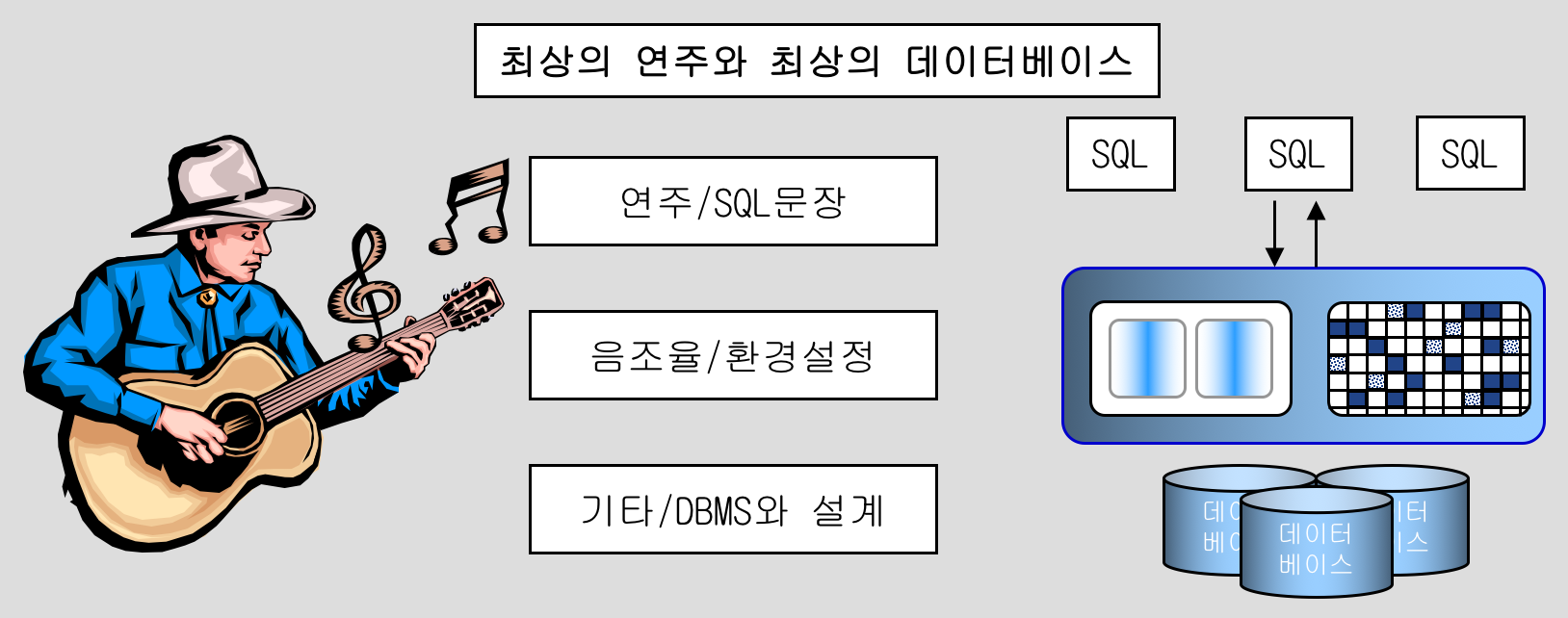
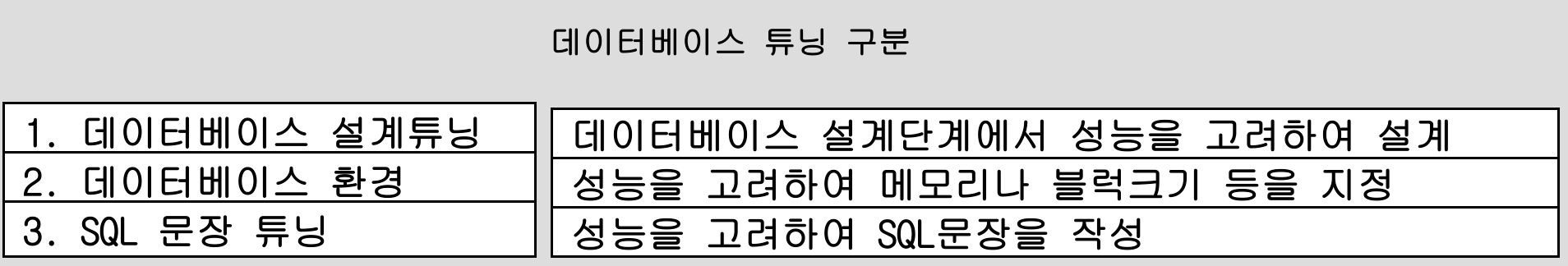
최상의 데이터 관리와 성능에 데이터베이스의 기본 설계 및 DBMS 자체성능, 데이터베이스 환경설정, SQL 문장 작성 방법이 모두 영향을 미치게 한다.



