



03. Execution Plan (실행계획)

● 1. Execution Plan

■ 실행계획을 읽어야 하는 이유

사전	① 낙관하다 ② <u>가장 능률적으로 활용하다</u>
IT 용어	Optimize + er(행위자) 붙여 <u>최적화기</u> , 성능 향상기 라는 의미로 사용
RDBMS	Query + Optimizer 를 합성하여 <u>질의 최적화기</u> 라는 의미로 사용

■ 탄생

RDBMS의 등장과 함께 나타난 Query Optimizer는 1970년대에 IBM 연구소에서 최초로 제안하였으며, 오늘날 모든 RDBMS에 채택되어 사용되고 있다.

■ 비절차적 언어 SQL 와 Optimizer

- SQL 개발자(사용자)가 **처리 절차(처리 순서) 와 처리방법**을 지정하지 않고 원하는 결과를 요청하는 언어
- 절차적인 언어는 개발자(사용자)가 처리방법, 처리순서, 효율적 방안을 직접 기술(프로그래밍)
- 비절차적 언어에서는 사용자가 얻고자 하는 요구사항을 SQL로 표현 하기만 하면 RDBMS내의 Query Optimizer가 효율적인 처리 방법,처리순서를 찾아내고 서버프로세스가 수행합니다.
- C, Java 같은 절차적인 언어에서는 개발자가 고민해야 하는 수행 알고리즘과 효율성을 비절차적인 SQL에서는 RDBMS내의 Query Optimizer(질의 최적화기)가 대신 수행

● 1. Execution Plan

■ 실행계획을 읽어야 하는 이유

역사 드라마를 상상 해보자

왕이 명령을 내리면 책사가 전쟁 계획을 수립하고 장군은 전쟁을 수행한다. 올바른 계획을 수립하면 승리 하지만 책사가 잘못된 판단을 내리면 장군은 전쟁에서 패한다. 책사가 수립한 계획의 중요성을 보여 준다.

비유	실제
왕	사용자 or 어플리케이션(Application Program)
명령	SQL
책사	옵티마이저 (Optimizer)
전쟁 수행 계획	실행계획 (Execution plan)
장군	서버 프로세스 (Server Process)

위의 비유표를 보면 SQL을 사용하는 여러분은 DBMS에게 명령을 내리는 왕의 역할을 한다.

왜? 왕은 책사가 수립한 전쟁 수행 계획을 읽을수 있어야 하고 읽어야 할까?

● 1. Execution Plan

■ 실행계획을 읽어야 하는 이유

책사(Optimizer)는 일반적으로 효율적인 계획을 수립하지만 그가 만든 모든 계획이 완전하지는 않기 때문이다. 계획을 수립하기 위해 필수적으로 필요한 정보의 부실, 주변 정황에 대한 잘못된 이해, 왕의 잘못된 명령등의 여러가지 이유로 오판 할수 있다.

역사를 보면 책사에게 모든 판단을 위임하고 명령만 내리는 왕은 쉽게 몰락 한다. 1번의 판단 실수가 왕권의 몰락의 원인이 되기도 한다. 현명한 왕은 책사가 수립한 계획을 이해하고 판단하여 책사의 실수를 지적 하고 새로운 계획을 수립하도록 한다.

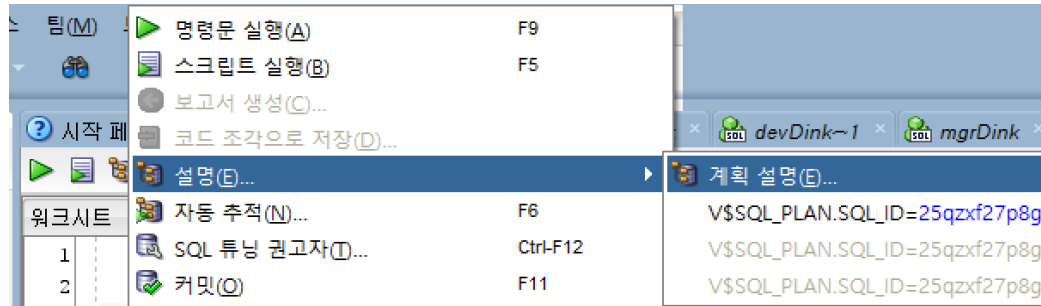
비유	실제
정보 부실	Object 통계정보(데이터 디렉터리에 저장된)
주변 정황	Index 부재, 잘못된 설계, 데이터 저장정책, 시스템 자원 부족등
잘못된 명령	비효율적인 SQL

현실에서도 단 1개의 악성 SQL에 의해 DBMS 전체에 성능 장애가 발생하여 서비스가 원활히 수행되지 않는 사례가 발생한다. DBMS 왕국을 무너지게 하지 않도록 개발자는 SQL을 작성한후 실행계획을 확인하는 과정을 가져야 한다. 수행계획을 검토하고 이해하여 책사가 오판 했다면 원인을 찾아 스스로 해결 하거나 DBA 또는 DA(Data Architect)에게 도움을 요청하도록 해야 한다.

● 1. Execution Plan

■ SQLDEV에서 실행계획 조회

- ① `SELECT * FROM EMP WHERE SAL = '3000';`
- ② SQL 선택 > 마우스 오른쪽 버튼 > 설명(E) > 계획설명 (Explain execution plan)



- ③ [에러 발생시] 권한이 없는 경우 DBA 계정으로 SQL 워크시트 생성

- 도구(T) > SQL 워크시트 > mgrDinkDBMS 클릭
- `grant select_any_catalog to scott;`
- 도구(T) > SQL 워크시트 > devDinkDBMS 클릭

권한부여

신규 session 생성부터 적용

④

OPERATION	OBJECT_NAME	CARDINALITY	COST
SELECT STATEMENT		2	3
TABLE ACCESS (FULL)	EMP	2	3
Filter Predicates SAL=3000			

● 1. Execution Plan

■ 실행계획 해석

SQL

```
SQL> SELECT d.dname,count(e.empno) as cnt,avg(e.sal) as avg_sal
      FROM emp e, dept d
      WHERE e.deptno = d.deptno and
            e.job != 'PRESIDENT'
      GROUP BY d.dname;
```

// 1회 실행후 , 실행계획 조회

결과

SQL 0초			
OPERATION	OBJECT_NAME	CARDINALITY	COST
SELECT STATEMENT		1	7
HASH (GROUP BY)		1	7
HASH JOIN		14	6
Access Predicates			
E,DEPTNO=D,DEPTNO			
TABLE ACCESS (FULL)	DEPT	7	3
TABLE ACCESS (FULL)	EMP	14	3
Filter Predicates			
E,JOB<>'PRESIDENT'			
Other XML			
{info}			
info type="has_user_tab"			
info type="db_version"			
19.0.0.0			
info type="parse_schema"			
"SCOTT"			
info type="plan_hash_full"			
info type="plan_hash"			
2006461124			

● 1. Execution Plan

■ 실행계획 해석

① Plan hash value는 SQL 튜닝시 유용한 정보를 제공한다.

