

RealMySql 8.0

🕒 생성일	@2022년 4월 3일 오후 12:14
☰ 태그	

8. 인덱스

디스크 읽기 방식

- 데이터 저장 장치에 따른 읽기 속도 차이 (HDD vs SSD)
 - HDD : 데이터 저장용 원판
 - SSD : 플래시 메모리
- 디스크의 I/O
 - 랜덤 I/O : 디스크의 헤더를 읽고 쓸 위치로 움직임
 - 순차 I/O : 디스크의 헤더를 움직이지 않고 순차적으로 데이터 읽고 씀
- 디스크의 랜덤 I/O를 줄이기 위해 꼭 필요한 데이터만 읽도록 쿼리 구성 (검색 범위의 최소화)
- 인덱스 레인지 스캔 → 랜덤 I/O
- 풀 테이블 스캔 → 순차 I/O
- 데이터 웨어하우스에서는 순차 I/O를 위해 풀 테이블 스캔을 하는 경우가 있다.
- 디스크를 적게 읽기 위한 중국 게임 프로그래머의 방법

중국의 어떤 서버 개발자의 디비 설계

구글 번역!! Google Translate!!

=====

🌐 <https://202psj.tistory.com/1483>



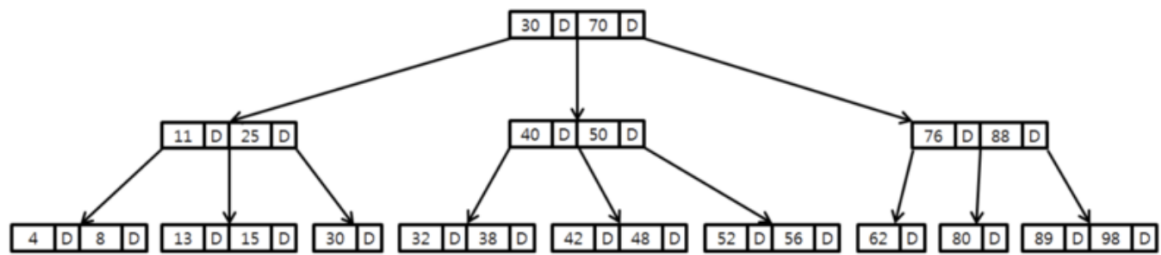
인덱스?

- 색인 (예. 전화번호 부, 책의 뒷페이지 찾아보기)
 - SortedList : 자료 저장 시, 정렬 상태를 유지
- 데이터 (책 내용)
 - ArrayList
- 두 알고리즘 특성 차이로 발생하는 성능으로 데이터 저장구조 고려
 - 데이터 저장 성능을 희생하여 읽기 성능을 올릴 것인가?
- 인덱스의 역할
 - Primary key
 - Secondary key
- 대표적인 인덱스 알고리즘
 - B-Tree : 컬럼의 값을 이용해 인덱싱
 - Hash : 해시 값을 계산하여 인덱싱, 값의 일부 검색에 제한
 - 검색 속도 $O(1)$
 - 왜 근데 인덱스로 안쓸까?
 - 여러가지 알고리즘 중 왜 b-tree 일까?

데이터베이스 인덱스는 왜 'B-Tree'를 선택하였는가
 데이터베이스의 탐색 성능을 좌우하는 인덱스. 인덱스는 데이터 저장, 수정, 삭제에 대한 성능을 희생시켜 탐색에 대한 성능을 대폭 상승하는 방식이라 볼 수 있다. DB의 인덱스는
 :: <https://helloinyong.tistory.com/296>

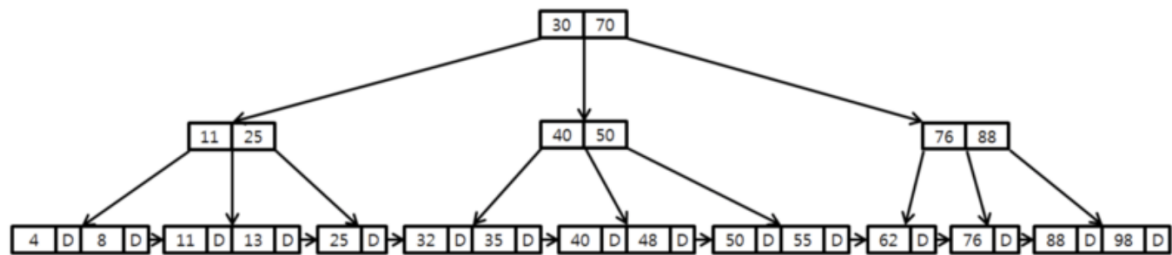


B-Tree, B+Tree



B-tree

- blanced tree → read black tree?
- 노드에 여러 키를 가질 수 있음
- 여러키는 대응되는 데이터를 한개씩 갖고 있음
- 리프 노드는 동일한 Depth
- 항상 정렬을 유지함



