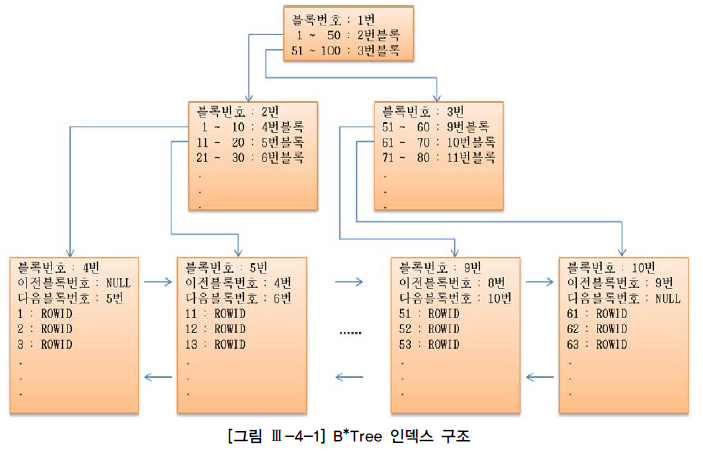
**제 1절 인덱스 구조 및 종류의 이해**

**1. 인덱스의 구조**



- 인덱스 탐색 순서

1) 수직적 탐색 : Root - Branch - Leaf. 읽고자 하는 시작점 검색. Random I/O

2) 수평적 탐색 : Leaf Block의 시작점부터 종료점 까지. Sequential I/O

3) 테이블 Random Access : Data 블록을 읽는 경우

\* 비용 : 테이블 Radom Access >수직적 탐색 > 수평적 탐색

- 기본 구조 : Root Node, Branch Node, Leaf Node

- Root Node : 가장 상위노드로 하위 Branch Node 수만큼의 Row

- Branch Node : Root와 Leaf의 연결고리로 자기 하위 Leaf Node 수만큼의 Row

- Leaf Node : Key + Row ID로 구성.

**2. 인덱스의 기본 원리**

- 인덱스의 사용이 불가능 하거나 범위스캔이 불가능한 경우

1) 인덱스 컬럼의 좌변가공 -> Where절의 좌변을 임의로 변경한 경우 -> Index Full Scan

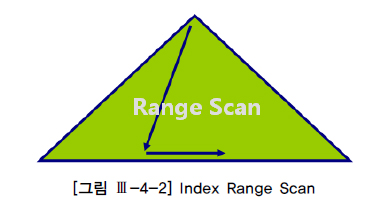
2) Null 값의 검색 -> Index Full Scan

3) 묵시적 형 변환 : 컬럼과 상수의 Data Type이 상이할 경우

- 문자를 숫자로 변환 후 비교. 변환이 불가능 할 경우 에러 발생

\* Like 연산자일 경우 반대로 숫자를 문자로 변경.

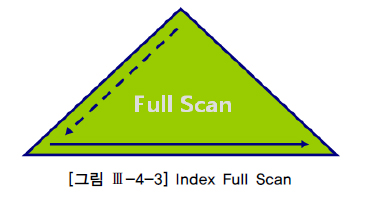
**3. 다양한 인덱스 스캔 방식**



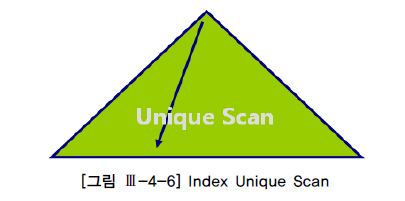
- Index Range Scan : SQL 튜닝의 핵심 원리로서 항상 빠른 속도를 보장하지는 않으며 테이블로 액세스 하는 횟수를 얼마만큼 줄이고 스캔하는 범위를 줄일수록 좋다.

\* 이 경우 인덱스를 구성하는 선두 컬럼을 조건절에 사용한다. -> 정렬 원칙!

