데이터베이스

색인

학습목표

- 색인의 개념을 설명할 수 있다.
- 색인의 내부 구조를 파악하여 데이터의 삽입과 삭제에 따른 변화를 설명할 수 있다.

학습내용

- 색인의 개념
- 색인의 내부 구조

색인의 필요성

데이터 조직 방법

01 데이터 조직 방법

순차 방법	데이터 (튜플)을 일정한 순서대로 저장하는 방법
색인 방법	부가적인 자료 구조를 이용하여 데이터의 빠른 접근을 지원하는 방법
해싱 방법	해싱 함수(Hashing function)를 이용하여 데이터를 그룹핑 하는 방법

순차 방법

엔트리 순차 파일	레코드가 생성되는 순서대로 저장하는 방법
키 순차 파일	레코드의 키값 순서대로 저장하는 방법

색인의 필요성

데이터 조직 방법

01 데이터 조직 방법

순차 방법

- 엔트리 순차 파일의 특징
  - 장점: 쉬운 레코드 저장
  - 단점: 필요한 레코드를 찾기 위해서 전체를 다 검색해야 함

s4	s5	s1
s2	s7	

- 키 순차 파일의 장·단점
  - 장점: 키값으로 정렬되어 있으므로 키값을 이용한 검색이 효율적임
  - 단점: 새로운 레코드 삽입 시 매우 느림

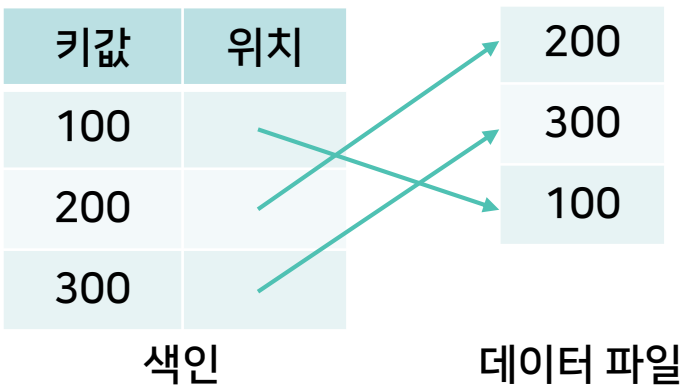
s1	s5	s7
s8	s9	

색인의 필요성

색인의 개념

01 색인(Index)

- 검색 성능을 향상 시키기 위한 부가적인 자료 구조
- SQL 명령문의 검색 속도를 향상시키기 위해 칼럼에 대해 생성하는 객체
  - 사용 목적
    - 포인트를 이용하여 테이블에 저장된 데이터를 랜덤 액세스하기 위함



색인의 필요성

색인의 개념

02 인덱스가 효율적인 경우

01 WHERE절이나 조인 조건절에서 자주 사용되는 칼럼

02 전체 데이터 중에서 10~15% 이내의 데이터를 검색하는 경우

03 두 개 이상의 칼럼이 WHERE절이나 조인 조건에서 자주 사용되는 경우

04 테이블에 저장된 데이터의 변경이 드문 경우

03 인덱스 생성/삭제 구문

색인 생성

- CREATE INDEX 색인명
- ON 테이블명(속성명, 속성명,…)

색인 삭제

- DROP INDEX 색인명
- ON 테이블명

색인의 필요성

색인의 종류

01 종류

- 고유 인덱스 VS 비고유 인덱스
- 단일 인덱스 VS 결합 인덱스
- 내림차순 인덱스(Descending Index)
- 집중 인덱스 VS 비집중 인덱스

02 고유 인덱스 VS 비고유 인덱스

고유 인덱스		비고유 인덱스
유일값을 가지는 속성에 대하여 생성하는 색인으로 각 키값은 테이블의 하나의 튜플과 연관됨	VS	중복된 값을 가지는 속성에 생성하는 인덱스로 키값은 여러 개의 튜플들과 연관됨

색인의 필요성

색인의 종류

02 고유 인덱스 VS 비고유 인덱스

기본키

- 테이블이 기본키에 대해서는 자동으로 고유 색인이 생성됨
  → 기본 색인(Primary Index)
  - 기본키는 중복을 허용하지 않음
- 새로운 튜플을 삽입 할 때 마다 키값이 고유값인지 검사해야 함
- 테이블에 속한 튜플들이 많다면 매우 느림
- 고유 색인을 이용해야 함

관계형 테이블의 검색

- 테이블 검색 시 기본키만을 사용하지 않음

> 예 | 학생 테이블에서 학번이 100번인 학생을 검색하시오.

- 실제로는 학생을 검색할 때 학번보다 이름을 이용하는 경우가 더 많음
- 검색을 빨리 하려면 조건에 많이 사용되는 컬럼에 대하여 색인을 생성함
  → 보조 색인(Secondary Index)

색인의 필요성

색인의 종류

02 고유 인덱스 VS 비고유 인덱스

고유 인덱스의 생성

- 고유 인덱스를 생성할 때는 UNIQUE 키워드를 사용

예 | 부서 테이블에 부서 이름에 대하여 고유 색인을 생성하자.

```
CREATE UNIQUE INDEX idx_dname_unique
ON DEPARTMENT(dname)
```

```
Messages
Command(s) completed successfully.
```

비고유 인덱스의 생성

- UNIQUE 없이 색인을 생성하면 비고유 색인이 됨

예 | 부서 테이블에 위치에 대하여 비고유 색인을 생성하자.

```
CREATE INDEX idx_loc_unique
ON DEPARTMENT(loc)
```

```
Messages
Command(s) completed successfully.
```

색인의 필요성

색인의 종류

03 단일 인덱스 VS 결합 인덱스

단일 인덱스		결합 인덱스
하나의 속성만으로 구성된 색인	VS	두 개 이상의 속성들에 대하여 생성된 색인

결합 인덱스의 생성

예 | 직원 테이블에서 부서 번호와 급여에 대하여 결합 인덱스를 생성해 보자.

```
CREATE INDEX idx_dnosalary
ON EMPLOYEE(dno, salary)
```

```
Messages
Command(s) completed successfully.
```

색인의 필요성

색인의 종류

03 단일 인덱스 VS 결합 인덱스

내림차순 인덱스(Descending Index)

- 특별히 속성별로 정렬 순서를 지정하여 결합 인덱스를 생성하는 방법
- 특징
  - 일반적인 색인들은 속성값에 대하여 오름차순으로 정렬되어 저장됨
  - 색인 생성 시에 각 속성별로 정렬순서(DESC, ASC)를 정해주면 됨

Descending Index의 생성

예 | 사원에 대하여 부서 번호는 오름차순, 급여는 내림차순으로 하여 색인을 생성하자.

```
CREATE INDEX idx_dnosalary_desc
ON EMPLOYEE(dno asc, salary desc)
```

```
Messages
Command(s) completed successfully.
```

색인의 필요성

색인의 종류

04 집중 인덱스 VS 비집중 인덱스

집중 인덱스(Clustered Index)

- 테이블의 튜플이 저장된 물리적 순서 해당 색인의 키값 순서와 동일하게 유지되도록 구성된 색인
- 기본키에 대하여 생성된 색인
- 테이블의 튜플들이 기본키에 오름차순으로 정렬되어 저장되어 있고 기본키 색인 또한 기본키에 따라서 오름차순으로 정렬되어 있음
- 하나의 테이블에 대하여 하나만 생성할 수 있음

비집중 인덱스(Unclustered Index)

