Real MySQL - 인덱스

디스크 읽기 방식

우선 `Random I/O`와 `Sequential I/O`에 대해서 간단히 알아보고 인덱스를 살펴보자.

최근에는 SSD 드라이브가 많이 활용되고 있지만, 그럼에도 여전히 데이터 저장 매체는 컴퓨터에서 가장 느린 부분이다. 그렇기 때문에 데이터베이스의 성능 튜닝은 어떻게 디스크 I/O를 줄이느냐가 관건일때가 많다.

HDD와 SSD

컴퓨터에서 CPU나 메모리 같은 주요 장치는 대부분 전자식 장치이다. 하지만 하드 디스크 드라이브는 기계식 장치이고, 그렇기 때문에 서버에서는 항상 디스크 장치가 병목이 된다. 그리고 이러한 기계식 하드 디스크 드라이브를 대체하기 위해 전자식 저장 매체인 SSD(Solid State Drive)가 많이 출시되고 있다.

SSD는 기존 하드 디스크 드라이브에서 데이터 저장용 플래터를 제거하고 그 대신 플래시 메모리를 장착하고 있다. 그렇기 때문에 원판을 기계적으로 회전시킬 필요가 없고 전원이 공급되지 않아도 데이터가 삭제되지 않는다.

컴퓨터에 있는 주요 장치의 초당 처리 횟수를 비교하면 아래와 같다:

- CPU (1,000,000,000 op/sec) > DRAM (100,000,000 op/sec) > SSD (100,000 op/sec) > HDD (200 op/sec)

디스크의 헤더를 움직이지 않고 한 번에 많은 데이터를 읽는 순차 I/O에서는 SSD가 HDD보다 조금 빠르거나 거의 비슷하지만, 랜덤 I/O를 통해 작은 데이터를 읽고 쓰는 작업에서는 압도적으로 빠르다. 그렇기 때문에 일반적인 웹 서비스(OLTP) 환경의 데이터베이스에서는 SSD가 훨씬 빠르다.

랜덤 I/O와 순차 I/O

랜덤 I/O와 순차 I/O의 차이는 디스크 헤더의 위치 이동이 얼마나 많은가이다. 순차 I/O는 3개의 페이지를 디스크에 기록하기 위해 한번의 헤더 이동이 있지만, 랜덤 I/O는 3개의 페이지를 쓰기 위해 3번의 헤더 이동이 있다.

한마디로 디스크의 성능은 디스크 헤더의 위치 이동 없이 얼마나 많은 데이터를 한 번에 기록하느냐에 따라 결정된다.

디스크 원판을 가지지 않은 SSD는 랜덤 I/O와 순차 I/O의 차이가 없을 것이라고 생각할 수 있지만, 실제로 SSD에서도 랜덤 I/O는 순차 I/O보다 전체 throughput이 떨어진다.

인덱스란?

만약 원하는 검색 결과를 가져오기 위해 데이터베이스 테이블의 모든 테이블을 검색해서 가져오려면 시간이 오래 걸릴 것이다. 그렇기 때문에 칼럼의 값과 해당 레코드가 저장된 주소를 키와 값의 쌍으로 삼아 인덱스를 만들어 둔다. 그리고 이러한 인덱스들은 칼럼의 값을 주어진 순서로 미리 정렬해서 보관한다.

그리고 이렇게 미리 정렬되어 보관되는 성질 때문에 DBMS에서 **인덱스는 데이터의 저장(INSERT, UPDATE, DELETE) 성능을 희생하고 그 대신 데이터의 읽기 속도를 높이는 기능이다.** 그렇기 때문에 테이블의 인덱스를 하나 더 추가할지 말지는 데이터의 저장 속도를 어디까지 희생할 수 있는지, 읽기 속도를 얼마나 더 빠르게 만들어야 하느냐에 따라 결정해야 한다.

그리고 이러한 인덱스를 역할별로 구분해 본다면 **프라이머리 키**와 **보조 키**로 구분할 수 있다.

- 프라이머리 키는 레코드를 대표하는 칼럼의 값으로 만들어진 인덱스이다. 이 칼럼은 테이블에서 해당 레코드를 식별할 수 있는 기준값이 되기 때문에 식별자라고도 부른다. 프라이머리ㄴ 키는 NULL 값을 허용하지 않으며 중복을 허용하지 않는다.
- 프라미어리 키를 제외한 나머지 모든 인덱스는 세컨더리 인덱스로 분류한다. 유니크 인덱스는 프라이머리 키와 성격이 비슷하고 프라이머리 키를 대체해서 사용할 수도 있다고 해서 대체 키라고도 하는데, 별도로 분류하기도 하고 그냥 세컨더리 인덱스로 분류하기도 한다.