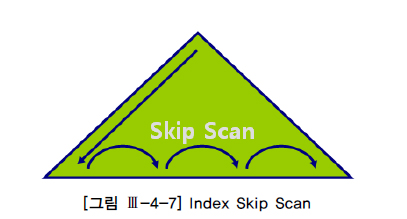
ex) Select \* From 사원 Where 이름 = ‘홍길동’ 인 경우 인덱스에 이름이 선두에 온다.



- Index Full Scan : 적당한 인덱스가 없을 경우 Table Full Scan을 수행하지만 조회 조건의 인덱스가 있고 선두 컬럼이 아니지만 옵티마지어가 인덱스 활용 시 이익이 판단할 경우 실행. \* 최종 결과 값이 적을 때 Full Table Scan보다 효율적이지만 많을 경우 Full Table Scan이 더 효율적.



- Index Unique Scan : 수직적 스캔만 발생하며 Unique 인덱스 일 경우 사용. ‘= 조건’ 일 경우만 사용한다.



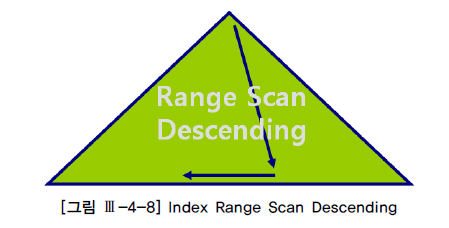
- Index Skip Scan : 조회 조건이 인덱스 선두 칼럼이 아니고 선두 칼럼의 Distinct가 매우 낮을 때 사용. Between, Like, 부등호 일 때도 사용 가능.

\*Distinct란 Data 종류의 개수를 의미.

- Index Fast Full Scan : 전체 인덱스를 Full Scan하며 Multi-Block I/O (여러개를 한꺼번에)며 속도가 빠르고 Index의 논리적 순서와 무관하게 물리적으로 읽으며 키 컬럼 순서와 무관하게 결과가 출력.

|  |  |
| --- | --- |
| Index Full Scan | Index Fast Full Scan |
| 1. 인덱스 구조를 따라 스캔  2. 결과집합 순서 보장  3. Single Block i/O  4. 병렬스캔 불가 (파티션시 가능)  5. 인덱스에 포함되지 않은 컬럼 조회시에도 사용 가능 | 1. 세그먼트 전체를 스캔  2. 결과집합 순서 보장 안됨  3. Multi Block I/O  4. 병렬스캔 가능  5. 인덱스에 포함된 컬럼만으로만 조회할 때 가능 -> Select 절에 인덱스에 있는 것만! |

- In-List Iterator : In의 데이터 개수만큼 진행을 하는 것



- Index Range Scan Descending : Index Range Scan을 반대방향부터 진행하는 것

- 그 외 Index Combine의 방식이 있다.

**I/O 튜닝의 핵심 원리**

- Sequential 액세스의 선택도를 높인다,

- Random 액세스의 발생량을 줄인다.

**4. 테이블 Random Access 부하**

- Row ID의 구조

1) 확장적구조 (8i 부터)

- 데이터 오브젝트(Index, Table 등) 번호 (6자리)

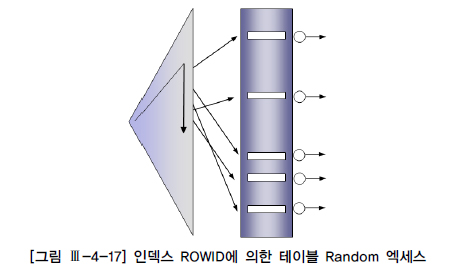
- 데이터 파일 번호 (3자리)

- 블록번호 (6자리)

- 로우번호 (3자리)

2) Restricted 구조 (8i 이전)