- 집중 인덱스가 아닌 인덱스들

집중 인덱스 (Clustered Index)	VS	비집중 인덱스 (Unclustered index)
▶ 테이블의 튜플이 저장된 물리적 순서로 해당 색인의 키값 순서와 동일하게 유지되도록 구성된 색인 ▶ 기본키에 대하여 생성된 색인 ※ 테이블의 튜플들이 기본키에 오름차순으로 정렬/저장되어 있고 기본키 색인 또한 기본키에 따라서 오름차순으로 정렬되어 있음 ▶ 하나의 테이블에 대하여 하나만 생성할 수 있음		집중 인덱스가 아닌 인덱스들

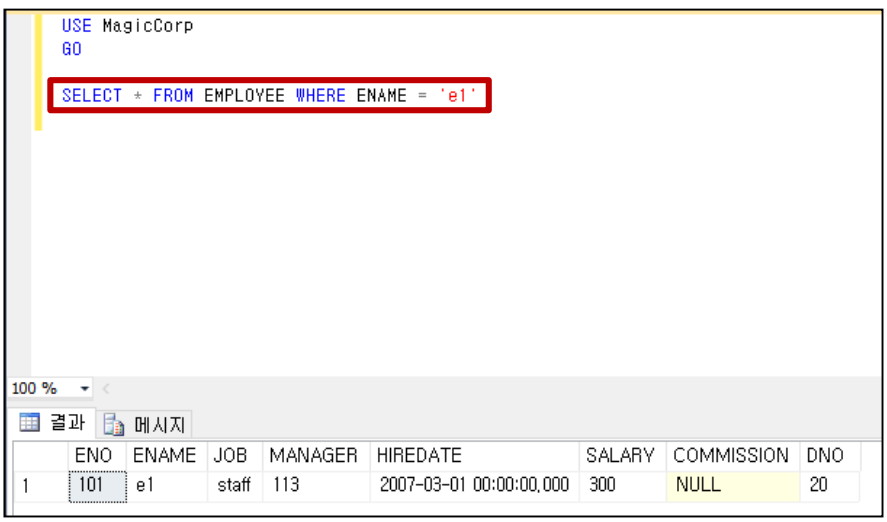
색인의 필요성

색인의 활용

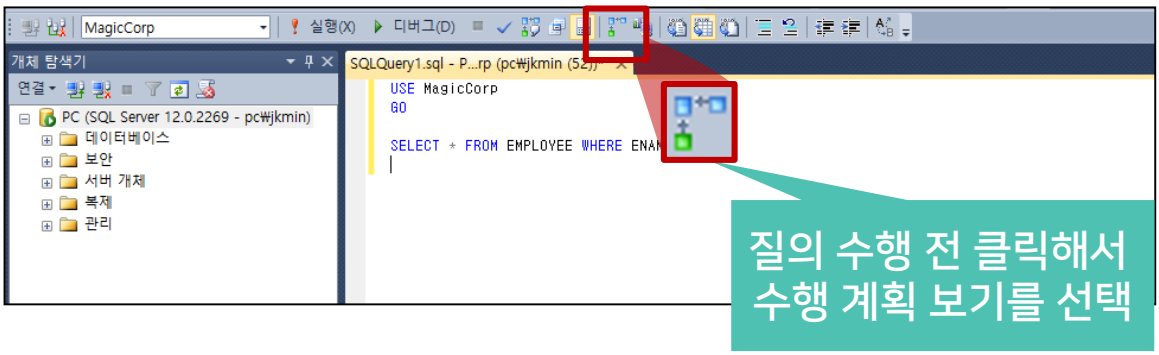
01 질의 수행 시 인덱스를 사용하는 방법

예 | 사원 이름이 'e1'인 사원의 정보를 검색하시오.

```
SELECT * FROM EMPLOYEE WHERE ENAME = 'e1'
```



MS-SQL에서 질의 수행 계획을 보는 방법-①

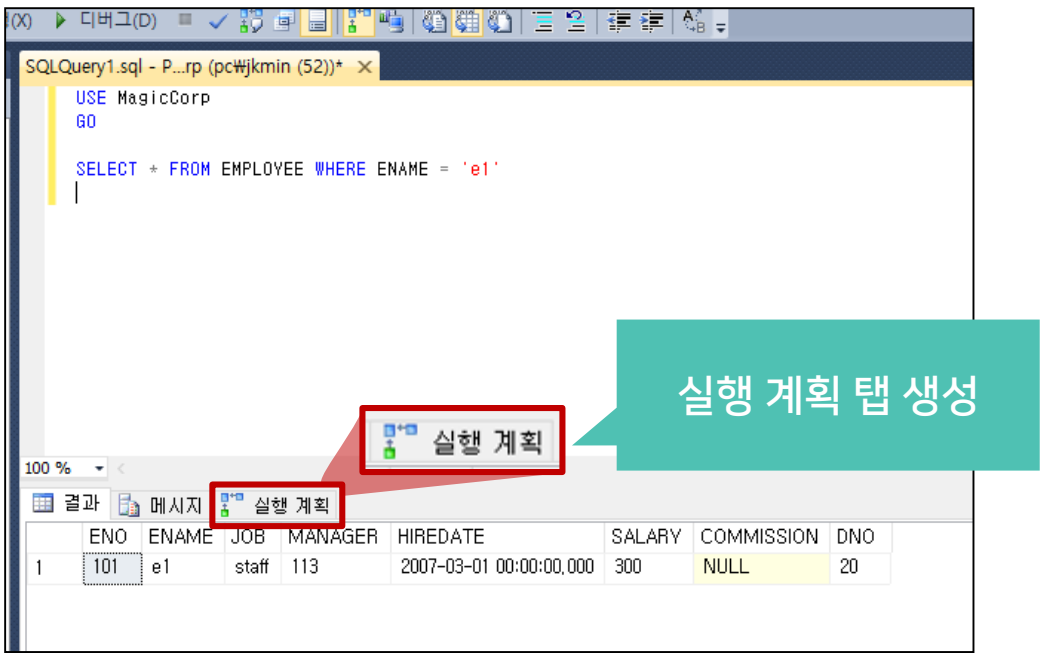


색인의 필요성

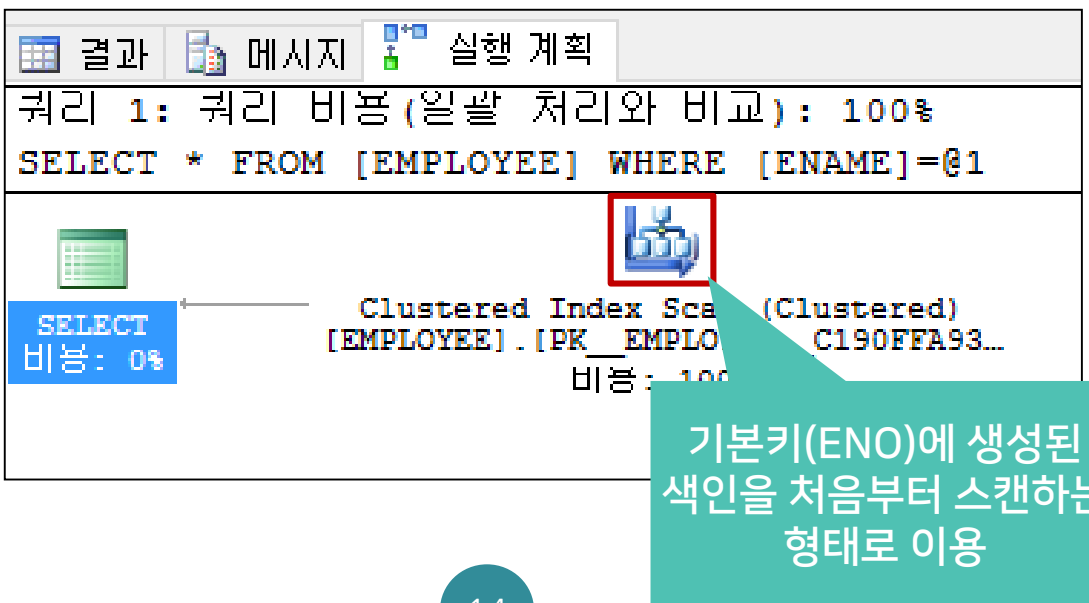
색인의 활용

01 질의 수행 시 인덱스를 사용하는 방법

MS-SQL에서 질의 수행 계획을 보는 방법-②



MS-SQL에서 질의 수행 계획을 보는 방법-③ | 질의 수행 방법



색인의 필요성

색인의 활용

02 질의 수행 시 인덱스를 강제로 사용하는 방법

- 질의는 기본키(ENO)로 탐색하는 것이 아니라 ENAME을 가지고 탐색

```
SELECT * FROM EMPLOYEE WHERE ENAME = 'e1'
```