AUTOTRACE 나 Explain plan을 통해서 나타나는 실행계획은 예상 실행계획(Estimated Execution Plan) 이고 SQL Trace를 통해서 실제 실행계획(Actual Execution Plan)이 출력 된다. 이로 인해 SQLDEV에서 튜닝을 수행 할때는 효율적인 실행계획으로 판단되었으나 어플리케이션 프로그램안에서 사용될때는 느린 경우가 발생할수 있다. 여러가지 이유에 의해서 예상 실행계획과 실제 실행계획이 다를수 있다.

실제 실행계획(Actual Execution Plan)은 SQL 실행시 실제로 사용된 실행 계획이고 예상 실행계획(Estimated Execution Plan)은 튜닝툴로 실행계획을 조회할때 보여지는 실행계획이다. 9i 이전 버전에서는 실제 실행계획을 조회하려면 SQL Trace를 사용하는게 유일한 방법이었다. 10g 버전부터 실제 사용된 실행 계획을 SGA내의 Shared Pool 영역에 캐싱(Caching)하고 **V\$SQL_PLAN 을 통해서 조회가 가능** 하다.

일반 개발자 계정에서는 V\$SQL_PLAN을 조회하는 권한이 없는 경우가 대부분이어서 DBA에게 권한을 요청을 해야 하지만 SQL Developer에서 자동 추적 기능을 사용하기 위해 이전 실습에서 SELECT ANY DICTIONARY 권한을 SCOTT 계정에 부여해서V\$SQL_PLAN을 조회 할수 있다.

```
SELECT id,parent_id, lpad(' ',depth*5)||operation||' '||options as Operation,
       object_name,cardinality,bytes,cost,time
FROM   v$sql_plan
WHERE  plan_hash_value = 2006461124;
```

● 1. Execution Plan

■ 실행계획 해석

ID	PARENT_ID	OPERATION	OBJECT_NAME	CARDINALITY	BYTES	COST	TIME
1	0	SELECT STATEMENT				7	
2	1	0 HASH GROUP BY		2	56	7	1
3	2	1 HASH JOIN		13	364	6	1
4	3	2 TABLE ACCESS FULL	DEPT	4	52	3	1
5	4	2 TABLE ACCESS FULL	EMP	13	195	3	1

실습 환경(ex DBMS버전) 마다 plan_hash_value 값은 다르게 나타날수 있다. 실행계획 출력시 나타나는 값을 위의 예제에 적용해서 사용한다. 위의 결과에 따르면 예상 실행계획과 실제 실행 계획이 동일 하다.

- ② V\$SQL_PLAN에서는 PARENT_ID가 표현되지만 AUTOTRACE에서 출력되는 실행계획에는 PARENT_ID가 나타나지 않는다. PARENT_ID가 나타나지 않지만 Operation의 인덴테이션(들여쓰기,depth)를 보면 PARENT_ID를 유추 할수 있다.

ID는 각 Operation에 부여된 일련번호이고 PARENT_ID는 부모 Operation의 ID이다.
TABLE ACCESS FULL DEPT, EMP Operation의 실행 결과가 부모 Operation을 수행하는 입력이 된다.
ID 와 PARENT_ID를 해석하면 실행계획의 실행 순서를 알수 있다.
Id 2번,4번에 * 표시 와 Predicate Information를 연결해서 해석해야 한다. Predicate Information은 where절의 조건문을 의미한다.

Id * 2 와 2 - access("E"."DEPTNO"="D"."DEPTNO")를 해석하면 조인 처리 알고리즘에 HASH JOIN이 사용되었으며 테이블간 연결 고리(조건)에 DEPTNO 컬럼을 사용 한다.

● 1. Execution Plan

■ 실행계획 해석

Id * 4 와 4 – filter("E"."JOB" <> 'PRESIDENT')를 해석하면 EMP를 Full Table Scan을 하여 데이터에 접근(ACCESS) 하고 JOB <> 'PRESIDENT'를 원하지 않는 데이터를 필터링(FILTER)하는 조건으로 사용한다.

SQL 튜닝시 WHERE에 사용되는 여러 조건문중 어떤 조건문이 access 로 사용되고 어떤 조건이 filter로 사용되는지 구분 해야 한다. access는 대상 데이터에 어떻게 접근 하는지를 나타내는 방법으로 주고 접근 경로(Access Path)라고 표현 한다.

접근 경로(Access Path)는 SQL 실행성능을 결정하는 가장 중요한 요소중 하나이다.SQL 튜닝시 어떤 조건이 Access Path를 결정하는지 결정된 Access Path가 효율적인지를 판단해야 한다.

③ 상위의 Operation에서 보여지는 수치는 하위 단계 Operation을 수행하기 위한 예상 비용(Cost),예상처리시간(Time), 예상되는 Row 개수, 예상되는 결과 Bytes 의 누적된 합이다. 위의 실행계획은 실제 실행계획이 아닌 실행하기전에 사전에 예측되는 실행계획으로 표시되는 값역시 실제 처리된 결과값 아닌 예측값이다.

● 1. Execution Plan

■ 실행계획 해석

Rows or Cadinality	<p>각 실행계획 연산별(단계별) 예상 출력 Row 개수 이다. TABLE ACCESS FULL DEPT 4 는 DEPT 테이블 전체를 읽어서 4건의 Row를 결과 집합으로 출력 한다.</p> <p>TABLE ACCESS FULL EMP 13 는 EMP 테이블 전체를 읽은후 JOB <>'PRESIDENT'조건에 의해서 1건 Row가 필터(Filter) 된후 13건의 Row를 결과 집합으로 출력 한다.</p> <p>HASH JOIN 결과 13건의 Row가 출력된다. HASH GROUP BY 2 는 GROUP BY d.dname 조건을 수행하기 위해 HASH를 사용하여 GROUP BY을 수행하고 그룹핑이 된 결과 2건 Row를 출력한다.</p>
Bytes	<p>예상 결과 Rows의 Bytes수</p>
Cost	<p>각 Operation을 실행하는데 소요되는 예상 비용이다.</p> <p>최상위 단계인 SELECT STATEMENT에서 Cost는 전체 처리 비용이 7 이라는 의미이다. SQL 처리 비용의 85%(6/7)는 HASH JOIN 연산에 사용되었다.</p> <p>HASH JOIN의 비용은 2개의 테이블 각각을 FULL TABLE SCAN 하는데 사용 되었다.</p> <p>전체 총 Cost 와 각 Operation별 단위 Cost를 비교해보면 SQL 실행을 위해 많은 자원을 사용하는 부분이 어디인지를 쉽게 알수 있다.</p> <p>(% CPU) 9i 버전은 I/O 횟수, I/O량을 기준으로 비용(Cost)을 계산하고 선택적(시스템 통계가 생성되어 있는 경우)으로 CPU,IO 성능을 비용(Cost)에 포함 시켰다..</p>

