1. 옵티마이저 소개

가. 옵티마이저란?

옵티마이저(Optimizer)는 SQL을 가장 빠르고 효율적으로 수행할 최적(최저비용)의 처리경로를 생성해 주는 DBMS 내부의 핵심엔진이다. 사용자가 구조화된 질의언어(SQL)로 결과집합을 요구하면, 이를 생성하는데 필요한 처리경로는 DBMS에 내장된 옵티마이저가 자동으로 생성해준다. 옵티마이저가 생성한 SQL 처리경로를 실행계획(Execution Plan)이라고 부른다. 옵티마이저의 SQL 최적화 과정을 요약하면 다음과 같다.

- 사용자가 던진 쿼리수행을 위해, 후보군이 될만한 실행계획을 찾는다.
- 데이터 딕셔너리(Data Dictionary)에 미리 수집해 놓은 오브젝트 통계 및 시스템 통계 정보를 이용해 각 실행계획의 예상비용을 산정한다.
- 각 실행계획을 비교해서 최저비용을 갖는 하나를 선택한다.

나. 옵티마이저 종류

옵티마이저는 다음 두 가지로 나뉘며, 앞서 설명한 SQL 최적화 과정은 비용기반 옵티마이저에 관한 것이다.

1) 규칙기반 옵티마이저

규칙기반 옵티마이저(Rule-Based Optimizer, 이하 RBO)는 다른 말로 '휴리스틱(Heuristic) 옵티마이저'라고 불리며, 미리 정해 놓은 규칙에 따라 액세스 경로를 평가하고 실행계획을 선택한다. 여기서 규칙이란 액세스 경로별 우선순위로서, 인덱스 구조, 연산자, 조건절 형태가 순위를 결정짓는 주요인이다.

2) 비용기반 옵티마이저

비용기반 옵티마이저(Cost-Based Optimizer, 이하 CBO)는 말 그대로 비용을 기반으로 최적화를 수행한다. 여기서 '비용(Cost)'이란, 쿼리를 수행하는데 소요되는 일량 또는 시간을 뜻한다. CBO가 실행계획을 수립할 때 판단 기준이 되는 비용은 어디까지나 예상치다. 미리 구해 놓은 테이블과 인덱스에 대한 여러 통계정보를 기초로 각 오퍼레이션 단계별 예상 비용을 산정하고, 이를 합산한 총비용이 가장 낮은 실행계획을 선택한다. 비용을 산정할 때 사용되는 오브젝트 통계 항목으로는 레코드 개수, 블록 개수, 평균 행 길이, 칼럼 값의 수, 칼럼 값 분포, 인덱스 높이(Height), 클러스터링 팩터 같은 것들이 있다. 오브젝트 통계뿐만 아니라 최근에는 하드웨어적 특성을 반영한 시스템 통계정보(CPU 속도, 디스크 I/O 속도 등)까지 이용한다. 역사가 오래된 Oracle은 RBO에서 출발하였으나 다른 상용 RDBMS는 탄생 초기부터 CBO를 채택하였다. Oracle도 10g 버전부터 RBO에 대한 지원을 중단하였으므로 본서는 CBO를 중심으

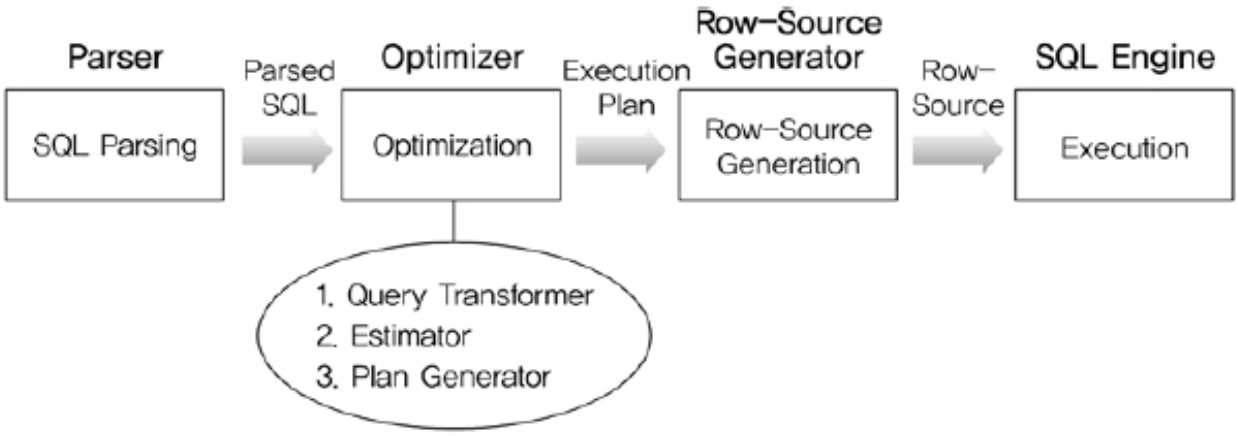
로 설명한다.

