**3.3. 인덱스 스캔 효율화**

**1. 인덱스 탐색**: 수직적 탐색 + 수평적 탐색

\* 루트 블록: 자신이 가리키는 주소(화살표)로 찾아간 블록에는 자신의 키 값보다 크거나 같은 값을 가진 레코드가 저장돼 있음(루트 블록 키 값 <= 리프 블록 데이터)

\* 점선 화살표(-->): 블록 내에서 시작점을 찾는 과정을 표시

\* 실선 화살표(🡪): 논리적인 스캔 시작점과 끝점을 표시

\* 수직적 탐색: 인덱스 스캔의 시작점을 찾는 과정

\* 수평적 탐색: 인덱스 스캔의 종료 지점을 찾는 과정

**<조건절 1: 그림 3-20>**

|  |  |  |  |
| --- | --- | --- | --- |
| 루트 블록 | | | |
| L | A | B | C |
| M | 3 | 3 | 3 |
| C |  |  |  |

|  |
| --- |
| C1 |
| C2 |

|  |  |  |  |
| --- | --- | --- | --- |
| A | A | A | A |
| 1 | 1 | 2 | 2 |
| 리프 1 | | | |

|  |  |  |  |
| --- | --- | --- | --- |
| A | B | B | B |
| 3 | 1 | 2 | 3 |
| 리프 2 | | | |

|  |  |  |  |
| --- | --- | --- | --- |
| B | B | B | C |
| 3 | 3 | 4 | 1 |
| 리프 3 | | | |

|  |  |  |  |
| --- | --- | --- | --- |
| C | C | C | C |
| 2 | 2 | 4 | 5 |
| 리프 4 | | | |

|  |
| --- |
| C1 |
| C2 |

WHERE C1 = ‘B’

: 수직적 탐색을 통해 C1=‘B’인 레코드를 찾고, C1=‘C’인 레코드를 만나는 순간 스캔을 멈춤.

**<조건절 2: 그림 3-21>**

|  |
| --- |
| C1 |
| C2 |

|  |  |  |  |
| --- | --- | --- | --- |
| 루트 블록 | | | |
| L | A | B | C |
| M | 3 | 3 | 3 |
| C |  |  |  |

WHERE C1 = ‘B’

AND C2 = 3

|  |
| --- |
| C1 |
| C2 |

|  |  |  |  |
| --- | --- | --- | --- |
| A | A | A | A |
| 1 | 1 | 2 | 2 |
| 리프 1 | | | |

|  |  |  |  |
| --- | --- | --- | --- |
| A | B | B | B |
| 3 | 1 | 2 | 3 |
| 리프 2 | | | |

|  |  |  |  |
| --- | --- | --- | --- |
| B | B | B | C |
| 3 | 3 | 4 | 1 |
| 리프 3 | | | |

|  |  |  |  |
| --- | --- | --- | --- |
| C | C | C | C |
| 2 | 2 | 4 | 5 |
| 리프 4 | | | |

: 수직적 탐색을 통해 C1=‘B’이고 C2=3인 첫 번째 레코드를 찾고, C1=‘B’인 레코드 중에서 C2=4인 레코드를 만나는 순간 스캔을 멈춤.

**<조건절 3: 그림 3-22>**

|  |
| --- |
| C1 |
| C2 |

|  |  |  |  |
| --- | --- | --- | --- |
| 루트 블록 | | | |
| L | A | B | C |
| M | 3 | 3 | 3 |
| C |  |  |  |

WHERE C1 = ‘B’

AND C2 >= 3

|  |
| --- |
| C1 |
| C2 |

|  |  |  |  |
| --- | --- | --- | --- |
| A | A | A | A |
| 1 | 1 | 2 | 2 |
| 리프 1 | | | |

|  |  |  |  |
| --- | --- | --- | --- |
| A | B | B | B |
| 3 | 1 | 2 | 3 |
| 리프 2 | | | |

|  |  |  |  |
| --- | --- | --- | --- |
| B | B | B | C |
| 3 | 3 | 4 | 1 |
| 리프 3 | | | |

|  |  |  |  |
| --- | --- | --- | --- |
| C | C | C | C |
| 2 | 2 | 4 | 5 |
| 리프 4 | | | |

