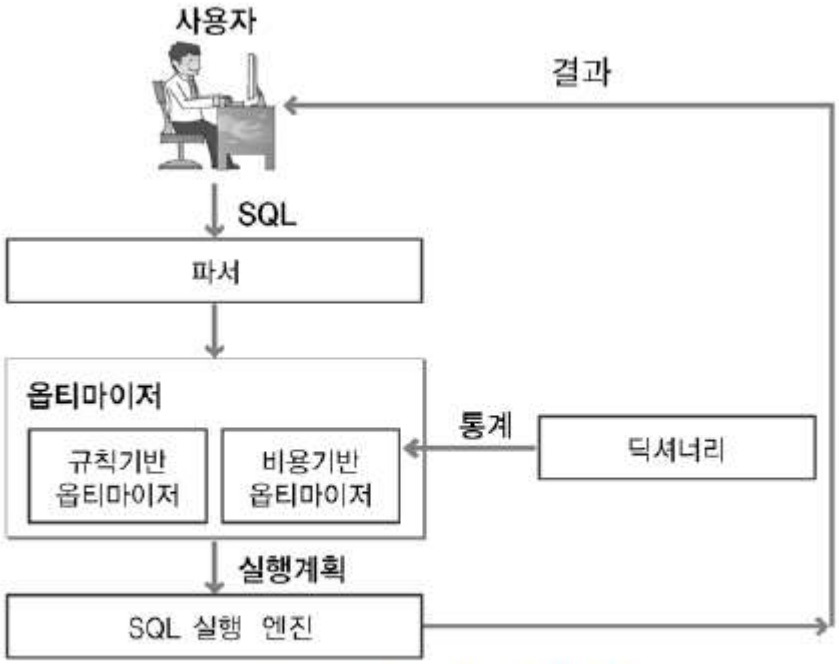
01.옵티마이저와 실행계획

1. 옵티마이저

옵티마이저(Optimizer)는 사용자가 질의한 SQL 문에 대해 최적의 실행 방법을 결정하는 역할을 수행한다. 이러한 최적의 실행 방법을 실행계획(Execution Plan)이라고 한다. 관계형 데이터베이스는 궁극적으로 SQL 문을 통해서만 데이터를 처리할 수 있다. JAVA, C 등과 같은 프로그램 언어와는 달리 SQL 은 사용자의 요구사항만 기술할 뿐 처리과정에 대한 기술은 하지 않는다.

그러므로 사용자의 요구사항을 만족하는 결과를 추출할 수 있는 다양한 실행 방법이 존재할 수 있다. 다양한 실행 방법들 중에서 최적의 실행 방법을 결정하는 것이 바로 옵티마이저의 역할이다. 관계형 데이터베이스는 옵티마이저가 결정한 실행 방법대로 실행 엔진이 데이터를 처리하여 결과 데이터를 사용자에게 전달할 뿐이다. 옵티마이저가 선택한 실행 방법의 적절성 여부는 질의의 수행 속도에 가장 큰 영향 미치게 된다.

이런 의미에서 관계형 데이터베이스에서 진정한 프로그래머는 옵티마이저라고 할 수 있다. 최적의 실행 방법 결정이라는 것은 어떤 방법으로 처리하는 것이 최소 일량으로 동일한 일을 처리할 수 있을지 결정하는 것이다. 그러나 이러한 결정을 옵티마이저는 실제로 SQL 문을 처리해보지 않은 상태에서 결정해야 하는 어려움이 있다. 옵티마이저가 최적의 실행 방법을 결정하는 방식에 따라 [그림 Ⅱ-3-1]과 같이 규칙기반 옵티마이저(RBO, Rule Based Optimizer)와 비용기반 옵티마이저(CBO, Cost Based Optimizer)로 구분할 수 있다.



[그림 Ⅱ-3-1] 옵티마이저의 종류

현재 대부분의 관계형 데이터베이스는 비용기반 옵티마이저만을 제공한다. 비록 규칙기반 옵티마이저를 제공하더라도 신규 기능들에 대해서는 더 이상 지원하지 않는다. 다만 하위 버전 호환성을 위해서만 규칙기반 옵티마이저가 남아 있을 뿐이다. 그렇지만 규칙기반 옵티마이저의 규칙은 보편 타당성에 근거한 것들이다. 이러한 규칙을 알고 있는 것은 옵티마이저의 최적화 작업을 이해하는데 도움이 된다.

가. 규칙기반 옵티마이저

규칙기반 옵티마이저는 규칙(우선 순위)을 가지고 실행계획을 생성한다. 실행계획을 생성하는 규칙을 이해하면 누구나 실행계획을 비교적 쉽게 예측할 수 있다. 규칙기반 옵티마이저가 실행계획을 생성할 때 참조하는 정보에는 SQL 문을 실행하기 위해서 이용 가능한 인덱스 유무와 (유일, 비유일, 단일, 복합 인덱스)종류, SQL 문에서 사용하는 연산자(=, <, <>, LIKE, BETWEEN 등)의 종류 그리고 SQL 문에서 참조하는 객체(힙 테이블, 클러스터 테이블 등)의 종류 등이 있다. 이러한 정보에 따라 우선 순위(규칙)가 정해져 있고, 이 우선 순위를 기반으로 실행계획을 생성한다.

결과적으로 규칙기반 옵티마이저는 우선 순위가 높은 규칙이 적은 일량으로 해당 작업을 수행하는 방법이라고 판단하는 것이다. [그림 II-3-2]는 Oracle 의 규칙기반 옵티마이저의 15 가지 규칙이다. 순위의 숫자가 낮을수록 높은 우선 순위이다.

순위	액세스 기법
1	Single row by rowid
2	Single row by cluster join
3	Single row by hash cluster key with unique or primary key
4	Single row by unique or primary key
5	Cluster join
6	Hash cluster key
7	Indexed cluster key
8	Composite index
9	Single column index
10	Bounded range search on indexed columns
11	Unbounded range search on indexed columns
12	Sort merge join
13	MAX or MIN of indexed column
14	ORDER BY on indexed column
15	Full table scan

[그림 Ⅱ-3-2] 규칙기반 옵티마이저의 규칙

규칙기반 옵티마이저의 우선 순위 규칙 중에서 주요한 규칙에 대해서만 간략히 설명한다.

규칙 1. Single row by rowid : ROWID 를 통해서 테이블에서 하나의 행을 액세스하는 방식이다. ROWID 는 행이 포함된 데이터 파일, 블록 등의 정보를 가지고 있기 때문에 다른 정보를 참조하지 않고도 바로 원하는 행을 액세스할 수 있다. 하나의 행을 액세스하는 가장 빠른 방법이다.

규칙 4. Single row by unique or primary key : 유일 인덱스(Unique Index)를 통해서 하나의 행을 액세스하는 방식이다. 이 방식은 인덱스를 먼저 액세스하고 인덱스에 존재하는 ROWID 를 추출하여 테이블의 행을 액세스한다.

칙 8. Composite index : 복합 인덱스에 동등('=' 연산자) 조건으로 검색하는 경우이다. 예를 들어, 만약 A+B 칼럼으로 복합 인덱스가 생성되어 있고, 조건절에서 WHERE A=10 AND B=1 형태로 검색하는 방식이다. 복합 인덱스 사이의 우선 순위 규칙은 다음과 같다. 인덱스 구성 칼럼의 개수가 더 많고 해당 인덱스의 모든 구성 칼럼에 대해 '=' 로 값이 주어질 수록 우선순위가 더 높다. 예를 들어, A+B 로 구성된 인덱스와 A+B+C 로 구성된 인덱스가 각각 존재하고 조건절에서 A, B, C 칼럼 모

두에 대해 '=' 로 값이 주어진다면 A+B+C 인덱스가 우선 순위가 높다. 만약 조건절에서 A, B 칼럼에만 '=' 로 값이 주어진다면 A+B 는 인덱스의 모든 구성 칼럼에 대해 값이 주어지고 A+B+C 인덱스 입장에서는 인덱스의 일부 칼럼에 대해서만 값이 주어졌기 때문에 A+B 인덱스가 우선 순위가 높게 된다.

규칙 9. Single column index : 단일 칼럼 인덱스에 '=' 조건으로 검색하는 경우이다. 만약 A 칼럼에 단일 칼럼 인덱스가 생성되어 있고, 조건절에서 A=10 형태로 검색하는 방식이다.

규칙 10. Bounded range search on indexed columns : 인덱스가 생성되어 있는 칼럼에 양쪽 범위를 한정하는 형태로 검색하는 방식이다. 이러한 연산자에는 BETWEEN, LIKE 등이 있다. 만약 A 칼럼에 인덱스가 생성되어 있고, A BETWEEN '10' AND '20' 또는 A LIKE '1%' 형태로 검색하는 방식이다.

규칙 11. Unbounded range search on indexed columns : 인덱스가 생성되어 있는 칼럼에 한쪽 범위만 한정하는 형태로 검색하는 방식이다. 이러한 연산자에는 >, >=, <, <= 등이 있다. 만약 A 칼럼에 인덱스가 생성되어 있고, A > '10' 또는 A < '20' 형태로 검색하는 방식이다.

