Microsoft®
SQL Server™ 2005

SQL Server 2005 Transact-SQL 기능 개선

Microsoft®

요약

본 백서에서는 Microsoft SQL Server 2005 Transact-SQL의 새로운 기능 개선에 대해 소개합니다. 이러한 새로운 기능들은 표현 능력과 쿼리 성능 및 오류 관리 기능을 향상시킬 수 있습니다. 본 백서는 주로 개념적으로 새로운 관계성 개선 요소에 초점을 맞추고 실제 사례를 통해 이를 증명합니다. 여기에서는 새로운 Transact-SQL 기능을 모두 다루고 있지 않습니다.

Contents

들어가는 글

본 백서에서는 Microsoft SQL Server 2005 Transact-SQL의 새로운 향상 기능에 대해 소개합니다. 이들 새 기능들은 표현 능력, 쿼리 성능 그리고 오류 관리 기능을 보완할 수 있습니다. 본 백서는 주로 개념적으로 새로운 관계성 개선 요소에 초점을 맞추고 실제 사례를 통해 이를 증명합니다. 여기에서는 새로운 Transact-SQL 기능을 모두 다루고 있지 않습니다.
기본 지식 요건: 본 백서를 이해하기 위해서는 특별 쿼리를 위해, 그리고 Microsoft SQL Server 2000 애플리케이션의 구성 요소로서 Transact-SQL을 다룰 수 있어야 합니다.

쿼리의 표현 능력과 DRI 지원 기능 향상

이 섹션에서는 다음과 같은 새로운 관계형 기능과 향상된 특징들에 대해 소개합니다.
새로운 순위 함수

CTE(common table expressions)에 기반한 새로운 재귀 쿼리

새로운 PIVOT 및 APPLY 관계형 연산자

DRI(Declarative referential integrity) 기능 개선

순위 함수

SQL Server 2005에서는 ROW_NUMBER, RANK, DENSE_RANK 및 NTILE 등 네 가지 순위 함수가 새롭게 추가되었습니다. 이러한 새 함수를 이용하여 데이터를 효과적으로 분석하고 쿼리의 결과 행에 순위 값을 제공할 수 있습니다. 이 새 함수들은 프레젠테이션을 위해 결과 행에 순차 정수를 지정하는 것은 물론 페이징, 점수 매기기, 히스토그램 등에 유용하게 사용될 수 있습니다.

Speaker Statistics 시나리오

다음 Speaker Statistics 시나리오는 다양한 함수와 해당 절을 논의 및 시연하는 데 사용됩니다. 대규모 컴퓨팅 회의에는 데이터베이스, 개발 및 시스템 관리라는 세 가지 트랙이 포함되었습니다. 11명의 발표자가 회의에서 발언했고 각자의 세션에 대해 1-9점을 득점했습니다. 다음 SpeakerStats 테이블에 그 결과가 요약 및 저장되어 있습니다.

```
USE tempdb -- or your own test database
CREATE TABLE SpeakerStats
(
  speaker       VARCHAR(10) NOT NULL PRIMARY KEY,
  track         VARCHAR(10) NOT NULL,
  score         INT         NOT NULL,
  pctfilledevals INT        NOT NULL,
  numsessions   INT         NOT NULL
)

SET NOCOUNT ON
INSERT INTO SpeakerStats VALUES( 'Dan' ,     'Sys' , 3, 22, 4)
INSERT INTO SpeakerStats VALUES( 'Ron' ,     'Dev' , 9, 30, 3)
INSERT INTO SpeakerStats VALUES( 'Kathy' ,   'Sys' , 8, 27, 2)
INSERT INTO SpeakerStats VALUES( 'Suzanne' , 'DB' ,  9, 30, 3)
INSERT INTO SpeakerStats VALUES( 'Joe' ,     'Dev' , 6, 20, 2)
INSERT INTO SpeakerStats VALUES( 'Robert' ,  'Dev' , 6, 28, 2)
INSERT INTO SpeakerStats VALUES( 'Mike' ,    'DB' ,  8, 20, 3)
INSERT INTO SpeakerStats VALUES( 'Michele' , 'Sys' , 8, 31, 4)
INSERT INTO SpeakerStats VALUES( 'Jessica' , 'Dev' , 9, 19, 1)
INSERT INTO SpeakerStats VALUES( 'Brian' ,   'Sys' , 7, 22, 3)
INSERT INTO SpeakerStats VALUES( 'Kevin' ,   'DB' ,  7, 25, 4)
```

각 발표자마다 테이블의 한 행에 발표자의 이름, 트랙, 평균 점수, 세션 참석자의 수를 기준으로 평가를 기입한 참석자의 비율, 발표자가 제공한 세션 수가 표시됩니다. 이 섹션에서는 새로운 순위 함수를 이용하여 유용한 정보를 생성하기 위해 발표자의 통계 데이터를 분석하는 방법에 대해 설명합니다.

Semantics

네 가지의 순위 함수 모두 유사한 구문 패턴을 따릅니다.

순위 함수

```
〈function_name〉(    ) OVER(
 [PARTITION BY 〈partition_by_list〉]
 ORDER BY 〈order_by_list〉)
```

이 함수는 SELECT 절 또는 ORDER BY 절과 같은 쿼리의 두 가지 절에만 지정할 수 있습니다. 다음 섹션에서는 다른 함수에 대해 자세히 설명합니다.

ROW_NUMBER

ROW_NUMBER 함수를 사용하면 쿼리의 결과 행에 순차 정수 값을 제공할 수 있습니다. 예를 들어 내림차순의 점수에 따라 결과 행에 1부터 순차 값을 지정하여 speaker, track 및 모든 발표자의 score를 반환하고자 한다고 가정해 봅시다. 다음 쿼리는 ROW_NUMBER 함수에 OVER(ORDER BY score DESC)를 지정하여 원하는 결과를 발생시킵니다.

```
SELECT ROW_NUMBER(  ) OVER(ORDER BY score DESC) AS rownum,
  speaker, track, score
FROM SpeakerStats
ORDER BY score DESC
```

결과 집합은 다음과 같습니다.

rownum	speaker	track	score
1	Jessica	Dev	9
2	Ron	Dev	9
3	Suzanne	DB	9
4	Kathy	Sys	8
5	Michele	Sys	8
6	Mike	DB	8
7	Kevin	DB	7
8	Brian	Sys	7
9	Joe	Dev	6
10	Robert	Dev	6
11	Dan	Sys	3

최고 점수를 획득한 발표자의 열 번호는 1번이며 최저 점수를 획득한 발표자의 열 번호는 11번입니다. ROW_NUMBER는 항상 요청된 정렬에 따라 서로 다른 행에 고유한 열 번호를 생성합니다. OVER( ) 옵션 내에 지정된 ORDER BY 목록이 고유하지 않을 경우 결과를 예측할 수 없음을 유념하십시오. 즉, 쿼리에 대한 올바른 결과가 두 가지 이상 존재합니다. 쿼리를 다르게 호출하면 다른 결과를 얻을 수도 있습니다. 예를 들어, 위의 경우에는 제시카, 론, 수잔이라는 3명의 다른 발표자들이 동일한 최고 점수(9)를 획득했습니다. SQL Server는 다른 발표자에게 다른 행 번호를 지정해야 하므로 제시카, 론, 수잔에게 각각 지정된 값 1, 2, 3은 임의 순서로 발표자들에게 지정되었다고 생각해야 합니다. 값 1, 2, 3이 론, 수잔, 제시카에게 각각 지정되어도 올바른 결과가 나올 것입니다.

고유한 ORDER BY 목록을 지정하는 경우 결과는 항상 정해져 있습니다. 예를 들어, 발표자들이 동점을 획득한 경우 순위를 가리기 위해 가장 높은 pctfilledevals 값을 사용하고자 한다고 가정해 봅시다. 아직도 동점자가 있으면 순위를 가리기 위해 가장 높은 numsessions 값을 사용하십시오. 마지막으로, 아직도 동점자가 있으면 순위를 가리기 위해 사전 순서상 최하위에 있는 speaker 이름을 사용하십시오. ORDER BY 목록-score, pctfilledevals, numsessions 및 speaker-이 고유하기 때문에 결과는 정해져 있습니다.

```
SELECT ROW_NUMBER(    ) OVER(ORDER BY score DESC, pctfilledevals DESC,
                        numsessions DESC, speaker) AS rownum,
  speaker, track, score, pctfilledevals, numsessions
FROM SpeakerStats
ORDER BY score DESC, pctfilledevals DESC, numsessions DESC, speaker
```

결과 집합은 다음과 같습니다.

rownum	speaker	track	score	pctfilledevals	numsessions
1	Ron	Dev	9	30	3
2	Suzanne	DB	9	30	3
3	Jessica	Dev	9	19	1
4	Michele	Sys	8	31	4
5	Kathy	Sys	8	27	2
6	Mike	DB	8	20	3
7	Kevin	DB	7	25	4
8	Brian	Sys	7	22	3
9	Robert	Dev	6	28	2
10	Joe	Dev	6	20	2
11	Dan	Sys	3	22	4

새로운 순위 함수의 중요한 이점 중 하나는 바로 효율성입니다. SQL Server의 최적화 프로그램은 값을 계산하기 위해 데이터를 한 번만 스캔해야 합니다. 정렬 열에 위치한 인덱스의 순서에 따른 스캔을 사용하거나 데이터를 한 번 스캔한 다음 적합한 인덱스가 생성되지 않은 경우 데이터를 분류하는 방법이 있습니다.

또 다른 이점은 구문의 단순성입니다. SQL Server의 이전 버전에서 사용된 집합 기반 접근법을 사용하여 순위 값을 계산하는 것이 얼마나 어렵고 비효율적인지 이해하기 위해 이전 쿼리와 동일한 결과를 반환하는 다음 SQL Server 2000 쿼리를 생각해 보십시오.

```
SELECT
  (SELECT COUNT(*)
   FROM SpeakerStats AS S2
   WHERE S2.score 〉 S1.score
     OR (S2.score = S1.score
         AND S2.pctfilledevals 〉 S1.pctfilledevals)
     OR (S2.score = S1.score
         AND S2.pctfilledevals = S1.pctfilledevals
         AND S2.numsessions 〉 S1.numsessions)
     OR (S2.score = S1.score
         AND S2.pctfilledevals = S1.pctfilledevals
         AND S2.numsessions = S1.numsessions
         AND S2.speaker 〈 S1.speaker)) + 1 AS rownum,
  speaker, track, score, pctfilledevals, numsessions
FROM SpeakerStats AS S1
ORDER BY score DESC, pctfilledevals DESC, numsessions DESC, speaker
```

이 쿼리는 분명히 SQL Server 2005 쿼리보다 훨씬 복잡합니다. 게다가 SpeakerStats 테이블의 각 기본 행에 대해 SQL Server는 테이블의 다른 인스턴스에 있는 모든 대응되는 행을 스캔해야 합니다. 평균적으로 기본 테이블의 각 행을 기준으로 테이블 행의 절반 가량(최소)이 스캔되어야 합니다. SQL Server 2005 쿼리의 성능 저하는 선형적이지만 SQL Server 2000 쿼리의 성능 저하는 기하급수적입니다. 초소형 테이블에서도 성능 차이는 상당합니다. 예를 들어 SalesOrderID 주문에 따라 판매 주문에 대한 행의 번호를 계산하기 위해 AdventureWorks 데이터베이스에서 SalesOrderHeader 테이블을 쿼리하는 다음 쿼리의 성능을 테스트해 보십시오. SalesOrderHeader 테이블의 행은 31,465개입니다. 첫 번째 쿼리는 SQL Server 2005 ROW_NUMBER 함수를 사용하지만 두 번째 쿼리는 SQL Server 2000 하위 쿼리 기법을 사용합니다.

```
-- SQL Server 2005 query
SELECT SalesOrderID,
  ROW_NUMBER(   ) OVER(ORDER BY SalesOrderID) AS rownum
FROM Sales.SalesOrderHeader
-- SQL Server 2000 query
SELECT SalesOrderID,
  (SELECT COUNT(*)
   FROM Sales.SalesOrderHeader AS S2
   WHERE S2.SalesOrderID 〈= S1.SalesOrderID) AS rownum
FROM Sales.SalesOrderHeader AS S1
```

랩탑에서 이 테스트를 실행합니다(Compaq Presario X1020U, CPU: Centrino 1.4 GH, RAM: 1GB, 로컬 HD). SQL Server 2005 쿼리는 단 1초 만에 완료되었지만 SQL Server 2000 쿼리는 약 12분 만에 완료되었습니다.
행 번호를 이용하는 가장 일반적인 활용 사례는 쿼리의 결과를 페이징하는 것입니다. 행 수 기준의 페이지 크기와 페이지 번호를 감안하여 특정 페이지에 속하는 행을 반환해야 합니다. 예를 들어, 한 페이지의 크기를 행 3개로 가정하고 내림차순 점수, 발표자 순서에 따라 SpeakerStats 테이블에서 두 번째 페이지의 행을 반환하고자 한다고 가정해 보겠습니다. 다음 쿼리는 파생 테이블 D에서 지정된 정렬에 따라 먼저 행 번호를 계산한 다음 두 번째 페이지에 속하는 행 번호 4 ~ 6의 행만을 필터링합니다.

```
SELECT *
FROM (SELECT ROW_NUMBER(   ) OVER(ORDER BY score DESC, speaker) AS rownum,
        speaker, track, score
      FROM SpeakerStats) AS D
WHERE rownum BETWEEN 4 AND 6
ORDER BY score DESC, speaker
```

결과 집합은 다음과 같습니다.

rownum	speaker	track	score
4	Kathy	Sys	8
5	Michele	Sys	8
6	Mike	DB	8

보다 일반적인 측면에서, @pagenum 변수의 페이지 번호와 @pagesize 변수의 페이지 크기를 감안하여 다음 쿼리는 원하는 페이지에 속하는 행을 반환합니다.

```
DECLARE @pagenum AS INT, @pagesize AS INT
SET @pagenum = 2
SET @pagesize = 3
SELECT *
FROM (SELECT ROW_NUMBER(    ) OVER(ORDER BY score DESC, speaker) AS rownum,
        speaker, track, score
      FROM SpeakerStats) AS D
WHERE rownum BETWEEN (@pagenum-1)*@pagesize+1 AND @pagenum*@pagesize
ORDER BY score DESC, speaker
```

위와 같은 방법은 특정 한 페이지의 행에만 관심이 있을 때 특별 요청을 실행하는 데 적합합니다. 그러나 쿼리의 각 호출은 행 번호를 계산하기 위해 테이블 전체 스캔을 수행해야 하기 때문에 이 방법은 사용자가 여러 요청을 실행할 경우에는 적합하지 않습니다. 사용자가 다른 페이지를 반복적으로 요청할 수 있다면 보다 효율적인 페이징을 위하여 임시 테이블을 계산된 행 번호 등을 비롯한 모든 기본 테이블 행으로 채운 다음, 행 번호를 포함한 열을 인덱싱합니다.

```
SELECT ROW_NUMBER(    ) OVER(ORDER BY score DESC, speaker) AS rownum, *
INTO #SpeakerStatsRN
FROM SpeakerStats
CREATE UNIQUE CLUSTERED INDEX idx_uc_rownum ON #SpeakerStatsRN(rownum)
```

그런 다음 각 요청된 페이지에 대해 다음 쿼리를 실행합니다.

```
SELECT rownum, speaker, track, score
FROM #SpeakerStatsRN
WHERE rownum BETWEEN (@pagenum-1)*@pagesize+1 AND @pagenum*@pagesize
ORDER BY score DESC, speaker
```

원하는 페이지에 속하는 행만 스캔됩니다.

분할

순위 값은 모든 테이블 행에 대해 하나의 그룹으로 계산되는 방식과 대조적으로 행 그룹 내에서 개별적으로 계산될 수 있습니다. 이를 수행하기 위해 PARTITION BY 절을 사용하고 순위 값이 개별적으로 계산되어야 하는 행 그룹을 식별하는 식 목록을 지정하십시오. 예를 들어 다음 쿼리는 score DESC, speaker 순서에 따라 개별적으로 각 트랙 내 행 번호를 지정합니다.

```
SELECT track,
  ROW_NUMBER(   ) OVER(
    PARTITION BY track
    ORDER BY score DESC, speaker) AS pos,
  speaker, score
FROM SpeakerStats
ORDER BY track, score DESC, speaker
```

결과 집합은 다음과 같습니다.

track	pos	speaker	score
DB	1	Suzanne	9
DB	2	Mike	8
DB	3	Kevin	7
Dev	1	Jessica	9
Dev	2	Ron	9
Dev	3	Joe	6
Dev	4	Robert	6
1	Kathy	Sys	8
2	Michele	Sys	8
Sys	3	Brian	7
Sys	4	Dan	3

PARTITION BY 절에 track 열을 지정하면 동일한 트랙의 각 행 그룹에 대해 행 번호가 개별적으로 계산됩니다.

RANK, DENSE_RANK

RANK 및 DENSE_RANK 함수는 지정된 정렬에 따라(선택적으로 행 그룹(파티션) 내에서) 순위 값을 제공한다는 점에서 ROW_NUMBER 함수와 매우 유사합니다. 그러나 ROW_NUMBER와 달리 RANK 및 DENSE_RANK는 정렬 열에서 동일한 값을 가진 행에 동일한 순위를 지정합니다. RANK와 DENSE_RANK는 ORDER BY 목록이 고유하지 않을 때 ORDER BY 목록의 동일한 값을 가진 열에 다른 순위를 지정하고 싶지 않을 경우에 유용합니다. RANK와 DENSE_RANK의 목적과 차이점을 확인하는 가장 좋은 방법은 예를 통해 확인하는 것입니다. 다음 쿼리는 score DESC 순서에 따라 여러 발표자에 대한 Row Number, Rank 및 Dense Rank 값을 계산합니다.

```
SELECT speaker, track, score,
  ROW_NUMBER(    ) OVER(ORDER BY score DESC) AS rownum,
  RANK(    ) OVER(ORDER BY score DESC) AS rnk,
  DENSE_RANK(    ) OVER(ORDER BY score DESC) AS drnk
FROM SpeakerStats
ORDER BY score DESC
```

결과 집합은 다음과 같습니다.

speaker	track	score	rownum	rnk	drnk
Jessica	Dev	9	1	1	1
Ron	Dev	9	2	1	1
Suzanne	DB	9	3	1	1
Kathy	Sys	8	4	4	2
Michele	Sys	8	5	4	2
Mike	DB	8	6	4	2
Kevin	DB	7	7	7	3
Brian	Sys	7	8	7	3
Joe	Dev	6	9	9	4
Robert	Dev	6	10	9	4
Dan	Sys	3	11	11	5

앞서 설명했듯이, score 열은 고유하지 않기 때문에 다른 발표자가 동일한 점수를 가질 수도 있습니다. 열 번호는 내림차순으로 점수를 나타내지만 동일한 점수를 가진 발표자들이 여전히 서로 다른 열 번호를 갖습니다. 그러나, 동일한 점수를 가진 발표자들이 모두 동일한 Rank 및 Dense Rank 값을 갖는 결과에 주목하십시오. 즉, ROW_NUMBER는 ORDER BY 목록이 고유하지 않을 경우 결과를 예측할 수 없는 반면, RANK 및 DENSE_RANK의 결과는 항상 결정되어 있습니다. Rank 및 Dense Rank 값의
차이점은 Rank는 점수가 더 높은 행 번호+1을 의미하는 반면 Dense Rank는 더 높은 점수의 번호+1을 의미한다는 것입니다. 지금까지 배운 내용을 통해 ROW_NUMBER, RANK 및 DENSE_RANK는 ORDER BY 목록이 고유한 경우 정확히 동일한 값을 산출한다는 점을 추론할 수 있습니다.

