관계형 DBMS 옵티마이저의 한계와 해결 방안

글 | 이상원<성균관대학교 교수> swlee@ex-em.com

관계형 DBMS의 두뇌 역할을 담당하는 비용 기반 옵티마이저(Cost Based Optimizer : CBO)의 근본적인 한계들을 정확히 아는 것은 필요한 경우에 SQL의 정확한 튜닝을 위해서 반드시 필요하다. 이 한계들을 극복하기 위해 관계형 DBMS 벤더들은 꾸준한 기능 개선 노력을 하고 있지만, 해결책이 결코 쉽지 않다. 이 글은 SQL 튜닝 전문가나 SQL 개발자 등 사용자 입장에서 이 문제의 근본적인 이유를 제대로 이해하고, 이를 해결하기 위해 어떤 기능들이 개별 DBMS에서 제공되는지를 파악하며, 현재 관계형 DBMS의 옵티마이저의 한계를 해당 기능들을 이용해서 어떻게 극복할 것인지 대해 체계적으로 생각하는 방법을 제공할 것이다.

관계형 DBMS 옵티마이저는 기본적으로 통계정보, 선택도, 카디널리티, 비용 순으로 특정 실행계획(Execution Plan)의 수행 비용을 예측한다. 이 글에서는 이 관계형 DBMS의 옵티마이저가 비용을 계산하는 과정에서 발생할 수 있는 오류의 종류와 원인을 체계적으로 알아보고, 각각의 오류에 대해 일반적인 해결방안을 설명하겠다. 이 글은 이 한계에 대한 완전한 해결책을 제시할 수는 없지만, 사용자 입장(특히 SQL 튜닝 전문가나 SQL 개발자)에서 1) 이 문제의 근본적인 이유를 제대로 이해하고, 2) 이를 해결하기 위해 어떤 기능들이 개별 DBMS에서 제공되는지를 파악하며, 3) 현재 관계형 DBMS의 옵티마이저의 한계를 해당 기능들을 이용해서 어떻게 극복할 것인지 대해 체계적으로 생각하는 방법을 제공할 것이다.

또한 이 한계들을 극복하기 위해서 최근 대두되고 있는 '스스로 학습하는 옵티마이저'의 기본적인 프레임워크와 이를 구현하는 방안들에 대해 오라클 DBMS를 중심으로 설명할 것이다.

이 글의 이해를 돕기 위해

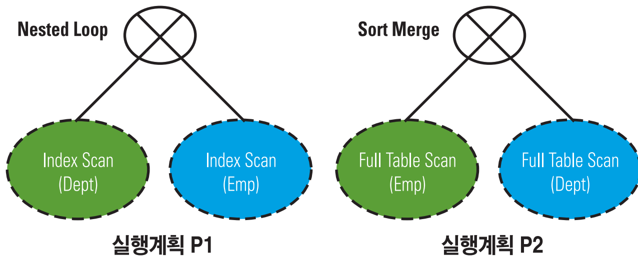
- **주 1** : 이 기사는 관계형 데이타베이스 개념에 일정 정도 익숙한 독자들을 대상으로 작성되었기 때문에 초보자는 이해하기가 어려울 수도 있다.
- **주 2** : 여기서 소개하는 관계형 DBMS의 한계는 특정 DBMS의 문제가 아니라, 상용 DBMS별로 조금씩 차이는 있겠지만, 거의 모든 DBMS에 공통적으로 나타나는 문제점이다.
- **주 3** : 여기서 주로 관계형 DBMS 옵티마이저 기술의 근본적인 한계를 논하겠지만, 그렇다고 해서 이 글이 관계형 DBMS 옵티마이저 기술을 폄하하는 것은 절대 아니다. 또한 특정 DBMS의 경우 규칙기반 옵티마이저(Rule Based Optimizer) 모드도 제공하는데, 여기서 비용기반 옵티마이저(Cost Based Optimizer)의 한계를 언급한다고 해서 비용기반 옵티마이저가 규칙 기반 옵티마이저에 비해 나쁘다고 단정하는 것은 아니다. 반대로, 규칙 기반 옵티마이저는 앞으로는 복잡한 SQL을 제대로 최적화하지 못하고, 또한 새로 도입된 관계형 DBMS의 최신 기술들을 활용할 수 없기 때문에 머지않아 사라지게 될 기술이라 본다.
- **주 4** : 이 기사는 관계형 DBMS의 아주 근본적인 문제점들의 기본 원인을 설명한다. 그런데 이 기술은 워낙 광범위하고 복잡한 기술이라 모든 세부적인 내용을 다 다룰 수도 없을 뿐더러, 필자도 관계형 DBMS 옵티마이저의 모든 사항을 완전히 이해하고 있지 않다. 미진하고 잘못된 부분은 모두 필자의 책임임을 밝혀 둔다.

관계형 DBMS 옵티마이저

현재의 모든 관계형 DBMS는 사용자의 SQL 질의를 효율적으로 수행하는 방법을 찾아내는 옵티마이저(Query Optimizer, '질의 최적화기'라고도 한다)를 제공하고 있다. 예를 들어, 다음과 같은 간단한 SQL문을 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 실행계획은 다양할 수 있다. <그림 1>은 두 가지 실행계획 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(중첩루프(Nested Loop) 조인 방법 이용). 한편, P2는 emp/dept 테이블을 전체 테이블 스캔해서(Full Table Scan) 이들을 정렬합병(Sort Merge) 방식으로 조인해서 질의 처리를 수행한다. 주목할 점은, 이 두 실행계획 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.



