Indexing

상태	진행 중
분류	Database
주차	2주차

📌 목차

Index

인덱스란?

- 데이터베이스에서 테이블에 대한 검색 속도를 높여주는 자료구조
- 메모리 영역에서 일종의 목차를 생성하는 개념
- 이런 목차를 이용해 검색 범위를 줄여 속도를 높일 수 있음
- 데이터 - ROWID(주소) 쌍으로 저장됨 (Oracle)

인덱스가 없을 때

- 일반적인 SELECT 쿼리 실행 시 먼저 메모리의 database buffer cache를 체크
- buffer cache에는 자주 사용되는 테이블이 캐싱되어 있어 cache에 있을 경우 바로 데이터를 찾아 출력
- 그렇지 않으면 물리적인 저장장치에 존재하는 데이터 파일에서 데이터를 찾음
- 인덱스를 사용하면 위 과정을 거치지 않고 바로 주소를 통해 찾아감

인덱스가 존재할 때

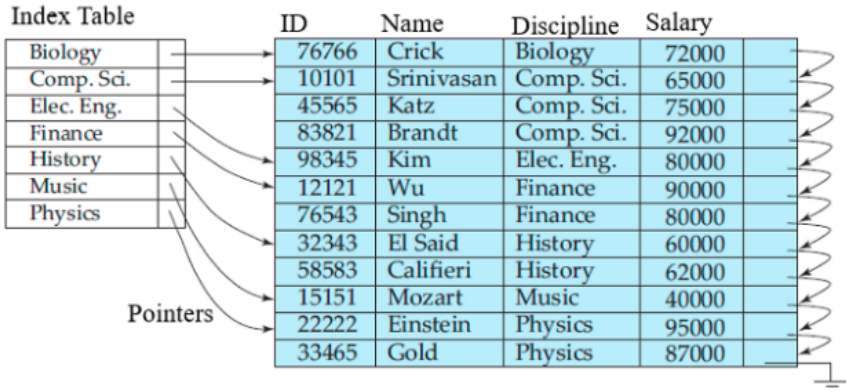
- server process가 구문 분석 과정을 마친 후 database buffer cache에 조건에 부합하는 데이터가 존재하는지 확인
- 해당 정보가 buffer cache에 없다면 디스크 파일에서 조건에 부합하는 블럭을 찾아 database buffer cache에 가져온 뒤 사용자에게 보여줌
    - 위 과정에서 인덱스가 있으면 WHERE절 조건의 컬럼이 인덱스의 키로 생성되어있는지 확인
    - 그 후 인덱스에 먼저 가서 조건에 부합하는 정보가 어떤 ROWID를 갖고 있는지 확인
    - ROWID에 있는 블럭을 찾아가 해당 블럭만 buffer cache에 복사

인덱스를 사용하는 이유

- 속도
    - 데이터베이스 내 데이터가 많아질수록 원하는 데이터를 찾기 위한 시간 증가
    - 인덱싱은 이 검색 시간을 크게 감소시켜 줌
- 최적화된 쿼리 수행
    - 인덱싱은 쿼리 성능을 최적화 함
    - 이를 통해 전체 시스템 성능도 향상

인덱스 예시

- 일반적으로 테이블의 Physics 값을 조회한다 가정
  - SELECT 절을 활용하여 조회
  - 어느 위치에 데이터가 존재하는지 모르기 때문에 Table Full scan 진행



- 테이블의 전체 데이터를 조회하기 때문에 데이터의 수가 적은 테이블이면 영향이 덜할 수 있다
- 그렇지만 만약 수십만개의 데이터가 존재하는 테이블의 데이터를 Full scan으로 조회한다면 성능이 매우 감소할 수 있다
  - 따라서, 인덱스를 따로 생성해 해당 데이터만 빠르게 찾을 수 있게 한다

인덱스의 특징

인덱스 동작 과정

- 테이블 생성 시 생성하고 싶은 인덱스 컬럼을 지정
- 생성 후 인덱스 조회 시, WHERE절이 포함된 Query로 조회
- 인덱스로 저장된 Key-Value값을 참조해서 결과 출력

인덱스 장점

- 빠른 데이터 검색
- 인덱스를 통해 정렬된 순서로 데이터에 접근 가능

인덱스 단점

- 추가적인 저장 공간 필요
- 삽입, 삭제 오버헤드 존재
    - 인덱스가 있는 테이블에 데이터를 삽입하거나 삭제할 때 인덱스도 함께 업데이트 해주어야 함

인덱스를 사용하면 좋은 경우

- 규모가 큰 테이블
- INSERT, UPDATE, DELETE가 자주 발생하지 않는 컬럼
- JOIN, WHERE, ORDER BY에 자주 사용되는 컬럼
- 데이터 중복도가 낮은 컬럼

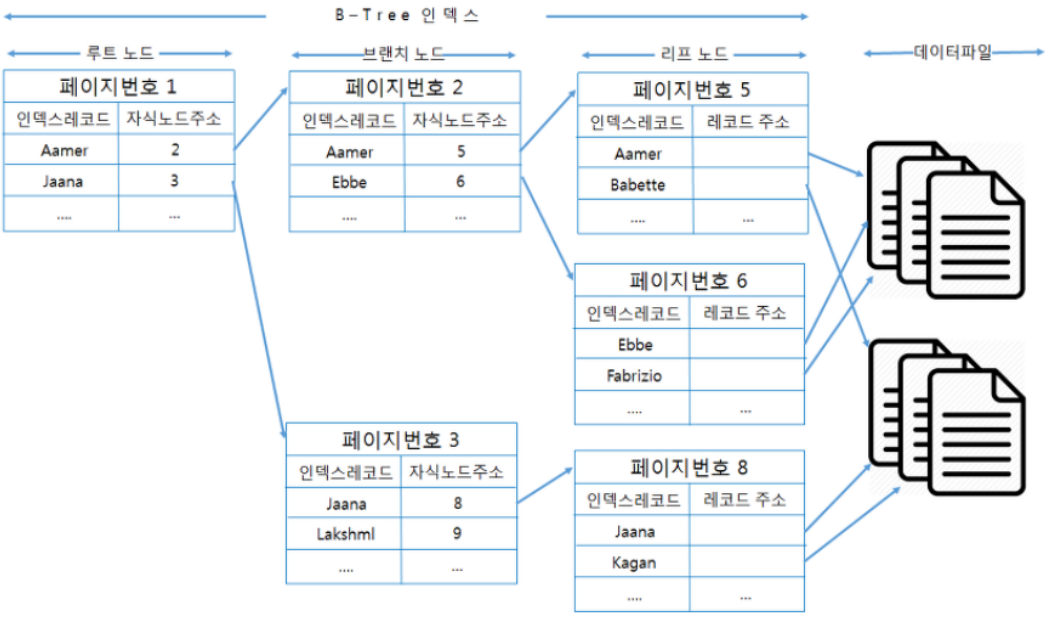
인덱스 사용이 불가능한 경우

- 인덱스 컬럼을 조건절에서 가공하면 인덱스 사용 불가능
- 부정형 비교는 인덱스 사용 불가능
- IS NULL 조건만으로는 인덱스 사용 불가능

인덱스 알고리즘 종류

- B-tree Index
- Hash Index
- Fractal-Tree Index

B-tree (Balanced Tree)



