

# 인덱스

## Motivation

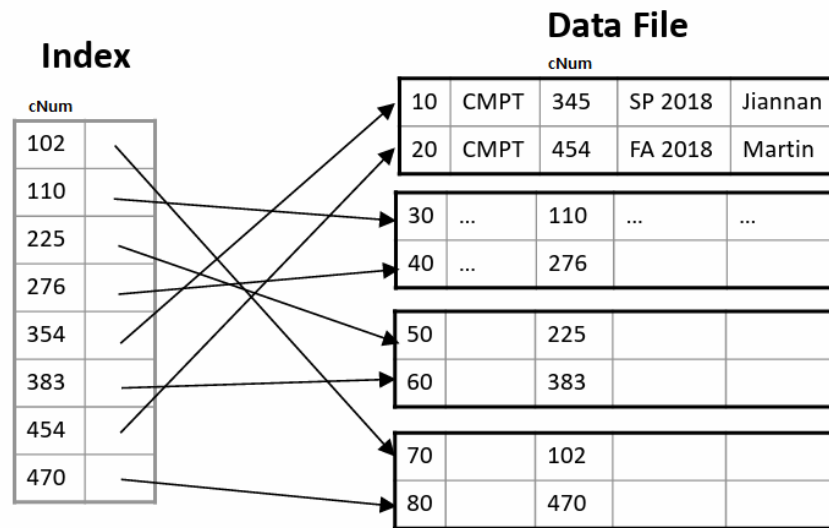
- 예를 들어 N개의 records를 가진 학생 테이블에서 **특정 나이**를 가진 학생을 찾고 싶다고 하자.
    - 만약 나이로 정렬되어 있다면 Binary Search를 이용하여  $O(\log N)$ 에 탐색이 가능하다.
    - 정렬 되어 있지 않다면 Full Scan하여  $O(N)$ 에 탐색이 가능하다.
  - 하지만 추가적으로 **특정 성적**을 가진 학생 또한 빠르게 찾고 싶다면?
    - 성적에 대해 정렬한 records가 또 필요하다.
    - 즉 성적에 대해 정렬한 records를 가져야 함 (복사본)
    - 결국 records에 대한 다수의 복사본을 가져야 하므로 많은 공간을 낭비한다.
- ⇒ Index라고 부르는 별도의 데이터 파일을 생성!

## 인덱스 만드는 방법

### 인덱스

- 인덱스: 검색 키가 주어질 때, 데이터 파일에서 records에 빠른 접근을 하기 위한 **추가적인 !파일!**
- 하나의 테이블은 여러개의 인덱스를 가질 수 있음

▪ Example) Index on cNum

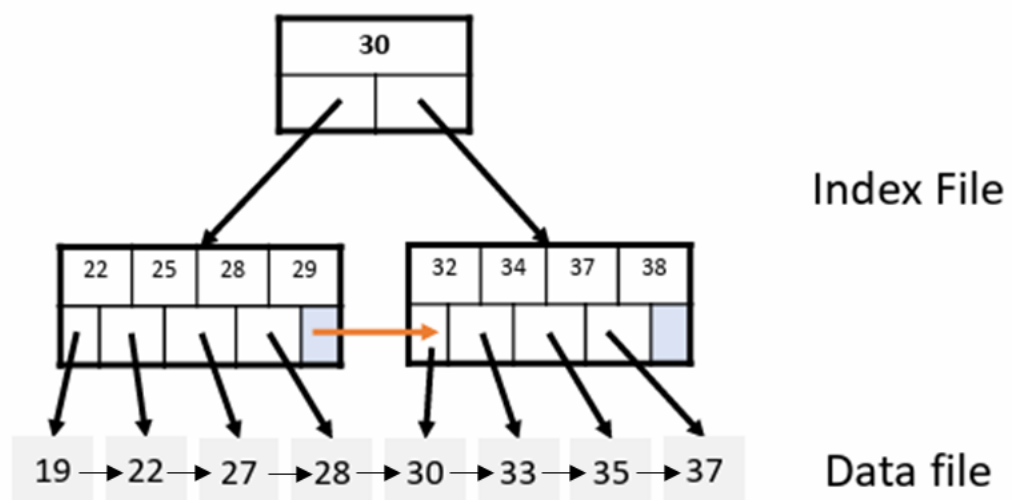


- 위 그림은 cNum에 대해 인덱스 파일을 생성한 것
- **Index파일은 정렬되어 있으므로 이진탐색**을 통해 빠르게 탐색 가능
- Key는 cNum, Value는 실제 record에 대한 포인터로 구성

