

2장 인덱스 기본

2.1 인덱스 구조 및 탐색

- 데이터베이스 테이블에서 데이터 찾는 법

1. 테이블 전체 스캔
2. 인덱스 이용

인덱스는 큰 테이블에서 소량의 데이터를 검색할 때 사용
(온라인 트랜잭션 처리 OLTP)

- 인덱스 튜닝 방법 핵심요소

1. 인덱스 스캔 과정에서 발생하는 비효율을 줄인다. (인덱스 스캔 효율화 튜닝)
2. 테이블 액세스를 줄인다 (랜덤 I/O방식 사용, 랜덤 액세스 최소화 튜닝)

- DB 성능이 느린 이유

디스크 I/O (특히 랜덤 I/O)가 많이 발생하면 느리다. 따라서 SQL튜닝은 랜덤 I/O와의 전쟁

2.1.2 인덱스 구조

- 루트블록 : 최상위 블록
- 브랜치블록 : 루트와 리프 사이에 위치한 블록
- 리프블록 : 키값 순 정렬, ROWID(테이블 레코드를 가리키는 주소값)

ROWID : 데이터 블록 주소(DBA, Data Block Address) + 로우 번호

데이터 블록 주소 : 데이터 파일 번호 + 블록 번호

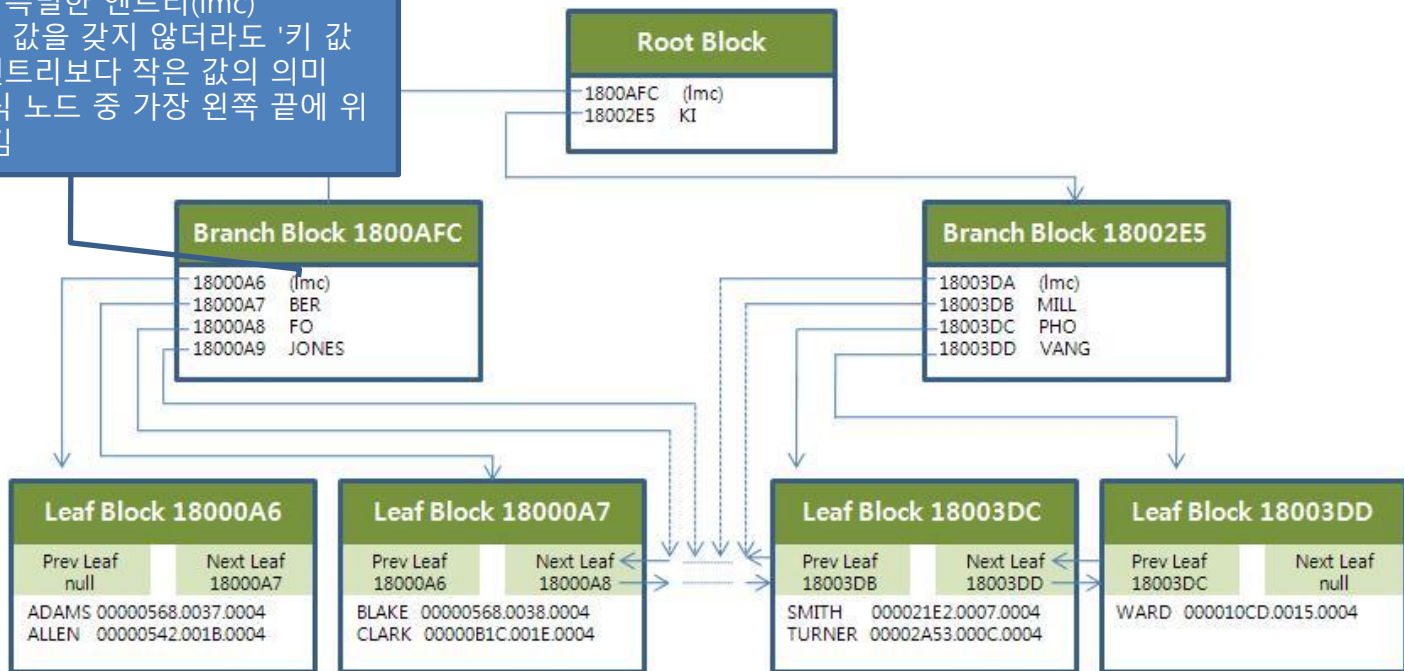
블록 번호 : 데이터파일 내에서 부여한 상대적 순번

로우 번호 : 블록 내 순번

LeftMost Child(LMC)

키 값을 갖지 않는 특별한 엔트리(lmc)

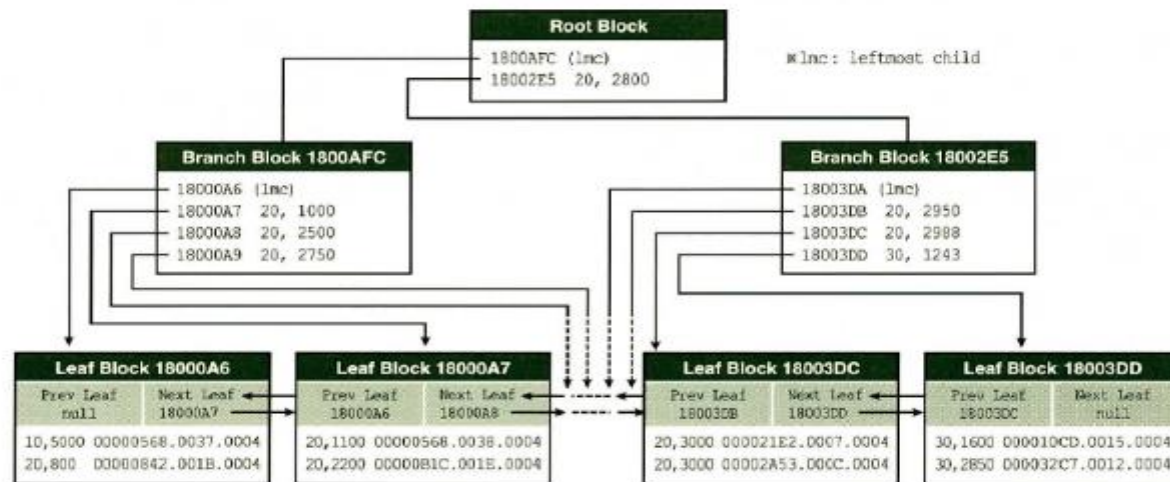
lmc는 명시적인 키 값을 갖지 않더라도 '키 값을 가진 첫 번째 엔트리보다 작은 값의 의미'를 가진 첫 번째 엔트리보다 작은 값의 의미. 브랜치 블록의 자식 노드 중 가장 왼쪽 끝에 위치한 블록을 가리킴



- 루트(Root)를 포함한 브랜치(Branch)블록에 저장된 엔트리에는 하위 노드 블록을 찾아가기 위한 DBA(Data Block Address)정보를 갖음
- 최말단 리프(Leaf) 블록에는 인덱스 키 컬럼과 함께 해당 테이블 레코드를 찾아가기 위한 주소정보(rowid)를 갖음
- 오라클은 인덱스 구성 컬럼이 모두 null인 레코드를 저장하지 않음
- 인덱스와 테이블 레코드 간에는 서로 1:1 대응 관계를 갖음
- 브랜치에 저장된 레코드 개수는 바로 하위 레벨 블록 개수와 일치
- 테이블 레코드에서 값이 갱신되면 리프 노드 인덱스 키 값도 같이 갱신
- 브랜치 노드는 인덱스 분할(split)에 의해 새로운 블록이 추가되거나 삭제될 때만 갱신