현재 사용하고 있는 대부분의 DBMS는 이미 대용량 데이터베이스에서도 어느 정도 성능을 보장한다. 그렇다면 데이터베이스를 튜닝하는 부분으로는 데이터베이스 설계와 데이터베이스와 관련된 환경 설정 및 SQL무장 작성으로 나눌 수 있다.

이 세가지를 별도로 구분하지 않고 서로 연관지어 튜닝하는 경우도 있지만 이미 설계단계에서 성능을 고려한 데이터베이스 설계를 진행하므로 구축과 운영 시점에는 데이터베이스 환경 설정과 SQL문장에 대한 튜닝이 발생하게 된다.



데이터 용량이 증가할수록 데이터베이스 튜닝의 필요성이 늘어가고 있다.

10.1 데이터 베이스 튜닝의 목적

튜닝을 수행하는 이유는 물론 데이터베이스의 활용 성능을 최상/최적으로 만들기 위해서이다.  
이를 좀더 구체적으로 표현하면 다음과 같이 정리할 수 있다.

첫째 업무적인 환경과 시스템적 환경에 적합한 데이터베이스 파라미터를 설정한다.

예를 들면, 정렬 작업이 많이 발생하는 업무에서는 정렬 작업을 위한 메모리 공간을 충분히 확보해야 하지만, 읽어오는 순서대로 데이터를 조회하는 업무에서는 정렬 공간을 많이 두어 메모리를 낭비할 필요가 없다.

둘째 데이터베이스에 접근하는 SQL 문장은 가능한 한 디스크 블록에 최소로 접근하도록 한다. 디스크를 적절하게 분산시켜 디스크 I/O가 집중되는 현상을 막으면 경합이 발생하지 않기 때문에 성능을 보장할 수 있다.

셋째 디스크 블록에서 한 번 읽은 데이터는 가능하면 메모리 영역에 보관하여 나중에 데이터를 빨리 가져오게 한다. 메모리에서 데이터 가져오기와 디스크에서 메모리로 로드한 이후에 데이터를 가져오는 경우에 둘 사이의 성능 차이가 크다. 당연히 메모리에서 데이터를 읽어오는 편이 빠르므로 디스크 입출력을 최소화할 수 있도록 튜닝한다.

넷째 모든 사용자의 SQL 문장은 공유가 가능하도록 대소문자 등 명명 표준을 준수하여 작성한다. 처음에 실행된 SQL 문장은 내부 파싱절차를 거쳐 DBMS안에서 최적의 실행 계획을 수립한다.

동일한 SQL문장에 대해서는 파싱작업을 다시 수행하지 않으므로 성능을 향상시킬 수 있다.

다섯째 잠금 발생이 최소가 되도록 한다. 데이터베이스에 여러 명의 사용자가 안정적으로 접근할 수 있도록 관리하는 잠금 기능이 전체적인 데이터베이스 성능을 저하시키는 경우가 있다. 잠금이 최소한으로 발생하도록 트랜잭션의 분산과 같은 방법을 활용해야 한다.

