Microsoft®
SQL Server™ 2005

SQL Server 2005 의 데이터베이스 동시 작업 및 행 수준 버전 관리

Microsoft®

본 문서는 예비 문서이며 여기에서 설명된 소프트웨어의 최종 상용 버전이 출시되기 전에 상당 부분 변경될 수 있습니다.
이 문서에 포함된 정보는 문서 발행 시에 논의된 문제들에 대한 Microsoft Corporation의 당시 관점을 나타냅니다. Microsoft는 변화하는 시장 상황에 부응해야 하므로 이를 Microsoft 측의 공약으로 해석해서는 안되며 발행일 이후 소개된 어떠한 정보에 대해서도 Microsoft는 그 정확성을 보증하지 않습니다.
이 문서는 오직 정보를 제공하기 위한 것입니다. Microsoft는 이 설명서에서 어떠한 명시적이거나 묵시적인 보증도 하지 않습니다.
해당 저작권법을 준수하는 것은 사용자의 책임입니다. 저작권에서의 권리와는 별도로, 이 설명서의 어떠한 부분도 Microsoft의 명시적인 서면 승인 없이는 어떠한 형식이나 수단(전기적, 기계적, 복사기에 의한 복사, 디스크 복사 또는 다른 방법) 또는 목적으로도 복제 되거나, 검색 시스템에 저장 또는 도입 되거나, 전송될 수 없습니다.
Microsoft가 이 설명서 본안에 관련된 특허권, 상표권, 저작권 또는 기타 지적 재산권 등을 보유할 수도 있습니다. 서면 사용권 계약에 따라 Microsoft로부터 귀하에게 명시적으로 제공된 권리 이외에, 이 설명서의 제공은 귀하에게 이러한 특허권, 상표권, 저작권 또는 기타 지적 재산권 등에 대한 어떠한 사용권도 허여하지 않습니다.
특별한 언급이 없는 한, 용례에 사용된 회사, 기관, 제품, 도메인 이름, 전자 메일 주소, 로고, 사람, 장소, 이벤트 등은 실제 데이터가 아닙니다. 어떠한 실제 회사, 기관, 제품, 도메인 이름, 전자 메일 주소, 로고, 사람, 장소 또는 이벤트와도 연관시

ⓒ 2005 Microsoft Corporation, 전권 보유.
Microsoft와 ActiveX는 미국, 대한민국 및/또는 기타 국가에서 Microsoft 의 등록 상표 또는 상표입니다. 여기에 인용된 실제 회사와 제품 이름은 해당 소유자의 상표일 수 있습니다.

Contents

데이터베이스 동시 작업 개요

현대의 관계형 데이터베이스 시스템을 사용하면 데이터에 수 천은 아니라도 수 백의 동시 연결이 가능합니다. 데이터베이스 시스템 자체의 아키텍처는 동일한 데이터에 대한 동시 액세스를 사용자나 연결 사이에 가능하면 방해가 적도록 관리할 수 있는 다양한 방법을 결정합니다. 대부분의 데이터베이스 시스템은 응용 프로그램 개발자가 동시 액세스를 관리하는 방법을 어느 정도 제어하는 데 사용할 수 있는 기능을 제공하여 개발자는 동시 작업과 데이터 일관성 사이에 균형을 찾을 수 있도록 합니다.

Microsoft® SQL Server™ 2005에는 새로운 방식으로 동시 액세스를 처리할 수 있는 행 수준 버전 관리(Row Level Versioning)라는 신기술이 포함되어 있습니다. SQL Server 2005의 많은 기능은 RLV로 설계되었으며 이 새 기능을 이용하기 위해 추가 응용 프로그램 제어는 필요하지 않습니다.
새로운 격리 수준 같은 다른 기능의 경우 데이터베이스 관리자가 데이터베이스별로 데이터베이스에서 특별히 RLV를 허용해야 합니다. 따라서 이전 SQL Server 버전의 잠금 동작에 의존하는 응용 프로그램은 이전 버전과의 호환성을 유지할 수 있습니다.

이 백서는 SQL Server 2005의 동시 작업 향상을 중점적으로 다룹니다. 서버측에서는 RLV 기술을 이용하는 SQL Server의 모든 기능을 다룹니다. 여기에는 스냅샷 격리, MARS(Multiple Active Result Sets) 및 온라인 인덱스 재구축과 같은 새로운 기능이 포함됩니다. RLV는 SQL Server 2005에서 데이터베이스 트리거를 지원하는 데도 사용되므로 SQL Server 2000과 2005 간에 트리거 동작의 차이점도 논의됩니다. 클라이언트측의 동시 작업 향상은 CLR 개체의 동시 작업, 새로운 SQL Native Client의 트랜잭션 제어, Windows Enterprise 서비스와 대기열 구성 요소 및 Service Broker 지원 응용 프로그램을 사용한 동시 작업을 포함합니다.

RLV와 클라이언트측 향상의 주요 이점 중 하나는 SQL Server가 이제 데이터 동시 작업과 같거나 더 나은 높은 수준의 데이터베이스 동시 작업을 제공할 수 있다는 점입니다. 이 백서는 새 기능을 기존 기능과 비교하기 위해서만 기존의 동시 작업 기능을 설명할 것입니다.

데이터베이스 동시 작업 정의

동시 작업은 여러 프로세스가 동시에 공유되는 데이터에 액세스하거나 변경하는 능력으로 정의할 수 있습니다. 서로를 방해하지 않고 실행할 수 있는 동시 사용자 프로세스 수가 많을 수록 데이터베이스 시스템의 동시 작업은 더욱 커집니다.

데이터를 변경하는 프로세스가 변경 중인 데이터를 다른 프로세스가 읽지 못하게 하거나 데이터를 읽고 있는 프로세스가 해당 데이터를 다른 프로세스가 변경하지 못하도록 할 때 동시 작업은 영향을 받습니다. 동시 작업은 여러 프로세스가 동일한 데이터를 동시에 변경하려고 시도할 때도 영향을 받으며 데이터 동시 작업을 희생하지 않고는 이러한 시도는 모두 성공할 수 없습니다.

데이터베이스 시스템이 동시 작업을 줄이는 상황을 해결하는 방법은 시스템이 낙관적 또는 비관적 동시 작업 제어를 사용하는지 여부에 의해 부분적으로 결정됩니다. 비관적 동시 작업 제어는 시스템에 충분한 데이터 수정 작업이 있어 읽기 작업이 다른 사용자의 데이터 수정에 의해 영향을 받을 가능성이 있다는 가정 하에서 작동합니다. 즉, 시스템은 비관적이 되며 충돌이 발생할 것이라고 가정합니다. 비관적 동시 작업 제어를 사용할 때의 기본 동작은 잠금을 사용하여 다른 사용자가 사용 중인 데이터에 대한 액세스를 차단하는 것입니다. 낙관적 동시 작업 제어는 데이터 수정 작업이 충분히 적어 모든 프로세스가 다른 프로세스가 읽고 있는 데이터를 수정할 가능성이 없다는(수정은 가능함) 가정 하에서 작동합니다. 낙관적 동시 작업 제어를 사용할 때의 기본 동작은 행 버전 제어를 사용하여 데이터를 읽는 사용자가 수정이 발생

이전의 실행 방법

지금까지 서버 수준에서 SQL Server의 동시 작업 제어 모델은 비관적 동시 작업의 잠금을 기반으로 했습니다. 잠금이 여전히 대부분의 응용 프로그램에 대해 동시 작업 제어를 위한 최상의 선택이지만 작은 응용 프로그램인 경우에는 상당한 차단 문제를 야기할 수 있습니다.

가장 큰 문제는 잠금이 writer-block-reader 또는 reader-block-writer 문제를 일으킬 때 발생합니다. 트랜잭션이 행을 변경하는 경우 변경된 데이터에 독점적 잠금을 유지합니다. SQL Server의 기본 동작은 쓰는 사용자가 커밋할 때까지 다른 트랜잭션이 행을 읽을 수 없는 것입니다. 또한 SQL Server는 연결의 격리 수준을 설정하거나 NOLOCK 테이블 힌트를 지정하여 응용 프로그램에서 요청할 수 있는 'read uncommitted isolation' 을 지원합니다. 이 비잠금 스캔은 트랜잭션이 일관성 있는 결과를 반환한다는 보장이 없기 때문에 항상 사용하기 전에 신중히 고려해야 합니다.

SQL Server 2005 이전의 동시 작업 솔루션에서는 일관성 없는 데이터의 위험을 감수하고 쓰는 사용자가 읽는 사용자를 차단할 수 없도록 했습니다. 결과가 항상 커밋된 데이터를 기준으로 해야 하는 경우에는 변경 내용이 커밋될 때까지 기꺼이 기다려야 했습니다.

