TECHNOLOGY & DEVELOPER

'혁명'을 꿈꾸다

나는 평생 주기적으로 '혁명'을 꿈꾸었다.
누군들 그렇지 않겠는가.
내게 혁명이란, 세계를 송두리째 바꾸는 것이 아니라
내가 선험적으로, 혹은 환경이나 습관의 축적에 의해
결정되었다고 느끼는 일상 속의 나를
통째로 뒤집어 변화시키는 일이다.
나를 근본적으로 변혁시키지 않고선
세계가 변화하지 않기 때문이다.

– 박범신의 〈비우니 향기롭다〉 中에서 –

옵티마이저의 비용계산 방법과 실행원리

시스템의 성능 향상을 위한 노력

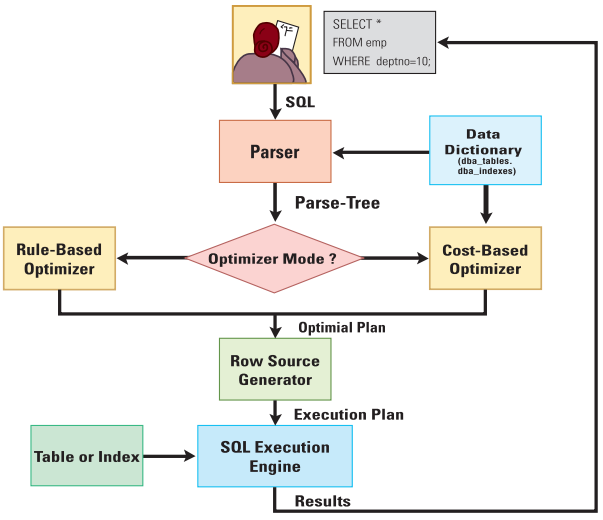
저자_주종면, 오라클 ACE, PLAN 정보기술(www.plandb.co.kr / Jina6678@paran.com)

이번 호 technical tips에서는 SQL문의 성능을 결정하는 비용기반 옵티마이저의 핵심 원리 중에 비용계산 방법에 대해 알아 보도록 하겠다. 대부분의 개발자들이 작성하는 SQL문은 비용기반 옵티마이저 환경에서 작성된 실행계획을 통해 실행되어지는데, 이때 옵티마이저의 비용계산 방법에 대해 정확히 이해한다면 더 좋은 성능을 보장 받는 SQL문을 작성할 수 있을 것이다.

1. SQL문 처리과정

옵티마이저의 비용계산 방법을 소개하기 전에 우선 SQL문의 처리과정에 대해 알아보자.

사용자가 실행하는 SQL문은 파서(Parser)에게 전달되고 파서는 데이터 딕셔너리 정보를 참조하여 SQL문에 대한 구문분석(Syntax와 Symantics)을 수행한다. 이 결과를 파스-트리(Parse-Tree)라고 한다.



〈그림 1〉 SQL문의 처리과정

파스-트리는 옵티마이저에게 전달되는데 오라클 데이터베이스에는 공식 기반 옵티마이저(Rule-Based Optimizer)와 비용기반 옵티마이저(Cost-Based Optimizer)가 있다. 비용기반 옵티마이저에 의해 산출된 적정 플랜(Optimal Plan)은 로우-소스 생성기(Row Source Generator)에게 전달되고 이것은 실행계획(Execution Plan)으로 결정된다.

우리가 SET AUTOTRACE, SQL*TRACE와 TKPROF와 같은 튜닝 도구들을 통해 참조할 수 있는 결과에 바로 이 실행계획이 포함되어 있다. 이 실행계획은 SQL 실행엔진(SQL Execution Engine)에 의해 테이블과 인덱스를 참조하여 그 결과를 사용자에게 리턴하게 되는 것이다.

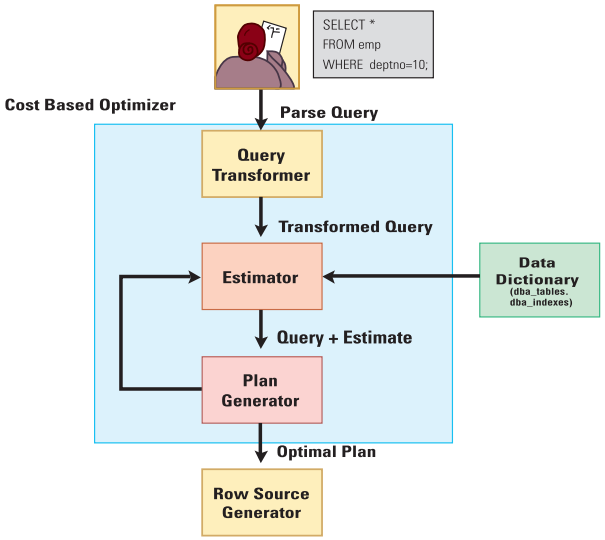
이어서, 비용기반 옵티마이저가 어떤 비용계산 방법을 통해 적절한 실행계획을 찾아내는지를 소개할 것이다. 개발작업 때 처음부터 좋은 실행계획을 작성할 수 있도록 SQL문을 작성한다면 SQL 튜닝에 대한 불필요한 시간과 비용을 줄여 나감으로써 좋은 성능의 시스템을 개발할 수 있는 첫걸음이 되는 것이다.

