Oracle Optimizer의 원리 이해 및 SQL & 애플리케이션의 튜닝 (상)

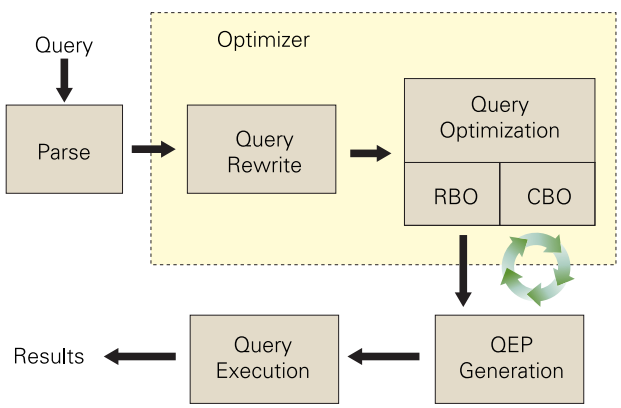
옵티마이저의 원리와 특징

글 | 최세훈 (한국오라클 Tech Sales Consulting 본부 DB Tech팀) sehoon.choi@oracle.com

다수의 데이타베이스 튜닝과 SQL / 애플리케이션 튜닝을 통해 튜닝의 효과를 확신하는 필자가 유익한 튜닝 정보를 제공한다. 여기에서 필자는 SQL 문장 개별 단위의 튜닝보다는 우선 옵티마이저의 원리를 이해하고, 전체 구조적인 문제, 유형 문제 또한 옵티마이저 관련 파라미터의 설정이 먼저 최적으로 설정된 상황하에서 SQL 문장의 단위 플랜에 대한 튜닝이 이루어져야 한다고 강조한다. 그런 취지에서 이 글에서는 옵티마이저의 원리에 대한 이해를 기반으로 SQL 및 애플리케이션 튜닝에 필요한 기본 지식을 개발자와 DBA가 쉽게 이해할 수 있도록 2회에 걸쳐 소개할 예정이다.

옵티마이저의 질의 처리 단계에 대한 이해

오라클에서 사용하는 옵티마이저(Optimizer)는 크게 RBO(Rule Base Optimizer)와 CBO(Cost Base Optimizer) 2개로 구분된다. 1992년 Oracle 7에서 처음 CBO가 지원된 이래 새로운 기능들이 적용되면서 CBO가 계속 향상되고 있는 데 반해, RBO는 오라클이 더 이상은 추구하지 않는 옵티마이저로서, 현재의 Oracle Database 10g에서도 명맥은 남아 있지만 향후는 더 이상 지원되지 않을 것이다.



<그림 1> 질의 처리 단계와 옵티마이저의 역할

옵티마이저의 입장에서 질의(query) 처리는 5단계로 나눌 수 있는데, 옵티마이저는 서브질의와 뷰의 병합(merge) 등을 수행하는 'Query Rewrite' 단계와 'Query Optimization' 단계에 참여한다. 여기서 옵티마이저는 ▲ 데이타를 어떠한 방법으로 액세스할 것이며 ▲ 올바른 결과를 어떻게 제공할 것이며 ▲ 데이타를 얼마나 효과적으로 액세스할 것인가를 결정한다. 'QEP Generation' 단계는 'Query Optimization' 단계에서 제공된 정보를 이용해서 질의에 대한 최적의 실행계획(execution plan)을 만들어 내는 단계이다. CBO에서는 질의 실행계획(QEP)을 구하기 위하여 RBO보다 복잡한 단계를 거치게 된다<그림 1>, <표 1>.

소프트 파싱과 하드 파싱

SQL 문장이 옵티마이저에 의해 처리되고 그 결과물로서, SQL 문장이 어떻게 실행될 것인지의 정보, 즉, QEP가 생기게 된다. 이들 정보는 한번 쓰고 버리는 것이 아니라 오라클의 캐쉬(cache) 영역인 SGA의 공유 풀(shared pool)에 이들 모든 정보를 캐쉬화해 관리한다. 다음 번에 같은 SQL 문장이 사용자에 의해서 실행되면, 이를 재활용하게 된다.

SQL 문장이 실행되면, 우선 SQL 문장 텍스트의 스트링을 해쉬 함수을 통과시켜 결과 값에 해당되는 버킷(어레이형 구조)에 매달린 체인 정보에

Query Process 단계	처리내용
Parse 단계	Syntax, Security, Semantics의 체크 및 Simple transformation 을 수행한다<표 2>.
Query Rewrite 단계	서브질의와 뷰의 병합을 수행하고, OR Expansion 작업을 수행한다. 서브질의와 뷰 병합이란, 옵티마이저가 더욱 효과적인 QEP를 찾기 위하여 더 효과적인 플랜이 있는지 그 가능성을 확인하는 과정이다<표 3>.
Optimization 단계	질의에 대한 액세스 경로를 결정한다.
QEP Generation 단계	질의를 실행하는 데 필요한 상세한 정보를 만들며, 이를 질의 실행계획(QEP : Query Execution Plan)이라고 한다. • 질의 실행계획(QEP) QEP는 시리얼 플랜(serial plan)과 패러렐 플랜(parallel plan)이 있다. 시리얼 플랜이란 질의에 대해서 병렬성이 적용되지 않은 플랜이며, 패러렐 플랜이란 질의에 대해서 병렬로 실행할 정보를 생성해 내는 것이다. 경우에 따라서 시리얼 플랜만 생성하거나, 시리얼과 패러렐 플랜을 동시에 생성하기도 한다. 오라클의 시리얼 플랜은 질의가 병렬로 수행할 정보가 없을 경우, 즉 테이블의 Degree이나 힌트 등이 없는 경우는 시리얼 플랜만 만들게 되며, 병렬성이 적용될 경우 시리얼 플랜과 패러렐 플랜을 모두 만들게 된다. 오라클의 시리얼 플랜을 RSO(Row Source Operator) Tree라 하며, 패러렐 플랜을 DFO(Data Flow Operator) Tree라고 한다. 이들 QEP로 병렬로 실행하려고 하나 실행시 리소스의 부족으로 원하는 Degree으로 병렬적으로 실행할 수 없는 경우 RSO Tree를 쓰기도 한다.
Query Exectution 단계	QEP에 따라 SQL 문장을 실행한다.