<그림 1> 질의 Q1에 대한 두 가지 실행계획 예

P1, P2 이외에도 질의 Q1을 수행할 수 있는 많은 실행계획이 있을 수 있다. 실행계획이란, 여러 개 테이블들의 조인에 대해, 특정한 조인 순서(Join Ordering), 조인 방법(Join Method), 그리고 테이블 액세스 방법(Access Method)을 선택하는 것이다. 옵티마이저는 가능한 실행계획들을 모두 검토하고, 이 중에서 가장 효과적으로, 즉 가장 빨리 Q1의 결과를 구할 수 있는 실행계획을 결정한다. 옵티마이저가 최적의 실행계획을 찾는 과정을 '질의 최적화(Query Optimization)' 또는 단순히 '최적화'라고 한다. 관계형 DBMS의 옵티마이저의 기본 원리에 대해서는 본지의 2002년 겨울호에 본 저자가 기고한 글[참고문헌 1]과 거기에 언급된 여러 문헌들을 참고하기 바란다.

<그림 2>는 관계형 DBMS 옵티마이저의 내부 아키텍처를 나타낸 것으로, SQL 질의는 크게 다음 4단계를 거쳐서 최적화된다(<그림 2>에서 번호가 매겨진 부분).

1. SQL : 사용자가 입력한 SQL
2. 실행계획 생성(Plan Generator) : 주어진 SQL을 수행할 수 있는 실행계획들 (P1, P2, ..., Pn)을 생성
3. 비용산정(Cost Estimator) : 각 실행계획에 대해 옵티마이저의 비용산정(Cost Estimator) 모듈이 예상비용을 계산
4. 실행계획 선택 : 각 실행계획들의 비용을 비교해서 가장 좋은 비용을 선정

이와 같이, 다양한 실행계획들에 대해 각 실행계획의 비용을 예측해서 최선의 실행계획을 선택하는 비용 기반 최적화(Cost Based Optimization) 기법이 Oracle, DB2, MS SQL Server를 포함한 모든 상용 DBMS 옵티마이저의 기본 아키텍처이다. 이 아키텍처는 관계형 DBMS의 옵티마이저는 IBM DB2의 모태인 System-R 프로토타입 시스템을 개발할 당시에 처음으로 고안되었고, 이 아키텍처를 주도적으로 제안한 IBM의 여성 전산학자 Pat. Selinger의 이름을 따서, 'Selinger-스타일 옵티마이저'라 부른다([참고문헌 2], 본격적으로 관계형 DBMS의 옵티마이저에 대해 깊이 있게 공부하고 싶은 독자들은 이 논문을 반드시 읽어야 한다).

