DB 인덱스

인덱스의 기본 개념과 자료구조

데이터베이스는 어떻게 자료를 찾을까?

> DB는 실제 하드디스크에 저장되는데, 파일 기반 접근은 메모리 기반 접근보다 시간이 매우 오래 걸립니다.

- DB에서는 Insert된 자료들을 어떻게 효율적으로 찾을 수 있을까요?

```sql
-- 이메일 기반 사용자 조회
SELECT * FROM user_account
WHERE email = 'test@example.com';

-- ID가 100번부터 200번 사이인 사용자 조회
SELECT * FROM user_account
WHERE id BETWEEN 100 AND 200;
```

- 자료가 100만개라면 SELECT 쿼리 수행 시 모든 행을 전부 찾아봐야 할까요?

> 인덱스를 통해 데이터의 위치를 찾고, 최소한의 파일 접근을 통해 DB에서 데이터를 조회합니다.

인덱스(Index)

- 책의 목차와 비슷한 역할
- 데이터를 정렬된 상태로 유지해서 O(log N)의 시간 복잡도로 탐색 가능하게 함

인덱스 예시 코드

```sql
-- 회원 정보
CREATE TABLE user_account (
    id BIGINT UNSIGNED NOT NULL AUTO_INCREMENT,   -- PK (AUTO_INCREMENT)
    username VARCHAR(50) NOT NULL,                -- 사용자 이름
    email VARCHAR(100) NOT NULL,                  -- 이메일
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    PRIMARY KEY (id)
);

-- 이메일 컬럼 인덱스
CREATE INDEX idx_user_email ON user_account(email);
```

인덱스 자료구조

B-Tree

- 모든 페이지는 m개의 자손을 가집니다.
- Root와 leaf 노드를 제외한 모든 페이지는 최소 m/2개의 자손을 가져야 합니다.
- 모든 노드는 같은 레벨에 존재해야 합니다.
- 데이터를 많이 저장해도 트리의 높이를 매우 낮게 유지할 수 있습니다.

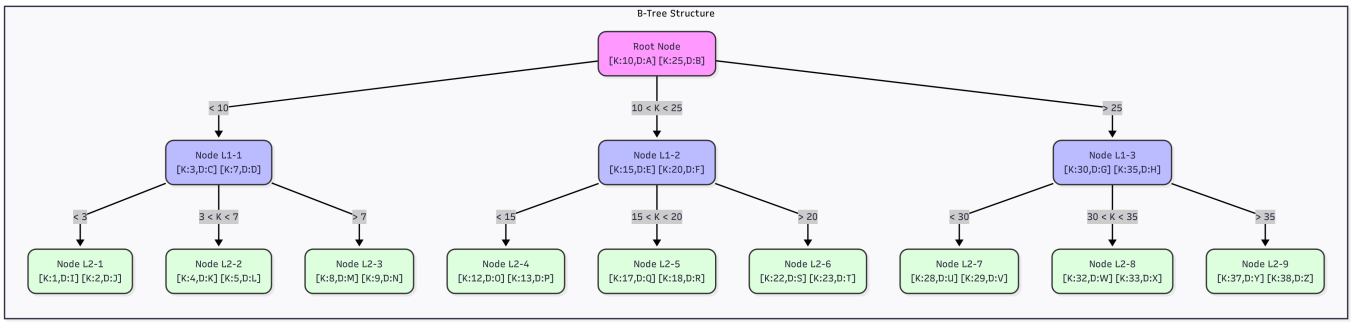
> N개의 키가 있을 때, 트리의 높이 d는 다음과 같습니다

$$N \geq 2 \cdot \lceil m/2 \rceil^{d-1}$$

$$d \leq 1 + \log_{\lceil m/2 \rceil}(\frac{N}{2})$$

$$N \ge 2 \cdot \lceil m/2 \rceil^{d-1}$$

$$d \le 1 + \log_{\lceil m/2 \rceil}(\frac{N}{2})$$



만약 차수(m)이 512이고, 키(N)이 100만 개 인 상황에서도 트리의 높이는 4 이하가 됩니다.