여섯째 배치 작업과 백업 작업 수행을 빠른 시간 안에 완료할 수 있어야 한다, 일반적으로 배치 작업에 걸리는 시간이 길기 때문에 작업을 수행하는 동안 메모리나 CPU등 하드웨어 자원을 많이 사용하게 된다. 가능하면 온라인 시스템이 가동되는 시간에는 배치 작업을 수행하지 않도록 하고, 꼭 필요하다면 빠른 시간 안에 수행될 수 있도록 배려해야 전체 시스템의 성능 저하 현상을 예방할 수 있다.

10.2 데이터베이스 설계 단계에서의 튜닝

설계단계에서 성능에 영향을 주는 주 요소는 데이터 모델의 반정규화, PRIMARY KEY설계, 접근 방법에 대한 설계가 있다.

데이터 정합성을 유지할 수 있는 대책을 마련하고, 성능을 위해 필요하다면 테이블, 컬럼, 관계에 대해 반정규화를 적용한다.

대용량 테이블의 경우 필요한 데이터에 대해서는 파티셔닝을 이용하여 테이블 분할을 검토한다,

이력을 관리해야 하는 테이블에 대해서는 필요하다면 시작과 종료나 현재 진행상태 등을 명확하게 명시하여 SQL문장의 실행 성능을 보장하도록 한다.

테이블 접근 유형에 따라 전체 스캔 방식이나 B-TREE인덱스, 비트맵 인덱스, 클러스터링과 해싱 적용 등을 고려한다.  
 테이블이 조회를 주로 하는지 입력, 수정, 삭제 작업이 주로 발생하는지를 고려하여 작당한 인덱스 개수를 지정해야 한다. 극단적으로 조회만 발생하는 경우에는 테이블에 인덱스가 많이 있어도 무관하지만 입력, 수정, 삭제가 빈번한 테이블에서는 인덱스가 많으면 매번 인덱스를 수정해야 하므로 성능을 저하시키는 원인이 된다.

분산 데이터베이스를 적용했을 경우 원격 데이터베이스를 이용할 때 성능 저하가 예상된다면 스냅샷을 이용한 복제 테이블 생성 등을 고려한다.

공통적으로 관리하는 코드 데이터에 대한 접근이 빈번하다면 응용 애플리케이션의 메모리에 상주시키고 함수를 사용하여 코드 변환을 하도록 유도한다,.

PRIMARY KEY는 일반적으로 지정된 순서로 인덱스를 생성하므로 KEY를 구성하는 컬럼 순서를 복합 컬럼 인덱스를 지정하는 규칙에 따라 나열한다.

FOREIGN KEY에 대해서는 가급적 인덱스를 생성하여 전체 스캔이 발생하는 경우와 불필요하게 발생하는 잠금을 피하돌고 한다.

복합 컬럼 인덱스를 지정하는 규칙은 첫째, 항상 조회 조건으로 사용하는 컬럼이 존재한다면 맨 앞에 오게 한다. 기업의 업무에서는 일반적으로 사업소코드, 지사코드, 지점코드 또는 회사코드와 같이 지여ㅛㄱ을 나타내거나 데이터를 분리하는 단위를 첫째 컬럼으로 이용한다. 둘째는 WHERE조건분포도가 낮은 컬럼이 앞쪽에 오도록 한다. 넷째는 정렬이 자주 발생하는 컬럼이 앞쪽에 오게 한다.

SYSTEM테이블 스페이스에는 데이터를 관리하는 딕셔러리 정보만 포함하고, 일반 오브젝트는 저장하지 않도록 한다.

테이블을 위한 테이블 스페이스와 인덱스를 위한 테이블 스페이스를 분리한다.

롤백 세그먼트에 대한 경합을 피하기 위해 롤백 세그먼트를 여러 개로 구성한다.

자주 수정하거나 변경 혹은 삭제되는 불안전한 데이터는 별도의 테이블 스페이스를 만들어 생성한다.

