Oracle Technical Note

오라클 옵티마이저의 기본 원리

현재 모든 관계형 DBMS에서는 사용자의 SQL 질의를 효율적으로 처리하기 위해 옵티마이저를 사용하고 있다. 개발자나 관리자들이 이 옵티마이저의 기본 동작 원리를 이해한다면, 여러 면에서 도움이 되리라는 생각에 이 글을 쓰게 되었다. 먼저, 옵티마이저에 대한 기본적인 이해를 구한 다음, 오라클 옵티마이저에 대해 알아보도록 한다.

본론에 앞서, 이 글은 관계형 데이타베이스 개념에 일정 정도 익숙한 독자들을 대상으로 작성되었기 때문에, 초보자는 이해하기가 난해할 수도 있다는 점과 오라클 옵티마이저가 워낙 광범위한 기술이라 모든 세부적인 내용을 다 다룰 수는 없으며, 필자도 오라클 옵티마이저의 모든 내부 동작 원리를 완전히 이해하고 있지는 못하다는 점에 양해를 바란다. 미진하고 잘못된 부분은 모두 필자의 책임임을 밝혀 둔다.

글 : 이상원 (주)엑셈 연구소장 겸 성균관대학교 교수
| swlee@ex-em.com |

ORACLE®

관계형 DBMS와 옵티마이저

21세기에 들어서도 여전히 관계형 DBMS가 데이타베이스 시장을 지배하고 있다. 그리고, 관계형 DBMS에서 사용되는 핵심 언어는 SQL(Structured Query Language)이다. 이 SQL 언어의 가장 큰 특징은 사용자가 데이타베이스에서 자신이 원하는 데이타(What)만 지정하면, 그 데이타를 어떻게 구하는가(How)는 DBMS가 자동적으로 결정해서 처리해 준다는 점이다.

이런 면에서 SQL을 선언적(declarative) 언어(주 1)라고 부르며, 사용자는 데이타베이스의 물리적 구조의 변경에 상관 없이 항상 원하는 정확한 결과 데이타를 구할 수 있다. 이러한 물리적 데이타 독립성(physical data independence)이 관계형 DBMS를 상업적 성공으로 이끈 가장 큰 이유 중의 하나이다.

아무리 SQL이 이와 같은 장점을 갖고 있더라도, DBMS가 내부적으로 질의를 처리하는 방식이 비효율적이라면 관계형 DBMS는 누구도 사용하지 않을 것이다. 다행히도 현재의 모든 관계형 DBMS는 사용자의 SQL 질의를 효율적으로 수행하는 방법을 찾아내는 옵티마이저(Query Optimizer : 혹자는 '질의 최적화기'라고도 부르는데 이 글에서는 '옵티마이저'라 부르겠다.)를 제공하고 있다.

예를 들어, 다음과 같은 간단한 SQL 문을 보자.

```
Q1 :    select ename, sal
        from emp e, dept d
        where e.deptno = d.deptno and d.loc = 'SEOUL'
```

emp와 dept는 각각 deptno와 loc 칼럼에 대해 B 트리 인덱스가 있다고 가정한다. 이 같은 단순한 질의 Q1의 경우에도 질의 결과를 구하는 방법, 즉 실행 계획(execution plan)은 다양할 수 있다.

〈 그림 1 〉은 두 가지 실행 계획 P1과 P2를 보여주고 있다. P1은 우선 loc = 'SEOUL' 조건을 만족하는 dept 레코드를 인덱스를 이용해서 찾고, 각 dept 레코드에 대해 deptno 값이 일치하는 emp의 레코드를 인덱스를 이용해서 찾아 값을 출력한다(Nested Loop 조인 방법 이용).

〔 그림 1 〕 질의 Q1에 대한 두 가지 실행 계획의 예

한편, P2는 emp/dept 테이블을 Full Table Scan해서 이들을 Sort Merge 조인 방식으로 조인해서 질의 처리를 수행한다. 주목할 점은, 이 두 실행 계획 모두 정확한 질의 결과를 구하지만, 두 방식의 수행 시간에는 차이가 많이 날 수도 있다는 점이다. 예를 들어, P1 방식은 1초에 원하는 결과를 구하는 반면, P2 방식은 1시간이 걸릴 수도 있다.

P1, P2 이외에도 질의 Q1을 수행할 수 있는 많은 실행 계획이 있을 수 있다. 실행계획이란, 여러 개 테이블들의 조인에 대해, 특정한 1) 조인 순서(join ordering), 2) 조인 방법(join method), 그리고 3) 테이블 액세스 방법(access method)을 선택하는 것이다.

옵티마이저는 가능한 실행 계획들을 모두 검토하고, 이 중에서 가장 효과적으로, 즉 가장 빨리, Q1의 결과를 구할 수 있는 실행 계획을 결정한다. 이 글에서는, 옵티마이저가 최적의 실행 계획을 찾는 과정을 '질의 최적화(Query Optimization)' 또는 단순히 '최적화' 라고 부르겠다.

관계형 DBMS 옵티마이저의 핵심 기능

관계형 DBMS 옵티마이저의 핵심 기능은 다음과 같다.

- 실행 계획 탐색(Search Space Enumeration) : 주어진 SQL 질의를 처리할 수 있는 실행 계획들을 나열(P1, ..., Pn) ?
- 비용 산정(Cost Estimation) : 각 실행 계획의 예상 비용을 계산

많은 실행 계획들 중에서 최종적으로 가장 비용이 적게 드는 실행 계획 Pi를 선택해서 SQL을 실행하고 결과를 사용자에게 보여 준다.

실행 계획 탐색

예를 들어, 3개의 테이블, T1, T2, T3에 대해 조인을 수행하는 SQL 문이 있다고 가정하자. 그럼 이 질의를 수행할 수 있는 가능한 실행 계획은 몇 가지일까? 우리는 앞에서 조인 순서, 조인 방법, 그리고 테이블 액세스 방법에 따라 서로 다른 실행계획이 만들어진다고 했다. 그렇다면, 3개의 테이블 T1, T2, T3에 대한 조인 순서는 3!, 즉 6개의 조인 순서가 있다.