규칙 15. Full table scan : 전체 테이블을 액세스하면서 조건절에 주어진 조건을 만족하는 행만을 결과로 추출한다.

규칙기반 옵티마이저는 인덱스를 이용한 액세스 방식이 전체 테이블 액세스 방식보다 우선 순위가 높다. 따라서 규칙기반 옵티마이저는 해당 SQL 문에서 이용 가능한 인덱스가 존재한다면 전체 테이블 액세스 방식보다는 항상 인덱스를 사용하는 실행계획을 생성한다. 규칙기반 옵티마이저가 조인 순서를 결정할 때는 조인 칼럼 인덱스의 존재 유무가 중요한 판단의 기준이다. 조인 칼럼에 대한 인덱스가 양쪽 테이블에 모두 존재한다면 앞에서 설명한 규칙에 따라 우선 순위가 높은 테이블을 선행 테이블(Driving Table)로 선택한다. 한쪽 조인 칼럼에만 인덱스가 존재하는 경우에는 인덱스가 없는 테이블을 선행 테이블로 선택해서 조인을 수행한다. 조인 칼럼에 모두 인덱스가 존재하지 않으면 FROM 절의 뒤에 나열된 테이블을 선행 테이블로 선택한다. 만약 조인 테이블의 우선 순위가 동일하다면 FROM 절에 나열된 테이블의 역순으로 선행 테이블을 선택한다. 규칙기반 옵티마이저의 조인 기법의 선택은 다음과 같다. 양쪽 조인 칼럼에 모두 인덱스가 없는 경우에는 Sort Merge Join 을 사용하고 둘 중하나라도 조인 칼럼에 인덱스가 존재한다면 일반적으로 NL Join 을 사용한다.

다음 SQL 문을 이용해서 규칙기반 옵티마이저의 최적화 과정을 알아보자.

```
SELECT ENAME FROM EMP WHERE JOB = 'SALESMAN' AND SAL BETWEEN 3000 AND 6000 INDEX ------------------------------- EMP_JOB : JOB EMP_SAL : SAL PK_EMP : EMPNO (UNIQUE)
```

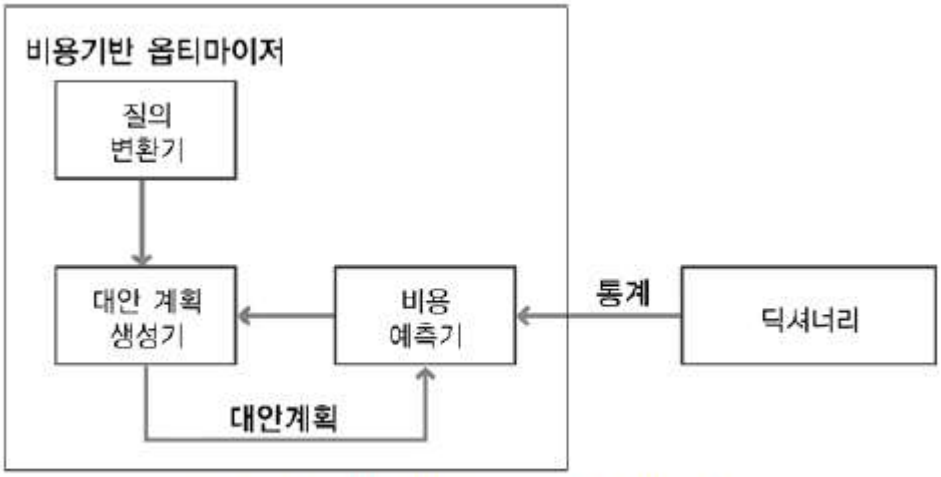
조건절에서 JOB 칼럼의 조건은 '=', SAL 칼럼의 조건은 'BETWEEN' 으로 값이 주어졌고 각각의 칼럼에 단일 칼럼 인덱스가 존재한다. 우선 순위 규칙에 따라 JOB 조건은 규칙 9 의 단일 칼럼 인덱스를 만족하고 SAL 조건은 규칙 10 의 인덱스상의 양쪽 한정 검색을 만족한다. 따라서 우선 순위가 높은 EMP_JOB 인덱스를 이용해서 조건을 만족하는 행에 대해 EMP 테이블을 액세스하는 방식을 선택할 것이다. 다음은 규칙기반 옵티마이저가 생성한 실행계획이다.

```
Execution Plan ---------------------------------------------------------- SELECT STATEMENT Optimizer=CHOOSE TABLE ACCESS (BY INDEX ROWID) OF 'EMP' INDEX (RANGE SCAN) OF 'EMP_JOB' (NON-UNIQUE)
```

나. 비용기반 옵티마이저

규칙기반 옵티마이저는 조건절에서 '=' 연산자와 'BETWEEN' 연산자가 사용되면 규칙에 따라 '=' 칼럼의 인덱스를 사용하는 것이 보다 적은 일량 즉, 보다 적은 처리 범위로 작업을 할 것이라고 판

단한다. 그러나 실제로는 'BETWEEN' 칼럼을 사용한 인덱스가 보다 일량이 적을 수 있다. 단순한 몇 개의 규칙만으로 현실의 모든 사항을 정확히 예측할 수는 없다. 비용기반 옵티마이저는 이러한 규칙기반 옵티마이저의 단점을 극복하기 위해서 출현하였다. 비용기반 옵티마이저는 SQL 문을 처리하는데 필요한 비용이 가장 적은 실행계획을 선택하는 방식이다. 여기서 비용이란 SQL 문을 처리하기 위해 예상되는 소요시간 또는 자원 사용량을 의미한다. 비용기반 옵티마이저는 비용을 예측하기 위해서 규칙기반 옵티마이저가 사용하지 않는 테이블, 인덱스, 칼럼 등의 다양한 객체 통계정보와 시스템 통계정보 등을 이용한다. 통계정보가 없는 경우 비용기반 옵티마이저는 정확한 비용 예측이 불가능해져서 비효율적인 실행계획을 생성할 수 있다. 그렇기 때문에 정확한 통계정보를 유지하는 것은 비용기반 최적화에서 중요한 요소이다.



[그림 Ⅱ-3-3] 비용기반 옵티마이저의 구성 요소

[그림 II-3-3]과 같이 비용기반 옵티마이저는 질의 변환기, 대안 계획 생성기, 비용 예측기 등의 모듈로 구성되어 있다. 질의 변환기는 사용자가 작성한 SQL 문을 처리하기에 보다 용이한 형태로 변환하는 모듈이다. 대안 계획 생성기는 동일한 결과를 생성하는 다양한 대안 계획을 생성하는 모듈이다. 대안 계획은 연산의 적용 순서 변경, 연산 방법 변경, 조인 순서 변경 등을 통해서 생성된다. 동일한 결과를 생성하는 가능한 모든 대안 계획을 생성해야 보다 나은 최적화를 수행할 수 있다. 그러나 대안 계획의 생성이 너무 많아지면 최적화를 수행하는 시간이 그만큼 오래 걸린 수 있다.

그래서 대부분의 상용 옵티마이저들은 대안 계획의 수를 제약하는 다양한 방법을 사용한다. 이러한 현실적인 제약으로 인해 생성된 대안 계획들 중에서 최적의 대안 계획이 포함되지 않을 수도 있다. 비용 예측기는 대안 계획 생성기에 의해서 생성된 대안 계획의 비용을 예측하는 모듈이다. 대안 계획의 정확한 비용을 예측하기 위해서 연산의 중간 집합의 크기 및 결과 집합의 크기, 분포도 등의 예측이 정확해야 한다. 보다 나은 예측을 위해 옵티마이저는 정확한 통계정보를 필요로 한다. 또한 대안 계획을 구성하는 각 연산에 대한 비용 계산식이 정확해야 한다.

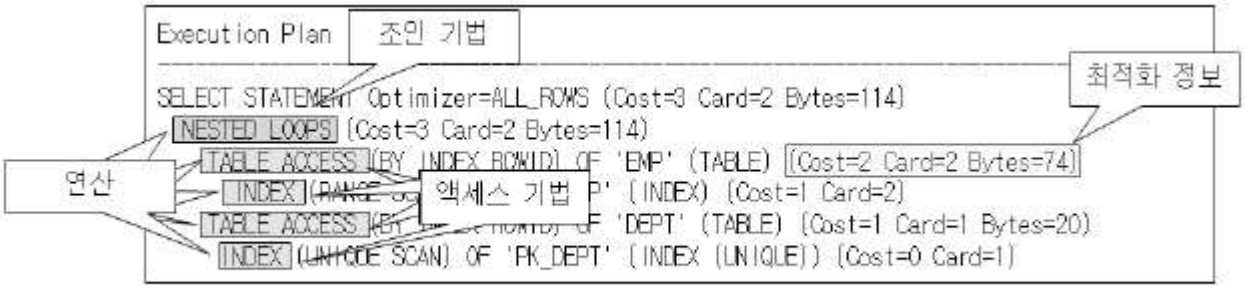
앞에서 규칙기반 옵티마이저는 항상 인덱스를 사용할 수 있다면 전체 테이블 스캔 보다는 인덱스를 사용하는 실행계획을 생성한다고 했다. 그렇지만 비용기반 옵티마이저는 인덱스를 사용하는 비용이 전체 테이블 스캔 비용보다 크다고 판단되면 전체 테이블 스캔을 수행하는 방법으로 실행계획을 생성할 수도 있다.