: 수직적 탐색을 통해 C1=’B’이고 C2>=3인 첫 번째 레코드를 찾고, C1=’C’인 레코드를 만날 때까지 스캔하다가 멈춤. C2>=3 조건절은 스캔을 멈추는데 역할을 하지 못함

**<조건절 4: 그림 3-23>**

|  |
| --- |
| C1 |
| C2 |

|  |  |  |  |
| --- | --- | --- | --- |
| 루트 블록 | | | |
| L | A | B | C |
| M | 3 | 3 | 3 |
| C |  |  |  |

WHERE C1 = ‘B’

AND C2 <= 3

|  |
| --- |
| C1 |
| C2 |

|  |  |  |  |
| --- | --- | --- | --- |
| A | A | A | A |
| 1 | 1 | 2 | 2 |
| 리프 1 | | | |

|  |  |  |  |
| --- | --- | --- | --- |
| A | B | B | B |
| 3 | 1 | 2 | 3 |
| 리프 2 | | | |

|  |  |  |  |
| --- | --- | --- | --- |
| B | B | B | C |
| 3 | 3 | 4 | 1 |
| 리프 3 | | | |

|  |  |  |  |
| --- | --- | --- | --- |
| C | C | C | C |
| 2 | 2 | 4 | 5 |
| 리프 4 | | | |

: 수직적 탐색을 통해 C1=’B’인 첫 번째 레코드를 찾고, 스캔하다가 C2>3인 첫 번째 레코드를 만나는 순간 스캔을 멈춤. C2<=3 조건절은 수직적 탐색 과정에 사용되지 않았고, 스캔을 멈추는 데 중요한 역할을 수행

**<조건절 5: 그림 3-24>**

|  |
| --- |
| C1 |
| C2 |

|  |  |  |  |
| --- | --- | --- | --- |
| 루트 블록 | | | |
| L | A | B | C |
| M | 3 | 3 | 3 |
| C |  |  |  |

WHERE C1 = ‘B’

AND C2 BETWEEN 2 AND 3

|  |
| --- |
| C1 |
| C2 |

|  |  |  |  |
| --- | --- | --- | --- |
| A | A | A | A |
| 1 | 1 | 2 | 2 |
| 리프 1 | | | |

|  |  |  |  |
| --- | --- | --- | --- |
| A | B | B | B |
| 3 | 1 | 2 | 3 |
| 리프 2 | | | |

|  |  |  |  |
| --- | --- | --- | --- |
| B | B | B | C |
| 3 | 3 | 4 | 1 |
| 리프 3 | | | |

|  |  |  |  |
| --- | --- | --- | --- |
| C | C | C | C |
| 2 | 2 | 4 | 5 |
| 리프 4 | | | |

: C1과 C2 조건절 모두 스캔 시작과 끝 지점을 결정하는데 중요한 역할을 수행

**<조건절 6: 그림 3-25>**

|  |
| --- |
| C1 |
| C2 |

|  |  |  |  |
| --- | --- | --- | --- |
| 루트 블록 | | | |
| L | A | B | C |
| M | 3 | 3 | 3 |
| C |  |  |  |

WHERE C1 BETWEEN ‘A’ AND ‘C’

AND C2 BETWEEN 2 AND 3

|  |
| --- |
| C1 |
| C2 |

|  |  |  |  |
| --- | --- | --- | --- |
| A | A | A | A |
| 1 | 1 | 2 | 2 |
| 리프 1 | | | |

|  |  |  |  |
| --- | --- | --- | --- |
| A | B | B | B |
| 3 | 1 | 2 | 3 |
| 리프 2 | | | |

|  |  |  |  |
| --- | --- | --- | --- |
| B | B | B | C |
| 3 | 3 | 4 | 1 |
| 리프 3 | | | |

|  |  |  |  |
| --- | --- | --- | --- |
| C | C | C | C |
| 2 | 2 | 4 | 5 |
| 리프 4 | | | |

: C1 조건절은 스캔 시작과 끝 지점을 결정하는 데 중요한 역할을 수행, C2의 경우 스캔량을 줄이는 데 거의 역할을 수행하지 못함 – 인덱스 선행 컬럼 또는 선두 컬럼에서 범위검색 조건이 사용 될 경우 후행 조건절은 인덱스 필터 조건으로…