인덱스 수직적,수평적 탐색

- 인덱스는 수직적 탐색으로 인덱스 스캔의 시작점을 찾고, 이후 수평적 탐색으로 본격적으로 데이터를 찾음
- 수직적 탐색 : 루트에서 시작해 리프 블록까지 수직적 탐색을 통해 조건을 만족하는 첫 번째 레코드를 찾는 과정
- 수평적 탐색 : 리프 블록끼리는 양방향 링크드 리스트이다. 따라서 수직적 탐색을 통해 첫 번째 레코드인 리프에 도달한 후, 해당 조건을 만족하는 다른 리프 블록을 찾기 위해 수평적으로 탐색하는 과정
- 수평적 탐색을 하는 이유 :
 1. 조건절을 만족하는 데이터를 모두 찾기 위해
 2. ROWID를 찾기 위해(인덱스 스캔 후 테이블을 스캔하는 경우도 있기 때문)



2.1.5 결합 인덱스 구조와 탐색

- 두 개 이상의 칼럼을 결합해서 만든 인덱스

Ex) 고객 테이블의 (성별, 고객명)

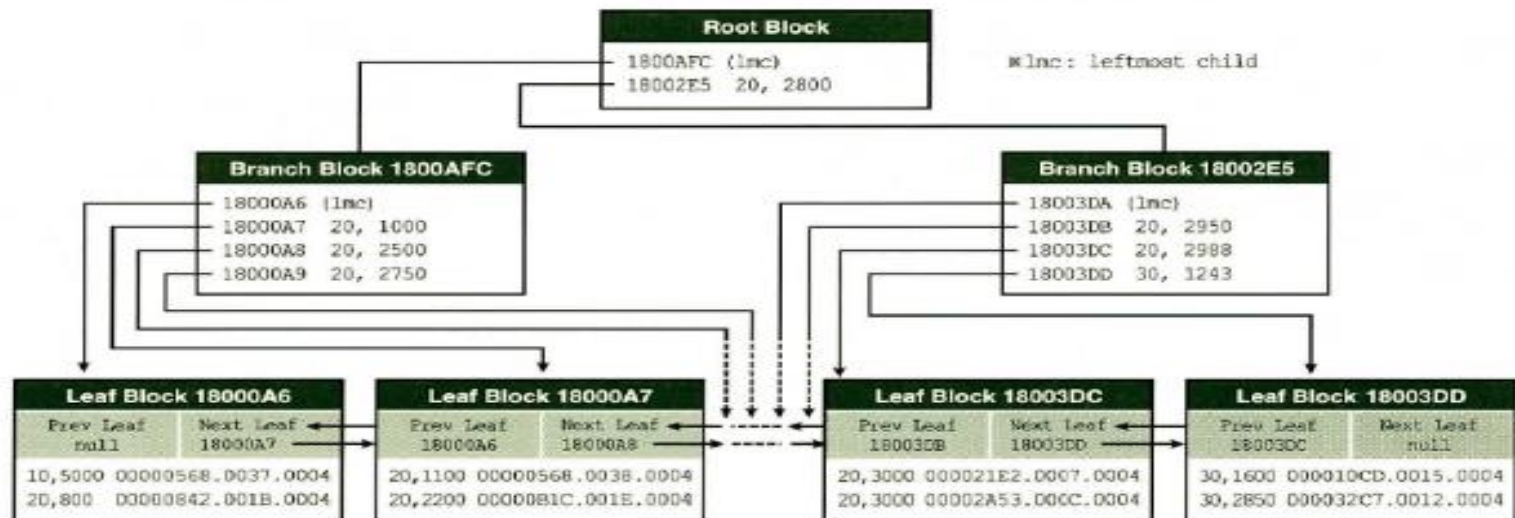
- 결합인덱스로 (성별, 고객명)을 설정한 경우, 인덱스 스캔 시작점은 성별='남'인 첫 번째 레코드가 아니라, 성별='남', 고객명='이재희' 라는 레코드 사실을 알아야 한다.
- 엑셀의 평면구조인 필터(1번 조건 필터 후 2번 조건을 거는 방식)와 동작방식이 다르다.

- emp 테이블에 deptno + sal 컬럼 순으로 생성한 결합 인덱스

- deptno=20 and sal >=2000' 조건으로 쿼리할 경우 두번째 리프 블록, 두번째 레코드부터 스캔이 시작

- 수직적 탐색 과정에서 deptno뿐만 아니라 sal값에 대한 필터링도 같이 이루어지기 때문

- 마지막 블록 첫 번째 레코드에서 스캔을 멈추게 될 것



2.2 인덱스 기본 사용법

- 인덱스를 정상적으로 사용(Index Range Scan) 하기 위한 가장 첫 번째 조건은 **인덱스 선두 컬럼이 조건절에 있어야 함.**
- 두 번째 조건은 **인덱스의 선두 컬럼을 가공하지 않은 상태로 사용해야 함.**
- 즉, 인덱스를 정상적으로 사용한다라는 의미는 **조건절에 인덱스 선두 컬럼을 가공하지 않은 상태로 사용해야 함을** 의미한다.
- 인덱스 컬럼을 가공해서 사용하면 **인덱스 스캔 시작점을 찾을 수 없으므로 Range Scan이 불가능**
 - '이름' 컬럼을 인덱스 컬럼으로 가진 인덱스에서 이름이 '홍길동'이라는 사람을 찾는다면 수직적 탐색을 통해 '홍길동' 리프를 찾고 수평적 탐색을 통해 '홍길동'이 나올 때까지 스캔하면 종료.
 - 동일한 조건에서 이름에 '길동'을 포함한 사람을 찾는다면 수직적 탐색을 통해 시작점을 알 수도 없고 스캔의 끝도 알 수 없음.
- 데이터를 가공하는 **SUBSTR, NVL, LIKE** 등을 잘못 사용하면 인덱스를 정상적으로 사용하지 않음

2.2 인덱스 기본 사용법

- BTree 인덱스를 정상적으로 사용하려면 범위 스캔 시작지점을 찾기 위해 루트 블록부터 리프 블록까지의 수직적 탐색 과정을 거쳐야 함
 - 인덱스컬럼에 SUBSTR 사용금지
 - 인덱스컬럼에 NULL값 미포함
 - 인덱스컬럼에 LIKE 조건 주의('%지수%', '지수' 와 같이 % 중간이나 %뒤에 사용금지)
 - OR 조건 지양
 - IN 조건 지양 > UNION ALL로 풀어서 인덱스 사용 가능
- IN 조건절에 대해서 SQL 옵티마이저는 IN-List Iterator 방식을 사용(IN-List 개수만큼 Index Range Scan을 반복)
- 부정형으로 비교된 조건 사용금지 (!=, NOT IN)
 - IS (NOT) NULL 로 비교된 경우 : 인덱스에서는 NULL에 대한 정보가 없어서 FULL TABLE SCAN이 발생(문자형의 경우, 컬럼명 > ' ' 을 사용하여 인덱스 사용 가능)