Notes. 스스로 학습하는 옵티마이저(Self-Learning Optimizer)

전통적으로 옵티마이저는, 오브젝트 통계와 시스템 통계로부터 산정한 '예상' 비용만으로 실행계획을 수립해 왔다. 하지만 앞으로는 예상치와 런타임 수행 결과를 비교하고, 예상치가 빗나갔을 때 실행계획을 조정하는 옵티마이저로 발전할 것이다. 최근에 발표된 각 DBMS 버전은 이미 이런 기능들을 포함하고 있다.

다. SQL 최적화 과정

Oracle 기준으로, SQL 최적화 및 수행 과정을 좀 더 자세히 표현하면 [그림 III-3-1]과 같다.



[그림 Ⅲ-3-1] SQL 최적화 및 수행 과정

[표 III-3-1]은 [그림 III-3-1]에 표현된 각 서브엔진의 역할을 요약한 것이다.

[표 Ⅲ-3-1] 서브엔진별 역할

엔진		역할
Parser		SQL 문장을 이루는 개별 구성요소를 분석하고 파싱해서 파싱 트리(내부적인 구조체)를 만든다. 이 과정에서 사용자 SQL에 문법적 오류가 없는지(→ Syntax 체크), 의미상 오류가 없는지(→ Semantic 체크) 확인한다.
Optimizer	Query Transformer	파싱된 SQL을 좀 더 일반적이고 표준적인 형태로 변환한다.
	Estimator	오브젝트 및 시스템 통계정보를 이용해 쿼리 수행 각 단계의 선택도, 카디널리티, 비용을 계산하고, 궁극적으로는 실행계획 전체에 대한 총 비용을 계산해 낸다.
	Plan Generator	하나의 쿼리를 수행하는 데 있어, 후보군이 될만한 실행계획들을 생성해 낸다.
Row-Source Generator		옵티마이저가 생성한 실행계획을 SQL 엔진이 실제 실행할 수 있는 코드(또는 프로시저) 형태로 포맷팅한다.
SQL Engine		SQL을 실행한다.

Oracle 뿐만 아니라 다른 DBMS도 비슷한 처리과정을 통해 실행계획을 생성한다. 참고로 M.Jarke와 J.Koch가 펴낸 "Query Optimization in Database Systems"를 보면, 쿼리 최적화 과정을 다음과 같이 설명하고 있는데, [그림 III-3-1]과 [표 III-3-1]에서 설명한 Parser와

Optimizer 역할에 해당하는 내용임을 알 수 있다.

- 쿼리를 내부 표현방식으로 변환
- 표준적인(canonical) 형태로 변환
- 후보군이 될만한 (낮은 레벨의) 프로시저를 선택
- 실행계획을 생성하고, 가장 비용이 적은 것을 선택

라. 최적화 목표

1) 전체 처리속도 최적화

쿼리 최종 결과집합을 끝까지 읽는 것을 전제로, 시스템 리소스(I/O, CPU, 메모리 등)를 가장 적게 사용하는 실행계획을 선택한다. Oracle, SQL Server 등을 포함해 대부분 DBMS의 기본 옵티마이저 모드는 전체 처리속도 최적화에 맞춰져 있다. Oracle에서 옵티마이저 모드를 바꾸는 방법은 다음과 같다.

alter system set optimizer_mode = all_rows; -- 시스템 레벨 변경 alter session set optimizer_mode = all_rows; -- 세션 레벨 변경 select /*+ all_rows */ * from t where … ; -- 쿼리 레벨 변경

2) 최초 응답속도 최적화

전체 결과집합 중 일부만 읽다가 멈추는 것을 전제로, 가장 빠른 응답 속도를 낼 수 있는 실행계획을 선택한다. 만약 이 모드에서 생성한 실행계획으로 데이터를 끝까지 읽는다면 전체 처리속도 최적화 실행계획보다 더 많은 리소스를 사용하고 전체 수행 속도도 느려질 수 있다. Oracle 옵티마이저에게 최초 응답속도 최적화를 요구하려면, 옵티마이저 모드를 first_rows로 바꿔주면 된다. SQL 서버에서는 테이블 힌트로 fastfirstrow를 지정하면 된다. Oracle에서 옵티마이저 모드를 first_rows_n으로 지정하면, 예를 들어 시스템 또는 세션 레벨에서 first_rows_10으로 지정하면, 사용자가 전체 결과집합 중 처음 10개 로우만 읽고 멈추는 것을 전제로 가장 빠른 응답 속도를 낼 수 있는 실행계획을 선택한다. 쿼리 레벨에서 힌트를 사용하려면 아래와 같이 하면 된다.

