2과목 3장 최적화의 원리

● 옵티마이저

- SQL의 실행계획을 수립하고 SQL을 실행하는 DBMS의 소프트웨어
- 사용자가 질의한 SQL문에 대해 최적의 실행 방법을 결정하는 역할 수행
- 옵티마이저의 실행계획은 SQL 성능에 아주 중요한 역할
- 동일한 결과가 나오는 SQL도 어떻게 실행하느냐에 따라 성능이 달라진다.
- 비용기반 옵티마이저를 기본적으로 사용한다.

● 실행계획 : SQL에서 요구한 사항을 처리하기 위한 절차와 방법을 의미, 실행계획을 구성하는 요소에는 조인 순서, 조인 기법, 액세스 기법, 최적화 정보, 연산 등이 있다.

● SQL 처리 흐름도 : SQL의 내부적인 처리 절차를 시각적으로 표현한 도표, 조인순서, 액세스기법, 조인기법 등 표현 가능하다.

● 힌트(HINT) : 개발자가 옵티마이저에게 실행계획을 알려주는 것

● 옵티마이저 엔진

(1) Query Transformer : SQL문을 효율적으로 실행하기 위해서 옵티마이저가 변환, SQL이 변환되어도 그 결과는 동일
(2) Estimator : 통계정보를 사용해서 SQL 실행비용을 계산, 총비용은 최적의 실행계획을 수립하기 위한 비용
(3) Plan Generator : SQL을 실행할 실행계획 수립

● 규칙 기반 옵티마이저 (RBO)

- 15개의 규칙에 우선순위를 두어 실행계획을 생성
- 인덱스 유무와 SQL문에서 참조하는 객체 등을 참고

(1) ROWID를 사용한 단일 행인 경우
(2) 클러스터 조인에 의한 단일 행인 경우
(3) 유일하거나 기본키를 가진 해시 클러스터 키에 의한 단일 행인 경우
(4) 유일하거나 기본키에 의한 단일 행인 경우
(5) 클러스터 조인인 경우
(6) 해시 클러스터 조인인 경우
(7) 인덱스 클러스터 키인 경우
(8) 복합 칼럼 인덱스인 경우
(9) 단일 칼럼 인덱스인 경우
(10)인덱스가 구성된 칼럼에서 제한된 범위를 검색하는 경우
(11)인덱스가 구성된 칼럼에서 무제한 범위를 검색하는 경우
(12) 정렬-병합(Sort-Merge) 조인인 경우
(13) 인덱스가 구성된 칼럼에서 MAX/MIN을 구하는 경우
(14) 인덱스가 구성된 칼럼에서 ORDER BY를 실행하는 경우
(15) 전체 테이블 스캔하는 경우

● 비용 기반 옵티마이저 (CBO)

- 테이블, 인덱스, 칼럼, 오브젝트 통계 및 시스템 통계를 사용해 총비용을 계산하여 비용이 가장 적은 실행계획을 수립
- 통계정보가 부적절한 경우 성능 저하가 발생
- 총비용이란 SQL문을 실행하기 위해 예상되는 소요시간 혹은 자원의 사용량을 의미
- 현재 대부분의 DB에서 사용

● 인덱스

- 데이터를 빠르게 검색할 수 있는 방법을 제공
- 인덱스 키로 정렬되어 있기 때문에 원하는 데이터를 빠르게 조회
- 오름차순/내림차순 탐색이 가능
- 하나의 테이블에 여러 개의 인덱스를 생성할 수 있고 하나의 인덱스는 여러 개의 칼럼으로 구성될 수 있다.
- 테이블을 생성할 때 기본키는 자동으로 인덱스 생성

● 인덱스의 구조

- 인덱스의 구조는 Root / Branch / Leaf 로 구성

(1) Root : 인덱스 트리에서 가장 상위에 있는 노드
(2) Branch : 다음 단계의 주소를 가지고 있는 포인터
(3) Leaf : 인덱스 키와 ROWID로 구성, 인덱스 키는 정렬되어서 저장되어 있다. Double Linked List 형태여서 양방향 탐색이 가능하다.

