1. 인덱스의 구조

모든 DBMS는 다양한 인덱스를 제공한다.

가장 일반적으로 사용되는 인덱스의 구조는 B*Tree 인덱스 구조이다. B*TREE 인덱스는 맨 위쪽 루트노드에서 브랜치 노드를 거쳐 맨 아래 리프 노드까지 연결되는 구조다.

루트에서 리프블록까지의 거리를 '인덱스 깊이' 라고 부르고 인덱스를 반복적으로 탐색할 때 성능에 영향을 미친다.

인덱스의 블록은 컨텐츠(KEY)와 페이지(ROWID)로 구성된다.

루트와 브랜치 블록은 각 하위 노드들의 데이터 값 범위를 키 값으로 그 키 값에 해당하는 블록을 찾는데 필요한 주소 정보를 페이지(ROWID)로 가진다.

블록내부의 인덱스는 KEY 값에 따라 항상 정렬되어있다. 키 값이 같을때는 ROWID 순으로 정렬된다. 리프 블록은 항상 인덱스 키 값 순으로 정렬돼 있기 때문에 범위 스캔이 가능하고, 정방향과 역방향 스캔이 둘 다 가능하도록 양방향 연결 리스트 구조로 연결돼 있다.

만약 데이터가 10억건이있다면 데이터에 따른 인덱스도 10억개가 존재한다. 10억개의 데이터를 저장하는데 대략 1억개 정도의 데이터블록이 필요하다면 10억개의 인덱스를 저장하는데는 대략 300만개의 데이터 블록이 필요하다. 이는 데이터 자체의 볼륨보다 데이터의 정보를 가진 인덱스의 볼륨이 1/3 정도 더 적다는 것을 의미한다.

ORACLE에서 인덱스 구성 칼럼이 모두 Null인 레코드는 인덱스에 저장하지 않는다. 반대로 인덱스 구성 칼럼 중 하나라도 NUll 값이 아닌 레코드는 인덱스에 저장한다. 이런 특성이 null 값 조회에 인덱스가 사용될 수 있는지를 결정하므로 인덱스를 설계하거나 SQL을 개발할 때 반드시 숙지해야한다.

인덱스를 생성하면 루트블록와 리프블록은 같다.

하나의 데이터 블록에 대략 400개 정도의 인덱스가 저장된다고 가정하면, 401번째 인덱스가 생성되면 루트블록와 리프블록은 분리된다.

루트와 리프블록이 하나의 블록에 위치하면 즉, 깊이 레벨이 0이면 최대 400개의 데이터를 인덱싱할 수 있다. 1레벨의 깊이의 브랜치블록이 존재할 경우 최대 400^2 개의 데이터를 인덱싱 할 수 있다. 같은 원리로 4레밸 깊이의 브랜치노드가 존재할 경우 최대 400^5(10조개) 개의 데이터를 인덱싱 할 수 있다.

루트 블록은 인덱스 트리의 최상위 블록으로 각 인덱스에 하위블록에 속한 인덱스의 값의 범위 및 블록에 대한 주소(BLOCK ADDRESS)를 가지고 있다. 루트 블록은 자신의 하위 블록의 수 만큼의 인덱스를 가진다.

브랜치 블록은 루트 블록과 리프 블록의 연결고리 역할을 하는 블록으로 각 인덱스에 하위블록에 속한 인데스의 값의 범위 및 블록에 대한 (BLOCK ADDRESS)를 가지고 있다. 각 브랜치 블록은 자신의 하위 브랜치 블록 또는 리프 블록의 수 만큼 인덱스를 가진다.

리프 블록의 각 인덱스는 키값과 하위 테이블의 데이터에 대한 ROWID로 구성되어있다. 키값 순서대로 정렬 되어있으며, 이전 이후의 리프 블록과 연결되어있다.

```
-- INDEX 블럭을 한 번 읽고 테이블 블록을 10번 읽는다.
SELECT * FROM 사원 WHERE 사번 BETWEEN 1 AND 10

-- INDEX 블럭을 한 번 읽고 테이블 블록을 10번 읽는다.
```

```
SELECT * FROM 사원 WHERE 사번 BETWEEN 10억 - 10 AND 10억

-- kEY값의 범위를 가지고 있는 BRANCH 인덱스를 탐색
SELECT * FROM 사원 WHERE 사번 BETWEEN 5억 AND 5억 + 10;
```

전체 데이터 대비 찾는 데이터의 수가 10%가 넘으면 인덱스를 활용한 탐색보다 전체 스캔이 더 빠르다.