select /*+ first_rows(10) */ * from t where ;

SQL 서버에서는 쿼리 힌트로 fast 10을 지정하면 된다.

select * from t where OPTION(fast 10);

2. 옵티마이저 행동에 영향을 미치는 요소

가. SQL과 연산자 형태

결과가 같더라도 SQL을 어떤 형태로 작성했는지 또는 어떤 연산자를 사용했는지에 따라 옵티마이저가 다른 선택을 할 수 있고, 이는 쿼리 성능에 영향을 미친다.

나. 옵티마이징 팩터

쿼리를 똑같이 작성하더라도 인덱스, IOT, 클러스터링, 파티셔닝, MV 등을 어떻게 구성했는지에 따라 실행계획과 성능이 크게 달라진다.

다. DBMS 제약 설정

개체 무결성, 참조 무결성, 도메인 무결성 등을 위해 DBMS가 제공하는 PK, FK, Check, Not Null 같은 제약 설정 기능을 이용할 수 있고, 이들 제약 설정은 옵티마이저가 쿼리 성능을 최적화하는 데에 매우 중요한 정보를 제공한다. 예를 들어, 인덱스 칼럼에 Not Null 제약이 설정돼 있으면 옵티마이저는 전체 개수를 구하는 Count 쿼리에 이 인덱스를 활용할 수 있다.

라. 옵티마이저 힌트

옵티마이저의 판단보다 사용자가 지정한 옵티마이저 힌트가 우선한다. 옵티마이저 힌트에 대해서는 뒤에서 좀 더 자세히 다룬다.

마. 통계정보

통계정보가 옵티마이저에게 미치는 영향력은 절대적이다. 뒤에서 통계정보를 이용한 비용계산 원리를 설명할 때 느끼겠지만 CBO의 모든 판단 기준은 통계정보에서 나온다.

바. 옵티마이저 관련 파라미터

SQL, 데이터, 통계정보, 하드웨어 등 모든 환경이 동일하더라도 DBMS 버전을 업그레이드하면 옵티마이저가 다르게 작동할 수 있다. 이는 옵티마이저 관련 파라미터가 추가 또는 변경되면서 나타나는 현상이다.

사. DBMS 버전과 종류

옵티마이저 관련 파라미터가 같더라도 버전에 따라 실행계획이 다를 수 있다. 또한, 같은 SQL이더라도 DBMS 종류에 따라 내부적으로 처리하는 방식이 다를 수 있다.

3. 옵티마이저의 한계

옵티마이저가 사람이 만든 소프트웨어 엔진에 불과하며 결코 완벽할 수 없음을 이해하는 것은 매우 중요하다. 현재의 기술수준으로 해결하기 어려운 문제가 있는가 하면, 기술적으론 가능한데 현실적인 제약(통계정보 수집량과 최적화를 위해 허락된 시간) 때문에 아직 적용하지 못하는 것들도 있다. 옵티마이저가 완벽하지 못하게 만드는 요인이 어디에 있는지 구체적으

로 살펴보자.

가. 옵티마이징 팩터의 부족

옵티마이저는 주어진 환경에서 가장 최적의 실행계획을 수립하기 위해 정해진 기능을 수행할 뿐이다. 옵티마이저가 아무리 정교하고 기술적으로 발전하더라도 사용자가 적절한 옵티마이징 팩터(효과적으로 구성된 인덱스, IOT, 클러스터링, 파티셔닝 등)를 제공하지 않는다면 결코 좋은 실행계획을 수립할 수 있다.