## 클러스터형 인덱스(Clustered Index) vs 보조 인덱스(Secondary Index)

### 클러스터형 인덱스

- 클러스터형 인덱스는 **테이블당 하나**를 설정할 수 있다.
  - 실제 Data File을 인덱스와 동일하게 정렬한다.



**Clustered**

- 실제 Data File을 정렬하기 때문에, 특정 필드로 **범위 연산**을 할 때 효과적이다
  - 예를 들어 학생 **age가 20살 이상**인 records R개를 가져온다고 하자.
    - **1번의 Random IO**가 발생하고 **R번의 Sequential IO**가 발생한다.
    - 즉, Clusterd 인덱스가 없다면 R번의 Random IO가 발생한다. (**R번 Random IO → R번 Sequential IO**)



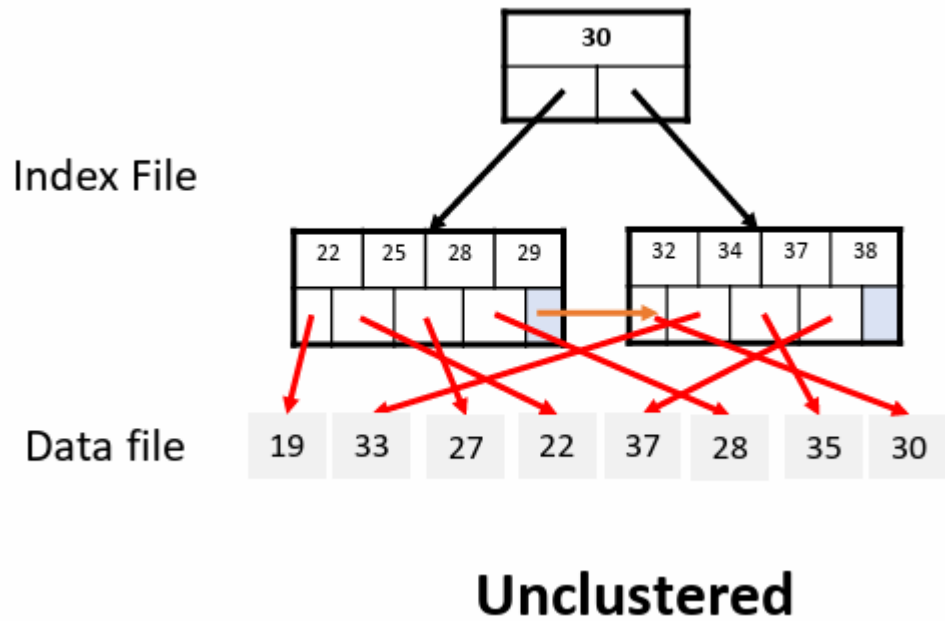
데이터의 1-2% Random read는 전체 파일을 Sequential scan하는 것보다 성능이 좋지 않다.

⇒ Sequential IO가 Random IO에 비해 약 50~100배 빠르다.

- 클러스터형 인덱스는 **1. Primary Key로 지정한 칼럼 2. UNIQUE NOT NULL로 지정한 칼럼**에 대해 생성된다.
  - **Primary Key를 탐색 키로 사용한 것이 기본 인덱스**라고 한다.
  - 위 2개가 함께 있으면 Primary Key에만 클러스터형 인덱스 생성
    - 클러스터형 인덱스는 테이블당 하나만 가질 수 있기 때문

## 보조 인덱스

- 보조 인덱스는 **Unclusterd Index**라고 한다.
- 테이블은 여러개의 보조 인덱스를 가질 수 있다.
  - 다양한 필드를 기반으로 쿼리를 보낼 수 있음
    - 즉 age, name, email 3개의 조건을 태워서 조회할 때
- Data File과 Index File의 정렬이 일치하지 않다.
  - 범위 검색(<, >)에 효과적이지 않다.
    - 범위 검색에서 records R개를 가져온다고 하면, R번의 Random IO가 필요
  - 하지만 존재 검색(=)은 **Clustered Index와 동일한 성능** (1번의 Random IO)