- EMPLOYEE 테이블의 ENAME을 가지고 색인 emp_name_idx을 만들 경우

  - 질의 수행기가 해당 색인을 사용하지 않음
  - 강제로 emp_name_idx를 사용하게 할 수 없을까?

```
USE MagicCorp
GO

CREATE INDEX emp_name_idx
ON EMPLOYEE(ename)
```

색인의 필요성

색인의 활용

02 질의 수행 시 인덱스를 강제로 사용하는 방법

- FROM절에 WITH(INDEX=INDEX_NAME)를 추가하여 강제로 특정 색인을 사용하게 할 수 있음
  - emp_name_idx를 사용

```
USE MagicCorp
GO

SELECT * FROM EMPLOYEE WITH (INDEX = emp_name_idx) WHERE ENAME = 'e1'
```

100 %

결과 | 메시지 | 실행 계획

	ENO	ENAME	JOB	MANAGER	HIREDATE	SALARY	COMMISSION	DNO
1	101	e1	staff	113	2007-03-01 00:00:00.000	300	NULL	20

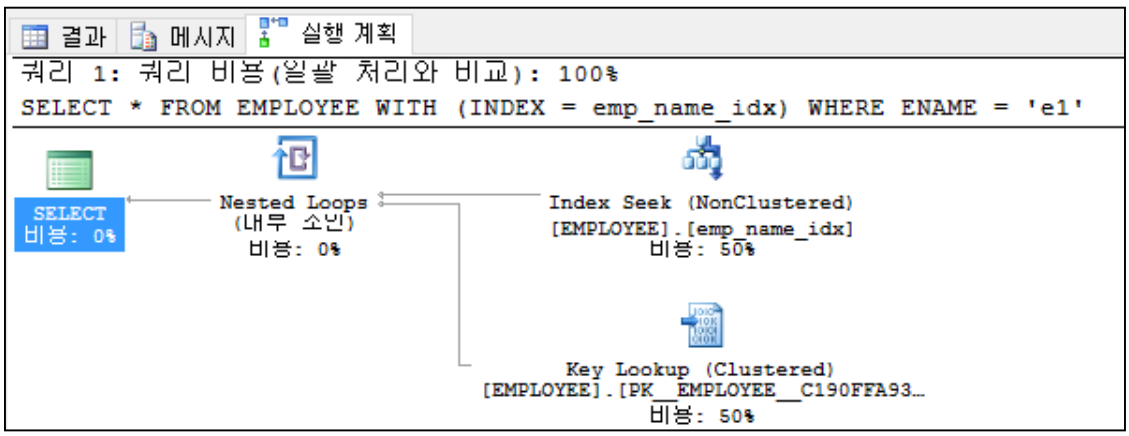
색인의 필요성

색인의 활용

02 질의 수행 시 인덱스를 강제로 사용하는 방법

색인을 사용하는지 질의 수행 계획 확인

- emp_name_idx를 통해서 이름이 e1인 튜플의 기본키값을 파악(Index Seek)
- 파악된 기본키를 이용하여 기본 색인(PK_EMPLOYEE__...)을 검색하여 결과를 찾도록 수행



색인의 내부 구조

B-tree

01 트리(Tree)

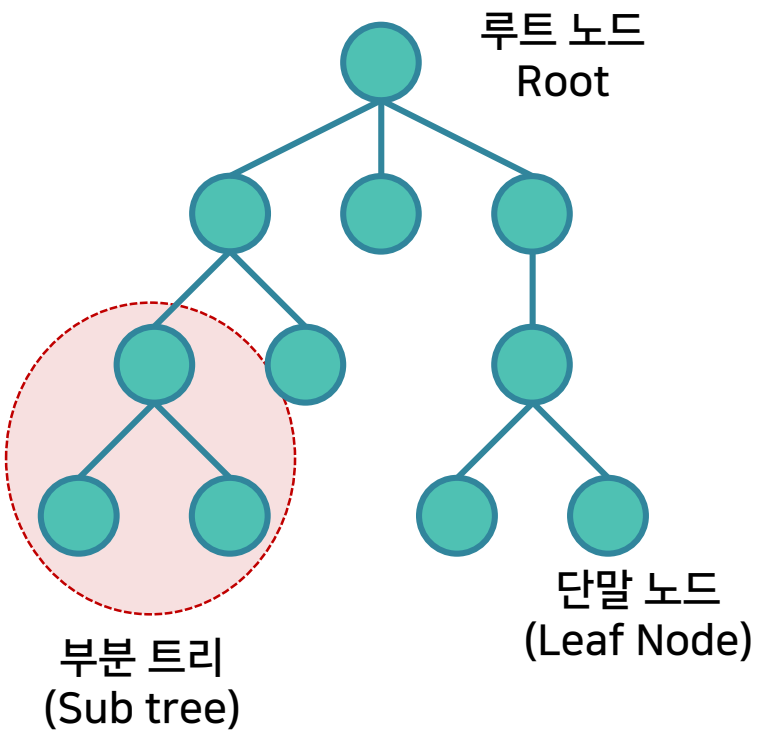
01 실무에서 많이 사용되는 자료 구조

02 루트(Root)라고 하는 유일한 시작 노드를 가짐

03 각 노드는 여러 개의 자식 노드를 가지는 구조

04 루트에서 각 노드까지의 경로가 유일한 구조

02 트리 구조



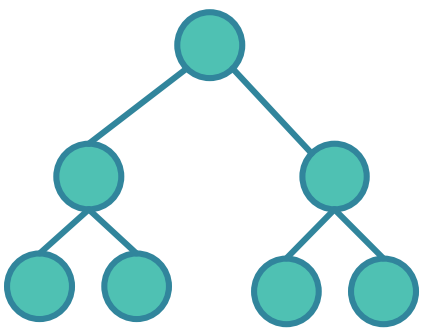
색인의 내부 구조

B-tree

03 트리의 종류

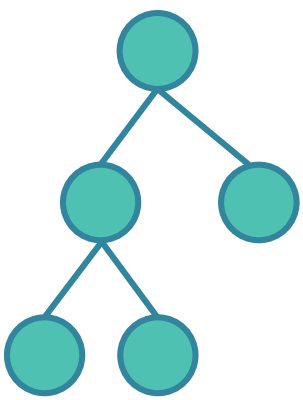
포화 트리 (Full tree)

- 모든 단말 노드가 같은 레벨에 존재
- 모든 중간 노드는 미리 정의된 최대한의 자식 노드를 가짐



완전 트리 (Complete tree)

- 마지막 레벨을 제외하면 포화 트리임
- 최하위 레벨의 단말 노드들은 왼쪽에서부터 채워진 형태로 되어 있는 경우



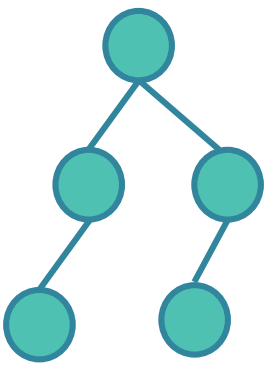
색인의 내부 구조

B-tree

03 트리의 종류

균형 트리 (Balanced tree)