데이터 저장 방식은 상당히 다양한 종류가 존재하지만 대표적으로 **B-Tree 인덱스와 Hash 인덱스**로 구분할 수 있다.

- B-Tree 알고리즘은 가장 일반적으로 사용되는 인덱스 알고리즘으로서, 칼럼의 값을 변형하지 않고 원래의 값을 이용해 인덱싱하는 알고리즘이다.
- Hash 인덱스 알고리즘은 칼럼의 값으로 해시값을 계산해서 인덱싱하는 알고리즘으로, 매우 빠른 검색을 지원한다. 하지만 값을 변형해서 인덱싱하므로 Prefix(전방) 일치와 같이 값의 일부만 검색하거나 범위를 검색할 때는 해시 인덱스를 사용할 수 없다. 메모리 기반의 데이터베이스에서 많이 사용한다.

B-Tree 인덱스

`Balanced-Tree`는 데이터베이스의 인덱싱 알고리즘 가운데 가장 일반적으로 사용되고, 가장 먼저 도입된 알고리즘이다. 일반적으로 DBMS에서는 주로 `B+-Tree` 또는 `B*-Tree`가 사용된다. B-Tree는 칼럼의 원래 값을 변형시키지 않고 인덱스 구조체 내에서는 항상 정렬된 상태로 유지한다.

구조 및 특성

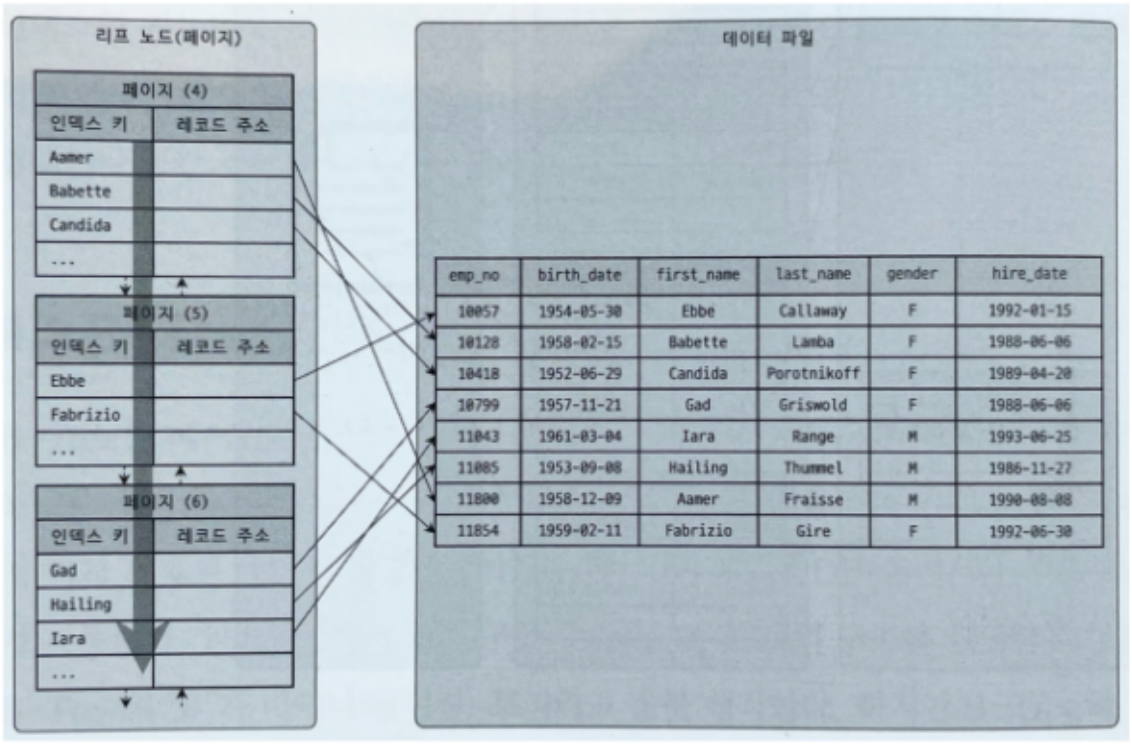
B-Tree는 트리 구조의 최상위에 하나의 Root Node가 존재하고 그 하위에 자식 노드가 붙어 있는 형태다. 트리 구조의 가장 하위에 있는 노드를 Leaf Node라 하고, 트리 구조에서 루트 노드도 아니고 리프 노드도 아닌 중간의 노드를 Branch Node라고 한다.