2. 비용기반 옵티마이저의 구조

이번에는 구체적으로 비용기반 옵티마이저의 아키텍처에 대해 알아보도록 하겠다.

CBO(Cost Based Optimizer)가 어떻게 비용을 계산하고 어떻게 실행계획을 작성하는지를 알기 위해서는 보다 구체적으로 CBO의 아키텍처에 대해 알고 있어야 한다.

〈그림 2〉에서와 같이 CBO는 쿼리 변형기(Query Transformer), 비용 계산기(Estimator), 쿼리 작성기(Query Generator) 3가지 구조로 구성되어 있다. 그럼, 각 구성 요소와 비용계산 알고리즘을 통해 실행계획 작성 방법에 대해 알아 보자.



〈그림 2〉 CBO의 아키텍처

3. 쿼리 변형기 (Query Transformer)

쿼리 변형기는 파서 (Parser)에 의해 구문 분석된 결과를 전달 받아 잘못 작성된 SQL문을 정확한 문장으로 변형시키는 역할을 수행한다.

· 잘못된 데이터 타입으로 조건 값을 검색하면 변형된다.
(S_DATE 컬럼은 날자 컬럼인데 문자 값을 검색할 때 사용하는 인용부호를 사용한 경우)

```
SQL> SELECT * FROM emp  WHERE s_date = '1999-01-01';
--> SQL> SELECT * FROM emp  WHERE s_date = TO_DATE('1999-01-01');
```

· LIKE 연산자는 %(와일드 카드)와 함께 검색하는 경우 사용되지만, 그렇지 않은 경우
= (동등) 조건으로 변형되어 검색된다.

```
SQL> SELECT * FROM emp WHERE ename LIKE '주종면';
--> SQL> SELECT * FROM emp WHERE ename = '주종면';
```

· BETWEEN ~ AND 조건은 > AND < 조건으로 변형되어 검색된다.

```
SQL> SELECT * FROM emp WHERE salary BETWEEN 100000 AND 200000;
--> SQL> SELECT * FROM emp WHERE salary >= 100000 and salary <= 200000;
```

· 인덱스가 생성되어 있는 컬럼의 IN 연산자의 조건은 OR 연산자의 조건으로 변형된다.

```
SQL> SELECT * FROM emp WHERE ename IN ('SMITH', 'KING');
--> SQL> SELECT * FROM emp WHERE ename = 'SMITH' or ename = 'KING';
```

· 인덱스가 생성되어 있는 컬럼의 OR 연산자의 조건은 UNION ALL로 변형된다.

```
SQL> SELECT * FROM emp WHERE ename = 'SMITH'  or  sal = 1000;
SQL> SELECT * FROM emp  WHERE ename = 'SMITH'
       UNION ALL
       SELECT * FROM emp  WHERE sal = 1000;
```

이와 같이 쿼리 변형기는 부적절하거나 잘못 작성된 SQL문장을 정확한 문장으로 변형시켜주는 역할을 수행하는 알고리즘이다.

4. 비용 계산기(Estimator)

비용 계산기는 비용기반 옵티마이저가 가지고 있는 비용 계산 공식에 의해 다양한 실행방법 중에 가장 좋은 성능의 실행계획을 찾아 주는 알고리즘이다.

1) 테이블과 인덱스의 통계정보

먼저, ANALYZE 명령어에 의해 수집되는 통계정보의 상태와 용어에 대해 설명하겠다. 먼저, 테이블에 대한 통계정보이다.

```
SQL> ANALYZE TABLE  big_emp  COMPUTE   STATISTICS;
SQL> ANALYZE TABLE  big_dept  COMPUTE   STATISTICS;

SQL> SELECT table_name, blocks, num_rows, avg_row_len
       FROM   user_tables
       WHERE  table_name =  'BIG_EMP'  or table_name =  'BIG_DEPT' ;
```

```
TABLE_NAME    BLOCKS    NUM_ROWS    AVG_ROW_LEN
-----------------------------------------------------------
BIG_DEPT           1         289             23
BIG_EMP          180       28955             43
```

NUM_ROWS : 해당 테이블의 전체 행수
AVG_ROW_LEN : 행 하나의 평균 길이

```
SQL> SELECT table_name, column_name, low_value, high_value, num_distinct
     FROM   user_tab_columns
     WHERE table_name = 'BIG_EMP' or table_name = 'BIG_DEPT' ;

TABLE_NAME  COLUMN_NAME LOW_VALUE        HIGH_VALUE NUM_DISTINCT
-----------------------------------------------------------------
BIG_EMP     EMPNO       C102             C3036464          28955
BIG_EMP     ENAME       4144414D53       57415244             14
BIG_EMP     JOB         414E414C595354   53414C45534D414E      8
BIG_EMP     MGR         C24C43           C25003                6
BIG_EMP     HIREDATE    77B7060D010101   78680604010101      713
BIG_EMP     SAL         80               C24E6233           3982
BIG_EMP     COMM        80               C20F                  5
BIG_EMP     DEPTNO      80               C164                 98
BIG_EMP     GROUPNO     31               32                    2
```