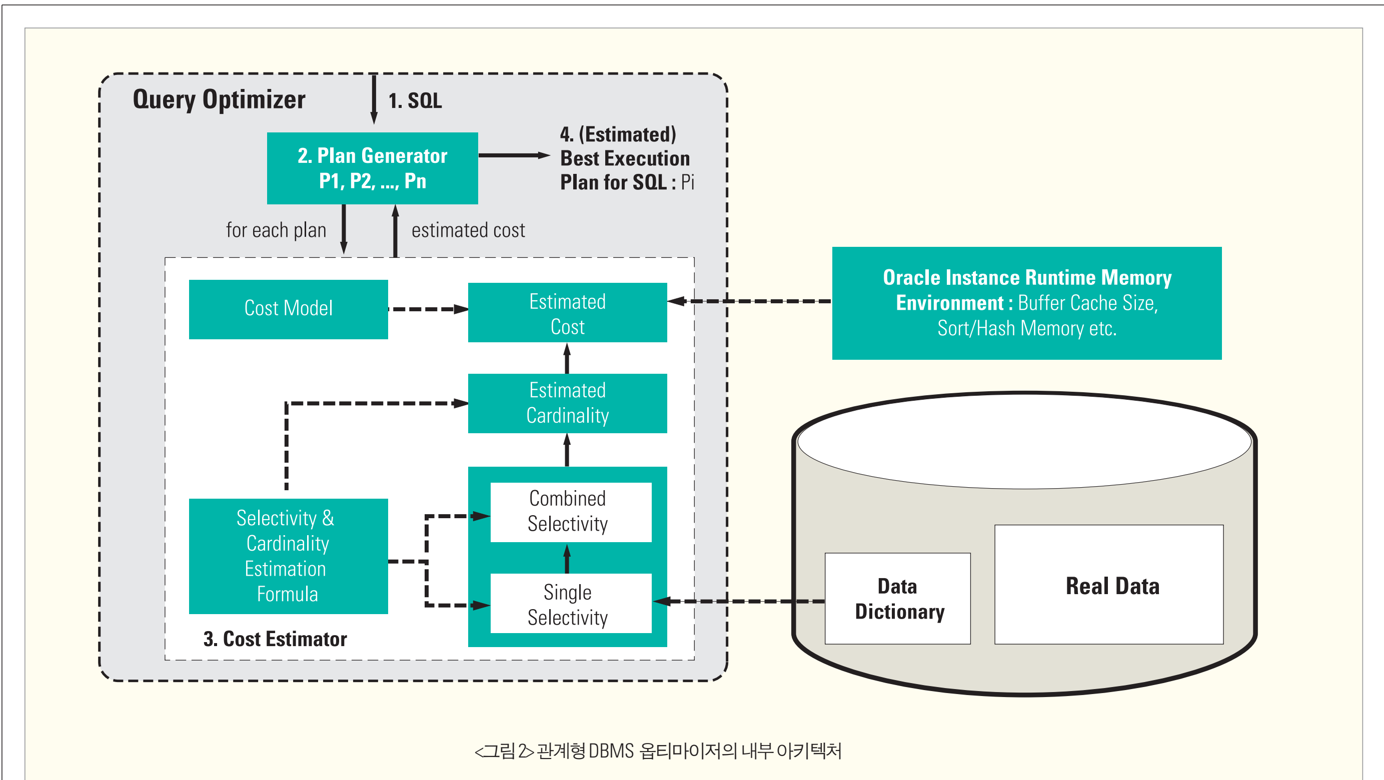
옵티마이저의 생명 - 비용산정의 정확성

그런데 여기서 주목할 점은, 옵티마이저가 실행계획들을 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 실행계획의 예상 비용이 적은가를 보고서 이를 실제로 수행하게 되는 것이다. 결국 옵티마이저의 생명은 예상 비용의 정확성에 달려 있다. 즉, <그림 2>의 비용산정(Cost Estimator) 모듈이 얼마나 정확하게 비용을 산정하느냐에 옵티마이저의 정확성이 결정된다.

많은 경우에, 옵티마이저의 비용산정 모듈은 아주 정확하게 비용을 예상하기 때문에, 현재 90% 이상의 실제 SQL들에 대해서 옵티마이저는 최선의 실행계획을 선택한다. 얼마나 위대한 기술인가! 그런데, 이 비용산정 모듈이 항상 정확한 예상 비용을 계산하는 것은 아니다. 그렇다면 왜 그럴까?

비용산정의 정확성에 영향을 미치는 요소

<그림 2>에서 알 수 있듯이, 특정 실행계획의 비용을 결정하는 요소는 크게 다음 3가지이다(물론, 이들 3가지 요소 이외에도 비용산정의 정확성에 영향을



<그림 2> 관계형 DBMS 옵티마이저의 내부 아키텍처

미치는 미묘한 여러 가지 문제들 - Bind 변수, 리터럴 변수, 특수한 조건식 '%xxx%', CPU와 저장장치 등의 접근 비용의 가정[참고문헌 8] 등 - 이 있지만, 이런 변수들은 여기서는 다루지 않겠다. 자세한 내용은 [참고문헌 5]를 참고하기 바란다).

- 통계정보 : 데이타 딕셔너리(Data Dictionary)
- 비용산정 모듈에서의 기본 가정들 : 선택도(Selectivity)와 카디널리티(Cardinality) 계산식
- 실제 질의 수행 환경에 대한 가정 : 비용 계산식의 런타임 수행환경(예 : 메모리)에 대한 가정

<그림 2>의 비용산정 모듈의 동작 순서를 따르면 1) 통계 정보를 바탕으로, 2) where 절의 조건식과 조인 조건에 대해 선택도와 카디널리티를 계산하고, 3) 이를 바탕으로 각 테이블에 대한 액세스 방법(예 : 전체 테이블 스캔 또는 인덱스)과 조인 방법(예 : 중첩루프나 정렬합병, 해시)의 비용을 계산한다.

그런데, 이 세 가지 과정 모두에서 잘못된 비용을 산정할 수 있는 가능성이 있다. 즉, 현재의 통계정보가 데이타베이스의 실제 데이타들의 분포와 다를 수 있고, 선택도와 카디널리티 계산 공식의 가정이 실제 데이타의 분포와 일치하지 않고, 비용 계산 공식에서 가정하고 있는 수행 환경이 실제 질의 수행 환경과는 많은 차이가 날 수 있는 것이다.

그러면 이제 이 세 가지 과정에서 발생할 수 있는 오류의 원인과 종류에 대해서 좀 더 자세히 알아보고, 이들 오류의 일반적인 해결책에 대해서도 간단히 알아보도록 하자. 지금부터는 오라클 DBMS를 중심으로 설명하겠지만, DB2나 MS SQL Server의 경우에도 유사하다고 이해하면 될 것이다.

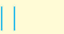
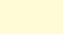
통계정보의 수준

일반적으로 관계형 DBMS의 경우, '시스템 카탈로그' 또는 '데이타 딕셔너리'라는 이름으로 테이블/인덱스/칼럼들에 대한 통계정보들을 유지한다<표 1>. 그러나 이 통계정보를 아주 정확하게 관리하는 것은 비용이 많이 들기 때문에, 테이블에 튜플들이 삽입/삭제/갱신될 때마다 자동적으로 유지, 관리되지 않고, 대신에 사용자가 주기적으로 통계정보를 수집하기 위한 명령을 수행한다.

구 분	필요한 통계정보
테이블	NUM_ROWS, BLOCKS, AVG_ROW_LENGTH 등
인덱스	BLEVEL, LEAF_BLOCKS, DISTINCT_KEYS, CLUSTERING_FACTOR 등
칼 럼	LOW_VALUE, HIGH_VALUE, DISTINCT_NUM, NULL_NUM, AVG_COL_LENGTH, HISTOGRAM 등

<표 1> 옵티마이저가 필요로 하는 기본적인 통계정보들

질의에서 참조된 특정 테이블, 칼럼, 인덱스들에 대한 통계정보를 딕셔



너리에서 참조해야 하는데, 이들에 대한 통계정보의 수준은 크게 다음과 같은 단계로 구분할 수 있다.

