Tech Notes (vol.2)

병렬 처리와 데이터 웨어하우스

서론

데이터 웨어하우스 환경은 기본적으로 매우 많은 자료를 축적하고 관리하기 위한 것입니다. 실제로 데이터 웨어하우스 환경에서 많은 자료를 제대로 관리할 수만 있다면 데이터 웨어하우스 디자인이나 사용의 다른 측면은 쉬워집니다. 마찬가지로 데이터 웨어하우스 환경에서 많은 자료를 제대로 관리하지 못하면 자료 관리 자체를 실패할 것이기 때문에 다른 사항은 문제삼을 수 조차 없을 것입니다. 사실 자료가 매우 많을 경우 자료 관리 자체가 데이터 웨어하우스를 성공적으로 수립하고 사용하는 첫 걸음이자 결정적인 요인이 됩니다.

데이터 웨어하우스 환경에서 많은 자료를 관리하는 구성 방법과 기술에는 다음과 같이 여러가지가 있습니다.

- 데이터를 여러 저장 매체에 저장하기
- 새로운 세부 데이터가 생겼을 때 데이터 요약하기
- 데이터 가공의 관점에서 데이터 관계(data relationship) 저장하기
- 적당한 데이터의 참조와 암호화
- 서로 다른 분할 영역의 독립적 관리를 위한 데이터 분할
- 데이터 웨어하우스에 대한 적당한 밀도(granularity)의 레벨과 요약 레벨 선택하기

주제

이런 디자인이나 아키텍처 기술 및 접근 방법은 모두 유효하고 어떤 하드웨어 환경에서나 사용할 수 있지만 데이터 웨어하우스 환경에서 많은 자료를 관리하는 데는 또 하나의 방법이 있습니다. 바로 데이터를 병렬로 관리할 수 있는 기술을 선택하는 방법입니다. 병렬 기술은 데이터베이스 시스템 기술로도 알려져 있습니다.

이 글은 병렬 환경에서의 많은 양의 데이터 웨어하우스 자료 관리에 관한 것입니다. 여기서는 데이터 웨어하우스만을 다루고자 합니다. 개발자들이 연산 처리를 하기 위해 데이터베이스 시스템이나 병렬 기술을 사

용하려 할 수도 있습니다. 또한 디자이너들은 같은 시스템에서 같은 데이터를 가지고 연산 트랜잭션 처리와 데이터 웨어하우스 처리를 동시에 시도하려는 환경에서 데이터베이스 시스템 기술을 사용하려 할 수도 있습니다. 여기서는 이러한 환경들을 다루고자 하는 것은 아닙니다. 단지 기본 저장 방법과 접근 방법에 병렬 처리 기술을 도입하고자 하는 데이터 웨어하우스에 대하여 전개하고자 합니다.

병렬기술의 장점

병렬 기술은 서로 다른 시스템들이 긴밀하게 연관되지만 작업은 독립적으로 하는 기술입니다. 각 시스템은 자료의 모음을 독립적으로 관리합니다. 데이터 웨어하우스에서는 직각 방향으로 자료가 전개됩니다.

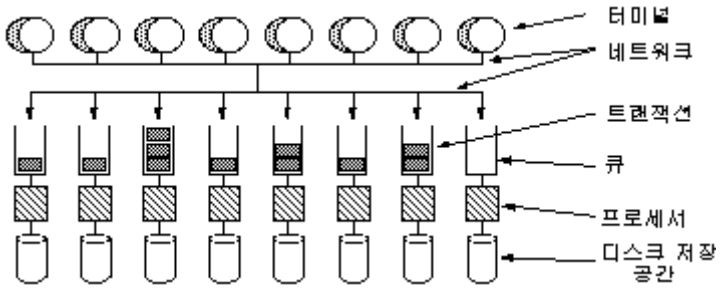


그림 1: 많은 자료를 관리하는 전형적인 병렬 기술 방법

그림 1은 같이 병행하여 작업하지만 자료를 독립적으로 관리하는 프로세서의 기본 구성을 보여줍니다.

그림 1에는 트랜잭션(또는 요구), 트랜잭션이 들어가는 큐, 프로세서를 연결하는 네트워크, 프로세서, 프로세서가 제어하는 자료 등 다섯 가지 기본 구성 요소가 있습니다.

시스템에 요구가 들어오면 요구를 전할 프로세서가 요구를 보낼 큐를 결정합니다. 트랜잭션이 큰 경우에는 트랜잭션을 자료와 처리에 대한 일련의 요구로 나누어 해당 프로세서로 보내게 됩니다. 요구가 큐로 들어가면 프로세서가 요구를 실행하기 시작합니다. 요구에 대한 작업이 완료되자마자 그 결과가 요구자에게 보내집니다.

한 프로세서가 프로세서에 속한 자료에 작업하는 요구를 처리하는 동안 다른 프로세서는 그와 상관없이 독립적으로 자신에게 전해진 요구를 처리할 수 있습니다. 이러한 독립적인 처리야말로 데이터 웨어하우스 아키텍처의 큰 매력입니다. 독립적인 처리는 기술적으로 많은 자료 관리를 가능하게 해 주기 때문입니다. 많은 자료를 관리하려면 프로세서를 여러 개 준비하기만 하면 됩니다. 다시 말해, 더 많은 데이터 웨어하우스 자

료를 관리하려면 데이터 웨어하우스 아키텍처는 그림 2와 같이 긴밀하게 네트워크화된 구성에 프로세서를 추가하기만 하면 됩니다.

병렬 환경이 아닌 시스템에서는 이미 커질대로 커진 환경에 많은 자료를 추가하게 되면 기본적인 운영 체제 수준에서 어려운 점이 너무 많습니다. 비병렬 환경에서는 일종의 자료의 한계점이 있어 그 값을 넘으면 작업이 비효율적이 됩니다. 데이터 한계점에 도달하면 다른 기술을 사용하는 것 외에는 방법이 없습니다. 그러나, 자료를 다른 기술로 처리하거나 전송하면 혼란이 일어나고 추가 비용이 크며 복잡하므로 될 수 있으면 피하는 것이 좋습니다. 반면, 병렬 처리를 하면 기술 자체가 거의 무제한으로 확장되므로 데이터 웨어하우스를 다른 기술로 변환하지 않아도 됩니다.

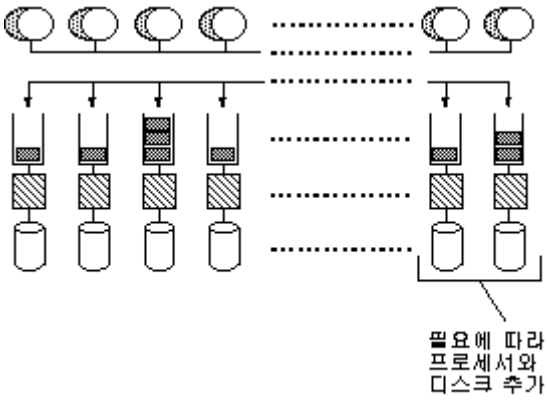


그림 2: 병렬 환경에서는 프로세서를 추가하는 것으로 쉽게 용량을 확장시킬 수 있습니다.

병렬 환경에서는 처리가 독립적이므로 자료 접근 속도는 자료가 전개되는 프로세서 수에 비례합니다. 병렬 구성에 자료가 고르게 최적화된 상태로 전개된 m개의 독립적인 프로세서가 있을 때, 이 프로세서를 하나의 큰(비병렬) 프로세서로 n 단위의 수만큼 취해 하나의 요구를 처리한다면, 이때 서비스를 실행하는 병렬 구성에 필요한 경과 시간은 n/m입니다.

그림 3은 이 차이를 보여 줍니다.

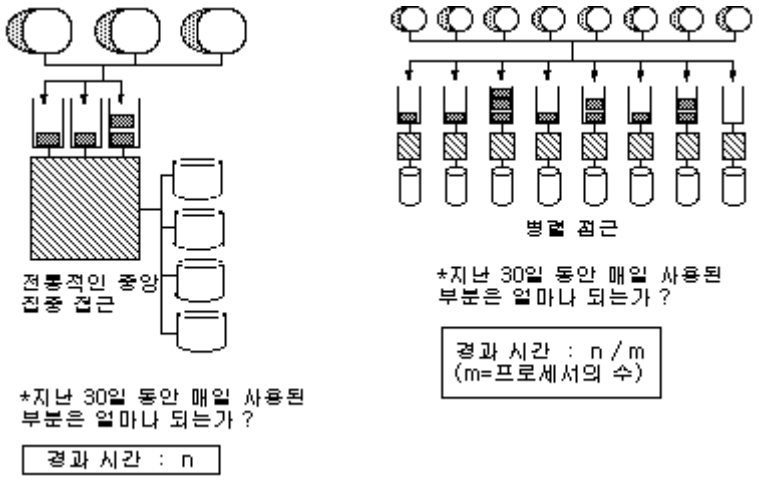


그림 3: 처리 시간을 줄이는 병렬 접근

병렬 환경으로 이동했을 때 절약되는 경과 시간은 다음의 식으로 나타낼 수 있습니다.

n - (n/m) = 경과 시간 차이

시스템(병렬이든 비병렬이든)에서 이루어진 작업은 I/O의 관점에서는 모두 같습니다. 실제 차이는 이루어진 전체 작업량에 있는 것이 아니라 그 작업을 하는 데 걸린 시간입니다.

또 병렬 환경에서는 추가되는 프로세서 수가 늘어남에 따라 줄어드는 처리 시간이 정비례하는 것은 아닙니다. 즉, 병렬 환경에서 프로세서 수가 하나에서 둘로 늘어날 때는 처리 시간이 눈에 띄게 짧아집니다. 그러나 프로세서 수가 늘어남에 따라 처리 시간이 줄어드는 폭은 작아질 것입니다. 가령, 프로세서 수가 20개에서 21개로 늘어날 때는 처리 시간의 변화가 거의 드러나지 않을 수도 있습니다.

데이터 웨어하우스 기술에서는 대개 자료 양에 따라 병렬 접근(parallel approach)과 표준 중앙 집중 접근(standard centralized approach) 중에서 선택하게 됩니다. 중소 규모의 자료로 구성되는 데이터 웨어하우스에는 중앙 집중 접근이 경제적이면서도 기술적인 면에서도 바람직합니다. 어느 기준을 지나면(데이터 웨어하우스에 아주 많은 자료가 포함되어), 병렬 접근이 바람직할 수 있습니다. 선택은 항상 기술적인 면과 경제적 사항을 고려하여야 할 것입니다.

물리적 조직

여러 가지 방법으로 병렬 처리 기술을 열거할 수 있습니다만 가장 일반적인 방법들만을 논의하고자 합니다. 물론 모든 병렬 기술 구성 요소의 각각의 구성과 그 방법들은 나름대로 장단점이 있으며, 따라서 모든 가능성을 다 언급할 수는 없을 것입니다.

그림 4는 병렬 환경의 각 구성 요소들이 진행하는 전형적인 내부 작동을 나타냅니다.

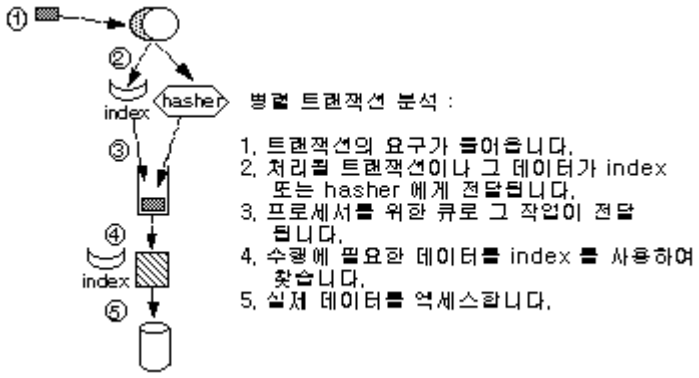


그림 4: 병렬 트랜잭션 분석

요구에는 해당 프로세서에 보내지는 단일 요구(singular request)와, 일련의 특정 요구들로 나뉘어 따로따로 각 프로세서로 보내지는 일반 요구(general request)가 있습니다.

요구가 들어가는 큐도 모든 프로세서에 접근하는 하나의 큰 큐로 이루어지는 경우와 여러 프로세서 각각에게 유일한 일련의 큐로 구성되는 경우가 있습니다. 후자의 경우, 요구를 실행하기 전에 특정 프로세서에 할당해야 합니다.