<표 1> 질의 처리 단계별 역할

Example Expression (from)	Transformation (to)
ename LIKE 'WARD'	ename= 'WARD'
ename IN ('KING', 'WARD')	ename= 'KING' OR ename= 'WARD'
ename=ANY/SOME('KING', 'WARD')	ename= 'KING' OR ename= 'WARD'
deptno != ALL(10,20)	deptno != 10 AND deptno != 20
sal BETWEEN 2000 and 3000	sal >= 2000 AND sal <= 3000
NOT(sal<1000 OR comm is null)	sal >= 1000 and comm is not null

<표 2> 간단한 변형(simple transformations)의 예

서 같은 SQL 문장이 존재하는지 찾는 처리절차를 수행하게 된다. 또한 같은 SQL 문장을 찾았어도 여러 버전이 존재할 수 있다. 여러 버전이란, 같은 SQL 문장(대/소문자, 화이트 스페이스 등이 모두 같아야 함)이지만 서로 다른 스키마의 테이블(예, scott의 emp, sys의 emp)이거나, 바인드 변수를 사용한 경우는 바인드 변수의 타입, 길이 등에 의해서도 서로 다른 버전

◉ **뷰 병합 예**

Example Expression (from)	View Merging (to)
create view emp_d10 as select * from emp where deptno=10; select empno from emp_d10 Where empno>11910	select empno from emp where deptno = 10 and empno > 11910;

◉ **서브질의 병합 예** (Single Row Sub-Query)

Example Expression (from)	Sub-Query Merging (to)
select ... from dept where deptno = (select deptno from emp where empno < 12501);	select ... from dept where deptno = <evaluated_value>;

<표 3> 서브질의와 뷰의 병합 예

이 된다는 것이다. 이와 같이 같은 SQL 문장에 같은 버전을 찾았다면 이를 '소프트 파싱(soft parsing)' 이라고 한다.

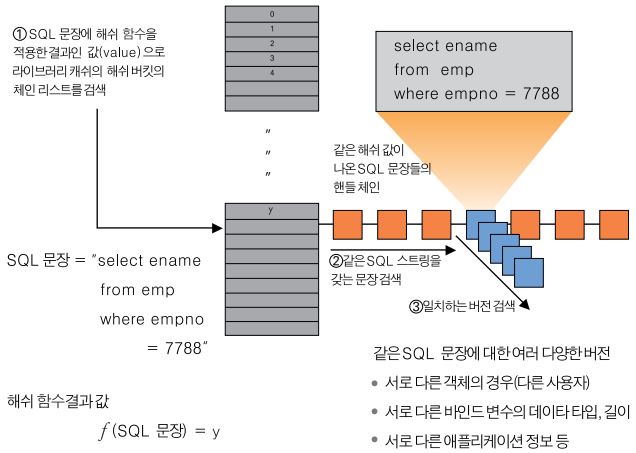
그렇지만, 체인을 다 찾았는데 같은 문장을 발견하지 못했다면, 해당 SQL 문장이 Parsing/Optimizing 단계를 거친 결과로 나온 정보를 저장하기 위해 공유 풀로부터 메모리를 확보 받고, 기록한 정보를 체인에 매달게 된다. 이를 '하드 파싱(hard parsing)' 이라고 한다. 당연히 하드 파싱의 작업량이 소프트 파싱의 작업량에 비해 월등히 클 것이다<그림 2>.

이와 같은 소프트 파싱과 하드 파싱의 과정을 생각해 볼 때, 집중적인 SQL 문장이 실행되는 OLTP(초당 수천 ~ 수만 개 이상)에서 하드 파싱이 많다면 어떻게 될까? 한정된 메모리인 캐쉬에 새로운 메모리를 계속 할당하고, LRU 알고리즘에 의해 제거하고, 체인에 매달고 끊는 등의 일들을 반복해야 할 것이다. 또한 하드 파싱은 복잡한 처리과정을 거치므로 많은 자원(CPU)을 사용하게 된다. 그러므로 OLTP 환경에서는 이와 같은 하드 파싱을 가능한 줄이도록 해야 한다. 특히 SQL 실행 규모가 큰 OLTP 업무는 1% 미만을 권장한다.

애플리케이션을 개발할 때 이러한 하드 파싱을 줄이기 위한 방법으로 거의 대부분의 데이타베이스 접속 방식(JDBC, ODBC, ADO, PRO*C 등)에서 자주 사용되는 SQL 문장들은 바인드 변수 기법들을 사용하여 개발하는 방법들을 제공하고 있다. 또한 일부에서는 소프트 파싱 자체도 줄일 수 있는 기법들을 제공하고 있다. 실제 이러한 기법을 적용해서 튜닝한 결과, 시스템 CPU/메모리 측면에서 40~50% 이상 개선된 사례가 많이 있다. 혹시 현재 운영중인 시스템이 사용자가 많아지면서 CPU 리소스가 급격히 증가해, 라이브러리 캐쉬, 공유 풀 경합 현상이 발생한다면, 이러한 점을 의심해 볼 수 있다.

<표 4>는 SQL 문장을 바인드 변수를 사용한 공유 SQL과, 상수를 결합한 형태로 SQL 문장을 만들어 실행시키는 비공유 SQL을 9,999회 실행시켜 오라클의 공유 풀 메모리 사용현황과 파싱시 CPU 사용시간을 테스트한 것이다(단, 그 결과치는 실행 서버별로 차이가 있다).