1. 디폴트(No Statistics) **단계**
테이블과 인덱스를 생성한 뒤, 사용자가 전혀 통계정보를 수집하지 않은 경우인데, 이 경우에 옵티마이저는 <표 1>에서 필요로 하는 정보들에 대해 기본적인 가정을 한다. 이 경우는 실제 데이타 분포와 완전히 틀릴 수 있다. 따라서 사용자는 다음 단계인 기본 통계정보는 유지할 필요가 있다.

2. 기본(Base Statistics) **단계**
실제 데이타를 분석해서 테이블, 인덱스, 칼럼들에 대해 기본적인 요약된 통계 정보를 유지하는 것이다. 이를 위해서 사용자는 Analyze table/index 등의 명령을 사용해야 한다.

3. 히스토그램(Histogram) **단계**
특정 칼럼에 대해서 데이타의 분포를 기본 단계보다 훨씬 더 정확하게 유지하는 것이다. 해당 칼럼에 대해 값들의 분포가 고르게 분포되지 않고(Non-Uniform), 치우친(Skewed) 경우에 해당 칼럼에 대해서는 히스토그램 정보를 생성해야 한다.

특정한 시점에 기본 또는 히스토그램 단계의 통계정보를 수집한 이후에, 실제 테이블에 데이타가 아주 많이 변경된 경우에는 새로이 통계정보를 재생성해 주어야 한다. 그렇지 않으면, 아무리 옵티마이저의 비용산정 모듈이 지능적이라도 결국은 틀린 비용 예측을 하게 되고, 궁극적으로 좋지 않은 실행계획을 선택하게 된다.

비용산정 모듈에서 사용하는 기본 가정들

<그림 2>의 비용산정 모듈에서 보듯이, 옵티마이저는 앞의 통계정보를 기반으로 선택도와 카디널리티를 계산하는 내부 공식을 갖고 있다. 이 내부 공식들은 다음 세 가지 가정에 기반한다.

- 균등 분포(Uniform Distribution)
- 조건식 상호독립(Predicate Independence)
- 조인 독립(Join Independence)

그런데 이 가정들은 많은 경우에 잘 성립하지만, 모든 경우에 성립하지는 않는 불완전한 가정이다. 따라서, 이 불완전한 가정으로 인해서 선택도와 카디널리티, 비용을 잘못 산정하게 될 수도 있는 것이다. 그러면 이들 각 가정의 의미와 문제점, 그리고 해결 방안에 대해 알아보자.

균등 분포 가정

균등 분포 가정은 어떤 테이블의 특정한 칼럼에 대해 튜플들이 갖는 '값이 골고루 분포되어 있다'는 가정이다. 예를 들어, d.loc에 대한 히스토그램 통계 정보가 없으면, 옵티마이저는 이 균등분포 가정에 기반해서 d.loc칼럼의 기본 단계의 서로 다른 값(Distinct Value) 개수(이를 NDV라 하자.)를 기준으로 해당 조건의 선택도를 1/NDV로 계산한다. 그러나, 실제 데이타가 d.loc에 대해 치우친(Skewed) 분포를 보이면, 옵티마이저의 선택도, 카디널리티, 비용이 순차적으로 틀리게 되는 것이다.

따라서, 이 가정이 일반적으로 맞지 않은 칼럼에 대해서는 사용자가 해당 칼럼에 대해 히스토그램 정보를 유지해야 한다.

조건식 상호독립 가정

조건식 상호독립 가정은 하나의 테이블에 대해 주어진 where절의 각 조건식이 서로 독립적이라는 것이다. 예를 들어, 질의 "select * from emp where deptno = 10 and salary < 40000"의 경우, deptno와 salary 각각에 대해 완전한 통계정보를 유지해서, deptno = 10의 선택도가 0.1이고 salary < 40000의 선택도는 0.4였다. 일반적으로 조건식 P1과 P2의 선택도를 Sel(P1), Sel(P2)라 할 때, '각 칼럼의 값들의 분포는 서로 독립적이다'는 가정에 기반해서, (P1 AND P2) 조건식의 결합 선택도(Combined Selectivity)는 Sel(P1) * Sel(P2)로 계산하게 된다. 이때 옵티마이저는 emp 테이블에서 where 절 조건의 전체 선택도를 0.1 x 0.4, 즉 0.04로 계산한다. 실제로 deptno와 salary는 별로 상관관계가 없기 때문에, 이 가정에 따라 계산된 선택도 0.04는 거의 정확하다고 볼 수 있다. 그러나 질의 "select * from emp where job_title = '부장' and salary < 40000"의 경우, job_title = '부장'과 salary < 40000에 대해 정확한 선택도는 0.2이고 salary < 40000의 선택도는 0.4를 유지하고 있지만, 부장이면서 연봉이 40000이하인 경우는 거의 없기 때문에 실제 선택도는 0에 가까울 것이다. 즉, 이 가정은 실제 데이타 분포상 서로 밀접한 상관관계가 있는 두 칼럼에 대한 선택도를 구할 때 아주 잘못된 선택도를 산정하게 되고, 결국 잘못된 카디널리티와 예상 비용을 산정하게 된다.

이에 대한 해결책은 이러한 조건식 상호독립이 성립하지 않는 칼럼들의 조합에 대해 히스토그램을 별도로 유지하는 것이다. 그렇지만, 이는 유지해야 할 히스토그램의 정보가 기하급수적으로 늘어나게 되는 문제가 있기 때문에 현실적인 대안이 되기는 힘들다. 오라클은 Oracle9*i* Database부터 동적 샘플링(Dynamic Sampling)을 제공해서 좀 더 정확한 선택도 계산을 가능하게 하고 있다. 자세한 내용은 [참고문헌 6]을 보기 바란다.

