk head just above a particular cylinder)
+
Rotational delay (Wait for the start of the desired block to come to be positioned under the head)
+
Transfer time (the time it takes to transmit between disk and RAM)

6. Given disk geometry figures, compute the minimum, average, and maximum seek, rotation, and transfer times to / from a disk



* Seek time varies from 1-20 ms
* Rotational delay varies from 0-10 ms
* Transfer time is determined by the disk's rotation rate, less than 1 ms / 4K pages
↳ Assume extra transfer time (when all data for requested page passes under the head)
= transfer time

Calculation: See example

Note: 当题目给出: moving time from C1 to C2 is 1ms + 1ms per n cylinders moved
那 average seek time: $1ms + \frac{1}{3}\cdot\left(\underbrace{\frac{\#\text{Total moved track}}{n}}_{\text{maximum seek time}}\right)$

Note: Average rotational latency: $\frac{1}{2}$ spin
spin 就是转一圈的时间

Note: tracfer time for a page or anything

1. Calculate # sector needed for the page or anything

2. Calculate the portion of one track that needed to store a page
↳ 1 page = n sectors
we need $\frac{n \text{ sectors}}{m \text{ sec/track}} = x$ tracks
↳ a fraction, hopefully

3. Use x to multiply the rotational delay for one spin → x · spin ✓

7. Compare and contrast the relative speeds of seek, rotation, and transfer times —— when accessing a given size of data on disk.

8. Explain how a large file, broken up into pages, can be optimally placed on a disk to improve performance

- ↳ prefetching : load a data before it's needed
  - ↳ 具体来说，DBMS 会预测用户的行为，再在这个行为的基础上进行 prefetch
    - ↳ 例如，一些用户 query 的目的可能只是 preview，但在 preview 之后就会下载整页内容

既然 DBMS 在从 disk 截取了位置了，如果 prefetch 到用来 preview 的内容，就会提高效率.

具体来说，该如何安排存放数据呢？

- ↳ Blocks in file should be arranged sequentially on disk (by "next" order), to minimize seek and rotational delay
  - ↳ "next" order :

We like to access :

Blocks on same track

> Blocks on same cylinder

> Blocks on adjacent cylinder

SAN: Storage Area Networks: Group of networked disk devices. Can be shared by group of machines.

- High performance, reliable
- RAID: Redundant Array of Independent Disks (at least two ways to read a data)

▷ UBC used to use SAN, now Isilon (Network Attached storage)

9. Compare and contrast HDD to solid state disks (SSD)

	HDD	SSD
Memory	Direct access?	flash memory
Parts		No moving parts/spinning disks
power		($\frac{1}{3}$, $\frac{1}{2}$) • HDD
Noise		quieter
reads	5-10ms	0-1 ms
price	\$60~70 (1TB) ,	\$115 – 140 (1TB)
capacity		Smaller
write cycle		limited : (1~5 million)

Random write/read : relevant when dealing with lots of small files scattered around the disk

↳ For HDD : The head moves, before the next piece of the file can be read/written

For SSD : Since no moving parts,

↳ erase a block of NAND flash

No matter the size of written data

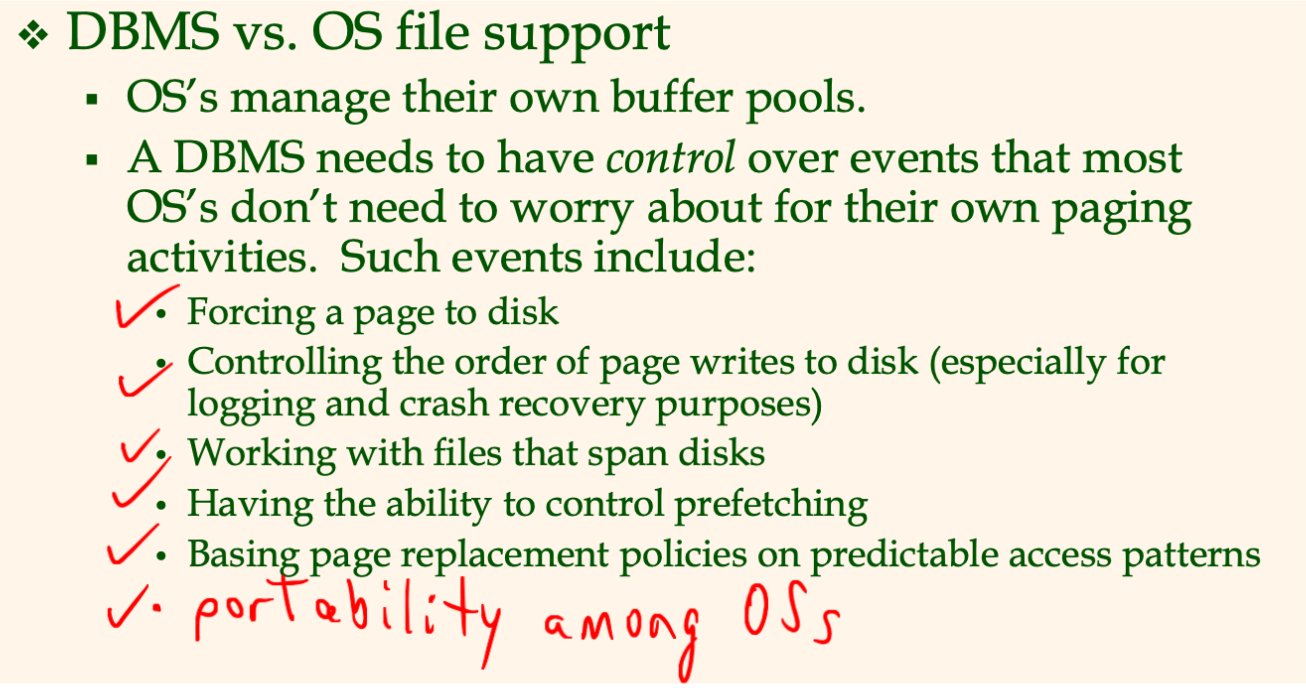
* SSD get us 1 order of magnitude closer to speed of RAM

RAM – HDD : 5 order of magnitude

Buffer pool management

Learning Goals

- Explain the purpose of a DBMS buffer pool, and justify why a DBMS should manage its own buffer pool, rather than let the OS handle it.
  - Buffer pool managed by DBMS is an area of main memory that has been allocated by the database manager for the purpose of caching table and index data as it is read from disk
  - When the buffer pool is managed by the DBMS, DBMS will choose algorithm/act on discarding pages by maximizing the IO needs. However, if the buffer pool is being managed by OS, the way it handles discarding pages will be pretty general since OS has so many other things to worry about. Moreover, if managing its own buffer pool, number of systems calls can be reduced.
  - Summary: DBMS maintain their own buffer rather than use that of the OS so that they control when to let out pages from it. It also avoids a system callfor each OS buffer read, although that could even be avoided by OS design.



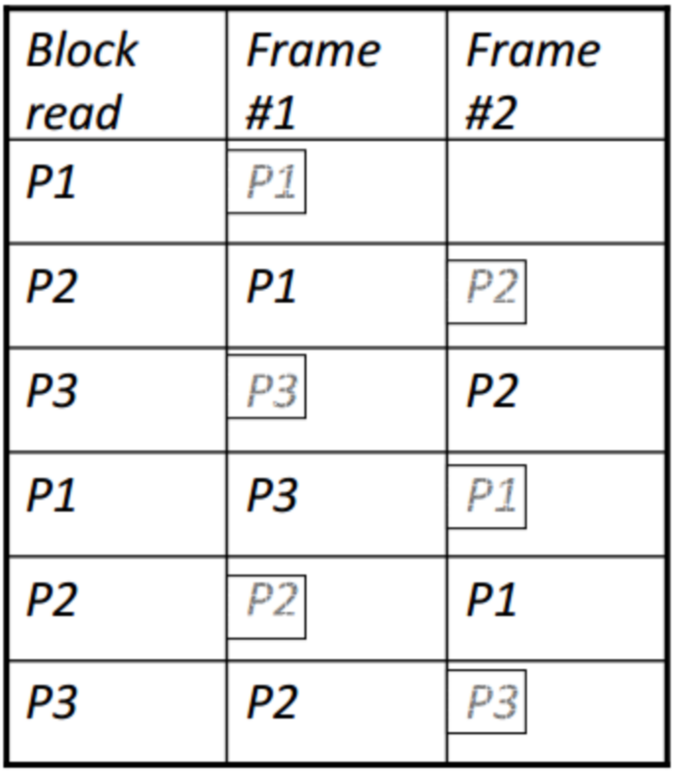
❖ DBMS vs. OS file support
- OS's manage their own buffer pools.
- A DBMS needs to have *control* over events that most OS's don't need to worry about for their own paging activities. Such events include:
  - ✓ Forcing a page to disk
  - ✓ Controlling the order of page writes to disk (especially for logging and crash recovery purposes)
  - ✓ Working with files that span disks
  - ✓ Having the ability to control prefetching
  - ✓ Basing page replacement policies on predictable access patterns
  - ✓ portability among OSs

- Provide an example of sequential flooding in a DBMS buffer pool.

  A nasty situation caused by **LRU + repeated sequential scans**

  也就是每个page request都是miss

Let us say there is `2` buffer frames `Frame #1` and `Frame #2`, `3` pages in file `P1`, `P2` and `P3`. What would happen if we scan the file twice(P1, P2, P3, P1, P2, P3) with `sequential scan`?



Block read	Frame #1	Frame #2
P1	P1	
P2	P1	P2
P3	P3	P2
P1	P3	P1
P2	P2	P1
P3	P2	P3

- Explain the tradeoffs between force/no force and steal/no steal page management in a buffer pool. Justify the use of the ARIES algorithm to exploit these properties.

	No Steal	**Steal**
No Force		Fastest
Force	Slowest	

	No Steal	**Steal**
No Force	No Undo/Redo	Undo/Redo
Force	No Undo/No Redo	Undo/No Redo

Basically, If employing steal approach, we can use pages that is being utilized by uncommitted as an empty frame for the page replacing algorithm, so the speed will be increased. But an undo log need to be 维护 when employing steal approach and dangerous(?). As for no force approach, it does increase the speed since we can use temporary storage to store transaction's updated pages, and then write them in disk in batch -> fewer IO operations.

ARIES???

- For a given reference string, compute the behaviour of the these page replacement algorithms: FIFO, LRU, MRU, Clock (reference bit), and Extended Clock (reference bit + dirty bit).

  - FIFO: victim = oldest page
  - Least Recently Use(LRU): victim = page that hasn't been referenced for the longest time
  - Most Recently Used(MRU): victim = page that has been most recently used
  - Clock: If a page is referenced often enough, its reference bit (RB) will stay set, and it won't be a victim.
    - if an empty frame in BP:
      - Use it to store the new page's data
      - Set the RB to 1
      - Set the timestamp to current time
    - else:
      - Find the oldest page(page with the oldest timestamp)
      - If that page's RB is set to 0, then:
        - This is the victim page, replace it with the new page
        - Set the new page's RB to 1
        - Set timestamp to the current time.
      - Else:
        - Decrement that page's RB to 0
        - Update that page's timestamp to the current time
  - Extended Clock
    - if an empty frame in BP:
      - Use it to store the new page's data
      - Set the RB to 1， DB to 1(?)
      - Set the timestamp to current time
    - else:
      - Find the oldest page(page with the oldest timestamp)
      - If that page's RB is set to 0/0 or 0/0* then:
        - This is the victim page, replace it with the new page
        - Set the new page's RB to 1, DB to 1(?)
        - Set timestamp to the current time.
      - Else:

```
switch (RB/DB){
  case (0/1):
      set to 0/0*;
  case (1/0 || 1/0*):
      set to 0/0 || 0/0*;
  case (1/1):
      set to 0/1;
}
```

- Create a reference string that produces worst-case performance for a given page replacement algorithm.

FIFO: 1 3 0 4 1 3 0 for 3 available frames

LRU: 1 2 3 1 2 3 for 2 available frames

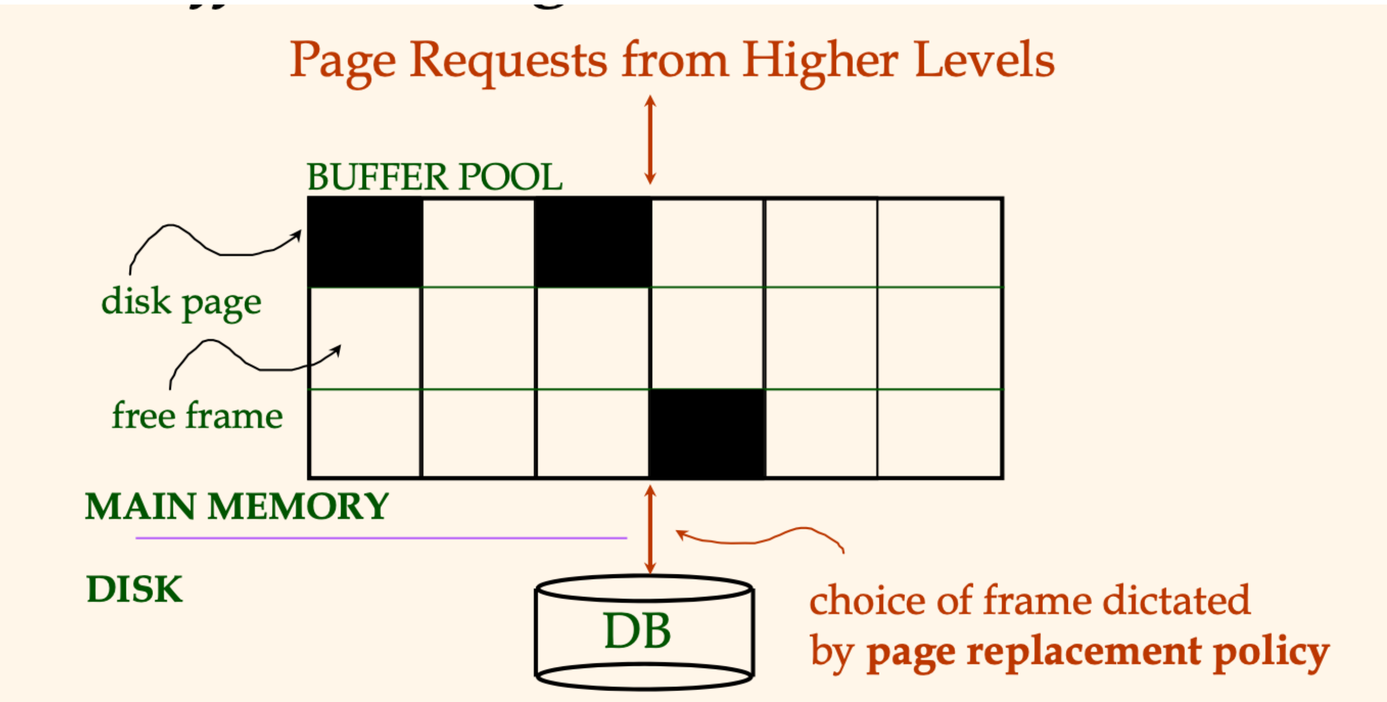
MRU: 1 2 3 2 3 for 2 available frames

- [Later] Explain how the page replacement policy and the access plan can have a big impact on the number of I/Os required by a given application.
- [Later] Predict which buffer pool page replacement strategy works best with a given workload (e.g., table scan, index scan, index lookup, logging, returning many rows, running the RUNSTATS utility).

Other

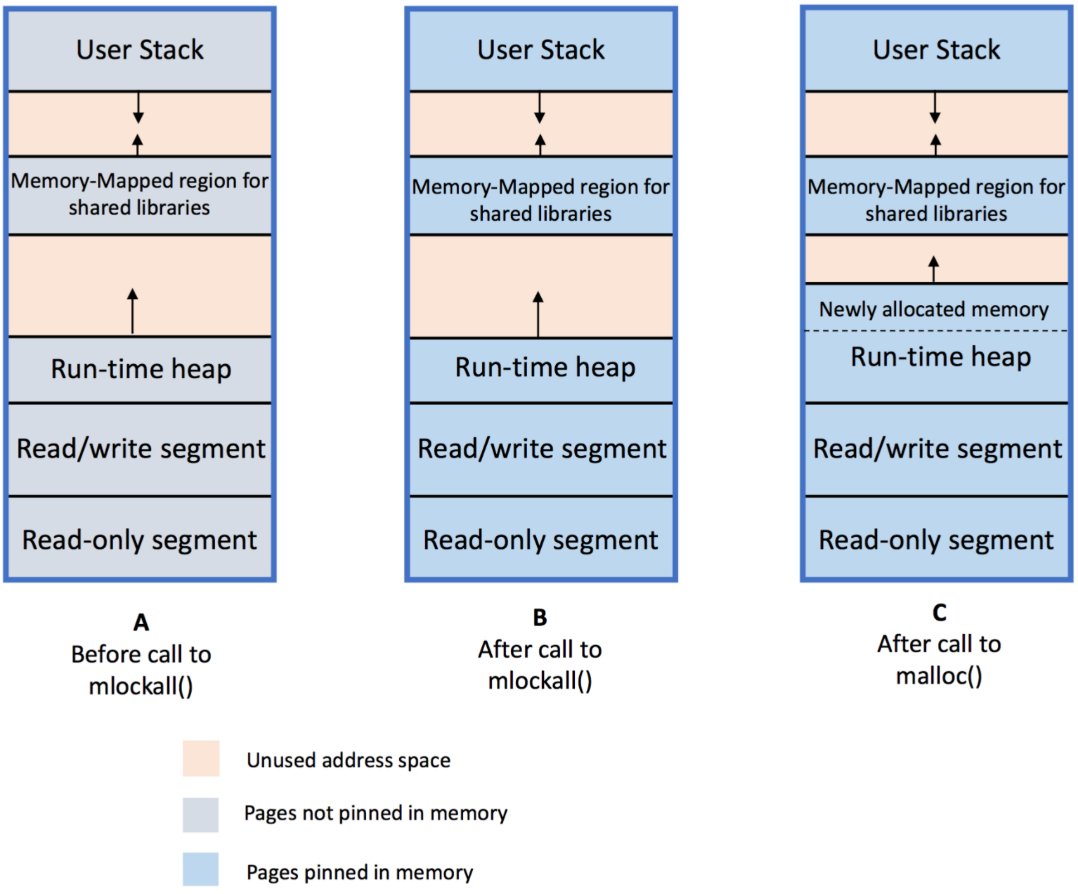
- a **page** is the smallest unit of transfer between disk and main memory
  - logical memory: page
  - physical memory: frames
- The Translation Lookaside Buffer (TLB) is a very fast L1 hardware cache. It is used to determine whether or not a particular page is currently in memory.
- Dirty bit/frame: The bit indicates that its associated block of memory has been modified and has not been saved to storage yet. When a block of memory is to be replaced, its corresponding dirty bit is checked to see if the block needs to be written back to secondary memory before being replaced or if it can simply be removed.

Page Replacement



```
public void requestPage(byte address){
  if (isAddressExitstInPool(address)){//use that}
  else{
    byte replacedFrame = pageReplacementAlgorithm();
    if (isPin(replacedFrame)){unPin(replacedFrame);}
    if (isDirty(replacedFrame)){writeToDisk(replacedFrame);}
    readToFrame(address, replacedFrame);
  }
}
```

- 其他操作：Pin
    - Pinning the pages in main memory is one way to ensure that a process stays in main memory and is exempt from paging. 并且当有一个新的request pin了这个page的时候，他的page count++



- transactions is a series of one or more SQL statements
    - commit: **A transaction is said to be committed when its log records reach disk.**
    - A transaction that is in progress is said to be **in-flight**. It hasn't been **committed**.
    - Locks held by the transaction are released at COMMIT time.
    - **Force**: At transaction commit time, we force(i.e. write) the transaction's updated pages to disk (after writing the log records to disk).
        - force策略表示事务在committed之后必须将所有更新立刻持久化到磁盘，这样会导致磁盘发生很多小的写操作（更可能是随机写）。no-force表示事务在committed之后可以不立即持久化到磁盘， 这样可以缓存很多的更新批量持久化到磁盘，这样可以降低磁盘操作次数（提升顺序写），

        - 但是如果committed之后发生crash，那么此时已经committed的事务数据将会丢失（因为还没有持久化到磁盘），因此系统需要记录redo log，在系统重启时候进行前滚（roll-forward）操作。
    - **Steal**: When the BP desperately **No Force** needs a free page, we can write a dirty page for an uncommitted transaction to disk (i.e., we steal frame from an in-flight transaction).
        - 是否允许一个uncommitted的事务将修改更新到磁盘，如果是steal策略，那么此时磁盘上就可能包含uncommitted的数据，因此系统需要记录undo log，以防事务abort时进行回滚（roll-back）。如果是no steal策略，就表示磁盘上不会存在uncommitted数据，因此无需回滚操作，也就无需记录undo log。

	No Steal	**Steal**
No Force		Fastest
Force	Slowest	

- A newly requested disk page that is not currently in the buffer pool causes a **page fault**.
- The page to be replaced is called the **victim** page

Disk Scheduling, Metadata, Records, and Pages

Learning Goals

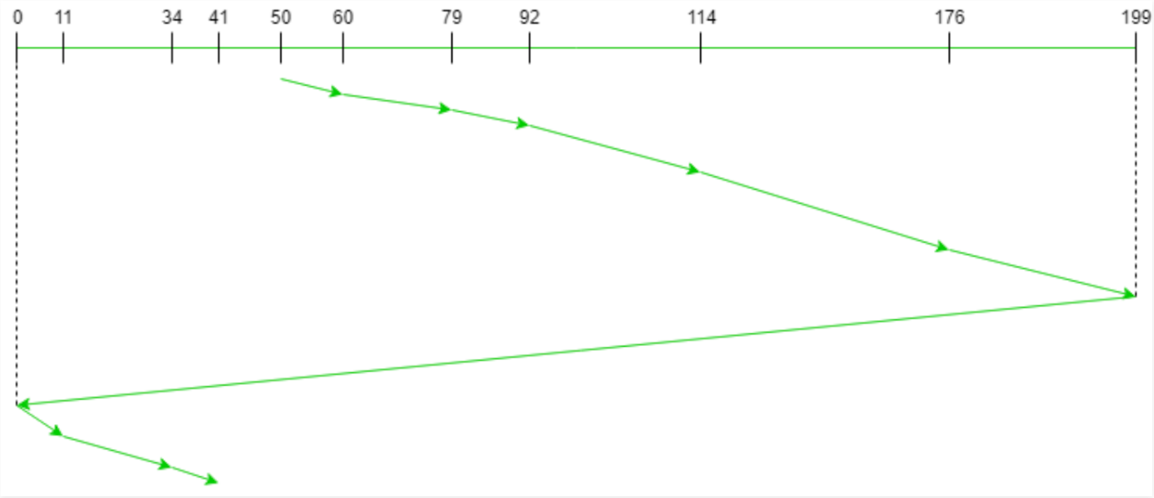
- Explain why page requests from disk may not be serviced immediately. List some of the reasons for contention.
    - Many DB users wanting access to the objects on the disk drive
    - Non-DB users wanting access to files on the same disk drive
    - Single user, but many processes/applications requesting service
    - Overhead service routines(DBMS, OS, ..)
- Explain the relationship among disk geometry, buffer pool management, and disk scheduling in providing good performance for data requests from a user of a DBMS. List the bottlenecks that may contribute to poor I/O performance in this disk “chain”.
    - 所以整个的顺序是，一个request被提出，然后
        - 从磁盘里取出这个page/block
            - 这个时候往往需要disk scheduling算法的参与，这个算法决定了seek time，也就是head什么时候移到哪个cylinder/track
            - 移动到相应的cylinder/track之后，磁盘开始转动，找到那个block最开始的那个sector，然后读取这个block（page），丢回给main memory
        - 把这个page/block放入bp
            - 在main memory中，buffer pool manager利用page replacement算法确定要替换哪个page，然后把这个读取到的page给放进去
- Compute the service order for a queue of track/cylinder/page requests using each of these disk scheduling algorithms: FCFS (First Come, First Serve), SSTF (Shortest Seek Time First), and Elevator (Scan with, and without, Look).
    - Disk Scheduling Algorithms:

