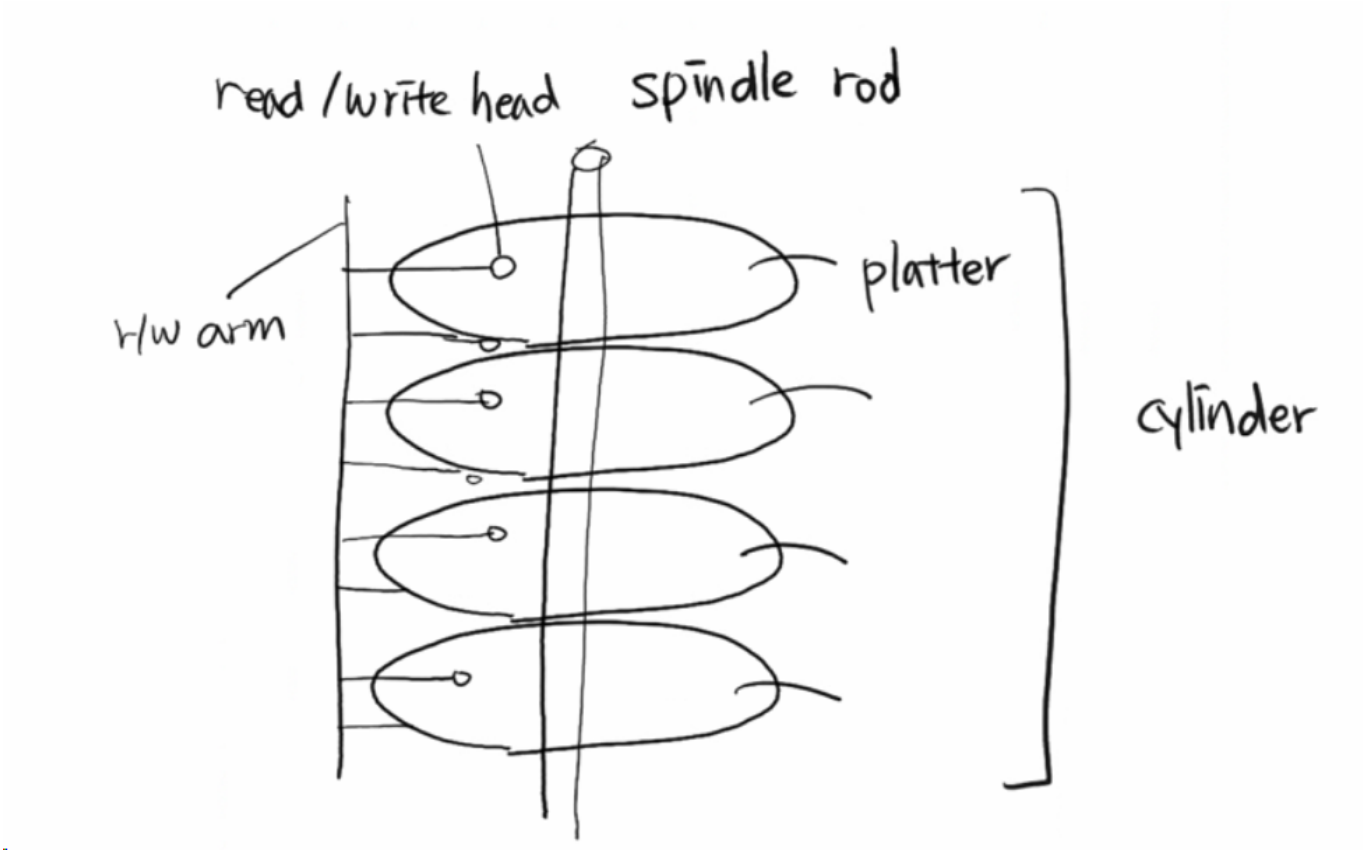
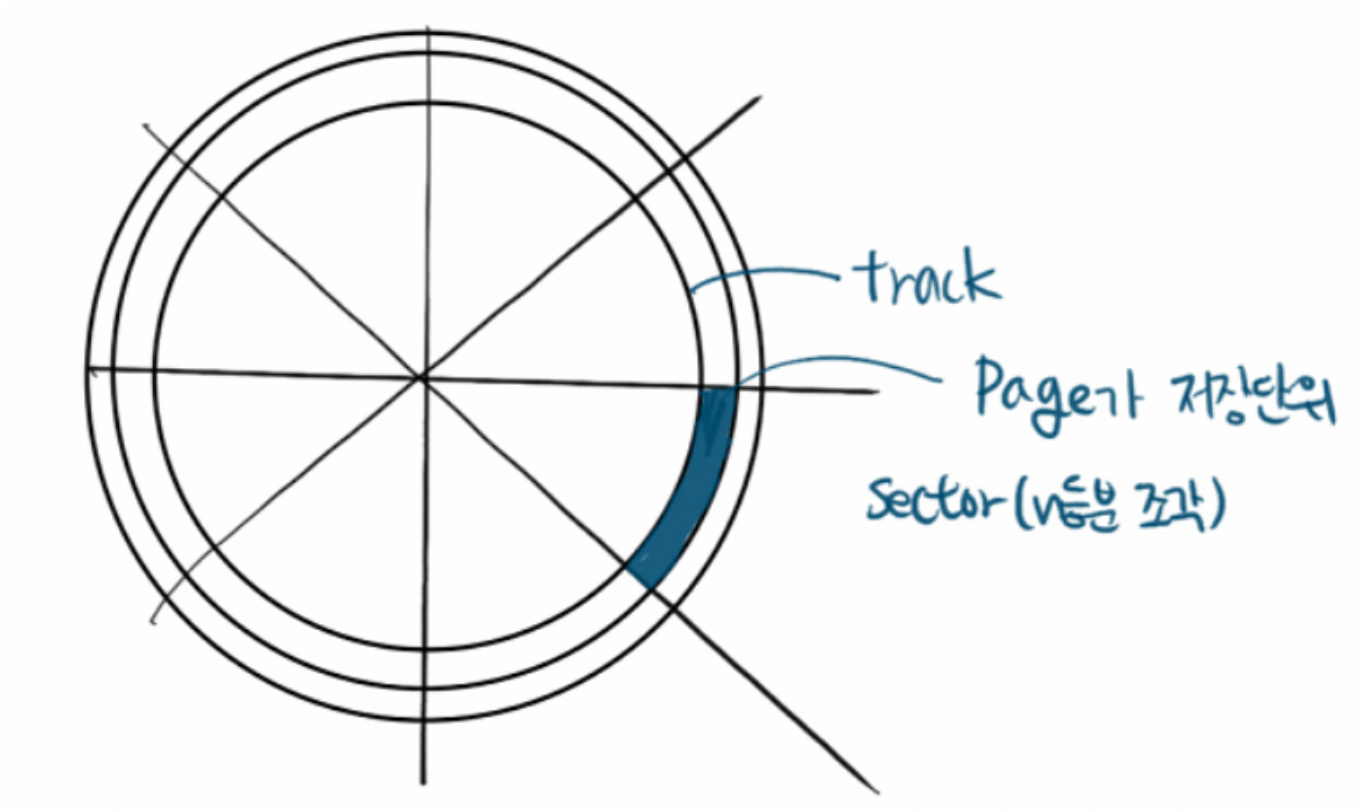