데이터베이스에서 인덱스와 실제 데이터가 저장된 데이터는 따로 관리가 되는데, 인덱스의 리프 노드는 항상 실제 데이터 레코드를 찾아가기 위한 주솟값을 가지고 있다. 인덱스는 테이블의 키 칼럼만 가지고 있으므로 나머지 칼럼을 읽으려면 데이터 파일에서 해당 레코드를 찾아야 하고, 이를 위해서 리프 노드가 주솟값을 가지고 있는 것이다.

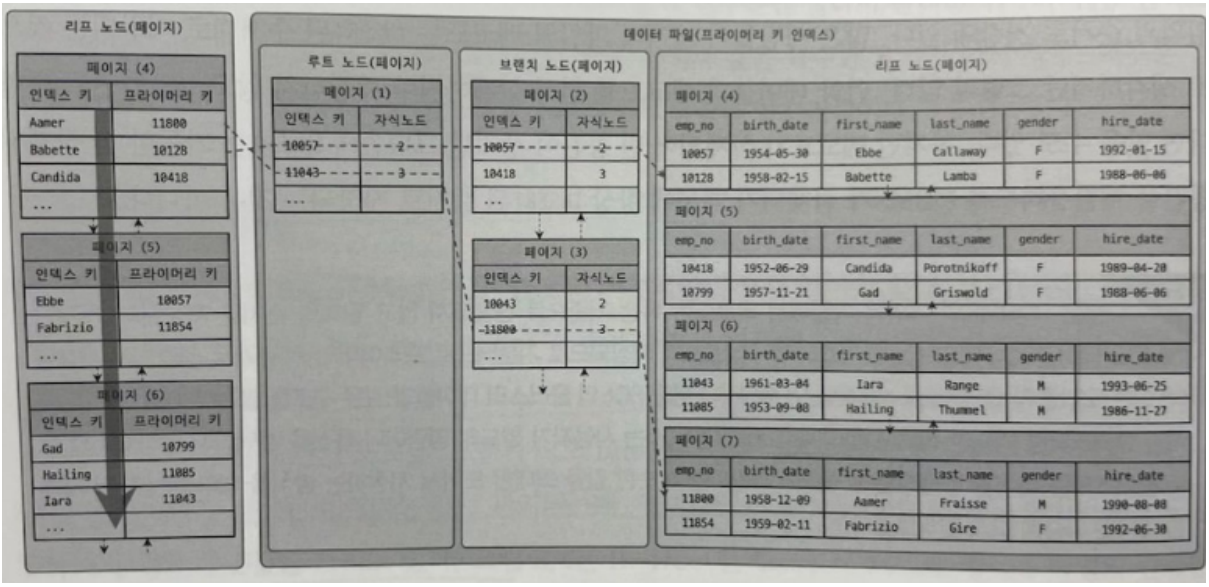
많은 사람들이 데이터 파일의 레코드가 INSERT된 순서대로 저장된다고 생각하지만 그렇지 않다. 만약 테이블의 레코드를 전혀 삭제하지 않는다면 그럴수도 있지만, 삭제하여 빈 공간이 생기면 그 공간을 재활용하기 때문에 INSERT된 순서대로 저장되지 않는다.

하지만 InnoDB 테이블에서 레코드는 프라이머리 키를 기준으로 정렬되어 클러스터되어 디스크에 저장된다.

MyISAM vs InnoDB



MyISAM 테이블의 인덱스와 데이터 파일의 관계



InnoDB 테이블의 인덱스와 데이터 레코드의 관계

두 스토리지 엔진의 인덱스에서 가장 큰 차이점은 세컨더리 인덱스를 통해 데이터 파일의 레코드를 찾아가는 방법에 있다. MyISAM 테이블은 세컨더리 인덱스가 물리적인 줏를 가지는 반면 InnoDB 테이블은 프라이머리 키를 주소처럼 사용하기 때문에 논리적인 주소를 가진다고 볼 수 있다.

그렇기 때문에 InnoDB 테이블에서 인덱스를 통해 레코드를 읽을 때는 데이터 파일을 바로 찾아가지 못하고, 인덱스에 저장돼 있는 프라이머리 키 값을 이용해 프라이머리 키 인덱스를

한 번 더 검색한 후, 프라이머리 키 인덱스의 리프 페이지에 저장돼 있는 레코드를 읽는다.

한마디로, InnoDB 스토리지 엔진에서는 모든 세컨더리 인덱스 검색에서 데이터 레코드를 읽기 위해서 반드시 프라이머리 키를 저장하고 있는 B-Tree를 다시 한번 검색해야 한다!