RANDOM ACCESS VS. SEQUENTIAL ACCESS

RANDOM ACCESS는 하나의 블록에서 하나의 레코드(ROW)만 꺼내 읽는다. 수직적 탐색에 사용되는 방식이다. LEAF 블록에서 테이블에 접근하기 위해 사용하는 방식이다.

SEQEUNTIAL ACCESS는 블록에 존재하는 모든 레코드(ROW)를 읽는다. 수평적 탐색에 사용되는 방식이다. 낭비되는 레코드가 없어 적은비용으로 효율적인 탐색이 가능하다. MULTI BLOCK IO를 활용해 디스크 접근 횟수를 줄인다.

---

2. 인덱스 기본 원리

인덱스의 사용이 불가능 하거나 범위스캔이 불가능 한 경우

인덱스 컬럼의 가공(좌변 가공)

가공된 데이터의 형태로 정렬돼있지 않기 때문이다.

```
-- INDEX 사용 X
SELECT * FROM 업체 WHERE SUBSTR(업체명, 1, 2) = '대한';
-- INDEX 사용 O
SELECT * FROM 업체 WHERE 업체명 like '대한%';
-- INDEX 사용 X
SELECT * FROM 업체 WHERE 월급여 * 12 = 500000000;
-- INDEX 사용 O
SELECT * FROM 업체 WHERE 월급여 = 50000000/12

SELECT * FROM 주문 WHERE TO_CHAR(일시, 'yyyymmdd') - :dt
SELECT * FROM 주문 WHERE 일시 > to_date(:dt, 'yyyymmdd') and 일시 <
to_date(:dt , 'yyyymmdd') + 1
SELECT * FROM 주문 WHERE 계약구분 || 년월 || 일련번호 = 'C1312001'
SELECT * FROM 주문 WHERE 계약구분 = 'C' AND 년월 = '1312' AND 일련번호 = '001'
```

NULL 검색

INDEX 가 C1 + C2 + C3 일때 'WHERE C1 IS NULL' 조건이면 이면 인덱스를 사용하지 않고 'TABLE FULL SCAN' 한다.

그 이유는 C1, C2, C3 가 모두 NULL인 경우도 위의 조건을 만족하지만 인덱스를 생성하지 않아 'INDEX RANGE SCAN' 으로 찾아낼 수 없기 때문이다.

WHERE C1 IS NULL C2 IS NULL AND C3 = 'X' 이라는 조건이면 인덱스를 사용해 INDEX RANGE SCAN 한다. 그 이유는 인덱스에 포함되는 컬럼의 값 중 하나라도 NULL이 아니면 인덱스를 생성하기 때문이다.

묵시적 형변환

컬럼의 데이터타입과 상수의 데이터타입이 상이할 경우 문자를 숫자로 변환한 후 비교한다. 변환이 불가능 할 경우 에러가 발생한다. 단, LIKE 연산자일 경우 반대로 숫자를 문자로 변환한다.

---

3. 다양한 인덱스 스캔 방식

INDEX RANGE SCAN

작은 범위의 탐색에서는 빠른 속도를 보장하지만 탐색의 범위가 일정 정도를 넘어가면 테이블 전체를 스캔하는 것보다 비효율적일 수 있다. 인덱스 스캔하는 범위와 테이블 액세스 횟수를 얼마나 줄일수 있는지가 탐색 속도의 핵심이다. INDEX RAGE SCAN을 위해선 인덱스를 구성하는 선두 컬럼이 조건절에 반드시 포함되야한다.

```
-- index : 부서코드 + 이름
-- 조건절에 쓰인 이름 컬럼이 인덱스의 선두컬럼이 아니기때문에 INDEX RANGE SCAN이 불가하다.

SELECT *
FROM 사원
WHERE 이름 = '홍길동'
```

INDEX FULL SCAN

```
-- index : 성별 + 연봉
-- 조건절에 쓰인 이름 컬럼이 인덱스의 선두컬럼이 아니기때문에 INDEX RANGE SCAN이 불가하다.
-- 하지만 조건절에 있는 모든 컬럼이 인덱스의 컬럼에 속한다.

SELECT *FROM 사원
WHERE 연봉 > 1억
```

기본적으로 적장한 인덱스가 없을 경우 TABLE FULL SCAN을 수행한다.

조건절에 쓰인 컬럼이 인덱스의 선두 컬럼은 아니지만 조건절의 모든 컬럼이 인덱스 구성컬럼에 속하면 옵티마이저가 인덱스 활용시 이익이 있다고 판단할 경우 INDEX FULL SCAN 한다.

