제5절 데이터베이스 튜닝

1. 데이터베이스 튜닝의 개념

데이터베이스 튜닝(Database Tuning)이란 데이터베이스 어플리케이션, 데이터베이스 자체, 운영체제 등의 조정을 통하여 데이터베이스 시스템의 성능을 향상시키는 작업을 말한다. 특히 데이터베이스 튜닝을 통해 데이터베이스 어플리케이션이 높은 작업 처리량과 짧은 응답시간을 갖도록 하는 것이 중요하다. 이러한 데이터베이스 튜닝이 필요한 이유는 데이터베이스 시스템 운영 중에 다양한 어플리케이션의 도입과 데이터의 대용량화로 인해 데이터베이스 시스템의 성능이 저하될 수 있기 때문이다.

갈수록 복잡화, 대량화되고 있는 시스템을 그대로 유지하면서 데이터베이스 시스템을 최적화하는 방안으로 데이터베이스 튜닝은 투자한 비용에 비해서 탁월한 효과를 거둘 수 있다는 점에서도 크게 주목받고 있다. 특히, 데이터베이스 시스템 설계 초기 단계에서는 정확하고 구체적인 작업부하에 대한 정보를 얻기 어렵기 때문에 데이터베이스를 구축하고 실제로 운용하면서 여러 가지 성능 관련 요소들을 분석함으로써 성능 개선을 위한 데이터베이스 튜닝을 수행하여야 한다.

2. 데이터베이스 튜닝의 목적

데이터베이스 튜닝의 목적은 데이터베이스 설계 및 활용에 존재하는 문제점을 파악하여 분석하고, 이렇게 분석된 문제점들을 해결함으로써 데이터베이스의 활용 성능을 최적화하는데 있다. 이와 같이 데이터베이스 튜닝을 수행함으로써 데이터베이스를 활용하는 시스템을 안정시키고, 또한 사용자의 만족과 관리자의 관리 능력을 향상시킬 수 있다.

데이터베이스 튜닝 과정에서 추구하는 목적을 달성하기 위한 방법들을 구체적으로 기술하면 다음과 같다.

첫째, 업무 환경과 시스템 환경에 적합하게 데이터베이스 파라미터를 설정한다. 예를 들어, 정렬 작업이 많이 발생하는 업무에서는 정렬 작업을 위한 메모리 공간을 충분히 확보해야 하지만, 읽어오는 순서대로 데이터를 조회하는 업무에서는 정렬 공간이 많이 필요하지 않기 때문에 메모리 공간을 낭비하지 않도록 적절하게 환경 설정을 해야 한다.

둘째, 데이터베이스에 접근하는 SQL 문장의 수행시 디스크 블록에 대한 접근 횟수가 가능한 최소가 되도록 한다. 특히, 디스크를 적절하게 분산시켜 디스크 입출력이 집중되는 현상을 막으면 경합이 발생되지 않기 때문에 바람직한 성능을 보장할 수 있다.

셋째, 디스크 블록에서 한 번 읽어온 데이터는 가능하면 메모리 영역에 보관하여 다시 그 데이터가 필요할 때 메모리로부터 빨리 가져오게 한다. 이것은 디스크로부터 메모리로 로드하여 데이터를 가져오는 경우에 비해 메모리에서 데이터를 직접 읽어오는 경우가 디스크 입출력을 최소화할 수 있어서 속도가 훨씬 빠르기 때문이다.

넷째, 모든 SQL 문장은 공유가 가능하도록 대소문자 등 명명 표준을 준수하여 작성한다. 처음 실행된 SQL 문장은 구문검사, 실행권한 체크와 같은 내부 파싱 절차를 거쳐 DBMS 안에서 최적의 실행 계획을 수립한다. 그러므로, 동일한 SQL 문장에 대해서는 가능하면 파싱 작업을 다시 수행하지 않도록 하면 성능을 향상시킬 수 있다.

다섯째, 락(lock) 발생을 최소화한다. 데이터

베이스에 다중 사용자가 안정적으로 동시에 접근할 수 있도록 하는 락 기능이 전체적인 데이터베이스 성능을 저하시키는 경우가 있다. 이런 경우에는 락이 최소한으로 발생하도록 트랜잭션의 분산과 같은 방법을 활용해야 한다.

여섯째, 배치(batch) 작업과 백업 작업 수행이 빠른 시간 안에 완료될 수 있도록 한다. 일반적으로 배치 작업에 걸리는 시간이 길기 때문에 작업을 수행하는 동안 메모리나 CPU 등 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.