- 칼럼의 값을 변형하지 않고 원래의 값을 이용해 인덱싱하는 알고리즘
- 최상위에 하나의 루트 노드가 존재하고, 그 하위에 자식 노드가 붙어있음
- 가장 하위는 리프 노드라 하고, 루트 노드도 아니고 리프 노드도 아닌 중간 노드를 브랜치 노드라고 한다

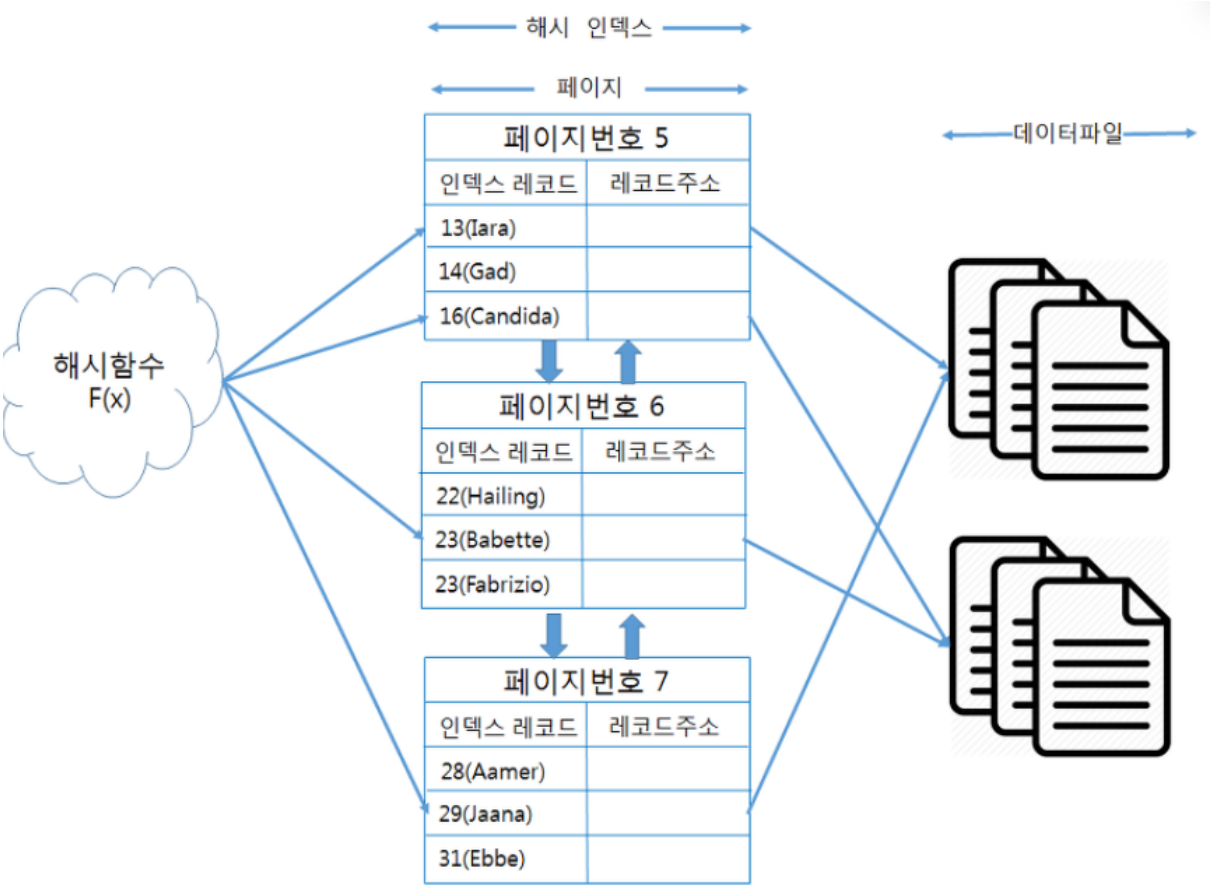
장점

- 어떤 데이터를 조회하든지 이에 사용하는 조회 과정의 길이 및 비용이 균등

단점

- 단순 조회로 데이터를 조회하는 과정 대비 조회 속도가 느리다

Hash Index



- 칼럼의 값으로 키 값을 계산해서 인덱싱
- 값을 변경해서 인덱싱하므로 동등 비교 검색에는 최적화 되어있지만. 범위를 검색하거나 정렬된 결과를 가져오는 목적으로는 사용할 수 없다
- 주로 인메모리 DB에서 사용하는 인덱스 종류
    - 인메모리 DB? : 메모리가 디스크 스토리지의 메인 메모리에 설치되어 운영되는 DB

장점

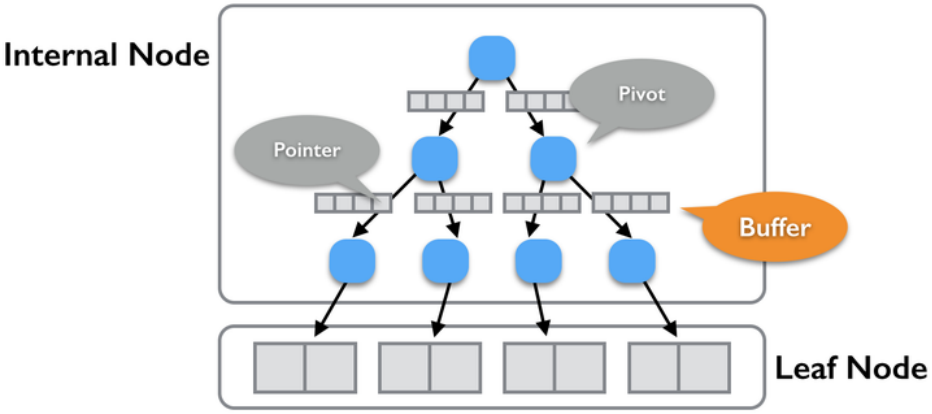
- 실제 키 값과는 관계 없이 함수의 결과만을 저장하므로 인덱스의 크기가 작다
- 타 인덱스 대비 조회 속도가 매우 빠르다

단점

- 범위로 조회하는 작업은 느리다
- 범위로 묶어서 보관하는 인덱스가 아니므로 데이터 갯수가 증가하면 더 큰 저장공간을 필요로 한다

Fractal-Tree 알고리즘 (TokuDB)

- B-Tree의 단점을 보완하기 위해 고안된 알고리즘
- 값을 변형하지 않고 인덱싱하며 범용적인 목적으로 사용할 수 있다는 측면에서 B-Tree와 거의 유사
- 그러나, 데이터가 저장되거나 삭제될 때 처리 비용을 상당히 줄임
- 추가로, 잦은 Disk I/O 시간을 줄이기 위해 고안



- 각 pivot에 Buffer 공간이 존재
- 데이터가 유입되면, B-Tree처럼 데이터를 바로 child node에 전달하지 않고 Buffer에 데이터가 가득 찬 순간 데이터를 child node에 전달
    - 따라서, Disk I/O가 발생하는 횟수를 줄임

인덱스 타입

- Primary(클러스터) 인덱스
    - 처음부터 정렬이 되어 있는 영어 사전과 같은 개념
- Secondary(보조) 인덱스
    - 책 뒤의 찾아보기의 개념과 유사

