인덱스

8.1 디스크 읽기 방식

데이터 저장 매체는 컴퓨터에서 가장 느린 부분이라는 사실에는 변함이 없습니다. 때문에 데이터 베이스 성능 튜닝은 어떻게 디스크 I/O를 줄이느냐가 관건일때가 상당히 많습니다.

두가지의 I/O 방식을 알아봅시다.

- 랜덤 I/O
- 순차 I/O

8.1.1 하드 디스크 드라이브(HDD)와 솔리드 스테이드 드라이브(SSD);

기계식 저장 장치인 **하드 디스크 드라이브(HDD)**를 대체 하기 위해 저장 매체인**SSD(Solid State Drive)**가 많이 출시되고 있습니다.

SDD도 기존 HDD와 같은 인터페이스(SATA나 SAS)를 지원하므로 내장 디스크나 DAS 또는 SAN에 그대로 사용할 수 있습니다.

SSD는 데이터 저장용 플래터를 제거하고, 그 대신 플레시 메모리를 장착했습니다.

- 디스크 원판을 회전시킬 필요가 없어 빠르게 데이터를 읽고 쓸 수 있습니다.
- 전원이 끊어져도 데이터가 삭제되지 않습니다.
- 컴퓨터 메모리(D-RAM)보단 느리지만 HDD보다 빠릅니다.

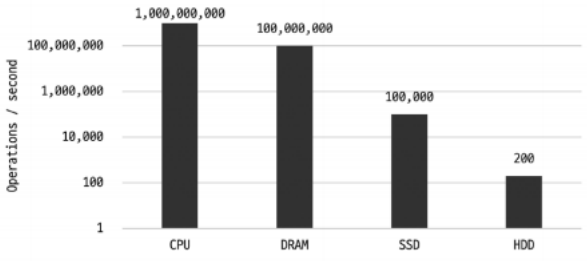


그림 8.1 주요 장치의 초당 처리 횟수(수치가 클수록 빠른 장치를 의미)

메모리와 디스크의 처리 속도는 10만 배 이상의 차이를 보입니다.

그에 비해 SSD는 1000배 가량의 차이를 보입니다. 요즘은 DBMS용으로 사용할 서버에는 대부분 SSD를 채택하고 있습니다.

SSD의 순차 I/O는 HDD와 성능이 비슷하거나 조금 더 빠르지만 랜덤 I/O 작업은 훨씬 빠르기 때문에 랜덤 I/O를 통해 작은 데이터를 읽고 쓰는 작업이 대부분인 DMS용 스토리지에 최적이라고 볼 수 있습니다.

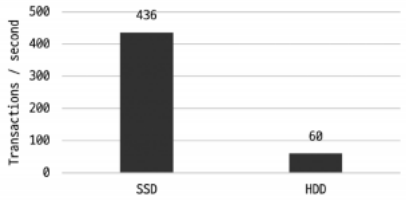


그림 8.2 솔리드 스테이트 드라이브(SSD)와 하드 디스크 드라이브(HDD)의 성능 벤치마크

8.1.2 랜덤 I/O와 순차 I/O

랜덤 I/O라는 표현은 HDD의 플래터를 돌려서 읽어야 할 데이터가 저장된 위치로 디스크 헤더를 이동시킨 다음 데이터를 읽는 것을 의미합니다.

랜덤 I/O 순차 I/O를 비교해봅시다.

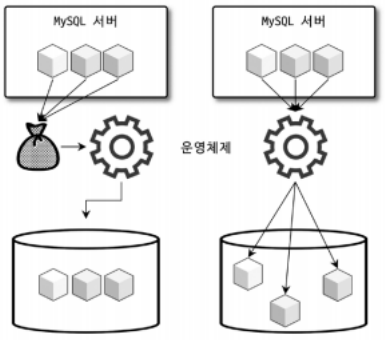


그림 8.3 순차 I/O(왼쪽)와 랜덤 I/O(오른쪽) 비교

순차 I/O는 3개의 페이지(3X16KB) 위해 1번의 시스템 콜을 요청했지만, 랜덤 I/O는 3개의 페이지를 디스크에 기록하기 위해 3번 시스템 콜을 요청합니다.

디스크의 I/O의 시간은 디스크 헤더를 움직여서 읽고 쓸 위치로 옮기는 단계에서 결정되는데 위 그림의 경우 순차 I/O가 랜덤 I/O보다 거의 3배 정도 빠릅니다.

그래서 여러번 쓰기 또는 읽기 요청하는 랜덤 I/O작업이 작업 부하가 훨씬 더 큽니다.

데이터베이스의 대부분의 작업은 이러한 작은 데이터를 빈번히 읽고 쓰기 때문에 MySQL서버에는 그룹 커밋, 바이너리로그,버퍼,로그버퍼 등이 기능이 내장되어 있습니다.

SDD는 원판을 가지지 않아서 랜덤과 순차의 차이가 없을 것으로 보이지만, 실제로는 랜덤I/O는 여전히 순차I/O보다 전체 스루풋(Throughput)이 떨어집니다.

> **참고** 이 책에서 소개하는 순차 I/O와 랜덤 I/O의 비교는 쉽게 이해할 수 있게 단순하게 비교해서 설명한 것이다. 랜덤 I/O나 순차 I/O 모두 파일에 쓰기를 실행하면 반드시 동기화(fsync 또는 flush 작업)가 필요하다. 그런데 순차 I/O인 경우에도 이러한 파일 동기화 작업이 빈번히 발생한다면 랜덤 I/O와 같이 비효율적인 형태로 처리될 때가 많다. 기업용으로 사용하는 데이터베이스 서버에는 캐시 메모리가 장착된 RAID 컨트롤러가 일반적으로 사용되는데, RAID 컨트롤러의 캐시 메모리는 아주 빈번한 파일 동기화 작업이 호출되는 순차 I/O를 효율적으로 처리될 수 있게 변환하는 역할을 한다. 하드 디스크 드라이브뿐만 아니라 SSD를 사용하는 경우에도 여전히 RAID 컨트롤러는 중요한 역할을 하기 때문에 RAID 컨트롤러와 RAID 컨트롤러에 장착된 캐시의 성능을 무시하지 말자.

> **참고** 인덱스 레인지 스캔은 데이터를 읽기 위해 주로 랜덤 I/O를 사용하며, 풀 테이블 스캔은 순차 I/O를 사용한다. 그래서 큰 테이블의 레코드 대부분을 읽는 작업에서는 인덱스를 사용하지 않고 풀 테이블 스캔을 사용하도록 유도할 때도 있다. 이는 순차 I/O가 랜덤 I/O보다 훨씬 빨리 많은 레코드를 읽어올 수 있기 때문인데, 이런 형태는 OLTP(On-Line Transaction Processing) 성격의 웹 서비스보다는 데이터 웨어하우스나 통계 작업에서 자주 사용된다.

8.2 인덱스란

책의내용 → 데이터 파일

인덱스 → 색인, 찾아보기

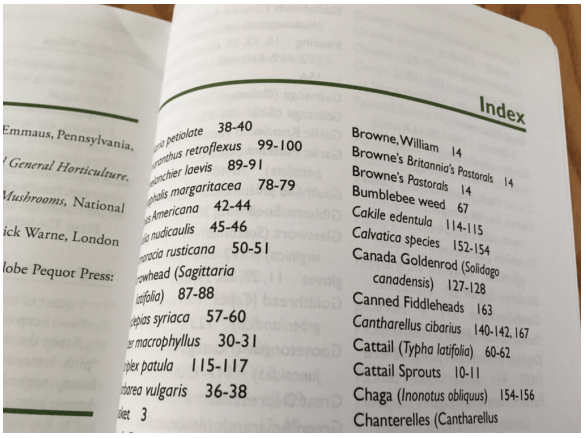
페이지 번호→ 데이터 파일에 저장된 레코드의 주소

DB의 모든 테이블을 돌면서 검색하려면 오랜 시간이 걸립니다. 그래서 칼럼(또는 칼럼들)의 값과 해당 레코드가 저장된 주소를 키와 값의 상으로 삼아 인덱스를 만들어 두는 것입니다.

그리고 책의 "**찾아보기**"와 DBMS 인덱스의 공통점 가운데 중요한 것은 바로 정렬입니다.

책의 찾아보기도 내용이 많아지면 우리가 원하는 검색어를 찾아내는 데 시간이 걸립니다.

그래서 최대한 빠르게 찾아갈 수 있게 "ㄱ", "ㄴ", "ㄷ"과 같은 순서로 정렬돼 있는데, DBMS의 인덱스도 마찬가지로 칼럼의 값을 주어진 순소러 미리 정렬해서 보관합니다.



프로그래밍 언어의 자료구조와 인덱스를 비교해 가면서 살펴봅시다.

SortedList (정렬 o)→ DBMS의 인덱스와 같은 자료 구조

ArrayList (정렬 x)→ 데이터 파일과 같은 자료 구조

SortedList는 자료 구조가 저장될 때마다 항상 값을 정렬해야 하기 때문에 저장 과정이 느립니다. 하지만 원하는 값을 빠르게 찾을 수 있습니다.

마찬가지로 DBMS의 인덱스도 인덱스가 많은 테이블은 당연히 INSERT,UPDATE,DELETE 문장의 처리가 느려집니다. 하지만 SELECT 문장은 매우 빠르게 처리됩니다.

인덱스의 사용 결정기준

- 데이터의 저장 속도를 어디까지 희생해도 되는지
- 읽기 속도를 얼마나 더 빠르게 만들어야 하는지

WHERE 조건 절에 사용되는 칼럼이라고 전부 인덱스로 생성하면 오히려 역효과를 불러 일으킬 수 있습니다.

인덱스는 데이터를 관리하는 방식(알고리즘)과 중복 값의 허용 여부 등에 따라 여러 가지로 나눠볼 수 있습니다.(저자의 임의의 분류이며, KEY라는 말과 INDEX는 같은 의미로 사용하겠습니다.)

인덱스를 역할별로 구분한다면 **프라이머리 키(Primary key)**와 **보조 키(세컨더리 인덱스, Secondary key)**로 구분할 수 있다.

프라이머리 키(Primary key)

- 프라이머리 키는 이미 잘 아는 것처럼 그 레코드를 대표하는 칼럼의 값으로 만들어진 인덱스 입니다.
- 테이블에서 해당 레코드를 식별할 수 있는 기준값입니다.
- Null 값을 허용하지 않으면 중복을 허용하지 않습니다.

보조 키(세컨더리 인덱스, Secondary key)

- 프라이머리 키를 제외한 나머지 모든 인덱스는 세컨더리 인덱스입니다.

데이터 저장 방식 별로 구분할 경우 대표적인 데이터 저장 방식 B-Tree 인덱스와 Hash 인덱스로 구분할 수 있습니다.