3. 데이터베이스 튜닝 방법론

데이터베이스 튜닝에서는 튜닝 방법론에 따라 기존 시스템에 존재하는 문제점을 정확히 분석하고, 그 분석을 통해 튜닝의 목적과 대상을 정하고, 마지막으로 각 단계별로 하나씩 해결해 나가야 한다. 그리고, 실제와 유사한 환경에서 최소의 반복 테스트를 수행함으로써 튜닝 이전보다 나쁜 결과를 가져오지 않도록 해야 한다.

가. 비율 기반 분석

비율 기반 분석 방법론은 주로 1990년대 초반에 데이터베이스 컨설턴트들이 사용했던 방법론으로 적은 용량의 데이터베이스에서는 효율적인 데이터베이스 튜닝 방법이었다. 예를 들면, 데이터베이스 성능을 분석하기 위해서 데이터베이스 시스템 관리자(DBA)가 주로 사용하는 방법은 데이터베이스 버퍼, 캐시 히트 비율로 이 값이 90% 이상이면 데이터베이스 성능에 문제가 없다는 식의 접근 방식이다.

그러나, 최근 낮은 하드웨어 비용으로 인해 많은 데이터베이스 시스템의 데이터베이스 버퍼 캐시 히트율이 99%가 넘는 경우가 많이 있지만, 이러한 경우에도 데이터베이스 성능을 저하시키는 병목 현상이 존재한다. 따라서 복잡한 데이터베이스 시스템 환경에서 비율 기반 분석 방법으로는 효율적으로 데이터베이스 성능을 분석하는 데 한계가 있다.

나. 대기 이벤트 기반 분석

주로 1990년대 중반부터 사용된 대기 이벤트 기반 분석 방법론은 비율 기반 분석 방법보다는 좀 더 발전된 형태로써 데이터베이스에 연결된 동시 세션들이 데이터베이스의 어떤 자원에 대해 대기하고 있는가를 진단하고 이러한 대기 시간을 줄이는 접근 방법이다.

그러나, 응답 시간에서 대기 시간만을 고려하고 서비스 처리 시간과 운영체제 자원에 대한 대기 시간은 고려되지 않으므로 서비스 처리 시간의 비중도가 큰 데이터베이스 시스템을 분석하는 데는 한계가 있다.

다. 응답 시간 분석

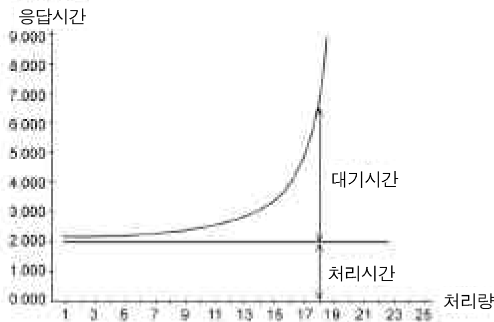
앞에서 언급된 두 가지 분석 방법론에서는 데이터베이스 성능을 효율적으로 분석하고 튜닝하는데 있어 다소 한계가 있다. 현재까지는 응답 시간 분석 방법론이 가장 효율적인 데이터베이스 튜닝 방법이라고 할 수 있다. 이 방법론은 주로 2000년대부터 사용하기 시작한 분석 방법으로써 이 방법론에서는 응답 시간이 처리 시간과 대기 시간으로 나뉘어 진다.

응답 시간 분석 방법론에서는 처리 시간에 대한 비중이 큰 데이터베이스 시스템인 경우 대기 시간보다는 처리 시간을 줄이는 것이 낫다. 반대로 대기 시간에 대한 비중이 큰 데이터베이스 시스템인 경우에는 처리 시간보다는 대기 시간을 줄이는 것이 전체 시스템의 응답 시간을 개선하는 데 좋다. 이 방법론에서 가장 중요한 요소는 분석 대상 시스템의 응답 시간 추이 분석이다. 어느 순간에 시스템이 최대한 사용됐는지를 파악한 후 시스템, 데이터베이스, 프로그램, SQL 단위로 처리 시간과 대기 시간의 비중을 분석한다.