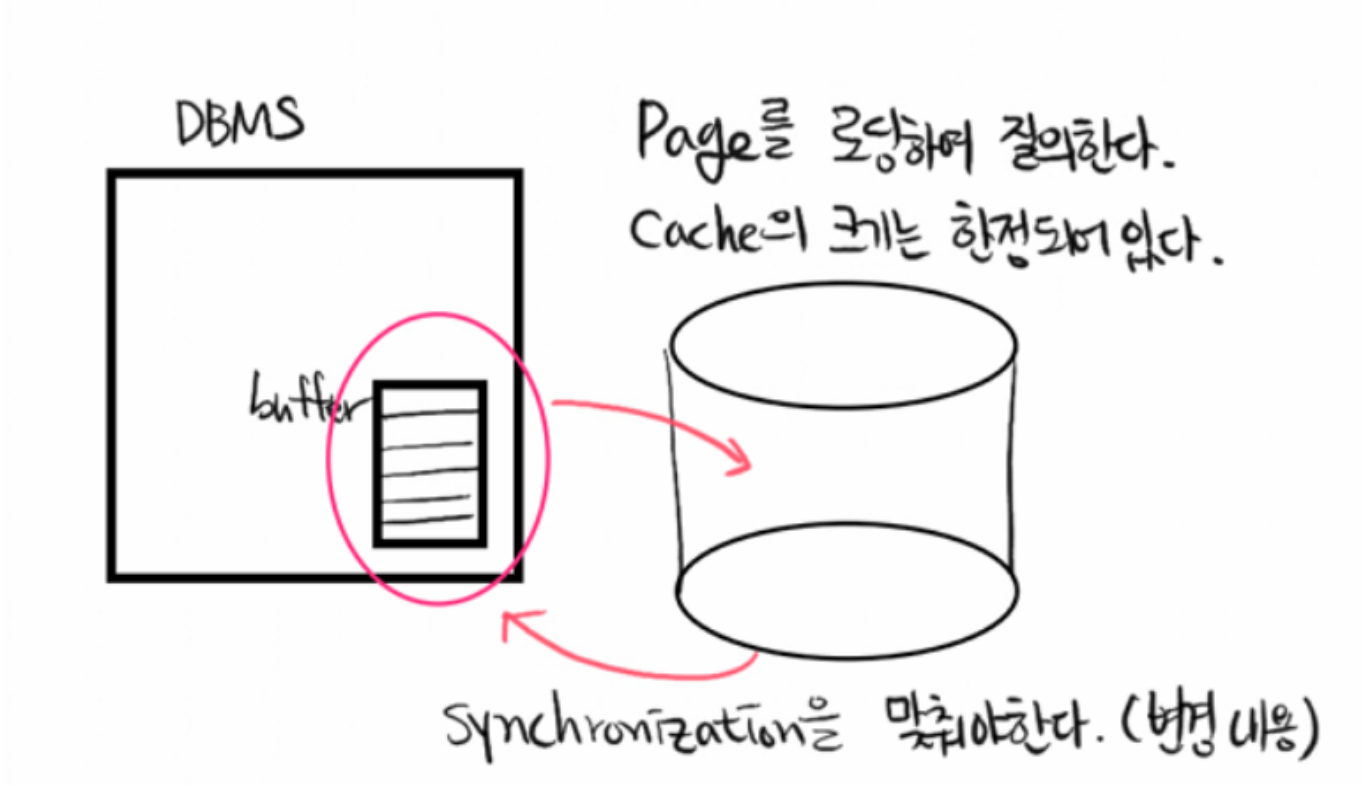