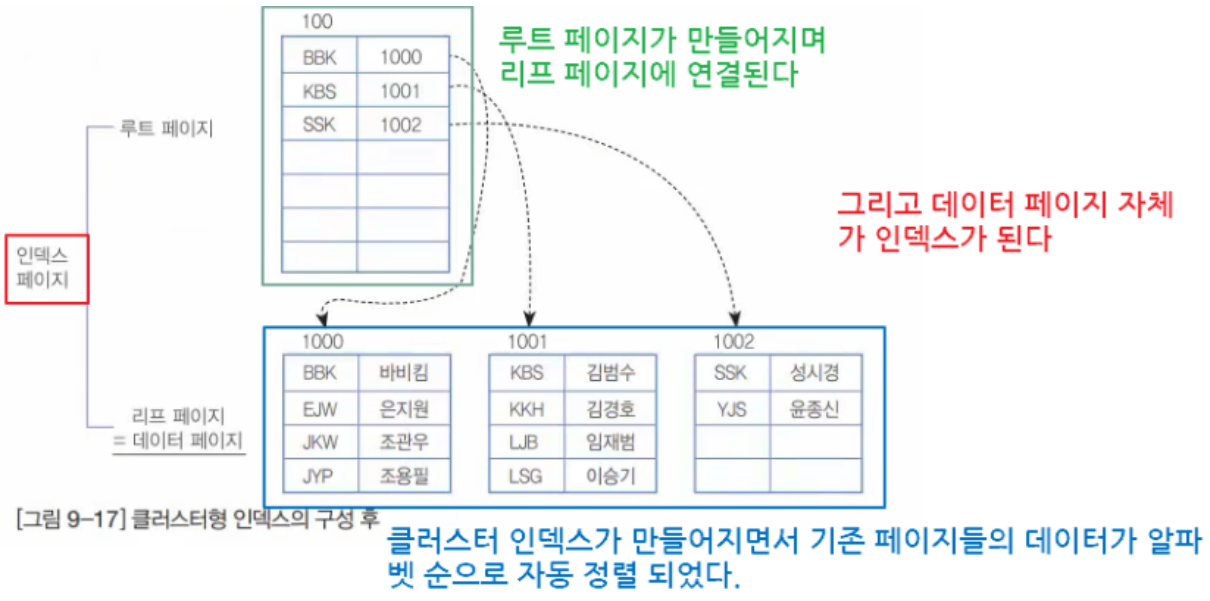
	클러스터 인덱스	보조 인덱스
속도	빠르다	느리다
사용 메모리	적다	많다
인덱스	인덱스가 주요 데이터	인덱스가 데이터의 사본
개수	한 테이블에 한 개	한 테이블에 여러 개

	클러스터 인덱스	보조 인덱스
리프 노드	리프 노드 자체가 데이터	리프 노드는 데이터가 저장되는 위치
저장값	데이터를 저장한 블록의 포인터	값과 데이터의 위치를 가리키는 포인터
정렬	인덱스 순서와 물리적 순서가 일치	인덱스 순서와 물리적 순서 불일치

클러스터 인덱스

- 나열된 데이터를 일정 기준으로 정렬
  - 따라서 클러스터형 인덱스 생성 시 데이터 페이지 전체가 다시 정렬 됨
  - 만약 대용량의 데이터가 이미 저장된 상태면 클러스터형 인덱스 생성시 심각한 시스템 부하를 줄 수 있음
- 한개의 테이블에 한개씩만 만들 수 있음
- 보조 인덱스보다 검색 속도는 더 빠르지만 입력, 수정, 삭제는 더 느리다
- 일반적으로 Primary Key를 clustered index로 지정

클러스터 인덱스 생성



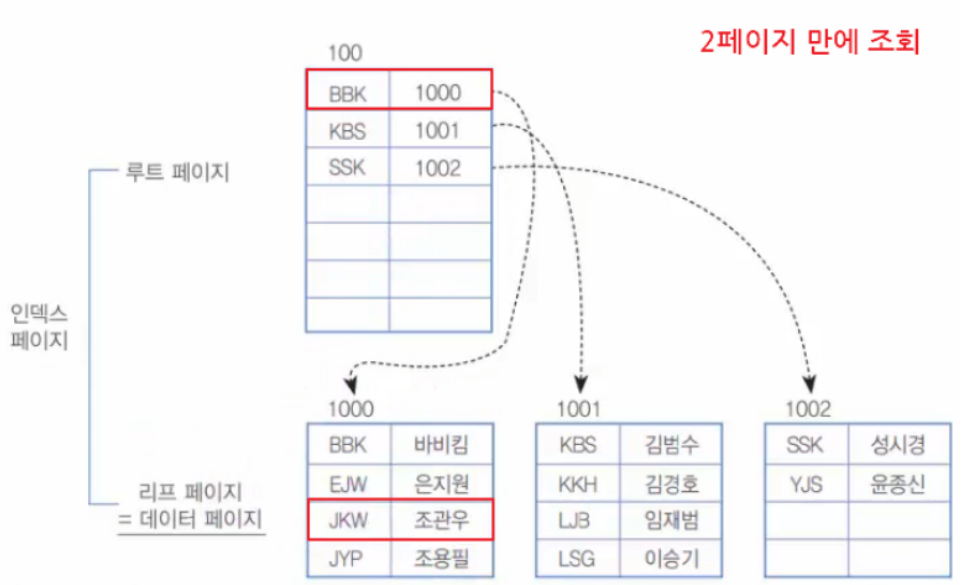
[그림 9-17] 클러스터형 인덱스의 구성 후

- 인덱싱을 하면 루트 페이지가 만들어짐
  - 루트 페이지? : 각 데이터 페이지의 첫번째 데이터만 따와서 모아 매핑시키는 페이지

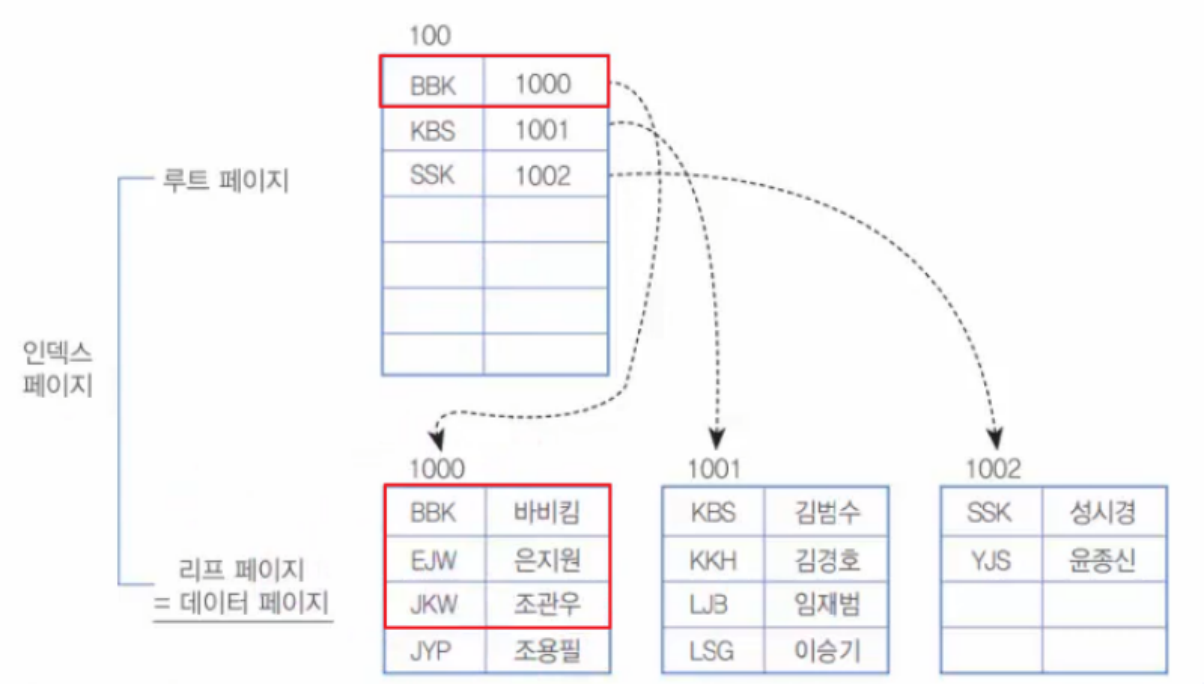
💡 MySQL은 데이터를 한 곳에다 다 저장하지 않고 페이지(page) 단위로 쪼개서 저장한다. 각 페이지의 기본 크기값은 16Kb이다.