비용기반 옵티마이저는 통계정보, DBMS 버전, DBMS 설정 정보 등의 차이로 인해 동일 SQL 문도 서로 다른 실행계획이 생성될 수 있다. 또한 비용기반 옵티마이저의 다양한 한계들로 인해 실행계획의 예측 및 제어가 어렵다는 단점이 있다.

2. 실행계획

실행계획(Execution Plan)이란 SQL 에서 요구한 사항을 처리하기 위한 절차와 방법을 의미한다. 실행계획을 생성한다는 것은 SQL 을 어떤 순서로 어떻게 실행할 지를 결정하는 작업이다. 동일한 SQL 에 대해 결과를 낼 수 있는 다양한 처리 방법(실행계획)이 존재할 수 있지만 각 처리 방법마다 실행 시간(성능)은 서로 다를 수 있다. 옵티마이저는 다양한 처리 방법들 중에서 가장 효율적인 방법을 찾아준다. 즉, 옵티마이저는 최적의 실행계획을 생성해 준다.

생성된 실행계획을 보는 방법은 데이터베이스 벤더마다 서로 다르다. 실행계획에서 표시되는 내용 및 형태도 약간씩 차이는 있지만 실행계획이 SQL 처리를 위한 절차와 방법을 의미한다는 기본적인 사항은 모두 동일하다. 실행계획을 보고 SQL 이 어떻게 실행되는지 정확히 이해할 수 있다면 보다 향상된 SQL 의 이해 및 활용이 가능하다. Oracle 의 실행계획 형태는 [그림 Ⅱ-3-4]와 같다. 실행계획을 구성하는 요소에는 조인 순서(Join Order), 조인 기법(Join Method), 액세스 기법(Access Method), 최적화 정보(Optimization Information), 연산(Operation) 등이 있다



[그림 Ⅱ-3-4] 실행계획 정보의 구성요소

조인 순서는 조인작업을 수행할 때 참조하는 테이블의 순서이다. 예를 들어, FROM 절에 A, B 두 개의 테이블이 존재할 때 조인 작업을 위해 먼저 A 테이블을 읽고 B 테이블을 읽는 작업을 수행한다면 조인 순서는 A → B 이다. [그림 Ⅱ-3-4]에서 조인 순서는 EMP → DEPT 이다. 논리적으로 가능한 조인 순서는 n! 만큼 존재한다. 여기서는 n 은 FROM 절에 존재하는 테이블 수이다.

그러나 현실적으로 옵티마이저가 적용 가능한 조인 순서는 이보다는 적거나 같다. 조인 기법은 두 개의 테이블을 조인할 때 사용할 수 있는 방법으로서 여기에는 NL Join, Hash Join, Sort Merge Join 등이 있다. [그림 Ⅱ-3-4]에서 조인 기법은 NL Join 을 사용하고 있다. 액세스 기법은 하나의 테이블을 액세스할 때 사용할 수 있는 방법이다. 여기에는 인덱스를 이용하여 테이블을 액세스하는 인덱스 스캔(Index Scan)과 테이블 전체를 모두 읽으면서 조건을 만족하는 행을 찾는 전체 테이블 스캔(Full Table Scan) 등이 있다. [그림 Ⅱ-3-4]에서 액세스 기법은 인덱스 스캔을 사용하고 있다. 최적화 정보는 옵티마이저가 실행계획의 각 단계마다 예상되는 비용 사항을 표시한 것이다.

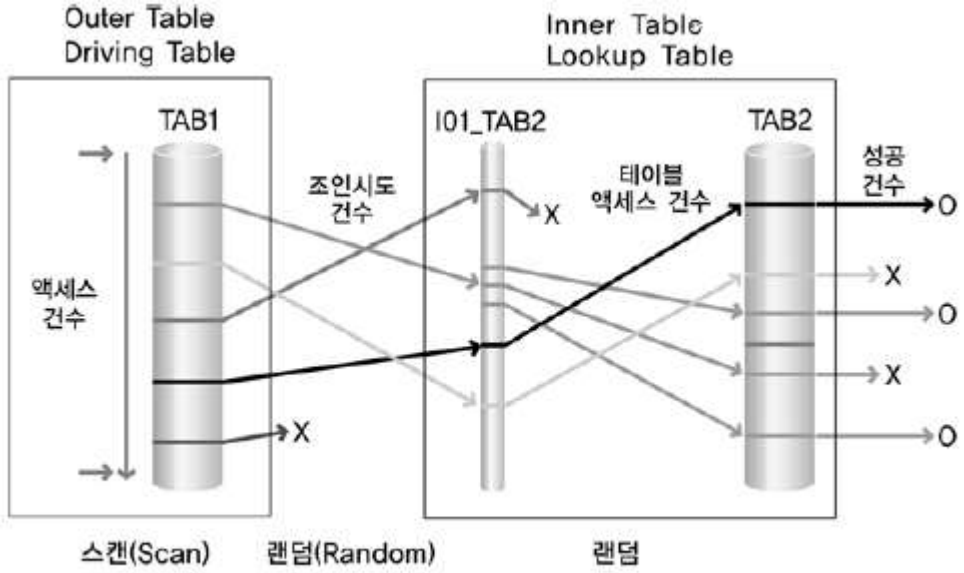
실행계획에 비용 사항이 표시된다는 것은 비용기반 최적화 방식으로 실행계획을 생성했다는 것을 의미한다. 최적화 정보에는 Cost, Card, Bytes 가 있다. Cost 는 상대적인 비용 정보이고 Card 는 Cardinality 의 약자로서 주어진 조건을 만족한 결과 집합 혹은 조인 조건을 만족한 결과 집합의 건수를 의미한다. Bytes 는 결과 집합이 차지하는 메모리 양을 바이트로 표시한 것이다. 이러한 비용 정보는 실제로 SQL 을 실행하고 얻은 결과가 아니라 통계 정보를 바탕으로 옵티마이저가 계산한 예상치이다. 만약 이러한 비용 사항이 실행계획에 표시되지 않았다면 이것은 규칙기반 최적화 방식으로 실행계획을 생성한 것이다.

[그림 Ⅱ-3-4] 실행계획의 예에서는 비용 정보가 표시되어 있으므로 비용기반 최적화 방식으로 생성된 실행계획이다. 연산(Operation)은 여러 가지 조작을 통해서 원하는 결과를 얻어내는 일련의 작업이다. 연산에는 조인 기법(NL Join, Hash Join, Sort Merge Join 등), 액세스 기법(인덱스 스캔, 전체 테이블 스캔 등), 필터, 정렬, 집계, 뷰 등 다양한 종류가 존재한다. 예를 들어, SQL 에서 정렬을 목적으로 ORDER BY 를 수행했다면 정렬 연산이 표시되다.

3. SQL 처리 흐름도

SQL 처리 흐름도(Access Flow Diagram)란 SQL 의 내부적인 처리 절차를 시각적으로 표현한 도표이다. 이것은 실행계획을 시각화한 것이다. [그림 II-3-5]와 같이 액세스 처리 흐름도에는 SQL 문의 처리를 위해 어떤 테이블을 먼저 읽었는지(조인 순서), 테이블을 읽기 위해서 인덱스 스캔을 수행했는지 또는 테이블 전체 스캔을 수행했는지(액세스 기법)과 조인 기법 등을 표현할 수 있다.

예를 들어, [그림 II-3-5]에서 조인 순서는 TAB1 → TAB2 이다. 여기서 TAB1 을 Outer Table 또는 Driving Table 이라고 하고, TAB2 를 Inner Table 또는 Lookup Table 이라고 한다. 테이블의 액세스 방법은 TAB1 은 테이블 전체 스캔을 의미하고 TAB2 는 I01_TAB2 이라는 인덱스를 통한 인덱스 스캔을 했음을 표시한 것이다. 조인 방법은 NL Join 을 수행했음을 표시한 것이다. [그림 II-3-5]에서 TAB1 에 대한 액세스는 스캔(Scan) 방식이고 조인시도 및 I01_TAB2 인덱스를 통한 TAB2 액세스는 랜덤(Random) 방식이다. 대량의 데이터를 랜덤 방식으로 액세스하게 되면 많은 I/O 가 발생하여 성능상 좋지 않다.



[그림 II-3-5] SQL 처리 흐름도

성능적인 관점을 살펴보기 위해서 SQL 처리 흐름도에 일량을 함께 표시할 수 있다. [그림 II-3-5]에서 건수(액세스 건수, 조인 시도 건수, 테이블 액세스 건수, 성공 건수)라고 표시된 곳에 SQL 처리를 위해 작업한 건수 또는 처리 결과 건수 등의 일량을 함께 표시할 수 있다. 이것을 통해 어느 부분에서 비효율이 발생하고 있는지에 대한 힌트를 얻을 수 있다.