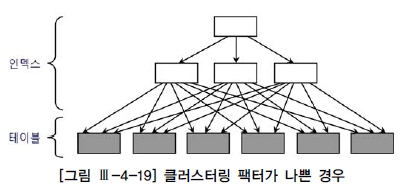
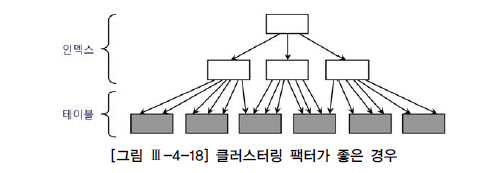
- 블록번호, 파일번호, 로우번호



|  |  |
| --- | --- |
| Sequential Access | Random Access |
| - 하나의 블록에서 순차적으로 읽는다.  - 알차게 빼 먹는다  - Index Leaf Block에서 Read / Full Scan  - 적은 비용  - Full Scan일 경우 Multi Block I/O 가능 | - 주로 하나의 블록에서 하나의 레코드만  - 높은 비용 . 낮은 효율  - Rowid 이용 테이블 Access  - DBA를 이용한 인덱스 수직적 탐색  - 클러스터링 팩터가 낮을 때 높은 성능  - Single Block I/O |

- Clustering Factor (CF) : 인덱스를 순차적으로 읽어 이전 Rowid 블록과 다음 Rowid 블록이 상이할 때 +1 증가

|  |  |
| --- | --- |
| C.F 좋을 때 (수치가 낮은 경우) | C.F 나쁠 때 (수치가 높은 경우) |
| - 블록 수에 근접  - 랜덤 액세스 효율이 좋다.  - 버퍼 Pinning효과로 인해 Random I/O (Logical Read) 미 발생 | - 로우 수에 근접  - 랜덤 액세스 효율이 매우 나쁘다. |



**버퍼Pinning 효과란?**

- Latch 획득의 부하 (Cache Buffer Latch) = 하나의 블록 조회 시

- Buffer Block 대기 (Buffer Busy Wait)

- LRU 알고리즘에 의해 메모리에서 Age Out 되었을 경우 LRU Latch 획득 필요

\* Buffer Pinning : Logical Read Count로 잡히지 않으며(PGA를 사용하기 때문에) 다음 번 Read시 동일 Block를 Read 할 경우 대상 Block이 Age Out 되지 않도록 Pin을 걸어두고 DB Block Address가 가르키는 메모리 번지수를 PGA에 저장하여 바로 찾아가는 기법.

비용 (Cost) = b level --인덱스 수직적 탐색 비용

+ (리프블록 수 x 유효 인덱스 선택도)  --인덱스 수평적 탐색 비용

+ (클러스터링 팩터 x 유효 테이블 선택도) --테이블 Random 액세스 비용

- b level : 리프블록에 도달하기 전까지 읽게 될 루트 및 브랜치 블록 개수

- 유효 인덱스 선택도 : 전체 인덱스 레코드 중에서 조건절을 만족하는 레코드를 찾기 위해 스캔 활용할 것으로 예상되는 비율(%)

- 유효 테이블 선택도 : 전체 레코드 중에서 인덱스 스캔을 완료하고서 최종적으로 테이블을 방문할 것으로 예상되는 비율(%)

- Table Full Scan과 Index Scan의 손익 분기점

\* 통상적으로 찾고자하는 레코드가 전체 용량 대비 10% But Not always.

-> C.F에 의해 좌우되며 이를 극복하기 위해 SQL Server의 Cluster Index, Oracle의 Clustered Index, IOT 존재

**5. 테이블 Random 액세스 최소화 튜닝**

- 인덱스 칼럼 추가

\* 이 경우 DML 속도가 느려지거나 디스크 사용량이 증가, 최악의 경우 실행계획이 바뀜!

- PK 인덱스에 칼럼 추가

\* 칼럼을 추가할 경우 클러스터링 팩터가 매우 높아짐(나쁨)

- 인덱스만 읽고 처리

\* 그 칼럼들이 자주 쓰인다면 사용할 것 (필요한 모든 변수 인덱스 선언)

**6. IOT와 클러스터**

|  |  |  |
| --- | --- | --- |
| 구분 | Heap Table | IOT |
| 인덱스를 이용한 Table Search | RowID | - |
| PK | Key + RowID | Key |
| Secondary Key | Key + RowID | Key + PK |
| 테이블의 RowID 변화 | 비 휘발성 | 휘발성 |
| 속도 | C.F 따라 상이 | PK 조회 : 매우 높음  Secondary Index : 낮음 |

\* IOT ( Index Organized Table)의 경우 Data 영역의 RowID가 휘발성이므로 Secondary Index는 RowID 대신 PK의 Key 값을 이용.