나. 통계정보의 부정확성

최적화에 필요한 모든 정보를 수집해서 보관할 수 있다면 옵티마이저도 그만큼 고성능 실행계획을 수립하겠지만, 100% 정확한 통계정보를 유지하기는 현실적으로 불가능하다. 특히, 칼럼 분포가 고르지 않을 때 칼럼 히스토그램이 반드시 필요한데, 이를 수집하고 유지하는 비용이 만만치 않다. 칼럼을 결합했을 때의 모든 결합 분포를 미리 구해두기 어려운 것도 큰 제약 중 하나다. 이는 상관관계에 있는 두 칼럼이 조건절에 사용될 때 옵티마이저가 잘못된 실행계획을 수립하게 만드는 주요인이다. 아래 쿼리를 예를 들어 보자.

select * from 사원 where 직급 = '부장' and 연봉 >= 5000;

직급이 {부장, 과장, 대리, 사원}의 집합이고 각각 25%의 비중을 갖는다. 그리고 전체 사원이 1,000명이고 히스토그램상 '연봉 >= 5000' 조건에 부합하는 사원 비중이 10%이면, 옵티마이저는 위 쿼리 조건에 해당하는 사원 수를 25(=1,000×0.25×0.1)명으로 추정한다. 하지만 잘 알다시피 직급과 연봉 간에는 상관관계가 매우 높아서, 만약 모든 부장의 연봉이 5,000만원 이상이라면 실제 위 쿼리 결과는 250(=1,000×0.25×1)건이다. 이런 조건절에 대비해 모든 칼럼 간 상관관계와 결합 분포를 미리 저장해 두면 좋겠지만 이것은 거의 불가능에 가깝다. 테이블 칼럼이 많을수록 잠재적인 칼럼 조합의 수는 기하급수적으로 증가하기 때문이다.

다. 바인드 변수 사용 시 균등분포 가정

아무리 정확한 칼럼 히스토그램을 보유하더라도 바인드 변수를 사용한 SQL에는 무용지물이다. 조건절에 바인드 변수를 사용하면 옵티마이저가 균등분포를 가정하고 비용을 계산하기 때문이다.

라. 비현실적인 가정

옵티마이저는 쿼리 수행 비용을 평가할 때 여러 가정을 사용하는데, 그 중 일부는 상당히 비현실적이어서 종종 이해할 수 없는 실행계획을 수립하곤 한다. 예전 Oracle 버전에선 Single Block I/O와 Multiblock I/O의 비용을 같게 평가하고 데이터 블록의 캐싱 효과도 고려하지 않았는데, 그런 것들이 비현실적인 가정의 좋은 예다. DBMS 버전이 올라가면서 이런 비현실적

인 가정들이 계속 보완되고 있지만 완벽하지 않고, 모두 해결되리라고 기대하는 것도 무리다.

마. 규칙에 의존하는 CBO

아무리 비용기반 옵티마이저라 하더라도 부분적으로는 규칙에 의존한다. 예를 들어, 최적화 목표를 최초 응답속도에 맞추면(Oracle을 예로 들면, optimizer_mode = first_rows), order by 소트를 대체할 인덱스가 있을 때 무조건 그 인덱스를 사용한다. 다음 절에서 설명할 휴리스틱(Heuristic) 쿼리 변환도 좋은 예라고 할 수 있다.

바. 하드웨어 성능

옵티마이저는 기본적으로 옵티마이저 개발팀이 사용한 하드웨어 사양에 맞춰져 있다. 따라서 실제 운영 시스템의 하드웨어 사양이 그것과 다를 때 옵티마이저가 잘못된 실행계획을 수립할 가능성이 높아진다. 또한 애플리케이션 특성(I/O 패턴, 부하 정도 등)에 의해서도 하드웨어 성능은 달라진다.

4. 통계정보를 이용한 비용계산 원리

실행계획을 수립할 때 CBO는 SQL 문장에서 액세스할 데이터 특성을 고려하기 위해 통계정보를 이용한다. 최적의 실행계획을 위해 통계정보가 항상 데이터 상태를 정확하게 반영하고 있어야 하는 이유다. DBMS 버전이 올라갈수록 자동 통계관리 방식으로 바뀌고 있지만, 가끔 DB 관리자가 수동으로 수집관리해 주어야 할 때도 있다. 옵티마이저가 참조하는 통계정보 종류로 아래 네 가지가 있다.

[표 Ⅲ-3-2] 옵티마이저 통계 유형

통계 유형	세부 통계 항목
테이블 통계	전체 레코드 수, 총 블록 수, 빈 블록 수, 한 행당 평균 크기 등
인덱스 통계	인덱스 높이, 리프 블록 수, 클러스터링 팩터, 인덱스 레코드 수 등
칼럼 통계	값의 수, 최저 값, 최고 값, 밀도, null값 개수, 칼럼 히스토그램 등
시스템 통계	CPU 속도, 평균적인 I/O 속도, 초당 I/O 처리량 등

지금부터, 데이터 딕셔너리에 미리 수집해 둔 통계정보가 옵티마이저에 의해 구체적으로 어떻게 활용되는지 살펴보자.