다음은 [그림 II-3-5]가 다음 SQL 문에 대한 SQL 처리 흐름도라는 가정으로 설명한다.

```
SELECT … FROM TAB1 A, TAB2 B WHERE A.KEY = B.KEY AND A.COL1 = :condition1 AND B.COL2 = :condition2
```

[그림 II-3-5]에서 액세스 건수는 SQL 처리를 위해 TAB1 을 액세스한 건수이다. 여기서는 TAB1 의 A.COL1 칼럼에 이용 가능한 인덱스가 존재하지 않아 전체 테이블 스캔을 수행했음을 의미한다. 따라서 액세스 건수는 TAB1 테이블의 총 건수와 동일하다. 조인 시도 건수는 TAB1 을 액세스한 후 즉, 테이블에서 읽은 해당 건에 대해 A.COL1 = :condition1 조건을 만족한 건만이 TAB2 와 조인을 시도한다. 즉, TAB1 을 액세스한 후 A.COL1 = :condition1 조건을 만족하지 않는다면 더 이상 조인 작업을 진행할 필요가 없다. 따라서 조인 시도 건수는 TAB1 에 주어진 조건인 A.COL1 = :condition1 을 만족한 건수가 된다.

테이블 액세스 건수는 B.KEY 칼럼만으로 구성된 인덱스(B.KEY 칼럼만으로 구성된 인덱스라고 가정함)인 I01_TAB2 에서 B.KEY = A.KEY (TAB1 은 이미 읽혀졌기 때문에 A.KEY 값은 상수임) 조건

을 만족한 건만이 TAB2 테이블을 액세스한다. 즉, 조인 시도한 건들 중에서 B.KEY = A.KEY 조건까지 만족한 건과 같다. 성공 건수는 SQL 실행을 통해 사용자에게 답으로서 보여지는 결과 건수이다. TAB2 테이블을 액세스해서 B.COL2 = :condition2 조건까지 만족해야 비로서 사용자에게 보여질 수 있다.

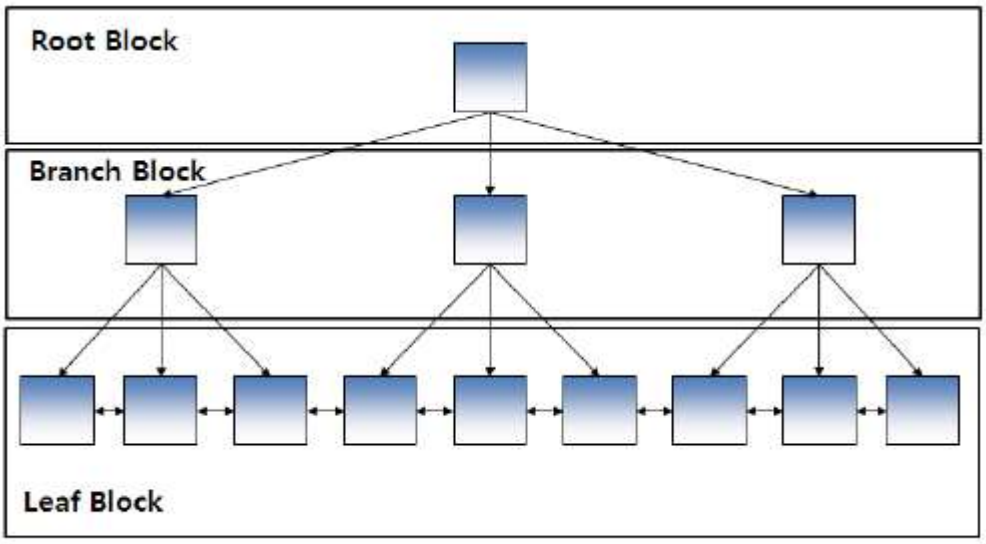
02.인덱스 기본

1. 인덱스 특징과 종류

인덱스는 원하는 데이터를 쉽게 찾을 수 있도록 돕는 책의 찾아보기와 유사한 개념이다. 인덱스는 테이블을 기반으로 선택적으로 생성할 수 있는 구조이다. 테이블에 인덱스를 생성하지 않아도 되고 여러 개를 생성해도 된다. 인덱스의 기본적인 목적은 검색 성능의 최적화이다. 즉, 검색 조건을 만족하는 데이터를 인덱스를 통해 효과적으로 찾을 수 있도록 돕는다. 그렇지만 Insert, Update, Delete 등과 같은 DML 작업은 테이블과 인덱스를 함께 변경해야 하기 때문에 오히려 느려질 수 있다는 단점이 존재한다.

가. 트리 기반 인덱스

DBMS 에서 가장 일반적인 인덱스는 B-트리 인덱스이다.



[그림 Ⅱ-3-6] B-트리 인덱스 구조

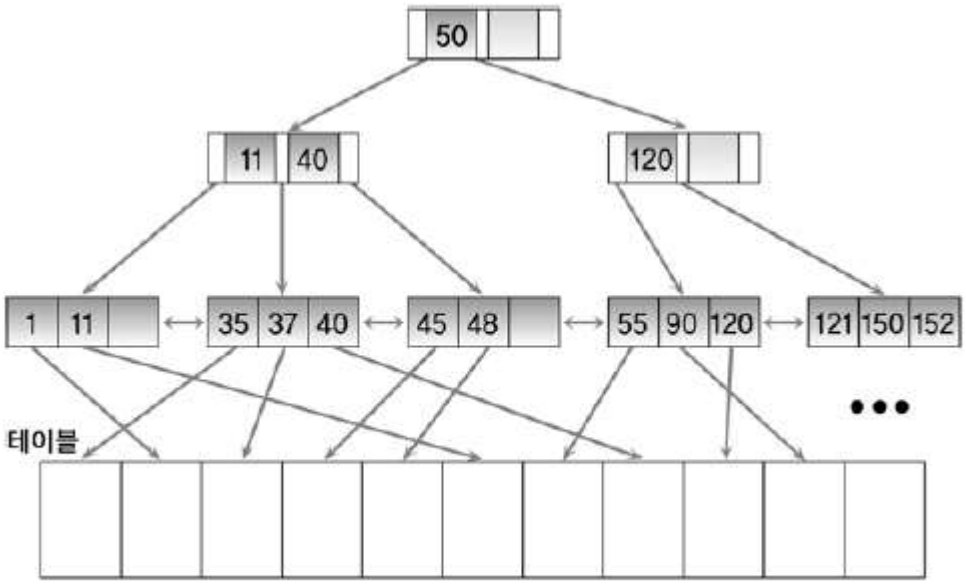
[그림 Ⅱ-3-6]과 같이 B-트리 인덱스는 브랜치 블록(Branch Block)과 리프 블록(Leaf Block)으로 구성된다. 브랜치 블록 중에서 가장 상위에서 있는 블록을 루트 블록(Root Block)이라고 한다.

브랜치 블록은 분기를 목적으로 하는 블록이다. 브랜치 블록은 다음 단계의 블록을 가리키는 포인터를 가지고 있다. 리프 블록은 트리의 가장 아래 단계에 존재한다.

리프 블록은 인덱스를 구성하는 칼럼의 데이터와 해당 데이터를 가지고 있는 행의 위치를 가리키는 레코드 식별자(RID, Record Identifier/Rowid)로 구성되어 있다. 인덱스 데이터는 인덱스를 구성하는 칼럼의 값으로 정렬된다. 만약 인덱스 데이터의 값이 동일하면 레코드 식별자의 순서로 저장된다. 리프 블록은 양방향 링크(Double Link)를 가지고 있다. 이것을 통해서 오름 차순(Ascending Order)과 내림 차순(Descending Order) 검색을 쉽게 할 수 있다. B-트리 인덱스는 '=' 로 검색하는 일치(Exact Match) 검색과 'BETWEEN', '>' 등과 같은 연산자로 검색하는 범위(Range) 검색 모두에 적합한 구조이다. [그림 Ⅱ-3-7]은 브랜치 브록이 3 개의 포인터로 구성된 B-트리 인덱스의 예이다. 인덱스에서 원하는 값을 찾는 과정은 다음과 같다.

1 단계. 브랜치 블록의 가장 왼쪽 값이 찾고자 하는 값보다 작거나 같으면 왼쪽 포인터로 이동 2 단계. 찾고자 하는 값이 브랜치 블록의 값 사이에 존재하면 가운데 포인터로 이동 3 단계. 오른쪽에 있는 값보다 크면 오른쪽 포인터로 이동

이 과정을 리프 블록을 찾을 때까지 반복한다. 리프 블록에서 찾고자 하는 값이 존재하면 해당 값을 찾은 것이고, 해당 값이 없으면 해당 값은 존재하지 않아 검색에 실패하게 된다.