또 다른 해결책으로 두 조건식 칼럼의 함수적 종속성(Functional Dependency)이나 다중값 종속성(Multivalued Dependency)을 이용해서 의미적 질의 최적화를 수행할 수도 있지만, 이 또한 현 단계에서는 현실적인 대안은 되지 못하고 이론적인 해결책일 뿐이다.

조인 독립 가정

조인 독립은 조인되는 두 테이블 T1과 T2에 대해, T1의 한 튜플이 T2의 모든 튜플들과 똑같은 확률로 조인된다고 가정(반대의 경우도 성립)하는 것이다. 예를 들어, 질의 "select * from emp, dept where emp.deptno = dept.deptno"의 결과 카디널리티는 Card(emp) x Card(dept) x 1/MAX(emp.deptno의 NDV, deptno.deptno의 NDV)가 된다. 이 가정은 조인되는 두 칼럼이 주키(primary key)/외래키(foreign key) 관계(N:1 의 관계)에 있다고 볼 때 가장 잘 들어맞는다. 만일에, emp.deptno와 dept.deptno가 거의 겹치는 값이 없다면, 조인 결과 튜플의 수도 거의 0에 가까울 것이다.

또 다른 극단적인 경우는, 아래 예처럼, 양쪽 칼럼 모두 특정 값이 많이 나타나는 경우로, 이들이 조인될 때는 조인 조건의 선택도는 1에 가까울 것이다. 즉, 조건식의 선택도는 1/3이 아니라, 실제 18/36, 즉 1/2이 된다. 이 조인 독립 가정에 위배되는 경우에 찾아내기 힘들고, 이는 실제 데이타 분포를 확인한 후 수동으로 찾아야 하고, 이를 바탕으로 사용자가 힌트(Hint) 기능을 사용함으로써 SQL 튜닝을 수행할 수밖에 없다.

...	deptno
	10
	10
	10
	10
	20
	30

emp

deptno	...
10	
10	
10	
10	
20	
30	

dept

메모리 실행환경에 대한 가정

현재의 관계형 DBMS들은 대부분 중첩루프(Nested Loop), 정렬합병(Sort Merge), 해시(Hash) 의 세 가지 조인 방법을 지원하고 있다. 그런데, 이들 각각의 조인 방법에 대한 비용 계산식에서는 실제 수행시에 사용 가능한 메모리 자원에 대해 특정한 가정을 하게 된다. 다음 두 가지가 대표적인 경우이다.

- 중첩루프 조인시 해당 데이타 및 인덱스 블록은 디스크로부터 읽어야 한다는 가정
- 정렬합병 또는 해시 조인에 필요한 메모리 크기에 대한 가정

중첩루프 조인의 가정

중첩루프의 경우 조인 비용(즉, 필요한 디스크 I/O 수)을 다음과 같이 산정한다.

Nested Loop Join Cost = cost of accessing outer table + (cardinality of outer table * cost of accessing inner table)

이 수식에 따르면, 중첩루프의 안쪽 테이블(inner table)의 데이타는 매번 디스크에서 읽어와야 한다는 가정이 숨어 있는 것이다. 그러나 실제 일정 정도 이상의 버퍼 캐시를 갖고 있고 인덱스를 통해서 안쪽 테이블을 액세스하는 경우, 항상 디스크 I/O가 발생하는 것은 아니고 인덱스 블록들이 버퍼 캐시에 존재할 확률이 높기 때문에 예상치보다 훨씬 적은 물리적인 디스크 I/O가 발생하게 된다. 따라서, 위 수식은 질의 수행 환경에 있어서 이용 가능한 메모리 자원에 대해 최악의 가정을 하고 있는 것이다. 결국은 중첩루프 조인 방법의 비용을

비용 산정에 영향을 미치는 요소	세부 요소	해결 방안
통계정보의 정확성	통계정보와 실제 데이타의 불일치	적절한 통계정보의 생성 및 주기적인 갱신
비용 산정 모듈의 가정과 실제 데이타 분포의 불일치	균등분포 가정	히스토그램(Histogram) 생성
	조건식 상호독립 가정	동적 샘플링(Dynamic Sampling)
	조인 독립 가정	자동적인 해결책 없음; 정확한 데이타 분석 후, 힌트 기능을 이용한 SQL 튜닝
조인 비용 계산식과 실제 수행환경의 차이	중첩루프 조인 버퍼캐시 가정	OPTIMIZER_INDEX_CACHING,OPTIMIZER_INDEX_COST_ADJ 조정
	Sort/Hash 메모리 가정	자동적인 해결책 없음, PGA_TARGET_SIZE 조정

<표 2> 옵티마이저의 기본적인 문제점 및 해결책

실제보다 너무 높이 책정하기 때문에 정렬합병이나 해시 조인 방법이 선택될 가능성이 높은 것이다.

이에 대한 해결책으로 init.ora 파일에서 OPTIMIZER_INDEX_CHACHING (디폴트로 0) 값과 OPTMIZER_INDEX_COST_ADJ(디폴트로 100)의 값을 조정해서 인덱스를 이용한 중첩루프 조인의 비용을 보정하는 방법이 있다. 물론, 정확한 값을 시스템 전체적으로 얼마로 조정할 것인가의 문제는 또 다른 어려운 문제이다.