B-Tree 인덱스 키 추가 및 삭제

(1) 인덱스 키 추가

새로운 키 값이 B-Tree에 저장될 때 테이블의 스토리지 엔진에 따라 새로운 키 값이 즉시 인덱스에 저장될 수도 있고 그렇지 않을 수도 있다. B-Tree에 저장될 때는 저장될 키 값을 이용해 B-Tree상의 적절한 위치를 검색해야 한다.

저장될 위치가 결정되면 레코드의 (키 값, 주소 정보)를 B-Tree의 리프 노드에 저장하고, 리프노드가 꽉 차서 더 저장할 수 없을 때는 리프 노드가 분리돼야 하는데, 이는 상위 브랜치 노드까지 처리의 범위가 넓어진다. 그렇기 때문에 B-Tree는 상대적으로 쓰기 작업에 비용이 많이 든다.

인덱스 추가로 인해 INSERT나 UPDATE 문장이 어떤 영향을 받는지는 테이블의 칼럼 수, 칼럼의 크기, 칼럼의 특성 등에 따라 다르다. 하지만 대략적으로 테이블에 레코드를 추가하는 작업 비용을 1이라고 가정하면, 해당 테이블의 인덱스에 키를 추가하는 작업 비용을 1.5 정도로 예측하면 된다.

만일 인덱스가 하나도 없는 경우 작업 비용이 1이라면, 3개인 경우에는 1.5*3+1 = 5.5 정도로 예측하면 된다. 그리고 또 중요한 점은 이 비용의 대부분이 메모리와 CPU에서 처리하는 시간이 아니라 디스크로부터 인덱스 페이지를 읽고 쓰기를 하기위해 걸리는 시간이라는 점이다.

MyISAM이나 MEMORY 스토리지 엔진을 사용하는 테이블에서는 INSERT 문장이 실행되면 즉시 새로운 키 값을 B-Tree 인덱스에 적용한다. **하지만 InnoDB 스토리지 엔진은 이 작업을 조금 더 지능적으로 처리하는데, 필요하다면 인덱스 키 추가 작업을 지연시켜 나중에 처리할 수 있다 (체인지 버퍼 - 아키텍처 참고)**. 하지만 프라이머리 키나 유니크 인덱스의 경우 중복 체크가 필요하기 때문에 즉시 B-Tree에 추가하거나 삭제한다.

(2) 인덱스 키 삭제

B-Tree의 키 값이 삭제될때는 리프 노드를 찾아서 삭제 마크만 하면 작업이 완료된다. 이 공간은 다시 사용될 수 있다. 이 작업 역시 디스크 I/O가 필요하므로 InnoDB 스토리지 엔진에서는 이 역시 지연 처리가 가능하다. MyISAM이나 MEMORY 스토리지 엔진의 테이블에서는 체인지 버퍼 같은 기능이 없으므로 인덱스 키 삭제가 완료된 후 쿼리 실행이 완료된다.

(3) 인덱스 키 변경

B-Tree의 키 값 변경 작업은 먼저 키 값을 삭제한 후, 다시 새로운 키 값을 추가하는 형태로 처리된다. 이 역시 InnoDB 스토리지 엔진을 사용하는 테이블에 대해서는 체인지 버퍼를 활용해 지연 처리될 수 있다.]

(4) 인덱스 키 검색

인덱스를 검색하는 작업은 B-Tree의 루트 노드부터 시작해 브랜치 노드를 거쳐 최종 리프 노드까지 이동하면서 비교 작업을 수행하고, 이 과정을 `트리 탐색`이라고 한다. 트리 탐색은 SELECT에서만 사용하는 것이 아니라 UPDATE나 DELETE를 처리하기 위해 항상 해당 레코드를 먼저 검색해야 할 경우에도 사용된다.

B-Tree 인덱스를 이용한 검색은 100% 일치 또는 값의 앞부분만 일치하는 경우에 사용할 수 있다. 부등호(<, >) 비교 조건에서도 인덱스를 활용할 수 있지만, 인덱스를 구성하는 키 값의 뒷부분만 검색하는 용도로는 인덱스를 사용할 수 없다.

그리고 인덱스의 키 값에 변형이 가해진 후 비교되는 경우에는 B-Tree의 빠른 검색 기능을 사용할 수 없다. **따라서 함수나 연산을 수행한 결과로 정렬한다거나 검색하는 작업은 B-Tree의 장점을 이용할 수 없다.**

InnoDB 테이블은 UPDATE나 DELETE 문장이 실행될 때 적절히 사용할 수 있는 인덱스가 없으면 불필요하게 많은 레코드를 잠근다. 그렇기 때문에 InnoDB 스토리지 엔진에서는 인덱스의 설계가 매우 중요하다.