      Current Status: 165, coming from 164, receiving requests: 1400, 2500, 170, 160, 161, 3500, 162

        - FCFS, First come First Serve
            - 1400, 2500, 170, 160, 161, 3500, 162
        - SSTF, Shortest Seek Time(也就是找距离最近的cylinder) First
            - 162, 161, 160,170,1400, 2500, 3500
        - Elevator Algorithm: 这个算法向一个方向持续扫描直至这个方向上没有别的request
            - 因为刚从164到165，所以现在从小到大扫描
            - 165, 170, 1400, 2500, 3500
            - 这时候这个方向的全部扫描好了，就会回来
            - 162, 161, 160

          补充问：

            - Service order: 170, 1400, 2500, 3500, 162, 161, 160
            - What is the updated service order if: while serving cyl. 1400, we suddenly get these new requests: 1250, 1400, 1500
            - 1400, 1500, 2500, 3500, 162, 161, 160 X(他问的是整体的service order，全都要写出来，并且需要把1250也写上)
            - 170, 1400, 1400, 2500, 3500, 1250., 162, 161, 160
        - C SCAN: 基本上就是不管在哪都先朝着一个方向走一直走到这个方向上最后一个cyl，然后再直接回到这个方向的0，再走

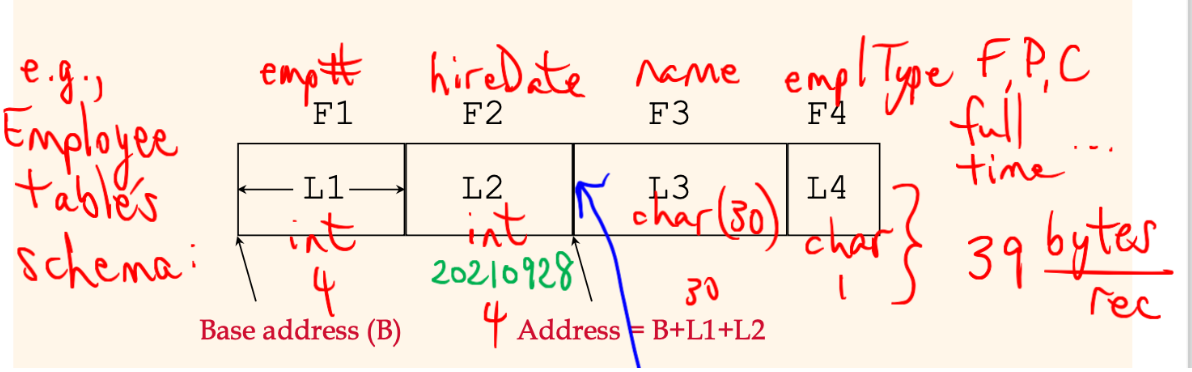
        - Unfairness: 比如如果一个request要request的是一个很远的cylinder，那么sstf可能就永远也触及不到他
- Give at least ten examples of the kinds of metadata stored for a DBMS.
    - number of records
    - number of unique keys
    - column names
    - data types
    - field sizes
    - flags
    - permissions
    - creation times
    - creator ids

    - record layouts
    - buffer pool sizes
- Justify the use of metadata from the perspective of both a DBMS and a DBA.
- Write simple SQL queries (on paper) to query an RDBMS catalog for metadata that is of interest to a DBA. For example, write simple SQL queries to join catalog tables and gather information from selected DB2 catalog tables.
- **Provide arguments for storing RDBMS metadata as a table rather than as a flat file or some other data structure.**
- Compare and contrast the record layouts for fixed-length and variable-length records in a DBMS. Provide an advantage for each.
- Explain why rows on a page might be relocated.
- Compare and contrast the page layouts for fixed-length and variable-length records in a DBMS. Provide an advantage for each.
- Justify the use of free space within a page, and intermittent free pages within a file, for an RDBMS table.
- Given probabilities of average string lengths, determine whether it makes more sense to use a fixed-length field, rather than a variable-length field.

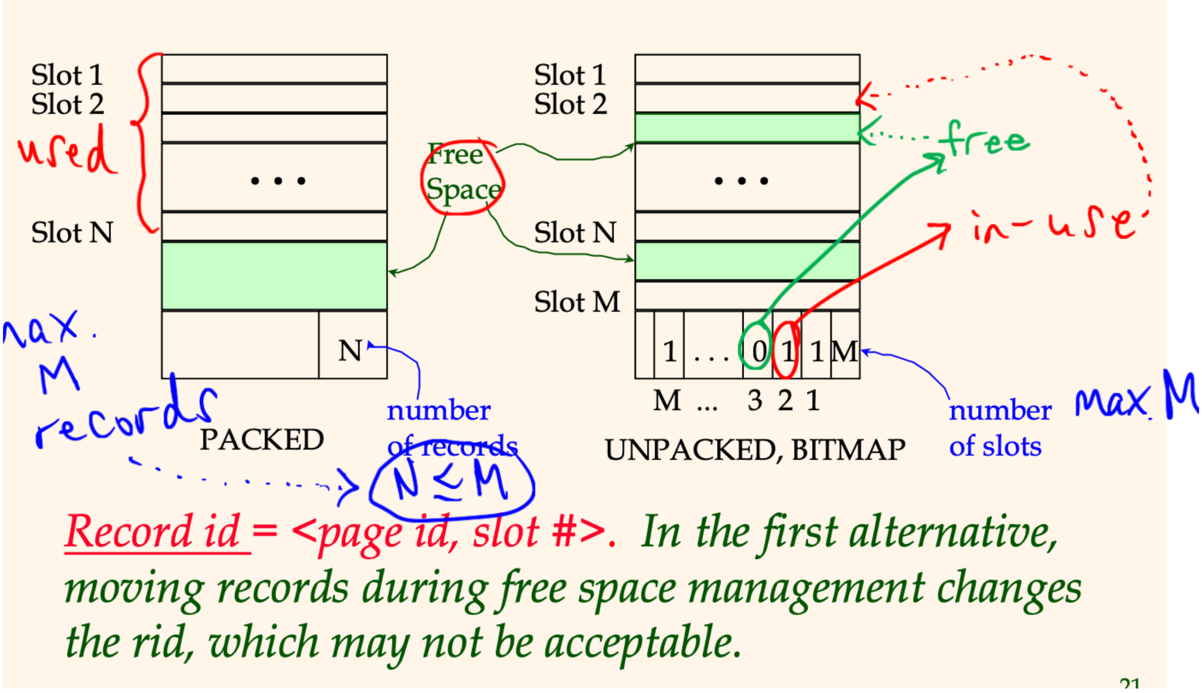
Other

- Indexes
    - We can retrieve records by:
        - Scanning all records in a file sequentially
        - Specifying the record id and going there directly
    - File structures that enable us to answer such value-based queries efficiently
        - value based queries:希望通过这个数据中的某个字段的值去找到这个数据
            - Find the name of the student with student id 86753091
            - Find all students with GPA $> 85\%$
        - **Dense Indexes** store one key/value pair per record in the table. The value is often the rid pointer that points to the full record on disk corresponding to the key
        - **Clustering Indexes**: 直接根据这个index的数据大小排序了数据在disk中的位置，所以如果这个clustering index里定义的是什么被用作“where” query的数值就会很方便（都根据数据大小排序好了）
- Metadata：data about data
- System Catalogs
    - contains metadata
        - e.g., # of records, # of unique keys, record layouts, column names, data types, field size, flags, permissions, creation times, creator IDs
    - There are about 150 catalog tables in Db2 version 11:
        - SYSIBM.SYSTABLES
        - SYSIBM.SYSINDEXES
        - SYSIBM.SYSKEYS

Stored in database DSNDB06, can be accessed through sql

- Record Formats
    - Fixed Length
        - 

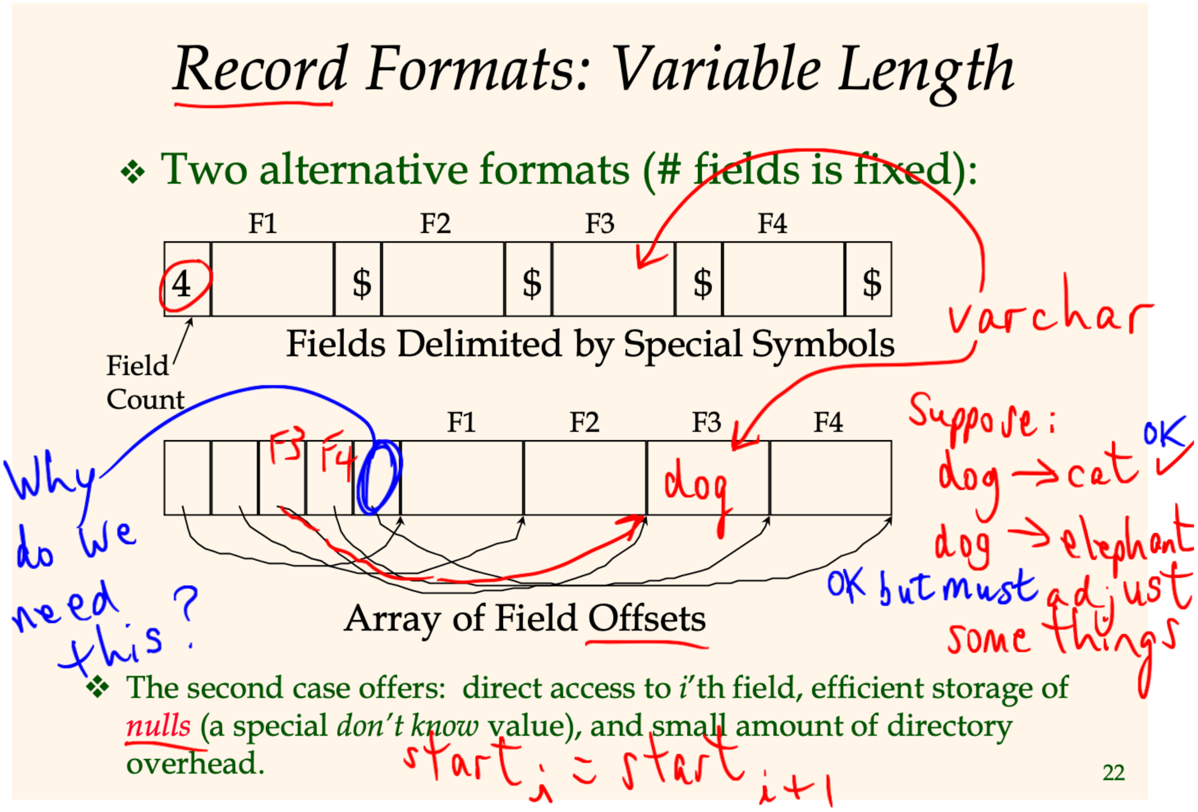
        - 不同的字段间相隔的长度都是一样的，是fixed的
            - **那么为什么只需要L1和L2就可以确定地址了呢**
        - Page format for fixed length records



Record id = <page id, slot #>. In the first alternative, moving records during free space management changes the rid, which may not be acceptable.

左边的就是按照顺序存入记录，前面的是全部存着的，而后面的是空的；而右边的则利用了一个东西来记录这个slot是不是空的，右边的明显比左边的更好，因为在左边的实现里，如果我们要删掉slot1的记录，我们就会把record id都给改变（向上移动）

    - Variable Length
        -



Record Formats: Variable Length

- Two alternative formats (# fields is fixed):

varchar

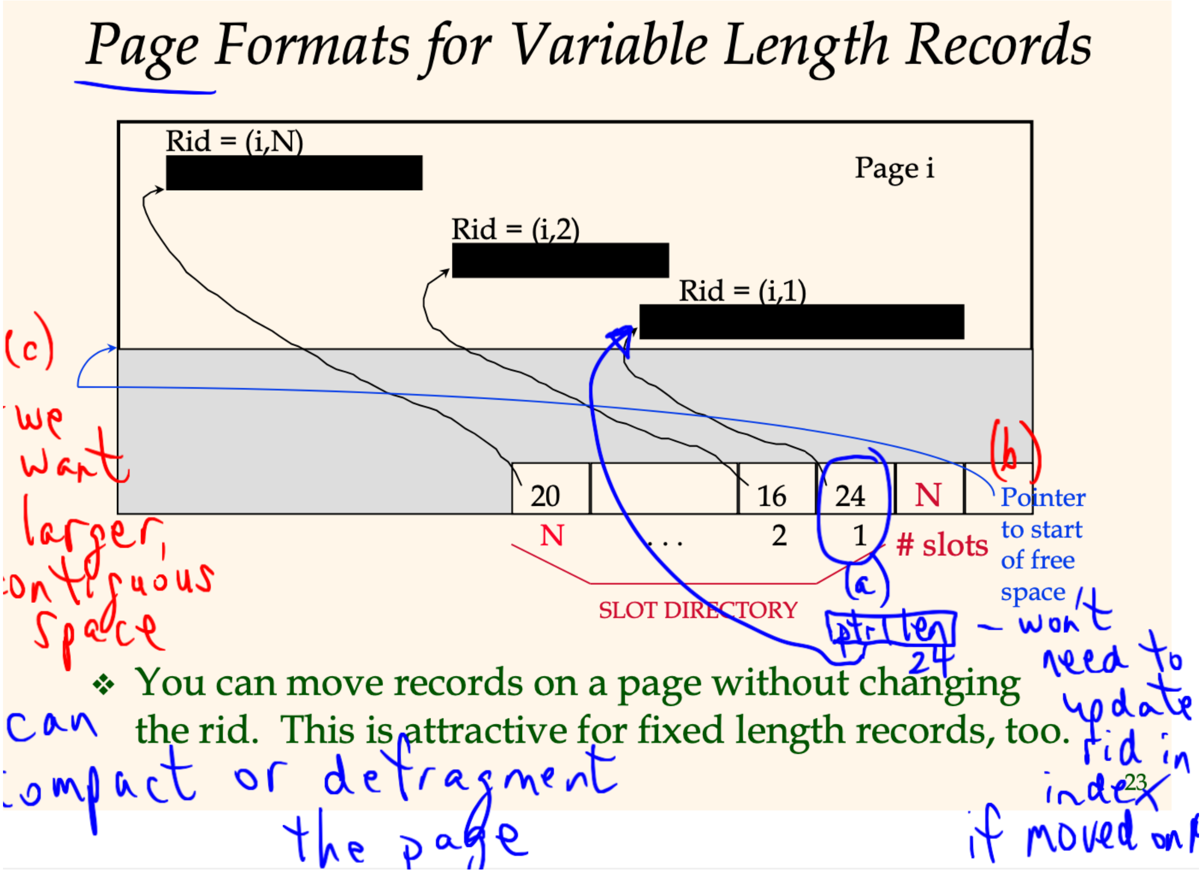
Why do we need this?

Suppose:
dog → cat OK
dog → elephant
OK but must adjust some things

- The second case offers: direct access to *i'th* field, efficient storage of *nulls* (a special *don't know* value), and small amount of directory overhead.

$start_i = start_{i+1}$

- 在第一种的record format里，前面有个小小的指示符指示这个记录中有多少个字段，然后后面的不同长度的字段间都会有个分隔符来把字段和字段分开
- 在第二种的record format里，前面分别先存着各个字段的base address，然后base address之后有个固定的分隔符和字段分开，接下来就可以找到各个字段
- Page format



Page Formats for Variable Length Records

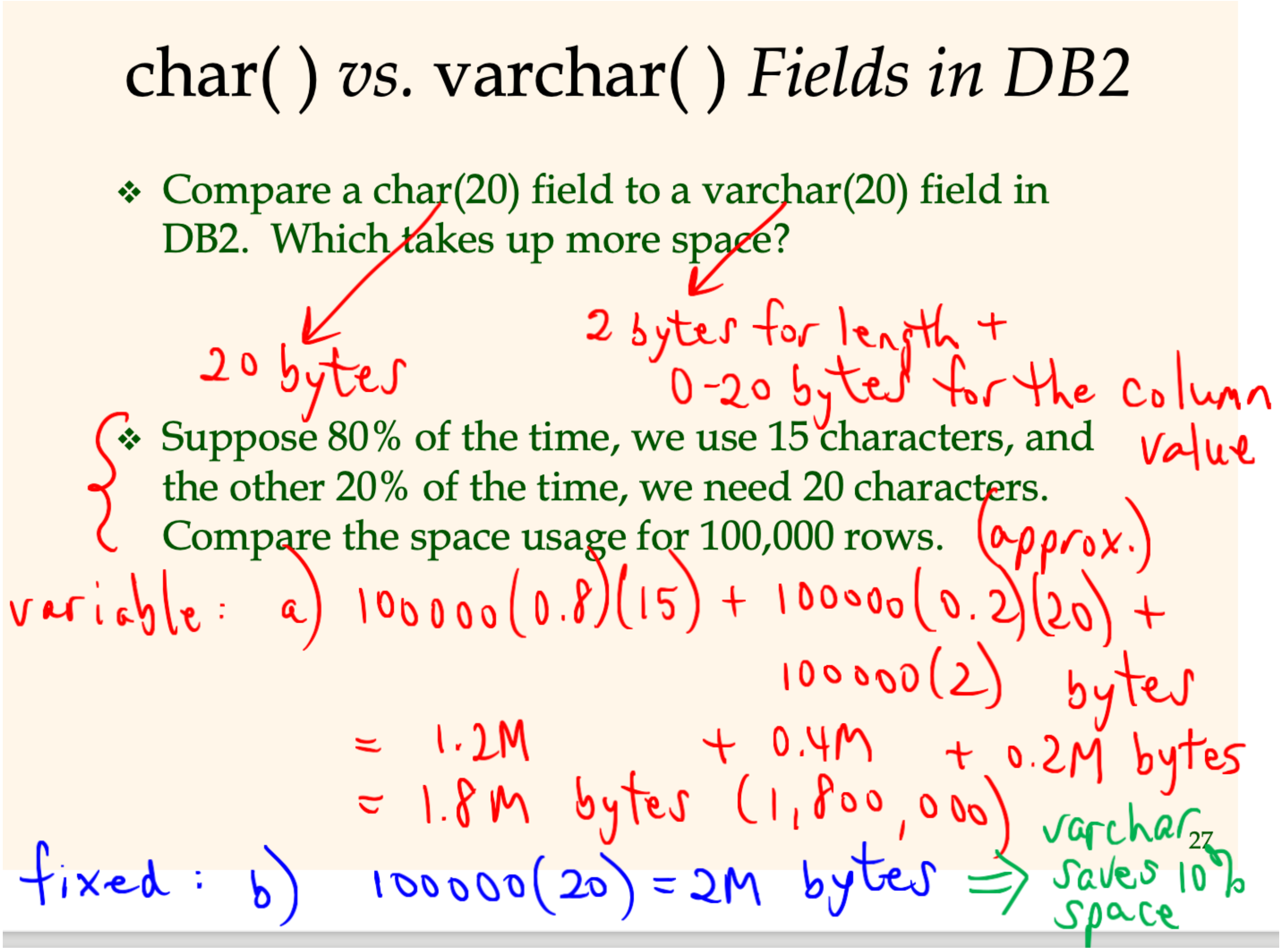
we want larger contiguous space

- won't need to update rid in index if moved on p

- You can move records on a page without changing the rid. This is attractive for fixed length records, too.

can compact or defragment the page

- PCTFREE、FREEPACE、REORG



char( ) *vs.* varchar( ) *Fields in DB2*

- Compare a char(20) field to a varchar(20) field in DB2. Which takes up more space?

20 bytes

2 bytes for length +
0-20 bytes for the column value

- Suppose 80% of the time, we use 15 characters, and the other 20% of the time, we need 20 characters. Compare the space usage for 100,000 rows. (approx.)

variable: a) 100000(0.8)(15) + 100000(0.2)(20) + 100000(2) bytes
= 1.2M + 0.4M + 0.2M bytes
= 1.8M bytes (1,800,000)

fixed: b) 100000(20) = 2M bytes ⇒ varchar saves 10% space

Lecture 01

MT2

Notation:

s^* 代表 data entry, <s, rid>

B+ tree 是一个 dynamic balanced tree structure

dynamic 指的是 easy to change the idea (grow & shrink)

balanced 指的是 height 不会疯狂增加

B+ tree 复杂度:

Search
Insert
Delete

$$\left.\begin{matrix}\text{Search}\\ \text{Insert}\\ \text{Delete}\end{matrix}\right\} O(\log_F N)$$

F : fan out : 一个 node 有多少个 child node

N : # leaf pages

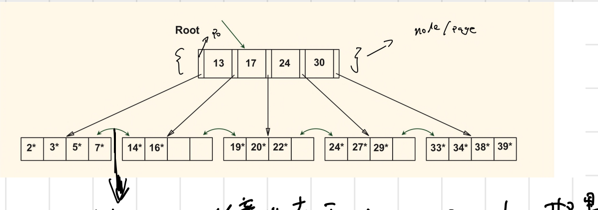
Fan out = K

root : 1 Page

height = 1 : K^1

height = 2 : K^2

- - -



这个 pointer 的意义在于 range search, 如果此时我们要找到 <19 的所有且没有 pointer 的话, 在找完 <13 这个 node 之后, 要重新从 root 找到 14* 这个 node 时间复杂度很高

dense vs Sparse

dense: one key, rid pair for each record in the table

sparse: one <key, rid> for a page, so in that page, every thing will be greater than key

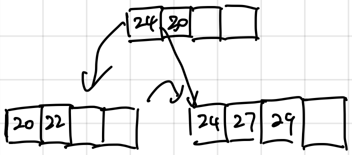
```
# Insert ( data, L)
  1. Find correct Leaf L
  2. Put data entry onto L
       if L has enough space, put!
       else:
           split (L)

Split (L) :
      if L is leaf page:
          1. find middle
          2. split L into $L_1$ and $L_2$ delimilited by
             the middle , $L_1 = L[\ :middle]$ , $L_2 = L[middle:\ ]$
          3. copy up the middle
               ↳ insert (middle, parent)
      else if L is internal page:
          1. find middle
          2. split L into $L_1$ and $L_2$ delimilited by
             the middle , $L_1 = L[\ :middle]$ , $L_2 = L[middle+1:\ ]$
          3. move up the middle
               ↳ insert (middle, parent)
      else if L is root page ;
          1. find middle
          2. split L into $L_1$ and $L_2$ delimilited by
             the middle , $L_1 = L[\ :middle]$ , $L_2 = L[middle+1:\ ]$
          3. New root with one Key: middle
          4. height ++;
      else:
          //
```

Redistribution

1. between leaf nodes

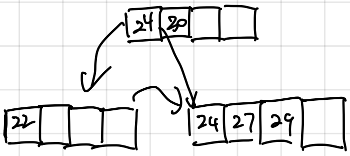
↓



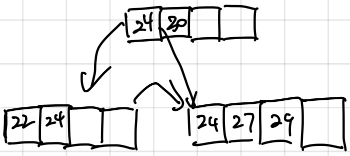
目前要删的是 20，但删完之后，minimum occupancy (2) 就无法满足

1. 检查是否满足 redistribute 的要求 (redisburite) (满足)

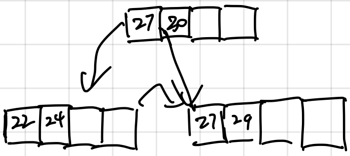
2、删除20*



3、Move 24* to ⤴



4、把 27* copy up，replace 24



2、between non leaf node

为什么要换 both 17，20 去隔壁 → 只换一个 20 也可以!