자료는 해싱 알고리즘이나 색인, 또는 양쪽 모두를 사용하여 프로세서에 지정합니다. 해싱 알고리즘을 사용하면 자료는 레코드의 기본 키에 따라 여러 개의 프로세서로 나뉩니다. 자료가 색인으로 프로세서에 지정되면 대개(꼭 그런 것은 아니지만) 그룹으로 한 프로세서에 할당됩니다.

일단 자료가 할당된 프로세서에 도달하면, 자료는 디스크 공간에 저장되고 색인은 또 다른 자료 할당을 추적하기 위해서 계속 사용됩니다. 자료는 행(레코드)과 칼럼(필드)으로 된 테이블의 물리적 블록에 저장됩니다.

위에서 설명한 대로 병렬 환경 구성 요소 배열 방법은 다양하며 각각 장단점이 있습니다.

Hot Spot

병렬 기술의 매력은 자료가 점차로 많아져서 많은 자료를 관리할 때에도 혼란을 주지 않고 점증적인 방법으로 리소스 처리를 추가할 수 있다는 점에서 돋보입니다. 언뜻 보면 병렬 접근이 데이터 웨어하우스에서 많은 자료를 관리하는 궁극적 해결 방법인 것처럼 보입니다.

하지만 많은 자료를 관리할 때 병렬 접근을 사용하는 것이 전통적인 단일 프로세서 접근보다 오히려 성능을 떨어뜨리는 경우가 있습니다.

많은 자료를 관리할 때 병렬 접근의 성능과 효율성은 자료가 각 프로세서에 얼마나 고르게 분포될 수 있는가에 달려 있습니다. 데이터가 고르고 적당하게 전개되어있으면 병렬 관리가 아주 효과적입니다. 그러나 병렬 프로세서에 자료 분포가 한 쪽으로 치우치게 되어 그 결과로 각 프로세서에 걸리는 작업 부하가 불균형을 이루게 되면 이른바 'Hot Spot'이 생기고 병렬 환경의 효율성은 현저히 떨어집니다.

그림 5는 Hot Spot을 보여 줍니다.

하나의 문제점은 데이터 웨어하우스의 자료 접근 패턴을 예측할 수 없다는 것입니다. 접근 비율과 엑세스되는 특정 레코드 모두 매우 다양합니다. 따라서 DSS 처리에서는 Hot Spot이 흔히 일어나게 됩니다.

물론 Hot Spot은 개선할 수 있습니다. Hot Spot을 개선하려면 여러 프로세서에 자료를 재분산해야 합니다. 그림 6은 Hot Spot을 개선한 내용입니다.

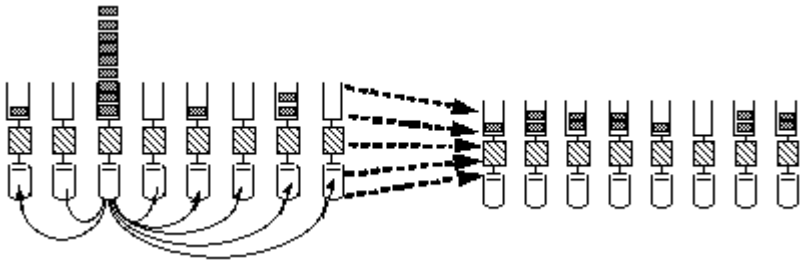


그림 6: 자료 분산을 변경하여 병렬 환경에서 작업 부하가 고르게 걸리도록 합니다.

데이터 웨어하우스 환경에서 Hot Spot을 개선할 때의 문제는 자료 사용을 미리 알아야 한다는 점입니다. 이전의 자료 사용 패턴이 앞으로도 똑같이 사용되리라는 보장은 없기 때문입니다. 따라서 병렬 환경에서 Hot Spot을 식별하고 개선하는 일은 어려운 작업입니다.

운용시스템과 DSS

병렬 환경은 데이터 웨어하우스, 즉 DSS 처리와 운용 시스템(operational)에 사용될 수 있지만 동시에 양쪽 모두를 사용할 수는 없습니다. 두 환경이 병렬 환경에서 혼합될 수 없는 몇 가지 이유가 있습니다. 그림 7.1, 그림 7.2, 그림 7.3 , 그림 7.4는 이런 차이점들을 보여 줍니다.

두 환경의 첫번째 차이점은 실행되는 트랜잭션 형식입니다. 운용 시스템 환경에서는 처리를 단일 프로세서에 연결하는 작은 트랜잭션을 많이 실행합니다. 그러나 데이터 웨어하우스 환경의 트랜잭션 작업은 매우 다릅니다. 즉, 데이터 웨어하우스 환경에서는 병렬 환경 전체에 분포된 자료에서 작업해야 하는 큰 트랜잭션을 몇 개만 실행하게 됩니다.

두 번째 차이점은 자료 내부 구조에 있습니다. 데이터 웨어하우스 환경에는 거대하고 연속적이면서 정적인 자료에 대한 최적화된 구조로서의 자료가 포함됩니다. 반면 운용 시스템 환경은 언제라도 갱신할 수 있는

제한된 양의 자료를 빠르게 접근하기 위한 구조로 구성되어 있습니다.

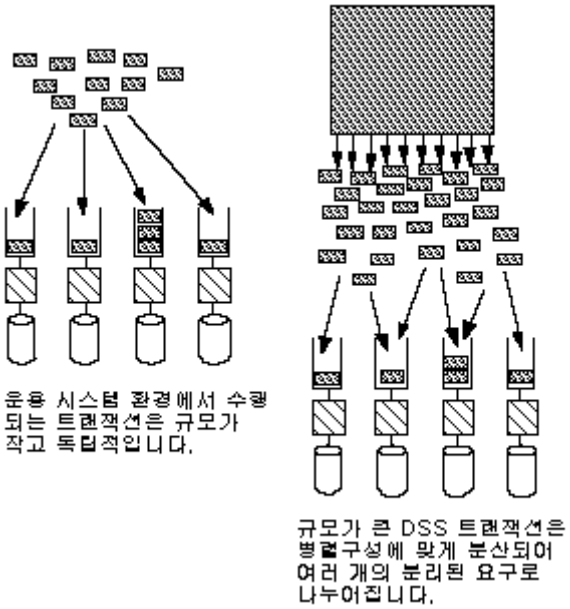


그림 7.1: 운용 시스템 환경과 데이터 웨어하우스 환경에서 실행되는 트랜잭션은 매우 다르므로 관리 방법도 다릅니다.

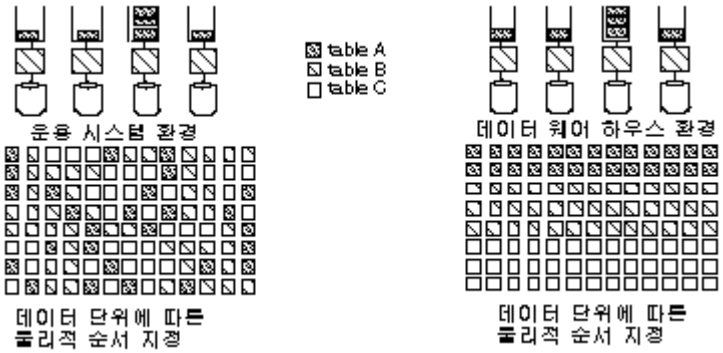


그림 7.2: 두 환경에서 자료의 물리적 구조가 매우 다릅니다.

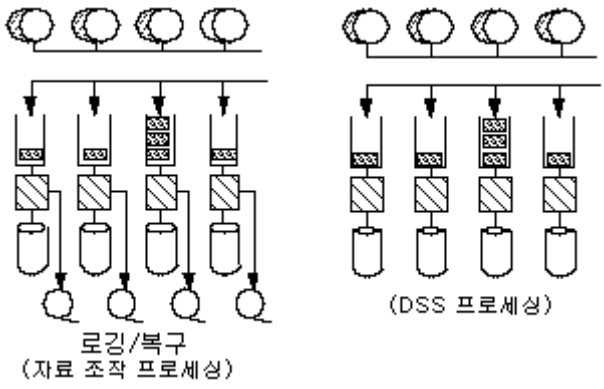


그림 7.3: 운용 시스템 환경의 기본적인 부분은 로깅과 복구인데 이는 DSS 환경에는 없는 기능입니다 .

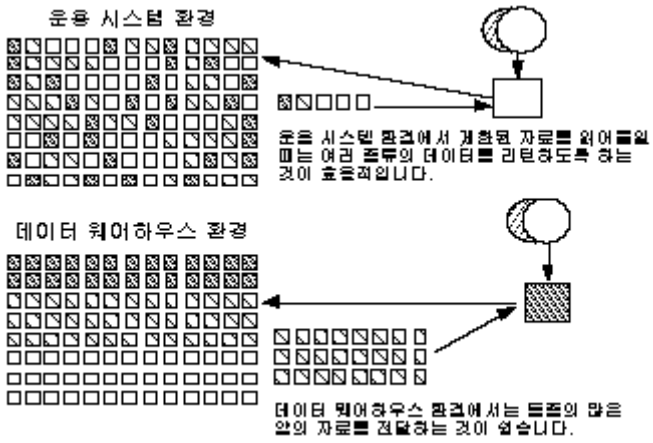


그림 7.4: 자료의 물리적 구조는 처리 성능에 막대한 영향을 미칩니다.

또한, 운용 시스템 환경에서는 일반적으로 자료를 형식에 따라 묶으므로 트랜잭션이 자료를 찾을 때 다른 위치를 검색해 볼 필요가 없습니다. 반면 데이터 웨어하우스 자료는 한꺼번에 저장되는 것이 일반적입니다.

운용 시스템 환경과 데이터 웨어하우스 환경의 또 하나의 중요한 차이는 자료 로깅에 있습니다. 자료를 조작하는 과정에는 갱신이 포함되므로 오버헤드가 요구됩니다. 로깅은 갱신 작업과 함께 수반되는 일종의 오버헤드입니다. 하지만 데이터 웨어하우스 자료는 갱신되지 않으므로 로그가 필요 없습니다. 따라서 운영 체제의 기본 특성이 다릅니다.

이런 점에서, 자료를 병렬 관리할 때에도 운용 시스템 환경과 데이터 웨어하우스 환경이 혼합될 수 없는 이유입니다.

데이터 웨어하우스의 레벨

병렬 기술로 구성한 데이터 웨어하우스의 가장 큰 잇점은 자료를 빠르게 또한 무작위로 접근할 수 있다는 것입니다. 이렇게 관리되는 자료의 세부 사항은 상대적으로 검색하기 쉽습니다. 그러나 자료의 병렬 관리는 그 비용이 만만치 않습니다.

대부분의 데이터 웨어하우스는 현재 세부 레벨만을 병렬 환경에 두고 다른 단계에는 별도의 기술을 사용합

니다. 그림 8은 이 내용을 자세히 보여 줍니다.

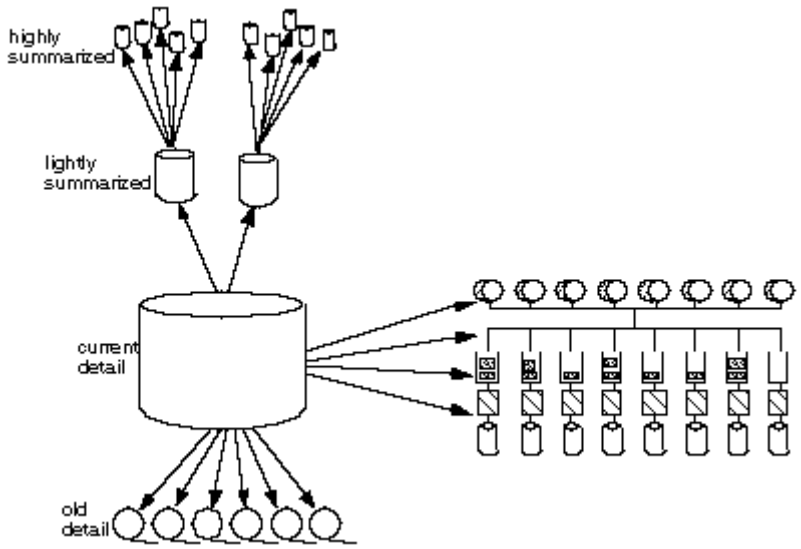


그림 8: 병렬 기술로는 데이터 웨어하우스의 현재 세부 사항 부분만 관리합니다.

여러 가지의 경제적 기술적인 이유로 현재 세부 자료는 병렬 기술로, 다른 자료는 별도의 기술로 관리하는 것이 좋습니다.