## 요약

- 클러스터형 인덱스를 적용하면 실제 Data File을 Index File과 동일하게 정렬.
  - 한 번의 Random IO로 연속적인 데이터 접근
  - 범위 검색에 효과적
- 보조 인덱스는 비클러스터형 인덱스로 여러개의 인덱스 파일을 가질 수 있다.
  - 여러개의 필드를 기반으로 조회 쿼리를 할 때 유용
  - 범위 검색에 약함. 존재 검색은 클러스터형과 동일

## 인덱스 최적화 기법

### SQL에서 인덱스 생성

- Create an index with **CREATE INDEX** command
  - **CREATE INDEX** <name> **ON** <relation-name> (attribute);

```
create index dept_index
on instructor (dept_name);
```

- Delete an index with **DROP INDEX** command
  - **DROP INDEX** <index-name>

```
drop index dept_index;
```

- Default는 보조 인덱스로 생성


## 인덱스 최적화

### 1. 인덱스는 비용이다

- 인덱스는 탐색이 두 번 필요하다.
  - 먼저 1. **인덱스 리스트를 탐색**한 후, 2. **컬렉션을 탐색**한다.
  - 예시로 해시함수를 인덱스 자료구조로 사용하는 경우를 보자.

- A naïve hash function

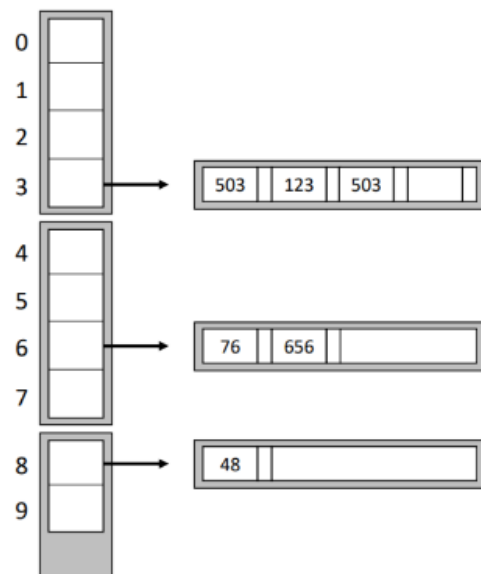
$$h(x) = x \bmod 10$$

 : A disk block

➔ Cost per lookup:

- One access in array
- One access in list

- No range queries



- 503을 탐색하기 위해,  $503 \bmod 10 = 3$ 이므로 **인덱스 파일의 3을 이진탐색으로 빠르게 찾는다.**
- 이 후 3이 가리키는 List에서 **Sequential**하게 탐색한다.

- 컬렉션이 수정되었을 때 인덱스도 수정되어야 한다.
  - + B-트리의 높이를 균형 있게 조절하는 비용
  - + 데이터를 효율적으로 조회할 수 있도록 분산시키는 비용

⇒ 꼭 필요한 곳에만 인덱스를 사용하자. 인덱스를 설정하면 update시 느려진다.

## 2. 항상 테스트하라

- 서비스 특징에 따라 인덱스 최적화 기법이 달라진다.
  - 정확하게 딱 떨어지는 답이 있는 것은 아니다.
  - 서비스에서 사용하는 객체의 깊이, 테이블의 양 등이 다르기 때문
- 항상 테스트하여 성능을 측정
  - explain() 함수를 통해 인덱스를 만들고 쿼리를 보낸 이후에 테스트하여 걸리는 시간을 최소화
  -

