맹준영_11장

▼ 11.1 인덱스의 동작

- 인덱스와 색인은 비슷
- 색인이 커지는 경우 키워드를 찾는데 시간이 걸림

> **색인 : 키워드가 문자 순서로 정렬돼 있고 해당 키워드가 있는 페이지의 번호가 기록되어 있다.**

RDB의 인덱스

- RDB의 인덱스는 일반적으로 B+트리로 구성
- **B+트리는 데이터(인덱스의 값)가 저장된 리프 노드 + 리프 노드까지의 경로 역할을 하는 논리프 노드**
- 탐색 경로의 출발점은 루트 노드
- 논리프 노드에는 자식 노드가 보유한 값 중에 최솟값이 저장
- 자식 노드는 페이지 ID에 대한 포인터가 저장
- 인덱스의 노드 수 = 테이블의 행 수

인덱스의 왼쪽과 검색 범위

- B+트리는 등가 비교와 범위 검색(Between)에 사용 가능
- LIKE 절 검색의 경우 와일드카드 위치에 따라 검색이 달라지므로 전방 일치를 권장한다.
  - B+트리는 왼쪽의 문자부터 순서대로 정렬
  - ‘a%’인 경우 맨 앞 문자를 알기 때문에 루트 노드부터 간단하게 탐색 가능
  - ‘%a’인 경우 앞 문자를 모르기 때문에 인덱스 스캔
  - ‘%a%’인 경우도 앞 문자를 모르기 때문에 인덱스 스캔
- 다중 컬럼 인덱스의 경우 조건에 왼쪽 컬럼부터 지정하지 않으면 인덱스를 스캔
- 검색에 인덱스를 사용하기 위해선 왼쪽 끝의 컬럼부터 순차적으로 인덱스를 지정
  - 왼쪽 끝 컬럼이 지정되면 나머지 값을 몰라도 범위 검색에서 인덱스를 효과적으로 활용 가능

보조 인덱스의 갱신

- 인덱스의 갱신은 기존 인덱스를 지우고 새로운 인덱스를 추가하는 것 → 삽입, 삭제 오버헤드가 발생
- 인덱스의 갱신 비용은 비싸다.

▼ 11.2 인덱스의 종류

- 일반적으로 B+트리 인덱스지만, RDB 벤더에 따라 인덱스가 다를 수 있다.

해시 인덱스

- 해시 테이블을 이용한 인덱스
- 해시 값을 이용 → 범위 검색이 불가능, 등가 비교 가능 (속도가 빠르다)
- 등가 비교만 가능하고 범위 검색에 필요 없는 곳에 사용
  - ex) 기본키 검색에 사용

전문 검색 인덱스 (Full Text Index)

- B+트리 후방일치, 중간 일치 개선
- “OO라는 단어를 포함한 행을 검색하고 싶은 경우” 사용
- 전치 인덱스 : 행에 포함된 단어나 부분 문자열에 행에 대한 위치가 저장된 구조
  - 단어 : 행 ≠ 1:1 → 인덱스 항목 수 > 테이블의 행 수
- 행에 포함된 단어를 인덱스로 사용하기 위해 단어를 구분하는 작업 필요
  - 형태 분석법
  - N그램

형태 분석법

- 사전을 사용해 문장의 문법을 분석하여 단어를 추출
- 단어를 딱 맞게 구분 가능 → 낭비가 적다.
- 인덱스 사이즌 작은 편 but, B+트리 보단 큼
- 사전에 실리지 않은 단어는 추출 불가능
- 구분 방법에 따라 의미가 달라지는 경우 어떤 단어를 추출할지 판단 불가
  - ex) “아빠가방에들어간다.”
  - → “아빠가 방에 들어간다”, “아빠 가방에 들어간다” → 모호!
- 구어체 처리의 어려움

N 그램

- 문장을 N 문자씩 부분 문자열로 나누는 방법
  - 2문자 : 바이그램, 3문자 : 트리그램
- 단어 단위 부분 문자열 추출이 아니기 때문에 의미 없는 문자열로 잘리는 경우가 존재
- 많은 문자열이 생기므로 인덱스가 커지고 검색 노이즈가 많음, 인덱스의 크기가 커짐

R트리 인덱스

- 지도상의 지점을 검색하는데 사용하는 인덱스, 공간 인덱스
- 한 개의 평면에서 어떤 도형이 다른 도형에 포함되는지, 중복되는지 등을 판별하기 위해 사용

함수 인덱스

- 컬럼 값에 함수를 사용한 WHERE 절은 B+트리 인덱스를 통한 검색 불가능
- 함수 우선 평가 후 컬럼 값에 대해 B+트리 인덱스 작성 → 함수값을 통한 검색 가능

