인덱스(2)

8.6. 함수 기반 인덱스

- **칼럼의 값을 변형**해서 만들어진 값에 대한 **인덱스**
    - 일반적인 인덱스 : 칼럼의 값 일부(칼럼의 값 앞 부분) / 전체
- **인덱싱할 값을 계산**하는 **과정**의 **차이**만 있음
- 실제 인덱스의 **내부적 구조** 및 **유지관리 방법이: B-Tree 인덱스와 동일**

> *구현 방법*

1. 가상 칼럼을 이용한 인덱스
2. 함수를 이용한 인덱스

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)
  );
```

> 예제 : `first_name`과 `last_name`을 합쳐서 검색해야 할 때 !

- **가상 칼럼**을 추가하고 그 가상 칼럼에 인덱스 생성하면 됨
- 이전 버전의 경우, `full_name`이라는 칼럼을 추가하고 모든 레코드에 대해 `full_name`을 업데이트하는 작업을 거쳐야 했음.

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

- full_name 칼럼에 대한 검색을 새로 만들어진 `ix_fullname` 인덱스를 이용해 실행 계획이 만들어지는 것을 확인할 수 있음

```
> EXPLAIN SELECT * FROM user WHERE full_name='Matt Lee';
+----+-------------+-------+------+-------------+---------+-------+
| id | select_type | table | type | key         | key_len | Extra |
+----+-------------+-------+------+-------------+---------+-------+
|  1 | SIMPLE      | user  | ref  | ix_fullname | 1023    | NULL  |
+----+-------------+-------+------+-------------+---------+-------+
```

- **가상 칼럼**이 `VIRTUAL` / `STORED` 옵션 중 **어떤 옵션으로 생성됐든 관계없이** 해당 가상 칼럼에 **인덱스 생성 가능**
- 그러나 가상 칼럼은 테이블에 새로운 칼럼을 추가하는 것과 같은 효과를 냄.
  → **실제 테이블 구조 변경됨**

8.6.2. 함수를 이용한 인덱스

- MySQL 8.0 버전부터 생성 가능한 인덱스
- **테이블의 구조를 변경하지 않고** 함수를 직접 사용하는 인덱스
- **계산된 결괏값의 검색을 빠르게** 만들어줌

```
> 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)))
  );
```

- 제대로 활용하려면 **반드시 조건절**에 함수 기반 **인덱스에 명시된 표현식이 그대로** 사용돼야 함

  만약, 함수 생성 시 명시된 표현식 ≠ 쿼리의 WHERE 조건절에 사용된 표현식 (결과가 같아도)

  MySQL 옵티마이저는 **다른 표현식으로 간주**하여 함수 기반 인덱스를 사용하지 못함.