10.3 데이터베이스 환경 튜닝

공유풀 튜닝

오라클 공유풀 영역은 라이브러리 캐시와 딕셔러리 캐시로 구분이 되며 라이브러리 캐시에는 SQL문장이 저장되고, 딕셔러리 캐시에는 데이터베이스에 관련된 각종 정보들을 딕셔러리로 저장된다. 오라클 데이터베이스 환경을 튜닝할 때 공유풀 튜닝이 가장 중요하고 가장 먼저 고려해야 한다.

1) 라이브러리 캐시튜닝

라이브러리 캐시에서는 SQL문을 실행하면 실행된 문장을 파싱하여 메모리 영역에 LRU알고리즘에 따라 보관하였다가 동일한 문장이 다시 실행되면 다시 파싱하지 않고 실행하므로 실행 성능을 향상시킨다.

HIT율이 90%보다 크게 나와야 하며 만약 작을 경우 SHARED\_POOL\_SIZE를 크게 하고, SQL문장에 대한 명명표준을 준수하여 동일한 문장에 대해 재파싱 작업이 일어나지 않도록 한다. 상수를 사용하기보다는 바인딩 변수를 사용한다.

2) 딕셔러리 캐시튜닝

딕셔러리 캐시에는 데이터베이스에 대한 각종 환경 정보와 오브젝트 생성 정보 등이 저장된 공간이다. 딕셔러리 캐시를 위해 ㅜ별도로 메모리를 할당하지 않으며 SHARED\_POOL\_SIZE에 의해 할당된다.

GETS에 대한 GETMISSES 비율이 15%보다 작게 나와야 한다. 만약 이보다 크게 나왔다면 SHARED\_POOL\_SIZE를 크게 한다.

데이터베이스 버퍼 캐시 튜닝

SQL문장을 실행하면 오라클은 디스크에서 데이터를 읽어들여 이를 메모리로 즉, 데이터베이스 버퍼 캐시에 저장한다. 데이터베이스에서 원하는 데이터를 빠르게 조회하기 위해서는 디스크보다는 메모리에서 읽는 편이 성능이 좋으므로 자주 사용되는 데이터를 버퍼 캐시에 저장할 수 있도록 환경을 설정한다. 버퍼 캐시에 영향을 주는 파라미터가 DB\_BLOCK\_BUFFER의 크기를 조절해야 한다.

시스템의 특성에 따라 다르지만 보통 온라인 프로그램 환경에서는 90%이상이면 아주 좋은 경우이고, 80% 이하로 값이 나온다면 버퍼 캐시의 크기를 증가시키는 편을 고려한다.

리두로그 버퍼 튜닝

리두로그 버퍼는 모든 SQL문장이 실행될 때 이용하는 메모리 영역이므로 작업하는 동안은 이 메모리 공간을 이용해야 한다. 데이터 대해 수정이 발생하면 잠금을 설정하듯이 메모리 영역에서 다른 프로세스가 이용하지 못하도록 독점하는 방법으로 래치(LATCH)를 할당 받아 작업 한다.

래치를 할당받지 못한 SQL문장들은 래치를 할당 받을 때까지 기다려야 한다. 그러므로 대기율을 분석하여 만약 오랜 시간동안 대기하고 있거나 잦은 대기 상태가 되는 경우에는 리두로그 버퍼의 크기를 조정해야 한다.

만약 WAIT\_RATIO가 1% 이상이 나오면 리두로그가 발생할 때 경합이 발생한 경우이다.

리두로그 버퍼를 얻기 위해 기다린 횟수를 정기적으로 점검하여 만약 횟수가 지속적으로 중가하거나 비 이상적으로 증가했다면 리두로그 버퍼의 크기를 늘리는 편을 고려한다.