B-Tree

- 가장 일반적으로 사용되는 인덱스 알고리즘 입니다.
- 칼럼의 값을 변형하지 않고 원래의 값을 이용해 인덱싱하는 알고리즘입니다.
- 위치 기반 검색을 지원하기 위한 R-Tree 인덱스 알고리즘도 있지만, R-Tree는 B-Tree의 응용 알고리즘 입니다.

Hash

- 칼럼의 값으로 해시값을 계산해서 인덱싱하는 알고리즘입니다.
- 매우 빠른 검색을 지원하지만 값을 변형해서 인덱싱하므로 전방(prepix) 일치와 같이 값의 일부만 검색하거나 범위를 검색할 때는 해시 인덱스를 사용할 수 없습니다.
- 주로 메모리 기반의 데이터베이스에서 많이 사용합니다.

데이터의 중복 허용 여부로 분류하면 유니크 인덱스(Unique)와 유니크 하지않은 인덱스(Non-Unique)로 구분할 수 있습니다.

인덱스가 유니크한지 아닌지는 단순히 값이 1개만 존재하는지 1개 이상 존재할 수 있는지를 의미하지만, 실제 DBMS를 실행해야 하는 옵티마이저는 유니크 인덱스에 대해 동등 조건으로 검색해 항상 1건의 레코드만 검색하면 됩니다..

인덱스의 기능별로 분류해보면 전문 검색용 인덱스와 공간 검색용 인덱스 등을 예로 들 수 있습니다.

8.3 B-Tree 인덱스

B-Tree(Balanced-Tree)는 데이터베이스의 인덱싱 알고리즘 가운데 가장 일반적으로 사용되고, 가장 먼저 도입된 알고리즘입니다.

- B-Tree는 칼럼의 원래 값을 변형시키지 않고(값의 앞부분만 잘라서 관리하기는 합니다.) 인덱스 구조체 내에서 항상 정렬된 상태로 유지합니다.
- 전문 검색과 같은 특수한 요건이 아닌 경우, 대부분 인덱스는 거의 B-Tree를 사용할 정도로 일반적인 용도에 적합한 알고리즘입니다.

8.3.1 구조 및 특성

- B-Tree는 트리 구조의 최상위에 하나의 **“루트 노드(Root node)”**가 존재합니다. 그 하위에 자식 노드가 붙어 있는 형태입니다
- 트리 구조의 가장 하위에 있는 노드를 **“리프 노드(Leaf node)”**라고 하며, 항상 실제 데이터 레코드를 찾아가기 위한 주솟값을 가지고 있습니다.
- 루트 노드도 아니고 리프 노드도 아닌 중간 노드를 **“브랜치 노드(Branch node)”**라고 합니다.

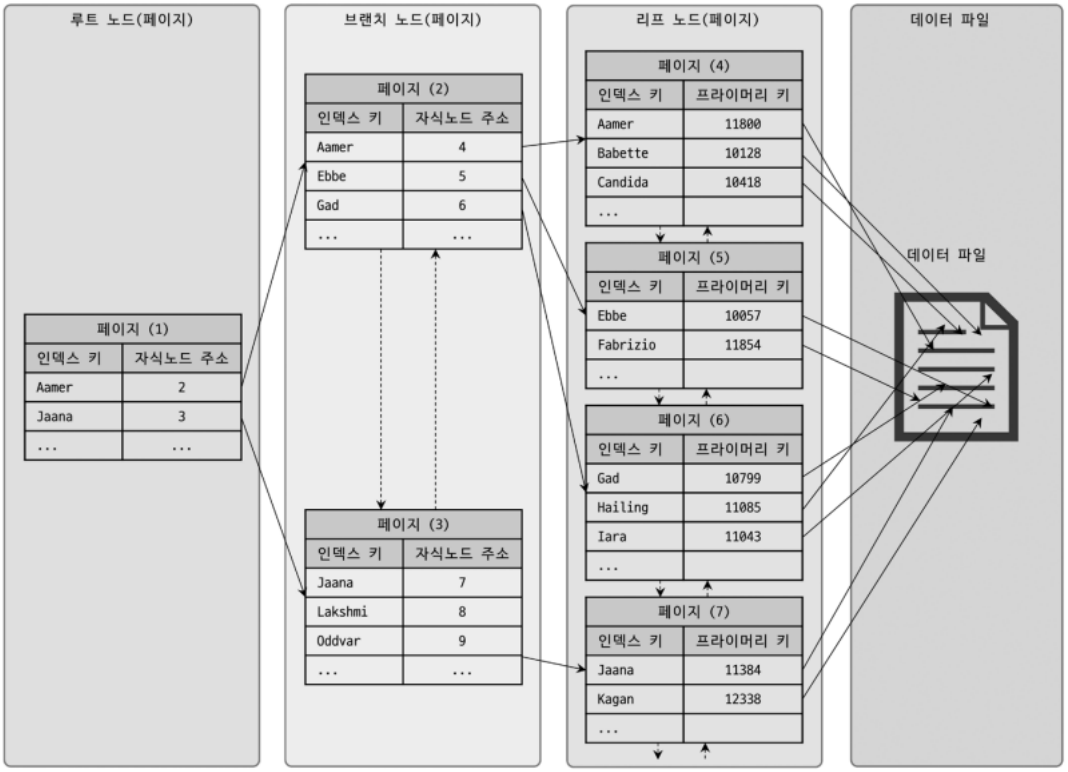


그림 8.4 B-Tree 인덱스의 구조

데이터 파일의 레코드는 INSERT된 순서대로 저장되는 것이 아니라 레코드가 삭제되어 빈 공간이 생기면 그다음의 INSERT는 가능한 한 삭제된 공간을 재활용하도록 DBMS가 설계되기 때문에 항상 INSERT된 순서로 저장되지 않습니다.

> 💡 대부분의 RDBMS의 데이터 파일에서 레코드는 정렬되지 않고 임의의 순서로 저장되지만 InnoDB테이블 에서 레코드는 클러스터되어 디스크에 저장되므로 기본적으로 프라이머리 키 순서로 정렬되어 저장됩니다.

인덱스는 테이블의 키 칼럼만 가지고 있으므로 나머지 칼럼을 읽으려면 데이터 파일에서 해당 레코드를 찾아야합니다.

이를 위해 인덱스의 리프 노드는 데이터 파일에 저장된 레코드의 주소를 가집니다.

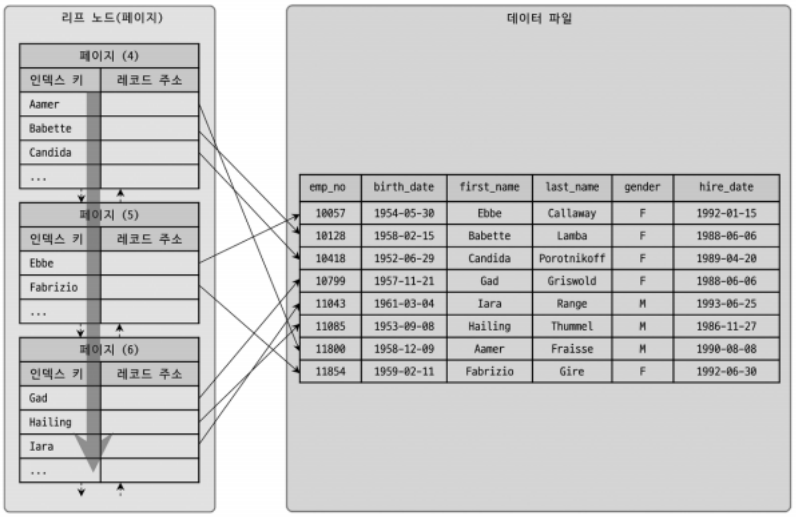
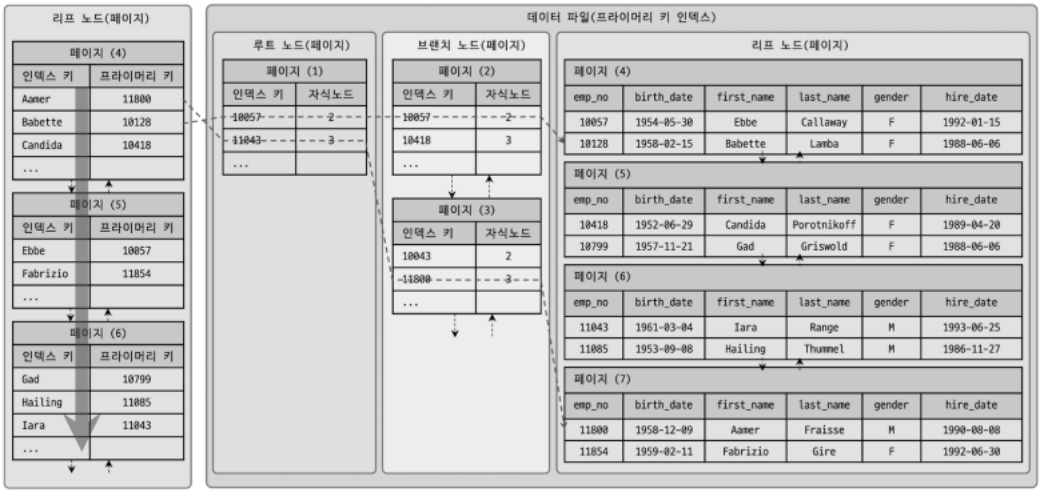


그림 8.5 B-Tree의 리프 노드와 테이블 데이터 레코드(MyISAM)



B-Tree의 리프 노드와 테이블 데이터 레코드(InnoDB)

위의 그림은 테이블의 인덱스와 데이터 파일의 관계를 보여줍니다.

MyISAM

- 레코드 주소는 MyISAM 테이블의 생성 옵션에 따라 레코드가 테이블에 INSERT된 순번이거나 데이터 파일 내의 위치 입니다.
- 세컨더리 인덱스가 물리적인 주소를 가집니다.

InnoDB

- InnoDB를 사용하는 테이블은 프라이머리 key가 ROWID의 역할을 합니다.
- 프라이머리 키를 주소처럼 사용하기 때문에 논리적인 주소를 가집니다.

  InnoDB는MyISAM과 달리 세컨더리 키가 논리적인 주소를 가지기 때문에 8.5그림처럼 데이터 파일을 곧바로 찾지 못하고, 프라이머리 키를 통해 프라이머리 키 인덱스를 찾은 뒤, 프라이머리 키 인덱스의 리프 페이지에 저장돼 있는 레코드를 읽습니다.