행 수준 버전 제어의 개요

응용 프로그램이 결과가 항상 커밋된 데이터를 기준으로 해야 한다는 것을 요구하는 경우에도 두 가지 가능성이 있습니다.
읽는 사용자가 데이터의 최근 커밋된 값을 반드시 가져야 하는 경우 쓰는 사용자가 트랜잭션을 완료하고 변경 내용을 커밋할 때까지 읽는 사용자는 반드시 잠금 상태에서 기다려야 합니다. 다른 상황에서는 가장 최신 버전이 아닌 경우에도 충분히 커밋된 데이터 값을 가질 수 있습니다. 이 경우 SQL Server가 행의 이전에 커밋된 값, 즉 이전 버전을 제공할 수 있는 경우 읽는 사용자는 문제가 없을 수 있습니다.

SQL Server 2005는 'read committed snapshot isolation' (RCSI)라고 하는 읽기 커밋 격리의 의미를 가진 '스냅샷 격리' 및 새로운 비차단을 소개합니다. 이러한 행 버전 관리 기반의 격리 수준을 사용하면 읽는 사용자는 차단 없이 행의 이전에 커밋된 값을 가져올 수 있으므로 시스템의 동시 작업은 증가됩니다. 이를 위해서는 SQL Server가 행을 업데이트할 때 행의 이전 버전을 보관해야 합니다. 동일한 행의 여러 이전 버전을 유지해야 할 수 있기 때문에 이 새로운 동작을 다중 버전 동시 작업 제어 또는 행 수준 버전 관리라고도 합니다.

행의 여러 이전 버전 저장을 지원하기 위해 tempdb 데이터베이스에서 추가 디스크 공간이 사용됩니다. 버전 저장을 위한 디스크 공간은 적절히 모니터링하고 관리해야 합니다.
버전 관리는 데이터를 변경하는 트랜잭션이 데이터의 이전 버전을 보관하는 방법을 통해 작동하므로 데이터베이스의 '스냅샷' (또는 데이터베이스의 일부)을 이러한 이전 버전에서 구성할 수 있습니다.

테이블이나 색인의 레코드가 업데이트되면 새 레코드에 업데이트를 수행 중인 트랜잭션의 sequence_number가 스탬프됩니다. 레코드의 이전 버전은 버전 저장소에 저장되며 새 레코드는 버전 저장소에 있는 이전 레코드에 대한 포인터를 포함합니다. 버전 저장소의 기존 레코드는 더 오래된 버전의 포인터를 포함할 수도 있습니다. 특정 레코드의 모든 이전 버전은 링크된 목록에 체인으로 연결되며 SQL Server는 올바른 버전에 도달하려면 목록의 여러 포인터를 따라가야 할 수 있습니다. 버전 레코드는 버전 레코드를 필요로 할 수 있는 작업이 있는 경우에만 버전 저장소에 보관해야 합니다.

다음 그림에서 레코드의 최신 버전은 트랜잭션 T3에 의해 생성되고 일반 데이터 페이지에 저장됩니다. 트랜잭션 T2와 트랜잭션 Tx 에 의해 생성되는 레코드의 이전 버전은 버전 저장소(tempdb에 있음)의 페이지에 저장됩니다.

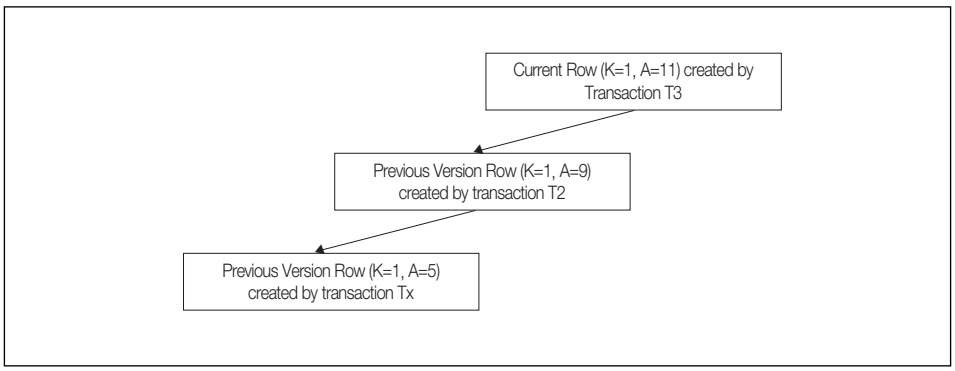

그림 1: 레코드의 버전

행 수준 버전 제어는 응용 프로그램이 이를 요구하거나 기본 비관적 모델을 사용한 동시 작업 감소를 허용할 수 없을 때 SQL Server가 낙관적 동시 작업 모델을 사용할 수 있도록 합니다. 행 버전 관리 기반 격리 수준으로 전환하려면 이 새 동시 작업 모델 사용의 장단점을 신중하게 고려해야 합니다. 버전 저장소로 tempdb의 증가된 사용을 모니터링하기 위한 추가 관리 요구 사항 이외에 버전 관리를 사용하면 이전 버전의 유지 관리에 필요한 추가 작업으로 인해 업데이트 작업의 성능이 느려집니다. 현재 데이터를 읽는 사용자가 없는 경우에도 업데이트 작업은 이 비용을 떠 안아야 합니다. 행 수준 버전 관리를 사용하여 읽는 사용자가 있는 경우 요청한 행의 적절한 버전을 찾기 위해 링크 포인터를 이동하는 추가 비용이 있습니다.

또한 스냅샷 격리의 낙관적 동시 작업 모델은 많은 업데이트 충돌이 발생하지 않을 것으로 낙관적으로 가정하기 때문에 동일한 데이터를 동시에 업데이트하기 위한 경합이 예상되는 경우 SI 격리 수준을 선택해서는 안 됩니다. 스냅샷 격리는 읽는 사용자가 쓰는 사용자에 의해 차단되지 않도록 잘 작동하지만 동시 쓰는 사용자는 여전히 허용되지 않습니다. 기본 비관적 모델에서 첫 번째 쓰는 사용자는 이후의 모든 쓰는 사용자를 차단하지만 SI를 사용하면 이후의 쓰는 사용자에게 실제 오류 메시지가 나타날 수 있으며 응용 프로그램은 원래 요청을 다시 제출해야 합니다. 이러한 업데이트 충돌은 SI에서만 발생하며 향상된 읽기-커밋 격리 수준, RCSI에서는 발생하지 않습니다. 또한 스냅샷 격리를 사용할 때 업데이트 충돌을 줄이기 위해 따를 수 있는 지침이 있습니다.

행 수준 버전 관리를 사용하여 스냅샷 격리를 지원하는 방법에 대한 자세한 내용은 Kimberly Tripp의 백서를 참조하십시오. SQL Server 2005 베타 2 스냅샷 격리, http://www.microsoft.com/technet/prodtechnol/sql/2005/SQL05B.mspx

행 수준 버전 관리를 이용하는 추가 SQL Server 2005 기능

행 수준 버전 관리를 SQL Server 2005에 추가하는 이유가 낙관적인 동시 작업을 위한 버전 저장소를 유지하고 행 버전 관리 기반의 격리 수준을 지원하여 데이터를 쓰는 사용자가 모든 읽는 사용자를 차단하는 문제를 해결하는 것이지만 이 새로운 데이터 관리 메커니즘을 이용하는 다른 SQL Server 2005 기능이 있습니다. 이러한 두 가지 기능, MARS(multiple active result sets)와 온라인 색인 재구축은 제품에 새로 소개된 기능이며 세 번째 기능은 기존 기능인 트리거를 관리하는 새로운 방법입니다. 이 절에서는 이러한 SQL Server 기능의 행 수준 버전 관리의 사용에 대한 간단한 설명을 제공합니다.

트리거 및 행 수준 버전 관리

트리거는 최초 버전 이후로 SQL Server의 일부였으며 기록/버전 데이터 유형을 제공한 SQL Server 2005 이전에는 제품의 유일한 기능이었습니다. 트리거의 특별한 기능 중 하나는 'deleted' 라는 의사 테이블에 액세스하는 능력입니다. 트리거가 DELETE 트리거인 경우 'deleted' 테이블에는 트리거를 실행하도록 한 작업에 의해 삭제된 모든 행이 포함되어 있습니다. 트리거가 UPDATE 트리거인 경우 'deleted' 테이블에는 트리거를 실행하도록 한 업데이트 명령문에 의해 변경된 모든 행에 있는 데이터의 이전 버전 즉, 업데이트가 발생하기 전의 데이터가 포함되어 있습니다. SQL Server의 이전 버전은 트리거가 연결된 테이블을 변경한 현재 트랜잭션의 모든 로그 레코드를 조회하는 트랜잭션 로그를 스캔하여 삭제된 테이블을 채웁니다. 로그 레코드 스캔은 트랜잭션 로그가 읽기가 아닌 쓰기에 최적화되어 있으므로 매우 비용이 많이 들 수 있습니다. 현재 트랜잭션의 로그 레코드가 이미 디스크에 기록되었을 수 있는 볼륨이 큰 OLTP 시스템의 경우 이는 실제 물리적 I/O 작업을 초래할 수 있습니다. 새 메커니즘은 기존 트리거의 성능에 도움을 줄 수 있습니다.