- 데이터 페이지는 자동 정렬이 된다
- 데이터 페이지 자체가 인덱스 페이지가 된다

클러스터 인덱스 이용한 조회

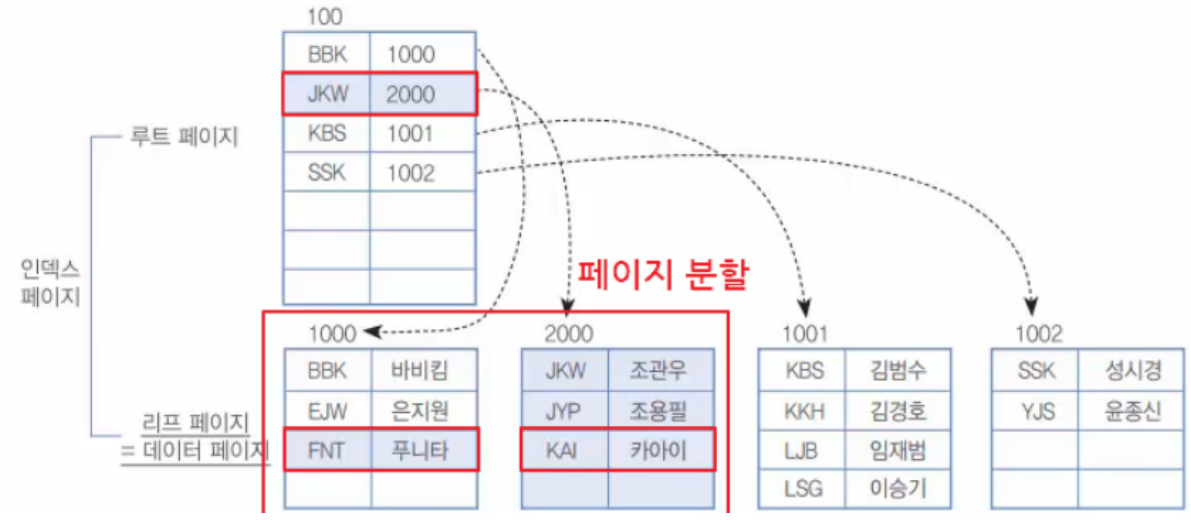


- JKW만 찾고 싶다면 두 페이지만에 조회가 가능하다
  - JKW는 BBK ≤ JKW < KBS 이므로 BBK가 가르키는 주소 확인
  - 해당 데이터 페이지에서 JKW를 찾으면 된다



- 여러개의 데이터를 범위로 검색한다 가정
  - 유저 아이디가 A~J로 시작하는 사용자를 모두 검색
- 두 페이지만에 조회가 가능해진다

클러스터 인덱스 이용한 데이터 삽입



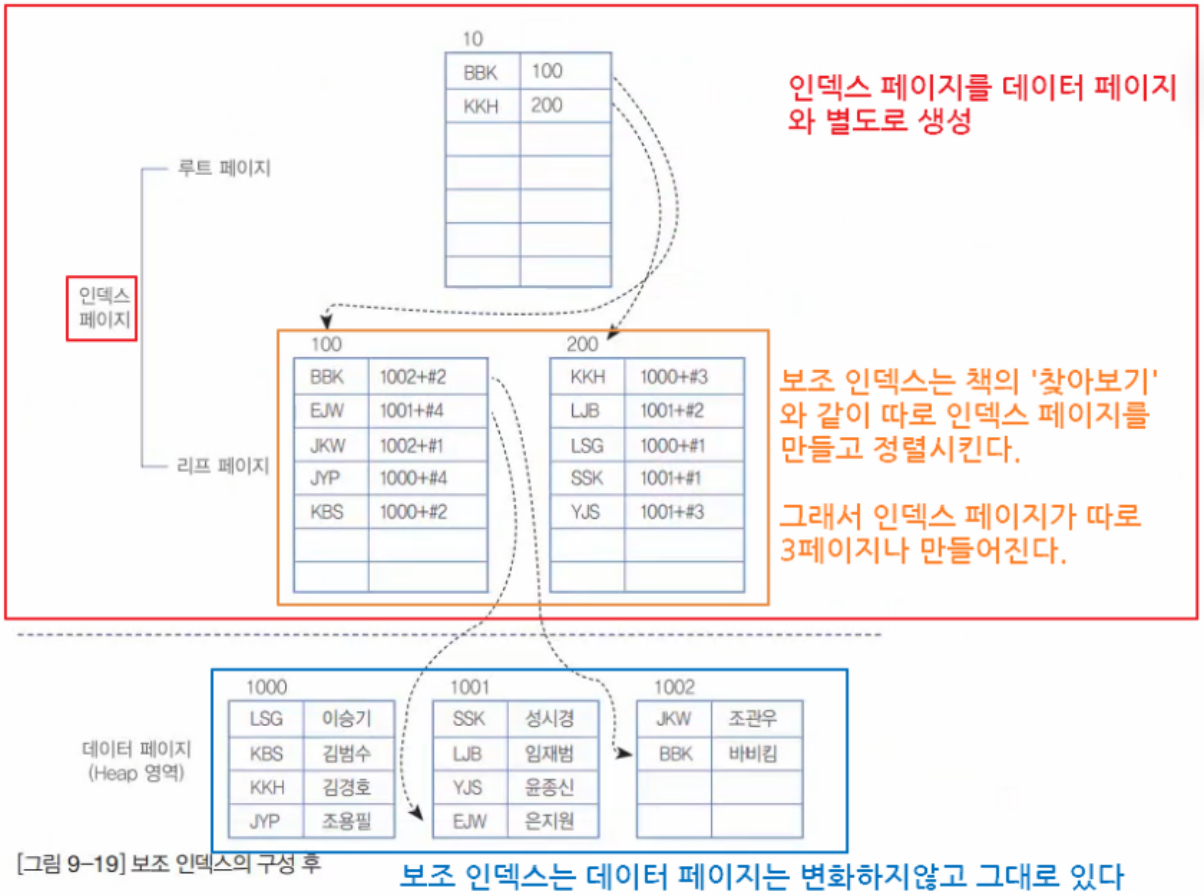
- 위 예제에서 “FNT” 데이터를 추가한다 가정
- 1000번 페이지에 공간이 없기 때문에 페이지 분할이 일어나 2000번 페이지가 생성이 됨