(T1§_T2)§_T3, (T1§_T3)§_T2, (T2§_T1)§_T3, (T2§_T3)§_T1, (T3§_T1)§_T2, (T3§_T2)§_T1

그리고, 하나의 조인 순서에는 2개의 조인을 포함하는데, 이용 가능한 조인 방법이 Nested Loop, Sort Merge, Hash Join의 세 가지가 있다면, 각 조인 순서에 대해 총 32, 즉 9개의 조합이 가능하다. 그리고, 이 각각의 경우 테이블을 접근하는 액세스 방법이 Full Table Scan과 Index Scan의 두 가지가 있다면 23, 즉 8개의 서로 다른 조합이 가능하다.

따라서, 3! x 32 x 23 = 432가지의 실행 계획이 가능하다. 그런데, 옵티마이저가 고려해야 할 실행 계획의 개수는 SQL에 포함된 테이블의 개수가 증가함에 따라 기하급수적으로 늘어나게 된다. 만일 from 절의 테이블의 개수가 5개인 경우, 5! x 35 x 25 = 933,120개가 가능해진다.

〈주 1〉 엄밀하게는 SQL은 절차적(procedural) 언어의 요소를 포함하고 있다. SQL은 이론적으로 관계대수(relational algebra)와 관계 해석(relational calculus)의 두 요소를 모두 포함하고 있고, 관계대수는 절차적 언어이다. 따라서, SQL은 '상대적으로(relatively)' 선언적 언어라 불러야 한다고 본다.

그리고, 여기서 각 실행 계획의 예상 비용을 계산하는 데 걸리는 시간이 0.01초라고 가정했을 때, 모든 실행 계획의 예상 비용을 구하는 데 약 9,300초(약 2시간 36분)이 걸린다. 만일 테이블의 개수가 10개라고 가정하면, 아마도 모든 실행 계획의 예상 비용을 계산하는 데만도 몇 년이 걸릴지도 모른다.

21세기 IT 환경에서는 하나의 SQL 문에 5 ~ 10개 정도의 테이블이 포함되는 경우가 일반적이다. 그런데, 옵티마이저가 실행 계획을 선정하는 데 걸리는 시간이 이와 같다면, 옵티마이저는 차라리 없는 것이 더 나을지도 모른다.

따라서, 옵티마이저는 모든 가능한 실행 계획을 다 고려할 수는 없다. 즉, 질의 최적화에 걸리는 시간을 줄이기 위해 어떤 실행 계획들은 아예 비용 계산에서 제외해야 할 필요도 있다. 옵티마이저는 모든 가능한 실행 계획 조합들을 탐색하는 방법 - 즉 어떤 실행 계획을 먼저 고려하고, 어떤 순서로 다음 실행 계획을 찾고, 어떤 실행 계획은 제외할 것인가? - 을 갖고 있어야 한다.

비용 산정

앞의 실행 계획 탐색 단계에서 만들어내는 각각의 실행 계획에 대해, 그 실행 계획을 실제로 수행할 때 비용 - 단순하게는 시간이 얼마나 걸릴지? - 을 예측해서, 가장 비용이 적은 실행 계획을 선택해야 한다.

이를 위해서 옵티마이저는 데이타베이스 내의 데이타들에 대해 갖고 있는 통계정보와 비용을 예측하는 다양한 모델을 사용해서 각 실행 계획의 비용을 계산할 수 있어야 한다.

여기서 주목할 점은, 옵티마이저가 실행 계획들을 비교할 때 사용하는 기준은 '예상 비용'이라는 점이다. 앞의 예에서 P1과 P2 방법을 실제로 수행해 보고 더 좋은 방법을 결정하는 것이 아니라, 옵티마이저가 갖고 있는 통계정보를 활용해서 P1과 P2로 수행했을 때 어느 실행 계획의 예상 비용이 작은가를 보고서 이를 실제로 수행하게 되는 것이다.

Selinger 스타일의 옵티마이저

필자가 아는 한에서, 현재의 모든 상용 관계형 DBMS의 옵티마이저는 IBM DB2의 모태인 System-R 프로토타입 시스템을 개발할 당시에 고안된 아키텍처에 기반하고 있다〈참고자료 4〉. 이 아키텍처를 주도적으로 제안한 IBM의 여성 전산학자 Pat. Selinger의 이름을 따서, 'Selinger 스타일 옵티마이저'라 부른다.

참고로 Pat. Selinger의 논문은 아직까지도 데이타베이스 분야에서 가장 많이 인용되는 논문 중의 하나이고, Pat. Selinger는 이 한 편의 논문으로 데이타베이스 연구 분야에서 슈퍼스타의 반열에 올라섰다.

이 Selinger 스타일 옵티마이저 아키텍처의 가장 큰 두 가지 특징은 1) 동적 프로그래밍 기반에 의한 실행 계획 탐색(Search Space Pruning based on Dynamic Programming)과 2) 비용 기반 최적화(Cost-Based Optimization)인데, 각각 위에서 나열한 옵티마이저 핵심 기능의 첫째, 둘째 기능에 해당된다.

옵티마이저의 이상과 현실

가장 이상적인 옵티마이저는, 모든 질의에 대해 옵티마이저가 선택한 실행 계획이 실제로 수행될 때도 가장 좋은 수행 속도를 보장하는 경우이다.

그러나, 현재의 옵티마이저는 비록 대부분의 경우에 상대적으로 아주 좋은 실행 계획〈주 2〉을 선택하지만, 실제로는 아주 나쁜 실행 계획을 선택하는 경우도 있다. 현재의 옵티마이저의 한계, 원인 그리고 앞으로의 개선 방향에 대해서는 뒤에서 자세히 설명하겠다.

옵티마이저는 30년 이상 축적된 기술을 포함하고 있는, 인간의 지능이 가장 많이 녹아 있는 복잡한 소프트웨어이다. 이 세상의 어떤 누구도 상용 관계형 DBMS 옵티마이저의 복잡한 내부 동작 원리를 완전히 이해하고 있는 사람은 없다고 단언할 수 있다.

실제로 필자도 이 글에서 주로 설명할 오라클 옵티마이저에 대해서 어쩌면 기본적인 지식밖에 없다고 할 것이다. 다만, 많은 오라클 개발자나 관리자들이 현대의 옵티마이저의 가장 기본적인 동작 원리를 쉽게 이해했으면 하는 마음으로 이 글을 썼다.

옵티마이저의 아키텍쳐

이제까지는 현재 보편적으로 사용되고 있는 관계형 DBMS 옵티마이저의 기능, 간단한 역사적 배경, 그리고 동작 원리에 대해 간략히 알아보았다. 이제부터는 이러한 배경 지식을 바탕으로 Oracle DBMS의 옵티마이저의 기본 구조와 동작 원리에 대해 알아보겠다.