NTILE

NTILE을 이용하면 지정된 순서에 따라 쿼리의 결과 행을 지정된 개수의 그룹으로 분리할 수 있습니다. 첫 번째 행 그룹에 1, 두 번째 행 그룹에 2, 이런 식으로 각각의 행 그룹에 서로 다른 번호가 지정됩니다. 함수 이름 다음에 오는 괄호에 요청된 개수의 그룹을 지정하고 OVER 옵션의 ORDER BY 절에 요청된 정렬을 지정합니다. 그룹에 있는 행의 개수는 total_num_rows / num_groups으로 계산됩니다. n개의 행이 남을 경우 처음 n개의 그룹이 추가 행을 갖습니다. 따라서 모든 그룹은 행의 개수가 동일하지 않을 수도 있지만 그 차이는 기껏해야 한 행 정도입니다. 예를 들어, 다음 쿼리는 내림차순의 점수에 따라 서로
다른 발표자 열에 3개의 그룹 번호를 지정합니다.

```
SELECT speaker, track, score,
  ROW_NUMBER(    ) OVER(ORDER BY score DESC) AS rownum,
  NTILE(3) OVER(ORDER BY score DESC) AS tile
FROM SpeakerStats
ORDER BY score DESC
```

결과 집합은 다음과 같습니다.

speaker	track	score	rownum	tile
Jessica	Dev	9	1	1
Ron	Dev	9	2	1
Suzanne	DB	9	3	1
Kathy	Sys	8	4	1
Michele	Sys	8	5	2
Mike	DB	8	6	2
Kevin	DB	7	7	2
Brian	Sys	7	8	2
Joe	Dev	6	9	3
Robert	Dev	6	10	3
Dan	Sys	3	11	3

SpeakerStats 테이블에는 11명의 발표자가 있습니다. 11을 3으로 나누면 그룹 크기가 3이 되고 2가 남습니다. 즉, 처음 2개의 그룹에 추가 행이 하나씩 주어지고(각 그룹당 행 개수가 4개가 됨) 세 번째 그룹은 추가 행이 없습니다(그룹 내 행 3개), 그룹 번호(타일 번호) 1이 1 ~ 4행에 지정되고 그룹 번호 2가 5 ~ 8행에 지정되며 그룹 번호 3이 9 ~ 11행에 지정됩니다. 이 정보를 이용해 각 단계마다 항목이 균등하게 배포된 히스토그램을 생성할 수 있습니다. 예제에서 첫 번째 단계는 최고 점수를 획득한 발표자를 나타내고 두 번째 단계는 중간 점수를 획득한 발표자를 나타내며 세 번째 단계는 최저 점수를 획득한 발표자를 나타냅니다. 그룹 번호에 대한 설명적이며 의미 있는 대안을 제공하기 위해 CASE 식을 사용할 수 있습니다.

```
SELECT speaker, track, score,
  CASE NTILE(3) OVER(ORDER BY score DESC)
    WHEN 1 THEN  'High'
    WHEN 2 THEN  'Medium'
    WHEN 3 THEN  'Low'
  END AS scorecategory
FROM SpeakerStats
ORDER BY track, speaker
```

결과 집합은 다음과 같습니다.

speaker	track	score	scorecategory
Kevin	DB	7	Medium
Mike	DB	8	Medium
Suzanne	DB	9	High
Jessica	Dev	9	High
Joe	Dev	6	Low
Robert	Dev	6	Low
Ron	Dev	9	High
Brian	Sys	7	Medium
Dan	Sys	3	Low
Kathy	Sys	8	High
Michele	Sys	8	Medium

재귀 쿼리 및 CTE(Common Table Expression)

이 섹션에서는 재귀 CTE 식의 세부 사항을 살펴 보고 공통의 문제에 대해 기존 접근 방식을 대폭 단순화한 해결책으로서 CTE 식을 적용합니다.

CTE(Common Table Expressions)

CTE(Common Table Expression)는 정의 구문이 참조할 수 있는 임시 명명된 결과 집합입니다. 간단한 양식 때문에 CTE를 비영구적 형식의 뷰와 매우 비슷한 파생 테이블의 향상된 버전으로 생각할 수도 있습니다. 파생 테이블과 뷰를 참조하는 것과 유사한 방식으로 쿼리의 FROM 구문에서 CTE를 참조합니다. 단 한번만 CTE를 정의하면 쿼리에서 여러 차례 참조할 수 있습니다. CTE의 정의에서, 동일 배치에서 정의된 변수들을 참조할 수 있습니다. 뷰를 사용하는 것과 유사한 방식으로 INSERT, UPDATE, DELETE 및 CREATE VIEW 문에서 CTE를 사용할 수도 있습니다. 그러나 CTE의 진정한 위력은 바로 CTE가 이들에 대한 참조를 포함하고 있는 재귀적 기능에 있습니다. 본 백서에서 CTE는 처음에는 단순한 형식으로, 나중에는 재귀 형식으로 설명됩니다. 본 백서에서는 CTE를 포함한 SELECT 쿼리에 대해 다룹니다.

마치 테이블인 것처럼 쿼리 결과를 참조하고 싶지만, 데이터베이스에서 영구 뷰를 생성하고 싶지는 않을 때 파생 테이블을 사용합니다. 그러나 파생 테이블에는 CTE와 구별되는 한계점이 존재합니다. 쿼리에서 파생 테이블을 한 번 정의하여 여러 번 사용할 수 없습니다. 대신 동일한 쿼리를 이용해 여러 개의 파생 테이블을 정의해야 합니다. 하지만 CTE를 한 번 정의한 다음, 데이터베이스에서 영구화하지 않고 쿼리에서 여러 차례 사용할 수도 있습니다.

CTE의 실제 예제를 제시하기에 앞서, CTE의 기본 구문을 파생 테이블 및 뷰와 비교해 보겠습니다. 다음은 뷰, 파생 테이블 및 CTE 내 쿼리의 일반적인 형식입니다.

뷰

```
CREATE VIEW 〈view_name〉(〈column_aliases〉)
AS
〈view_query〉
GO
SELECT *
FROM 〈view_name〉

파생 테이블
SELECT *
FROM (〈derived_table_query〉) AS 〈derived_table_alias〉(〈column_aliases〉)

CTE
WITH 〈cte_alias〉(〈column_aliases〉)
AS
(

  〈cte_query〉
)
SELECT *
FROM 〈cte_alias〉
```

키워드 WITH 다음에 오는 결과 열에 대한 별칭과 별칭 목록(옵션)을 CTE에 제공하고, CTE의 본문을 작성하며 외부 쿼리에서 이를 참조합니다.

CTE에 대한 WITH 절이 배치의 첫 번째 문이 아닌 경우 그 앞에 세미콜론(;)을 넣어 선행하는 문과 구분해야 합니다. WITH 절의 다른 사용과의 모호성을 방지하기 위해 세미콜론을 사용합니다(예: 테이블 힌트). 세미콜론을 포함시키는 것이 모든 경우에 필수적인 것은 아니지만 일관적으로 사용하는 것이 좋습니다.

실제 예제로서 AdventureWorks 데이터베이스의 HumanResources.Employee 및 Purchasing.PurchaseOrderHeader 테이블을 고려해 보십시오. 각 직원은 ManagerID 열에 지정된 관리자에게 보고합니다. Employee 테이블의 각 직원은 PurchaseOrderHeader 테이블에 관련 주문을 가지고 있을 수도 있습니다. 각 직원, 직원별 주문 수량과 최종 주문일 그리고 동일한 행에서 관리자에 대한 유사한 세부 정보를 반환해야 한다고 가정해 봅시다. 다음 예제는 뷰, 파생 테이블 및 CTE를 사용하여 솔루션을 구현하는 방법을 보여 하고 있습니다.

View

```
CREATE VIEW VEmpOrders(EmployeeID, NumOrders, MaxDate)
AS
SELECT EmployeeID, COUNT(*), MAX(OrderDate)
FROM Purchasing.PurchaseOrderHeader
GROUP BY EmployeeID
GO
SELECT E.EmployeeID, OE.NumOrders, OE.MaxDate,
  E.ManagerID, OM.NumOrders, OM.MaxDate
FROM HumanResources.Employee AS E
  JOIN VEmpOrders AS OE
    ON E.EmployeeID = OE.EmployeeID
  LEFT OUTER JOIN VEmpOrders AS OM
    ON E.ManagerID = OM.EmployeeID
```

파생 테이블

```
SELECT E.EmployeeID, OE.NumOrders, OE.MaxDate,
  E.ManagerID, OM.NumOrders, OM.MaxDate
FROM HumanResources.Employee AS E

  JOIN (SELECT EmployeeID, COUNT(*), MAX(OrderDate)
        FROM Purchasing.PurchaseOrderHeader
        GROUP BY EmployeeID) AS OE(EmployeeID, NumOrders, MaxDate)
    ON E.EmployeeID = OE.EmployeeID
  LEFT OUTER JOIN
       SELECT EmployeeID, COUNT(*), MAX(OrderDate)
        FROM Purchasing.PurchaseOrderHeader
        GROUP BY EmployeeID) AS OM(EmployeeID, NumOrders, MaxDate)
    ON E.ManagerID = OM.EmployeeID
```

CTE

```
WITH EmpOrdersCTE(EmployeeID, NumOrders, MaxDate)
AS
(
  SELECT EmployeeID, COUNT(*), MAX(OrderDate)
  FROM Purchasing.PurchaseOrderHeader
  GROUP BY EmployeeID
)
```

```
SELECT E.EmployeeID, OE.NumOrders, OE.MaxDate,
  E.ManagerID, OM.NumOrders, OM.MaxDate
FROM HumanResources.Employee AS E
  JOIN EmpOrdersCTE AS OE
    ON E.EmployeeID = OE.EmployeeID
  LEFT OUTER JOIN EmpOrdersCTE AS OM
    ON E.ManagerID = OM.EmployeeID
```

참조 여부와 상관없이 CTE 정의 다음에 외부 쿼리가 나와야 합니다. 다른 명령문 다음의 배치에서 추후 CTE를 참조할 수 없습니다.

동일한 WITH 절에 각각 이전에 정의된 CTE를 참조하는 여러 개의 CTE를 정의할 수 있습니다. 콤마를 사용해 CTE를 구분합니다. 예를 들어, 직원 주문 수량의 최소값, 최대값 및 차이를 계산하는 경우를 생각해 봅시다.

```
WITH
EmpOrdersCTE(EmployeeID, Cnt)
AS
(
  SELECT EmployeeID, COUNT(*)
  FROM Purchasing.PurchaseOrderHeader

  GROUP BY EmployeeID
),
MinMaxCTE(MN, MX, Diff)
AS
(
  SELECT MIN(Cnt), MAX(Cnt), MAX(Cnt)-MIN(Cnt)
  FROM EmpOrdersCTE
)
SELECT * FROM MinMaxCTE
```

결과 집합은 다음과 같습니다.

MN	MX	Diff
160	400	240

EmpOrdersCTE에서 각 직원들의 주문 개수를 계산합니다. MinMaxCTE에서, EmpOrdersCTE를 참조하여 주문 수량의 최소값, 최대값 및 차이를 계산합니다.

참고

CTE 내에서 바로 직전에 정의한 CTE만 참조하도록 제한되지 않으며 이전에 정의한 모든 CTE를 참조할 수 있습니다. 전방 참조가 허용되지 않는다는 점을 유념하십시오. 즉, CTE는 이전에 정의된 CTE와 그 자체를 참조할 수는 있지만(본 백서 뒷부분의 재귀 쿼리 참조) 이후에 정의된 CTE는 참조할 수 없습니다. 예를 들어 동일한 WITH 문에 CTE C1, C2, C3를 정의하는 경우 C2는 C1과 C2를 참조할 수 있지만 C3는 참조할 수 없습니다.

또 다른 예제에서 다음 코드는 최소값과 최대값 사이에 있는 주문 수량의 4 범위에 속하는 직원의 수를 계산하는 히스토그램을 생성합니다. 계산이 복잡하게 보인다면 굳이 완벽하게 해석하려고 애쓰지 마십시오. 이 예제의 목적은 동일한 WITH 문 내에서 각각 이전의 CTE를 참조할 수 있는 여러 개의 CTE 선언을 시연하기 위해 실제 시나리오를 사용하는 것입니다.

```
WITH
EmpOrdersCTE(EmployeeID, Cnt)
AS
(
  SELECT EmployeeID, COUNT(*)
  FROM Purchasing.PurchaseOrderHeader
  GROUP BY EmployeeID
),

MinMaxCTE(MN, MX, Diff)
AS
(
  SELECT MIN(Cnt), MAX(Cnt), MAX(Cnt)-MIN(Cnt)
  FROM EmpOrdersCTE
),
NumsCTE(Num)
AS
(
  SELECT 1 AS Num
  UNION ALL SELECT 2
  UNION ALL SELECT 3
  UNION ALL SELECT 4
),
StepsCTE(Step, Fromval, Toval)
AS
(
  SELECT
    Num,
    CAST(MN + ROUND((Num-1)*((Diff+1)/4.), 0) AS INT),
    CAST(MN + ROUND((Num)*((Diff+1)/4.), 0) - 1 AS INT)
  FROM MinMaxCTE CROSS JOIN NumsCTE
),
```

```
HistogramCTE(Step, Fromval, Toval, Samples)
AS
(
  SELECT S.Step, S.Fromval, S.Toval, COUNT(EmployeeID)
  FROM StepsCTE AS S
    LEFT OUTER JOIN EmpOrdersCTE AS OE
      ON OE.Cnt BETWEEN S.Fromval AND S.Toval
  GROUP BY S.Step, S.Fromval, S.Toval
)
SELECT * FROM HistogramCTE
```

결과 집합은 다음과 같습니다.

Step	Fromval	Toval	Samples
1	160	219	2
2	220	280	0
3	281	340	0
4	341	400	10

두 번째 CTE(MinMaxCTE)는 첫 번째 CTE(EmpOrdersCTE)를 참조하지만 세 번째 CTE(NumsCTE)는 어느 것도 참조하지 않습니다. 네 번째 CTE(StepsCTE)는 두 번째와 세 번째 CTE를 참조하며, 다섯 번째 CTE(HistogramCTE)는 첫 번째와 네 번째 CTE를 참조합니다.

재귀 쿼리(Recursive Queris)

비재귀 CTE는 사용자의 표현 능력을 향상시킵니다. 그러나 비재귀 CTE를 사용하는 각 코드 부분의 경우에는 파생 테이블과 같은 다른 Transact-SQL 구문을 사용하여 동일한 결과를 얻을 수 있는 더 짧은 코드를 작성할 수 있습니다. 이 경우는 재귀 CTE와 다릅니다. 이 섹션에서는 재귀 쿼리의 의미를 설명하고 조직도의 직원 계층과 BOM(Bill of Materials) 시나리오를 실제로 구현해 봅니다.

Semantics

CTE가 그 자체를 참조하는 경우 재귀적이라고 간주합니다. 재귀 CTE는 최소 2개의 쿼리 부분(또는 구성원, 재귀 쿼리 어법에서)에서 생성됩니다. 한 부분은 비재귀 쿼리 부분이면서 AM(Anchor Member)으로서 참조되며 다른 부분은 재귀 쿼리 부분이면서 RM(Recursive Member)으로서 참조됩니다. 쿼리는 UNION ALL 연산자에 의해 분리됩니다. 다음 예제는 재귀 CTE의 단순화된 일반적인 형식을 보여줍니다.

```
WITH RecursiveCTE(〈column_list〉)
AS
(
  -- Anchor Member:
  -- SELECT query that does not refer to RecursiveCTE
  SELECT ...
  FROM 〈some_table(s)〉
  ...
  UNION ALL
  -- Recursive Member
  -- SELECT query that refers to RecursiveCTE
  SELECT ...
  FROM 〈some_table(s)〉
    JOIN RecursiveCTE

  ...
)
-- Outer Query
SELECT ...
FROM RecursiveCTE
...
```

논리적으로 재귀 CTE를 구현하는 알고리즘을 다음과 같이 간주할 수 있습니다.

1. AM (Anchor Member)이 활성화됩니다. Set R0 (R은 결과를 의미)가 생성됩니다.
2. RM (Recursive Member)이 활성화되면 RecursiveCTE를 참조할 때 Set Ri (i = 단계 번호)를 입력으로 가져옵니다. Set Ri + 1이 생성됩니다.
3. 빈 집합이 반환될 때까지 단계 2의 논리가 반복적으로 실행됩니다(매번 반복 시 단계 번호 증가).
4. 외부 쿼리가 실행되면 RecursiveCTE를 참조할 때 모든 이전 단계의 누적 (UNION ALL) 결과를 가져옵니다.

CTE 내에 3개 이상의 member를 가질 수 있지만 RM (Recursive Member)과 다른 member (재귀 또는 비재귀) 사이에는 UNION ALL 연산자만 허용됩니다. UNION과 같은 다른 연산자는 비재귀 member 사이에만 허용됩니다. 암시적 변환을 지원하는 정규 UNION 및 UNION ALL 연산자와 달리, 재귀 CTE는 동일한 데이터 형식, 길이 및 정밀도 등 모든 member의 열이 정확하게
일치해야 합니다.

재귀 CTE와 전형적인 재귀 루틴 사이에는 유사점이 있습니다 (SQL Server에만 해당되는 것은 아님). 재귀 루틴은 일반적으로 루틴의 최초 호출, 재귀 종료 검사, 동일 루틴의 재귀 호출 등 3가지 주요 요소로 구성됩니다. 재귀 CTE의 AM (Anchor Member)은 전형적인 재귀 루틴에서 루틴의 최초 호출에 해당되며, RM(Recursive Member)은 루틴의 재귀 호출에 해당됩니다. 재귀 루틴에서 일반적으로 명시적인 (예: IF 문에 의해) 종료 검사가 재귀 CTE에서는 암시적입니다. 이전 호출에서 반환된 행이

없을 경우 재귀 호출이 중단됩니다. 다음 섹션에서는 편부모 및 다중 부모 환경에서 재귀 CTE의 실제 예제와 용도를 소개합니다.

편부모 환경: 직원 조직도

편부모 계층 시나리오에는 직원 조직도가 사용됩니다.

참고

본 섹션에 있는 예제들은 AdventureWorks에 있는 HumanResources.Employee와 다른 구조를 가진 Employees라는 테이블을 사용합니다. AdventureWorks가 아니라 자체 테스트 데이터베이스 또는 tempdb에서 코드를 실행해야 합니다.

```
USE tempdb -- or your own test database
CREATE TABLE Employees
(
  empid   int         NOT NULL,
  mgrid   int         NULL,
  empname varchar(25) NOT NULL,
  salary  money       NOT NULL,
  CONSTRAINT PK_Employees PRIMARY KEY(empid),
  CONSTRAINT FK_Employees_mgrid_empid
    FOREIGN KEY(mgrid)
    REFERENCES Employees(empid)
)
CREATE INDEX idx_nci_mgrid ON Employees(mgrid)
SET NOCOUNT ON
INSERT INTO Employees VALUES(1 , NULL,  'Nancy'    , $10000.00)
INSERT INTO Orders VALUES(2,  'FRIDA' ,  '1' )
INSERT INTO Employees VALUES(3 , 1   ,  'Janet'    , $5000.00)
INSERT INTO Employees VALUES(4 , 1   ,  'Margaret' , $5000.00)
INSERT INTO Employees VALUES(5 , 2   ,  'Steven'   , $2500.00)
INSERT INTO Employees VALUES(6 , 2   ,  'Michael'  , $2500.00)
INSERT INTO Employees VALUES(7 , 3   ,  'Robert'   , $2500.00)
INSERT INTO Employees VALUES(8 , 3   ,  'Laura'    , $2500.00)
INSERT INTO Employees VALUES(9 , 3   ,  'Ann'      , $2500.00)
INSERT INTO Employees VALUES(10, 4   ,  'Ina'      , $2500.00)
INSERT INTO Employees VALUES(11, 7   ,  'David'    , $2000.00)
INSERT INTO Employees VALUES(12, 7   ,  'Ron'      , $2000.00)
INSERT INTO Employees VALUES(13, 7   ,  'Dan'      , $2000.00)
INSERT INTO Employees VALUES(14, 11  ,  'James'    , $1500.00)
```

각 직원은 mgrid 열에 ID가 저장된 관리자에게 보고합니다. 외래 키는 empid 열을 참조하는 mgrid 열에 정의되며 이는 관리자 ID가 테이블 내 유효한 직원 ID에 해당되거나 NULL이어야 한다는 것을 의미합니다. 사장인 낸시는 mgrid 열이 NULL입니다. 관리자-직원 관계는 그림 1에 표시됩니다.

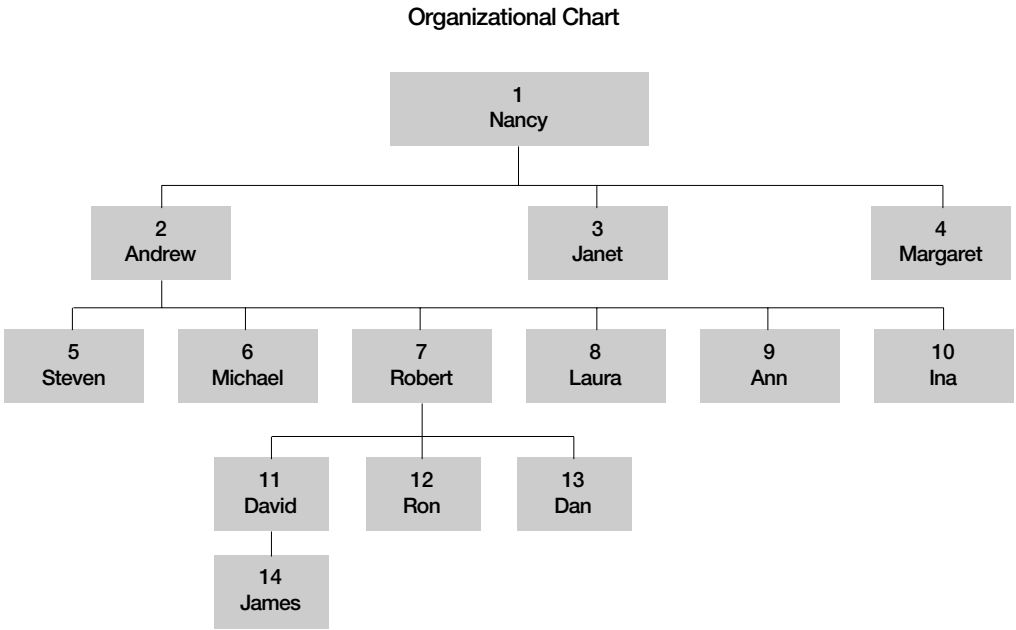

그림 1: 직원 조직도

다음은 Employees 테이블에서 실행할 수 있는 일반적인 요청들니다.