- 위와 같이 삽입, 삭제를 할 때 페이지 분할 혹은 추가적인 정렬이 필요해져 오히려 성능이 감소

보조 인덱스

- Non-clustered Index 라고도 불린다
- 데이터 페이지는 그대로 두고 별도의 페이지에 인덱스를 구성한다
  - 따라서, 자동 정렬을 하지 않는다
- 보조 인덱스의 리프 페이지는 데이터가 아닌 데이터가 위치하는 주소값
- 클러스터형보다 검색 속도는 더 느리지만 데이터의 입력, 수정, 삭제는 더 빠르다
- 여러개 생성할 수 있으나, 너무 많이 생성할 경우 오히려 성능을 떨어뜨릴 수 있다

보조 인덱스 생성

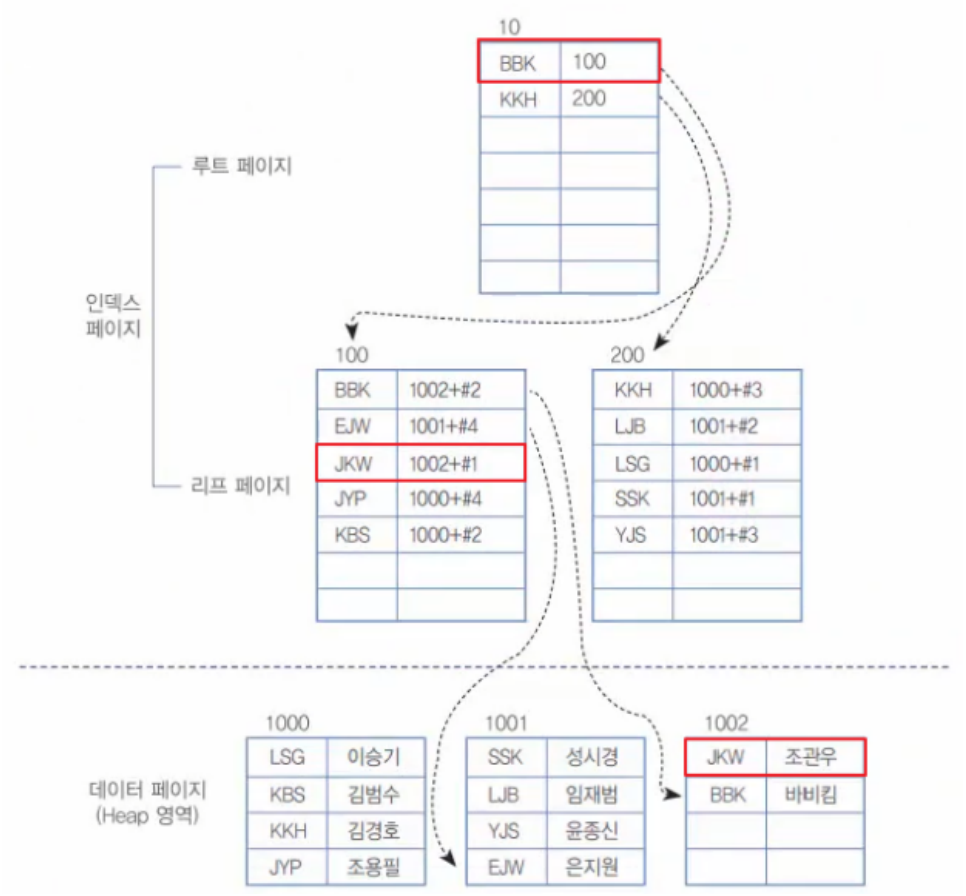


[그림 9-19] 보조 인덱스의 구성 후

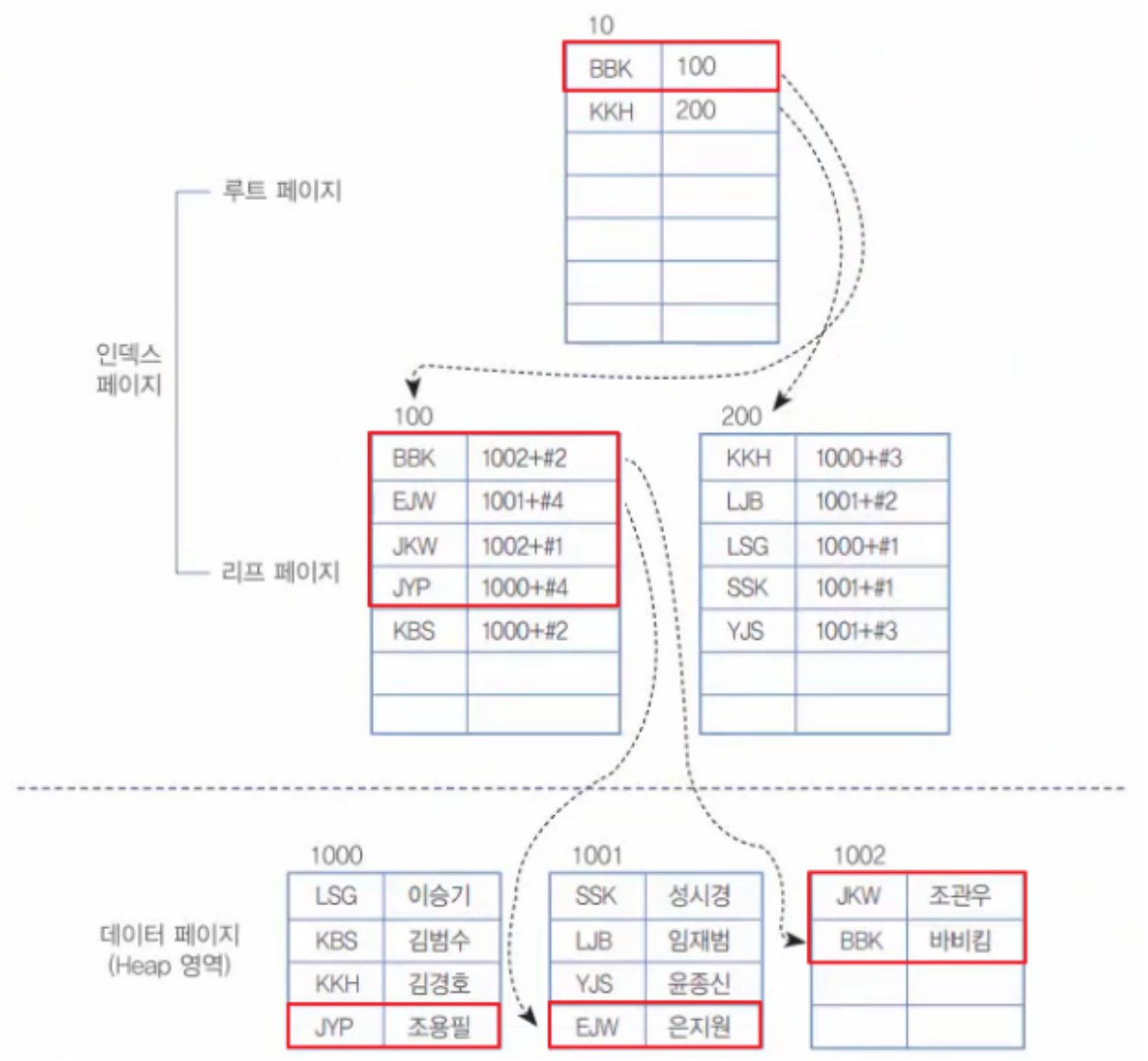
- 보조 인덱스 역시 루트 페이지를 생성한다
  - 그렇지만 데이터 페이지에 바로 연결하지 않고 따로 리프 페이지를 생성

