DB

SQL 성능은 인덱스에 좌우된다

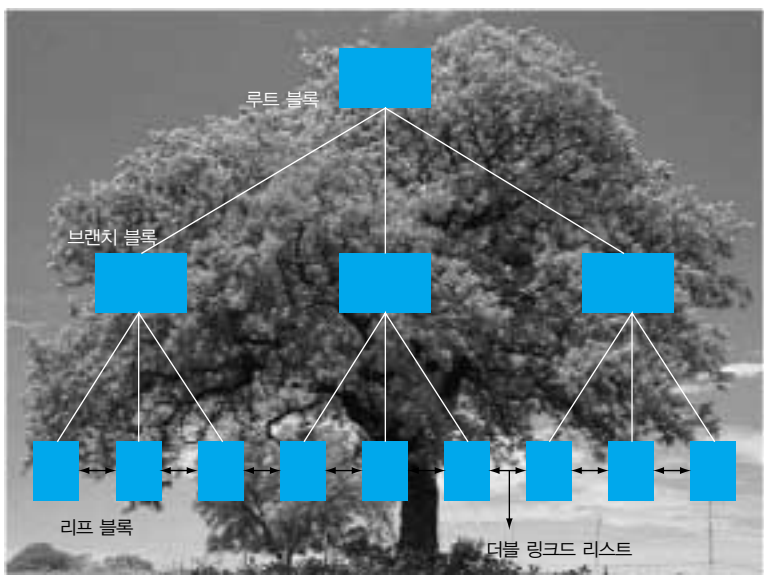
B*TREE 인덱스의 구조

SQL의 성능은 인덱스와 분리하여 생각할 수 없다. 인덱스의 구성 및 종류에 따라 SQL의 성능이 좌우되는 것이 현실이기 때문이다. 오라클에서 제공하는 인덱스에는 B*TREE 인덱스, 비트맵 인덱스, 역방향 키 인덱스 및 함수 기반 인덱스 등이 있다. 이 가운데 가장 많이 사용하는 인덱스가 B*TREE 인덱스이다. 이번 호에는 B*TREE 인덱스의 구조에 대해 살펴보자.

B*TREE 인덱스는 균형을 이룬 인덱스이다

B*TREE 인덱스는 B Tree 인덱스의 공간 사용에서 진화된 형태로 Balanced Tree의 약자이다. 언뜻 단어의 생김새만 보면 '균형 잡힌 나무'라는 뜻이 된다. 그렇다면 균형 잡힌 나무라는 것이 무엇을 의미하는 것일까? 균형 잡힌 나무에 대해 이해하기 위해서는 B*TREE 인덱스가 어떻게 생겼는지 먼저 알아야 한다.

B*TREE 인덱스는 〈그림 1〉과 같은 형태로 생성된다. 이처럼 인덱스가 나무 모양을 하고 있기 때문에 B*TREE 인덱스라고 불리게 되었다. B*TREE 인덱스는 균형 잡힌 형태를 가지고 있게 되며 이러한 균형에 의해 B*TREE 인덱스의 특징이 생기게 된다.



〈그림 1〉 B*TREE 인덱스의 형태

B*TREE 인덱스의 구성은 절대 복잡하지 않다

많은 사람들은 인덱스 구조가 매우 복잡하다고 생각한다. 하지만 인덱스의 구조를 이해한 후에는 B*TREE 인덱스의 구조가 매우 간단하다는 것을 이해할 것이다. 〈그림 1〉을 참조하여 B*TREE 인덱스의 구조를 파헤쳐 보자.

인덱스는 데이터베이스에서 세그먼트에 속한다. 세그먼트란 저장 공간을 가지고 있는 데이터베이스 오브젝트를 의미한다. 따라서 인덱스는 디스크에 저장 공간을 가지게 된다. 디스크에 저장 공간을 가진다는 뜻은 디스크에 인덱스를 위한 블록이 존재한다는 의미이다. 이러한 인덱스를 위한 블록은 크게 세가지로 구분된다.

나무의 뿌리에서 시작하는 수분의 최종 목표 지점은 나무의 잎이 될 것이다. 나무의 잎까지 수분을 공급해야 그 나무는 정상적으로 성장하게 된다. 수분이 뿌리에서 잎까지 가기 위해서는 나뭇가지를 거쳐야만 한다. 바로 이것이 B*TREE 인덱스이다. B*TREE 인덱스에서는 나무의 뿌리를 루트 블록이라 하며 나뭇가지를 브랜치 블록이라 한다. 최종 목적지인 나무의 잎을 리프 블록이라고 한다.

첫 번째 인덱스 블록이 루트 블록(Root Block)이다. 루트 블록은 나무의 뿌리와 같다. 나무가 생명을 유지하기 위한 수분을 공급 받기 위해서는 뿌리와 연결되어 있어야 한다. 나무에 공급되는 수분의 시작은 뿌리이기 때문이다. 인덱스에서는 루트 블록이 나무의 뿌리와 같은 역할을 수행하게 된다. 루트 블록을 통해서만 해당 인덱스의 모든 데이터를 액세스할 수 있으며 인덱스의 데이터 삽입 및 변경은 루트 블록 액세스를 시작으로 수행되게 된다.

이렇기 때문에 루트 블록을 인덱스의 뿌리라고 말하는 것이다. 루트 블록에는 분기 값과 블록 주소(DBA, Data Block Address)를 가지게 된다. 해당 분기 값은 액세스하고자 하는

값에 따라 좌측으로 갈지, 우측으로 갈지를 결정해주는 값이 된다. 결국 해당 나무의 잎을 찾아가기 위해 뿌리를 거쳐 어떤 나뭇가지를 찾아갈 것인지를 알려주는 역할을 하게 된다.