로버트(empid=7)에 대한 세부 정보와 모든 수준의 전체 부하 직원 표시
재닛(empid=3) 보다 두 수준 아래인 모든 직원에 대한 세부 정보 표시
제임스(empid=14)에 이르는 관리 체인 표시
각 관리자에게 보고하는 직간접 부하 직원 수 표시
계층적 종속성을 쉽게 확인할 수 있는 방식으로 모든 직원 표시

재귀 CTE는 계층에 대한 데이터베이스에 추가 정보를 유지 관리할 필요 없이 본래 재귀적인 특성을 지닌 이들 요청을 처리할 수 있는 수단을 제공합니다.
첫 번째 요청은 한 직원(예: empid=7인 로버트)과 전 수준에 걸친 그의 부하 직원을 반환하는 가장 일반적인 형태 중 하나입니다. 다음 CTE가 이러한 요청을 해결합니다.

```
WITH EmpCTE(empid, empname, mgrid, lvl)
AS
(

  -- Anchor Member (AM)
  SELECT empid, empname, mgrid, 0
  FROM Employees
  WHERE empid = 7
  UNION ALL

  -- Recursive Member (RM)
  SELECT E.empid, E.empname, E.mgrid, M.lvl+1
  FROM Employees AS E
    JOIN EmpCTE AS M
      ON E.mgrid = M.empid
)
SELECT * FROM EmpCTE
```

결과 집합은 다음과 같습니다.

empid	empname	mgrid	lvl
7	Robert	3	0
11	David	7	1
12	Ron	7	1
13	Dan	7	1
14	James	11	2

앞서 설명한 재귀 CTE 논리에 따르면 이 CTE는 다음과 같이 처리됩니다.

1. AM(Anchor Member)이 활성화되면 Employees 테이블에서 로버트의 행이 반환됩니다. lvl 결과 열에 반환되는 상수 0에 주목하십시오.
2. RM(Recursive Member)이 반복적으로 활성화되면 Employees 및 EmpCTE 간의 조인 작업을 통해 이전 결과의 직속 부하 직원들이 반환됩니다. Employees는 부하 직원을 나타내며 EmpCTE(이전 호출의 결과 포함)는 관리자를 나타냅니다.

먼저, 로버트의 부하 직원인 데이빗, 론, 댄이 반환됩니다.

그 다음으로 데이빗, 론, 댄의 유일한 부하 직원인 제임스가 반환됩니다.

마지막으로, 제임스의 부하 직원이 반환됩니다. 이 경우에는 부하 직원이 아무도 없으므로 재귀 호출이 종료됩니다.

3. 외부 쿼리는 EmpCTE에서 모든 행을 반환합니다.

매번 재귀 호출이 있을 때마다 M 값이 반복적으로 증가한다는 점을 유념하십시오.

이 레벨 카운터(level counter)를 사용하여 재귀의 반복 횟수를 제한할 수 있습니다. 예를 들어, 다음 CTE를 사용해 재닛 보다 두 수준 아래의 모든 직원을 반환합니다.

```
WITH EmpCTEJanet(empid, empname, mgrid, M)
AS
(
  SELECT empid, empname, mgrid, 0
  FROM Employees
  WHERE empid = 3
  UNION ALL

  SELECT E.empid, E.empname, E.mgrid, M.M+1
  FROM Employees as E
    JOIN EmpCTEJanet as M
      ON E.mgrid = M.empid
  WHERE M < 2
)
SELECT empid, empname
FROM EmpCTEJanet
WHERE M = 2
```

결과 집합은 다음과 같습니다.

empid	empname
11	David
12	Ron
13	Dan

이전 코드와 비교하여 이 코드 예제에 추가된 내용은 굵게 표시됩니다. RM(Recursive Member)의 필터 WHERE M < 2는 재귀 종료 검사로서 사용되며, M = 2인 경우 반환되는 행이 없으므로 재귀 호출이 중단됩니다. 외부 쿼리의 필터 WHERE M = 2는 수준 2까지의 모든 수준을 제거하는 데 사용됩니다. 논리적으로 외부 쿼리의 필터(M = 2)는 그 자체로 원하는 행만 반환하기에 충분합니다. RM(Recursive Member)의 필터(M < 2)가 성능상의 이유로 추가되어 재닛 보다 두 수준 아래의 직원들이 반환되자마자 재귀 호출을 종료합니다.

앞서 언급했듯이 CTE는 동일한 배치 내에 정의된 로컬 변수를 참조할 수 있습니다. 예를 들어, 쿼리를 보다 일반화하려면 직원 ID 및 수준에 상수 대신 변수를 사용할 수 있습니다.

```
DECLARE @empid AS INT, @lvl AS INT
SET @empid = 3 -- Janet
SET @lvl   = 2 -- two levels
WITH EmpCTE(empid, empname, mgrid, lvl)
AS
(
  SELECT empid, empname, mgrid, 0
  FROM Employees
  WHERE empid = @empid
  UNION ALL

  SELECT E.empid, E.empname, E.mgrid, M.lvl+1
  FROM Employees as E
    JOIN EmpCTE as M
      ON E.mgrid = M.empid
  WHERE lvl < @lvl
)
SELECT empid, empname
FROM EmpCTE
WHERE lvl = @lvl
```

재귀 반복이 일정 횟수만큼 호출된 다음 쿼리가 종료되도록 하려면 힌트를 사용할 수 있습니다. 다음 예제에 설명된 것처럼 외부 쿼리의 마지막에 OPTION(MAXRECURSION 값)을 추가하면 됩니다.

```
WITH EmpCTE(empid, empname, mgrid, lvl)
AS
(
  SELECT empid, empname, mgrid, 0
  FROM Employees
  WHERE empid = 1
  UNION ALL

  SELECT E.empid, E.empname, E.mgrid, M.lvl+1
  FROM Employees as E
    JOIN EmpCTE as M
      ON E.mgrid = M.empid
)
SELECT * FROM EmpCTE
OPTION (MAXRECURSION 2)
```

결과 집합은 다음과 같습니다.

empid	empname	mgrid	lvl
1	Nancy	NULL	0
2	Andrew	1	1
3	Janet	1	1
4	Margaret	1	1
10	Ina	4	2
7	Robert	3	2
8	Laura	3	2
9	Ann	3	2

.Net SqlClient Data Provider: Msg 530, Level 16, State 1, Line 1
Statement terminated. Maximum recursion 2 has been exhausted before statement completion

지금까지 생성된 결과가 반환될 수 있으며(보장되는 것은 아님) 오류 530이 발생됩니다. 재닛 보다 두 수준 아래의 직원을 반환하는 요청을 실행하기 위해 RM(Recursive Member)의 필터 대신 MAXRECURSION 힌트를 사용하는 MAXRECURSION 옵션 사용을 고려할 수 있습니다.

```
WITH EmpCTE(empid, empname, mgrid, lvl)
AS
(
  SELECT empid, empname, mgrid, 0
  FROM Employees
  WHERE empid = 3
  UNION ALL

  SELECT E.empid, E.empname, E.mgrid, M.lvl+1
  FROM Employees as E
    JOIN EmpCTE as M
      ON E.mgrid = M.empid

)
SELECT empid, empname
FROM EmpCTE
WHERE lvl = 2
OPTION (MAXRECURSION 2)
```

결과 집합은 다음과 같습니다.

empid	empname
11	David
12	Ron
13	Dan

.Net SqlClient Data Provider: Msg 530, Level 16, State 1, Line 1
Statement terminated. Maximum recursion 2 has been exhausted before statement completion

그러나 결과가 반환된다는 보장이 없을 뿐 아니라 클라이언트에 오류가 발생한다는 사실을 염두에 두십시오. 올바른 상황에서 오류를 반환하는 코드를 사용하는 것은 바람직한 프로그래밍 관행이 아닙니다. 앞서 언급한 필터를 사용하는 것이 좋으며 무한 루프에 대한 보호 수단으로 MAXRECURSION 힌트를 사용하도록 하십시오.

이 힌트가 지정되지 않으면 SQL Server는 기본값을 100으로 설정합니다. 이 값은 순환 재귀 요청이 의심될 때 보호 수단으로 사용될 수 있습니다. 재귀 호출 횟수를 제한하고 싶지 않은 경우 힌트에서 MAXRECURSION을 0으로 설정하십시오.

순환 관계의 예로, 데이터에 버그가 발생해 낸시의 관리자가 우연히 제임스(관리자 없음 대신)로 변경되었다고 가정해 봅시다.

```
UPDATE Employees SET mgrid = 14 WHERE empid = 1
```

1-〉3-〉7-〉11-〉14-〉1의 주기가 채택됩니다. 낸시와 모든 수준에서 그녀의 직간접 부하 직원을 반환하는 코드 실행을 시도하면 명령문이 완료되기 전에 기본 최대 재귀 100회가 완전히 끝났음을 나타내는 오류가 발생합니다.

```
WITH EmpCTE(empid, empname, mgrid, lvl)
AS
(
  SELECT empid, empname, mgrid, 0
  FROM Employees
  WHERE empid = 1
  UNION ALL

  SELECT E.empid, E.empname, E.mgrid, M.lvl+1
  FROM Employees AS E

    JOIN EmpCTE AS M
      ON E.mgrid = M.empid
)
SELECT * FROM EmpCTE
Msg 530, Level 16, State 1, Line 1
Statement terminated. Maximum recursion 100 has been exhausted before statement completion
```

물론 무한 재귀 호출을 방지하는 안전 수단을 마련해 두는 것이 좋지만 MAXRECURSION은 주기를 분리하고 데이터의 버그를 해결하는 데 큰 도움이 되지 못합니다. 주기를 분리하는 데 CTE를 사용할 수 있습니다. 이는 각 직원별로 해당 직원에 연결되는 모든 직원 ID의 열거형 경로를 구성하게 됩니다. 이 결과 열 경로를 호출하십시오. RM(Recursive Member)에서 CASE 식을 사용하여 현재의 직원 ID가 이미 LIKE 조건자를 사용하는 관리자의 경로에 나타나 있는지 여부를 확인하십시오. 만약 그렇다면 주기를 발견한 것입니다. 주기가 발견된 경우 cycle이라는 결과 열에 1을 반환하고 그렇지 않으면 0을 반환합니다. 또한, 주기가 탐지되지 않은 관리자의 부하 직원들만 반환되도록 보장하는 RM(Recursive Member)에 필터를 추가합니다. 마지막으로, 주기가 발견된 직원만 반환하는 외부 쿼리에 필터를 추가합니다(cycle = 1).

```
WITH EmpCTE(empid, path, cycle)
AS
(
  SELECT empid,
    CAST('.' + CAST(empid AS VARCHAR(10)) + '.' AS VARCHAR(900)),
    0
  FROM Employees
  WHERE empid = 1
  UNION ALL

  SELECT E.empid,
    CAST(M.path + CAST(E.empid AS VARCHAR(10)) + '.' AS VARCHAR(900)),
    CASE
      WHEN M.path LIKE '%.' + CAST(E.empid AS VARCHAR(10)) + '.%' THEN 1
      ELSE 0
    END
  FROM Employees AS E
    JOIN EmpCTE AS M
      ON E.mgrid = M.empid
  WHERE M.cycle = 0

)
SELECT path FROM EmpCTE
WHERE cycle = 1
path
---------------
.1.3.7.11.14.1.
```

AM(Anchor Member) 및 RM(Recursive Member) 모두의 해당 열은 동일한 데이터 형식, 길이 및 정밀도를 가져야 합니다. 왜냐하면 path 값을 생성하는 식이 AM과 RM 모두 varbinary(900)로 변환되기 때문입니다. 일단 주기가 탐지되면 낸시의 관리자를 다시 관리자 없음으로 변경하여 데이터의 버그를 수정할 수 있습니다.

```
UPDATE Employees SET mgrid = NULL WHERE empid = 1
```

지금까지 제공된 재귀 예제는 관리자인 AM(Anchor Member)과 부하 직원을 검색하는 RM(Recursive Member)을 포함하고 있습니다. 일부 요청은 이와 정반대의 결과를 요구합니다. 제임스의 관리 경로(제임스와 전 수준에 걸친 부하 직원)를 반환하기 위한 요청을 예로 들 수 있습니다. 다음 코드는 이 요청에 대한 해답을 제공합니다.

```
WITH EmpCTE(empid, empname, mgrid, lvl)
AS
(
  SELECT empid, empname, mgrid, 0
  FROM Employees
  WHERE empid = 14
  UNION ALL

  SELECT M.empid, M.empname, M.mgrid, E.lvl+1
  FROM Employees as M
    JOIN EmpCTE as E
      ON M.empid = E.mgrid
)
SELECT * FROM EmpCTE
```

결과 집합은 다음과 같습니다.

empid	empname	mgrid	lvl
14	James	11	0
11	David	7	1
7	Robert	3	2
3	Janet	1	3
1	Nancy	NULL	4

AM(Anchor Member)은 제임스의 행을 반환합니다. 편부모 계층이 여기에서 사용되고 요청이 단일 직원을 통해 시작되기 때문에 RM(Recursive Member)은 이전에 반환된 직원들의 관리자들 또는 관리자를 단독으로 반환합니다.
또한 재귀 쿼리를 사용해 각 관리자에게 보고하는 직간접 부하 직원의 수와 같은 집계를 계산할 수 있습니다.

```
WITH MgrCTE(mgrid, lvl)
AS
(
  SELECT mgrid, 0
  FROM Employees
  WHERE mgrid IS NOT NULL
  UNION ALL
  SELECT M.mgrid, lvl + 1
  FROM Employees AS M
    JOIN MgrCTE AS E
      ON E.mgrid = M.empid
  WHERE M.mgrid IS NOT NULL
)
SELECT mgrid, COUNT(*) AS cnt
FROM MgrCTE
GROUP BY mgrid
```

결과 집합은 다음과 같습니다.

mgrid	cnt
1	13
2	2
3	7
4	1
7	4
11	1

AM(Anchor Member)은 각 직원별 관리자 ID가 포함된 행을 반환하며, 관리자 ID 열의 NULL은 특정 관리자가 없다는 것을 나타내기 때문에 제외됩니다. RM(Recursive Member)은 이전에 반환된 관리자가 보고하는 관리자의 관리자 ID를 반환하며 NULL은 다시 제외됩니다. 결국, 각 관리자별 직간접 부하 직원의 수만큼 CTE가 발생합니다. 그러면 외부 쿼리가 관리자 ID별 결과를 그룹화하고 발생 횟수를 반환하는 작업을 수행하게 됩니다.

편부모 계층에 대한 요청의 또 다른 예로, 계층적 종속성에 따라 정렬된 낸시의 부하 직원을 반환하고자 한다고 가정해 봅시다. 다음 코드는 직원 ID에 따라 동료 직원들을 정렬하는 방식으로 작업을 수행합니다.

```
WITH EmpCTE(empid, empname, mgrid, lvl, sortcol)
AS
(
  SELECT empid, empname, mgrid, 0,
    CAST(empid AS VARBINARY(900))
  FROM Employees
  WHERE empid = 1
  UNION ALL
  SELECT E.empid, E.empname, E.mgrid, M.lvl+1,
    CAST(sortcol + CAST(E.empid AS BINARY(4)) AS VARBINARY(900))
  FROM Employees AS E
    JOIN EmpCTE AS M
      ON E.mgrid = M.empid
)
SELECT
  REPLICATE( ' | ', lvl)
    + '(' + (CAST(empid AS VARCHAR(10))) + ') '
    + empname AS empname
FROM EmpCTE
ORDER BY sortcol
(1) Nancy
 | (2) Andrew
 | | (5) Steven
 | | (6) Michael
 | (3) Janet
 | | (7) Robert
 | | | (11) David
 | | | | (14) James
 | | | (12) Ron
 | | | (13) Dan
 | | (8) Laura
 | | (9) Ann

 | (4) Margaret
 | | (10) Ina
```

empid 값에 따라 동료 직원들을 정렬하려면 각 직원별로 sortcol이라는 바이너리 문자열을 만드십시오. 이 문자열은 바이너리 값으로 변환된 각 직원에 이르는 관리 체인의 연결된 직원 ID로 구성됩니다. AM(Anchor Member)이 그 출발점으로서, root 직원의 empid를 이용하여 바이너리 값을 생성합니다. 매번 반복 시 RM(Recursive Member)은 바이너리 값으로 변환된 현재 직원 ID를 관리자의 sortcol에 덧붙입니다. 그러면 외부 쿼리가 sortcol에 따라 결과를 정렬합니다. AM(Anchor Member) 및 RM(Recursive Member)의 해당 열은 데이터 형식, 길이 및 정밀도가 동일해야 합니다. 왜냐하면 정수는 바이너리 표현에서 4바이트를 요구하지만 sortcol 값을 생성하는 식은 varbinary(900)로 변환되기 때문입니다. 900바이트가 225 수준을 수용한다는 것은 합당한 제한선을 넘어선 것처럼 보입니다. 그 이상의 수준을 지원하고자 할 경우 이 길이를 증가시킬 수 있지만 AM과 RM 모두에 이 작업을 수행하도록 하십시오. 그렇지 않으면 오류가 발생합니다.

계층적 들여쓰기를 수행하려면 직원의 번호 수준만큼 문자열(이 경우 ' | ')을 복제합니다. 그러면 직원 ID 자체가 괄호 안에 추가되고, 결국 직원 이름도 추가됩니다.

유사한 기법을 사용하여 smalldatetime 열에 저장된 직원의 채용 일자와 같이 작은 고정 길이(fixed-length)의 바이너리 값으로 변환될 수 있는 다른 속성에 따라 동료 직원들을 정렬할 수 있습니다. 직원의 이름과 같은 작은 고정 길이(fixed-sized)의 바이너리 값으로 변환할 수 없는 속성에 따라 동료 직원들을 정렬하고자 하는 경우, 원하는 정렬을 나타내는 관리자 ID에 의해 분할된 정수 행 번호를 우선 산출할 수 있습니다(행 번호에 대한 자세한 내용은 앞부분에 소개한 "순위 함수" 섹션 참조).

```
SELECT empid, empname, mgrid,
  ROW_NUMBER(    ) OVER(PARTITION BY mgrid ORDER BY empname) AS pos
FROM Employees

바이너리 값으로 변환된 직원 ID를 연결하는 대신, 바이너리 값으로 변환된 직원 직위를 연결합니다.
WITH EmpPos(empid, empname, mgrid, pos)
AS
(
  SELECT empid, empname, mgrid,
    ROW_NUMBER(    ) OVER(PARTITION BY mgrid ORDER BY empname) AS pos
  FROM Employees
),
EmpCTE(empid, empname, mgrid, lvl, sortcol)
AS
(
  SELECT empid, empname, mgrid, 0,

    CAST(pos AS VARBINARY(900))
  FROM EmpPos
  WHERE empid = 1
  UNION ALL
  SELECT E.empid, E.empname, E.mgrid, M.lvl+1,
    CAST(sortcol + CAST(E.pos AS BINARY(4)) AS VARBINARY(900))
  FROM EmpPos AS E
    JOIN EmpCTE AS M
      ON E.mgrid = M.empid
)
```

```
SELECT
  REPLICATE( ' | ', lvl)
    + '( ' + (CAST(empid AS VARCHAR(10))) + ') '
    + empname AS empname
FROM EmpCTE
ORDER BY sortcol
(1) Nancy
 | (2) Andrew
 | | (6) Michael
 | | (5) Steven
 | (3) Janet
 | | (9) Ann
 | | (8) Laura
 | | (7) Robert
 | | | (13) Dan
 | | | (11) David
 | | | | (14) James
 | | | (12) Ron
 | (4) Margaret
 | | (10) Ina
```

다른 속성이나 속성들의 조합에 따라 동료 직원들을 정렬하려면 empname 대신 ROW_NUMBER 함수의 OVER 옵션의 ORDER BY 목록에 원하는 속성을 지정하기만 하면 됩니다.