```
> EXPLAIN SELECT * FROM user WHERE ((CONCAT(first_name, ' ', last_name)))='Matt Lee';
+----+-------------+-------+------+-------------+---------+-------+-------+
| id | select_type | table | type | key         | key_len | ref   | Extra |
+----+-------------+-------+------+-------------+---------+-------+-------+
|  1 | SIMPLE      | user  | ref  | ix_fullname | 87      | const | NULL  |
+----+-------------+-------+------+-------------+---------+-------+-------+
```

🏷️ 가상 칼럼을 이용하든 직접 함수를 이용하든 함수 기반 인덱스는 **내부적으로 동일한 구현 방법**을 사용함 !! → 어떤 방법을 쓰든 **둘의 성능은 같다**

8.7. 멀티 밸류 인덱스

- 모든 인덱스는 하나의 레코드가 1개의 인덱스 키 값을 가짐(1:1 관계) (전문 검색 인덱스 제외)
- 멀티 밸류 인덱스는 **하나의 데이터 레코드**가 **여러 개의 인덱스 키 값**을 가짐 (1:n 관계)

- 일반적인 RDBMS를 기준으로 생각하면 이러한 인덱스는 **정규화에 위배되는 형태**
- 하지만 최근 RDBMS들이 JSON 데이터 타입을 지원하기 시작하면서 **JSON 배열 타입**의 필드에 저장된 **원소**들에 대한 **인덱스 요건**이 발생한 것
- 다음 함수들을 이용해서 검색해야 옵티마이저가 인덱스를 활용한 실행 계획을 수립함
  - `MEMBER OF()`
  - `JSON_CONTAINS()`
  - `JSON_OVERLAPS()`

8.8. 클러스터링 인덱스

- 클러스터링 : 여러 개를 하나로 묶는다
- 클러스터링 인덱스 : 테이블의 레코드를 비슷한 것끼리(**PK 기준으로) 묶어서** 저장한다

  ⇒ **비슷한 값들을 동시에 조회**하는 경우가 **많기 때문**
- InnoDB 스토리지 엔진에서만 지원됨. 나머지는 지원 안 함.

8.8.1. 클러스터링 인덱스 (PK가 데이터 레코드 저장에 미치는 영향)

- PK값이 비슷한 레코드끼리 묶어서 저장하는 것
- PK값에 의해 **레코드 저장 위치** 또한 결정됨

  ⇒ 즉, **PK값 변경** 시, 그 레코드의 **물리적인 저장 위치 또한 변경**됨.

  → 인덱스 알고리즘이라기보다는 테이블 레코드의 저장 방식이라고 할 수 있음 ('클러스터링 인덱스' == '클러스터링 테이블', PK == '클러스터링 키')
- InnoDB처럼 항상 클러스터링 인덱스로 저장되는 테이블은 PK기반의 검색이 매우 빠르며, 대신 레코드의 저장이나 PK의 변경이 상대적으로 느림.
- **PK값에 대한 의존도가 상당히 크기에, 신중히 PK를 결정해야 함.**
- 클러스터링 테이블 구조 자체는 일반적인 B-Tree와 비슷하지만, **리프 노드에 레코드의 모든 칼럼**이 같이 저장됨.

  ⇒ 즉, 클러스터링 인덱스는 **하나의 거대한 인덱스 구조**로 관리되는 것.

8.8.2. 세컨더리 인덱스에 미치는 영향

- 데이터 레코드가 저장된 주소는 내부적인 레코드 아이디(ROWID) 역할을 함

  PK이나 세컨더리 인덱스의 각 키는 그 주소를 이용하여 실제 데이터 레코드를 찾음

- InnoDB의 세컨더리 인덱스가 실제 레코드가 저장된 주소를 가지고 있다면, 클러스터링 키가 변경될 때마다 주소가 변경되고, 모든 인덱스에 저장된 주솟값을 변경해야함 → 클러스터링 테이블의 **모든 세컨더리 인덱스는 PK 값을 저장함**