(1) 처리 시간 분석

분석 대상의 응답 시간이 처리 시간에 집중되었을 경우에는 어느 프로그램, 어느 SQL에서 처리 시간을 집중적으로 사용했는가를 분석하고, 단위 SQL 튜닝을 통해 수행 시간의 향상과 수행 비중을 계산해 어느 정도의 처리 시간을 줄일 수 있는가를 분석한다. 물론 단위 SQL 튜닝 과정에서는 집중 사용된 시간대의 수행 속도와 튜닝 단계의 수행 속도, 사용자 수, 변수 사용 유형 등 가변 요소가 많기 때문에 정확한 수치를 측정하는 것은 불가능하지만, 단순 SQL 뿐 아니라 복합 요소를 분석해 처리 시간의 요소를 예측하고 향후 사용자나 데이터의 증가량에 관계없이 안정적으로 운영할 수 있도록 해야 한다.

처리 시간은 시스템의 한정적인 자원을 사용하기 때문에 급격히 증가할 수는 없다. 따라서 일정 수준에 도달하면 처리 시간은 정적인 값을 유지하게 되고, 대기 시간은 급증하는 형태를 보여주기 때문에 처리 시간이 집중되는 시스템은 문제 발생 초기라는 분석이 가능하다. 따라서 이런 단계에서의 최적화가 선행되어야 최소의 비용으로 최적의 시스템을 유지할 수 있다. [그림 4-9-13]은 응답시간과 처리량과의 관계를 보여준다.



[그림 4-9-13] 응답시간과 처리량의 관계

(2) 대기 시간 분석

대기 시간 분석 시에 가장 먼저 접근해야 하는 항목은 자원 대기 현상이 발생한 위치가 시스템인지 데이터베이스인지를 분석하는 것이다. 시스템이라면 CPU, 메모리, 입출력 중에 어느 요소가 가장 심각한 대기 시간을 보여 주는가를 분석해야 한다. 이런 분석을 수행하는 것은 대기 시간의 정확한 원인을 분석하고 대기 시간을 최소화시켜서 전체 응답 시간을 개선하는데 목적이 있다.

CPU 대기가 많은 경우는 처리 시간 분석과의 연계 분석이 가능한데, 이는 처리 시간을 줄이는 분석 방법과 일치한다. 따라서 처리 시간을 줄일 수 있는 만큼 줄이고, 그 이후에는 CPU를 늘려 성능 향상을 고려해야 한다. 메모리 대기가 많다면 시스템 환경에서는 커널 메모리와 버퍼 캐시의 크기, 페이징과 스와핑의 비율 등에 대한 분석이 필요하다. 즉, 사용하고 있는 메모리를 줄여서 최적화하는 방안과 최후의 대처 방안인 가용 메모리를 늘리는 방안을 고려해야 할 것이다.

가장 많은 튜닝 시간이 소요되는 항목인 입출력 대기 시간이 많다면 정확한 원인 분석이 필수적이며 분석 순서도 준수해야 한다. 가장 먼저 분석해야 할 사항은 어플리케이션의 사용 유형과 사용 시간대 분석이다. 우선 입출력 대기 시간이 급증하는 시점의 원인이 사용자의 증가 때문인지, 수행되는 어플리케이션의 수가 많아서인지, 아니면 배치와 온라인이 공존하는 시간대인지를 분석해야 한다. 그 다음은 어플리케이션이 배치나 온라인 또는 온라인을 통한 배치인지를 분석한 후 해당 시간대에 이들 어플리케이션이 수행되어야만 하는지에 대한 정당성을 검증해야 한다.

이런 분석 작업이 완료되면 반드시 어플리케이션 분석을 수행해야 한다. 이를 통해 실행 계획을 검증하고 적절한 실행 계획을 보장하는가에 대한 점검을 통하여 실행 계획이 최적화될 수 있도록 하여야 한다.

4. 데이터베이스 튜닝의 단계

일반적으로 데이터베이스 튜닝은 튜닝 방법론 절차에 따라 현행 정보 시스템에 존재하는 문제점을 분석하고, 분석된 문제점을 단계적으로 해

결하는 것이 필요하다. 여기에서 일반적인 데이터베이스 튜닝 단계에 대해 기술한다.

가. 비즈니스 규칙 튜닝

최적의 성능을 위해서는 비즈니스 규칙에 대한 적절한 조정은 필수적인데, 이것은 전체 시스템의 설계와 구현에 대한 고수준 분석에 기반해야 한다. 실제로 DBA들이 직면한 성능 문제들은 실제로 시스템의 설계와 구현에 대한 정확하지 않은 분석이나 부적절한 비즈니스 규칙에 의해 유발된다. 특히, 비즈니스 규칙은 많은 동시 사용자들이 존재하는 환경에 근거한 현실적인 기대치를 고려해야 한다.