[그림 Ⅱ-3-7] B-트리 인덱스 검색

예를 들어, [그림 Ⅱ-3-7]에서 37 을 찾고자 한다면 루트 블록에서 50 보다 작으므로 왼쪽 포인터로 이동한다. 37 는 왼쪽 브랜치 블록의 11 과 40 사이의 값이므로 가운데 포인터로 이동한다. 이동한 결과 해당 블록이 리프 블록이므로 37 이 블록 내에 존재하는지 검색한다.

본 예에서는 리프 블록에 37 이 존재한다. 검색하고자 하는 값을 찾은 것이다. 만약, SQL 문에서 다른 칼럼이 더 필요하면 리프 블록에 존재하는 레코드 식별자를 이용해서 테이블을 액세스한다. 만약, 37 과 50 사이의 모든 값을 찾고자 한다면(BETWEEN 37 AND 50) 위와 동일한 방법으로 리프 블록에서 37 을 찾고 50 보다 큰 값을 만날 때까지 오른쪽으로 이동하면서 인덱스를 읽는다. 이것은 인덱스 데이터가 정렬되어 있고 리프 블록이 양방향 링크로 연결되어 있기 때문에 가능하다. 인덱스를 경유해서 반환된 결과 데이터는 인덱스 데이터와 동일한 순서로 갖게 되는 특징을 갖는다. 인덱스를 생성할 때 동일 칼럼으로 구성된 인덱스를 중복해서 생성할 수 없다.

그렇지만 인덱스 구성 칼럼은 동일하지만 칼럼의 순서가 다르면 서로 다른 인덱스로 생성할 수 있다. 예를 들어, JOB+SAL 칼럼 순서의 인덱스와 SAL+JOB 칼럼 순서의 인덱스를 별도의 인덱스를 생성할 수 있다. 인덱스의 칼럼 순서는 질의의 성능에 중요한 영향을 미치는 요소이다. Oracle 에서 트리 기반 인덱스에는 B-트리 인덱스 외에도 비트맵 인덱스(Bitmap Index), 리버스 키 인덱스(Reverse Key Index), 함수기반 인덱스(FBI, Function-Based Index) 등이 존재한다.

나. SQL Server 의 클러스터형 인덱스

SQL Server 의 인덱스 종류는 저장 구조에 따라 클러스터형(clustered) 인덱스와 비클러스터형(nonclustered) 인덱스로 나뉜다. 여기서는 클러스터형 인덱스에 대해서만 설명하기로 한다. 클러스터형 인덱스는 두 가지 중요한 특징이 있다.

첫째, 인덱스의 리프 페이지가 곧 데이터 페이지다. 따라서 테이블 탐색에 필요한 레코드 식별자가 리프 페이지에 없다(인덱스 키 칼럼과 나머지 칼럼을 리프 페이지에 같이 저장하기 때문에 테이블을 랜덤 액세스할 필요가 없다). 클러스터형 인덱스의 리프 페이지를 탐색하면 해당 테이블의 모든 칼럼 값을 곧바로 얻을 수 있다. 흔히 클러스터형 인덱스를 사전에 비유한다.

예를 들어, 영한사전은 알파벳 순으로 정렬되어 있으며 각 단어 바로 옆에 한글 설명이 붙어 있다. 전문서적 끝 부분에 있는 찾아보기(=색인)가 페이지 번호만 알려주는 것과 비교하면 그 차이점을 알

수 있다. 둘째, 리프 페이지의 모든 로우(=데이터)는 인덱스 키 칼럼 순으로 물리적으로 정렬되어 저장된다. 테이블 로우는 물리적으로 한 가지 순서로만 정렬될 수 있다. 그러므로 클러스터형 인덱스는 테이블당 한 개만 생성할 수 있다.(전화번호부 한 권을 상호와 인명으로 동시에 정렬할 수 없는 것과 마찬가지다.)

[그림 II-3-8]은 Employee ID, Last Name, First Name, Hire Date 로 구성된 Employees 테이블에 대해 Employee ID 에 기반한 클러스터형 인덱스를 생성한 모습이다. B-트리 구조를 편의상, 삼각형 모양을 왼쪽으로 90 도 돌려서 나타냈다.

EmployeeID	LastName	FirstName	HireDate
1	Davolio	Nancy	1992-05-01 00:00:00.000
2	Fuller	Andrew	1992-08-14 00:00:00.000
3	Leverling	Janet	1992-04-01 00:00:00.000
4	peacock	Margaret	1993-05-03 00:00:00.000
5	Buchanan	Steven	1993-10-17 00:00:00.000
6	Suyama	Michael	1993-10-17 00:00:00.000
7	King	Robart	1994-01-02 00:00:00.000
8	Callahan	Laura	1994-03-05 00:00:00.000
9	Dodsworth	Anne	1994-11-15 00:00:00.000

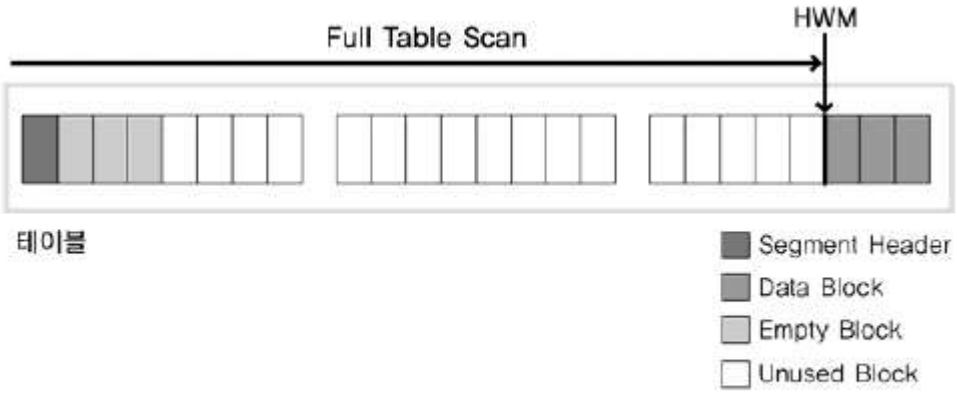
[그림 Ⅱ-3-8] Employees_pk 클러스터형 인덱스

[그림 II-3-8]에 표시된 것처럼, 리프 블록에 인덱스 키 칼럼 외에도 테이블의 나머지 칼럼이 모두 함께 있다.

2. 전체 테이블 스캔과 인덱스 스캔

가. 전체 테이블 스캔

전체 테이블 스캔 방식으로 데이터를 검색한다는 것은 테이블에 존재하는 모든 데이터를 읽어 가면서 조건에 맞으면 결과로서 추출하고 조건에 맞지 않으면 버리는 방식으로 검색한다.



[그림 Ⅱ-3-9] 전체 테이블 스캔

Oracle 의 경우 [그림 II-3-9]와 같이 검색 조건에 맞는 데이터를 찾기 위해서 테이블의 고수위 마크(HWM, High Water Mark) 아래의 모든 블록을 읽는다. 고수위 마크는 테이블에 데이터가 쓰여졌던 블록 상의 최상위 위치(현재는 지워져서 데이터가 존재하지 않을 수도 있음)를 의미한다. 전체 테이블 스캔 방식으로 데이터를 검색할 때 고수위 마크까지의 블록 내 모든 데이터를 읽어야 하기 때

문에 모든 결과를 찾을 때까지 시간이 오래 걸릴 수 있다. 이와 같이 전체 테이블 스캔 방식은 테이블에 존재하는 모든 블록의 데이터를 읽는다.

그러나 이것은 결과를 찾기 위해 꼭 필요해서 모든 블록을 읽었다기 보다 Full Table Scan 연산이었기 때문에 모든 블록을 읽은 것이다. 따라서 이렇게 읽은 블록들은 재사용성이 떨어진다. 그래서 전체 테이블 스캔 방식으로 읽은 블록들은 메모리에서 곧 제거될 수 있도록 관리된다.

옵티마이저가 연산으로서 전체 테이블 스캔 방식을 선택하는 이유는 일반적으로 다음과 같다.

1) SQL 문에 조건이 존재하지 않는 경우

SQL 문에 조건이 존재하지 않는다는 것은 테이블에 존재하는 모든 데이터가 답이 된다는 것이다. 그렇기 때문에 테이블의 모든 블록을 읽으면서 무조건 결과로서 반환하면 된다.

2) SQL 문의 주어진 조건에 사용 가능한 인덱스가 존재하는 않는 경우

사용 가능한 인덱스가 존재하지 않는다면 데이터를 액세스할 수 있는 방법은 테이블의 모든 데이터를 읽으면서 주어진 조건을 만족하는지를 검사하는 방법뿐이다. 또한 주어진 조건에 사용 가능한 인덱스는 존재하나 함수를 사용하여 인덱스 칼럼을 변형한 경우에도 인덱스를 사용할 수 없다.

3) 옵티마이저의 취사 선택

조건을 만족하는 데이터가 많은 경우, 결과를 추출하기 위해서 테이블의 대부분의 블록을 액세스해야 한다고 옵티마이저가 판단하면 조건에 사용 가능한 인덱스가 존재해도 전체 테이블 스캔 방식으로 읽을 수 있다.

4) 그 밖의 경우

병렬처리 방식으로 처리하는 경우 또는 전체 테이블 스캔 방식의 힌트를 사용한 경우에 전체 테이블 스캔 방식으로 데이터를 읽을 수 있다.