8.3.2 B-Tree 인덱스 키 추가 및 삭제

- 테이블의 레코드를 저장하거나 변경하는 경우 인덱스 키 추가나 삭제 작업이 발생합니다. 인덱스 키 추가 나 삭제가 어떻게 처리되는지 알아두면 쿼리의 성능을 쉽게 예측할 수 있습니다.

8.3.2.1 인덱스 키 추가

- B-Tree에 저장될 때는 저장될 키 값을 이용해 B-Tree상의 적절한 위치를 검색해야 합니다.
- 저장될 위치가 결정되면 레코드의 키 값과 대상 레코드의 주소 정보를 B-Tree의 리프 노드에 저장합니다.
- 리프 노드가 꽉 차서 더는 저장할 수 없으면 분리(split)되야 하는데, 이는 상위 브랜치 노드까지의 처리의 범위가 넓어집니다. 이러한 작업 탓에 상대적으로 쓰기 작업에 비용이 많이 듭니다.
- InnoDB 스토리지 엔진은 필요하다면 인덱스 키 추가 작업을 지연시켜서 나중에 처리할 수 있습니다. 하지만 프라이머리 키나 유니크 인덱스의 경우 중복 체크가 필요하기 때문에 B-Tree에 추가하거나 삭제합니다.

8.3.2.2 인덱스 키 삭제

B-Tree의 키 값이 삭제되는 경우는 상당히 간단합니다.

해당 키 값이 저장된 B-Tree의 리프 노드를 찾아서 그냥 삭제 마크만 하면 작업이 완료됩니다.

- 삭제 마킹된 인덱스 키 공간은 계속 그대로 방치 하거나 재활용 할 수 있습니다.
- 인덱스 키 삭제로 인한 마킹 작업 또한 디스크 쓰기가 필요한 디스크 I/O 작업입니다.
- 처리가 지연된 인덱스 키 삭제 또한 사용자에게 특별한 악영향이 없습니다.

8.3.2.3 인덱스 키 변경

인덱스의 키 값은 그 값에 따라 저장될 리프 노드의 위치가 결정되므로 B-Tree 키 값이 변경되는 경우 단순히 인덱스상의 키 값만 변경하는 것은 불가능합니다.

B-Tree의 키 값 변경 작업은 먼저 키 값을 삭제한 후, 다시 새로운 키 값을 추가하는 형태로 처리됩니다.

인덱스 키 값을 변경하는 작업은 인덱스 키 값을 삭제한 후 인덱스 키 값을 추가하는 작업이므로 앞에서 설명한 내용대로 처리되며, InnoDB 스토리지 엔진을 사용하는 테이블에 대해서는 이 작업 모두 체인지 버퍼를 활용해 지연 처리 될 수 있습니다.

8.3.2.4 인덱스 키 검색

인덱스를 검색하는 작업은 B-Tree 루트 노드부터 시작해 브랜치 노드를 거쳐 최종 리프 노드까지 이동하면서 비교작업을 수행하며, 이 과정을 **“트리 탐색”**이라고 부릅니다.

- B-Tree 인덱스를 이용한 검색은 100% 일치 또는 값의 앞부분만 일치하는 경우에 사용할 수 있습니다.
- 인덱스를 구성하는 키 값의 뒷부분만 검색하는 용도로는 인덱스를 사용할 수 없습니다.
- 인덱스를 이용한 검색에서 중요한 사실은 인덱스의 키 값에 변형이 가해진 후 비교되는 경우에는 절대 B-Tree 빠른 검색 기능을 사용할 수 없습니다(이미 변형된 값은 B-Tree 인덱스에 존재하는 값이 아니기 때문).
- 따라서 함수나 연산을 수행한 결과로 정렬한다거나 검색하는 작업은 B-Tree의 장점을 이용할 수 없습니다.

8.3.3 B-Tree 인덱스 사용에 영향을 미치는 요소

B-Tree 인덱스는 인덱스를 구성하는 **칼럼의 크기**와 **레코드의 건수**, 그리고 **유니크한 인덱스 키 값의 개수** 등에 의해 검색이나 변경 작업의 성능이 영향을 받습니다.

8.3.3.1 인덱스 키 값의 크기

인덱스는 페이지 단위로 관리되며, 그림 8.4에서 루트와 브랜치,그리고 리프 노드를 구분한 기준이 바로 페이지 단위입니다.

DBMS의 B-Tree는 자식 노드의 개수가 가변적인 구조입니다. 그렇다면 MySQL의 B-Tree는 자식 노드를 몇 개까지 가지게 될까요?

자식 노드의 개수는 인덱스의 페이지 크기와 키 값의 크기에 따라 결정됩니다. 인덱스의 키가 16byte라고 가정하면 다음 그림과 같이 인덱스 페이지가 구성 될 것입니다.

그림 8.7 인덱스 페이지의 구성

자식 노드 주소라는 것은 여러 가지 복합적인 정보가 담긴 영역이며, 페이지의 종류별로 대략 6바이트에서 12바이트까지 다양한 크기의 값을 가질 수 있습니다.(편의상 자식 노드 주소 영역이 평균 12바이트로 구성된다고 가정합시다.)

16*1042/(16+12) = 585개를 저장 할 수 있습니다.

(인덱스 페이지 크기) / (인덱스 키 크기 + 자식노드 주소 크기)

인덱스를 구성하는 키값의 크기가 커지면 디스크로부터 읽어야하는 횟수가 늘어나고, 그만큼 느려지게 됩니다.

8.3.3.2 B-Tree 깊이

인덱스 키 값의 크기가 커지면 커질수록 하나의 인덱스 페이지가 담을 수 있는 인덱스 키 값의 개수가 적어지고, 그 때문에 같은 레코드 건수라 하더라도 B-Tree의 깊이가 깊어져서 디스크 읽기가 더 많이 필요합니다.

8.3.3.3 선택도(기수성)

인덱스에서 **선택도(Selectivity)** 또는 **기수성(Cardinality)**는 거의 같은 의미로 사용되며, 모든 인덱스 키 값 가운데 유니크한 값의 수를 의미합니다.

예를들어 전체 인덱스 키 값은 100개인데, 그중에서 유니크한 값의 수는 10개라면 기수성은 10입니다.

즉 인덱스 키 값 가운데 중복된 값이 많아지면 기수성은 낮아지고 동시에 선택도 또한 떨어지며, 인덱스는 선택도가 높을수록 검색 대상이 줄어들기 때문에 중복값이 적을수록 그만큼 빠르게 처리됩니다.

> 💡 선택도가 좋지 않다고 하더라도 정렬이나 그루핑과 같은 작업을 위해 인덱스를 만드는 것이 훨씬 나은 경우도 많습니다. 인덱스가 항상 검색에만 사용되는 것이 아니므로 여러 가지 용도를 적절히 고려해야합니다.

country라는 칼럼과 city라는 칼럼이 포함된 tb_test 테이블을 예로 들어봅시다. 전체 레코드 건수는 1만 건이며, country 칼럼으로만 인덱스가 생성된 상태에서 아래의 두 케이스를 살펴봅시다.

- 케이스 A: country 칼럼의 유니크한 값의 개수가 10개
- 케이스 B : country 칼럼의 유니크한 값의 개수가 1000개

```
SELECT * FROM tb_test
WHERE country='KOREA' AND city 'SEOUL';
```

- country 칼럼의 유니크 값이 10개일 때

  country 칼럼의 유니크 값이 10개이므로 tb_city 테이블에는 10개 국가(country)의 도시(city) 정보가 저장돼 있는 것이다. MySQL 서버는 인덱스된 칼럼(country)에 대해서는 전체 레코드의 건수나 유니크한 값의 개수 등에 대한 통계 정보를 가지고 있다. 여기서 전체 레코드 건수를 유니크한 값의 개수로 나눠보면 하나의 키 값으로 검색했을 때 대략 몇 건의 레코드가 일치할지 예측할 수 있게 된다. 즉, 이 케이스의 tb_city 테이블에서는 country='KOREA'라는 조건으로 인덱스를 검색하면 1000건(10,000/10)이 일치하리라는 것을 예상할 수 있다. 그런데 인덱스를 통해 검색된 1000건 가운데 city='SEOUL'인 레코드는 1건이므로 999건은 불필요하게 읽은 것으로 볼 수 있다.

- country 칼럼의 유니크 값이 1000개일 때

  country 칼럼의 유니크 값이 1000개이므로 tb_city 테이블에는 1000개 국가(country)의 도시(city) 정보가 저장돼 있는 것이다. 이 케이스에서도 전체 레코드 건수를 국가 칼럼의 유니크 값 개수로 나눠보면 대략 한 국가당 대략 10개 정도의 도시 정보가 저장돼 있으리라는 것을 예측할 수 있다. 그래서 이 케이스에서는 tb_city 테이블에서 country='KOREA'라는 조건으로 인덱스를 검색하면 10건(10,000/1,000)이 일치할 것이며, 그 10건 중에서 city='SEOUL'인 레코드는 1건이므로 9건만 불필요하게 읽은 것이다.

8.3.3.4 읽어야 하는 레코드 건수

인덱스를 사용하여 레코드를 읽는 것은 테이블에서 직접 레코드를 읽는 것보다 비용이 4~5배 더 많이 든다고 합니다. 따라서 쿼리가 전체 테이블 레코드의 20~25% 이상을 읽어야 한다면, 인덱스를 사용하지 않고 전체 테이블을 읽는 것이 더 효율적입니다.

8.3.4 B-Tree 인덱스를 통한 데이터 읽기

어떤 경우에 인덱스를 사용하게 유도할지, 또는 사용하지 못하게 할지 판단하려면 MySQL이 어떻게 인덱스를 이용해서 실제 레코드를 읽어 내는지 알아야 합니다.

8.3.4.1 인덱스 레인지 스캔

인덱스 접근 방법 가운데 가장 대표적인 접근 방식 입니다.

다음 쿼리를 예시로 들어봅시다.

```
SELECT * FROM employees WHERE first_name BETWEEN 'Ebbe' AND 'Gad'
```

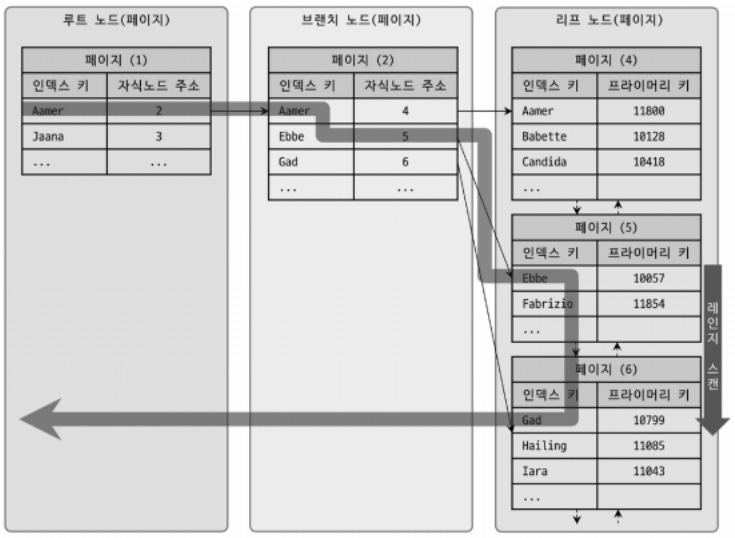


그림 8.8 인덱스를 이용한 레인지 스캔

인덱스 레인지 스캔은 검색해야 할 인덱스의 범위가 결정됐을 때 사용하는 방식입니다. 검색하려는 값의 수나 검색 결과 레코드 건수와 관계없이 레인지 스캔이라고 표현합니다.