● 1. Execution Plan

■ 실행계획 해석

Cost	10g버전 부터는 CPU 사용량을 필수적으로 비용계산에 포함
Time	각 Operation이 실행시 예측되는 경과시간(Elapsed Time) 을 초단위로 출력 한다. 전체 처리를 위해서 1초가 예상 된다. 나머지연산도 전부 1초로 표현되는데 초단위의 시간을 출력하기 위해 1초 이하의 단위를 초로 환산 하면서 생긴 오차이다. 시간이 오래걸리는 실행계획을 표현하는 경우 정상적인 예측 시간이 표시 된다. 각종 통계(Object 통계, System 통계)를 기반으로 예측된 시간이므로 실제 처리와 일치하지 않는 경우가 많기에 사용자 측면의 응답시간 개념으로 접근하는게 아니라 전체 처리시간중에 특정 Operation이 차지하는 비율측면으로 접근하면 SQL 처리중 병목지점이 어디인지 쉽게 찾을수 있다.

● 1. Execution Plan

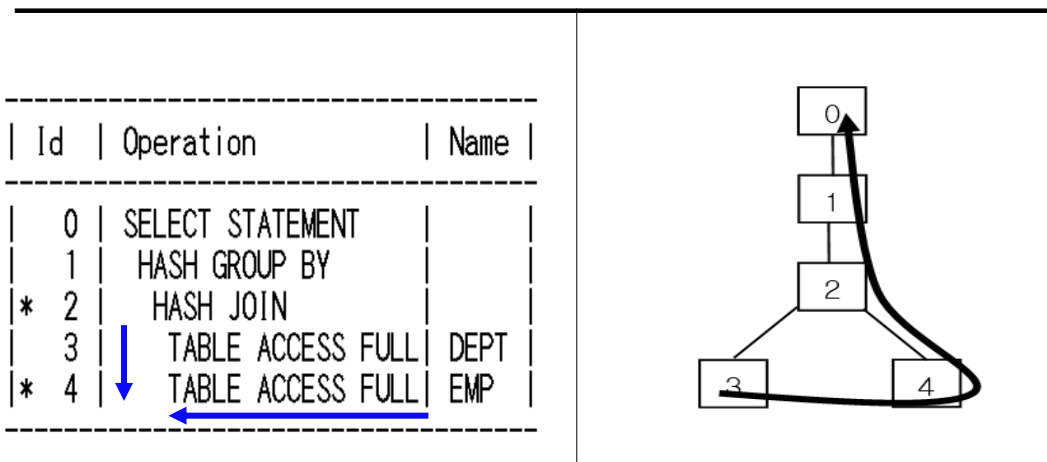
■ 실행계획 해석

실행계획은 계층적 트리 구조로 구성되며 실행계획의 순서는 하위 레벨의 Operation이 먼저 실행되고 동일한 레벨이면 위에(위치상으로 윗쪽) 있는 Operation이 먼저 실행 된다.

첫 번째 [트리]

튜닝 학습하는 과정에서는 실행계획의 각 Operation을 트리 구조로 그림을 그리면 이해하기 쉽다.

실행계획의 구조가 트리구조로 정식적인 읽는 방법



두 번째 [안위]

직관적으로 읽는 방법으로 인덴테이션(Indentation) 가장 깊은(가장 안쪽에) Operation이 먼저 실행되고 동일한 인덴테이션 레벨은 위쪽 Operation이 먼저 실행.

안쪽에서 밖으로 위에서 아래로 읽는다.

● 1. Execution Plan

■ SQLDEV 에서 autotrace (explain execution plan + Statistics)

SQL Developer는 설명(Explain Execution Plan) 과 자동추적(Autotrace) 을 사용하여 실행 계획을 조회할수 있다. 설명 보다는 자동추적이 더 많은 분석 정보를 제공 한다.

자동추적 기능을 사용하여 실습해보자, 아래의 SQL을 2번 실행한후 각각 응답시간을 비교한후 확인한후 자동 추적 실습(응답시간이 각각 다른 이유는?)

SQL	<pre>SELECT E.EMPNO,C.GENDER,COUNT(C.ID) AS CNT_CUSTOMER FROM EMP E, CUSTOMER C WHERE E.EMPNO = C.ACCOUNT_MGR AND E.JOB != 'PRESIDENT' GROUP BY E.EMPNO,C.GENDER;</pre>
-----	---

인출된 모든 행: 26(3.902초)			
	EMPNO	GENDER	CNT_CUSTOMER
1	7499	M	404288
2	7566	F	223022
3	7900	F	222004
4	7499	F	443241
5	7782	M	201719

해당 SQL을 선택한후 F6키를 누르거나 마우스 오른쪽 버튼을 클릭하여 자동 추적을 실행한다

```
SELECT  E.EMPNO,C.GENDER,COUNT(C.ID) AS CNT_CUSTOMER
FROM    EMP E, CUSTOMER C
WHERE   E.EMPNO = C.ACCOUNT_MGR AND
        E.JOB != 'PRESIDENT'
GROUP BY E.EMPNO,C.GENDER;
```

- 명령문 실행(A) F9
- 스크립트 실행(B) F5
- 보고서 생성(C)...
- 코드 조각으로 저장(D)...
- 설명(E)
- 자동 추적(N)...** F6

[에러 발생시] 권한이 없는 경우 DBA 계정으로 SQL 워크시트 생성

① grant select_any_catalog to scott; // granted Role은 신규 connection 부터 적용

② grant select on sys.v_\$session to demo;

grant select on sys.v_\$mystat to demo;

grant select on sys.v_\$statname to demo;

* system 계정에 권한이 없는경우 sys 계정에서 각각의 v_\$에 대한 권한을 system계정에게 부여한후 재실행

grant select on v_\$session to system with grant option;

● 1. Execution Plan

■ SQLDEV 에서 autotrace 사용

결과

SQL 핫스팟 12,015초						
OPERATION	OBJECT_NAME	CARDINALITY	COST	LAST_CR_BUFFER_GETS	LAST_ELAPSED_TIME	LAST_OUTPUT_ROWS
SELECT STATEMENT			34368			
HASH (GROUP BY)		19	34368	124816	11605902	26
MERGE JOIN		19	34367	124816	11605810	26
TABLE ACCESS (BY INDEX ROWIDEMP)		13	2	2	28	13
Filter Predicates						
INDEX (FULL SCAN)	EMP_EMPNO_PK	14	1	1	9	14
SORT (JOIN)		19	34365	124814	11605773	26
Access Predicates						
Filter Predicates						
VIEW	VW_GBF_7	19	34364	124814	11605740	26
HASH (GROUP BY)		19	34364	124814	11605736	26
TABLE ACCESS (FULL) CUSTOMER		5350128	34102	124814	10674934	5652476

- ① SQL을 클릭하면 실행계획과 관련된 SQL을 출력해 보여준다. 12.015초는 SQL을 실행한후 결과를 보여주는데 소요된 응답시간 이다. **핫스팟은 실행계획 분석시 요긴하게 사용하는 유틸리티**이다. 여러번 클릭 해보자.
- 성능 문제가 발생하는 부분을 찾는 쉬운 방법은 Cost가 많이 사용된 Operation을 우선 분석하는 것이다.
- 핫스팟을 통해 이를 쉽게 찾을 수 있다. 위의 SQL 실행에 사용되는 **전체 Cost 34,368**(가장 상위 Operation에 표시되는 비용이 각 단위 Operation이 누적된 최종값)중 핫스팟으로 표시된 TABLE ACCESS (FULL) CUSTOMER Operation에서 Cost 34,102이 사용되었다. **전체 비용의 99%(34,102/34,368) 가 CUSTOMER 테이블 FULL SCAN 하는데 사용되었다. 튜닝시 주요 고려 대상**이다.