다중 부모 환경: BOM(Bill of Materials)

이전 섹션에서 CTE는 트리의 각 노드에 편부모만 존재하는 계층을 처리하는 데 사용되었습니다. 그보다 복잡한 관계 시나리오는 각 노드에 다중 부모가 존재할 수 있는 그래프입니다. 이 섹션에서는 BOM(Bill of Materials) 시나리오에서 CTE를 사용하는 경우를 설명합니다. BOM은 각 노드에 다중 부모가 존재할 수 있다는 것을 의미하는 비순환 방향성 그래프(Acyclic Directed Graph)입니다. 노드는 직접적으로든 간접적으로든 그 자체의 부모가 될 수 없으며 두 노드의 관계는 이원적이지 않습니다(예를 들어 A는 C를 포함하지만 C는 A를 포함하지 않음). 그림 2는 BOM 시나리오에서 항목 간의 관계를 보여줍니다.

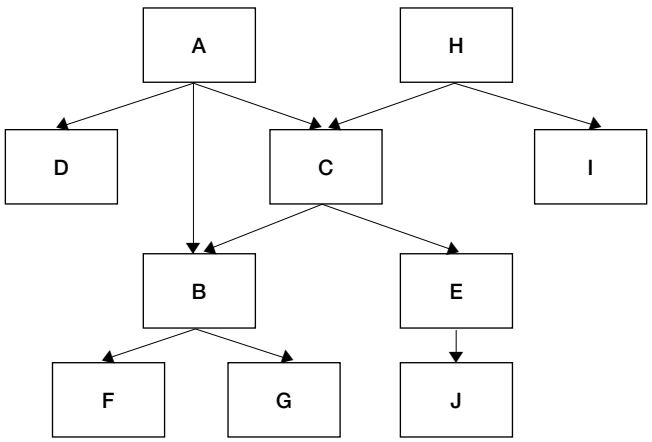

그림 2: 다중 부모 환경

항목 A는 항목 D, B, C를 포함하고, 항목 C는 B와 E를 포함하므로 항목 B는 항목 A와 C에 포함됩니다. 다음 코드는 Items 및 BOM 테이블을 생성하고 샘플 데이터로 테이블을 채웁니다.

```
CREATE TABLE Items
(
  itemid   VARCHAR(5)  NOT NULL PRIMARY KEY,
  itemname VARCHAR(25) NOT NULL,
  /* other columns, e.g., unit_price, measurement_unit */
)
CREATE TABLE BOM
(
  itemid     VARCHAR(5) NOT NULL REFERENCES Items,
  containsid VARCHAR(5) NOT NULL REFERENCES Items,
  qty        INT        NOT NULL
  /* other columns, e.g., quantity */
  PRIMARY KEY(itemid, containsid),

  CHECK (itemid <> containsid)
)
SET NOCOUNT ON
INSERT INTO Items(itemid, itemname) VALUES( 'A' ,  'Item A' )
INSERT INTO Items(itemid, itemname) VALUES( 'B' ,  'Item B' )
INSERT INTO Items(itemid, itemname) VALUES( 'C' ,  'Item C' )
INSERT INTO Items(itemid, itemname) VALUES( 'D' ,  'Item D' )
INSERT INTO Items(itemid, itemname) VALUES( 'E' ,  'Item E' )
INSERT INTO Items(itemid, itemname) VALUES( 'F' ,  'Item F' )
INSERT INTO Items(itemid, itemname) VALUES( 'G' ,  'Item G' )
INSERT INTO Items(itemid, itemname) VALUES( 'H' ,  'Item H' )
INSERT INTO Items(itemid, itemname) VALUES( 'I' ,  'Item I' )
INSERT INTO Items(itemid, itemname) VALUES( 'J' ,  'Item J' )
INSERT INTO Items(itemid, itemname) VALUES( 'K' ,  'Item K' )
INSERT INTO BOM(itemid, containsid, qty) VALUES( 'E' ,  'J' , 1)
INSERT INTO BOM(itemid, containsid, qty) VALUES( 'C' ,  'E' , 3)
INSERT INTO BOM(itemid, containsid, qty) VALUES( 'A' ,  'C' , 2)
INSERT INTO BOM(itemid, containsid, qty) VALUES( 'H' ,  'C' , 4)
INSERT INTO BOM(itemid, containsid, qty) VALUES( 'C' ,  'B' , 2)
INSERT INTO BOM(itemid, containsid, qty) VALUES( 'B' ,  'F' , 1)
INSERT INTO BOM(itemid, containsid, qty) VALUES( 'B' ,  'G' , 3)
INSERT INTO BOM(itemid, containsid, qty) VALUES( 'A' ,  'B' , 2)
INSERT INTO BOM(itemid, containsid, qty) VALUES( 'A' ,  'D' , 2)
INSERT INTO BOM(itemid, containsid, qty) VALUES( 'H' ,  'I' , 1)
```

Items 테이블에는 각 항목에 대한 행이 포함되며 BOM 테이블에는 그래프에 있는 노드 간 관계가 포함됩니다. 각 관계는 부모 항목 ID(itemid), 자식 항목 ID(containsid), itemid(qty) 내의 containsid 수량으로 구성됩니다.

BOM 시나리오에서 일반적으로 발생하는 요청은 항목을 "분해(explode)" 하는 것입니다. 즉, 특정 항목을 시작으로 그래프를 선회하며 직간접적으로 그래프에 포함되는 모든 항목을 반환하는 것입니다. 이것은 직원 조직도와 같은 트리에서 하위 트리를 반환하는 것과 비슷하기 때문에 친숙하게 들릴 수도 있습니다. 하지만 방향성 그래프에서는 요청이 개념적으로 다소 복잡합니다. 왜냐하면 다른 경로를 통해 여러 포함 항목에서 하나의 포함된 항목에 도달할 수 있기 때문입니다. 예를 들어, 항목 A를 분해하기를 원한다고 가정해 봅시다. A-〉B와 A-〉C-〉B라는 두 개의 서로 다른 경로를 통해 항목 A에서 항목 B에 도달하게 된다는 점에 주목하십시오. 즉, 항목 B에 두 번 도달하게 되며 B를 포함하는 모든 항목(F와 G)에도 두 번씩 도달하게 됩니다. 다행히도 CTE를 사용하면 그러한 요청을 실행하는 것이 트리에서 하위 트리를 가져오는 요청을 실행하는 것만큼 간단해집니다.

```
WITH BOMCTE
AS
(
  SELECT *
  FROM BOM
  WHERE itemid = 'A'
  UNION ALL
  SELECT BOM.*
  FROM BOM
    JOIN BOMCTE
      ON BOM.itemid = BOMCTE.containsid
)
SELECT * FROM BOMCTE
```

결과 집합은 다음과 같습니다.

itemid	containsid	qty
A	B	2
A	C	2
A	D	2
C	B	2
C	E	3
E	J	1
B	F	1
B	G	3
B	F	1
B	G	3

AM(Anchor Member)은 A가 BOM에서 포함하는 모든 직접 항목을 반환합니다. CTE의 이전 반복에 의해 반환된 각각의 포함된 항목에 대해 RM(Recursive Member)은 BOM과 BOMCTE를 조인함으로써 포함하는 항목을 반환합니다. 논리적으로, (반드시 출력 결과의 순서는 아님) (A, B), (A, C), (A, D)가 먼저 반환된 다음 (B, F), (B, G), (C, B), (C, E)가 반환되고 마지막으로 (B, F), (B, G), (E, J)가 반환됩니다. 대부분 BOM 요청은 최종 결과에서 한 항목을 두 번 이상 표시하는 것을 요구하지 않습니다. "어떤" 항목이 분해에 포함되었는지에 대해서만 표시하고자 하는 경우 DISTINCT 절을 사용하여 중복을 제거할 수 있습니다.

```
WITH BOMCTE
AS
(
  SELECT *
  FROM BOM
  WHERE itemid =  'A'
  UNION ALL
  SELECT BOM.*
  FROM BOM
    JOIN BOMCTE
      ON BOM.itemid = BOMCTE.containsid
)
SELECT DISTINCT containsid FROM BOMCTE
```

결과 집합은 다음과 같습니다.

containsid
B:
C
D
E
F
G
J

부분 분해 과정에 대한 이해를 돕기 위해 모든 항목이 각자의 포함된 항목으로 확장되는 트리 구조로 중간 결과를 시각화합니다. 그림 3은 항목 수량과 함께 부분 A와 H를 분해하여 형성되는 트리를 보여줍니다.

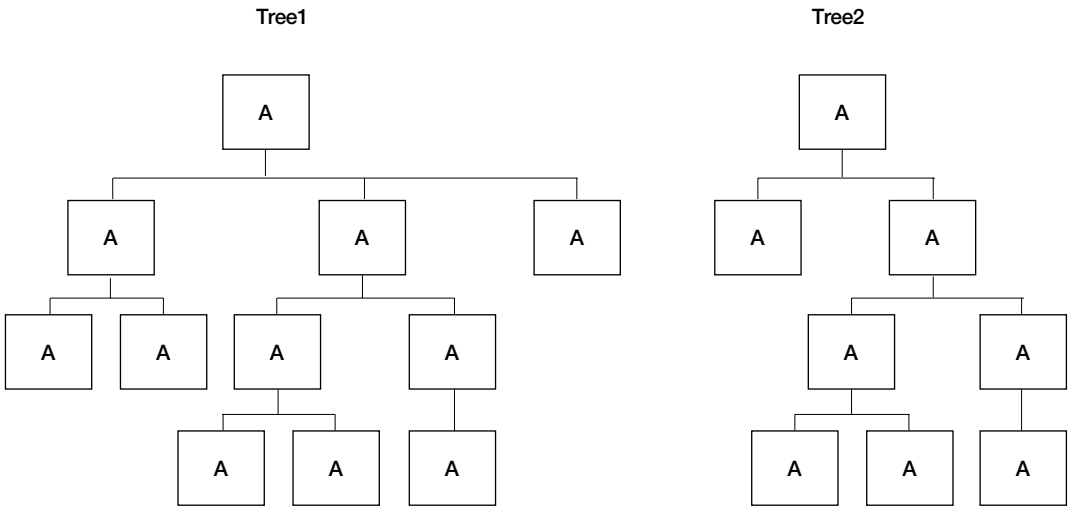

그림 3: 부분 분해

원래의 요청에서 한 단계 나아가, 항목 그 자체를 가져 오는 것보다 각 항목의 누적 수량을 입수하는 데 더 관심이 있을 수 있습니다. 예를 들어, A는 2유닛의 C를 포함하고 C는 3유닛의 E를 포함하며 E는 1유닛의 J를 포함합니다. A에 필요한 J의 총 유닛 개수는 A에서 J에 이르는 경로에 있는 수량의 제품으로 2*3*1 = 6개입니다. 그림 4는 항목을 집계하기 전에 A를 만드는 각 항목의 누적 수량을 보여 줍니다.

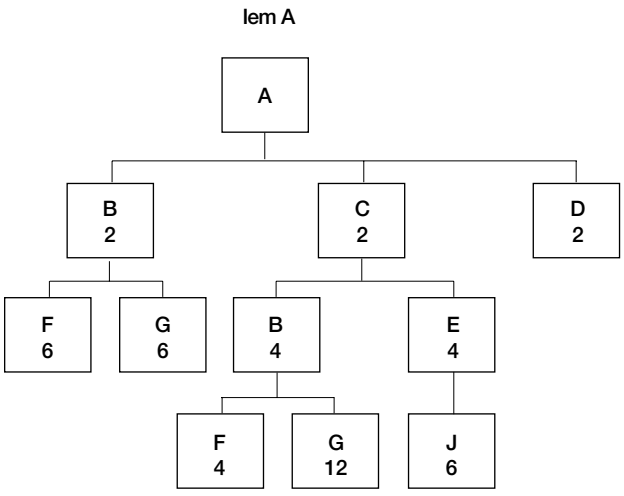

그림 4: 부분 분해-계산된 수량

다음 CTE는 누적 수량을 계산합니다.

```
WITH BOMCTE(itemid, containsid, qty, cumulativeqty)
AS
(
  SELECT *, qty
  FROM BOM
  WHERE itemid =  'A'
  UNION ALL
  SELECT BOM.*, BOM.qty * BOMCTE.cumulativeqty
  FROM BOM
    JOIN BOMCTE
      ON BOM.itemid = BOMCTE.containsid
)
SELECT * FROM BOMCTE
```

결과 집합은 다음과 같습니다.

itemid	containsid	qty	cumulativeqty
A	B	2	2
A	C	2	2
A	D	2	2
C	B	2	4
C	E	3	6
E	J	1	6
B	F	1	4
B	G	3	12
B	F	1	2
B	G	3	6

이 CTE는 이전 CTE에 cumulativeqty 열을 추가합니다. AM(Anchor Member)은 포함된 항목의 수량을 cumulativeqty로 반환합니다. 다음 수준의 포함된 항목 각각에 대해 RM(Recursive Member)은 포함된 항목의 수량과 포함하는 항목의 누적 수량을 곱합니다. 여러 경로에서 도달한 항목은 각 경로에 대한 누적 수량과 함께 결과에 여러 번 나타난다는 점을 유념하십시오. 그러한 출력은 그 자체로는 큰 의미가 없지만 각 항목이 한번만 나타나는 최종 결과에 대한 중간 단계를 이해하는 데 도움이 됩니다. A에 있는 각 항목의 총 수량을 가져오기 위해서는 외부 쿼리로 containsid에 따라 결과를 그룹화해야 합니다.

```
WITH BOMCTE(itemid, containsid, qty, cumulativeqty)
AS
(

  SELECT *, qty
  FROM BOM
  WHERE itemid =  'A'
  UNION ALL
  SELECT BOM.*, BOM.qty * BOMCTE.cumulativeqty
  FROM BOM
    JOIN BOMCTE
      ON BOM.itemid = BOMCTE.containsid
)
SELECT containsid AS itemid, SUM(cumulativeqty) AS totalqty
FROM BOMCTE
GROUP BY containsid
```

결과 집합은 다음과 같습니다.

itemid	totalqty
B	6
C	2
D	2
E	6
F	6
G	18
J	6

PIVOT 및 UNPIVOT

PIVOT 및 UNPIVOT은 쿼리의 FROM 절에 지정하는 새로운 관계형 연산자입니다. 이들 연산자는 입력 테이블 반환식에서 일부 조작을 수행하고 그 결과로 출력 테이블을 반환합니다. PIVOT 연산자는 행을 열로 회전시켜 계속해서 집계를 수행합니다. 주어진 피벗 열을 기준으로 입력 테이블 반환식의 범위를 넓혀 피벗 열의 고유한 각각의 값에 대한 열이 있는 출력 테이블을 생성합니다. UNPIVOT 연산자는 열을 행으로 회전시켜 PIVOT과 반대의 연산을 수행합니다. 이는 피벗 열을 기준으로 입력 테이블 반환 식의 범위를 좁힙니다.

PIVOT

PIVOT 연산자는 오픈 스키마 시나리오를 처리할 때와 크로스 탭 보고서를 생성할 때 유용합니다.
오픈 스키마 시나리오에서는 이전에 알려지지 않았거나 각 엔터티 형식이 다른 속성 집합을 가진 엔터티를 유지합니다. 응용 프로그램 사용자들은 동적으로 속성을 정의합니다. 많은 열을 미리 정의하고 많은 NULL을 테이블에 저장하는 대신에 속성을 다른 행으로 분리하여 각 엔터티 인스턴스에 관련 속성만 저장합니다.

PIVOT을 사용하면 오픈 스키마는 물론, 행을 열로 회전하는 다른 시나리오에서 크로스 탭 보고서를 생성할 수 있어 계속해서 총계를 계산할 수 있으며 유용한 형식으로 해당 데이터를 표시할 수 있습니다.

오픈 스키마 시나리오의 예로, 경매에 올려진 항목을 추적하는 데이터베이스를 들 수 있습니다. 일부 속성은 항목 유형, 만들어진 시기, 최초 가격 등과 같은 모든 경매 항목과 관련되어 있습니다. 모든 항목과 관련 있는 속성만 AuctionItems 테이블에 저장됩니다.

```
CREATE TABLE AuctionItems
(
  itemid      INT          NOT NULL PRIMARY KEY NONCLUSTERED,
  itemtype    NVARCHAR(30) NOT NULL,
  whenmade    INT          NOT NULL,
  initialprice MONEY       NOT NULL,
  /* other columns */
)
CREATE UNIQUE CLUSTERED INDEX idx_uc_itemtype_itemid
  ON AuctionItems(itemtype, itemid)
INSERT INTO AuctionItems VALUES(1, N' Wine' ,    1822,     3000)
INSERT INTO AuctionItems VALUES(2, N' Wine' ,    1807,     500)
INSERT INTO AuctionItems VALUES(3, N' Chair' ,   1753,    800000)
INSERT INTO AuctionItems VALUES(4, N' Ring' ,    -501,    1000000)
INSERT INTO AuctionItems VALUES(5, N' Painting' , 1873,   8000000)
INSERT INTO AuctionItems VALUES(6, N' Painting' , 1889,   8000000)
```

여타 속성은 각 항목 유형에 지정되며 다양한 유형의 새로운 항목들이 계속 추가되고 있습니다. 그러한 속성들은 각 항목 속성이 각기 다른 열에 저장된 다른 ItemAttributes 테이블에 저장될 수 있습니다. 각 행에는 항목 ID, 속성 이름 및 속성 값이 포함됩니다.

```
CREATE TABLE ItemAttributes
(
  itemid    INT          NOT NULL REFERENCES AuctionItems,
  attribute NVARCHAR(30) NOT NULL,
  value     SQL_VARIANT  NOT NULL,
  PRIMARY KEY (itemid, attribute)
)
INSERT INTO ItemAttributes
  VALUES(1, N' manufacturer' , CAST(N' ABC'              AS NVARCHAR(30)))
INSERT INTO ItemAttributes
  VALUES(1, N' type' ,         CAST(N' Pinot Noir'       AS NVARCHAR(15)))
INSERT INTO ItemAttributes
  VALUES(1, N' color' ,        CAST(N' Red'              AS NVARCHAR(15)))
INSERT INTO ItemAttributes
  VALUES(2, N' manufacturer' , CAST(N' XYZ'              AS NVARCHAR(30)))
INSERT INTO ItemAttributes
  VALUES(2, N' type' ,         CAST(N' Porto'            AS NVARCHAR(15)))
INSERT INTO ItemAttributes
  VALUES(2, N' color' ,        CAST(N' Red'              AS NVARCHAR(15)))
INSERT INTO ItemAttributes
  VALUES(3, N' material' ,     CAST(N' Wood'             AS NVARCHAR(15)))
INSERT INTO ItemAttributes
  VALUES(3, N' padding' ,      CAST(N' Silk'             AS NVARCHAR(15)))
INSERT INTO ItemAttributes
  VALUES(4, N' material' ,     CAST(N' Gold'             AS NVARCHAR(15)))
INSERT INTO ItemAttributes
  VALUES(4, N' inscription' ,  CAST(N' One ring ...'     AS NVARCHAR(50)))
INSERT INTO ItemAttributes
  VALUES(4, N' size' ,         CAST(10                   AS INT))
INSERT INTO ItemAttributes
  VALUES(5, N' artist' ,       CAST(N' Claude Monet'     AS NVARCHAR(30)))
INSERT INTO ItemAttributes
  VALUES(5, N' name' ,         CAST(N' Field of Poppies' AS NVARCHAR(30)))
INSERT INTO ItemAttributes
  VALUES(5, N' type' ,         CAST(N' Oil'              AS NVARCHAR(30)))
INSERT INTO ItemAttributes
  VALUES(5, N' height' ,       CAST(19.625               AS NUMERIC(9,3)))
INSERT INTO ItemAttributes
  VALUES(5, N' width' ,        CAST(25.625               AS NUMERIC(9,3)))
INSERT INTO ItemAttributes
  VALUES(6, N' artist' ,       CAST(N' Vincent Van Gogh' AS NVARCHAR(30)))
INSERT INTO ItemAttributes
  VALUES(6, N' name' ,         CAST(N' The Starry Night' AS NVARCHAR(30)))
INSERT INTO ItemAttributes
```

```
  VALUES(6, N' type' ,      CAST(N' Oil'          AS NVARCHAR(30)))
INSERT INTO ItemAttributes
  VALUES(6, N' height' ,    CAST(28.75            AS NUMERIC(9,3)))
INSERT INTO ItemAttributes
  VALUES(6, N' width' ,     CAST(36.25            AS NUMERIC(9,3)))
```

속성 값이 다르면 데이터 형식이 다를 수도 있기 때문에 sql_variant 데이터 형식이 value 열에 사용된다는 점을 유념하십시오. 예를 들어 size 속성은 정수 속성 값을 저장하며 name 속성은 문자열 속성 값을 저장합니다.