    - 데이터 페이지의 주소와 매핑을 하고 리프 페이지를 정렬 시킨다
- 이처럼 추가 공간이 필요하므로 남용시 공간 낭비가 될 수 있다
- 데이터 페이지에 변화를 주지 않는다
    - 클러스터 인덱스와는 달리 여러개 생성이 가능한 이유

보조 인덱스 이용한 조회

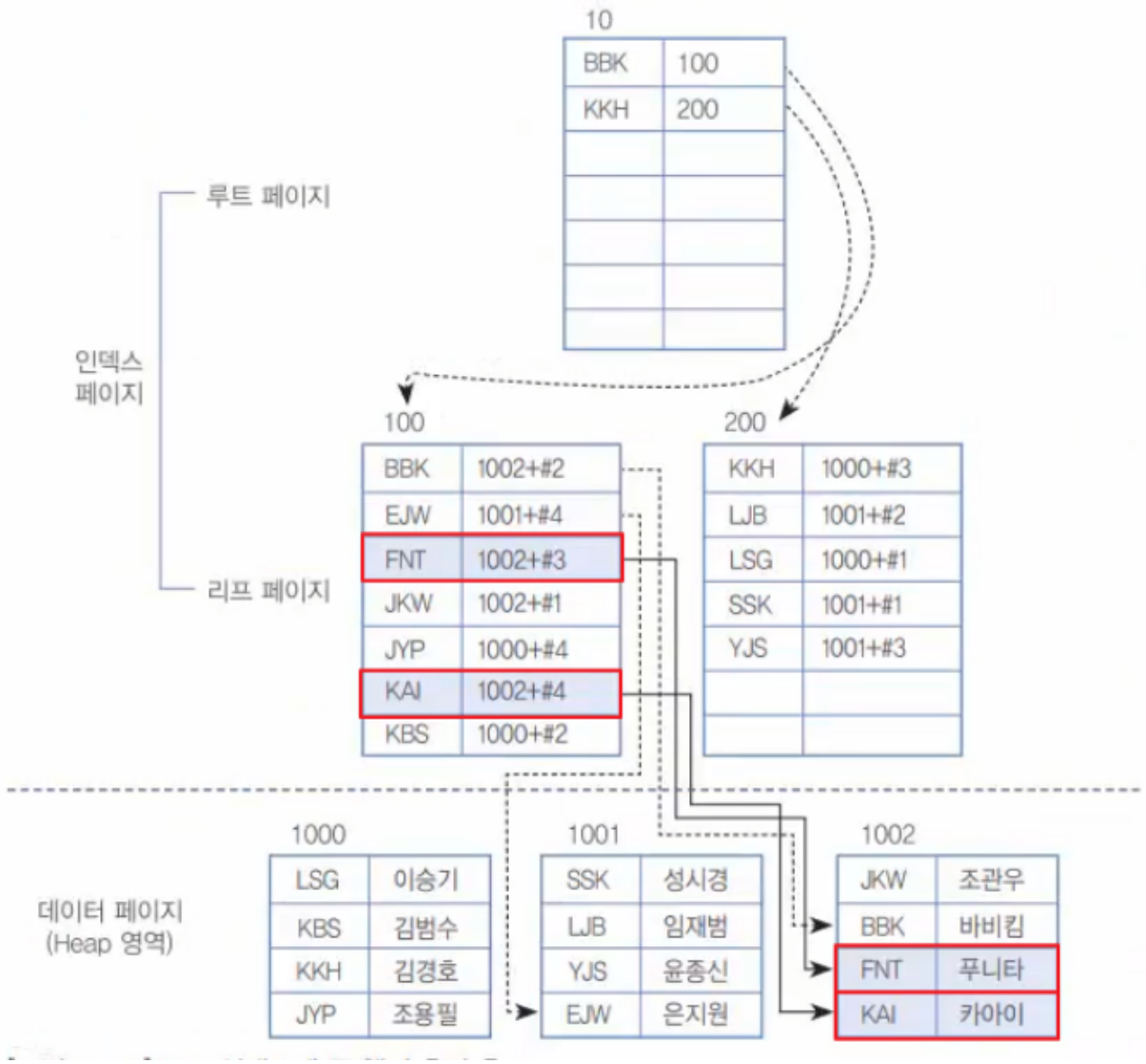


- JKW만 찾고 싶다면 세 페이지를 조회해야 함
    - 만약 데이터가 매우 많다면 클러스터 인덱스와 페이지 조회수 차이가 더 커질 것임



- 여러개의 데이터를 범위로 검색한다 가정
  - 유저 아이디가 A~J로 시작하는 사용자를 모두 검색
- 다섯 페이지를 조회해야 검색이 가능

보조 인덱스 데이터 삽입



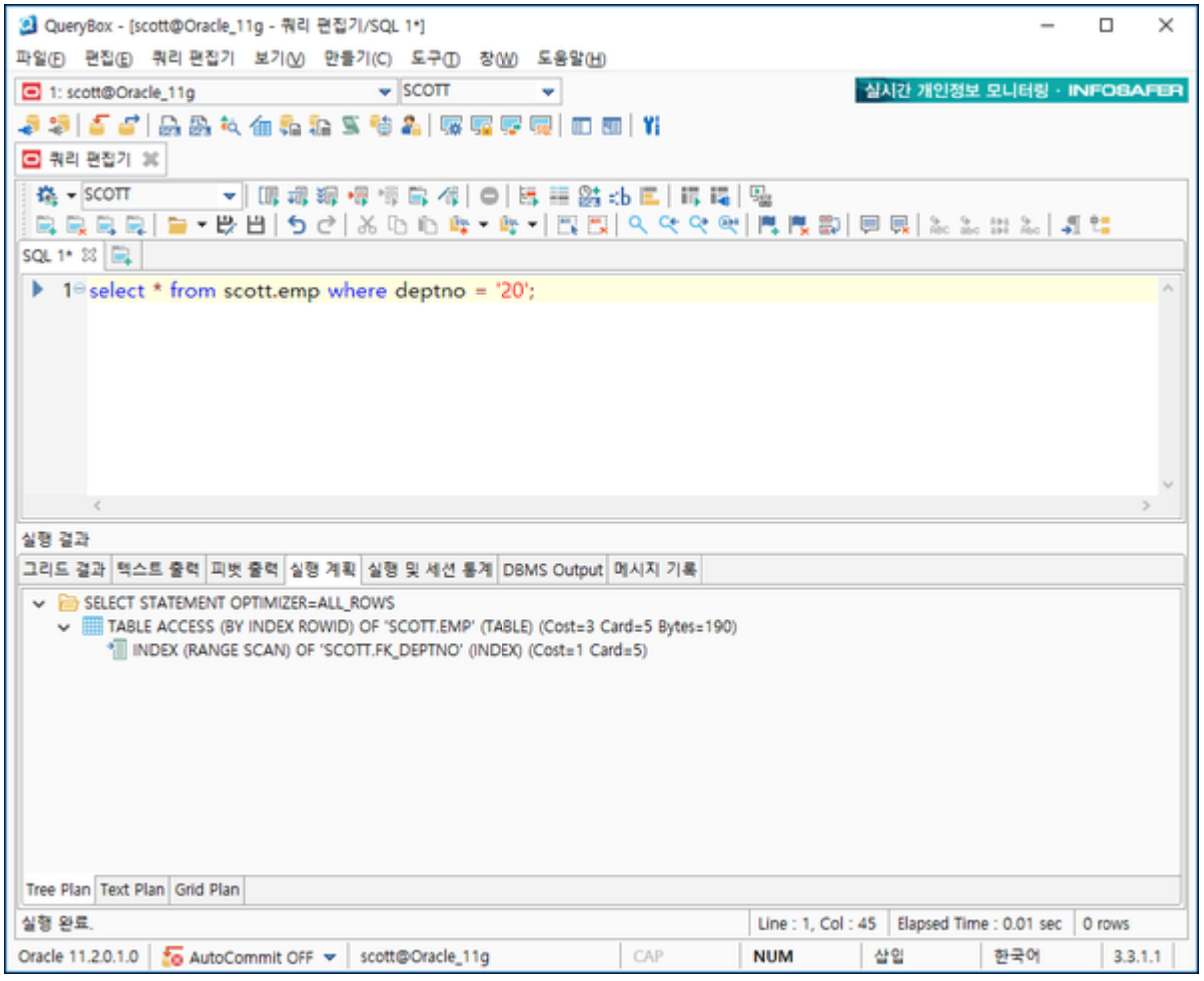
- 페이지 빈곳에 데이터를 넣고 리프 페이지에서 매핑만 하면 되기 때문에 테이블 분할이 일어나지 않는다
- 따라서 '삽입' 측면에서 클러스터 인덱스보다 성능 이점을 갖게 됨

