

TIBERO 튜닝 기초교육

(3. 인덱스 스캔 효율화)

YOON



III

인덱스 스캔 효율화

- 3.1 테이블 랜덤 액세스
- 3.2 인덱스 클러스터링 팩터(CF)
- 3.3 손익분기점
- 3.4 인덱스 컬럼 추가
- 3.5 부분범위 처리 활용
- 3.6 인덱스 스캔 효율화
- 3.7 인덱스 선행 컬럼이 '=' 아닐 때 생기는 비효율
- 3.8 Between을 IN-List로 전환
- 3.9 Index Skip Scan 활용
- 3.10 IN 조건은 항상 '='인가?
- 3.11 Between vs. Like
- 3.12 범위검색 조건을 남용할 때의 비효율
- 3.13 인덱스와 NULL
- 3.14 다양한 옵션 조건 처리 방식
- 3.15 함수호출부하 해소를 위한 인덱스 구성
- 3.16 인덱스 설계

3.1 테이블 랜덤 액세스

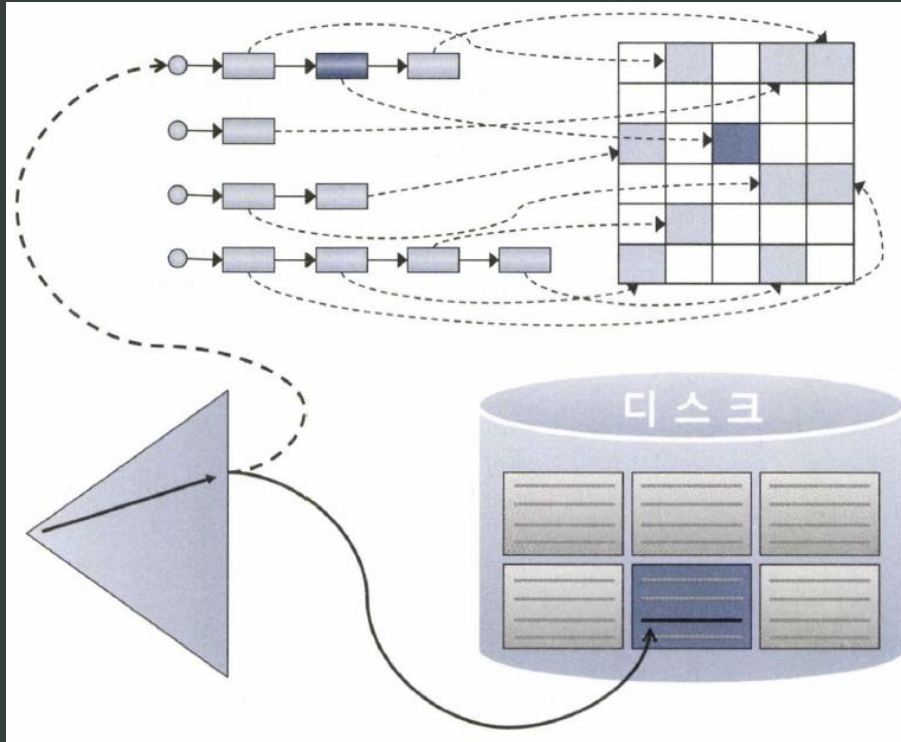
인덱스 ROWID

- 인덱스를 스캔하는 이유는 조건을 만족하는 데이터를 인덱스에서 빨리 찾고 테이블 레코드를 찾아가기 위한 주소값(ROWID)를 얻는데 있음
- 물리적 주소 vs. 논리적 주소
- (프로그래밍의) 포인터가 아니며, 디스크 상에서 테이블 레코드를 찾아가기 위한 위치 정보를 담는 논리적 주소임
- 프로그래밍 언어에서 포인터(Pointer)는 메모리 주소값을 담는 변수

- ROWID 구조 - 18 Byte(6.3.6.3)
AAAAxAAACAAAE2VAAA
AAAAxA · AAC · AA AE2V · AAA
6 : Segment 식별을 위한 Data Object Number
3 : Tablespace의 상대적인 Data File Number
6 : Row를 포함하는 Data Block Number
3 : Block에서의 Row의 Slot
- 행 이전과 체인화로 인해, 그리고 익스포트 및 임포트 후 행의 위치가 이동될 수 있기 때문에 데이터에 액세스하는 방법으로 권장되지 않음

3.1 테이블 랜덤 액세스 (계속)

인덱스를 이용한 데이터 탐색 과정(고비용 구조)



인덱스를 이용해 테이블 블록을 찾아가는 과정

- ① I/O 성능을 높이려면 버퍼캐시를 활용해야 하기에 블록을 읽을 때 디스크로 가기 전 버퍼캐시 탐색
- ② 리프 블록에서 읽은 ROWID를 분해해서 DBA 정보를 얻음($\text{DBA} = \text{DataFile No} + \text{Block No}$)
- ③ 읽고자 하는 DBA를 해시 함수에 입력해서 해시 체인을 찾고 거기서 버퍼 헤더를 찾고, 거기서 얻은 포인터로 버퍼 블록을 찾아감
- ④ ROWID가 가르키는 테이블 블록을 버퍼캐시에서 먼저 찾아보고(점선), 없을 경우 디스크 블록에서 읽음 (물론, 버퍼캐시에 적재한 후에 읽음)

3.1 테이블 랜덤 액세스 (계속)

DBMS_ROWID

```
CREATE OR REPLACE PACKAGE SYS.DBMS_ROWID AUTHID CURRENT_USER
IS ROWID_INVALID EXCEPTION;
PRAGMA EXCEPTION_INIT(ROWID_INVALID, -15130);

PROCEDURE ROWID_INFO (
    ROWID_IN      IN  ROWID
    ,SEGMENT_NUMBER OUT NUMBER
    ,ABSOLUTE_FNO  OUT NUMBER
    ,BLOCK_NUMBER  OUT NUMBER
    ,ROW_NUMBER    OUT NUMBER)
PRAGMA BUILTIN('pkg_dbms_rowid__rowid_info');

FUNCTION ROWID_CREATE(
    SEGMENT_NUMBER IN NUMBER
    ,ABSOLUTE_FNO  IN NUMBER
    ,BLOCK_NUMBER  IN NUMBER
    ,ROW_NUMBER    IN NUMBER)
RETURN ROWID PRAGMA BUILTIN('pkg_dbms_rowid__rowid_create');

FUNCTION ROWID_SEGMENT      (ROW_ID IN ROWID) RETURN NUMBER PRAGMA BUILTIN('pkg_dbms_rowid__rowid_segment');
FUNCTION ROWID_BLOCK_NUMBER (ROW_ID IN ROWID) RETURN NUMBER PRAGMA BUILTIN('pkg_dbms_rowid__rowid_block_number');
FUNCTION ROWID_ROW_NUMBER   (ROW_ID IN ROWID) RETURN NUMBER PRAGMA BUILTIN('pkg_dbms_rowid__rowid_row_number');
FUNCTION ROWID_ABSOLUTE_FNO (ROW_ID IN ROWID) RETURN NUMBER PRAGMA BUILTIN('pkg_dbms_rowid__rowid_absolute_fno');
FUNCTION ROWID_TO_RELATIVE_FNO(ROW_ID IN ROWID) RETURN NUMBER PRAGMA BUILTIN('pkg_dbms_rowid__rowid_to_relative_fno');

FUNCTION ROWID_CREATE_WITH_RELATIVE_FNO(
    SEGMENT_NUMBER IN NUMBER
    ,RELATIVE_FNO   IN NUMBER
    ,BLOCK_NUMBER   IN NUMBER
    ,ROW_NUMBER     IN NUMBER
) RETURN ROWID PRAGMA BUILTIN('pkg_dbms_rowid__rowid_create_with_relative_fno');
END;
```

3.1 테이블 랜덤 액세스 (계속)

DBMS_ROWID (활용)

```
select empno
      , dbms_rowid.rowid_segment(rowid) object
      , dbms_rowid.rowid_to_relative_fno(rowid) file_no
      , dbms_rowid.rowid_block_number(rowid) block_no
      , dbms_rowid.rowid_row_number(rowid) row_number
from emp_hash e;
```