가. 선택도

선택도(Selectivity)는 전체 대상 레코드 중에서 특정 조건에 의해 선택될 것으로 예상되는 레코드 비율을 말한다. 선택도를 가지고 카디널리티를 구하고, 다시 비용을 구해 인덱스 사용 여부, 조인 순서와 방법 등을 결정하므로 선택도는 최적의 실행계획을 수립하는 데 있어 가장 중요한 요인이라고 하겠다.

- 선택도 → 카디널리티 → 비용 → 액세스 방식, 조인 순서, 조인 방법 등 결정 히스토그램이 있으면 그것으로 선택도를 산정하며, 단일 칼럼에 대해서는 비교적 정확한 값을 구한다. 히스토그램이 없거나, 있더라도 조건절에 바인드 변수를 사용하면 옵티마이저는 데이터 분포가 균일하다고 가정한 상태에서 선택도를 구한다. 히스토그램 없이 등치(=) 조건에 대한 선택도를 구하는 공식은 다음과 같다.
-

$$\text{선택도} = \frac{1}{\text{Distinct Value 개수}} = \frac{1}{\text{num distinct}}$$

나. 카디널리티

카디널리티(Cardinality)는 특정 액세스 단계를 거치고 난 후 출력될 것으로 예상되는 결과 건수를 말하며, 아래와 같이 총 로우 수에 선택도를 곱해서 구한다.

- 카디널리티 = 총 로우 수 × 선택도 칼럼 히스토그램이 없을 때 '=' 조건에 대한 선택도가 1/num_distinct이므로 카디널리티는 아래와 같이 구해진다.
- 카디널리티 = 총 로우 수 × 선택도 = num_rows / num_distinct

  select * from 사원 where 부서 = :부서

  예를 들어, 위 쿼리에서 부서 칼럼의 Distinct Value 개수가 10이면 선택도는 0.1(=1/10)이고, 총 사원 수가 1,000명일 때 카디널리티는 100이 된다. 옵티마이저는 위 조건절에 의한 결과집합이 100건일 것으로 예상한다는 뜻이다. 조건절이 두 개 이상일 때는 각 칼럼의 선택도와 전체 로우 수를 곱해 주기만 하면 된다.

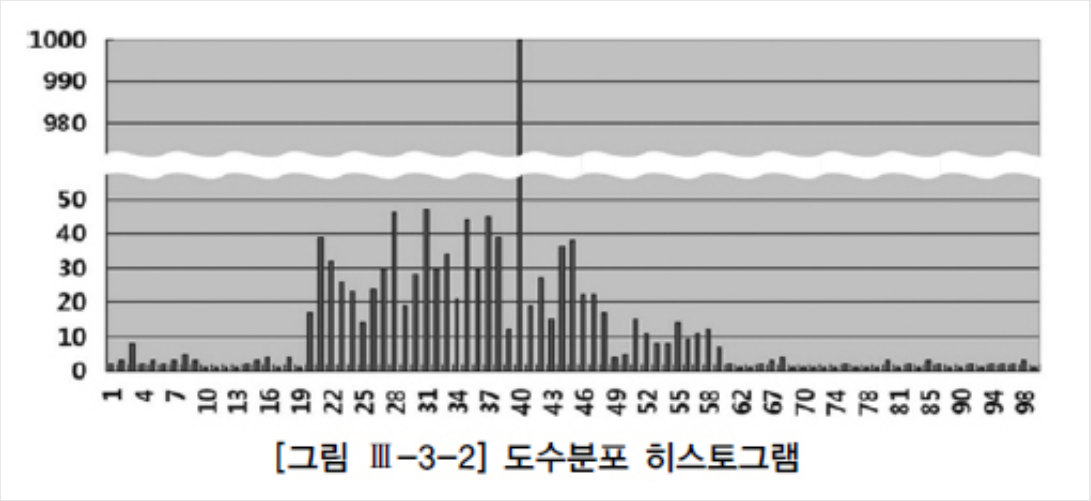
  select * from 사원 where 부서 = :부서 and 직급 = :직급;

  직급의 도메인이 {부장, 과장, 대리, 사원}이면 Distinct Value 개수가 4이므로 선택도는 0.25(=1/4)다. 따라서 위 쿼리의 카디널리티는 25(=1000 × 0.1 × 0.25)로 계산된다.

다. 히스토그램

미리 저장된 히스토그램 정보가 있으면, 옵티마이저는 그것을 사용해 더 정확하게 카디널리티를 구할 수 있다. 특히, 분포가 균일하지 않은 칼럼으로 조회할 때 효과를 발휘한다. 히스토그램에는 아래 두 가지 유형이 있다.