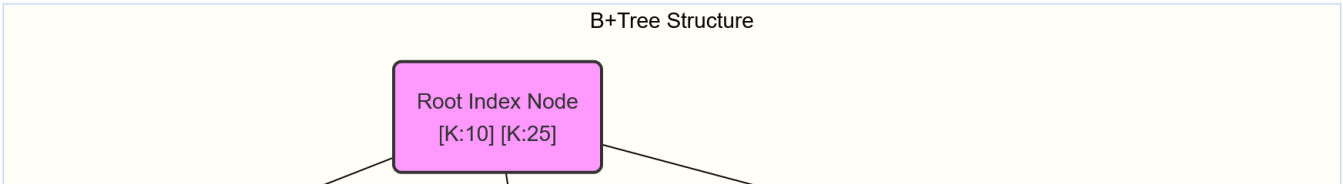
즉 100만개의 데이터가 저장된 상황에서도 4번의 디스크 접근으로 원하는 값을 찾을 수 있습니다.

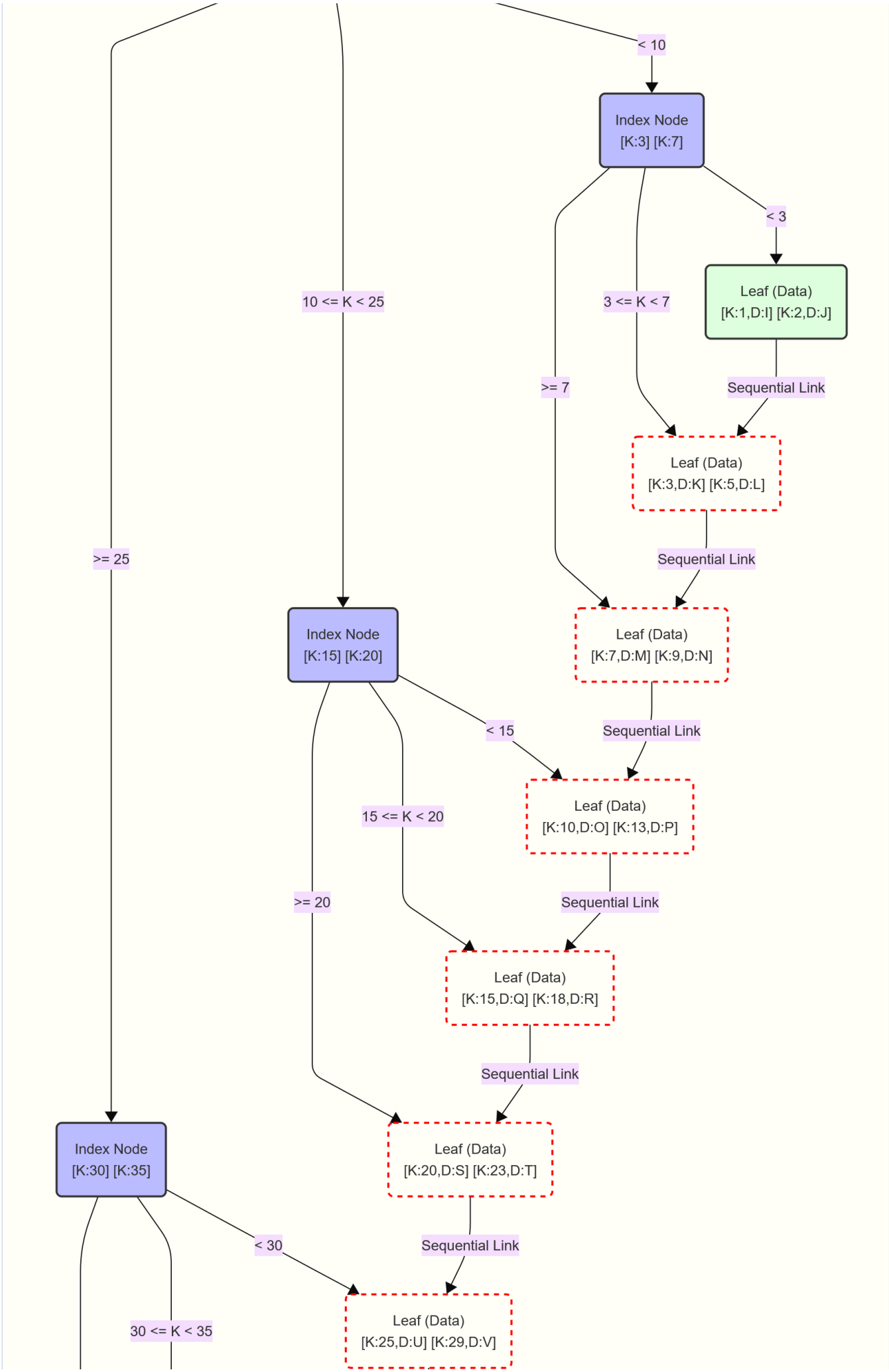
B+Tree

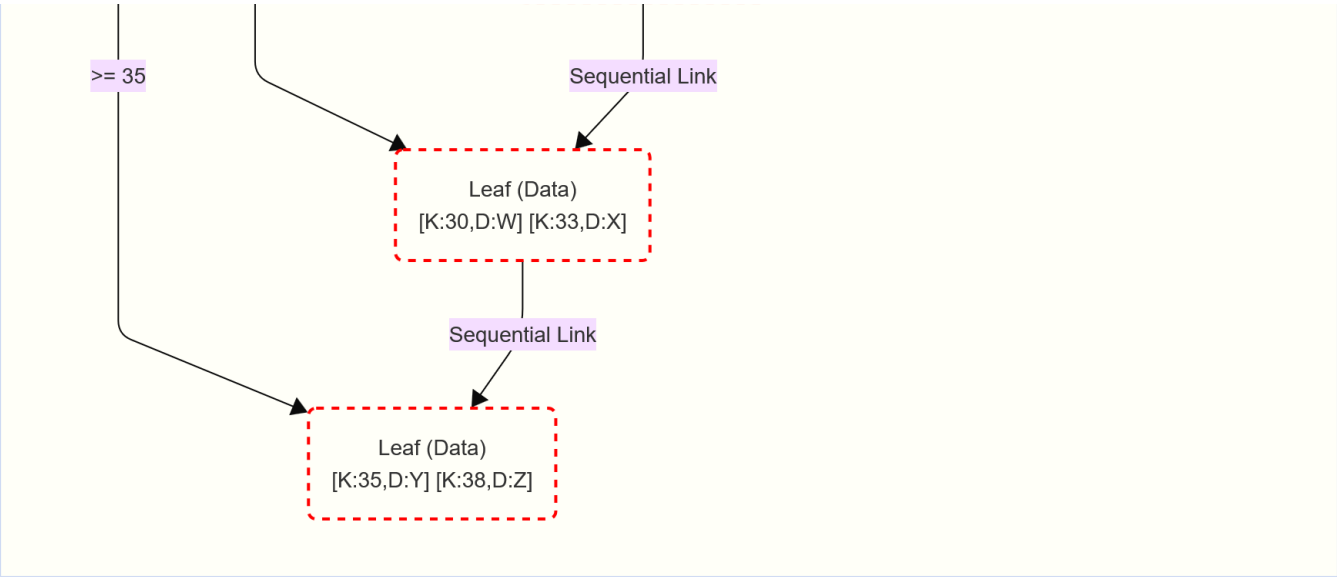
> B-Tree에서 몇 가지 규칙이 추가된 형태입니다.