디스크 I/O튜닝

특정 디스크에 있는 데이터 파일에 I/O가 집중되면 이는 성능 저하의 요인이 되므로 I/O현황을 분석하여 집중된 I/O를 분산시켜야 한다. 특히 테이블 스페이스와 데이터 파일을 설계할 때 가장 밀접한 관련이ㅏ 있는 부분으로 보통 핵심 업무 예를 들면, 인사관리에서는 급여 사항, 보험업무에서는 사건이나 고객과 같이 주로 발생하는 업무 프로세스가 특정 데이터 파일을 이용할 때 I/O집중 현상이 많이 발생된다.

물리적인 읽기와 쓰기 작업이 특정 디스크에 집중적으로 발생한다면 해당 데이터 파일을 다른 디스크로 옮기거나 테이블 스페이스를 여유가 있는 데이터 파일에 생성해야 한다.

시스템 테이블 스페이스를 별도로 생성한다.

임시 테이블 스페이스 테이블과 인덱스의 테이블 스페이스와 분리하여 생성한다.

롤백 세그먼트와 온라인 리두로그 파일은 분리하여 생성한다.

온라인 리두로그 파일과 아카이브 리두로그 파일은 분리하여 생성한다,

롤백 세그먼트와 테이블과 인덱스 테이블 스페이스와 분리하여 생성한다,

테이블의 테이블 스페이스와 인덱스의 테이블 스페이스는 분리하여 생성한다.

리두로그 파일은 지속적으로 I/O가 발생하므로 I/O가 가장 적은 디스크에 배치한다.

테이;블과 인덱스 테이블 스페이스 중에 I/O가 많은 테이블 스페이스는 별도의 디스크에 배치한다.

로우 마이그레이션과 로우 체이닝

로우 마이그레이션은 데이터 블록에서 수정이 발생하면 수정되ㅏㄴ 데이터를 해당 데이터블록에서 저장하지 못하고, 다른 블록의 빈 공 간을 찾아 데이터를 저장하는 방식이 로우 마이그레이션이다. 로우 체이닝은 로우 길이가 너무 길어서 데이터 블록 하나에 데이터가 모두 저장되지 않고 두 개 이상의 블록에 걸쳐 하나의 로우가 저장되어 있는 형태이다.

로우 마이그레이션과 로우 체이닝 모두 데이터를 입력하거나 수정할 때 추가적으로 작업이 발생되어야 하므로 성능 저하의 원인이 된다. 로우 마이그레이션 현상은 보통 PCTFREE가 낮게 설정되어 있는 경우에 나타나고 로우 체이닝 현상은 데이터베이스를 생성할 때 지정하는 DB\_BLOCK\_SIZE를 너무 작게 지정하거나 데이터 타입이 LOB타입과 같이 대용량 타입일 경우에 발생한다.

로우 마이그레이션 확인

로우 마이그레이션은 테이블을 분석하면 DBA\_TABLES에서 마이그레이션이 발생한 건수를 조회할 수가 있다.

만약 테이블의 로우에 비해 마이그레이션된 건수가 많다면 테이블을 생성할 때 PCTFREE를 증가시켜 재생성하여 테이터를 이관하도록 한다.

로우 체이닝 확인

로우 체이닝을 확인하기 위해서는 오라클에서 제공하는 UTLCHAIN.SQL을 실행해야 한다. 이 문장을 실행하면 CHAINDEC\_ROWS라는 테이블이 생성되어, 분석 작업을 수행할 경우 체이닝에 관련된 데이터가 테이블에 저장된다.

10.4 SQL문장 튜닝

SQL 문장 작성할 때 꼭 알고 있어야 할 사항

대부분 업무에서 사용하는 SQL문장은 단순히 결과만을 조화하는 것이 아니라 일정한 성능을 보장해야 한다. 아무리 훌륭하게 테이블을 생성하여도 SQL문장을 잘못 작성하면 절대 성능을 보장할 수 없다. 그러므로 데이터베이스에서 데이터를 처리할 때는 단순히 결과만 나타내며 하지말고, 반드시 일정한 수준의 성능이 보장되는가를 항상 염두에 두어야 한다.

\* 접근 경로에 대한 우선순위를 알고 있어야 한다.

\* 옵티마아이저 모드가 비용기반 모드인지 규칙기반 모드인지 알고 있어야 한다.