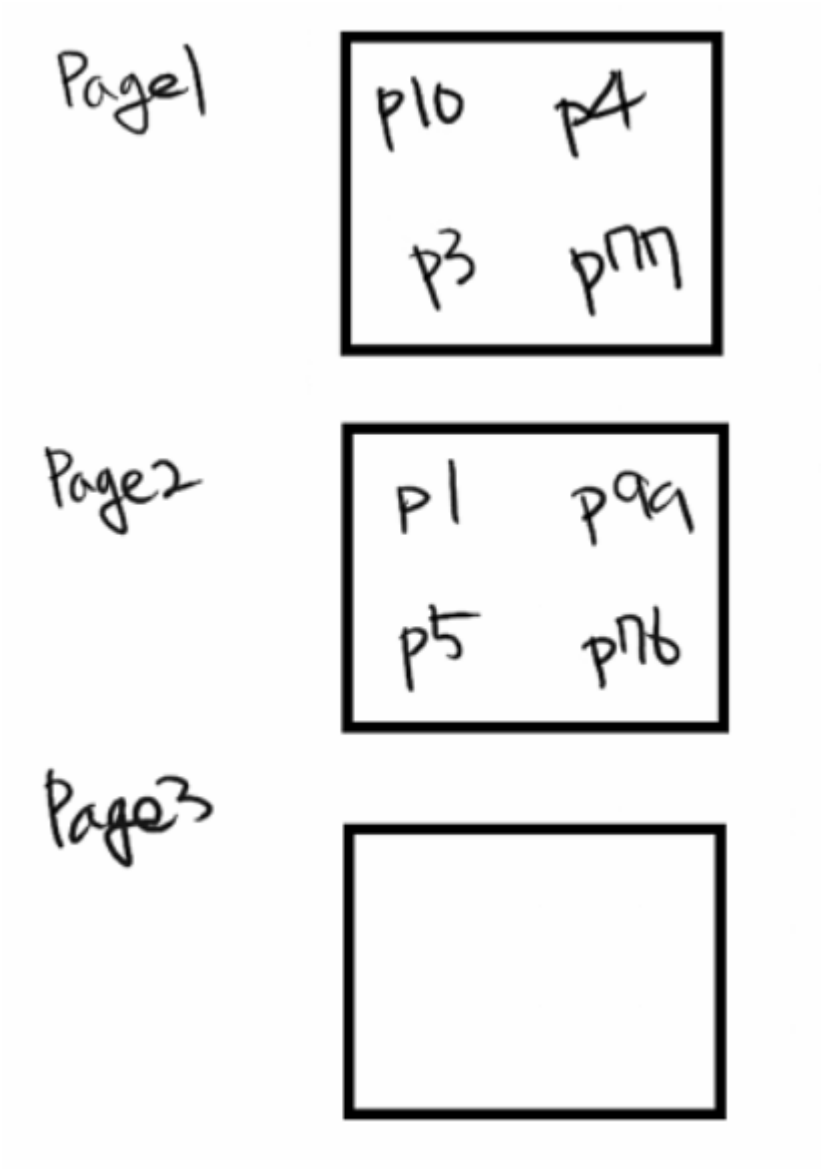


- Disk I/O Time = seek time(track) + rotation latency(sector)
- (Page)Allocation Unit : 4K~8K