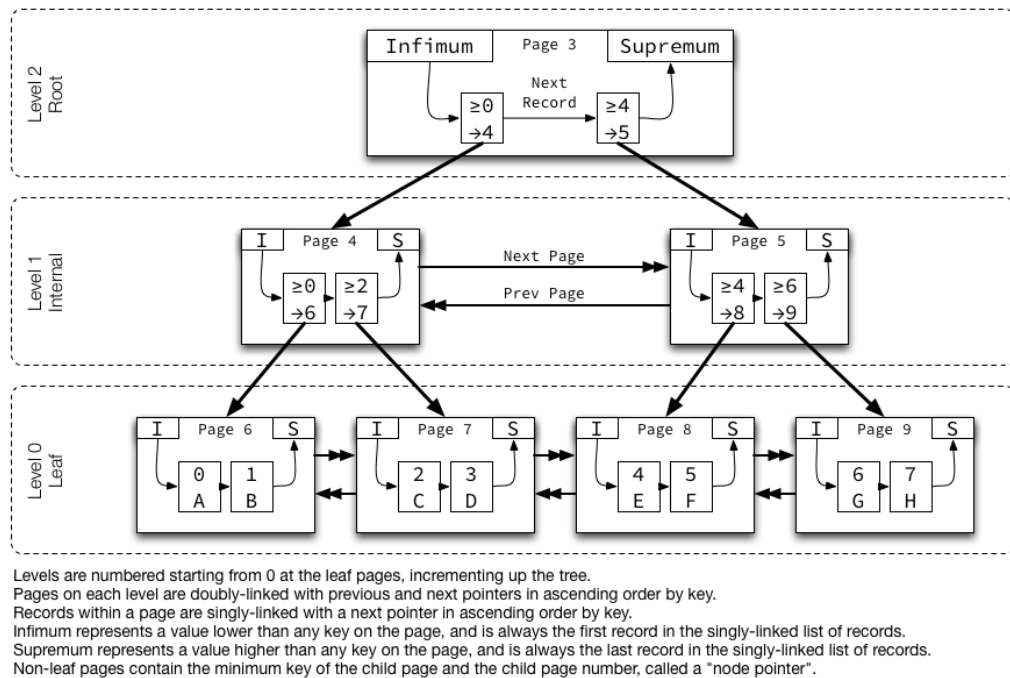
B+tree

- 리프 노드를 제외한 모든 노드는 키만 가지고 있음

B-Tree 인덱스

- 개요
 - 가장 범용적으로 사용되는 인덱싱 알고리즘
 - B+tree, B*tree가 사용됨

B+Tree Structure



- 리프 노드를 제외하고 모든 노드에서 데이터를 담지 않기 때문에 노드에 더 많은 키 저장
 - 트리의 높이가 낮아짐
- 구조 및 특성
 - 루트 노드 → 브랜치 노드 → 리프 노드
 - 브랜치 노드
 - 리프 노드의 키를 찾아가기 위한 정보 저장
 - 리프 노드
 - 실제 데이터 저장 (데이터 파일 주소 값)
 - 노드는 정렬 된 상태로 유지
- 엔진 별 인덱스 구조
 - MyISAM : 데이터 파일과 프라이머리 키
 - 힙 구조의 데이터 파일
 - 프라이머리 키 값과 무관하게 INSERT 순서대로 데이터가 저장
 - 레코드는 모두 ROWID 라는 물리적 주소값을 소유
 - 프라이머리 키, 세컨더리 인덱스 모두 ROWID를 포인터로 가짐