- Internal 노드와 Leaf(말단) 노드로 노드의 역할이 구분됩니다.
- 말단인 Leaf 노드에만 데이터 레코드를 가리키는 데이터가 저장됩니다.
- 리프 노드들은 서로 Linked-List로 연결되어 있습니다. 바로 옆 데이터를 조회할 수 있어서 범위 검색 시 유리합니다.

```sql
-- ID가 100번부터 200번 사이인 사용자 조회
SELECT * FROM user_account
WHERE id BETWEEN 100 AND 200;
```




< 10
Index Node
[K:3] [K:7]
< 3
Leaf (Data)
[K:1,D:I] [K:2,D:J]
10 <= K < 25
3 <= K < 7
>= 7
Sequential Link
Leaf (Data)
[K:3,D:K] [K:5,D:L]
>= 25
Sequential Link
Index Node
[K:15] [K:20]
Leaf (Data)
[K:7,D:M] [K:9,D:N]
< 15
Sequential Link
Leaf (Data)
[K:10,D:O] [K:13,D:P]
15 <= K < 20
>= 20
Sequential Link
Leaf (Data)
[K:15,D:Q] [K:18,D:R]
Sequential Link
Index Node
[K:30] [K:35]
Leaf (Data)
[K:20,D:S] [K:23,D:T]
< 30
Sequential Link
30 <= K < 35
Leaf (Data)
[K:25,D:U] [K:29,D:V]



B-Tree, B+Tree 특징 비교

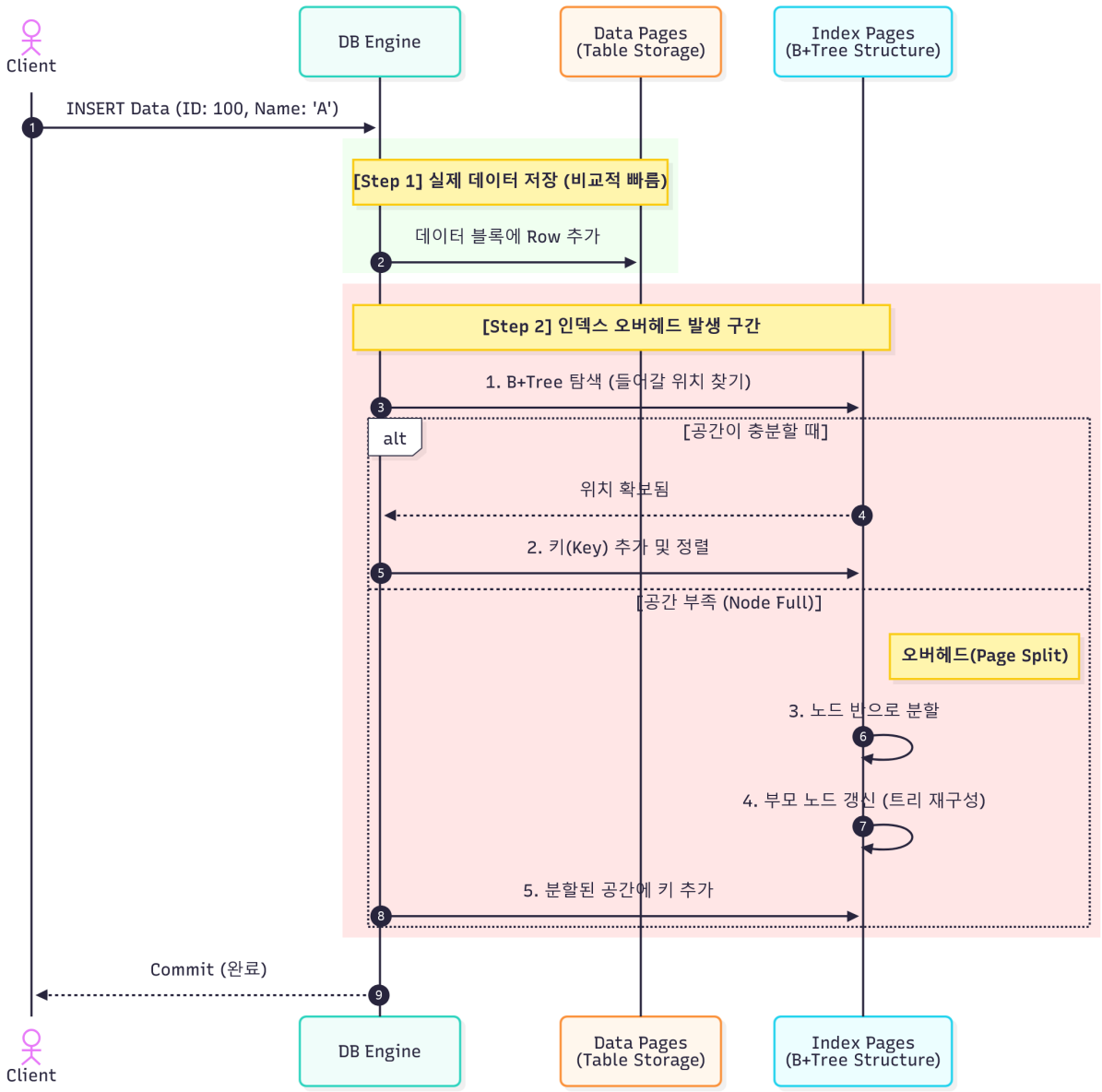
비교 항목	B-Tree	B+Tree
데이터 저장 위치	모든 노드 (내부 + 리프)에 데이터 포인터 존재	리프 노드에만 데이터 포인터 존재
순차 검색 (Range Scan)	트리를 오르락내리락하며 순회해야 함 (비효율적)	리프 노드의 Linked List를 통해 선형 탐색 가능 (효율적)
검색 속도	루트 노드 ~ 리프 노드 범위에 데이터 포인터가 존재함	무조건 리프 노드까지 내려가야 함 (But 경로가 매우 짧음)