- 모든 단말 노드가 같은 레벨에 있는 트리

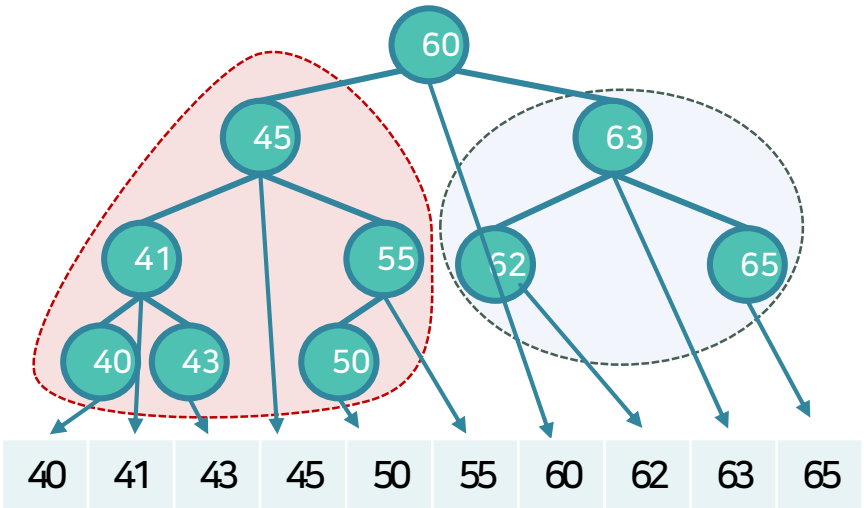


색인의 내부 구조

B-tree

04 2원 검색 트리

- 자식이 최대 2개인 트리이며, 키값은 하나임
  - 각 노드의 키값이 왼쪽 자손 노드들의 키값보다는 항상 크고, 오른쪽 자손 노드들의 키값보다는 항상 작은 이진 트리를 말함



- 노드의 구조

색인의 내부 구조

B-tree

04 2원 검색 트리

- B-tree
  - 2원 검색 트리의 일반화로 2원 트리는 노드의 크기가 작음
  - 한 노드에 많은 정보를 넣음 → m-원 트리

- 균형 m-원 트리
  - 모든 단말 노드는 같은 레벨에 위치
  - 자식을 최대 m개 가질 수 있음

05 차수가 m인 B-tree의 특성

- 루트와 리프를 제외한 노드의 서브트리 수 ≥ 2
  - $m/2 \leq$ 개수 $\leq m$

- 키값의 수
  - 리프: $m/2 - 1 \sim (m-1)$
  - 리프가 아닌 노드: 서브트리수 - 1

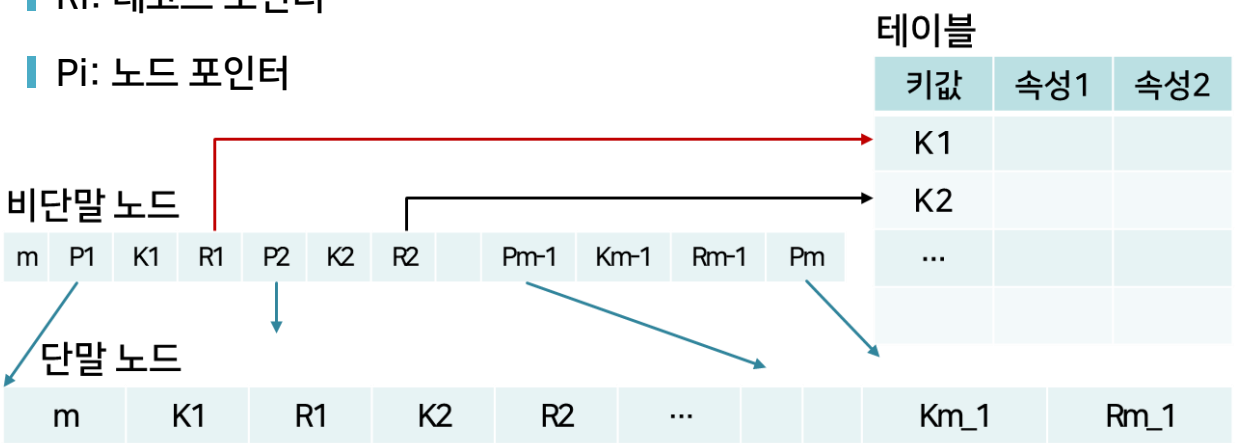
- 한 노드 내의 키값
  - 오름차순

색인의 내부 구조

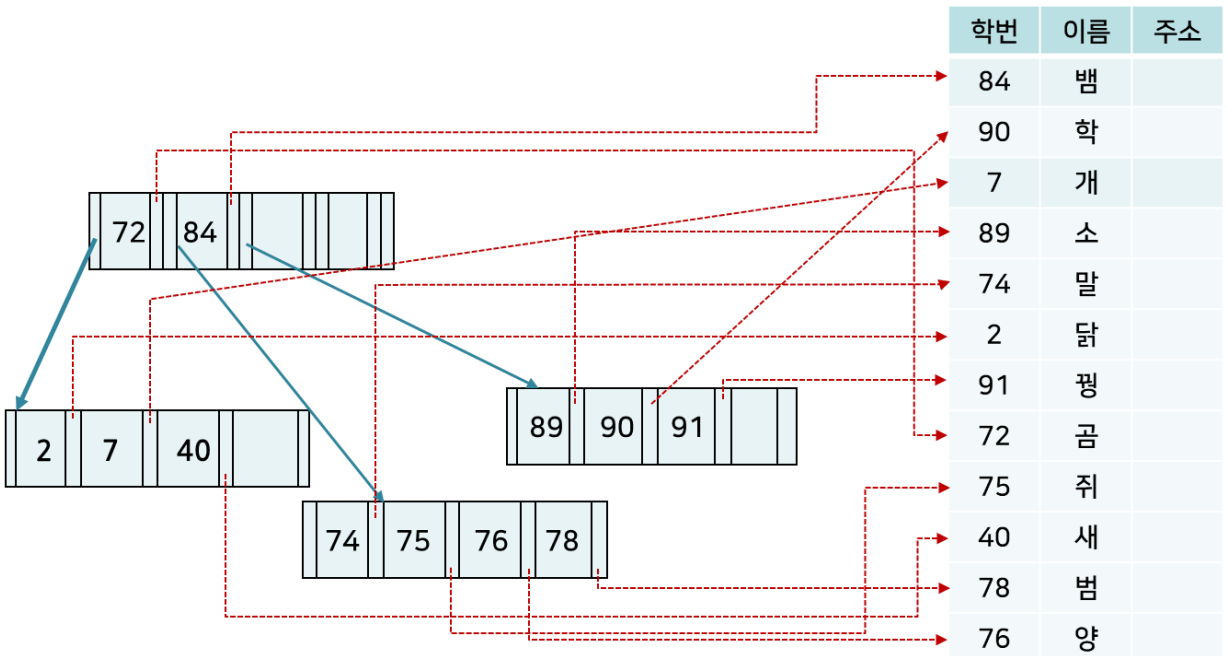
B-tree

06 물리적 구조가 다른 단말 노드와 비단말 노드

- Ki: 키값
- Ri: 레코드 포인터
- Pi: 노드 포인터



07 B-tree의 예(m= 5)



색인의 내부 구조

B-tree

08 B-tree 연산

단일키 검색

- 단일키값의 크기 비교를 통한 직접 탐색

  예 | 학번이 7번인 학생 정보를 검색하시오.