병렬 데이터 웨어하우스 환경의 메타 데이터(Meta Data)

메타 데이터(Meta Data)는 데이터 웨어하우스 환경에서 가장 중요한 구성 요소 중 하나입니다. 데이터 웨어하우스가 병렬 기술을 사용한다고 해도 메타 데이터의 역할에는 변함이 없습니다.

일반적으로 데이터 웨어하우스로 저장된 메타 데이터에는 다음의 내용이 포함됩니다.

- 자료 내용
- 자료 구조
- 운용 시스템 환경으로부터의 자료의 맵핑 정보
- 추출, 버전 정보 등의 기록

병렬 데이터 웨어하우스 환경에서의 물리적 디자인

병렬 환경에서의 데이터 웨어하우스 디자인은 디자인 초기 단계에서는 비병렬 환경과 마찬가지입니다. 자료 모델 정의, 주요 주제 영역 정의, 레코드 체계 정의, 미래에의 예측 등은 두 환경에 모두 같습니다. 병렬 환

경 디자인에서 달라지는 것은 물리적 디자인을 만들 때입니다. 다른 프로세서들에게 자료를 분산하는 일이 디자인에서는 매우 중요한 문제입니다.

우선, 프로세서를 몇 개나 둘 것인가 하는 점이고, 다음 문제는 각 프로세서에 자료를 어떻게 분포시킬 것인가 하는 점입니다. 이 결정에 영향을 미치는 요소는 다음과 같습니다.

- 자료 증가 패턴
- 초기 자료량
- 자료 접근 패턴

디자인에서 또 하나의 중요한 문제는 무엇으로 자료의 기본 키를 삼을 것인가 하는 것입니다. 기본 키는 프로세서에서 처음 자료를 분산하는 구분자 역할을 하므로 여러 병렬 프로세서에서의 자료 분포에 영향을 미칩니다. 중요한 것은 키에 직접 관련되지 않은 데이터 집합에 대한 자료의 보조 키 배치입니다. 보조 키는 병렬 프로세서에 무작위로 배치되거나, 또는 기본 키에 직접 관련되는 자료와 같은 물리적 위치에 놓이게 됩니다. 어느 쪽이 좋은 방법인지는 자료의 사용 방법에 따라 결정 지을 수 있습니다. 병렬 환경에서 자료 분할은 중앙 집중 데이터 웨어하우스 환경에서만큼이나 중요합니다. 테이블을 분할하면 색인 작성, 자료 재구조화, 자료 관리 등을 독립적으로 할 수 있습니다.

키 지정으로 자료를 병렬 프로세서에 할당하는 것은 데이터 웨어하우스 환경의 디자인에서 매우 중요한 측면입니다. 자료의 물리적 배치가 자료 접근에 중요한 영향을 미치고, 따라서 자료 병렬 관리의 효율성을 결정하기 때문입니다. 즉, 자료가 병렬 프로세서에 고르게 분포되지 않으면 병렬 처리하는 장점이 사라지므로, 단일 중앙 프로세서로 관리하는 것이 더 좋습니다.

물리적 디자인의 두 번째 문제는 데이터 웨어하우스 환경에서 파생 자료(즉, 요약 테이블)의 식별과 지원입니다. 요약 자료를 저장하는 것은, 자주 사용되는 자료를 계산하여 재저장하는 작업이 필요할 때 계산을 여러 번 하면 오류가 일어날 우려가 있는 경우에 사용할 수 있는 좋은 방법입니다. 이런 상황에서는 병렬 환경에 자료를 요약하고 저장하는 것이 합리적입니다.

병렬 데이터 웨어하우스 환경에서 중요한 물리적 디자인 기술은, 자료가 정규적으로 조인될 것을 예측할 수

있다면 미리 자료를 조인해 놓는 것입니다. 자료를 조인할 확률이 많다면 자료를 동적으로 조인할 때보다는 로드할 때 미리 조인하는 것이 더 효율적입니다.

또 하나의 물리적 디자인 기술은 데이터 웨어하우스에 자료 개개의 연관성을 생성하는 것입니다. 자료의 연관성은 데이터 웨어하우스에서 매우 중요합니다. 하지만 구현은 운용 시스템 환경에서와는 다른 경우가 대부분입니다.

병렬 데이터 웨어하우스 환경에서는 자료에 훨씬 빠르게 접근할 수 있다는 장점 때문에 언제 필요하게 될지 모르는 자료까지도 될 수 있으면 많은 양의 자료를 자세히 저장하기를 원할 수 있습니다. 그러나 데이타 웨어하우스에 자료를 저장하는 비용은 병렬 기술을 쓴다 해도 무시할 수는 없습니다. 따라서 다음 자료 관리 규칙을 지키는 것이 좋습니다.

- DSS 처리에 사용되지 않는 자료는 데이터 웨어하우스에 저장하지 않습니다.
- 자료가 너무 오래되면 '심층 동결(deep freeze)', 벌크 저장을 고려합니다.
- 세부 사항 수준이 너무 조밀하면 자료를 요약합니다.

요약

데이터 웨어하우스는 병렬 접근 기술로 관리해야 합니다. 병렬 접근에서는 여러 프로세서가 자료에 대하여 독립적으로 작업하고 독립적으로 자료를 관리합니다. 이러한 자료 관리의 독립성 때문에 프로세서 역시 독립적으로 추가할 수 있습니다.

병렬 접근과 중앙 집중 접근 중 어느 쪽을 선택할 것인가는 관리할 자료의 양과 자료 접근 빈도에 달려 있습니다.

병렬 접근은 자료 관리의 강력한 대안을 제공하지만 데이터 웨어하우스에서 물리적 디자인은 여전히 중요한 문제입니다.

롤(Role) : Informix Online Dynamic Server 버전 7.10 UDI 의 새로운 보안기능

서론

Informix OnLine Dynamic Server, 버전 7.10.UD1에서는 "롤(Role)"이라고 하는 새로운 기능을 도입하였습니다. 롤은 특정 권한부여나 박탈을 사용자 개별적으로 하는 대신에 어떤 특성에 맞춰 그룹으로 처리할 수 있는 기능입니다. 게다가, 사용자에게는 하나 이상의 롤을 사용할 수 있도록 권한을 부여할 수 있습니다. 사용자가 롤의 권한에 접근할 필요가 있을 때, 사용자 또는 응용프로그램은 현재 접근 수준을 해당 롤로 설정합니다. 그런 다음, 사용자가 롤이 부여된 함수를 실행한 후에는 그 권한이 더 이상 효력을 발생하지 않도록 롤을 해제할 수도 있습니다. 이 글에서는 롤을 사용하여 보안성을 개선하는 몇 가지 사례를 검토하고 롤의 한계에 대하여 논의합니다. 이 글에 사용된 예는 Sun Sparc 하드웨어 플랫폼의 Informix OnLine Dynamic Server, 버전 7.10.UD1에서 데모 데이터베이스 stores7을 사용하여 작성한 것입니다.

롤의 용도를 설명해 주는 간단한 예로 시작하겠습니다. 이 예에서는 영업부에 속한 사용자들의 삽입, 갱신

및 삭제 작업을 제한하려 합니다. 먼저, 테이블 내의 모든 사용자에게서 모든 권한을 박탈합니다. 그런 다음, 각 개인에게 질의, 삽입, 갱신 및 삭제 권한을 부여하는 대신 세 개의 롤을 작성합니다. 이 중 read_ord라는 롤은 질의 전용 권한으로, 다음 롤인 upd_ord는 질의, 갱신 및 삽입 권한으로, 마지막 롤인 del_ord에는 삭제 권한으로 부여할 것입니다. 각 사용자에게는 이 롤을 사용할 수 있는 권한이 부여됩니다. 마지막으로 이 글에서 이러한 롤을 응용프로그램에서 어떻게 사용할 수 있는가를 설명합니다.

참고

여기서 설명하는 예는 stores7 데이터베이스와 이 데이터베이스 내의 orders 테이블을 사용합니다.

롤 생성

롤을 생성하는 것은 CREATE ROLE role_name 문으로 시작합니다. 여기에서 role_name은 8개 이하의 문자로 이루어진 롤의 이름입니다. role_name은 시스템 카탈로그 sysusers 테이블에 저장되기 때문에 시스템의 사용자명이 되어서는 안 됩니다. 롤을 작성하기 위해서는 해당 데이터베이스의 dba 권한을 가지고 있어야 합니다. 다음의 문을 실행함으로써 위 세 개의 롤을 각각 생성할 수 있습니다.

```
create role read_ord;
create role upd_ord;
create role del_ord;
```

롤을 생성하고 나면 시스템 카탈로그 테이블 sysusers를 검색하여 생성된 롤을 확인할 수 있습니다.

```
select * from sysusers where usertype = "G";
```

이 질의의 결과는 다음과 같습니다.

```
username usertype priority password
read_ord G 5
upd_ord  G 5
```

del_ord G 5

sysusers 테이블에서 usertype G는 Informix OnLine Dynamic Server, 버전 7.10.UD1의 새로운 usertype 으로서 롤을 의미합니다.

롤에대한 권한

롤에 권한을 부여하는 것은 사용자에게 권한을 부여하는 것과 같으며, 따라서 같은 구문을 사용할 수 있습니다. 특정 롤에 권한을 부여하도록 하기 위하여 먼저 orders 테이블의 모든 권한을 박탈해야 합니다. 다음의 SQL 문은 orders 테이블에 부여된 모든 권한을 표시합니다.

```
select * from systabauth where tabid in
(select tabid from systables where tabname ="orders");
```

username이 *public*으로 된 권한을 박탈하려면 다음과 같은 SQL문을 사용하십시오.

이 결과는 다음과 같습니다.

```
grantor   grantee   tabid   tabauth
Informix   del_ord   101    s---d---
Informix   read_ord  101    s-------
Informix   upd_ord   101    su-i----
```

위 내용으로 사용자 Informix가 권한을 부여하였다는 것과 권한을 받은 grantee 칼럼은 롤 이름을 나타내고 tabauth 칼럼은 받은 권한을 나타내고 있다는 것을 알 수 있습니다.

롤에 사용자 추가

이제 롤에 적절한 사용자를 추가해야 합니다. 이 영업부에는 bylee, jhkim, ucpark, jhhong, ykkim 등 각각의 5명의 사용자가 있는 걸로 간주합니다. 이 그룹의 모든 사용자는 orders 테이블을 읽을 수 있어야 하지만 orders를 추가하거나 갱신할 수 있는 사람은 bylee와 jkkim뿐이어야 합니다. 또한 bylee는 orders를 삭제할 수도 있어야 합니다. 이러한 작업을 수행하기 위해서는 다음 SQL문을 사용하십시오.

```
grant read_ord to bylee, jhkim, ucpark, jhhong, ykkim ;
grant upd_ord to bylee, jkkim;
grant del_ord to bylee;
```

read_ord	jhhong	n
read_ord	ykkim	n
upd_ord	bylee	n
upd_ord	ykkim	n
del_ord	bylee	n

위 정보는 롤 이름과 그 롤에 접근할 수 있는 사용자 및 "N"(이 롤을 다른 사람에게 부여할 수 없음) 또는 "Y"(이 롤을 다른 사람에게 부여할 수 있음)를 표시합니다.

롤의 사용: set roll문

한 사용자에게 롤을 사용할 수 있는 권한이 부여되었다고 하더라도, 아직 그 사용자는 롤의 권한에 자동으로 접근할 수 없습니다. 사용자 또는 사용자가 실행한 응용프로그램은 먼저 SET ROLE문을 실행해야 합니다. 여기서 주의해야 할 점은 데이터베이스에 연결할 수 있는 connect 권한이 있고 SQL에 대한 지식이 있다면 어떤 사용자라도 SET ROLE 명령을 사용하여 롤을 활성화할 수 있다는 것입니다.

가령 사용자 jhkim이 SET ROLE 문이 실행되기 전에 orders 테이블에서 데이터를 선택하려고 시도한다면 다음과 같은 오류 메시지를 보게 될 것입니다.

사용자가 더 이상 롤이 필요하지 않을 때에는 롤을 NONE 또는 NULL로 설정할 수 있는데 이렇게 하면 롤의 권한이 제거됩니다. 응용프로그램에서도 롤의 권한이 더 이상 요구되지 하지 않을 때 SET ROLE NONE 또는 SET ROLE NULL문을 사용하여 이 권한을 끝내십시오. 다음 구문을 잘 살펴보십시오.

```
set role none;
select * from orders;
#
# 272: No SELECT permission.
```

응용 프로그램에서의 롤

롤은 응용프로그램에 사용하기 위해 고안된 것입니다. 응용프로그램은 롤을 설정하여 작업을 수행한 다음, 롤을 해제할 수 있습니다. 이 경우, 사용자는 응용프로그램이 실행되는 동안에만 권한을 가지게 됩니다. 응용프로그램이 끝나면 사용자의 권한이 박탈됩니다.