SQL Server 2005에서 삭제된 테이블은 행 수준 버전 관리를 사용하여 구체화됩니다. 관련 트리거가 정의된 테이블에서 업데이트나 삭제가 수행되면 행 버전 관리 기반의 격리 수준 사용 여부에 관계 없이 테이블 변경 내용이 버전 관리됩니다. 트리거가 'deleted' ' 테이블에 액세스해야 하는 경우 버전 저장소에서 데이터를 검색합니다. 업데이트 또는 삽입한 새 데이터는 'inserted' 테이블을 통해 액세스할 수 있습니다. SQL Server 2005 트리거가 inserted를 스캔할 때 행의 가장 최신 버전을 찾습니다.

tempdb가 버전 저장소에 사용된다는 사실 때문에 SQL Server 2000에서 트리거를 많이 사용하는 응용 프로그램은 SQL Server 2005로 업그레이드한 후에 tempdb에 대한 수요가 증가할 수 있다는 것을 인식해야 합니다.

온라인 색인 생성 및 행 수준 버전 관리

색인 생성과 재구축은 사실 SQL Server의 새 기능은 아니지만 SQL Server 2005에서는 이제 테이블이나 색인을 오프라인으로 만들지 않고도 색인을 구축 또는 재구축할 수 있습니다. 이전 버전에서는 클러스터된 색인의 구축 또는 재구축은 모든 데이터를 완전히 액세스할 수 없도록 전체 테이블을 독점적으로 잠궜습니다. 클러스터되지 않은 색인의 구축 또는 재구축은 테이블에 공유 잠금을 배치하므로 데이터를 읽을 수 있지만 수정할 수는 없습니다. 또한 클러스터되지 않은 색인을 재구축하는 동안 색인 자체를 완전히 사용할 수 없었으며 인덱스를 사용한 쿼리는 성능이 저하될 수 있습니다.

행 수준 버전 관리를 사용하면 SQL Server 2005에서 색인을 완전히 온라인으로 구축 또는 재구축할 수 있습니다. 색인이 구축되면 SQL Server는 색인 구축을 시작한 당시의 데이터 버전을 기존 테이블에서 스캔합니다. 스냅숏 격리를 사용하는지 여부에 관계 없이 테이블 수정은 버전 관리됩니다. 테이블에서 데이터를 읽으려는 요청은 버전 관리되는 데이터에 액세스합니다.

Multiple Active Result Sets 및 행 수준 버전 관리

MARS(Multiple Active Result Sets)는 SQL Server 2005의 클라이언트측 기능이지만 구현은 거의 서버측 기능인 버전 저장소를 사용합니다. 이런 이유로 MARS에 대한 설명은 다른 행 수준 버전 관리 기능에 포함되며 클라이언트측 동시 작업 고려 사항은 아닙니다.

Microsoft SQL Server 2005는 데이터베이스 엔진에 액세스하는 응용 프로그램에서 MARS에 대한 지원을 확장합니다. 이전 버전의 SQL Server에서는 데이터베이스 응용 프로그램이 기본 결과 집합을 사용할 때 연결에서 여러 활성 명령문을 유지할 수 없습니다. 응용 프로그램은 해당 연결에서 다른 배치를 실행하기 전에 한 배치에서 모든 결과 집합을 처리하거나 취소해야 합니다. SQL Server 2005는 응용 프로그램이 연결 당 둘 이상의 보류 중인 요청, 특히 연결 당 둘 이상의 활성 기본 결과 집합을 가질 수 있는 새로운 특성을 선보입니다.

SELECT 및 BULK INSERT 명령문만 비원자적으로 실행하는 것이 허용되며 MARS 지원 연결에서 다른 배치를 실행하는 다른 명령문과 인터리브됩니다. DDL과 데이터 수정 명령문은 모두 원자적으로 실행됩니다. 실행을 기다리는 다른 명령문이 있는 경우 원자 명령문이 실행을 완료할 때까지 차단됩니다.

두 배치가 MARS 연결에 제출되고, 그 중 하나에는 SELECT 명령문이 포함되어 있고 다른 하나에는 UPDATE 명령문이 포함되어 있는 경우 SELECT가 전체 결과를 처리하기 전에 UPDATE가 실행을 시작할 수 있습니다. 그러나 SELECT 명령문을 실행하려면 UPDATE 명령문의 실행을 완료해야 하며 UPDATE에 의한 모든 변경이 버전 관리됩니다. 두 명령문 모두 동일한 트랜잭션을 실행하는 경우 SELECT는 필요한 데이터 행의 이전 버전에 액세스하기 때문에 SELECT 명령문이 실행된 후 UPDATE 명령문에 의한 변경을 SELECT 명령문에서 볼 수 없습니다.

동시 작업과 관련된 클라이언트측 문제

SQL Server 2005의 저장소 엔진은 트랜잭션과 잠금을 관리하는 동시 작업 제어를 제공합니다. 그러나 사용자는 클라이언트 응용 프로그램, 구성 요소 및 서비스를 통해 SQL Server에 액세스하며 이러한 모든 프로그래밍 구성 요소는 SQL Server가 동시 작업을 관리하는 방식에 영향을 받습니다. 또한 연결 설정이 동시 작업에 어떤 영향을 미치며 클라이언트 응용 프로그램에서 트랜잭션을 효과적으로 관리하는 방법을 이해하는 것도 중요합니다.

SQL Server의 이전 버전에서 SQL Server에 의해 실제로 실행된 명령은 어느 응용 프로그램 레이어가 특정 트랜잭션을 시작했는지에 관계 없이 Transact-SQL 명령이 되어야 합니다.

SQL Server 2005의 새로운 SQL CLR 기능을 사용하면 Service Broker 같은 기능을 포함하여 새로운 서버측 프로그래밍 인프라와 새로운 데이터 액세스 공급자, 동시 작업 및 트랜잭션 관리가 훨씬 강력해집니다. 그러나 강력해지기는 했지만 더 복잡해졌습니다.

SQL CLR 개체의 동시 작업 제어

SQL CLR 프로시저 내에서 수행한 작업은 DTC-관리 트랜잭션을 시작할 필요가 없으며 SQL Server 데이터를 업데이트, 삽입 또는 삭제하는 작업은 표준 Transact-SQL 프로시저와 같은 트랜잭션 원칙을 따릅니다.

SQL CLR 개체는 SqlContext 개체를 통해 기본 제공되는 데이터 액세스 공급자를 사용하여, SqlContext 개체를 통해 SQL Server 데이터를 사용할 수 있으며 이러한 SQL CLR 개체는 다음 예에서와 같이 SqlContext 개체의 Pipe 속성을 사용하여 결과를 호출하는 연결로 보낼 수 있습니다.

```
〈SQLProcedure(  )〉 _
Public Shared Sub GetServerVersion (  )
    Dim cmdGetVersion as SqlCommand = SqlContext.CreateCommand(  )

    cmdGetVersion.Commandtext = "SELECT @@VERSION"
    cmdGetVersion.CommandType = CommandType.Text

    SqlContext.Pipe.Send(cmdGetVersion.ExecuteScalar(  ).ToString(  ))
End Sub
```

매우 간단해 보이지만 실제로 뒤에서 어떤 일이 이루어지는 것일까요? 위의 명령을 실행할 때 실제로 어떤 일이 발생하는지 보기 위해 추적을 설정하면 다음 이벤트가 나타납니다.

```
SQL:BatchStarting       EXEC cbo.GetServerVersion
SQL:StmtStarting        EXEC cbo.GetServerVersion
SP:Starting             EXEC cbo.GetServerVersion
SP:StmtStarting         SELECT @@VERSION
```

이것은 다음 Transact-SQL 저장 프로시저를 정의하고 실행한 경우 실제로 표시되는 것과 정확히 동일한 이벤트입니다.

```
CREATE PROCEDURE dbo.GetServerVersion
AS
    SELECT @@VERSION
GO
```

이제 AdventureWorks 데이터베이스에서 Production.Product 테이블의 데이터를 업데이트하는 CLR 저장 프로시저를 만들어 봅시다.

```
〈SqlProcedure(  )〉 _
Public Shared Sub UpdateListPriceByProductID(ByVal ProductID As SqlInt32)
Try
Dim cmdUpdate As SqlCommand = SqlContext.CreateCommand(  )
Dim parProductID As SqlParameter = _
cmdUpdate.Parameters.Add("@ProductID" , SqlDbType.Int)

parProductID.Direction = ParameterDirection.Input
cmdUpdate.CommandText = "UPDATE Production.Product " _
+ "SET ListPrice = ListPrice * 1.1 " _
+ "WHERE ProductID = @ProductID"

parProductID.Value = ProductID

cmdUpdate.ExecuteNonQuery(  )

Catch e As Exception
SqlContext.Pipe.Send(e.Message)
End Try
End Sub
```

이 저장 프로시저를 실행할 때 SQL Server에서 실제로 어떤 활동이 일어나는지를 확인하기 위해 다시 추적을 사용할 수 있습니다. 다음 표는 서버에서 발생한 이벤트를 나열한 것이며 다음 설명에서 특정 이벤트를 참조할 수 있도록 이벤트 번호를 포함하고 있습니다.