영역 검색

- 특정 범위에 있는 키값들 검색(순차 탐색)

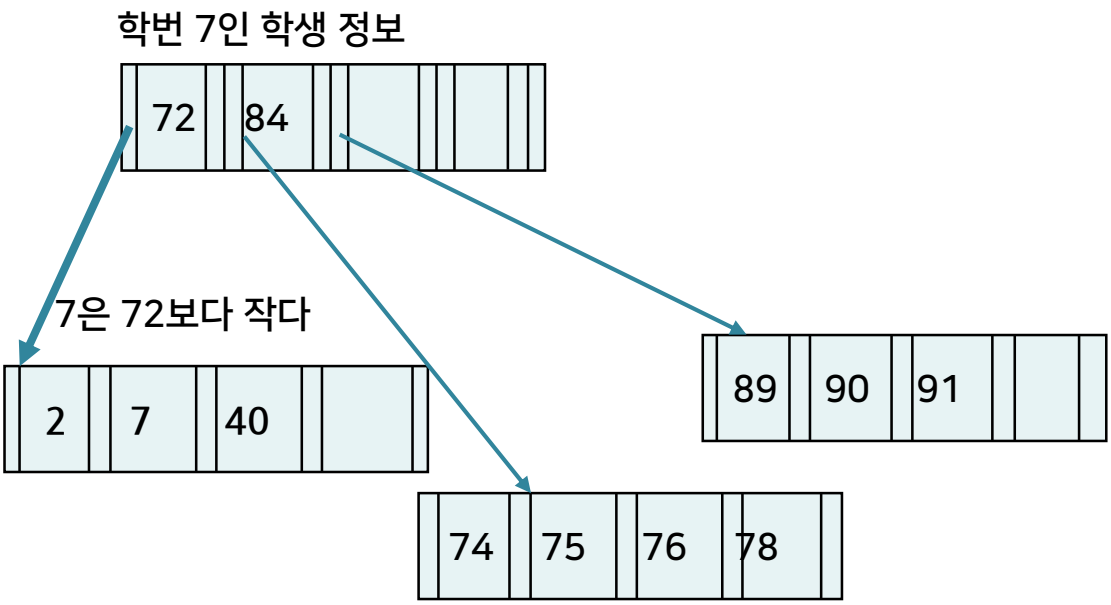
  예 | 학번이 40 이상인 큰 학생들의 정보를 검색하시오.