응용프로그램에 롤을 사용하기 위해서는 그 응용프로그램을 위하여 롤을 설정하는 문을 준비하고 실행해야 합니다. 다음 문은 Informix 4GL 프로그램 내용 중에서 롤을 read_ord로 설정하는 예를 나타낸 것입니다.

```
prepare role_stmt from "SET ROLE read_ord"
```

… …

SET ROLE문을 실행하는 EXECUTE문이 처리된 후 성공적으로 실행되었는지를 반드시 확인해야 합니다. 그러지 않다면, 사용자는 적절한 권한이 없이 함수를 실행할 것이며, 결국 이것은 또 다른 수많은 SQL 오류를 유발하게 될 것입니다.

Informix OnLine Dynamic Server, 버전 7.10.UD1에는 롤을 처리하는 몇 개의 새로운 오류 메시지가 있습니다. 예를 들어, 사용자에게 롤의 사용 허가가 없는 경우, sqlca.sqlcode는 "19805: No privilege to set to the Role."의 내용을 리턴합니다.

결론

많은 사용자가 한 데이터베이스에 참여할 때, 롤의 보안 기능이 더욱 유용합니다. 롤을 사용함으로써 데이터베이스 관리자(DBA)는 데이터베이스 권한을 효과적으로 제어할 수 있습니다.

사용자가 구현하는 SQL의 품질

서론

이 글은 SQL에도 품질 보증(QA: Quality Assurance)이 필요하다는 전제에서 시작합니다. 또한 SET EXPLAIN의 출력을 해석하는 방법을 비롯하여 Informix 옵티마이저의 역할을 설명하고 SQL 성능 향상을 위한 방법을 제시합니다.

이 글의 주제는 사용된 툴과 상관없이, Informix 데이터베이스가 필요한 모든 SQL 개발에 관한 것입니다. SQL은 Informix 4GL, Informix SQL, Informix ESQL, Informix NewEra와 같은 Informix 툴이나 Informix 데이터베이스를 사용하는 기타 업체의 툴로 개발할 수 있습니다.

먼저, 왜 SQL을 QA해야 하는가에 대해 생각해 보도록 하겠습니다. SQL 개발을 엄격하게 제어하기 위해서는 SQL의 QA를 수행할 것을 일반적으로 권장하고 있습니다. 데이터베이스 크기가 계속 높은 비율로 증가할 때 QA가 특히 중요합니다. 데이터베이스가 커짐에 따라 오류도 많이 눈에 띄고 문제도 많아지기 때문입니

다. 방대한 데이터베이스는, 특히 성능에 관한 한, 조금도 소홀히 할 수 없습니다. 개발 단계에서 충분히 주의를 기울이면 앞으로 발생할 문제를 덜 수 있을 것입니다.

QA는 성능 문제 외에도 다음과 같은 심각한 문제들을 예방해 줍니다.

- Root DBspace를 무심코 큰 임시 테이블로 채우면 다른 Informix 응용프로그램이 실행되지 않을 수 있습니다.
- UNIX 파일 시스템 공간을 큰 정렬 파일로 채우면 UNIX 프로세스에서 오류가 일어날 수 있습니다.
- 논리 로그를 긴 트랜잭션으로 채우면 보관 파일로부터 데이터베이스를 복구해야 하는 경우가 생길 수 있습니다.

이런 문제들을 피하려면 SQL 개발 표준을 강화하는 것이 가장 좋은 방법이며, SQL QA 과정을 개발 과정과 분리시키는 것이 좋습니다. 일반적인 QA 작업에 많은 인원을 배치하는 기관도 많지만 QA 작업을 한 사람이 도맡아 하는 곳도 있습니다.

대개, 데이터베이스 관리자(DBA)가 SQL 문에 대한 QA 내부 작업의 적임자일 것입니다. DBA야말로 데이터베이스 스키마를 가장 잘 알고 있으며, 또한 DB 엔진 내부 작동을 가장 잘 이해하고 있는 사람이기 때문입니다. 또 DBA는 어떤 SQL이 실행되는지 알고 있어야 합니다. 이것이 흔히 데이터베이스 스키마 수정 여부(예를 들어, 색인을 추가해야 하는지)의 지표가 되기 때문입니다. 어느 누가 QA를 하든 가장 중요한 것은 QA가 제대로 이루어져야 한다는 사실입니다. 개발자와 각 QA 책임자가 모두 QA 절차와 해당 기관에 필요한 표준을 잘 이해하고 합의하는 것이 필요합니다.

QA 절차에 대한 질의를 최적화하는 도구로는 SET EXPLAIN ON 명령이 가장 좋습니다. 질의를 최적화할 수도 있지만 데이터베이스 설계 역시 질의 성능에 중요한 역할을 합니다. 따라서, 테이블을 다시 디자인하고 효율적인 색인 방법을 사용하여 문제를 해결할 수 있는 경우가 많습니다. 이 글에서는 SET EXPLAIN ON 옵션을 중심으로 질의를 최적화하는 방법을 설명하고자 합니다.

인포믹스 옵티마이저의 이해

최적화를 제대로 논의하려면 먼저 Informix 옵티마이저를 이해해야 합니다.

옵티마이저란?

옵티마이저는 주어진 질의를 실행하기 전 가장 좋은 최선의 경로를 찾는 Informix 엔진의 한 부분입니다. 데이터에 접근하는 데는 여러 경로가 있지만 시간이 더 많이 걸리는 경로가 있게 마련입니다. 여러 테이블이 하나로 조인되어 있을 때는 더더욱 그렇습니다. 개발자들은 질의를 작성한 후 최선의 경로를 찾는 일은 옵티마이저에 맡깁니다. 엔진이 가장 좋은 방법을 찾아 사용할 것이라고 여기기 때문입니다. 옵티마이저는 대부분 이 작업을 성공적으로 완수하지만, 반면 이 때문에 엔진이 항상 옳은 결정을 내리는 것은 아니라는 것을 잊어 버리기 쉽습니다. SET EXPLAIN ON의 출력은 옵티마이저가 데이터에 접근하는 데 선택한 경로를 나타냅니다. 이 정보로 옵티마이저가 시간이 많이 걸리는 경로를 선택했는지 확인할 수 있습니다. 이러한 경우, 질의를 다시 작성해야 할 필요가 있을 것입니다. 그러나 때때로 질의를 조금만 변경해도 옵티마이저가 다른 경로를 찾아 실행할 수도 있으며, 성능을 향상시키기 위해 색인을 추가하거나 정렬할 수 있도록 임시 공간을 추가해야 하는 경우도 있습니다.

옵티마이저의 결정 방법

옵티마이저가 어떻게 결정을 내리는가?

물론 이에 대한 대답은 간단하지 않습니다. 옵티마이저의 주 목표는, 필요한 데이터 양을 제한함으로써 I/O를 줄여 검색할 때 필요한 데이터를 가장 효율적인 방법으로 찾을 수 있게 하는 것입니다. 옵티마이저는 시스템 카탈로그의 정보를 바탕으로 결정을 내립니다. Informix OnLine, 버전 5.x에서는 이 정보가 다음 항목으로 구성되어 있습니다.

- 질의에 사용되는 각 테이블의 행 수 - systables.nrows
- 데이터에 사용되는 페이지 수와 색인에 사용되는 페이지 수 - systables.npused
- 칼럼 값의 유일성 여부 - sysconstraints
- 색인 존재 여부 - sysindexes
- 색인 순서(오름차순/내림차순) - sysindexes(Informix OnLine Dynamic Server, 버전 7에서는 필수

사항 아님.)

- 데이터가 색인과 같은 순서 즉, 클러스터 색인인지 여부 - sysindexes.clust
- 루트 노드에서 리프 노드까지 색인의 레벨 수
- 각 칼럼에서 두 번째로 큰 값과 두 번째로 작은 값. 옵티마이저는 이 정보로 값의 범위를 대략적으로 알아낼 수 있음. - syscolumns.colmin과 colmax(Informix OnLine Dynamic Server, 버전 7에서는 데이터 분산에 대하여 더 자세한 내용을 알아낼 수 있음. - sysdistrib table).

이러한 정보를 이용하여 옵티마이저는 가능한 모든 경로를 검색하고 각각 비용을 추정하여 평가합니다. 비용은 디스크 접근, 필요한 CPU 자원, 네트워크 접근 등 몇 가지 사항에 따라 계산됩니다. 이 과정에서 옵티마이저는 다음 작업을 합니다.

- 테이블 조인 순서 결정
- 순차적 스캔의 수행 여부
- 임시 테이블의 작성 여부
- 선택 목록, 필터, 정렬에 색인을 사용할 수 있는지, 또는 그룹별로 색인을 사용할지 결정

옵티마이저는 비용면에서 가장 효율적인 계획을 선택한 다음 질의를 처리합니다. SET EXPLAIN ON을 실행중이면 선택된 방법이 파일에 기록됩니다.

옵티마이저에 정확한 정보를 제공하는 것이 얼마나 중요한지 이해하려면, 다음의 간단한 질의와 옵티마이저의 조인 결정 예를 보십시오. 이 예에는 200개의 행이 있는 테이블(tab1)과 500,000개의 행이 있는 테이블(tab2)을 사용합니다. 두 테이블 모두 조인 칼럼에 고유 색인이 있습니다. 다음과 같이 간단한 select 문으로 두 테이블의 관련 행을 결정하고 있습니다.

```
SELECT * FROM tab1, tab2 WHERE tab1.col1=tab2.col2
```

최선의 경우 옵티마이저는 먼저 작은 테이블에서 선택하고, 색인을 이용하여 그 테이블을 두 번째 테이블에

조인합니다. 계산이 용이하도록 두 테이블 모두 한 페이지에 한 행만이 기록되어 있다고 가정하면, 첫 번째 테이블의 한 행당 두 번째 테이블은 세번의 색인 읽기(색인이 3 레벨) 및 한번의 데이터 읽기, 즉 tab1의 200번과 tab2의 200 x 4번이 되어 디스크를 1,000번 읽게 됩니다.

하지만 옵티마이저에 행 수와 유일 값에 대한 정확한 정보가 없다면 tab2를 먼저 선택했을 수도 있습니다. 그러면 tab2의 한 행당 tab1을 색인 두 번(색인 레벨 2), 데이터 한 번을 읽는 것이 되므로 500,000번에다 500,000 x 3번을 더하여 디스크를 2백만 번 읽게 됩니다.

또, 옵티마이저가 부정확한 정보(즉, 색인을 사용할 수 있다는 사실을 인식하지 못한 경우 등)로 작업하여 tab1의 한 행당 tab2의 모든 행을 하나하나 연속해서 읽어 나간다면 약 10억번(500,000 x 200)을 읽어야 합니다.

이 예를 보면 잘못된 정보와 결정이 성능(1,000번과 1,000,000,000번의 차이)에 치명적인 영향을 미친다는 사실을 뚜렷하게 알 수 있습니다. WHERE절에 필터 조건이 많아지고 테이블이 많이 포함되면 결정 과정이 더욱 복잡해지고 정확한 통계의 중요성도 더욱 커집니다.

통계의 정확도 옵티마이저가 사용하는 시스템 카탈로그 정보는 UPDATE STATISTICS가 실행될 때만 갱신된다는 것을 반드시 염두에 두어야 합니다. 옵티마이저는 제공된 정보가 좋은 만큼만 제몫을 하므로 가능한 한 UPDATE STATISTICS를 정기적으로 실행하는 것이 좋습니다. 이때 거의 변경되지 않는 테이블보다는 매우 동적인 테이블에 대해서 더 자주 실행하는 것이 좋습니다. 시스템 카탈로그 정보의 시효가 지나면 옵티마이저가 잘못된 결정을 내려 심각한 성능 문제가 생길 수 있습니다. 많은 업체에서 UPDATE STATISTICS를 정기적으로 실행하지 않는 것은 매우 유감스러운 일입니다. UPDATE STATISTICS는 데이터베이스 전체나 개별 테이블, 테이블의 칼럼, 내장 프로시저에 대하여 실행할 수 있습니다.

기억에 의존하는 것보다는 cron 명령으로 매일 밤 UPDATE STATISTICS가 실행되도록 자동화하는 것이 좋습니다. 카탈로그가 갱신될 때는 시스템 테이블에 잠시 로크가 걸리므로 다른 응용프로그램이 실행중인 동안 UPDATE STATISTICS를 실행할 때는 주의하십시오. 이 때문에 다른 프로세스에서도 오류를 발생시킬 수 있습니다. UPDATE STATISTICS를 처리하는 시간도 꽤 길기 때문에 데이터베이스를 사용하고 있는 동

안에는 너무 자주 갱신하지 마십시오.