VIEW	VW_GBF_7	19	34364	124814	11605740	26
HASH (GROUP BY)		19	34364	124814	11605736	26
TABLE ACCESS (FULL) CUSTOMER		5350128	34102	124814	10674934	5652476

● 1. Execution Plan

■ SQLDEV 에서 autotrace 사용

SQL의 **응답시간(Response Time)**은 **Response time = Elapsed time + Wait time** 이다.

응답시간은 사용자 관점에서 사용자가 원하는 결과를 얻는데 까지 걸린 시간이고 경과시간(Elapsed time)은 DBMS내부에서 SQL을 처리하는데 사용된 시간이며 대기시간(Wait time)은DBMS내부 공유 자원에 대한 대기 시간이다.

실행계획에 표시되는 시간은 **마이크로세컨드(microsecond,µs,백만분의 일초)**로 초단위로 변환하여 계산하면 아래와 같다.

응답시간 (Response time)	12.05초
경과시간 (Elapsed time)	11.60초 (소수점 2자리 절삭)
대기시간 (Wait time)	0.45초

전체 경과시간 11.60초의 92%(10.67/11.60)가 CUSTOMER에 대한 FULL SCAN에 사용되었다.

HASH(GROUP BY) 연산에 표시된 핫스팟 지점에 마우스 커서를 위치 하면 아래와 같은 정보를 표시한다.

VIEW	VW_GBF_7	19	34364	124814	11605740	26
HASH (GROUP BY)		19	34364	124814	11605736	26
TABLE ACCESS (FULL) CUSTOMER		5350128	34102	124814	10674934	5652476

개별 노드 런타임=930802
LAST_ELAPSED_TIME=11605736

CUSTOMER에 대한 FULL SCAN에 10.674934초가 소요 되었고 HASH(GROUP BY) 연산까지 누적된 전체 경과시간은 11.605736초 이고 HASH(GROUP BY) 연산 수행시간은 0.930802 초가 소요 되었다.

● 1. Execution Plan

■ SQLDEV 에서 autotrace 사용

② 자동추적 결과의 각 항목은 아래와 같다. * 도구 > 환경설정 > 데이터베이스 > 유틸리티 > 자동추적/계획 > 추가

OPERATION	Row Source Operation
OBJECT_NAME	Operation 수행 대상이 되는 테이블 ,인덱스의 이름
CARDINALITY	Operation 실행시 생성될거라 예측(예상)되는 Row 개수
COST	Operation의 수행 비용
LAST_CR_BUFFER_GETS	현재 실행된 Operation이 Data Buffer Cache내에서 CR(Consistent Read) 모드로 읽은 Block의 개수 (SELECT시 Block를 CR모드로 읽는다)
LAST_ELAPSED_TIME	현재 실행된 Operation의 경과 시간 (microsecond , 백만분의 1초단위)
LAST_OUTPUT_ROWS	현재 실행된 Operation에 의해 생성된 Row 개수

<참고> 실행계획 분석시 **CARDINALITY**(예상 되는 Row 개수) 와 **LAST_OUTPUT_ROWS**(실제 생성된 Row 개수)의 차이가 크게 발생하는 이유는 오브젝트에 대한 통계정보가 정확하지 않기 때문이다.

CARDINALITY는 실행계획 수립시 데이터 디크셔너리에 저장되어 있는 **오브젝트 통계정보**를 기반으로 예측 하기에 정확하지 않는 통계정보는 비효율적인 실행계획을 생성하는 주요 원인이 된다.

통계정보와 실제 데이터를 비교한후 통계정보가 정확하지 않는 경우 DBA에게 통계정보 재생성 요청을 하거나 ANALYZE ,DBMS_STATS를 사용하여 직접 통계정보를 재생성해야 한다. (개발자가 직접하는 경우 ??)

● 1. Execution Plan

■ SQLDEV 에서 autotrace 사용

② 자동추적 결과의 각 항목은 아래와 같다.

VIEW	VW_GBF_7	19	34364	124814	11605740	26
HASH (GROUP BY)		19	34364	124814	11605736	26
TABLE ACCESS (FULL) CUSTOMER		5350128	34102	124814	10674934	5652476

개별 노드 런타임=930802
LAST_ELAPSED_TIME=11605736

TABLE ACCESS(FULL) CUSTOMER Operation을 해석해보자.

CUSTOMER 테이블을 FULL SCAN 검색하는데 Data Buffer Cache에서 124,814 Block(1 Block 8KB)을 읽어 5,652,476 Row를 추출하는데 10.674934초 소요되었고 이를 옵티마이저 관점의 비용으로 환산하면 34,102 Cost가 사용되었다.

HASH(GROUP BY)를 해석해보자. 자식 Operation(Child,하위 Operation)인 FULL SCAN의 출력 데이터를 입력으로 사용하여 HASH 알고리즘으로 GROUP BY ACCOUNT_MGR,GENDER를 수행한후 26 Row 결과를 생성한다. 상위 Operation에 나타나는 값은 하위값을 포함하는 누적값이다.










0.93초 (11.605734초 – 10.674934초) 소요가 되었고 206 Cost (34364 Cost – 34102 Cost) 가 사용 되었다.