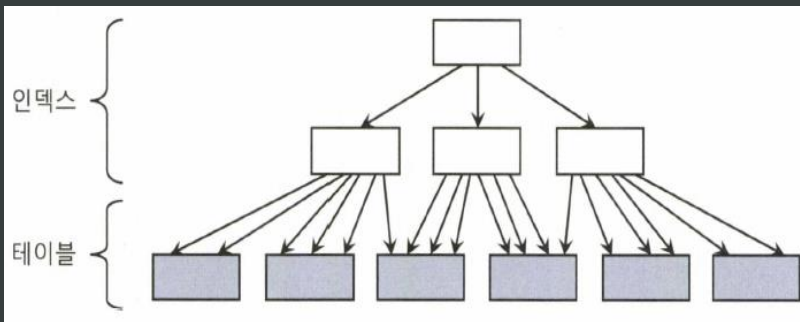
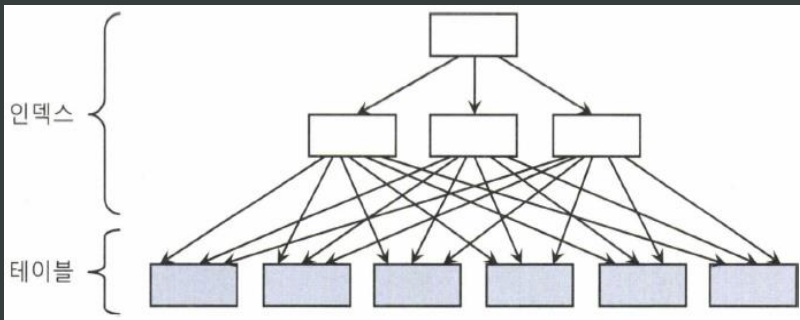
	EMPNO	OBJECT	FILE_NO	BLOCK_NO	ROW_NUMBER
1	246	3868	2	95866	0
2	246	3868	2	95866	1
3	256	3868	2	95866	2
4	256	3868	2	95866	3
5	256	3868	2	95866	4
6	256	3868	2	95866	5
7	256	3868	2	95866	6
8	256	3868	2	95866	7
9	256	3868	2	95866	8
10	256	3868	2	95866	9
11	256	3868	2	95866	10
12	256	3868	2	95866	11
13	256	3868	2	95866	12
14	256	3868	2	95866	13
15	256	3868	2	95866	14
16	256	3868	2	95866	15

- 오라클과 티베로의 경우 일부 함수 차이가 있음
- 테이블에 대한 Hot 블록이 발생하는 경우 블록 경합을 회피하기 위해 데이터 레코드의 상세 정보가 추가적으로 필요할 경우 활용할 수 있음
- 대표적으로 해시 파티션 적용 이후 비교

3.2 인덱스 클러스터링 팩터(CF)

정의

- 특정 컬럼을 기준으로 같은 값을 갖는 데이터가 모여 있는 정도를 의미하는 군집성 계수
- 테이블 랜덤 액세스를 감소 효과



효과

- CF가 좋은 컬럼 인덱스의 검색 효율은 테이블 액세스 수에 비해 블록 I/O가 적게 발생함을 의미
- 버퍼 Pinning 효과
(래치 획득과 해시 체인 스캔 과정을 생략하고 테이블을 읽을 수 있기에 논리적 블록 I/O 과정 생략)

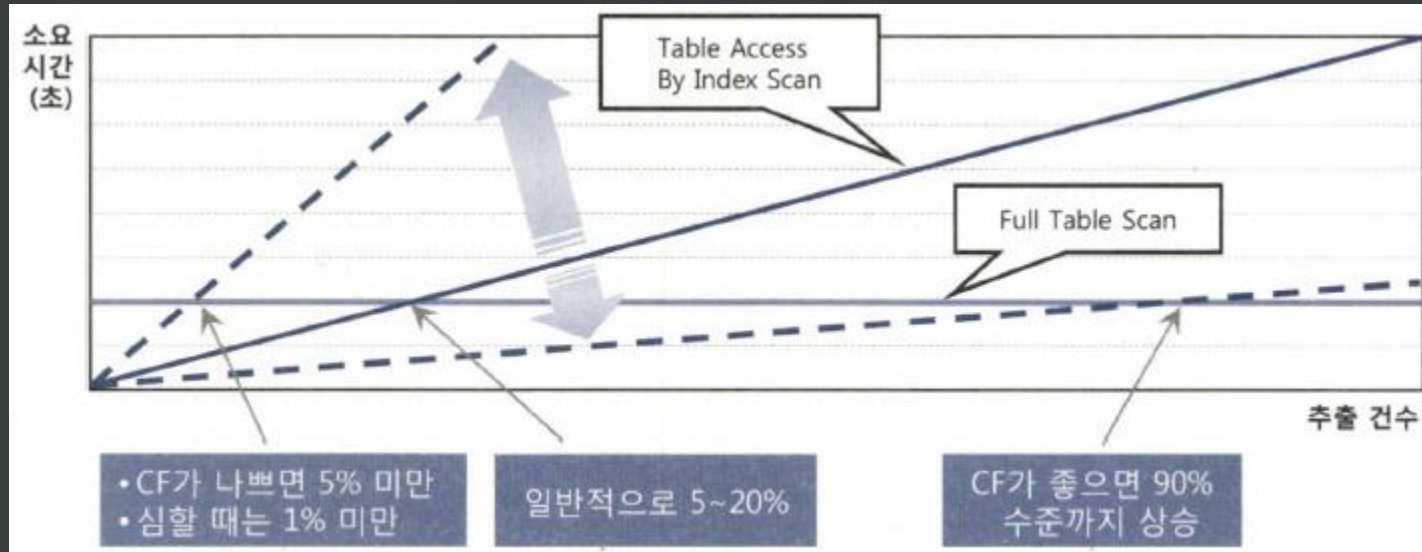
비교

NAME	VALUE
db block gets	304
consistent gets	221
...(중간 생략)...	
rows processed	112

NAME	VALUE
db block gets	148
consistent gets	113
...(중간 생략)...	
rows processed	112

3.3 손익분기점

인덱스 손익분기점과 버퍼캐시 히트율



- 일반적인 5~20%의 손익분기점은 데이터 건수가 작은 100만 건 이내 테이블에 적용되는 수치임
- 1,000만 건의 10%면 100만 건이며 인덱스로 추출할 경우 OLTP 운영 시스템에 영향을 줄 수 있음
- 또한, 이런 경우라면 캐시 히트율을 극히 낮출 수 밖에 없음

3.3 손익분기점 (계속)

온라인 프로그램 튜닝 vs. 배치 프로그램 튜닝

- 모든 성능 문제를 인덱스로 해결하려 해선 안 됨(인덱스는 다양한 튜닝 도구 중 하나일 뿐)
- 온라인 프로그램(Online, OLPT)은 보통 데이터를 읽고 갱신하므로 인덱스를 효과적으로 활용하는 것이 중요
- 반면, 대량 데이터를 읽고 갱신하는 배치 프로그램(Batch, OLAP)은 넓은 범위 처리 기준으로 튜닝을 해야 하기에 일부가 아닌 전체를 빠르게 처리하는 것을 목표로 삼아야 함(대량 데이터를 빠르게 처리하려면, 인덱스와 NL 조인보다 Full Scan과 해시 조인이 유리함)
- 초대용량 테이블을 Full Scan 하면 이 또한 시스템에 주는 부담이 적지 않음.
(파티션 전략 및 병렬처리, 집계 테이블 등이 고려되어야 함)
- 결론적으로는 상황에 맞는 선택을 해야함

3.4 인덱스 컬럼 추가

변경 전

DEPTNO	JOB
10	CLERK
10	MANAGER
10	PRESIDENT
20	ANALYST
20	ANALYST
20	CLERK
20	CLERK
20	MANAGER
30	CLERK
30	MANAGER
30	SALESMAN
30	SALESMAN
30	SALESMAN
30	SALESMAN
30	SALESMAN

ENAME	DEPTNO	JOB	SAL
ADAMS	20	CLERK	1100
KING	10	PRESIDENT	5000
BLAKE	30	MANAGER	2850
JONES	20	MANAGER	2975
SMITH	20	CLERK	800
CLARK	10	MANAGER	2450
ALLEN	30	SALESMAN	1600
MILLER	10	CLERK	1300
MARTIN	30	SALESMAN	1250
SCOTT	20	ANALYST	3000
JAMES	30	CLERK	950
FORD	20	ANALYST	3000
WARD	30	SALESMAN	1250
TURNER	30	SALESMAN	1500

EMP_X01 : deptno, job

```
select * from emp where deptno = 30 and sal >= 2000
```

변경 후

DEPTNO	JOB	SAL
10	CLERK	1300
10	MANAGER	2450
10	PRESIDENT	5000
20	ANALYST	3000
20	ANALYST	3000
20	CLERK	800
20	CLERK	1100
20	MANAGER	2975
30	CLERK	950
30	MANAGER	2850
30	SALESMAN	1250
30	SALESMAN	1250
30	SALESMAN	1250
30	SALESMAN	1500
30	SALESMAN	1600

ENAME	DEPTNO	JOB	SAL
ADAMS	20	CLERK	1100
KING	10	PRESIDENT	5000
BLAKE	30	MANAGER	2850
JONES	20	MANAGER	2975
SMITH	20	CLERK	800
CLARK	10	MANAGER	2450
ALLEN	30	SALESMAN	1600
MILLER	10	CLERK	1300
MARTIN	30	SALESMAN	1250
SCOTT	20	ANALYST	3000
JAMES	30	CLERK	950
FORD	20	ANALYST	3000
WARD	30	SALESMAN	1250
TURNER	30	SALESMAN	1500

EMP_X01 : deptno, job, sal

```
select * from emp where deptno = 30 and job = 'CLERK'  
select * from emp where deptno = 30 and sal >= 2000
```

차이점은?