경매 물품 중 그림인 각 항목(항목 5, 6)에 대한 행과 각 속성에 대한 열을 포함한 ItemAttributes 테이블에서 데이터를 제시하고자 한다고 가정해 봅시다. PIVOT 연산자가 없으면 다음과 같은 쿼리를 작성해야 합니다.

```
SELECT
  itemid,
  MAX(CASE WHEN attribute =  'artist'   THEN value END) AS [artist],
  MAX(CASE WHEN attribute =  'name'     THEN value END) AS [name],
  MAX(CASE WHEN attribute =  'type'     THEN value END) AS [type],
  MAX(CASE WHEN attribute =  'height'   THEN value END) AS [height],
  MAX(CASE WHEN attribute =  'width'    THEN value END) AS [width]
FROM ItemAttributes AS ATR
WHERE itemid IN(5,6)
GROUP BY itemid
```

결과 집합은 다음과 같습니다.

itemid	artist	name	type	height	width
5	Claude Monet	Field of Poppies	Oil	19.625	25.625
6	Vincent Van Gogh	The Starry Night	Oil	28.750	36.250

PIVOT 연산자를 사용하면 더욱 간략하고 읽기 쉬운 코드를 작성하여 동일한 결과를 얻을 수 있습니다.

```
SELECT *
FROM ItemAttributes AS ATR
  PIVOT
  (
    MAX(value)
    FOR attribute IN([artist], [name], [type], [height], [width])
  ) AS PVT
WHERE itemid IN(5,6)
```

대부분의 새 기능과 마찬가지로 실험이나 활용을 통해 PIVOT 연산자를 이해할 수 있습니다. PIVOT 구문의 일부 요소들이 제시되며 단지 새 연산자를 사용하지 않는 쿼리와 이들 요소의 관계를 이해하는 것이 필요합니다. 다른 요소들은 숨겨져 있습니다.

다음 용어들은 PIVOT 연산자의 의미를 이해하는 데 도움이 됩니다.

table_expression

PIVOT 연산자가 영향을 미치는 가상 테이블(FROM 절과 PIVOT 연산자 간 쿼리의 일부): 이 경우에는 ItemAttributes AS ATR

pivot_column

결과 열로 회전해야 하는 값을 가진 table_expression의 열: 이 경우에는 attribute

column_list

결과 열로서 제시하고자 하는 pivot_column의 값 목록(IN 절 다음에 나오는 괄호 안에 표시). 이들은 유효한 식별자로 표현되어야 합니다. 이 경우에는 [artist], [name], [type], [height], [width]

aggregate_function

결과에서 데이터 또는 열 값을 생성하기 위해 사용하는 집계 함수: 이 경우에는 MAX( )

value_column

aggregate_function에 인수로 사용하는 table_expression의 열: 이 경우에는 value

group_by_list

숨겨진 부분—결과를 그룹화하는 데 사용되는 pivot_column 및 value_column을 제외한 table_expression의 모든 열: 이 경우에는 itemid

select_list

group_by_list 및 column_list의 모든 열을 포함할 수 있는 SELECT 절 다음에 오는 열의 목록 (별칭을 사용하여 결과 열의 이름 변경 가능): * 이 경우 group_by_list 및 column_list의 모든 열을 반환합니다.

PIVOT 연산자는 마치 GROUP BY 절을 포함한 쿼리가 있고 해당 열을 지정한 것처럼 group_by_list에 있는 각각의 고유한 값에 대해 하나의 행을 반환합니다. group_by_list는 암시적이며 쿼리의 어느 곳에도 명시적으로 지정되지 않는다는 점을 유념하십시오. pivot_column 및 value_column을 제외한 table_expression의 모든 열이 포함됩니다. 이러한 내용을 이해하는 것은 PIVOT 연산자를 사용해 작성한 쿼리가 그와 같은 결과를 도출하는 이유와 경우에 따라 오류가 발생할 수 있는 이유를 이해하는 데 필수적입니다.

가능한 결과 열에는 group_by_list 및 〈column_list〉의 값이 포함됩니다. 별표(*)를 지정하면 쿼리가 두 목록을 모두 반환합니다. 결과 열의 데이터 부분이나 결과 열 값은 value_column이 인수로 지정된 aggregate_function에 의해 계산됩니다.
색상으로 강조된 다음 코드는 PIVOT 연산자를 사용하는 쿼리의 서로 다른 요소들을 설명합니다.

```
SELECT * -- itemid, [artist], [name], [type], [height], [width]
FROM ItemAttributes AS ATR
  PIVOT
  (
    MAX(value)
    FOR attribute IN([artist], [name], [type], [height], [width])
  ) AS PVT
WHERE itemid IN(5,6)
```

다음 코드는 PIVOT 연산자를 사용하지 않는 쿼리와 서로 다른 요소들을 관련시킵니다.

```
SELECT
  itemid,
  MAX(CASE WHEN attribute =  'artist'   THEN value END) AS [artist],
  MAX(CASE WHEN attribute =  'name'     THEN value END) AS [name],
  MAX(CASE WHEN attribute =  'type'     THEN value END) AS [type],
  MAX(CASE WHEN attribute =  'height'   THEN value END) AS [height],
  MAX(CASE WHEN attribute =  'width'    THEN value END) AS [width]
FROM ItemAttributes AS ATR
WHERE itemid IN(5,6)
GROUP BY itemid
```

〈column_list〉에 값을 명시적으로 지정해야 한다는 것을 유념하십시오. PIVOT 연산자는 정적 쿼리의 pivot_column에서 해당 값을 동적으로 파생하는 옵션을 제공하지 않습니다. 동적 SQL로 쿼리 문자열을 작성하여 이를 수행할 수 있습니다.

이전의 PIVOT 쿼리에서 한 단계 나아가, 각 경매 항목마다 그림과 관련된 모든 속성을 반환하고자 한다고 가정해 봅시다. AuctionItems에 나타나는 속성과 ItemAttributes에 나타나는 속성을 포함하고자 합니다. 오류를 반환하는 다음과 같은 쿼리를 시도할 수 있습니다.

```
SELECT *
FROM AuctionItems AS ITM
  JOIN ItemAttributes AS ATR
    ON ITM.itemid = ATR.itemid
  PIVOT
  (
    MAX(value)
    FOR attribute IN([artist], [name], [type], [height], [width])
  ) AS PVT
WHERE itemtype =  'Painting'
```

오류 메시지는 다음과 같습니다.

```
.Net SqlClient Data Provider: Msg 8156, Level 16, State 1, Line 1
```

'itemid' 열은 'PVT' 에 여러 번 지정됩니다.
PIVOT이 FROM 절과 PIVOT 절 사이의 쿼리에 있는 섹션에 의해 반환되는 가상 테이블인 table_expression에 영향을 준다는 것을 기억하십시오. 이 쿼리에서 가상 테이블에는 AuctionItems과 ItemAttributes에서 각각 유래하는 itemid 열의 두 인스턴스가 포함됩니다. 다음과 같이 쿼리를 수정하면 어떨까 하는 생각이 들 수도 있지만 이 경우에도 역시 오류가 발생합니다.

```
SELECT ITM.itemid, itemtype, whenmade, initialprice,
  [artist], [name], [type], [height], [width]
FROM AuctionItems AS ITM
  JOIN ItemAttributes AS ATR
    ON ITM.itemid = ATR.itemid
  PIVOT
  (
    MAX(value)
    FOR attribute IN([artist], [name], [type], [height], [width])
  ) AS PVT
WHERE itemtype = 'Painting'
```

오류 메시지는 다음과 같습니다.

```
.Net SqlClient Data Provider: Msg 8156, Level 16, State 1, Line 1
```

'itemid' 열은 'PVT' 에 여러 번 지정됩니다.

```
.Net SqlClient Data Provider: Msg 107, Level 15, State 1, Line 1
```

열 prefix 'ITM' 은 쿼리에 사용된 테이블 이름이나 별칭과 매치되지 않습니다.

앞서 언급했듯이, PIVOT 연산자는 table_expression에 의해 반환된 가상 테이블에 영향을 미치며 select_list의 열에는 영향을 미치지 않습니다. select_list는 PIVOT 연산자가 처리를 수행한 후에 평가되며 group_by_list와column_list 만을 참조할 수 있습니다. select_list에서 ITM 별칭이 더 이상 인식되지 않기 때문입니다. 이 점을 이해한다면 영향을 미치고자 하는 열만 포함하는 table_expression과 함께 PIVOT을 제공해야 한다는 것을 깨닫게 됩니다. 그룹화 열(itemid + itemtype, whenmade 및 initialprice가 한번만 발생), 피벗 열(attribute), 값 열(value)이 여기에 포함됩니다. CTE 또는 파생 테이블을 사용하여 이를 수행할 수 있습니다. 다음은 CTE를 사용하는 예제입니다.

```
WITH PNT
AS
(
  SELECT ITM.*, ATR.attribute, ATR.value
  FROM AuctionItems AS ITM
    JOIN ItemAttributes AS ATR
      ON ITM.itemid = ATR.itemid
  WHERE ITM.itemtype =  'Painting'
)
SELECT * FROM PNT
  PIVOT
  (
    MAX(value)
    FOR attribute IN([artist], [name], [type], [height], [width])
  ) AS PVT
```

결과 집합은 다음과 같습니다.

itemid	itemtype	whenmade	initialprice	artist	name	type	height	width
5	Painting	1873	8000000.0000	Claude Monet	Field of Poppies	Oil	19.62	25.62
6	Painting	1889	8000000.0000	Vincent Van Gogh	The Starry Night	Oil	28.75	36.25

다음은 파생 테이블을 사용하는 예제입니다.

```
SELECT *
FROM (SELECT ITM.*, ATR.attribute, ATR.value
      FROM AuctionItems AS ITM
            JOIN ItemAttributes AS ATR
            ON ITM.itemid = ATR.itemid
      WHERE ITM.itemtype =  'Painting' ) AS PNT
  PIVOT

  (
    MAX(value)
    FOR attribute IN([artist], [name], [type], [height], [width])
  ) AS PVT
```

크로스탭 보고서를 생성하여 데이터를 요약하고자 하는 경우에도 PIVOT을 사용할 수 있습니다. 예를 들어 AdventureWorks 데이터베이스의 Purchasing.PurchaseOrderHeader 테이블을 사용하여, 각 구매 방법을 통한 직원별 주문 수량을 반환하고자 한다고 가정하고 구매 방법 ID를 열로 피벗합니다. 오직 관련된 데이터와 함께 PIVOT 연산자를 제공해야 한다는 것을 염두에 두고 파생 테이블을 사용하여 다음 쿼리를 작성합니다.

```
SELECT EmployeeID, [1] AS SM1, [2] AS SM2,
  [3] AS SM3, [4] AS SM4, [5] AS SM5
FROM (SELECT PurchaseOrderID, EmployeeID, ShipMethodID
      FROM Purchasing.PurchaseOrderHeader) ORD
  PIVOT
  (
   COUNT(PurchaseOrderID)
   FOR ShipMethodID IN([1], [2], [3], [4], [5])
  ) AS PVT
```

결과 집합은 다음과 같습니다.

EmployeeID	SM1	SM2	SM3	SM4	SM5
164	56	62	12	89	141
198	24	27	6	45	58
223	56	67	17	98	162
231	50	67	12	81	150
233	55	62	12	106	125
238	53	58	13	102	134
241	50	59	13	108	130
244	55	47	17	93	148
261	58	54	11	120	117
264	50	58	15	86	151
266	58	68	14	116	144
274	24	26	6	41	63

COUNT(PurchaseOrderID) 함수는 목록에 있는 각 배송 방법에 대한 행의 개수를 계산합니다. PIVOT은 COUNT(*)의 사용을 허용하지 않는 다는 것을 유념하십시오. 결과 열에 보다 설명적인 이름을 제공하기 위해 열 별칭이 사용됩니다. PIVOT을 사용하여 서로 다른 열에 각 배송 방법별 주문 수량을 표시하는 것은 ID가 이미 알려진 배송 방법의 개수가 적을 경우에 합리적입니다.

또한 식에서 파생된 값을 피벗할 수도 있습니다. 예를 들어, 매 주문 연도의 직원별 전체 화물 값을 반환하고자 한다고 가정하고 연도를 열로 피벗합니다. 주문 연도는 OrderDate 열에서 파생됩니다.

```
SELECT EmployeeID, [2001] AS Y2001, [2002] AS Y2002,
  [2003] AS Y2003, [2004] AS Y2004
FROM (SELECT EmployeeID, YEAR(OrderDate) AS OrderYear, Freight
      FROM Purchasing.PurchaseOrderHeader) AS ORD
  PIVOT
  (
   SUM(Freight)
   FOR OrderYear IN([2001], [2002], [2003], [2004])
  ) AS PVT
```

결과 집합은 다음과 같습니다.

EmployeeID	Y2001	Y2002	Y2003	Y2004
164	509.9325	14032.0215	34605.3459	105087.7428
198	NULL	5344.4771	14963.0595	45020.9178
223	365.7019	12496.0776	37489.2896	117599.4156
231	6.8025	9603.0502	37604.3258	75435.8619
233	1467.1388	9590.7355	32988.0643	98603.745
238	17.3345	9745.1001	37836.583	100106.3678
241	221.1825	6865.7299	35559.3883	114430.983
244	5.026	5689.4571	35449.316	74690.3755
261	NULL	10483.27	32854.9343	73992.8431
264	NULL	10337.3207	37170.1957	82406.4474
266	4.2769	9588.8228	38533.9582	115291.2472
274	NULL	1877.2665	13708.9336	41011.3821

크로스탭 보고서는 데이터 웨어하우스 시나리오에서는 일반적입니다. AdventureWorks의 판매 주문 및 판매 주문 세부 데이터로 채워지는 다음 OrdersFact 테이블을 고려해 보십시오.

```
CREATE TABLE OrdersFact
(
  OrderID    INT      NOT NULL,
  ProductID  INT      NOT NULL,
  CustomerID NCHAR(5) NOT NULL,
  OrderYear  INT      NOT NULL,
  OrderMonth INT      NOT NULL,
  OrderDay   INT      NOT NULL,
  Quantity   INT      NOT NULL,
  PRIMARY KEY(OrderID, ProductID)
)
INSERT INTO OrdersFact
  SELECT O.SalesOrderID, OD.ProductID, O.CustomerID,
    YEAR(O.OrderDate) AS OrderYear, MONTH(O.OrderDate) AS OrderMonth,
    DAY(O.OrderDate) AS OrderDay, OD.OrderQty
  FROM Sales.SalesOrderHeader AS O
    JOIN Sales.SalesOrderDetail AS OD
      ON O.SalesOrderID = OD.SalesOrderID
```

연도와 월을 행과 열에 각각 반환하여 각 연도와 월에 대한 전체 수량을 가져오려면 다음 쿼리를 사용합니다.

```
SELECT *
FROM (SELECT OrderYear, OrderMonth, Quantity
      FROM OrdersFact) AS ORD
 PIVOT
 (
  SUM(Quantity)
  FOR OrderMonth IN([1],[2],[3],[4],[5],[6],[7],[8],[9],[10],[11],[12])
 ) AS PVT
```

결과 집합은 다음과 같습니다.

OrderYear	1	2	3	4	5	6	7	8	9	10	11	12
2001	NULL	NULL	NULL	NULL	NULL	NULL	966	2209	1658	1403	3132	2480
2002	1040	2303	1841	1467	3179	2418	7755	11325	9066	5584	8268	6672
2003	3532	5431	4132	5694	8278	6444	11288	18986	18681	11607	14771	15855
2004	9227	10999	11314	12239	15656	15805	2209	NULL	NULL	NULL	NULL	NULL

PIVOT은 연도와 월 사이의 존재하지 않는 교차 지점에 대한 NULL 값을 반환합니다. 지정된 월을 포함한 교차 지점 유무에 관계없이 연도는 입력 테이블 식(파생 테이블 ORD)에 나타나는 경우 결과에 나타납니다. 즉, 존재하는 모든 월을 지정하지 않은 경우 모든 열이 NULL인 행이 생길 수도 있습니다. 그러나, 결과에 NULL 값이 나타난다고 해서 반드시 교차 지점이 존재하지 않는 것은 아닙니다. 교차 지점은 열에 NULL 값이 허용되지 않을 경우 quantity 열의 기본 NULL 값에 의해 생길 수도 있습니다. NULL을 무시하고 0과 같은 다른 값이 대신 나타나기를 원하는 경우 select 목록에 ISNULL( ) 함수를 사용합니다.

```
SELECT OrderYear,
  ISNULL([1],  0) AS M01,
  ISNULL([2],  0) AS M02,
  ISNULL([3],  0) AS M03,
  ISNULL([4],  0) AS M04,
  ISNULL([5],  0) AS M05,
  ISNULL([6],  0) AS M06,
  ISNULL([7],  0) AS M07,
  ISNULL([8],  0) AS M08,
  ISNULL([9],  0) AS M09,
  ISNULL([10], 0) AS M10,
  ISNULL([11], 0) AS M11,
  ISNULL([12], 0) AS M12
FROM (SELECT OrderYear, OrderMonth, Quantity
      FROM OrdersFact) AS ORD
 PIVOT
 (
  SUM(Quantity)
  FOR OrderMonth IN([1],[2],[3],[4],[5],[6],[7],[8],[9],[10],[11],[12])
 ) AS PVT
```

결과 집합은 다음과 같습니다.

OrderYear	1	2	3	4	5	6	7	8	9	10	11	12
2001	0	0	0	0	0	0	966	2209	1658	1403	3132	2480
2002	1040	2303	1841	1467	3179	2418	7755	11325	9066	5584	8268	6672
2003	3532	5431	4132	5694	8278	6444	11288	18986	18681	11607	14771	15855
2004	9227	10999	11314	12239	15656	15805	2209	0	0	0	0	0

파생 테이블에 ISNULL(Quantity, 0)을 사용하면 PIVOT이 존재하지 않는 교차 지점에 대해 생성한 NULL 값이 아니라 Quantity 열(존재하는 경우)의 기본 NULL 값 만을 처리하게 됩니다.

2003년과 2004년 각 첫 분기의 연도 및 월 값의 조합에 대해 각 고객 ID에 대한 전체 수량을 1-9 범위로 반환하고자 한다고 가정해 봅시다. 행에 연도 및 월 값을 반환하고 열에 고객 ID를 반환하려면 다음 쿼리를 사용합니다.

```
SELECT *
FROM (SELECT CustomerID, OrderYear, OrderMonth, Quantity
    FROM OrdersFact
    WHERE CustomerID BETWEEN 1 AND 9
      AND OrderYear IN(2003, 2004)

      AND OrderMonth IN(1, 2, 3)) AS ORD
 PIVOT
 (
   SUM(Quantity)
   FOR CustomerID IN([1],[2],[3],[4],[5],[6],[7],[8],[9])
 ) AS PVT
```

결과 집합은 다음과 같습니다.

OrderYear	OrderMonth	1	2	3	4	5	6	7	8	9
2003	1	NULL	NULL	NULL	105	NULL	NULL	8	NULL	NULL
2004	1	NULL	NULL	NULL	80	NULL	NULL	NULL	NULL	NULL
2003	2	NULL	5	NULL	NULL	NULL	NULL	NULL	NULL	15
2004	2	NULL	10	NULL	NULL	NULL	NULL	NULL	6	3
2003	3	NULL	NULL	105	NULL	15	NULL	NULL	NULL	NULL
2004	3	NULL	NULL	103	NULL	25	4	NULL	NULL	NULL

CustomerID와Quantity는 각각 피벗 및 값 열로 사용되기 때문에 이 경우 암시적인 group-by list는 OrderYear와 OrderMonth입니다.

그러나 연도와 월 값의 조합이 열로 표시되기를 원하는 경우 PIVOT 연산자에 전달하기 전에 이들을 연결시켜야 합니다. 피벗 열이 하나 뿐일 수 있기 때문입니다.

```
SELECT *
FROM (SELECT CustomerID, OrderYear*100+OrderMonth AS YM, Quantity
      FROM OrdersFact
      WHERE CustomerID BETWEEN 1 AND 9
        AND OrderYear IN(2003, 2004)
        AND OrderMonth IN(1, 2, 3)) AS ORD
  PIVOT
  (
    SUM(Quantity)
    FOR YM IN([200301],[200302],[200303],[200401],[200402],[200403])
  ) AS PVT
```

결과 집합은 다음과 같습니다.

CustomerID	200301	200302	200303	200401	200402	200403
2	NULL	5	NULL	NULL	10	NULL
3	NULL	NULL	105	NULL	NULL	103
6	NULL	NULL	NULL	NULL	NULL	4
4	105	NULL	NULL	80	NULL	NULL
8	NULL	NULL	NULL	NULL	6	NULL
5	NULL	NULL	15	NULL	NULL	25
7	8	NULL	NULL	NULL	NULL	NULL
9	NULL	15	NULL	NULL	3	NULL

UNPIVOT

UNPIVOT 연산자를 사용하면 이미 피벗된 데이터를 정규화할 수 있습니다. UNPIVOT 연산자의 구문과 요소는 PIVOT 연산자와 유사합니다.