비트맵 인덱스

- OLAP (Online Analytical Processing)에 사용되는 인덱스
- 컬럼별로 여러 개의 비트맵이 생성되고 비트 중 1이 있으면 행의 값이 비트맵의 값이 됨을 나타냄

- B+트리에 비해 크기가 작아 인덱스 스캔이 빠르다.
- 비트맵의 갱신에 시간이 걸리므로 갱신이 많은 작업에 적합하지 않음
- 컬럼이 얻는 값이 많은 경우 비트맵 수 증가로인한 성능 저하

클러스터 인덱스

- 테이블 자체가 인덱스 되어 있는 것
- 기본키의 값을 검색 → 같은 페이지의 데이터가 존재
- 리프 노드에 기본키에 해당하는 페이지 데이터가 존재하고, 데이터를 포함

▼ 11.3 파티셔닝

- 수직 파티셔닝 : 컬럼을 기준으로 테이블을 나누는 것, 특정 컬럼에 대해 고속 스캔시 유용
- 수평 파티셔닝 : 행을 기준으로 테이블을 나누는 것 (*** 아래부터의 파티셔닝은 모두 수평 파티셔닝 기준**)
- 파티셔닝한 테이블은 키의 값에 따라 어떤 파티션에 속하는 행인지 분배
- 파티셔닝 테이블은 각각 동일한 구조를 가진 테이블, 인덱스가 파티션 마다 존재
- 파티션 프루닝 (Partition Pruning) : 키를 통해 파티션의 검색 대상을 좁혀가는 동작
  - 파티션 프루닝 진행 후 각 파티션의 인덱스를 이용해 검색이 가능

> 파티셔닝의 종류 (파티션을 분할하는 방법)

방식	설명
레인지 (Range)	키의 범위에 따라 파티션을 정한다. ex) 날짜별 데이터 분할 (2022-01-01 ~ 2022-12-31)
리스트 (List)	사전에 부여된 키의 값에 따라 파티션을 정한다. ex) 컬럼의 값이 적을 때 효과적
해시 (Hash)	키로 계산한 해시값의 나머지 연산 등에 따라 파티션을 정한다. → 키 파티셔닝

파티셔닝이 적합한 경우

- 파티션 프루닝을 할 수 있는 검색 조건 쿼리가 빈번한 경우 파티셔닝이 효율적
- 인덱스만 사용하더라도 검색 성능 효과적 but, 키의 카디널리티가 낮은 경우에는 파티셔닝이 명확히 효율적 (하나의 검색으로 대량의 데이터가 조회되는 경우)
- 최신의 데이터 참조하는 경우 “레인지 파티셔닝”은 효율적
- 엑세스 데이터의 국소성이 있다면 캐시 효율이 증가

> 엑세스의 국소성 : 데이터의 전체 접근 부위중 어느 한 곳만 엑세스하는 것 (캐시의 지역성)

파티셔닝과 고유성 제약

로컬 인덱스 : 파티션 별로 갖는 인덱스
글로벌 인덱스 : 파티셔닝 수행 중 테이블 전체를 대상으로 하는 인덱스

- 로컬 인덱스의 고유성은 파티션 범위이므로 테이블 전체에 대한 고유성을 보장하지 못함
  - 파티셔닝 된 테이블에 대해 기본키나 유니크 인덱스를 반드시 파티션 키에 포함
- **파티셔닝을 통해 테이블의 고유성을 보장하지 못하는 경우 파티셔닝을 하지 않는 것이 권장**
- 글로벌 인덱스는 테이블의 고유성을 보장 but, 파티션 프루닝 불가능, 파티션 행 갱신 시 글로벌 인덱스 갱신 필요

파티셔닝에 관한 일반적인 오해

- 파티션 프루닝이 불가능한 경우 모든 파티션을 풀스캔하므로 성능이 더 나빠질 수 있다.

- 파티션 프루닝을 못해 인덱스를 통해 검색을 진행하는 경우 모든 로컬 인덱스에 대해 검색이 필요하므로 오버헤드 발생
- 파티셔닝 시 처리가 고속화되는지 검증 필요

▼ 11.4 관계형 모델과 인덱스

- 인덱스는 실행계획과 관련된 물리적인 구현체이다.
- 쿼리 → 인덱스 설계 (O), 인덱스 결정 → 쿼리 작성 (X)

정규화와 인덱스

- 정규화 되지 않은 테이블은 컬럼이 많기 때문에 인덱스도 늘어남
  - 인덱스가 많으면 갱신 성능 저하, 디스크 공간 낭비 증가
  - 정규화된 테이블은 적어도 한 개의 후보키를 가지기 때문에 테이블에 대해 명시적으로 기본키를 작성 가능
- 정규화가 되지 않으면 NULL을 가질 수 있다.
  - 인덱스에서 NULL은 맨 앞 혹은 맨 뒤에 존재
  - 인덱스를 사용한 검색에서 NULL에 대한 조건이 추가되야함

```
WHERE AGE > 20 # Null이 없는 컬럼인 경우

WHERE AGE > 20 OR AGE IS NULL # NULL을 갖는 경우
```

▼ 11.5 지령: 최적의 인덱스를 찾아라!