3.5 부분범위 처리 활용

개념

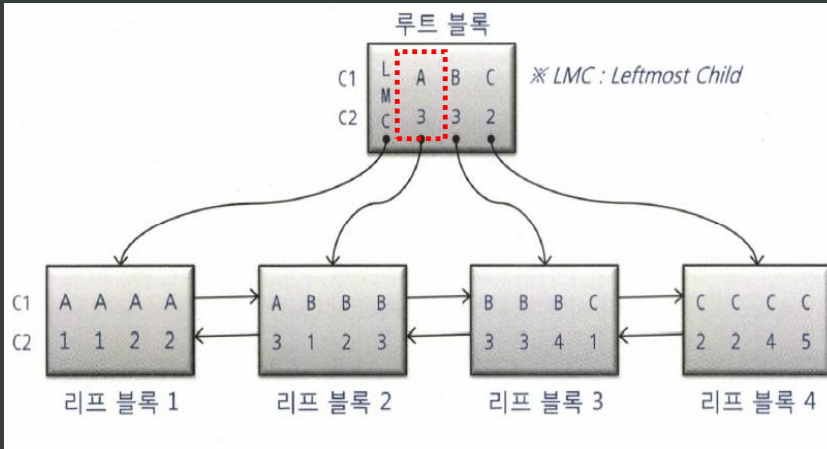
- 쿼리의 결과집합 전체를 쉼 없이 연속적으로 전송하지 않고 사용자로부터 Fetch Call이 있을 때마다 일정량씩 나누어 전송하는 것을 의미함
- 전송 단위인 Array Size는 클라이언트 프로그램에서 설정(Java : 10, SQL*Plus : 15, tSQL : ?)
- 1억 건짜리 테이블도 결과를 빨리 출력할 수 있는 이유는, DBMS가 데이터를 모두 읽어 한 번에 전송하지 않고 먼저 읽는 데이터부터 일정량을 전송하고 멈추기 때문(전송 후 서버 프로세스는 CPU를 OS에 반환하고 대기 큐에서 Sleep)

활용

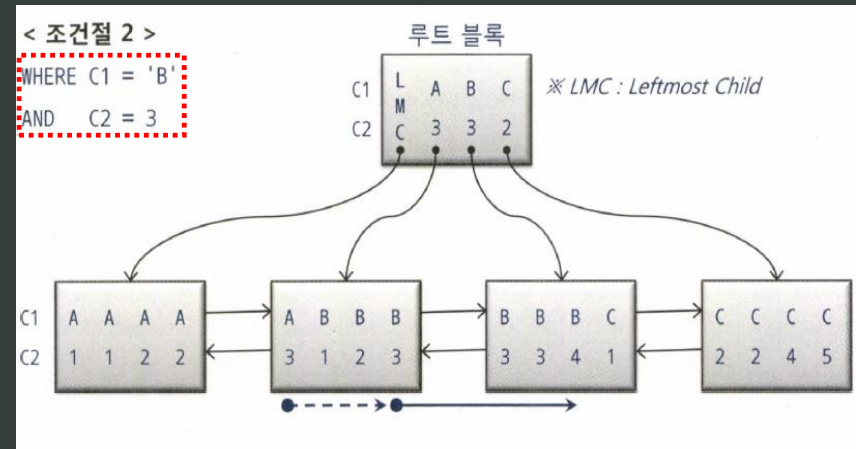
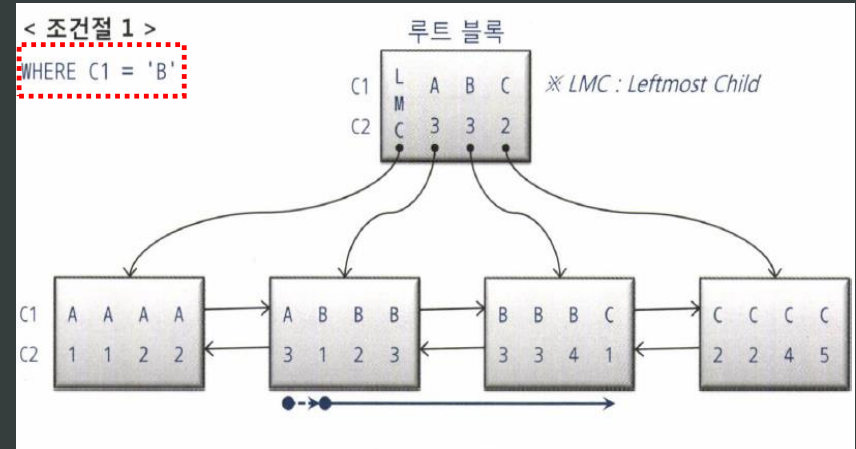
- 테이블 랜덤 액세스로 인한 인덱스 손익분기점의 한계를 극복하기 위한 기능으로 인덱스로 액세스할 대상 레코드가 아무리 많아도 빠른 응답속도를 낼 수 있음
- 정렬 조건이 있는 경우 일반적으로는 모든 데이터를 다 읽고 나서 클라이언트에게 전송을 시작할 수 있지만 해당 컬럼이 선두인 인덱스가 있다면 부분범위 처리가 가능
- Array Size 조정을 통한 Fetch Call 횟수를 줄일 수 있음(전송해야 하는 총량은 불변)

3.6 인덱스 스캔 효율화

1. 인덱스 탐색 및 스캔 효율성



- 루트 블록에는 키 값을 갖지 않는 특별한 레코드가 하나 있음 - 가장 왼쪽의 LMC(Leftmost Child)
- LMC는 자식 노드 중 가장 왼쪽 끝에 위치한 블록을 가리킴 (리프 블록 1)
- LMC가 가리키는 주소로 찾아간 블록에는 첫번째 레코드 (위의 C1='A', C2=3인 레코드)보다 작거나 같은(<=) 값을 갖는 레코드가 저장돼 있음

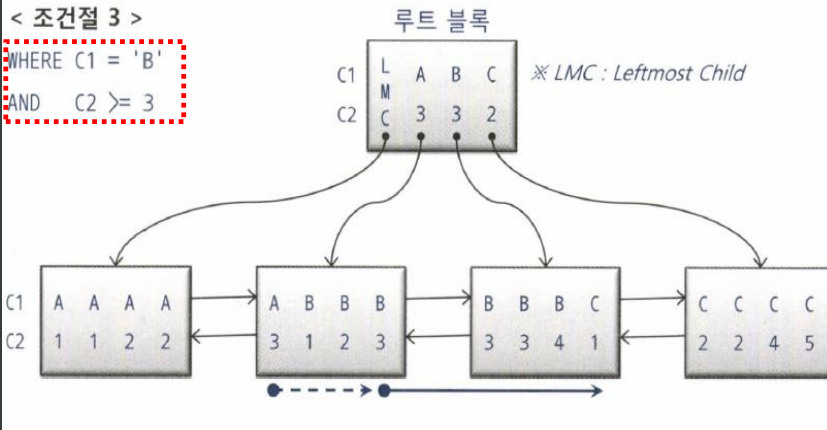


3.6 인덱스 스캔 효율화

1. 인덱스 탐색 및 스캔 효율성 (계속)