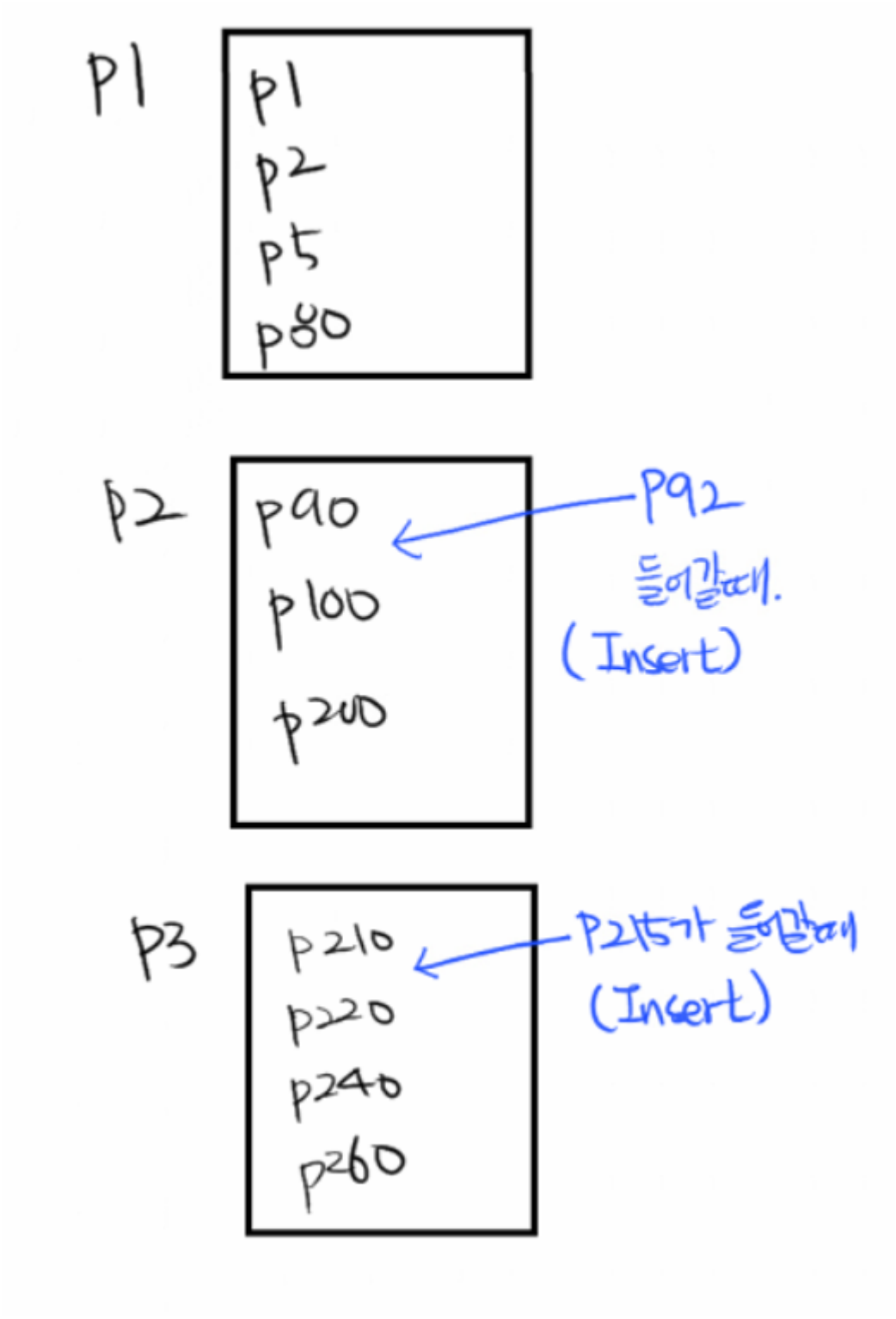
- eviction policy : LRU(가장 오래전에 쓴 Page를 Replace 대상으로 삼는다)