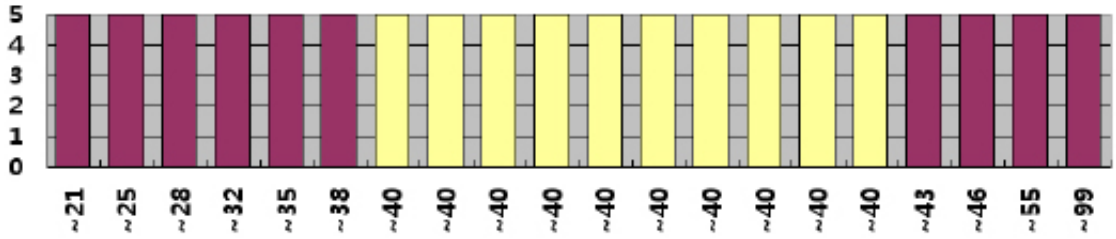
- 도수분포 히스토그램
  [그림 III-3-2]처럼 값별로 빈도수(frequency number)를 저장하는 히스토그램을 말한다.



[그림 Ⅲ-3-2] 도수분포 히스토그램

칼럼이 가진 값의 수가 적을 때 사용되며, 칼럼 값의 수가 적기 때문에 각각 하나의 버킷을 할당(값의 수 = 버킷 개수)하는 것이 가능하다.

- 높이균형 히스토그램
  칼럼이 가진 값의 수가 아주 많아 각각 하나의 버킷을 할당하기 어려울 때 사용된다. 히스토그램 버킷을 값의 수보다 적게 할당하기 때문에 하나의 버킷이 여러 개 값을 담당한다. 예를 들어, 값의 수가 1,000개인데 히스토그램을 위해 할당된 버킷 개수가 100개이면, 하나의 버킷이 평균적으로 10개의 값을 대표한다. 높이균형 히스토그램에서는 말 그대로 각 버킷의 높이가 같다. 각 버킷은 {1/(버킷 개수) × 100}%의 데이터 분포를 갖는다. 따라서 각 버킷(→ 값이 아니라 버킷)이 갖는 빈도수는 {(총 레코드 개수) / (버킷 개수)}로써 구할 수 있다. 빈도 수가 많은 값(popular value)에 대해서는 두 개 이상의 버킷이 할당된다. [그림 III-3-3]에서 x 축은 연령대를 의미하는데, age = 40인 레코드 비중이 50%이어서 총 20개 중 10개 버킷을 차지한 것을 볼 수 있다.



[그림 Ⅲ-3-3] 높이균형 히스토그램

Notes. 바인드 변수 사용 시 카디널리티 계산

바인드 변수를 사용하면, 최초 수행할 때 최적화를 거친 실행계획을 캐시에 적재하고 실행시점에는 그것을 그대로 가져와 값만 다르게 바인딩하면서 반복 재사용하게 된다. 여기서, 변수를 바인딩하는 시점이 (최적화 시점보다 나중인) 실행시점이라는 사실이 중요하다. 즉, SQL을 최적화하는 시점에 조건절 칼럼의 데이터 분포를 활용하지 못한다.
바인드 변수를 사용할 때 옵티마이저가 평균 분포를 가정한 실행계획을 생성하는 것도 이 때문이다. 칼럼 분포가 균일할 때는 상관없겠지만, 그렇지 않을 때는 실행 시점에 바인딩되는 값에 따라 쿼리 성능이 다르게 나타날 수 있어 문제다. 따라서 DW, OLAP, 배치 프로그램(Loop 내에서 수행되는 쿼리 제외)에서 수행되는 쿼리는 바인드 변수보다 상수를 사용하는 것이 좋은데, 날짜 칼럼처럼 부등호, between 같은 범위 조건으로 자주 검색되는 칼럼일 때 특히 그렇다. OLTP성 쿼리이더라도 값의 종류가 적고 분포가 균일하지 않을 때는 상수 조건을 쓰는 것이 유용할 수 있다.

라. 비용

CBO는 비용(Cost)을 기반으로 최적화를 수행하고 실행계획을 생성한다고 설명했다. 여기서 '비용(Cost)'이란, 쿼리를 수행하는데 소요되는 일량 또는 시간을 뜻하며, 어디까지나 예상치다. 옵티마이저 비용 모델에는 I/O 비용 모델과 CPU 비용 모델 두 가지가 있다. I/O 비용 모델은 예상되는 I/O 요청(Call) 횟수만을 쿼리 수행 비용으로 간주해 실행계획을 평가하는 반면 CPU 비용 모델은 여기에 시간 개념을 더해 비용을 산정한다. 지면 관계상 본서는 I/O 비용 모델만 다루기로 하겠다.