● 1. Execution Plan

■ Execution 변화

① DBMS 버전별 실행 계획

다음은 동일한 SQL을 다른 DBMS 버전인 9i R2 와 12c R1에서 실행한 결과 이다. 버전에 따른 실행계획의 차이를 비교 해보자

SQL	SELECT deptno,count(*) as cnt, sum(sal) as sum_sal FROM emp_large GROUP BY deptno;																												
9i R2 실행결과	<table><thead><tr><th>DEPTNO</th><th>CNT</th><th>SUM_SAL</th></tr></thead><tbody><tr><td>10</td><td>85723</td><td>250043200</td></tr><tr><td>20</td><td>142851</td><td>310718325</td></tr><tr><td>30</td><td>171426</td><td>268571750</td></tr></tbody></table> <p>경과: 00:00:01.01</p> <p>Execution Plan</p> <table><tbody><tr><td>0</td><td colspan="3">SELECT STATEMENT Optimizer=CHOOSE (Cost=850 Card=3 Bytes=15)</td></tr><tr><td>1</td><td>0</td><td colspan="2">SORT (GROUP BY) (Cost=850 Card=3 Bytes=15)</td></tr><tr><td>2</td><td>1</td><td colspan="2">TABLE ACCESS (FULL) OF 'EMP_LARGE' (Cost=261 Card=400000 Bytes=2000000)</td></tr></tbody></table>	DEPTNO	CNT	SUM_SAL	10	85723	250043200	20	142851	310718325	30	171426	268571750	0	SELECT STATEMENT Optimizer=CHOOSE (Cost=850 Card=3 Bytes=15)			1	0	SORT (GROUP BY) (Cost=850 Card=3 Bytes=15)		2	1	TABLE ACCESS (FULL) OF 'EMP_LARGE' (Cost=261 Card=400000 Bytes=2000000)					
DEPTNO	CNT	SUM_SAL																											
10	85723	250043200																											
20	142851	310718325																											
30	171426	268571750																											
0	SELECT STATEMENT Optimizer=CHOOSE (Cost=850 Card=3 Bytes=15)																												
1	0	SORT (GROUP BY) (Cost=850 Card=3 Bytes=15)																											
2	1	TABLE ACCESS (FULL) OF 'EMP_LARGE' (Cost=261 Card=400000 Bytes=2000000)																											
12c R1 실행결과	<table><thead><tr><th>DEPTNO</th><th>CNT</th><th>SUM_SAL</th></tr></thead><tbody><tr><td>30</td><td>171437</td><td>268571750</td></tr><tr><td>20</td><td>142851</td><td>310718325</td></tr><tr><td>10</td><td>85723</td><td>268572750</td></tr></tbody></table> <table><thead><tr><th>OPERATION</th><th>OBJECT_NAME</th><th>CARDINALITY</th><th>COST</th></tr></thead><tbody><tr><td> SELECT STATEMENT</td><td></td><td>3</td><td>1706</td></tr><tr><td> HASH (GROUP BY)</td><td></td><td>3</td><td>1706</td></tr><tr><td> TABLE ACCESS (FULL)</td><td>EMP_LARGE</td><td>1001303</td><td>1681</td></tr></tbody></table>	DEPTNO	CNT	SUM_SAL	30	171437	268571750	20	142851	310718325	10	85723	268572750	OPERATION	OBJECT_NAME	CARDINALITY	COST	 SELECT STATEMENT		3	1706	 HASH (GROUP BY)		3	1706	 TABLE ACCESS (FULL)	EMP_LARGE	1001303	1681
DEPTNO	CNT	SUM_SAL																											
30	171437	268571750																											
20	142851	310718325																											
10	85723	268572750																											
OPERATION	OBJECT_NAME	CARDINALITY	COST																										
 SELECT STATEMENT		3	1706																										
 HASH (GROUP BY)		3	1706																										
 TABLE ACCESS (FULL)	EMP_LARGE	1001303	1681																										

● 1. Execution Plan

■ Execution 변화

① DBMS 버전별 실행 계획

- 정렬된 데이터 차이

9i에서는 deptno 컬럼을 기준으로 정렬된 데이터 결과가 나오지만 12c에서는 정렬되지 않은 결과 생성된다.

- 실행계획 Operation 차이

GROUP BY 조건을 수행하기 위해 DBMS내부에서 버전에 따라 다른 Operation을 수행한다.

9i	SORT(GROUP BY) Operation 수행
12c	HASH GROUP BY Operation 수행

SORT(GROUP BY) Operation은 GROUP BY 조건을 실행하기 위해 정렬(sort) 연산을 수행하기 때문에 정렬된(오름차순) 결과가 나타난다. HASH GROUP BY Operation은 GROUP BY 조건을 실행하기 위해 HASH 연산을 수행하기 때문에 정렬된 결과가 나타나지 않는다.

- 비용(Cost) 차이

9i 와 12c는 비용(Cost)계산 하는 방법 및 요소가 다르기 때문에 직접적인 비용을 비교하는 것은 의미가 없지만 전체 비용중 특정 Operation의 비율을 비교 해보는 것은 의미가 있다.

버전	Operation	Cost(전체)	Cost(group by)	비율
9i	SORT(GROUP BY)	850	589	69.2%
12c	HASH GROUP BY	774	17	2.1%

● 1. Execution Plan

■ Execution 변화

① DBMS 버전별 실행 계획

GROUP BY 조건 처리를 위해 9i에서는 전체 비용의 69.2%를 사용하고 12c에서는 정렬된 결과를 포기하는 대신 전체 비용의 2.1%를 사용 한다.

버전에 따라 GROUP BY 조건의 내부 처리 방식이 SORT에서 HASH로 달라 질수 있다는 것을 알고 있어야 한다. 실행계획은 여러가지 요소에 의해서 변경될수 있다. 예를들면 데이터량, 데이터 분포도, Instance 파라미터,인덱스의 변경, Oracle 버전 변경등 여러 요소가 실행계획을 변경 시킨다.

실행계획을 분석할수 있게 됨에 따라 DBMS 내부의 처리 방식을 이해 할수 있다.실행계획을 읽을수 없다면 12c에서 group by 실행시 정렬된 결과가 나오지 않는지 이해할수 없다.

DBMS가 운영되는 H/W서버가 과거에 비해 더 높은 CPU성능 과 더 많은 CPU개수 및 대용량 메모리를 채택하게 됨에 따라 데이터를 그룹화하기 위해 IO 집약적인 SORT 연산 보다 CPU 집약적인 HASH 연산이 효율적인 결과를 나타냄에 따라 HASH 연산이 SORT 연산을 대체 되었다.

● 1. Execution Plan

■ Execution 변화

② Optimizer 모드별 실행 계획

10g 버전부터는 RBO(Rule Based Optimizer)를 더 이상 지원하지 않지만 실행계획이 RBO 모드에 작성되었는지 CBO(Cost Based Optimizer) 모드에서 작성 되었는지를 실행계획을 보고 구분할수 있어야 한다.

SQL	<pre>SELECT /*+ RULE */ deptno,count(*) as cnt, sum(sal) as sum_sal FROM emp_large GROUP BY deptno;</pre>
RBO	<pre>Execution Plan ----- Plan hash value: 3369992899 ----- Id Operation Name ----- ----- ----- 0 SELECT STATEMENT 1 SORT GROUP BY 2 TABLE ACCESS FULL EMP_LARGE -----</pre>
CBO	<pre>Execution Plan ----- Plan hash value: 855116971 ----- Id Operation Name Rows Bytes Cost (%CPU) Time ----- ----- ----- ----- ----- ----- ----- 0 SELECT STATEMENT 409K 10M 774 (4) 00:00:01 1 HASH GROUP BY 409K 10M 774 (4) 00:00:01 2 TABLE ACCESS FULL EMP_LARGE 409K 10M 757 (1) 00:00:01 -----</pre>

* CBO 모드에서 생성된 실행계획에는 비용(Cost)이 나타나고 RBO 모드에서 생성된 실행계획에는 비용(Cost)이 나타나지 않는다.