Heap File

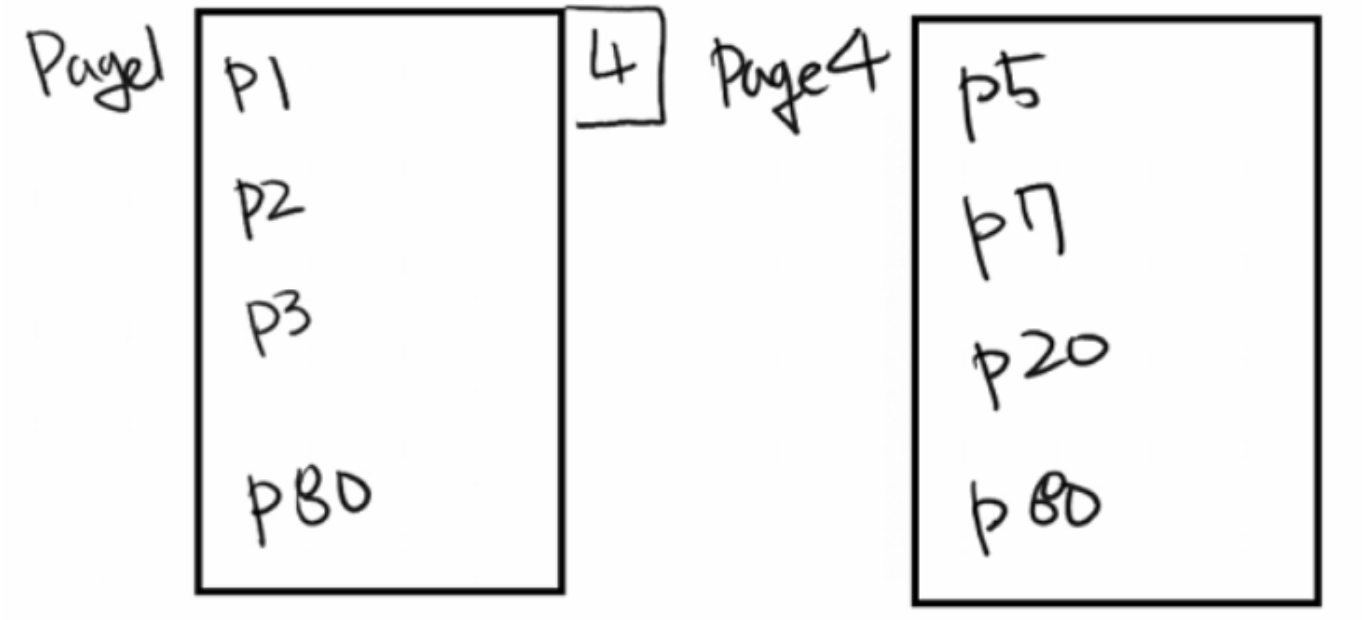


- 순서대로 저장하기 때문에 저장할 때는 효율이 좋다.
- Search, Delete시에 Full Scan을 요한다.
- 뺀 곳에 다시 넣기 위해 뺄때마다 Compaction이 필요하다.
- 평균 F/2

Sorted File



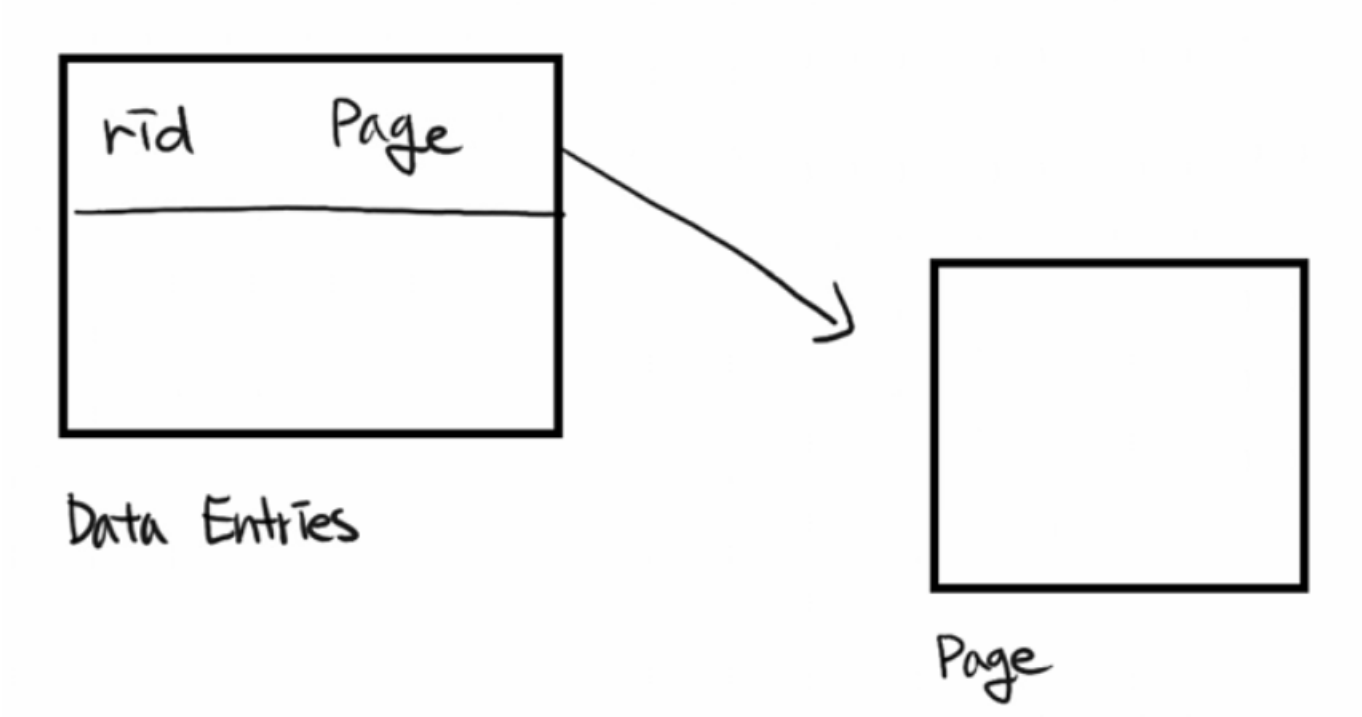
- Search가 빠르다. $\log_2 F$가 든다.
- Insert시에 뒤의 Page들이 다 밀려야한다, 비용이 큰 작업이다.
- 링크드 리스트는 파일 입장에서 효율적이지 않다. 메모리 입장에서만 좋다.
- 임시 해결책으로 들어갈 수도 있다는 가정으로 미리 비워둘 수 있는데 비효율적이다.