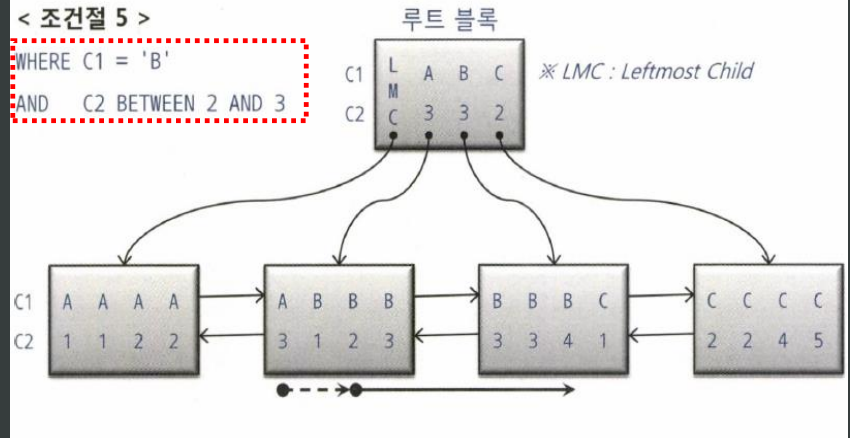
< 조건절 3 >

WHERE C1 = 'B'
AND C2 >= 3



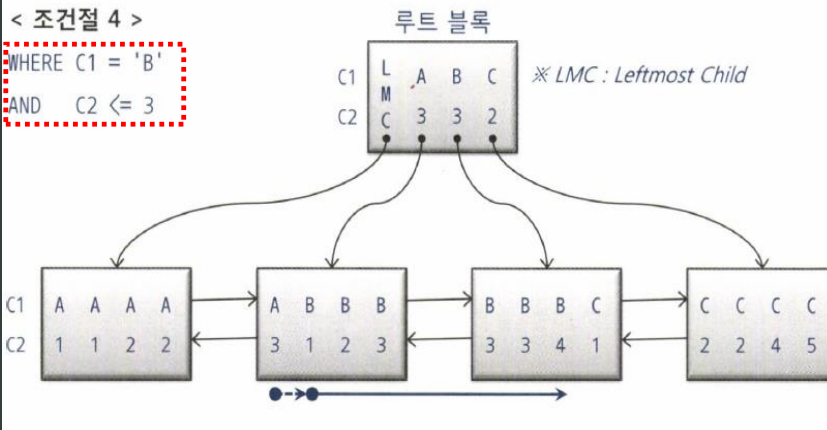
< 조건절 5 >

WHERE C1 = 'B'
AND C2 BETWEEN 2 AND 3



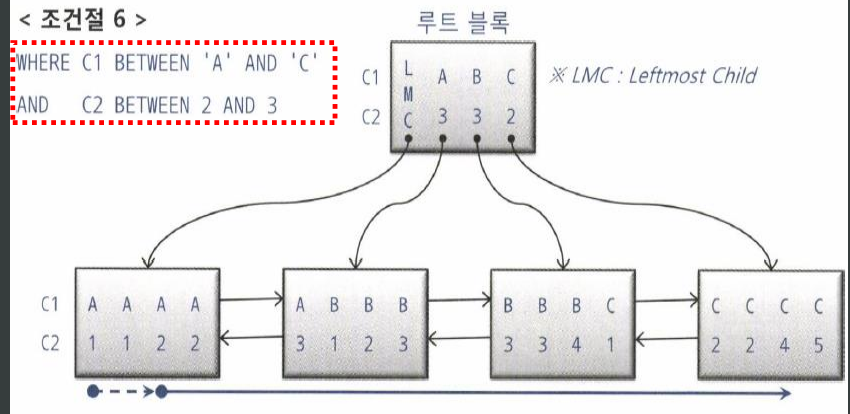
< 조건절 4 >

WHERE C1 = 'B'
AND C2 <= 3



< 조건절 6 >

WHERE C1 BETWEEN 'A' AND 'C'
AND C2 BETWEEN 2 AND 3



3.6 인덱스 스캔 효율화

1. 인덱스 탐색 및 스캔 효율성 (계속)

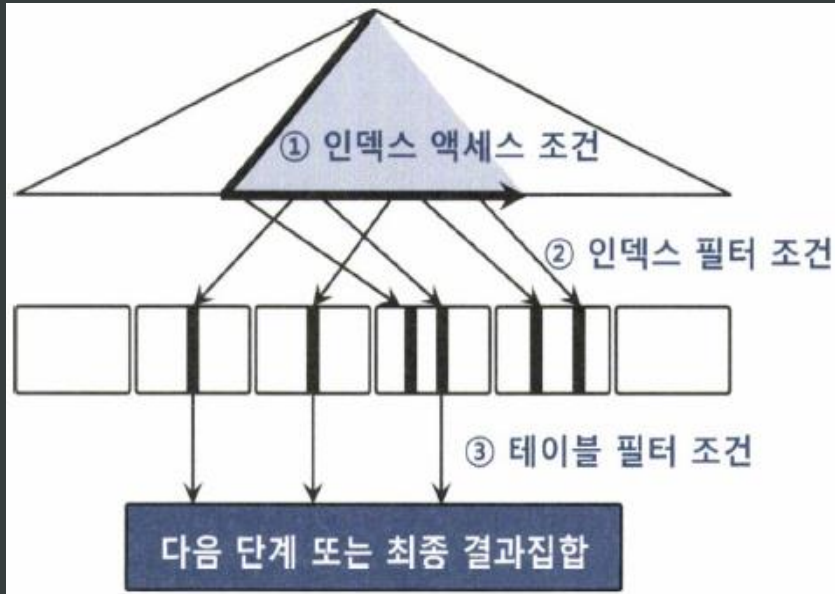
- 교재가 의미하는 바는?
- 인덱스 선두 컬럼의 사용 형태 : Equal, Range(Between, Like, <=)
- 인덱스 선행 컬럼의 조건절 생략에 따른 비효율
- XPLAN – 인덱스 효율성 측정

ID	Operation	Name	Rows	Elaps. Time	Pstart	Pend	CR Gets	Starts
1	PARTITION HASH (ALL PART)		18	00:00:00.0000	1	4	0	1
2	TABLE ACCESS (ROWID) LOCAL	TB\$OBJECTS_HASH	18	00:00:00.0064			142	1
3	INDEX (RANGE SCAN)	TB\$OBJECTS_HASH_IX2	68	00:00:00.0170			140	1

- 인덱스를 스캔하고 얻은 레코드가 68개이고 140개의 블록을 읽음
- 인덱스 리프 블록에는 테이블 블록보다 훨씬 더 많은 레코드가 담기며, 한 블록당 평균 500개의 레코드라고 가정하면 위의 경우는 70,000(500 * 140)개의 레코드를 읽고 68개을 얻은 경우임

3.6 인덱스 스캔 효율화

2. 액세스 조건과 필터 조건



C1	C2	C3	C4
성	능	감	시
성	능	개	량
성	능	개	선
성	능	검	사
성	능	검	증
성	능	계	수
성	능	계	측
성	능	곡	선
성	능	관	리
성	능	시	험
성	능	이	론
성	능	지	수
성	능	측	정
성	능	튜	닝
성	능	평	가
성	능	호	환

Arrows indicate the flow of data from the left table to the right table. The right table is a filtered version of the left table, showing only rows that satisfy the conditions. The conditions are: C1 = '성' and C2 = '능' and C3 = '검' (left side) and C1 = '성' and C2 = '능' and C4 = '선' (right side). The rows that satisfy both conditions are highlighted in dark blue in the right table.

- 인덱스 액세스 조건(①)은 인덱스 스캔 범위를 결정하는 조건절이고 필터 조건(②)은 테이블로 액세스할지를 결정하는 조건절
- 테이블 액세스 단계에서 처리되는 조건절은 모두 필터(③)이며, 쿼리 수행 다음 단계로 전달하거나 포함할지 결정

- 인덱스 구성 : c1, c2, c3, c4
- 좌측 : c1, c2, c3 모두 인덱스 액세스 (c1 = '성' and c2 = '능' and c3 = '검')
- 우측 : c1, c2 인덱스 액세스, c4는 인덱스 필터 (c1 = '성' and c2 = '능' and c4 = '선')

3.6 인덱스 스캔 효율화

3. 비교 연산자와 컬럼 순서

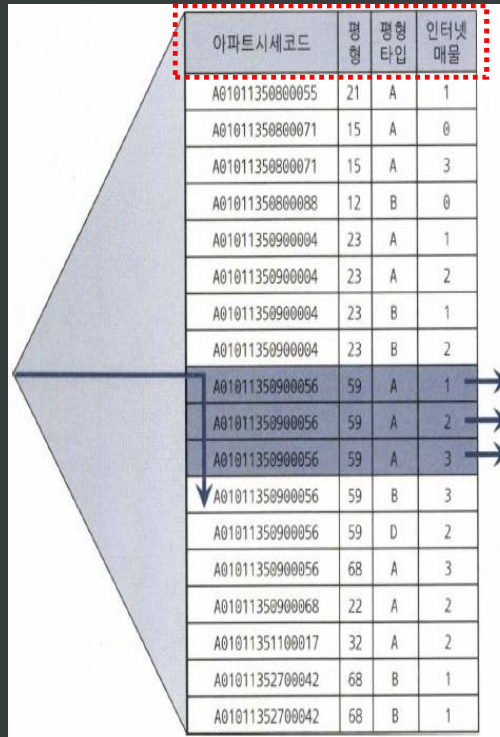
구분	조건절	Index Access	Index Filter
1	Where c1 = 1 and c2 = 'A' and c3 = '나' and c4 = 'a'	C1, C2, C3, C4	
2	Where c1 = 1 and c2 = 'A' and c3 = '나' and c4 >= 'a'	C1, C2, C3, C4	
3	Where c1 = 1 and c2 = 'A' and c3 between '가' and '다' and c4 = 'a'	C1, C2, C3	C4
4	Where c1 = 1 and c2 <= 'B' and c3 = '나' and c4 between 'a' and 'b'	C1, C2	C3, C4
5	Where c1 between 1 and 3 and c2 = 'A' and c3 = '나' and c4 = 'a'	C1	C2, C3, C4