\* 숫자형식은 숫자 형식의 타입으로 문자 형식은 문자형식의 타입으로 비교한다.

\* where절 안에 비교하는 조건에 인덱스를 이용해야 할 경우 인덱스 컬럼을 기술한다,

\* 여러 개의 컬럼이 인덱스 하나로 지정된 경우 where 절에 모두 기술한다.

\* 여러 개의 컬럼이 인덱스 하나로 지정된 경우 앞쪽에 있는 컬럼을 모두 지정하여 인덱스 범위를 조회할 수 있게 한다.

\* 인덱스를 이용하고자 하는 컬럼에 대해 변형을 하지 않는다.

\* 인덱스 칼럼에 NULL 값을 사용하지 않는다.

\* 부정형으로 지정할 경우에는 인덱스를 사용하지 않는다. 가능하면 부정형을 사용하지 않는다,

\* OR를 사용하면 인덱스를 사용하지 않는다.

\* 불필요하게 DUAL테이블을 이용하지 않는다.

\* where 조건절에 걸린 인덱스가 데이터를 20%이상 반환하리라 예상되면 인덱스 스캔을 피한다,.

\* 불필요하게 DBMS에서 제공하는 함수(sum, sign, decode,NVL,…)를 사용하지 않는다.

\* 인덱스가 걸려있는 컬럼에 대해 LIKE형식으로 비교하는 경우에는 반드시 뒤쪽에 비교자%가 위치해야 한다, 앞쪽에 위치할 경우에는 인덱스를 이용하지 않는다.

\* 힌트를 적절하게 사용한다.

접근 경로에 대한 우선순위를 알고 있어야 한다.

오라클에서 제시한 접근 경로에 대한 우선순위 표와 예이다. 우선순위 표에는 ROWID나 클러스터, 인덱스 등의 우선순위가 높게 나타나고, 테이블 전체를 읽는 전체 스캔 방식이 맨 마지막에 나타난다.

1순위 : ROWID에 의한 단일 로우

2순위 : 클러스터 조인에 의한 단일 로우

3순위 : 유일하건아 primary key를 가진 해시 클러스터 키에 의한 단일 로우

4순위 : 유일하거나 primary key에 의한 단일 로우

5순위 : 클러스터 조인

6순위 : 해시 클러스터 키

7순위 : 인덱스 클러스터 키

8순위 : 복합 컬럼 인덱스

9순위 : 단일 컬럼 인덱스

10순위 : 인덱스가 구성된 컬럼에서 제한된 범위 검색

11순위 : 인덱스가 구성된 컬럼에서 무제한 범위 검색

12순위 : 정렬 – 병합 조인

13순위 : 인덱스가 구성된 열에서 MAX또는 MIN

14순위 : 인덱스가 구성된 열에서 ORDER BY

15순위 : FULL테이블 스캔

옵티마이저 모드가 비용기반 모든인지 규칙기반 모드인지 알고 있어야 한다.

오라클의 옵티마이저는 SQL문장을 분석하여 인덱스나 조인 방법 등에 대해 종합적인 정보를 이용하여 가장 적절하다고 판단한 수행 방법을 결정한다, 오라클의 옵티마이저는 비용기반 모드(COST BASED)와 규칙기반 모드(RULE BASED MODE)를 제공한다.

규칙기반 모드는 ANALYSYS를 이용하여 생성한 통계 정보를 활용하지 않고, 오라클에서 정한 접근 경로에 대한 우선순위 근거하여 실행 계획을 선택한다.

비용기반 모드에서는 ANALYSE를 이용하여 생성한 통계 정보를 활용하여 가장 빠른 접근 경로가 무엇인지를 결정하여 실행계획을 수립한다.

어떤 모드를 사용하는가에 따라 SQL 문장의 실행 경로나 인덱스 사용 규칙이 달라진다. 그러므로 데이터베이스의 운영 모드가 무엇인지 정확하게 모른다면 SQL문장을 효율적으로 작성할 수 없게 된다.