나. 인덱스 스캔

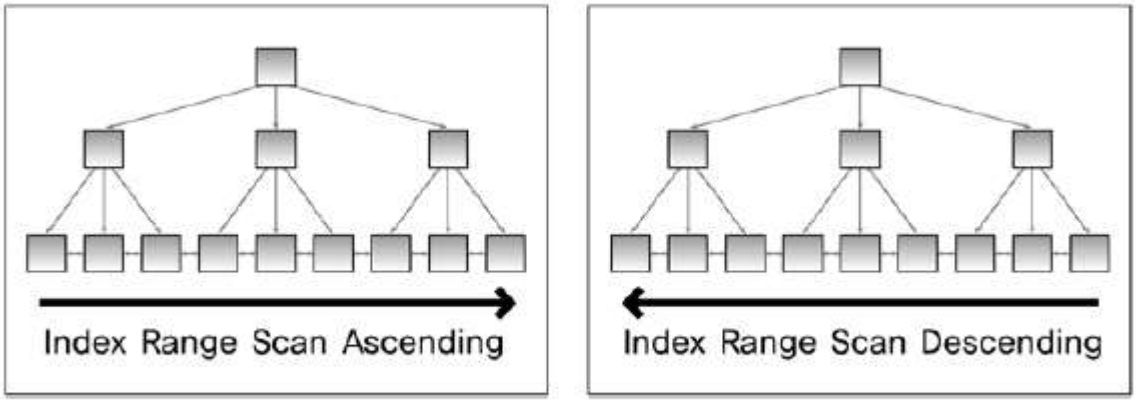
여기서는 데이터베이스에서 주로 사용되는 트리 기반 인덱스를 중심으로 설명한다. 인덱스 스캔은 인덱스를 구성하는 칼럼의 값을 기반으로 데이터를 추출하는 액세스 기법이다. 인덱스의 리프 블록은 인덱스 구성하는 칼럼과 레코드 식별자로 구성되어 있다. 따라서 검색을 위해 인덱스의 리프 블록을 읽으면 인덱스 구성 칼럼의 값과 테이블의 레코드 식별자를 알 수 있다. 인덱스에 존재하지 않는 칼럼의 값이 필요한 경우에는 현재 읽은 레코드 식별자를 이용하여 테이블을 액세스해야 한다.

그러나 SQL 문에서 필요로 하는 모든 칼럼이 인덱스 구성 칼럼에 포함된 경우 테이블에 대한 액세스는 발생하지 않는다. 인덱스는 인덱스 구성 칼럼의 순서로 정렬되어 있다. 인덱스의 구성 칼럼이 A+B 라면 먼저 칼럼 A 로 정렬되고 칼럼 A 의 값이 동일할 경우에는 칼럼 B 로 정렬된다. 그리고 칼럼 B 까지 모두 동일하면 레코드 식별자로 정렬된다. 인덱스가 구성 칼럼으로 정렬되어 있기 때문에 인덱스를 경유하여 데이터를 읽으면 그 결과 또한 정렬되어 반환된다.

따라서 인덱스의 순서와 동일한 정렬 순서를 사용자가 원하는 경우에는 정렬 작업을 하지 않을 수 있다. 인덱스 스캔 중에서 자주 사용되는 인덱스 유일 스캔(Index Unique Scan), 인덱스 범위 스캔(Index Range Scan), 인덱스 역순 범위 스캔(Index Range Scan Descending)에 대해 간단히 설명하면 다음과 같다. 제공되는 인덱스 스캔 방식은 데이터베이스 벤더마다 다를 수 있다.

1) 인덱스 유일 스캔은 유일 인덱스(Unique Index)를 사용하여 단 하나의 데이터를 추출하는 방식이다. 유일 인덱스는 중복을 허락하지 않는 인덱스이다. 유일 인덱스 구성 칼럼에 모두 '='로 값이 주어지면 결과는 최대 1 건이 된다. 인덱스 유일 스캔은 유일 인덱스 구성 칼럼에 대해 모두 '=' 로 값이 주어진 경우에만 가능한 인덱스 스캔 방식이다.

2) 인덱스 범위 스캔은 인덱스를 이용하여 한 건 이상의 데이터를 추출하는 방식이다. 유일 인덱스의 구성 칼럼 모두에 대해 '=' 로 값이 주어지지 않은 경우와 비유일 인덱스(Non-Unique Index)를 이용하는 모든 액세스 방식은 인덱스 범위 스캔 방식으로 데이터를 액세스하는 것이다. [그림 II-3-10]의 왼쪽 그림과 같은 방식으로 인덱스를 읽는다.



[그림 Ⅱ-3-10] 인덱스 오름/내림 차순 범위 스캔

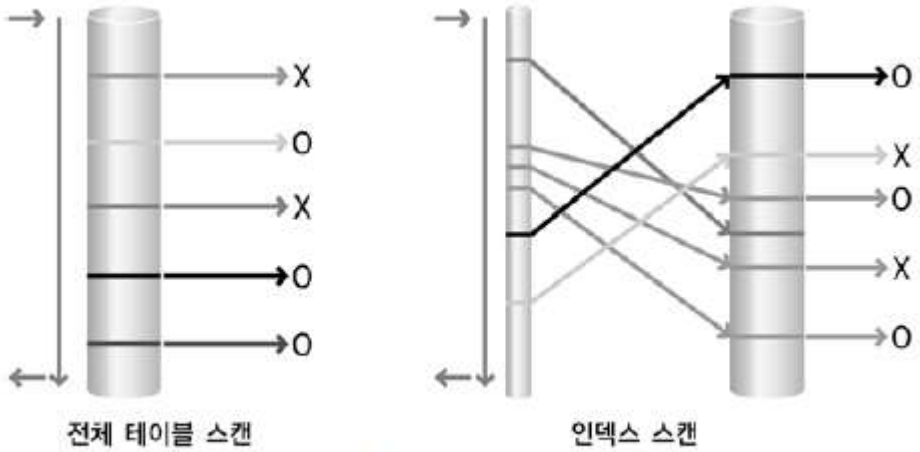
다. 전체 테이블 스캔과 인덱스 스캔 방식의 비교

3) 인덱스 역순 범위 스캔은 [그림 II-3-10]의 오른쪽 그림과 같이 인덱스의 리프 블록의 양방향 링크를 이용하여 내림 차순으로 데이터를 읽는 방식이다. 이 방식을 이용하여 최대값(Max Value)을 쉽게 찾을 수 있다. 이 또한 인덱스 범위 스캔의 일종이다.

이외에도 인덱스 전체 스캔(Index Full Scan), 인덱스 고속 전체 스캔(Fast Full Index Scan), 인덱스 스킵 스캔(Index Skip Scan) 등이 존재한다.

가. 규칙기반 옵티마이저

데이터를 액세스하는 방법은 크게 두 가지로 나눠 볼 수 있다. 인덱스를 경유해서 읽는 인덱스 스캔 방식과 테이블의 전체 데이터를 모두 읽으면서 데이터를 추출하는 전체 테이블 스캔 방식이다. 인덱스 스캔 방식은 사용 가능한 적절한 인덱스가 존재할 때만 이용할 수 있는 스캔 방식이지만 전체 테이블 스캔 방식은 인덱스의 존재 유무와 상관없이 항상 이용 가능한 스캔 방식이다. 앞에서 설명한 것처럼 옵티마이저는 인덱스가 존재하더라도 전체 테이블 스캔 방식을 취사 선택할 수 있다. [그림 II-3-11]은 전체 테이블 스캔과 인덱스 스캔에 대한 SQL 처리 흐름도 표현의 예이다.



[그림 Ⅱ-3-11] 전체 테이블 스캔과 인덱스 스캔에 대한 SQL 처리 흐름도 표현 예시

인덱스 스캔은 인덱스에 존재하는 레코드 식별자를 이용해서 검색하는 데이터의 정확한 위치를 알고서 데이터를 읽는다. 그렇기 때문에 인덱스 스캔 방식에서는 불필요하게 다른 블록을 더 읽을 필요가 없다. 따라서 한번의 I/O 요청에 한 블록씩 데이터를 읽는다. 그러나 전체 테이블 스캔은 데이터를 읽을 때 한번의 I/O 요청으로 여러 블록을 한꺼번에 읽는다. 어차피 테이블의 모든 데이터를 읽을 것이라면 한 번 읽기 작업을 할 때 여러 블록을 함께 읽는 것이 효율적이다. 대용량 데이터 중에서 극히 일부의 데이터를 찾을 때, 인덱스 스캔 방식은 인덱스를 이용해 몇 번의 I/O 만으로 원하는 데이터를 쉽게 찾을 수 있다.

그러나 전체 테이블 스캔은 테이블의 모든 데이터를 읽으면서 원하는 데이터를 찾아야 하기 때문에 비효율적인 검색을 하게 된다. 그러나 반대로 테이블의 대부분의 데이터를 찾을 때는 한 블록씩 읽는 인덱스 스캔 방식 보다는 어차피 대부분의 데이터를 읽을 거라면 한번에 여러 블록씩 읽는 전체 테이블 스캔 방식이 유리할 수 있다.

