# 16. Buffer and Indexing

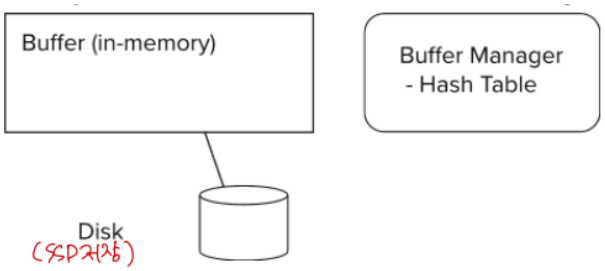
## ▼ Storage Access

- block → storage allocation과 data transfer의 단위
- database system → disk와 memory 사이의 block transfer 횟수를 줄임 ⇒최소화

→ main memory 내부에서 가능한 block 수만큼 저장해놓음

→ disk access 횟수 줄일 수 있음 (main memory에 있으니까~!!)

→ buffer에 그 block들의 복사본 저장! + buffer manger가 memory 관리

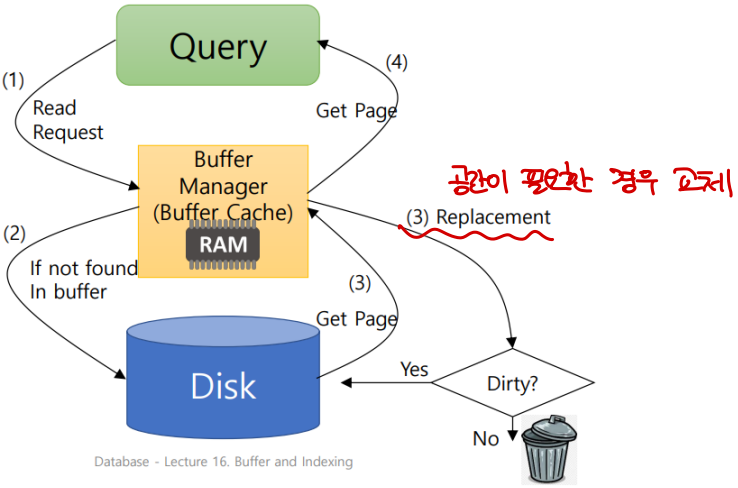



## ▼ Buffer Manager

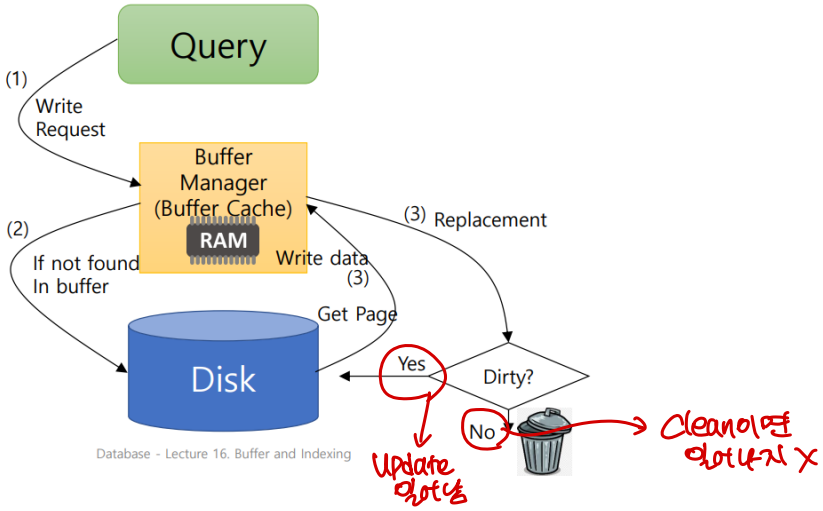
disk로부터 block 필요로 할 때 program은 buffer manager 호출

- block이 buffer 내에 이미 존재
  - buffer manger : main memory 내부 block의 주소를 return
- block이 buffer 내에 존재 x
  - 공간 할당
    - 공간이 필요한 경우 → 새 block을 위해 다른 block 교체(폐기)
    - 수정된 경우 → disk에 다시 기록
  - buffer에 disk로부터 block 읽어온 뒤 main memory 내부 주소 return

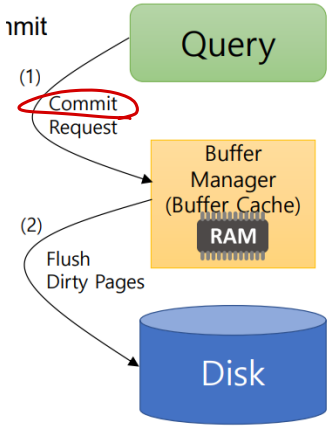
- operation
  - read




  - write




  - transaction commit



## ▼ Buffer-Replacement Policy

어떤 기준으로 page 삭제할 지에 대한 정책 필요

① • **LRU strategy**(least recently used)
  - 대부분의 OS가 사용
  - 안 쓸 것 같은 놈을 우선적으로 replace