오라클은 RBO(Rule-Based Optimization : 규칙기반 최적화)와 CBO(Cost-Based Optimization : 비용기반 최적화)를 모두 지원하고 있다. CBO의 경우, 1992년 Oracle 버전 7부터 도입되었는데, 향후 이 글에서 오라클 옵티마이저라 함은 CBO를 지칭하는 것이다. RBO는 간단한 규칙 위주로 최적화를 수행하는 방법으로, 앞으로는 널리 사용되지 않을 것이며, 오라클도 공식적으로 CBO 사용을 권장하고 있다.

| 그림 2 |는 오라클 옵티마이저의 아키텍처를 보여주고 있는데, 사용자의 SQL 질의는 크게 다음 4단계를 거쳐서 수행된다.

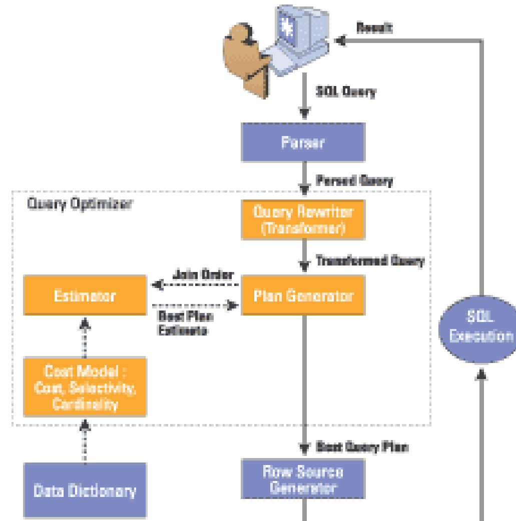
1. 파싱(Parser)
2. 옵티마이저(Query Optimizer)
3. 로우소스 생성(Row Source Generator)
4. SQL 실행(SQL Execution Engine)

파싱(Parser) 단계는 SQL은 구문(syntax)과 의미(semantics) 검사를 수행한다. 예를 들어, SQL 구문이 정확한지를 검사하고, 참조된 테이블에 대해 사용자의 접근 권한 등을 검사한다. 이 단계가 끝나면, SQL 문은 파싱 트리(parsed tree) 형태로 변형되어 옵티마이저에게 넘겨진다.

〈주 2〉 어떤 실험에서는 80% 정도의 경우, 옵티마이저가 선택하는 실행 계획이 실제 수행했을 때도 결과가 가장 좋았다. 나머지 20%의 경우에, 옵티마이저가 선택한 실행 계획은 평균적으로 세 번째로 시간이 많이 걸렸다. 이 실험 전체에 걸쳐서 가장 빠른 실행 계획과 옵티마이저가 선택한 실행 계획의 수행 시간은 1.33배 정도였다.

옵티마이저(Query Optimizer) 단계는 앞에서 넘겨받은 파싱 트리를 이용해서 최적의 실행 계획을 고른다.

| 그림 2 |에서 점선 형태의 사각형으로 표시된 부분이 옵티마이저의 주요 구성 요소를 보여주고 있는데, 뒤에서 각 구성 요소의 역할에 대해 자세히 설명하겠다.



〔 그림 2 〕 Oracle Query Optimizer의 아키텍처

로우소스 생성(Row Source Generator) 단계는 옵티마이저에서 넘겨받은 실행 계획을 내부적으로 처리하는 자세한 방법을 생성하는 단계이다. '로우 소스'란 실행 계획을 실제로 구현하는 인터페이스 각각을 지칭하는 말로, 테이블 액세스 방법, 조인 방법, 그리고 정렬(sorting) 등을 위한 다양한 로우 소스가 제공된다. 따라서, 이 단계에서는 실행 계획에 해당하는 트리 구조의 로우 소스들이 생성된다.

마지막으로, SQL 실행(SQL Execution Engine) 단계는 위에서 생성된 로우 소스를 SQL 수행 엔진에서 수행해서 결과를 사용자에게 돌려주는 과정이다.

여기서 한 가지 주목할 점은, 소프트 파싱(soft parsing)과 하드 파싱(hard parsing)은 크게 옵티마이저 단계의 포함 여부에 따른 차이이다. 즉, 소프트 파싱은 이미 최적화를 한 번 수행한 SQL 질의에 대해 옵티마이저 단계와 로우 소스 생성 단계를 생략하는 것이고, 하드 파싱은 이 두 단계를 새로 수행하는 것이다. 따라서, 하드 파싱은 통계정보 접근과 실행 계획 탐색 때문에 시간이 많이 걸린다. 이 차이가 주로 SQL 튜닝 전문가들이 가급적이면 하드 파싱을 피하라고 권하는 이유이다.

오라클 옵티마이저의 동작 원리

이제부터는 오라클 옵티마이저의 각 구성 요소의 기능에 대해 좀 더 자세히 알아보자. | 그림 2 |에서 보듯이, 오라클 옵티마이저는 크게 다음 3가지 모듈로 구성된다.

- 질의 변환(Query Rewriter)
- 실행 계획 생성(Plan Generator)
- 비용 산정(Estimator)

질의 변환 모듈

질의 변환(Query Rewriter 또는 Transformer) 단계는 파싱 트리(parsed tree)를 받아들여서 질의 변환을 수행한다. 이 변환 과정을 통해서 의미적으로 같은 결과를 수행하지만, 더 나은 실행 계획을 찾을 수 있는 SQL 문으로 변환함으로써 질의의 수행 처리 속도를 높이는 데 그 목적이 있다. 오라클 옵티마이저가 수행하는 질의 변환은 크게 다음 두 종류로 구분할 수 있다.

- 휴리스틱(Heuristic based) 질의 변환 : 이 변환의 종류로는 크게 View Merging, Subquery Unnesting, Predicate Push Down, Partition Pruning 등이 있는데, 이들 변환은 가능한 경우에 항상 질의 변환을 수행한다. 왜냐하면, 이와 같은 변환은 경험적으로 거의 항상 원래 질의보다 더 빠른 수행 속도를 보장하기 때문이다.
- 비용 기반(Cost based) 질의 변환 : 이 변환의 예로는, MV Rewrite, Star Query Transformation, OR-expansion 등을 들 수 있다. 그런데, 이 방법을 사용해서 변환된 SQL 문이 원래 SQL 문보다 속도가 더 빠르다는 보장이 없다. 따라서, 변환 전/후의 두 SQL 문에 대해 각각 최선의 실행 계획을 구하고, 이들의 비용을 비교해서 더 효율적인 실행 계획을 최종적으로 선택한다.