최종 결과 값이 적을 땐 TABLE FULL SCAN 보다 INDEX FULL SCAN이 효율적이고 최종 결과 값이 많을 땐 TABLE FULL SCAN이 효율적이다.

INDEX UNIQUE SCAN

UNIQUE 인덱스를 '=' 조건으로 조회하면 INDEX UNIQUE SCAN 한다.

INDEX SKIP SCAN

```
-- INDEX 선두컬럼이 조건절에 없으므로 INDEX RANGE SCAN 불가

-- 인덱스 : 성별 + 연봉

SELECT 성별, 연봉
FROM 사원
WHERE 연봉 > 7000
```

조회 조건이 인덱스 선두 컬럼이 아니기 때문에 INDEX RANGE SCAN의 사용이 불가하고 성별 처럼 인덱스 선두 컬럼의 DISTINCT 가 매우 낮아야 유리하다.

DISTINCT가 높으면 DISTINCT 의 수만큼 SKIP이 일어나기 때문이다. 인덱스 선두 컬럼이 BETWEEN LIKE, 부등호 일 때도 사용 가능하다.

INDEX FAST FULL SCAN

INDEX FAST FULL SCAN 은 INDEX FULL SCAN 보다 빠르다. 그 이유는 인덱스 트리 구조를 무시하고 인덱스 세그먼트 전체를 MULTOBLOCK READ 방식으로 스캔하기 때문이다.

조회 조건이 인덱스 선두 컬럼이 아니어서 INDEX RANGE SCAN의 사용이 불가하면 옵티마이저는 INDEX FULL SCAN을 선택한다. 이때 블록단위가 아닌 익스텐트(8개의 블록) 단위로 읽는다.

인덱스 리프블록을 MULTI-BLOCK I/O로 읽기 때문에 스캔 속도가 빠르다. TABLE RANDOM I/O가 발생하는 경우 사용이 불가하므로 인덱스에 포함되는 컬럼으로만 조회할 때 사용 가능하다.

INDEX RANGE SCAN DECENDING

INDEX RANGE SCAN 과 기본적으로 동일한 스캔 방식이다. 단 인덱스를 뒤쪽에서 부터 앞쪽으로 스캔하기 때문에 내림차순으로 정렬된 결과 집합을 얻는다.

---

4. ORACLE DBMS 구조

- SERVER PROCESS 는 CLIENT 의 USER PROCESS 만큼 생성된다.

- SERVER PROCESS 는 하나의 독립적인 공간인 PGA를 갖는다.

- ORECLE DATA BASE SERVER는 DATABASE와 INSTANCE 로 구성된다.

- DATABASE 는 3가지 종류의 파일(DATAFILES, CONTROL FILES, REDO LOG FILES)로 구성된다.

- DATAFILE 은 사용자가 저장한 데이터, DBMS 유지를 위한 스키마 정보가 저장된다.

- CONTROL FILE에는 파일의 경로가 저장된다.

- REDO LOG FILE 은 DATA에 변경사항이 생기면 그 기록을 쌓는다.

- SGA 메모리 영역은 DATA BUFFER CACHE/ REDO LOG BUFFER / Shared AREA 로 구성된다.

로그 버퍼

REDO LOG BUFFER 는 INSERT , DELETE , UPDATE 가 발생하면 그 기록을 저장.

DB 복구 시 과거에 실행한 SQL을 재 실행하기 위해 필요한 정보를 보관한다.

DML 발생시 DATA BUFFER CACHE 보다 앞서서 먼저 REDO LOG BUFFER에 저장하고 DATA BUFFER CACHE에 저장한다.

USER COMMIT 등 상황에서 COMMIT을 만나면 LGWR 가 REDO LOG BUFFFER에 있는 정보를 REDO LOG FILE에 저장

DATA BUFFER CACHE에 있는 값을 REDO LOG FILE 로 옮기지 않는 이유는 블록 단위 / RANDOM ACCESS 로 데이터를 읽어야 하는 비효율이 있기 때문이다.

REDO LOG BUFFER는 APPEND 방식으로 저장되고 순차적으로 저장된 데이터를 읽어 올수 있어 정합성 보장에 유리하고 빠르고 부하가 적다.

SHARED AREA

LIBRARY CACHE 에는 SQL FULL TEXT와 실행계획이 담겨있다.

DICTONARY CACHE 는 테이블 구조, 컬럼구조 , 인덱스 구조등에 대한 정보가 담겨있다.