**2. 인덱스 스캔 효율성**

- 인덱스 선행 컬럼이 조건절에 없거나 ‘=’ 조건이 아니면 스캔 과정에서 비효율이 발생 – p183. [그림 3-29] 참조

**> 인덱스 스캔 효율성 측정**: SQL트레이스 활용 – p184. SQL 트레이스 구문 예제: 읽어들인 ROW 개수 대비 CR이 적을수록 그 효율성이 높음.

\* cr: Block Read Count

\* 선두 컬럼: 인덱스 구성상 ‘맨 앞쪽’에 있는 컬럼을 지칭할 때 사용

\* 선행 컬럼: 어떤 컬럼보다 ‘상대적으로 앞쪽’에 놓인 컬럼을 지칭할 때 사용

**3.3.3. 액세스 조건과 필터 조건 –** p185. [그림 3-30] 참조

① 인덱스 액세스 조건: 인덱스 스캔의 범위를 결정 – 스캔의 시작과 종료 지점

② 인덱스 필터 조건: 테이블로 액세스할지를 결정

③ 테이블 필터 조건: 쿼리 수행 다음 단계로 전달하거나 최종 결과집합에 포함할지를 결정

다음 단계 또는 최종 결과집합

**\* 옵티마이저의 비용 계산 원리** – 7.1 상세 설명

비용 = 인덱스 수직적 탐색 비용 + 인덱스 수평적 탐색 비용 + 테이블 랜덤 액세스 비용

= 인덱스 루트와 브랜치 레벨에서 읽는 블록 수

+ 인덱스 리프 블록을 스캔하는 과정에 읽는 블록 수

+ 테이블 액세스 과정에 읽는 블록 수

**3.3.4 비교 연산자 종류와 컬럼 순서에 따른 군집성**

- 테이블과 달리 인덱스에는 ‘같은 값(=)’을 갖는 레코드들이 서로 군집

- 조건절에서 인덱스 중에서 어느 하나를 누락하거나 ‘=’ 조건이 아닌 연산자(범위검색 연산자 = 부등호, BETWEEN, LIKE)로 조회하면 조건절을 만족하는 레코드가 서로 흩어진 상태(인덱스 필터 조건 적용 대상)가 됨(p186. [그림 3-31], p189. [표 3-1] 참조).

**3.3.5 인덱스 선행 컬럼이 등치(=) 조건이 아닐 때 생기는 비효율**

**- 인덱스 스캔 효율성 극대화**

: 인덱스 컬럼을 조건절에 모두 등치(=) 조건으로 사용할 때 가장 좋음: 인덱스 리프 블록을 스캔하면서 읽은 레코드는 하나도 걸러지지(인덱스 필터 과정) 않고 모두 테이블 액세스로 이어짐: p192. [그림 3-32] 참조: 인덱스 구성 = 『아파트시세코드+평형+평형타입+인터넷매물』

**- 인덱스 스캔 비효율이 발생하는 경우**

① 인덱스 선행 컬럼이 조건절에 없는 경우

② 인덱스 선두 컬럼 또는 인덱스 선행 컬럼의 조건절에서 범위검색(부등호, BETWEEN, LIKE) 조건 사용

-> p193. [그림 3-33] 참조: 인덱스 구성 = 『인터넷매물+아파트시세코드+평형+평형타입』

**3.3.6 BETWEEN을 IN-List로 전환**

- IN-List 개수만큼 UNION ALL 브랜치가 생성되고 각 브랜치마다 모든 컬럼을 ‘=’(등치) 조건으로 검색하여 범위검색의 비효율성 개선

- 실행계획에서 INLIST ITERATOR 오퍼레이션 수행: p194 SQL 구문, p195. [그림 3-34], 실행계획 참조: 인덱스 구성 = 『인터넷매물+아파트시세코드+평형+평형타입』

- IN-List 개수만큼 인덱스 스캔 수행: P196. dbms\_xplan.display\_cursor 함수를 이용한 Row Source별 수행 통계 자료, p197. [그림 3-35] 참조

**> BETWEEN 조건을 IN-List로 전환할 때 주의 사항**

- IN-List 개수가 많지 않아야 됨, IN-List 개수가 많을 경우 BETWEEN 조건에서 리프 블록을 많이 스캔하는 비효율보다 IN-List 개수만큼 브랜치 블록을 반복 탐색하는(수직적 탐색) 비효율이 더 클 수 있음: p198. [그림 3-36] 참조

- 인덱스 스캔 과정에서 선택되는 레코드들이 서로 멀리 떨어져 있을 때만 유용: p199. [그림 3-37] 참조

**3.3.7 Index Skip Scan 활용**

: 선두 컬럼이 BETWEEN(범위검색 조건)이어서 나머지 검색 조건을 만족하는 데이터들이 서로 멀리 떨어져 있을 때 큰 효과를 얻을 수 있음.