결론적으로 보면, 비공유 SQL 방식의 사용 메모리와 CPU 사용률이



<그림 2> SQL 문장의 파싱된 정보를 찾기 위한 라이브러리 캐쉬 검색절차

실행 규모에 비례해 증가하고, 실행된 SQL 문장이 기존에 캐쉬화되어 있는 SQL 문장들을 밀어내는 역할을 한다는 것을 알 수 있다.

이와 같은 SQL 문장을 공유하기 위해서 오라클 입장에서 처리해주는 CURSOR_SHARING이라는 파라미터를 제공하기도 한다. 그러나, CURSOR_SHARING은 모든 상수를 다 바인드 변수로 바꿔버리기 때문에 개발자가 의도하지 않은 Literal까지도 바꾸게 되므로, 애플리케이션을 수정할 수 있다면 가능한 애플리케이션 단에서 바인드 변수를 사용하는 것이 효과적이다.

하드 파싱을 줄이기 위해 모든 업무에 바인드 변수 사용방법을 적용하는 것은 잘못된 생각이다. 옵티마이저의 입장에서 보면, 바인드 변수 기법보다는 Literal을 사용한 비공유 SQL 방식을 좋아한다. Literal SQL 문장일 경우는 상수 값에 따라서 범위를 정확히 알 수 있기 때문에 효과적인 플랜을 결정하는 주요 결정요소로 작용하기 때문이다. 즉 바인드 변수 기법은 옵티마이저의 판단에는 좋지 않지만 SQL 문장이 집중적으로 실행되는 OLTP 환경에서 하드 파싱의 비율을 줄이기 위한 방법인 것이다. 즉, 업무의 특징에 따라서 다른 적용방식이 사용되어야 한다. 다음은 OLTP와 DW의 특징에 따라 다르게 고려되어야 할 사항이다.

• OLTP의 특징

- 목표 : 신속한 응답 시간, 적은 데이타 처리량
- 파싱 타임을 최소화하고 SQL 등이 공유될 수 있도록 바인드 변수를 사용해야 한다.
- 인덱스의 사용률이 높아야 한다.
- 정렬(sorting)을 최소화해야 한다.
- Nested Loop Join(FIRST_ROWS_n) 방식으로 많이 유도한다.

• DW의 특징

- 목표 : 최고의 처리량, 방대한 데이타 처리량
- 인덱스의 참조는 중요한 사항이 아니다.
- 정렬 또는 Aggregate함수 등이 중요한 역할을 한다.
- Hash Join 등을 많이 사용하도록 유도한다.
- 파싱 타임 등은 그리 중요하지 않으며, 바인드 변수의 사용이 문제가 될 수 있다.
- 병렬 질의 등의 사용률을 높인다.

Rule Base Optimizer

질의 최적화(query optimization)에서 RBO(Rule Base Optimizer)는 정해진 랭킹(ranking)에 의해 플랜을 결정한다. 같은 랭킹이라면 Where 절의 뒤부터, From절 뒤의 객체가 우선 순위를 갖는다. 한 객체(예 : 테이블)에서 같은 랭킹의 인덱스가 있다면 가장 최근에 만들어진 인덱스를 사용한다. 이는 CBO(Cost Base Optimizer)에서도 같이 적용되는 사항이다. 다분히 RBO는 개발자들이 프로그래밍 단계에서 SQL 문장 구조의 인위적인 조정 등으로 인덱스를 사용 못하게 하는 등 개발자가 코딩에 신경을 많이 써야 하는 문제점이 있다. 또한 RBO는 해당 질의에 대한 테이블의 인덱스가 존재한다면 전체 90% 이상의 대상이어도 인덱스를 선택한다는 것이다. 즉, RBO는 무조건 다음과 같은 미리 정해진 룰을 기준으로 플랜을 결정하게 된다. 1992년 Oracle 7에서 CBO가 지원되면서 CBO는 계속적인 신기능의 적용으로 발전해 온 반면, RBO는 더 이상의 기능 향상은 없으며, 향후는 CBO만 지원될 계획이다. 그러므로 RBO에 더 이상의 미련을 갖지 말기 바라며, CBO의 훌륭한 기능들을 적극 활용하길 바란다.

다음은 RBO의 랭킹을 정리한 것이다.

	SQL 유형	공유 풀의 메모리 사용	하드 파싱 수	실행 수	파싱 CPU 사용률
공유 SQL	select ename from emp where empno = :1	9,807	1	9,999	0.01sec
비공유 SQL	select ename from emp where empno = 1 select ename from emp where empno = 2 select ename from emp where empno = 3	93,219,148(92MB)	9,999	9,999	14.33sec

<표 4> 공유 SQL과 비공유 SQL의 비교

Path 1 : Single Row by Rowid
Path 2 : Single Row by Cluster Join
Path 3 : Single Row by Hash Cluster Key with Unique or Primary Key
Path 4 : Single Row by Unique or Primary Key
Path 5 : Clustered Join
Path 6 : Hash Cluster Key
Path 7 : Indexed Cluster Key
Path 8 : Composite Index
Path 9 : Single-Column Indexes
Path 10 : Bounded Range Search on Indexed Columns
Path 11 : Unbounded Range Search on Indexed Columns
Path 12 : Sort-Merge Join
Path 13 : MAX or MIN of Indexed Column
Path 14 : ORDER BY on Indexed Column
Path 15 : Full Table Scan

특히 Path 8, 9, 10에 주의를 해야 한다. 예를 들면, 'emp' 테이블에 'A' 인덱스가 "deptno"로 구성되어 있고, 'B' 인덱스가 "deptno + empno"로 구성되어 있다면, 다음과 같은 SQL 문장은 'A' 인덱스를 사용하게 된다. 조건이 Bounded Range Search(Between)로 왔기 때문에 아래의 SQL 문장에서 (A)와 (B)의 랭킹은 (A) ==> Rank 9 , (B) ==> Rank 10 조건이 되므로 싱글 칼럼 인덱스를 사용한다는 것이다.

```
select /*+ rule */ * from emp where deptno = 10 and empno between 7888 and 8888;
                                    A                B
```

그러면, 이제 CBO에 대해 살펴보기 전에, 참고로 RBO를 CBO로 전환한 사례를 잠깐 소개하겠다.