3.7 인덱스 선행 컬럼이 '=' 아닐 때 생기는 비효율

스캔 범위 비교

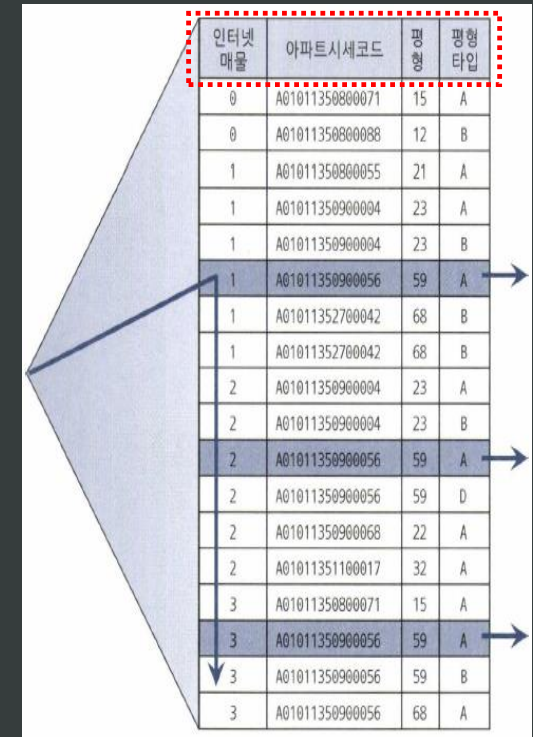
• 쿼리

```
select 해당층
      , 평당가
      , 입력일
      , 해당동
      , 매물구분
      , 연사용일수
      , 중개업소코드
from 매물아파트매매
where 아파트시세코드 = 'A01011350900056'
      and 평형 = '59'
      and 평형타입 = 'A'
      and 인터넷매물 between '1' and '3'
order by 입력일 desc
```



아파트시세코드	평형	평형 타입	인터넷 매물
A01011350800055	21	A	1
A01011350800071	15	A	0
A01011350800071	15	A	3
A01011350800088	12	B	0
A01011350900004	23	A	1
A01011350900004	23	A	2
A01011350900004	23	B	1
A01011350900004	23	B	2
A01011350900056	59	A	1
A01011350900056	59	A	2
A01011350900056	59	A	3
A01011350900056	59	B	3
A01011350900056	59	D	2
A01011350900056	68	A	3
A01011350900068	22	A	2
A01011351100017	32	A	2
A01011352700042	68	B	1
A01011352700042	68	B	1

- 인덱스 선행 컬럼이 모두 '=' 조건일 때 필요한 범위만 스캔하고 멈춤



인터넷 매물	아파트시세코드	평형	평형 타입
0	A01011350800071	15	A
0	A01011350800088	12	B
1	A01011350800055	21	A
1	A01011350900004	23	A
1	A01011350900004	23	B
1	A01011350900056	59	A
1	A01011352700042	68	B
1	A01011352700042	68	B
2	A01011350900004	23	A
2	A01011350900004	23	B
2	A01011350900056	59	A
2	A01011350900056	59	D
2	A01011350900068	22	A
2	A01011351100017	32	A
3	A01011350800071	15	A
3	A01011350900056	59	A
3	A01011350900056	59	B
3	A01011350900056	68	A

- 인덱스 선두 컬럼에 Between을 사용하면 조건을 만족하지 않는 레코드까지 스캔하고 버리는 비효율 발생

3.8 Between을 IN-List로 전환

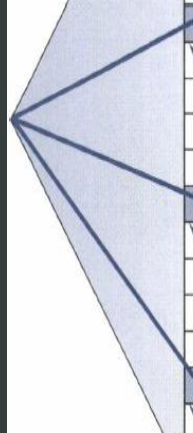
1. 조건절 쿼리 변경

- Between

```
select 해당층  
      , 평당가  
      , 입력일  
      , 해당동  
      , 매울구분  
      , 연사용일수  
      , 중개업소코드  
from 매물아파트매매  
where 아파트시세코드 = 'A01011350900056'  
      and 평형 = '59'  
      and 평형타입 = 'A'  
      and 인터넷매물 between '1' and '3'  
order by 입력일 desc
```

- In-List

```
select 해당층  
      , 평당가  
      , 입력일  
      , 해당동  
      , 매울구분  
      , 연사용일수  
      , 중개업소코드  
from 매물아파트매매  
where 아파트시세코드 = 'A01011350900056'  
      and 평형 = '59'  
      and 평형타입 = 'A'  
      and 인터넷매물 in ('1', '2', '3')  
order by 입력일 desc
```

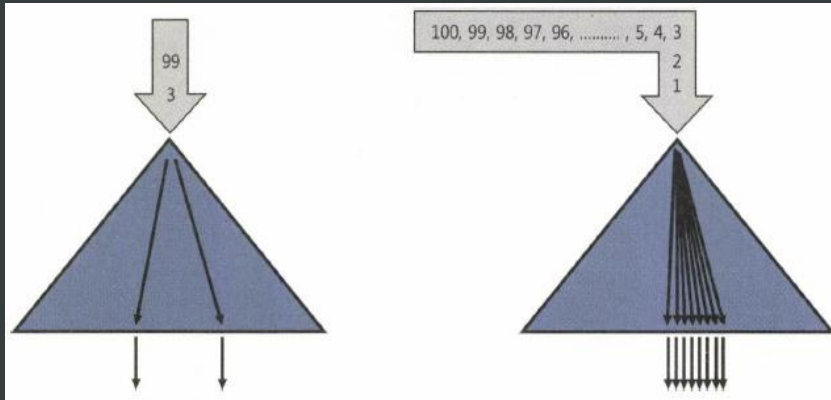


인터넷매물	아파트시세코드	평형	평형타입
0	A01011350800071	15	A
0	A01011350800088	12	B
1	A01011350800055	21	A
1	A01011350900004	23	A
1	A01011350900004	23	B
1	A01011350900056	59	A
1	A01011352700042	68	B
1	A01011352700042	68	B
2	A01011350900004	23	A
2	A01011350900004	23	B
2	A01011350900056	59	A
2	A01011350900056	59	D
2	A01011350900068	22	A
2	A01011351100017	32	A
3	A01011350800071	15	A
3	A01011350900056	59	A
3	A01011350900056	59	B
3	A01011350900056	68	A

- IN-List 개수만큼 Union All 계획 생성
- 각 브랜치마다 모든 컬럼이 '=' 조건으로 검색되어 Between의 비효율이 사라짐

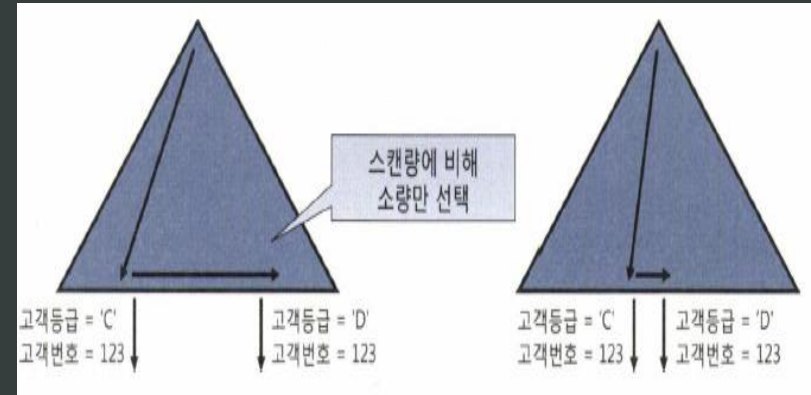
3.8 Between을 IN-List로 전환

2. In-List 조건의 개수



- IN-List 개수가 많으면 우측과 같이 수직 탐색이 많이 발생함
- Between 조건의 리프 블록 탐색 비용 대비 IN-List 개수만큼 탐색하는 비효율이 더 클 수 있음

3. 인덱스 스캔 과정의 만족 레코드 수



- IN-List의 효과를 보려면 조건을 만족하는 레코드가 서로 멀리 떨어져 있어야 함
- 가까이 있을 경우 그 효과가 미비하거나 수직적 탐색 때문에 오히려 블록 I/O가 더 많이 발생할 수 있음

3.9 Index Skip Scan 활용

선두 컬럼의 비효율 제거

- Between 조건을 IN-List 조건으로 변환하면 도움이 되는 상황에서 조건절을 바꾸지 않고 같은 효과를 내는 다른 방법으로 Index Skip Scan을 활용할 수 있음
- 인덱스 선두 컬럼이 '='가 아니면서 나머지 검색 조건을 만족하는 데이터들이 멀리 떨어져 있는 경우 효과
- (결론) - 별도 예제 참조, 실습