2.2 인덱스 컬럼의 가공

인덱스 컬럼 가공 사례	튜닝 방안
SUBSTR(업체명, 1, 2) = '대한'	업체명 LIKE '대한%'
월급여 * 12 = 36000000	월급여 = 36000000 / 12
TO_CHAR(일시, 'yyyymmdd') = :dt	일시 >= TO_DATE(:dt, 'yyyymmdd') AND 일시 < TO_DATE(:dt, 'yyyymmdd') + 1
연령 직업 = '30공무원'	연령 = 30 AND 직업 = '공무원'
회원번호 지점번호 = :str	회원번호 = SUBSTR(:str, 1, 2) AND 지점번호 = SUBSTR(:str, 3, 4)
NVL(주문수량, 0) >= 100	주문수량 >= 100
NVL(주문수량, 0) < 100	not null 컬럼이면 NVL제거, 아니면 함수기반 인덱스(FBI) 생성 고려 => create index 주문_x01 on 주문(nvl(주문수량, 0));
dept 컬럼이 char type 인 경우 DEPT = 7788	DEPT = TO_CHAR(7788)
EMPNO != '1234'	EMPNO > '1234' OR EMPNO < '1234'
Date 컬럼이 날짜인 경우 DATE IS NOT NULL	DATE > 0;

```
CREATE OR REPLACE FUNCTION GET_KORNM
```

```
(  
    V_FROM IN VARCHAR2,  
    V_TO IN VARCHAR2  
)  
RETURN VARCHAR2  
IS
```

```
    OUT_REAL_NM VARCHAR2(100);  
    TYPE V_ARR IS TABLE OF VARCHAR2(10);  
    V_FIRST V_ARR;  
    V_LAST V_ARR;  
    V_MID V_ARR;
```

```
BEGIN
```

```
V_LAST := V_ARR('김', '이', '박', '최', '정'  
    , '강', '조', '윤', '장', '임'  
    , '오', '한', '신', '서', '권'  
    , '황', '안', '송', '유', '홍'  
    , '전', '고', '문', '손', '양'  
    , '배', '조', '백', '허', '남');
```

```
V_MID := V_ARR('민', '현', '동', '인', '지'  
    , '현', '재', '우', '건', '준'  
    , '승', '영', '성', '진', '준'  
    , '정', '수', '광', '영', '호'  
    , '중', '훈', '후', '우', '상'  
    , '연', '철', '아', '윤', '은');
```

```
V_FIRST := V_ARR('유', '자', '도', '성', '상'  
    , '남', '식', '일', '철', '병'  
    , '혜', '영', '미', '환', '식'  
    , '숙', '자', '희', '순', '진'  
    , '서', '빈', '정', '지', '하'  
    , '연', '성', '공', '안', '원');
```

```
SELECT SUBSTR(V_LAST(ROUND(DBMS_RANDOM.VALUE(1 , 30), 0)) ||  
    V_MID(ROUND(DBMS_RANDOM.VALUE(1 , 30), 0)) ||  
    V_FIRST(ROUND(DBMS_RANDOM.VALUE(1 , 30), 0)) ||  
    V_MID(ROUND(DBMS_RANDOM.VALUE(1 , 30), 0)) ||  
    V_FIRST(ROUND(DBMS_RANDOM.VALUE(1 , 30), 0))  
    ,V_FROM,V_TO)  
    INTO OUT_REAL_NM  
    FROM DUAL;
```

```
RETURN OUT_REAL_NM;
```

```
END;
```

```
/
```

```
--출처: https://cyh0214.tistory.com/entry/무작위-한글이름-만들기 [맑은안개이야기:티스토리]
```

```
CREATE TABLE TUNN_TEST
(
EMPNO VARCHAR2(6) PRIMARY KEY,
DEPTNO VARCHAR2(2),
SAL NUMBER(5),
BONUS NUMBER(3),
GRADE VARCHAR2(1),
GENDER VARCHAR2(6),
KOR_NM VARCHAR2(12),
ENG_NM VARCHAR2(15),
BIRTHDAY VARCHAR2(8),
CELLPHONE VARCHAR2(11),
REGION VARCHAR2(12),
REG_DATE VARCHAR2(8),
EXT_DATE VARCHAR2(8)
);
```

```
CREATE TABLE TAX_TUNN
(
TAXNO VARCHAR2(6) PRIMARY KEY,
YYYY NUMBER(4),
TAX_CODE VARCHAR2(10),
REF_FORWARD NUMBER(2),
CONFIRM_NUM NUMBER(2),
TAX_AMOUNT NUMBER(6)
);
```

```
CREATE TABLE EQ_TUNN
(
EQ_NO VARCHAR2(1),
EQ_DATE VARCHAR2(8),
EQ_CHG_NUM VARCHAR2(6),
EQ_SPEC_NO VARCHAR2(4),
EQ_NM VARCHAR2(15),
EQ_STATUS VARCHAR2(1)
);
```

```
CREATE INDEX TUNN_IDX1
ON TUNN_TEST(BIRTHDAY);
```

```
CREATE INDEX TUNN_IDX2
ON TUNN_TEST(BONUS);
```

```
CREATE INDEX TUNN_IDX3
ON TUNN_TEST(KOR_NM);
```

```
CREATE INDEX TUNN_IDX4
ON TUNN_TEST(REGION);
```

```
CREATE INDEX TUNN_IDX5
ON TUNN_TEST(CELLPHONE);
```

```
CREATE INDEX TAX_IDX1
ON TAX_TUNN(YYYY, TAX_CODE,
REF_FORWARD, CONFIRM_NUM);
```

```
CREATE UNIQUE INDEX PK_EQ_IDX1
ON
EQ_TUNN(EQ_NO,EQ_DATE,EQ_CHG_NUM);
```

```
(UNIQUE INDEX 없을 때 확인용)
-- CREATE INDEX EQ_IDX2
--ON EQ_TUNN(EQ_NO,EQ_DATE);
```

```

INSERT INTO TUNN_TEST
SELECT T.EMPNO
, CASE WHEN T.DEPT13 = 1 AND DR19 = 1 AND DR09 = 1 THEN '40'
      WHEN T.DEPT13 = 2 THEN '20'
      WHEN T.DEPT13 = 3 THEN '30'
      ELSE '10' END AS DEPTNO
      , SAL
      , GRADE*20 AS BONUS
      , DECODE(GRADE,4,'A',3,'B',2,'C',1,'D') AS GRADE
      , DECODE(GEN12, '1', 'MALE', 'FEMALE') AS GENDER
      , GET_KORNM('1', '3') AS KOR_NM
      , NM1 || NM2 || NM3 || NM4 AS ENG_NM
      , YYYY || MM || DD AS BIRTHDAY
      , '010' || CEL1 || CEL2 AS CELL
      , CASE WHEN DR19 = 9 AND DR09 IN (9,0) THEN '경상'
            WHEN DR19 = 9 AND DR09 = 8 THEN '충청'
            WHEN DR19 = 9 AND DR09 = 7 THEN '전라'
            WHEN DR19 = 9 AND DR09 = 6 AND DEPT13 = 1 THEN '제주'
            WHEN DR19 = 9 AND DR09 IN (1,2,3,4,5) THEN '광주'
            WHEN DR19 = 8 AND DR09 IN (1,2,3,4,5) THEN '대전'
            WHEN DR19 = 8 AND DR09 IN (6,7,8,9,0) THEN '대구'
            WHEN DR19 = 7 AND DR09 = 7 THEN '부산'
            WHEN DR19 IN (4,5,6) THEN '경기'
            ELSE '서울' END REGION
      , TO_CHAR(ADD_MONTHS(TO_DATE('202112', 'YYYYMM'), -REG_MM), 'YYYYMM') AS REG_DATE
      , CASE WHEN DR19 = 3 AND YYYY = 2000 THEN TO_CHAR(ADD_MONTHS(TO_DATE('202112', 'YYYYMM'), -REG_MM + DR19), 'YYYYMM') ELSE '' END AS EXT_DATE
FROM (
  SELECT LEVEL AS LVL
        , LEVEL+1000 AS EMPNO
        , ROUND(DBMS_RANDOM.VALUE(1,9),0) AS DR19
        , ROUND(DBMS_RANDOM.VALUE(0,9),0) AS DR09
        , ROUND(DBMS_RANDOM.VALUE(100,9999),0) AS CEL1
        , LPAD(ROUND(DBMS_RANDOM.VALUE(1,9999),0), 4, '0') AS CEL2
        , DBMS_RANDOM.STRING('U', 1) AS NM1
        , DBMS_RANDOM.STRING('L', ROUND(DBMS_RANDOM.VALUE(2,3),0)) AS NM2
        , DBMS_RANDOM.STRING('U', 1) AS NM3
        , DBMS_RANDOM.STRING('L', ROUND(DBMS_RANDOM.VALUE(2,3),0)) AS NM4
        , ROUND(DBMS_RANDOM.VALUE(1,2),0) AS GEN12
        , ROUND(DBMS_RANDOM.VALUE(1,3),0) AS DEPT13
        , ROUND(DBMS_RANDOM.VALUE(3000, 10000)/100,0)*100 AS SAL
        , ROUND(DBMS_RANDOM.VALUE(1925,2002),0) AS YYYY
        , LPAD(ROUND(DBMS_RANDOM.VALUE(1,12),0),2,'0') AS MM
        , LPAD(ROUND(DBMS_RANDOM.VALUE(1,31),0),2,'0') AS DD
        , ROUND(DBMS_RANDOM.VALUE(1, 500), 0) AS REG_MM
        , ROUND(DBMS_RANDOM.VALUE(1,4),0) AS GRADE
FROM DUAL
CONNECT BY LEVEL <= 100000
) T
;
COMMIT;