- 중위 순회(In-order Traversal) 시행
  - 중위 순회: 왼쪽 자식- 부모- 오른쪽 자식

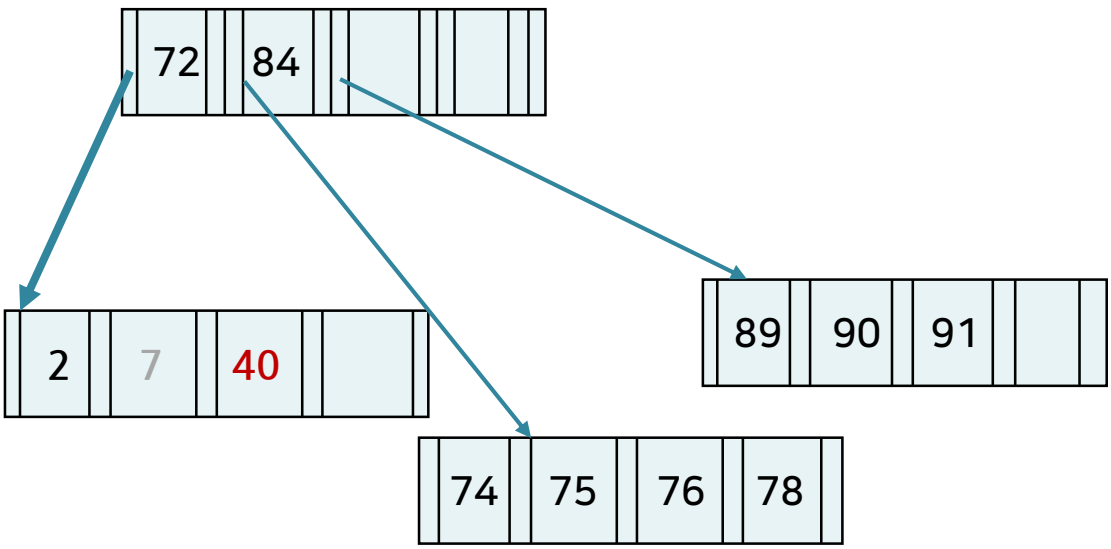
색인의 내부 구조

B-tree

09 B-tree연산(탐색)



학번 40번 이상인 학생들: 40부터 중위 탐색

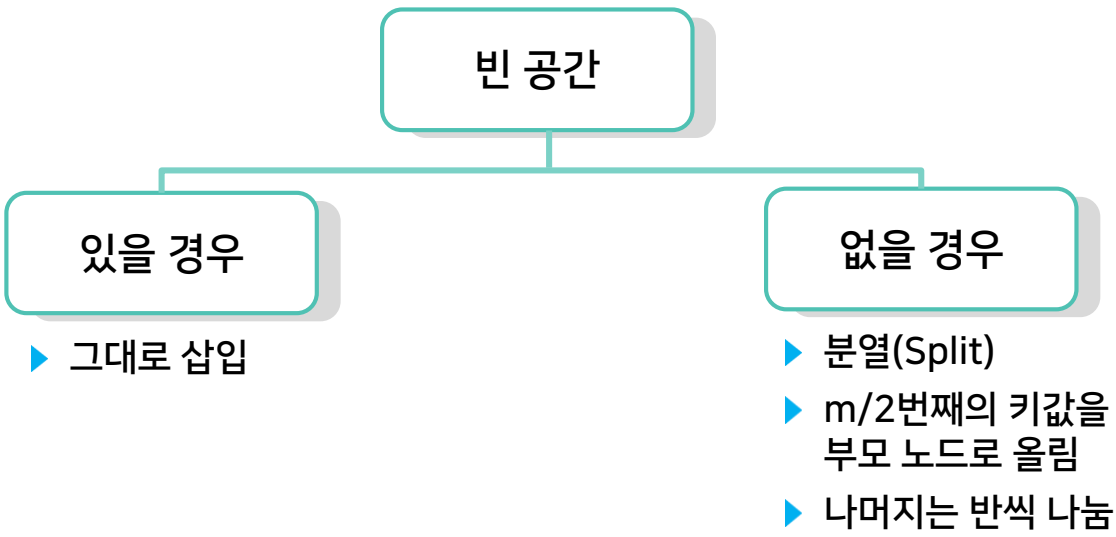


색인의 내부 구조

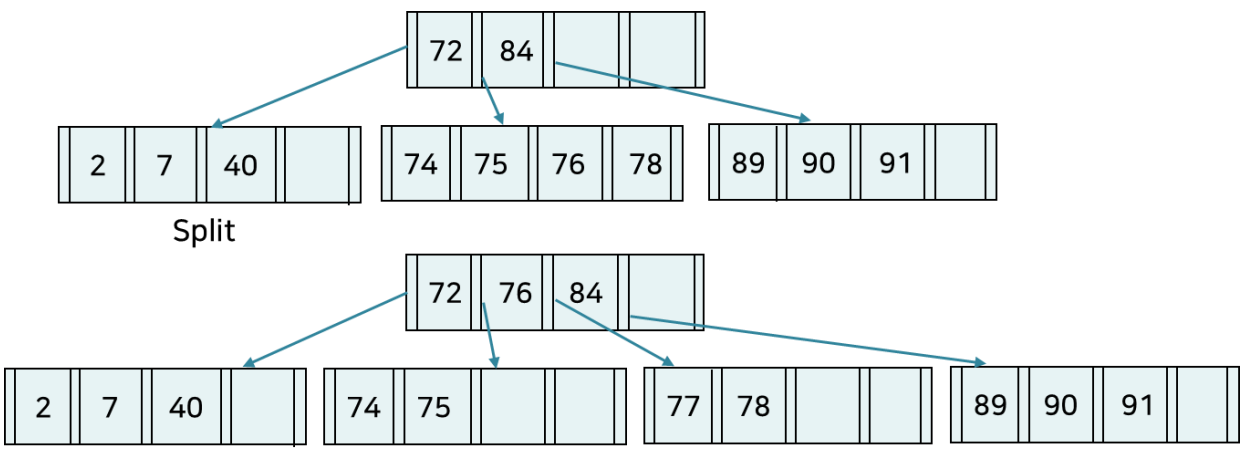
B-tree

10 B-tree연산(삽입)

- 새로운 키값의 삽입은 항상 단말 노드에서 일어남



- Insert 77



색인의 내부 구조

B-tree

11 B-tree연산(삭제)

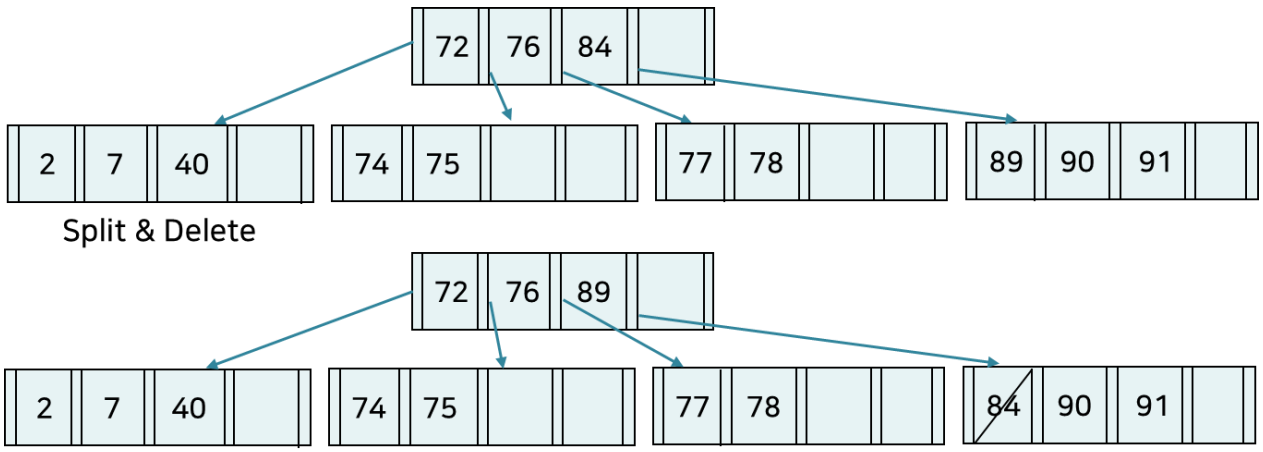
- B-tree에서의 삭제 또한 항상 단말노드에서 발생
- 삭제키가 비단말 노드에 있는 경우
  - 삭제키값보다 큰 키값들 중 가장 작은 키값(후행키값)과 자리 교환
  - 후행키값은 항상 단말 노드에 있음
- 키값 삭제 후 언더플로우 발생 시
  - 키의 수 < ⌈m/2⌉-1
  - 재분배(부유한 형제한테서 가지고 오기)
    - 최소키 수보다 많은 키를 가지고 있는 형제가 있다면 가지고 오기
    - 형제키값은 부모로, 부모 키값은 자신에게로 전달
  - 합병(재분배 불가능 시: 형제들도 다 가난함)
    - 서로 같이 살기