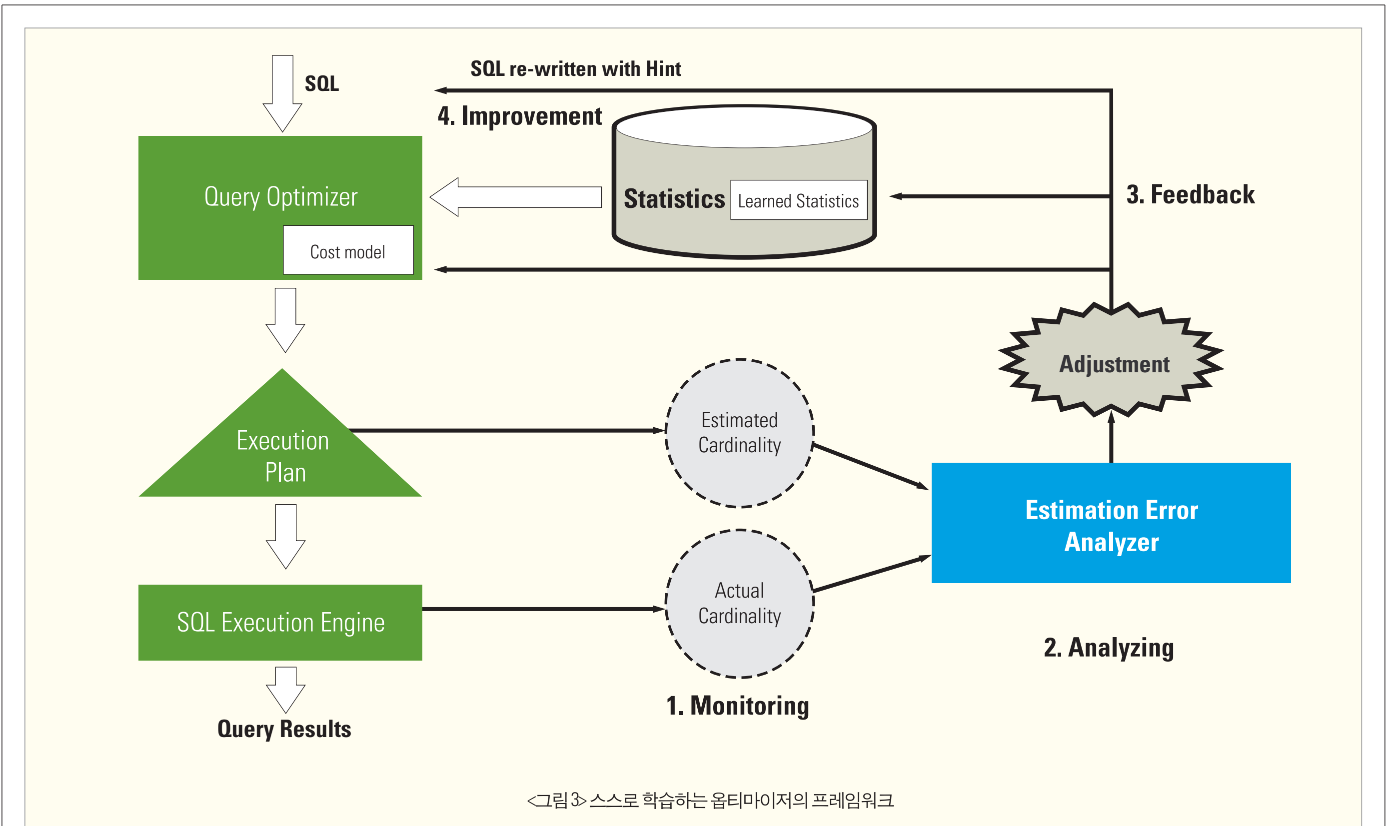
정렬합병 또는 해시 조인의 가정

정렬합병 또는 해시 조인의 경우, Oracle9*i* Database 이전에는 각 오라클 인스턴스마다 고정된 SORT_AREA_SIZE 또는 HASH_AREA_SIZE가 실제 할당되기 때문에 이 두 조인에 대한 비용은 상대적으로 정확했다고 볼 수 있다. 그러나, Oracle9*i* Database부터는 시스템 전체적으로 PGA_TARGET_SIZE 라는 목표치를 정해 놓고, 동적으로 각 사용자마다 필요에 의해 정렬이나 해싱을 위한 공간을 동적으로 할당하기 때문에, 이들 비용을 계산할 때 특정한 메모리 공간을 수식에 의해 가정해야만 한다. 필자도 이 경우에 옵티마이저가 메모리 공간에 어떤 가정을 하고, 동적으로 어떻게 할당되는지에 대해서는 자세히 알 방법이 없다. 따라서, 이 경우에 가정한 메모리 공간보다 실제 수행시 메모리가 훨씬 많거나 적을 경우, 옵티마이저가 예측한 공간보다 작을 경우 예상 비용이 틀리게 된다는 점은 분명하다.

이상의 내용을 정리하면, 관계형 DBMS의 옵티마이저는 기본적으로 통계정보, 선택도, 카디널리티, 비용 순으로 특정 실행계획의 수행 비용을 예측한다. 앞에서 이 관계형 DBMS의 옵티마이저가 비용을 계산하는 과정에서 발생할 수 있는 오류를 체계적으로 알아보았고, 각각의 오류에 대해 일반적으로 어떤 식으로 해결할 것인지에 대해 알아보았다. <표 2>는 이상에서 살펴본 관계형 DBMS 옵티마이저의 비용산정에 영향을 미치는 요소와 해결 방안을 정리한 것이다.