오라클의 질의 변환 모듈에서 지원하는 다양한 변환의 종류와 내용에 대해서는 〈참고자료 2, 3〉을 참고하기 바란다.

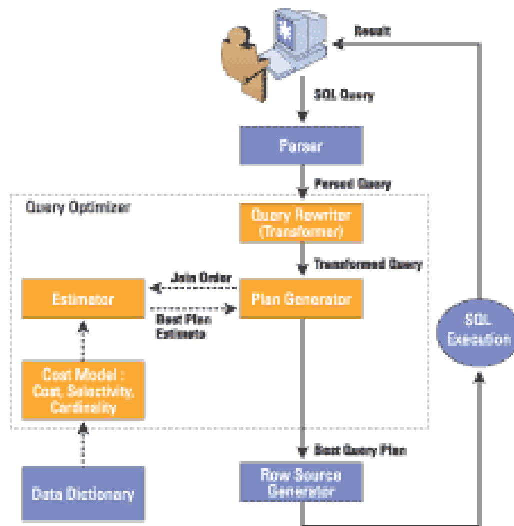
질의 변환 단계가 끝나면, 오라클 옵티마이저는 실행 계획 생성과 비용 산정 모듈을 수행하기 앞서, 질의에서 사용된 모든 테이블들과 각 테이블에 정의된 인덱스들에 관한 기본적인 통계정보들(예를 들어, 테이블의 블록 개수, 로우 평균 길이, 인덱스의 높이, 인덱스 리프 블록의 개수 등)과 각 테이블에 대한 다양한 액세스 경로(예를 들어, Full Table Scan, Index Scan 등)에 대한 비용 정보를 미리 구해 둔다.

옵티마이저의 아키텍쳐

실행 계획 생성 모듈

이 모듈은 옵티마이저가 새로운 실행 계획을 만드는 것이다. 오라클 옵티마이저는 제일 먼저 각 테이블의 레코드 수를 기준으로 오름차순으로 결정한다. 예를 들어, SQL 질의의 from 절에서 T1, T2, T3 순서로 참조한 경우, 각 테이블의 카디널러티(cardinality : 테이블의 튜플 수)가 T1 〉 T2 〉 T3 순이라면 제일 처음 고려하는 조인 순서는 (T3§_T2)§_T1 이 된다. 이 조인 순서에 대해서 다음 단계인 비용 산정 모듈을 호출해서 이 조인 순서에 따르는 실행 계획들과 각 실행 계획의 비용을 구한다.

그리고, 더 이상의 새로운 조인 순서가 없을 때까지 계속 새로운 조인 순서를 만들어서 비용을 계산한다. 이 모듈은 지금까지 찾아낸 가장 좋은 실행 계획과 그 비용을 저장하고 있다. 이 단계는 최종적으로 구해진 최적의 실행 계획을 | 그림 2 |의 로우 소스 생성 단계에 넘겨준다.



〔 그림 2 〕 Oracle Query Optimizer의 아키텍처

오라클의 실행 계획 생성 모듈은 테이블 개수가 정해진 값(디폴트 5)보다 작은 경우에는 모든 조인 순서에 대해 고려하지만, 테이블 개수가 이 값을 넘어서면 where 절에 명시적으로 조인 조건이 테이블들을 앞에 포함하는 조인 순서만 고려한다. 예를 들어, 6개의 테이블 T1, T2, ... , T6에 대한 조인을 수행하는 SQL 문에서 조인 조건이 T1, T2, T5, T6에 대해서만 주어졌다면, T1, T2, T5, T6가 먼저 조인되고, T3, T4는 항상 나중에 조인되는 조인 순서만 고려한다.

그런데, 이런 경우에도, 앞에서 언급한 것처럼, 테이블의 개수가 많으면 가능한 조인 순서의 조합이 기하급수적으로 늘어나게 된다. 이렇게 되면 옵티마이저 시간이 너무 많이 걸리기 때문에, 옵티마이저는 일정한 수(디폴트로는 최대 80,000)의 조인 순서에 대해서만 비용을 계산하고, 이 중에서 가장 최선의 실행 계획을 찾게 된다. 즉, 모든 가능한 조인 순서 조합들 중에서 일부분만 비용을 계산하고, 나머지는 고려하지 않는 것이다. 이를 실행 계획 탐색에 대한 '가지치기(pruning)' 또는 '컷오프(cutoff)'라 부른다.

그런데, 고려되지 않은 조인 순서 중에서 실제로 최선의 실행 계획이 포함되어 있을

수 있다. 옵티마이저가 제일 처음 고려하는 조인 순서를 테이블의 레코드 수의 오름차순 순서로 정하는 이유는 경험적으로 이 순서 근처에 실제로 최적의 실행 계획이 존재하기 때문이다. 이와 같이 초기 조인 순서를 선택하는 휴리스틱(heuristic)을 사용함으로써 임의로 조인 순서를 시작했을 때 최적의 좋은 실행 계획이 컷오프되는 것을 막을 수 있다.

오라클 옵티마이저 실행 계획 생성 모듈(Oracle9i부터 도입된)의 또 다른 특징은, 조인 순서를 바꾸어가면서 지금까지 구한 최적의 실행 계획의 예상 비용이 그리 크지 않은 경우, 최적화 단계를 일찍 끝내버린다.

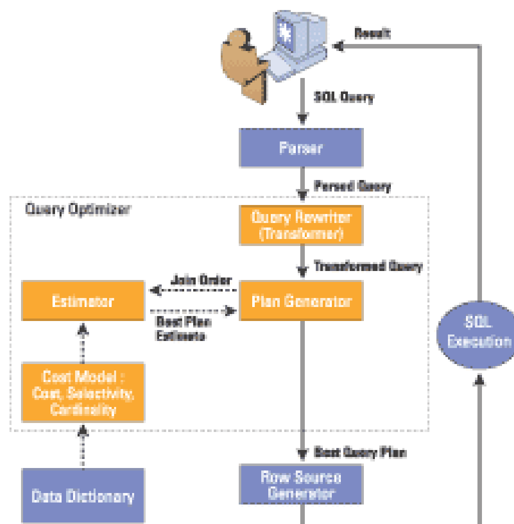
예를 들어, 어떤 질의에 대해 10초 동안 최적화를 수행해서 찾은 최적 실행 계획의 예상 수행 시간이 1분이면, 남은 조인 순서가 더 있더라도 옵티마이저 단계를 종료한다. 반면에, 지금까지 구한 최적 예상 수행 시간이 2시간이면, 더 나은 실행 계획을 찾기 위해 새로운 조인 순서에 대해 계속 탐색할 필요가 있다. 이를 '적응적 탐색 전략(adaptive search strategy)'이라 부른다.