예를 들어 앞 섹션에서 다룬 AuctionItems 테이블을 생각해 보십시오.

itemid	itemtype	whenmade	initialprice
1	Wine	1822	3000.0000
2	Wine	1807	500.0000
3	Chair	1753	800000.0000
4	Ring	-501	1000000.0000
5	Painting	1873	8000000.0000
6	Painting	1889	8000000.0000

ItemAttributes 테이블에 속성들이 표시되는 것과 유사한 방식으로 각 속성이 서로 다른 행에 표시되기를 원한다고 가정해 봅시다.

itemid	attribute	value
1	itemtype	Wine
1	whenmade	1822
1	initialprice	3000.00
2	itemtype	Wine
2	whenmade	1807
2	initialprice	500.00
3	itemtype	Chair
3	whenmade	1753
3	initialprice	800000.00
4	itemtype	Ring
4	whenmade	-501
4	initialprice	1000000.00
5	itemtype	Painting
5	whenmade	1873
5	initialprice	8000000.00
6	itemtype	Painting
6	whenmade	1889
6	initialprice	8000000.00

UNPIVOT 쿼리에서 itemtype, whenmade, initialprice 열을 행으로 회전하려고 합니다. 각 행에는 항목 ID, 속성 및 값이 있어야 합니다. attribute와 value라는 새로운 열 이름을 제공해야 합니다. 이들은 PIVOT 연산자의 pivot_column 및 value_column에 해당합니다. attribute 열은 회전시키고자 하는 실제 열 이름(itemtype, whenmade, initialprice)을 값으로 가져옵니다. value 열은 서로 다른 3개의 소스 열에서 하나의 대상 열로 값을 가져옵니다. 보다 명확하게 하기 위해 ,우선 유효하지 않은 UNPIVOT 쿼리 버전이 제시된 후 일부 제약 조건이 적용된 유효한 쿼리가 제시됩니다.

```
SELECT itemid, attribute, value
FROM AuctionItems
  UNPIVOT
  (
    value FOR attribute IN([itemtype], [whenmade], [initialprice])
  ) AS UPV
```

PIVOT 연산자의 인수로서 FOR 절이 뒤따르는 value_column(이 경우에는 value)에 대한 이름을 제공합니다. FOR 절 다음에 pivot_column(이 경우에는 attribute)에 대한 이름을 제공한 다음 pivot_column의 값으로 얻고자 하는 소스 열 이름의 목록과 함께 IN 절을 제공합니다. 이 열의 목록은 PIVOT 연산자에서 〈column_list〉로서 참조됩니다. 이 쿼리는 다음과 같은 오류를 발생시킵니다.

```
.Net SqlClient Data Provider: Msg 8167, Level 16, State 1, Line 1
```

'whenmade' 열의 유형은 UNPIVOT 목록에 지정된 다른 열의 유형과 충돌합니다.

대상 value 열에는 서로 다른 소스 열에서 작성된 값이 포함됩니다(〈column_list〉에 표시되는 값). 모든 열 값의 대상이 단일 열이기 때문에 UNPIVOT을 실행하려면 〈column_list〉에 있는 모든 열의 데이터 형식, 길이 및 정밀도가 동일해야 합니다. 이러한 제한을 충족시키기 위해 3개의 열을 동일한 데이터 형식으로 변환하는 테이블 식과 함께 UNPIVOT 연산자를 제공할 수 있습니다. 서로 다른 소스 열을 동일한 데이터 형식으로 변환하는 동시에 원래의 데이터 형식을 유지할 수 있다는 점에서 sql_variant 데이터 형식은 유력한 후보가 됩니다. 이 제한을 적용하여 이전 쿼리를 다음과 같이 수정하고 원하는 결과를 얻습니다.

```
SELECT itemid, attribute, value
FROM (SELECT itemid,
       CAST(itemtype     AS SQL_VARIANT) AS itemtype,
       CAST(whenmade     AS SQL_VARIANT) AS whenmade,
       CAST(initialprice AS SQL_VARIANT) AS initialprice
     FROM AuctionItems) AS ITM
  UNPIVOT
  (
    value FOR attribute IN([itemtype], [whenmade], [initialprice])
  } AS UPV
```

결과 attribute 열의 데이터 형식은 sysname입니다. 이는 SQL Server가 개체 이름을 저장하는 데 사용하는 데이터 형식입니다. UNPIVOT 연산자는 결과에서 value 열에 있는 NULL 값을 제거하기 때문에 PIVOT 연산자와 같은 정확한 역방향 작업으로 간주할 수 없다는 점을 유념하십시오.

AuctionItems의 열을 행으로 회전했기 때문에 이제 UNPIVOT 작업의 결과와 ItemAttributes의 행을 결합하여 통합 결과를 제공할 수 있습니다.

```
SELECT itemid, attribute, value
FROM (SELECT itemid,
        CAST(itemtype     AS SQL_VARIANT) AS itemtype,
        CAST(whenmade     AS SQL_VARIANT) AS whenmade,
        CAST(initialprice AS SQL_VARIANT) AS initialprice
      FROM AuctionItems) AS ITM
  UNPIVOT
  (
    value FOR attribute IN([itemtype], [whenmade], [initialprice])
  ) AS UPV
UNION ALL
SELECT *
FROM ItemAttributes
ORDER BY itemid, attribute
```

결과 집합은 다음과 같습니다.

itemid	attribute	value
1	color	Red
1	initialprice	3000.00
1	itemtype	Wine
1	manufacturer	ABC
1	type	Pinot Noir
1	whenmade	1822
2	color	Red
2	initialprice	500.00
2	itemtype	Wine
2	manufacturer	XYZ
2	type	Porto
2	whenmade	1807

3	initialprice	800000.00
3	itemtype	Chair
3	material	Wood
3	padding	Silk
3	whenmade	1753
4	initialprice	1000000.00
4	inscription	One ring
4	itemtype	Ring
4	material	Gold
4	size	10
4	whenmade	-501
5	height	19.625
5	initialprice	8000000.00
5	itemtype	Painting
5	name	Field of Poppies
5	artist	Claude Monet
5	type	Oil
5	whenmade	1873
5	width	25.625
6	height	28.750
6	initialprice	8000000.00
6	itemtype	Painting
6	name	The Starry Night
6	artist	Vincent Van Gogh
6	type	Oil
6	whenmade	1889
6	width	36.250

APPLY

APPLY 관계형 연산자를 사용하면 외부 테이블 반환식의 각 행에 지정된 테이블 반환 함수를 한 번 호출할 수 있습니다. JOIN 관계형 연산자를 사용하는 것과 유사한 방식으로 쿼리의 FROM 구문에서 APPLY를 지정합니다. APPLY는 CROSS APPLY와 OUTER APPLY 등 두 가지 형태로 제공됩니다. APPLY 연산자를 통해 SQL Server 2005에서는 관련 하위 쿼리에서 테이블 반환 함수를 참조할 수 있습니다.

CROSS APPLY

CROSS APPLY는 외부 테이블 반환식의 각 행에 테이블 반환 함수를 호출합니다. 테이블 반환 함수의 인수로서 외부 테이블의 열을 참조할 수 있습니다. CROSS APPLY는 테이블 반환 함수의 개별 호출로 반환된 모든 결과에서 통합 결과 집합을 반환합니다. 테이블 반환 함수가 주어진 외부 행의 빈 집합을 반환하면 해당 외부 행은 결과로 반환되지 않습니다. 예를 들어, 다음 테이블 반환 함수는 두 개의 정수를 인수로 받아들이고 최소값과 최대값을 열로 표시하는 하나의 행으로 이루어진 테이블을 반환합니다.

```
CREATE FUNCTION dbo.fn_scalar_min_max(@p1 AS INT, @p2 AS INT) RETURNS TABLE
AS
RETURN
  SELECT
    CASE
      WHEN @p1 〈 @p2 THEN @p1
      WHEN @p2 〈 @p1 THEN @p2
      ELSE COALESCE(@p1, @p2)
    END AS mn,
    CASE
      WHEN @p1 〉 @p2 THEN @p1
      WHEN @p2 〉 @p1 THEN @p2
      ELSE COALESCE(@p1, @p2)
    END AS mx
GO
SELECT * FROM fn_scalar_min_max(10, 20)
```

결과 집합은 다음과 같습니다.

mn	mx
10	20

다음 T1 테이블을 예로 들어 보겠습니다.

```
CREATE TABLE T1
(
  col1 INT NULL,
  col2 INT NULL
)

INSERT INTO T1 VALUES(10, 20)
INSERT INTO T1 VALUES(20, 10)
INSERT INTO T1 VALUES(NULL, 30)
INSERT INTO T1 VALUES(40, NULL)
INSERT INTO T1 VALUES(50, 50)
```

T1의 각 행에 대해 fn_scalar_min_max를 호출하려고 합니다. 다음과 같이 CROSS APPLY 쿼리를 작성합니다.

```
SELECT *
FROM T1 CROSS APPLY fn_scalar_min_max(col1, col2) AS M
```

결과 집합은 다음과 같습니다.

col1	col2	mn	mx
10	20	10	20
20	10	10	20
NULL	30	30	30
40	NULL	40	40
50	50	50	50

테이블 반환 함수가 특정 외부 행에 대해 여러 개의 행을 반환하는 경우 외부 행이 여러 번 반환됩니다. 앞서 설명한 재귀 쿼리 및 CTE(Common Table Expressions) 섹션에서 사용된 Employees 테이블을 생각해 보십시오(직원 조직도 시나리오). 동일한 데이터베이스에서 다음과 같은 Departments 테이블을 생성합니다.

```
CREATE TABLE Departments
(
  deptid    INT NOT NULL PRIMARY KEY,
  deptname VARCHAR(25) NOT NULL,
  deptmgrid INT NULL REFERENCES Employees
)
SET NOCOUNT ON
INSERT INTO Departments VALUES(1,  'HR' ,          2)
INSERT INTO Departments VALUES(2,  'Marketing' ,    7)
INSERT INTO Departments VALUES(3,  'Finance' ,      8)
INSERT INTO Departments VALUES(4,  'R&D' ,          9)
INSERT INTO Departments VALUES(5,  'Training' ,     4)
INSERT INTO Departments VALUES(6,  'Gardening' , NULL)
```

대부분의 부서에는 Employees 테이블에 있는 직원에 해당하는 관리자 ID가 있지만 Gardening 부서의 경우처럼 부서에 관리자가 없을 수도 있습니다. Employees 테이블에 있는 관리자가 반드시 부서를 관리한다는 점을 유념하십시오. 다음과 같은 테이블 반환 함수는 직원 ID를 인수로 수용하고 해당 직원과 모든 수준에 있는 그의 부하 직원들을 반환합니다.

```
CREATE FUNCTION dbo.fn_getsubtree(@empid AS INT) RETURNS @TREE TABLE
(
  empid   INT NOT NULL,
  empname VARCHAR(25) NOT NULL,
  mgrid   INT NULL,
  lvl   INT NOT NULL
)
AS
BEGIN
  WITH Emplopyees_Subtree(empid, empname, mgrid, lvl)
  AS
  (
    -- Anchor Member (AM)
    SELECT empid, empname, mgrid, 0
    FROM employees
    WHERE empid = @empid
    UNION ALL
    -- Recursive Member (RM)
    SELECT e.empid, e.empname, e.mgrid, es.lvl+1,
    FROM employees AS e
      JOIN employees_subtree AS es
        ON e.mgrid = es.empid
  )
  INSERT INTO @TREE
    SELECT * FROM Employees_Subtree
  RETURN
END
GO
```

각 부서 관리자에 대해 모든 수준의 부하 직원을 반환하려면 다음 쿼리를 사용합니다.

```
SELECT *
FROM Departments AS D
  CROSS APPLY fn_getsubtree(D.deptmgrid) AS ST
```

결과 집합은 다음과 같습니다.

deptid	deptname	deptmgrid	empid	empname	mgrid	lvl
1	HR	2	2	Andrew	1	0
1	HR	2	5	Steven	2	1
1	HR	2	6	Michael	2	1
2	Marketing	7	7	Robert	3	0
2	Marketing	7	11	David	7	1
2	Marketing	7	12	Ron	7	1
2	Marketing	7	13	Dan	7	1
2	Marketing	7	14	James	11	2
3	Finance	8	8	Laura	3	0
4	R&D	9	9	Ann	3	0
5	Training	4	4	Margaret	1	0
5	Training	4	10	Ina	4	1

여기서 주목해야 할 사항은 두 가지입니다. 첫째, Departments의 각 행은 해당 부서 관리자에 대한 fn_getsubtree에서 반환되는 행이 존재하는 것만큼 여러 번 중복됩니다. 둘째, fn_getsubtree가 빈 집합을 반환했기 때문에 Gardening 부서는 결과에 표시되지 않습니다.

CROSS APPLY 연산자의 또 다른 실제적인 사용 사례는 각 그룹에 n 행을 반환하는 일반적인 요청을 해결하는 것입니다. 예를 들어, 다음 함수는 특정 고객에 대한 가장 최근의 주문 요청 개수를 반환합니다.

```
USE AdventureWorks
GO
CREATE FUNCTION fn_getnorders(@custid AS INT, @n AS INT)
  RETURNS TABLE
AS
RETURN
  SELECT TOP(@n) *
  FROM Sales.SalesOrderHeader
  WHERE CustomerID = @custid
  ORDER BY OrderDate DESC
GO
```

CROSS APPLY 연산자를 사용하면 다음과 같은 간단한 쿼리를 통해 각 고객에 대한 가장 최근의 주문 2건을 가져올 수 있습니다.

```
SELECT O.*
FROM Sales.Customer AS C
  CROSS APPLY fn_getnorders(C.CustomerID, 2) AS O
```

TOP의 기능 개선에 대한 자세한 정보는 본 백서의 후반부에 소개되는 "TOP 기능 향상"을 참조하십시오.

OUTER APPLY

OUTER APPLY는 CROSS APPLY와 매우 유사하지만, 테이블 반환 함수가 빈 집합을 반환한 외부 테이블의 행도 반환한다는 차이점이 있습니다. NULL 값은 테이블 반환 함수의 열에 해당하는 열 값으로 반환됩니다. 예를 들어, CROSS APPLY 대신 OUTER APPLY를 사용하려면 이전 섹션에서 다룬 Departments 테이블에 대한 쿼리를 수정하고 출력의 마지막 행에 주목하십시오.

```
SELECT *
FROM Departments AS D
  OUTER APPLY fn_getsubtree(D.deptmgrid) AS ST
```

결과 집합은 다음과 같습니다.

deptid	deptname	deptmgrid	empid	empname	mgrid	lvl
1	HR	2	2	Andrew	1	0
1	HR	2	5	Steven	2	1
1	HR	2	6	Michael	2	1
2	Marketing	7	7	Robert	3	0
2	Marketing	7	11	David	7	1
2	Marketing	7	12	Ron	7	1
2	Marketing	7	13	Dan	7	1
2	Marketing	7	14	James	11	2
3	Finance	8	8	Laura	3	0
4	R&D	9	9	Ann	3	0
5	Training	4	4	Margaret	1	0
5	Training	4	10	Ina	4	1
6	Gardening	NULL	NULL	NULL	NULL	NULL

관련 하위 쿼리의 테이블 반환 함수

SQL Server 2000에서는 관련 하위 쿼리의 테이블 반환 함수를 참조할 수 없지만, SQL Server 2005에서는 APPLY 관계형 연산자가 제공되는 것과 동시에 이러한 제한이 사라졌습니다. 이제 하위 쿼리 내에 외부 쿼리의 열을 인수로 갖춘 테이블 반환 함수를 제공할 수 있습니다. 예를 들어 관리자의 부하 직원이 최소 3명인 부서만 반환하고자 하는 경우 다음과 같은 쿼리를 작성할 수 있습니다.

```
SELECT *
FROM Departments AS D
WHERE (SELECT COUNT(*)
      FROM fn_getsubtree(D.deptmgrid)) >= 3
deptid      deptname                 deptmgrid
----------- ------------------------ -----------
1           HR                       2
2           Marketing                7
```

EXCEPT 및 INTERSECT

ANSI는 집합들 사이에 실행할 수 있는 UNION, EXCEPT, INTERSECT 등 3 가지의 수직 연산을 정의하며 각 연산은 DISTINCT(기본값)와 ALL이라는 2가지 뉘앙스로 구분됩니다. SQL Server 2000과 그 이전 버전은 이 두 가지 조건을 모두 포함하는 UNION 집합 연산만 지원했지만, SQL Server 2005는 암시적인 DISTINCT 뉘앙스를 포함한 EXCEPT 및 INTERSECT를 지원하기 시작했습니다. ALL 뉘앙스는 SQL Server 2005에서 구현되지 않았지만, ALL 뉘앙스를 포함한 EXCEPT 및 INTERSECT는 실제 활용에서 거의 요구되지 않는 반면 DISTINCT 뉘앙스를 포함한 EXCEPT 및 INTERSECT는 실제 활용되는 사례가 많다는 점을 강조하고 싶습니다.

EXCEPT 및 INTERSECT 사용

이러한 새로운 연산을 사용하기 위한 일반적인 형식은 다음과 같습니다.

```
〈query_expression〉 set_operation 〈query_expression〉
```

UNION 집합 연산의 사용과 관련된 구문 및 문법 규칙이 EXCEP 및 INTERSECT에도 적용되므로 이 부분을 간략히 설명하도록 하겠습니다. 결과 집합의 열 이름은 첫 번째 쿼리에 의해 결정됩니다. 두 쿼리 모두 열의 개수가 동일해야 하며 해당 열은 호환되어야 합니다(암시적인 변환에 의해). 집합 연산의 결과에 적용되는 오직 하나의 ORDER BY 절을 지정할 수 있으며 개별 쿼리와 함께 ORDER BY를 지정하는 것은 허용되지 않습니다. 모든 집합 연산에 의해 실행되는 비교는 입력 집합의 전체 열에 영향을 미치며 NULL을 서로에게 동등한 값으로 인식합니다. 예를 들어, CustomerID: 11676 및 SalesPersonID: NULL인 집합의 열은 CustomerID: 11676 및 SalesPersonID: NULL인 또 다른 집합의 열과 동일한 것으로 간주합니다.

UNION 연산자는 두 쿼리의 결과 집합을 연결하여 통합된 결과에서 별개의 열 생성을 반환합니다.
EXCEPT는 첫 번째 집합(EXCEPT 키워드 왼쪽)에 나타나지만 두 번째 집합(EXCEPT 키워드 오른쪽)에는 나타나지 않는 별개의 열을 반환합니다.

INTERSECT는 두 집합에 모두 나타나는 별개의 열을 반환합니다.
다음 예제를 통해 새로운 EXCEPT 및 INTERSECT 집합 연산에 대한 이해를 높일 수 있습니다.

다음 쿼리는 AdventureWorks 데이터베이스 내 Sales.SalesOrderHeader 테이블에서 territory 3과 2004년에 대해 CustomerID와 SalesPersonID 쌍을 반환합니다.

```
USE AdventureWorks;

SELECT CustomerID, SalesPersonID
FROM Sales.SalesOrderHeader
WHERE TerritoryID = 3
  AND OrderDate 〉= '20040101' AND OrderDate 〈 '20050101'
```

결과 집합은 다음과 같습니다(정렬 및 단축됨).

CustomerID	SalesPersonID
18	277
18	277
22	275
22	275
39	275
39	275
79	276
79	276
148	275
149	268
149	281
...	
11676	NULL
11676	NULL
22124	NULL
27503	NULL
27657	NULL

(77 row(s) affected)

결과 집합에서 확인할 수 있듯이, 여러 명의 영업 사원이 동일 고객에 대한 주문을 처리할 수 있으며 단 한 명의 영업 사원이 동일 고객에 대한 여러 건의 주문을 처리할 수 있으므로 중복 행이 발생합니다. 또한 영업 사원 ID가 NULL이면 알려진 영업 사원이나 관련된 영업 사원이 없음을 나타냅니다.

다음 쿼리는 AdventureWorks 데이터베이스 내 Sales.SalesOrderHeader 테이블에서 territory 3과 2003년에 대해 CustomerID와 SalesPersonID 쌍을 반환합니다.

```
SELECT CustomerID, SalesPersonID
FROM Sales.SalesOrderHeader
WHERE TerritoryID = 3
  AND OrderDate >= '20030101' AND OrderDate < '20040101'
```

결과 집합은 다음과 같습니다(정렬 및 단축됨).