- 인덱스를 경유한 테이블 액세스 비용
  I/O 비용 모델에서의 비용은 디스크 I/O Call 횟수(논리적/물리적으로 읽은 블록 개수가 아닌 I/O Call 횟수)를 의미한다. 그리고 인덱스를 경유한 테이블 액세스 시에는 Single Block I/O 방식이 사용된다. 이는 디스크에서 한 블록을 읽을 때마다 한 번의 I/O Call을 일으키는 방식이므로 읽게 될 물리적 블록 개수가 I/O Call 횟수와 일치한다. 따라서 인덱스를 이용한 테이블 액세스 비용은 아래와 같은 공식으로 구할 수 있다. 비용 = blevel -- 인덱스 수직적 탐색 비용 + (리프 블록 수 × 유효 인덱스 선택도) -- 인덱스 수평적 탐색 비용 + (클러스터링 팩터 × 유효 테이블 선택도) -- 테이블 Random 액세스 비용

[표 Ⅲ-3-3] 인덱스를 경유한 테이블 액세스 비용 항목

항목	설명
blevel	브랜치 레벨을 의미하며, 리프 블록에 도달하기 전에 읽게 될 브랜치 블록 개수임
클러스터링 팩터	특정 칼럼을 기준으로 같은 값을 갖는 데이터가 서로 모여있는 정도. 인덱스를 경유해 테이블 전체 로우를 액세스할 때 읽을 것으로 예상되는 논리적인 블록 개수로 계수화 함
유효 인덱스 선택도	전체 인덱스 레코드 중에서 조건절을 만족하는 레코드를 찾기 위해 스캔할 것으로 예상되는 비율(%). 리프 블록에는 인덱스 레코드가 정렬된 상태로 저장되므로 이 비율이 곧, 방문할 리프 블록 비율임
유효 테이블 선택도	전체 레코드 중에서 인덱스 스캔을 완료하고서 최종적으로 테이블을 방문할 것으로 예상되는 비율(%). 클러스터링 팩터는 인덱스를 경유해 전체 로우를 액세스할 때 읽힐 것으로 예상되는 테이블 블록 개수이므로 여기에 유효 테이블 선택도를 곱함으로써 조건절에 대해 읽힐 것으로 예상되는 테이블 블록 개수를 구할 수 있음

- Full Scan에 의한 테이블 액세스 비용

Full Scan에 대해서는, 테이블 전체를 순차적으로 읽어 들이는 과정에서 발생하는 I/O Call 횟수로 비용을 계산한다. Full Scan할 때는 한 번의 I/O Call로써 여러 블록을 읽어 들이는 Multiblock I/O 방식을 사용하므로 총 블록 수를 Multiblock I/O 단위로 나눈 만큼 I/O Call이 발생한다. 예를 들어, 100블록을 8개씩 나누어 읽는다면 13번의 I/O Call이 발생하고, I/O Call 횟수로써 Full Scan 비용을 추정한다. 따라서 Multiblock I/O 단위가 증가할수록 I/O Call 횟수가 줄고 예상비용도 줄게 된다.

5. 옵티마이저

통계정보가 정확하지 않거나 기타 다른 이유로 옵티마이저가 잘못된 판단을 할 수 있다. 그럴 때 프로그램이나 데이터 특성 정보를 정확히 알고 있는 개발자가 직접 인덱스를 지정하거나 조인 방식을 변경함으로써 더 좋은 실행계획으로 유도하는 메커니즘이 필요한데, 옵티마이저 힌트가 바로 그것이다. 힌트 종류와 구체적인 사용법은 DBMS마다 천차만별이다. 지면 관계상 모두 다룰 수 없으므로 Oracle과 SQL Server에 대해서만 설명하기로 한다.

가. Oracle 힌트

1) 힌트 기술 방법

Oracle에서 힌트를 기술하는 방법은 다음과 같다.

```
SELECT /*+ LEADING(e2 e1) USE_NL(e1) INDEX(e1 emp_emp_id_pk) USE_MERGE(j)
FULL(j) */ e1.first_name, e1.last_name, j.job_id, sum(e2.salary) total_sal FROM employees
e1, employees e2, job_history j WHERE e1.employee_id = e2.manager_id AND
e1.employee_id = j.employee_id AND e1.hire_date = j.start_date GROUP BY e1.first_name,
e1.last_name, j.job_id ORDER BY total_sal;
```

2) 힌트가 무시되는 경우

다음과 같은 경우에 Oracle 옵티마이저는 힌트를 무시하고 최적화를 진행한다.