```

```

INSERT INTO TAX_TUNN
SELECT T.TAXNO
      , YYYY      -- 기준연도
      , CASE WHEN T.TAXCD = 1 THEN 'AAA'
        WHEN T.TAXCD = 2 THEN 'BBB'
        WHEN T.TAXCD = 3 THEN 'CCC'
        ELSE 'DDD' END || RAN_STR || YYYY || MM AS TAX_CODE      -- 과세구분코드
      , DR09 AS REP_FORWARD      -- 보고회차
      , MM AS CONFIRM_NUM      -- 실명확인번호
      , SAL AS TAX_AMOUNT      -- 과세금액
FROM (
  SELECT LEVEL AS LVL
        , LEVEL+1000 AS TAXNO
        , ROUND(DBMS_RANDOM.VALUE(1,9),0) AS DR09
        , ROUND(DBMS_RANDOM.VALUE(1,5),0) AS TAXCD
        , ROUND(DBMS_RANDOM.VALUE(3000, 10000)/100,0)*100 AS SAL
        , ROUND(DBMS_RANDOM.VALUE(2000,2022),0) AS YYYY
        , ROUND(DBMS_RANDOM.VALUE(1,12),0) AS MM
        , DBMS_RANDOM.STRING('U', 1) AS RAN_STR
  FROM DUAL
  CONNECT BY LEVEL <= 100000
) T
;
COMMIT;

```

```

INSERT INTO EQ_TUNN
SELECT EQ_NO, EQ_DATE, LPAD(ROW_NUMBER() OVER (PARTITION BY EQ_NO, EQ_DATE ORDER BY
      EQ_NO, EQ_DATE), 6, '0') AS EQ_CHG_NUM
      , EQ_NO || EQ_NO3 AS EQ_SPEC_NO
      , EQ_NO || EQ_NO3 || '장비' AS EQ_NM
      , DECODE(EQ_STATUS, 1, 'S', 2, 'S', 3, 'A', 4, 'B', '') AS EQ_STATUS
FROM (
      SELECT DBMS_RANDOM.STRING('U', 1) AS EQ_NO
      , TO_CHAR(TO_DATE('20200102', 'YYYYMMDD') + ROUND(DBMS_RANDOM.VALUE(1,20),0),
      'YYYYMMDD') AS EQ_DATE
      , LPAD(ROUND(DBMS_RANDOM.VALUE(1,5), 0),3,'0') AS EQ_NO3
      , ROUND(DBMS_RANDOM.VALUE(1,5),0) AS EQ_STATUS
      FROM DUAL
      CONNECT BY LEVEL <= 100000
      )
ORDER BY EQ_DATE DESC
;
COMMIT;

```

-- 인덱스 목록 및 정보

```

SELECT A.TABLE_NAME, A.INDEX_NAME, B.COLUMN_NAME, A.UNIQUENESS, LEAF_BLOCKS, DISTINCT_KEYS,
      AVG_LEAF_BLOCKS_PER_KEY, AVG_DATA_BLOCKS_PER_KEY, CLUSTERING_FACTOR
FROM ALL_INDEXES A
      , ALL_IND_COLUMNS B
WHERE A.OWNER = 'SCOTT'
      AND A.TABLE_NAME = 'TUNN_TEST'
      AND A.INDEX_NAME = B.INDEX_NAME
ORDER BY TABLE_NAME, INDEX_NAME
;

```

-- LIKE 연산 (88p)

```
SELECT EMPNO  
FROM TUNN_TEST  
WHERE KOR_NM LIKE '%미%';
```

```
SELECT EMPNO  
FROM TUNN_TEST  
WHERE KOR_NM LIKE '미%';
```

결과값이 동일하지는 않지만 INDEX RANGE SCAN을 타는지 여부만 확인한다.

처음과 같이 LIKE 조건에서 % 사이에 문자가 들어가는 경우 TABLE FULL SCAN을 탄다
두번째에서는 인덱스컬럼에 LIKE를 사용했지만 **문자가 먼저 오는 경우 INDEX RANGE SCAN**을 탄다

Operation	Object Name	Rows	Bytes	Cost	Object Node	In/Out	PStart	PStop	Access Predicates	Filter Predicates
SELECT STATEMENT Optimizer Mode=ALL_ROWS		5 K	63 K	310						
TABLE ACCESS FULL	SCOTT.TUNN_TEST	5 K	63 K	310					"KOR_NM" LIKE '%미%' AND "KOR_NM" IS NOT NULL	

Operation	Object Name	Rows	Bytes	Cost	Object Node	In/Out	PStart	PStop	Access Predicates	Filter Predicates
SELECT STATEMENT Optimizer Mode=ALL_ROWS		21	273	24						
TABLE ACCESS BY INDEX ROWID	SCOTT.TUNN_TEST	21	273	24						
INDEX RANGE SCAN	SCOTT.TUNN_IDX3	21		2					"KOR_NM" LIKE '미%' "KOR_NM" LIKE '미%'	

-- OR 또는 IN-List 조건을 OR-Expansion으로 유도(88p)

```
SELECT /* USE_CONCAT */ *
FROM TUNN_TEST
WHERE KOR_NM = '조성자'
OR CELLPHONE = '01061359190'
;
```

```
SELECT *
FROM TUNN_TEST
WHERE KOR_NM = '조성자'
UNION ALL
SELECT *
FROM TUNN_TEST
WHERE CELLPHONE = '01061359190'
AND (KOR_NM <> '조성자' OR KOR_NM IS NULL)
;
```

1번째 실행계획을 통해서 **OR 조건의 경우 인덱스가 적절하게 있다면** 옵티마이저가 **USE_CONCAT** 힌트 사용하는 것과 같이 INDEX_RANGE_SCAN 으로 수행하는 것을 확인할 수 있다.

2번째 **UNION ALL** 로 변환해서 수행하는 경우, **자동으로 INDEX RANGE SCAN**으로 수행되는 것을 확인할 수 있다.