만약 REGIST\_DEPT 10의 분포도가 25% 정도이고, 2002년 1월 1이부터 1월 3일까지 ORDER\_DATE 분포도가 3% 정도인 인덱스가 모두 지정되어 있다고 가정하자.

규칙기반 모드에서 위 문장은 전체 테이블을 스캔하는 것은 15순위이고, REGIST\_DEPT가 ‘=’에 의해 비교 되었으므로 9순위 그리고 ORDER\_DATE가 제한된 범위를 검색하는 조건이므로 10순위가 된다. 그래서 ORDER\_DATE의 데이터의 분포가 40% 정도로 매우 좋지 않음에도 불구하고 인덱스를 이용하게 되어 성능이 저하된다.

비용기반 모드에서는 데이터에 대한 통계 정보를 이용하므로 분포도가 좋지 않은 인덱스는 이용하지 않게 된다. 그래서 이와 같은 예에서도 분포도가 25%인 REGIST\_DATE를 이용하여 실행 계획을 수행한다.

또 다른 예로, 가져오려는 데이터의 분포도가 20%를 훨씬 넘을 때 전체 테이블을 스캔하는 편이 더 빠름에도 불구하고, 규칙기반 모드에서는 항상 인덱스를 이용하려 하루 것이다.

규칙기반 모드에서 발견되는 이러한 문제에도 불구하고 규칙기반 모드를 실제 데이터베이스를 운영하는 업무에서 사용하는 이유는 실행 계획의 명시로 인해 SQL문장의 튜닝이 용이하기 때문이다.

반대로 비용기반 모드에서도 옵티마이저가 모든 SQL에 대한 실행 계획을 완벽하게 수렵하지는 못하므로 원하는 성능이 나오지 않는 경우가 종종 있다. 그러므로 비용기반 모드일지라도 접근 경로에 대한 순위와 데이터의 분포도를 고려하여 SQL문장을 튜닝해야 한다.

숫자 형식은 숫자형식의 타입으로 문자형식은 문자 형식의 타임으로 비교한다. Where 절에 기술된 비교 형식을 일치시켜주지 않으면 비록 변환에 문제가 없는 형식이라 할지라도 DBMS내부적으로 한 번 더 작업아 발생하게 되므로 성능에 영향을 준다.

예를 들어, 숫자형식인 DEPTNO=’10’과 같이 작성하지 말고, DEPTNO = 10과 같이 작성한다, 또한 문자 형식인 DEPTNM = 영업부로 작성하지 말고 DEPTNM=’영업부’와 같이 작성해야 한다.

Where 절에 비교하는 조건이 인덱스를 이용해야 할 경우 인덱스 컬럼을 기술해 준다.

아주 당연한 이야기 같지만 실제 프로젝트에서는 적절한 인덱스를 비교할 수 있음에도 불구하고 where 조건절에 기술하지 않는 경우가 많다, 데이터를 가져오고 싶은 테이블의 인덱스가 무엇인지 항상 확인해야 한다.

조회한 컬럼에 대해서 인덱스를 사용하고 있는지 반드시 확인하면서 SQL문장을 작성한다,

여러 개의 컬럼이 인덱스 하나로 지정된 경우 where 절에 모두 기술한다.

여러 개의 컬럼이 하나의 인덱스로 지정된 복합 컬럼 인덱스인 경우 where절에 모든 컬럼을 기술한다.

예를 들어, 인덱스가 BRANCH\_NO + ORDER\_DATE + ORDER\_CD로 지정된 테이블이 존재한다고 가정하고 SQL문장을 작성해 보자.

Select order\_no from order

Where order\_date = ‘2020-07-01’ and order\_cd = ‘01’;

만약 이와 같이 문장을 작성하였다면 인덱스를 이용하지 않게 되고, 그렇게 이 문장은 수행 속도가 저하될 것이다. 위에 보이는 where 조건절에서 인덱스를 사용하도록 where branch\_no = ‘1111’ and order\_date = ‘2020-07-01’ and order\_cd = ‘01’을 추가한다.