나. 데이터 설계 튜닝

데이터 설계 단계에서는 구현된 응용 어플리케이션에서 필요한 데이터가 무엇인지 정확히 파악해야 한다. 그리고, 어떤 관계들이 중요하며 어떠한 속성들이 존재해야 하는지도 잘 파악해야 한다. 이러한 기반 위에서 최적의 성능을 위해 정보의 구조화 작업이 필요하다. 또한, 성능 개선을 위한 정규화나 비정규화 단계의 작업이 필요하며, 데이터의 경합을 최대한 피할 수 있는 작업도 필요하다.

다. 어플리케이션 설계 튜닝

어플리케이션은 구현 목적에 맞게 효과적으로 각각의 프로세스를 구성해야 하는데, 같은 시스템을 액세스하는 어플리케이션이라도 구현 목적에 따라 상이한 설계가 가능하다. 그러므로, 각 프로세스의 성능을 조사하여 부하가 발생하는 시점의 해당 어플리케이션 수행에 필요한 시간 및 필요로 하는 데이터들을 조사한다. 그리고, 튜닝을 필요로 하는 대상 어플리케이션을 선정하고, 선정된 어플리케이션을 위주로 집중적인 튜닝을 수행한다.

라. 데이터베이스의 논리적 구조 튜닝

데이터베이스의 논리적 구조는 데이터베이스 스키마에 의해 정의된다. 이 단계에서의 핵심은 작업부하로 예상되는 질의와 갱신을 고려하여 스키마를 작성함으로써 데이터베이스 시스템 성능을 향상시키는 데이터베이스 논리적 구조를 생성하는 것이다.

마. 데이터베이스 접근 방식 튜닝

최상의 시스템 성능을 위하여 SQL의 장점과 어플리케이션의 작업처리를 최대화시키기 위해 보유하고 있는 DBMS의 기능을 충분히 활용하고 있는지를 검증한다.

바. 액세스 경로 튜닝

효과적인 데이터 액세스를 위하여 트리 인덱스, 비트맵 인덱스, 클러스터, 해시 클러스터 등의 사용을 고려한다. 또한, 어플리케이션 테스트 단계를 거치면서 원하는 응답속도를 얻기 위해서 인덱스의 추가 및 삭제는 물론 설계의 개선도 고려되어야 한다.

사. 메모리 운영 튜닝

성능 개선에 긍정적인 효과를 가져올 수 있도록 메모리 자원을 효율적으로 할당하여 캐시의 성능을 개선하고 SQL 문장의 파싱 작업을 감소시켜야 한다.

아. 물리적 구조 및 입출력 튜닝

디스크 입출력 성능은 많은 어플리케이션들의 성능을 저하시키는 주된 원인이 되므로 디스크간에 데이터를 분산하여 입출력 경합을 감소시키고, 또한 최소의 액세스 비용을 위한 효율적인 데이터 블록 운영도 고려하여 튜닝한다.

자. 자원의 경합에 대한 튜닝

다수의 사용자가 동일한 자원에 대해 동시에 액세스를 원하는 운영 환경에서 자원에 대한 경합의 유발은 불가피하지만 블록, 공유 풀, 락 등의 경합 형태를 감소시키기 위한 노력을 계속하여야 한다.

차. H/W 시스템에 특화된 부분의 튜닝

데이터베이스는 사용되는 H/W 시스템의 종류와 특성에 따라서 성능의 차이가 생긴다. 그러므로, 사용 중인 H/W 시스템 플랫폼에 특화된 부분에 대한 튜닝 방안을 강구해야 한다.

5. 데이터베이스 튜닝의 구분

데이터베이스 튜닝은 크게 시스템 영역과 어플리케이션 영역으로 나눌 수 있다. 시스템 영역은 주로 데이터의 처리 속도에 영향을 미치게 되며 디스크, OS, DBMS 등이 이 영역에 속하게 된다. 어플리케이션 영역은 일의 양에 영향을 미치게 되는데, 일의 양을 결정하는 요소로는 업무 데이터 량, 데이터 설계, 어플리케이션 설계, 로직, 스키마 설계, SQL 등이 있다. 특히, 어플리케이션은 데이터베이스 성능 저하 원인의 약 60%를 차지할 만큼 비중이 매우 높아 튜닝이 반드시 필요하므로 관리자들은 어플리케이션에 대한 정확한 원인 분석에 따라 성능 향상의 최적화 방법을 이용하여 효과적이고 지속적인 어플리케이션 튜닝을 실시해야 한다.