SGA vs. PGA

SGA는 서버 프로세스들의 공용 공간이므로 경합과 그로인한 LOCK과 경합이 있다.

PGA는 서버 프로세스의 개인 공간으로 엑세스 속도는 빠르지만 용량이 적다.

BACKGROUND PROCESS

- PMON

좀비 프로세스 정리 및 관리

- SMON

인스턴스 리커버리는 다음과 같은 과정을 거친다.

ROLL FORWARD - ROLL BACK - CHECKPOINT

트랜잭션 중 인스턴스에 문제가 발생하면 DB를 복구하면서 REDO LOG FILE에 존재하는 소스를 재실행한다. 이를 ROLL FORWARD라 한다.

커밋하지 않은 트랜잭션을 되돌리는 것을 ROLL BACK 이라고 한다.

이 과정을 마치면 CHECKPOINT를 표시하고 DB를 정상화 한다.

이런 과정을 인스턴스 리커버리라 한다.

- DBWR 데이터 BUFFER CACHE 에 있는 데이터를 DATAFILE로 write 한다.

- LGWR RODO LOG BUFFER 에 있는 데이터를 REDO LOG FILES 로 write 한다.

- CKPT

CKPT를 만나면 다른 거래를 모두 막고 LGWR을 기동한 후 DBWR를 기동한다. 즉, 메모리와 데이터 파일을 동기화 하고 이를 보장해주는 역할을 한다.

---

5. 테이블 RANDOM ACCESS 부하

리프노드가 가진 ROWID는 다음과 같이 구성되어있다

- 데이터 오브젝트 번호(6) - 데이터 파일 번호(3) - 블록번호 - 로우번호

DATA BLOCK ADDRESS(데이터 오브젝트 번호 + 데이터 파일 번호 + 블록번호) + 로우번호(Location)

DB BLOCK ADDRESS 를 활용한 블록 접근법

DBA를 HASH FUNCTION에 투입한 결과값을 기반으로 메모리주소를 획득한다. DBA에 있는 정보는 메모리의 주소가 아닌 디스크의 주소다.

하지만 조회할때는 메모리를 먼저 조회해야하기때문에 HASH FUNCTION을 통해 해시 버킷을 찾아가 메모리의 블록주소를 얻는다.

HASH BUFFER CHAIN LATCH

BUFFER HEADER 에서 메모리 주소를 얻는 과정에서 다른 프로세스로인해 대기(BUFFER BLOCK 대기)와 획득시도를 반복한다.

BUFFER PINNING

한 프로세스의 다음 번 READ 시 현재 읽은 동일 block을 read 할 경우 대상 block이 메모리에서 떠나지 않도록 pin을 걸어두고, 해당 주소인 DBA가 가리키는 메모리 번지수를 PGA에 저장해 다음번에는 다이렉트로 찾아갈 수 있게하는 기법이다.

LOGICAL READ COUNT에 잡히지 않아 block count의 수가 감소한다.

CLUSTERING FACTOR

실제 데이터가 인덱스 키 순서대로 정렬되어있는 정도를 나타내는 지표.

인덱스를 순차적으로 읽어 이전 rowid 블록과 다음 rowid 블록이 상이할 때 + 1 증가.

인덱스 수직적 탐색 비용 + 인덱스 수평적 탐색 비용(리프블록 * 유효 인덱스 선택도) + 테이블 RANDOM 엑세스 비용(클러스터링 팩터 * 유효 테이블 선택도)

---

6. 테이블 RANDOM ACCESS 최소화 튜닝

인덱스 컬럼 추가

결과값에 대한 모든 필터링을 테이블이 아닌 인덱스에서 한다면 테이블 RANDOM 액세스에 대한 비효율은 없다.

```
-- index : 성별 + 연봉
```

```
-- 근무지에 대한 필터는 인덱스가 아닌 테이블에서 이루어진다.
SELECT *
FROM TSTOWN.사원
WHERE 성별 = '남'
AND 근무지 = '광주'
```

테이블 당 인덱스가 과도하게 많으면 실행계획 재사용이 힘들어 효율이 감소한다. 그렇기 때문에 기존 인덱스에 새로운 컬럼을 추가하는 방식이 추천된다.

```
-- index : 성별 + 연봉 + 근무지
-- 성별 + 연봉 인덱스에비해 리프노드의 스캔 범위는 늘지만 TABLE RANDOM ACCESS에 대한 비효율
은 없다.
SELECT *
FROM TSTOWN.사원
WHERE 성별 = '남'
AND 근무지 = '광주'
```