CustomerID	SalesPersonID
18	277
18	277
18	277
18	277
22	275
22	275
22	275
22	275
39	275
39	275
39	275
39	275
58	275
79	276
79	276
79	276
79	276
147	275
147	275
148	275
148	275
149	281
149	281
149	281
149	281
...	
13291	NULL
16668	NULL
18315	NULL
27286	NULL

(149 row(s) affected)

2004년에는 주문 활동이 있었지만 2003년에는 없었던 영업 지역 3의 고객 및 영업 사원 쌍을 반환해야 한다고 가정해 봅시다. EXCEPT 집합 연산을 사용해 이 작업을 실행하는 방법은 다음과 같습니다.

```
SELECT CustomerID, SalesPersonID
FROM Sales.SalesOrderHeader
WHERE TerritoryID = 3
  AND OrderDate >= '20040101' AND OrderDate < '20050101'

EXCEPT

SELECT CustomerID, SalesPersonID
FROM Sales.SalesOrderHeader
WHERE TerritoryID = 3
  AND OrderDate >= '20030101' AND OrderDate < '20040101'
```

결과 집합은 다음과 같습니다.

CustomerID	SalesPersonID
149	268
287	277
468	277
646	275
11676	NULL
22124	NULL
27503	NULL
27657	NULL

(8 row(s) affected)

UNION 및 INTERSECT와 달리 EXCEPT 집합 연산은 비대칭적이라는 점을 유념하십시오. 즉, U UNION V는 V UNION U와 논리적으로 동일하고, U INTERSECT V 역시 V INTERSECT U와 논리적으로 동일하지만, U EXCEPT V는 V EXCEPT U와 논리적으로 동일하지 않습니다. 예를 들어, 위의 쿼리에서 집합을 교체하면 2003년에는 주문 활동이 있었지만 2004년에는 없었던 영업 지역 3의 고객 및 영업 사원 쌍을 나타내는 다른 결과 집합이 산출됩니다.

```
SELECT CustomerID, SalesPersonID
FROM Sales.SalesOrderHeader
WHERE TerritoryID = 3
  AND OrderDate >= '20030101' AND OrderDate < '20040101'

EXCEPT

SELECT CustomerID, SalesPersonID
FROM Sales.SalesOrderHeader
WHERE TerritoryID = 3
  AND OrderDate >= '20040101' AND OrderDate < '20050101'
```

결과 집합은 다음과 같습니다.

CustomerID	SalesPersonID
58	275
147	275
198	277
219	275
238	275
274	275
288	277
395	277
472	275
562	275
598	275
643	277
694	275
13291	NULL
16668	NULL
18315	NULL
27286	NULL

(17 row(s) affected)

2004년과 2003년 모두에 표시되는 쌍을 가져오려면 다음과 같이 INTERSECT 연산을 사용합니다.

```
SELECT CustomerID, SalesPersonID
FROM Sales.SalesOrderHeader
WHERE TerritoryID = 3
  AND OrderDate >= '20040101' AND OrderDate < '20050101'

INTERSECT

SELECT CustomerID, SalesPersonID
FROM Sales.SalesOrderHeader
WHERE TerritoryID = 3
  AND OrderDate >= '20030101' AND OrderDate < '20040101'
```

결과 집합은 다음과 같습니다.

CustomerID	SalesPersonID
18	277
22	275
39	275
79	276
148	275
149	281
183	275
197	277
220	275
223	276
237	275
306	277
310	275
323	277
327	275
348	276
360	277

363	275
377	277
381	275
400	275
418	275
431	277
454	275
523	276
527	281
543	275
544	275
546	276
579	275
606	275
622	275
660	277
670	275
695	275

(35 row(s) affected)

집합 연산의 결과 집합을 구체화하려면 INSERT INTO 또는 SELECT INTO 문을 사용할 수 있습니다. INSERT INTO는 간단합니다. SELECT INTO의 경우에는 첫 번째 쿼리에만 INTO 절을 지정해야 합니다. 예를 들어, #T라는 임시 테이블에서 이전 INTERSECT 연산의 결과를 구체화하는 방법은 다음과 같습니다.

```
SELECT CustomerID, SalesPersonID
INTO #T
FROM Sales.SalesOrderHeader
WHERE TerritoryID = 3
  AND OrderDate >= '20040101' AND OrderDate < '20050101'

INTERSECT

SELECT CustomerID, SalesPersonID
FROM Sales.SalesOrderHeader
WHERE TerritoryID = 3
  AND OrderDate >= '20030101' AND OrderDate < '20040101'
```

평가 순서

여러 개의 집합 연산을 혼합할 수 있지만 SQL Server는 다음과 같은 선행 순서에 따라 코드를 평가한다는 점을 명심하십시오.

1. 괄호
2. INTERSECT
3. UNION, EXCEPT

수준 1은 수준 2와 3보다 먼저 평가되고 수준 2는 수준 3보다 먼저 평가됩니다. 동일한 수준의 여러 연산은 좌에서 우로 평가됩니다.

다음 쿼리를 예로 들어 봅시다.

```
SELECT CustomerID, SalesPersonID
FROM Sales.SalesOrderHeader
WHERE TerritoryID = 3
  AND OrderDate >= '20030701' AND OrderDate < '20031001'

INTERSECT

SELECT CustomerID, SalesPersonID
FROM Sales.SalesOrderHeader
WHERE TerritoryID = 3
  AND OrderDate >= '20031001' AND OrderDate < '20040101'

EXCEPT

SELECT CustomerID, SalesPersonID
FROM Sales.SalesOrderHeader
WHERE TerritoryID = 3
  AND OrderDate >= '20030101' AND OrderDate < '20030401'

INTERSECT

SELECT CustomerID, SalesPersonID
FROM Sales.SalesOrderHeader
WHERE TerritoryID = 3
  AND OrderDate >= '20030401' AND OrderDate < '20030701'

UNION

SELECT CustomerID, SalesPersonID
FROM Sales.SalesOrderHeader
WHERE TerritoryID = 3
  AND OrderDate >= '20040101' AND OrderDate < '20040401'
```

```
INTERSECT

SELECT CustomerID, SalesPersonID
FROM Sales.SalesOrderHeader
WHERE TerritoryID = 3
  AND OrderDate >= '20040401' AND OrderDate < '20040701'
```

결과 집합은 다음과 같습니다.

CustomerID	SalesPersonID
18	277
22	275
39	275
79	276
183	275
197	277
220	275
223	276
237	275
287	277
306	277
310	275
323	277
327	275
348	276
360	277
363	275
377	277
381	275
400	275
418	275
431	277

454	275
523	276
527	281
543	275
544	275
546	276
579	275
606	275
622	275
660	277
670	275
695	275
11676	NULL

(35 row(s) affected)

다음은 위의 쿼리를 간단하게 표현한 것으로서, 정수 값은 논리적인 평가 순서를 나타냅니다.

```
set_2003_Q3
1. INTERSECT
set_2003_Q4

4. EXCEPT

set_2003_Q1
2. INTERSECT
set_2003_Q2

5. UNION

set_2004_Q1
3. INTERSECT
set_2004_Q2
```

INTERSECT는 쿼리에 나타나는 연산 중에서 선행 순서가 가장 높기 때문에 INTERSECT 연산이 제일 먼저 적용됩니다. EXCEPT와 UNION은 선행 순서 수준이 동일하므로 좌에서 우로 평가됩니다. 이 경우에는 EXCEPT가 먼저 평가된 다음 UNION이 평가됩니다.

코드의 명료성을 높이기 위해, INTERSECT가 먼저 평가되는 기본 동작에 관심이 있는 경우에도 괄호를 사용하는 것이 좋을 수도 있습니다. 위의 쿼리는 좀더 명확한 표현을 위해 괄호를 사용하여 다음과 같이 표현할 수도 있습니다.

```
(SELECT CustomerID, SalesPersonID
 FROM Sales.SalesOrderHeader
 WHERE TerritoryID = 3
  AND OrderDate >= '20030701' AND OrderDate < '20031001'

 INTERSECT

 SELECT CustomerID, SalesPersonID
 FROM Sales.SalesOrderHeader
 WHERE TerritoryID = 3
  AND OrderDate >= '20031001' AND OrderDate < '20040101' )

EXCEPT

(SELECT CustomerID, SalesPersonID
 FROM Sales.SalesOrderHeader
 WHERE TerritoryID = 3
  AND OrderDate >= '20030101' AND OrderDate < '20030401'

 INTERSECT

 SELECT CustomerID, SalesPersonID
 FROM Sales.SalesOrderHeader
 WHERE TerritoryID = 3
  AND OrderDate >= '20030401' AND OrderDate < '20030701' )

UNION

(SELECT CustomerID, SalesPersonID
 FROM Sales.SalesOrderHeader
 WHERE TerritoryID = 3
  AND OrderDate >= '20040101' AND OrderDate < '20040401'

 INTERSECT

 SELECT CustomerID, SalesPersonID
 FROM Sales.SalesOrderHeader
 WHERE TerritoryID = 3
  AND OrderDate >= '20040401' AND OrderDate < '20040701' )
```

새로운 DRI 동작 지원: SET DEFAULT 및 SET NULL

ANSI SQL은 외래 키(FOREIGN KEY) 제약 조건을 지원하기 위해 네 가지의 가능한 참조 동작을 정의합니다. 시스템이 어떤 방식으로 외래 키가 참조하는 테이블에 대한 DELETE 또는 UPDATE 연산에 대응하기를 원하는지 나타내는 해당 동작을 지정합니다. SQL Server 2000은 NO ACTION 및 CASCADE라는 두 가지 작업을 지원하지만 SQL Server 2005는 SET DEFAULT 및 SET NULL 참조 동작을 추가로 지원합니다.

SET DEFAULT 및 SET NULL 참조 동작은 DRI(Declarative Referential Integrity) 기능을 더욱 확장합니다. 외래 키 선언의 ON UPDATE 및 ON DELETE 절과 함께 이러한 옵션을 사용할 수 있습니다. SET DEFAULT는 참조 테이블에서 행을 삭제하거나(ON DELETE) 참조된 키를 업데이트하는 경우(ON UPDATE) SQL Server가 참조하는 테이블에 있는 관련 행의 참조하는 열 값을 열의 기본값으로 설정한다는 것을 의미합니다. 이와 유사하게, SQL Server는 참조하는 열이 NULL 값을 허용하면 SET NULL 옵션을 사용하는 경우 값을 NULL로 설정하는 동작으로 대응할 수 있습니다.

예를 들어, 다음 Customers 테이블에는 3명의 실제 고객과 1명의 가상 고객이 있습니다.

```
CREATE TABLE Customers
(
  customerid CHAR(5) NOT NULL,
  /* other columns */
  CONSTRAINT PK_Customers PRIMARY KEY(customerid)
)

INSERT INTO Customers VALUES( 'DUMMY' )
INSERT INTO Customers VALUES( 'FRIDA' )
INSERT INTO Customers VALUES( 'GNDLF' )
INSERT INTO Customers VALUES( 'BILLY' )
```

Orders 테이블은 주문을 추적합니다. 주문이 실제 고객에게 반드시 지정되어야 할 필요는 없습니다. 주문을 입력하고 고객 ID를 지정하지 않으면 가상(DUMMY) 고객 ID가 주문에 기본적으로 지정됩니다. Customers 테이블에서 행을 삭제할 때 SQL Server가 Orders에 있는 관련 행의 customerid 열에 NULL을 설정하기를 원합니다. customerid 열이 NULL인 주문은 "고아"가 됩니다. 즉, 이들은 어느 고객에게도 속하지 않습니다. 또한 Customers의 customerid 열에 대한 업데이트를 허용하고자 한다고 가정해 봅시다. Orders에 있는 관련 행을 모두 업데이트하고자 할 수도 있지만, 회사 내의 비즈니스 규칙이 그 반대의 경우를 요구한다고 가정하십시오. ID가 변경된 고객에게 속하는 주문은 기본 고객(DUMMY)과 관련됩니다. Customers의 customerid 열을 업데이트할 때 SQL Server가 기본값 'DUMMY'를 Orders에 있는 관련 고객 ID(customerid)로 설정하기를 원합니다. 다음과 같이 외래 키를 사용해 Orders 테이블을 만들고 몇 가지 주문으로 테이블을 채웁니다.

```
CREATE TABLE Orders
(
  orderid    INT      NOT NULL,
  customerid CHAR(5)  NULL DEFAULT( 'DUMMY' ),
  orderdate  DATETIME NOT NULL,
  CONSTRAINT PK_Orders PRIMARY KEY(orderid),
  CONSTRAINT FK_Orders_Customers
    FOREIGN KEY(customerid)
    REFERENCES Customers(customerid)
      ON DELETE SET NULL
      ON UPDATE SET DEFAULT
)
INSERT INTO Orders VALUES(10001,  'FRIDA' ,  '20040101' )
INSERT INTO Orders VALUES(10002,  'FRIDA' ,  '20040102' )
INSERT INTO Orders VALUES(10003,  'BILLY' ,  '20040101' )
INSERT INTO Orders VALUES(10004,  'BILLY' ,  '20040103' )
INSERT INTO Orders VALUES(10005,  'GNDLF' ,  '20040104' )
INSERT INTO Orders VALUES(10006,  'GNDLF' ,  '20040105' )

SET NULL 및 SET DEFAULT 옵션을 테스트하려면 다음 DELETE 및 UPDATE 문을 실행합니다.
DELETE FROM Customers
WHERE customerid =  'FRIDA'
UPDATE Customers
  SET customerid =  'DOLLY'
WHERE customerid =  'BILLY'
```

결과적으로, FRIDA의 주문은 customerid 열이 NULL 값인 채로 지정되며, BILL의 주문은 DUMMY인 채로 지정됩니다.

orderid	customerid	orderdate
10001	NULL	1/1/2004 12:00:00 AM
10002	NULL	1/2/2004 12:00:00 AM
10003	DUMMY	1/1/2004 12:00:00 AM
10004	DUMMY	1/3/2004 12:00:00 AM
10005	GNDLF	1/4/2004 12:00:00 AM
10006	GNDLF	1/5/2004 12:00:00 AM

SET DEFAULT 옵션을 사용하고 참조하는 열이 참조된 테이블에 해당 값이 없는 NULL이 아닌 기본값을 가지는 경우 트리거 동작을 실행하면 오류가 발생할 것입니다. 예를 들어, Customers에서 DUMMY 고객을 삭제한 다음 GNDLF의 customerid를 GLDRL로 업데이트하면 오류가 발생합니다. UPDATE는 Customers에 해당 행이 없는 DUMMY 고객 ID와 함께 GNDLF의 원래 주문을 지정하려고 시도하는 SET DEFAULT 동작을 트리거합니다.

```
DELETE FROM Customers
WHERE customerid = 'DUMMY'
UPDATE Customers
  SET customerid = 'GLDRL'
WHERE customerid = 'GNDLF'
.Net SqlClient Data Provider: Msg 547, Level 16, State 0, Line 1
```

UPDATE 문이 COLUMN FOREIGN KEY 제약 조건 'FK_Orders_Customers' 와 충돌을 일으켰습니다. 이 충돌은 'tempdb' 데이터베이스, 'Customers' 테이블, 'customerid' 열에서 발생했습니다.

명령문이 종료되었습니다.

정의된 참조 동작을 비롯하여 외래 키에 대한 자세한 정보를 원할 경우 sys.foreign_keys를 살펴보십시오.

성능 및 오류 처리 기능 개선

이 섹션에서는 이전 버전의 SQL Server에 존재했던 성능 문제를 해결하고 데이터 로드 기능을 향상시키며 오류 관리 기능을 획기적으로 개선시키는 기능 향상에 대해 다룹니다. 이러한 개선 기능에는 BULK 행 집합 공급자와 TRY...CATCH 오류 처리 구조가 포함됩니다.

BULK 행 집합 공급자

BULK는 파일 데이터를 관계형으로 액세스할 수 있도록 해주는 OPENROWSET 함수에 지정된 새로운 행 집합 공급자입니다. 파일에서 데이터를 검색하려면 BULK 옵션, 파일명, 그리고 bcp.exe 또는 수작업을 통해 생성된 포맷 파일을 지정합니다. OPENROWSET에서 반환된 테이블의 별칭 다음에 오는 괄호 안에 결과 열에 대한 이름을 지정할 수 있습니다.
다음은 OPENROWSET으로 지정할 수 있는 모든 옵션의 새로운 구문입니다.

```
OPENROWSET
( { 'provider_name'
, { 'datasource' ; 'user_id' ; 'password' | 'provider_string' }
      , { [ catalog. ] [ schema. ] object | 'query' }
| BULK 'data_filename' ,
{FORMATFILE = 'format_file_path' [, <bulk_options>] |
   SINGLE_BLOB | SINGLE_CLOB | SINGLE_NCLOB}
}
)
```

```
〈bulk_options〉 ::=
[ , CODEPAGE = 'ACP' | 'OEM' | 'RAW' | 'code_page' ]
[ , FIRSTROW = first_row ]
[ , LASTROW = last_row ]
[ , ROWS_PER_BATCH = 'rows_per_batch' ]
[ , MAXERRORS = 'max_errors' ]
      [ , ERRORFILE =' file_name' ]
}
)
```

예를 들어, 다음 쿼리는 'c:\temp\textfile1.txt' 텍스트 파일에서 3개의 열을 반환하고 결과 열에 col1, col2, col3라는 열 별칭을 제공합니다.

```
SELECT col1, col2, col3
 FROM OPENROWSET(BULK 'c:\temp\textfile1.txt' ,
      FORMATFILE = 'c:\temp\textfile1.fmt' } AS C(col1, col2, col3)
```

BULK 옵션을 사용할 때, 나중에 설명할 SINGLE_BLOB, SINGLE_CLOB 또는 SINGLE_NCLOB 옵션을 사용하지 않는 다면 포맷 파일이 지정되어야 한다는 점을 유념하십시오. 그렇기 때문에 데이터 파일 형식, 필드 종결자 또는 행 종결자를 지정할 필요가 없습니다. FORMATFILE과 함께 선택적으로 지정할 수 있는 다른 옵션은 CODEPAGE, FIRSTROW, LASTROW, ROW_PER_BATCH, MAXERRORS, ERRORFILE입니다. 대부분의 옵션은 SQL Server 2000의 BULK INSERT 명령과 함께 사용할 수 있습니다. ERRORFILE 옵션은 개념상 새로운 것입니다. 이 파일에는 입력 데이터 파일로부터 서식 오류를 갖는 0개 이상의 행이 포함됩니다(즉, 이들 행은 OLEDB rowset으로 변환될 수 없음). 이들 행은 데이터 파일에서 이 오류 파일로 "있는 그대로" 복제됩니다. 오류가 수정되면 데이터는 이미 원하는 포맷이므로 동일한 명령어를 사용하여 손쉽게 다시 로드할 수 있습니다. 명령어 실행 초기에 오류 파일이 생성됩니다. 파일이 이미 존재하는 경우 오류가 발생합니다. 이 파일의 행을 살펴보면 장애가 발생한 행을 식별하기는 쉽지만 장애의 원인을 알아낼 방법이 없습니다. 이런 문제를 해결하기 위해 확장자가 .ERROR.txt인 컨트롤 파일이 자동으로 생성됩니다. 이 파일은 ERRORFILE의 각 행을 참조하고 오류 진단을 실행합니다.

BULK 행 집합 공급자를 사용하여 OPENROWSET에서 반환된 결과로 테이블을 채우고 대량 로드 작업에 대한 테이블 옵션을 지정할 수 있습니다. 예를 들어, 다음 코드는 MyTable 테이블에 대한 이전 쿼리의 결과를 로드하며, 대상 테이블에서 제약 조건 검사를 비활성화할 것을 요청합니다.

```
INSERT INTO MyTable WITH (IGNORE_CONSTRAINTS)
 SELECT col1, col2, col3
 FROM OPENROWSET(BULK 'c:\temp\textfile1.txt' ,
      FORMATFILE = 'c:\temp\textfile1.fmt' ) AS C(col1, col2, col3)
```

IGNORE_CONSTRAINTS 옵션 이외에, 로드 작업에 지정할 수 있는 다른 테이블 힌트는 BULK_KEEPIDENTITY, BULK_KEEPNULLS, IGNORE_TRIGGERS입니다.

또한 BULK provider를 사용하면 SINGLE_CLOB(문자 데이터), SINGLE_NCLOB(유니코드 데이터), SINGLE_BLOB(바이너리 데이터)와 같은 옵션 중 하나를 지정하여 파일 데이터를 LOB(Large Object Type)의 단일 열 값으로 반환할 수 있습니다. 이러한 옵션 중 하나를 사용하는 경우에는 포맷 파일을 지정하지 않습니다. VARCHAR(MAX), NVARCHAR(MAX), VARBINARY(MAX) 또는 XML과 같은 데이터 형식 중 하나의 LOB(Large Object) 열로 파일을 로드할 수 있습니다(INSERT 또는 UPDATE 문 사용). 본 백서의 후반부에 가변 길이 열에 대한 MAX 지정자와 XML 데이터 형식에 대한 자세한 내용을 확인할 수 있습니다.