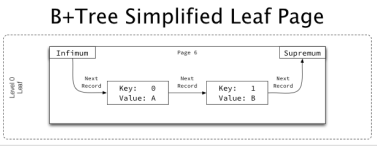
루트 노드에서부터 비교를 시작해 브랜치 노드를 거치고 최종적으로 리프 노드까지 찾아 들어가야만 비로소 필요한 레코드의 시작지점을 찾을 수 있다.

시작 지점을 찾았다면 그때 부터는 리프 노드의 순서대로 읽으면 됩니다. 스캔 중에 리프 노드의 끝까지 읽으면 리프 노드 간의 링크를 이용해서 다음 리프 노드를 찾아서 다시 스캔합니다.

B+Tree index structures in InnoDB

[This post refers to innodb_ruby version 0.8.8 as of February 3, 2014.] In On learning InnoDB: A journey to the core, I introduced the innodb_diagrams project to document the InnoDB internals, whic…

https://blog.jcole.us/2013/01/10/btree-index-structures-in-innodb/



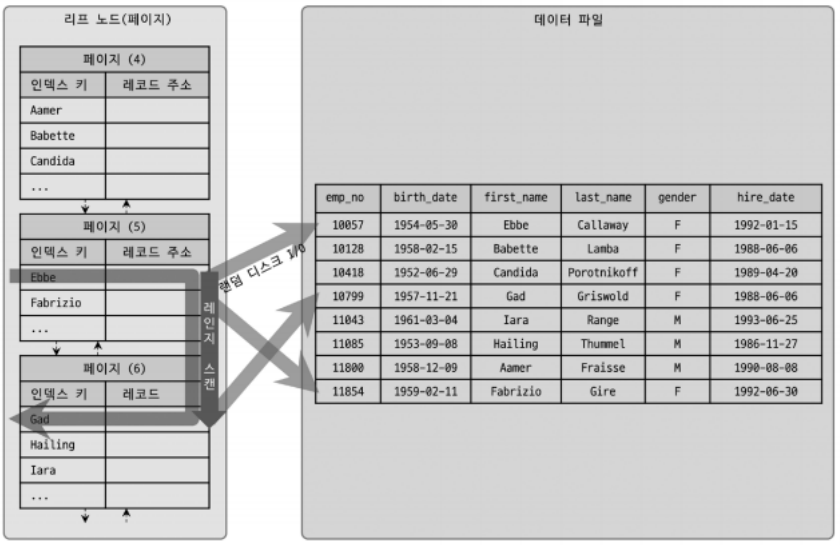


그림 8.9 인덱스 레인지 스캔을 통한 데이터 레코드 읽기

B-Tree 인덱스에서 루트와 브랜치 노드를 이용해 스캔 시작 위치를 검색하고, 그 지점부터 필요한 방향(오름차순 또는 내림차순)으로 인덱스를 읽어 나가는 과정을 위 그림에서 확인할 수 있습니다.

중요한 것은 어떤 방식으로 스캔하든 관계없이, 해당 인덱스를 구성하는 칼럼의 정순 또는 역순으로 정렬된 상태로 레코드를 가져온다는 것입니다.

이는 별도의 정렬 과정이 수반되는 것이 아니라 인덱스 자체의 정렬 특성 때문에 자동으로 그렇게 됩니다.

1. B-Tree 인덱스에서 검색 조건과 일치하는 데이터를 찾을 때, 레코드 주소를 리프 노드에서 가져와야 합니다.
2. 이 과정에서 레코드 한 건마다 랜덤 I/O가 발생하며, 검색 결과에 따라 랜덤 I/O 횟수가 달라집니다.
3. 예를 들어, 그림 8.9에서 3건의 레코드가 검색 조건과 일치한다면, 최대 3번의 랜덤 I/O가 필요합니다.
4. 이로 인해 인덱스를 통해 데이터 레코드를 읽는 작업은 비용이 많이 드는 작업으로 분류됩니다.
5. 만약 인덱스를 통해 읽어야 할 데이터 레코드가 전체 데이터의 20~25%를 넘어간다면, 직접 테이블의 데이터를 읽는 것이 더 효율적일 수 있습니다

인덱스 레인지 스캔은 다음과 같이 크게 3단계를 거칩니다.

1. 인덱스에서 조건을 만족하는 값이 저장된 위치를 찾는다, 이 과정을 인덱스 탐색이라고 합니다.
2. 1번에서 탐색된 위치부터 필요한 만큼 인덱스를 차례대로 쭉 읽습니다. 이 과정을 인덱스 스캔이라고 합니다.
3. 2번에서 읽어 들인 인덱스 키와 레코드 주소를 이용해 레코드가 저장된 페이지를 가져오고 최종 레코드를 읽어 옵니다.

쿼리가 필요로 하는 데이터에 따라 3번 과정은 필요하지 않을 수도 있는데 , 이를 커버링 인덱스라고 합니다. 커버링 인덱스로 처리되는 쿼리는 디스크의 레코드를 읽지 않아도 되기 때문에 랜덤 읽기가 상당히 줄어들고 성능은 그만큼 빨라집니다.

아래의 명령어를 통해 1번과 2번의 작업이 얼마나 수행됐는지를 확인할 수 있습니다.

```
SHOW STATUS LIKE 'HANDLER_%'
```

```
+----------------------------+---------+
| Variable_name              | Value   |
+----------------------------+---------+
| Handler_read_first         | 71      |
| Handler_read_last          | 1       |
| Handler_read_key           | 567     |
| Handler_read_next          | 3447233 |
| Handler_read_prev          | 19      |
```

Handler_read_key → 1번 단계가 실행된 횟수

Handler_read_next → 2번 단계를 정순 읽은 레코드 건수

Handler_read_prev →2번 단계를 역순으로 읽은 레코드 건수

8.4.3.2 인덱스 풀 스캔

인덱스의 처음부터 끝까지 모두 읽는 방식을 인덱스 풀 스캔이라고합니다.

대표적으로 쿼리의 조건절에 사용된 칼럼이 인덱스의 첫 번째 칼럼이 아닌 경우 인덱스 풀 스캔 방식이 사용됩니다.

ex)인덱스는(A,B,C)칼럼의 순서로 만들어져 있지만 쿼리의 조건절은 B칼럼이나 C칼럼으로 검색하는 경우

쿼리가 인덱스에 명시된 칼럼만으로 조건을 처리할 수 있는 경우 주로 이 방식이 사용됩니다.

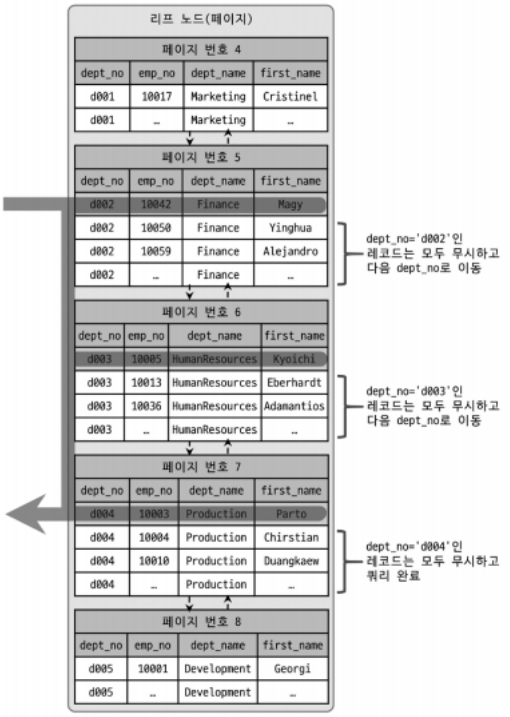
그림 8.10 인덱스 풀 스캔

1. 인덱스 리프 노드의 제일 앞 또는 제일 뒤로 이동
2. 인덱스의 리프 노드를 연결하는 링크드 리스트를 따라 처음부터 끝까지 스캔

이 방식은 인덱스 레인지 스캔보다는 빠르지 않지만 테이블 풀 스캔보다는 효율적입니다.

8.3.4.3 루스 인덱스 스캔

말 그대로 느슨하게 또는 듬성듬성하게 인덱스를 읽는 방식입니다.



.11 루스 인덱스 스캔(dept_name과 first_name 칼럼은 참조용으로 표시됨)

인덱스 레인지 스캔과 비슷하게 작동하지만 중간에 필요치 않은 인덱스 키 값은 무시하고 다음으로 넘어가는 형태로 처리됩니다.