1	SQL:BatchStarting	EXECUTE dbo.UpdateListPriceByProductID 444; Management Studio에서 실행하는 배치가 실행을 시작합니다.
2	SQL:StmtStarting	EXECUTE dbo.UpdateListPriceByProductID 444; 이 배치의 유일한 명령문이 실행을 시작합니다.
3	SP:Starting	EXECUTE dbo.UpdateListPriceByProductID 444; 이 명령문은 dbo.UpdateListPriceByProductID CLR 저장 프로시저를 호출하며, 실행을 시작합니다.
4	SP:StmtStarting	UPDATE Production.Product SET ListPrice = ListPrice * 1.1 WHERE ProductID = @ProductID
		이것은 이 저장 프로시저에서 데이터베이스 작업을 실행하는 유일한 명령문입니다. 이 프로시저가 SQLContext 개체를 사용하여 현재 연결 컨텍스트에 대한 참조를 얻기 때문에 연결 이벤트는 없습니다.
5	SQLTransaction	75691 UPDATE 0 - Begin 실행할 명령문이 데이터 수정 작업이기 때문에 SQL Server는 자동으로 암시적 트랜잭션을 시작합니다.
6	SP:Starting	UPDATE Production.Product SET ListPrice = ListPrice * 1.1 WHERE ProductID = @ProductID

		uProduct라는 Production.Product 테이블에서 정의된 AFTER UPDATE 트리거가 있습니다. 이 트리거는 저장 프로시저와 마찬가지로 내부적으로 관리되기 때문에 SP:Starting 이벤트가 나타납니다.
		이 트리거를 실행하면 트랜잭션 번호 75691의 제어를 통해 발생합니다.
		다음은 이 트리거를 만드는 코드입니다.
		
		CREATE TRIGGER [Production].[uProduct]
		ON [Production].[Product]
		AFTER UPDATE NOT FOR REPLICATION AS
		BEGIN
		SET NOCOUNT ON;
		
		UPDATE [Production].[Product]
		SET [Production].[Product].[ModifiedDate]
		= GETDATE( )
		FROM inserted
		WHERE inserted.[ProductID] =
		[Production].[Product].[ProductID];
		END;

7	SP:StmtStarting	SET NOCOUNT ON;
		트리거 내부의 첫 번째 명령문이 시작됩니다.
8	SP:StmtCompleted	SET NOCOUNT ON;
		트리거 내부의 첫 번째 명령문이 완료됩니다.
9	SP:StmtStarting	UPDATE [Production].[Product] SET [Production].[Product].[ModifiedDate] = GETDATE( ) FROM inserted WHERE inserted.[ProductID] = [Production].[Product].[ProductID];
		트리거 내부의 두 번째 명령문이 시작됩니다.
10	SP:StmtCompleted	UPDATE [Production].[Product] SET [Production].[Product].[ModifiedDate] = GETDATE( ) FROM inserted WHERE inserted.[ProductID] = [Production].[Product].[ProductID];
		트리거 내부의 두 번째 명령문이 완료됩니다.
11	SP:Completed	UPDATE Production.Product SET ListPrice = ListPrice * 1.1 WHERE ProductID = @ProductID
		uProduct 트리거의 실행이 완료됩니다.
12	SP:StmtCompleted	UPDATE Production.Product SET ListPrice = ListPrice * 1.1 WHERE ProductID = @ProductID
		4단계에서 시작된 원래의 UPDATE 문의 실행이 완료됩니다.
13	SQLTransaction	75691 UPDATE 1 - Commit
		이 명령문이 데이터베이스에 써야 할 때 이 트랜잭션이 자동으로 열렸기 때문에 트랜잭션 75691은 명령문이 완료되면 자동으로 커밋됩니다.
14	SP:Completed	EXECUTE [dbo].[UpdateListPriceByProductID] 444;
		3단계에서 시작된 저장 프로시저가 완료됩니다.
15	SQL:StmtCompleted	EXECUTE [dbo].[UpdateListPriceByProductID] 444;
		2단계에서 시작된 명령문이 완료됩니다.
16	SQL:BatchCompleted	EXECUTE [dbo].[UpdateListPriceByProductID] 444;
		The batch started on step 1 completes.

추적 이벤트는 실제로 실행된 것이 CLR 코드인지 알려주지 못합니다. CLR 저장 프로시저가 Transact-SQL 트리거를 강제로 시작했다는 사실은 트랜잭션 관점에서 고려 사항이 되지 않습니다.

위의 추적 정보는 SqlContext 개체가 사용되는 한 SQL CLR 저장 프로시저 내에서 보낸 코드 실행이 이 프로시저를 호출하는 연결과 동일한 SPID 하에서 실행되는 것을 보여줍니다.

SQL CLR 개체가 .NET 데이터 공급자를 사용하여 외부 데이터베이스 시스템에 액세스하면 마치 표준 .NET 응용 프로그램에서 연결된 것처럼 동작합니다. 트랜잭션 관점에서 보면 차이가 없을 것입니다. 다음 절에서는 ADO.NET 2.0의 동시 작업 관리를 설명합니다.

SQL CLR 개체는 항상 인프로세스(in-process) 데이터 공급자를 통해 SQL Server의 현재 인스턴스에 연결되어야 합니다.

ADO.NET 2.0의 동시 관리

ADO.NET은 데이터베이스 응용 프로그램과 구성 요소를 만들기 위해 두 가지 다른 프로그래밍 모델을 제공함으로써 이전 클라이언트 데이터 액세스 패러다임을 깼습니다.

낙관적 동시 작업을 기반으로 한 연결이 끊긴 모델은 DataSet 및 DataAdapter 유형에 구축되며
비관적 동시 작업을 기반으로 한 연결된 모델은 SQLCommand 및 SQLDataReader 유형에 구축됩니다.

연결이 끊긴 모델을 사용하는 응용 프로그램은 SqlDataAdapter 또는 TableAdapter를 사용하여 데이터베이스에서 요청된 데이터를 읽고, 이 작업 동안 충분한 공유 잠금을 사용하여 데이터 행을 일관성 있게 읽을 수 있도록 한 다음 데이터베이스에서 연결을 끊어 이 읽기 작업이 갖고 있을 수 있는 모든 잠금을 해제합니다. 여기에서 트랜잭션 동작은 ADO.NET의 이전 릴리스와 거의 동일하며 MSDN에서 이 항목에 사용할 수 있는 포괄적인 공개 정보를 참조할 수 있습니다.

ADO.NET 2.0은 SQL Server의 새 버전에서 사용할 수 있는 새로운 기능을 제시하는 향상된 SQL Server .NET 데이터 공급자를 제공합니다. 이 새 공급자가 제시하는 MARS 및 새 스냅샷 격리 수준 같은 몇 가지 새로운 기능은 이 문서 앞부분에서 설명했습니다.

트랜잭션 동작을 변경할 수 있는 또 다른 ADO.NET 향상은 명령을 비동기적으로 실행할 수 있다는 것입니다. 이는 SQL Server의 새로운 MARS 기능을 사용합니다. 그러나 ADO.NET과 관련하여 특별한 사항은 없으며 동작은 이 문서 앞 부분에서 설명한 것과 정확히 같습니다.

.NET Framework Version 2.0은 LTM(Lightweight Transaction Manager)과 OleTx Transaction Manager 등 두 가지 새로운 트랜잭션 관리자를 제공합니다. 이러한 두 트랜잭션 관리자에 대한 액세스는 System.Transactions 네임스페이스를 통해 캡슐화됩니다.

ADO.NET 2.0은 System.Transactions를 자동으로 이용하도록 다시 디자인되었습니다. 즉, ADO.NET 2.0에서 엔터프라이즈 서비스를 사용하는 코드는 필요할 때 LTM 또는 OleTx를 사용합니다. 이런 식으로 응용 프로그램은 선언적인 방식으로 이 트랜잭션 모델을 사용하게 됩니다.

이 새로운 방법을 보여주기 위해 System.Transactions를 사용하고 사용하지 않는 동일한 예제 응용 프로그램을 실행할 것입니다.