- PK를 확인한 후 PK인덱스를 검색하는 과정을 거치기에, InnoDB가 MyISAM보다 좀 더 복잡하게 처리되지만, InnoDB 테이블에서 클러스터링 인덱스는 더 큰 장점을 제공하기 때문에 성능 저하에 대한 걱정 안 해도 됨

8.8.3. 클러스터링 인덱스의 장점과 단점

MyISAM과 같은 클러스터링되지 않은 일반 PK와 클러스터링 인덱스를 비교했을 때 상대적인 장단점

장점 - 빠른 읽기 (SELECT)

- PK(클러스터링 키)로 검색했을 때 처리 성능이 매우 빠름.(특히 PK를 범위 검색할 때)
- 테이블의 모든 세컨더리 인덱스가 PK를 가지고 있기 때문에 **인덱스만으로 처리될 수 있는 경우가 많음**(커버링 인덱스)

단점 - 느린 쓰기 (INSERT, UPDATE, DELETE)

- 테이블의 모든 세컨더리 인덱스가 PK를 가지고 있기 때문에, 클러스터링 키 값이 커질 경우 전체적으로 인덱스의 크기가 커짐.
- 세컨더리 인덱스를 통해 검색할 때, **PK로 다시 한번 검색해야 함** → 처리 성능이 느림.
- INSERT할 때 **PK에 의해 레코드의 저장 위치**가 결정됨 → 처리 성능 느림
- PK를 변경할 때 레코드를 DELETE하고 INSERT하는 작업이 필요함 → 처리 성능 느림.

🏷️ 일반적으로 웹 서비스같은 온라인 트랜잭션 환경(OLTP)에서는 쓰기:읽기 비율이 1:9 또는 2:8이므로 **조금 느린 쓰기를 감수하고 읽기를 빠르게 유지**하는 것이 중요하다!

8.8.4. 클러스터링 테이블 사용 시 주의 사항 (InnoDB)

8.8.4.1. 클러스터링 인덱스 키의 크기

- 클러스터링 테이블의 경우 **모든 세컨더리 인덱스(보조 인덱스)가 PK값을 포함함**

  → PK의 크기가 커지면 세컨더리 인덱스(보조 인덱스)의 크기도 자동으로 커짐
- 일반적으로 한 테이블에 세컨더리 인덱스는 4~5개
- 게다가 인덱스가 커질 수록 같은 성능을 내기 위해 **그만큼의 메모리가 더 필요함**

  ⇒ InnoDB의 PK는 신중히 선택하자!

8.8.4.2. PK는 `AUTO-INCREMENT`보다는 업무적인 칼럼으로 생성(가능한 경우)

- InnoDB의 PK는 클러스터링 키로 사용되고, 이 값에 의해 레코드의 위치 결정됨.
- PK(클러스터링 키)로 **검색했을 때 처리 성능이 매우 빠름**.(특히 PK를 범위 검색할 때)

→ 크기가 크더라도 **업무적으로 해당 레코드를 대표하는 칼럼**이라면 **PK**로 설정하는 것이 좋음

8.8.4.3. PK는 반드시 명시할 것

- `AUTO_INCREMENT` **칼럼을 사용해서라도 PK는 생성하는 것을 권장함.**

  InnoDB 테이블에서 PK를 정의하지 않으면 InnoDB 스토리지 엔진이 내부적으로 일련번호 칼럼을 추가함. 하지만 자동으로 추가된 칼럼은 사용자에게 보이지 않기에 사용자가 전혀 접근 불가함. 어차피 같다면 사용자가 사용할 수 있는 값(`AUTO_INCREMENT` 값)을 PK로 설정하자

- ROW 기반의 복제나 InnoDB Cluster에서는 모든 테이블이 PK를 가져야만 하는 정상적인 복제 성능을 보장하기도 함

8.8.4.4. AUTO-INCREMENT 칼럼을 인조 식별자로 사용할 경우

- 여러 개의 칼럼이 복합으로 PK를 만들어지는 경우, PK의 크기가 길어질 때가 가끔 있음.
- PK가 크기가 길어도 세컨더리 인덱스(보조인덱스)가 필요하지 않다면 그대로 PK를 사용하자.
- **세컨더리 인덱스도 필요하고 PK의 크기도 길다면** `AUTO_INCREMENT` 칼럼을 추가하고 이를 PK로 하자.

  **인조 식별자 : PK를 대체하기 위해 인위적으로 추가된 프라이머리 키
- 조회보다 **INSERT 위주의 테이블** (예. 로그 테이블)의 경우 해당 방법이 성능 향상에 도움됨.

8.9. 유니크 인덱스

- 유니크는 사실 인덱스라기 보다는 **제약 조건**에 가깝다고 볼 수 있음. (중복값 저장 불가)