실제 프로젝트에서는 from 절에 테이블이 5개 이상 지정되는 경우도 있고, 테이블 하나에 여러 개의 인덱스가 걸려있는 경우도 많으므로 인덱스 컬럼을 모두 기술하지 않아 성능을 저하시키는 경우가 많이 있다. 앞에서 기술한 인덱스를 조회하는 SQL문장을 실행하여 해당 테이블에 어떤 인덱스가 존재하는지를 반드시 파악하고, 적절한 인덱스를 이용할 수 있도록 SQL문장을 작성한다,

여러 개의 컬럼이 인덱스 하나로 지정된 경우 앞쪽에 있는 컬럼을 모두 지정하여 인덱스 범위를 조회할 수 있게 한다.

복합 컬럼 인덱스를 설정할 때 자주 사용하는 컬럼을 맨 앞쪽에 놓고 이용 순서대로 설계해야 하는 내용은 인덱스 설계 단계에서 설명하였다. 비록 뒤에 오는 컬럼에 대한 데이터가 입력되지 않더라도 앞쪽에 위치한 컬럼에 대해서는 정확하게 기술하도록 한다. 이렇게 하면 인덱스에 대한 조회(RANGESCAN)가 가능하게 되어 성능 향상에 도움이 된다. 단, 앞쪽에 위치한 컬럼에 값이 누락되면 인덱스는 쓰이지 않는다.

인덱스를 이용하고자 하는 컬럼에 대해 변형을 하지 않는다.

인덱스를 지정한 컬럼이 모두 where 절에서 비교되어 있음에도 불구하고, 인덱스를 이용하지 못하게 인덱스가 변경된 경우가 있다. 인덱스 변형이 발생하여 사용하지 못하게 되는 경우를 알아보자.

인덱스를 사용하지 못하게 되는 경우를 알게 되면 규칙기반 모드에서 SQL문장의 실행계획을 수립할 때 인덱스 사용여부를 이용하여 실행 계획을 외부에서 조정할 수 있다.

비교되는 컬럼에 변형을 발생시켜 외부 변수나 상수와 비교하게 되면 인덱스를 이용할 수 없게 된다.

* 문자열 변형이 일어나는 경우

아주 간단한 사실이지만 컬럼 타입에 따라 혹은 정확한 상수를 지정했는지에 따라 인덱스를 이용할 수도 있고 전체 스캔이 발생할 수도 있다. 보통 숫자처럼 보이는 컬럼도 가끔 문자열을 지정하여 사용하기도 한다. 이때 인덱스를 지정 했는지만 비교하는 상수나 바인드 변수에 정확한 유형을 지정하지 않았다면 인덱스를 사용할 수 없다.

Substr을 이용하여 컬럼의 값을 잘라버리면 인덱스 영역에 존재하는 컬럼과 비교하지 않고, 바로 테이블에 존재하는 데이터 영역에서 비교하므로 전체 스캔이 발생한다. LIKE로 수정하여 비교하면 동일한 SQL문장을 구사할 수 있으면서 인덱스 영역에서 인덱스를 이용하여 비교하게 된다.

* 숫자에 변형이 일어나는 경우
* 날짜의 변형이 일어나는 경우
* 인덱스 컬럼에 NULL/NOT NULL 값을 사용하지 않는다

부정형으로 지정할 경우에는 인덱스를 사용하지 않는다

조건절에서 두 개의 컬럼을 비교할 때 하나의 컬럼에는 인덱스가 있고 한개의 컬럼에는 인덱스가 없는 경우 인덱스를 사용하지 않는다.

불필요하게 DUAL테이블을 이용하지 않는다.

Where 조건절에 걸린 인덱스가 데이터의 15% 이상을 반환하리라 예상되면 인덱스 스캔을 피하라.

불필요하게 DBMS에서 제공하는 함수를 사용하지 않는다.