이 첫 번째 예제는 명시적 트랜잭션 컨트롤을 사용하지 않고 SQLCommand 개체를 통해 SQL 명령문을 실행합니다.

```
Private Sub TestSystemTransactions(  )
  Dim conAW As New SqlConnection(sConnString)
  Dim sQuery1 As String, count1 As Integer
  Dim cmd1 As SqlCommand = conAW.CreateCommand(  )

  sQuery1 = "UPDATE Production.Product " _
    + "SET ListPrice = ListPrice * 1.1 " _
    + "WHERE ProductNumber LIKE 'EC%' "

  cmd1.CommandText = sQuery1

    Try
      conAW.Open(  )
      Dim result1 As IAsyncResult = cmd1.BeginExecuteNonQuery(  )

      While result1.IsCompleted = False
        Console.WriteLine("Waiting ({0})", count1)
         ' Wait for 1/10 second, so the counter
         ' doesn't consume all available resources
         ' on the main thread.
        Threading.Thread.Sleep(100)
        If result1.IsCompleted = False Then count1 += 1
      End While

      Console.WriteLine("Command complete. Affected {0} rows.", _
        cmd1.EndExecuteNonQuery(result1))

    Catch ex As SqlException
      Console.WriteLine("Error ({0}): {1}", ex.Number, ex.Message)
    Catch ex As InvalidOperationException
      Console.WriteLine("Error: {0}", ex.Message)
    Catch ex As Exception
      Console.WriteLine("Error: {0}", ex.Message)
    Finally
      conAW.Close(  )
      Console.ReadLine(  )
    End Try
End Sub
```

프로파일러를 사용하여 이 코드가 SQL Server에서 생성하는 작업 시퀀스를 검토할 수 있습니다.

1	SQL:BatchStarting	UPDATE Production.Product SET ListPrice = ListPrice * 1.1 WHERE ProductNumber LIKE 'EC%'
		cmd1 SqlCommand 객체를 통해 VB 콘솔 응용 프로그램에서 실행하는 배치.
2	SQL:StmtStarting	UPDATE Production.Product SET ListPrice = ListPrice * 1.1 WHERE ProductNumber LIKE 'EC%'
		1단계와 동일.
3	SQLTransaction	162198 UPDATE 0 - Begin
		실행할 명령문이 데이터 수정 작업이기 때문에 SQL Server는 자동으로 암시적 트랜잭션을 시작합니다.
4	SP:Starting	UPDATE Production.Product SET ListPrice = ListPrice * 1.1 WHERE ProductNumber LIKE 'EC%'
		uProduct라는 Production.Product 테이블에서 정의된 AFTER UPDATE 트리거가 있습니다. 이 트리거는 저장 프로시저와 마찬가지로 내부적으로 관리되기 때문에 SP:Starting 이벤트가 나타납니다. 이 트리거를 실행하면 트랜잭션 번호 162198의 제어를 통해 발생합니다. 이 문서 앞 부분에서 설명한 것과 같은 트리거입니다.
5	SP:StmtStarting	SET NOCOUNT ON;
		트리거 내부의 첫 번째 명령문이 시작됩니다.
6	SP:StmtCompleted	SET NOCOUNT ON;
		트리거 내부의 첫 번째 명령문이 완료됩니다.
7	SP:StmtStarting	UPDATE [Production].[Product] SET [Production].[Product].[ModifiedDate] = GETDATE( ) FROM inserted WHERE inserted.[ProductID] = [Production].[Product].[ProductID];
		트리거 내부의 두 번째 명령문이 시작됩니다.
8	SP:StmtCompleted	UPDATE [Production].[Product] SET [Production].[Product].[ModifiedDate] = GETDATE( ) FROM inserted WHERE inserted.[ProductID] = [Production].[Product].[ProductID];
		트리거 내부의 두 번째 명령문이 완료됩니다.

9	SP:Completed	UPDATE Production.Product SET ListPrice = ListPrice * 1.1 WHERE ProductID = @ProductID
		uProduct 트리거의 실행이 완료됩니다.
10	SP:StmtCompleted	UPDATE Production.Product SET ListPrice = ListPrice * 1.1 WHERE ProductNumber LIKE 'EC%'
		4단계에서 시작된 원래의 UPDATE 문의 실행이 완료됩니다.
11	SQLTransaction	162198 UPDATE 1 - Commit
		이 명령문이 데이터베이스에 써야 할 때 이 트랜잭션이 자동으로 열렸기 때문에 트랜잭션 162198은 명령문이 완료되면 자동으로 커밋됩니다.
12	SQL:BatchCompleted	UPDATE Production.Product SET ListPrice = ListPrice * 1.1 WHERE ProductNumber LIKE 'EC%'
		The batch started on step 1 completes.

이제 이 두 번째 예제는 명시적 트랜잭션 컨트롤을 사용하여 동일한 SQLCommand 개체를 통해 동일한 SQL 명령문을 실행합니다(코드의 새 줄은 굵게 표시되어 있음):

```
Private Sub TestSystemTransactions(  )
  Dim conAW As New SqlConnection(sConnString)
  Dim sQuery1 As String, count1 As Integer
  Dim cmd1 As SqlCommand = conAW.CreateCommand(  )

  sQuery1 = "UPDATE Production.Product " _
    + "SET ListPrice = ListPrice * 1.1 " _
    + "WHERE ProductNumber LIKE 'EC%' "

  cmd1.CommandText = sQuery1

  Dim myTSScope As New TransactionScope

  Using myTSScope
    Try
      conAW.Open(  )
      Dim result1 As IAsyncResult = cmd1.BeginExecuteNonQuery(  )

      While result1.IsCompleted = False
        Console.WriteLine("Waiting ({0})" , count1)
        ' Wait for 1/10 second, so the counter
```

```
            ' doesn't consume all available resources
            ' on the main thread.
            Threading.Thread.Sleep(100)
            If result1.IsCompleted = False Then count1 += 1
          End While

          myTSScope.Complete()

          Console.WriteLine("Command complete. Affected {0} rows.", _
            cmd1.EndExecuteNonQuery(result1))

        Catch ex As SqlException
          Console.WriteLine("Error ({0}): {1}", ex.Number, ex.Message)
        Catch ex As InvalidOperationException
          Console.WriteLine("Error: {0}", ex.Message)
        Catch ex As Exception
          Console.WriteLine("Error: {0}", ex.Message)
        Finally
          conAW.Close()
          Console.ReadLine()
        End Try
      End Using
    End Sub
```

프로파일러를 사용하여 이 코드가 SQL Server에서 생성하는 작업 시퀀스를 검토할 수 있습니다. 여기서 새 작업은 굵게 표시됩니다.

1	TM: Begin Tran starting	BEGIN TRANSACTION
		연결이 설정되면 트랜잭션 관리자는 코드가 TransactionScope 개체를 사용하기 때문에 트랜잭션을 만들려고 시도합니다.
2	SQLTransaction	164587 user_transaction 0 - Begin
		이것은 UPDATE 명령문이 아직 실행되지 않았기 때문에 앞의 예제처럼 UPDATE 명령문에 의해 트리거된 것이 아니라 트랜잭션 관리자가 시작한 트랜잭션입니다. 사실 이 SQLTransaction은 1단계에서 실행한 BEGIN TRANSACTION 명령문에 의해 생성된 것입니다.
3	TM: Begin Tran completed	BEGIN TRANSACTION
		사용자 트랜잭션 만들기는 이 연결에서 성공했으며 이 트랜잭션의 경우 XactSequence 번호를 얻습니다. 이 경우 XactSequence 번소는 2362232012810이며 이 트랜잭션 아래의 트랜잭션 관리자가 관리할 모든 작업을 식별합니다.
4	SQL:BatchStarting	UPDATE Production.Product SET ListPrice = ListPrice * 1.1 WHERE ProductNumber LIKE 'EC%'
		cmd1 SqlCommand 객체를 통해 VB 콘솔 응용 프로그램에서 실행하는 배치.
5	SQL:StmtStarting	UPDATE Production.Product SET ListPrice = ListPrice * 1.1 WHERE ProductNumber LIKE 'EC%'
		1단계와 동일.
6	SP:Starting	UPDATE Production.Product SET ListPrice = ListPrice * 1.1 WHERE ProductNumber LIKE 'EC%'
		uProduct라는 Production.Product 테이블에서 정의된 AFTER UPDATE 트리거가 있습니다. 이 트리거는 저장 프로시저와 마찬가지로 내부적으로 관리되기 때문에 SP:Starting 이벤트가 나타납니다. 이 트리거를 실행하면 XactSequence 번호 236223201281의 제어를 통해 발생합니다. 이 문서 앞 부분에서 설명한 것과 같은 트리거입니다.
7	SP:StmtStarting	SET NOCOUNT ON;
		트리거 내부의 첫 번째 명령문이 시작됩니다.
8	SP:StmtCompleted	SET NOCOUNT ON;
		트리거 내부의 첫 번째 명령문이 완료됩니다.