비용 산정 모듈

자, 그럼 다음으로 비용 산정 모듈에 대해 알아보자. 실행 계획 생성 모듈에서 넘겨받은 특정 조인 순서의 각 조인에 대해 Nested Loop, Sort Merge, Hash Join 방식과 각 테이블의 다양한 액세스 방법을 반복 적용하면서 각 단계별로 비용을 계산해서 궁극적으로 해당 조인 순서에서 찾을 수 있는 최선의 실행 계획과 그 예상 비용을 구해서 실행 계획 생성 모듈에게 넘겨 준다.

현재의 조인 순서에 대해 중간 단계까지의 수행 비용이 실행 계획 생성 모듈에서 지금까지 구한 최선의 예상 비용보다 더 크다면, 해당 조인 순서에 대해서는 더 이상 비용 산정을 수행하지 않고 끝낸다. 예를 들어, T1, T2, T3에 대해 (T1§_T2)§_T3 순서에 대해 비용이 1000이었는데, (T§_T3)§_T2 순서의 (T1§_T3) 비용이 1200이었다면 더 이상 비용을 계산할 필요가 없다.

| 그림 2 |에 나와 있는 것처럼, 옵티마이저는 실행 계획의 비용을 계산하기 위한 비용 모델(Cost Model)을 갖고 있고, 이 비용 모델은 Oracle Data Dictionary에서 관리하는 다양한 통계정보를 기반으로 크게 다음과 같은 세 가지 값(measure)의 예상치를 계산한다.



〔 그림 2 〕Oracle Query Optimizer의 아키텍처

- 선택도(Selectivity) : Where 절에 있는 다양한 조건들의 선택도 계산
- 카디널러티(Cardinality) : 실행 계획상의 각각의 연산의 결과 카디널러티 수 계산
- 비용(Cost) : 실행 계획상의 각각의 연산을 수행하는 데 소요되는 시간 비용 계산

이 비용 산정을 위한 통계정보를 저장하는 Data Dictionary 테이블들은 DBA_TABLES, DBA_INDEXES, DBA_TAB_COL_STATISTICS, DBA_HISTOGRAMS 등이다. 이 통계정보는 앞에서 언급한 테이블의 액세스 경로의 비용 정보를 결정하는 데도 사용된다.

이들 테이블의 정보는 사용자가 ANALYZE 명령어나 DBMS_STATS 패키지를 이용해서 관리하게 되는데, 이들 테이블에서 관리하는 통계정보의 종류와 통계정보를 수집/관리하고, 통계정보를 확인하는 자세한 방법은 이 글에서는 설명하지 않겠다〈참고자료 5〉.

만일에 테이블과 인덱스에 대한 통계정보가 존재하지 않는 경우, 옵티마이저는 해당 테이블과 인덱스에 대해 디폴트로 가정하는 값들이 있다〈참고자료 5〉.

- 선택도

우선 선택도(selectivity)의 개념을 예로 들자. 앞에서 예로 든 질의 Q1에서 d.loc = 'SEOUL' 이라는 조건의 선택도는 dept 테이블 전체 중에서 loc의 값이 'SEOUL' 인 레코드의 비율을 일컫는다. 옵티마이저는 선택도 계산을 통해서 해당 조건을 만족하는 레코드가 몇 건 정도가 되는지를 예측하게 된다.

옵티마이저는 만일 DBA_TABLES에 dept 테이블의 loc 칼럼의 distinct column values가 10이라면 옵티마이저는 선택도가 0.1이라고 판단하게 된다. 이때 선택도를 이와 같이 정하는 이유는 dept 테이블이 loc 칼럼들에 골고루 분포되어 있다고 가정할 때 성립한다. 그러나, 실제로 loc 칼럼의 값들이 skew되어서 분포할 수도 있다.

예를 들어, 전체 레코드의 50%가 loc 값으로 'SEOUL' 을 갖는다면 잘못된 선택도 값을 얻게 된다. 이와 같이 데이타 분포가 skew되어 있는 경우, 해당 칼럼에 대한 히스토그램 정보를 DBA_HISTOGRAM 테이블에 만들어 주어야 정확한 선택도 값을 계산할 수 있다(이 경우는 0.5). 오라클 옵티마이저는 다양한 조건식의 종류에 대해 선택도를 통계정보에 기반해서 계산하는 수식을 내부적으로 갖고 있다.

그렇지만, 만일 dept 테이블이 아직 분석되지 않아서 통계정보가 없는 경우, 옵티마이저는 내부적으로 갖고 있는 디폴트 값을 선택도로 지정한다(예를 들어, 0.01).

- 카디널러티

앞의 dept 테이블의 전체 레코드 건수가 1000일 때, 앞에서 설명한 loc = 'SEOUL' 의 선택도가 0.1로 계산되었을 때, 조건을 만족하는 레코드 건수는 1000 x 0.1, 즉 100개로 예상할 수 있다. 이와 같이 어떤 연산을 수행한 결과로 나오는 레코드 건수를 '카디널러티(cardinality)' 라 하는데, 정확한 카디널러티를 계산하는 것은 좋은 실행 계획을 만드는 데 굉장히 중요하다.

예를 들어, (T1§_T2)§_T3 순서로 테이블을 조인할 때 (T1§_T2)의 결과와 T3를 조인할 때 어떤 조인 방법을 선택하는 것이 좋을지를 결정하기 위해서는 (T1§_T2)의 크기를 정확하게 알아야 한다. 이를 위해서는 (T1§_T2) 조인의 결과 레코드가 몇 개인지를 예상할 수 있어야 한다.

이를 위해 오라클 옵티마이저는 다양한 연산의 결과 레코드의 카디널러티를 통계정보와 수식에 의해서 계산한다. T1과 T2의 조인 조건이 T1.c1 = T2.c2(이를 'P' 라 표기)라 했을 때, 앞에서 설명한 선택도 계산 공식에 의해 이 조건식의 선택도 Sel(P)를 먼저 계산한 후, 이 조인의 결과 카디널러티는 Card(T1) x Card(T2) x Sel(P)가 된다.