* 월별고객별판매집계 인덱스 구성, 조건절, 그리고 스캔 방식에 따른 성능 비교

|  |  |  |  |
| --- | --- | --- | --- |
| 인덱스 구성 | 판매구분 + 판매월 | 판매월 + 판매구분 | |
| 판매구분 데이터 군집성 | 판매구분을 기준으로 정렬 | 판매월 기준으로 정렬 | |
| 블록 I/O | 281 | BETWEEN | 3,090 |
| IN-List | 314 |
| Skip Scan | 300 |

> 예제로 제시된 월별고객별판매집계 테이블의 특정 인덱스 기준 데이터(판매구분 -A: 10%, B: 90%)가 가지고 있는 각각의 Distinct Value 점유율 차이로 인덱스 구성에 따른 군집성의 차이가 발생하고, 이로 인한 블록 I/O 처리 성능에 큰 영향을 미치게 됨. 따라서 각각의 케이스에 맞는 조건절과 스캔 방식을 적용할 필요가 있음.

**3.3.8 IN 조건은 ‘=’인가**

- IN 조건은 ‘=’이 아님 -> 인덱스 구성에 따라 성능의 차이 발생

- 스캔 대상 인덱스 리프 블록에 있는 데이터의 군집성에 따라 스캔 성능의 차이가 결정됨:

- 군집성이 떨어져 있을 경우 IN-List Iterator 방식으로 처리하는 것이 효과적이지만, 군집성이 좋을 경우 선두 컬럼을 인덱스 엑세스 조건으로 사용하고 IN 조건에서 사용되는 후행 컬럼을 인덱스 필터 조건으로 처리하는 것을 권장함.

> SQL 구문: P203. 참조

select \*

from 고객별가입상품

where 고객번호 = :cust\_no

and 상품ID in (‘NH00037’, ‘NH00041’, ‘NH00050’)

> 인덱스 구성에 따른 인덱스 스캔 처리 방식 비교

|  |  |  |
| --- | --- | --- |
| 인덱스 | 상품ID + 고객번호 | 고객번호 + 상품ID |
| 처리 방식 | IN-List Iterator 방식(인덱스 엑세스 조건) | 인덱스 필터 조건으로 처리 |
| 참고 자료 | p204. [그림 3-39] | p204. [그림 3-40] |

> NUM\_INDEX\_KEYS 힌트 활용

- IN-List Iterator를 액세스 조건 또는 필터 조건으로 유도

- NUM\_INDEX\_KEYS 힌트의 세 번째 인자: 해당 순번의 인덱스 컬럼까지만 인덱스 액세스 조건으로 사용

Ex> select /\*+ num\_index\_keys(a 고객별가입상품\_X 1)\*/ \* - p207 실행계획 참조

select /\*+ num\_index\_keys(a 고객별가입상품\_X 2)\*/ \* - p208 실행계획 참조

**3.3.9 BETWEEN과 LIKE 스캔 범위 비교**: **BETWEEN < LIKE**

- LIKE보다 BETWEEN이 스캔 검색 범위가 보다 좁아 인덱스 스캔 과정에서 효용성이 보다 뛰어남: p210. [그림 3-41], [그림 3-42] 참조

**3.3.10 범위검색 조건을 남용할 때 생기는 비효율**

- 인덱스 컬럼에 범위검색 조건을 남용하면 인덱스 스캔 비효율이 발생, 대량 테이블을 넒은 범위로 검색할 때는 그 영향이 매우 클 수 있음

**3.3.11 다양한 옵션 조건 처리 방식의 장단점 비교**

**> OR 조건 활용**

* 인덱스 액세스 조건으로 사용 불가
* 인덱스 필터 조건으로도 사용 불가
* 테이블 필터 조건으로만 사용 가능
* 단, 인덱스 구성 컬럼 중 하나 이상이 Not Null 컬럼이면, 18c부터 인덱스 필터 조건으로 사용 가능