B-Tree 인덱스 사용에 영향을 미치는 요소

(1) 인덱스 키 값의 크기

InnoDB 스토리지 엔진은 디스크에 데이터를 저장하는 가장 기본 단위를 페이지 또는 블록이라고 하고, 디스크의 모든 읽기 및 쓰기 작업의 최소 작업 단위가 된다. 인덱스도 페이지 단위로 관리된다. **MySQL의 B-Tree가 자식 노드를 몇 개까지 가지는지는 인덱스의 페이지 크기와 키 값의 크기에 따라 결정된다.**

그러므로 인덱스를 구성하는 키 값의 크기가 커지면 디스크로부터 읽어야 하는 횟수가 늘어나고, 그만큼 느려진다는 것을 의미한다.

(2) B-Tree 깊이

B-Tree 인덱스의 깊이는 상당히 중요하지만 직접 제어할 방법은 없다. B-Tree의 깊이는 MySQL에서 값을 검색할 때 몇 번이나 랜덤하게 디스크를 읽어야 하는지와 직결되는 문제다. 결론적으로 인덱스 키 값의 크기가 커질수록 하나의 인덱스 페이지가 담을 수 있는 키 값의 개수가 적어지고, 그로인해 같은 레코드 건수라 하더라도 깊이가 깊어져서 디스크 읽기가 더 많이 필요하게 된다.

인덱스 키 값의 크기는 가능하면 작게 만드는 것이 좋다는 것을 강조하기 위한 설명이고, 실제로는 아무리 대용량 DB라도 깊이가 5단계 이상 깊어지는 경우는 흔치 않다.

(3) 선택도 (기수성)

인덱스에서 선택도(Selectivity) 또는 기수성(Cardinality)는 거의 같은 의미로 사용되며, **모든 인덱스 키 값 가운데 유니크한 값의 수를 의미한다**. 인덱스 키 값 가운데 중복된 값이 많아질수록 기수성은 낮아지고 선택도 또한 떨어진다. 인덱스는 선택도가 높을수록 검색 대상이 줄어들기 때문에 그만큼 빠르게 처리된다.

그렇기 때문에 인덱스에서 유니크한 값의 개수는 인덱스나 쿼리의 효율성에 큰 영향을 미친다.

(4) 읽어야 하는 레코드의 건수

인덱스를 통해 테이블의 레코드를 읽는 것은 인덱스를 거치지 않고 바로 테이블의 레코드를 읽는 것보다 높은 비용이 드는 작업이다. 그렇기 때문에 인덱스를 이용한 읽기의 손익 분기점이 얼마인지 판단할 필요가 있는데, 일반적인 DBMS의 옵티마이저에서는 인덱스를 통해 레코드 1건을 읽는 것이 테이블에서 직접 레코드 1건을 읽는 것보다 4~5배정도 비용이 더 많이 드는 작업인 것으로 예측한다.

그렇기 때문에 인덱스를 통해 읽어야 할 레코드의 건수가 전체 테이블 레코드의 20~25%를 넘어서면 인덱스를 이용하지 않는 것이 좋다.

물론 손익분기점을 넘을 경우 MySQL의 옵티마이저가 기본적으로 인덱스를 무시하고 테이블을 직접 읽는 방식으로 처리하겠지만 기본적으로 알고 있어야 할 사항이다.

B-Tree 인덱스를 통한 데이터 읽기

(1) 인덱스 레인지 스캔

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

인덱스 레인지 스캔은 다음과 같이 크게 3단계를 거친다:

1. 인덱스에서 조건을 만족하는 값이 저장된 위치를 찾는다. 이 과정을 인덱스 탐색(index seek)이라고 한다.
2. 1번에서 탐색된 위치부터 필요한 만큼 인덱스를 차례대로 쭉 읽는다. 이 과정을 인덱스 스캔이라고 한다.
3. 2번에서 읽어 들인 인덱스 키와 레코드 주소를 이용해 레코드가 저장된 페이지를 가져오고, 최종 레코드를 읽어온다.

쿼리가 필요로 하는 데이터에 따라 3번 과정은 불필요할 수 있는데, 이를 커버링 인덱스라고 한다. 커버링 인덱스로 처리되는 쿼리는 디스크의 레코드를 읽지 않아도 되기 때문에 랜덤 읽기가 상당히 줄어들고 성능은 그만큼 빨라진다.

(2) 인덱스 풀 스캔

인덱스의 처음부터 끝까지 모두 읽는 방식을 인덱스 풀 스캔이라고 한다. 대표적으로 쿼리의 조건절에 사용된 칼럼이 인덱스의 첫 번째 칼럼이 아닌 경우 인덱스 풀 스캔 방식이 사용된다. **쿼리가 인덱스에 명시된 칼럼만으로 조건을 처리할 수 있는 경우 이 방식이 사용된다**. 인덱스 뿐만 아니라 데이터 레코드까지 모두 읽어야 한다면 절대 이 방식으로 처리되지 않는다.