- Page가 가득 찼을때 Insert를 해야하면, 새로운 페이지에 넣고 포인터를 만드는 식으로 가능하다. 성능이 log_2F보다 느려진다.

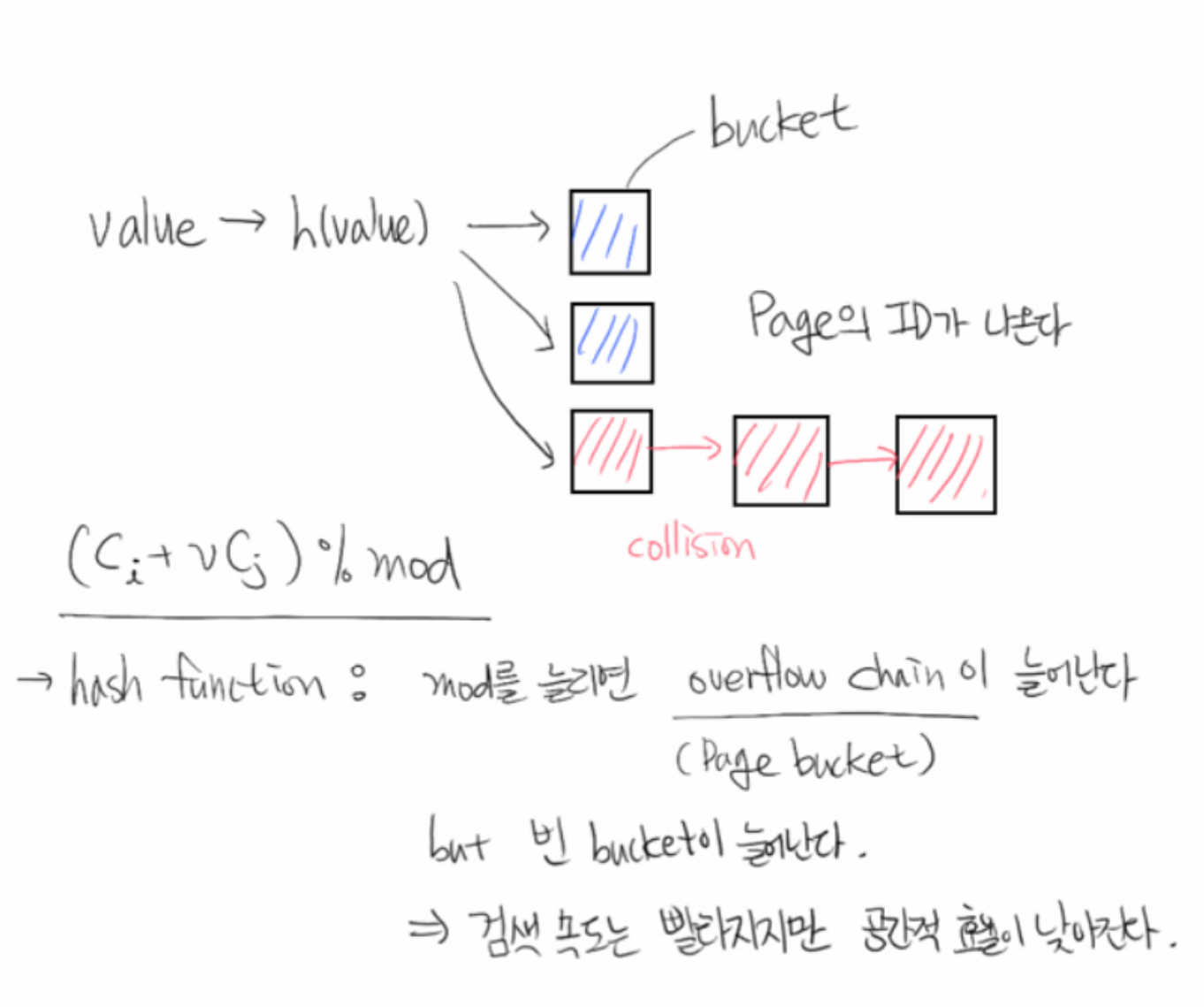
Heap, Sorted File은 index를 사용하지 않는다.