- InnoDB : 프라이머리 키
 - 프라이머리 키 순서대로 데이터가 클러스터링 되어 저장
 - 인덱스 값으로 프라이머리 키를 가짐
- 물리적 주소(MyISAM) vs 논리적 주소(InnoDB)
- 인덱스 키 추가
 - 키 값을 이용해 저장될 위치 검색
 - 위치 결정 후 → 키 값과 레코드의 주소 정보를 리프노드에 저장
 - 리프노드가 full일 경우 split을 이용해 리프 노드 분리
 - 브랜치 노드 생성
 - 디스크 I/O 발생으로 인해 지연 작업 필요
 - 프라이머리 키 or 유니크 인덱스의 경우에는 중복 체크로 인해 즉시 작업
- 인덱스 키 삭제
 - 키 값에 해당 하는 리프노드 탐색 후 삭제 마크 → 디스크 I/O 발생
 - InnoDB에서는 버퍼링 지연 처리
- 인덱스 키 변경
 - 단순 키 값의 변경 불가
 - 키 값 삭제 후, 새로운 키 값을 추가하는 형태
 - 결국 2번의 작업 수행 필요
- 인덱스 키 검색
 - 100% 일치 혹은 값의 앞부분(Left-most part)일치 경우 사용
 - 키 값의 뒷부분을 활용한 검색은 불가능
 - 인덱스 키 값 변형 후 빠른 검색 기능 사용 불가
 - 함수나 연산을 수행시, 키 값 변경으로 인덱스 내에 검색 불가
- 사용 영향 요소
 - 인덱스 키 값의 크기
 - 페이지 단위로 인덱스의 키 저장 형태
 - 키 값이 커지면 → 페이지의 수가 늘어남 → 탐색해야 할 페이지 증가 → 그에 따른 디스크 I/O 증가

- 키 값의 크기 증가 → 인덱스 크기 증가 → 메모리 공간 한정적 → 캐싱 대상 레코드 감소
 - Btree 깊이
 - 키 값의 크기 증가 → 페이지 깊이 깊어짐 → 디스크 I/O 증가
 - 선택도(기수성)
 - 전체 인덱스 100개, 유니크한 값의 수 10개 → 기수성 10
 - 중복된 값이 많으면 기수성과 선택도가 감소
 - 선택도가 높을수록 검색 대상 감소 → 빠른 검색 속도
 - 읽기 대상 레코드 건수
 - 100만건 중에 50만건을 읽는다면?
 - 인덱스 or 풀스캔
- 읽기 방식
 - 인덱스 레인지 스캔 : 검색해야할 인덱스의 범위가 결정 됐을 때
 - 정렬된 Btree의 특성 상, 정렬된 상태로 데이터 읽기 실행
 - 스캔 위치 검색 → 오름차순 or 내림차순 인덱스 읽기 → 최종 레코드 읽기
 - 커버링 인덱스를 이용하여 3번째 단계의 랜덤 읽기 감소
 - 인덱스 풀 스캔 : 쿼리의 조건절 컬럼이 인덱스의 첫번째 컬럼이 아닌 경우
 - (A,B,C) 순으로 구성된 인덱스 → B,C를 조건절로 사용할 경우
 - A를 이용할 수 없으므로 B와 C의 인덱스를 모조리 탐색
 - 데이터 풀 스캔보다는 효과적 → 테이블의 크기보다는 인덱스 범위가 작기 때문
 - 루스 인덱스 스캔 (인덱스 스킵 스캔)
 - 필요하지 않은 인덱스 키값은 무시하고 넘어감
 - GROUP BY 또는 MAX(), MIN()
 - MAX, MIN 경우 → 조건의 처음 혹은 마지막 인덱스 키만 scan 후 다음 조건으로 넘어감
 - 예) 부서별 직원번호가 높은 or 낮은 사람
 - 8.0 이전에는 GROUP BY에만 적용

- 8.0 이후에는 WHERE 조건절에서도 사용 가능
- 단점
 - WHERE 조건절에서 선행 컬럼의 인덱스의 유니크 수가(중복 수) 적을 때만 최적화 적용 가능
 - 쿼리가 인덱스의 존재하는 컬럼으로만 처리해야 함
 - 인덱스에 존재하지 않는 컬럼 조회시 데이터 풀스캔이 실행 하므로 인덱스 사용X
- 다중 컬럼 인덱스
 - 두 개 이상의 컬럼으로 구성된 인덱스
 - 첫번째 이후의 컬럼은 이전 컬럼에 의존해서 정렬
 - 두번째 컬럼의 데이터 값은 첫번째 데이터 값의 정렬 후 정렬
 - (1, 300)
 - (2, 202)
 - (2, 203)
 - 두번째 컬럼의 값이 더 크더라도, 앞에 위치 할 수 있음 (첫번째 컬럼의 값이 작을 경우)
- B-Tree 인덱스의 정렬 및 스캔 방향
 - 인덱스 생성 시 설정한 정렬 규칙에 따라 데이터 정렬
 - 오름차순, 내림차순 설정 → 사실 별로 필요 없어 보임
 - 읽기 방향만 바꾸면 되므로 그다지 중요치 않음
 - 그래서 인덱스는 항상 오름차순으로 정렬 됨
 - 2개 이상의 복합 컬럼에서 오름차순 & 내림차순 복합 설정시 → 내림차순 인덱스로 해결 (???? 왜)
 - InnoDB에서 역순 스캔이 정순 스캔보다 느린 이유