가. 병목 위치 튜닝

대부분의 시스템의 성능은 병목이라는 몇 가지 요소의 성능에 의해 주로 제한된다. 병목을 예를 들어 설명하면 어떤 프로그램에서 코드 내의 작은 루프에 80%의 시간을 소비하고 코드의 나머지 부분에 20%의 시간을 소비한다고 할 때 작은 루프를 병목이라고 할 수 있다. 이때 병목인 루프의 속도를 향상시키면 최상의 경우 전체적으로 80%의 성능 향상을 얻을 수 있게 된다. 이처럼 시스템을 튜닝할 때는 병목의 위치를 알아내고 그 요소의 성능을 향상시킴으로서 병목을 제거해야 한다.

데이터베이스 시스템은 큐잉 시스템(Queueing System)으로 모델링될 수 있는데, 여기서 트랜잭션은 입력에서부터 서버 프로세스까지의 기동, 실행 중 디스크 읽기, CPU 사이클 및 동시성 제어를 위한 락 등 데이터베이스 시스템에게 다양한 서비스를 요청한다. 그리고, 이들 각 서비스는 관련된 큐를 가지며 어떤 트랜잭션들은 코드의 실행보다는 큐에서 대기하는 데 대부분의 시간을 보낼 수도 있다(특히 디스크 입출력 큐에서). 만약 서비스의 요청이 일정한 간격으로 이루어지고 다음 요청이 도달하는 시간 전에 끝난다면, 각 요청은 이용 가능한 자원을 찾아서 기다리지 않고 즉시 실행될 수 있지만 이것은 거의 불가능하다.

자원의 이용률에 따라 큐의 길이는 지수적으로 증가하기 때문에 큐의 길이를 되도록 짧게 하여 낮은 이용률을 유지해야 한다. 엄지손가락 규칙(rule of thumb)에 따르면, 일반적으로 70% 가량의 이용률은 좋은 것으로 간주되나 90% 이상의 이용률은 심한 지연을 초래할 수 있으므로 과다한 것으로 간주되고 있다.

나. 매개변수 튜닝

데이터베이스 관리자는 3단계로 데이터베이스 시스템을 튜닝할 수 있다. 첫 번째 최하위 단계가 하드웨어 단계이다. 이 단계에서의 시스템 튜닝으로는 디스크 입출력이 병목일 때는 디스크 추가나 RAID 시스템 사용, 디스크 버퍼 크기가 병목일 때는 메모리 추가, CPU 사용이 병목일 때는 보다 빠른 처리기로 대체 등이 있다. 두 번째 단계는 버퍼 크기와 체크포인트 간격과 같은 데이터베이스 시스템 매개변수로 구성되는데, 잘 설계된 데이터베이스 시스템은 사용자나 데이터베이스 관리자에게 부담을 주지 않고 가능한 많은 튜닝을 자동으로 수행할 수 있다. 세 번째 단계가 스키마와 트랜잭션을 포함한 상위 단계이다. 성능을 향상시키기 위해 사용자는 스키마 설계, 생성된 인덱스 및 트랜잭션을 튜닝할 수 있다.

이러한 세 단계의 튜닝은 서로에게 영향을 미칠 수 있으므로 시스템을 튜닝할 때는 이들을 함께 고려하여야 한다.

다. 하드웨어 튜닝

(1) 디스크 관련 튜닝

잘 설계된 트랜잭션 처리 시스템이라 하더라도 보통 각 트랜잭션마다 적어도 서너 개의 입출력 연산을 가지고 있다. 그리고, 현재 사용되고 있는 대부분의 디스크들은 약 10밀리세컨드의 액세스 시간과 초당 20MB의 전송 시간을 갖고 있으며, 1KB씩 약 100번의 무작위 액세스 입출력이 가능하다. 따라서 각 트랜잭션이 2번의 입출력 연산을 요청한다고 할 때 하나의 디스크는 겨우 초당 50개의 트랜잭션만을 처리할 수 있으므로, 결국 초당 더 많은 트랜잭션을 처리하기 위한 방법은 단지 디스크의 수를 늘이는 것뿐이다. 즉, 어떤 시스템이 초당 n개의 트랜잭션을 처리해야 한다면 데이터는 n/50개의 디스크에 나누어 저장되어 있어야한다.