ISAM(Index Sequential Access Method)



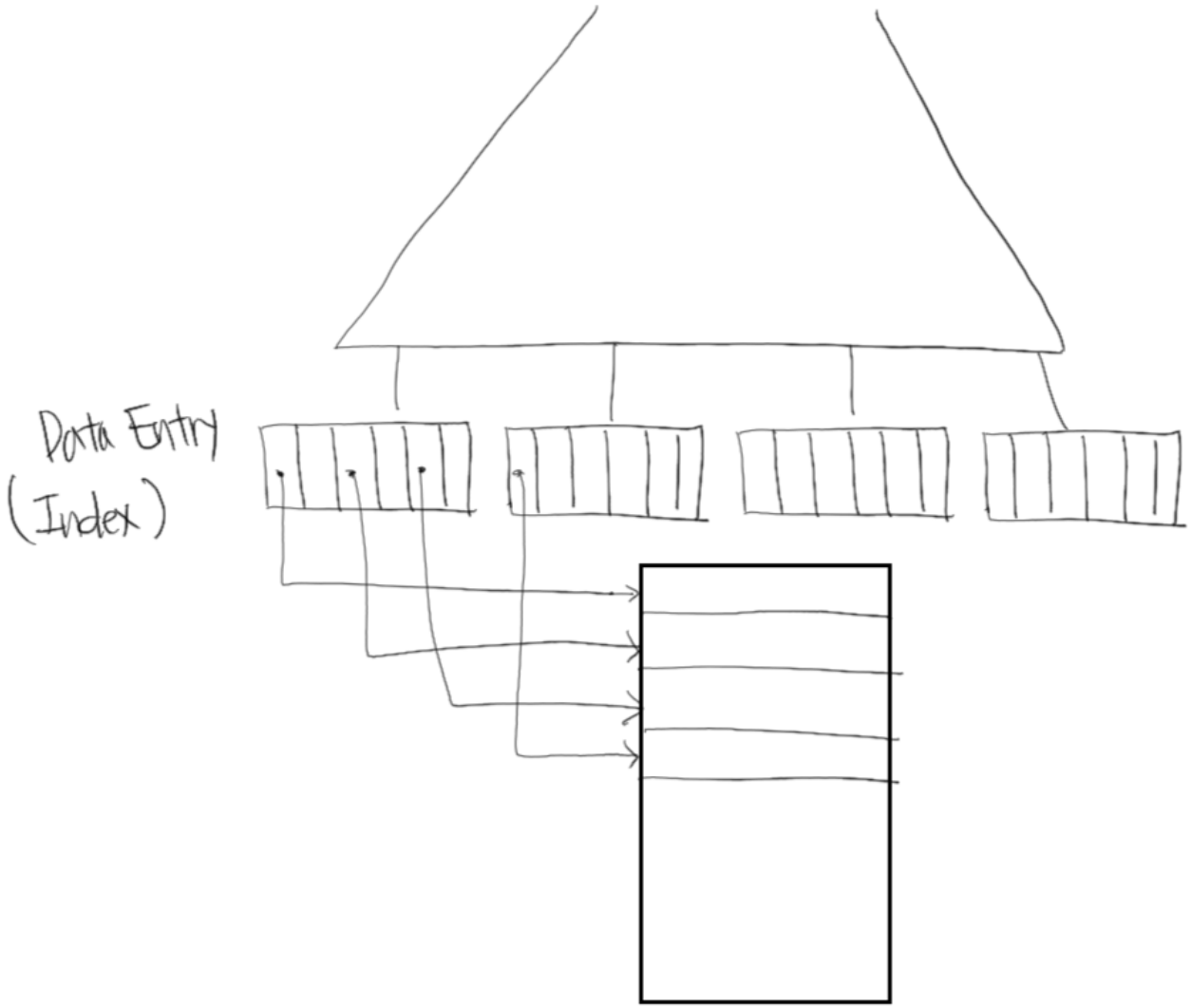
- Data Entry에 Data가 있거나, Page에 Data가 있을 수 있다.
- log_F(Page의 수)만큼 걸린다.
    - F : fan-out(분기 갯수)
- node의 길이가 기렁지면 순차탐색 시간이 늘어나지만 tree의 깊이가 작아진다.
- B+ Tree는 roof-leaf의 거리가 balancing 되어 있다.
- leaf node끼리는 서로 Linked List로 되어있다.
- range query시에 leaf node를 본다.