--> 이 방식의 유일한 장점은 옵션 조건 컬럼이 NULL 허용 컬럼이더라도 결과집합을 보장한다는 것, 되도록 사용 자제 요망

**> LIKE/BETWEEN 조건 활용**

① 인덱스 선두 컬럼에 사용하는 것은 금물

② NULL 허용 컬럼에 사용하는 것은 금물 – null인 데이터가 결과 집합에서 누락됨

③ 숫자형 컬럼에 사용하는 것은 금물(숫자형이면서 인덱스 액세스 조건으로 사용 가능한 컬럼) – 자동 형변환이 발생하여 인덱스 필터 조건으로 변경됨

④ 가변 길이 컬럼에 사용할 경우 특히, LIKE 구문에서 사용시 컬럼 값 길이가 고정적 이어야 함

--> 변별력이 좋은 필수 조건이 있는 상황에서만 유효: 필수 조건 컬럼을 인덱스 선두에 두고 액세스 조건으로 활용할 경우

**> UNION ALL 활용**

- 옵션 조건 컬럼도 인덱스 액세스 조건으로 사용.

- SQL 코딩량이 길어진다는 단점을 가짐

: 『 :variable is null/:variable is not null 』 - 특정 변수에 값을 입력 여부에 따라 실행 인덱스 선택 스캔 실행: p221. 실행계획 참조

**> NVL/DECODE 함수 활용**

: UNION ALL 보다 단순하면서도 UNION ALL과 같은 성능을 냄

- 옵션 조건 처리용 NVL/DECODE 함수를 여러 개 사용하면 그중 변별력이 가장 좋은 컬럼 기준으로 한 번만 OR Expansion이 발생

**\* Dynamic SQL** – p224. 참조

- iBatis SQL 매핑 프레임워크에서 옵션 조건을 <isEmpty> 구문으로 처리

- 인덱스를 잘 구성할 경우 <isEmpty> 구문에서 옵션 조건에 ‘=’ 연산자 사용할 경우, 변별력 있는 컬럼을 액세스 조건으로 사용 가능

* + 1. **함수호출부하 해소를 위한 인덱스 구성**

**> PL/SQL 함수의 성능적 특성**

* PL/SQL 사용자 정의 함수가 느린 이유

1. 가상머신(VM) 상에서 실행되는 인터프리터 언어
2. 호출 시마다 컨텍스트 스위칭 발생
3. 내장 SQL에 대한 Recursive Call 발생 – 함수 적용 대상 건수 만큼 발생

\* Context Switch: CPU가 해당 프로세스를 실행하기 위한 해당 프로세스의 정보들로 PCB(Process Control Block)에 저장

\* PCB의 저장정보

- 프로세스 상태 : 생성, 준비, 수행, 대기, 중지

- 프로그램 카운터 : 프로세스가 다음에 실행할 명령어 주소

- 레지스터 : 누산기, 스택, 색인 레지스터

- 프로세스 번호

\* 참고로 Context Switching 때 해당 CPU는 아무런 일을 하지 못한다. 따라서 컨텍스트 스위칭이 잦아지면 오히려 오버헤드가 발생해 효율(성능)이 떨어진다.  
출처: <https://jeong-pro.tistory.com/93> [기본기를 쌓는 정아마추어 코딩블로그]

**> 효과적인 인덱스 구성을 통한 함수 호출 최소화**

* 인덱스 구성에 따른 PL/SQL 함수[ex->encryption( :phone\_no )] 사용 예제

|  |  |  |  |
| --- | --- | --- | --- |
| SQL | 인덱스 | 인덱스 구성 | 후행 조건 엑세스 여부 |
| select \* from 회원  where 생년 = ‘1987’  and 암호화된\_전화번호 = encryption( :phone\_no ) | 회원\_X01 | 생년 | X – 인덱스 필터 조건 |
| 회원\_X02 | 생년, 생월일, 암호화된\_전화번호 | X – 인덱스 필터 조건 |
| 회원\_X03 | 생년, 암호화된\_전화번호 | O – 인덱스 액세스 조건 |