일반적으로 GROUP BY 또는 집합 함수 가운데 MAX()또는 MIN() 함수에 대해 최적화를 하는 경우에 사용됩니다.

```
SELECT dept_no, MIN(emp_no)
FROM dept_emp
WHERE dept_no BETWEEN 'd002' AND 'd004'
GROUP BY dept_no
```

8.3.4.4 인덱스 스킵 스캔

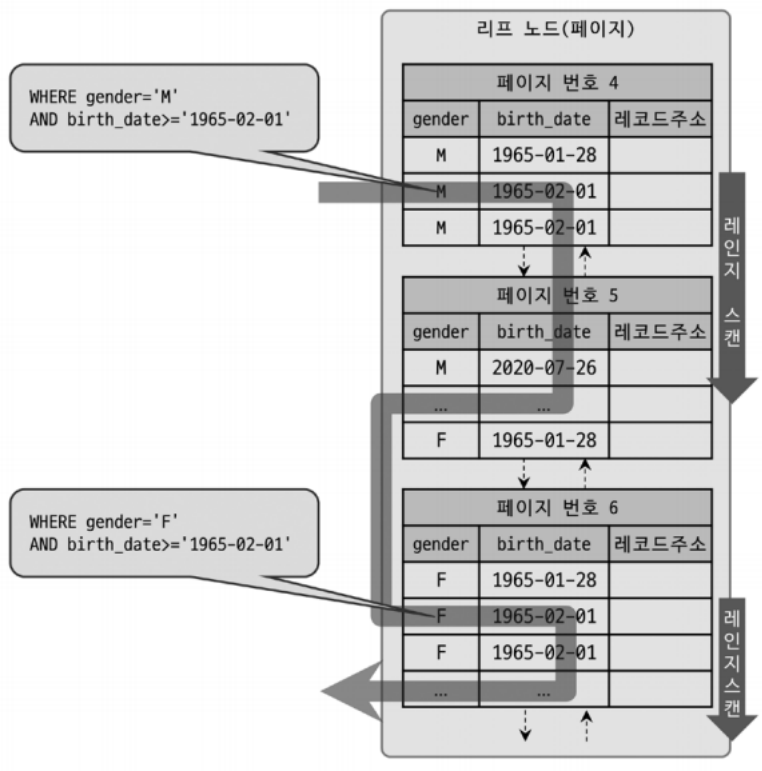


그림 8.12 인덱스 스킵 스캔

```
ALTER TABLE employees ADD INDEX ix_gender_birthdate(gender, birth_date);
```

위와 같은 다중 컬럼을 만들었다고 생각해 봅시다.

```
- // 인덱스를 사용하지 못하는 쿼리
ysql> SELECT * FROM employees WHERE birth_date>='1965-02-01';

- // 인덱스를 사용할 수 있는 쿼리
ysql> SELECT * FROM employees WHERE gender='M' AND birth_date>='1965-02-01';
```

예를 들어, gender 컬럼에는 'M'과 'F'라는 두 가지 값만 존재하고, birth_date 컬럼에는 많은 고유한 값들이 존재한다고 가정해봅시다. gender를 사용하지 않는 쿼리의 경우, 인덱스 스킵 스캔은 gender 컬럼을 건너뛰고 birth_date 값들에 직접 접근하여 검색을 최적화할 수 있습니다

```
SET optimizer_switch='skip_scan=off';
```

인덱스 스킵 스캔은 mysql 8.0버전에 새로이 도입된 기능이어서 아직 다음과 같은 단점이 존재합니다.

- WHERE 조건절에 조건이 없는 인덱스의 선행 칼럼의 유니크한 값의 개수가 적어야 합니다.
- 쿼리가 인덱스에 존재하는 칼럼으로만 처리 가능해야 합니다.

8.3.5 다중 컬럼(Multi-column) 인덱스

2개 이상의 컬럼을 가진 인덱스를 **다중 컬럼 인덱스**라고 합니다.

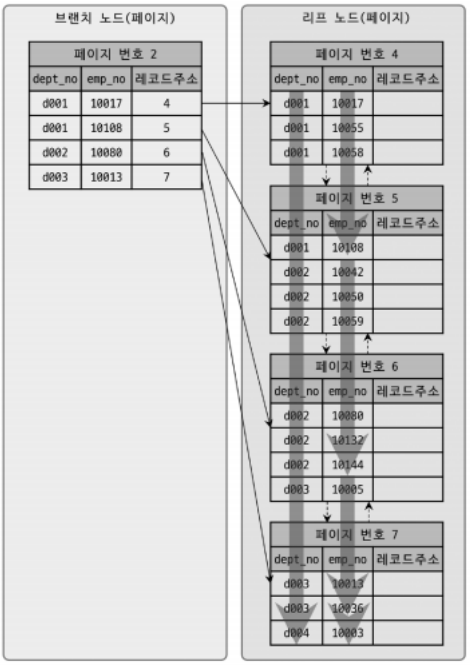


그림 8.13 다중 칼럼 인덱스

다중 컬럼 인덱스의 두 번째 칼럼은 첫 번째 칼럼에 의존해서 정렬돼 있습니다. 즉 두 번째 칼럼의 정렬은 첫 번째 칼럼이 똑같은 레코드에서만 의미가 있습니다. 그렇기 때문에 자주사용하는 칼럼을 앞에 위치 시켜야 합니다.

8.3.6 B-Tree 인덱스의 정렬 및 스캔 방향

8.3.6.1 인덱스의 정렬

일반적인 상용 DBMS는 인덱스를 생성하는 시점에 인덱스를 구성하는 각 칼럼의 정렬을 오름차순 또는 내림차순으로 설정할 수 있습니다.

5.7버전까지는 칼럼 단위로 정렬 순서를 혼합할 수 없었으나.

MySQL 8.0버전 부터는 다음과 같은 형태의 정렬 순서를 혼합한 인덱스도 생성할 수 있게 됐습니다.

```
CREATE INDEX it_teamname_userscore
ON employees (team_name ASC,user_score DESC)
```

이렇게 혼합해서 사용할 시 “각 팀별로 최고 점수를 가진 사용자는 누구인가?”에 대한 쿼리를 쉽고 빠르게 찾을 수 있습니다.

8.3.6.1.1 인덱스 스캔 방향

인덱스 생성 시점에 오름차순 또는 내림차순으로 정렬이 결정 되지만 쿼리가 그 인덱스를 사용하는 시점에 인덱스를 읽는 방향에 오름차순 또는 내림 차순 정렬 효과를 얻을 수 있습니다.(옵티마이저가 판단함)

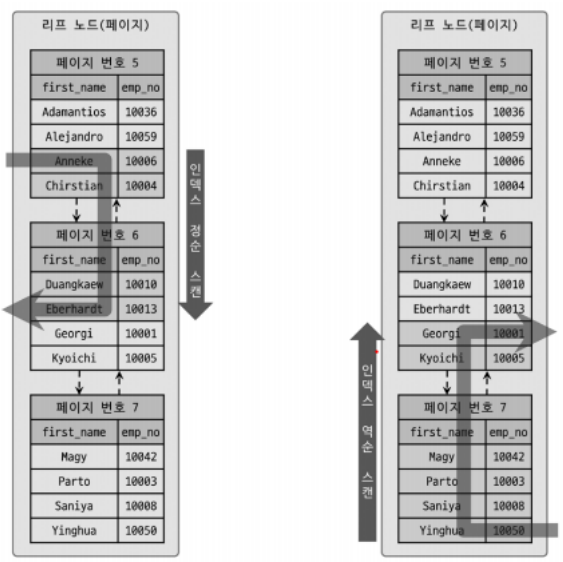


그림 8.14 인덱스의 오름차순(ASC)과 내림차순(DESC) 읽기

8.3.6.1.2 내림차순 인덱스

복합 인덱스에서 인덱스 역순 스캔이 인덱스 정순 스캔에 비해 느릴 수 밖에 없는 다음의 두 가지 이유가 있습니다.

- 페이지 잠금은 인덱스 정순 스캔(Foward index scan)에 적합한 구조입니다.
- 페이지 내에서 인덱스 레코드가 단방향으로만 연결된 구조입니다.

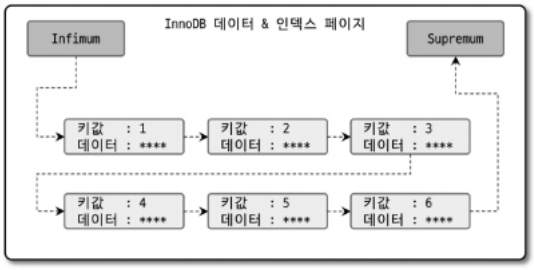


그림 8.16 InnoDB 페이지 내에서 레코드들의 연결

하지만 그렇다고 인덱스를 꼭 내림차순으로 정렬할 필요는 없습니다. 내림차순으로 읽는 쿼리가 많이 실행되는 상황에 사용하면 됩니다.

많은 쿼리가 앞쪽만 또는 뒤쪽만 집중적으로 읽어서 인덱스의 특정 페이지 잠금이 병목 될 것으로 예상된다면 쿼리에서 자주 사용되는 정렬 순서대로 인덱스를 생성하는 것이 잠금,병목 현상을 완화화는데 도움이 됩니다.

8.3.7 B-Tree 인덱스를 활용한 쿼리 최적화

B-Tree 인덱스는 데이터베이스 쿼리의 효율성을 극대화하는 데 중요한 역할을 합니다. `WHERE`, `GROUP BY`, `ORDER BY` 절 등에서 인덱스의 사용 여부와 방식을 파악하는 것이 쿼리 성능을 결정짓습니다.

8.3.7.1 조건별 인덱스 활용과 효율성

다중 컬럼 인덱스에서, 인덱스를 구성하는 컬럼들의 순서와 각 컬럼에 사용된 조건이 동등 비교(`=`), 크거나 같음(`>=`), 작거나 같음(`<=`)과 같은 범위 비교에 따라 인덱스의 활용도가 달라집니다. 예를 들어:

```
SELECT * FROM dept_emp WHERE dept_no = 'd002' AND emp_no >= 10144;
```

이 쿼리에 대해 다음 두 가지 인덱스를 고려해 봅시다:

- **케이스 A**: `INDEX(dept_no, emp_no)`
- **케이스 B**: `INDEX(emp_no, dept_no)`

케이스 A의 인덱스를 사용하면, `dept_no`가 'd002'이고 `emp_no`가 10144 이상인 레코드를 찾은 후, 더 이상 'd002'가 아닌 레코드는 살펴볼 필요가 없습니다. 그러나 케이스 B에서는 `emp_no`가 10144 이상인 모든 레코드를 검토하면서 각각이 'd002' 부서에 속하는지 확인해야 합니다. 이는 불필요하게 많은 레코드를 읽게 만들 수 있습니다.

8.3.7.2 인덱스 가용성

B-Tree 인덱스는 왼쪽 기준 정렬을 사용합니다. 따라서 왼쪽에서 시작하는 컬럼의 조건이 없으면 인덱스를 효율적으로 사용할 수 없습니다. 예를 들어:

- **케이스 A**: `SELECT * FROM employees WHERE first_name LIKE '%mer';`
  - 이 쿼리는 인덱스를 사용할 수 없습니다. 왜냐하면 `%`가 앞에 오기 때문에 인덱스 레인지 스캔을 할 수 없기 때문입니다.
- **케이스 B**: `SELECT * FROM dept_emp WHERE emp_no >= 10144;`
  - 이 쿼리는 `dept_no` 조건 없이 `emp_no`로만 필터링하기 때문에, 인덱스의 효율적 사용이 어렵습니다.

인덱스를 설계할 때는 쿼리의 조건과 맞게 인덱스를 생성하여, 데이터 접근 시간을 단축시키고 성능을 최적화해야 합니다. 이는 적절한 인덱스 선택과 정렬 순서를 통해 데이터베이스의 읽기와 쓰기 성능을 크게 향상시킬 수 있습니다.

8.3.7.3 가용성과 효율성 판단

기본적으로 B-Tree 인덱스의 특성상 다음 조건에서는 사용할 수 없습니다.(작업 범위 결정 조건으로 사용할 수 없습니다. 경우에 따라 체크 조건으로 사용할 순 있습니다.)