● 1. Execution Plan

■ Execution 변화

③ 통계정보별 실행계획

옵티마이저는 2가지 종류의 통계정보를 참조하여 실행계획을 수립 한다.

- 시스템(System) 통계정보는 CPU 성능(Mhz) ,Disk I/O 성능에 대한 정보
- 오브젝트(Object) 통계정보는 테이블 ,인덱스등에 대한통계정보로 테이블의 전체 Rows 의 개수, 평균 Row 길이, Block의 개수 ,컬럼의 데이터 분포도, 컬럼내의 고유한 값의 개수 등에 대한 정보

시스템 통계정보는 DBA 나 튜너의 영역으로 개발자인 여러분은 System 통계정보가 실행계획을 결정하는데 미치는 영향을 직접 볼수 없지만 오브젝트 통계정보에 따라 실행 계획이 달라지는 것을 볼수 있다.

오브젝트 통계정보를 수집 하는 2가지 방법은 아래와 같다.

수집방법	설명	예
ANALYZE	8i 버전까지 사용	ANALYZE TABLE EMP COMPUTE STATISTICS;
DBMS_STAT	ANALYZE 명령어의 단점을 개선한 패키지	DBMS_STATS.GATHER_TABLE_STAT(OWNNAME=>'SCOTT',TABNAME=>'EMP');

● 1. Execution Plan

■ Execution 변화

③ 통계정보별 실행계획

부정확한 통계정보로 인해 실행계획이 비효율적으로 수립되는 아래의 사례를 보자

EMP_LARGE에는 100만명의 사원 정보가 있으며 직원의 직무는 PRESIDENT가 1명이고 ANALYST가 10명이다.

JOB컬럼에는 INDEX가 생성되어 있다.

SQL	<div>Select * From emp_large Where job = 'CLERK';</div> <div>Select * From emp_large Where job = 'PRESIDENT'; // ?? president</div> <div>Select * From emp_large Where job = 'ANALYST';</div>																																																												
결과	<table><thead><tr><th>JOB</th><th>COUNT(*)</th></tr></thead><tbody><tr><td>1 CLERK</td><td>428553</td></tr><tr><td>2 SALESMAN</td><td>285717</td></tr><tr><td>3 ANALYST</td><td>10</td></tr><tr><td>4 MANAGER</td><td>214291</td></tr><tr><td>5 ENGINEER</td><td>71428</td></tr><tr><td>6 PRESIDENT</td><td>1</td></tr></tbody></table>	JOB	COUNT(*)	1 CLERK	428553	2 SALESMAN	285717	3 ANALYST	10	4 MANAGER	214291	5 ENGINEER	71428	6 PRESIDENT	1																																														
JOB	COUNT(*)																																																												
1 CLERK	428553																																																												
2 SALESMAN	285717																																																												
3 ANALYST	10																																																												
4 MANAGER	214291																																																												
5 ENGINEER	71428																																																												
6 PRESIDENT	1																																																												
실행계획	<div><table><thead><tr><th>OPERATION</th><th>OBJECT_NAME</th><th>CARDINALITY</th><th>COST</th></tr></thead><tbody><tr><td> SELECT STATEMENT</td><td></td><td>250326</td><td>1680</td></tr><tr><td> TABLE ACCESS (FULL)</td><td>EMP_LARGE</td><td>250326</td><td>1680</td></tr><tr><td> Filter Predicates</td><td></td><td></td><td></td></tr><tr><td></td><td>JOB='CLERK'</td><td></td><td></td></tr></tbody></table></div> <div><table><thead><tr><th>OPERATION</th><th>OBJECT_NAME</th><th>CARDINALITY</th><th>COST</th></tr></thead><tbody><tr><td> SELECT STATEMENT</td><td></td><td>250326</td><td>1680</td></tr><tr><td> TABLE ACCESS (FULL)</td><td>EMP_LARGE</td><td>250326</td><td>1680</td></tr><tr><td> Filter Predicates</td><td></td><td></td><td></td></tr><tr><td></td><td>JOB='PRESIDENT'</td><td></td><td></td></tr></tbody></table></div> <div><table><thead><tr><th>OPERATION</th><th>OBJECT_NAME</th><th>CARDINALITY</th><th>COST</th></tr></thead><tbody><tr><td> SELECT STATEMENT</td><td></td><td>219073</td><td>1680</td></tr><tr><td> TABLE ACCESS (FULL)</td><td>EMP_LARGE</td><td>219073</td><td>1680</td></tr><tr><td> Filter Predicates</td><td></td><td></td><td></td></tr><tr><td></td><td>JOB='ANALYST'</td><td></td><td></td></tr></tbody></table></div>	OPERATION	OBJECT_NAME	CARDINALITY	COST	SELECT STATEMENT		250326	1680	TABLE ACCESS (FULL)	EMP_LARGE	250326	1680	Filter Predicates					JOB='CLERK'			OPERATION	OBJECT_NAME	CARDINALITY	COST	SELECT STATEMENT		250326	1680	TABLE ACCESS (FULL)	EMP_LARGE	250326	1680	Filter Predicates					JOB='PRESIDENT'			OPERATION	OBJECT_NAME	CARDINALITY	COST	SELECT STATEMENT		219073	1680	TABLE ACCESS (FULL)	EMP_LARGE	219073	1680	Filter Predicates					JOB='ANALYST'		
OPERATION	OBJECT_NAME	CARDINALITY	COST																																																										
SELECT STATEMENT		250326	1680																																																										
TABLE ACCESS (FULL)	EMP_LARGE	250326	1680																																																										
Filter Predicates																																																													
	JOB='CLERK'																																																												
OPERATION	OBJECT_NAME	CARDINALITY	COST																																																										
SELECT STATEMENT		250326	1680																																																										
TABLE ACCESS (FULL)	EMP_LARGE	250326	1680																																																										
Filter Predicates																																																													
	JOB='PRESIDENT'																																																												
OPERATION	OBJECT_NAME	CARDINALITY	COST																																																										
SELECT STATEMENT		219073	1680																																																										
TABLE ACCESS (FULL)	EMP_LARGE	219073	1680																																																										
Filter Predicates																																																													
	JOB='ANALYST'																																																												

● 1. Execution Plan

■ Execution 변화

③ 통계정보별 실행계획

SQL 실행결과 'PRESIDENT'는 1 Rows 이지만 실행계획의 예상되는 Rows에는 80,000건으로 나타나는 이유는 옵티마이저가 filter 조건인 JOB = 'PRESIDENT'가 80,000 Rows를 리턴 할거라는 예측을 했기 때문이다. 전체 데이터의 20%에 (80,000/400,000)에 해당하는 데이터를 가져 오기 위해서는 해당 컬럼에 Index가 있지만 Full Table Scan 접근경로(Access Path)를 결정하는 것이 유리하다는 판단을 했다. 올바른 판단일까 ? 왜 이런 판단을 했을까? 옵티마이저가 Full table scan이 효율적이라고 판단한 이유를 찾아보자. USER_TABLES에는 테이블 정보와 통계정보가 저장된다. 주요한 통계정보를 보자.