(2) 메모리 관련 튜닝

제한된 요소가 디스크의 용량이 아니라 임의의 데이터에 접근하는 속도일 경우 트랜잭션당 입출력 연산의 수는 메모리 상에 더 많은 데이터를 저장함으로써 줄일 수 있다. 만약 모든 데이터가 메모리 상에 있다면 쓰기 연산을 제외한 디스크 입출력은 없게 되는 것이다. 따라서 자주 사용되는 데이터를 메모리에 존재하게 하면 디스크 입출력 횟수를 줄일 수 있다. 그러나, 거의 사용되지 않는 데이터를 메모리에 저장하게 되면 메모리 낭비이기 때문에 디스크나 메모리를 늘리는데 사용 가능한 주어진 비용으로 초당 트랜잭션의 수를 최대로 하는 방법을 알아내는 것이 중요하다.

먼저 디스크에 한번 접근하는데 드는 비용을 (디스크 드라이브당 가격) / (하나의 디스크에 초당 액세스 횟수)으로 구한다. 그 다음에는 한 페이지를 저장하는데 드는 메모리 비용을 (메모리의 MB당 페이지의 가격) / (메모리의 MB당 페이지의 수)으로 구한다. 그리고, 특정 페이지가 초당 n번 액세스된다면 n번은 메모리에 있게 되므로, n * (디스크에 한번 접근하는데 드는 비용) = (한 페이지를 저장하는데 드는 메모리 비용)의 식을 만족하는 n값을 구한다. 구해진 n값을 기준으로 해서 어떤 페이지가 n보다 더 자주 액세스된다면 메모리를 늘리는 것이 더 유리하다.

현재의 디스크 기술과 메모리와 디스크의 가격으로 n값을 구해보면 무작위로 액세스되는 페이지일 때 초당 1/300번(혹은 5분에 한번)이 나오게 된다. 이러한 결과는 이른바 5분 규칙이라고 알려진 엄지손가락 규칙(rule of thumb)에 의해서도 얻어낼 수 있다. 5분 규칙이란 만약 어떤 페이지가 5분 이내에 한번 또는 그보다 더 자주 사용된다면 그 페이지는 반드시 메모리에 저장되어 있어야 한다는 것이다. 즉, 평균적으로 5분에 한번 이상 액세스되는 모든 페이지들이 저장될 수 있을 만큼의 충분한 메모리가 있어야 한다. 반면 자주 액세스되지 않는 데이터에 대해서는 그 데이터가 요청되는 입출력율을 지원할 수 있는 충분한 디스크가 필요하다.

엄지손가락 규칙은 단지 입출력 횟수로만 계산하기 때문에 응답 시간과 같은 요소들은 고려되지 않는다. 따라서 어떤 어플리케이션에서 디스크 액세스 시간과 같거나 그보다 더 빠른 응답 시간을 제공하기 위해서는 자주 사용되지 않는 데이터들까지도 메모리에 저장되어 있어야 한다.

(3) RAID 레벨 관련 튜닝

RAID 1 또는 RAID 5의 사용에 따른 튜닝은 데이터가 얼마나 자주 갱신되는 가와 관련된다. RAID 5는 RAID 1보다 무작위 쓰기에 있어서 훨씬 더 느리게 수행되는데, 그것은 RAID 5가 한번의 무작위 쓰기 연산을 위해 2번의 읽기와 2번의 쓰기를 수행하기 때문이다. 따라서 RAID 1은 쓰기 연산이 많은 로그 파일에 적합하며, RAID 5는 저장되는 데이터의 양은 매우 많지만 데이터의 변환이 적고 읽기 연산이 많은 데이터 파일이나 인덱스 파일에 적합하다.

라. 개념 스키마 튜닝

데이터베이스 설계 과정에서, 현재 사용하고

있는 릴레이션 스키마가 주어진 작업부하에 대한 성능 목표를 달성하지 못할 경우가 발생할 수 있다. 이러한 경우에는 개념 스키마를 재설계할 필요가 있다. 개념 스키마는 주요한 작업부하로 예상되는 질의와 갱신을 고려하여 설계되는 것이 중요하다. 다음은 개념 스키마 튜닝시 고려해야 할 몇 가지 선택 사항들이다.

첫째는 스키마 설계에 있어서 BCNF와 3NF의 선택이다. 스키마 설계시 BCNF 설계 대신 3NF 설계를 받아들여야 할 경우가 있다. 즉, 어떤 릴레이션이 BCNF로 정규화가 불가능할 경우가 있으며, BCNF로 정규화가 가능하더라도 3NF로 작업부하를 해결하는 것이 더 효율적이라면 3NF를 선택하는 것이 바람직하다. 또한, 이미 BCNF 상태인 릴레이션도 더 분해할 필요가 있을 경우도 있다.