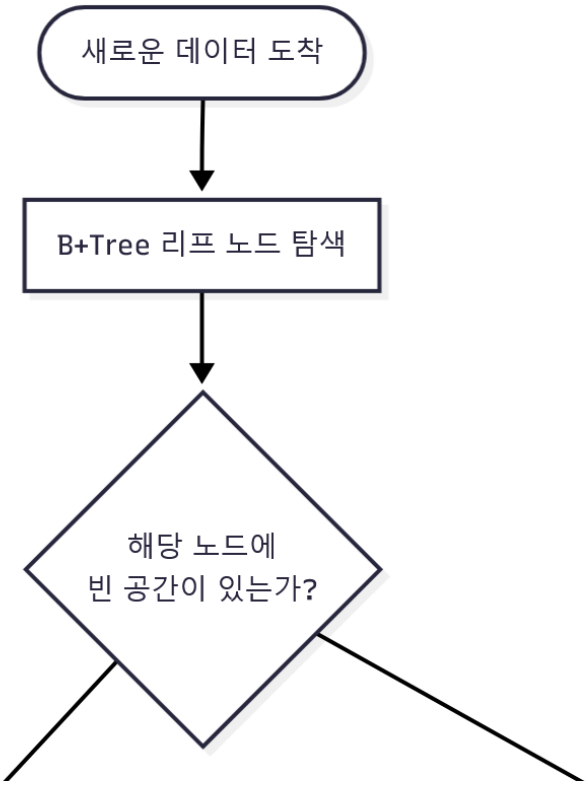
> 일반적으로 데이터베이스의 인덱스는 B+Tree를 통해 구현합니다.