색인의 내부 구조

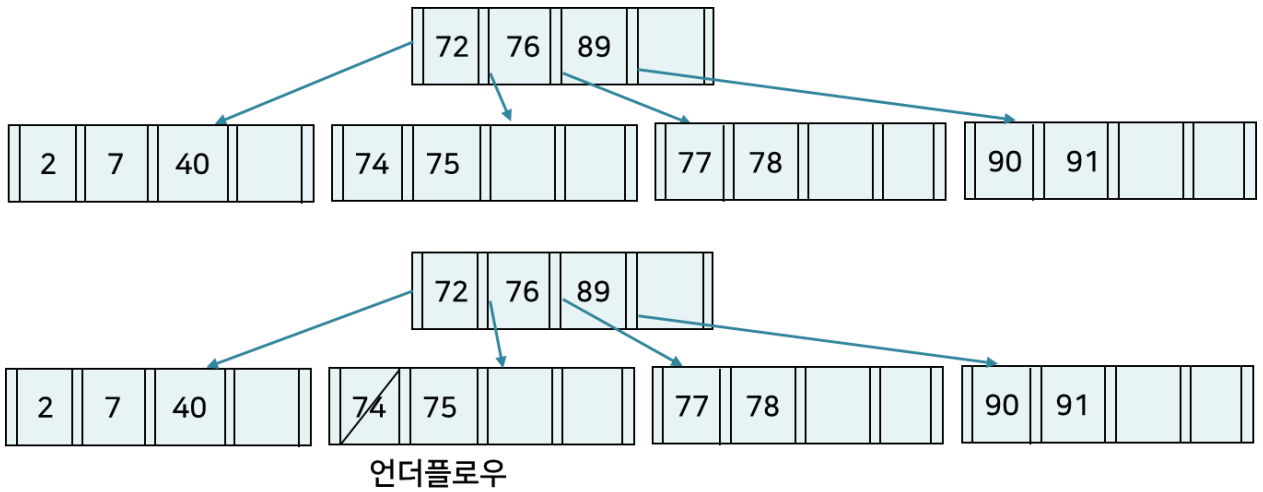
B-tree

11 B-tree연산(삭제)

Delete 84



Delete 74

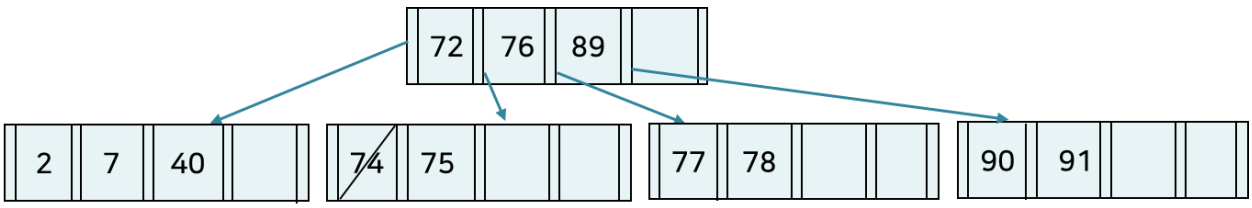


색인의 내부 구조

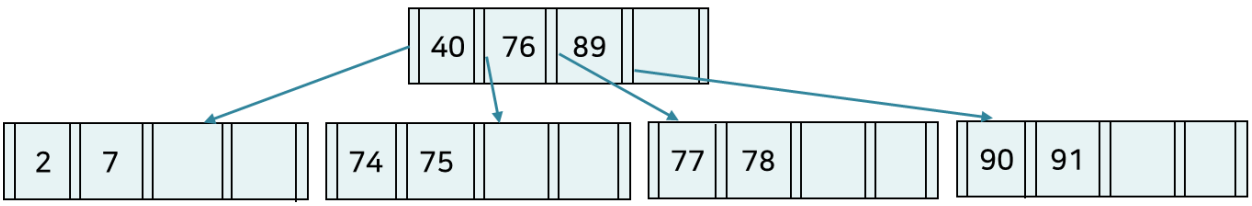
B-tree

11 B-tree연산(삭제)

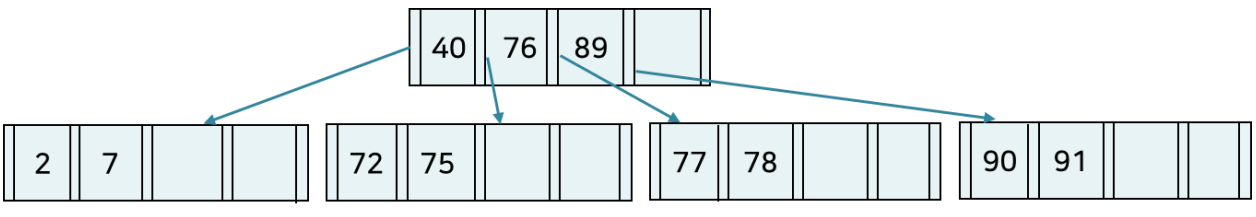
Delete 84

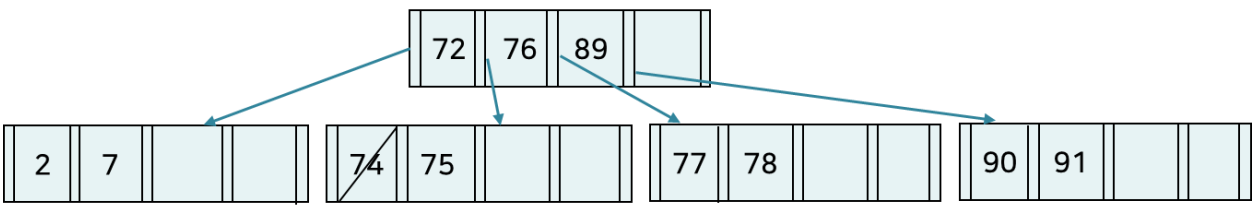


{2, 7, 40, 72,75}를 재분배, 이때[m/2]번째 값(즉, 40)이 기준이 됨



Delete 40





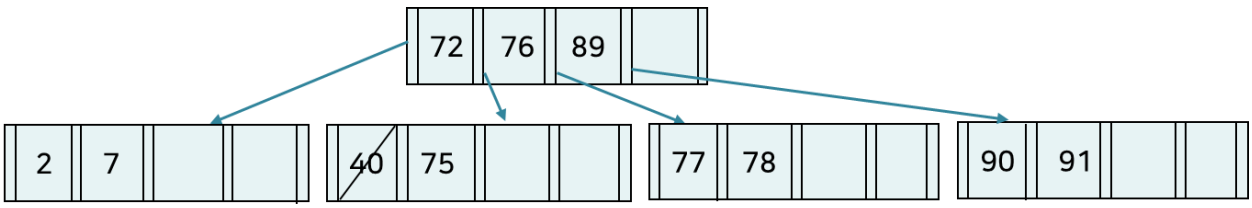
Swap
Redistribution 불가

색인의 내부 구조

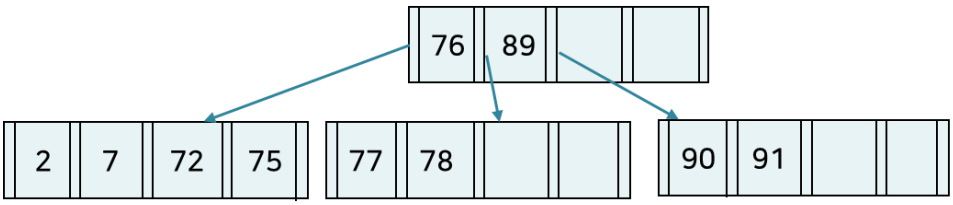
B-tree

11 B-tree연산(삭제)

Delete 40



Merge with left sibling and parent



색인의 내부 구조

B+tree

01 B-tree의 문제점

- 단말 노드와 비단말 노드의 구조가 다름
  - 저장 공간의 낭비: 단말 노드의 경우 1/3의 공간을 사용하지 않음
  - 단말 노드 ↔ 비단말 노드 변환 시 비용이 많이 듦

단말 노드와 비단말 노드의 물리적 구조를 같이 하자.

비단말 노드

m	P1	K1	R1	P2	K2	R2		Pm-1	Km-1	Rm-1	Pm

m	K1	R1	K2	R2	···			Km_1	Rm_1	

단말 노드

색인의 내부 구조

B+tree

02 구조

m	P0	K1	R1	P2	K2	R2	···			Km_1	Rm-1

- 비단말 노드: Ri는 노드 포인터
- 단말 노드: Ri는 레코드 포인터

2단계 구조

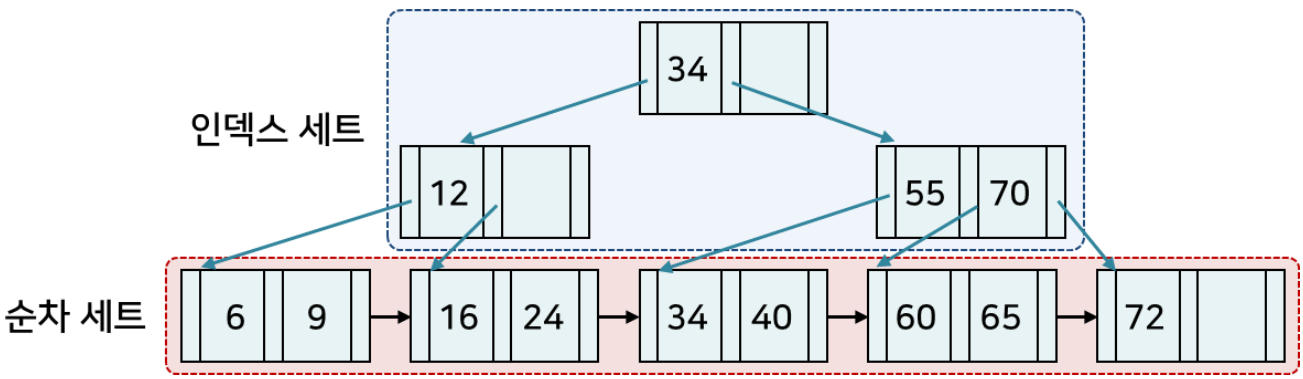
- 인덱스 세트와 순차 세트

순차 세트

- 단말 노드들로 구성
- 테이블에 있는 모든 키값들을 포함
- 키값 순서대로 연결되어 있음

인덱스 세트

- 비단말 노드들로 구성
- 단말 노드에 있는 키값들에 대한 접근 경로 제공
- 인덱스 세트에 있는 키값은 테이블에 존재하지 않을 수도 있음



색인의 내부 구조

B+tree

03 특징

- B-tree와 유사함
  - 루트 노드와 단말 노드를 제외한 노드들의 자식
  - 노드 수: m/2 ~ m
  - 루트 노드의 자식의 수: 0, 2 ~ m
  - 단말 노드가 아닌 노드의 키값의 수: 자식 노드의 수 -1
  - 단말 노드: Linked List 형태

04 검색

- 인덱스 세트에서는 m-원 탐색트리처럼 검색해서 단말 노드로 접근
- 단말 노드에서는 순차 탐색(Linear Search)

05 삽입

- B-tree와 유사, 단말 노드에서 발생
- 넣을 공간이 없을 시 분열

색인의 내부 구조

B+tree

06 삭제

- 언더플로우가 없으면 단말 노드에서만 삭제
  - 키값이 인덱스 세트에 남아 있을 수 있음
- 언더플로우 발생시
  - 재분배: 인텍스 세트의 키값 변화, 트리 구조 유지
  - 합병: 인덱스 세트에 있는 키값도 삭제