(3) 루스 인덱스 스캔

루스 인덱스 스캔이란 느슨하게(듬성듬성) 인덱스를 읽는 것을 의미한다. 앞에서 소개한 두 가지 방법은 타이트(Tight) 인덱스 스캔으로 분류된다. 루스 인덱스 스캔은 레인지 스캔과 비슷하게 작동하지만 중간에 필요치 않은 인덱스 키 값은 무시하고 다음으로 넘어간다. **일반적으로 GROUP BY 또는 집함 함수 가운데 MAX() 또는 MIN() 함수에 대해 최적화를 하는 경우에 사용된다.**

(4) 인덱스 스킵 스캔

데이터베이스 서버에서 인덱스의 핵심은 값이 정렬돼 있다는 것이며, 이로 인해 인덱스를 구성하는 칼럼의 순서가 매우 중요하다. 예를 들어 employees 테이블에 아래와 같은 인덱스를 생성해보자:

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

이 인덱스를 사용하려면 WHERE 조건절에 gender 칼럼에 대한 비교 조건이 필수다.

그렇기 때문에 brith_date 칼럼의 조건만 가진 쿼리는 이 인덱스를 사용할 수가 없었다. 하지만 MySQL 8.0 버전부터는 옵티마이저가 gender 칼럼을 건너뛰어서 birth_date 칼럼만으로도 인덱스 검색이 가능하게 해주고, 이를 인덱스 스킵 스캔이라 부른다. 아래는 인덱스 스킵 스캔이 어떻게 처리되는지를 보여준다:

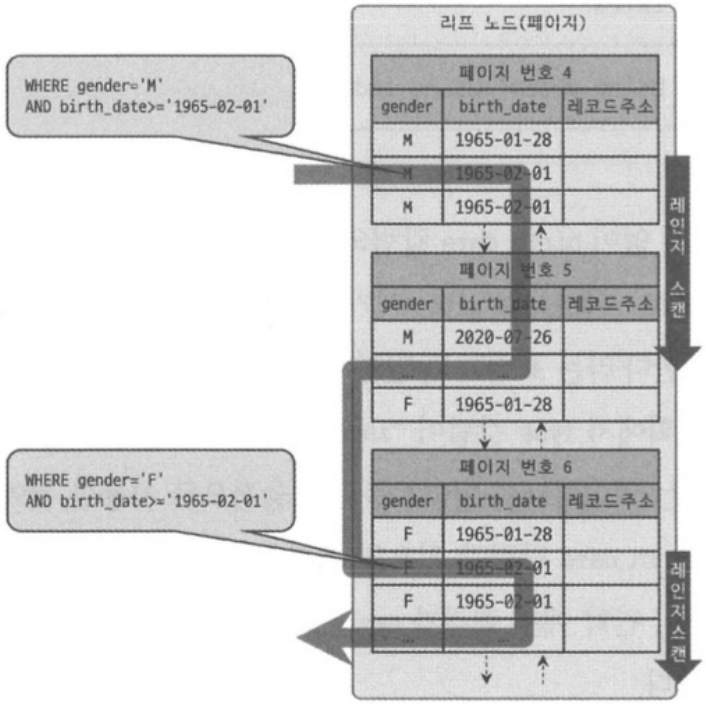


그림 8.12 인덱스 스킵 스캔

인덱스 스킵 스캔에는 아직 단점이 존재하는데:

1. WHERE 조건절에 조건이 없는 인덱스의 선행 칼럼의 유니크한 값의 개수가 적어야 한다.
2. 쿼리가 인덱스에 존재하는 칼럼만으로 처리 가능해야 한다 (커버링 인덱스).

인덱스의 선행 칼럼이 유니크한 값의 개수가 많다면 레인지 스캔 시작 지점을 검색하는 작업이 많아지기 때문에 쿼리의 성능이 매우 떨어진다.

다중 칼럼(Multi-column) 인덱스

2개 이상의 칼럼으로 구성된 인덱스를 얘기한다. Concatenated Index라고도 불리며 데이터 레코드 건수가 작은 경우에는 브랜치 노드가 존재하지 않을 수도 있다. 하지만 루트 노드와 리프 노드는 항상 존재한다. 인덱스의 두 번째 칼럼은 첫 번째 칼럼에 의존하여 정렬되게 된다. 그렇기 때문에 다중 칼럼 인덱스에서는 인덱스의 순서가 상당히 중요하고, 이를 매우 신중히 결정해야 한다.