-> Secondary Index로조회할시성능이좋지않음.

- IOT의 활용

1) 크기가 작고 NL조인으로 반복 Lookup (Driving Table의 반대 급부) 하는 테이블

2) 폭이 좁고 긴(row수가 많은) 테이블

3) 넓은 범위를 주로 검색하는 테이블

4) 데이터 입력과 조회 패턴이 서로 다른 테이블

**8. 인덱스 스캔 효율**

Distinct : Data 종류 개수 -> 높을수록 Good

선택도 : 1/Distinct -> 낮을수록 Good

Cardinality : 선택도 x Row 수 -> 낮을수록 Good

분포도 : Data 종류별 개수 / Row 수 -> 낮을수록 Good

(1) 인덱스 매칭도

결합 인덱스 우선순위 결정

1. 항상(자주) 사용 되는가?

2.＂＝＂조건

3. Cardinality

4. 소트 연산대체

(2) 인덱스 선행컬럼이 등치 조건이 아닐 때 발생하는 비효율

- Sequential 액세스 효율은 선택도에 의해 결정 -> 얼마나 적게 스캔하는지

- 인덱스 컬럼이 조건절에 모두 등치 조건일 때 가장 높다.

- 리프블록을 스캔하면서 읽은 레코드는 모두 필터링되지 않고, 테이블 액세스

- 인덱스 컬럼 중 조건절 생략되거나, 등치조건이 아니라도 뒤쪽 컬럼일 때 비효율은 없다.

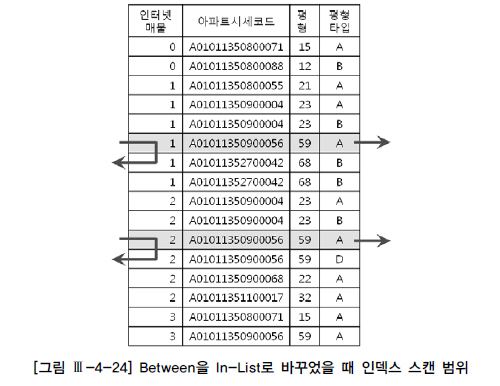


(3) Between 조건을 IN-List로 변경했을 때 인덱스 스캔 효율

- IN에 들어가 있는 수만큼 수직적 탐색이 발생 -> 수가 적을 때는 Between보다 우위

-> IN-List 개수가 많지 않아야하며, 수직적 탐색에 대한 비효율이 더 클 수 있으며 인덱스 높이가 높을 때 비효율이 증가한다.

->> IN-List가 많을 경우 Leaf Block 비효율(Check조건으로)이 더 효율적일 수 있다.



(4) Index Skip Scan을 이용한 비효율 해소

- 인덱스가 Check 조건으로 되어있을 때 어떠한 경우는 IN-List보다 우위를 보임

(5) 범위검색 조건을 남용할 때 발생하는 비효율

- 인덱스가 3개의 컬럼으로 구성되어 있고 중간값이 빠져있을 경우 발생되는 비효율.

- 인덱스의 가운데 컬럼이 빠져있을 경우 Union all과 Bind 변수를 활용해서 분기하여 Select절을 작성할 것.

ex) Select ~ From 가입상품

Where **:지역 is NULL** and 회사 = :회사 and 상품명 like :상품명

**Union all**

Select ~ From 가입상품

Where **:지역 is not NULL** and 회사 = :회사 and 상품명 like :상품명

(6) 같은 컬럼에 두 개의 범위검색 조건 사용 시 주의사항

- 같은 컬럼을 두 번 범위검색을 할 때 스캔 범위가 더 적을 것을 Driving 조건으로 활용해서 할 것 -> 만약 범위가 더 넓은 것을 Driving 할 경우 좌변가공을 통해 인위적으로 변경.