9	SP:StmtStarting	UPDATE [Production].[Product] SET [Production].[Product].[ModifiedDate] = GETDATE( ) FROM inserted WHERE inserted.[ProductID] = [Production].[Product].[ProductID];
		트리거 내부의 두 번째 명령문이 시작됩니다.
10	SP:StmtCompleted	UPDATE [Production].[Product] SET [Production].[Product].[ModifiedDate] = GETDATE( ) FROM inserted WHERE inserted.[ProductID] = [Production].[Product].[ProductID];
		트리거 내부의 두 번째 명령문이 완료됩니다.
11	SP:Completed	UPDATE Production.Product SET ListPrice = ListPrice * 1.1 WHERE ProductID = @ProductID
		uProduct 트리거의 실행이 완료됩니다.
12	SP:StmtCompleted	UPDATE Production.Product SET ListPrice = ListPrice * 1.1 WHERE ProductNumber LIKE 'EC%'
		4단계에서 시작된 원래의 UPDATE 문의 실행이 완료됩니다.
13	SQL:BatchCompleted	UPDATE Production.Product SET ListPrice = ListPrice * 1.1 WHERE ProductNumber LIKE 'EC%'
		The batch started on step 4 completes.
14	TM: Commit Tran starting	COMMIT TRANSACTION
		myTSScope 개체의 Complete 메서드는 트랜잭션 관리자에게 SQL 트랜잭션 164587로 식별된 XactSequence 번호 2362232012810로 정의된 트랜잭션을 커밋하도록 요청합니다.
15	SQLTransaction	164587 user_transaction 1 - Commit
		이것은 2단계에서 트랜잭션 관리자가 시작한 사용자 트랜잭션이며 이 단계에서 커밋됩니다.
16	TM: Commit Tran completed	COMMIT TRANSACTION
		사용자 트랜잭션이 이 연결에서 커밋되었습니다.

트랜잭션이 앞의 예제와 다른 방식으로 만들어진 것을 명확하게 표시하며 트랜잭션 개체가 트랜잭션을 커밋하도록 명시적으로 요청할 때까지 유지됩니다. 이는 개발자에게 보다 세부적인 컨트롤을 제공하며 같은 트랜잭션의 일부로 더 많은 작업을 포함할 수 있습니다.
한 데이터 저장소만 필요하며 배포된 트랜잭션은 필요하지 않기 때문에 이 경우 System.Transactions는 모든 트랜잭션 작업을 표준 SQL 트랜잭션을 기반으로 한다는 것을 인식하는 것이 중요합니다. 응용 프로그램이 두 개의 다른 서버에서 데이터를 업데이트해야 하는 경우 System.Transactions는 대신 DTC 트랜잭션을 시작합니다.

새 SQL Native Client의 동시 작업과 트랜잭션 컨트롤

새로운 SQL Server 2005용 SQL Native Client는 SQL OLE DB 공급자와 SQL ODBC 드라이버를 하나의 기본 동적 링크 라이브러리(DLL)로 통합하고 이러한 두 인터페이스의 기능을 확장하는 MDAC를 대체하도록 설계되었습니다. 추가된 성능과 기능을 가진 ADO 기반 데이터베이스 응용 프로그램에 대한 마이그레이션 경로를 제공하는 ADO(ActiveX Data Objects)와 호환됩니다.

SQL Native Client의 주 목적은 SQL Server 2005 관련 기능에 대한 지원을 현재 ODBC 또는 OLEDB를 기본적으로 사용하는 C++ 개발자에게 제공하는 것입니다. "개발자에게 익숙한" 프로그래밍 모델이 응용 프로그램 성능을 저하시키는 응용 프로그램 처리의 추가 레이어를 포함하고 있기 때문에 이러한 개발자는 ADO 또는 ADO.NET을 피할 수 있습니다.
예를 들어, SQL Native Client는 이러한 옵션을 이용하기 위해 ADO.NET을 요구하지 않고 MARS와 비동기 작업을 기본적으로 지원합니다.

3.1 SQL Native Client를 사용한 SQLOLEDB 트랜잭션 처리 고려 사항

기본적으로 SQLOLEDB는 자동 세션 기반 트랜잭션 제어를 사용합니다. 각 실행 단위는 클라이언트 응용 프로그램에서 수동 제어를 요구하지 않는 단일 트랜잭션으로 간주됩니다.

그러나 여러 작업 단위가 동일한 트랜잭션에 참가할 수 있습니다. SQLOLEDB 사용자는 StartTransaction, Commit 및 Abort 메서드를 제시하는 ITransactionLocal 인터페이스를 통해 SQL Server 트랜잭션을 보다 정밀하게 제어할 수 있습니다.

트랜잭션을 보다 폭 넓게 제어하기 위해 SQL Server 2005 SQLOLEDB 트랜잭션은 MS-DTC가 관리하는 기존의 배포 트랜잭션에 참가할 수 있습니다. SQLOLEDB 사용자는 ITransactionJoin::JoinTransaction 메서드를 사용하여 배포 트랜잭션에 참가할 수 있습니다.

SQLOLEDB 프로그래밍 모델을 사용하는 클라이언트 응용 프로그램은 여러 가지 방식으로 트랜잭션 격리를 제어할 수 있습니다.

SQLOLEDB 기본 자동 커밋 모드의 경우 DBPROPSET_SESSION 속성 DBPROP_SESS_AUTOCOMMITISOLEVELS for SQLOLEDB.

local manual-commit transactions의 경우 ITransactionLocal::StartTransaction 메서드의 isoLevel 매개 변수.
MS DTC-조정된 배포 트랜잭션의 경우 ITransactionDispenser::BeginTransaction 메서드의 isoLevel 매개 변수.

대부분의 데이터베이스 응용 프로그램의 경우 적절한 트랜잭션 격리 수준을 선택하는 것이 중요합니다. 한편으로는, 비즈니스 요구 사항이 데이터베이스 액세스를 비즈니스 요구 사항과 호환되는 방식으로 수행할 수 있도록 격리 수준을 보다 제한적으로 요구할 수 있습니다. 다른 한편으로는, 높은 트랜잭션 격리 수준은 동시 작업을 줄입니다. SQLOLEDB 응용 프로그램의 기본 격리 수준은 DBPROPVAL_TI_READCOMMITTED이며, 이는 SQL Server 기본 트랜잭션 격리 수준인 READ COMMITTED와 동일합니다.

3.2 SQL Native Client를 사용한 ODBC 트랜잭션 처리 고려 사항

SQLOLEDB와 반대로 SQL Native Client를 통해 ODBC 프로그래밍 모델을 사용하는 데이터베이스 응용 프로그램은 연결 수준에서 트랜잭션을 관리할 수 있습니다. ODBC 모델은 두 가지 다른 트랜잭션 모델을 제시합니다.

자동 커밋 모드 - SQL Server의 기본 명령문 당 자동 커밋 모드를 기본적으로 사용합니다. 이러한 작업은 명령문 기준으로 명령문에서 SQL Server가 자동으로 관리하기 때문에 데이터베이스 응용 프로그램은 연결이 열린 후 만들어진 트랜잭션을 커밋하거나 롤백하는 작업을 수행할 필요가 없습니다.

수동 커밋 모드 - SQL Server의 IMPLICIT_TRANSACTION 기능을 기반으로 합니다. 연결이 설정된 이후(또는 이전 SqlEndTran 이후) 수행한 모든 작업을 동일 트랜잭션의 일부로 간주합니다. 이 트랜잭션은 응용 프로그램이 SqlEndTran 메서드를 실행하는 경우에만 종료됩니다. 응용 프로그램이 작업을 커밋하지 않고 연결을 닫는 경우 모든 작업은 자동으로 롤백됩니다.

ODBC 응용 프로그램은 자동 커밋 모드를 해제하기 위해 SQLSetConnectAttr 메서드를 호출하여 언제든지 자동 커밋에서 수동 모드로 전환할 수 있습니다.

데이터베이스 응용 프로그램은 수동 모드에서 트랜잭션을 종료하기 위해 SQL_COMMIT 또는 SQL_ROLLBACK 옵션을 사용하여 SqlEndTran 메서드를 호출해야 합니다.

분산 트랜잭션의 경우 ODBC 응용 프로그램은 자동 커밋 모드를 사용하지 않고 실행해야 하지만 MS-DTC가 직접 관리하기 때문에 SQLEndTran을 호출할 필요가 없습니다. 그러나 분산 트랜잭션에 참가하는 데 필요한 프로세스는 SQLOLEDB 모델보다 약간 더 복잡합니다.

클라이언트 응용 프로그램은 MS-DTC OLE 인터페이스의 ITRansactionDispenser::BeginTransaction 메서드를 호출하여 트랜잭션 컨텍스트에 대한 참조를 얻습니다.

클라이언트 응용 프로그램은 SQLSetConnectAttr 메서드를 호출하여 SQL_COPT_SS_ENLIST_IN_DTC 특성을 이전 단계에서 얻은 DTC 개체로 설정합니다.

이제 이 연결은 분산 트랜잭션의 일부로 등록됩니다.

ODBC 프로그래밍 모델이 트랜잭션 격리 수준 반복 가능한 읽기를 Serializable로 구현합니다. SQL_ATR_TXN_ISOLATION을 사용하여 트랜잭션 격리 수준을 다음 값으로 설정할 수 있습니다.

SQL_TXN_READ_UNCOMMITTED, SQL_TXN_READ_COMMITTED, SQL_TXN_REPEATABLE_READ, SQL_TXN_SS_SNAPSHOT, or SQL_TXN_ SERIALIZABLE.

ODBC 모델은 기본적으로 사용되는 MARS를 지원합니다. 응용 프로그램은 SQLSetConnectAttr 메서드를 호출하여 SQL_COPT_SS_MARS_ENABLED를 SQL_MARS_ENABLED_YES 또는 SQL_MARS_ENABLED_NO로 설정하여 이 동작을 제어할 수 있습니다.