LOW_VALUE : 해당 컬럼에 저장되어 있는 가장 최소값에 대한 암호화 결과
HIGH_VALUE : 해당 컬럼에 저장되어 있는 가장 최대값에 대한 암호화 결과
NUM_DISTINCT : 해당 컬럼에 저장되어 있는 유일한 값의 개수

다음은 인덱스에 대한 통계정보이다.

```
SQL> CREATE  INDEX  i_big_emp_deptno ON big_emp (deptno);
SQL> ANALYZE INDEX  i_big_emp_deptno  COMPUTE STATISTICS;

SQL> SELECT   index_name              index_name,
              num_rows                num_rows,
              avg_leaf_blocks_per_key l_blocks,
              avg_data_blocks_per_key d_blocks,
              clustering_factor       cl_fac,
              blevel                  blevel,
              leaf_blocks             leaf
     FROM     user_indexes;

INDEX_NAME          NUM_ROWS L_BLOCKS D_BLOCKS CL_FAC BLEVEL LEAF
-----------------------------------------------------------------
I_BIG_EMP_DEPTNO    28853         1       51   5036      1   57
```

NUM_ROWS : 인덱스 행 수
L_BLOCKS : 하나의 LEAF 블록에 저장되어 있는 인덱스 키의 수
D_BLOCKS : 하나의 DATA 블록에 저장되어 있는 인덱스 키의 수
CL-FAC : CLUSTER FACTOR
BLEVEL : INDEX의 DEPTH
LEAF : LEAF 블록의 수

통계 정보는 오라클 10g 이전 버전까지는 사용자가 실행하는 ANALYZE 명령어에 의해 생성되었으며 10g 버전부터는 오라클 서버의 자동화된 알고리즘에 의해 자동 생성된다.
오라클 9i 버전 때까지는 사용자에 의해 통계정보를 생성해 주지 않으면 비용기반 옵티마이저는 부정확한 실행계획을 작성함으로써 성능이 저하되는 경우들이 많이 발생했었다.
〈그림 3〉은 통계정보가 생성되어 있지 않은 경우 비용기반 옵티마이저가 참조하는 통계정보의 기본 값이다. 데이터를 저장하고 있는 테이블과 인덱스의 실제 구조정보와 다른 값을 참조하기 때문에 결론적으로 좋은 실행계획을 작성하지 못하는 것이다.

통 계 정 보	기 본 값
Cardinality	Num_of_blocks*(block_size - cache_layer) / avg_row_len
Average Row Length	100 Bytes
Number of Blocks	Null
Levels	1
Leaf Blocks	25
Leaf Blocks / Key	1
Data Blocks / Key	1
Distinct Keys	100
Cluster Factor	800 (8 * number of blocks)

〈그림 3〉 통계정보의 기본 값

2) 용어에 대한 이해

```
SQL> SELECT * FROM  big_emp;

Execution Plan
----------------------------------------------------------------------
0    SELECT STATEMENT Optimizer=CHOOSE (Cost=19 Card=28955 Bytes=1042380)
1 0    TABLE ACCESS (FULL) OF 'BIG_EMP' (Cost=19 Card=28955 Bytes=1042380)
```

COST : SQL문을 실행하여 조건을 만족하는 행을 검색하는데 소요되는 횟수
CARDINALITY : 전체 테이블에서 SQL문의 조건을 만족하는 행 수

3) Cardinality

일반적으로 cardinlity는 SQL문이 실행되었을 때 조건을 만족하는 행수를 의미하는 것이긴 하지만 이것은 검색되는 컬럼이 어떤 속성을 가지고 있느냐에 따라 계산 공식이 달라진다.

3-1) Distinct Cardinality(Unique-Key)인 경우

이 경우는 주로 Full Table Scan과 같이 테이블 전체 행을 검색하는 경우의 cardinlity를 계산하는 공식이다.

```
SQL> SELECT count(*) FROM big_dept;
 count(*)
----------
       289
```

Cardinality = 조건을 만족하는 테이블의 행 수 = 289

```
SQL> ANALYZE  TABLE  big_dept  COMPUTE  STATISTICS;
SQL> SELECT  * FROM big_dept ;

Execution Plan
----------------------------------------------------------------------------
0     SELECT STATEMENT Optimizer=CHOOSE (Cost=1 Card=289 Bytes=5202)
1 0    TABLE ACCESS (FULL) OF 'BIG_DEPT' (Cost=1 Card=289 Bytes=5202)
```

3-2) Efficient Cardinality(Non-Unique-Key)인 경우

```
SQL> SELECT count(*) FROM big_dept;         -->  289 행
SQL> SELECT distinct loc FROM big_dept;     -->    7 행
```

Cardinality = 테이블의 전체 행수 / Distinct-Key 수 = 289 / 7
= 41

```
SQL> ANALYZE  TABLE  big_dept  COMPUTE  STATISTICS;
SQL> SELECT  * FROM big_dept  WHERE  loc =  'LA' ;

Execution Plan
----------------------------------------------------------------------------
0     SELECT STATEMENT Optimizer=CHOOSE (Cost=1 Card=41 Bytes=738)
1 0    TABLE ACCESS (BY INDEX ROWID) OF 'BIG_DEPT' (Cost=1 Card=41 Bytes=738)
```

3-3) Group Cardinality(Group by 절)인 경우

```
SQL> ANALYZE  TABLE   big_emp  COMPUTE  STATISTICS;
```