자동적인 해결책 - 스스로 학습하는 옵티마이저를 향해

지금까지는 관계형 DBMS 옵티마이저의 문제점과 이에 대해 개발자, DBA, 또는 SQL 튜닝 전문가가 취할 수 있는 해결 방안에 대해 알아보았다. 그렇지만, 관계형 DBMS는 궁극적으로는 사용자의 개입 없이도 항상 최선의 실행계획을 생성할 수 있어야 할 것이다. 현재 많은 관계형 DBMS 벤더들은 이 목표를 위해 새로운 기술 개발을 경주하고 있다. 여기서는 이러한 노력들 중에서 머지 않아 실용적으로 사용될 '스스로 학습을 통해 더 효율적인 실행계획을 생성하는 옵티마이저(Self-Learning Optimizer)'에 대해 알아보고자 한다. 이 내용은 필자가 대학



<그림 3> 스스로 학습하는 옵티마이저의 프레임워크

원에 재학중 생각한 아이디어이기도 하지만, 현재 학계나 산업계에서 활발히 연구가 진행중이다.

<그림 3>은 스스로 학습하는 옵티마이저의 일반적인 프레임워크이다. 그림 왼편은 지금까지의 일반적인 옵티마이저의 동작 과정이다. 앞에서 설명한 옵티마이저의 여러 가지 한계점으로 인해 특정 SQL에 대해 좋지 않은 실행계획을 생성하는 경우, 사용자가 원인을 분석해서 <표 2>와 같은 조처를 취하지 않는 경우 해당 질의에 대해서는 항상 똑같은 실행계획을 생성하고 수행하게 된다.

<그림 3>의 오른편은 스스로 학습하는 옵티마이저의 개념적인 동작 과정을 보여주는데, 크게 다음 4단계로 이루어진다.

- 모니터링(Monitoring)
- 분석(Analyzing)
- 피드백(Feedback)
- 개선(Improvement)

모니터링 단계

이 단계는 옵티마이저가 선택한 실행계획을 수행했을 때, 실행계획상의 각 로우 소스(row source)의 실행정보(예를 들어 실제 조건을 만족하는 튜플 수, 중첩루프 조인의 결과 튜플 수 및 논리적/물리적 버퍼 읽기 등)를 정확하게 산출한다. 이 단계는 SQL 실행 엔진 모듈에서 담당해야 하며, 이 모니터링을 위해서 약간의(일반적으로 5% 미만) 오버헤드가 발생한다.

분석 단계

분석 단계는 옵티마이저의 예상치(예를 들어, 로우 소스별 결과 카디널리티)와 모니터링 단계의 실제 값과 비교를 해서, 특정 로우 소스에서 예상치 대비 실제값 사이에 심각한 차이가 발생한 경우 최적화 과정에서 어떤 이유로 예상치가 틀렸는지(예를 들어 균등 분포의 오류, 조인 독립 가정의 위배)를 분석하는 단계이다. 그런데, 다양한 옵티마이저 오류의 원인들과 이 원인들이 어떤 식으로 실행계획의 선택에 나쁜 영향을 미쳤는지를 자동적으로 정확하게 분석하기란 쉽지 않다.

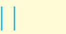
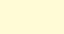
피드백 단계

피드백 단계는 앞에서 분석된 오류의 원인을 제거 또는 보정하기 위해 통계치, SQL 문, 옵티마이저의 내부 비용 모델 등을 조정하는 단계를 일컫는다. 이 단계 또한 앞에서 분석된 원인을 어떤 식으로 보정하는 것이 완벽한 보정인지를 결정하기란 쉽지 않다.

개선 단계

개선 단계는 피드백 단계에서 새로이 반영된 통계정보, 옵티마이저 내부 비용모델 등을 이용해서 최적화를 수행해서, 주어진 SQL 문에 대해 더 나은 실행계획을 생성하는 단계이다.

그러나, 스스로 학습하는 옵티마이저의 개념과 프레임워크는 아주 초보

적인 수준에 머물러 있으며 실용화되기 위해서는 많은 연구와 기술 개발 및 필드 테스트를 통한 검증이 있어야 할 것이다.

스스로 학습하는 옵티마이저의 구현 방안들

스스로 학습하는 옵티마이저 프레임워크의 구현 방안은 여러 가지가 있을 수 있다.

첫째, 이 프레임워크를 DBMS내에서 흡수해서 기존의 옵티마이저를 확장/구현하는 것이다. 하지만, 이 기술이 완전하게 검증된 것이 아니라는 문제가 있다. 오라클 DBMS의 경우도 이 학습하는 옵티마이저 기능을 현재 단계에서 제공하는 것은 힘들지 않겠지만, 현재 기술로는 완전한 자동화가 거의 불가능하기 때문에 이 기능을 제공하지 않는 것으로 판단된다.

둘째, 써드파티 도구로서 분석/피드백/개선 단계를 지원하는 방안이다. DBMS 엔진에서는 모니터링 단계의 정보만 생산하고, 도구에서 자동적으로 주어진 SQL에 대해 옵티마이저의 오류 원인을 분석하고, 피드백 정보를 어떻게 저장하고, 주어진 SQL을 어떻게 개선할 것인지에 대한 기능을 제공하는 것이다.