인덱스 스캔 방식

- Index Range Scan
- Index Full Scan
- Index Unique Scan
- Index Skip Scan

Index Range Scan

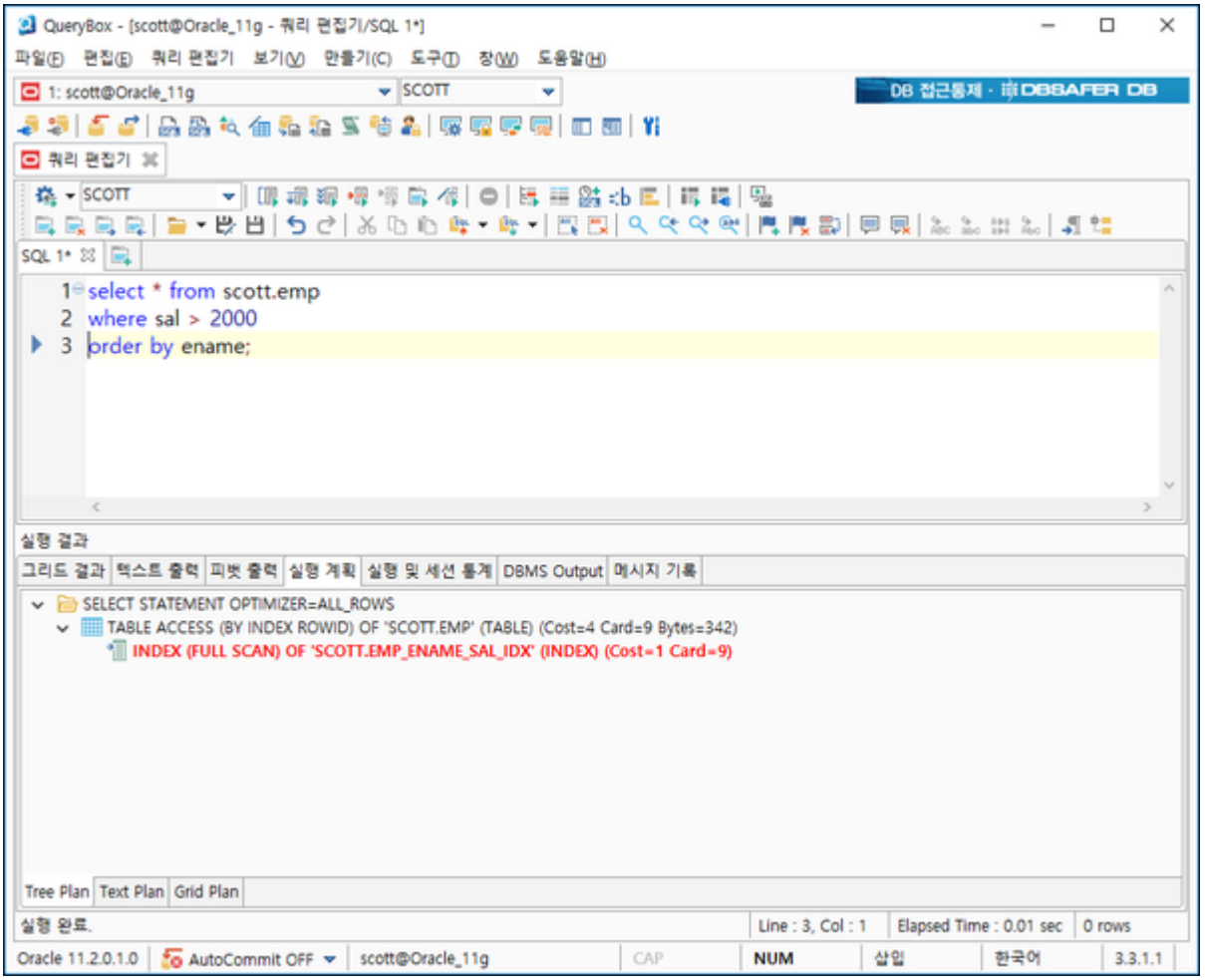
- B-tree 인덱스의 가장 일반적인 형태의 액세스 방식
- 인덱스 루트에서 리프 블록까지 수직적 탐색 후 필요한 범위만큼 수평적 탐색하는 방식
- 선두 컬럼을 가공하지 않은 상태로 조건절에 사용해야 함
    - 선두 컬럼? : 인덱스로 사용되는 첫 번째 열을 가리킴, 일반적으로 자주 검색에 사용되거나 필터링에 유용한 열이 선택됨