B-Tree 인덱스의 정렬 및 스캔 방향

(1) 인덱스의 정렬

MySQL 8.0 이전에는 칼럼 단위로 정렬 순서를 혼합해서 인덱스를 생성할수 없었지만, 8.0 이후로는 아래와 같이 혼합 인덱스를 생성할 수 있게 되었다.

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

(2) 인덱스 스캔 방향

아래 쿼리가 실행될 때 MySQL은 인덱스를 어떻게 읽어올까?

```
SELECT * FROM employees ORDER BY first_name DESC LIMIT 1;
```

오름차순으로 정렬돼 있는 인덱스를 최댓값부터 거꾸로 읽으면 내림차순으로 가져올 수 있다는 것을 MySQL 옵티마이저는 알고 있기 때문에 오름차순으로 정렬돼 있는 인덱스라도 역순으로 읽어온다.

(2) 내림차순 인덱스

MySQL 8.0 부터 지원하는 내림차순 인덱스는 과연 오름차순 인덱스와 같은 성능을 보여줄까? 우선 오름차순 인덱스와 내림차순 인덱스의 정렬 차이를 살펴보자:

- 오름차순 인덱스: 작은 값의 인덱스 키가 B-Tree의 왼쪽으로 정렬된 인덱스
- 내림차순 인덱스: 큰 값의 인덱스 키가 B-Tree의 왼쪽으로 정렬된 인덱스
- 인덱스 정순 스캔: 인덱스 키의 크고 작음에 관계없이 인덱스 리프 노드의 왼쪽 페이지부터 오른쪽으로 스캔
- 인덱스 역순 스캔: 인덱스 키의 크고 작음에 관계없이 인덱스 리프 노드의 오른쪽 페이지부터 왼쪽으로 스캔

실제로 1천만 건의 레코드가 있는 테이블에서 다음 아래 쿼리를 실행하면 1.2초정도의 차이가 난다:

```
SELECT * FROM t1 ORDER BY tid ASC LIMIT 12619775, 1;  //~4.15sec
SELECT * FROM t1 ORDER BY tid DESC LIMIT 12619775, 1;  //~5.35sec
```

그 이유로는 두가지를 꼽을 수 있다:

- 페이지 잠금이 인덱스 정순 스캔에 적합한 구조
- 페이지 내에서 인덱스 레코드가 단방향으로만 연결된 구조

하지만 인덱스를 역순으로 정렬하는 쿼리가 많지 않다면, 굳이 인덱스를 역순으로 정렬할 고려를 할 필요는 없다.

B-Tree 인덱스의 가용성과 효율성

(1) 비교 조건의 종류와 효율성

```
SELECT * FROM dept_emp
WHERE dept_no='d002' AND emp_no >= 10114;
```

위 쿼리가 실행될 때 아래 두 가지 인덱스에 어떤 차이가 있는지 살펴보자:

- Case A: INDEX(dept_no, emp_no)
- Case B: INDEX(emp_no, dept_no)

Case A에서는 `dept_no='d002' AND emp_no >= 10114` 조건에 맞는 레코드를 찾고 그 이후에 d002가 아닐때까지 인덱스를 쭉 읽게된다.

하지만 Case B 인덱스에서는 우선 `emp_no>=10144 AND dept_no='d002'` 인 레코드를 찾고 그 이후 모든 레코드에 대해 dept_no가 'd002'인지 비교하게 된다.

이처럼 인덱스를 통해 읽은 레코드가 나머지 조건에 맞는지 비교하면서 취사 선택하는 작업을 `필터링` 이라고도 한다.

(2) 인덱스의 가용성

B-Tree 인덱스는 왼쪽 값에 기준해서 오른쪽 값이 정렬된다. 여기서 왼쪽이란 하나의 칼럼 내에서뿐만 아니라 다중 칼럼 인덱스의 칼럼에 대해서도 적용된다. 그렇기 때문에 하나의 칼럼으로 검색해도 값의 왼쪽 부분이 없으면 인덱스 레인지 스캔 방식의 검색이 불가능하다. 또한 다중 칼럼 인덱스에서도 왼쪽 칼럼의 값을 모르면 인덱스 레인지 스캔을 사용할 수 없다.