엔진이 최적화를 수행하는 시기

주 변수가 없다면 SQL 문을 준비할 때마다 최적화가 수행됩니다. 주 변수가 있으면 옵티마이저는 변수가 전달될 때(커서가 오픈될 때)까지 필요한 정보를 얻을 수 없으며, 이 경우 OPEN 문에서 최적화가 일어납니다. 그래서 표준 SQL(즉, 준비되지 않은 SQL)로 사용될 때마다 질의가 최적화됩니다. SQL 문이 프로그램에서 반복 사용되면 SQL 문을 PREPARE하여 성능을 향상시키는 것이 좋습니다. 이렇게 하면 SQL은 재최적화 키워드로 다시 최적화하라고 요구하지 않는 한 프로그램에서 한 번만 최적화됩니다. 내장 프로시저의 경우 프로시저가 만들어지거나 그 내장 프로시저에 대하여 UPDATE STATISTICS가 실행될 때 SQL이 최적화됩니다.

질의가 갑자기 느려질 때

UPDATE STATISTICS를 정기적으로 실행하는데도 10분 걸리던 질의가 갑자기 한 시간 걸리는 경우가 있습니다. 질의 시간이 갑자기 오래 걸리는 것은 옵티마이저가 시스템 카탈로그의 새 정보에 따라 다른 경로를 선택하도록 결정했기 때문입니다. 옵티마이저가 더 나은 선택을 하게 하려면 질의를 다시 만들어야 하는 경우도 있습니다. 해결 방법은 프로그램을 다시 컴파일하여 질의 계획을 찾는 것이 아니라 프로그램에 SET EXPLAIN ON을 사용할 수 있는 기능을 제공하는 것입니다. 이 문제를 해결하려면 매개변수를 전달하도록 설정하거나 프로그램에 SET EXPLAIN ON을 선택할 환경 변수를 설정하는 단순 함수를 만듭니다.

개발과 생산

옵티마이저가 사용하는 통계는 자주 바뀝니다. 또한 작은 데이터베이스에서 개발하는 동안 질의를 시험할

티마이저가 같은 결정을 내리게 됩니다. 나중 방법은 손상이 일어났을 때 지원이 보류되므로 사용하지 않는 것이 좋습니다.

옵티마이저 제어

최적화 수준

Informix OnLine, 버전 5 이상에서는 엔진의 정보량에 영향을 미치는 기능이 제공됩니다. 이는 SET OPTIMIZATION HIGH 또는 LOW 명령으로 가능합니다. HIGH는 엔진이 모든 액세스 경로를 검사하도록 하고 LOW는 초기 단계에서 가능성이 적은 옵션을 제거하여 최적화 시간을 줄입니다. 여러 테이블(예를 들어, 다섯 개 이상)이 조인되어 최적화에 걸리는 시간이 너무 길어질 때는 LOW가 유용합니다. OPTIMIZATION LOW를 사용할 때의 문제점은 최적의 경로가 될 수 있는 액세스 경로가 너무 일찍 제거되어 버릴 수 있다는 점입니다.

OPTCOMPIND

Informix OnLine Dynamic Server, 버전 7 이후부터는 옵티마이저를 좀 더 잘 제어할 수 있습니다. 이전 버전에서는 색인이 가장 효율적인 액세스 경로를 제공하는 것으로 간주했으나 버전 7부터는 옵티마이저로 색인 읽기 비용과 테이블 스캔 비용을 비교하여 가장 효율적인 경로를 선택할 수 있습니다. 이 방법을 사용하려면 OPTCOMPIND=2(기본값)로 설정합니다. 이전 버전처럼 색인을 사용하도록 하는 기능은 OPTCOMPIND=0으로 설정할 수 있습니다. OPTCOMPIND=1로 설정하면 옵티마이저가 2로 설정되었을 때처럼 작동합니다. 단, REPEATABLE READ가 선택되면 OPTCOMPIND가 0으로 설정되었을 때처럼 작동합니다. 이 옵션으로 읽기를 반복하여 연속 스캔하면-행을 모두 스캔하면서-읽는 동안 효과적으로 전체 테이블을 공유할 수 있습니다.

옵티마이저를 위한 데이터 생산

Informix OnLine Dynamic Server, 버전 6부터는 데이터 분포 분석과 저장을 도입하여 옵티마이저가 보다 많은 정보를 갖고 결정을 내리도록 했습니다. 데이터 분포 분석을 사용하면 옵티마이저가 칼럼에 포함되는 값, 예를 들어 테이블 각 영역의 데이터 수와 그 데이터들 중 유일한 값은 얼마나 되는지 그 정도를 더 잘

알게 됩니다. 이 작업은 칼럼에서 데이터 표본을 채취, 테이블 영역에 관한 정보를 다양한 빈(bin)에 저장하는 것으로 이루어집니다. 이 정보는 특히 대형 테이블을 다루는 경우 옵티마이저에 아주 귀중한 자료가 됩니다. UPDATE STATISTICS 명령으로 각 칼럼 분포 정보를 생성할 수 있습니다. 표본 추출하는 데이터 양은 키워드 MEDIUM과 HIGH로 제어합니다. 예를 들어, UPDATE STATISTICS HIGH FOR 테이블명(칼럼명)과 같이 씁니다. MEDIUM은 데이터 표본만 추출하여 빨리 실행하지만, HIGH는 테이블의 행을 모두 평가하므로 더 느린 대신 정확합니다. LOW는 기능면에서 이전 버전의 UPDATE STATISTICS를 사용하는 경우와 비슷하며 데이터 분포 정보를 얻지는 않습니다. 데이터 분포 분석은 키워드 RESOLUTION을 사용하거나 밀도 값(bin의 수)과 신뢰 값(샘플링 수준) 등의 지정에 영향을 받습니다. 이 매개변수 사용에 대한 자세한 내용은 제품 설명서를 참고하십시오.

Informix OnLine Dynamic Server, 버전 7의 UPDATE STATISTICS

Informix OnLine Dynamic Server, 버전 7에서 UPDATE STATISTICS 방법을 사용할 때는 다음 작업을 순서대로 하는 것이 좋습니다.

1. 칼럼 리스트를 주지 않은 DISTRIBUTIONS ONLY와 RESOLUTION 매개변수는 기본값으로 하는

위 명령은 단편화 방법(fragmentation strategy) 결정에 유용하게 쓸 수 있습니다.

참고

대형 정적 테이블에 대해서는 UPDATE STATISTICS를 재실행할 필요가 없습니다.

옵티마이저의 생산고찰

옵티마이저에 충분한 정보를 제공하고 나면 옵티마이저가 선택한 질의 계획을 볼 수 있습니다. 질의나 Informix NewEra, Informix 4GL 또는 Informix ESQL 프로그램에서 SET EXPLAIN ON을 사용하십시오. 명령이 설정되면 옵티마이저는 현재 디렉토리에 sqexplain.out을 생성하여 같은 프로세스의 모든 질의에 대한 기록을 남깁니다. 보통 파일명과 위치는 운영체제에 따라 혹은 원격 호스트에서 질의를 실행할 때 달라질 수 있습니다.

다음은 SET EXPLAIN의 전형적인 출력입니다.

```
QUERY:

--------

select cust_id, order.* from orders, customers where order_date

> "01/12/1995" AND order_date < "01/01/1996" AND

customers.cust_id = orders.cust_id order by order_date DESC

Estimated Cost: 10
```

Estimated# of Rows Returned: 200

Temporary Files Required For: Order By

1) Informix.orders: INDEX PATH

(1) Index keys: order_date

Lower Index Filter: Informix.orders.order_date >

"01/12/1995"

Upper Index Filter: Informix.orders.order_date <

"01/01/1996"

2) Informix.customers: INDEX PATH

(1) Index keys: cust_id (Key-Only)

Lower Index Filter: Informix.customers.cust_id =

Informix.orders.cust_id

SET EXPLAIN 출력 이해

Query: {LOW}

이 영역에는 출력의 이전 영역에는 최적화된 실제 질의가 표시됩니다. 현재 SET OPTIMIZATION을 LOW로 설정하였다면 LOW가 함께 표시될 것입니다. sqexplain.out 파일이 이미 생성되어 있을 경우 sqexplain.out 은 계속 추가됩니다.

Estimated Cost:

"Estimated Cost" 값은 옵티마이저가 선택한 액세스 방법에 할당하는 숫자로서 옵티마이저에게만 의미가 있으며 실시간 처리와는 아무런 관련이 없는 상대적인 값입니다. 이 값은 다른 질의의 추정 비용과 비교하기보다는 같은 질의를 실행함에 있어서 어떤 변경 사항(예를 들어, 색인 변경)에 따른 비용 효과를 비교하기에 유용하게 사용할 수 있습니다.

Estimated# of Rows Returned:

옵티마이저가 시스템 카탈로그 테이블 정보를 바탕으로 추정한 값입니다. 카탈로그 정보는, 특히 Informix OnLine Dynamic Server, 버전 7 이전 버전의 경우 상당히 제한되어 있으므로 값이 부정확한 경우가 있습니다. 질의가 조인과 관련되어 있을 때 더욱 그런 경우가 많습니다. Informix OnLine Dynamic Server, 버전 7에서는 데이터에 대한 분포 정보를 얻어 옵티마이저가 질의에 리턴된 행 수를 더 정확히 추정할 수 있습니다.

Temporary Files Required For: Order By Group By:

Temporary Files Required For: Order By Group By 항목이 표시되었다면 질의에 GROUP BY문이나 ORDER BY문이 있으나 옵티마이저가 요구된 순서로 데이터를 얻을 수 있는 색인이 없을 경우입니다. 임시 파일은 질의 결과에 순서를 지정하기 위하여 만듭니다. 임시 파일은 테이블 크기에 따라 매우 커질 수 있으므로 사용할 수 있는 디스크 공간을 확인하고 성능에 미칠 영향을 고려해야 합니다. 정렬할 칼럼이 여러 테이블에서 온 경우에는 색인을 사용할 수 없습니다. Informix OnLine Dynamic Server, 버전 7에서는 색인 지정 순서가 ORDER BY와 같지 않아도 옵티마이저가 ORDER BY 방향으로 색인을 고찰할 수 있습니다. Informix OnLine Dynamic Server, 버전 7 이전에는 색인의 ASCENDING/DESCENDING의 여부와 ORDER

BY에 명시된 칼럼의 그 순서가 일치하는가에 따라 옵티마이저가 ORDER BY에 대해 색인을 사용할 수 있었습니다.

1) owner.table: INDEX PATH (Key-Only)

owner.table은 *1)*이라고 표시된 대로 옵티마이저가 가장 먼저 읽는 테이블입니다. 읽어야 하는 테이블이 또 필요하다면(예를 들어, 중첩 루프 조인의 경우) *2)*, *3)* 등으로 더 큰 숫자와 함께 표현될 것입니다. 이 단계에서 반환된 각 행에 대해서 엔진은 좀더 하위 단계에서 테이블을 질의합니다. INDEX PATH는 테이블 검색에 사용된 색인을 나타냅니다.

Informix OnLine의 Key-Only 표시는 단지 색인만 읽을 뿐 실제 데이터 값(행)은 이 테이블에서 읽어오지 않음을 뜻합니다. 일반적으로 행에 비해 색인 크기가 작으므로 Key-only 검색은 매우 효율적입니다(Informix OnLine Dynamic Server, 버전 7.2 이전 버전). 이는 실제 데이터 행 읽기가 제거될 뿐 아니라 색인 키 값은 한 페이지에 실제 데이터보다 훨씬 많이 들어가므로 I/O가 줄어듭니다. 그러나 Key-only 검색은 색인이 부여된 칼럼 외에 다른 칼럼은 선택 리스트에 없을 경우에만 가능합니다. 따라서 될 수 있으면 *SELECT **을 사용하지 않는 것이 좋으며, 필요한 칼럼만 선택해야 합니다. Informix OnLine Dynamic Server, 버전 7.2에서는 미리 읽기(Read-Ahead) 용량 때문에 Key-Only 읽기가 간혹 더 느려지는 일이 있습니다.