- 문법적으로 안 맞게 힌트를 기술
- 의미적으로 안 맞게 힌트를 기술
  예를 들어, 서브쿼리에 unnest와 push_subq를 같이 기술한 경우(unnest되지 않은 서브쿼리만이 push_subq 힌트의 적용 대상임)
- 잘못된 참조 사용
  없는 테이블이나 별칭(Alias)을 사용하거나, 없는 인덱스명을 지정한 경우 등
- 논리적으로 불가능한 액세스 경로
  조인절에 등치(=) 조건이 하나도 없는데 Hash Join으로 유도하거나, 아래 처럼 null 허용칼럼에 대한 인덱스를 이용해 전체 건수를 세려고 시도하는 등

select /*+ index(e emp_ename_idx) */ count(*) from emp e

- 버그
  위 경우에 해당하지 않는 한 옵티마이저는 힌트를 가장 우선적으로 따른다. 즉, 옵티마이저는 힌트를 선택 가능한 옵션 정도로 여기는 게 아니라 사용자로부터 주어진 명령어(directives)로 인식한다. 여기서 주의할 점이 있다. Oracle은 사용자가 힌트를 잘못 기술하거나 잘못된 참조를 사용하더라도 에러가 발생하지 않는다는 사실이다. 힌트와 관련한 Oracle의 이런 정책은 프로그램 안정성 측면에 도움이 되는가 하면, 성능 측면에서 불안할 때도 있다. 예를 들어, 힌트에 사용된 인덱스를 어느 날 DBA가 삭제하거나 이름을 바꾸었다고 하자. 그럴 때 SQL Server에선 에러가 발생하므로 해당 프로그램을 수정하고 다시 컴파일해야 한다. 프로그램을 수정하다 보면 인덱스 변경이 발생했다는 사실을 발견하게 되고, 성능에 문제가 생기지 않도록 적절한 조치를 취할 것이다. 반면, Oracle에선 프로그램을 수정할 필요가 없어 좋지만 내부적으로 Full Table Scan하거나 다른 인덱스가 사용되면서 성능이 갑자기 나빠질 수 있다. 애플리케이션 운영자는 사용자가 불평하기 전까지 그런 사실을 알지 못하며, 사용 빈도가 높은 프로그램에서 그런 현상이 발생해 시스템이 멎기도 한다. DBMS마다 이처럼 차이가 있다는 사실을 미리 숙지하고, 애플리케이션 특성(안정성 우선, 성능 우선 등)에 맞게 개발 표준과 DB 관리정책을 수립할 필요가 있다.

3) 힌트 종류

Oracle은 공식적으로 아래와 같이 많은 종류의 힌트를 제공하며, 비공식 힌트까지 합치면 150여 개에 이른다. 비공식 힌트까지 모두 알 필요는 없지만, 최소한 [표 III-3-4]에 나열한 힌트는 그 용도와 사용법을 숙지할 필요가 있다. 자세한 설명은 Oracle 매뉴얼을 참조하기 바란다.

[표 Ⅲ-3-4] Oracle 힌트

분류	힌트
최적화 목표	all_rows forst_rows(n)
액세스 경로	full cluster hash index, no_index index_asc, index_desc index_combine index_join index_ffs, no_index_ffs index_ss, no_index_ss index_ss_asc, index_ss_desc
쿼리 변환	no_query_transformation use_concat no_expand rewrite, no_rewrite merge, no_merge star_transformation, no_star_transformation fact, no_fact unnest, no_unnest
조인 순서	ordered leading
조인 방식	use_nl, no_use_nl use_nl_with_index use_merge, no_use_merge use_hash, no_use_hash
병렬 처리	parallel, no_parallel pq_distribute parallel_index, no_parallel_index
기타	append, noappend cache, nocache push_pred, no_push_pred push_subq, no_push_subq qb_name cursor_sharing_exact driving_site dynamic_sampling model_min_analysis

나. SQL Server 힌트

SQL Server에서 옵티마이저 힌트를 지정하는 방법으로는 크게 3가지가 있다.

- 테이블 힌트
  테이블명 다음에 WITH절을 통해 지정한다. fastfirstrow, holdlock, nolock 등
- 조인 힌트

FROM절에 지정하며, 두 테이블 간 조인 전략에 영향을 미친다. loop, hash, merge, remote 등

- 쿼리 힌트
  쿼리당 맨 마지막에 한번만 지정할 수 있는 쿼리 힌트는 아래와 같이 OPTION절을 이용한다.

앞에서 설명했듯이, SQL Server는 문법이나 의미적으로 맞지 않게 힌트를 기술하면 프로그램에 에러가 발생한다.