    → query에는 bad → 모든 tuple 다 뒤져야 할 수도 있음

    → Query : sequential scan과 같은 잘 정의된 pattern으로 disk 접근

② • Toss-immediate strategy
  - 바로바로 block 삭제

③ • Most-recently used(MRU) strategy
  - block이 처리되면 교체될 block 선택

⇒ file sytem : wirte reorder 가능(buffer를 통해서 rw할 경우 disk로 바로 x)

⇒ 다양한 문제들에 대비해야 함! (corruption 발생 가능)

## ▼ Pinned Block ⇒ disk에 write 허가받지 못한 Block

- Pin: 지금 r/w 할 block → pin list에 입장 ⇒ 누군가 쓰고 있다면 바로 쓰지 X
- Unpin: 이미 r/w가 끝난 block → 더 이상 내가 사용하지 않음을 알림

⇒ 여러 개의 pin/unpin 동시에 가능

- pin count 계속 저장 및 update
  - 0 → block eviction 가능
  - ≥ 1 → 아직 pin list에 block 존재(pin list에 wait 중인 놈 존재)

## ▼ Shared and Exclusive locks on buffer → critical section 만들어줌.

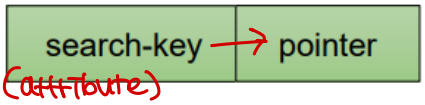
- 하나의 page에는 여러 transaction이 concurrent하게 접근 가능함

  → 그 접근 코드는 다 critical section이어야 함
- Reader → shared lock(read lock)
- Writer → exclusive lock(write lock)
- Locking Rules
  - exclusive lock : 한 번에 오직 하나의 process만 접근 가능
  - shared lock
    - exclusive lock과 concurrent하게 접근 불가
    - 동시에 여러 process 접근 가능

# ▼ Indexing

data 접근 빨리 함

- search key : record 찾을 때 사용되는 attribute set (idx building 시에 기준이되는 key)
  ⇒ 찾을 수 있도록 충분한 정보가 있어야 함.
- index file(index entry)




  - index building 시에 기준이 되는 key → pointer로 구성되어 있음
- index : orginal table보다 훨씬 작음
  - ordered indices : key가 정렬되어서 저장
  - hash indicies : search key가 hash function을 사용 ⇒ "bucket"
- index evaluation metrics ⇒ index에서 우리 query문을 실행 (해결) ?
  1. accross type
     - point query (key = value) : 값
     - range query (low < key < high) : 범위 내
  2. access time : 접근시간
  3. insertion time : 데이터 추가하는데 걸린 시간
  4. deletion time : 데이터 삭제하는데 걸린 시간
  5. space overhead : 너무 많은 메모리가 제한되어 있기 때문에 자제해야 함

## ▼ Ordered indicies

다 search key value로 정렬이 되어서 저장되는 애들

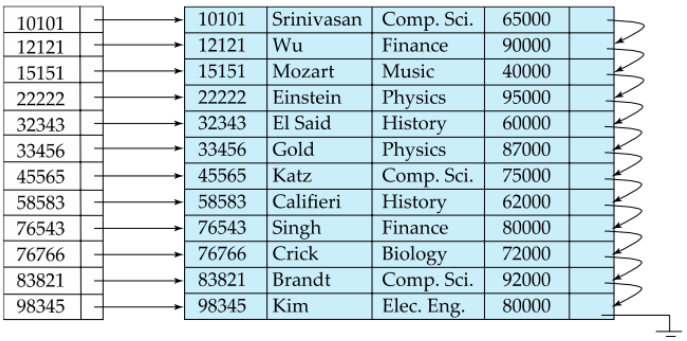
- clusterd index(primary index) : file 내에서도 정렬됨
- secondary index(nonclustered index) : file과 다른 순서로 정렬됨
- indexed-sequential file(ISAM) : index 자체로 file 관리

## ▼ Dense index Files ⇒ 일대일

- Dense index : index record가 file 내의 모든 search-key value에 의해 일대일 대응

- example
  - relation : instructor
  - index : ID attribute
    → search key

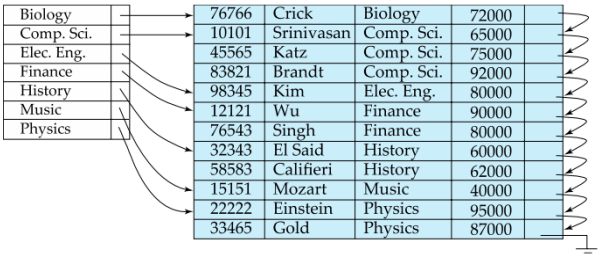
search key → pointer



| | | | | |
|---|---|---|---|---|
| 10101 | Srinivasan | Comp. Sci. | 65000 | |
| 12121 | Wu | Finance | 90000 | |
| 15151 | Mozart | Music | 40000 | |
| 22222 | Einstein | Physics | 95000 | |
| 32343 | El Said | History | 60000 | |
| 33456 | Gold | Physics | 87000 | |
| 45565 | Katz | Comp. Sci. | 75000 | |
| 58583 | Califieri | History | 62000 | |
| 76543 | Singh | Finance | 80000 | |
| 76766 | Crick | Biology | 72000 | |
| 83821 | Brandt | Comp. Sci. | 92000 | |
| 98345 | Kim | Elec. Eng. | 80000 | |