현재 RBO를 사용하고 있는 상황에서 마이그레이션시 CBO로 전환하고 싶으나, 막연히 두려운 부분도 많을 것이다. 실제 RBO에서 CBO로 전환하고 나서 가장 효과를 보는 부분은 배치 잡 형태이다. 특히 Oracle9*i* Database 이상의 WORKAREA_SIZE_POLICY=AUTO로 운영하는 곳이라면 더욱 더 그럴 것이다. 그러나 OLTP의 변화는 조심해야 한다. 아래의 경우는, 이전하면서 옵티마이저 모드를 RBO에서 CBO로 전환한 것 뿐만 아니라, 블록 사이즈와 CBO 옵티마이저에 민감한 db_file_multiblock_read_count 값도 크게 늘렸다. 특히 WORKAREA_SIZE_POLICY=AUTO로 필요한 워킹 메모리(Sort, Hash, Bitmap 등)를 옵티마이저가 판단하에 가능한 충분히 사용하게 하는 방식을 사용하였다.

그러다 보니, CBO에 영향을 주는 소트 메모리와 해쉬 메모리가 풍부하게 되었고, 블록 사이즈도 커졌으며, 풀 테이블 스캔의 정도를 결정하는 db_file_multiblock_read_count 값도 아주 커진 상태이다. 또한 마이그레이션되면서 데이타가 재정리되어 있는 상태이므로, 풀 테이블 스캔과 Sort Merge Join, Hash Join의 경향이 커진 상태이다. 그러므로 배치 잡의 경우는 최적의 조건이 되었으나, 기존에 주로 Nested Loop Join을 선호하던 RBO 환경의 OLTP들은 많은 플랜의 변화에 직면하게 된다. 그러면 이러한 부분을 어떻게 보정해 줄 것인가? optimizer_index_caching, optimizer_index_cost_adj의 파라미터가 그 해답일 것이다. 가능한 Nested Loop Join를 선호하고, CBO의 옵티마이저 모드가 인덱스에 점수를 더 주어서 인덱스의 비중을 키울 수가 있는 것이다. 물론 이러한 전환형태 말고 옵티마이저 모드를 FIRST_ROWS_n으로 운영하거나, 아웃라인을 이용하는 방법 등도 있을 것이다. 여러 방법이 있겠지만, 필자는 아래와 같은 방법을 선호한다.

다음은 마이그래이션시 RBO에서 CBO로 전환한 사례이다.

Oracle 7 --> Oracle 9 (RBO to CBO 전환 사례)

db_block_size	: 2KB --> 8KB
db_file_multiblock_read_count	: 8 --> 32
optimizer_mode	: RULE --> CHOOSE
hash_join_enabled	: FALSE --> TRUE
workarea_size_policy	: AUTO (New)
optimizer_index_caching	: 80 (New)
optimizer_index_cost_adj	: 20 (New)

Cost Base Optimizer

질의 최적화에서 CBO(Cost Base Optimizer)는 해당 SQL 문장이 참조하고 있는 객체들(테이블, 인덱스 등)에 대한 수집된 통계정보(statistics)의 값과 데이타베이스 파라미터(init.ora) 설정 값을 기초로 가장 적은 비용(cost)이 발생되는 플랜을 결정하는 옵티마이저 방식이다.

여기서 중요한 사실은 RBO에서는 전혀 사용되지 않았던 통계정보를 CBO에서는 이용한다는 것이다. 이들 통계정보는 DBA에 의해서 또는 자동 수집 기능(Oracle9*i* Database Release 2, Oracle Database10g)에 의해 객체들의 통계정보를 관리하는 시스템 딕셔널리(Dictionary)에 저장되고, 이 정보를 CBO 옵티마이저가 이용하는 것이다. 이들 정보는 SQL 문장을 실행하는 데 얼마만큼의 I/O 횟수가 발생할 것인가를 계산하기 위한 각종 데이타를 가지고 있다. 여기서 중요한 사실은 I/O 크기는 중요하지 않으며 I/O 횟수가 중요하다는 것이다. 즉, CBO 옵티마이저는 SQL 문장에 대한 여러 가지 경우의 수별로 I/O의 횟수에 비례한 비용을 산출해내고, 이들 비용에서 가장 작은 비용을 갖는 플랜을 결정한다는 것이다. 즉, 비용은 I/O 횟수에 비례하는 값이라고 보면 쉬울 것이다.

그러나, Oracle Database 10*g*부터는 비용의 단위 기준이 I/O에서 처리시간으로 바뀌었다(time base). 또한 Oracle9*i* Database부터 시스템 통계정보(CPU, 디스크 액세스 타임)를 이용해서 I/O로 환산한 방식을 제공하였으나, 이것은 단지 옵션이었다. 그러나, Oracle Database 10*g*부터는

시스템 통계정보(CPU, 디스크 액세스 타임)를 이용해서 처리시간으로 환산한 방식을 디폴트로 사용하므로 상당히 정확한 플랜을 만들어내며, 실행 예측 시간도 상당히 정확하다.

그러면, 여기서 잠깐 CBO에서 사용되는 통계정보가 저장된 Dictinary 정보 예를 참고로 살펴보자.