Operation	Object Name	Rows	Bytes	Cost	Object Node	In/Out	PStart	PStop	Access Predicates	Filter Predicates
SELECT STATEMENT Optimizer Mode=ALL_ROWS		6	432	3						
TABLE ACCESS BY INDEX ROWID	SCOTT.TUNN_TEST	6	432	3						
BITMAP CONVERSION TO ROWIDS										
BITMAP OR										
BITMAP CONVERSION FROM ROWIDS										
INDEX RANGE SCAN	SCOTT.TUNN_IDX3			1					"KOR_NM"='조성자'	
BITMAP CONVERSION FROM ROWIDS										
INDEX RANGE SCAN	SCOTT.TUNN_IDX5			1					"CELLPHONE"='01061359190'	

Operation	Object Name	Rows	Bytes	Cost	Object Node	In/Out	PStart	PStop	Access Predicates	Filter Predicates
SELECT STATEMENT Optimizer Mode=ALL_ROWS		6	432	9						
UNION-ALL										
TABLE ACCESS BY INDEX ROWID	SCOTT.TUNN_TEST	5	360	7						
INDEX RANGE SCAN	SCOTT.TUNN_IDX3	5		1					"KOR_NM"='조성자'	
TABLE ACCESS BY INDEX ROWID	SCOTT.TUNN_TEST	1	72	2						"KOR_NM" <> '조성자' OR 'KOR_NM' IS NULL
INDEX RANGE SCAN	SCOTT.TUNN_IDX5	1		1					"CELLPHONE"='01061359190'	

-- IN 조건절(89p)

```
SELECT *  
FROM TUNN_TEST  
WHERE CELLPHONE IN ('01061359190',  
                    '01089132248')  
;
```

```
SELECT *  
FROM TUNN_TEST  
WHERE CELLPHONE = '01061359190'  
UNION ALL  
SELECT *  
FROM TUNN_TEST  
WHERE CELLPHONE = '01089132248'  
;
```

1번째 실행계획을 통해서 **IN 조건의 경우 인덱스가 적절하게 있다면** 옵티마이저가 INDEX_RANGE_SCAN 으로 수행하는 것을 확인할 수 있다.

2번째 **UNION ALL 로 변환**해서 수행하는 경우, **자동으로 INDEX RANGE SCAN**으로 수행되는 것을 확인할 수 있다.

INLIST ITERATOR 과정이 없는 2번째 UNION ALL에서 Cost가 좋게 나오는 것을 확인할 수 있다.

Operation	Object Name	Rows	Bytes	Cost	Object Node	In/Out	PStart	PStop	Access Predicates	Filter Predicates
SELECT STATEMENT Optimizer Mode=ALL_ROWS		2	144	5						
INLIST ITERATOR										
TABLE ACCESS BY INDEX ROWID	SCOTT.TUNN_TEST	2	144	5						
INDEX RANGE SCAN	SCOTT.TUNN_IDX5	2		3					"CELLPHONE"='01061359190' OR "CELLPHONE"='01089132248'	

Operation	Object Name	Rows	Bytes	Cost	Object Node	In/Out	PStart	PStop	Access Predicates	Filter Predicates
SELECT STATEMENT Optimizer Mode=ALL_ROWS		2	144	4						
UNION-ALL										
TABLE ACCESS BY INDEX ROWID	SCOTT.TUNN_TEST	1	72	2						
INDEX RANGE SCAN	SCOTT.TUNN_IDX5	1		1					"CELLPHONE"='01061359190'	
TABLE ACCESS BY INDEX ROWID	SCOTT.TUNN_TEST	1	72	2						
INDEX RANGE SCAN	SCOTT.TUNN_IDX5	1		1					"CELLPHONE"='01089132248'	

-- 결합인덱스에서 인덱스 처음값(92p)

```
SELECT *  
FROM TAX_TUNN  
WHERE YYYY = 2018  
      AND SUBSTR(TAX_CODE, 1, 4) = 'AAAA'  
      AND REF_FORWARD = 8  
      AND CONFIRM_NUM = 6  
;
```

조건절의 YYYY, TAX_CODE, REF_FORWARD, CONFIRM_NUM 으로 인덱스가 생성되어 있는 경우
중간조건에 SUBSTR이 들어가도 **첫번째 조건에 의해 INDEX RANGE SCAN**이 수행된다

Operation	Object Name	Rows	Bytes	Cost
SELECT STATEMENT Optimizer Mode=ALL_ROWS		1	64	24
TABLE ACCESS BY INDEX ROWID	SCOTT.TAX_TUNN	1	64	24
INDEX RANGE SCAN	SCOTT.TAX_IDX1	1		23

-- 인덱스 범위가 많은 경우 비효율(94p)

```
SELECT /*+ FULL(TUNN_TEST) */ EMPNO
FROM TUNN_TEST
WHERE BONUS > 20;
```

```
SELECT /*+ INDEX(TUNN_TEST TUNN_IDX2) */ EMPNO
FROM TUNN_TEST
WHERE BONUS > 20;
```

BONUS 값은 20, 40, 60, 80 총 4가지로 구성되어 있고, 최소값이 20이고,
20보다 큰 건수는 총 10만건의 데이터 중에서 8만건 이상이다. (80퍼센트 이상)

실행계획을 보면 2번째 INDEX RANGE SCAN 보다 **TABLE FULL SCAN**에서 Cost가 훨씬 좋게 나오는 것을 확인할 수 있다.

Operation	Object Name	Rows	Bytes	Cost	Object Node	In/Out	PStart	PStop	Access Predicates	Filter Predicates
SELECT STATEMENT Optimizer Mode=ALL_ROWS		83 K	731 K	309						
TABLE ACCESS FULL	SCOTT.TUNN_TEST	83 K	731 K	309						"BONUS">20

Operation	Object Name	Rows	Bytes	Cost	Object Node	In/Out	PStart	PStop	Access Predicates	Filter Predicates
SELECT STATEMENT Optimizer Mode=ALL_ROWS		83 K	731 K	3673						
TABLE ACCESS BY INDEX ROWID	SCOTT.TUNN_TEST	83 K	731 K	3673						
INDEX RANGE SCAN	SCOTT.TUNN_IDX2	83 K		165					"BONUS">20	

-- UNIQUE 인덱스 생성을 통해서 정렬 수행(95p)

```
SELECT *  
FROM EQ_TUNN  
WHERE EQ_NO = 'C'  
      AND EQ_DATE = '20200120'  
;
```

```
CREATE UNIQUE INDEX PK_EQ_IDX1 ON EQ_TUNN(EQ_NO,EQ_DATE,EQ_CHG_NUM);
```

EQ_NO, EQ_DATE, EQ_CHG_NUM 으로 **UNIQUE INDEX가 생성되어 있는 경우**
ORDER BY 절을 사용하지 않아도 인덱스 컬럼 순서대로 정렬이 수행된다

	EQ_NO	EQ_DATE	EQ_CHG_NUM	EQ_SPEC_NO	EQ_NM	EQ_STATUS
1	C	20200120	000001	C004	C004장비	S
2	C	20200120	000002	C001	C001장비	S
3	C	20200120	000003	C005	C005장비	B
4	C	20200120	000004	C002	C002장비	A
5	C	20200120	000005	C001	C001장비	A
6	C	20200120	000006	C003	C003장비	A
7	C	20200120	000007	C002	C002장비	B
8	C	20200120	000008	C002	C002장비	S
9	C	20200120	000009	C003	C003장비	B
10	C	20200120	000010	C002	C002장비	A

2.2.5 ORDER BY 절에서 컬럼 가공

- ORDER BY 절에 || 연산을 사용하여 가공한 컬럼을 지정한 경우
- SELECT 절에 TO_CHAR 등을 사용하여 컬럼을 가공한 뒤 ORDER BY 절에 가공한 컬럼명으로 지정한 경우
- 실행계획에서 SORT ORDER BY STOPKEY 단계가 추가되어 성능이 저하된다.