## 3. 복합 인덱스는 같음, 정렬, 다중 값, 카디널리티 순이다.

- 여러 필드를 기반으로 조회를 할 때 복합 인덱스를 생성
  - 이 때 순서가 중요하다. 순서에 따라 성능이 달라진다.
  - 같음, 정렬, 다중 값, 카디널리티 순으로 생성
    1. 어떠한 값과 같음을 비교하는 ==이나 equal이 포함되는 쿼리가 있다면 제일 먼저 인덱스로 설정
    2. 정렬에 쓰는 필드라면 그다음 인덱스로 설정
    3. 다중 값을 출력해야 하는 필드, 즉 쿼리 자체가 >이거나 <등 많은 값을 출력해야 하는 쿼리에 쓰는 필드라면 나중에 인덱스를 설정
    4. 유니크한 값의 정도를 카디널리티라고 한다. 이 카디널리티가 높은 순서를 기반으로 인덱스를 생성해야 한다.
      - 예를 들어 age와 email이 있을 때, email이 카디널리티가 높으므로 email필드에 대한 인덱스를 먼저 생성한다.
      - 같은 의미로 데이터 비율이 낮을수록 효과적이다. (선택도가 낮을수록 효과적)
- 복합 인덱스 아래 사진 참고

- Index on multiple attributes
  - When the WHERE clause frequently uses the combination of several columns,

```
SELECT * FROM instructor
WHERE dept_name = 'CSE' AND name = 'Albert'
```

AND only



```
create index dept_index
on instructor (dept_name, name);
```

- What is the difference between (dept\_name, name) and (name, dept\_name)?
  - (dept\_name, name) → works when “where dept\_name = ‘CSE’”, but does not work when “where name = ‘Albert’”.
  - (name, dept\_name) → works when “where name = ‘Albert’”, but does not work when “where dept\_name = ‘CSE’”.
  - (A, B, C) → works when “where A = ?”, “where A = ? AND B = ?”, and “where A = ? AND B = ? AND C = ?”.

## 인덱스 선택 문제

- DBA가 적절한 인덱스를 선택한다.
- 주어진 테이블에 대해 어떤 인덱스를 생성해야하고 하지 말아야할까?
- 만약 WHERE절에 다음이 포함된다면, 속성 집합 K를 인덱스로 설정을 고려하자
  - K에 exact match (=) → Unclustered
  - K에 범위 연산 (<, >) → Clustered
  - K로 Join (=, <, >)

## 추가) 예제문제

1.

▪ Example)

Offering (oID, dept, cNum, term, instructor)

```
CREATE INDEX IDX1 ON Offering(dept)
```

➔ Which query(s) could be affected by IDX1?

(A)

```
SELECT oID FROM Offering  
WHERE dept = 'CMPT'
```

(B)

```
SELECT oID FROM Offering  
WHERE cNum = '354'
```

(C)

```
SELECT oID FROM Offering  
WHERE dept = 'CMPT' AND cNum = '354'
```

- Ans: (A), (C)

2.

▪ Example)

Offering (oID, dept, cNum, term, instructor)

```
CREATE INDEX IDX2 ON Offering(dept, cNum)
```

➔ Which query(s) could be affected by IDX2?

(A)

```
SELECT oID FROM Offering  
WHERE dept = 'CMPT'
```

(B)

```
SELECT oID FROM Offering  
WHERE cNum = '354'
```

(C)

```
SELECT oID FROM Offering  
WHERE dept = 'CMPT' AND cNum = '354'
```

- Ans: (A), (C)

- cNum에 대해 인덱스가 걸려 있지만 dept를 기준으로 정렬된 상태에서, cNum으로 정렬하기 때문에 Full Scan과 동일



3.

▪ Example 1)

If your workload is

100000 queries

```
SELECT *  
FROM R  
WHERE N = ?
```

100 queries

```
SELECT *  
FROM R  
WHERE P = ?
```

Which one(s) are useful?

- A. Index on N
- B. Index on P

- **Ans: 둘 다 유용할 수 있지만, A가 더 유용**
  - 쿼리 수

4.

▪ Example 2)

If your workload is

100,000 queries

```
SELECT *  
FROM R  
WHERE N > ? AND N < ?
```

100 queries

```
SELECT *  
FROM R  
WHERE P = ?
```

100,000 queries

```
INSERT INTO R  
VALUES (?, ?, ?)
```

Which one(s) are useful?