Cardinality = Distinct-Key 수 – 1

```
SQL> SELECT deptno, sum(sal) FROM big_emp   Group by deptno;   --> 99 행

Execution Plan
----------------------------------------------------------------------------
0     SELECT STATEMENT Optimizer=CHOOSE (Cost=131 Card=98 Bytes=588)
1 0    SORT (GROUP BY) (Cost=131 Card=98 Bytes=588)
2 1      TABLE ACCESS (FULL) OF 'BIG_EMP' (Cost=57 Card=28955 Bytes=173730)
```

4) Selectivity

선택도는 전체 테이블에서 SQL문의 조건을 만족하는 행이 분포되어 있는 비율을 의미하며 검색 되어지는 컬럼의 성격에 따라 계산 공식이 달라진다.

4-1) Unique-Key/Primary-Key의 경우

```
SELECT * FROM emp WHERE empno = 200;
--> Selectivity = 0.01                (좋은 선택도)
```

4-2) Non Unique-Key의 경우

```
SELECT * FROM emp WHERE ename =  'SMITH' ;
--> Selectivity = 1  / distinct-keys
--> 1/ 4   = 0.25
```

4-3) 값을 가진 비동등 조건식의 경우

```
SELECT * FROM emp WHERE empno < 200;
--> Selectivity = (범위값 – 최소값) / (최대값 – 최소값)
                = (200 - 1) / (29999 - 1)
                = 199 / 29998 = 0.007
```

```
SELECT * FROM emp WHERE empno > 200;
--> Selectivity = (범위값 – 최소값) / (최대값 – 최소값)
                = (29799 - 1) / (29999 - 1)
                = 29798 / 29998 = 0.9

SELECT * FROM emp WHERE empno BETWEEN 100 AND 200;
--> Selectivity = (최대 조건값 – 최소 조건값) / (최대값 – 최소값)
                = (200 - 100) / (29999 - 1)
                = 100 / 29998 = 0.003
```

4-4) 바인드 변수를 가진 비동등식의 경우

```
SELECT * FROM emp WHERE empno < :a ;
--> Selectivity = 0.25 %     (나쁜 선택도)

SELECT * FROM emp WHERE empno BETWEEN :a AND :b ;
--> Selectivity = 0.5 %      (나쁜 선택도)
```

5. 비용 계산 방법

지금까지 비용기반 옵티마이저가 비용을 계산하기 위해 알아야 할 여러 가지 내용에 대해 알아보았다. 그럼 지금부터는 다양한 SQL문의 비용 계산 공식에 대해 알아보자.

5-1) Full Table Scan인 경우

Cost = 전체 블록 수 / DB_FILE_MULTIBLOCK_READ_COUNT의 보정 값

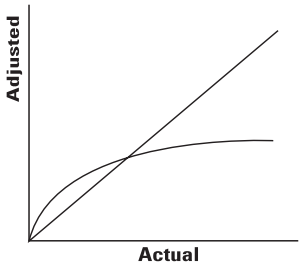
인덱스를 사용하지 않고 해당 테이블의 첫 번째 블록부터 전체 블록을 검색해야 하는 전체 테이블 스캔의 경우에는 init<SID>.ora 파일에 정의되어 있는 DB_FILE_MULTOBLOCK_READ_COUNT 파라메터 값에 의해 비용이 계산된다.
이 파라메터는 FULL TABLE SCAN의 경우 한번에 하나의 I-O로는 성능을 기대할 수 없기 때문에 보다 빠른 성능을 기대하기 위해 제공되는 다중 블록 읽기를 위한 파라메터이다. 즉, 한번 I-O에 8개, 16개, 32개, 64개의 다중 블록을 읽게 하기 위함이다.

```
SQL> SHOW  PARAMETER   db_file_multiblock_read_count
 NAME                                 TYPE             VALUE
------------------------------------------------------------------------------
db_file_multiblock_read_count         integer          16

SQL> SELECT * FROM  big_emp;

Execution Plan
------------------------------------------------------------------------------
0     SELECT STATEMENT Optimizer=CHOOSE (Cost=19 Card=28955 Bytes=1042380)
1 0     TABLE ACCESS (FULL) OF 'BIG_EMP' (Cost=19 Card=28955 Bytes=1042380)
```

앞에서 소개된 비용 계산 공식을 적용해보면 COST = 180 / 16 = 11.25 의 결과가 나와야 하는데 실제 비용은 COST=19의 결과가 계산되었다 !!
이것은 DB_FILE_MULTIBLOCK_READ_COUNT 파라메터의 실제 값처럼 한번 I-O에 8, 16, 32, 64개의 블록을 읽을 수는 없기 때문에 파라메터의 실제 값이 아닌 보정 값으로 비용을 계산했기 때문이다. 〈그림 4〉의 왼쪽 표는 ACTUAL (DB_FILE_MULTIBLOCK_READ_COUNT 파라메터 값)에 따른 Adjusted(보정 값)이며 〈그림 4〉의 오른쪽 그림은 이 파라메터가 실제로 성능에 영향을 미치게 되는 영향도를 그림으로 나타낸 것이다.