컬럼	내용
table_name	테이블명
num_rows	테이블내의 Row의 개수
blocks	테이블이 사용한 블록의 개수 (데이터가 저장된 블록의 개수)
avg_row_len	1 Row의 평균길이 (Byte 단위)
sample_size	테이블 통계 분석(Analyze)에 사용된 샘플의 크기
last_analyzed	테이블 통계가 분석(수집)된 날짜

● 1. Execution Plan

■ Execution 변화

③ 통계정보별 실행계획

SQL	SELECT num_rows,blocks,avg_row_len,sample_size,last_analyzed FROM USER_TABLES WHERE TABLE_NAME = 'EMP_LARGE';				
결과	NUM_ROWS	BLOCKS	AVG_ROW_LEN	SAMPLE_...	LAST_ANALYZED
	1001303	6166	46	1044	22/10/13

EMP_LARGE 테이블은 1,000,000만 Rows를 6166개의 Block에 저장하고 있으며 통계정보는 2022/10/13일에 수집되었다. SAMPLE_SIZE 1044의 의미는 테이블에서 1044개를 샘플링하여 EMP_LARGE 테이블의 통계정보를 생성하였다는 의미이다. 일반적으로 테이블의 데이터가 대용량일 경우 테이블의 일부 데이터를 샘플링 하여 통계정보를 생성한다. 튜닝 실습에 사용되는 예제 데이터 생성 스크립트(부록)을 참조하면 테스트 데이터를 생성후 아래와 같은 통계정보 수집 명령어를 수행한다. ANALYZE TABLE EMP_LARGE COMPUTE STATISTICS; USER_TABLES 에서는 특이점이 없다. USER_TAB_COLUMNS은 테이블내의 컬럼에 대한 정보 와 통계정보가 저장된다.

컬럼	내용
column_name	컬럼명
num_distinct	컬럼의 고유한 값의 개수(Number of distinct values)
low_value	컬럼의 하한값 (RAW 타입으로 이진 데이터로 표현)
high_value	컬럼의 상한값 (RAW 타입으로 이진 데이터로 표현)
density	밀도 (1 / num_distinct)
num_nulls	컬럼의 null 개수

● 1. Execution Plan

■ Execution 변화

③ 통계정보별 실행계획

SQL	SELECT column_name,num_distinct,low_value,high_value,density,num_nulls FROM user_tab_columns WHERE table_name= 'EMP_LARGE';					
결과	⚙ COLUMN_NAME	⚙ NUM_DISTINCT	LOW_VALUE	HIGH_VALUE	⚙ DENSITY	⚙ NUM_NULLS
	EMPNO	1001303	C2093F	C3645747	0.000000998698695599634	0
	ENAME	1001303	414152466A4F67775A6C	7A7947456A45505A6C70	0.000000998698695599634	0
	JOB	4	434C45524B	53414C45534D414E	0.25	0
	MGR	6	C24C43	C25003	0.166666666666667	63301
	HIREDATE	13	77B40C11010101	77BB0417010101	0.0769230769230769	0
	SAL	12	C209	C233	0.0833333333333333	0
	COMM	4	80	C20F	0.25	711655
	DEPTNO	3	C10B	C11F	0.333333333333333	0

user_tab_columns는 테이블내의 각 컬럼별 통계정보를 가지고 있다. EMPNO,JOB 컬럼 을 분석을 해보자.

Primary key 컬럼의 모든 데이터는 고유(Unique)하고 존재(Not Null)해야 한다.

EMPNO 컬럼은 Primary Key로 NUM_DISTINCT 1,000,000이고 NUM_NULLS가 0이다.

밀집도(Density)는 0.00000001 (1/1,000,000) 이다.

low_value , high_value는 이진데이터 타입으로텍스트 데이터로 보기 위해서는 타입 변환 함수

(utl_raw.cast_to_varchar2,number)를 사용 해야 한다.

Job 컬럼의 데이터는 SALESMAN,CLERK,PRESIDENT,MANAGER,ANALYST 으로 5개의 NUM_DISTINCT 이고 NUM_NULLS가 0이다.

밀집도는 데이터가 모여있는 밀도를 의미 하며 0.25 이다. 0.25는 1 / NUM_DISTINCT를 계산하여 도출 되었다.

실제 데이터를 보면 Job 컬럼내의 데이터 밀도는 0.25 처럼 균일하지 않다. 옆에 있는 데이터건수를 보면 불균등 하게 분포 하여 실제로는 밀도가 일정하지 않다. Density 부분이 이상하다.

● 1. Execution Plan

■ Execution 변화

③ 통계정보별 실행계획

Density가 0.25로 되어 있으니

WHERE JOB = 'PRESIDENT' 조건으로 검색할때 예상 데이터를 10만 Rows * 0.25 = 25만건 계산한 것이다.

옵티마이저가 예상 데이터를 25만건으로 부정확하게 잘못 예측한 이유는 데이터 딕셔너리의 부정확한 정보를 기반으로 계산한 결과이다. 딕셔너리의 정보가 불완전하거나 현재성을 반영하지 못하는 경우에 옵티마이저는 효율적인 실행계획을 생성하지 못하게 된다.

위의 문제를 해결해보자.

10g 버전부터는 ANALYZE를 더 이상 사용하지 말고 DBMS_STATS를 사용하도록 공식적으로 권고하지만 튜닝을 처음 접하는 개발자에게는 ANALYZE 명령어가 쉽다. 튜닝 학습후에 실제 사용시에는 DBMS_STATS 패키지를 사용하기 바란다.

오브젝트 통계정보 조회 SQL	SELECT num_rows,sample_size,last_analyzed FROM user_tables WHERE table_name= 'EMP_LARGE';
	SELECT column_name,num_distinct,low_value,high_value,density,num_nulls, histogram,num_buckets FROM user_tab_columns WHERE table_name= 'EMP_LARGE';
	SELECT column_name,endpoint_number,endpoint_value,endpoint_actual_value FROM user_histograms WHERE table_name='EMP_LARGE' ORDER BY column_name,endpoint_number;
	SELECT index_name,leaf_blocks,distinct_keys,clustering_factor, num_rows ,avg_data_blocks_per_key FROM user_indexes WHERE table_name = 'EMP_LARGE';

● 1. Execution Plan

■ Execution 변화

③ 통계정보별 실행계획

ANALYZE 명령어를 사용하여 통계정보의 변화를 비교해 보자.