[USER|ALL|DBA]_TABLES : Table의 통계정보
NUM_ROWS,BLOCKS,AVG_ROW_LEN,SAMPLE_SIZE,LAST_ANALYZED

[USER|ALL|DBA]_INDEXES : Index의 통계정보
BLEVEL,LEAF_BLOCKS,DISTINCT_KEYS,AVG_LEAF_BLOCKS_PER_KEY,
AVG_DATA_BLOCKS_PER_KEY,CLUSTERING_FACTOR,NUM_ROWS,SAMPLE_SIZE,
LAST_ANALYZED

[USER|ALL|DBA]_TAB_COLUMNS : Column의 통계정보
NUM_DISTINCT,LOW_VALUE,HIGH_VALUE,DENSITY,NUM_NULLS,NUM_BUCKETS,
LAST_ANALYZED,SAMPLE_SIZE,AVG_COL_LEN

[USER|ALL|DBA]_TAB_HISTOGRAMS : Column의 Data 분포도 정보
TABLE_NAME,COLUMN_NAME,ENDPOINT_NUMBER,ENDPOINT_VALUE,
ENDPOINT_ACTUAL_VALUE

기타 파티션 / 클러스터 등에 대한 통계정보

그러면 "select * from dept where deptno = 10"과 같은 SQL 문장을 실행해야 한다고 가정하자. 여기서 dept Table은 deptno에 대한 인덱스가 있고, 테이블은 전체 10 블록으로 구성되어 있으며, 풀 테이블 스캔일 경우 I/O 단위를 결정하는 파라미터는 DBA가 db_file_multiblock_read_count=8로 지정하여 운영하고 있다고 가정하자. 여기서 RBO라면 무조건 인덱스를 타는 플랜을 결정하였을 것이다.

그러나, CBO의 경우는 deptno의 인덱스를 이용해 실행하면, 3회(인덱스는 싱글 블록 단위 I/O)의 I/O가 발생한다고 가정하고, 풀 테이블 스캔의 경우는 2회(8블록 + 2블록)의 I/O가 발생한다고 가정하면, CBO에서는 인덱스가 있음에도 I/O 횟수 측면에서 더 효과적인 풀 테이블 스캔을 선택한다는 것이다. 그러므로 CBO는 이와 같이 가능한 정확한(현실 데이타와 맞는) 통계정보와 적절한 데이타베이스의 파라미터인 init.ora에 의해 플랜이 결정되는 것이다.

CBO에서만 가능한 기능들

CBO는 오라클의 신기능을 지원하도록 지속적으로 발전하고 있으며, 다음의 경우는 반드시 CBO에서만 플랜 결정시 검토되거나 무조건 CBO로 동작되는 경우이다. 예를 들어, 파티션 테이블을 사용한다면 통계정보가 없더라도 무조건 CBO로 동작된다는 것이다.

- Partitioned tables (*)
- Index-organized tables
- Reverse key indexes
- Function-based indexes
- SAMPLE clauses in a SELECT statement (*)
- Parallel execution and parallel DML
- Star transformations
- Star joins
- Extensible optimizer
- Query rewrite (materialized views)
- Progress meter
- Hash joins
- Bitmap indexes
- Partition views (release 7.3)
- Hint (*)
- Parallel DEGREE & INSTANCES - 'DEFAULT' 도 해당 (*)

CBO의 옵티마이저에 영향을 줄 수 있는 파라미터 예

옵티마이저가 플랜을 수립하는 데 영향을 줄 수 있는 파라미터 값이 무엇인지를 알고 있는 것이 무엇보다 중요하다. 실제 옵티마이저가 참조하는 파라미터는 Oracle9*i* Database 기준으로 보더라도 60여 개에 이른다. 특히 DBA는 이들 옵티마이저의 파라미터 설정에 신중해야 한다. 또한 이들 파라미터의 효과적인 설정은 개발 중이거나, 마이그래이션 중에 업무의 특징을 판단한 다음, 해당 업무에 가장 효과적인 것을 설정해야 한다. 기준이 잘못되면 개발자들은 SQL 문장마다 힌트를 넣기 바쁠 것이고, 많은 인적 자원을 튜닝에 소모해야 할 것이다. 그러므로 대부분의 업무들이 최적화되어 잘 운영될 수 있는 형태로 이들 파라미터를 바꿔가면서 기준을 정하는 것이 중요하다. 물론 이들 값보다도 CBO에서 사용되는 통계정보가 중요하다는 것은 당연한 사실이다. 오라클에서는 개발장비에도 운영장비에 있는 통계정보와 같게 운영할 수 있도록 DBMS_STATS 패키지를 제공한다.

다음은 질의 수행시 옵티마이저가 플랜을 수립하기 위해 참조한 파라미터 중 일부이다(버전마다 다르다).

OPTIMIZER_PERCENT_PARALLEL (Default = 0)

Optimizer_Percent_Parallel의 Parameter는 CBO가 비용을 계산하는 데 영향을 주는 파라미터이다. 즉 수치가 높을수록 병렬성을 이용하여 풀 테이블 스캔으로 테이블을 액세스하려고 한다. 이 값이 0인 경우는 최적의 시리얼 플랜이나 패러렐 플랜을 사용하며, 1~100일 경우는 비용 계산에서 객체의 등급을 사용한다.

OPTIMIZER_MODE (Default=Choose(Oracle7 ~ Oracle9*i* Database),ALL_ROWS)
{Choose(<=9i)|Rule(<=9i)|First_rows|First_rows_n(>

=Oracle9*i*|All_rows}

기본적인 옵티마이저 모드를 결정한다(왼쪽상자기사 '옵티마이저 모드의 종류 및 특징' 참조).