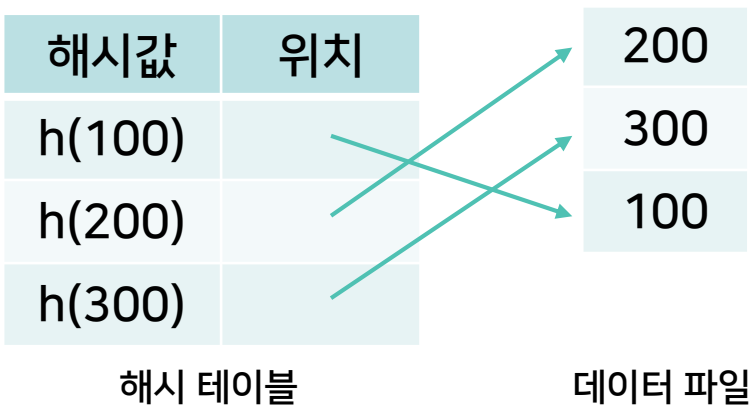
색인의 내부 구조

해싱

01 트리 기반 색인 기법

02 해싱 기법

- 계산에 의해서 얻은 주소에 데이터를 저장
- 자료 검색 시 키값을 비교하는 것이 아니라 키값을 이용하여 직접 접근 (해시 함수를 사용)



색인의 내부 구조

해싱

03 충돌(Collision)

- 2개 이상의 키가 같은 해시값을 갖는 경우 충돌이 발생

충돌방지를 하기 위해서는 어떤 방법이 있을까?

오픈 해싱(Open Hashing)

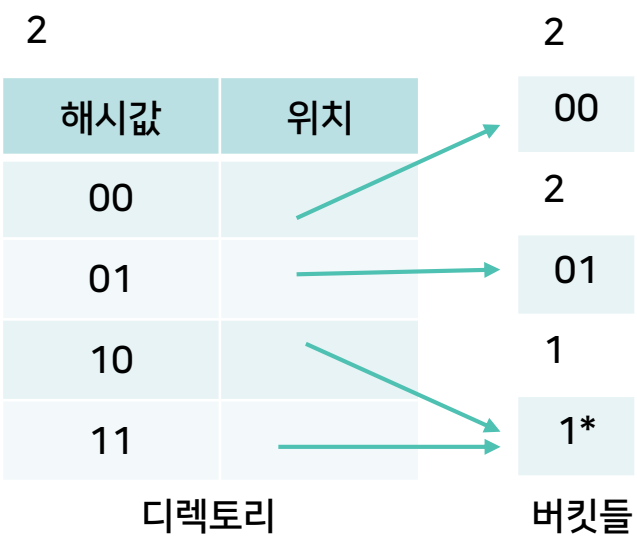
- 같은 해시값을 갖는 키들을 바구니에 모아 놓음
- 주로 바구니는 연결된 리스트(Linked List)로 구현함
- 나중에 바구니를 검색할 때는 순차 검색으로 함

색인의 내부 구조

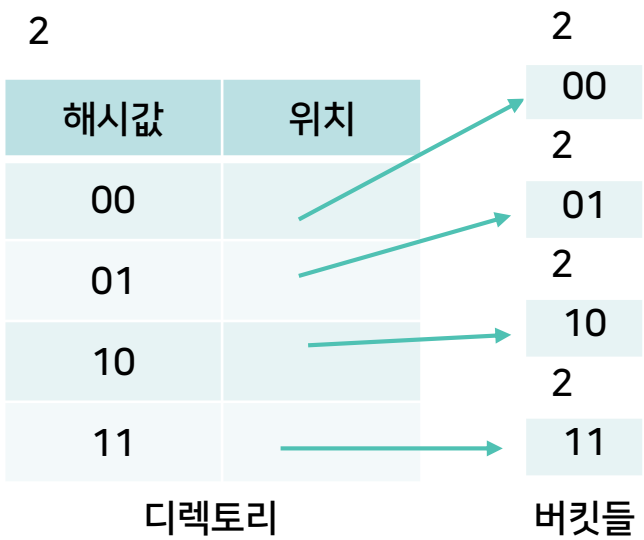
해싱

04 확장성 해싱(Extendable Hashing)

- 충돌 문제 해결 기법 중 하나
- 디렉토리와 버킷
  - 키값에서 일정 길이 부분의 비트를 해시값으로 사용



- 10으로 시작하는 레코들이 많아짐
  - 버킷을 쪼갬

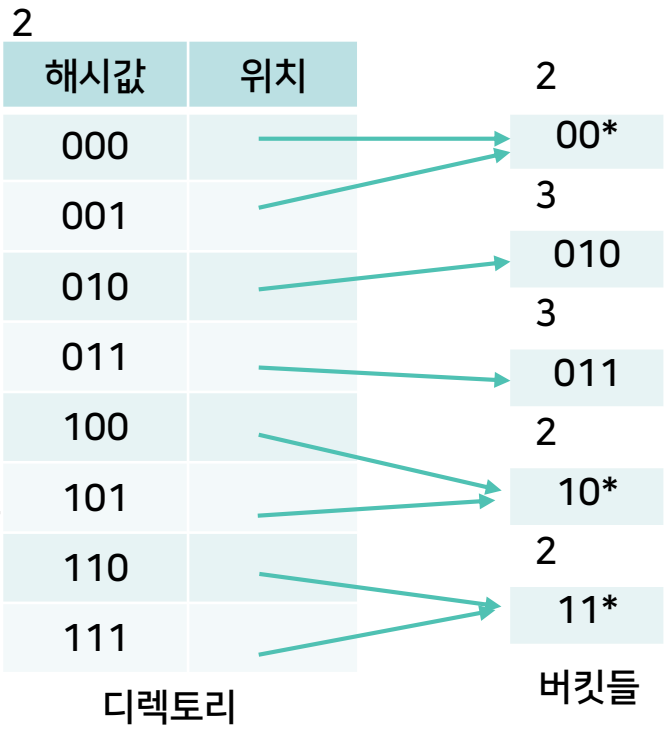


색인의 내부 구조

해싱

04 확장성 해싱(Extendable Hashing)

- 01로 시작하는 레코드들이 많아짐
  - 현재 디렉토리는 2비트만을 봄
  - 3비트를 보자
- 해싱 기법은 영역 탐색을 지원하지 못함
  - x < y → h(x) < h(y)이지 않음



1 색인의 필요성

- ✓ 색인: 검색 성능을 향상 시키기 위한 부가적인 자료 구조
- ✓ 색인 생성

```
CREATE INDEX 색인명
ON 테이블명(속성명, 속성명,…)
```

- ✓ 색인의 종류
  - 고유 색인: 유일 값을 갖는 칼럼에 대해 생성되는 인덱스로 모든 인덱스 키는 테이블 하나의 행과 연결
  - 비고유 색인: 중복 값을 갖는 칼럼에 대해 생성되는 인덱스로 하나의 인덱스 키는 여려 행과 연결 가능
  - 단일 색인: 하나의 칼럼에 대하여 구성된 색인
  - 결합 색인: 두 개 이상의 칼럼에 결합하여 생성되는 색인

2 색인의 내부 구조

✓ B-tree
- 2원 검색 트리의 일반화 형태
- 균형 m원 트리: 모든 단말 노드의 레벨은 같으며 자식은 최대 m개를 가짐

✓ B+tree
- 인덱스 세트
  비단말 노드들로 구성, 단말 노드에 있는 키값들에 대한 접근 경로 제공
- 순차 세트
  단말 노드들로 구성, 테이블에 있는 모든 키값을 포함하며 키값 순서대로 연결되어 있음

✓ 해싱: 키값을 이용 직접 레코드 접근