(7) Between과 Like 스캔 범위 비교

- 적어도 between을 사용할 경우 성능적으로 손해볼 것은 없다. -> 스캔 범위 지정의 문제

- 그래도 LIKE의 경우라도 스캔 범위를 정확히 지정해줄 경우 비효율은 없다.

(

8) 선분이력의 인덱스 스캔 효율

- 최근 데이터를 주로 읽을 때 : 인덱스 (종료일자 + 시작일자)

- 과거 데이터를 주로 읽을 때 : 인덱스 (시작일자 + 종료일자)

- 인덱스 수정이 불가하다면 Index\_Desc 힌트 활용

- 중간 지점을 읽을 때는 어떠한 인덱스든 비효율이 발생하지만 rownum <= 1 (수도컬럼)

**9. 인덱스의 종류**

**가. B\*Tree 인덱스**

1) Unbalanced 인덱스

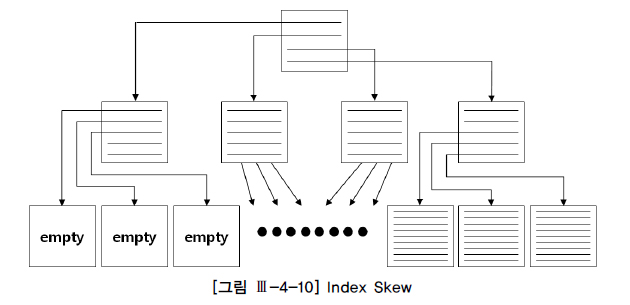
- B-Tree에선 발생하지 않음 (Balanced Tree 이므로)

2) Index Skew 현상

- 대량 삭제 후 발생현상

- 빈 블록은 Free-list로 등록되지만 반환하지는 않음

- 재사용 가능하나 다시 채워질 때 까지 인덱스 스캔 효율 저하



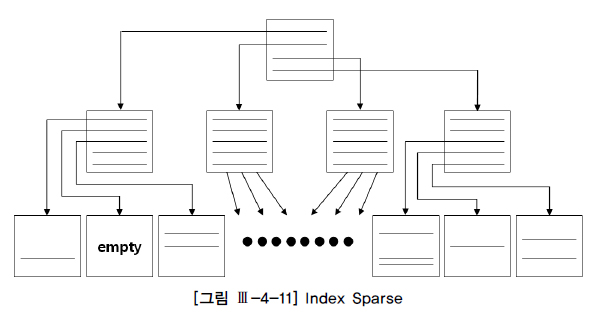
3) Index Sparse

- 대량의 삭제 작업 후 발생

- 인덱스 밀도가 낮은 상황

- Skew 현상과 같이 빈블록은 재사용되지만 동 현상은 Empty Block이 거의 없어 데이터가 채워질 때까지 인덱스 비효율 발생

- 총 레코드 건수가 일정한데도 인덱스 공간 사용량 지속증가는 주로 같은 현상



4) Index 재생성 요건

- 인덱스 분할에 의한 경합이 현저히 높을 때

- 자주 사용되는 인덱스 스캔 효율을 높이고자 할 때. 특히 NL join에서 반복 액세스 되는 인덱스 높이가 증가했을 때

- 대량의 delete 작업을 수행한 이후 다시 레코드가 입력되기까지 오랜 기간이 소요될 때

- 총 레코드 수가 일정한데도 인덱스가 계속 커질 때

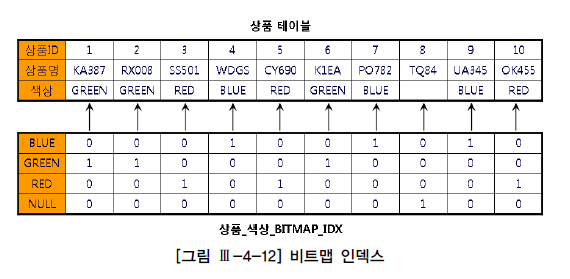
- 위와 같이 인덱스 재생성은 확실한 효과가 예상 될 때만 시행하는 것이 바람직.