- NOT-EQUAL로 비교된 경우
- LIKE “%??” 형태로 문자열 패턴이 비교된 경우
- 스토어드 함수나 다른 연산자로 인덱스 칼럼이 변형된 후 비교된 경우
- NOT-DETERMINISTIC 속성의 스토어드 함수가 비교 조건에 사용된 경우
- 데이터 타입이 서로 다른 비교
- 문자열 데이터 타입의 콜레이션이 다른 경우

다중 칼럼으로 만들어진 인덱스의 사용 조건을 살펴 봅시다.

```
INDEX ix_test(column_1,column_2,column_3, ...,column_n)
```

- 작업 범위 결정 조건으로 인덱스를 사용하지 못하는 경우
  - column_1 칼럼에 대한 조건이 없는 경우
  - column_1 칼럼의 비교 조건이 위의 인덱스 사용 불가 조건 중 하나인 경우
- 작업 범위 결정 조건으로 인덱스를 사용하는 경우
  - column_1 ~column_(i-1)칼럼까지 동등 비교 형태로 비교
  - column_i 칼럼에 대해 다음 연산자중 하나로 비교

    -동등 비교

    -크다 작다 형태

    -LIKE로 좌측 일치 패턴(LIKE “주노%”)

8.4 R-Tree 인덱스

R-Tree 인덱스 알고리즘을 이용해 2차원의 데이터를 인덱싱하고 검색하는 목적의 인덱스를 `공간 인덱스(Spatial Index)`라고합니다.

SNS서비스가 GIS와 GPS에 기반을 둔 서비스를 가지고 있다고 가정합시다. 이러한 기능을 MySQL의 `공간 확장(Spatial Extension)`을 이용하면 간단하게 구현할 수 있습니다.

공간 확장에는 다음과 같이 크게 세 가지 기능이 포함돼 있습니다.

- 공간 데이터를 저장할 수 있는 데이터 타입
- 공간 데이터의 검색을 위한 공간 인덱스(R-Tree 알고리즘)
- 공간 데이터의 연산 함수(거리 또는 포함 관계의 처리)

8.4.1 구조 및 특성

MySQL은 공간 정보 저장 및 검색을 위해 여러 가지 `기하학적 도형(Geometry)` 정보를 관리할 수 있는 데이터 타입을 제공합니다.

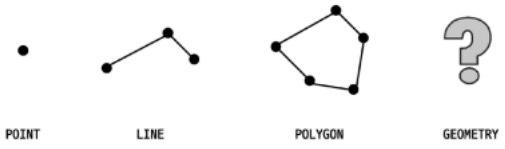


그림 8.19 GEOMETRY 데이터 타입

그림의 마지막에 있는 `GEOMETRY 타입`은 나머지 3개 타입의 슈퍼 타입으로, 나머지 객체를 모두 저장할 수 있습니다.

R-Tree를 이애하기 위해 `MBR(Minimum Bounding Rectangle)`을 알아봅시다. MBR이란 도형을 감싸는 최소 크기의 사각형을 의미합니다. 이 사각형들의 포함 관계를 B-Tree 형태로 구현한 인덱스가 R-Tree 인덱스입니다.

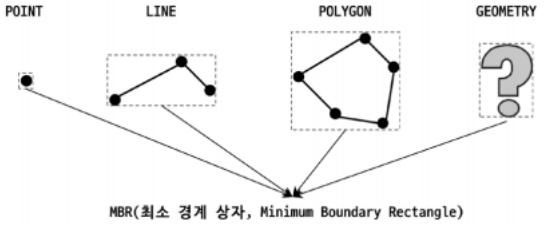


그림 8.20 최소 경계 상자(MBR, Minimum Bounding Rectangle)

간단히 R-Tree의 구조를 살펴보자면 다음과 같습니다.

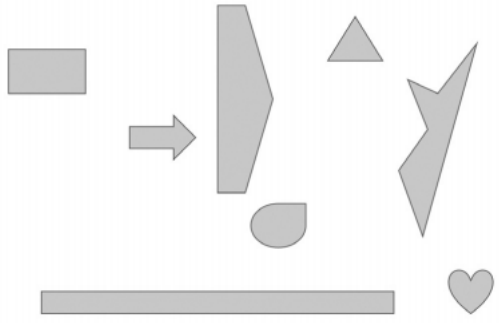


그림 8.21 공간(Spatial) 데이터

위 와같은 도형들의 MBR을 3개의 레벨로 나눠서 표현하자면 다음과 같습니다.

- 최상위 레벨: R1,R2
- 차상위 레벨: R3,R4,R5,R6
- 최하위 레벨: R7~R14

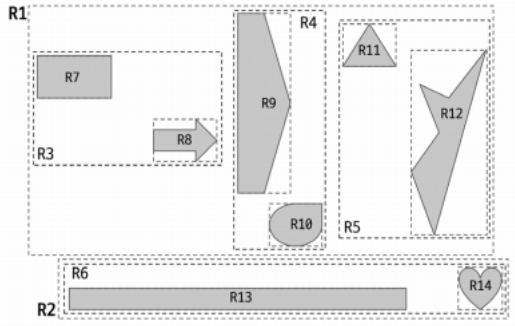


그림 8.22 공간(Spatial) 데이터의 MBR

최상위 MBR은 R-Tree의 루트 노드에 저장되는 정보이며,차상위 그룹 MBR은 R-Tree의 브랜치 노드 마지막으로 각 도형의 객체는 리프 노드에 저장됩니다.

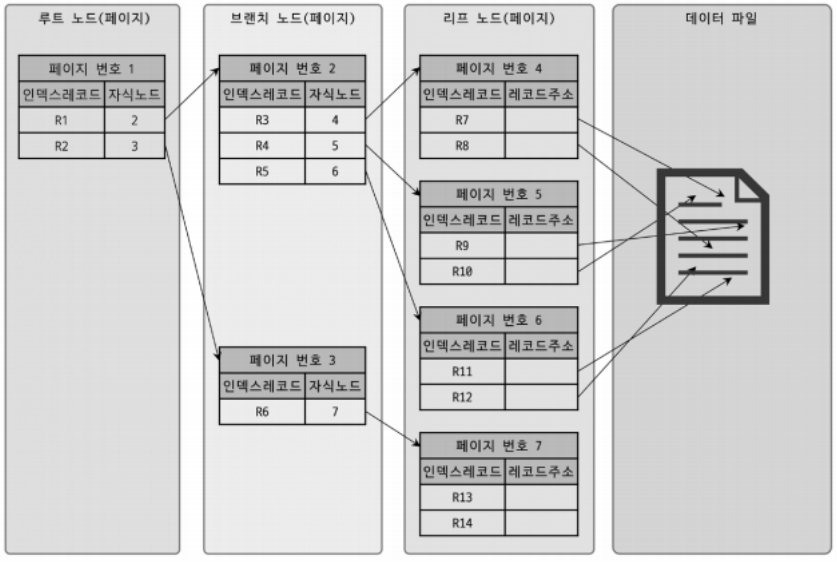


그림 8.23 공간(R-Tree, Spatial) 인덱스 구조

8.4.2 R-Tree 인덱스의 용도

`공간(Spatial) 인덱스`라고도 합니다. 일반적으로 **WGS84(GPS)** 기준의 **위도**, **경도 좌표** 저장에 주로 사용됩니다. 하지만 위도 경도 좌표 분 아니라 CAD/CAM 스포트웨어 또는 회로 디자인 등과 같이 좌표 시스템에 기반을 둔 정보에 대해 적용할 수 있습니다.

예를들어

1. **최상위 노드에 5KM 도시 저장**: 이 경우, 최상위 노드의 MBR은 전체 도시의 영역을 포함합니다.
2. **도시 내 건물 좌표 저장**: 도시 내의 각 건물은 자신의 MBR을 가지고, 이는 건물을 완전히 포함하는 최소 크기의 직사각형입니다. 이러한 건물들의 MBR은 R-Tree의 하위 노드들에 저장됩니다.

3. **특정 영역 내 건물 검색**: 5KM 도시 내에서 특정 영역(예: 특정 거리 내의 건물)을 찾을 때, R-Tree는 이 영역에 해당하는 MBR들을 빠르게 필터링하여 해당하는 건물들을 찾아냅니다.

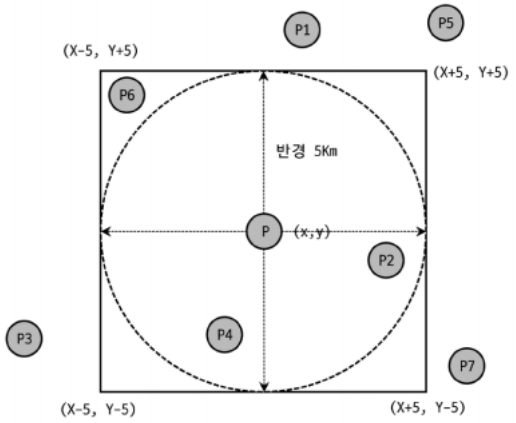


그림 8.24 특정 지점을 기준으로 사각 박스 이내의 위치를 검색

사용법은 257PAGE 참조 필요할때 !

8.5 전문 검색 인덱스

MySQL의 B-Tree 인덱스는 실제 칼럼의 값이 1MB이더라도 1MB 전체의 값을 인데스 키로 사용하는것이 아니라, 1000byte(MyISAM) 또는 3072바이트(InnoDB)까지만 잘라서 인덱스 키로 사용합니다.

문서 전체에 대한 분석과 검색을 위한 인덱싱 알고리즘을 **전문 검색(Full Text serch) 인덱스** 라고 하는데,전문 검색 인덱스는 일반화된 기능의 명칭이지 전문 검색 알고리즘의 이름을 지칭하는건 아닙니다.

8.5.1 인덱스 알고리즘

전문 검색 인덱스의 기법 크게 두 가지가 있습니다.

- 어근 분석
- n-gram 분석 알고리즘

8.5.1.1 어근 분석 알고리즘

MySQL 서버의 전문 검색 인덱스는 다음과 같은 두 가지 중요한 과정을 거쳐서 색인 작업이 수행됩니다.

불용어 처리: 가치가 없는 단어를 필터링 제거하는 작업

- 불용어의 개수는 많지 않기 때문에 알고리즘을 구현한 코드에 모두 상수로 정의해서 사용하는 경우가 많습니다.
- 유연성을 위해 불용어 자체를 데이터베이스화해서 사용자가 추가하거나 삭제할 수 있게 구현하는 경우도 있습니다.
- MySQL 서버는 불용어가 소스코드에 정의돼 있지만, 이를 무시하고 사용자가 별도로 불용어를 정의할 수 있는 기능을 제공합니다.