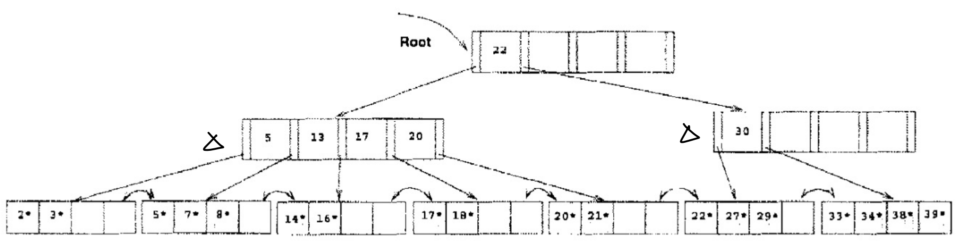
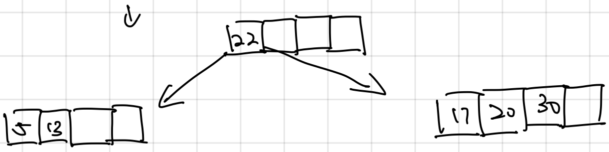


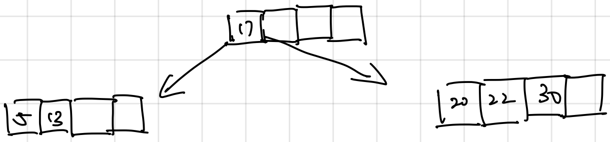
Figure 10.19 A B+ Tree during a Deletion

目前要 redistribute 的是 △ 的两个 node

1. 把 17，20 移动到右边的 node

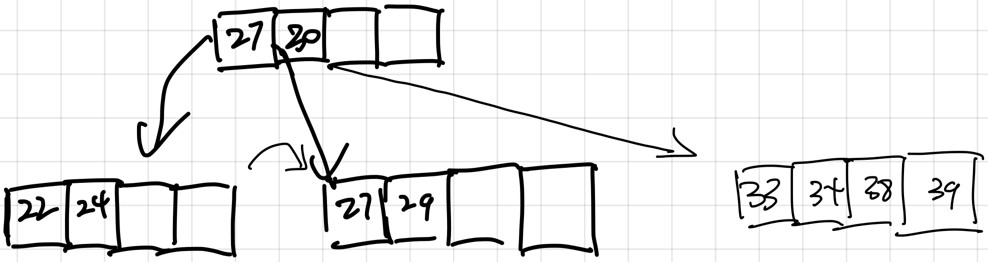


2、Replace 22 with 17



Merge

1. between leaf node



目前要删 24

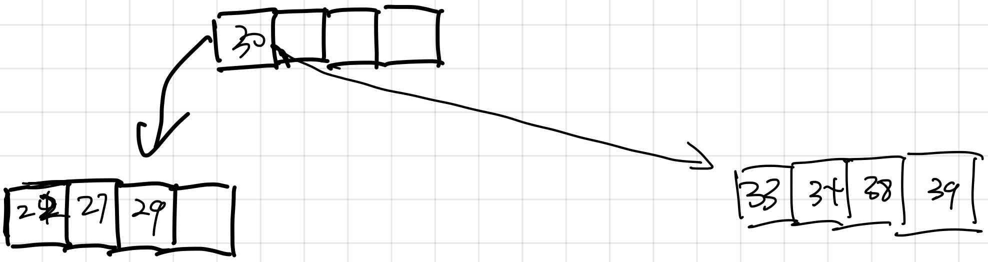
1. 检查能否 redistribute （不能）

2、检查能否 merge （能）

↳ 1、删除 24

2、Merge |22| | | 和 |27|29| | |

3、把 27 这个 ptr 串 上面删掉 （这里忽略这个 non leaf page 违反了 minimum occupancy 的要求）



2、between non leaf node

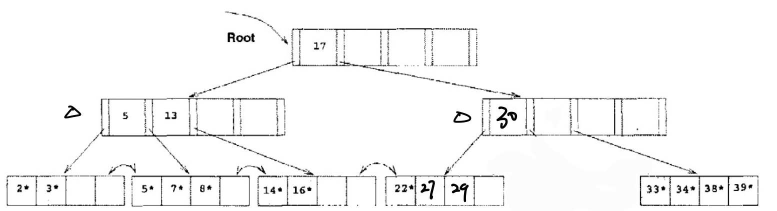


Figure 10.16 B+ Tree after Deleting Entries 19* and 20*

现在要merge △ 的两个 node

1、把 17 pull down

2、 merge

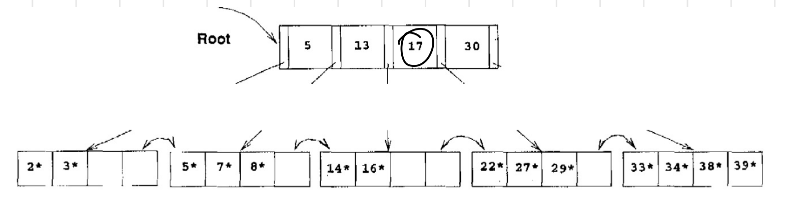


Figure 10.18 B+ Tree after Deleting Entry 24*

至此为止，完成了本页一开始的删除 24 的要求

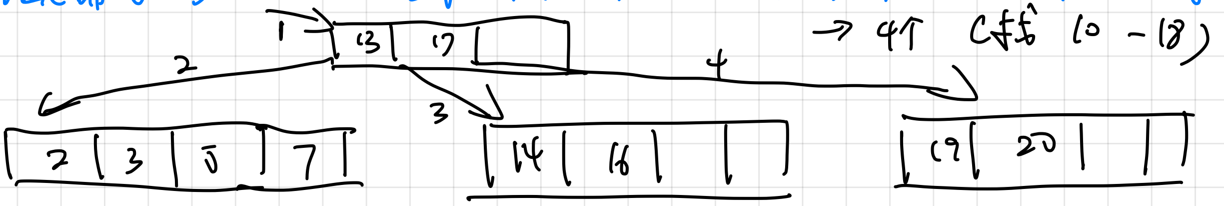
检查是否满足 redistribute 的要求

检查是否满足 merge 的要求

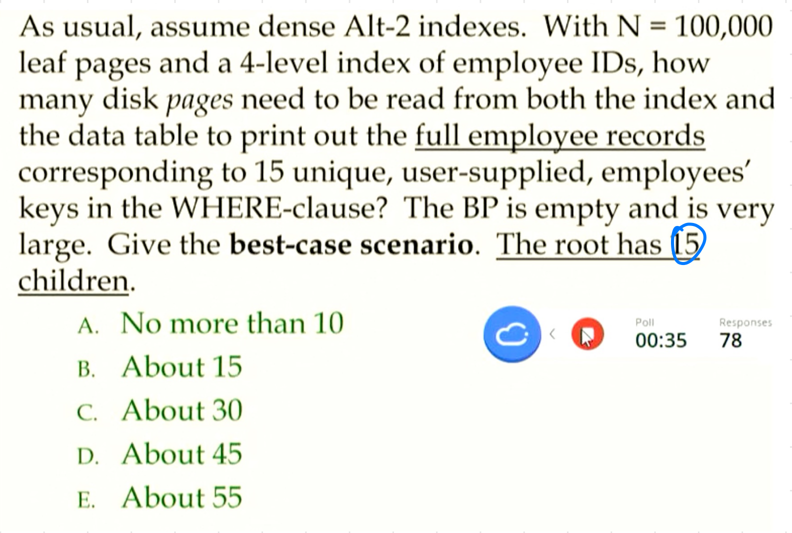
Lecture 02

\# Select every thing will never go to index page

\# 计算需要多少个 index page 才可以找到的时候，看一个 ptr 算一个 page → 4个 (共 10 - 18)



\#



As usual, assume dense Alt-2 indexes. With N = 100,000 leaf pages and a 4-level index of employee IDs, how many disk *pages* need to be read from both the index and the data table to print out the full employee records corresponding to 15 unique, user-supplied, employees' keys in the WHERE-clause? The BP is empty and is very large. Give the **best-case scenario**. The root has 15 children.

- A. No more than 10
- B. About 15
- C. About 30
- D. About 45
- E. About 55

\# 4-level : heigh 3 (0,1,2,3)

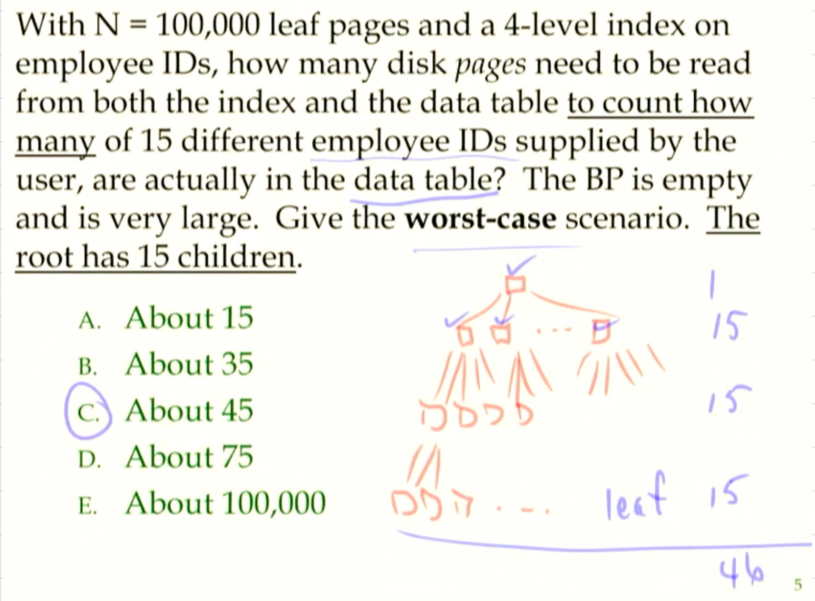
答案是 A

∵ best case scenairo

∴ index 就访问 4 遍

直接到 leaf

然后找 15个 k^*，其中 rid 指向的都是 一个 page



With N = 100,000 leaf pages and a 4-level index on employee IDs, how many disk *pages* need to be read from both the index and the data table to count how many of 15 different employee IDs supplied by the user, are actually in the data table? The BP is empty and is very large. Give the **worst-case** scenario. The root has 15 children.

- A. About 15
- B. About 35
- C. [✓ About 45]
- D. About 75
- E. About 100,000

\# Worst case

∴ root Always 1

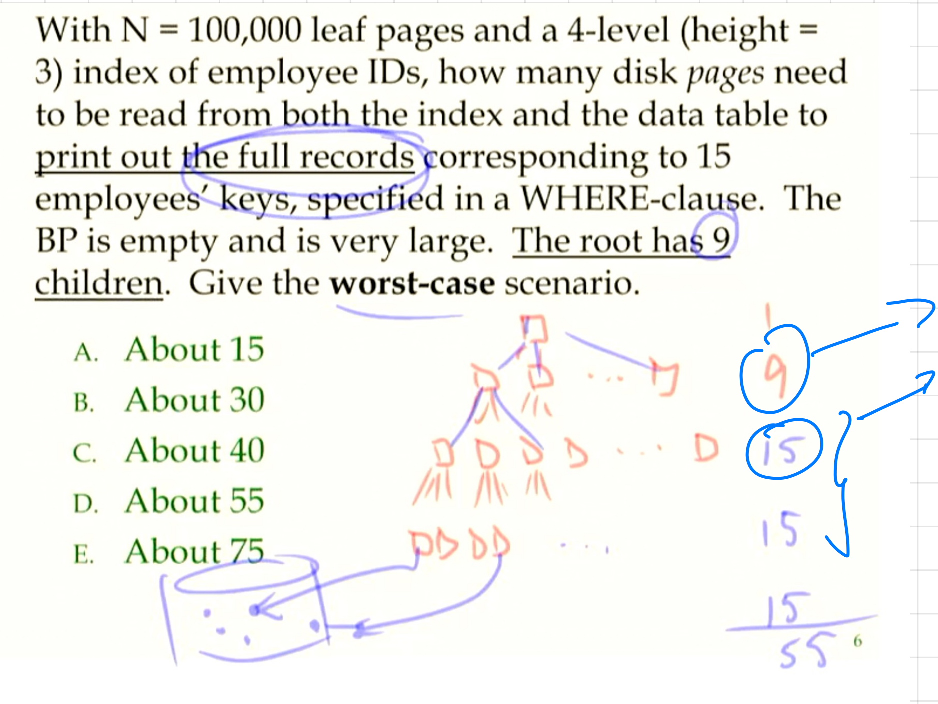
Each layer, always 15

因为我们在每一层，都需要访问 15个不一样的范围，所以 worst case 就是 15

然后定位到 leaf page

已经有 45 +1 pages from index

但不需要 data table



With N = 100,000 leaf pages and a 4-level (height = 3) index of employee IDs, how many disk *pages* need to be read from both the index and the data table to print out the full records corresponding to 15 employees' keys, specified in a WHERE-clause. The BP is empty and is very large. The root has 9 children. Give the **worst-case** scenario.

- A. About 15
- B. About 30
- C. About 40
- D. About 55
- E. About 75

这一层 9 是因为最多只有 9

这一层 15 是因为我们需访问 15 个不一样的

Lecture 03

—> 主要在讲 Exdendible hashing，在另一个文件中

Lecture 04

worst case scenairo for hashing : $O(N)$

* $h(key) = (a * key + b) \% N$ usually works pretty well

* a bucket is a page

Exdendible Hashing Calculation:

1. 根据 Alt * 来算出一个 data entry 所占的空间
2. 求出一个 index page 中可以有多少个 data entry
3. 总 record / # rec / page = # page

Lecture -1

cylindrification

phase 1 sort :

先算出一个 BP fill 需要的时间

Transfor time : # pages (number of pages in memory) · $\overbrace{x \text{ ms}/\text{pages}}^{\text{average transfor time}}$

\+ rotational delay : 0

\+ Long Seek : 10 ms

\+ Short Seek : (# cylinders (number of cyls in memory) − 1) · Short seek time

再算出 BP fill 的次数 : $\# \text{fill} = \dfrac{\text{fize size}}{\# \text{page in main memory}}$

再乘在一起 : $\underset{\substack{\downarrow \\ \text{R \& W}}}{2} \cdot \text{BP time} \cdot \# \text{fill}$

phase 2:

算出 # cylinder needed for file

再算出 time for 1 cylinder

Transfor time : # pages (number of pages in cylinder) · $\overbrace{x \text{ ms}/\text{pages}}^{\text{average transfor time}}$

\+ rotational delay : 0

\+ Average Seek : 10 ms (data - dependent 了，现在不同的数据都散落在不同的 SR，每次用的时候都 assume worst)

再算总时间：

$\underset{\substack{\swarrow \\ \text{R \& W}}}{2} \cdot \# \text{cylinder} \cdot \text{time}/\text{cylinder}$

* Cylindrification

在 phase 2 中，改变 input buffer 和 output buffer 对总的时间的影响

总时间： $\underset{\downarrow \atop R \cdot W}{2} \cdot \# \text{cylinder} \cdot \text{time}/\text{cylinder}$

R： # cylinder · time/cyl

W： # cylinder · time/cyl

注意，增加 # cylinder output buffer 会减少 writing 的时间

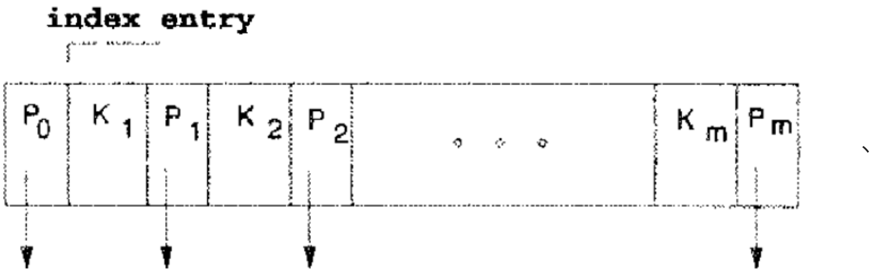
∵之前是 merge 好了之后，从 disk 上 seek 到位置 (long seek) 再写一个 cyl，所以如果 output buffer 有 1 cyl，那么每个写入的 cyl 都要经历一次 LS，但如果 output buffer 上升到 2 cyl 的话，现在每两个 cyl 做一次 LS，快很多！ → $W = \frac{\# \text{cylinder}}{\# q \text{ in output buffer}} \cdot \text{time}/\text{cyl}$

Lecture

2021-10-07

Pre Class

- index entry: <key, pointer>，然后key可以用某个field的值来替代，这样就可以根据key的值来进行快速搜索



- 上面这样子的一个东西就是index page，然后在一个index page里面，key的数量都比ptr的数量多一，因为key的作用是separator

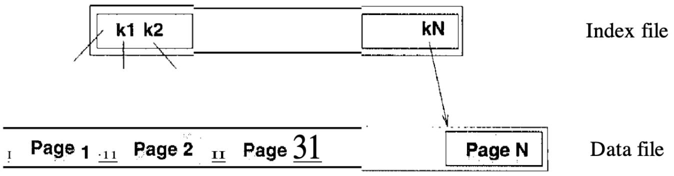


Figure 10.2 One-Level Index Structure

  - 动机：index file虽然比data file要小很多，但是他依然可以令insert和delectable很麻烦，所以我们何不在一堆index file上面，再建立一个index file：递归的建立index file until the smallest auxiliary structure fits on one page
- B+ Tree： **a balanced tree in which the internal nodes direct the search and the leaf nodes contain the data entries.**

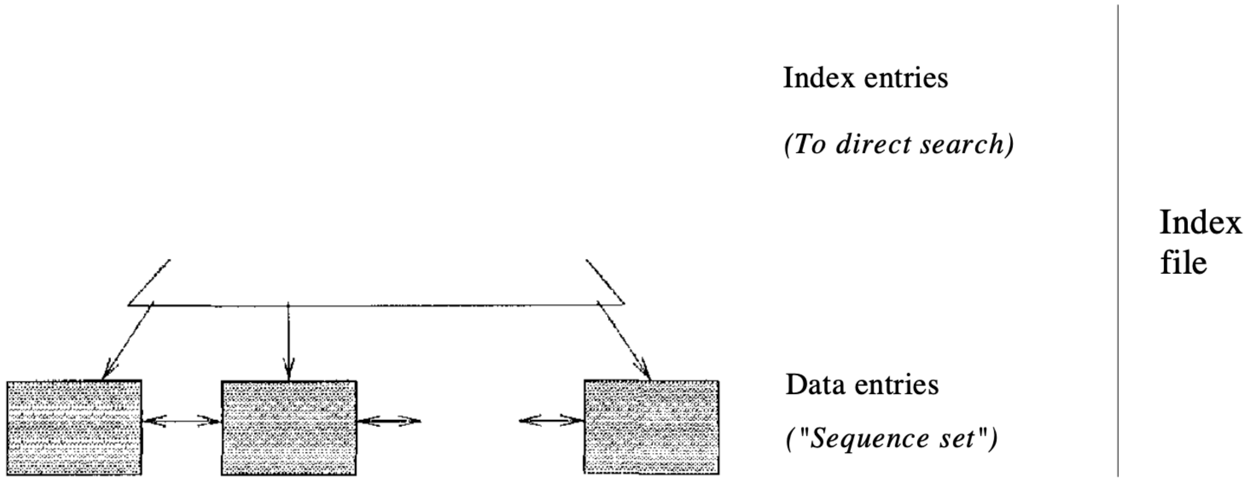
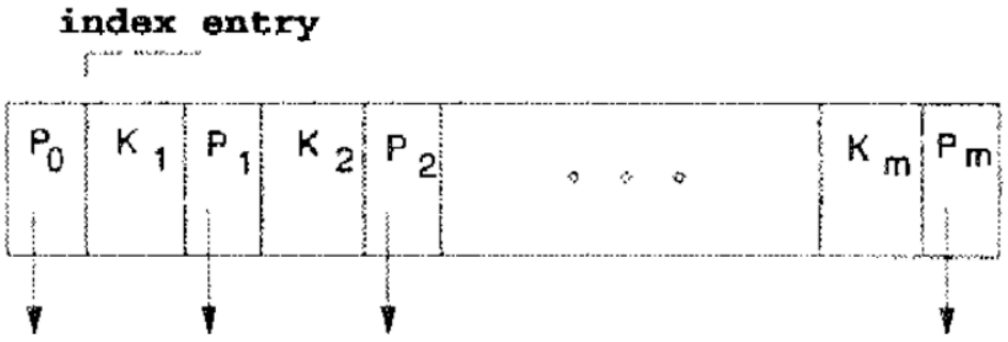


Figure 10.7 Structure of a B+ Tree

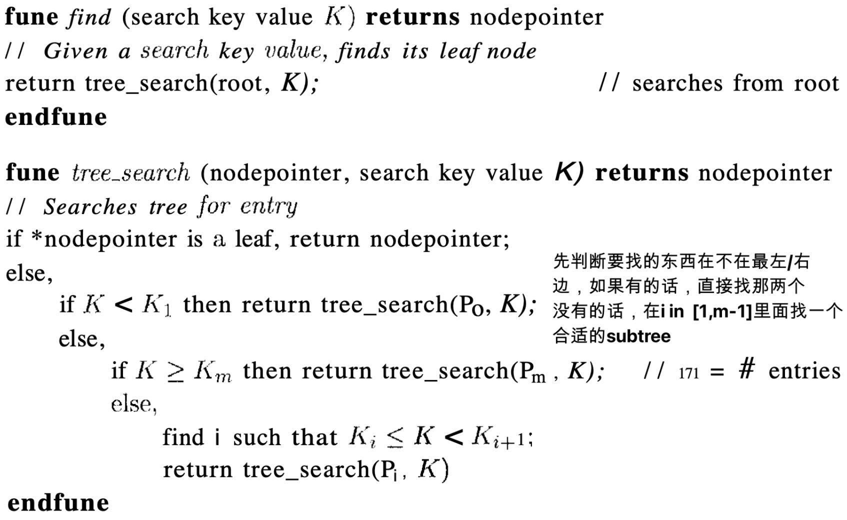
  - 还真的是只有leaf才存真正的data entry
  - A **data entry** is a <k, rid> pair, where rid is the record id of a data record with search key value k.
  - 其实entry都是这样的key ptr pair，只不过data entry pointer中要么是指向data page的指针，要么直接是data，data entry的ptr直接指向数据；而index entry的pointer指向的则是下一个index page
  - Operations (insert, delete) on the tree keep it balanced.
  - Minimum occupancy of 50 percent is guaranteed for each node except the root
  - Format of a Node
  - non-leaf node

    经典图重温：



        - P0指向的subtree中，所有的key都小于K1
        - if $0 < i < m$, Pi指向的subtree中，所有的key K都$k_i < K \le k_{i+1}$
        - Pm指向的subtree中，所有的key都大于K_m
    - leaf node: denoted as $k*$
        - 并且所有的leaf node都被chained成doubly linked list

- Search：



```
fune find (search key value $K$) returns nodepointer
// Given a search key value, finds its leaf node
return tree_search(root, $K$);                    // searches from root
endfune

fune tree_search (nodepointer, search key value $K$) returns nodepointer
// Searches tree for entry
if *nodepointer is a leaf, return nodepointer;
else,
    if $K < K_1$ then return tree_search($P_0$, $K$);
    else,
        if $K \geq K_m$ then return tree_search($P_m$, $K$);    // 171 = # entries
        else,
            find i such that $K_i \leq K < K_{i+1}$;
            return tree_search($P_i$, $K$)
endfune
```

先判断要找的东西在不在最左/右边，如果有的话，直接找那两个没有的话，在i in [1,m-1]里面找一个合适的subtree

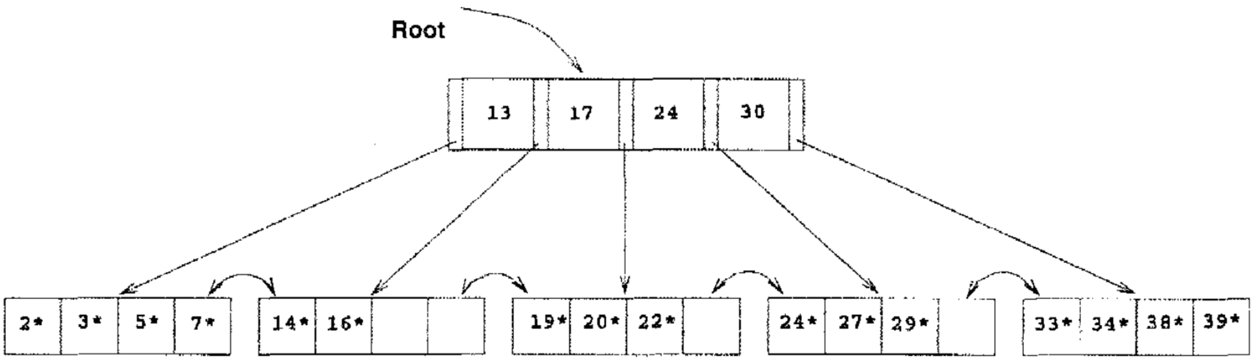


Figure 10.9 Example of a B+ Tree, Order d=2

2021-10-12

Pre Class

- insert的时候，如果一个node满了，并不是非得要split他，也可以尝试redistribute他，比如上面那个10.9的树，如果这时候要加入8进来，可以发现左数第二个node还剩下两个位置，这个时候完全可以把8插进去