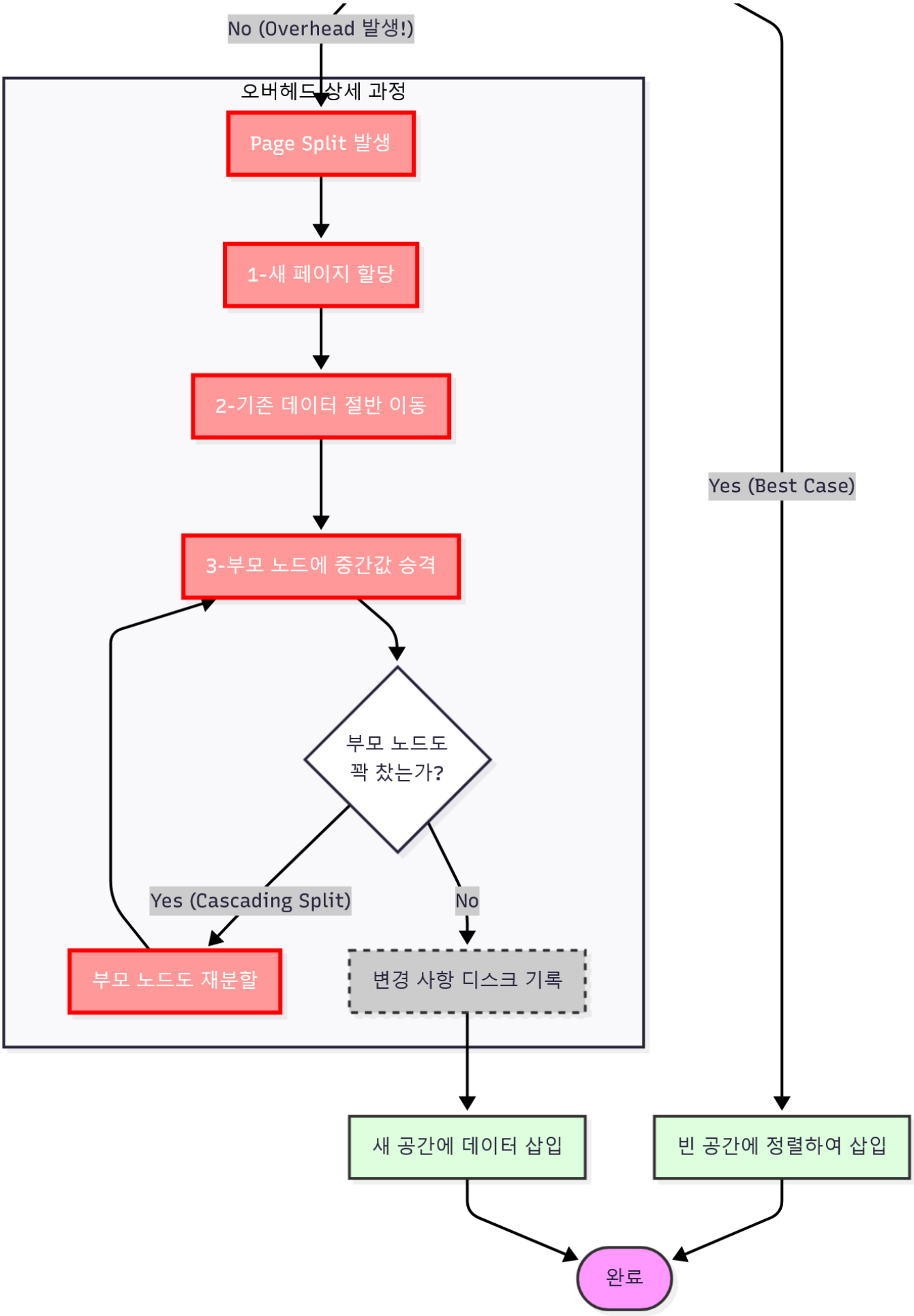
인덱스를 설정하면 좋은 컬럼

- SELECT 문 수행 시 기준 조건으로 자주 사용되는 컬럼
- 중복이 적은 컬럼 (c.f. 나쁜 예시 : 성별)

인덱스의 단점


Client
DB Engine
Data Pages (Table Storage)
Index Pages (B+Tree Structure)
INSERT Data (ID: 100, Name: 'A')
[Step 1] 실제 데이터 저장 (비교적 빠름)
데이터 블록에 Row 추가
[Step 2] 인덱스 오버헤드 발생 구간
1. B+Tree 탐색 (들어갈 위치 찾기)
alt
[공간이 충분할 때]
위치 확보됨
2. 키(Key) 추가 및 정렬
[공간 부족 (Node Full)]
오버헤드(Page Split)
3. 노드 반으로 분할
4. 부모 노드 갱신 (트리 재구성)
5. 분할된 공간에 키 추가
Commit (완료)
DB Engine
Data Pages (Table Storage)
Index Pages (B+Tree Structure)
Client


새로운 데이터 도착
B+Tree 리프 노드 탐색
해당 노드에
빈 공간이 있는가?



- 삽입, 삭제 시 DB 행(레코드) 외 Index 기반 B+Tree도 함께 업데이트해야 합니다.