둘째는 정규화와 비정규화를 선택하는 상황이다. 정규화 작업은 논리적인 모델을 검증하는 마지막 단계로 물리적인 설계에 선행되어 작업이 이루어지며 성능을 고려하여 수행되는 비정규화와 반대되는 개념이다. 정규화 작업은 유지보수를 효율적이고 쉽게 해주는 반면에 중복된 데이터를 가지는 릴레이션에 대한 과도한 조인을 발생시켜 성능을 떨어뜨리는 요인이 되기도 한다. 따라서 구현될 시스템의 성능을 고려하여 비정규화 하는 작업을 수행할 필요가 있다. 비정규화는 데이터를 액세스하는 속도를 빠르게 하기 위해서 수행되나 릴레이션 상에서 불필요한 데이터를 가질 가능성이 높기 때문에 저장 공간의 낭비를 야기할 수 있다.

셋째는 정규화에 있어서 수직 분할과 수평 분할 방법의 선택이다. 수직 분할은 릴레이션의 컬럼들을 나누는 것으로 일반적으로 사용되는 분할이다. 수평 분할은 릴레이션에서 하나 이상의 릴레이션들로 나누는 것으로 동등한 스키마를 갖는 두 개 이상의 릴레이션을 작성하는 것이다.

넷째는 뷰(view)의 사용이다. 특별히 기존의 데이터베이스 스키마를 튜닝하는 것이라면 스키마의 변경 없이 사용자에게 뷰를 제공하여 변경사항을 숨기는 방법을 사용할 수 있다.

마. 인덱스 튜닝

초기 데이터베이스는 중요한 작업부하로 예상되는 질의와 갱신들을 고려하여 데이터 검색 속도를 향상시키기 위해 인덱스를 사용한다. 그러나, 데이터베이스를 사용하면서 중요한 작업부하로 예상되었던 질의와 갱신들은 데이터 처리 내용의 변화 등의 이유로 더 이상 빈번하게 사용되지 않을 수 있다. 오히려 새로운 질의와 갱신이 중요한 작업부하로 확인되는 경우가 발생한다. 이렇게 새로이 확인된 작업부하에 대해서 기존의 불필요한 인덱스를 제거하고 새로이 필요한 인덱스를 추가하는 작업이 필요하다.

새로운 인덱스를 추가할 경우 인덱스 종류를 선택하는 것도 중요하다. 예를 들어, B-트리와 해시 인덱스를 비교해보면 범위 질의를 많이 사용할 경우에는 B-트리 인덱스가 해시 인덱스를 사용하는 것보다 효율적이고, 정확한 값에 의한 검색인 경우는 해시 인덱스가 더 효율적이다. 그리고, 튜닝 가능한 파라미터로 클러스터 인덱스가 있다. 클러스터는 초기 인덱스 설정에 없을 경우가 있다. 그러나, 일반적으로 대부분의 질의와 갱신에 있어서 성능 향상을 위해 인덱스를 클러스터하여 사용해야 한다. 따라서 상황에 따라 인덱스에 대해 클러스터를 해주어야 하는 경우가 생긴다.

위에서 언급한 바와 같이 데이터베이스의 데이터는 지속적으로 증가하고 변형되기 때문에 각 상황에 적합하게 인덱스를 사용할 수 있도록 지속적으로 모니터링하여 인덱스를 조정하여야 한다.

바. 실체 뷰 튜닝

실체 뷰란 데이터베이스 질의어 처리 속도를 증가시키기 위해 자주 사용될 만한 질의 결과를 미리 저장해 놓는 것을 의미한다. 실체 뷰는 데이터가 분산 환경에서 흩어져 있는 경우에도 유용한데, 자주 사용되는 질의나 직전의 질의 등의 결과를 클라이언트에 실체 뷰 형태로 복사한 후, 원격(remote) 지점 대신 클라이언트 지점에서

데이터를 읽어 성능 향상을 꾀하게 된다. 사용자는 실체 뷰의 존재에 대한 인식을 하지 않은 채 일반적인 질의를 하게 되고, 질의어 처리기에서 이러한 질의를 실체 뷰를 사용한 동일한 의미의 질의로 변환하여 수행하게 된다. 그러나, 실체 뷰는 저장 및 유지를 위해 공간 오버헤드와 시간 오버헤드가 있다. 또한, 트랜잭션이 갱신되면 실체 뷰는 그 후에 갱신되기 때문에 시간적인 차이로 불일치성 문제가 발생할 수 있다.