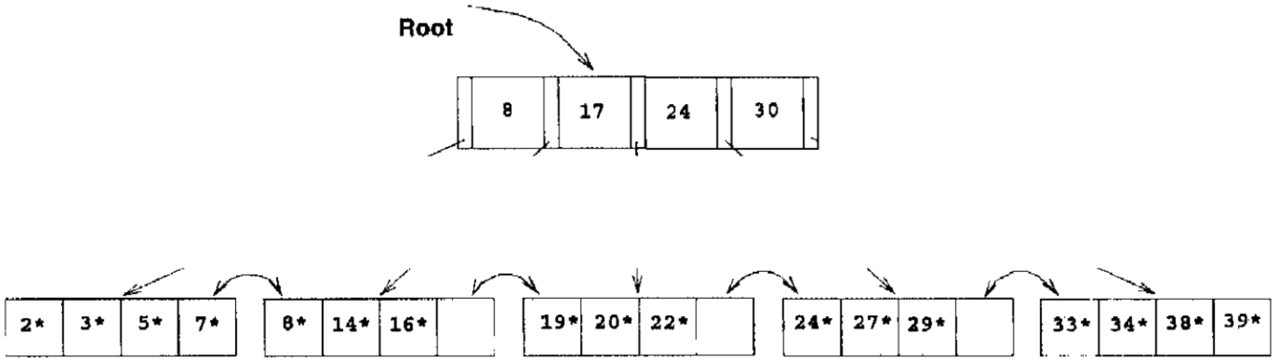


Figure 10.14 B+ Tree after Inserting Entry 8* Using Redistribution

但是并不是每次insert都满足redistribution的条件的，如果sibling满了的话就不行，所以需要先检查，再看看是split还是redistribution。但是如果这个insertion发生在leaf level的话，我们本来就要retrieve neighbour node，因为prev-next neighbour存在于leaf node，当insert发生的时候，即使要split，我们也需要调整prev-next neighbour（split的话在现有的siblings中间加多了一个node）

- delete
    - The situation when we have to merge two non-leaf nodes is exactly the opposite of the situation when we have to split a non-leaf node. We have to split a non- leaf node when it contains $2d$ keys and $2d+1$ pointers, and we have to add another key--pointer pair. Since we resort to merging two non-leaf nodes only when we cannot redistribute entries between them, the two nodes must be minimally full; that is, each must contain d keys and $d+1$ pointers prior to the deletion.
    - redistribute的意思就是，当我把目标要删除的值删除之后，目标要删除的值所在的page不满足minimum occupancy的要求，但是发现如果从sibling拉一个entry过来的话，**目前的page和sibling都符合minimum occupancy的要求**，那么就拉，这就叫做redistribute
    - merge的意思就是，当我把目标要删除的值删除之后，目标要删除的值所在的page不满足minimum occupancy的要求，但是发现如果从sibling拉一个entry过来的话，**目前的page和sibling都不符合minimum occupancy的要求**，并且发现，如果和目前的page和sibling的entry放在一起的话，entry的数量正好在d和2d中间，那么这时候就把他们放在一起，这就是merge
    - 书里面提出的小小的refinement就是，在检查sibling的时候检查两个sibling
    - **redistribution is guaranteed to propagate no further than the parent node.**

2021-10-14

Pre Class

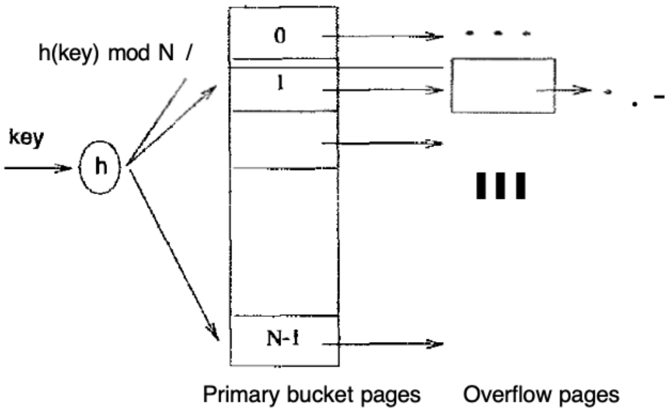


Figure 11.1 Static Hashing

- static hashing scheme是一种存储模式
- 很像一个bucket的集合，其中每个bucket都配备了一个possibly additional overflow pages
- 一个bucket装的是data entry（三个alternative）
- 一个文件享有bucket a到N-1
- Search：

  searching a bucket requires us to search (in general) all pages in its overflow chain

  - apply hash function to data entry : $hash(data)$
  - use $hash(data)$去找到bucket
- Insert：
  - apply hash function to data entry : $hash(data)$
  - use $hash(data)$去找到对应的bucket
  - 如果bucket没满：放入data
  - 如果bucket满了：
    - 在这个bucket后面allocate一个新的overflow page
    - 把data放在page上
    - 把这个page加入这个bucket的overflow chain上
- Delete：
  - apply hash function to data entry : $hash(data)$
  - use $hash(data)$去找到对应的bucket
  - 如果data不是在overflow chain中的一个overflow page上或者在overflow chain中的一个overflow page但不是这个page的最后一个：直接删掉
  - 如果data在overflow chain中的一个overflow page但是这个page的最后一个：

        - data删掉
        - 把page从这个bucket的overflow chain中移除，放入free page
- If we have *N* buckets, numbered athrough *N* ~ 1, a hash function *h* of the form *h(value) = (a *value + b)* works well in practice. (The bucket identified is *h(value) mod N.)*
- the number of buckets is fixed.
- Note that two different search keys can have the same hash value.
- **Extendible Hashing**

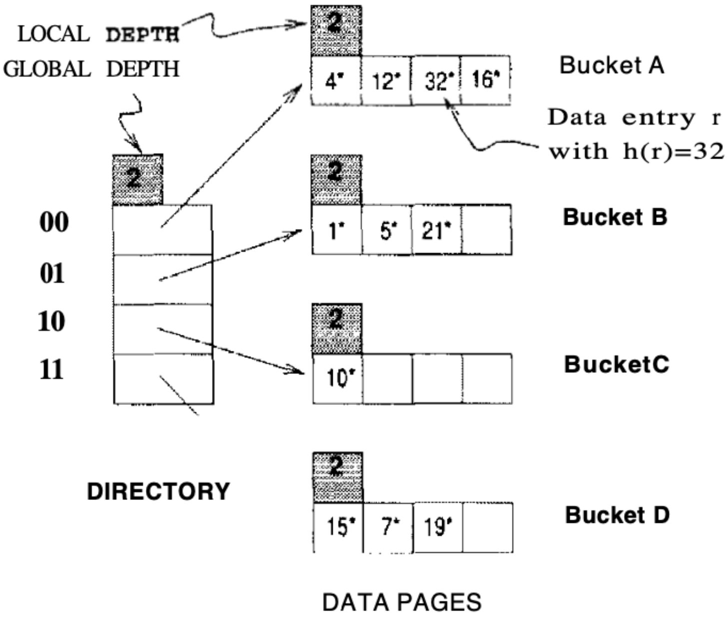


Figure 11.2 Example of an Extendible Hashed File

把bucket这一层给分隔开，先用一个directory去指向bucket

Search for data entry<k, rid>：

- 先算出hash（k）
- 使用hash（k）的二进制表示的后两位去定位到具体的bucket
- 从bucket中找到具体的数字

Insert a data entry <k, rid>

- 先算出hash（k）
- 使用hash（k）的二进制表示的后两位去定位到具体的bucket
- 如果bucket有位置：放入到bucket
- 如果bucket没有位置：
    - 需要split满了的bucket
        - 重新计算bucket中所有的hash(k)

        - 根据hash(k)的二进制表示的后三位去来区分现在的bucket和bucket image
        - 如果新的bucket image的bit并不在directory里面的话：
            - double现在的directory以存放bucket

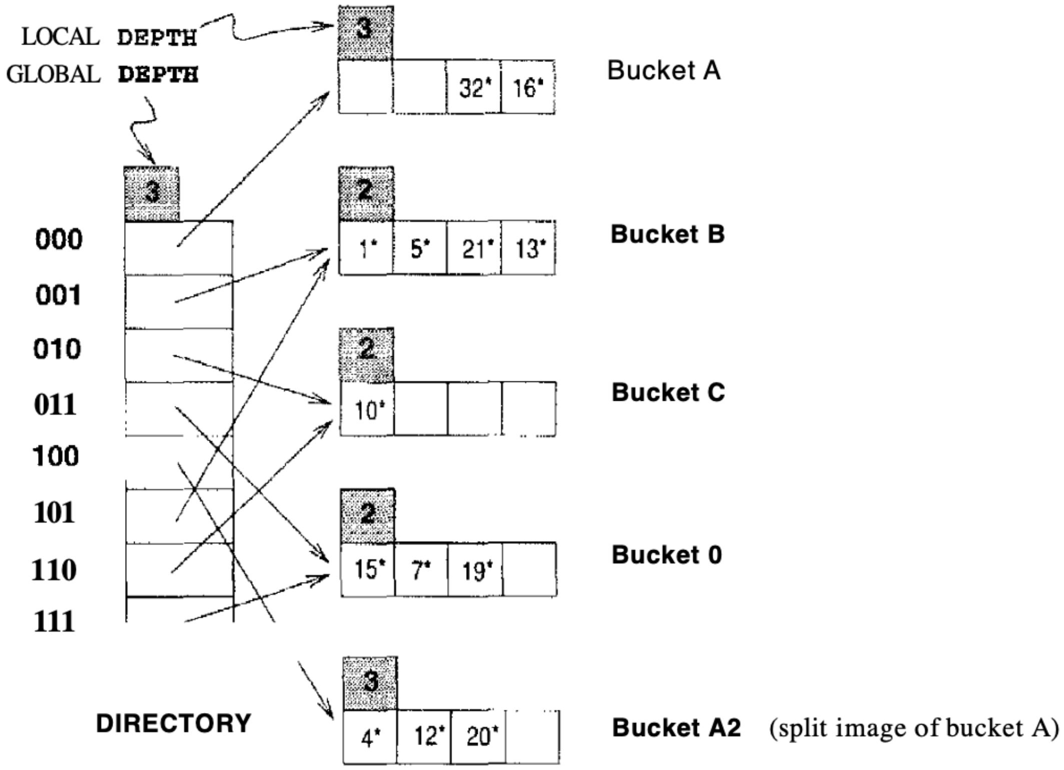


Figure 11.5 After Inserting Entry r with $h(r)$ = 20

- d指的是directory所需要的位数，成为global depth，当directory doubling occur，global depth++；local depth是per bucket，当bucket split了一次之后，local depth++，并且split出来的image bucket会有和之前的bucket一样的local depth（这一轮split加了一之后）
- 如果d == local depth，当这个local depth的bucket被split之后，必须要double directory
- deletion:
    - if the removal of a data entry makes a bucket empty, the bucket can be merged with its split image
    - if each directory element points to same bucket as its split image, we can halve the directory
- Bulk Loading：
    - The first step is to sort the data entries according to a search key in ascending order.
    - We allocate an empty page to serve as the root, and insert a pointer to the first page of entries into it.
    - When the root is full, we split the root, and create a new root page.
    - Keep inserting entries to the right most index page just above the leaf level, until all entries are indexed.

2021-10-19

Pre Class

- Linear Hashing
- 需要利用family of hash functions: $h_a, h_1, h_2...$ with the property that each function's range is twice that of its predecessor.
    - 怎么做到的：by choosing a hash function hand an initial number *N* of buckets 2 and defining h_i(value) = *h(value) mod* (2^iN)
    - If *N* is chosen to be a power of 2, then we apply *h* and look at the last d_i bits; *do* is the number of bits needed to represent *N,* and d_i = d_0 + i.
    - 比如N是32，那么d0就是5（d0 is the number of bits needed to represent N) ,h0就是简单的h mod 32，h1就是h mod （$2^1 \times 32$)
    - level: indicate the current round number and is initialized to 0
    - Next: the bucket to split
    - N_{Level}: number of buckets in the file at the beginning of round Level

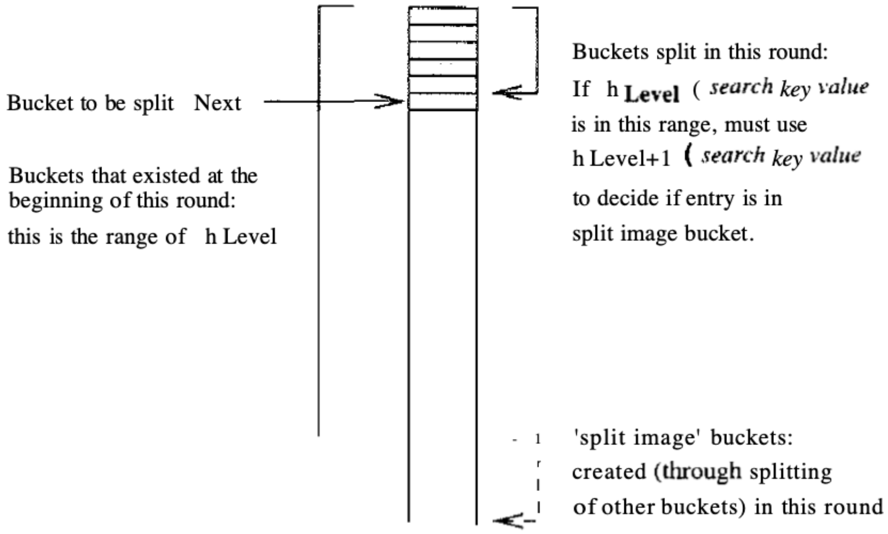


Figure 11.7 Buckets during a Round in Linear Hashing

    - basically, 在round level里面，只有h_{level}和$h_{level+1}$会被使用，file前面的bucket会被一个一个的split，这样一轮下来，bucket的数量会乘二
    - Search for <k, rid>：
        - calculate $h_{level}(k)$
        - 找到bucket，看看是否被split了
            - split了：用$h_{level+1}(k)$去查

        - 没split：直接在这个bucket里面找
  - For our examples, a split is 'triggered' when inserting a new data entry causes the creation of an overflow page.
  - Split：
    - use hash function $h_{level+1}$ to redistribute entry between current and split image
    - assign bucket number b +N_{level} to the split image(b是现在的bucket)
  - Insert <k, rid>：
    - 根据$h_{level}(k)$找到对应的bucket，看看有没有被split
    - 被split了，用$h_{level+1}$找到这个k归属于现在的bucket还是split image，找到对应的位置，看看满了没满（包括之前存在的overflow page）
    - 没满：直接加入
    - 满了：
      - split current！要注意的是没，这个insert触发了一次split，但这个split并没有发生在overflow的地方，而发生在目前next所指向的地方，round robin轮到的位置
      - 把现在的这个k加入这个bucket的overflow page
      - next++
  - 只有split了之后才会increment next

2021-10-21

Pre Class

- internal sort用的都是in memory sort，比如quick sort
- Run: refer to each sorted subfile. 就比如merge sort创造的那些什么copy
- Simple Two Way Merge Sort：
  - 这玩意儿和merge sort很像，就是merge sort breaks down了之后

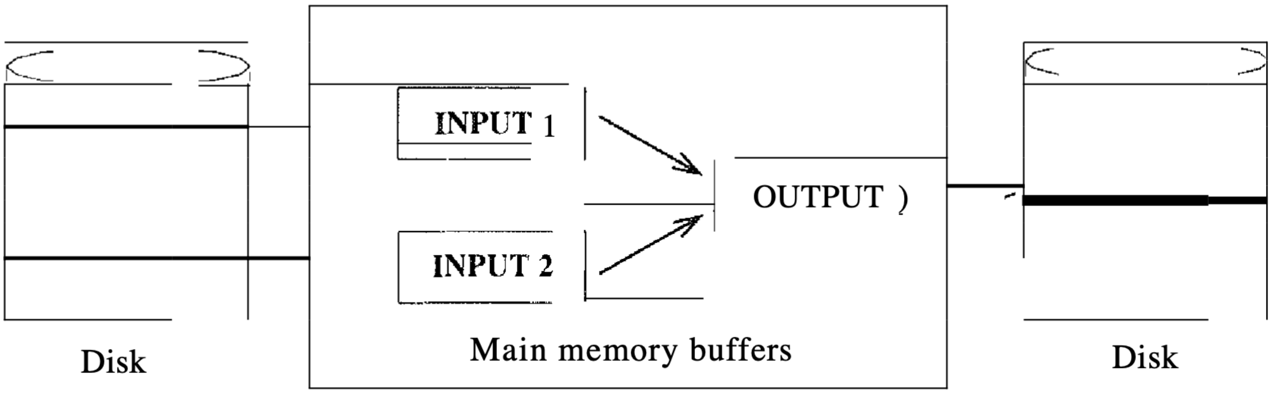


Figure 13.3 Two-'Way Merge Sort with Three Buffer Pages

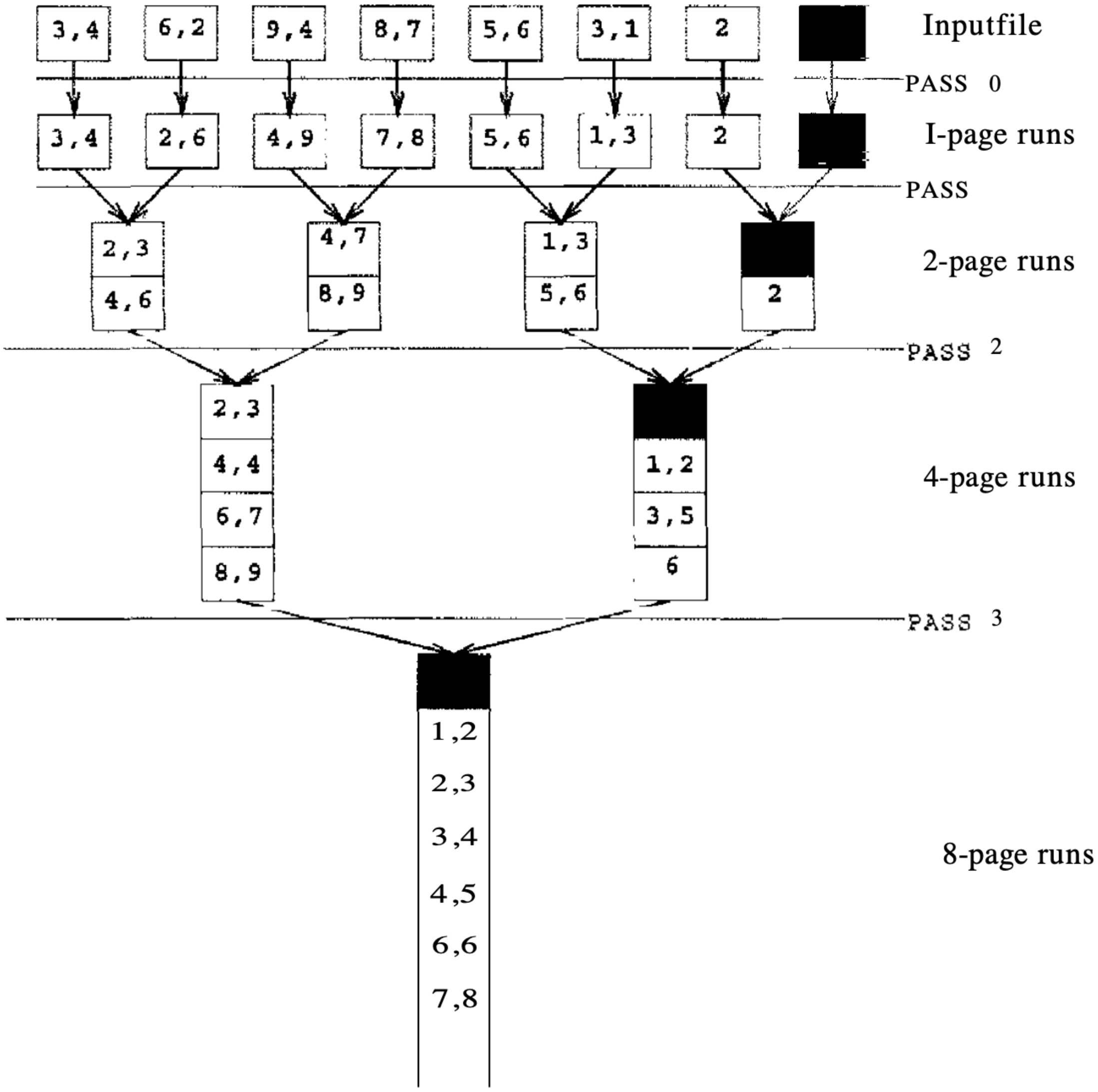


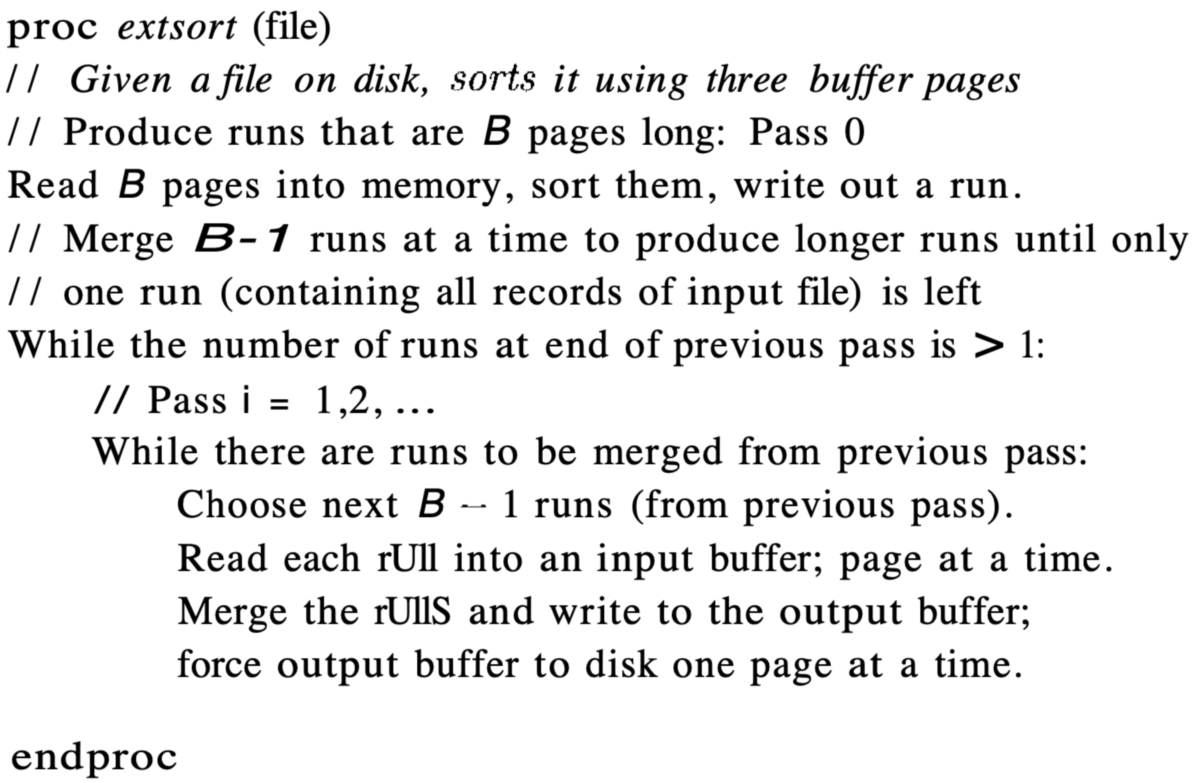
Figure 13.2 Two-Way Merge Sort of a Seven-Page File