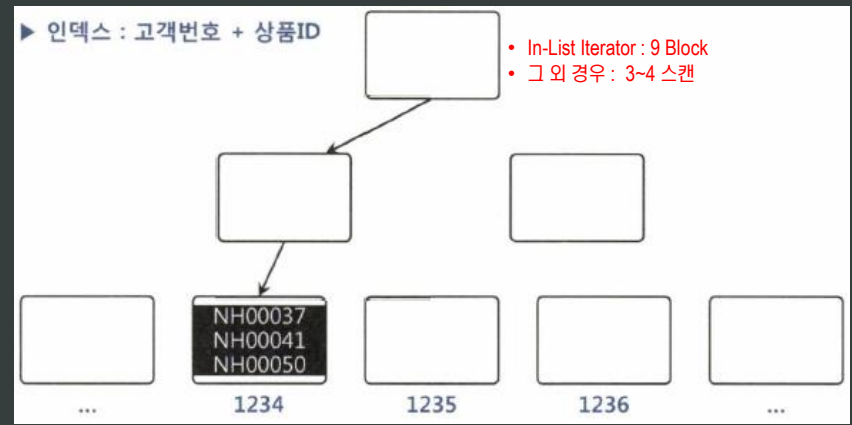
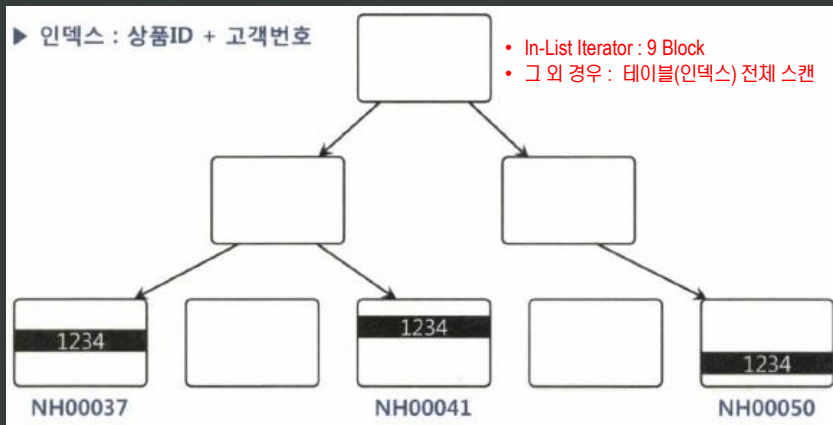
		Elapsed Time	Logical Reads	비고
Table Full Scan		0.0998	3,812	
Index Range(=, =)		0.0073	353	Between > IN-List 변환
Index(판매구분, 판매월)	=, >=	0.0080	287	힌트 제어(단, 선행 컬럼 Like 처리)
	Skip	0.1544	3,392	
Index(판매월, 판매구분)	>=, =	0.0812	3,128	힌트 제어
	Skip	0.0128	341	

3.10 IN 조건은 항상 '='인가?

IN은 항상 '='가 아님

- IN 조건이 '='이 되려면 IN-List Iterator 방식으로 풀려야만 하며, 아닌 경우라면 IN 조건은 필터 조건임
- 인덱스 구성에 따라 성능도 달리질 수 있음
(액세스 조건으로 만들기 위해 IN-List Iterator 방식으로 푸는 것이 항상 효과적이지는 않음)

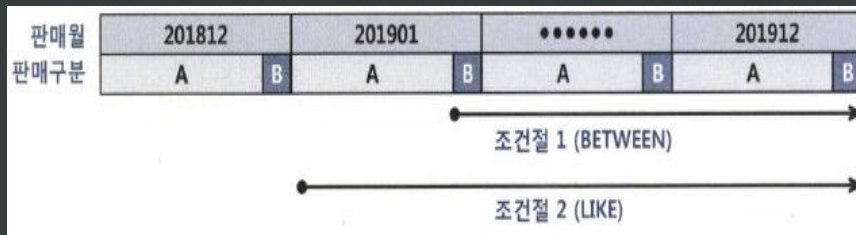
```
select *  
  from 고객별가입상품  
 where 고객번호 = :cust_no  
    and 상품ID in ('NH00037', 'NH00041', 'NH00050')
```



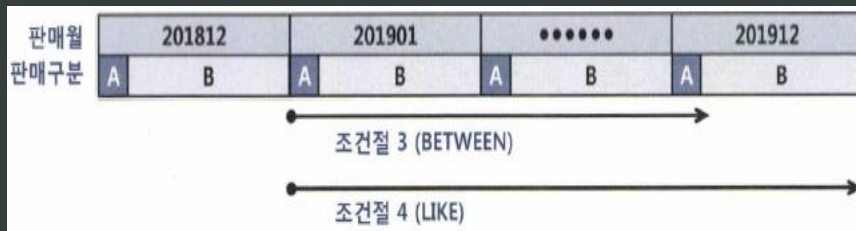
3.11 Between vs. Like

Between

where 판매월 between '201901' and '201912'
and 판매구분 = 'B'



where 판매월 between '201901' and '201912'
and 판매구분 = 'A'



Like

where 판매월 like '2019%'
and 판매구분 = 'B'

- Like는 혹시라도 '201900'이 저장돼 있다면 그 값도 읽어야 하므로 판매구분 = 'B'로 바로 내려갈 수 없음

where 판매월 like '2019%'
and 판매구분 = 'A'

- Between은 '201912', 'B'를 만나는 순간 스캔을 멈춤
- Like는 혹시라도 '201913'이 저장돼 있다면 그 값도 읽어야 하므로 중간에 멈출 수가 없음

• 결론적으로, 조건절을 범위 검색으로 처리하는 경우 가능한 Like 대신 Between으로 처리하는 것이 유리함

3.12 범위검색 조건을 남용할 때의 비효율

1. 정상적인 개발 패턴

인덱스 구성 : 회사코드, 지역코드, 상품명

<쿼리 1> 회사코드, 지역코드, 상품명을 모두 입력

```
select 고객ID
  , 상품명
  , 지역코드
  , ...
from 가입상품
where 회사코드 = :com
      and 지역코드 = :reg
      and 상품명 like :prod || '%'
```

<쿼리 2> 회사코드, 상품명만 입력

```
select 고객ID
  , 상품명
  , 지역코드
  , ...
from 가입상품
where 회사코드 = :com
      and 상품명 like :prod || '%'
```

회사코드	지역코드	상품명
C60	03	일반형
C70	02	고급형
C70	02	보급형1
C70	02	보급형2
C70	02	일반형
C70	02	일반형
C70	02	일반형
C70	02	일반형
C70	02	일반형
C70	03	고급형
C70	03	보급형1
C80	02	고급형

회사코드	지역코드	상품명
C60	03	일반형
C70	02	고급형
C70	02	보급형1
C70	02	보급형2
C70	02	일반형
C70	02	일반형
C70	02	일반형
C70	02	일반형
C70	02	일반형
C70	03	고급형
C70	03	보급형1
C80	02	고급형

- 인덱스 중간 컬럼(지역코드)이 없을 때는 어쩔 수 없는 넓은 범위를 스캔하지만, 조건이 있는 경우는 아주 적은 범위만 스캔하고 빠르게 결과를 출력할 수 있음

3.12 범위검색 조건을 남용할 때의 비효율

2. 과도한 Like 사용 개발 패턴

인덱스 구성 : 회사코드, 지역코드, 상품명

<통합 쿼리>

```
select 고객ID  
      , 상품명  
      , 지역코드  
      , ...  
from 가입상품  
where 회사코드 = :com  
      and 지역코드 like :reg || '%'  
      and 상품명 like :prod || '%'
```

회사코드	지역코드	상품명
C60	03	일반형
C70	02	고급형
C70	02	보급형1
C70	02	보급형2
C70	02	일반형
C70	02	일반형
C70	02	일반형
C70	02	일반형
C70	02	일반형
C70	02	일반형
C70	03	고급형
C70	03	보급형1
C80	02	고급형

회사코드	지역코드	상품명
C60	03	일반형
C70	02	고급형
C70	02	보급형1
C70	02	보급형2
C70	02	일반형
C70	02	일반형
C70	02	일반형
C70	02	일반형
C70	02	일반형
C70	02	일반형
C70	03	고급형
C70	03	보급형1
C80	02	고급형

- 지역코드를 입력 안 한 경우는 기존과 같지만, 지역코드를 입력한 경우는 기존보다 인덱스 스캔 범위가 늘어났음
- 이유는 액세스 조건이던 상품명에 필터 조건으로 바뀜

3.13 인덱스와 NULL

특별한 조건(NULL)

- 단일 컬럼으로 구성된 인덱스는 NULL을 저장하지 않음
(DBMS 마다 다름. PostgreSQL은 단일 컬럼으로 구성된 인덱스에도 NULL 저장)
- 결합 컬럼으로 구성된 인덱스는 NULL을 저장함
(dummy 값도 추가 가능. create index tab1_x01 on tab1(col1, 1)로도 구성 가능 - 문제점은?)