- 테이블의 크기가 커질 수록 인덱스를 통한 검색 속도의 개선효과는 크다.
- 인덱스가 늘어나면 테이블 갱신 시 오버헤드도 커지고 공간도 늘어난다.
- 검색에 필요한 컬럼이 포함된 인덱스 조합 중 가장 효율 높은 것을 선택

인덱스의 엑세스 특성

- 인덱스는 테이블 풀스캔을 피하는 것이 목적
- 데이터↓, 인덱스 이용한 쿼리 호출 빈도↓, 검색 결과가 매우 많은 행을 가져오는 경우 테이블 풀스캔 하는 것이 더 효율적
- 실행 계획에 따라 인덱스를 사용하지 않는 것이 효율적인 경우도 있기 때문에 설계 시 주의

인덱스가 사용되는 구문

WHERE

- WHERE절에 등호나 부등호로 비교하는 경우 인덱스 효과 가능성 존재
- 다중 컬럼인 경우 다중 컬럼 인덱스 설정 효과적

> 🔥 다중 컬럼 인덱스를 사용하는 경우 검색 조건이 전방일치 해야한다.

```
SELECT * FROM T WHERE COL1 > 100 AND COL2 = 'abc';
```

- (col2, col1)로 인덱스를 설계하면 (’abc’, 100) < (col2, col1) < (’abc’, col1의 최댓값) 범위 인덱스 내에서 검색 가능

JOIN

- JOIN은 내부 테이블 (결합하는 테이블)에 대한 엑세스는 인덱스가 사용
- WHERE 절의 검색 조건이 중요

```
SELECT * FROM t1 JOIN t2 ON t1.COL1 = t2.col2
WHERE t1.col3 = 1000 AND t2.col4 = 'abc'
```

- col2가 기본키거나 유니크 인덱스인 경우 선택되는 데이터가 1개이므로 효과적
- col2가 기본키거나 유니크 인덱스가 아닌 경우 WHERE절의 검색 조건으로 행을 줄인 후 옵티마이저가 JOIN을 수행하는 순서를 바꿀 수 있다.
  - col2가 유니크가 아닌 보조 인덱스 존재 가능성
  - `t1.col1 = t2.col2` → 여러 개의 행이 선택될 수 있다. (1번 작업)
  - WHERE 절의 `t2.col2 = 'abc'`를 이용해 검색 필요 (2번 작업)
- (col2, col4) 다중 컬럼 인덱스가 있는 경우
  - 데이터 선택 시점에 `t1.col1 = t2.col2, t2.col4 = 'abc'` 조건이 모두 적용됨
  - 1번, 2번 작업을 하지 않아 불필요한 작업이 없어짐

상관 서브쿼리

- WHERE 절과 서브쿼리 양 쪽을 모두 고려한 인덱스 설계가 필요

```
# IN 절 서브쿼리
SELECT * FROM t1 WHERE t1.col1 IN
(SELECT col2 FROM t2 WHERE col3 = 100)
AND t1.col4 = 'abc';

# EXISTS
SELECT * FROM t1 WHERE EXISTS
(SELECT * FROM t2 WHERE t2.col3 = 100 AND t1.col1 = t2.col2)
AND t1.col4 = 'abc'
```

- t2 테이블에 (col2, col3) 다중 컬럼 인덱스가 존재하는 경우 인덱스를 통한 해결 가능
- select list에 있는 col2는 IN절의 col과 비교 → `t1.col1 = t2.col2`
  - EXISTS 서브쿼리로 나타내면 col3, col2가 인덱스로 구성되야하는 것이 명시적으로 보인다.