COM+ 서비스 관리 구성 요소의 트랜잭션 관리

COM+는 응용 프로그램에 여러 가지 서비스를 제공하며, 그 중 일부는 분산된 환경에서 트랜잭션을 관리하도록 설계되었습니다. 이러한 트랜잭션 관련 서비스는 다음과 같습니다.

선언적 트랜잭션 처리 기능을 적용하는 자동 트랜잭션 처리.
BYOT(Bring Your Own Transaction), BYOT 기능을 사용하여 COM+ Distributed Transaction Coordinator (DTC)에 액세스하기 위해 트랜잭션 상속의 형태를 허용합니다.

Compensating Resource Manager (CRM), 원자가 및 내구성 속성을 비트랜잭션 리소스에 적용합니다.
대기열 구성 요소, 비동기 메시지 대기열을 제공합니다.

4.1 자동 트랜잭션 처리

이 서비스는 설계 시간에 선언적 방식으로 클래스의 트랜잭션 동작을 구성하는 기능을 제공하며 이는 이 클래스에서 만든 개체가 트랜잭션 시작과 종료 지점을 제어하는 추가 코드 없이 런타임에 트랜잭션에 참가하는 방법을 나타냅니다.
이 기술은 응용 프로그램 코드에서 트랜잭션 관리를 간소화하지만 비용이 발생합니다. 성능 저하가 발생하지 않도록 이러한 암시를 이해하는 것이 중요합니다.

트랜잭션 처리를 위한 일반적인 최상의 방식은 반드시 열 필요가 있을 때까지 연결을 열지 말고 가능한 빨리 연결을 닫는 것이 좋습니다. 같은 맥락에서 반드시 필요할 때까지는 트랜잭션을 시작하지 않고 더 이상 필요하지 않을 때는 트랜잭션을 완료하는 것을 고려해야 합니다.

서비스되는 구성 요소의 일반적인 문제 중 하나는 응용 프로그램이 연결을 여는 즉시 새 트랜잭션을 만드는 것입니다(또는 기존 트랜잭션을 사용하여 현재 연결을 등록). 즉, 이 연결에서 무엇을 수행하던 트랜잭션의 일부가 되며 연결을 처음 연 이후로 실행된 모든 명령문을 포함할 것입니다. 이렇게 하면 트랜잭션이 연결 범위의 트랜잭션의 일부가 될 필요가 없고 SQL Server의 기본 자동 커밋 동작 하에서 더 잘 수행될 수 있는 명령문의 실행을 포함하기 때문에 동시 작업이 감소합니다.

그리 많지 않은 개발자가 인식하지 못하는 또 다른 문제는 COM+ 트랜잭션은 SERIALIZABLE 트랜잭션 격리 수준을 사용하여 기본적으로 만들어진다는 사실입니다. 즉, 이 연결에서 실행된 명령문에서 얻은 공유 잠금은 트랜잭션이 종료될 때까지 해제되지 않는 것을 의미합니다. COM+ 트랜잭션의 트랜잭션 격리 수준을 변경하는 것은 쉽지만, 개발자는 격리 수준의 변경이 필요할 수 있음을 인식해야 합니다. 문제를 더 복잡하게 만드는 것은 설계하는 개체가 분산 트랜잭션에서 루트 개체인지 여부가 항상 명확하지 않습니다. 따라서 현재 구성 요소의 트랜잭션 격리 수준을 항상 변경하지 못할 수 있으며 개발자는 필요한 방식으로 동시 작업을 제어하지 못할 수 있습니다.

데이터베이스 응용 프로그램이 한 SQL Server 인스턴스의 데이터만 사용하고 자동 트랜잭션 이외에 COM+의 다른 서비스를 요구하지 않는 경우 이 응용 프로그램은 COM+ 기반 분산 트랜잭션 대신 로컬 트랜잭션을 기반으로 다시 작성해야 합니다. COM+는 개발을 간소화하지만 트랜잭션 디버깅을 더 어렵게 만듭니다. COM+는 각 트랜잭션에 대한 성능 페널티(대개 20%와 50% 사이)를 부과하기도 하므로, 분산 트랜잭션이 필요하지 않은 경우 이를 피할 수 있습니다.

4.2 BYOT(Bring Your Own Transaction).

BYOT 기능은 트랜잭션 상속의 형태를 허용하므로 COM+ 구성 요소는 기존의 DTC(Distributed Transaction Protocol) 또는 TIP(Transaction Internet Protocol) 트랜잭션을 사용하는 연결을 등록할 수 있습니다. 이 기술은 자동 트랜잭션보다 세부적인 제어가 가능하기는 하지만 약간 복잡한 경우에는 관리가 어려울 수 있습니다.

4.3 CRM(Compensating Resource Manager)

COM+는 DTC 트랜잭션에서 비트랜잭션 개체를 포함할 수 있도록 Compensating Resource Manager를 제공합니다. CRM은 최소한의 트랜잭션 원자가(전체 또는 아무 행동도 없음) 및 복구 로그를 통한 내구성을 제공하는 간단한 리소스 관리자입니다. 비트랜잭션 개체를 DTC 트랜잭션에 추가할 경우 더 복잡해지고 DTC 트랜잭션 실행 시간이 더 길어지므로 이러한 암시를 이해하는 것이 중요합니다. 게다가, 이러한 비트랜잭션 개체는 고려해야 하는 가능한 오류 조건 수를 확장하여 트랜잭션이 롤백될 가능성을 더욱 크게 만들고 사용자 관점에서 실감할 정도의 성능 저하를 발생시킵니다.

이는 이 서비스를 사용해서는 안 된다는 것을 의미하는 것이 아니라 사용하기 전에 이 서비스가 필요한지 평가하는 것이 중요하다는 뜻입니다. 업무 상 이 서비스가 반드시 필요하며 다른 방법으로는 쉽게 해결할 수 없는 경우 CRM이 가장 좋은 솔루션이 될 수 있습니다. 그러나 사용자에게 적합한 방식으로 특정 비즈니스 요구에 서비스 할 수 있도록 응용 프로그램을 모니터링해야 합니다. CRM을 설명하는 것은 이 문서의 범위를 벗어나지만 다음 웹 사이트에서 이 서비스에 대한 자세한 설명을 볼 수 있습니다.

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cossdk/html/3d490da6-1577-4a77-9f7d-6188f96f2914.asp

4.4 대기열 구성 요소.

대기열 구성 요소는 Microsoft Message Queue(MSMQ) 기능을 구성 요소에 제시하는 COM+ 서버스이며 비동기 메시지 대기열을 제공합니다. 이러한 메시지는 트랜잭션이 될 수 있어 메시지가 순서대로 전달되고, 한 번만 전달되며 대상 대기열에서 성공적으로 검색할 수 있도록 합니다.

트랜잭션 내에서 메시지 집합을 보내는 것은 단일 트랜잭션을 구성하기 위해 메시지를 함께 그룹화하는 것을 포함합니다. 대기열 기술은 내부와 외부 트랜잭션 사이를 구분합니다. 내부 트랜잭션은 메시지 관리와 직접 관련이 있으며 이 문서의 범위를 벗어납니다.

외부 대기열 트랜잭션은 데이터베이스 같은 외부 리소스를 포함합니다. 이러한 트랜잭션은 일반적으로 MS-DTC인 메시지 대기열 시스템의 일부가 아닌 외부 트랜잭션 조정자에 의존합니다. 외부 트랜잭션 조정자는 내부 트랜잭션에 사용되는 것보다 더 복잡한 프로그래밍 모델이 있습니다.

원칙적으로 대기열 구성 요소를 사용하는 유일한 이유가 SQL Server의 한 인스턴스를 포함하는 데이터베이스 응용 프로그램의 확장성을 확장하는 것인 경우 Service Broker를 기반으로 하는 아키텍처 외에 이 아키텍처를 다시 고려하는 것이 좋습니다.

동시 작업 및 비동기 Service Broker 지원 응용 프로그램

Service Broker는 안정적이고 확장성 있는 데이터베이스 응용 프로그램을 만들기 위해 대기열 기반 메시징 프레임워크용 인프라를 제공하는 SQL Server 2005 Storage Engine의 한 구성 요소입니다.
일반적인 사용자 지정 대기열 데이터베이스 응용 프로그램은 대기열 테이블을 많이 사용하며, 클라이언트 응용 프로그램과 대기열 서비스는 지속적으로 적은 수의 행을 삽입, 선택 및 삭제하여 빈번한 동시 작업 문제를 일으킵니다. 이러한 동시 작업 문제는 주로 SQL Server 잠금 관리자가 이러한 대기열 테이블을 표준 테이블 이외의 것으로 간주하지 않고 응용 프로그램이 대기열 테이블에 있는 데이터에 액세스할 때 보통의 잠금 메커니즘을 적용하기 때문에 발생합니다. 액세스가 심한 테이블에 사용되는 일반 잠금 메커니즘은 자주 차단되고 데드록이 많이 발생하기 때문에 개발자가 잘 수행되는 사용자 지정 대기열 메커니즘을 설계하는 것은 매우 어렵습니다.