Actual	Adjusted
4	4.175
8	6.589
16	10.398
32	16.409
64	25.895
128	40.865

*** add 1 on v9.x**



〈그림 4〉 Full Table Scan의 비용계산 보정값

즉, COST = 19는 (180 / 10.398) + 1의 계산 공식 결과임을 알 수 있다.
사용자가 실행하는 SQL문의 실행계획이 FULL TABLE SCAN으로 결정되도록 유도하기 위해서는 이 파라메터 값을 조절하면 된다.

```
SQL> ALTER  SESSION SET   db_file_multiblock_read_count = 8;
SQL> SELECT * FROM  big_emp;

Execution Plan
------------------------------------------------------------------------------
0     SELECT STATEMENT Optimizer=CHOOSE (Cost=29 Card=28955 Bytes=1042380)
1 0     TABLE ACCESS (FULL) OF 'BIG_EMP' (Cost=29 Card=28955 Bytes=1042380)
```

위 SQL문의 비용은 Cost = (180 / 6.589) + 1 = 29 이다.
파라메터 값의 변경에 따라 비용이 달라지는 것을 확인할 수 있을 것이다.

```
SQL> ALTER  SESSION SET   db_file_multiblock_read_count = 32;
SQL> SELECT * FROM  big_emp;

Execution Plan
----------------------------------------------------------------------------
0     SELECT STATEMENT Optimizer=CHOOSE (Cost=13 Card=28955 Bytes=1042380)
1 0     TABLE ACCESS (FULL) OF 'BIG_EMP' (Cost=13 Card=28955 Bytes=1042380)
```

위 SQL문의 비용은 Cost = (180 / 16.409) + 1 = 13이다.

5-2) Unique Index Scan인 경우

Cost = blevel + 1

UNIQUE INDEX를 이용한 비용은 LEAF 블록의 DEPTH + 1 이 된다.

```
SQL> CREATE UNIQUE INDEX I_big_emp_empno ON  BIG_EMP (EMPNO);
SQL> ANALYZE INDEX I_big_emp_empno compute statistics;

SQL> SELECT INDEX_NAME, BLEVEL FROM USER_INDEXES
        WHERE INDEX_NAME = 'I_BIG_EMP_EMPNO';

INDEX_NAME                    BLEVEL
-----------------------------------------------
I_BIG_EMP_EMPNO                    1

SQL> SELECT  /*+index(big_emp I_big_emp_empno )*/   ename
                  FROM     big_emp
                  WHERE    empno = 7499;
----------------------------------------------------------------------------
0     SELECT STATEMENT Optimizer=CHOOSE (Cost=2 Card=1 Bytes=20)
1 0     TABLE ACCESS (BY INDEX ROWID) OF 'BIG_EMP' (Cost=2 Card=1 Bytes=20)
2 1       INDEX (UNIQUE SCAN) OF 'I_BIG_EMP_EMPNO' (UNIQUE) (Cost=1 Card=100)
```

위 SQL문의 비용은 Cost = 1 + 1 = 2이다.

5-3) Fast Full Index Scan인 경우

Cost = leaf_blocks / db_block_size

```
SQL> CREATE INDEX emp_job_deptno  ON  BIG_EMP (job, deptno);
SQL> ANALYZE INDEX emp_job_deptno compute statistics;
SQL> SELECT INDEX_NAME, LEAF_BLOCKS FROM USER_INDEXES
       WHERE INDEX_NAME = 'EMP_JOB_DEPTNO';

INDEX_NAME                    LEAF_BLOCKS
-----------------------------------------------
EMP_JOB_DEPTNO                        89

SQL> SHOW  PARAMETER  DB_BLOCK_SIZE
   NAME                        VALUES
  ----------------------------------------
   DB_BLOCK_SIZE                    8

SQL> SELECT /*+index_ffs(big_emp big_emp_job_deptno)*/   job, deptno
        FROM    big_emp
        WHERE deptno >= 1 and deptno <= 100
----------------------------------------------------------------------------
0     SELECT STATEMENT Optimizer=CHOOSE (Cost=10 Card=37 Bytes=703)
1 0     INDEX (FAST FULL SCAN) OF 'EMP_JOB_DEPTNO (Cost=10 Card=37 Bytes=703)
```

위 SQL문의 비용은 Cost = 89 / 8 = 10 이다.

5-4) Index Range Scan인 경우

Cost = blevel + (Selectivity X leaf_blocks) + (Selectivity X Cluster Factor)

SQL> ALTER SESSION SET optimizer_index_cost_adj = 100; SQL> ANALYZE TABLE big_emp COMPUTE STATISTICS;

SQL> ANALYZE INDEX i_big_emp_deptno COMPUTE STATISTICS;

선택도= 1/98 (Distinct한 deptno가 98가지 종류가 있을 때 USER_TAB_COLUMNS 참조)