● B-TREE 인덱스에서 원하는 값을 찾는 과정

- 가장 일반적인 인덱스, 관계형 DB에서 가장 많이 사용
- 데이터 중 10% 미만 데이터를 검색할 때 유용

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

● 인덱스 생성

- CREATE INDEX 문을 사용해서 생성
- 인덱스를 생성할 때는 한 개 이상의 칼럼을 사용해서 생성할 수 있다.
- 인덱스 키는 기본적으로 오름차순 정렬, DESC 구를 포함하여 내림차순으로 정렬 가능

● 인덱스 스캔

- 인덱스를 구성하는 칼럼의 값을 기반으로 데이터를 추출하는 액세스 기법

(1) 인덱스 유일 스캔(Index Unique SCAN)
- 인덱스의 키 값이 중복되지 않은 경우, 해당 인덱스를 사용할 때 발생

(2) 인덱스 범위 스캔(Index Range SCAN)
- SELECT 문에서 특정 범위를 조회하는 WHERE 문을 사용한 경우 발생, (Like, Betwwen)이 대표적이다.
- 데이터 양이 적은 경우는 인덱스 자체를 실행하지 않고 전체 테이블 스캔(TABLE FULL SCAN)이 될 수 있다.
- 인덱스 범위 스캔은 Leaf Block의 특정 범위를 스캔한 것

(3) 인덱스 역순 범위 스캔(Index Reverse Range SCAN)
- 인덱스의 리프 블록의 양방향 링크를 이용하여 내림차순으로 데이터를 읽는다.

(4) 전체 테이블 스캔 (TABLE FULL SCAN)
- 테이블에 존재하는 모든 데이터를 읽으면서 조건에 맞으면 결과로 추출하고 조건에 맞지 않으면 버리는 방식으로 검색

● 전체 테이블 스캔을 하는 경우
- SQL 문에 조건이 존재하지 않는 경우
- SQL 문에 주어진 조건에 사용 가능한 인덱스가 없는 경우
- 옵티마이저의 취사 선택
- 병렬처리 방식으로 처리하는 경우

● 전체 테이블 스캔 시에 High Water Mark 의 의미
- 테이블을 읽을 때 High Water Mark 이하 까지만 전체 테이블 스캔을 한다.
- High Water Mark 는 테이블에 데이터가 저장된 블록에서 최상의 위치를 의미하고 데이터가 삭제되면 High Water Mark 가 변경된다.

● 파티션 인덱스(Partition Index)
(1) Global Index
- 여러 개의 파티션에서 하나의 인덱스를 사용
(2) Local Index
- 해당 파티션 별로 각자의 인덱스를 사용
(3) Prefixed Index
- 파티션 키와 인덱스 키가 동일
(4) Non Prefixed Index
- 파티션 키와 인덱스 키가 다름

√ 카디널리티 : 행의 수

● 실행계획을 읽는 순서
(1) 안 -> 밖
(2) 위 -> 아래

● 옵티마이저 조인
(1) Nested Join (NL Join)
- 프로그래밍에서 사용하는 중첩된 반복문과 유사한 방식으로 조인을 수행
- 랜덤 엑세스 방식으로 데이터를 읽는다. 랜덤 엑세스가 많이 발생하면 성능 저하가 발생
- 하나의 테이블에서 데이터를 먼저 찾고 그 다음 테이블을 조인하는 방식

(2) Sort Merge Join
- 조인 칼럼을 기준으로 데이터를 정렬하여 조인 수행
- 스캔 방식으로 데이터를 읽음
- 두 개의 테이블을 SORT_AREA 라는 메모리 공간에 모두 로딩하고 정렬을 수행
- 두 개의 테이블에 대해서 정렬이 완료되면 병합
- 정렬이 발생하기 때문에 데이터 양이 많아지면 성능 저하
- 정렬 데이터 양이 너무 많으면 정렬은 임시 영역에서 수행하며 임시 영역은 디스크에 있기 때문에 성능이 급격하게 떨어진다.

(3) Hash Join
- CPU 작업 위주로 처리, 해싱 기법 이용
- NL Join의 랜덤 엑세스 문제와 Sort Merge Join 의 정렬 작업 부담을 해결하기 위한 대안으로 등장
- 두 개의 테이블 중에서 작은 테이블을 Hash메모리에 로딩
- 두 개의 테이블의 조인 키를 사용해서 해시 테이블 생성
- 해시함수를 사용해서 주소 계산, 해당 주소를 사용해서 테이블을 조인하기 때문에 CPU 연산을 많이 한다.
- Hash Join 시에는 선행 테이블의 크기가 작아서 충분히 메모리에 로딩되어야 한다.

√ From 절에 기술한 테이블 순서대로 조인을 하는 힌트
Oracle : ORDERED
SQL Server : option(force order)