第一个pass有2^k个page，之后每一个pass：k--（两两合并）

number of passes: $[log_2N]+1$，N是number of pages, 每一轮，每一个page有一个io

overall cost：$2\times N\times([log_2N]+1)$

- External Merge Sort(对上述的refinement)



```
proc extsort (file)
// Given a file on disk, sorts it using three buffer pages
// Produce runs that are B pages long: Pass 0
Read B pages into memory, sort them, write out a run.
// Merge B-1 runs at a time to produce longer runs until only
// one run (containing all records of input file) is left
While the number of runs at end of previous pass is > 1:
    // Pass i = 1,2, ...
    While there are runs to be merged from previous pass:
        Choose next B - 1 runs (from previous pass).
        Read each rUll into an input buffer; page at a time.
        Merge the rUllS and write to the output buffer;
        force output buffer to disk one page at a time.

endproc
```

Figure 13.6 External Merge Sort

  - refinement1：

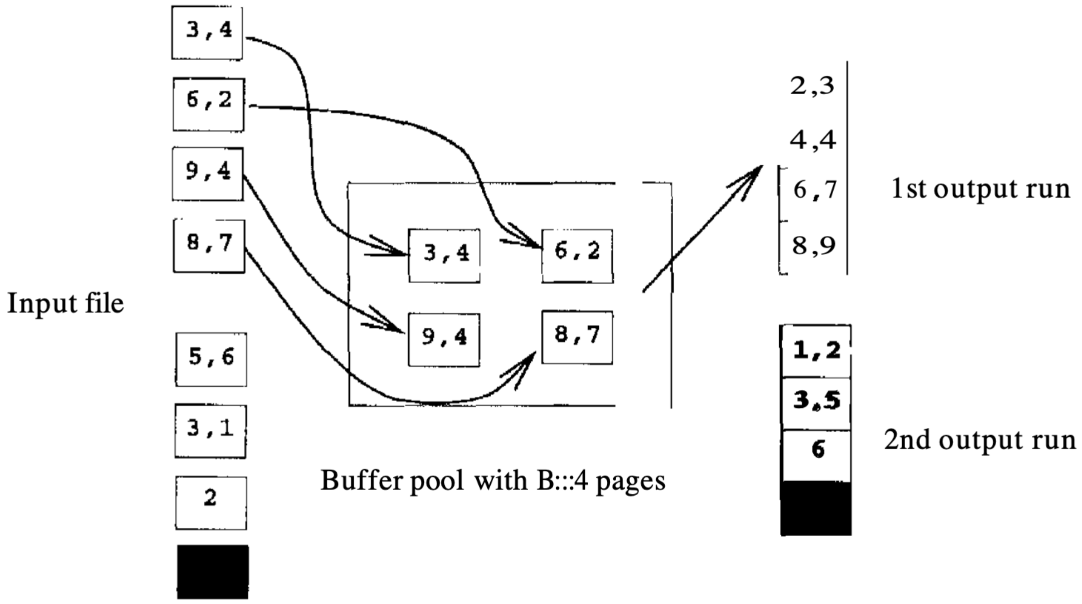


Figure 13.4 External Merge Sort with B Buffer Pages: Pass 0

    在pass 0的时候，之前只是简单的读，然后sort internally一个page；比如[2, 1] -> [1, 2]；而现在是一下读B个page，sort这B个page，像上面的图展示的那样，一下子读了四个page；这样可以把run从N减少到N/B

  - refinement2:

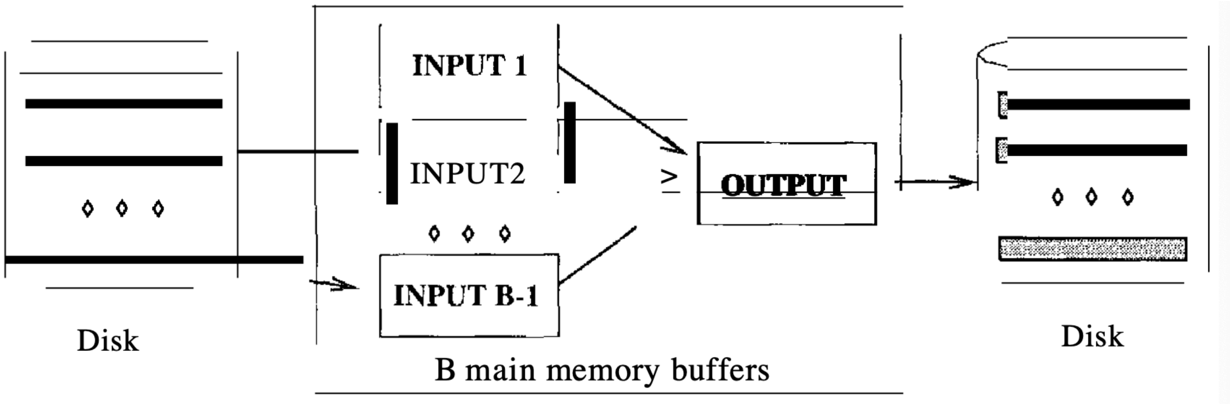


    Figure 13.5 External IVlerge Sort with B Buffer Pages: Pass $i > 0$

    每一轮，用B-1个page做input，然后一起merge到剩下的那个output

  - In doing a *(B* - I)-way merge, we have to repeatedly pick the 'lowest' record in the *B-1* runs being merged and write it to the output buffer. This operation can be implemented simply by examining the first (remaining) element in each of the *B - 1* input buffers.

- External Merge Sort
  - 比如现在要sort 108个pages，有5个buffer pages
    1. 利用5个buffer pages，五个五个地sort这108个pages，创造ceil(108/5) = 22的sorted run
    2. 把四个buffer pages当作是input，一个buffer page当作是output，然后四个四个sorted run这样merge，merge成ceil(22/4) = 6个sorted run
    3. 把六个buffer pages当作是input，一个buffer page当作是output，然后四个四个sorted run这样merge，merge成ceil(6/4) = 2个sorted run
    4. 最后两个sorted run，merge一次，变成1个sorted run，就merge成功

2021-10-26

Pre Class

- clustered index：该索引中键值的逻辑顺序决定了表中相应行的物理顺序。所以说如果一个table是clustered index的话，我们只需要扫过他的b+ tree就好了，cost相当于traverse the tree from root to the left most leaf, plus the cost of retrieving the pages in sequence set, plus the cost of retrieving the pages containing the data records

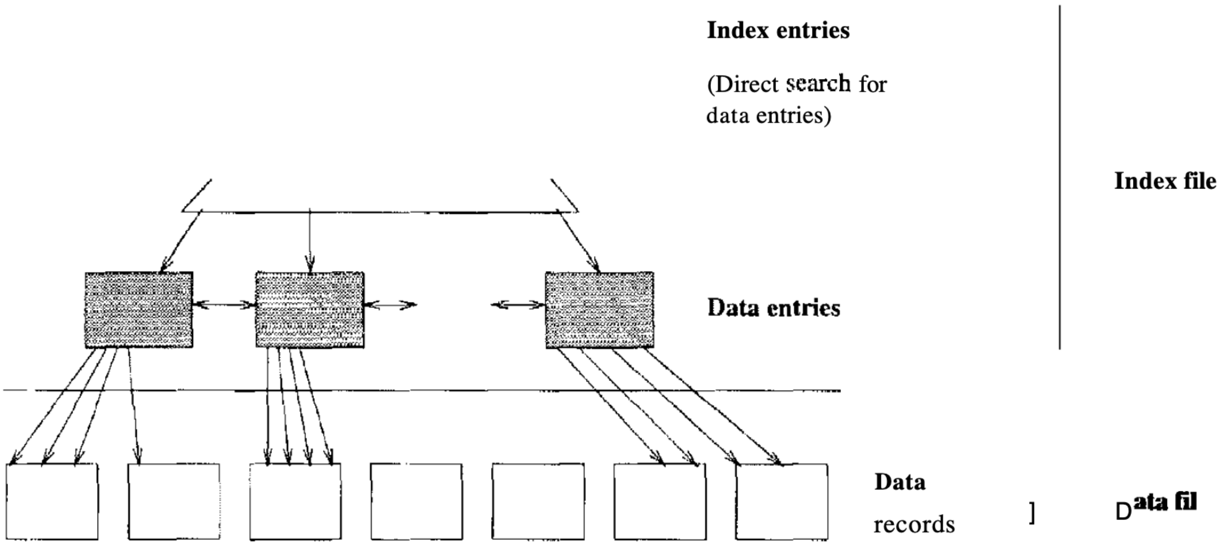


Figure 13.11 Clustered B+ Tree for Sorting

- unclustered index: 就是说键值的逻辑顺序和物理顺序无关

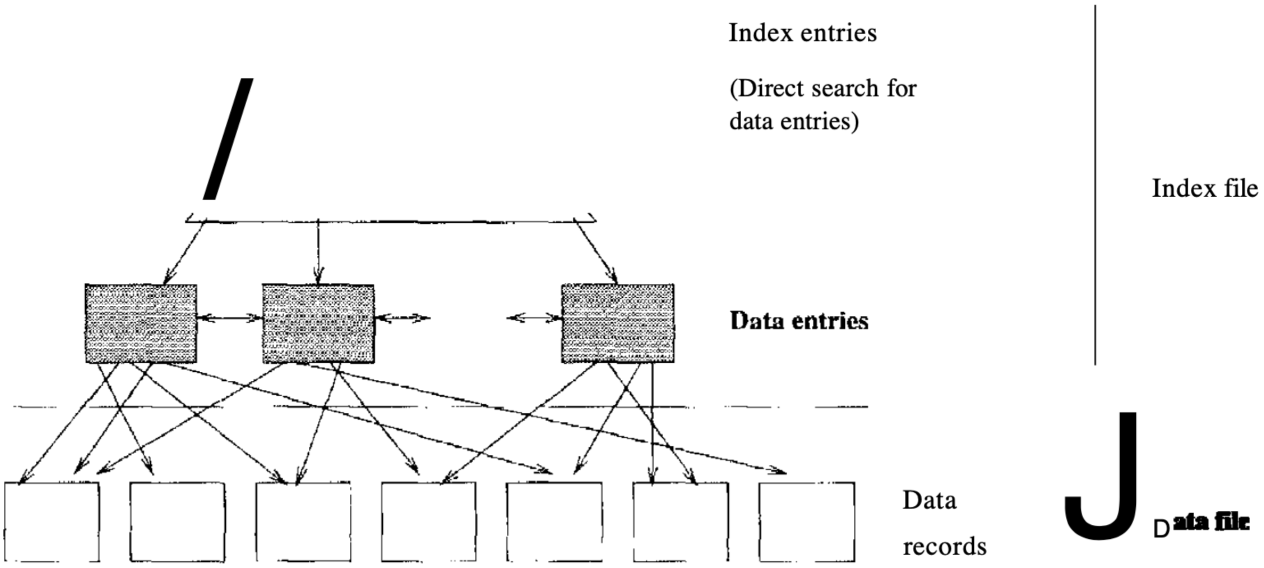


Figure 13.12 Unclustered B+ Tree for Sorting

- worst case等于data records的数量
- double buffering: 把所有的bp分为两部分，在使用一半的bp的同时，refill另一半bp

- # Chapter 10 Tree-Structure Indexes

Learning Goals

- List the 3 ways or “alternatives” of representing data entries k* having key k, in a general index
    1. whole data record with search key k: <k, record>
    2. <k, rid> where the rid signifies the location of a jingle data record (row in the actual table) that has search key k
        - that rid points to the **location** instead of the real data of the data entry
    3. <k, list of rids of data table records having search key k>
- Justify the use of indexes in database systems.
    - Using indexing可以
- Explain how a tree-structured index supports both range and point queries.
- Build a B+ tree for a given set of data. Show how to insert and delete keys in a B+ tree.
- Analyze the complexity of: (a) searching, (b) inserting into, and (c) deleting from, a B+ tree.
- Explain why a B+ tree needs sibling pointers for its leaf pages.
- Explain why B+ trees tend to be very shallow, even after storing many millions of data entries. Equivalently, provide arguments for why B+ trees can store large numbers of data entries in their pages.
- Explain the pros and cons of allowing prefix key compression in an index.
- Given a set of data, build a B+ tree using bulk loading.
- Provide several advantages of bulk loading compared to using a large number of SQL INSERT statements.
- Estimate the number of I/Os required to traverse the records of a large file (whose records have key k) using B+ trees for: (a) the clustered case, and (b) the unclustered case. Justify any assumptions that are needed.
- Using a diagram, show the principal difference between a clustered and an unclustered index.
- Provide examples of when clustering doesn’t help performance, and may actually hinder the performance of certain kinds of queries.

Take away

- index索引指的是一个建立在data entry上的东西，如果没有index的话，那么在搜索某一个具体条件的data entry的时候，就得按照顺序一条一条的找，然后看看这个数据是不是符合要求，建立index的话，可以有一个类似<index, data>的东西，我们可以先通过索引定位到具体的data，再去找这个data。而存放index的方式也可以被优化

- 所以tree structured indexing指的是存放k这个index的技巧
- B+ tree
  - m is the number of keys in a node
    - is m consistent for every node?
  - for each node, we want to occupy at least 50% of the node's available entries:
    - each node contains d <= m <= 2d entries
  - d is called the order of the tree
  - fan-out = m+1 => number of pointers to child ndoes in a node
  - N = number of leaf + internal pages => 所以是#total pages - 1?
  - height指的是总的层数（包括root）-1
  - 基本上d决定着一个node有多少个entries，m则决定了一个node指向了多少个child nodes
  - look up
    - look up b+ tree基本上和look up binary tree的操作是一样的
  - insertion
    - copy up VS move up
      - copy up发生在leaf node，当需要split leaf node的时候，把leaf node的中间要分割的地方取出来，然后copy到parent中，这个时候的parent node和leaf node都各有一个要分割的东西
      - move up发生在parent/root node，当需要分割root node的时候，会直接把delimiter给向上移动，这个是时候，delimiter只会存在于新的root
  - deleting
    - look up
    - delete
    - check for needs for redistribute
      - 如果删了之后node的entry数量少于d，那么就需要redistribute
        - borrow entries from adjacent nodes that have same parent
        - if redistribute fails -> merge

MT3

Lecture 01 & Lecture 02

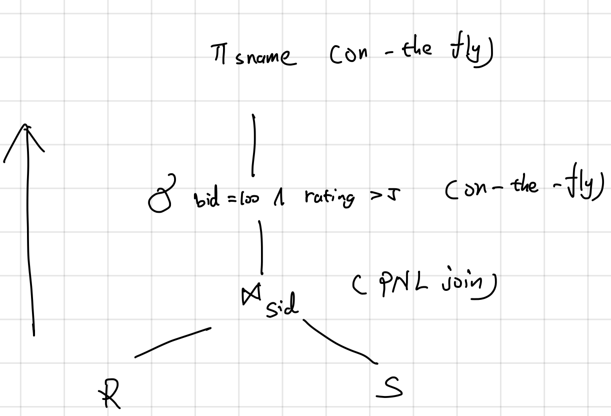
* Access Path: Different ways in which rows of a table can be retrieved

* Query vs Relational Algebra

select	π (Projection)
where	σ
join	⋈

* Plan:

on the fly: as you are doing the previous step, do this too



* table Scan

↳ if the target data page is a very tiny table, just perform a table scan

* 选择 index 最好选择最能 narrow down entry 的

例: SELET Cout(*)
From Sailors
where rating = 10 AND
age > 30

有 10 个 ratings 和 100 ages

⇒ Rating vs Age 肯定选 rating 因为假如每个rating 都有一个 age / 每个 age 都有一个 rating 的话，总数是 10 * 100 = 1000

if rating: 找到 rating = 10，接下来只需 query 100 个 DE

if age: 找剩下 70个 age，需 query 70 * 10 = 700 个 DE

* tree index better for range query

* hash index better for equality Search

* Calculations for an Alt 2. B+ Tree Index

4K byte/page, (22+10) bytes / DE, there are 100,000 tuples

∴ We can fit 4K/32 = 128 DEs / page

Since we have 100,000 records, we need $\left\lceil \frac{100,000}{128} \right\rceil$: 782 leaf page

Assume For 782 leaf pages, we have x parent nodes

$\therefore\ x \cdot \underbrace{(128+1)\text{ pages / node}}_{\text{因为能指向128个key，但 fanout = 128+1}} = 782$

$x = \left\lceil \frac{782}{129} \right\rceil = 7$ parent nodes

And 1 root

∴
1 root
7
782 leaf

* Calculation of cost:

Clustered Index

* 首先算出 index page需要多少个: root + index + RF * leaf pages

再算出需要 access 多少 data pages: RF · # Data pages → 因为 clustered index 中, physical order 和 logical order 保持一致, 那么我们可以假设符合条件的 tuples 总是在 data page 上连在一起, 那么我们只需访问这些 table: RF · #Data page 即可

Unclustered index

首先算出 index page需要多少个: root + index + RF * # leaf pages

再算出一个 data page 需 access 多少 tuples: RF * # tuples / Data pages

↳ 乘以 # data pages

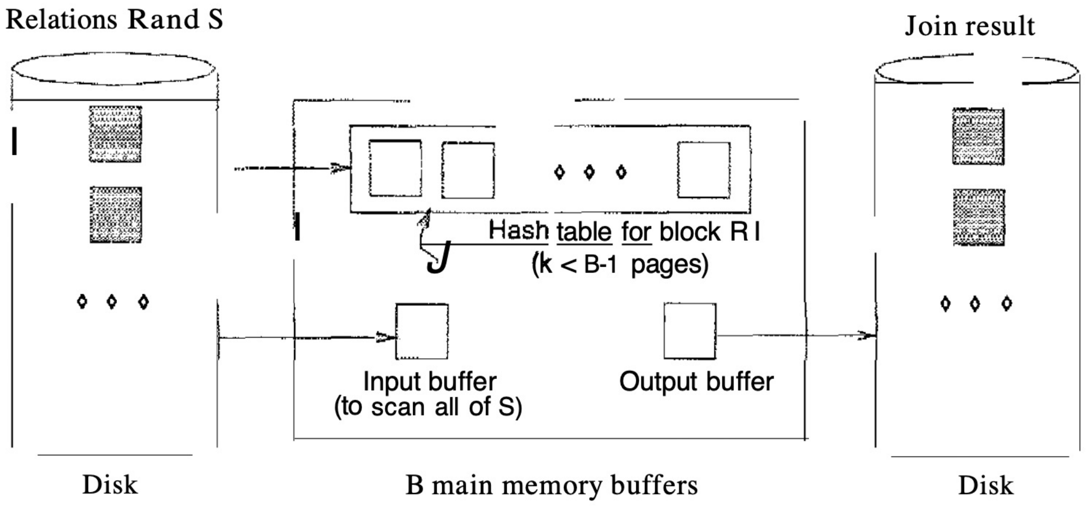
⇒ 对于 un clustered index 中, 我们的记录有可能散落在各处, 平均每个 data page 我们需访问 RF * # tuples / page 并需访问所有的 data page

假设R有m pages，S有n pages，我们现在想join这两个relation

- Page Nested Loop(PNL) Join (当buffer page == 3，这时候我们只能一个bp给R，一个给S，一个给output，只能一个一个读取)
  - 他的cost就相当于m(读取每一个R的pages)+m*n(对于读取的每一个R的page，我都扫过S中所有的pages)
- Block Nested Loop(BNL) Join (当buffer page >= 3的时候，我们可以一个bp给S，一个给output，剩下B-2个全部给R)
  - 他的cost就是m(读取每一个R的pages)+ceil($\frac{m}{B-2}$)*n(每一次我们都读取$\frac{m}{B-2}$个R pages，然后每一次扫过全部的S)
- smaller table as outer table

* 可以用 in memory hash table 来代替 match

```
for each $\frac{m}{B-2}$ pages from R:
    foreach page from S:
        for all matching in-memory tuples, add
```

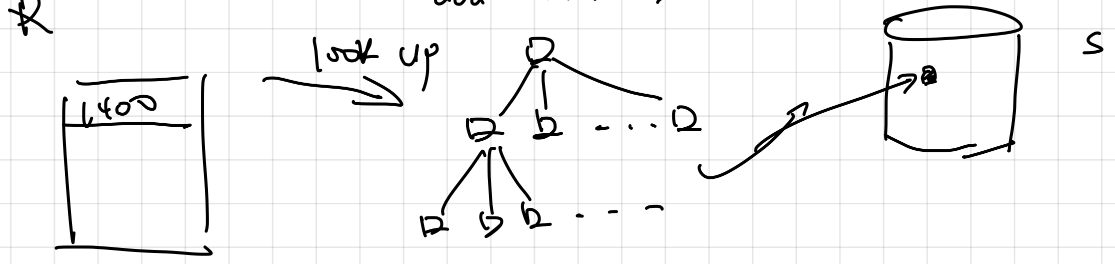


以上用的是 1个BP for output, 1个BP for inner, B-2个BP for outer, 我们也可以把 BP-1 (除去 output) 个 page evenly splited between input & output

Index Nested Join

* Use index to retrieve inner table

```
For each pages from R:
        foreach page from S that satisfied condition based on index:
            add <r, s>
```



Cost:

1.1 Assume B+ tree index, the cost to find the leaf is $\boxed{3}$ I/Os.

1.2 Assume hach index, the cost to find the bucket is 1.2 I/os.

2. Assume we already found the bucket:

if clustered: one more I/os per outer entry

if unclustered: one I/o per matching inner

* Probe : 从index找到 matching tubles.

Cost: $M + (M \text{ pages}) \cdot (x \text{ tuples/pages}) * (\# \text{look up} + \underset{\substack{\uparrow \\ \text{if we need} \\ \text{to fetch tuple}}}{1})$

Cost of Sort Merge Join:

Assume R: 1000 pages, S: 500 pages, both sort in two passe

Sort: $\underbrace{1500}_{R} * \underbrace{2}_{r/w} * \underbrace{2}_{2 \text{ passes}} + \underbrace{500}_{S} * 2 * 2 = 6000$

Join: Additional scan for both tables

1000 + 500 = 1500

→ 7500

Program Preparation in DB2 for z/OS

The following diagrams are from: (a) Fig 2.2 in C.J. Date & Colin J. White's *A Guide to DB2* (3rd edition), (b) Figs. 13.1 and 14.4 in Craig S. Mullin's *DB2 Developer's Guide* (5th edition). Program preparation is required before any DB2 application code can run (e.g., Java, C, C++, COBOL programs). The end results are an executable load module and a DB2 program plan, both of which run together to access a DB2 database.