- Page와 Data Entry를 따로 두어서 Insertion, Delete시에 비용을 줄인다.

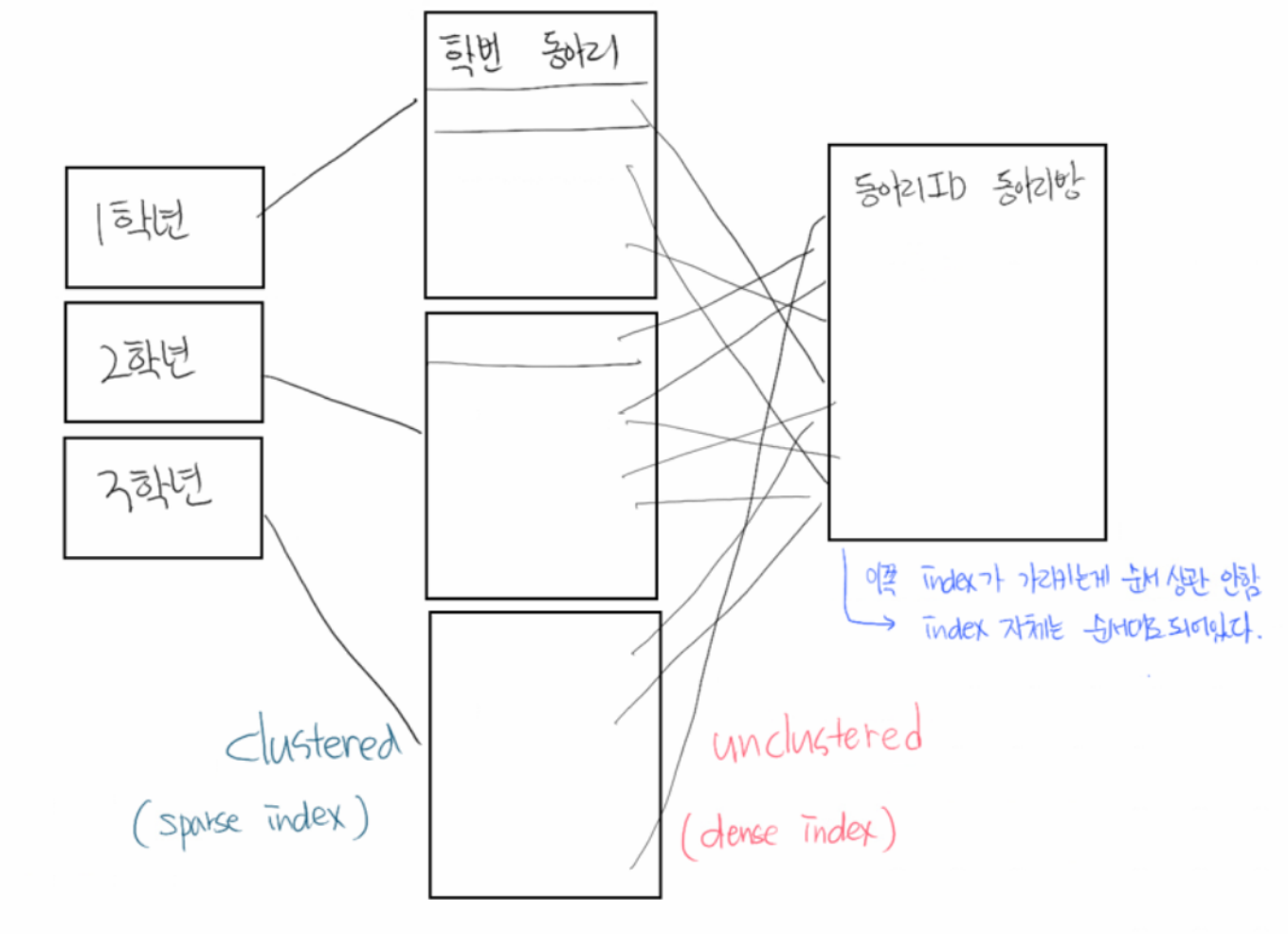


- 어떤게 더 좋은지 평가하기 위해 TPC Benchmark를 할 수 있다.

Cluster와 Uncluster



- cluster : Data Entry 순서에 맞게 Data Record의 순서가 같을 때.
- uncluster : 위의 경우에서 순서가 다를때.
- 책의 목차가 cluster, 책의 색인이 uncluster.


학번 동아리
1학년
2학년
3학년
동아리ID 동아리방
이때 index가 가리키는게 순서 상관 안함
index 자체는 순서대로 되어있다.
clustered
(sparse index)
unclustered
(dense index)