HASH_AREA_SIZE, HASH_JOIN_ENABLED (Oracle Database 10*g* : _hash_join_enabled=true)

위의 파라미터 값에 따라서 Hash Join으로 유도할 수 있다. Hash Join이 가능하고 해쉬 메모리가 충분하다면, 플랜에 Hash Join의 경향이 커진다.

OPTIMIZER_SEARCH_LIMIT (Default = 5)

옵티마이저에게 조인 비용을 계산할 경우, From절에 나오는 테이블의 개수에 따라서 조인의 경우의 수가 있을 수 있으며, 옵티마이저는 이들 각각의 경우의 수에 대한 조인 비용을 계산하게 된다. 물론 일부 예외사항은 있다. 예를 들어, Cartesian Production Join 등은 우선 순위가 낮으므로 뒤로 미뤄질 것이다. 이 파라미터의 값이 5일 경우 From절에 5개의 테이블에 대해서 모든 조인의 경우의 수를 가지고 비용을 계산하게 되며, 그 개수는 5!=120개의 경우의 수에 대한 조인 비용을 계산하게 되므로 옵티마이저가 많은 시간을 소모하게 되므로 성능에 영향을 미칠 수도 있다.

SORT_AREA_SIZE , SORT_MULTIBLOCK_READ_COUNT

위의 파라미터의 값에 따라서 Sort Merge Join으로 유도할 수 있다. 소트 메모리가 충분하다면, 플랜에 Sort Merge Join의 경향이 커진다.

DB_FILE_MULTIBLOCK_READ_COUNT

이 파라미터의 수치가 클수록 인덱스 스캔보다는 풀 테이블 스캔의 비중이 높아진다. 이 파라미터는 옵티마이저의 플랜 결정에 민감하게 영향을 주는 값이다. 즉, 이 값이 커지면 풀 테이블 스캔과 병행해서 Sort Merge Join 또는 Hash Join의 경향이 커진다.

OPTIMIZER_INDEX_CACHING (Default = 0)

CBO가 Nested Loop Join을 선호하도록 조절하는 파라미터, Nested Loop Join시 버퍼 캐쉬 내에 이너 테이블의 인덱스를 캐쉬화하는 비율(%)을 지정하므로 Nested Loop Join시 성능이 향상되며, 옵티마이저는 비용 계산시 이 비율을 반영하여 Nested Loop Join을 선호하도록 플랜이 선택된다(0~100). 100에 근접할수록 인덱스 액세스 경로가 결정될 가능성이 높다. 기존의 RBO를 CBO로 전환시 옵티마이저를 RBO 성향으로 보정하는 데 효과적이다.

OPTIMIZER_INDEX_COST_ADJ (Default = 100)

옵티마이저가 인덱스를 사용하는 위주의 플랜으로 풀릴 것인지 또는 가능한 사용하지 않을 쪽으로 풀릴 것인지의 비중을 지정한다. CBO는 RBO처럼 인덱스를 사용하도록 플랜이 주로 만들어지게 되나, 인덱스가 있다고 해서 RBO처럼 인덱스를 이용한 플랜으로 처리되는 것은 아니다. 인덱스를 이용하는 플랜 위주로 하고자 한다면 100(%) 이하를, 가능한 인덱스를 사용하지 않고자 한다면 100 이상을 지정한다(1 ~ 10000). 이 파라미터는 기존의 RBO를 CBO로 전환시 옵티마이저를 RBO의 인덱스 위주 성향으로 보정하는 데 효과적이다.

WORKAREA_SIZE_POLICY (AUTO | MANUAL)

옵티마이저가 [HASH|SORT|BITMAP_MERGE|CREATE_ BITMAP] *_AREA_SIZE를 자동으로 결정하는 PGA 자동 관리 방식으로, 인스턴스에 속한 모든 PGA의 메모리의 합이 PGA_AGGREGATE_TARGET에서 설정된 메모리를 가능한 넘지 않는 범위 내에서 Workarea(Sort, Hash, Bitmap 등)를 충분히 사용하고자 하는 방식이다. 플랜은 할당된 Workarea를 가지고 플랜을 결정하게 되므로 풍부한 메모리에 의해 Hash Join, Sort Merge Join등을 선호하는 경향이 높다. 내부적으로 히든 파라미터로 *_AREA_SIZE의 값을 가지고 플랜을 결정할 수도 있으나 인위적인 설정 없이는 자동 할당된 메모리로 플랜이 결정된다.

옵티마이저 모드의 종류 및 특징

- **인스턴스 레벨** : optimizer_mode = {Choose|Rule|First_rows|First_rows_n| All_rows}
- **세션 레벨 :** 인스턴스 레벨에 우선
  ALTER SESSION SET optimizer_mode = {Choose|Rule|First_rows|First_rows_n|All_rows}
- **스테이트먼트 레벨 :** 힌트를 사용하며, 인스턴스, 세션 레벨에 우선
- Oracle9*i* Database에서 FIRST_ROWS_n 옵티마이저 모드가 추가되었음(N : 1, 10, 100, 1000).
- Oracle Database 10*g*에서는 CHOOSE, RULE 모드는 더 이상 지원되지 않으나, 기능은 남아 있다.
- **OPTIMIZER_MODE=CHOOSE 일 경우**
  통계정보가 없다면 기본적으로 RBO로 플랜이 결정된다. 그러나, 'RULE', 'DRIVING_SITE' 힌트 이외의 힌트가 왔다면 CBO로 결정된다(힌트는 룰의 규정을 깨므로 CBO로 동작됨).
- Parallel Degree, Partition Table, SAMPLE절 등이 있으면 무조건 CBO
- OPTIMIZER_MODE=First_rows|First_rows_n|All_rows일 경우 통계정보의 존재 여부와 관계 없이 무조건 CBO로 처리하려고 함.
  통계정보가 없다면 Heuristics Value를 이용하거나, Oracle9*i* Database 이상일 경우는 다이나믹 샘플링의 레벨에 따라 테이블의 데이타를 샘플링해서 CBO로 플랜이 결정된다. 그러나, 플랜이 비효율적일 수 있다.
- 통계정보가 있으나 옵티마이저 모드가 RULE일 경우, 다른 힌트가 오지 않은 경우와 Parallel Degree, Partition Table, SAMPLE절 등이 나오지 않은 경우는 RBO로 처리된다.