인덱스 구성	사용 형태	Operation	Predicate
MGR, JOB	mgr is null mgr is null and job = 'CLERK'	Table Full Scan Index Range Scan	Filter Access
JOB, MGR	job = 'CLERK' job = 'CLERK' and mgr is null	Index Range Scan	Access
MGR, 1	mgr is null	Index Range Scan	Access
NVL(MGR, 0)	nvl(mgr, 0) = 0	Index Full Scan	Filter

3.14 다양한 옵션 조건 처리 방식

1. OR 조건 활용

```
select * from 거래
where (:cust_id is null or 고객ID = :cust_id)
and   거래일자 between :dt1 and :dt2
```

Execution Plan

```
0      SELECT STATEMENT Optimizer=ALL_ROWS
1      0      TABLE ACCESS (FULL) OF '거래' (TABLE)
```

- CUST_ID IS NULL 조건이 아님
- 이 방식은 옵션 조건 컬럼을 선두로 두고 [고객ID + 거래일자] 인덱스를 구성해도 사용할 수 없음
- 인덱스 선두 컬럼에 대한 옵션 조건에 OR를 사용하면 안됨

Execution Plan

```
0      SELECT STATEMENT Optimizer=ALL_ROWS
1      0      FILTER
2      1      TABLE ACCESS (BY INDEX ROWID) OF '거래' (TABLE)
3      2      INDEX (RANGE SCAN) OF '거래_IDX3' (INDEX)
```

Predicate information (identified by operation id):

```
1 - filter(TO_DATE(:DT1)<=TO_DATE(:DT2))
2 - filter(:CUST_ID IS NULL OR "고객ID"=TO_NUMBER(:CUST_ID))
3 - access("거래일자">=:DT1 AND "거래일자"<=:DT2)
```

- [거래일자 + 고객ID]로 구성된 인덱스를 사용할 수 있지만, 고객ID를 필터 조건으로 사용하는 문제 발생
- 인덱스에 100만 건을 스캔하여, 그만큼 고객ID를 필터링 하는 것이라면 고객ID를 인덱스에 포함할 필요 없음
- 쿼리를 분리하는 방안(Union All, Nvl, Decode) / 파티션 기법등이 고려되어야 함

3.14 다양한 옵션 조건 처리 방식

2. Like / Between 조건 활용

- Like / Between은 옵션 조건 처리를 위해 많이 사용하는 대표적인 방식
- 변별력이 좋은 필수 조건이 있는 상황에선 이들 패턴을 사용하는 것은 나쁘지 않음
- 문제는 필수 조건의 변별력이 좋지 않을 경우 성능에 문제가 발생
- Like / Between 패턴을 사용하고자 할 때는 아래 네 가지 경우에 속하는 점검이 필요함(Between은 1, 2번)
 1. 인덱스 선두 컬럼
 2. Null 허용 컬럼
 3. 숫자형 컬럼
 4. 가변 길이 컬럼

3.14 다양한 옵션 조건 처리 방식

2-1. Like / Between 패턴 점검 - ① 인덱스 선두 컬럼

- 인덱스 선두 컬럼에 대한 옵션 조건을 Like / Between 연산자로 처리하는 것은 금물
- 인덱스 구성 : 고객ID + 거래일자

```
select * from 거래
where 고객ID like :cust_id || '%'
and 거래일자 between :dt1 and :dt2
```

- 고객ID 값이 입력되면 약간의 비효율이 있어도 변별력이 종기에 빠르게 조회되지만, 입력하지 않을 경우 인덱스를 통해 모든 거래 데이터를 스캔하면서 조건을 필터링 함
- 직측과 같은 경우라면 인덱스를 [거래일자 + 고객ID] 순으로 구성해야 함

2-2. Like / Between 패턴 점검 - ② Null 허용 컬럼

- 위 쿼리에서 :cust_id 별수에 Null 값을 입력하면 아래와 같음

```
select * from 거래
where 고객ID like '%'
and 거래일자 between :dt1 and :dt2
```

- 일차적인 성능문제를 떠나 결과 집합의 오류가 발생함
- 실제로 원하는 데이터는 고객ID가 없는 데이터를 찾고하 함인데 전체 데이터 출력됨

3.14 다양한 옵션 조건 처리 방식

2-3. Like / Between 패턴 점검 - ③ 해당 컬럼의 데이터 타입(숫자형)

- 숫자형이면서 인덱스 액세스 조건으로 사용 가능한 컬럼에 Like 방식의 옵션 처리는 금물
- 인덱스 구성 : 거래일자 + 고객ID

```
select * from 거래
where 거래일자 = :trd_dt
and 고객ID like :cust_id || '%'
```

- 좌측 SQL의 경우 고객ID가 숫자형 컬럼이면, 아래와 같이 자동 형변환 발생
- 특정 고객의 하루 거래를 조회하고 싶은 데 해당 일자의 모든 거래를 스캔하면서 고객 ID 조건을 필터링 (형변환으로 인해 고객ID가 필터 조건으로 사용)

2-4. Like / Between 패턴 점검 - ④ 가변 길이 컬럼 확인

- 불 필요한 데이터 스캔을 줄이기 위한 기법

```
Where 고객명 like :cust_nm || '%'
```

```
Where 고객명 like :cust_nm || '%'  
and length(고객명)  
= length(nvl(:cust_nm, 고객명))
```

```
Where 고객명 like :cust_nm
```

- Like 패턴으로 “김훈” 고객을 찾기 위해 :cust_nm 변수에 “김훈 ” 을 입력하면 " 김훈 남“ 고객도 같이 조회됨
- 컬럼 값 길이가 가변적일 때는 변수 값 길이가 같은 레코드만 조회되도록 조건을 일부 추가할 수 있음
- 다른 방법으로는 입력한 값과 정확히 일치하는 고객명만 출력하도록 '%'가 없는 Like 조건절을 사용할 수도 있음

3.14 다양한 옵션 조건 처리 방식

3. Union All 활용

```
select * from 거래
where (:cust_id is null or 고객ID = :cust_id)
and 거래일자 between :dt1 and :dt2
```

Execution Plan

```
0      SELECT STATEMENT Optimizer=ALL_ROWS
1      0      TABLE ACCESS (FULL) OF '거래' (TABLE)
```

- :cust_id 변수에 따라 값을 입력여부에 따라 SQL 중 하나만 실행되게 하는 방식
- 두 개의 쿼리를 만족할 인덱스가 모두 존재해야 함

```
select * from 거래
where :cust_id is null
and 거래일자 between :dt1 and :dt2
union all
select * from 거래
where :cust_id is not null
and 고객ID = :cust_id
and 거래일자 between :dt1 and :dt2
```

Execution Plan

```
0      SELECT STATEMENT Optimizer=ALL_ROWS
1      0      UNION ALL
2      1      FILTER -- :cust_id is null
3      2      TABLE ACCESS (BY LOCAL INDEX ROWID) OF '거래' (TABLE)
4      3      INDEX (RANGE SCAN) OF '거래_IDX1' (INDEX) -- 거래일자
5      1      FILTER -- :cust_id is not null
6      5      TABLE ACCESS (BY LOCAL INDEX ROWID) OF '거래' (TABLE)
7      6      INDEX (RANGE SCAN) OF '거래_IDX2' (INDEX) -- 고객ID + 거래일자
```

3.14 다양한 옵션 조건 처리 방식

4. Nvl / Decode 활용

```
select * from 거래
where  고객ID = nvl(:cust_id, 고객ID)
and    거래일자 between :dt1 and :dt2
```

```
select * from 거래
where  고객ID = decode(:cust_id, null, 고객ID, :cust_id)
and    거래일자 between :dt1 and :dt2
```

Execution Plan

```
-----
0      SELECT STATEMENT Optimizer=ALL_ROWS
1      0      CONCATENATION
2      1      FILTER      -- :cust_id is null
3      2      TABLE ACCESS (BY LOCAL INDEX ROWID) OF '거래' (TABLE)
4      3      INDEX (RANGE SCAN) OF '거래_IDX1' (INDEX)  -- 거래일자
5      1      FILTER      -- :cust_id is not null
6      5      TABLE ACCESS (BY LOCAL INDEX ROWID) OF '거래' (TABLE)
7      6      INDEX (RANGE SCAN) OF '거래_IDX2' (INDEX)  -- 고객ID + 거래일자
```

- Nvl, Decode 둘 다 사용 가능, 실행계획 동일
- 고객ID 컬럼을 함수 인자로 사용했는데도 쿼리 변환 (OR Expansion)이 발생하여 Union All 방식으로 변환
- Nvl, Decode 대신 Coalesce, Case 문을 사용하면 쿼리 변환이 작동하지 않음
- 좌측은 Oracle 기준이며, Tiberio의 경우는 다름 (Tiberio의 경우 Bug로 의심됨)
- 좌측 쿼리의 단점은 앞서 언급한 것처럼 Null인 레코드는 추출할 수 없음(누락됨)