실체 뷰는 적당한 질의들을 선택한 후 실험을 통해서 작성된다. 최적화기를 사용하면 쉽게 실체 뷰를 만들 수 있으나 최적화기를 사용해도 실제 실험을 통하지 않으면 정확한 비용 측정이 어렵기 때문에 작업부하를 고려하여 성능 향상을 시킬 수 있는 실체 뷰를 찾기 위해서는 수동적인 방법도 사용된다.

사. 질의 튜닝

과거에는 데이터베이스 시스템의 최적화기가 좋지 않아 질의를 어떻게 작성하느냐에 따라 실행에 큰 영향을 주었으며 그에 따라 성능에도 크게 영향을 미쳤다. 오늘날은 최적화기가 발전되어 나쁘게 작성된 질의조차도 내부에서 좋은 질의로 변환되어 효율적으로 실행될 수 있다. 그러나, 최적화기가 할 수 있는 범위가 한정되어 있다. 현재 대부분의 시스템은 정확한 질의 실행 계획을 찾을 수 있는 기법을 제공하므로 이것을 사용해 질의를 튜닝할 수 있다.

질의 수행이 예상보다 느리다면 어떤 문제가 있는지 질의를 조심스럽게 시험해야 한다. 인덱스 튜닝과 함께 질의를 새로 작성해 본다면 문제가 무엇인지 알아낼 수 있다. 또한, 시스템의 최적화기가 최상의 정책을 찾을 수 없다면, 사용자는 질의에 최적화기를 위한 정보를 추가하여 적절한 정책을 선택하도록 할 수 있다. 예를 들어, 사용자가 특정 인덱스를 사용하도록 하거나 조인 명령, 조인 방법을 강제적으로 사용하도록 하는 경우가 있다.

인덱스는 데이터의 빠른 검색을 위한 것으로 대부분의 경우 인덱스를 사용하는 것이 성능향상을 가져오지만 원리를 정확하게 알지 못하면 인덱스를 사용하는 것이 오히려 성능을 저해할 수 있으므로 상황에 따라 적절하게 인덱스를 사용해야 한다. 인덱스의 무분별한 증가는 릴레이션의 데이터 갱신시 인덱스 갱신이 늘어나는 문제를 가져온다. 또한, 최적화기는 데이터의 빠른 검색에도 불구하고 작업부하 때문에 인덱스를 사용하지 않는 경우도 있다.

아. 트랜잭션 튜닝

트랜잭션 성능을 향상시키는 데에는 처리 단위를 집합으로 묶어서 다루는 방법과 트랜잭션 사이에 락 다툼을 줄이는 방법을 생각해 볼 수 있다.

Embedded SQL에서 질의가 매개 변수에 대해 다른 값을 가지고 빈번히 실행된다면 집합 지향적인 질의로 호출들을 결합하여 한번만 실행되도록 할 수 있다. 예를 들어, 클라이언트-서버 시스템에서는 Embedded SQL 호출들을 결합하여 SQL 질의의 통신비용을 줄일 수 있다. 또한, SQL 질의의 통신비용과 번역 비용을 줄이기 위해 저장된 프로시저를 사용하는 기법이 있다. 여기서 질의는 서버에 미리 컴파일 된 프로시저의 형태로 저장되고, 클라이언트는 질의 전체를 통신하지 않고 이렇게 저장된 프로시저를 호출할 수 있다.

서로 다른 유형의 트랜잭션이 동시에 같은 자원을 접근하면 락 다툼으로 인해 성능 저하를 가져올 수 있다. 예를 들어, 은행에서는 고객의 계좌에 대한 수없이 많은 갱신 트랜잭션들이 실행된다. 이 때, 지점의 통계를 산출하는 질의가 동시에 실행된다고 하자. 이 통계 산출 질의가 한 릴레이션에 검색을 실행하면, 이 질의가 실행되는 동안 해당 릴레이션에 대한 모든 갱신이 불가능해져 시스템의 성능에 악영향을 미칠 수 있다. 또한, 수행 시간이 긴 장기 갱신 트랜잭션이 시스템 락과 시스템 고장으로부터의 회복하는 데 걸리는 시간으로 성능 문제를 야기할 수 있다. 이러한 문제를 해결하는 방법은 하나의 트랜잭션을 가급적 작은 갱신 트랜잭션들의 집합

으로 나누는 것이다. 그리하여, 작은 트랜잭션은 자원을 락(lock)하는 시간을 짧게 하여 락 다툼에서의 성능 저하를 줄이고, 장기 갱신 트랜잭션에서 시스템 고장시 고장 이후의 나머지 트랜잭션을 수행하면 되기 때문에 회복 성능도 향상시켜 줄 수 있게 된다.