예를 들어, T1, T2의 튜플 수가 각각 1000, 5000이고 Sel(P)가 0.01이면, 조인의 결과로 생기는 튜플 수는 1000 x 5000 x 0.01 = 5000이 된다. 그런데, Sel(P)가 조금이라도 틀리면 이후의 전체적인 비용 산정이 잘못되게 된다. 오라클 옵티마이저는 다양한 종류의 연산에 대해 내부 공식을 사용해 카디널러티를 계산한다.

- 비용

비용(cost)은 테이블 액세스, 조인 등을 수행하는 데 걸리는 시간을 의미하는데, 시간은 주로 디스크 I/O 수와 CPU 사용시간을 고려한다. 비용은 앞에서 계산한 통계정보와 내부 계산식에 의해 계산된다.

예를 들어, T1§_T2를 Nested Loop 방식으로 조인할 경우 조인비용은 (T1의 데이타 블록수) + ((T1의 레코드 건수)*(T2의 액세스 비용))이 된다. 이처럼 오라클 옵티마이저는 모든 연산에 대해 소요되는 비용을 계산하는 수식을 갖고 있다.

오라클 옵티마이저는 이 세 가지 예상 값(measure)을 기반으로, 현재의 실행 계획의 예상 비용을 구한다.

오라클 옵티마이저와 관련한 몇 가지 유용한 기능

이상에서 오라클 옵티마이저의 내부 동작 원리를 살펴보았다. 옵티마이저와 관련해서 오라클에서 제공하는 몇 가지 유용한 기능들에 대해서 간단히 알아보자(이들 기능에 대한 자세한 설명은 〈참고자료 3〉을 보기 바란다).

이 기능들은 크게 두 가지 - 즉, 옵티마이저가 사용할 통계정보를 수집/관리하는 기능과 옵티마이저의 활동을 자세히 추적할 수 있는 기능 - 로 구분할 수 있다.

먼저, 통계정보 수집/관리 기능으로는 ANALYZE 명령과 DBMS_ STATS 패키지를 들 수 있다. 비용 계산이 최적의 실행 계획을 구하는 데 중요한 역할을 하기 때문에, 이 기능을 사용해서 1) 어떤 테이블이나 칼럼에 변경사항이 많이 발생해서 새로 통계정보를 분석해야 하는지, 2) 어떤 칼럼에 대해 히스토그램을 만들어야 하는지에 대한 도움을 받을 수 있다.

다음으로 옵티마이저의 활동 추적과 관련해서, 1) 주어진 질의에 대해서 어떠한 최적화 과정을 거쳤는지, 2) 최종적으로 어떤 실행 계획을 선택했는지, 그리고, 선택된 실행 계획대로 수행했을 때 걸리는 시간과 자원이 얼마나 소요되었는지를 확인하는 기능들이 제공된다. 우선 옵티마이저의 최적화 과정을 확인하려면, 'Event 10053' 을 이용하면 된다. 이를 위해서는 SQL*Plus에서 다음 alter 명령을 수행하면 된다.

```
SQL> alter session set events '10053 trace name context
        forever';
```

이 명령을 수행하고 나면, 현재 세션에서 수행하는 모든 질의에 대해 옵티마이저의 최적화 과정의 모든 정보가 $ora_home/admin/udump /xxx.trc 파일에 기록된다. 이 트레이스 파일은 오라클 옵티마이저가 고려한 모든 조인 순서, 조인 방법, 테이블 액세스 방법, 선택도, 카디널리티, 비용 정보 등을 포함하고 있다.

다음으로, 단순히 옵티마이저가 최종적으로 선택한 실행 계획만 확인하고 싶으면, EXPLAIN 명령어나 SQL*Plus에서 제공되는 Autotrace 기능을 이용하면 된다. 그리고, 옵티마이저가 선택해서 수행한 실행 계획의 자세한 성능 정보를 알고 싶으면, SQL Trace, TKPROF 등의 기능을 이용하면 된다.

오라클 옵티마이저의 한계와 그 원인

오라클 옵티마이저를 포함한 현재의 관계형 DBMS의 옵티마이저는 항상 최적의 실행 계획을 고르지는 못한다. 옵티마이저의 한계는 실행 계획 생성 모듈과 비용 산정 모듈의 동작 원리에 그 원인이 숨어 있다.

실행 계획 생성 모듈의 제약

우선, 실행 계획 생성 모듈에서 최적화를 위해 사용할 수 있는 시간이 제한적이라는 점이다. 앞서 살펴보았지만, 10개 이상의 테이블 조인을 포함하는 질의의 경우 최적의 실행 계획을 구하기 위해 옵티마이저가 고려해야 할 탐색 공간이 너무 많기 때문에 다양한 형태의 컷오프를 수행한다. 이 과정에서 실제로 최적의 실행 계획이 고려되지 않고 잘려나갈 수 있다.

비용 산정 모듈의 불완전성

비용 산정 모듈에서는 특정 실행 계획의 비용을 통계정보와 내부적인 비용 산정 모델을 사용해서 계산한다. 그런데, 옵티마이저에서 사용하는 통계정보와 비용 산정 모델이 불완전하다. 따라서, 옵티마이저는 불완전한 정보를 바탕으로 일종의 추측(또는 가정)을 하는 것이다.

예를 들어, d.loc = 'SEOUL'의 선택도를 구하는 과정을 보자. d.loc에 대한 히스토그램 통계정보가 없으면, 옵티마이저는 '모든 값들이 골고루 분포되어 있다'는 가정하에 d.loc 칼럼의 distinct value 개수(이를 NDV라 하자)를 기준으로 해당 조건의 선택도를 1/NDV로 계산한다. 그러나, 실제로는 d.loc에 대해 skew된 분포를 보이면 옵티마이저의 실행 계획 비용 산정이 틀려지게 된다.

비용 산정 모듈이 한계를 갖게 되는 또 다른 예로, SQL에서 바인드 변수(bind variables)의 사용을 들 수 있다. 질의 Q1에서 d.loc = 'SEOUL' 대신에 d.loc = :loc_name 조건이 사용되었으면, 데이타베이스에 loc 칼럼에 아무리 정확한 통계치를 갖고 있어도 선택도에 대해서는 일정한 비율을 가정할 수밖에 없다.