어근 분석 : 검색어로 선정된 단어의 뿌리인 원형을 찾는 작업

- 오픈소스 형태소 분석 라이브러리 MeCab을 플러그인 형태로 사용할 수 있게 지원합니다.
- 서구권 언어는 MongoDb에서 사용되는 Snowball이라는 오픈소스가 존재합니다.
- 한국어는 일본어와 많이 비슷하기 때문에 MeCab을 이용해 한글 분석이 가능합니다.

8.5.1.2 n-gram 알고리즘

MeCab을 위한 형태소 분석은 매우 전문적이기 때문에 많은 노력과 시간이 필요합니다. 이를 보완하기 위해 **n-gram 알고리즘**이 도입됐습니다.

n-gram: :키워드를 검색해내기 위한 인덱싱 알고리즘

- 본문을 무조건 몇 글자씩 잘라서 인덱싱하는 방법입니다.
- 만들어진 인덱스의 크기가 형태소 분석보다 상당히 큽니다.
- n은 인덱싱할 키워드의 숫자이며 2글자 단위의 2-gram 방식이 많이 사용됩니다.

2-gram 알고리즘을 사용한 토큰 분리

다음의 문장을 토큰을 분리해 봅시다. `To be or not to be. That is the question`

표의 전체 내용은 다음과 같습니다:

단어	bi-gram(2-gram) 토큰						
To	To						
be	be						
or	or						
not	no	ot					
to	to						
be	be						
That	Th	ha	at				
is	is						
the	th	he					
question	qu	ue	es	st	ti	io	on

만약 단어가 10글자 단어라면 그 단어는 2-gram 알고리즘에서는 (10-1)개의 토큰으로 구분됩니다. 이렇게 구분된 토큰을 인덱스에 저장하기만 하면됩니다. 이때 중복된 토큰은 하나의 인덱스 엔트리로 병합되어 저장 됩니다.

MySQL 서버는 이렇게 생성된 토큰들에 대해 불용어를 걸러내는 작업을 수행하는데, 이때 불용어와 동일하거나 불용어를 포함하는 경우 걸러서 버립니다.

아래의 명령어를 통해 확인할 수 있습니다.

```
SELECT * FROM information_schema.innodb_ft_default_stopword;
```

최종적으로 저장되는 인덱스 엔트리는 다음과 같습니다.

입력	불용어 일치	불용어 포함	출력(최종 인덱스 등록)
at		O	
be	O		
be	O		
es			et
ha		O	
he			he
io		O	
is		O	
no			no
on	O		
or	O		
ot			ot
qu			qu
st			st
Th			Th
th			th
ti		O	
To	O		
to	O		
ue			ue

8.5.1.3 불용어 변경 및 삭제

불용어 처리가 사용자를 혼란스럽게 할 수도 있습니다. 그래서 불용어 처리 자체를 완전히 무시하거나 MySQL 서버에서 내장된 불용어 대신 사용자가 직접 불용어를 등록하는 방법을 권장합니다.

ex : Behance 사이트의 be

전문 검색 인덱스의 불용어 처리 무시

두가지의 불용어 처리 무시 방법이 존재합니다.

스토리지 엔진에 관계없이 MySQL 서버의 모든 전문 검색 인덱스에 대해 불용어를 완전히 제거하는 방법

- MySQL 서버의 설정 파일(my.cnf)의 fs_stopword_file 시스템 변수에 빈 문자열을 설정하면 됩니다. `ft_stopword_file=''`

InnoDB 스토리지 엔진을 사용하는 테이블의 전문 검색 인덱스에 대해서만 불용어 처리를 무시할 수 있습니다.

- `SET GLOBALA innodb_ft_enable_stopword=OFF` 의 명령어를 사용해 동적인 시스템 변수로 InnoDB 스토리지 엔진에서만 불용어 처리 무시가 가능합니다.

사용자 정의 불용어 사용

두 가지의 MySQL 사용자 정의 불용어 사용 방법이 있습니다.

불용어 목록을 파일로 저장하고, MySQL 서버 파일에서 파일의 경로를 ft_stopword_file 설정에 등록하면 됩니다.

- `ft_stopword_file='/data/my_cutom_stopword.txt'`

두 번째 방법은 InnoDb 스토리지 엔진을 사용하는 전문 검색 엔진에서만 사용할 수 있는데, 불용어의 목록을 테이블로 저장하는 방식입니다.

- Innodb_ft_server_stopword_table 시스템 변수에 불용어 테이블을 설정하바니다.
- 불용어 목록을 변경한 이후 전문 검색 인덱스가 생성돼야만 변경된 불용어가 적용됩니다.

```
CREATE TABLE my_stopword(value VARCHAR(30)) ENGINE = INNODB;
INSERT INTO my_stopword(value) VALUES ('MySQL');

SET GLOBAL innodb_ft_server_stopword_table='mydb/my_stopword';
ALTER TABLE tb_bi_gram
        ADD FULLTEXT INDEX fx_title_body(title, body) WITH PARSER ngram;
```

8.5.2 전문 검색 인덱스의 가용성

전문 검색 인덱스를 사용하려면 반드시 다음 두 가지 조건을 갖춰야 합니다.

- 쿼리 문장이 전문 검색을 위한 문법(WATCH… AGAINST …)을 사용
- 테이블이 전문 검색 대상 칼럼에 대해서 전문 인덱스 보유

8.6 함수 기반 인덱스

칼럼의 값을 변형해서 만들어진 값에 대해서 인덱스를 구축해야 하는 상황에 함수기반의 인덱스를 활용할 수 있습니다. MySQL 8.0 버전부터 지원합니다.

- 가상 칼럼을 이용한 인덱스
- 함수를 이용한 인덱스

인덱싱할 값을 계산하는 과정의 차이만 있을 뿐, 구조는 B-Tree와 동일합니다.

8.6.1 가상 칼럼을 이용한 인덱스

```
CREATE TABLE user(
        user_id BIGINT,
        first_name VARCHAR(10),
        last_name VARCHAR(10),
        PRIMARY KEY (user_id)
)
```

위와 같은 테이블이 존재한다고 가정할 때 first_name과 last name을 합쳐서 검색해야 하는 상황이면 모든 레코드에 full_name이라는 칼럼을 추가해야 할 것입니다.

하지만 8.0 부터는 가상 칼럼을 추가하고 그 가상 칼럼에 인덱스를 생성 할 수 있습니다.

```
ALTER TABLE user
ADD full_name VARCHAR(30) AS (CONCAT(first_name,' ',last_name)) VIRTUAL,
ADD INDEX ix_fullname(full_name);
```

하지만 가상 칼럼은 테이블에 새로운 칼럼을 추가하는 것과 같은 효과를 내기 때문에 실제 테이블의 구조가 변경된다는 단점이 존재합니다.

8.6.2 함수를 이용한 인덱스

MySQL 8.0 버전부터는 다음과 같이 테이블의 구조를 변경하지 않고,함수를 직접 사용하는 인덱스를 생성할 수 있게 됐습니다.

```
CREATE TABLE USER (
    user_id BIGINT,
    first_name VARCHAR(10),
    last_name VARCHAR(10),
    PRIMARY KEY (user_id)
    INDEX ix_fullname ((CONCAT(first_name,' ',last_name)))
);
```

주의점은 WHERE 조건절에 함수 기반 인덱스에 명시된 표현식을 그대로 사용하지 않으면 옵티마이저가 다른 표현식으로 인식해 함수 기반 인덱스를 사용할 수 없습니다.

```
EXPLAIN SELECT * FROM user WHERE CONCAT(first_name,' ',last_name) = 'lee geonhoe';
```

8.7 멀티 밸류 인덱스

멀티 밸류 인덱스는 하나의 데이터 레코드가 여러 개의 키 값을 가질 수 있는 형태의 인덱스입니다.

최근 RDBMS들이 JSON 데이터 타입을 지원하기 시작하면서 JSON의 배열 타입의 필드에 저장된 원소(element)들에 대한 인덱스 요건이 발생했습니다.

```
CREATE TABLE user (
    user_id BIGINT AUTO_INCREMENT PRIMARY KEY,
    first_name VARCHAR(10),
    last_name VARCHAR(10),
    credit_info JSON,
    INDEX mx_creditscores ( (CAST(credit_info->'$.credit_scores' AS UNSIGNED ARRAY)))
);
INSERT INTO user VALUES (1, 'harris', 'lee', '{"credit_scores":[360, 353, 351]}');
```

멀티 밸류 인덱스를 활용하기 위해서는 일반적인 조건 방식을 사용하면 안되며, 반드시 다음 함수들을 이용해야 옵티마이저가 인덱스를 활용한 실행 계획을 수립합니다.

- MEMBER OF()
- JSON_CONTAINS()
- JSON_OVERLAPS()

EX)

```
mysql> SELECT * FROM user WHERE 360 MEMBER OF(credit_info->'$.credit_scores');
+---------+------------+-----------+-----------------------------------+
| user_id | first_name | last_name | credit_info                       |
+---------+------------+-----------+-----------------------------------+
|       1 | Matt       | Lee       | {"credit_scores": [360, 353, 351]} |
+---------+------------+-----------+-----------------------------------+

mysql> EXPLAIN SELECT * FROM user WHERE 360 MEMBER OF(credit_info->'$.credit_scores');
+----+-------------+-------+------+---------------+---------+-------+-------------+
| id | select_type | table | type | key           | key_len | ref   | Extra       |
+----+-------------+-------+------+---------------+---------+-------+-------------+
|  1 | SIMPLE      | user  | ref  | mx_creditscore | 9       | const | Using where |
+----+-------------+-------+------+---------------+---------+-------+-------------+
```

8.8 클러스터링 인덱스

MySQL 서버에서 클러스터링은 테이블의 레코드를 비슷한 것(프라이머리 키 기준)들끼리 묶어 저장하는 형태로 구현됩니다.

이는 주로 비슷한 값들을 동시에 조회하는 경우가 많다는 점에서 착안한 것입니다. (InnoDB만 지원합니다.)

8.8.1 클러스터링 인덱스

클러스터링 인덱스는 테이블의 프라이머리 키에 대해서만 적용됩니다.

프라이머리 키값이 변경된다면 레코드의 저장 위치 또한 바뀌게 됩니다. 그렇기 때문에 신중하게 프라이머리 키를 결정해야 합니다.

클러스터링 인덱스는 프라이머리 키 값에 의해 레코드의 저장 위치가 결정되므로 사실 인덱스 알고리즘이라기보다는 테이블 레코드의 저장 방식입니다.

- InnoDB와 같이 항상 클러스터링 인덱스로 저장되는 테이블은 프라이머리 키 기반의 검색이 매우 빠릅니다.
- 레코드의 저장이나 프라이머리 키의 변경이 상대적으로 느립니다.