2.2.6 SELECT-LIST에서 컬럼 가공

- MAX, MIN 함수 사용시 수직탐색에 유리하게 인덱스가 짜여져있는 경우,
- 실행계획에서 INDEX RANGE SCAN과 FIRST ROW 단계로 성능이 우수하다.
- 만일 인덱스 생성시 문자열로 지정되어 있는 컬럼을 SELECT 절에서 TO_NUMBER로 변환 후 MAX, MIN 함수 수행시 인덱스 컬럼을 변형시킨 경우에 해당하기 때문에 실행 계획에서 SORT 과정을 수행하게 된다.
- 마지막 예제(102p)의 Top N 알고리즘의 경우 366p에서 확인 가능
- 스칼라 서브쿼리를 사용하거나 INDEX_DESC 힌트 활용
- 전체 데이터를 조회해야 하는 경우, 윈도우 함수와 KEEP 절을 사용

-- 인덱스를 이용한 최소, 최대값 계산(101p)

```
SELECT NVL(MAX(TO_NUMBER(EQ_CHG_NUM)), 0)
FROM EQ_TUNN
WHERE EQ_NO = 'C'
      AND EQ_DATE = '20200120';
```

```
SELECT NVL(TO_NUMBER(MAX(EQ_CHG_NUM)), 0)
FROM EQ_TUNN
WHERE EQ_NO = 'C'
      AND EQ_DATE = '20200120';
```

1번째는 인덱스 컬럼을 변형 후에 MAX 연산을 하고 있으며, 실행계획을 보면 INDEX RANGE SCAN 과정에서 Rows와 Bytes 값이 높게 출력된다.

2번째는 인덱스 컬럼을 그대로 MAX 연산을 하고 있으며, 실행계획을 보면 INDEX RANGE SCAN(MIN/MAX)를 수행하고 FIRST_ROW 과정을 수행하며

Rows와 Bytes 값이 1번째와 비교해서 상당히 좋게 나오는 것을 확인할 수 있다.

따라서 **MAX, MIN 사용시 인덱스 컬럼을 지정**하면 성능에 유리하게 할 수 있다.

Operation	Object Name	Rows	Bytes	Cost	Object Node	In/Out	PStart	PStop	Access Predicates	Filter Predicates
SELECT STATEMENT Optimizer Mode=ALL_ROWS		1	18	3						
SORT AGGREGATE		1	18							
INDEX RANGE SCAN	SCOTT.PK_EQ_IDX1	222	3 K	3					"EQ_NO"='C' AND "EQ_DATE"='20200120'	

Operation	Object Name	Rows	Bytes	Cost	Object Node	In/Out	PStart	PStop	Access Predicates	Filter Predicates
SELECT STATEMENT Optimizer Mode=ALL_ROWS		1	18	3						
SORT AGGREGATE		1	18							
FIRST ROW		1	18	3						
INDEX RANGE SCAN (MIN/MAX)	SCOTT.PK_EQ_IDX1	1	18	3					"EQ_NO"='C' AND "EQ_DATE"='20200120'	

2.2.7 자동 형변환

- 오라클에서는 숫자형과 문자형이 만나면 숫자형이 이긴다. 숫자형 컬럼 기준으로 문자형 컬럼을 변환한다.
- 날짜형과 문자형이 만나면 날짜형이 이긴다.
- 컬럼 타입은 문자형으로 지정하고, WHERE절에서는 생년월일 = 19821225와 같이 숫자형을 사용하면 실제로는 TO_NUMBER(생년월일)로 변환되어 인덱스 컬럼에 변경이 된 것으로 보고 INDEX Range Scan을 사용할 수 없다.
- 반면 LIKE 연산의 경우, 문자열 비교 연산자이므로 숫자형 컬럼이 문자형 컬럼으로 변경되어 처리된다.
- 계좌번호 컬럼과 같이 **숫자형일 때, LIKE 조건으로 검색하면 자동형변환이 발생**해 인덱스 액세스 조건으로 사용하지 못한다. (TO_CHAR(계좌번호) LIKE '10%')
- DECODE(A, B, C, D) 함수의 경우 반환값의 데이터 타입은 세 번째 인자인 C에 의해 결정된다. C가 문자형이고, D가 숫자형이면 D는 문자형으로 변환된다.
- 또한 C가 null 값이면, varchar2로 취급되므로 주의해서 사용해야 한다.
- **SQL 성능은 블록 I/O를 줄이는 데에서 결정된다. 컬럼의 타입을 확실히 체크하고 형변환 함수를 잘 사용해주어야 한다.**

2.3 인덱스 확장기능 사용법

2.3.1. Index Range Scan

- B*Tree 인덱스의 가장 일반적인 액세스 방식
- 인덱스 루트에서 리프 블록까지 **수직적으로 탐색 후 필요한 범위만 스캔**하는 방식
- 실행계획에서 INDEX (RANGE SCAN) 으로 표시
- 인덱스 선두 컬럼을 가공하지 않은 상태로 조건절에 사용해야 한다.

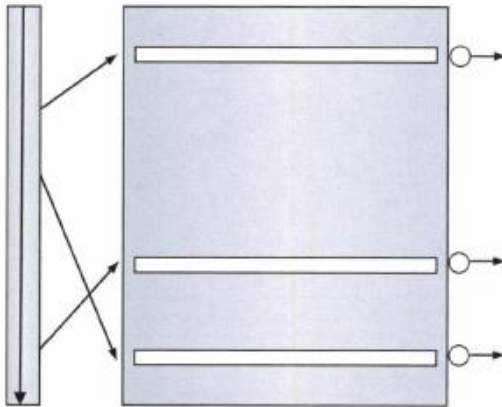
2.3.2. Index Full Scan

```
SQL> create index emp_ename_sal_idx on emp (ename, sal);
```

```
SQL> select *  
2 from emp  
3 where sal > 9000  
4 order by ename;
```

Execution Plan

```
0 SELECT STATEMENT Optimizer=ALL_ROWS  
1 0 TABLE ACCESS (BY INDEX ROWID) OF 'EMP' (TABLE)  
2 1 INDEX (FULL SCAN) OF 'EMP_ENAME_SAL_IDX' (INDEX)
```



- 인덱스 리프 블록을 처음부터 끝까지 수평적으로 탐색하는 방식

실행계획에서 INDEX (FULL SCAN) 으로 표시

- 인덱스 선두컬럼을 조건절에 쓰지 않더라도, 조건절에 의해 소량의 데이터만 조회하는 경우 옵티마이저가 TABLE FULL SCAN보다 유리하다 판단해서 INDEX FULL SCAN이 수행될 수 있다.

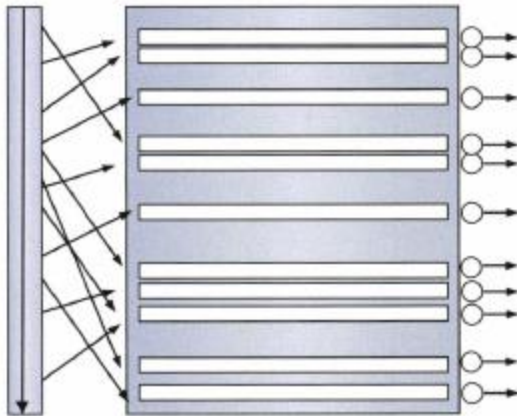
- FIRST_ROWS 와 같은 힌트 사용시 옵티마이저 모드를 변경하여 소트 연산을 생략하여 성능 개선 효과로 사용할 수 있다. 하지만 데이터를 많이 읽게 될수록 성능은 오히려 안좋아지게 되므로 조심해야 한다.