OPTIMIZER_DYNAMIC_SAMPLING (Default = 1(Oracle9*i* Database), 2(Oracle Database 10*g*))

더 나은 플랜을 결정하기 위한 목적으로 더 정확한 Selectivity & Cardinality를 구하기 위한 방법으로 0 ~ 10 레벨이 있으며, 레벨이 높을수록 SQL 문장의 실행 시점에 통계정보를 만들기 위해 테이블의 데이타를 샘플링하기 위한 추가적인 Recursive SQL이 발생된다. DYNAMIC_SAMPLING(0 ~ 10) 힌트를 통해서도 같은 기능을 할 수 있다. 그러나 내부적으로 추가적인 테이블 액세스의 비용이 발생하므로 OLTP에서는 주로 사용하지 않는다. 특히 OLTP 환경에서 레벨을 디폴트 값 이상 높여 놓지 않도록 한다. Oracle Database 10*g*의 경우 통계정보가 없다면 '다이나믹 샘플링' 이 적용된다.

다음은 Oracle Database 10*g*의 플랜 및 다이나믹 샘플링의 예이다.

```
SQL> analyze table dept delete statistics;

Table analyzed.

SQL> analyze table bigemp delete statistics;

Table analyzed.

SQL> explain plan for
        select * from bigemp e, dept d
       where e.deptno = d.deptno and
        d.deptno = 10;

Explained.

SQL>  select * from table(dbms_xplan.display);

PLAN_TABLE_OUTPUT

| Id | Operation          | Name   | Rows | Bytes | Cost(%CPU) | Time     |
|  0 | SELECT STATEMENT   |        | 11833 | 1352K |  176   (6) | 00:00:02 |
|  1 | MERGE JOIN CARTESIAN |      | 11833 | 1352K |  176   (6) | 00:00:02 |
|* 2 | TABLE ACCESS FULL  | DEPT   |     1 |    30 |    5   (0) | 00:00:01 |
|  3 | BUFFER SORT        |        | 11833 | 1005K |  171   (6) | 00:00:02 |
|* 4 | TABLE ACCESS FULL  | BIGEMP | 11833 | 1005K |  171   (6) | 00:00:02 |

Predicate Information (identified by operation id):

2- filter( "D" ." DEPTNO" =10)
4 - filter( "E" ." DEPTNO" =10)

Note
   - dynamic sampling used for this statement

20 rows selected.
```

통계정보가 없을 경우, 다이나믹 샘플링 기능을 확인하기 위해서 통계정보 삭제

Explain Plan으로 파싱 처리

플랜을 보기 위한 SQL

Oracle Database 10*g*부터는 비용 중 CPU 비중 및 예측실행 시간을 확인할 수 있다.

다이나믹 샘플링이 사용된 것을 확인할 수 있다.

< Dynamic Sampling에 의한 Recursive SQL문장 예 >

```
SELECT /* OPT_DYN_SAMP */ /*+ ALL_ROWS IGNORE_WHERE_CLAUSE
   NO_PARALLEL(SAMPLESUB) NO_PARALLEL_INDEX(SAMPLESUB) */ NVL(SUM(C1),0),
   NVL(SUM(C2),0)
FROM
  (SELECT /*+ IGNORE_WHERE_CLAUSE NO_PARALLEL( "E" ) FULL( "E" )
   NO_PARALLEL_INDEX( "E" ) */ 1 AS C1, CASE WHEN  "E" ." DEPTNO" =10 THEN 1 ELSE 0
   END AS C2 FROM  "BIGEMP"  SAMPLE BLOCK (4.225352 , 1) SEED (1)  "E" ) SAMPLESUB
```

CBO를 위한 통계정보 운영 방법

통계정보는 CBO의 플랜 결정에 사용되는 객체들의 물리적인 구성정보를 나타낸다. 즉, 테이블이 몇 블록으로 구성되어 있으며, 몇 건의 로우들을 가지고 있으며, 평균 로우 길이는 어느 정도이며, 칼럼의 Min/Max 값의 분포, Distinct 값, 인덱스의 레벨, 키(key)당 Leaf Block 수 등의 정보들을 나타낸다. 이들 정보는 CBO의 플랜 결정의 기초 자료로 사용된다. 이들 통계정보를 생성하기 위해서는 ANALYZE 명령어를 이용하거나 DBMS_STATS 패키지를 이용하면 된다. 그러나 2개의 차이점에 주의해야 하며, DBMS_STATS를 지원하는 Oracle8*i* Database 이상부터는 DBMS_STATS를 사용하기를 권장하고 있다.