- A. Index on N
- B. Index on P

- **Ans: A**
  - 범위 탐색이므로 B-Tree 구조를 가지며, Clustered Index를 사용해야 효과 있음
  - B는 알 수 없다.
    - Insert 쿼리가 많기 때문에 오버헤드가 크다.
    - 하지만 데이터 비율이 작을 경우 효과적일 수 있다.
      - 만약 테이블이 큰데, age=21인 학생이 한 명 뿐이라면?

- 매우 효율적
- 반면에 age=21인 사람이 많다면?
  - Full Scan과 유사

5.

▪ Example 3)

If your workload is

100,000 queries

```
SELECT *
FROM R
WHERE N=?
```

1,000,000 queries

```
SELECT *
FROM R
WHERE N= ? AND P>?
```

100,000 queries

```
INSERT INTO R
VALUES (?, ?, ?)
```

Which one(s) are useful?

- A. Index on (N,P)
- B. Index on (P,N)
- C. Index on N
- D. Index on P

• Ans: A, C, D

- A가 Best

- N에 대해 정렬된 상태에서 추가적으로 P에 대해 정렬하므로, N=?를 이진 탐색으로 찾고, 그 결과에서 P>?를 이진 탐색으로 빠르게 찾을 수 있다.

6.

▪ Example 4)

If your workload is

100000 queries

```
SELECT sID
FROM Student
WHERE name = ?
```

100000 queries

```
SELECT sID
FROM Student
WHERE gender = ?
```

Which one is better?

- A. Index on name
- B. Index on gender

• Ans: A

- A가 더 효과적이다.

- 이진 탐색해서 해당 name에 해당하는 rows만 가져온다.
- B의 경우 성별은 남/여 두가지 이므로, 카디널리티가 낮다. 즉, 선택도가 낮음.  
⇒ 선택도가 높으면 테이블에 값이 다양하다는 뜻이고, 인덱스가 효과적이다.

7.

■ Example 5)

If your workload is

100000 queries

```
SELECT sID
FROM Student
WHERE name like ?
```

100000 queries

```
SELECT sID
FROM Student
WHERE age = ?
```

Which one is better?

- A. Index on name
- B. Index on age

• Ans: B

- like는 pattern matching으로 정렬 유무와 관련이 적다.
  - ex) name like '\_\_ab'

8.

■ Example 6)

If your workload is

100000 queries

```
SELECT sID
FROM Student
WHERE name = ?
```

100 queries

```
SELECT sID
FROM Student
WHERE age = ?
```

Which one(s) are useful?

- A. Index on name
- B. Index on age
- C. Index on name, age
- D. Index on age, name

• Ans: 일반적인 경우 C-A-D-B순으로 유용

- 하지만 Student 테이블이 매우 큰데, age=24인 사람이 1명 밖에 없을 경우와 같이 전체 데이터 대비 해당 데이터가 작을 때(카디널리티가 높을 때) 더 효과적일 수 있다.

9.

▪ Example 7)

If your workload is

100000 queries

```
SELECT sID
FROM Student
WHERE fname = ?
```

100000 queries

```
SELECT sID
FROM Student
WHERE fname = ? AND age > ?
```

Which one is better?

- A. Index on (fname, age)
- B. Index on (age, fname)

• Ans: A

10.

▪ Example 8)

If your workload is

100000 queries

```
SELECT sID
FROM Student
WHERE name = ?
```

100 queries

```
SELECT sID
FROM Student
WHERE age = ?
```

100000 queries

```
INSERT INTO Student
VALUES (?, ..., ?)
```

Which one(s) are useful?

- A. Index on name
- B. Index on age
- C. Index on name, age
- D. Index on age, name

• Ans: 알 수 없다. 만약 선택한다면 A, C순으로 고려

- Insert 쿼리 수가 많다.
  - Insert와 존재 검색의 효율성을 비교해야 함.
  - C는 Insert 과부하 가능성이 높음

## 요약

- 인덱스 선택 가이드 라인
  1. 중요도에 따라 워크로드의 쿼리 고려
  2. 쿼리에 의해 접근되는 Relation을 고려
  3. 가능한 탐색 키에 대한 WHERE절을 관찰
  4. 다수의 쿼리에 대해 속도를 향상 시킬 수 있는 인덱스를 고르기
  5. 범위 탐색 쿼리는 클러스터 인덱스를 사용