(1) Index Keys: column_name

Lower Index Filter: owner.table.column > x

Upper Index Filter: owner.table.column < y

column_name은 INDEX PATH 읽기에서 사용된 칼럼명입니다. Lower Index Filter는 색인 읽기가 시작된 첫 키 값(x)을 나타냅니다. Upper Index Filter는 색인 읽기가 끝난 지점의 키 값(y)을 나타냅니다.

1) owner.table: SEQUENTIAL SCAN (Serial, fragments: ALL)

이렇게 표시되었다면 테이블에서 모든 행을 순차적 검색으로 읽습니다. 괄호로 묶은 부분은 Informix

OnLine Dynamic Server, 버전 7과 관련된 부분입니다. Serial 대신 Parallel이 표시되면 엔진이 병렬 검색을 실행했음을 의미합니다. 이 작업은 PDQPRIORITY 설정의 영향을 받습니다. ALL 표시는 분할된 모든 영역을 검색했음을 나타내며, 이는 옵티마이저가 WHERE절을 분석한 후 어떤 분할 영역도 검색 조건에서 제외할 수 없었음을 알 수 있습니다. NONE은 그 반대 상황, 즉 옵티마이저가 검색 조건을 분석한 후 어떤 분할 영역도 살펴볼 필요가 없었음을 나타냅니다. 숫자(또는 숫자 목록)는 나열된 분할 영역만 검색했음을 나타냅니다. 이때 숫자는 sysfragments 테이블의 순서에 따라 다릅니다. 질의 계획의 하위 단계(보통 더 큰 숫자로 표시됨)에서 순차적 검색이 실행되면, 이전 단계에서 반환된 각 행에 대하여 테이블 전체가 검색됨을 나타내므로 특히 주의하십시오. 이것은 질의 QA를 실행하거나 최적화할 때 주의하라는 경고가 됩니다. 큰 테이블에서 아주 적은 일부만을 검색하고자 할 때 그 테이블 모두를 순차적 검색하는 것은 당연히 필요없는 일입니다. 그러나 그것을 막는 질의 계획, 즉 질의의 첫 단계부터 순차적 검색을 실행한다든지, 작은 테이블이라도 순차적 검색을 계속하는 것은 좋은 방법이 아닐 것입니다.

AUTOINDEX PATH: owner.table.column

Informix OnLine, 버전 4에서 자주 사용되는 설정입니다. 순차적 검색을 피하기 위해 조인에 사용된 칼럼에 임시 색인을 만들어 조인을 실행합니다. 이 설정은 조인 칼럼에 색인이 없을 때 사용되며 일반적으로 영구 색인이 필요함을 나타냅니다.

SORT SCAN: owner.table.column

SORT SCAN은 칼럼에서 색인을 사용할 수 없을 때 순차적 검색과 결합되어 사용됩니다. 이 칼럼은 정렬 후 나중에 조인에 사용되며, MERGE JOIN이 될 수도 있습니다.

MERGE JOIN Merge Filters: owner.table.column =

owner.table.column

병합 조인(Merge Join)은 이전에 조인하기 위하여 준비된 두 선택 결과를 조인하는 데 사용됩니다. 조인 칼럼을 적절한 순서로 얻으면(색인이 없으면 SORT SCAN으로 얻을 수 있음) 서버는 이미 준비된 양쪽 결과를

순차적으로 읽어가면서 실제 데이터 행을 검색하기 전에 먼저 병합합니다. 이 방법이 중첩 루프 조인(Nested loop join)보다 일반적으로 빠르다고 여겨지는 경우가 많습니다.

DYNAMIC HASH JOIN (Build Outer) Dynamic Hash Filters:

owner.tab1.col = owner.tab2.column ...

Informix OnLine Dynamic Server, 버전 7에서만 사용하는 방법입니다. 해시 조인 역시 먼저 검색된 두 테이블을 조인하는 데 사용합니다. Build Outer 표시는 먼저 사용될 테이블을 나타내고 필터는 테이블 조인 방법을 나타냅니다. 복잡한 질의가 색인을 사용할 수 없을 때 해시 조인이 인수 받습니다. 해시 조인은 또한 정렬-병합 조인(sort-merge join) 대신 사용할 수 있고, 더 효율적입니다. 정렬-병합 조인이 두 테이블을 모두 정렬하는데 반해 해시 조인은 보통 한 테이블만 정렬합니다. 해시 조인은 많은 데이터를 다룰 때, 특히 분할 영역이 있는 PDQ에서 자주 사용됩니다. 내부 해시 알고리즘을 사용한 다음 행을 해시 테이블에 놓습니다. 해시 조인 비용은 색인을 사용할 때보다 낮은데 특히 큰 테이블에서 데이터 15% 이상을 검색해야 할 때 그렇습니다. 데이터가 클러스터링되어 있지 않을 때 색인을 읽고 추가적으로 실제 행을 검색해야 하는 비용은 해시 조인에 비해서 꽤 높은 편입니다. OPTCOMPIND=2(기본값 7.10.UD1 이상 버전)일 때 색인을 사용하는 대신 옵티마이저는 해시 조인을 사용할 것입니다. REPEATABLE READ를 사용할 때에는 OPTCOMPIND=1로 설정하십시오.

SQL 질의 품질 보증과 최적화

다음은 SQL 질의 품질 보증과 최적화할 때 고려해야 할 사항입니다.

1. 질의 계획의 첫번째 위치에 있지 않으면 큰 테이블에서는 순차적 검색을 하지 않는 것이 좋습니다. 바로 앞 테이블의 각 행마다 전체 테이블 스캔이 반복되기 때문입니다. 이렇게 하면 성능이 저하될 수 있습니다. 이는 실행 중인 질의에 뿐만 아니라 공유 메모리(전체 테이블을 공유 메모리로 계속 읽어 오게 될 때)에서 최근에 사용한 페이지를 바꾸어 다른 작업에도 영향을 미칩니다. 다른 프로세스도 디스크로부터 읽어야 하므로 일반적으로 디스크 I/O가 늘어납니다. 질의를 다시 생성할 수 없고 속도가 느리다면 색인 추가를 고려해 보십시오.

2. 임시 정렬 파일이 크게 생성되도록 하는 질의는 사용하지 마십시오. 이런 파일은 CPU 자원을 모두 소모하고, 디스크 I/O를 늘리며, 디스크 공간을 모두 소모하기 때문입니다. 정렬할 칼럼에 색인을

추가하십시오. 이 경우 ORDER BY절에 기존 색인 칼럼을 사용하지 않으면, 옵티마이저는 WHERE절에 명시한 칼럼에 의해 순서를 지정하지 않을 수도 있습니다. 이 때문에 옵티마이저가 임시 파일을 만드는 대신 색인을 사용할 수 있도록 하기 위하여 ORDER BY절을 명시하거나 WHERE절에 더미(dummy) 조건을 줄 수도 있습니다. Informix OnLine Dynamic Server, 버전 7 이전에는 색인이 ORDER BY에 명시한 내용과 같은 순서일 때 색인 사용이 가능했습니다. 정렬 파일을 사용할 때는 $PSORT_DBTEMP와 $DBSPACETEMP 값을 확인하십시오. 이 값을 사용하여 엔진이 정렬 파일에 여러 디스크를 사용하도록 하여 성능을 크게 향상시킬 수 있기 때문입니다.

3. 관련된 부질의(Correlated Subquery)를 사용하십시오. 주 질의에서 선택한 칼럼을 WHERE절에서 참조하는 부질의어는 성능에 큰 문제가 생길 수 있습니다. 이 참조는 부질의를 주 테이블의 각 행에 대하여 되풀이 실행합니다. 큰 테이블에서 EXISTS와 같은 명령문을 사용할 때는 그 프로세스가 논리 로그를 매우 빨리 채울 수 있으므로 주의하십시오. 비록 최종적으로 리턴되는 행이 없다고 하더라도, 임시 테이블 영역은 주 테이블의 각 행에 대하여 할당되고 로깅됩니다. 이때 발생할 수 있는 가장 최악의 상황은 논리 로그가 꽉 차는 경우이며 백업된 자료로부터 재저장해야만 합니다. 관련된 부질의는 가능하면 조인을 사용하도록 다시 작성되어야 합니다. 더욱 복잡하게 재작성될 때 임시 테이블을 만들어 그 테이블과 조인하도록 해야 할 수도 있습니다.

4. 서로 다른 칼럼에 사용된 OR 문은 옵티마이저의 색인 사용을 방해합니다. 색인이 있고 질의 계획에서 옵티마이저가 순차 스캔을 선택하면 UNION문(각 OR 조건에 하나씩) 사용을 고려해 보십시오. 옵티마이저가 색인을 사용할 기회를 제공할 수 있습니다.

5. 질의 실행이 느리다면, 질의 계획의 모든 단계에서 INDEX PATH가 선택되었어도 옵티마이저가 제대로 결정을 내렸다고 생각하지 마십시오. 질의 계획을 확인하여 테이블이 올바른 순서로 필터링되었는지 확인해 봐야 합니다. 목표는, 질의 초기 단계에 될 수 있으면 많은 행을 제거하는 것입니다. 하지만 옵티마이저에 항상 이 작업을 제대로 수행할 만한 정보가 있는 것은 아닙니다. 특히 Informix OnLine Dynamic Server, 버전 6 이전에는 그렇습니다. 이후 버전에서 UPDATE STATISTICS HIGH을 사용하면 옵티마이저에 데이터 분포에 대한 정보가 추가로 제공되므로 알맞은 테이블이 먼저 제거됩니다. INDEX 읽기가 항상 최선의 방법은 아니라는 점에 주의하십시오. 예를 들어, 데이터 페이지를 검색함에 있어서 각 데이터 페이지가 물리적인 순서로 되어 있지 않으면

테이블 순차 검색이 색인 검색보다 빠를 수도 있습니다.

6. 자료형을 변환하고 문자 칼럼을 비교하는 것, 예를 들어 tab1.character_col = tab2.integer_col은 처리 비용이 매우 큰 작업입니다. 가능하면 칼럼의 자료형을 숫자형으로 바꾸어 보십시오. 조인 칼럼이 문자형이면 칼럼을 매 행마다 한 바이트씩 비교해야 합니다. Informix OnLine Dynamic Server, 버전 7에서는 변환을 더 잘 처리하게 되었지만 아직도 변환 오버헤드 비용은 옵티마이저에서 고려되지 않습니다.

7. 색인을 사용하지 못하는 경우가 있는 WHERE절을 찾으십시오. 이 절은 대표 문자로 시작하며 OR, LIKE 또는 MATCHES(예를 들어 MATCHES "*NOUR"), 함수(예를 들어 MONTH, DAY, LENGTH 등)나 부정 표현(예를 들어 !="NOUR"), 그리고 첫 문자를 제외한 하위 문자열 검색(예를 들어 postcode[4,5]>10) 등이 있습니다.

8. 아주 오래된 버전 엔진을 제외하면 SELECT문의 선택 리스트나 테이블 순서, WHERE 절의 구성 요소 순서는 효력을 미치지 않습니다. 하지만 옵티마이저가 두 경로의 비용이 같다고 평가할 때는 WHERE 절의 명령문 순서가 영향을 미칩니다.

9. 로깅된 데이터베이스에서 긴 트랜잭션을 실행하지 마십시오. 긴 트랜잭션은 논리 로그를 채워 데이터베이스를 손상시킬 위험이 있습니다. LOAD문, INSERT INTO xx SELECT yy FROM zz, UPDATE 또는 여러 행에 걸쳐 있는 DELETE문 등은 주의 깊게 살펴보아야 합니다. 로크를 지나치게 많이 사용하는 것을 막으려면 테이블을 단독 모드(exclusive mode)로 로크하십시오. 로크를 지나치게 사용하면 성능이 저하되고 나아가 로크 최대수에 도달하면 명령문이 실행되지 않습니다.

10. WHERE 절에 필요한 모든 조건이 포함되었는지 WHERE 절에서 조인 칼럼과 조건을 확인하십시오. 예를 들어, WHERE tab1.x = tab2.x and tab1.x > 1000는 옵티마이저가 tab1.x에서 색인을 사용하게 하지만 tab2.x의 색인이 더 적합합니다. AND tab2.x > 1000 조건을 추가하면 옵티마이저의 선택 범위가 넓어집니다. 다른 예는 tab1.x = tab2.x AND tab2.x = t ab3.x입니다. Informix OnLine, 버전 5의 옵티마이저는 tab1을 바로 tab3에 조인하는 것은 고려하지 않으므로 tab1.x = tab3.x를 추가하면 선택의 폭이 넓어집니다. 하지만 Informix OnLine Dynamic Server, 버전 7에서는 옵티마이저가 질의를 다시 구성하여 가능한 모든 조합을 살펴보게 하므로 이 제안이 해당되지 않습니다.