03.조인 수행 원리

조인이란 두 개 이상의 테이블을 하나의 집합으로 만드는 연산이다. SQL 문에서 FROM 절에 두 개 이상의 테이블이 나열될 경우 조인이 수행된다. 조인 연산은 두 테이블 사이에서 수행된다. FROM 절에 A, B, C 라는 세 개의 테이블이 존재하더라도 세 개의 테이블이 동시에 조인이 수행되는 것은 아니다. 세 개의 테이블 중에서 먼저 두 개의 테이블에 대해 조인이 수행된다. 그리고 먼저 수행된 조인 결과와 나머지 테이블 사이에서 조인이 수행된다. 이러한 작업은 FROM 절에 나열된 모든 테이블을 조인할 때까지 반복 수행한다.

예를 들어, A, B, C 세 개의 테이블을 조인할 때를 가정으로 설명하면 다음과 같다. 먼저 A 와 B 두 테이블을 먼저 조인하면 해당 조인 결과와 나머지 C 테이블을 조인한다(A → B → C). 만약, A 와 C 테이블을 먼저 조인한다면 해당 조인 결과와 나머지 B 테이블을 조인한다(A → C → B). 테이블 또는 조인 결과를 이용하여 조인을 수행할 때 조인 단계별로 다른 조인 기법을 사용할 수 있다. 예를 들어, A 와 B 테이블을 조인할 때는 NL Join 기법을 사용하고 해당 조인 결과와 C 테이블을 조인할 때는 Hash Join 기법을 사용할 수 있다. 조인 기법은 두 개의 테이블을 조인할 때 사용할 수 있는 방법이다. 여기서는 조인 기법 중에서 자주 사용되는 NL Join, Hash Join, Sort Merge Join 에 대해서 조인 원리를 간단하게 설명한다.

1. NL Join

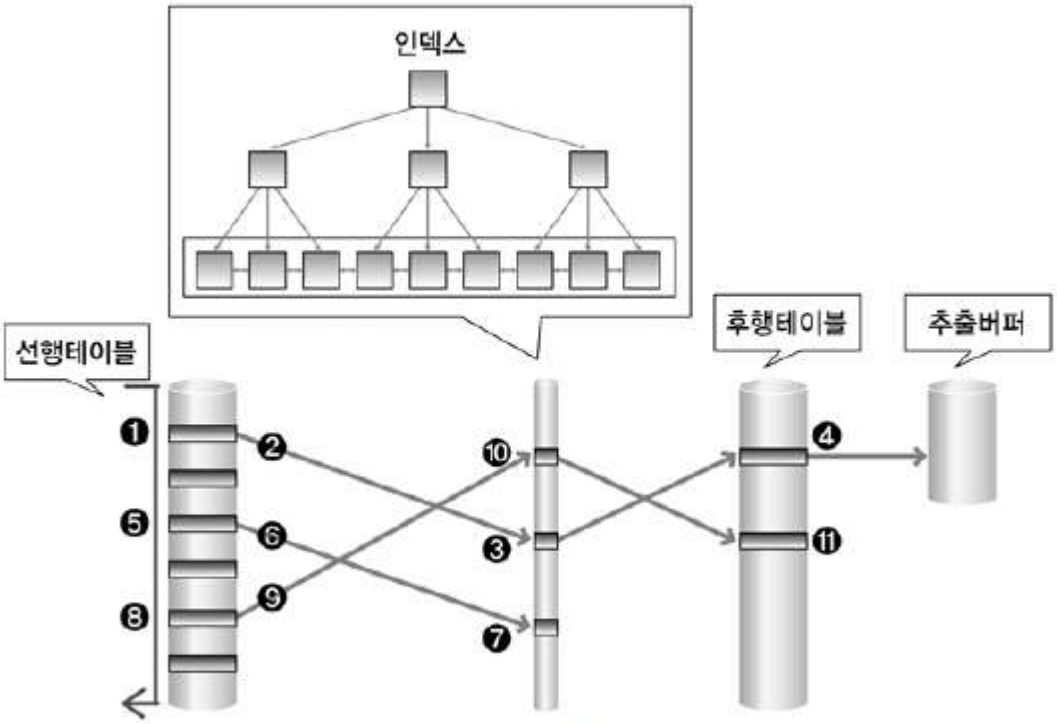
NL Join 은 프로그래밍에서 사용하는 중첩된 반복문과 유사한 방식으로 조인을 수행한다. 반복문의 외부에 있는 테이블을 선행 테이블 또는 외부 테이블(Outer Table)이라고 하고, 반복문의 내부에 있는 테이블을 후행 테이블 또는 내부 테이블(Inner Table)이라고 한다.

```
FOR 선행 테이블 읽음 → 외부 테이블(Outer Table) FOR 후행 테이블 읽음 → 내부 테이블(Inner Table)
(선행 테이블과 후행 테이블 조인)
```

먼저 선행 테이블의 조건을 만족하는 행을 추출하여 후행 테이블을 읽으면서 조인을 수행한다. 이 작업은 선행 테이블의 조건을 만족하는 모든 행의 수만큼 반복 수행한다. NL Join 에서는 선행 테이블의 조건을 만족하는 행의 수가 많으면(처리 주관 범위가 넓으면), 그 만큼 후행 테이블의 조인 작업은 반복 수행된다. 따라서 결과 행의 수가 적은(처리 주관 범위가 좁은) 테이블을 조인 순서상 선행 테이블로 선택하는 것이 전체 일량을 줄일 수 있다. NL Join 은 랜덤 방식으로 데이터를 액세스하기 때문에 처리 범위가 좁은 것이 유리하다.

NL Join 의 작업 방법은 다음과 같다.

① 선행 테이블에서 주어진 조건을 만족하는 행을 찾음 ② 선행 테이블의 조인 키 값을 가지고 후행 테이블에서 조인 수행 ③ 선행 테이블의 조건을 만족하는 모든 행에 대해 1 번 작업 반복 수행



[그림 Ⅱ-3-12] NL Join

[그림 Ⅱ-3-12]의 예를 이용하여 NL Join 의 수행 방식을 알아보도록 하자. [그림 Ⅱ-3-12]에서 인덱스는 B-트리 인덱스의 리프 블록만을 그린 것임을 표현한 것이다.

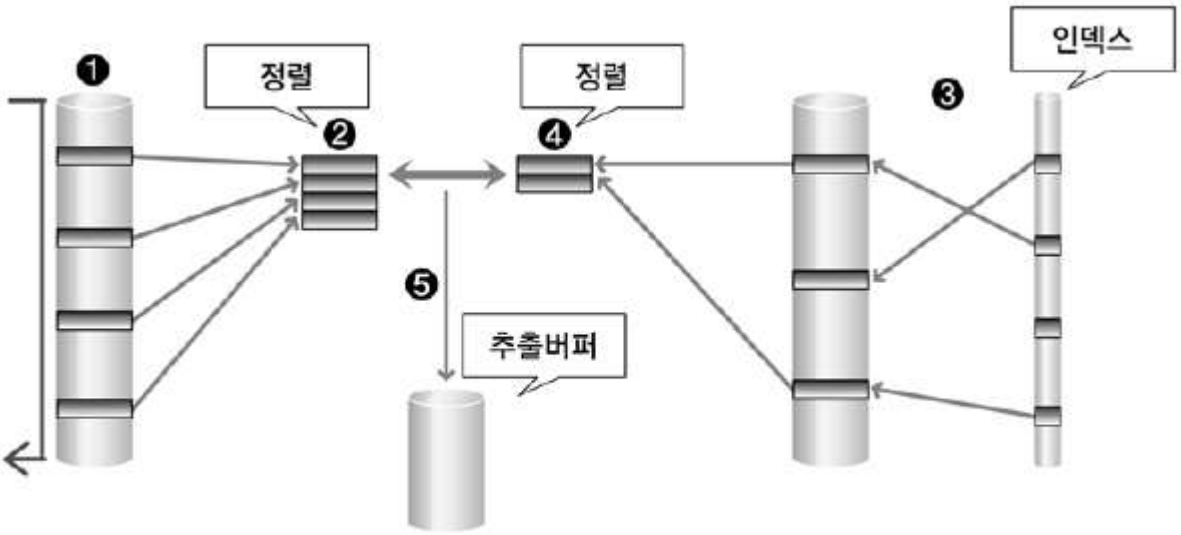
① 선행 테이블에서 조건을 만족하는 첫 번째 행을 찾음 → 이때 선행 테이블에 주어진 조건을 만족하지 않는 경우 해당 데이터는 필터링 됨 ② 선행 테이블의 조인 키를 가지고 후행 테이블에 조인 키가 존재하는지 찾으러 감 → 조인 시도 ③ 후행 테이블의 인덱스에 선행 테이블의 조인 키가 존재하는지 확인 → 선행 테이블의 조인 값이 후행 테이블에 존재하지 않으면 선행 테이블 데이터는 필터링 됨 (더 이상 조인 작업을 진행할 필요 없음) ④ 인덱스에서 추출한 레코드 식별자를 이용하여 후행 테이블을 액세스 → 인덱스 스캔을 통한 테이블 액세스 후행 테이블에 주어진 조건까지 모두 만족하면 해당 행을 추출버퍼에 넣음 ⑤ ~ ⑪ 앞의 작업을 반복 수행함

추출버퍼는 SQL 문의 실행결과를 보관하는 버퍼로서 일정 크기를 설정하여 추출버퍼에 결과가 모두 차거나 더 이상 결과가 없어서 추출버퍼를 채울 것이 없으면 결과를 사용자에게 반환한다. 추출버퍼는 운반단위, Array Size, Prefetch Size 라고도 한다. [그림 Ⅱ-3-12]에서 만약 선행 테이블에 사용 가능한 인덱스가 존재한다면 인덱스를 통해 선행 테이블을 액세스할 수 있다. (여기서는 사용할 인덱스가 없음을 가정으로 설명한 것임) NL Join 기법은 조인이 성공하면 바로 조인 결과를 사용자에게 보여 줄 수 있다. 그래서 결과를 가능한 빨리 화면에 보여줘야 하는 온라인 프로그램에 적당한 조인 기법이다.