**나. 비트맵 인덱스**

- Distinct Value 개수가 적을 때 적은 용량을 차지하므로 인덱스가 여러 개 필요한 대용량 테이블에 유용하며 다양한 분석관점을 가진 팩트성 테이블에 주로 사용

- 테이블 랜덤 액세스 발생으로 B-Tree 인덱스와 동일하여 좋은 성능은 아님

- > 즉, 단독으로 쓰임새는 없으나 여러 비트맵 인덱스를 동시 활용시 대용량 데이터 검색 성능 향상에 효과가 있으나 DML 부하 과다이므로 DW 환경에서 많이 사용



**다. 함수기반 인덱스**

- 데이터 입력, 수정 시 함수를 적용해야 하기에 다소 부하 발생가능

- 사용된 함수가 User-Defined 함수일 경우 부하 극심

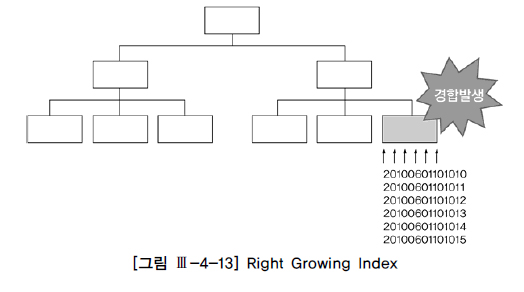
- 남용하지 말고 꼭 필요한 경우에만 사용

**라. 리버스 인덱스**

- 인덱스가 Right Growing 현상일 경우 유용

- 입력된 값을 거꾸로 변환해서 저장하므로 데이터가 고르게 분포

- 등치 검색만 가능하며 그 외엔 사용불가



**마. 클러스터 인덱스**

- 키 값이 같은 레코드가 한 블록에 모이도록 저장하는 구조

- 한 블록에 모두 담을 수 없을 때는 새로운 블록을 할당해 클러스터 체인으로 연결

- 다중테이블 인덱스 클러스터

- 물리적으로 같은 블록에 여러 테이블의 레코드가 저장 가능 -> 조인 상태로

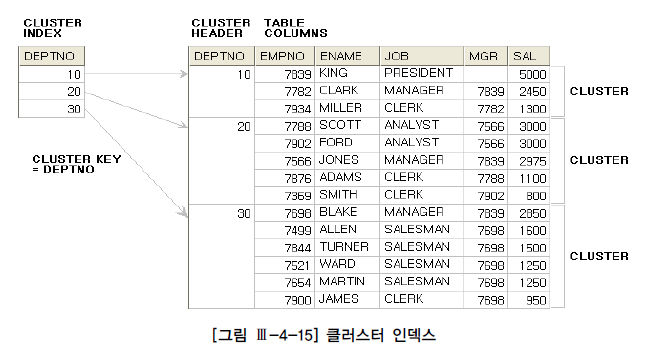
- 일반적은 경우 하나의 데이터 블록이 여러 테이블에 의해 공유될 수 없음.

- 일반적은 B-Tree 구조로 해당 키 값을 저장하는 첫 번째 데이터 블록만 가리킴

- 키 값은 항상 Unique하며 키 값과 테이블 레코드는 1:M 관계 유지

- 넓은 범위 검색에 유리 (클러스터링 팩터가 좋음)

- 새로운 키 값이 자주 입력되거나 수정이 자주 발생할 경우 좋지 않음



**10. 인덱스 설계**

1) 결합 인덱스 구성을 위한 기본 공식

- 조건절에 항상 사용되는가?

- 등치 조건으로 사용되는가?

- Cardinality가 좋은가?

- 소트 오퍼레이션을 생략 가능한가?

2) 추가적 고려사항

- 쿼리 수행빈도, 업무상 중요도, 클러스터링 팩터, 데이터량, DML 부하, 저장공간

- 인덱스 관리 비용