11. 필요한 칼럼만 선택하십시오. 그러면 프론트엔드와 백엔드 간의 통신이 줄고 I/O도 줄어듭니다. 되도록 SELECT *...문을 사용하지 마십시오. 이 질의는 모든 칼럼을 다 선택하지만 모든 칼럼이 항상 필요한 것은 아닐 것입니다.

12. 데이터 일정 부분 집합이 WHERE 술어로 다시 선택되면 임시 테이블을 사용하십시오. 예를 들어, SELECT ... FROM tab1 WHERE orderdate > "01/12/1995" and x = 1의 문과 SELECT ... FROM tab1 WHERE orderdate > "01/12/1995" and x = 2의 문이 바로 연이어 실행된다고 가정해 보십시오. 이 경우, 테이블이 매우 크면 먼저 orderdate > "01/12/1995"의 조건에 맞는 모든 행을 임시 테이블로 선택하고 임시 테이블에서 재검색을 해보십시오.

13. 옵티마이저가 가장 좋은 경로를 선택하도록 임시 테이블을 사용할 수 있습니다. 먼저 큰 테이블에서 임시 테이블로 행을 선택하고 임시 테이블을 나머지 테이블에 조인하면 됩니다. 옵티마이저가 임시 테이블이 원본 테이블보다 훨씬 작다는 사실을 인식하면 다른 질의 계획을 사용할 것입니다.

14. 임시 테이블에 색인을 만들어 보십시오. 임시 테이블에서 흔히 놓치기 쉬운 작업입니다.

15. 임시 테이블을 ORDER BY와 함께 연속 선택에 사용할 때는 ORDER BY로 임시 테이블을 만드십시오. 예를 들어, SELECT x FROM y ORDER BY x INTO TEMP temp_tab와 같이 씁니다. 이 구문은 Informix OnLine, 버전 4.1 이전에는 사용할 수 없습니다.

16. 큰 임시 테이블에 대해서는 응용프로그램에서 UPDATE STATISTICS를 사용하십시오. 이 작업도 임시 테이블에서 놓치기 쉽습니다.

17. 명시적으로 임시 테이블을 만들 때는 WITH NO LOG 문을 사용하십시오. 논리 로그에 쓰는 오버헤드가 없어지므로 성능이 향상됩니다. 또한 큰 임시 테이블에 긴 트랜잭션을 만들 가능성이 없어집니다. Informix OnLine Dynamic Server, 버전 7에서는 임시 테이블을 로깅되지 않은 특수 dbspace에 만들 수 있습니다. 될 수 있으면 이 기능을 사용하십시오.

18. 검사하거나 최적화할 질의 시간을 잴 때는 질의를 실행하기 전후에 UNIX의 time 또는 timex 명령, 또는 SELECT CURRENT FROM systables WHERE tabid = 1 형식의 문장을 사용하십시오.

19. 질의를 검사하거나 시간을 잴 때는 재실행하는 질의가 전에 읽힌 페이지가 메모리에 남아 있기 때

문에 더 빨라진다는 사실을 기억하십시오. onstat -p(tbstat -p)로 디스크나 버퍼 사용을 확인할 수 있습니다. 질의 시간을 잴 때는 검사 사이에 인스턴스를 재시작하여 공유 메모리를 초기화합니다.

20. 단편화와 병행 질의에 관해서는 옵티마이저가 단편화된 분할 영역을 쉽게 결정할 수 있도록 데이터를 분할하는 것이 중요합니다. WHERE 절이 명확해야 옵티마이저가 검색이 필요없는 분할 영역을 제거하여 I/O를 줄이고 병행 처리를 활성화할 수 있습니다.

요약

Informix 비용 기준 옵티마이저(cost-based optimizer)는 여러 정보를 다양하게 수용하고 있으므로 DBA 대신 가장 좋은 데이터 검색 방법을 결정합니다. 따라서 일반적으로 옵티마이저의 결정이 당연하게 받아들여지는 일이 많습니다. 하지만 옵티마이저와 옵티마이저가 내린 결정의 결과를 이해하는 것은 중요한 일입니다. 또한 더욱 더 중요한 것은 옵티마이저에 작업을 효율적으로 수행할 수 있는 정보를 제공하는 것입니다. Informix는 계속해서 옵티마이저를 개선하며 성능을 향상시킬 방법을 찾고 있습니다. 이 글에 언급된 사항 중에는 앞으로 출시될 버전에서는 오히려 쓸모 없는 내용이 될 수도 있을 것입니다.

사용가능 DB영역 결정

서론

디스크 드라이브가 많이 포함된 데이터베이스 인스턴스 관리 작업은 매우 복잡합니다. 그러나, 2,3백개의 디

스크 드라이브를 포함한 인스턴스가 빠른 속도로 표준으로 자리잡고 있습니다. 물리적 수준에서 큰 데이터베이스를 구현할 때 데이터베이스 관리자(DBA)는 고도로 분할된 테이블과 색인을 최대한으로 사용하는 것이 일반적입니다. 그러나, 그 결과 만들어진 DB 영역과 청크 수는 쉽게 관리할 수 없을 만큼 커집니다.

데이터베이스 인스턴스의 전 기간에 걸쳐 테이블이나 인덱스를 삭제하고 다시 만들 수도 있습니다. 하지만, 이 작업이 항상 원래 청크에서 이루어지는 것은 아닙니다. 결국, 이 활동에는 인스턴스 전용 드라이브 사이에서 이미 사용된 디스크 공간과 사용 가능한 공간이 끼어들게 됩니다. 청크 수가 늘어남에 따라 onstat -d와 같은 유틸리티는 그다지 유용하지 않을 수도 있습니다. 장치명과 그 안에서 사용할 수 있는 공간에 대한 정보와, 생성할 수 있는 최대 청크 크기와 오프셋을 나열하는 스크립트로 이를 해결할 수 있습니다. 다음의 get_avail_space가 바로 이 작업을 해주는 스크립트입니다.

스크립트설명

이 스크립트는 sysmaster 데이터베이스를 질의하고 해당 인스턴스의 모든 드라이브로부터 정보를 모읍니다. 그런 다음, 사용 가능한 공간이 있는지 확인하고 장치명과 해당 공간의 오프셋과 최대 크기를 표시합니다.

참고

다음 유틸리티는 사용자 Informix로 실행해야 합니다. 또한 이 유틸리티는 Solaris 2.5.1 환경의 Sun* Microsystems SparcCenter 2000과 Informix OnLine Dynamic Server, version 7.12에서 테스트했습니다.

사용 가능 DB 영역을 결정하는 스크립트

```
#############################################################

#!/bin/ksh

#

# 스크립트:    get_avail_space
```

```
# 날짜:          23 August 1996

#




export InformixDIR=${InformixDIR}

export InformixSERVER=${InformixSERVER}

export ONCONFIG=${ONCONFIG}

export PATH=${PATH}

outfile=/tmp/freespace$$.out




print "₩n₩n 드라이브의 빈 공간 정보 획득₩n₩n"

dbaccess sysmaster -<$outfile 2>/dev/null

set isolation to dirty read;

select b.fname, (b.offset*2) OFFSET, (b.chksize*2)

CHKSIZE, ((b.offset + b.chksize)*2) TOTAL
```

```
from sysdbspaces a, syschunks b

where a.dbsnum=b.dbsnum

order by 1,2 ;

!

# 모든 드라이브 크기가 같은 것으로 간주합니다.

drive_size=655360



#주의: 파이프 뒤에 공백을 주지말고 다음 줄에 역슬래시(₩)를 넣습니다.

cat $outfile|₩

nawk `BEGIN { x=1 }

    {

    # 다음은 개행(빈 줄)으로 시작하는 것을

    # 모두 삭제합니다.

      if( $1 ~ /fname/ ) {
```

```
        fname[x]=$2

    }

    if( $1 ~ /offset/ ) {

        offset[x]=$2

    }

    if( $1 ~ /chksize/ ) {

        chksize[x]=$2

    }

    if( $1 ~ /total/ ) {

        total[x]=$2

        x++

    }

}END{

    for( i=1; i<=x; i++) {
```

```
  if( fname[i] == fname[i+1] ) {

    if( total[i] != offset[i+1] ) {

      # 이전의 전체 크기와 다음 오프셋 크기 사이에

      # 차이가 있습니다.

      printf("DRIVE NAME = %-10s OFFSET = %-10d MAX SIZE = %-10d₩n, fname[i],

            total[i], offset[i+1] - total[i])

    }

  }else{

    if( total[i] < `$drive_size' ) {

      printf("DRIVE NAME = %-10s OFFSET = %-10d MAX SIZE = %-10d₩n, fname[i],

            total[i], offset[i+1] - total[i])

    }

  }

}
```

```
    }'|sort -k 5,6

print "₩n₩n"

rm $outfile
```

출력예

```
DRIVE NAME = /DRIVE/D270   OFFSET = 255000   MAX SIZE = 56000
DRIVE NAME = /DRIVE/D270   OFFSET = 586000   MAX SIZE = 69924
DRIVE NAME = /DRIVE/D271   OFFSET = 255000   MAX SIZE = 56000
DRIVE NAME = /DRIVE/D271   OFFSET = 586000   MAX SIZE = 69924
DRIVE NAME = /DRIVE/D272   OFFSET = 255000   MAX SIZE = 56000
DRIVE NAME = /DRIVE/D272   OFFSET = 586000   MAX SIZE = 69924
DRIVE NAME = /DRIVE/D273   OFFSET = 255000   MAX SIZE = 56000
DRIVE NAME = /DRIVE/D273   OFFSET = 596000   MAX SIZE = 59924
DRIVE NAME = /DRIVE/D274   OFFSET = 255000   MAX SIZE = 56000
DRIVE NAME = /DRIVE/D274   OFFSET = 594000   MAX SIZE = 61924
DRIVE NAME = /DRIVE/D275   OFFSET = 255000   MAX SIZE = 56000
DRIVE NAME = /DRIVE/D275   OFFSET = 586000   MAX SIZE = 69924
```

결론

제공된 스크립트는 drive_size를 상수로 정의합니다. 즉, 인스턴스 전용 드라이브는 모두 크기가 같다고 간주한 것입니다. 이 조건이 적용되지 않으면, 스크립트의 일부를 수정해야 할 것입니다.

스키마 최적화

서론

데이터베이스 스키마는 테이블 정의, 색인 정의, 권한 등으로 이루어집니다. 스키마 최적화는 시스템 자원을 효율적으로 이용(즉, 디스크 공간 절약, I/O 요구 감소)하고 데이터에 효과적으로 접근할 수 있도록 테이블을 구성하는 일 등을 포함합니다. 비효율적인 스키마는 해당 테이블의 디스크 공간을 많이 차지하고, 유지보수하기에도 힘들며, 데이터에 접근하는 I/O 양도 많아집니다. 따라서 성능이 떨어지게 됩니다. 이 글에서는 성능 향상을 위한 스키마 최적화 방법에 대해 설명하고자 합니다.

칼럼정의

최적화에 대해 논의하려면 먼저 테이블에서 정의하는 칼럼의 자료형부터 다루어야 합니다. 각 칼럼에는 정수보다 문자열이 많이 쓰이며, 이는 많은 사용자가 수학적 계산에 쓰이지 않는 필드를 일반적으로 문자열로 처리하기 때문입니다. 그러나 이는 다음의 두 가지 이유로서 비효율적이라고 하겠습니다.

첫번째 이유는 데이터 영역에 관련된 것입니다. 숫자 값만을 포함하는 8 문자 필드는 레코드당 8 바이트를 사용합니다. 그러나 이 칼럼을 정수와 같은 숫자 필드로 처리하면 레코드당 4 바이트로 충분히 자료를 저장할 수 있습니다. 따라서 이 칼럼을 색인으로 사용하면 키 값 크기도 작고 레코드 크기도 작게 처리할 수 있습니다. 행 크기가 작을 때의 이점은 이 글 뒷부분에서 설명합니다.

Customer 테이블	부정확한 칼럼 정의	바이트 수	정확한 칼럼 정의	바이트 수
customer_num	char (8)	8	integer	4
lname	char (30)	30	char (20)	20
fname	char (30)	30	char (15)	15
credit_limit	decimal (11,2)	7	decimal (7,2)	5
last_update	datetime year to fraction(5)	11	datetime year to minute	7
	합계	86	vs.	51

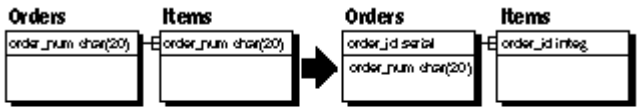
비효율성의 두번째 이유는 비교에 관련된 것입니다. 데이터베이스 엔진이 정수 필드 두 개를 비교할 때 8 문자 필드 두 개를 비교하는 것보다 훨씬 빠릅니다. 문자열이 동일한지 검사하기 위해서는 엔진이 각 글자를 비교해야 합니다. 그러나 이와 달리 정수 필드를 비교하는 데는 한 번이면 충분합니다.