MySQL Ascending index vs Descending index

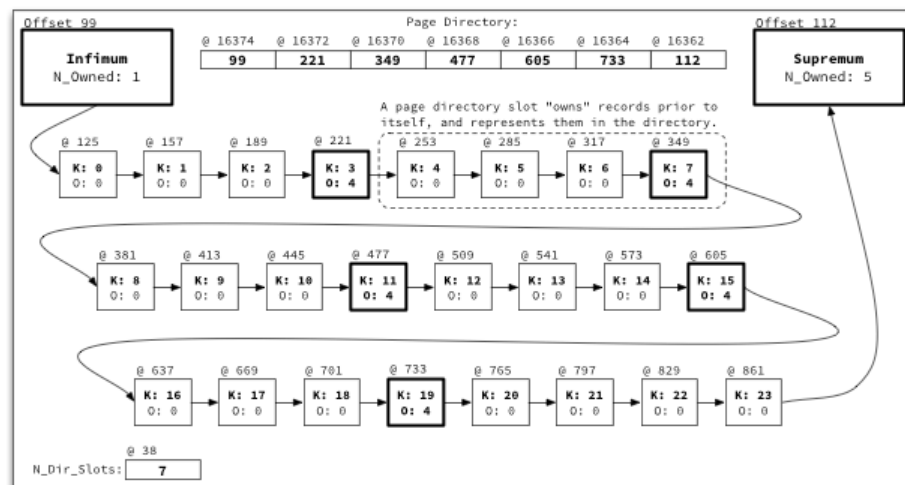
용어 정리 이 설명에서는 인덱스의 정렬 순서와 데이터 읽기 순서 등 방향에 대한 단어들이 혼재하면서, 여러 가지 혼란을 초래하기 쉬운 설명들이 있을 것으로 보인다. 그래서 우선 표

 <https://tech.kakao.com/2018/06/19/mysql-ascending-index-vs-descending-index/>



- 페이지 잠금이 정순 스캔에 유리
 - 리프 페이지는 Double linked list로 연결 되어 있지만
 - 페이지 잠금 과정에서 데드락 방지를 위해 B-Tree 원 → 오 순서로만 잠금을 획득
 - 정순 스캔은 잠금 획득이 쉽지만
 - 역순 스캔은 잠금 획득이 어렵다
- 페이지 내에서의 인덱스 레코드가 단방향으로만 연결된 구조
 - 페이지에 많은 키가 존재 한다. 20바이트의 키, 16K의 페이지 → 하나의 페이지당 600개의 키
 - 모든 키를 검색하면 시간이 오래걸림 → 정렬된 레코드 4~8개를 묶어 대표 키 관리
 - 이 대표키를 묶어서 관리하는 리스트 → 페이지 디렉토리

B+Tree Page Directory Structure



Infimum always owns only itself, so will always have a slot in the page directory with N_Owned = 1.
 Supremum always owns the last few records in the page, and is allowed to own less than 4 records (if the page has fewer).
 All directory slots will own a minimum of 4 and maximum of 8 records, except supremum, which may own fewer.
 The page directory grows "downwards" from offset 16376, the beginning of the FIL trailer; the first directory entry starts at 16374.

- 이 페이지 디렉토리는 Single linked list로 구성 → 오름 차순 검색은 쉽다. 내림 차순은 ??
- 가용성과 효율성
 - 효율성 : 인덱스 구성에 따른 차이 (효율 발생)
 - 동등 비교, 범위 조건인지에 따라 인덱스 활용이 다르다.
 - 작업 범위 결정 조건 (좋다)