3.15 함수호출부하 해소를 위한 인덱스 구성

1. PL/SQL 함수의 성능적 특성

```
select 회원번호, 회원명, 생년, 생월일, encryption(전화번호)
from   회원
where  회원번호 = :member_no  -- 한 건 조회
```

```
select 회원번호, 회원명, 생년, 생월일, encryption(전화번호)
from   회원
where  생월일 like '01%'  -- 수십 ~ 수백 만 건 조회
```

PL/SQL 사용자 정의 함수가 느린 3가지 이유

- ① 가상머신(VM) 상에서 실행되는 인터프리터 언어
- ② 호출 시마다 컨텍스트 스위칭 발생
- ③ 내장 SQL에 대한 Recursive Call 발생

- PL/SQL로 작성한 함수와 프로시저를 컴파일하면 JAVA 언어처럼 바이트코드(Bytecode, Machine-Readable Code)가 생성
- 이를 해석하고 실행할 수 있는 PL/SQL 엔진(Virtual Machine)만 있다면 어디서든 실행될 수 있음
- PL/SQL은 인터프리터 언어이므로 그것으로 작성한 함수 실행시 매 번 SQL 실행엔진과 PL/SQL 가상머신 사이에 컨텍스트 스위칭(Context Switching)이 발생
- 이러한 컨텍스트 스위칭은 CPU 사용의 주요 원인이 되기도 함
- 스칼라 서브쿼리로의 유도로 일부 성능 개선 가능

3.15 함수호출부하 해소를 위한 인덱스 구성

2. Context Switching 성능 부하 - Non Recursive

```
create table t_context (no number, char_time varchar(21));
```

```
create or replace function date_to_char(p_dt date)
return varchar
as
begin
    return to_char(p_dt, 'yyyy/mm/dd hh24:mi:ss');
end;
/
```

```
alter session set sql_trace = 'Y';
```

```
insert into t_context
select /*+ No Context Switching */ rownum as no
    , to_char(sysdate+rownum, 'yyyy/mm/dd hh24:mi:ss') as ch_tm
  from dual
Connect by level < 100000;
```

```
Commit;
```

```
Insert into t_context
Select /*+ Context Switching */ rownum as no
    , date_to_char(sysdate+rownum) as ch_tm
  from dual
connect by level < 100000;
```

```
commit;
```

```
alter session set sql_trace = 'N';
```

[내장 함수 사용]

stage	count	cpu	elapsed	current	query	disk	rows
parse	1	0.01	0.00	0	51	0	0
exec	1	0.21	0.22	3391	150	1	100009
fetch	0	0.00	0.00	0	0	0	0
sum	2	0.22	0.22	3391	201	1	100009

[사용자 정의 함수 사용]

stage	count	cpu	elapsed	current	query	disk	rows
parse	1	0.00	0.00	0	28	0	0
exec	1	2.28	2.29	3089	66	0	100002
fetch	0	0.00	0.00	0	0	0	0
sum	2	2.28	2.29	3089	94	0	100002

3.15 함수호출부하 해소를 위한 인덱스 구성

3. Context Switching 성능 부하 - Recursive Call

```
create or replace function date_to_char2(p_dt date)
return varchar
as
    curr_dt    varchar(21);
begin
    select to_char(p_dt, 'yyyy/mm/dd hh24:mi:ss')
        into curr_dt
        from dual;

    return curr_dt;
end;
/

alter session set sql_trace = 'Y';

insert into t_context
select /*+ Recursive Call */ rownum as no
      , date_to_char2(sysdate + rownum) as char_time
  from dual
 connect by level < 100000;

commit;

alter session set sql_trace = 'N';
```

[재귀 호출]

stage	count	cpu	elapsed	current	query	disk	rows
parse	1	0.00	0.00	0	28	0	0
exec	1	11.01	11.07	3106	72	0	200001
fetch	0	0.00	0.00	0	0	0	0
sum	2	11.01	11.07	3106	100	0	200001

3.15 함수호출부하 해소를 위한 인덱스 구성

4. 조건절의 함수 사용 호출

```
create or replace function emp_avg_sal
return number
is
    l_avg_sal number;
begin
    select avg(sal)
    into l_avg_sal
    from emp;

    return l_avg_sal;
end;
/
```

No	쿼리 형태	함수 호출 회수
1	select /*+ full(emp) */ * from emp where sal >= emp_avg_sal();	
2	create index emp_ix01 on emp(sal); select /*+ index(emp(sal)) */ * from emp where sal >= emp_avg_sal();	
3	create index emp_ix02 on emp(deptno); select /*+ index(emp(deptno)) */ * from emp where sal >= emp_avg_sal() and deptno = 20;	
4	create index emp_ix03 on emp(deptno, sal); select /*+ index(emp(deptno, sal)) */ * from emp where sal >= emp_avg_sal() and deptno = 20;	
5	create index emp_ix04 on emp(deptno, ename, sal); select /*+ index(emp(deptno, ename, sal)) */ * from emp where sal >= emp_avg_sal() and deptno = 20;	
6	select /*+ index(emp(deptno, sal)) */ * from emp where sal >= emp_avg_sal() and deptno >= 10;	

3.16 인덱스 설계

1. 유형 분석

인덱스	Case	조건유형	Index Access_Predicate	Index Filter_Predicate	Table Filter_Predicate
(A, B)	Case #1	<ul style="list-style-type: none">A = 100B = 'SEOUL'	A=100 and B='SEOUL'	-	-
	Case #2	<ul style="list-style-type: none">A = 100B != 'SEOUL'	A=100	B<>'SEOUL'	-
	Case #3	<ul style="list-style-type: none">A >= 100 and A <= 1000C = 1	A>=100 and A<=1000	-	C=1
	Case #4	<ul style="list-style-type: none">A > 100C = 1 and D <> 'A'	A>=100	-	C=1 and D<>'A'

3.16 인덱스 설계

2. 분석 및 제안

조건유형 (인덱스 : A + B)	분석 및 제안
<ul style="list-style-type: none"> A = 100 B != 'SEOUL' 	<ul style="list-style-type: none"> 분석 : 인덱스 2번째 컬럼을 필터 처리 (B<>'SEOUL') 하므로, 이 조건의 형태를 바꿀 수 있는지 여부에 따라 효율성을 높일 수 있는지 여부가 결정됨 제안 : 조건변경 [B in ('PUSAN', 'INCHUN', 'KWANGJU', 'DAEJEON', 'ULSAN')] 이러한 형태로 전환이 가능하면, 인덱스 2번째 컬럼까지 액세스 조건으로 인덱스 검색 효율을 높일 수 있음
<ul style="list-style-type: none"> A >= 100 A <= 1000 C = 1 	<ul style="list-style-type: none"> 분석 : 인덱스 2번째 컬럼은 조건절에 쓰이지 않고, 3번째 컬럼이 테이블에서 필터 조건(C=1)으로 사용되므로, 컬럼 C를 인덱스 컬럼의 후보로 활용할 수 있음 제안1 : C 컬럼을 추가 (A, B, C) 컬럼으로 기존 인덱스를 재구성한다면, (C=1)은 인덱스의 필터 조건으로 사용 제안2 : 신규 인덱스(A, C)를 만들 경우, A 컬럼으로 구성된 인덱스의 중복 발생, 이것을 허용할 지 여부를 고려. (위 사례만을 위한 인덱스로는 최적임) 제안3 : 기존인덱스에 C 컬럼을 2번째 컬럼으로 인덱스(A, C, B) 재구성 이 경우에만 국한한다면, 좋은 튜닝 방법 중 하나임 그러나, 기존에 A,B 컬럼 조합으로 사용되던 조건 조합이 있는 경우 이는 성능에 악영향을 끼칠 수 있음. 이것을 강행해도 될지 여부에 대해 고민해야 하며, 실전에서는 이런 경우, 대개는 이 방법을 선택하지 못한다. 검증이 어렵고, 분석 대상 범위가 방대할 수 있기 때문이다
<ul style="list-style-type: none"> A > 100 C = 1 D <> 'A' 	<ul style="list-style-type: none"> 분석 : B컬럼은 인덱스에는 구성되어 있으나, 조건절에는 사용되지 않았고, C, D 컬럼의 조건은 기존 인덱스 컬럼에 포함되지 않아, 인덱스 사용 효율이 낮다. 이에 조건절에 사용된 컬럼을 기존 인덱스에 어떻게 포함하여 재구성할 것인지를 결정해야 한다 제안1 : C, D 컬럼을 기존 인덱스의 필터 조건으로 추가하여 인덱스(A, B, C, D) 또는 인덱스(A, B, C) 또는 인덱스(A, B, D) 중 효율성이 좋은 것으로 재구성을 고려 한다.(기존 영향도 최소화) 제안2 : C, D 컬럼을 기존 인덱스의 중간에 추가하여 인덱스(A, C, D, B)로 재구성 고려함. 단, 기존 인덱스의 B 컬럼의 순서가 2번째에서 4번째로 변경되었기 때문에 기존에 사용하던 프로그램의 성능이 나빠질 여지가 있음.(검증 요구됨)