- index : dept_name

search key → pointer



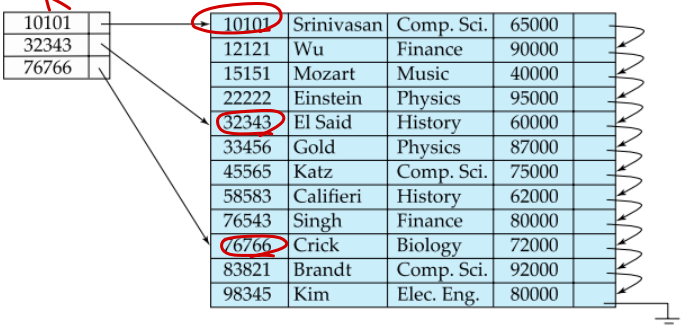
| | | | | |
|---|---|---|---|---|
| Biology | 76766 | Crick | Biology | 72000 |
| Comp. Sci. | 10101 | Srinivasan | Comp. Sci. | 65000 |
| Elec. Eng. | 45565 | Katz | Comp. Sci. | 75000 |
| Finance | 83821 | Brandt | Comp. Sci. | 92000 |
| History | 98345 | Kim | Elec. Eng. | 80000 |
| Music | 12121 | Wu | Finance | 90000 |
| Physics | 76543 | Singh | Finance | 80000 |
| | 32343 | El Said | History | 60000 |
| | 58583 | Califieri | History | 62000 |
| | 15151 | Mozart | Music | 40000 |
| | 22222 | Einstein | Physics | 95000 |
| | 33465 | Gold | Physics | 87000 |

## ▼ Sparse Index Files

오직 search-key value로만 sequential하게 정렬됨

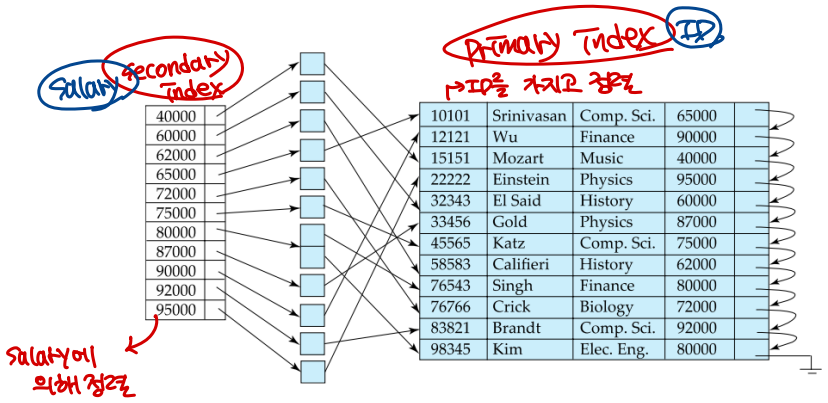
- search-key value K
  - index record point

    → 가장 앞에 있는 놈을 넣어 놓음



| | | | | |
|---|---|---|---|---|
| 10101 | 10101 | Srinivasan | Comp. Sci. | 65000 |
| 32343 | 12121 | Wu | Finance | 90000 |
| 76766 | 15151 | Mozart | Music | 40000 |
| | 22222 | Einstein | Physics | 95000 |
| | 32343 | El Said | History | 60000 |
| | 33456 | Gold | Physics | 87000 |
| | 45565 | Katz | Comp. Sci. | 75000 |
| | 58583 | Califieri | History | 62000 |
| | 76543 | Singh | Finance | 80000 |
| | 76766 | Crick | Biology | 72000 |
| | 83821 | Brandt | Comp. Sci. | 92000 |
| | 98345 | Kim | Elec. Eng. | 80000 |

## ▼ Secondary indicies (ordered index 중 하나) ⇒ dense index여야 함 (일대일 대응)




## ▼ Multilevel index

- 만약 index가 memory에 fit하지 않는다면 접근하는 것이 비쌀거임 → disk/memory 나눠서!
- solution : sequential file로써 disk에 접근 + sparce index 넣기
  - outer : sparse index
  - inner : basic index file
- outer도 너무 크면 또 다른 level의 index가 생성될 것임
- 모든 level의 index들이 file로부터 insertion, 혹은 deletion의 내용이 update 되어야 함