- MySQL에서는 인덱스 없이 유니크 제약만 설정할 방법이 없음
- 유니크 인덱스에서는 NULL도 저장될 수 있는데, **NULL은 2개 이상 저장 가능함**
- MySQL에서 PK는 기본적으로 NULL 허용하지 않는 유니크 속성이 자동으로 부여됨

  MyISAM / MEMORY 테이블에서 PK는 사실 NULL이 허용되지 않는 유니크 인덱스와 같음

  그러나 InnoDB 테이블의 PK는 클러스터링 키의 역할도 하므로 유니크 인덱스와는 근본적으로 다름

8.9.1. 유니크 인덱스와 일반 세컨더리 인덱스의 비교

유니크 인덱스와 유니크하지 않은 일반 세컨더리 인덱스는 인덱스 **구조상 차이는 없음**

읽기와 쓰기를 **성능** 관점에서 볼 수 있음

8.9.1.1. 인덱스 읽기

- 많은 사람들이 유니크 인덱스가 빠르다고 생각하지만 사실이 아님
- 유니크 인덱스는 1건만 읽으면 되지만, *유니크 하지 않은 세컨더리 인덱스에서는 레코드를 한 건 더 읽어야 한다?*

  → 유니크 하지 않은 세컨더리 인덱스에서 한 번 더 해야 하는 작업은 디스크 읽기가 아니라 **CPU에서 칼럼값을 비교**하는 작업임. **성능에 영향 없음**
- 유니크 하지 않은 세컨더리 인덱스는 중복된 값이 허용되므로 읽어야 할 레코드가 많아서 느린 것이지, **인덱스 자체 특성이 느린 것이 아님**

  레코드 1개를 읽느냐, 2개 이상을 읽느냐의 차이만 있다는 것을 의미함. (1건 0.1초, 2건 0.2초 일때 후자를 느리게 처리게 처리됐다고 할 수 없음)
- 사용되는 실행 계획이 다르지만, 이는 인덱스의 성격이 유니크한지에 대한 것임.
- **읽어야 할 레코드 수가 같다면 성능상 차이가 거의 없음**

8.9.1.2. 인덱스 쓰기

- 새로운 레코드가 INSERT되거나 인덱스 칼럼의 값이 변경되는 경우 인덱스 쓰기 작업이 이루어짐
- 그런데 유니크 인덱스의 키 값을 쓸 때는 **중복된 값이 있는지 체크**하는 과정이 필요함

  또 중복된 값을 체크할 때는 읽기 잠금을 사용하고, 쓰기를 할 때 쓰기 잠금을 사용하는데 이 과정에서 **데드락이 아주 빈번히 발생**함

  InnoDB 스토리지 엔진에는 인덱스 키의 저장을 버퍼링하기 위해 체인지 버퍼가 사용되는데(인덱스의 저장/변경 작업을 매우 빠르게 처리해줌), 유니크 인덱스는 중복 체크 과정 때문에 작업 자체를 버퍼링하지 못함

  ⇒ 유니크하지 않은 세컨더리 인덱스의 쓰기(변경 작업)보다 **느림.**

8.9.2. 유니크 인덱스 사용 시 주의 사항

- 꼭 필요한 경우라면 유니크 인덱스 생성하는 것은 당연함. (유일성이 꼭 보장되어야 하는 칼럼)
- 하지만 **성능 개선**을 위해서는 **비추천**.
- 하나의 테이블의 *같은 칼럼에서 유니크 인덱스와 일반 인덱스*를 각각 중복해서 생성해 둔 경우,

이미 유니크 인덱스도 일반 세컨더리 인덱스(보조인덱스)와 같은 역할을 동일하게 수행할 수 있으므로 중복으로 만들어줄 필요는 없다.

- 하나의 테이블의 같은 칼럼에서 PK*와 유니크 인덱스*를 동일하게 생성한 경우도 불필요한 중복
- 유일성이 꼭 보장되어야 하는 칼럼에 대해서는 유니크 인덱스를 생성 하되, 꼭 필요하지 않다면 그대신 **유니크하지 않은 세컨더리 인덱스**를 생성하는 방법을 한번씩 고려하자!

8.10. 외래키

- MySQL에서 외래키는 InnoDB 스토리지 엔진에서만 생성 가능
- **외래키 제약**이 설정되면 **자동**으로 **연관되는 테이블 칼럼**에 **인덱스까지 생성**됨.
- 외래키가 제거되지 않은 상태에서는 자동으로 생성된 인덱스 삭제 불가 → **외래키가 제거** 되어야 자동 생성된 **인덱스 삭제 가능**

> *InnoDB의 외래키 관리의 특징 (중요)*

1. 테이블의 변경(쓰기 잠금)이 발생하는 경우에만 잠금 경합(잠금 대기)가 발생함.
2. 외래키와 연관X 칼럼의 변경은 최대한 잠금 경합(잠금 대기)을 발생X