정렬

- B+트리는 키의 순서로 정렬되어 있다.
- WHERE 절을 전방일치 하도록 조건을 지정하는 경우 인덱스를 이용한 범위 검색이 가능
- 일치하는 인덱스에서 ASC인 경우 아래를 , DESC인 경위 위를 읽으면 된다.
- 다중 컬럼 인덱스도 범위가 명확하고 범위의 인덱스가 정렬키의 순서로 나열하는 조건을 만족하면 인덱스를 통해 정렬 해결 가능

```
ex) 다중 컬럼 인덱스가 (col1, col2, col3)인 경우
col1 정렬 -> col2 정렬 -> col3 순서 정렬된 상태로 인덱스 구성

WHERE col1 = 100 AND col2 = 'abc' ORDER BY col3
-> col1, col2로 특정 범위를 명확히 하는 것 가능
-> col1, col2 순서로 나열되어 있음
-> 범위를 찾았기 때문에 현재 범위 내 인덱스에서 col3은 정렬된 상태 = 정렬에 인덱스 적용 가능

WHERE col1 = 100 ORDER BY col3
-> col1까지는 인덱스 범위를 특정 가능하지만 col3은 col2의 범위가 명확히지 않으므로 정렬되 상태가 아님
=> 정렬에 인덱스 적용 불가능
```

커버링 인덱스(Index Only Scan)

- 검색 결과가 많은 행에는 커버링 인덱스 효과적
- 쿼리의 실행에 필요한 컬럼이 인덱스에 모두 포함된 경우 테이블 자체를 엑세스 하지 않고 인덱스에만 엑세스해 쿼리를 해결 가능
- 자주 실행되는 쿼리가 인덱스만 검색되도록 컬럼을 추가해 사용
  - 인덱스의 디스크 공간, 갱신 성능은 떨어짐
  - 높은 빈도로 사용되는 경우 속도의 성능이 매우 효과적

OR과 인덱스

- AND는 다중 컬럼 인덱스로 효과를 볼 수 있다.

- OR는 각각의 조건에 인덱스가 있는 경우에 해결할 수 있다.

```
WHERE col1 = val1 OR col2 = val2
```

- 위의 경우 col1, col2로 인덱스를 각각 구성하고 인덱스를 통해 얻어진 ROWID 집합에 대해 UNION을 통해 최종 행을 판단 (인덱스 머지)
- 인덱스 머지가 안되는 경우 `UNION DISTINCT`를 사용해 두 개의 `SELECT`를 연결하는 방법이 효과적

최적의 인덱스를 찾기 위한 전략

인덱스 ≠ 후보키

- 인덱스는 후보키의 컬럼만 되는 것이 아닌 모든 컬럼에 대해 적용 가능
- 대부분의 인덱스는 WHERE 절의 검색 조건을 빠르게 실행하기 위해 필요 → "어떻게 하면 범위검색에 필요한 인덱스를 설계하는가?"
- WHERE 절(제한, Restrict)에서의 범위 검색을 빠르게 하기 위한 것 → 인덱스
- select list에서 프로젝션을 빠르게 하기 위한 것 → 커버링 인덱스

컬럼의 정렬 순서

- 어떤 순서로 컬럼을 정렬할지가 중요하다.
- 범위 검색, 정렬이 있는 경우 검색조건이 전방일치가 되도록 컬럼 배치
- 같은 컬럼으로 배치 순서가 다른 여러개의 인덱스를 만들어야 되는 경우도 존재 → 간과하지 말것!

카디널리티

- 카디널리티 : 집합에 포함된 요수의 갯수
- 여러가지 검색조건에 대해 카디널리티가가 높은 컬럼만 포함한 인덱스를 만드는 것은 중요
- 하나의 인덱스를 통해 여러가지 검색 조건을 해결 가능 → 갱신 오버헤드 감소, 디스크 공간 절약

최적의 조합을 찾아라

1. 어떤 유형의 인덱스가 사용 가능한지, 실행 계획의 어디에 인덱스를 사용 가능한지 판단
2. 카디널리티나 테이블 사이즈, 쿼리 실행 빈도등을 검토해 어떤 컬럼을 인덱스에 넣을지 판단

- 인덱스는 컬럼의 조합을 최적화 하는 것 → 쿼리가 빨리 수행된다고 해도 갱신 오버헤드가 크다면 효율적이지 않음

> 🔥 좋지 않은 인덱스 설계
>
> 1. 모든 컬럼에 인덱스가 있는 경우
> 2. 인덱스에 포함된 컬림이 한 개인 경우
> 3. 여러 개의 다중 컬럼 인덱스에 같은 컬럼의 조합이 있고 모두 같은 정렬로 된 경우
> 4. 0,1 or Y,N 과 같은 플래그 컬럼에 인덱스가 존재하는 경우

▼ 11.6 요약

- 인덱스 설계에 앞서 DB의 논리적 설계나 정규화가 중요
- 인덱스 설계에는 정답이 존재하지 않으므로 여러가지 조건과 경험에 의해 최적의 인덱스를 설계하는 것이 중요