Index Full Scan

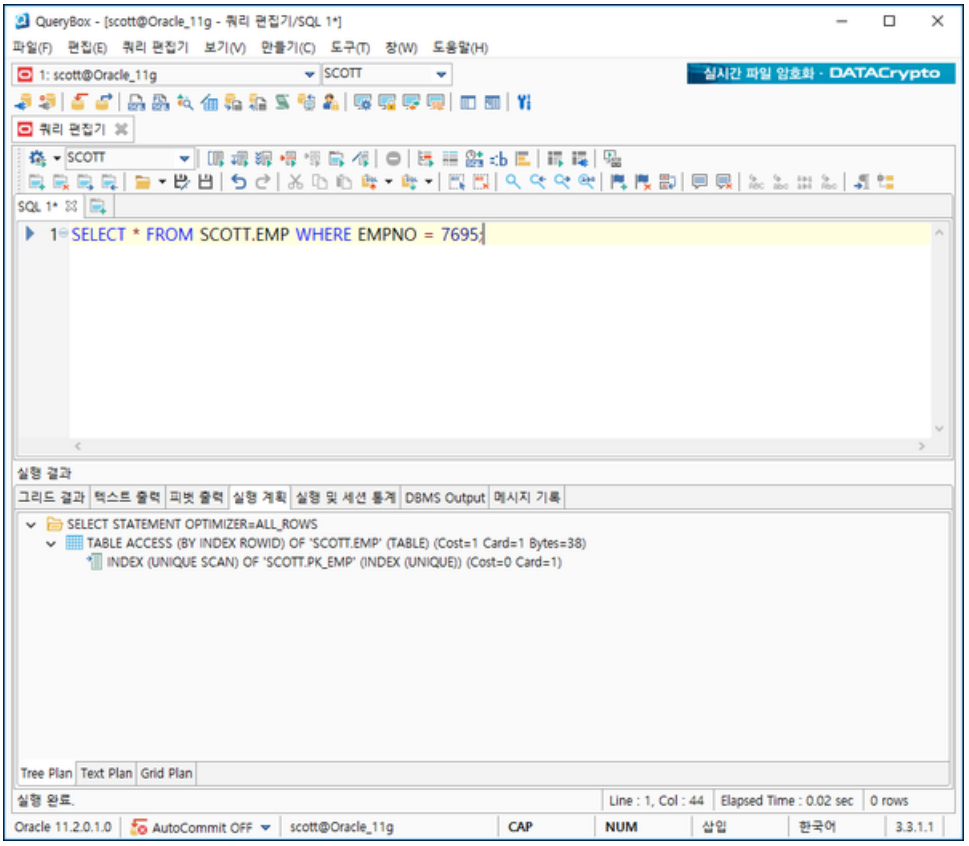
- 수직적 탐색 없이 인덱스 리프 블록 처음부터 끝까지 수평적으로 탐색
- 데이터 검색을 위한 최적의 인덱스가 없을 때 차선으로 선택
- 선두 컬럼인 ename이 조건절에 없으므로 Index Range Scan 불가능
- sal 컬럼이 인덱스에 있으므로 Index Full Scan을 통해 레코드를 찾는다

```
-- 인덱스 생성
CREATE INDEX emp_ename_sal_idx ON scott.emp(ename, sal);
```

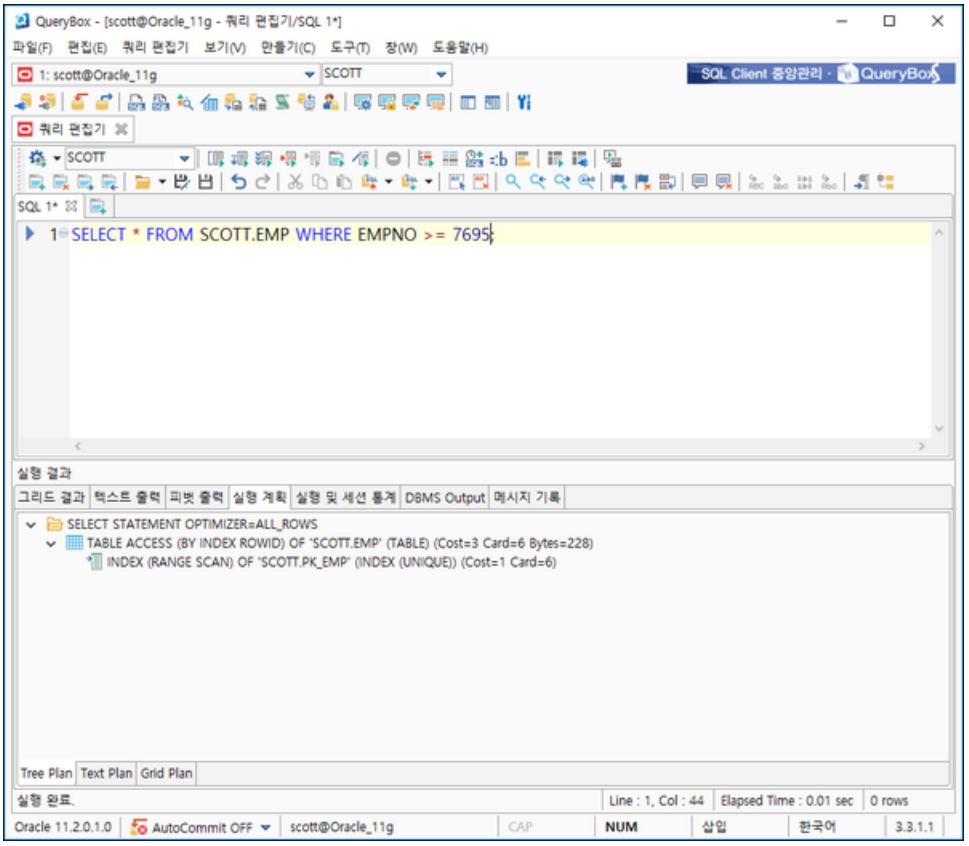


Index Unique Scan

- 수직적 탐색으로만 데이터를 찾는 스캔 방식
- Unique 인덱스를 '=' 조건으로 탐색하는 경우 동작
  - Unique 인덱스가 존재하는 컬럼은 중복 값이 존재하지 않음
  - 따라서 데이터를 한 건만 찾으면 되므로, 수직적 탐색만 이용



- 그렇지만, Unique 인덱스라고 해도 범위 조건(BETWEEN, >, <, LIKE 등) 조건으로 검색하게 되면 Index Range Scan으로 처리
    - 수직적 탐색으로만 레코드를 찾는 것은 무리가 있기 때문



Index Skip Scan

- 인덱스 선두 컬럼이 조건절에 없어도 인덱스를 활용하는 스캔 방식
- 조건절에 빠진 선행 인덱스 컬럼의 Distinct Value 갯수가 적고, 후행 컬럼의 Distinct Value 갯수가 많을 때 유용
  - Distinct Value? : 중복을 제외한 값
- 루트 또는 각 브랜치 블록에서 읽은 컬럼 값 정보를 이용해 조건절에 부합하는 레코드를 포함할 가능성이 있는 리프 블록만 액세스
  - Index Full Scan과 같이 모든 블록을 읽는 것이 아닌 가능성이 있는 블록만 액세스하는 방식

▼ 참고

Index Fast Full Scan

- Index Full Scan보다 빠른 스캔 방식
  - 논리적인 인덱스 트리 구조를 무시하고 인덱스 세그먼트 전체를 MultiBlock I/O 방식으로 스캔하기 때문에 더 빠르다
  - 물리적으로 디스크에 저장된 순서대로 인덱스 리프 블록을 읽음

- 빠르지만 인덱스 리프 노드가 갖는 연결 리스트 구조를 무시하기 때문에 인덱스 키 순서대로 정렬이 되어있지 않음
- 쿼리에 사용한 컬럼이 모두 인덱스에 포함되어 있을 경우만 사용 가능

DBMS 블록 Access 방식

- Sequential Access
- Radom Access
- Single Block I/O
- Multi Block I/O

Sequential Access

- 논리적 혹은 물리적으로 연결된 순서에 따라서 데이터를 순차적으로 접근
- 2가지 경우에 발생
  - 인덱스 리프노드 스캔 : 리프노드가 연결리스트 형태로 구성되어 있으므로 논리적인 연결고리가 존재하기 때문에 순차 탐색
  - Table Full Scan : 순차적으로 모든 테이블 탐색

Single Block I/O

- 하나의 블록을 읽기 위해 디스크 I/O를 하는 것
- 읽고난 후에는 데이터가 database buffer cache에 올라가게 됨

Multi Block I/O

- 한 번에 여러 블록을 디스크로부터 I/O 하는 것
- 마찬가지로 읽은 데이터는 database buffer cache에 캐싱 됨
- 주로 Table Full Scan을 할 때 일어남