칼럼 정의와 관련된 또 하나의 문제는 필드 크기에 관한 것입니다. 필드가 지나치게 커지는 것은 흔히 있는 일입니다. 특히 데이터베이스가 처음 만들어져 데이터의 실제 범위를 결정하기 힘들 때는 더욱 그렇습니다. 예를 들어, 데이터베이스에서 가격 칼럼을 money(11,2)로 정의하면 $999,999,999.99 까지의 값이 들어갈 수 있습니다. 그렇지만 그렇게 큰 금액은 거의 나오지 않으므로 칼럼을 money(5,2)로 정의하여 $999.99 까지의 값을 포함할 수 있게 하여 레코드당 2~3 바이트의 공간을 절약하는 것이 좋습니다.

기본키와 외부키

또 하나의 흔히 일어나는 스키마 디자인 오류는 사이즈가 큰 기본 키를 다른 테이블의 외부 키로 사용하는 것입니다. 예를 들어, orders 테이블의 기본 키가 20바이트 문자 칼럼인 주문 번호이므로 그 기본 키를 items 테이블에서 외부 키로 사용하는 것은 효율적입니다.

하지만 더 좋은 방법은 orders 테이블에서 serial 필드를 정의하고 주문의 연속되는 번호를 items 테이블로 가져오는 것입니다. 이렇게 하면 items 테이블의 공간이 절약되고 두 테이블이 정수 필드로 조인되므로 조인 속도가 빨라집니다. 한 가지 문제는 items 테이블을 주문 번호로 직접 접근할 수 없고 orders 테이블을 먼저 읽어서 주문 번호를 검색한 후 주문의 serial 값을 확인하여 다시 items 테이블에 접근해야 한다는 것입니다.

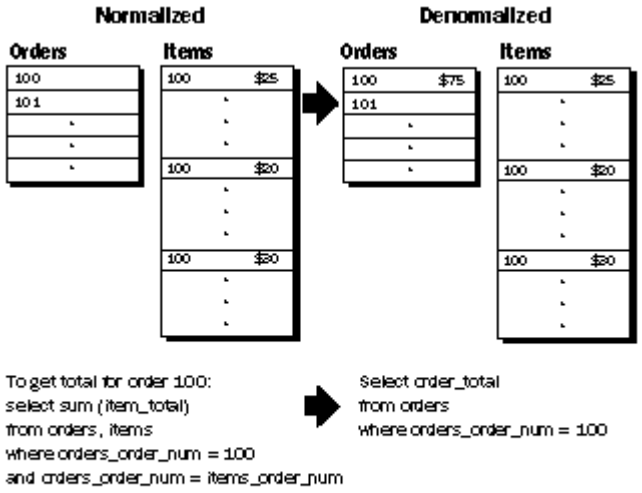


Fennimore Figure 02

그림 2: 문자열 대신 정수를 외부 키로 사용한 예

비정규화(Denormalization)

성능을 향상시킬 수 있는 또 하나의 방법은 데이터베이스를 "비정규화(Denormalization)"하는 것입니다. 일반적으로 정규화된 데이터베이스에는, 성능 향상을 가져오는 파생 데이터가 포함되지 않기 때문입니다. 가령 주문 총액을 자주 질의하며 그 질의가 값을 빨리 반환해야 하는 경우, 이 질의를 수행하는 한 가지 방법은 주문과 관련된 항목 전체의 합계를 구하는 것입니다. 주문에 항목이 많지 않은 시스템이라면 이런 방법도 문제가 되지 않습니다. 하지만 주문마다 수백 가지 항목이 있다면 합계를 구하는 데 I/O가 더 많이 요구될 것입니다. 대신 orders 테이블에 총액 필드를 만들면 데이터베이스에서 한 레코드만 읽으면 주문 총액을 구할 수 있을 것입니다.



Fennimore Figure 03

그림 3: 파생 데이터를 사용하여 질의 속도를 높인 예

비정규화로 성능을 향상시키는 또 다른 예는 세부 테이블의 데이터를 중복시키는 것입니다. 어떤 한 테이블(master)에는 재고 상자에 대한 정보만 들어있고, 또 다른 테이블(detail)에 각 상자의 내용을 나열한다고 가정합니다. carton 테이블에는 상자의 상태가 들어가고, carton_detail에는 상자 속의 실제 항목이 들어갑니다. 재고에서 특정 항목이 들어있는 상자를 모두 찾으려면 다음과 같이 carton 테이블과 carton_detail 테이블을 조인해야 합니다.

```
select *
from carton, carton_detail
where carton.id = carton_detail.id
and carton.status = "A"
and carton_detail.item = "some_item"
```

이 질의를 자주 사용하는 경우 status 칼럼을 carton_detail 테이블로 옮겨 carton 테이블과 조인할 필요가 없도록 하는 것이 좋습니다. 이 경우 색인은 item 칼럼에 만들고 status 칼럼은 carton_detail 테이블에 만들면 질의를 빠르게 실행할 수 있을 것입니다.

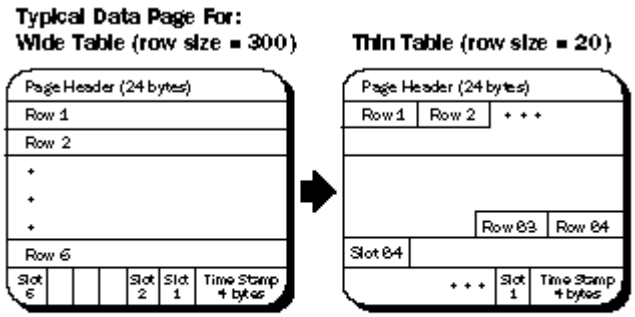
일반적으로 "비정규화"를 꺼리는 이유는 파생되고 반복되는 데이터에 필요한 프로그래밍과 추가 오버헤드 때문

입니다. 또한 파생되거나 반복되는 데이터가 제대로 갱신되지 않으면 자료 무결성이 손상될 위험도 있습니다. 예를 들어, carton 테이블의 status 칼럼 값이 갱신되고 carton_detail 테이블의 status는 갱신되지 않으면 자료 무결성이 깨지게 됩니다. 이 문제들은 트리거나 내장 프로시저를 사용하여 비정규화된 데이터의 무결성을 유지할 수 있습니다. 스키마를 비정규화할 때 그만한 오버헤드가 따르기는 하지만 중요한 것은 질의 성능을 향상시킨다는 점입니다.

한 행의 사이즈가 큰 테이블

사이즈가 매우 큰 행을 만들 수 있는 테이블은 엔티티(*주문*과 같은)에 대한 모든 정보를 저장하는 경우가 많습니다. 행 크기가 큰 테이블을 와이드 테이블(wide table)이라 하는데 몇 가지 이유로 비효율적입니다. 그 중 하나는 캐쉬할 수 있는 자료의 양입니다.

예를 들어, 행을 디스크에서 읽을 때는 먼저 그 행이 포함된 전체 페이지를 공유 메모리로 읽습니다. 그런 다음, 해당 행을 공유 메모리에서 읽습니다. 필요한 행이 이미 공유 메모리에 존재한다면 디스크에서 공유 메모리로 물리적으로 읽어 오는 데이터 캐쉬는 일어나지 않습니다. 즉 한 페이지에 저장할 수 있는 행이 많아질수록 메모리로 읽을 수 있는 행도 많아집니다.



Fennimore Figure 04

그림 4: 와이드 테이블(wide table)과 씬 테이블(thin table)의 자료 페이지 내용. 행 크기가 작으면 한 페이지에 들어가는 행 수가 많아지고 캐쉬 양도 늘어납니다.

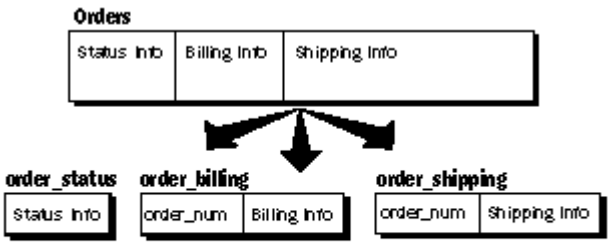
와이드 테이블과 관련된 또 하나의 성능 문제는 로깅입니다. 로깅을 하는 데이터베이스에서 행이 갱신, 삽입 또는 삭제되면, 전체 레코드가 트랜잭션 로그에 쓰여집니다. 갱신을 하면 수정되기 이전과 이후 자료(before

image, after image)가 디스크에 쓰여지므로 가장 큰 문제일 수 있습니다.

주문에 관한 정보가 모두(status, billing, shipping 정보 등) 들어있는 orders 테이블의 예를 살펴보겠습니다. 행의 길이가 300바이트(status 20바이트, billing 80 바이트, shipping 200바이트)라고 가정합니다.

주문이 들어오면 이 정보가 모두 한 테이블에 들어갑니다. 주문을 처리하면서 status 정보는 자주 접근하고 변경하지만, billing이나 shipping 정보는 그렇지 않습니다. 그러나 가장 자주 변경되는 status_flag(1문자)가 갱신될 때마다 갱신 전후의 자료(전체 600바이트)가 트랜잭션 로그에 쓰여집니다. 따라서 고작 한 바이트의 문자 필드를 갱신하기 위해 너무 많은 오버헤드가 생기게 됩니다. 또한 레코드의 status 정보를 검색할 때 행 전체가 읽힙니다. Informix OnLine Dynamic Server에서는 행이 포함된 페이지 전체가 읽히므로, 각 행의 총 300바이트 중 280바이트는 읽히기는 하지만 사용되지는 않습니다.

이런 비효율성을 바로 잡으려면 테이블을 일대일 대응하는 세 개의 테이블로 나눕니다. order_status 테이블(20바이트), order_billing 테이블(80바이트 + order_num), order_shipping 테이블(200바이트 + order_num)입니다. 이렇게 하면 order_status가 갱신될 때 40바이트만 로그에 쓰여집니다. 또한 페이지 크기가 2K인 Informix OnLine Dynamic Server 시스템에서, 레코드 크기가 300 바이트일 때는 페이지당 6개의 행 밖에 들어갈 수 없었지만 order_status 테이블에는 페이지당 84 행이 들어갑니다. 따라서 더 많은 행이 캐쉬될 수 있고 그러므로, 성능 또한 향상됩니다.



Fennimore Figure 05

그림 5: 와이드 테이블을 저장 정보 형식에 따라 나누기

긴 테이블

긴 테이블(long table)이란 행이 많은 테이블을 의미합니다. 일반적으로 시스템에는 몇 년 동안의 자료가 들어 있지만 검색하는 것은 그날그날 필요한 최근 자료 뿐입니다. 환자 정보 시스템이 이런 예입니다. 외래 환자에 대한 자료는 몇년씩 보존되지만 일상적인 보고에 필요한 내용은 지난 석달 동안의 자료에 대해 이루어지는 것이 일반적입니다. 그 이전의 자료는 데이터베이스의 일상적 작업에 부정적인 영향을 미친다고 볼 수 있습니다.

분할은 Informix OnLine Dynamic Server, 7.10 이후 버전에서 사용할 수 있습니다. 테이블은 표현식이나 라운드 로빈 방법으로 분할할 수 있습니다. 위의 예에서는 날짜 칼럼을 가지고, 최근 석달 부분과 그 이전 내용으로 테이블을 분할할 수 있습니다. 이 방법을 사용하면 테이블을 하나로 유지할 수 있습니다. 그러나, 최근 석달 동안에 날짜 기준으로 질의를 실행하면 옵티마이저가 최근 석달 동안의 분할 영역만 검색하고 다른 분할 영역은 건너뜁니다.

결론

이 글에서는 스키마 디자인과 관련된 가장 일반적인 문제들을 다루었습니다. 질의 최적화와 마찬가지로 스키마 최적화의 목표는 I/O와 처리 시간을 줄이는 것입니다. 적절한 자료형과 알맞은 외부 키를 사용하고, 데이터베이스를 비정규화하며, 한 행의 사이즈가 크고 행의 수가 많은 테이블을 분할하면 질의의 I/O와 처리 시간이 크게 줄어 성능이 향상될 것입니다.