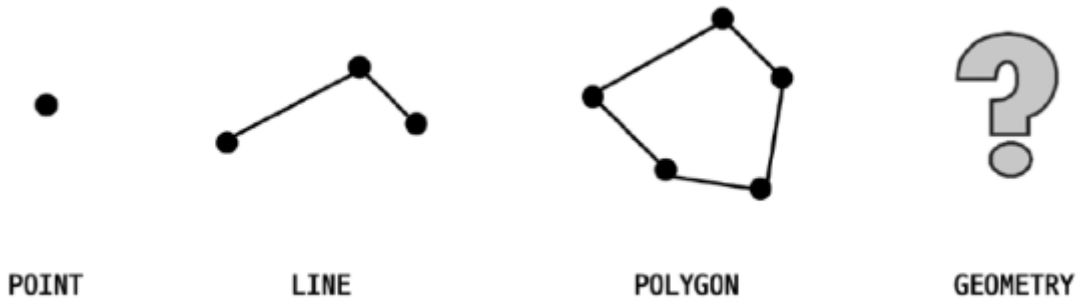
R-Tree 인덱스

R-Tree 인덱스는 공간 인덱스(Spatial Index)에서 사용되는 알고리즘이다. 2차원 데이터를 인덱싱하고 검색하기 위해 사용되며, B-Tree와 매커니즘은 흡사하지만 B-Tree가 인덱스를 구성하는 칼럼의 값이 1차원의 스칼라 값인 반면, R-Tree 인덱스는 2차원의 공간 개념 값이다. GIS와 GPS 기반의 위치 기반 서비스를 구현하기 위해서는 MySQL의 공간 확장(Spatial Extension)을 이용하면 되고, 아래 기능들을 제공한다:

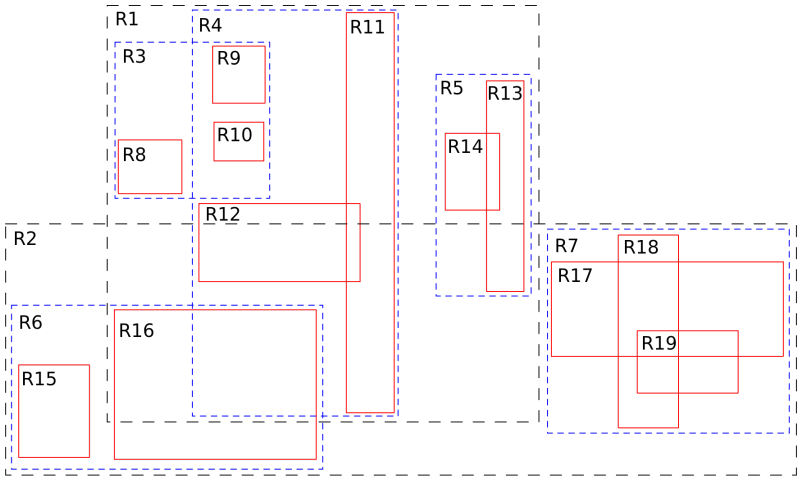
- 공간 데이터를 저장할 수 있는 데이터 타입
- 공간 데이터의 검색을 위한 공간 인덱스 알고리즘 (R-Tree)

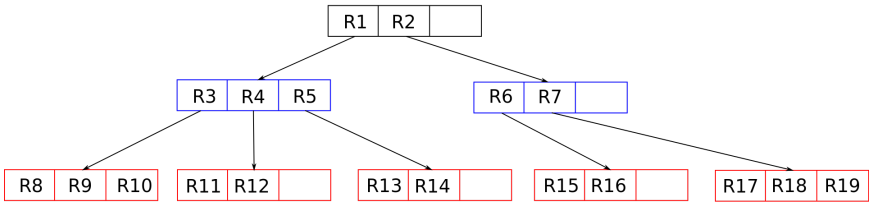
- 공간 데이터의 연산 함수 (거리 또는 포함 관계의 처리)

구조 및 특성



대표적으로 MySQL에서 지원하는 데이터 타입들이다. GEOMETRY 타입은 나머지 3개의 슈퍼 타입으로 3개 모두 저장할 수 있다. R-Tree 알고리즘은 각 도형들을 감싸는 최소 크기의 사각형 (Minimum Bounding Rectangle, MBR)로 이루어져 있고, 이 사각형들의 포함 관계를 B-Tree 형태로 구현하였다.





R-Tree 인덱스의 용도

주로 WGS84(GPS) 기준의 위도, 경도 좌표 저장에 사용된다. 하지만 CAD/CAM 소프트웨어 또는 회로 디자인 등과 같이 좌표 시스템에 기반을 둔 정보를 모두 저장할 수 있다. GPS

기반의 정보 사용 예시로는 ‘현재 사용자의 위치로부터 반경 5km 이내의 음식점 검색’ 등이 있다.

위와 같은 검색이 가능한 이유는 R-Tree는 각 도형의 포함 관계를 이용해 만들어졌기 때문이다. 그렇기 때무에 ST_Contains() 또는 ST_Within() 등과 같은 포함 관계를 비교하는 함수로 검색을 할때만 인덱스를 이용할 수 있다.