2.3.2. Index Full Scan

```
SQL> select /** first_rows */ *  
2 from emp  
3 where sal > 1000  
4 order by ename;
```

Execution Plan

```
0 SELECT STATEMENT Optimizer=HINT: FIRST_ROWS  
1 0 TABLE ACCESS (BY INDEX ROWID) OF 'EMP' (TABLE)  
2 1 INDEX (FULL SCAN) OF 'EMP_ENAME_SAL_IDX' (INDEX)
```



- Index Full Scan 역시 Range Scan과 마찬가지로 **인덱스 컬럼순으로 정렬**된다.

- 즉, Sort Order By 연산을 생략할 수 있다. 그러한 목적으로도 사용한다.

- 옵티마이저는 Table Full Scan이 오히려 더 유리한 상황이 와도, Order By가 있다면 이 소트 연산을 생략하기 위해 Index Full Scan을 선택할 경우도 있다.

2.3.2. Index Unique Scan

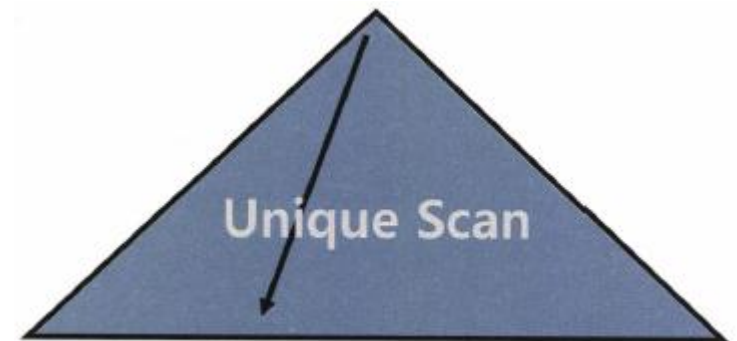
- 수직적 탐색만으로 데이터를 찾는 스캔 방식
- Unique 인덱스를 = 조건으로 탐색하는 경우 작동
- 실행계획에서 INDEX (UNIQUE SCAN) 으로 표시
- Unique 인덱스가 존재하는 컬럼은 중복값 없이 입력되지 않게 DBMS가 정합성을 관리해준다. 그래서 데이터를 = 조건으로 찾은 후 더이상 탐색이 필요없다.
- Unique 인덱스가 존재해도 BETWEEN, LIKE 등 범위 조건으로 검색하면 Index Range Scan을 한다. (수직적 탐색으로만 찾을 수 없기 때문)
- Unique 결합 인덱스에 대해 일부 컬럼으로만 검색 할 때에도, Index Range Scan을 사용한다.

```
SELECT *
```

```
FROM EQ_TUNN
```

```
WHERE EQ_NO = 'C' AND EQ_DATE = '20200120' AND EQ_CHG_NUM = '000198';
```

Operation	Object Name	Rows	Bytes	Cost
SELECT STATEMENT Optimizer Mode=ALL_ROWS		1	34	3
TABLE ACCESS BY INDEX ROWID	SCOTT.EQ_TUNN	1	34	3
INDEX UNIQUE SCAN	SCOTT.PK_EQ_IDX1	1		2



2.3.4. Index Skip Scan

- **인덱스 선두 컬럼이 조건절에 없어도 인덱스를 활용**하는 새로운 방식
- 루트 또는 브랜치 블록에서 읽은 컬럼 값 정보를 이용해 조건절에 부합하는 레코드를 포함할 '가능성이 있는' 리프 블록만 골라서 액세스 하는 스캔 방식
- **Distinct Value 개수가 적은**(성별과 같이 2개만 존재) **선두 컬럼이 조건절에 없고** 후행 컬럼의 Distinct Value 개수가 많을 때 유용(고객번호와 같이 고유한 값)
- 실행계획에서 INDEX (SKIP SCAN) 으로 표시
- INDEX_SS 힌트로 사용 / NO_INDEX_SS 힌트로 사용방지
- 인덱스 선두컬럼에 대한 조건절은 있고, 중간 컬럼에 대한 조건절이 없는 경우에도 Skip Scan 사용가능하다. 대신 후속 컬럼에 BETWEEN과 같은 범위가 조건으로 지정되어야 성능상 유리하다.
- Distinct Value가 적은 두 개의 선두컬럼이 모두 조건절에 없는 경우에도 유용하게 사용 가능하다.
- 선두컬럼이 부등호, BETWEEN, LIKE 같은 범위검색 조건일 때에도 사용가능하다.



2.3.4. Index Skip Scan

```
CREATE INDEX TUNN_IDX7 ON TUNN_TEST(GENDER, SAL, BONUS);
```

```
SELECT GENDER, SAL, BONUS  
FROM TUNN_TEST  
WHERE SAL > 8000  
      AND BONUS > 40;
```

```
SELECT /*+ INDEX_SS(TUNN_TEST TUNN_IDX7) */ GENDER, SAL, BONUS  
FROM TUNN_TEST  
WHERE SAL > 8000  
      AND BONUS > 40;
```

Index Skip Scan은 리프 블록을 순회하면서 값의 범위를 벗어나면 **다시 브랜치 블록을 읽어서** 어디로 Skip 해야할지 결정하므로 **그만큼 오버헤드가 발생**한다.
대부분의 경우 Index Range Scan보다 성능이 뒤떨어지므로 차선의 선택이라는 것을 기억해야 한다.

Operation	Object Name	Rows	Bytes	Cost	Object Node	In/Out	PStart	PStop	Access Predicates	Filter Predicates
SELECT STATEMENT Optimizer Mode=ALL_ROWS		12	216	2						
INDEX RANGE SCAN	SCOTT.TUNN_IDX7	12	216	2					"ENG_NM" LIKE 'Ab%' AND "SAL">5000	"ENG_NM" LIKE 'Ab%' AND "SAL">5000

Operation	Object Name	Rows	Bytes	Cost	Object Node	In/Out	PStart	PStop	Access Predicates	Filter Predicates
SELECT STATEMENT Optimizer Mode=ALL_ROWS		12	216	18						
INDEX SKIP SCAN	SCOTT.TUNN_IDX7	12	216	18					"ENG_NM" LIKE 'Ab%' AND "SAL">5000	"ENG_NM" LIKE 'Ab%' AND "SAL">5000

2.3.5. Index Fast Full Scan

- 논리적인 인덱스 트리 구조를 무시하고, **인덱스 세그먼트 전체를 Multiblock I/O 방식**으로 스캔하는 방식
- **쿼리에 사용한 컬럼이 모두 인덱스에 포함되어** 있을 때만 사용할 수 있다.
- INDEX_FFS 힌트로 사용 / NO_INDEX_FFS 힌트로 사용방지
- 디스크로부터 대량의 인덱스 블록을 읽어야 할 때 큰 효과를 발휘한다.
- 속도는 빠르지만 인덱스 리프 노드가 갖는 연결 리스트 구조를 무시하여 **정렬되지 않는다.**


Index Full Scan	Index Fast Full Scan
1. 인덱스 구조를 따라 스캔	1. 세그먼트 전체를 스캔
2. 결과집합 순서 보장	2. 결과집합 순서 보장 안 됨
3. Single Block I/O	3. Multi Block I/O
4. (파티션 돼 있지 않다면) 병렬스캔 불가	4. 병렬스캔 가능
5. 인덱스에 포함되지 않은 컬럼 조회시에도 사용 가능	5. 인덱스에 포함된 컬럼으로만 조회할 때 사용 가능