Cluster-Factor= 5036 (USER_INDEXES 참조)

```
SQL> SELECT INDEX_NAME, BLEVEL, LEAF_BLOCKS FROM USER_INDEXES
        WHERE INDEX_NAME = 'I_BIG_EMP_DEPTNO'

INDEX_NAME            BLEVEL        LEAF_BLOCKS
----------------------------------------------------------------------
I_BIG_EMP_DEPTNO          1                57
```

```
SQL> SELECT /*+INDEX(BIG_EMP  i_big_emp_deptno)*/  *
        FROM BIG_EMP WHERE DEPTNO = 10 ;

Execution Plan
----------------------------------------------------------------------------
0     SELECT STATEMENT Optimizer=CHOOSE (Cost=53 Card=294 Bytes=10584)
1 0     TABLE ACCESS (BY INDEX ROWID) OF 'BIG_EMP' (Cost=53 Card=294 Bytes=10584)
2 1       INDEX (RANGE SCAN) OF 'I_BIG_EMP_DEPTNO' (NON-UNIQUE) (Cost=1 Card=294)
```

위 SQL문의 비용은 Cost = 1 + (1/98 * 57) +(1/98 * 5036) = 53이다.

```
SQL> ALTER  SESSION SET   optimizer_index_cost_adj = 50;
SQL> SELECT /*+INDEX(BIG_EMP)*/    *  FROM BIG_EMP WHERE DEPTNO = 10 ;

Execution Plan
----------------------------------------------------------------------------
0     SELECT STATEMENT Optimizer=CHOOSE (Cost=27 Card=294 Bytes=10584)
1 0     TABLE ACCESS (BY INDEX ROWID) OF 'BIG_EMP' (Cost=27 Card=294 Bytes=10584)
2 1       INDEX (RANGE SCAN) OF 'I_BIG_EMP_DEPTNO' (NON-UNIQUE) (Cost=1 Card=294)
```

위 SQL문의 비용은 Cost = 53 X 0.5 = 27이다.

```
SQL> ALTER  SESSION SET   optimizer_index_cost_adj = 150;
SQL> SELECT /*+INDEX(BIG_EMP)*/ *  FROM BIG_EMP WHERE DEPTNO = 10 ;

Execution Plan
----------------------------------------------------------------------------
0     SELECT STATEMENT Optimizer=CHOOSE (Cost=80 Card=294 Bytes=10584)
1 0     TABLE ACCESS (BY INDEX ROWID) OF 'BIG_EMP' (Cost=80 Card=294 Bytes=10584)
2 1       INDEX (RANGE SCAN) OF 'I_BIG_EMP_DEPTNO' (NON-UNIQUE) (Cost=1 Card=294)
```

위 SQL문의 비용은 Cost = 53 X 1.5 = 80이다.

5-5) Sort-Merge Join인 경우

다음은 소트-머지 조인의 경우 비용 계산 공식이다.

Cost = (Outer 테이블의 Sort Cost + Inner 테이블의 Sort Cost) - 1

```
SQL> SELECT  /*+use_merge(big_dept  big_emp)*/ *
            FROM   big_emp, big_dept
            WHERE BIG_EMP.DEPTNO = BIG_DEPT.DEPTNO;

Execution Plan
----------------------------------------------------------------------------
0     SELECT STATEMENT Optimizer=CHOOSE (Cost=452 Card=28853 Bytes=1558062)
1 0     MERGE JOIN (Cost=452 Card=28853 Bytes=1558062)
2 1       SORT (JOIN) (Cost=7 Card=289 Bytes=5202)
3 2         TABLE ACCESS (FULL) OF 'BIG_DEPT' (Cost=1 Card=289 Bytes=5202)
4 1       SORT (JOIN) (Cost=446 Card=28955 Bytes=1042380)
5 4         TABLE ACCESS (FULL) OF 'BIG_EMP' (Cost=57 Card=28955 Bytes= 1042380)
```

위 SQL문의 비용은 Cost = (outer-sort-cost + inner-sort-cost) - 1
= 7 + 446 - 1 = 452 이다.

5-6) Nest-Loop 조인의 경우

다음은 중첩루프 조인의 경우 비용 계산 공식이다.

Cost = Outer 테이블의 Cost + (Inner 테이블의 Cost * Outer 테이블의 Cardinality)

```
SQL> SELECT  /*+ use_nl(big_emp  big_dept)
                index(big_emp   I_big_emp_deptno)*/ *
      FROM   big_emp, big_dept
      WHERE  BIG_EMP.DEPTNO = BIG_DEPT.DEPTNO;

Execution Plan
----------------------------------------------------------------------------
0     SELECT STATEMENT Optimizer=CHOOSE (Cost=18786 Card=28853 Bytes= 1558062)
1 0     TABLE ACCESS (BY INDEX ROWID) OF 'BIG_EMP' (Cost=65 Card=28955 Bytes=1042380)
2 1       NESTED LOOPS (Cost=18786 Card=28853 Bytes=1558062)
3 2         TABLE ACCESS (FULL) OF 'BIG_DEPT' (Cost=1 Card=289 Bytes=5202) <-- OUTER 테이블
4 2         INDEX (RANGE SCAN) OF 'I_BIG_EMP_DEPTNO' (NON-UNIQUE) (Cost=1 Card=28955)
```