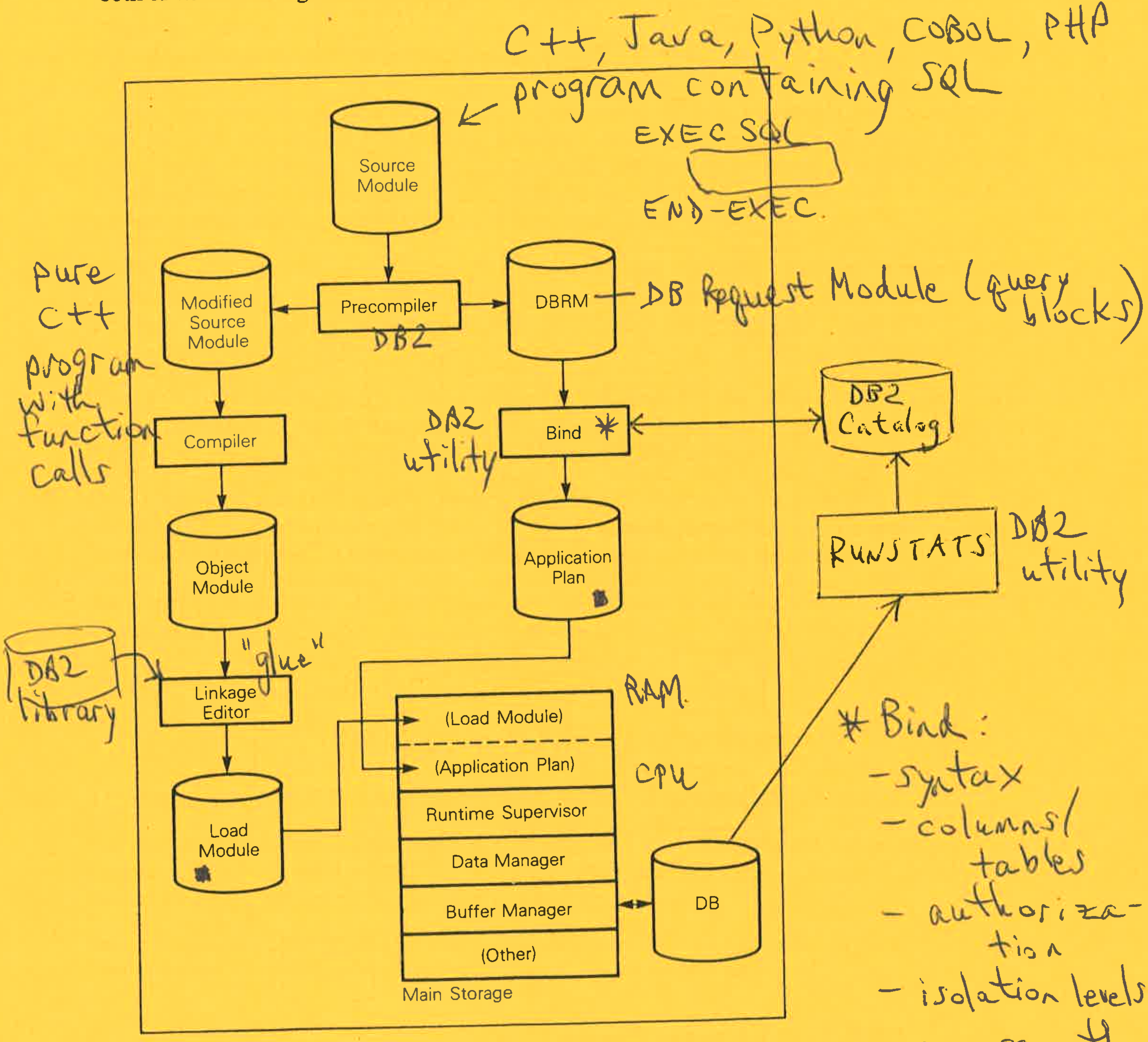


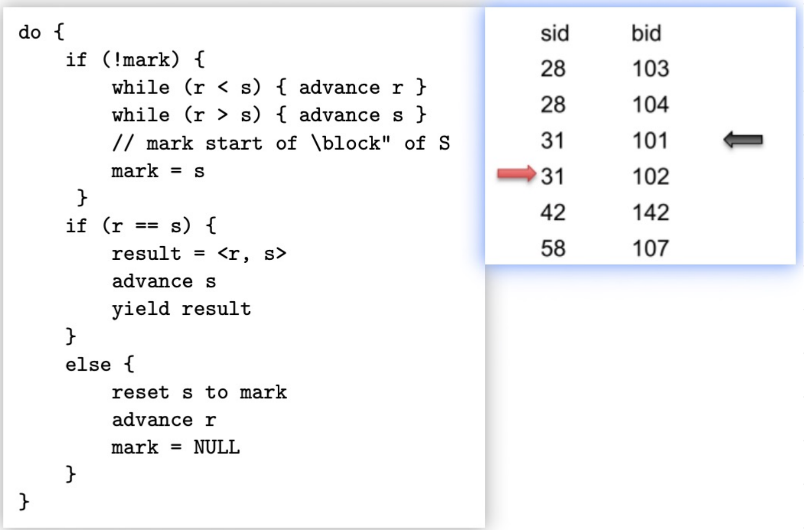
Fig. 2.2 DB2 application program preparation and execution (overview)

* Bind:
- syntax
- columns/tables
- authorization
- isolation levels
- access path & query plan

π, σ
sort

index?
join?
order?

Sort Merge Join Algorithm



```
do {
    if (!mark) {
        while (r < s) { advance r }
        while (r > s) { advance s }
        // mark start of \block" of S
        mark = s
     }
    if (r == s) {
        result = <r, s>
        advance s
        yield result
    }
    else {
        reset s to mark
        advance r
        mark = NULL
    }
}
```

sid	bid
28	103
28	104
31	101
31	102
42	142
58	107

query block :

SELECT attribute list
FROM relation list
WHERE $\underbrace{\text{term1}}$ And ... term k

↓

Reduction Factor : fractions of tuples that Satisfies a conjunct

↳ Reflects the impact of the term in reducing the result size

* Push Selection ahead of join

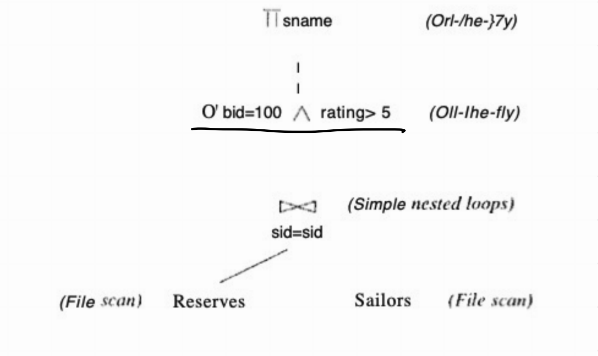


Figure 12.4 Query Evaluation Plan for Sample Query

=>

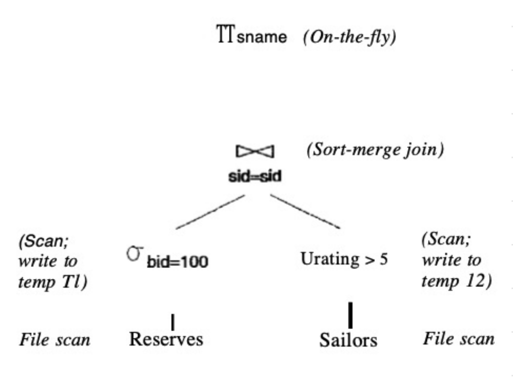
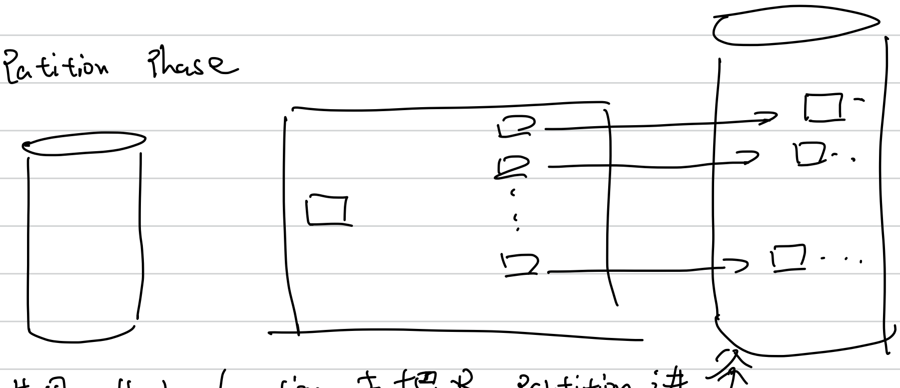


Figure 12.6 A Second Query Evaluation Plan

* join Cost too much, 但如果先 select ,再join会好很多

Hash Join

1. Partition Phase



① 先用 Hash function 去把 R partition 进

② 再用 Hash function 去把 S Partition 进

2. Join Phase

```
For i=1 to B-1 do:
    (a) Take partition Ri's pages and hash it again

    (b) For each page in S:
            read the page
            for each tuple on the page:
                apply h2 to see if there is a join
```

Clustered:

cost = expected # of internal pages to get to the leaf level
+ expected # of qualifying leaf page
+ expected # of data pages

2021. 11.25

-> Transactions without commit statement : automatic commit.
↘ crashes -> roll back
↓
To use multiple commits

-> Tx should be short

-> Consistent means that the DBMS will guarantee all constraints.

-> Roll back = Abort

-> Read must go before write

* Strict Two Phase locking (Strict 2PL)

↳ Each transaction obtain an S (shared) lock on the DB object before reading, and an X (exclusive) lock before writing it

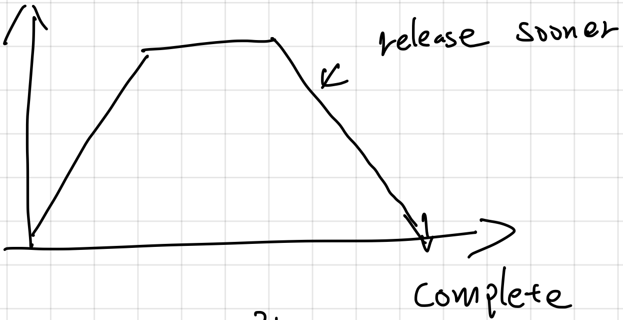
S -> Before Read

X -> Before Write

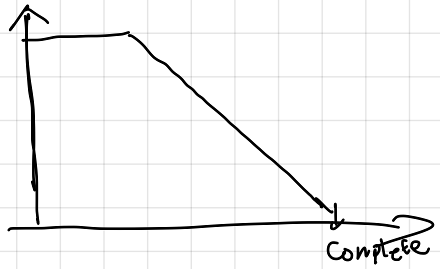
All locks released after completion of the tx

* lock Based Concurrency control

Strict 2PL

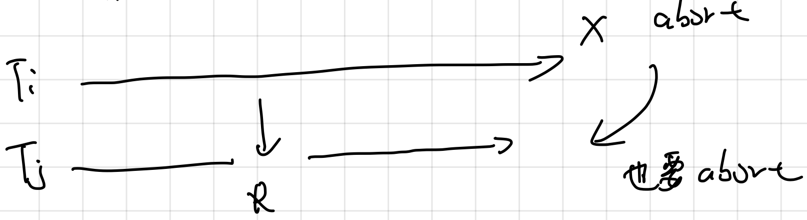


2PL



Conservative 2PL
↓
requires all lock from start

* Domino Effect



* Read → share lock

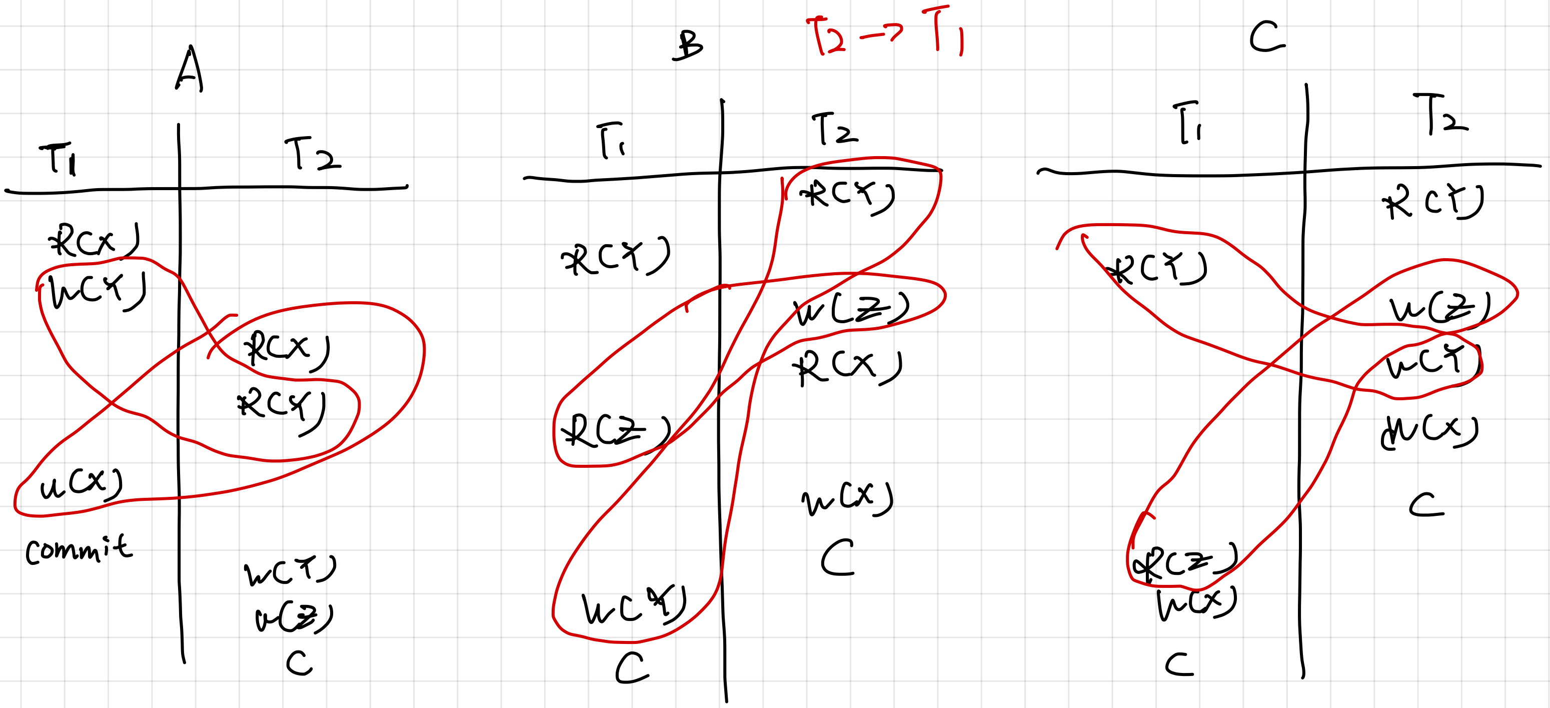
write → exclusive lock

* 通过不停地 swap, 可把 all conflict equivalent schedules 换成 serial

* Conflict serializable → serializable

serializable → conflict serializable

* 判断 conflict serializability, 好像只要画上 xx 的, 基本都不行, 因无论怎样交换, 都做不到 serializab



A

T1	T2
R(X)	
W(X)	
	R(X)
	R(Y)
W(X)	
commit	
	W(Y)
	W(Z)
	C

B T2 → T1

T1	T2
	R(Y)
R(Y)	
	W(Z)
	R(X)
R(Z)	
	W(X)
	C
W(Y)	
C	

C

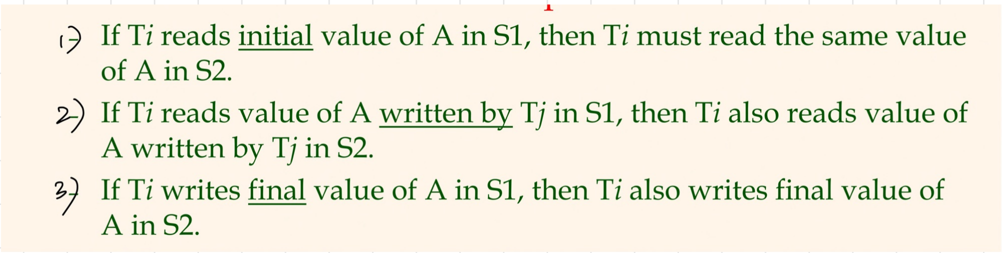
T1	T2
	R(Y)
R(Y)	
	W(Z)
	W(Y)
	W(X)
	C
R(Z)	
W(X)	
C	

1. Precedence Graph

1. Node : T (Transaction)
2. edge : $T_i \rightarrow T_j$ iff an action of T_i precedes and conflicts with T_j

2. View serializability

* Conflict serializable could prevent a legal transactions from going ahead.



1) If T*i* reads initial value of A in S1, then T*i* must read the same value of A in S2.
2) If T*i* reads value of A written by T*j* in S1, then T*i* also reads value of A written by T*j* in S2.
3) If T*i* writes final value of A in S1, then T*i* also writes final value of A in S2.

} V-S => S

view serializable

⇓

serializale

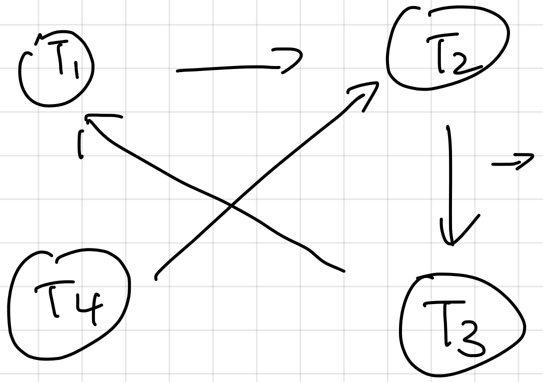
2PL : Strict Two-Phase Locking Protocol

Strict 2PL only allows conflict serializable and recoverable schedules.

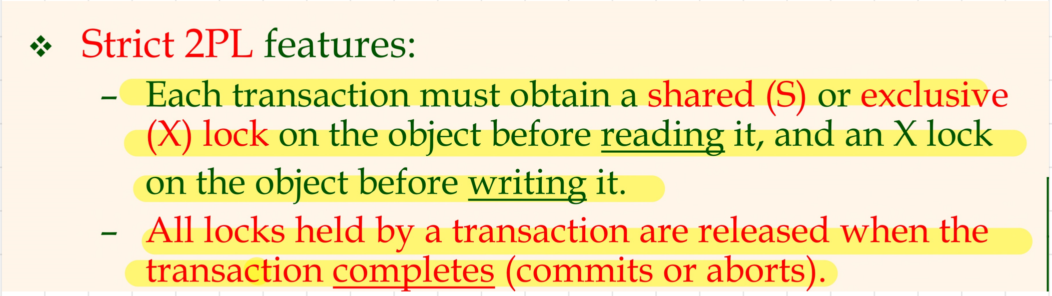
* Read → share lock (release of commit time)

write → exclusive lock

* Wait for graph:

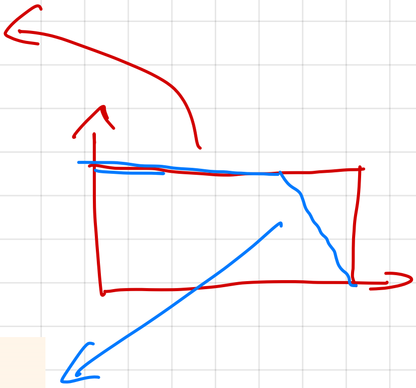


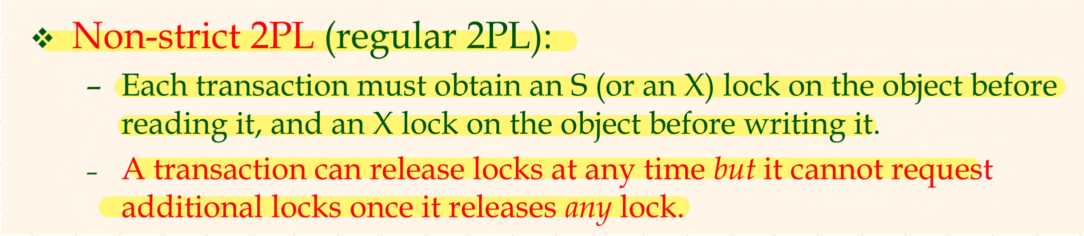
Arc from T_2 to T_3 means that T_2 is waiting for T_3's lock



❖ Strict 2PL features:

- Each transaction must obtain a shared (S) or exclusive (X) lock on the object before reading it, and an X lock on the object before writing it.
- All locks held by a transaction are released when the transaction completes (commits or aborts).





❖ Non-strict 2PL (regular 2PL):

- Each transaction must obtain an S (or an X) lock on the object before reading it, and an X lock on the object before writing it.
- A transaction can release locks at any time *but* it cannot request additional locks once it releases *any* lock.

Lock Management

X(A) → request A

Cascading abort: situation in which the abort of one tx forces the abort of another tx.

if safe (不用 insert) : release parent
else : not release parent

* Index locking →
* Predicate locking

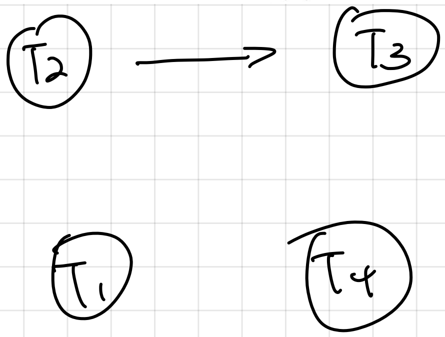
Re uncommitted : No locks

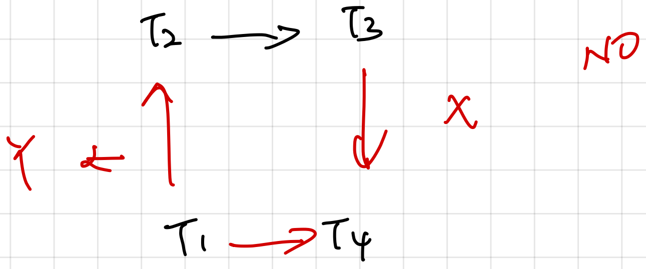
9. Consider the following ordering S of transactions:

```
T2:R(X); T3:W(X); T3:commit; T1: W(Y); T1:commit; T2:R(Y);
T2:W(Z); T2:commit; T4:R(X); T4:R(Y); T4:commit
```

a) Draw a precedence graph for S.
b) Is S conflict-serializable? Why or why not?
c) Is S serializable? If so, give all possible serial orderings.
d) Is S recoverable? Justify your answer.

(a)





(b) No, Not C-S, there contains no cycle at the graph

(c) T1 -> T2 -> T3 -> T4

(d) Since transaction only commit after all txs that they read from committed.

LSN : Log sequence number , LSNs are always increasing

Write Ahead log :

* Each data page contains a <u>PageLSN</u>

↳ The LSN of the most recent log record for a change done to that page

* flushed LSN : maximum LSN flushed

WAL protocol
- * Force the update log record to disk before the data page gets to disk
- * Send the log record out to disk before transaction commits

Log records :
- Update
- Commit
- Abort
- End
- CLR (Compensation log record)
  - ↳ created before undoing a previous update action
  - ↳ take place during an abort or recovery
  - ↳ contains LSN of next log to be undone

* Transaction Table (TT)
- ↳ TxID
- ↳ Last LSN ↗ change to u upon crash recovery
- ↳ Status (in-progress, committed, aborted)

* Dirty Page Table
- ↳ one entry for each dp in the bp
  - ↳ contains [recLSN, : the LSN of the log record which first caused the page to be dirty]
- ↳ Entry is removed when the page is written to disk

* checkpoint : periodic snapshot of the state of a DBMS, used to reduce recovery time.

begin_ checkpoint : record
end- checkpoint : record contains TT & DPT

↳ store the LSN of the checkpoint rec in a safe place

No attempt is made to force dirty pages to disk
↳ STEAL NO FORCE

At commit time

* Write the commit record to the log
* All log records up to the tx's lastLSN are flushed
    ↳ guarantees that flushedLSN ≥ lastLSN
    ↳ flushes → sequential, sychronous.

ARIES Algorithm :

① CRASH

1. go back to last checkpoint
2. store analysis (Rebuild TT & DPT)
3. Find Smallest recLSN in DPT after analysis
    ↳ earliest of the DPT
4. ↓ redo all step from that point until the crash
5. undoing all the uncommitted txs.

Analysis → Redo → Undone

CPSC 404:
More Practice Exam Questions on Concurrency and Crash Recovery

Last Update: December 12, 2016

1. Consider the following sequence of actions using the ARIES protocol:

LSN	**LOG**
00	begin checkpoint
10	end checkpoint
20	update: T1 writes P61
30	update: T2 writes P72
40	update: T3 writes P95
50	T1 commit
60	T1 end
70	T2 commit
80	update: T3 writes P77
90	T2 end
100	update: T4 writes P72
	<CRASH, RESTART>

   a) For this sequence of log records, what is done during Analysis after the crash occurs? For each LSN above (in order), explain how the Transaction Table and the Dirty Page Table are rebuilt.
   b) What is done during Redo? Indicate where Redo starts, and explain how each LSN is handled during Redo.
   c) For this sequence of log records, describe how the Undo phase proceeds, and show any new log records that result. At the end, take a new checkpoint.

2. Consider the following ordering S of transactions from the previous set of sample exam questions. This time, however, we want to convert S via a series of allowable swaps by the scheduler into a serial schedule S'. Draw a table ordering the actions of the transactions. Then, draw the final schedule S'. Explain how the swaps take place so that schedule S' is conflict equivalent to S.

```
T2:R(X); T3:W(X); T3:commit; T1: W(Y); T1:commit; T2:R(Y);
T2:W(Z); T2:commit; T4:R(X); T4:R(Y); T4:commit
```

3. This question deals with **multiple-granularity locking** (MGL). Suppose there are locks at the table level, the page level, and the row level. If the object that we want to lock is at the lowest level of the hierarchy, then we'll take warning locks (intention locks) at the higher levels. If we want to lock the whole relation explicitly, then we can take a shared or exclusive lock at the table level.