다른 예로서, 다음 질의를 살펴보자.

```
Q2:    select *
       from emp
       where job_title = 'vice_president' and salary < 40000
```

데이타베이스에는 job_title, salary 칼럼 모두에 대해 정확한 히스토그램 정보를 유

지하고 있고, job_title = ‘vice_president’의 선택도가 0.05이고, salary 〈 40000의 선택도는 0.4였다. 이때 옵티마이저는 emp 테이블에서 where 절의 조건의 전체 선택도를 0.05 x 0.4, 즉 0.02로 계산한다. 이는 ‘각 칼럼의 값들의 분포는 서로 독립적이다’는 가정에 기반하다.

그러나, 부사장이면서 연봉이 40,000 이하인 경우는 거의 없기 때문에 실제 선택도는 0에 가까울 것이다. 결국 앞의 가정은 이와 같이 서로 밀접한 상관관계가 있는 두 칼럼에 대한 선택도를 구할 때 문제가 되는 것이다.

마지막 예로, 조인 연산에 대한 비용을 예측할 때, 이 조인을 수행할 수 있는 메모리 공간을 고정 크기로 가정하고, Nested Loop, Sort Merge, Hash Join의 비용을 산정한다. 그러나, 질의를 수행할 때 실제 비용은 이용 가능한 메모리의 양에 따라 크게 차이가 날 수 있다.

결론적으로, 옵티마이저의 정확도는 비용 계산의 정확도에 따라 좌우되는데, 참고하는 통계정보가 부족하거나 계산 과정의 몇 가지 가정들이 실제 데이타 분포와 실행 계획의 런타임 환경과 차이가 있기 때문에 정확도에 문제가 발생하는 것이다.

힌트 기능을 이용한 옵티마이저 동작 제어

이와 같은 현재의 옵티마이저의 한계를 보완하기 위해, 오라클에서는 SQL에 힌트(hint)를 추가해서 사용자가 옵티마이저가 선택하는 실행 계획에 영향을 줄 수 있도록 하고 있다.

옵티마이저가 최선이 아닌 차선의 실행 계획을 선택하는 경우, 사용자가 SQL에 힌트를 주어서 실행 계획을 베스트 플랜으로 만들도록 하는 것이 목적이다. 힌트를 제공하는 것이 옵티마이저의 기능이 떨어지는 것을 의미하는 것은 아니다. 어떤 DBMS의 옵티마이저도 완전할 수는 없기 때문에, 힌트 기능의 제공은 반드시 필요하다.

옵티마이저의 실행 계획은 결국 조인 순서, 조인 방법, 테이블 액세스 경로를 결정하는 것이기 때문에, 힌트의 종류도 크게 이 세 가지를 제어하는 것으로 구분할 수 있다.

다음 예는 Q1에 대해 힌트를 사용한 예를 보여주고 있는데, 처음 ‘ordered’는 from 절에 나와 있는 순서대로 조인 순서를 정하는 것이고, ‘use_nl’의 경우 dept 테이블을 inner table로 사용할 때 Nested Loop 방식만을 사용하도록 지정하고, full(e)는 emp 테이블은 항상 Full Table Scan을 통해서 액세스하도록 지정하는 것이다.

오라클에서 제공하는 다양한 힌트의 종류와 자세한 의미에 대한 설명은 〈참고자료 5〉를 보기 바란다.

```
Q1 :    select /*+ ordered use_nl(d) full(e) */ ename, sal : Oracle 힌트 기능 사용 예
        from emp e, dept d
        where e.deptno = d.deptno and d.loc = 'SEOUL'
```

오라클 SQL에서 힌트를 제공하는 또 다른 목적은, 사용자가 다양한 실행 계획을 수행해 봄으로써 어떤 데이타 액세스 경로가 도움이 되는지를 테스트해 볼 수도 있다. 힌트의 사용은 아주 불가결한 경우 말고는 사용을 조심해야 한다. 실제로 이 힌트 기능이 남용되는 경우가 많다.

힌트를 사용하게 됨으로써 데이타베이스 환경의 변화(예를 들어, 테이블의 크기 변화, 인덱스의 추가/삭제)가 발생할 때 옵티마이저가 더 나은 실행 계획을 선택하는 것을 방해할 수도 있다. 실제로 Oracle E-Business Suite 11i의 경우 포함된 27만 개의 SQL 중에서 0.3%만이 힌트를 포함하고 있다고 한다.

SQL 튜닝과 옵티마이저의 관계

SQL 튜닝은 특정 SQL 질의의 수행 시간을 단축하기 위해 사용자가 취하는 다양한 방법을 통칭한다. SQL 튜닝의 범위는 굉장히 포괄적인데, 옵티마이저와 관련한 방법으로는 SQL 재작성, 힌트 사용, 새로운 인덱스 추가, 통계 데이타의 추가/갱신 등을 통해서 옵티마이저가 더욱더 효율적인 실행 계획을 생성하도록 하는 것이다.

- SQL 재작성

사용자가 원하는 데이타를 질의하는 방법은 실제로 매우 다양할 수 있다. 극단적인 예로, C. J. Date는 한 SQL 문을 50가지 이상의 다른 SQL 문으로 표현이 가능함을 보여 준다(http://www.dbpd.com/vault/9807xtra.htm 참조). SQL 재작성을 통한 SQL 튜닝은 원래의 SQL 문을, 같은 결과를 내지만, 옵티마이저가 더 효과적인 실행 계획을 생성할 수 있는 SQL 문으로 바꾸는 방법이다.

- 힌트 사용

앞에서 언급한 것처럼, 힌트 기능을 사용해서 옵티마이저가 선택하는 실행 계획을 바꾸는 방법이다.

- 새로운 인덱스 추가

SQL 문의 효율적인 처리를 위해서는 특정 테이블의 특정 칼럼 값을 이용해서 해당 데이타를 빨리 찾아야 하는데, 인덱스가 없기 때문에 옵티마이저가 어떤 실행 계획을 선택하더라도 그 SQL 문은 느릴 수밖에 없는 경우가 있다. 이와 같은 상황에서는 새로운 인덱스 생성을 통해서 옵티마이저가 해당 인덱스를 이용하는 새로운 실행 계획을 선택하도록 할 수 있다.

- 통계 데이타의 추가/갱신

앞에서 설명한 것처럼, 오라클 옵티마이저의 비용 산정 모듈에서는 테이블, 칼럼, 인덱스 등에 대한 통계정보를 이용해서 선택도, 카디널러티 등을 구하고 이를 통해서 궁극적으로 실행 계획의 비용을 계산한다.