마지막으로, SQL 튜닝 전문가가 수동으로 분석/피드백/개선을 수행하는 것이다.

현재 (주)엑셈과 성균관대 VLDB 연구실에서는 두 번째 방안을 지원하는 도구를 개발중이다. 이를 위해 Oracle9*i* Database부터 제공하는 V$SQL_PLAN_STATISTICS_ALL 등의 뷰를 이용하고 있는데, 이 뷰는 스스로 학습하는 옵티마이저를 위한 모니터링 단계에서 필요한 다양한 정보를 제공하고 있다. 개발중인 도구의 일차 목표는 1) 이 뷰에서 제공하는 실행계획상의 예상 카디널리티와 실제 카디널리티의 값을 비교해서 심각한 차이가 발생하는 경우, 2) 앞서의 다양한 옵티마이저의 오류의 근본 원인과 실행계획 생성에 미치는 영향을 분석하고, 이를 보정하기 위해 3) 힌트 기능을 이용한 SQL 재작성, 통계정보 생성 권고, 옵티마이저 관련 파라미터 값 조정 권고 등의 기능을 제공할 것이다.

좀 더 나아가서는 V$SQL_PLAN_STATISTICS_ALL 뷰에서 제공하는 카디널리티 이외에 I/O 비용, 버퍼캐시 적중률 등의 정보를 기반으로 학습하는 기능도 추가할 수 있다. 써드파티 도구 방식이 기능적으로 완전하기 위해서는 향후 오라클 DBMS에서 두 가지 인터페이스를 제공해야 할 것이다.

우선 <그림 3>의 분석 단계에서 학습한 새로운 통계정보를 저장하고, 옵티마이저가 새로 최적화하는 과정에서 이를 활용할 수 있도록 한다. 예를 들면, 균등분포를 가정했는데, 특정 칼럼이 특정한 값에 대해 치우친(skewed) 경우 이 값에 대한 통계치를 저장하고, 나중에 카디널리티를 계산할 때 이 새로운 값을 기반으로 카디널리티 예상치를 보정할 수 있어야 한다.

둘째로, 옵티마이저 내부의 비용 모델을 변경할 수 있는 인터페이스를 제공해야 한다. 예를 들어 두 테이블에서 조인되는 칼럼들 사이에 조인 독립성 조건이 만족되지 않는 경우, 조인 카디널리티 계산 공식에 가중치를 파라미터로 이용해서 변경할 수 있어야 한다.

학습하는 옵티마이저에 대한 관심을 기대하며

이 글에서는 관계형 DBMS의 옵티마이저의 근본적인 한계점에 대해 살펴보았다. 이 문제점에 대해 단시일 내에 관계형 DBMS 벤더들이 완벽한 해결책을 내놓기는 힘들 것이다. 따라서, 사용자 입장에서 1) 이 문제의 근본적인 이유를 제대로 이해하고 2) 이를 해결하기 위해 어떤 기능들이 개별 DBMS에서 제공되는지를 파악하고 3) 현재 관계형 DBMS의 옵티마이저의 한계를 해당 기능들을 이용해서 어떻게 극복할 것인지를 고민하는 수밖에 없다.

또한, 옵티마이저의 근본적인 한계를 극복하기 위한 다양한 노력들 중에서, 필자가 판단하기에 앞으로 주요한 방향이 될 학습하는 옵티마이저의 프레임워크와 구현 방안에 대해 간단히 언급하였다. 사실 이 분야는 아직까지 미개척 분야이고, 각 단계별로 데이타베이스 튜닝 전문가들이 나름대로 기여할 여지가 엄청나게 많은 부분이다.

이 글이 데이타베이스 관련 종사자들에게 관계형 옵티마이저의 근본적인 한계를 이해하는 데 큰 맥락에서 도움이 되기를 바라고, 조만간 오라클 DBMS 옵티마이저에 관한 좀더 완벽하고 포괄적인 내용을 포함하는 자료[참고문헌 5]를 제공할 수 있기를 희망한다. 또한, 현재 필드에 종사하는 여러 전문가들과의 교류를 통해서 스스로 학습하는 옵티마이저의 요소 기술들에 대해 깊이 있는 논의를 할 수 있는 기본 토대가 되었으면 하는 바람이다.

참고 문헌

1. 이상원, 오라클 옵티마이저의 기본원리, 오라클매거진코리아 2002년 겨울호
2. Pat. Selinger 외 다수, Access Path Selection in a Relational Database System, ACM SIGMOD 1979
3. Hector Garcia-Molina, Jeffrey D. Ullman, Jennifer Widom, Database Systems: The Complete Book(Chapter 16), Prentice Hall, 2001
4. Wolfgang Breitling, "Fallacies of Cost Based Optimizer," 2003 Hotsos Symposium on Oracle System Performance
5. 이상원, 오라클 옵티마이저 내부 동작원리의 모든 것, In Preparation
6. Oracle Corp. Query Optimization in Oracle9*i*, 2002 Technical White Paper
7. Benoit Dageville, Mohamed Zait, "SQL Memory Management in Oracle9*i*," VLDB2002
8. Frederick R. Reiss, Tapas Kanungo, "A Characterization of the Sensitivity of Query Optimization to Storage Access Cost Parameters," SIGMOD 2003
9. Oracle Corp., Oracle9*i* Database Performance Tuning Guide and Reference Release 2 (9.2), http://otn.oracle.com/docs/products/oracle9i/doc_library/release2/server.920/a96533/toc.htm