LSN	LOG
00	begin checkpoint
10	end checkpoint
~~20~~	~~update: T1 writes P61~~ ✓
~~30~~	~~update: T2 writes P72~~ ✓ ✓
40	update: T3 writes P95
~~50~~	~~T1 commit~~ ?
60	T1 end
~~70~~	~~T2 commit~~
80	update: T3 writes P77
90	T2 end
100	update: T4 writes P72 ✓
	<CRASH, RESTART>

undo T3, T4

100 ↓ 30 ↓ ∅

80 ↓ 40 ↓ ∅

1. Analysis

TT

tx	lastLSN	Status
~~T1~~	~~20~~	~~u~~
~~T2~~	~~30~~	~~u~~
T3	~~40~~80	u
T4	100	u

DPT

Page	recLSN
P61	20
P72	30
P95	40
P77	80

(2) Redo from smallest recLSN in DPT

(3) Undo all things that are still in progress.

T₁ :				w(Y)	C						
T₂ :	R(x)					R(Y)	w(Z)	C			
T₃ :		w(x)	C								
T₄ :									R(X)	R(Y)	C

For convenience, use the following **compatibility matrix** for making your decisions. (It's taken from the textbook. It's fixed and doesn't change; but, it saves you from memorizing it. I'll provide it on an exam if I ask such a question.)

	--	IS	IX	S	X
--	√	√	√	√	√
IS	√	√	√	√	
IX	√	√	√		
S	√	√		√	
X	√				

a) **Transaction 1 (T1)**: a single SQL statement. For the following SQL statement, describe the sequence of MGL locks you would take on the table in question, assuming you want to lock at each level of the hierarchy. Assume that we have no index on the table.

```
SELECT     sid, sname, age, rating
FROM       Sailors;
```

b) How would your answer to (a) change if we had an (Alt. 2) index on the `sid` field? (Let us ignore the part about locking the index, and deal with only the table.)

c) What is a potential problem with Part (a)? Is there a way we could have a more efficient locking strategy while still using MGL? Explain.

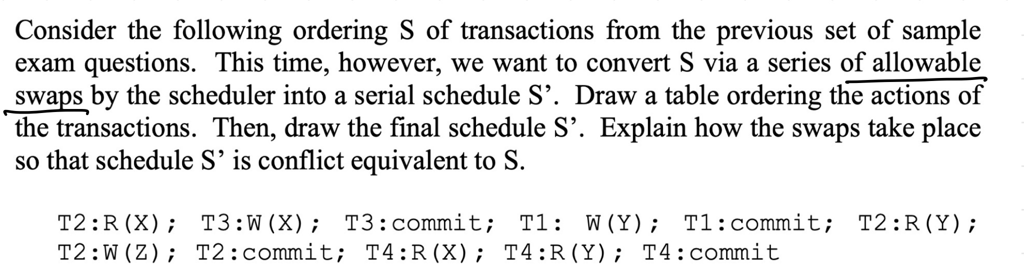
d) **Transaction 2 (T2)**: a single SQL statement. Suppose that *while T1 is executing*, we issue the following SQL statement. Again, describe the sequence of MGL locks you would take on the table in question, assuming you want to lock at each level of the hierarchy. Assume that there is no index on the table.

```
UPDATE     Sailors
SET        rating = 7
WHERE      sid = 1050;
```

e) How would your answer to (d) change if we had an (Alt. 2) index on the `sid` field? (Let us ignore the part about index locking, and deal with only the table.) Assume that we are using an index to find the *rid* for the row for Sailor #1050. Again, assume T1 is in progress. Let us assume that only T1 is running.

Final 可能题型 终极汇总

Module 4:



Consider the following ordering S of transactions from the previous set of sample exam questions. This time, however, we want to convert S via a series of allowable swaps by the scheduler into a serial schedule S'. Draw a table ordering the actions of the transactions. Then, draw the final schedule S'. Explain how the swaps take place so that schedule S' is conflict equivalent to S.

```
T2:R(X); T3:W(X); T3:commit; T1: W(Y); T1:commit; T2:R(Y);
T2:W(Z); T2:commit; T4:R(X); T4:R(Y); T4:commit
```

☆ 只有 pair of unconflicting action can be swapped

☆ Commit 跟着它上面的一起换

☆ 每次在换的时候应注意 → 先把一个东西换到最前，先不需要在意 在这一步换完之后会不会 introduce 新的 conflict

☆ 作表

S

T1	T2	T3	T4
	R(X)		
		w(X)	
		C	
w(Y)			
C			
	R(Y)		
	W(Z)		
	C		
			R(X)
			R(Y)
			C

→

S'

T1	T2	T3	T4
w(Y)			
C			
	R(x)		
	R(Y)		
	W(Z)		
	C		
		w(x)	
		C	
			R(X)
			R(Y)
			C

Consider the following sequence of actions using the ARIES protocol:

LSN	**LOG**
00	begin checkpoint
10	end checkpoint
20	update: T1 writes P61
30	update: T2 writes P72
40	update: T3 writes P95
50	T1 commit
60	T1 end
70	T2 commit
80	update: T3 writes P77
90	T2 end
100	update: T4 writes P72
	<CRASH, RESTART>

a) For this sequence of log records, what is done during Analysis after the crash occurs? For each LSN above (in order), explain how the Transaction Table and the Dirty Page Table are rebuilt.

b) What is done during Redo? Indicate where Redo starts, and explain how each LSN is handled during Redo.

c) For this sequence of log records, describe how the Undo phase proceeds, and show any new log records that result. At the end, take a new checkpoint.

(a) 所需知识点：

Analysis：重点是在维护 TT (transaction table) 和 DPT (Dirty Page Table)

TT :

Tx ID	lastLSN	Status

DPT :

PageID	recLSN

基本上，我们可以把这个过程认为是 ascendingly 扫过 LSN

```
For every Entry:
    if Entry == End:
        remove Tx from TT
        ⇨ 只有end了的，才能被认为是"结束"了的，在之后的 undo 中需要被
          undo
    else if Entry == Update:
        if PageID not in DPT:        ↙重点是 recLSN记录的是这个page最
            Add PageID to DPT          早被遗漏的 LSN.
            Set recLSN to LSN
        if Tx ID not in TT:
            Add Tx ID to TT
            Add LSN to Last LSN
        else :
            Replace stored LSN to current LSN in TT
    else :
        // For commit or abort
        Change the tx's status to ("C" or "A")
```

Consider the following sequence of actions using the ARIES protocol:

LSN	LOG
00	begin checkpoint
10	end checkpoint
20	update: T1 writes P61
30	update: T2 writes P72
40	update: T3 writes P95
50	T1 commit
60	T1 end
70	T2 commit
80	update: T3 writes P77
90	T2 end
100	update: T4 writes P72
	<CRASH, RESTART>

TT

TxID	LSN	Status
~~T1~~	~~20~~	~~uc~~
~~T2~~	~~30~~	~~uc~~
T3	~~40~~ 80	u
T4	100	

DPT

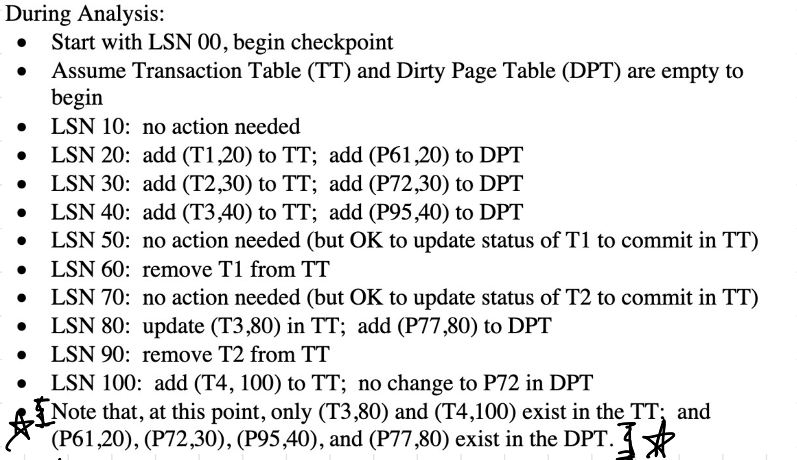
PageID	recLSN
P61	20
P72	30
P95	40
P77	80

a) For this sequence of log records, what is done during Analysis after the crash occurs? For each LSN above (in order), explain how the Transaction Table and the Dirty Page Table are rebuilt.

b) What is done during Redo? Indicate where Redo starts, and explain how each LSN is handled during Redo.

c) For this sequence of log records, describe how the Undo phase proceeds, and show any new log records that result. At the end, take a new checkpoint.

(a) 这种问题基本上照着 principle 走一遍就OK了, 问题是写的东西太多了

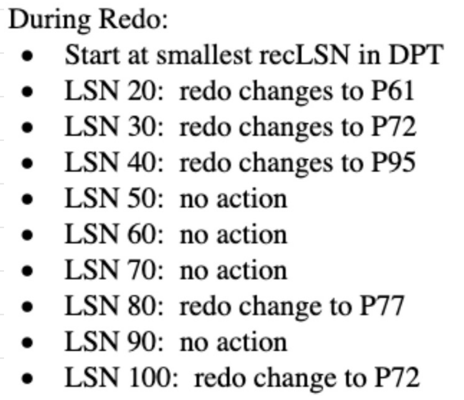


During Analysis:

- Start with LSN 00, begin checkpoint
- Assume Transaction Table (TT) and Dirty Page Table (DPT) are empty to begin
- LSN 10: no action needed
- LSN 20: add (T1,20) to TT; add (P61,20) to DPT
- LSN 30: add (T2,30) to TT; add (P72,30) to DPT
- LSN 40: add (T3,40) to TT; add (P95,40) to DPT
- LSN 50: no action needed (but OK to update status of T1 to commit in TT)
- LSN 60: remove T1 from TT
- LSN 70: no action needed (but OK to update status of T2 to commit in TT)
- LSN 80: update (T3,80) in TT; add (P77,80) to DPT
- LSN 90: remove T2 from TT
- LSN 100: add (T4, 100) to TT; no change to P72 in DPT
- ★重 Note that, at this point, only (T3,80) and (T4,100) exist in the TT; and (P61,20), (P72,30), (P95,40), and (P77,80) exist in the DPT. 重★

(b) Redo 的写法

↳ 其实就是把 (checkpoint, crash) 中间的 LSN 都 "Redo" changes, 除了 C/A.



During Redo:

- Start at smallest recLSN in DPT
- LSN 20: redo changes to P61
- LSN 30: redo changes to P72
- LSN 40: redo changes to P95
- LSN 50: no action
- LSN 60: no action
- LSN 70: no action
- LSN 80: redo change to P77
- LSN 90: no action
- LSN 100: redo change to P72

(c) Undo

Queue!

↓

* 重点 是要把 CLR 记录到 log

→ 从 TT 中的 latest LSN 开始找 → 把要undo的，加入到 loser set → { }

找到 last LSN → 找到 last LSN 改了什么 page

然后：

1. undo changes: For LSN id, undo changes to Page
2. 很重要的 : add CLR to log: Undo Tx, LSN id,
3. 判断 Tx 还有没有更早的 LSN，若有 → next UndoLSN = 那个LSN
4. 把 nextUndo LSN 加入 loser set

↓

找下一个 loser set 的东西

9. Consider the following ordering S of transactions:

```
T2:R(X); T3:W(X); T3:commit; T1: W(Y); T1:commit; T2:R(Y);
T2:W(Z); T2:commit; T4:R(X); T4:R(Y); T4:commit
```

a) Draw a precedence graph for S.
b) Is S conflict-serializable? Why or why not?
c) Is S serializable? If so, give all possible serial orderings.
d) Is S recoverable? Justify your answer.

所需知识：

Precedence graph:

* node = 一个 tx
* edge from T_i to T_j iff an action of T_i precedes and conflicts with T_j

↳ 一个action A conflict with action B iff:

1. A, B operate on the same object
2. At least one operation is W

* Conflict-Serializable:

对做题 no help 的概念： S is conflict serializable if S is conflict equivalent to any serial schedule.

↳ conflict equivalent (S_1, S_2)

1. S_1 and S_2 involve the same actions of the same transactions
2. Every pair of conflicting actions is ordered the same way

对做题有帮助： 一个 schedule 的 precedence graph 有 cycle 的话，就不是 C-S

9. Consider the following ordering S of transactions:

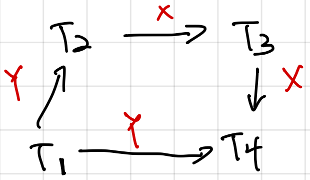
```
T2:R(X); T3:W(X); T3:commit; T1: W(Y); T1:commit; T2:R(Y);
T2:W(Z); T2:commit; T4:R(X); T4:R(Y); T4:commit
```

a) Draw a precedence graph for S.
b) Is S conflict-serializable? Why or why not?
c) Is S serializable? If so, give all possible serial orderings.
d) Is S recoverable? Justify your answer.

Serializable schedule: 当一个 schedule的结果 as if all tx runs in serial

recoverable: commit only after all transactions whose changes they read commit

(a)



(b) no cycle → C-S

(c) Yes

(d) Yes

Time	T1	T2	T3
0			
1			
2			start
3			READ `tax`
4			`tax = tax +` 100
5	start		
6	READ `salary`		
7	`salary = salary +` 500		
8			WRITE `tax`
9			commit
10			end
11		start	
12		READ `tax`	
13		READ `salary`	
14		`tax = tax + salary` * 0.01	
15		WRITE `tax`	
16		commit	
17		end	
18	---------------	*Begin Checkpoint*	--------------------
19	READ `tax`		
20	`tax = tax +` 1000		
21	WRITE `salary`		
22	-----------------	*End Checkpoint*	--------------------
23	commit		
24	end		

The following questions refer to the table of transactions above. For each question, be sure to justify your answer. Let us assume that a transaction can upgrade a lock from S to X unless another transaction currently holds a lock on it.

a) Would the above schedule be permitted by Strict 2PL?
b) Would the above schedule be permitted by non-strict 2PL?
c) Is this schedule *recoverable*?
d) Does this schedule *avoid cascading aborts*?

所需知识：

Strict two-phase: a transaction does not release its locks until the transaction has either committed or aborted.

Two-phase: Locks can be released before commit time providing the transaction doesn't ask for further locks.

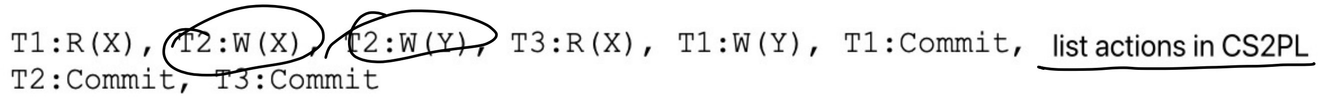
Conservative Two-phase: C2PL's transactions obtain all the locks they need before the transactions begin, if they can't obtain, they wait

Avoid Cascading Aborts: A schedule avoids cascading aborts if each transaction only reads values written by *committed* transactions.

* Lock:

Share Lock block Exclude Lock; doesn't block share lock

Exclude Lock block share Lock and Exclude Lock



```
T1:R(X), T2:W(X), T2:W(Y), T3:R(X), T1:W(Y), T1:Commit,
T2:Commit, T3:Commit
```

list actions in CS2PL

T1欲获X之读锁，Y之写锁，尚无人占，T1获之

T2欲获X之写锁，但写锁，而T1占之，T2等(wait)

T3欲获X之读锁，即T1已占X之读锁，缘是读锁本对另一读锁无阻拦之意，故T3仍获之。

T1完，释X之读锁，Y之写锁

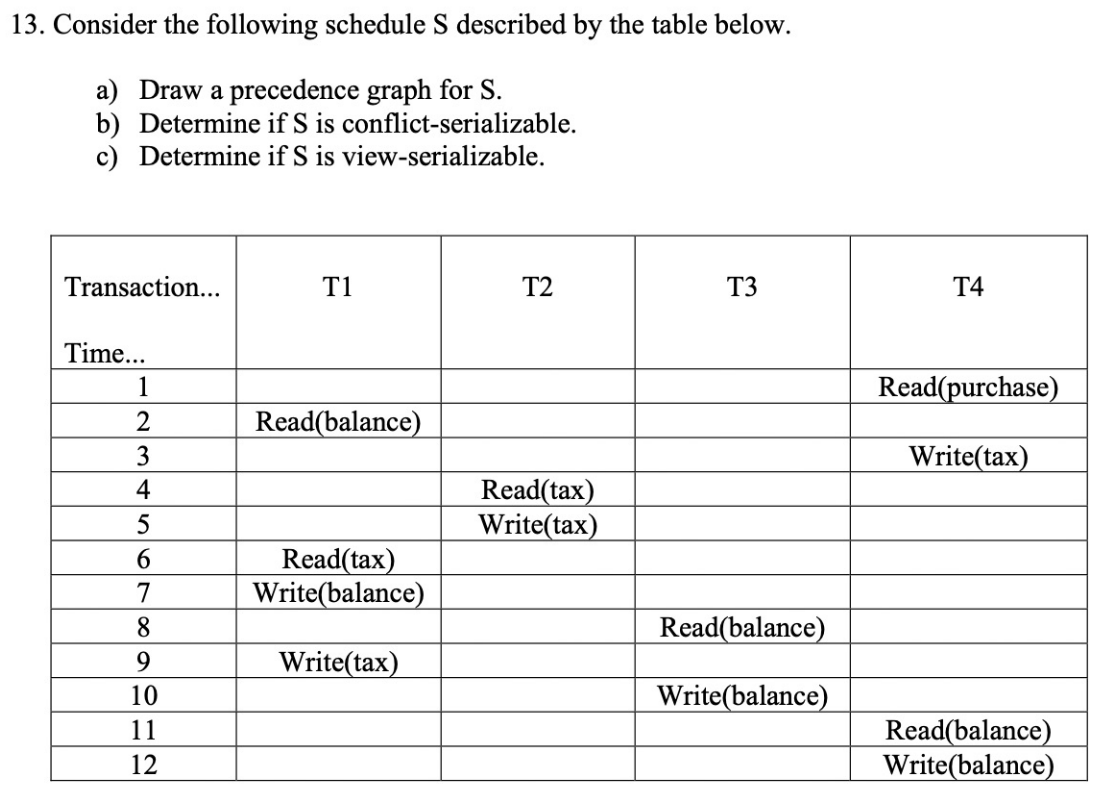
T2欲获X之写锁，但写锁，而T3占之，T2等(wait)

T3完，释X之读锁

T2欲获X之写锁，但写锁，而T1占之，T2等(wait)

T2完，释X之读锁，Y之写锁

此类题，应令于原S之首action，于事锁之事，为最优先者(T1 in this case)



13. Consider the following schedule S described by the table below.

a) Draw a precedence graph for S.
b) Determine if S is conflict-serializable.
c) Determine if S is view-serializable.

Transaction... Time...	T1	T2	T3	T4
1				Read(purchase)
2	Read(balance)			
3				Write(tax)
4		Read(tax)		
5		Write(tax)		
6	Read(tax)			
7	Write(balance)			
8			Read(balance)	
9	Write(tax)			
10			Write(balance)	
11				Read(balance)
12				Write(balance)

所需知识：

用以下的定义，看能不能找来一个 serial order

Schedules S1 and S2 are view equivalent if:
– If Ti reads initial value of A in S1, then Ti must read the same value of A in S2.
– If Ti reads value of A written by Tj in S1, then Ti also reads value of A written by Tj in S2.
– If Ti writes final value of A in S1, then Ti also writes final value of

* side: view serializable schedule and not conflict serializable | must have blind write, no read befor write

Module 3:

Cost Model:

此问题为全教材最深，最繁琐之问题，用定义 + 例子来记录

假设：现在要 join R (m pages), S (n pages)

没有index:
- Block Nested Loop Join
  - Page Nested Loop Join (B=3)
  - Block Nested Loop Join (B>3)
- Hash Join
- Sort Merge Join

有index: Index Nested Join
- Clustered Index Join
- Unclustered Index Join

Block Nested Loop Join:

配置: R: m pages, S: n pages, $m > n$, B buffer pages

* 简单，暴搜

* Cost = $m + \left\lceil \frac{m}{B-2} \right\rceil * n$

↳ 除了一个给 output，一个给 n、剩下的都是给 m 的

Sort Merge Join:

配置: R: m pages, S: n pages, $m > n$, B buffer pages

Cost: Sort + merge

$$\left.\begin{aligned} &= m \cdot 2 \cdot (\text{用 B sort m 的 \#passes}) \\ &+ n \cdot 2 \, (\text{用 B sort n 的 \#passes}) \end{aligned}\right\} \text{Sort}$$

$$\left. + \; m + n \right\} \text{merge}$$

Hash Join :

配置: R: m pages, S: n pages, $m > n$, B buffer pages

算法过程:

Partition phase:

读m, 写: 把 m partition 成 B-1个 partions, 每个 Partion m 有

$\left\lceil \frac{m}{B-\textcircled{1}} \right\rceil$ → 1个作 input 个 page

读n, 写: 把 n partition 成 B-1个 partions, 每个 Partion n 有

$\left\lceil \frac{n}{B-\textcircled{1}} \right\rceil$ → 1个作 input 个 page

确保 [Partion m < B-2 || Partion n < B-2]

如果不, 再 partion 一遍

Join phase:

读m: 把 min(Partion m, Partion n) load 进来

重新 用 新的 hash 去 partition → 分成 $\frac{\text{Partion}_m}{B-\textcircled{2}}$ 个 partions

1 for input, 1 for output

读n: 读入 Partion n (在第一个 phase 对应的那个)

应用 h_2(Partion n), 找 match, 写出

Cost: $\#\text{ Partion phases} \cdot (m+n) \underbrace{\cdot 2}_{R/W} + \underbrace{(m+n)}_{\text{Join phases}}$

Index Nested Loop join (on the fly)

配置: R: m pages, S: n pages, m > n, B buffer pages

假设目前要先 select R, 再 join S

Cost = (probe - 1) + (# of index pages in R to read)
+ (# of data pages in R to read)
+ (# of data pages in S)

probe-1: 对于 B+ tree index 和 hash index 各不同

For B+ tree: 即它的 height (x-level, x-1), 所以 probe 为 x-level 的数
一般 assume probe 为 3

For hash index: 差不多, 都是从 root 到 bucket, 一般 assume 为 1.2

# of index pages to read in R:

RF * Total Number of leaf pages

RF: 根据要 query 的来画来 (要 select 的 / 总)

# of data pages to read in R:

clustered: RF * Total Number of data pages in R

unclustered: $RF * \left(\frac{\#\text{Tuples}}{\text{data pages}}\right) * \#\text{ data pages}$

# of data pages to read is S:

[RF * Total Number of tuples in R] → # qualifying tuples in R

⇓

$RF * \text{Total Number of tuples in } R * (\underset{\substack{\downarrow \\ \text{找到 index}}}{\text{probe}} + \underset{\substack{\downarrow \\ \text{从 index 到 data}}}{1})$

注意, 以上所有的前提, 能在① 用 index 先选 R 和 能在② 用 index 直选 S, 当且仅当 它们有 index, 否则 只能用 table scan.