위 SQL문의 비용은 Cost = outer-cost + (inner-cost * outer card)
= 1 + (65 * 289)
= 1 + 18785
= 18786

```
SQL> CREATE  INDEX  i_big_dept_deptno ON big_dept (deptno);
SQL> ANALYZE TABLE big_dept COMPUTE STATISTICS;
SQL> SELECT  /*+use_nl(big_emp  big_dept)
                              index(big_dept  I_big_dept_deptno)
                              ordered*/ *
       FROM   big_emp, big_dept
       WHERE BIG_EMP.DEPTNO = BIG_DEPT.DEPTNO;

Execution Plan
----------------------------------------------------------------------------
0     SELECT STATEMENT Optimizer=CHOOSE (Cost=57967 Card=28853 Bytes= 1558062)
1 0    TABLE ACCESS (BY INDEX ROWID) OF 'BIG_DEPT' (Cost=2 Card=289 Bytes= 5202)
2 1      NESTED LOOPS (Cost=57967 Card=28853 Bytes=1558062)
3 2        TABLE ACCESS (FULL) OF 'BIG_EMP'(Cost=57 Card=28955 Bytes=1042380) <--OUTER 테이블
4 2        INDEX (RANGE SCAN) OF 'I_BIG_DEPT_DEPTNO' (NON-UNIQUE) (Cost=1 Card=289)
```

위 SQL문의 비용은 Cost = outer-cost + (inner-cost * outer card)

= 57 + (2 * 28955)

= 57 + 57910

= 57967

5-7) Hash Join인 경우

다음은 해시 조인의 경우 비용 계산 공식이다.

Cost = (Outer 테이블의 Cost x #Hash 파티션수 + Inner 테이블의 Cost) + 2

```
SQL> SELECT /*+hash(big_emp)*/ *
       FROM BIG_EMP,  BIG_DEPT
       WHERE BIG_EMP.DEPTNO = BIG_DEPT.DEPTNO;

Execution Plan
----------------------------------------------------------------------------
0     SELECT STATEMENT Optimizer=CHOOSE (Cost=60 Card=28853 Bytes=1558062)
1 0    HASH JOIN (Cost=60 Card=28853 Bytes=1558062)
2 1      TABLE ACCESS (FULL) OF 'BIG_DEPT' (Cost=1 Card=289 Bytes=5202)
3 1      TABLE ACCESS (FULL) OF 'BIG_EMP' (Cost=57 Card=28955 Bytes= 1042380)
```

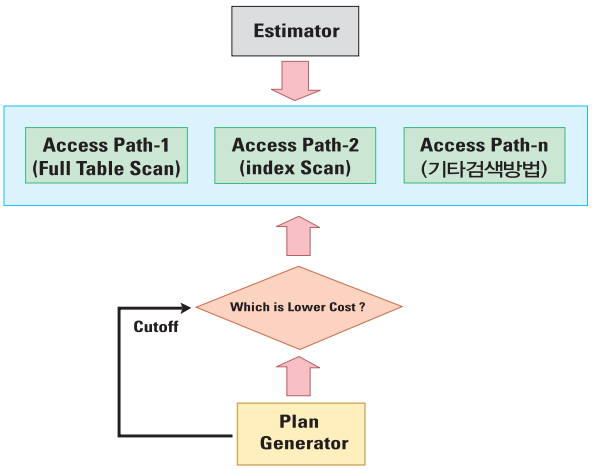
위 SQL문의 비용은 Cost = outer-cost + inner-cost + Sort Cost + 2

= 1 + 57 + 2

= 60

6. 실행계획 생성기(Plan Generator)

사용자가 실행한 SQL문은 쿼리 변형기와 비용 계산기에 의해 여러 가지 유형의 실행계획으로 비용 분석된다. 그 중에 가장 적은 비용으로 실행되어질 수 있는 실행계획 하나가 선택되는데 이것을 Optimal Plan이라고 한다.



〈그림 5〉 Plan Generator

다음 문장들은 동일한 결과를 제공하지만 실행계획 생성기에 의해 가장 적은 비용의 실행계획을 선택한 결과이다.

6-1) 적정 플랜(Optimal Plan)

Index Scan인 경우

```
SELECT  ename
FROM    big_emp
WHERE   deptno = 20 AND   empno BETWEEN 100 AND 200
ORDER BY ename;

Execution Plan
----------------------------------------------------------------------------
0     SELECT STATEMENT Optimizer=CHOOSE (Cost=9 Card=1 Bytes=12)
1 0    SORT (ORDER BY) (Cost=9 Card=1 Bytes=12)
2 1      TABLE ACCESS (BY INDEX ROWID) OF 'BIG_EMP' (Cost=5 Card=1 Bytes=12)
3 2        INDEX (RANGE SCAN) OF 'I_BIG_EMP_EMPNO' (UNIQUE) (Cost=2 Card=1)
```

이 실행계획은 비용기반 옵티마이저에 의해 I_BIG_EMP_EMPNO 인덱스가 선택되었으며 이때 계산된 I-O COST는 9이다.

I_BIG_EMP_DEPTNO 인덱스가 선택된 경우

```
SELECT  /*+index(big_emp I_BIG_EMP_DEPTNO)*/    ename
FROM    big_emp
WHERE   deptno = 20 AND   empno BETWEEN 100 AND 200
ORDER BY ename;

Execution Plan
----------------------------------------------------------------------------
0    SELECT STATEMENT Optimizer=CHOOSE (Cost=70 Card=1 Bytes=12)
1 0    SORT (ORDER BY) (Cost=70 Card=1 Bytes=12)
2 1      TABLE ACCESS (BY INDEX ROWID) OF 'BIG_EMP' (Cost=66 Card=1 Bytes=12)
3 2        INDEX (RANGE SCAN) OF 'I_BIG_EMP_DEPTNO' (NON-UNIQUE) (Cost=2 Card=1)
```

이 실행계획은 I_BIG_EMP_DEPTNO 인덱스가 선택되었으며 이때 계산된 I-O COST는 70이다.