Analyze 명령어와 DBMS_STATS의 차이점

- Analyze는 Serial Statistics Gathering 기능만 있는 반면, DBMS_STATS은 Parallel Gathering 기능이 있다.
- Analyze는 파티션의 통계정보를 각 파티션 테이블과 인덱스에 대해서 수집하고, Global Statistics는 파티션 정보를 가지고 계산하므로, 비정확할 수 있다. 그러므로 파티션 또는 서브파티션이 있는 객체에는 DBMS_STATS을 사용하여야 한다.
- DBMS_STATS은 전체 클러스터에 대해서는 통계정보를 수집하지 않는다. 그러므로 Analyze를 사용한다.
- DBMS_STATS은 CBO와 관련된 통계정보만을 수집한다. 즉, 테이블의 EMPTY_BLOCKS, AVG_SPACE,CHAIN_CNT 등은 수집되지 않는다.
- DBMS_STATS은 사용자가 지정한 통계정보 테이블에 수집된 통계정보를 저장할 수 있고, 딕셔너리로 각 칼럼, 테이블, 인덱스, 스키마 등을 반영할 수 있다.
- DBMS_STATS은 IMPORT/EXPORT 기능 및 추가적인 기능이 많다. 이 기능을 이용하여 운영 DB의 통계정보를 개발장비의 통계정보로 복사할 수 있으므

로 개발장비의 플랜을 운영장비와 같게 만들 수 있다(매뉴얼 참조).

다음은 Analyze 명령어에만 있는 기능이다.

• Structural Integrity Check 기능

```
analyze { index/table/cluster } (schema.){ index/table/cluster } validate structure (cascade) (into schema.table);
```

• Chained Rows 수집 기능

```
ANALYZE TABLE order_hist LIST CHAINED ROWS INTO <user_tab>;
```

시스템 통계정보 (>= Oracle9*i* Database)

시스템 통계정보는 객체의 통계정보와 같이 사용되는 정보로서, 기존의 Oracle8*i* Database까지의 I/O 중심의 플랜 방식에 CPU와 디스크 I/O 속도와 같은 시스템 자원의 효율을 반영하여 보다 효율적인 플랜을 결정하기 위한 방법으로, Oracle9*i* Database에서 처음 소개되었으며, 옵션 기능으로 DBA에 의해 사용될 수도 있고 사용하지 않을 수도 있었다. Oracle Database 10*g*에서는 시스템 통계정보가 기본적으로 수집되고 사용된다. 그러므로 기존의 I/O 횟수만 가지고 플랜을 결정하던 부분이 Oracle Database 10*g*에서는 시스템의 자원의 성능도 고려되어 보다 정확한 플랜을 결정할 수 있게 되었다.

이들 시스템 통계정보는 DBMS_STATS 패키지를 이용해서 수집된다. Oracle Database 10*g*에서는 기본적으로 수집되는 값들이 있으며, 또한 사용자가 수집해야 하는 항목도 있다. 다음은 Oracle Database 10*g*의 시스템 통계정보의 수집형태를 보여주고 있다.

```
SQL> select * from aux_stats$; -- System통계정보 확인
```

SNAME	PNAME	PVAL1	PVAL2	
SYSSTATS_INFO	STATUS		COMPLETED	
SYSSTATS_INFO	DSTART		09-30-2004 16:23	
SYSSTATS_INFO	DSTOP		09-30-2004 16:23	
SYSSTATS_INFO	FLAGS	0		
SYSSTATS_MAIN	CPUSPEEDNW	230.853	<<< Default (NOWORKLOAD상태 값)	
SYSSTATS_MAIN	IOSEEKTIM	10	<<< Default (NOWORKLOAD상태 값)	
SYSSTATS_MAIN	IOTFRSPEED	4096	<<< Default (NOWORKLOAD상태 값)	
SYSSTATS_MAIN	SREADTIM			디폴트로 수집된 시스템 통계정보. CPU Clock Speed, I/O Seek Time, I/O Transfer Time 등을 확인할 수 있다.
SYSSTATS_MAIN	MREADTIM			
SYSSTATS_MAIN	CPUSPEED			
SYSSTATS_MAIN	MBRC			
SYSSTATS_MAIN	MAXTHR			
SYSSTATS_MAIN	SLAVETHR			

```
SQL> EXECUTE dbms_stats.gather_system_stats(gathering_mode => 'START');
PL/SQL procedure successfully completed.
>>>> 이 시기 동안 발생된 워크로드를 분석해서 aux_stats$에 반영시킴.

SQL> EXECUTE dbms_stats.gather_system_stats(gathering_mode => 'STOP');
```

시스템 통계정보 수집의 시작과 종료. 시스템이 사용되는 시기에 일정시간 수집한다.

```
PL/SQL procedure successfully completed.

SQL> select * from aux_stats$;
```

SNAME	PNAME	PVAL1	PVAL2	
SYSSTATS_INFO	STATUS		COMPLETED	
SYSSTATS_INFO	DSTART		09-30-2004 17:02	
SYSSTATS_INFO	DSTOP		09-30-2004 17:04	
SYSSTATS_INFO	FLAGS	1		
SYSSTATS_MAIN	CPUSPEEDNW	243.637		
SYSSTATS_MAIN	IOSEEKTIM	13.776		
SYSSTATS_MAIN	IOTFRSPEED	4096		
SYSSTATS_MAIN	SREADTIM	9.167		Dbms_stats의 gather_system_stats를 통해 수집된 시스템 통계정보가 반영되었다.
SYSSTATS_MAIN	MREADTIM	9.091		
SYSSTATS_MAIN	CPUSPEED	238		
SYSSTATS_MAIN	MBRC	15		
SYSSTATS_MAIN	MAXTHR			
SYSSTATS_MAIN	SLAVETHR			

```
SQL> EXECUTE dbms_stats.DELETE_SYSTEM_STATS(); --- 수집된 시스템 통계정보 삭제
PL/SQL procedure successfully completed.
```

옵티마이저 원리에 바탕한 SQL 튜닝

지금까지 설명한 바와 같이 개략적으로나마 옵티마이저의 원리를 이해하기 위한 부분에 초점을 맞춰 설명하였다. 더 자세한 부분이 필요하면 오라클 매뉴얼인 [Database Performance Tuning Guide and Reference]를 권장하고 싶다. 다음 호엔 오라클이 사용하는 조인 방법들와 오라클이 제공하는 SQL 튜닝 방법에 대해 알아보도록 하자.