2. Sort Merge Join

Sort Merge Join 은 조인 칼럼을 기준으로 데이터를 정렬하여 조인을 수행한다. NL Join 은 주로 랜덤 액세스 방식으로 데이터를 읽는 반면 Sort Merge Join 은 주로 스캔 방식으로 데이터를 읽는다. Sort Merge Join 은 랜덤 액세스로 NL Join 에서 부담이 되던 넓은 범위의 데이터를 처리할 때 이용되던 조인 기법이다. 그러나 Sort Merge Join 은 정렬할 데이터가 많아 메모리에서 모든 정렬 작업을 수행하기 어려운 경우에는 임시 영역(디스크)을 사용하기 때문에 성능이 떨어질 수 있다.

일반적으로 대량의 조인 작업에서 정렬 작업을 필요로 하는 Sort Merge Join 보다는 CPU 작업 위주로 처리하는 Hash Join 이 성능상 유리하다. 그러나 Sort Merge Join 은 Hash Join 과는 달리 동등 조인 뿐만 아니라 비동등 조인에 대해서도 조인 작업이 가능하다는 장점이 있다.



[그림 Ⅱ-3-13] Sort Merge Join

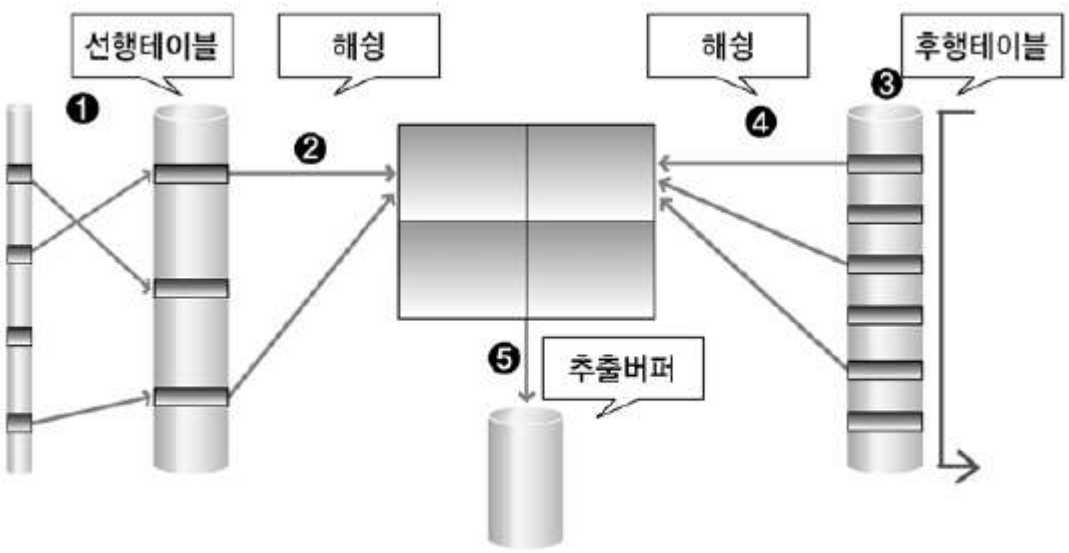
Sort Merge Join 의 동작은 [그림 Ⅱ-3-13]과 같다.

① 선행 테이블에서 주어진 조건을 만족하는 행을 찾음 ② 선행 테이블의 조인 키를 기준으로 정렬 작업을 수행 ① ~ ②번 작업을 선행 테이블의 조건을 만족하는 모든 행에 대해 반복 수행 ③ 후행 테이블에서 주어진 조건을 만족하는 행을 찾음 ④ 후행 테이블의 조인 키를 기준으로 정렬 작업을 수행 ③ ~ ④번 작업을 후행 테이블의 조건을 만족하는 모든 행에 대해 반복 수행 ⑤ 정렬된 결과를 이용하여 조인을 수행하며 조인에 성공하면 추출버퍼에 넣음

Sort Merge Join 은 조인 칼럼의 인덱스를 사용하지 않기 때문에 조인 칼럼의 인덱스가 존재하지 않을 경우에도 사용할 수 있는 조인 기법이다. Sort Merge Join 에서 조인 작업을 위해 항상 정렬 작업이 발생하는 것은 아니다. 예를 들어, 조인할 테이블 중에서 이미 앞 단계의 작업을 수행하는 도중에 정렬 작업이 미리 수행되었다면 조인을 위한 정렬 작업은 발생하지 않을 수 있다.

3. Hash Join

Hash Join 은 해쉉 기법을 이용하여 조인을 수행한다. 조인을 수행할 테이블의 조인 칼럼을 기준으로 해쉬 함수를 수행하여 서로 동일한 해쉬 값을 갖는 것들 사이에서 실제 값이 같은지를 비교하면서 조인을 수행한다. Hash Join 은 NL Join 의 랜덤 액세스 문제점과 Sort Merge Join 의 문제점인 정렬 작업의 부담을 해결 위한 대안으로 등장하였다.



[그림 Ⅱ-3-14] Hash Join

Hash Join 의 동작은 [그림 Ⅱ-3-14]와 같다.

① 선행 테이블에서 주어진 조건을 만족하는 행을 찾음 ② 선행 테이블의 조인 키를 기준으로 해쉬 함수를 적용하여 해쉬 테이블을 생성 → 조인 칼럼과 SELECT 절에서 필요로 하는 칼럼도 함께 저장됨 ① ~ ②번 작업을 선행 테이블의 조건을 만족하는 모든 행에 대해 반복 수행 ③ 후행 테이블에서 주어진 조건을 만족하는 행을 찾음 ④ 후행 테이블의 조인 키를 기준으로 해쉬 함수를 적용하여 해당 버킷을 찾음 → 조인 키를 이용해서 실제 조인될 데이터를 찾음 ⑤ 조인에 성공하면 추출버퍼에 넣음 ③ ~ ⑤번 작업을 후행 테이블의 조건을 만족하는 모든 행에 대해서 반복 수행

Hash Join 은 조인 칼럼의 인덱스를 사용하지 않기 때문에 조인 칼럼의 인덱스가 존재하지 않을 경우에도 사용할 수 있는 조인 기법이다. Hash Join 은 해쉬 함수를 이용하여 조인을 수행하기 때문에 '='로 수행하는 조인 즉, 동등 조인에서만 사용할 수 있다. 해쉬 함수를 적용한 값은 어떤 값으로 해쉬될 지 알 수 없다. 해쉬 함수가 적용될 때 동일한 값은 항상 같은 값으로 해쉬됨이 보장된다. 그러나 해쉬 함수를 적용할 때 보다 큰 값이 항상 큰 값으로 해쉬되고 작은 값이 항상 작은 값으로 해쉬된다는 보장은 없다. 그렇기 때문에 Hash Join 은 동등 조인에서만 사용할 수 있다.

[그림 Ⅱ-3-14]와 같이 Hash Join 은 조인 작업을 수행하기 위해 해쉬 테이블을 메모리에 생성해야 한다. 생성된 해쉬 테이블의 크기가 메모리에 적재할 수 있는 크기보다 더 커지면 임시 영역(디스크)에 해쉬 테이블을 저장한다. 그러면 추가적인 작업이 필요해 진다. 그렇기 때문에 Hash Join 을 할 때는 결과 행의 수가 적은 테이블을 선행 테이블로 사용하는 것이 좋다. 선행 테이블의 결과를 완전히 메모리에 저장할 수 있다면 임시 영역에 저장하는 작업이 발생하지 않기 때문이다. Hash Join 에서는 선행 테이블을 이용하여 먼저 해쉬 테이블을 생성한다고 해서 선행 테이블을 Build Input 이라고도 하며, 후행 테이블은 만들어진 해쉬 테이블에 대해 해쉬 값의 존재여부를 검사한다고 해서 Prove Input 이라고도 한다.