Full Table Scan인 경우

```
SELECT  /*+full(big_emp)*/ ename
FROM    big_emp
WHERE   deptno = 20 AND   empno BETWEEN 100 AND 200
ORDER BY ename;

Execution Plan
----------------------------------------------------------------------------
0    SELECT STATEMENT Optimizer=CHOOSE (Cost=61 Card=1 Bytes=12)
1 0    SORT (ORDER BY) (Cost=61 Card=1 Bytes=12)
2 1      TABLE ACCESS (FULL) OF 'BIG_EMP' (Cost=57 Card=1 Bytes=12)
```

이 실행계획은 Full Table Scan이 선택되었으며 이때 계산된 IO COST는 61이다. 결론적으로, 5-1, 5-2, 5-3의 SQL문장들은 동일한 문장, 동일한 결과를 제공하지만 이 문장이 실행될 수 있는 실행계획은 다양하다는 것을 알 수 있다.

이와 같이, 비용기반 옵티마이저는 여러 가지 실행계획 중에 가장 비용이 적게 발생하는 I_BIG_EMP_EMPNO 인덱스를 이용한 실행계획을 Optimal Plan으로 선택하게 된다.

6-2) 비용계산 분석 명령어

다음 문장은 비용분석기(Estimator)와 실행계획 생성기(Plan Generator)에 의해 비용 분석된 결과를 모니터링하는 방법이다.

```
SQL> ALTER SESSION SET EVENTS '10053 trace name context forever, level 1';
SQL>  SELECT *
        FROM BIG_EMP, BIG_DEPT, ACCOUNT
        WHERE BIG_EMP.DEPTNO = BIG_DEPT.DEPTNO
        AND ACCOUNT.CUSTOMER =  BIG_EMP.EMPNO
        SQL> EXIT

[C:\] CD  C:\ORACLE\ADMIN\ORA92\UDUMP
[C:\] DIR

ORA92_ORA_xxxx.trc      <-- 워드패드 편집기를 통해 결과 확인
```

〈분석결과〉

*** 2005-07-17 10:41:27.000
*** SESSION ID:(10.976) 2005-07-17 10:41:27.000
QUERY
SELECT * FROM BIG_EMP, BIG_DEPT, ACCOUNT
WHERE BIG_EMP.DEPTNO = BIG_DEPT.DEPTNO AND ACCOUNT.CUSTOMER = BIG_EMP.EMPNO

****
GENERAL PLANS
****
Join order[1]: BIG_DEPT [BIG_DEPT] BIG_EMP [BIG_EMP] ACCOUNT [ACCOUNT]
Now joining: BIG_EMP [BIG_EMP] *******
NL Join
Outer table: cost: 2 cdn: 289 rcz: 19 resp: 2
Inner table: BIG_EMP
Access path: tsc Resc: 19
Join: Resc: 5493 Resp: 5493
Join cardinality: 28853 = outer (289) * inner (28955) * sel (3.4480e-003) [flag=0]
Best NL cost: 5493 resp: 5493
SM Join
Outer table:
resc: 2 cdn: 289 rcz: 19 deg: 1 resp: 2
Inner table: BIG_EMP
Best SM cost : 257

****
Join order[2]: BIG_DEPT [BIG_DEPT] ACCOUNT [ACCOUNT] BIG_EMP [BIG_EMP]
Now joining: ACCOUNT [ACCOUNT] *******
****
Join order[3]: BIG_EMP [BIG_EMP] BIG_DEPT [BIG_DEPT] ACCOUNT [ACCOUNT]
Now joining: BIG_DEPT [BIG_DEPT] *******

****
Join order[4]: BIG_EMP [BIG_EMP] ACCOUNT [ACCOUNT] BIG_DEPT [BIG_DEPT]
Now joining: ACCOUNT [ACCOUNT] *******

****
Join order[5]: ACCOUNT [ACCOUNT] BIG_DEPT [BIG_DEPT] BIG_EMP [BIG_EMP]
Now joining: BIG_DEPT [BIG_DEPT] *******

****
Join order[6]: ACCOUNT [ACCOUNT] BIG_EMP [BIG_EMP] BIG_DEPT [BIG_DEPT]
Now joining: BIG_EMP [BIG_EMP] *******

〈분석결과 평가〉

분석된 결과 중에 Join order[n]는 여러 개의 테이블을 조인하는 경우 어떤 테이블부터 검색하여 어떤 순서에 의해 조인해 나가는 방법인지 분석하는 경우를 나타낸다.
Join order[1]에서 Best NL cost: 5493은 중첩루프 조인의 비용 결과이며, Best SM cost : 257은 소트-머지 조인의 경우 비용 결과이다.
비용기반 옵티마이저는 하나의 조인순서가 결정되면 다양한 실행 방법들에 대한 비용을 일일이 계산하게 된다. 그 중에 가장 적은 비용이 발생하는 조인 순서와 실행 방법을 실행 계획으로 선택하게 되는 것이다.