구조 자체는 B-Tree와 비슷하지만 세컨더리 인덱스를 위한 B-Tree 리프 노드와는 달리 클러스트링 인덱스의 리프 노드에는 레코드의 모든 칼럼이 같이 저장돼 있습니다.

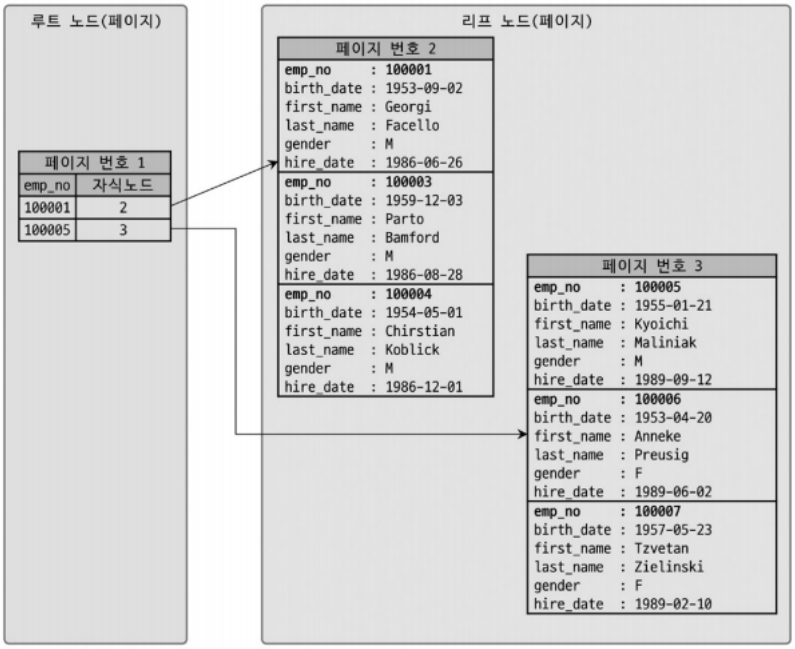


그림 8.25 클러스터링 테이블(인덱스) 구조

즉 클러스트링 테이블은 그 자체가 하나의 거대한 인덱스 구조로 관리되는 것입니다.

그렇다면 **프라이머리키가 없는 테이블**은 어떻게 클러스터링 테이블로 구성될까요?

프라이머리키가 없는 경우에는 InnoDB 스토리지 엔진이 아래와 같은 우선순위대로 프라이머리 키를 대체할 칼럼을 선택합니다.

1. 프라이머리 키가 있으면 프라이머리키 선택
2. NOT NULL 옵션의 UNIQUE INDEX 중에서 첫 번째 인덱스를 클러스터링 키로 선택
3. 자동으로 유니크한 값을 가지도록 증가되는 칼럼을 내부적으로 추가한 후, 클러스터링 키로 선택

하지만 이는 아무 의미없는 숫자 값으로 클러스팅 되는것이며 어떠한 혜택도 주지않는다..(무 의미)

8.8.2 세컨더리 인덱스에 미치는 영향

InnoDB 테이블에서 세컨더리 인덱스가 실제 레코드가 저장된 주소를 가지고 있다면 어떻게 될까요?

- 클러스터링 키 값이 변경될 때마다 데이터 레코드의 주소가 변경되고 그때마다 해당 테이블의 모든 인덱스에 저장된 주솟값을 변경해야 합니다.

위와같은 오버헤드를 제거하기 위해 InnoDB 테이블의 모든 세컨더리 인덱스는 해당 레코드가 저장된 주소가 아닌 프라이머리 키 값을 저장하도록 구현돼 있습니다.

8.8.3 클러스터링 인덱스의 장단점

장점

- 프라이머리 키(클러스터링 키)로 검색할 때 성능이 매우 빠릅니다.
- 테이블의 모든 세컨더리 인덱스가 프라이머리 키를 가지고 있기 때문에 인덱스만으로 처리될 수 있는 경우가 많습니다..

단점

- 테이블의 모든 세컨더리 인덱스가 클러스터링 키를 갖기 때문에 클러스터링 키 값의 크기가 클 경우 전체적으로 인덱스의 크기가 커집니다.
- 세컨더리 인덱스를 통해 검색할 때 프라이머리 키로 다시 한 번 검색해야 하므로 처리 성능이 느립니다..
- INSERT할 때 프라이머리 키에 의해 레코드의 저장 위치가 결정되므로 성능이 느립니다..
- 프라이머리 키를 변경할 때 레코드를 DELETE하고 다시 INSERT 해야합니다..

즉 빠른 읽기와, 느린 쓰기의 특징을 가집니다.

8.8.4 클러스터링 테이블 사용 시 주의사항

8.8.4.1 클러스터링 인덱스 키의 크기

프라이머리 키의 크기가 커지면 세컨더리 인덱스도 자동으로 크기가 커집니다. 세컨더리 인덱스가 많아 질수록 메모리를 더 많이 잡아먹습니다.

8.8.4.2 프라이머리 키는 AUTO_INCREMENT보다는 업무적인 칼럼으로 생성

- **의미 있는 식별자**: 업무에 중요한 의미를 가진 고유 식별자를 프라이머리 키로 사용합니다.
- **업무 프로세스 연계**: 키가 실제 업무 프로세스나 데이터의 특성과 직접 연결됩니다.
- **데이터 관리 용이성**: 업무적인 칼럼을 키로 사용하면 데이터 관리와 이해가 더 쉬워집니다.
- **AUTO_INCREMENT 대체**: 순차적이고 의미 없는 AUTO_INCREMENT 값 대신, 데이터의 의미를 반영하는 칼럼을 사용합니다.

8.8.4.3 프라이머리 키는 반드시 명시할 것

프라이머리 키가 없다면 AUTO_INCREMENT 칼럼을 이용해서라도 프라이머리 키는 생성하는 것을 권장합니다.

프라이머리 키를 정의하지 않으면 사용자가 접근할 수 없는 일련번호를 자동으로 칼럼에 추가하기 때문에 사용자가 접근할 수 있는 AUTO_INCREMENT가 낫습니다. 또한 ROW 기반의 복제나 InnoDb Cluster에서는 모든 테이블이 프라이머리 키를 가져야만 하는 정상적인 복제 성능을 보장합니다.

8.8.4.4 AUTO-INCREMENT 칼럼을 인조 식별자로 사용할 경우

하지만, 프라이머리 키의 크기가 길어도 세컨더리 인덱스가 필요치 않다면 그대로 프라이머리 키를 사용하는 것이 좋습니다.

세컨더리 인덱스도 필요한데 프라이머리 키도 길면AUTO_INCREMENT 칼럼을 추가하고 이를 프라이머리 키로 설정하면됩니다. 이와같이 프라이머리 키를 대체하기 위해 인위적으로 추가된 프라이머리 키를 인조 식별자(Surrogate key)라고 합니다.

로그 테이블 같이 조회보다는 **INSERT 위주**로 동작되는 테이블은 인조 식별자를 프라이머리 키로 설정하는 것이 성능에 더 도움이 된다.

8.9 유니크 인덱스

유니크 인덱스는 인덱스라기보다는 제약 조건에 가깝습니다.

테이블이나 인덱스에 같은 값이 2개 이상 저장될 수 없음을 의미하며 특징은 다음과 같습니다.

- 인덱스 없이 유니크 제약만 설정할 방법은 없습니다.
- Null은 특정 값이 아니므로 2개 이상 저장될 수 있습니다.
- 프라이머리 키는 기본적으로 null이 허용되지 않는 유니크 속성이 자동으로 부여됩니다.

8.9.1 유니크 인덱스와 일반 세컨더리 인덱스의 비교

8.9.1.1 인덱스 읽기

유니크 하지않은 세컨더리 인덱스에서 한 번 더 해야 하는 작업은 디스크 읽기가 아니라 CPU에서 칼럼값을 비교하는 작업(`WHERE ~~`)이기 때문에 이는 성능상 영향이 거의 없습니다.

유니크하지 않은 세컨더리 인덱스는 중복된 값이 허용되므로 읽어야 할 레코드가 많아서 느린 것이지,인덱스 자체의 특성 때문에 느린 것이 아닙니다.

(즉 레코드가 많아서 오래걸리는게 유니크 인덱스의 특성 때문은 아니다..?)

8.9.2 유니크 인덱스 사용 시 주의사항

하나의 테이블에서 같은 칼럼에 유니크 인덱스는 일반 인덱스와 같은 역할을 하므로 일반 인덱스를 굳이 따로 둘 필요는 없습니다.

마찬가지로 유니크인덱스와 프라이머리 키를 동일하게 생성할 필요가 없습니다.

8.10 외래키

MySQL에서 외래키는 InnoDB 스토리지 엔진에서만 생성할 수 있으며, 외래키 제약이 설정되면 자동으로 연관되는 테이블의 칼럼에 인덱스 까지 생성됩니다. 외래키가 제거되지 않은 상태에서 자동으로 생성된 인덱스를 삭제할 수 없습니다.

InnoDB 외래키 관리에는 중요한 두 가지 특징이 있습니다.

- 테이블의 변경(쓰기 잠금)이 발생하는 경우에만 잠금 경합(잠금 대기)이 발생합니다.
- 외래키와 연관되지 않은 칼럼의 변경은 최대한 잠금 경합을 발생시키지 않는다.

부모 테이블이 업데이트로 인해 잠금 상태일 때, 자식 테이블의 변경(외래키 관련)은 부모 테이블의 잠금이 해제될 때까지 대기합니다.

반대로, 자식 테이블이 잠금 상태일 때는 부모 테이블의 변경 작업(예: 삭제)이 자식 테이블의 잠금이 해제될 때까지 대기합니다.

8.10.1 자식 테이블의 변경이 대기하는 경우

작업 번호	커넥션-1	커넥션-2
1	`BEGIN;`	
2	`UPDATE tb_parent` `SET fd='changed-2' WHERE id=2;`	
3		`BEGIN;`
4		`UPDATE tb_child` `SET pid=2 WHERE id=100;`
5	`ROLLBACK;`	
6		`Query OK, 1 row affected (3.04 sec)`

반대로, 자식 테이블이 잠금 상태일 때는 부모 테이블의 변경 작업(예: 삭제)이 자식 테이블의 잠금이 해제될 때까지 대기합니다.

8.10.2 부모 테이블의 변경 작업이 대기하는 경우

작업 번호	커넥션-1	커넥션-2
1	BEGIN;	
2	UPDATE tb_child SET fd='changed-100' WHERE id=100;	
3		BEGIN;
4		DELETE FROM tb_parent WHERE id=1;
5	ROLLBACK;	
6		Query OK, 1 row affected (6.09 sec)