아래의 7개의 ANALYZE 명령어를 차례로 수행 해보면서 위의 4개의 데이터 디렉터리에서 있는 정보가 어떻게 변하는지를 실습

① ANALYZE TABLE EMP_LARGE ESTIMATE STATISTICS;

② ANALYZE TABLE EMP_LARGE ESTIMATE STATISTICS SAMPLE 20 PERCENT;

③ ANALYZE TABLE EMP_LARGE ESTIMATE STATISTICS SAMPLE 2000 ROWS;

④ ANALYZE TABLE EMP_LARGE DELETE STATISTICS;

⑤ ANALYZE TABLE EMP_LARGE COMPUTE STATISTICS;

⑥ ANALYZE TABLE EMP_LARGE COMPUTE STATISTICS

FOR TABLE FOR COLUMNS JOB;

⑦ ANALYZE TABLE EMP_LARGE COMPUTE STATISTICS

FOR TABLE FOR ALL INDEXED COLUMNS FOR ALL INDEXES;

● 1. Execution Plan

■ Execution 변화

③ 통계정보별 실행계획

COMPUTE	분석 대상 테이블을 FULL Table Scan을 통해 전체를 읽고통계정보를 분석 및 수집하기 때문에 100% 정확한 통계정보생성 한다. 테이블의 데이터가 대용량 데이터인경우 통계정보 수집에 많은 자원 및 시간이 소요될수 있으므로 데이터베이스를 사용하지 업무 시간대에 통계정보를 수집하거나 샘플링 (ESTIMATE) 방식을 권고한다.
ESTIMATE	분석 대상 테이블의 일부 데이터를 샘플링 하여 통계정보를 수집 한다. 대용량 데이터인경우 사용을 권고하며 빠른시간내에 비교적 정확도가 높은 통계정보를 수집 한다. 샘플링 방식이므로 일부 통계정보에 일부 오차가 발생하지만 옵티마이저가 실행계획을 수립하는데 영향은 미미 하다.
FOR COLUMNS	해당 컬럼의 데이터 분포도(Histogram)을 수집한다. 컬럼내의 데이터가 균등한 분포인 경우에는 밀집도를 통해 적절하게 구하지만 불균등 분포인 경우 분포도 (Histogram)을 수집해야만 옵티마이저가 정확한 실행계획을 수집한다.

분포도(Histogram)를 생성해야 하는 이유를 알아보자.

Deptno는 비교적 균등하게 분포가 되어있어서 분포도(Histogram)이 없어도 전체 40만건을 Distinct한 3개(10,20,30)의 값

으로 나누면 부서별로 평균 13만3천명 가량 근무한다고 옵티마이저는 판단 한다.

Job은 불균등하게 분포가 되어 있어서 분포도(Histogram)가 없는 경우 전체 40만건을 Distinct한 5개의 값으로 나누면

Job 별

로 평균 8만명 정도 근무를 한다고 옵티마이저는 판단한다

● 1. Execution Plan

■ Execution 변화

③ 통계정보별 실행계획

불균등 분포		균등분포	
JOB	CNT	DEPTNO	CNT
CLERK	171396	30	171437
MANAGER	114299	20	142847
SALESMAN	114294	10	85716
ANALYST	10		
PRESIDENT	1		

ANALYZE TABLE EMP_LARGE COMPUTE STATISTICS FOR TABLE FOR ALL INDEXED COLUMNS FOR ALL INDEXES;
를 실행후 위의 오브젝트 통계정보를 조회하는 SQL구문 4개를 실행해보자.

분포도 생성 전/후 실행계획을 비교해보자.

SQL

SELECT COUNT(*) AS CNT,SUM(SAL) AS SUM_SAL
FROM EMP_LARGE
WHERE JOB = 'PRESIDENT';

분포도
생성전
실행계획

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		1	10	756 (1)	00:00:01
1	SORT AGGREGATE		1	10		
* 2	TABLE ACCESS FULL	EMP_LARGE	80000	781K	756 (1)	00:00:01

분포도
생성후
실행계획

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		1	10	4 (0)	00:00:01
1	SORT AGGREGATE		1	10		
2	TABLE ACCESS BY INDEX ROWID BATCHED	EMP_LARGE	1	10	4 (0)	00:00:01
* 3	INDEX RANGE SCAN	EMP_LARGE_JOB_IDX	1		3 (0)	00:00:01

● 1. Execution Plan

■ Execution 변화

③ 통계정보별 실행계획

컬럼 분포도(Histogram)를 생성 전/후 실행계획을 비교해보면 데이터 접근경로(Access Path)가 변경되고 이로 인해 실행결과를 수행하는 산정되는 비용이 변경 된다. Full Table Scan에서 Index Scan으로 실행계획(접근경로)가 변경되고 예상 Rows가 8만에서 1로 정확하게 산정되었고 실행 비용이 756에서 10으로 감소한다. 통계정보에 따라 실행계획이 바뀌는 사례를 실습을 통해 반드시 확인 하고 옵티마이저 관점에서 판단하는 방식을 이해해야 한다.

Density	<p>밀도를 말하며 분포도"라고 부르기도 한다. Column 의 밀도를 결정</p> $\text{Density} = 1/10000 = 0.0001$ $\text{Density} = 1/\text{NDV}$ <p>Histogram이 없는 경우에는 $\text{Density} = 1/\text{NDV}$의 공식으로 계산</p> <p>Histogram이 있는 경우에는 분포도를 고려한 보정된 Density 가 계산</p> <p>Density 값은 Object Statistics 수집시 계산되며, DBA_TAB_COL_STATISTICS.DENSITY Column 을 통해 조회</p>
Cadinality	<p>Cardinality는 집합내의 원소 개수 의미.</p> <p>집합에 속하는 원소의 수가 100개라면 {Cardinality(집합) = 100 } 으로 표현 .</p> <p>Table T 의 전체 Row수가 100,000 개라면 {Cardinality(t1)=100000}이 된다. 만일 Where t.job ='clerk' 이라는 조건이 주어지고, 해당 조건을 만족하는 Row 수가 1,000 개 라면 {Cardinality(t.job)= 1,000} 이 된다.</p> $\text{Adjusted Cardinality} = \text{Base Cardinality} * \text{Selectivity}$
Selectivity	<p>선택도. 조건(Predicate)을 만족 하는 확률(선택) 비율로 Selectivity는 조건에 따라 변경된다.</p> <p>Column c1의 Density 가 0.1 이라고 하면{c1 = 1}, {c2 = :b1}, {c1 between 1 and 10}, {c1 between :b1 , :b2} 조건의 Selectivity는 모두 다르다. Histogram은 분포도로 data가 비대칭적(Skewed)일때 그 분포를 알 수 있는 유일한 방법</p> <p>Clustering Factor 는 Index를 경유한 Table Lookup의 비용을 예측하는데 사용.</p>