테이블을 생성하면 pK에 대한 인덱스가 하나 자동으로 생성되는데 그 인덱스에 조건절에 나타나는 컬럼을 추가해 RANDOM 엑세스 를 최소화할 수 있다. 단, 인덱스 컬럼이 많아지면 클러스터링 팩터가 안좋아지기도 한다.

```
-- index : 부서코드 + 성명 + 퇴사여부
-- SELECT 문을 위해 인덱스 컬럼 추가 시 비효율이 발생한다.
SELECT 부서코드 , 성명 , 퇴사여부
FROM 사원
WHERE 부서코드 = '1000'
AND 성명 = '홍길동'
```

---

7. IOT(INDEX ORGANIZED TABLE)와 클러스터

IOT

리프노드 : 키 + ROWID

IOT는 리프노드에 테이블의 모든 데이터가 포함되어 있는 구조다.

리프노드 저장공간이 부족해 인덱스 스플릿이 일어나면 데이터 테이블 자체가 리프노드 이기 때문에 rowid 변경되야한다. 이런 것을 막기 위해 IOT 리프노드는 rowid 대신 primary key를 갖는다.

rowid 대신 Primary key를 가지므로 Secondary INDEX로 테이블 access를 하기위해선 추가적인 수직적 탐색 + 수평적 탐색 과정을 거친다.

- 크기가 작고 NL 조인으로 반복되는 룩업 테이블 - 폭이 좁고 긴 테이블 - 넓은 범위를 주로 검색하는 테이블 - 데이터 입력과 조회 패턴이 서로 다른 테이블

클러스터드 인덱스

키 값이 같은 레코드가 한 블록에 모이도록 저장하는 구조다

한 블록에 모두 담을 수 없으면 새로운 블록을 할당해 클러스터 체인으로 연결한다.

물리적으로 같은 블록에 여러 테이블의 레코드 저장가능 - 조인 상태로 저장한다.

일반저인 B-TREE 구조로 해당 키 값을 저장하는 첫 번째 데이터 블록만을 가리킨다.

키 값은 항상 Unique이고 키 값과 테이블 레코드는 1:M 관계를 유지한다.

넓은 범위 검색에 유리하다.

PRIMARY 키에 대한 값이 자주 추가되는 경우나 수정이 발생하는 경우 클러스터드 인덱스의 사용에 좋지않다.

---

8. 인덱스 스캔 효율

- 인덱스 매칭도

- 비교 연산자 종류와 컬럼 순서에 따른 인덱스 레코드의 군집성

- 인덱스 선행 컬럼이 등치(=) 조건이 아닐 때 발생하는 비효율

- BETWEEN 조건을 IN-LIST로 바꾸었을 때 인덱스 스캔 효율

- INDEX SKIP SCAN을 이용한 비효유 해소

- 범위 조건을 남용할 때 발생하는 비효율

- 같은 컬럼에 두 개의 범위검색 조건 사용시 주의 사항

- BETWEEN 과 LIKE 스캔 범위 비교

- 선분 이력의 인덱스 스캔효율

- ACCESS PREDICATRE / FILTR PREDICAE

- INDEX FRAGMENAION

인덱스 매칭도

조회 조건이 = 이 아닌 컬럼은 가급적 인덱스 뒤쪽으로

- DRIVING 조건: 조건에 만족하는 데이터만 읽어왔다.

- CHECK 조건 : 조건에 만족하지 않는 데이터도 읽어왔다.

실행계획으로 효율성 보기

```
SELECT *
FROM EMP E, DEPT D
WHERE E.SAL > 1500
AND E.JOB = 'MANAGER'
AND D.DEPTNO = E.DEPTNO;

-- EXECUTION PLAN                                          A-ROWS
-- SELECT STATEMENT                                         3
--  NESTED LOOPS                                            3
--    TABLE ACCESS BY INDEX ROWID OF EMP (TABLE)            3   -- 인덱스 레인지
스캔을 통해 7건을 테이블 렌덤 액세스했지만 테이블에서 4건을 필터링 함(테이블랜덤엑세스 비효율)
--      INDEX RANGE SCAN OF IX_EMP_X01(INDEX)               7
--    INDEX UNIQUE SCAN OF PK_DEPT(INDEX)                   3
--  TABLE ACCESS BY INDEX ROWID OF DEPT(TABLE)              3

-- PREDICATE INFORMATION
-- 3 - FILTER("E", "JOB" == 'MANAGER')
-- 4 - ACCESS("E", "SAL" > 1500)
-- 5 - ACCESS("D", "DEPTNO" = "E"."DEPTNO")
```