그런데, 만일 특정 테이블/칼럼에 대한 통계정보가 없거나, 오래 전에 만들어진 경우는 비용 계산이 부정확하게 되고, 따라서 옵티마이저가 선택하는 실행 계획이 실제로는 안 좋은 실행 계획일 수가 있다.

이를 해결하기 위해서는 특정 통계정보를 추가하거나 새로 갱신해 주어서 옵티마이저가 정확한 비용 산정을 통해서 더 나은 실행 계획을 선택하도록 해주는 방법이다.

옵티마이저 기술의 발달은 궁극적으로 SQL 튜닝 관련 직종을 없앨 수도 있다. 그러나, 다행인지 불행인지 몰라도, 향후 10년 사이에 이런 일이 벌어지지는 않을 것 같다.

향후 옵티마이저의 기술의 발전 방향

비록 옵티마이저 기술은 지난 30년간 꾸준히 발전해오면서 인간이 만든 가장 지능적인 소프트웨어이지만, 앞으로도 끊임없이 기술 발전이 필요한 분야이다.

사용자가 사용하는 SQL 질의가 점점 더 복잡해지고, 데이타베이스에서 다루는 데이타 양이 엄청난 속도로 늘어나고, 새로운 데이타의 종류를 데이타베이스에서 다루어야 하기 때문에, 옵티마이저 기술의 중요성은 더욱 더 커질 것이다.

향후 옵티마이저 기술의 주요 발전 방향은 다음의 분야가 될 것이다. 여기서 나열한 분야는 주로 학계에서 연구가 활발히 진행중이거나 많은 진전이 있는 분야를 중심으로 판단한 필자의 개인적인 의견이다.

- 질의 변환

오라클 옵티마이저와 관련해서 간략히 설명했지만, 현재의 옵티마이저가 처리하는 질의변환의 형태는 상대적으로 정형화되고 단순한 형태의 질의 변환만을 주로 수행한다. 단일 SQL 블록(Select-From-Where)에 대한 질의변환 이외에, 복잡한 중첩 질의(nested query)를 단일 질의로 변환하는 방법, 중첩질의 내의 각 SQL 블록을 결합한 효과적인 실행 계획 생성이 가능해질 것이다.

- 비용 산정을 위한 정확한 통계정보 관리

실행 계획에 대한 정확한 비용 산정이 옵티마이저 기술의 핵심이다. 따라서, 지금 옵티마이저가 참고하는 통계정보보다 더 정교하고 복잡한 통계정보의 관리/유지 기법들이 도입될 것이다.

예를 들어, 애트리뷰트 값들의 분포가 서로 독립적이라는 가정 대신에 상호 연관성이 깊은 칼럼들에 대한 히스토그램 정보를 효과적으로 수집/활용하는 기술이 도입될 수도 있다.

- 런타임시 동적 질의 최적화

현재의 옵티마이저 기술은, 질의 수행 환경에 상관없이 옵티마이저가 선택한 실행 계획을 그대로 실행한다는 측면에서 정적(static) 질의 최적화 방법이다. 앞으로는 고정된 실행 계획을 그대로 수행하는 것이 아니라, SQL 실행 단계의 상황에 따라 실행 계획을 융통성 있게 바꾸는 기술이 개발될 것이다. 실제로 Oracle9i의 경우 초보적인 형태의 동적 질의 최적화 기능을 제공하고 있다.

- 학습하는 옵티마이저

현재의 옵티마이저 기술은 통계정보의 변화나 사용자의 힌트가 없다면, 같은 SQL 질의에 대해서는 항상 똑같은 실행 계획을 선택할 것이다. 옵티마이저가 특정 질의에 대해 생성한 실행 계획을 실제 수행했을 때, 예상과 달리 좋지 않은 성능을 보이면, 옵티마이저가 다음 번에 최적화를 수행할 때는 이전의 실행 계획을 제외한 다른 대안을 찾게 되는 것이다.

참고자료

1. Ken Jacobs, Query Optimization, Oracle Magazine 2002 July/August (http://www.oracle.com/oramag/oracle/02-jul /o42dba.html)

2. Oracle Corp., Query Optimization in Oracle9i, Technical White Paper (http://otn.oracle.com/products/bi/pdf/o9i_optimization_twp.pdf)

3. Oracle Corp., Oracle9i Performance Tuning Guide and Reference Release 2(9.2) (http://otn.oracle.com/docs/products/oracle9i/doc_library/release2/server.920/a96533/toc.htm)

4. Patricia Selinger 외 다수, Access Path Selection in a Relational Database System, ACM SIGMOD 1979

5. Surajit Chaudhuri, An Overview of Query Optimization in Relational Systems, AMM PODS Tutorial 1998v

6. Yannis E. Ioannidis, Query Optimization, Handbook for Computer Science(Chapter 45), CRC Press

7. Hector Garcia-Molina, Jeffrey D. Ullman, Jennifer Widom, Database Systems: The Complete Book, Prentice Hall, 2001

8. Raghu Ramakrishnan and Johannes Gehrke, Database Management Systems(2nd Edition), McGraw-Hill, 1999

9. Abraham Silberschatz, Henry F. Korth and S. Sudarshan, Database System Concepts(4th edition), McGraw-Hill, 2001

한국오라클(주)
서울특별시 강남구 삼성동 144-17
삼화빌딩
대표전화 : 2194-8000
FAX : 2194-8001

한국오라클교육센타
서울특별시 영등포구 여의도동 28-1
전경련회관 5층, 7층
대표전화 : 3779-4242~4
FAX : 3779-4100~1

대전사무소
대전광역시 서구 둔산동 929번지
대전둔산사학연금회관 18층
대표전화 : (042)483-4131~2
FAX : (042)483-4133

대구사무소
대구광역시 동구 신천동 111번지
영남타워빌딩 9층
대표전화 : (053)741-4513~4
FAX : (053)741-4515

부산사무소
부산광역시 동구 초량동 1211~7
정암빌딩 8층
대표전화 : (051)465-9996
FAX : (051)465-9958

울산사무소
울산광역시 남구 달동 1319-15번지
정우빌딩 3층
대표전화 : (052)267-4262
FAX : (052)267-4267

광주사무소
광주광역시 서구 양동 60-37
금호생명빌딩 8층
대표전화 : (062)350-0131
FAX : (062)350-0130

고객에게 완전하고 효과적인
정보관리 솔루션을 제공하기 위하여
오라클사는 전 세계 145개국에서
제품, 기술지원, 교육 및
컨설팅 서비스를
제공하고 있습니다.

http://www.oracle.com/
http://www.oracle.com/kr