두 번째 인덱스 블록이 브랜치 블록(Branch Block)이다. 브랜치 블록은 나뭇가지와 같다. 나뭇가지가 어떤 역할을 하게 되는가? 나뭇가지는 나무의 뿌리와 잎을 연결해 주는 역할을 하게 된다. 결국, 루트 블록과 가장 밑에 있는 리프 블록을 연결해 주는 역할을 수행하게 되는 것이다. 리프 블록은 루트 블록과 마찬가지로 분기 값과 블록 주소(DBA, Data Block Address)를 가지게 된다. 분기 값 또한 루트 블록의 분기 값과 동일하게 어떤 리프 블록으로 가야 할지를 알려주는 역할을 수행하게 된다.

세 번째 인덱스 블록이 리프 블록(Leaf Block)이다. 리프 블록은 인덱스 블록 중 가장 하단에 위치하고 있다. 리프 블록은 인덱스 키 값과 ROWID로 구성된다. 리프 블록에는 우리가 액세스하고자 하는 데이터를 저장하고 있다. 루트 블록 및 브랜치 블록을 통해 액세스하고자 하는 데이터를 저장하고 있는 리프 블록까지 액세스했다면 인덱스를 만들 때 사용한 컬럼의 값인 인덱스 키 값과 테이블의 해당 데이터를 찾아갈 수 있는 주소인 ROWID를 저장하고 있으므로 원하는 데이터를 모두 추출할 수 있게 된다.

위와 같이 B*TREE 인덱스의 블록들은 자신의 역할을 수행하여 원하는 값을 추출할 수 있게 구성되어 있다. 이처럼 우리가 복잡하다고 생각하는 인덱스의 구조는 절대 복잡하지 않다는 것을 알아야 할 것이다.

B*TREE 인덱스의 특징은 균형에서 발생한다

B*TREE 인덱스는 수많은 특징을 가지고 있다. 이러한 현상이 생기는 것은 B*TREE 인덱스가 균형 잡힌 인덱스이기 때문에 발생하는 현상이다.

하나의 데이터를 B*TREE 인덱스를 통해 액세스하는 경우를 확인해 보자. 테이블에 존재하는 하나의 데이터를 인덱스를 통해 액세스하기 위해 필요한 것은 무엇인가? 반드시 알아야 할 것은 인덱스 키 컬럼의 값과 ROWID이다. 예를 들어 사원 테이블에서 사원번호가 '1234'인 데이터를 액세스하고, 사원번호 컬럼에는 인덱스가 존재한다고 가정하자. 해당 인덱스를 이용하여 데이터를 액세스하고자 한다면 사원번호가 '1234'인 데이터가 테이블의 어디에 위치하는가에 대한 주소를 알아야 한다.

사원번호가 '1234'인 데이터의 주소를 인지한다는 것은 대상 데이터의 ROWID를 인지한다는 것이다. ROWID는 인덱스의 키 컬럼인 사원 이름 값과 함께 리프 블록에 모두 저장되어 있다. 그렇게 때문에 사원번호가 '1234'인 데이터를 인덱스를 통해 추출하기 위해서는 루트 블록, 브랜치 블록 및 리프 블록 순서로 액세스를 수행해야 한다.

리프 블록에 존재하는 각각의 인덱스 키 컬럼을 찾기 위해서는 루트 블록, 브랜치 블록 및 리프 블록의 동일한 순서에 의해 데이터를 추출한다. 위와 같은 이유에서 하나의 데이터를 액세스하는 경우 동일한 응답 속도를 보장해 줄 수 있다.

이러한 특성을 가지는 B*TREE 인덱스는 하나의 데이터를 추출하는데 동일한 응답 속도를 보장해 주기 때문에 온라인 트랜잭션에서 주로 사용되는 인덱스이다.

B*TREE 인덱스의 균형은 파괴될 수 있다

B*TREE 인덱스의 가장 중요한 특징인 인덱스의 균형은 파괴될 수 있다. 데이터가 인덱스의 어느 한쪽으로 집중된다면 해당 데이터가 저장되는 부분의 리프 블록이 계속적으로 증가될 것이다. 증가되는 데이터에 의해 인덱스 블록 분할이 발생하게 되고 이에 따라 인덱스의 균형은 파괴될 수 있다는 뜻이다.

이와 같은 경우는 해당 인덱스에 많은 데이터가 한쪽 리프 블록으로 삽입 또는 삭제될 경우이다. 결국에는 테이블에 많은 데이터가 삽입되고 삭제되는 현상이 자주 발생한다면 인덱스의 균형은 파괴되기 쉽다는 이야기와 동일하다.

B*TREE 인덱스의 균형이 파괴된다면 B*TREE 인덱스의 특징이 파괴되는 것과 같다. 인덱스의 균형 파괴로 인덱스를 이용하는 SQL의 응답 속도가 저하될 수 있다. 이런 경우 인덱스 재구축(Index Rebuild)을 수행해야만 인덱스의 균형이 바로 잡히게 된다. 필요에 따라 인덱스 재구성은 응답 속도 향상을 위해 반드시 필요한 작업이다.

권순용 kwontra@hanmail.net | SKC&C에서 DBA 업무를 담당하고 있다. 프로젝트에서는 DB 아키텍처, 튜닝 및 모델링을 주로 수행했다. 저서로는 정보문화사의 Perfect! 오라클 실전 튜닝 및 초보자를 위한 오라클 10g가 있다. 또한 많은 정보를 공유하고자 DB 관련 카페를 운영한다. 지금은 새로운 SQL 튜닝 서적과 개인적으로 DB 관련 특허를 준비하고 있다.