2.3.5. Index Fast Full Scan

```
CREATE INDEX TUNN_IDX6 ON TUNN_TEST(GENDER, SAL, KOR_NM);
```

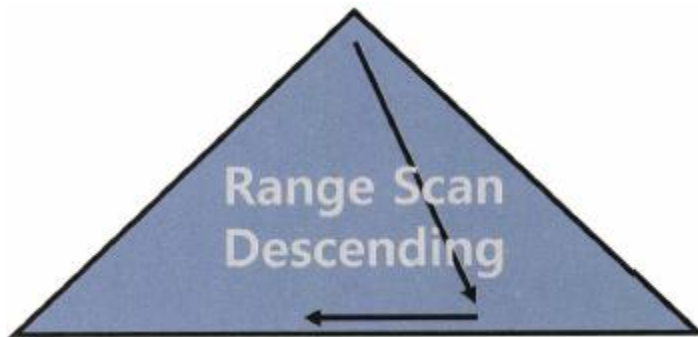
```
SELECT GENDER, SAL, KOR_NM  
FROM TUNN_TEST  
WHERE SAL >= 5000  
      AND SAL <= 6000  
;
```

쿼리에 사용한 GENDER, SAL, KOR_NM이 **모두 인덱스에 포함되어 있고, 범위 검색으로 일량이 많은 경우** 옵티마이저가 INDEX FAST FULL SCAN을 선택하는 것을 알 수 있다

Operation	Object Name	Rows	Bytes	Cost	Object Node	In/Out	PStart	PStop	Access Predicates	Filter Predicates
SELECT STATEMENT Optimizer Mode=ALL_ROWS		17 K	283 K	106						
 INDEX FAST FULL SCAN	SCOTT.TUNN_IDX6	17 K	283 K	106					"SAL" <= 6000 AND "SAL" >= 5000	

2.3.6. Index Range Scan Descending

- **내림차순 정렬시**, 조회컬럼에 인덱스가 있으면 옵티마이저가 알아서 인덱스를 거꾸로 읽는 실행계획을 수립한다.
- 만약 옵티마이저가 인덱스를 거꾸로 읽지 않는다면, INDEX_DESC 힌트로 사용



```
SQL> select * from emp  
2  where empno > 0  
3  order by empno desc
```

Execution Plan

```
0  SELECT STATEMENT Optimizer=ALL_ROWS  
1  0  TABLE ACCESS (BY INDEX ROWID) OF 'EMP' (TABLE)  
2  1  INDEX (RANGE SCAN DESCENDING) OF 'PK_EMP' (INDEX (UNIQUE))
```

만일 EMPNO 컬럼이 문자형인 경우, WHERE EMPNO > '' 로 변환하면 INDEX RANGE SCAN DESCENDING 을 탄다.

```
SELECT * FROM TUNN_TEST WHERE EMPNO > ''  
ORDER BY EMPNO DESC;
```

Operation	Object Name	Rows	Bytes	Cost	Object Node	In/Out	PStart	PStop	Access Predicates	Filter Predicates
SELECT STATEMENT Optimizer Mode=ALL_ROWS		5 K	351 K	177						
TABLE ACCESS BY INDEX ROWID	SCOTT.TUNN_TEST	5 K	351 K	177						
INDEX RANGE SCAN DESCENDING	SCOTT.SYS_C0024781	900		5					"EMPNO">"	

QUIZ

-- 총 10만건

SELECT *

FROM TUNN_TEST;

	EMPNO	DEPTNO	SAL	BONUS	GRADE	GENDER	KOR_NM	ENG_NM	BIRTHDAY	CELLPHONE	REGION	REG_DATE	EXT_DATE
1	1194	20	4100	20	D	MALE	조성자	OgcrIih	19581023	01080562446	경기	198112	
2	1195	20	5200	80	A	FEMALE	한현공	NhdoZrd	19770608	01061359190	경기	198405	
3	1196	20	8000	20	D	MALE	신영혜	ZweRwyl	19931028	01064231031	서울	199211	
4	1197	20	10000	20	D	MALE	정홍성	GqewGkd	19800911	01054608786	경기	198202	
5	1198	20	4500	40	C	FEMALE	양영미	DsnCzi	19890613	01097160030	경기	198803	
6	1199	20	3800	60	B	MALE	문영공	NwsfVfou	19570523	01052292732	경기	198303	
7	1200	10	4000	60	B	FEMALE	윤호자	JnkOkvi	19480110	01080697130	경기	198210	
8	1201	10	5100	20	D	MALE	백연빈	CzspBjeu	19881116	01046669106	서울	200107	
9	1202	30	9800	40	C	MALE	송후환	BvjPoj	19380429	0108134662	서울	200105	
10	1203	20	5500	20	D	FEMALE	문성숙	QzfwMbx	19670307	01081889633	경기	198901	

SELECT MIN(SAL), MAX(SAL) -- 최소연봉, 최대연봉
FROM TUNN_TEST
WHERE DEPTNO = '10';

현재 존재하는 인덱스가 없다면,
위 SQL의 성능을 최적화시키려면 무슨 작업들을 해줘야 할까요?
현재의 실행계획은 아래처럼 수행된다

Operation	Object Name	Rows	Bytes	Cost	Object Node	In/Out	PStart	PStop	Access Predicates	Filter Predicates
SELECT STATEMENT Optimizer Mode=ALL_ROWS		1	6	309						
SORT AGGREGATE		1	6							
TABLE ACCESS FULL	SCOTT.TUNN_TEST	25 K	146 K	309					"DEPTNO"='10'	

만일 부서번호 관계없이 전체 인원으로 구할 때

```
CREATE INDEX TUNN_IDX8 ON TUNN_TEST(SAL);
```

(AS-IS)

```
SELECT MAX(SAL) AS MAX_SAL  
       , MIN(SAL) AS MIN_SAL  
FROM TUNN_TEST;
```

(TO-BE)



QUIZ


```
SELECT MIN(SAL), MAX(SAL) -- 부서번호가 10인 사람들 중 최소연봉, 최대연봉  
FROM TUNN_TEST  
WHERE DEPTNO = '10'
```

부서번호 조건이 늘어난 경우 인덱스는 어떻게 생성하는게 좋을까요

1. 이미 SAL이 단독인덱스로 있으니 DEPTNO로 단독 인덱스를 생성하면 될지?
2. DEPTNO, SAL 결합인덱스로 생성해야 할지?

```
CREATE INDEX TUNN_IDX9 ON TUNN_TEST(DEPTNO);  
CREATE INDEX TUNN_IDX10 ON TUNN_TEST(DEPTNO, SAL);
```

(TO-BE)



11g COST와 CARDINALITY

- COST : 쿼리를 수행하는 소모한 논리적 비용.
- CARDINALITY : 쿼리 조건에 맞는 레코드 건수.
- SORT AGGREGATE : 전체 Row를 대상으로 집계를 수행할 때 나타나며, Oracle 실행계획에 'sort'라는 표현이 사용됐지만 실제 소트가 발생하진 않는다.

Ex) Cost=633K : 633,000 비용발생(논리적 비용 = IO + MEM + CPU + NET + ...)
Card=42M : 42,000,000건(접근하는 레코드 수)
Bytes=15G : 15,000,000,000(42,000,000 * 1 ROW의 총 길이)

위의 플랜을 종합하면, 해당 쿼리는 4,200만 건의 데이터를 읽어오며,
이때 15,000,000,000Byte의 네트워크 트래픽을 유발한다.
그리고 이때 비용은 633K가 발생함을 오라클 옵티마이저가 알려 주고 있다.

출처: <https://argolee.tistory.com/67> [놀멍:티스토리]

참고사이트

<http://wiki.gurubee.net/pages/viewpage.action?pageId=26739921>

인덱스 구조

<http://wiki.gurubee.net/pages/viewpage.action?pageId=26739954>

인덱스 컬럼의 가공

Optimizing Oracle Optimizer 조동욱 지음 2008년판

TOMATO_SQL 작성가이드

실행계획 스크린샷 : SQL Gate (Oracle 11.2.0.1.0 버전)