- 필터링 or 체크 조건 (별루다)
- 가용성
 - Left-most 기준의 검색 조건
 - 첫번째 컬럼을 기준으로 인덱싱 해줬는데 안쓴다???
- 인덱스를 사용할 수 없는 조건
 - NOT-EQUAL 비교 (<>, NOT IN, IS NOT NULL) → 이거 빼고 다요...??
 - LIKE '%??'
 - 스토어드 함수나 다른 연산자로 인덱스 컬럼이 변경된 후 비교
 - NOT_DETERMINISTIC 속성의 스토어드 함수가 비교 조건에 사용된 경우
 - 스토어드 함수의 결과값이 캐싱되지 않고 매번 새로 계산하는 함수
 - 데이터 타입이 서로 다른 비교(인덱스 컬럼의 타입 변환 후 비교)
 - 문자열 데이터 타입의 콜레이션이 다른 경우 (utf ↔ euckr)
 - 다중 컬럼의 경우
 - 작업 범위 결정 조건
 - 해당
 - 1 ~ n-1 번째 컬럼까지 동등 비교 후
 - 동등 비교
 - 범위 비교
 - LIKE를 이용한 좌측 일치 패턴 ('문자%')
 - 해당하지 않음
 - 1번째 컬럼에 대한 조건 없음
 - 1번째 컬럼이 인덱스 사용 불가 조건일 경우

R-Tree 인덱스

- 공간 인덱스, 2차원의 데이터를 인덱싱
- GPS 혹은 지도 서비스의 좌표 저장
- MBR → 도형을 감싸는 최소 사각형의 크기

- 함수를 이용한 거리 기반의 검색
 - ST_Contains()
 - ST_Within()

전문 검색 인덱스

- 단어 알고리즘
 - 어근 분석 : 단어의 뿌리인 원형을 찾는 검색
 - n-gram : 인덱싱할 키워드의 최소 글자 지정 후 키워드를 쪼개서 인덱싱하는 방법
- 함수 기반 인덱스
 - 컬럼의 값을 변형 하여 인덱스 구성
 - 가상 컬럼을 이용하는 경우
 - 테이블에 새로운 컬럼을 추가, 테이블의 구조 변경
 - 함수를 이용한 경우
 - 테이블의 구조 변경하지 않고 활용

멀티 밸류 인덱스

- 기존의 1:1 형태의 키 → 값의 맵핑 형태가 아닌 1:N의 맵핑 형태
- Json 데이터 타입 지원을 위해 사용

```
{
  key1: value,
  key2: [a,b,c,d]
}
```

- 멀티 밸류 인덱스를 활용하기 위한 함수
 - MEMBER OF()
 - JSON_CONTAINS()
 - JSON_OVERLAPS()

클러스터링 인덱스

- 프라이머리 키 기준으로 비슷한 데이터끼리 묶어서 저장하는 형태
- 비슷한 값들을 동시에 조회 하는 경우가 많기 때문에
- InnoDB에서만 사용가능
- 프라이머리 키 값의 의존도가 높음
 - 프라이머리 키 기반의 검색은 빠름
 - 레코드의 저장 혹은 키 변경시 느림
- 프라이머리 키가 없을 경우?
 1. 프라이머리 키가 있으면 해당 키를 클러스터링 키로 선택
 2. NOT NULL 옵션의 유니크 인덱스 중에서 첫 번째 인덱스를 클러스터링 키로 선택
 3. AUTO_INCREMENT와 같은 유니크한 키를 가질 수 있는 키를 내부적으로 추가하고 클러스터링 키로 선택
- 클러스터링 인덱스 vs 논 클러스터링 인덱스
 - 데이터 정렬 vs 인덱스 페이지만 정렬
 - 테이블당 1개 vs 최대 개수 한정
 - 프라이머리 키 vs 다른 인덱스 키
 - 리프노드에 모든 컬럼 vs 프라이머리 키
 - Btree 인덱스와 다른점

유니크 인덱스

- 테이블이나 인덱스에 값이 유니크 해야함 (중복이 없어야 함)
- NULL은 값이 아니므로 2개이상 가질 수 있다 (MySQL에서는 유니크 속성 부여로 NULL 불가)
- 일반 세컨더리 인덱스와 다른 점이 거의 없음 (그냥 인덱스가 유니크한 컬럼이다?)

외래키

- InnoDB에서만 생성 가능
- 외래키 제약이 설정되면, 자동으로 연관되는 테이블의 컬럼에 인덱스 생성

- 외래키가 제거되지 않으면 자동으로 생성된 인덱스 삭제 불가
- 특징
 - 외래키와 연관된 테이블의 변경이 있을 경우 잠금 경합이 발생
 - 외래키와 연관되지 않은 컬럼의 변경은 잠금 경합을 최대한 발생 시키지 않음