Calculate # of page at each level of B+ tree

1. 先算出一个 page 可以放入多少个 Data Entries → # DEs/pg

2、算出需要多少 leaf pages → # leaf

3、$\left\lceil \frac{(\# \text{ leaf})}{(\# \text{ DEs/pg} + 1)} \right\rceil$ = level -1 (倒数第一层的 nodes)

⋮

$$\left\lceil \frac{\text{level-1}}{(\text{DEs/pg} + 1)} \right\rceil = \text{level-2}$$ (倒数第二层的 nodes)

直到算出 1 → root node

Module 2

B+ Tree

Minimum 50% occupancy：每个子节点，entry的数量 $d \leq m \leq 2d$，d是 order
基本上就是我这个须占一半

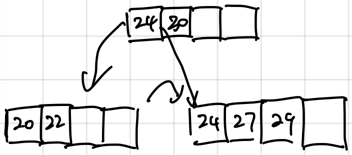
```
# Insert (data, L)
  1. Find correct Leaf L
  2. Put data entry onto L
       if L has enough space, put!
       else:
           split (L)
              [split时，多的跟右边] (保留的 middle，即于 leaf 情况中)

  split (L):
      if L is leaf page:
          1. find middle
          2. split L into L1 and L2 delimilited by
               the middle, L1 = L[ : middle], L2 = L[middle: ]
          3. copy up the middle
               ↳ insert (middle, parent)
      else if L is internal page:
          1. find middle
          2. split L into L1 and L2 delimilited by
               the middle, L1 = L[ : middle], L2 = L[middle+1: ]
          3. move up the middle
               ↳ insert (middle, parent)
      else if L is root page;
          1. find middle
          2. split L into L1 and L2 delimilited by
               the middle, L1 = L[ : middle], L2 = L[middle+1: ]
          3. New root with one key: middle
          4. height ++;
      else:
          //
```

Redistribution

1. between leaf nodes

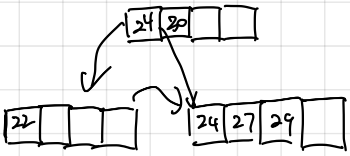
↓



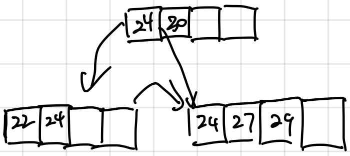
目前要删的是 20，但删完之后，minimum occupancy (2) 就无法满足

1. 检查是否满足 redistribute 的要求 (redisburite) (满足)

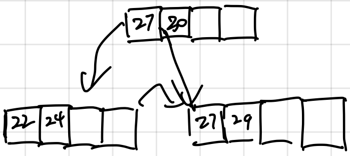
2、删除 20*



3、Move 24* to ⤴



4、把 27* copy up，replace 24



2、between non leaf node

为什么要换 both 17，20 去隔壁 → 只换一个 20 也可以！

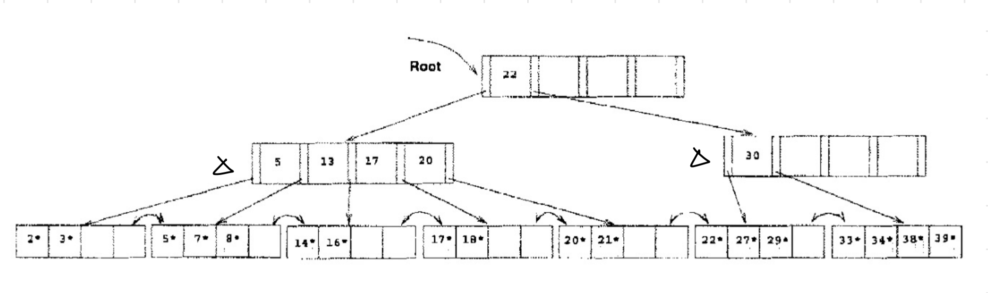
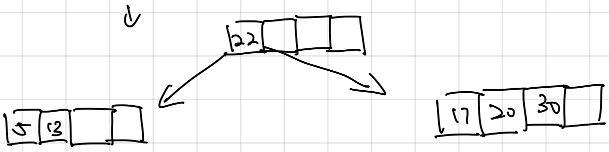


Figure 10.19 A B+ Tree during a Deletion

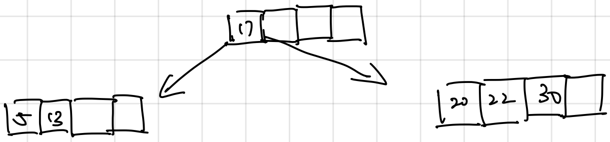
目前要 redistribute 的是 △ 的两个 node

1. 把 17，20 移动到右边的 node

↓

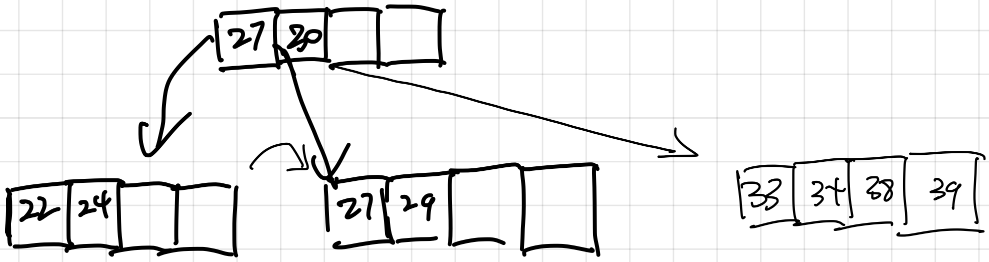


2、Replace 22 with 17



Merge

1. between leaf node

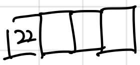
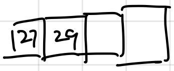


目前要删 24

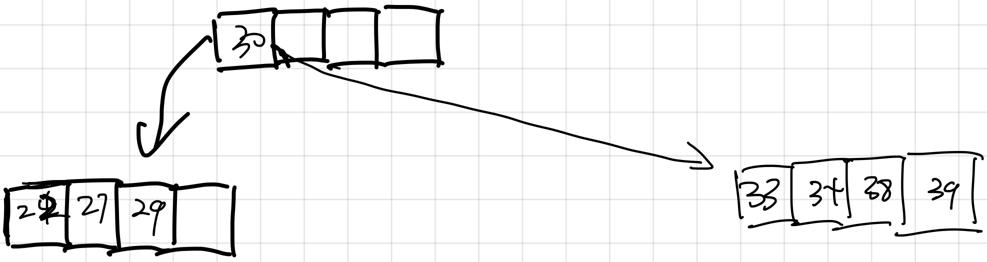
1. 检查能否 redistribute (不能)

2. 检查能否 merge (能)

↳ 1. 删除 24

2. Merge  和 

3. 把 27 这个 ptr 中 上面删掉 (这里忽略这个 non leaf page 违反了 minimum occupancy 的要求)



2. between non leaf node

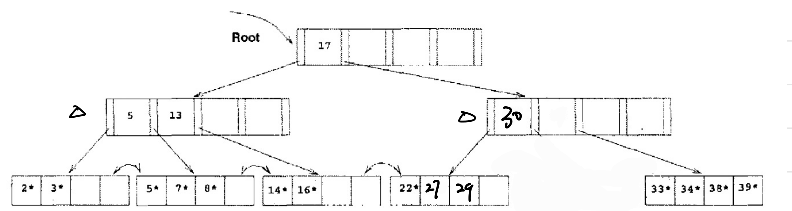


Figure 10.16 B+ Tree after Deleting Entries 19* and 20*

现在要merge ▷ 的两个 node

1. 把 17 pull down

2. merge

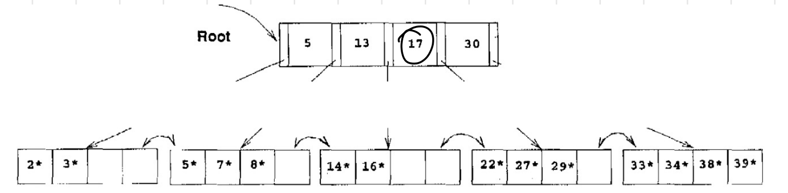


Figure 10.18 B+ Tree after Deleting Entry 24*

至此为止, 完成了本页一开始的删除 24 的要求

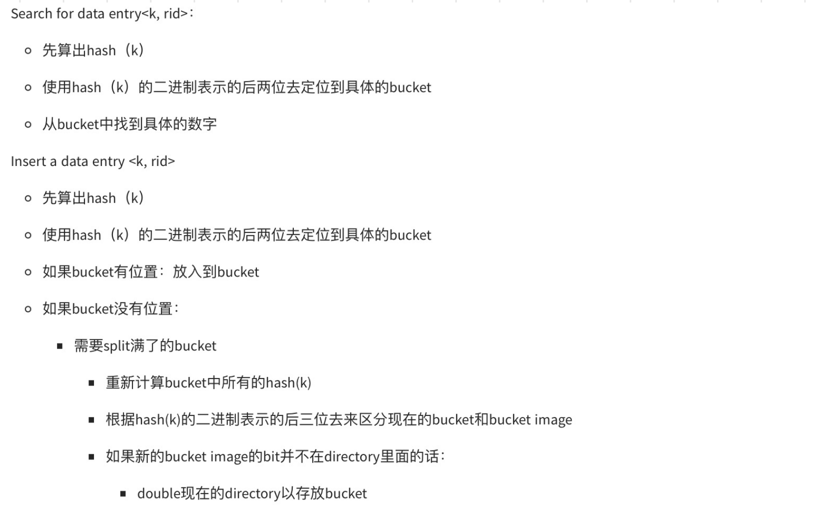
Hashing

Static Hashing

- Insert：
  - apply hash function to data entry : $hash(data)$
  - use $hash(data)$去找到对应的bucket
  - 如果bucket没满：放入data
  - 如果bucket满了：
    - 在这个bucket后面allocate一个新的overflow page
    - 把data放在page上
    - 把这个page加入这个bucket的overflow chain上

- Delete：
  - apply hash function to data entry : $hash(data)$
  - use $hash(data)$去找到对应的bucket
  - 如果data不是在overflow chain中的一个overflow page上或者在overflow chain中的一个overflow page但不是这个page的最后一个：直接删掉
  - 如果data在overflow chain中的一个overflow page但是这个page的最后一个：
    - data删掉
    - 把page从这个bucket的overflow chain中移除，放入free page

Extendible Hashing

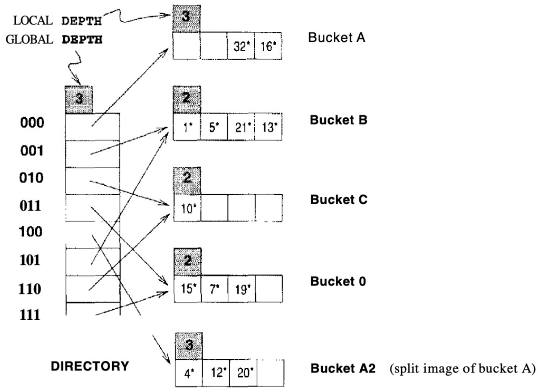


Search for data entry<k, rid>：

- 先算出hash（k）
- 使用hash（k）的二进制表示的后两位去定位到具体的bucket
- 从bucket中找到具体的数字

Insert a data entry <k, rid>

- 先算出hash（k）
- 使用hash（k）的二进制表示的后两位去定位到具体的bucket
- 如果bucket有位置：放入到bucket
- 如果bucket没有位置：
  - 需要split满了的bucket
    - 重新计算bucket中所有的hash(k)
    - 根据hash(k)的二进制表示的后三位去来区分现在的bucket和bucket image
    - 如果新的bucket image的bit并不在directory里面的话：
      - double现在的directory以存放bucket

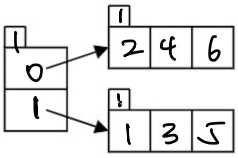


Global depth 和 local depth 如何成长?

只要有一个 split：

Global depth +=1
(需要多一位去定位)
local depth of that bucket
++

5. {6 marks} Suppose we have an Extendible Hash index structure shown below. You can assume that each bucket can hold up to 3 entries. Use the same hash function from class (i.e., modulo last *n* bits of each binary value). All of the nubs are 1 at the start; the first bucket holds keys 2, 4, and 6; the second bucket holds keys 1, 3, and 5.

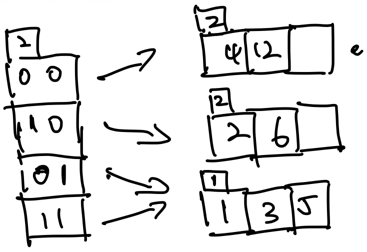


2: 00 10
4: 01 00
6: 01 10
12: 1100

1: 0001
3: 0011
5: 0101

a) Add this key: 12—and show the index structure after it is successfully inserted:

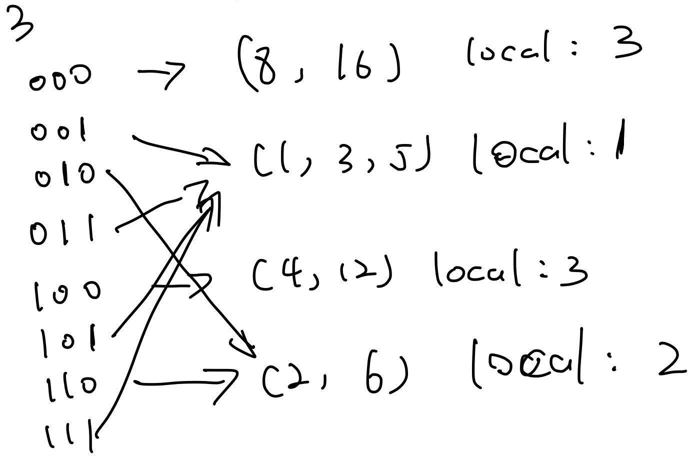
11100
10100



100
000

b) Let us build on your answer to (a). Provide an example of the minimum number of unique keys that need to be added to (a)'s result so that the directory is capable of holding 8 arrows (because the global depth nub is 3—note that $2^3 = 8$). Draw the resulting structure with the new keys that you plan to insert into the result of (a). Use the space below to draw the complete final index structure.

Add 8, 16



Linear Hashing

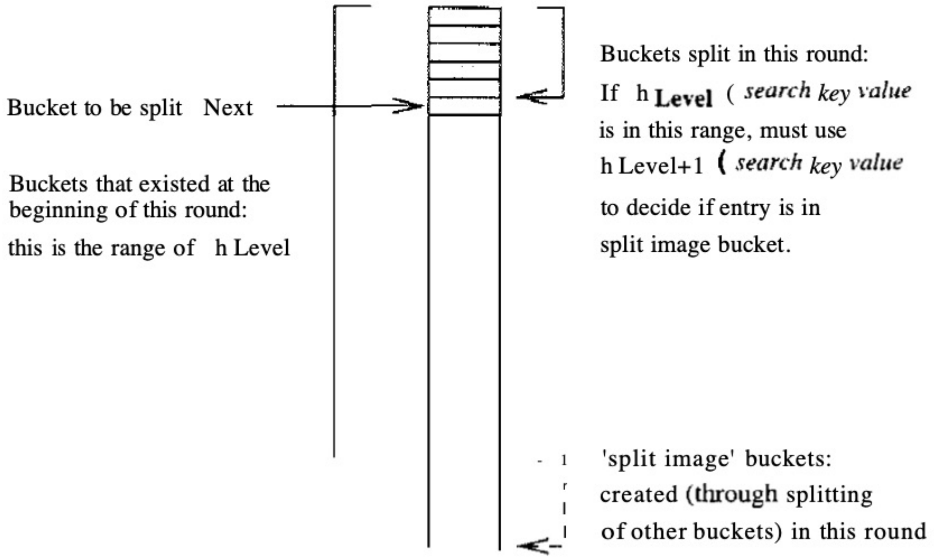


Figure 11.7 Buckets during a Round in Linear Hashing

- For our examples, a split is 'triggered' when inserting a new data entry causes the creation of an overflow page.
- Split：
  - use hash function $h_{level+1}$ to redistribute entry between current and split image
  - assign bucket number b +N_{level} to the split image(b是现在的bucket)
- Insert <k, rid>：
  - 根据$h_{level}(k)$找到对应的bucket，看看有没有被split ✓
  - 被split了，用$h_{level+1}$找到这个k归属于现在的bucket还是split image，找到对应的位置，看看满了没满（包括之前存在的overflow page）
  - 没满：直接加入
  - 满了：
    - split current！要注意的是没，这个insert触发了一次split，但这个split并没有发生在overflow的地方，而发生在目前next所指向的地方，round robin轮到的位置
    - 把现在的这个k加入这个bucket的overflow page
    - next++
- 只有split了之后才会increment next

External Merge Sort

分别计算 每一轮的 External Merge Sort 产生的 Run 及 分别的 Page Size

Sort X 个 Pages，有 B 个 buffer pages

Sort: 创造 $\left\lceil \frac{X}{B} \right\rceil = S_1$ SRs，每个 SR 有 x_1 个 page

Merge 1: 创造 $\left\lceil \frac{S_1 \text{ SRs}}{B-1} \right\rceil = S_2$ SRs …… 每个SR有 $x_1 \cdot (B-1)$ 个 page

until output only 1 SRs

一个 BP 放一个 SR

cylindrification

phase 1 sort:

先算出一个 BP fill 需要的时间

Transfor time : # pages (number of pages in memory) · $\overbrace{x \text{ ms}}^{\text{average transfor time}}$/pages

\+ rotational delay : 0

\+ Long Seek : 10 ms

\+ Short Seek : (# cylinders (number of cyls in memory) − 1) · Short seek time

再算出 BP fill 的 次数 : # fill = $\frac{\text{file size}}{\text{\# page in main memory}}$

再乘在一起 : $\underbrace{2}_{\text{R \& W}} \cdot$ BP time · # fill

phase 2:

算出 # cylinder needed for file

再算出 time for 1 cylinder

Transfor time : # pages (number of pages in cylinder) · $\overbrace{x \text{ ms}}^{\text{average transfor time}}$/pages

\+ rotational delay : 0

\+ Average Seek : 10 ms (data-dependent 了，现在不同的数据都散落在不同的 SR，每次用的时候都 assume worst)

再算总时间：

$\underset{\text{R \& W}}{2} \cdot$ # cylinder · time/cylinder

Sort phase time for 1 BP: 1个 long seek，Cyl − 1 个 SS

* Cylindrification

在 phase 2 中，改变 input buffer 和 output buffer
对总的时间的影响

总时间： $2 \cdot \# \text{cylinder} \cdot \text{time}/\text{cylinder}$

$\underset{R \cdot W}{\downarrow}$

$R: \; \# \text{cylinder} \cdot \text{time}/\text{cyl}$

$W: \; \# \text{cylinder} \cdot \text{time}/\text{cyl}$

注意，增加 # cylinder output buffer 会减少 writing 的时间
∵之前是 merge 好了之后，从 disk 上 seek 到位置 (long seek) 再写一个 cyl，所以如果 output buffer 有 1 cyl，那么每个写入的 cyl 都要经历一次 LS，但如果 output buffer 上升到 2 cyl 的话，现在每两个 cyl 做一次 LS，快很多！ → $W = \frac{\# \text{Cylinder}}{\# \text{cyl in output buffer}} \cdot \text{time}/\text{cyl}$

11. {10 marks} This question is about external mergesort. We have an unsorted input file that fills 600 contiguous cylinders. The buffer pool is 20 cylinders in size. Our disk geometry specs state that we have 10 tracks per cylinder, so that means the buffer pool holds 200 tracks. (Don't worry about the number of individual pages.)

Let us assume that the arm/head on the disk drive is currently at some unpredictable location. There are multiple users of the disk drive, but once you start a read or write operation, you do so uninterrupted until the number of pages of your request is completed, even if you need to access multiple tracks or cylinders in the same disk request. Furthermore, you always read and write contiguously.

A <u>short seek</u> (SS) is to an adjacent (neighbouring) cylinder *only*. *All other seeks* are called <u>long seeks</u> (LS), even if you move just two cylinders over. Do not include time for rotation or transfer.

There are 2 questions for you to answer, and you do <u>not</u> have to calculate any milliseconds.

a) Compute the number of long seeks (LS) and short seeks (SS) that it takes to **READ** the unsorted file into memory during the *first* phase or pass (i.e., during the sort phase), <u>and</u> the number of LS and SS operations that it takes to **WRITE** out the sorted runs from this phase. **Show your work.** (Do *not* compute the merge step on this page; we'll do that in part (b) on the next page.)

(a) 在 Sort phase 中，我们需要把 data 全跑一遍，所以第一步，求 Sort phase，需要的 write 和 Read 时，有两部分需要注意

(1) 一个 BP fill 需要的 LS 和 SS：

SS: seek in between 2 adjacent cyls

∴算一个 BP 有多少个 cyls，那么 SS 就是 #cyls −1

LS: 1 LS (非 adjacent cyls 的 seek)

(2) 需要多少 BP fills：

有 X 个 page 要 sort，一个 BP 有 y 个 page，

需 x/y 次 BP fill

注意要仔细看准 BP fill 的次数，若 BP 中只有 2 个 cyls，且只有 ~1.5 次 BP fills，我们只走了一次完整的 BP fill 和 .5 次 BP fill，第二次没有跨 cyl → 不需 SS.

b) We still have 20 cylinders (i.e., 200 tracks) of space available in the buffer pool for the merge phase, and we're going to break down the input buffers and output buffer, as follows. Use 5 tracks (0.5 cylinders) for every input buffer, and the rest for the output buffer. Compute the number of long seeks (LS) and short seeks (SS) that it takes to do the merge phase (both reading and writing) to produce the final sorted file. **Show your work.**

i) First, provide a sketch of the merge phase (like in class), labelling the input buffers and their sizes, and the output buffer and its size.

ii) Then, compute the number of LS and SS seek operations. Provide a total for the merge part (i.e., Part (b)), but you don't have to add in your calculations from Part (a).

类似这样

已知 30 way merge

∵ 0.5 cyls per input buffer

$\therefore$ Total: $30 \cdot 0.5 = \underbrace{15}_{\text{cyls}}$ for input

5 cyls for output

对于 Merge 来说，我们实打实地需要走完整个 file

$$\therefore \frac{\text{File size}}{\text{input buffer size}}$$

$$= \frac{600 \text{ cyls}}{0.5 \text{ cyls/bp}} = 1200 \text{ input buffer fill}$$

↓

1200 ls (每个 input buffer 只有 .5 cyl, 无 ss)

For write:

5个 cyl for output

$$\therefore \frac{600}{5} = 120 \text{ write}$$

For each: 1 LS + $\overset{(5-1)}{4}$ SS

$$\therefore 120 \cdot (1 + 4)$$

Seek time + Rotational Delay + Transfer time

↓

从当前的 cyl 到目标 cyl : 1ms + 1ms for 500 cyls moved

Rotational Delay:

转几圈 : 例 16,000 rpm =

$$16{,}000 \cdot \frac{1\text{min}}{60\text{s}} = 266.67 \text{ cyl/sec}$$

↓

倒数

$$\frac{1}{266.67} \approx 3.75 \text{ ms}$$

↓

一圈

Note: trasfer time for a page or anything

1. calculate # sector needed for the page or anything

2. Calculate the portion of one track that needed to store a page

↳ 1 page = n sectors

we need $\frac{n \text{ sectors}}{m \text{ sec/track}} = x$ tracks

↳ a fraction, hopefully

3. Use x to multiply the rotational delay ~~for~~ one spin → x · spin ✓

Note: 当题目给出: moving time from C1 to C2 is 1ms + 1ms per n cylinders moved

那 average seek time: 1ms + $\frac{1}{3} \cdot \left(\frac{\text{\# Total moved track}}{n}\right)$

↓

maximum seek time

Note: Average rotational latency: $\frac{1}{2}$ spin

↓

就是转一圈的时间

- Clock: If a page is referenced often enough, its reference bit (RB) will stay set, and it won't be a victim.
    - if an empty frame in BP:
        - Use it to store the new page's data
        - Set the RB to 1
        - Set the timestamp to current time
    - else:
        - Find the oldest page(page with the oldest timestamp)
        - If that page's RB is set to 0, then:
            - This is the victim page, replace it with the new page
            - Set the new page's RB to 1
            - Set timestamp to the current time.
        - Else:
            - Decrement that page's RB to 0
            - Update that page's timestamp to the current time
- Extended Clock
    - if an empty frame in BP:
        - Use it to store the new page's data
        - Set the RB to 1， DB to 1(?)
        - Set the timestamp to current time
    - else:
        - Find the oldest page(page with the oldest timestamp)
        - If that page's RB is set to 0/0 or 0/0* then:
            - This is the victim page, replace it with the new page
            - Set the new page's RB to 1, DB to 1(?)
            - Set timestamp to the current time.
        - Else:

```
switch (RB/DB){
  case (0/1):
      set to 0/0*;
  case (1/0 || 1/0*):
      set to 0/0 || 0/0*;
  case (1/1):
      set to 0/1;
}
```