Service Broker는 사용자 지정 대기열 응용 프로그램에 대안을 제공하며 전체 대기열과 메시징 인프라를 관리합니다. Service Broker를 사용하면 개발자는 기술 세부 사항을 다루지 않고도 제공되는 프레임워크를 사용하는 응용 프로그램 작성에 집중할 수 있습니다.

Service Broker를 사용하는 응용 프로그램은 변환의 일부로 대기열로 보낸 메시지를 기반으로 하며, 이는 변환 그룹으로 그룹화됩니다. 응용 프로그램은 트랜잭션의 일부로 변환에 메시지를 보내고, 각 변환의 모든 메시지가 Service Broker에 등록되거나 아무것도 등록되지 않도록 합니다.

응용 프로그램은 특정 Service Broker 서비스를 정의하며, 저장 프로시저 또는 외부 Windows나 콘솔 응용 프로그램으로 구현할 수 있습니다. 경우에 따라 응용 프로그램은 받은 순서대로 한 번에 한 메시지를 처리하도록 Service Broker 서비스를 외부 Windows 서비스로 구현할 수 있습니다. 대기열이 자동 내부 활성화를 사용하여 정의된 경우 Service Broker는 MAX_QUEUE_READERS 옵션에 정의된 한도까지 이러한 각 서비스의 인스턴스를 필요한 만큼 실행합니다. 이러한 서비스의 여러 인스턴스를 가지면 Service Broker는 이 대기열에서 모든 보류 중인 변환을 처리할 수 있습니다.

Service Broker의 트랜잭션은 두 개의 개별 단계로 처리됩니다.

응용 프로그램은 단일 비즈니스 트랜잭션의 일부로 보낸 메시지 또는 메시지 그룹의 동시 작업을 보장하기 위해 트랜잭션을 만듭니다. 트랜잭션은 이러한 메시지를 대기열에 추가하고 모두 논리적으로 그룹화되도록 Service Broker가 수행한 모든 필수 작업을 포함합니다. 이 트랜잭션을 롤백해야 하는 경우 커밋된 트랜잭션만 Service Broker 변환의 일부가 되므로 변환의 무결성은 여전히 보존됩니다.

Service broker 서비스는 이 변환에 포함된 메시지가 요청한 데이터베이스 작업 뿐만 아니라 이 변환을 반영하도록 인프라 테이블에 대한 모든 변경 내용을 포함할 자체 트랜잭션에서 각 변환 그룹을 처리해야 합니다. MSDN 문서 "A First Look at SQL Server 2005 Service Broker" (http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnsql90/html/sqlsvcbroker.asp)는 이 프로세스를 자세히 설명합니다.

첫 번째 프로세스는 SQL Server의 행 수준 잠금으로 인해 동시 작업 문제가 발생할 가능성이 없습니다. 각 변환 그룹은 작업 단위를 나타내며, 단일 트랜잭션으로 대기열에 삽입되고 단일 트랜잭션으로 처리됩니다. Service Broker는 각 메시지가 한 번만 처리되도록 하므로 동시에 동일한 데이터를 결정하는 두 프로세스가 있는 경우는 발생할 가능성이 거의 없으며 이벤트는 Service Broker에 의해 투명하게 관리됩니다.

변환 그룹 잠금은 Service Broker에 의해 자동으로 처리되며 수동으로 관리할 힌트 같은 메커니즘이 없습니다. 하나의 대기열 읽는 사용자 서비스만 한 번에 주어진 변환 그룹의 메시지를 처리할 수 있습니다.
다음 명령문은 변환 그룹 잠금을 획득합니다.

BEGIN DIALOG CONVERSATION (Transact-SQL)
- BEGIN CONVERSATION TIMER (Transact-SQL)
- END CONVERSATION (Transact-SQL)
- MOVE CONVERSATION (Transact-SQL)
- RECEIVE (Transact-SQL)
- SEND (Transact-SQL)

그러나 데이터베이스 응용 프로그램으로서, 연결이 끊어진 모델을 기반으로 하는 응용 프로그램에 더 중요하므로 데이터에 대한 업데이트가 업데이트할 데이터의 특정 상태에 의존하지 않는 방식으로 응용 프로그램에 중요합니다. 두 작업 모두 일련화할 수 있는 트랜잭션의 일부가 될 수 없는 경우 이전에 읽은 값이 업데이트 작업이 발생하는 시간에 동일하다는 보장이 없기 때문에 이전에 읽은 데이터 값에 의존하지 않고 가능하면 원자 데이터 업데이트를 수행하도록 응용 프로그램을 설계해야 합니다. 이것은 한 메시지는 SQL Server 데이터에 특정한 변경을 요청하고 다른 메시지는 같은 데이터의 다른 변경을 요청하는 경우 적절히 설계되지 않은 트랜잭션의 업데이트가 손실될 수 있는 Service Broker 응용 프로그램의 경우에도 적용됩니다. 업데이트 중 하나는 손실될 가능성이 많습니다. 이것은 Service broker 문제가 아니라 응용 프로그램 설계 문제입니다.

한 응용 프로그램이 데이터의 현재 상태를 기반으로 하는 일부 데이터베이스 수정을 요청할 때 문제가 발생하는 것이 방지되지만 이때 이러한 수정은 데이터베이스에 적용해야 하고 기본 데이터는 다른 프로세스에 의해 변경되었을 수 있으므로 이는 낙관적 동시 작업 기술을 사용하는 것이 유용할 수 있는 경우 일반적입니다. 낙관적 동시 작업을 구현하려면 각 변환은 이 메시지를 처리하는 일을 담당하는 서비스가 이러한 새 변경을 허용하거나 이 트랜잭션을 롤백할지 여부를 확인하는 데 충분한 정보를 가질 수 있도록 데이터의 현재와 미래 상태에 대한 충분한 정보를 포함해야 합니다.

즉, 응용 프로그램이 SQL Server에서 데이터를 읽으면 응용 프로그램은 이 데이터를 원래 값으로 유지해야 합니다(이것은 DataSets이 자동으로 사용되는 방법입니다). 응용 프로그램이 이러한 데이터 값에 대한 변경 요청을 보내면 메시지는 실제 값과 원래 값 모두 포함해야 합니다. 생성되는 UPDATE 명령문은 SET 구문에서 실제 값을, WHERE 구문에서 원래 값을 사용합니다. 이런 식으로 데이터가 원래 읽은 것과 여전히 동일한 경우 WHERE 구문은 이 동일한 행을 다시 발견하고 SET 구문은 이러한 값을 업데이트하는 방법을 결정합니다.

그러나 데이터를 원래 읽은 이후로 데이터가 다른 프로세스에 의해 변경된 경우 WHERE 구문은 기존 행과 일치하지 않으며 행은 이 UPDATE 작업으로 변경되지 않습니다. 이 경우는 SQL Server 오류를 생성하지 않지만 이 특정 Service broker 서비스를 구현하는 저장 프로시저는 클라이언트 응용 프로그램이 동시 작업 위반으로 고려해야 하는 이 이벤트의 알림을 보내야 합니다.

성공적으로 처리할 수 없는 메시지는 Poison Message가 되며 응용 프로그램은 포이즌 메시지를 정상적으로 관리하도록 설계해야 합니다. 특히, 이 포이즌 메시지의 이유가 동시 작업 문제인 경우 클라이언트 응용 프로그램은 이 이벤트에 대해 알려야 합니다. 클라이언트 응용 프로그램은 이 상황을 해결해야 하며 낙관적 동시 작업 환경에서 데이터가 손실되는 것으로 취급할 수 있습니다. 일반적인 솔루션은 데이터베이스에서 사용할 수 있는 새로운 기본 값을 사용자에게 제공하여 사용자에게 다시 업데이트할 수 있는 가능성을 제공하거나 제안된 변경을 취소하는 것입니다.

결론

SQL Server 2005는 새로운 기능 및 기존 기능을 변경하여 데이터에 동시 액세스를 개선했습니다. 많은 기능이 SQL Server에서 새로운 RLV(Row Level Versioning) 기술을 바탕으로 구축되었습니다. 트리거와 같은 RLV 기술을 기반으로 하는 일부 기능은 응용 프로그램을 변경할 필요가 없으며 개발자의 관점에서는 거의 느낄 수 없습니다. 스냅샷 격리의 두 수준 같이 새 데이터베이스 옵션을 사용해야 하며 약간의 코드 변경이 필요할 수 있습니다. 스냅샷 격리를 확실히 이용하려면 현재성과 동시 작업의 장단점을 재평가해야 합니다.

SQL Server 2005는 SQL Server와 통신하는 클라이언트를 위한 동시 작업 향상을 제공합니다. SQL Server가 내부적으로 동시 작업을 관리하는 방법에 대해 보다 자세히 알고 있다면 클라이언트 응용 프로그램에 대한 최적의 동시 작업 제어 값을 보다 현명하게 결정할 수 있을 것입니다.