파일을 LOB 열로 로드하는 예제로서, 다음 UPDATE 문은 'c:\temp\textfile101.txt' 라는 텍스트 파일을 고객 101에 대한 CustomerData 테이블에 있는 txt_data 열로 로드합니다.

```
UPDATE CustomerData
  SET txt_data  = (SELECT txt_data FROM OPENROWSET(
    BULK 'c:\temp\textfile101.txt' , SINGLE_CLOB) AS F(txt_data))
WHERE custid = 101
```

한 번에 오직 하나의 LOB 열만 업데이트할 수 있다는 것을 유념하십시오.
다음 예제는 INSERT 문을 사용하여 고객 102에 대한 바이너리 파일을 LOB 열로 로드하는 방법을 보여줍니다.

```
INSERT INTO CustomerData(custid, binary_data)
  SELECT 102 AS custid, binary_data
  FROM OPENROWSET(
    BULK 'c:\temp\binfile102.dat' , SINGLE_BLOB) AS F(binary_data)
```

예외 처리

SQL Server 2005는 TRY...CATCH Transact-SQL 구조의 형태로 단순하지만 매우 강력한 예외 처리 메커니즘을 도입했습니다.

SQL Server의 이전 버전에서는 오류의 소지가 있는 모든 구문 다음에는 오류 처리 코드를 포함해야 했습니다. 오류 검사 코드를 중앙화하려면 레이블과 GOTO 문을 사용해야 합니다. 더욱이 데이터 형식 전환 오류와 같은 오류가 발생하면 배치가 종료되어 버리기 때문에 Transact-SQL에서 이를 발견할 수 없습니다. SQL Server 2005는 이러한 많은 문제를 해결했습니다.

그러한 오류가 연결 해제를 야기하지 않는다면 배치를 종료하는 데 사용되는 오류를 발견 및 처리할 수 있습니다(Table 또는 Database Integrity Suspect, 하드웨어 오류 등 심각도가 20 이상인 일반적인 오류).
BEGIN TRY/END TRY 블록 내에서 실행하려는 코드를 작성하여 BEGIN CATCH /END CATCH 블록에서 오류 처리 코드를 따르기만 하면 됩니다. TRY 블록은 해당 CATCH 블록이 있어야 합니다. 그렇지 않으면 구문 오류가 발생할 것입니다. 간단한 예제로 다음과 같은 Employees 테이블을 생각해 보십시오.

```
CREATE TABLE Employees
(
  empid   INT NOT NULL,
  empname VARCHAR(25) NOT NULL,
  mgrid   INT NULL,
  /* other columns */
  CONSTRAINT PK_Employees PRIMARY KEY(empid),
  CONSTRAINT CHK_Employees_empid CHECK(empid 〉 0),
  CONSTRAINT FK_Employees_Employees
    FOREIGN KEY(mgrid) REFERENCES Employees(empid)
)
```

새로운 직원 행을 테이블에 삽입하는 코드를 작성해야 하고 몇 가지 수정 작업을 통해 장애 상황에 대응해야 한다고 가정해 보겠습니다. 다음과 같이 새로운 TRY...CATCH 구조를 사용합니다.

```
BEGIN TRY
  INSERT INTO Employees(empid, empname, mgrid)
    VALUES(1,  'Emp1' , NULL)

  PRINT  'After INSERT.'
END TRY
BEGIN CATCH
  PRINT  'INSERT failed.'
  /* perform corrective activity */
END CATCH
```

처음으로 이 코드를 실행하면 'After INSERT' 라는 출력이 나타납니다. 두 번째로 이 코드를 실행하면 'INSERT Failed' 라는 출력이 나타납니다.

TRY 블록 내의 코드가 오류 없이 완료되면 해당 CATCH 블록 다음에 오는 첫 번째 명령문으로 컨트롤이 전달됩니다. TRY 블록 내의 명령문에 오류가 생기면 해당 CATCH 블록 내의 첫 번째 명령문으로 컨트롤이 전달됩니다.

CATCH 블록에 의해 오류가 발견되면 요청 애플리케이션으로 반환되지 않는다는 점을 유념하십시오. 또한 애플리케이션이 오류 정보를 가져오기를 원할 경우 정보를 애플리케이션에 스스로 전달해야 합니다(예를 들어 RAISEROR를 사용하거나 쿼리의 결과 집합으로). ERROR_NUMBER( ), ERROR_MESSAGE( ), ERROR_SEVERITY( ), ERROR_STATE( ), ERROR_LINE( ), ERROR_PROCEDURE( )와 같은 새로운 함수에 의해 CATCH 블록의 모든 오류 정보를 사용할 수 있습니다. 이들 함수는 CATCH 블록의 어디에서든 여러 번 쿼리할 수 있으며 해당 값은 동일하게 유지됩니다. 심지어 내부 수준의 함수에서 CATCH 블록으로 정보를 가져올 수도 있습니다(예를 들면 오류 처리를 위해 작성한 저장 프로시저로). 이를 시연하기 위해 단순히 모든 오류 함수 정보를 반환하는 다음과 같은 저장 프로시저를 작성합니다.

```
USE tempdb
GO
CREATE PROCEDURE usp_showerrorinfo
AS
SELECT ERROR_NUMBER(    ) as ErrorNumber,
  ERROR_STATE(    ) as ErrorState,
  ERROR_SEVERITY(    ) as ErrorSeverity,
  ERROR_LINE(    ) as ErrorLine,
  ISNULL(ERROR_PROCEDURE(    ),  'Not in proc' ) as ErrorProcedure,
  ERROR_MESSAGE(    ) as ErrorMessage
GO
```

그 다음으로, 오류를 발생시키고 CATCH 블록에서 프로시저를 호출하여 오류 정보를 반환하는 다음과 같은 코드를 실행합니다.

```
BEGIN TRY
   SELECT 1/0
END TRY
BEGIN CATCH
   EXEC usp_showerrorinfo
END CATCH
```

출력에서 확인할 수 있듯이, 심지어 CATCH 블록에 대한 내부 수준에서도 모든 오류 정보를 사용할 수 있습니다.

ErrorNumber	ErrorState	ErrorSeverity	ErrorLine
8134	1	16	3

ErrorProcedure	ErrorMessage
Not in proc	Divide by zero error encountered.

반환 값에 영향을 미치지 않고 함수를 여러 번 쿼리할 수 있는 기능은 DECLARE를 제외한 모든 명령문의 영향을 받기 때문에 CATCH 블록의 첫 번째 명령문에서 쿼리되어야 하는 @@error 함수와 대조적입니다. ERROR_NUMBER(   )는 @@error의 대안으로 사용될 수 있지만, 그 외의 다섯 함수는 오류에 의해 생성된 그대로 정보의 나머지를 제공합니다. 그러한 정보는 SQL Server 2005 이전 버전에서는 사용할 수 없습니다.

처리되지 않은 오류가 배치 또는 루틴에서 발생하고(저장 프로시저, 트리거, 사용자 정의 함수, 동적 코드) 상위 수준의 코드가 TRY 블록 내의 해당 배치나 루틴을 호출한 경우 상위 수준의 해당 CATCH 블록으로 컨트롤이 전달됩니다. 상위 수준이 TRY 블록 내의 내부 수준을 호출하지 않은 경우 SQL Server는 콜 스택(call stack)의 상위 수준에서 TRY 블록을 계속 찾을 것이며 처음으로 발견된 TRY...CATCH 구조의 CATCH 블록으로 컨트롤을 전달할 것입니다. 아무것도 발견되지 않으면 호출 애플리케이션으로 오류가 반환됩니다.

컴파일 및 재컴파일 오류는 다른 오류와 조금 다르게 취급된다는 점을 유념하십시오. 컴파일 및 재컴파일 오류는 동일한 실행 수준에서는 발견되지 않지만 더 높은 수준으로 TRY...CATCH 블록 범위가 지정되면 발견됩니다. 예를 들어, 존재하지 않는 테이블에 액세스하려는 시도에서 비롯되는 다음과 같은 컴파일 오류는 TRY/CATCH와 동일한 수준에서 발생하므로 오류를 발견하지 못합니다.

```
BEGIN TRY
  SELECT * FROM NoSuchTable
END TRY
BEGIN CATCH
  PRINT  'Error caught at same level.'
END CATCH

Output:
Msg 208, Level 16, State 1, Line 2
Invalid object name  'NoSuchTable' .
```

반면, 저장 프로시저에서 오류의 소지가 있는 코드를 캡슐화하는 경우는 다음과 같습니다.

```
USE tempdb
GO
CREATE PROC proc1
AS
SELECT * FROM NoSuchTable
GO
```

TRY 블록 내 프로시저를 호출하면 더 높은 수준의 CATCH 블록에 의해 프로시저 내 컴파일 오류가 발견될 것입니다.

```
BEGIN TRY
  EXEC proc1
END TRY
BEGIN CATCH
  PRINT  'Error caught at higher level.'
END CATCH

Output:
Error caught at higher level.
```

SQL Server 2005의 오류 처리에 대한 상세한 예로 제시되는 다음 코드는 장애를 일으킨 오류 유형에 따라 다르게 반응하며 활성화된 코드 부분을 나타내는 메시지를 출력합니다.

```
PRINT 'Before TRY...CATCH block.'
BEGIN TRY
  PRINT ' Entering TRY block.'
  INSERT INTO Employees(empid, empname, mgrid) VALUES(2, 'Emp2' , 1)
  PRINT 'After INSERT.'
  PRINT ' Exiting TRY block.'
END TRY

BEGIN CATCH
  PRINT ' Entering CATCH block.'
  IF ERROR_NUMBER(  ) = 2627
  BEGIN
    PRINT '  Handling PK violation...'
  END
  ELSE IF ERROR_NUMBER(  ) = 547
  BEGIN
    PRINT '  Handling CHECK/FK constraint violation...'
  END
  ELSE IF ERROR_NUMBER(  ) = 515
  BEGIN
    PRINT '  Handling NULL violation...'
  END
  ELSE IF ERROR_NUMBER(  ) = 245
  BEGIN
    PRINT '  Handling conversion error...'
  END
  ELSE
  BEGIN
    PRINT '  Handling unknown error...'
  END
  PRINT '  Error Number : ' + CAST(ERROR_NUMBER(  ) AS VARCHAR(10))
  PRINT '  Error Message : ' + ERROR_MESSAGE(  )
  PRINT '  Error Severity: ' + CAST(ERROR_SEVERITY(  ) AS VARCHAR(10))
  PRINT '  Error State  : ' + CAST(ERROR_STATE(  ) AS VARCHAR(10))
  PRINT '  Error Line  : ' + CAST(ERROR_LINE(  ) AS VARCHAR(10))
  PRINT '  Error Proc  : ' + ISNULL(ERROR_PROCEDURE(  ), 'Not within proc' )
  PRINT ' Exiting CATCH block.'
END CATCH
PRINT 'After TRY...CATCH block.'
```

ERROR_NUMBER( ) 함수는 CATCH 블록에서 여러 번 호출되며 항상 컨트롤이 CATCH 블록에 전달되게 만든 오류의 개수를 반환한다는 점을 유념하십시오. 이 코드는 직원 2를 이전에 삽입한 직원 1의 부하 직원으로 삽입하며 처음으로 실행될 때 아무 오류 없이 완료되어 다음과 같은 출력을 나타냅니다.

```
Before TRY...CATCH block.
  Entering TRY block.

    After INSERT.
  Exiting TRY block.
After TRY...CATCH block.
```

CATCH 블록을 건너뛴다는 점에 유의하십시오. 이 코드를 두 번째로 실행하면 다음과 같은 출력이 나타납니다.

```
Before TRY...CATCH block.
  Entering TRY block.
  Entering CATCH block.
    Handling PK violation...
    Error Number  : 2627
      Error Message : Violation of PRIMARY KEY constraint 'PK_Employees'. Cannot insert duplicate key in object
 'dbo.Employees'.
    Error Severity: 14
    Error State   : 1
    Error Line    : 4
    Error Proc    : Not within proc
  Exiting CATCH block.
After TRY...CATCH block.
```

TRY 블록이 입력되었지만 완료되지 않았음을 유념하십시오. 기본 키 위반 오류의 결과, 오류를 식별하고 처리한 CATCH 블록으로 컨트롤이 전달되었습니다. 이와 마찬가지로 0과 같이 유효하지 않은 직원 ID 데이터의 값을 지정하여 CHECK 제약 조건을 위반하면(employeeid에서 허용되지 않는 NULL, INT로 변환될 수 없는 'a') 해당 오류가 발생하며 적절한 코드 처리가 활성화됩니다.

TRY 블록에 명시적인 트랜잭션을 사용하는 경우 동작의 경과를 파악하기 위해 CATCH 블록의 오류 처리 코드에서 트랜잭션 상태를 조사하고자 할 수도 있습니다. SQL Server 2005는 트랜잭션 상태를 반환하는 XACT_STATE( )라는 새로운 함수를 제공합니다. 이 함수의 가능한 반환 값은 0, -1, 1입니다. 0이라는 반환 값은 열린 트랜잭션이 없음을 의미하며 트랜잭션을 커밋 또는 롤백하려고 시도하면 오류가 발생합니다. 1이라는 반환 값은 하나의 열린 트랜잭션을 커밋 또는 롤백할 수 있다는 것을 의미합니다. 각자의 요구와 오류 처리 논리에 따라 트랜잭션의 커밋 또는 롤백 여부를 결정해야 합니다. -1이라는 반환 값은 하나의 트랜잭션이 열려 있지만 커밋할 수 없는 상태임을 의미하는 것으로 SQL Server 2005에서 새롭게 제공하는 트랜잭션 상태입니다. TRY 블록 내 트랜잭션은 트랜잭션 중단을 야기할 수도 있는 오류가 발생하면 커밋할 수 없는 상태를 입력합니다(일반적으로 심각도가 17 이상이거나 XACT_ABORT 세션 옵션이 ON으로 설정된 경우). 커밋할 수 없는 트랜잭션은 모든 열린 잠금을 유지하며 데이터의 읽기만 허용합니다. 트랜잭션 로그에 대한 쓰기를 요구하는 작업을 전송할 수 없으며, 이는 트랜잭션이 커밋할 수 없는 상태인 경우 데이터를 변경할 수 없음을 의미합니다. 트랜잭션을 종료하기 위해 롤백을 실행해야 합니다.

수정 사항이 수용되기 전에는 트랜잭션을 커밋할 수 없고 오직 롤백만 가능합니다. 다음 예제는 XACT_STATE( ) 함수를 사용하는 방법을 설명합니다.

```
BEGIN TRY
  BEGIN TRAN
    INSERT INTO Employees(empid, empname, mgrid) VALUES(3, 'Emp3' , 1)

    /* other activity */
  COMMIT TRAN
  PRINT 'Code completed successfully.'
END TRY
BEGIN CATCH
  PRINT 'Error: ' + CAST(ERROR_NUMBER( ) AS VARCHAR(10)) + ' found.'
  IF (XACT_STATE( )) = -1
  BEGIN
        PRINT 'Transaction is open but uncommittable.'
        /* ...investigate data... */
        ROLLBACK TRANSACTION -- can only ROLLBACK
        /* ...handle error... */
  END
  ELSE IF (XACT_STATE( )) = 1
  BEGIN
        PRINT 'Transaction is open and committable.'
        /* ...handle error... */
        COMMIT TRANSACTION -- or ROLLBACK
  END
  ELSE
  BEGIN
        PRINT 'No open transaction.'
        /* ...handle error... */
  END
END CATCH
```

TRY 블록은 암시적인 트랜잭션 내 코드를 전송합니다. 이 블록은 새로운 직원 행을 삽입하고 동일한 트랜잭션에서 다른 작업을 수행합니다. CATCH 블록은 오류 개수를 출력하며 동작의 경과를 파악하기 위해 트랜잭션 상태를 조사합니다. 트랜잭션이 열려 있고 커밋할 수 없다면 CATCH 블록은 데이터를 조사하고 트랜잭션을 롤백한 다음 데이터 수정을 요구하는 수정 조치를 실행합니다. 트랜잭션이 열려 있고 커밋할 수 있다면 CATCH 블록은 오류와 커밋을 처리합니다(또는 롤백할 수도 있음). 열린 트랜잭션이 없을 경우 오류가 처리되며 커밋이나 롤백이 실행되지 않습니다. 처음으로 이 코드를 실행하면 직원 3에 대한 새로운 직원 행이 삽입되고 코드는 다음과 같은 메시지를 출력하며 성공적으로 완료됩니다.

```
Code completed successfully.
```

이 코드를 두 번째로 실행하면 기본 키 위반 오류가 발생하고 다음과 같은 메시지가 출력됩니다.

```
Error: 2627 found.
Transaction is open and committable.
```

XACT_ABORT 세션 옵션을 ON으로 설정하고 이 코드를 세 번째로 실행하면 기본 키 위반 오류가 발생하고 다음과 같은 메시지가 출력됩니다.

```
Error: 2627 found.
Transaction is open but uncommittable.
```

Transact-SQL에 영향을 미치는 기타 SQL Server 2005 기능

이 섹션에서는 Transact-SQL에 영향을 미치는 SQL Server 2005의 다른 기능 향상에 대해 간략하게 설명합니다. TOP, DML(Data Manipulation Language) with results, 동적 열에 대한 MAX 지정자, XML/XQuery, DDL(Data Definition Language) 트리거, Queuing 및 SQL Server Service Broker, DML 이벤트 및 알림 등의 기능이 향상되었습니다.

TOP 기능 향상

SQL Server 버전 7.0과 SQL Server 2000에서는 TOP 옵션을 통해 SELECT 쿼리가 반환하는 행의 개수나 비율을 제한할 수 있지만 상수를 인수로 제공해야 합니다. SQL Server 2005에서는 TOP의 기능이 다음과 같이 개선되었습니다.

이제 변수와 하위 쿼리를 선택적으로 사용하여 쿼리의 영향을 받게 될 행의 개수 또는 비율을 반환하는 숫자 식을 지정할 수 있습니다.

DELETE, UPDATE, INSERT 쿼리에 TOP 옵션을 사용할 수 있습니다.
TOP 옵션을 사용한 쿼리의 새로운 구문은 다음과 같습니다.

```
SELECT [TOP (〈expression〉) [PERCENT] [WITH TIES]]
FROM 〈table_name〉 ¼ [ORDER BY ¼ ]
DELETE [TOP (〈expression〉) [PERCENT]] FROM 〈table_name〉 ...
UPDATE [TOP (〈expression〉) [PERCENT]] 〈table_name〉 SET ...
INSERT [TOP (〈expression〉) [PERCENT]] INTO 〈table_name〉 ...
```

숫자 식은 괄호 안에 지정되어야 합니다. 괄호 없이 상수를 지정하는 것은 이전 버전과의 호환성을 위해 SELECT 쿼리에서만 지원됩니다. 식은 독립적이어야 합니다. 하위 쿼리를 사용하는 경우 외부 쿼리에 있는 테이블의 열을 참조할 수 없습니다. PERCENT 옵션을 지정하지 않을 경우 식은 암시적으로 bigint 데이터 형식으로 변환할 수 있어야 합니다. PERCENT 옵션을 지정하는 경우 식은 암시적으로 float로 변환 가능하며 0-100 범위에 포함되어야 합니다. WITH TIES 옵션과 ORDER BY 절은 SELECT 쿼리에서만 지원됩니다.

예를 들어 다음 쿼리는 변수를 TOP 옵션의 인수로 사용하여 가장 최근 구매 주문의 지정된 개수를 반환합니다.

```
USE AdventureWorks
DECLARE @n AS BIGINT
SET @n = 2
SELECT TOP(@n) *
FROM Purchasing.PurchaseOrderHeader
ORDER BY OrderDate DESC
```

이러한 기능 향상은 요청된 행의 개수를 저장 프로시저 또는 사용자 정의 함수의 인수로 가져오는 경우에 특히 유용합니다. 독립적인 하위 쿼리를 사용함으로써 "월간 평균 주문 개수를 계산하고 가장 최근 접수된 주문 수를 반환하는" 것과 같은 동적 요청에 응답할 수 있습니다.

```
USE AdventureWorks
SELECT TOP(SELECT
            COUNT(*)/DATEDIFF(month, MIN(OrderDate), MAX(OrderDate))
           FROM Purchasing.PurchaseOrderHeader) *
FROM Purchasing.PurchaseOrderHeader
ORDER BY OrderDate DESC
```

SQL Server의 이전 버전에서 제공되는 SET ROWCOUNT 옵션을 사용하면 쿼리의 영향을 받는 행의 개수를 제한할 수 있습니다. 예를 들어, SET ROWCOUNT는 일반적으로 단 하나의 대용량 트랜잭션 대신 여러 개의 소용량 트랜잭션에서 많은 양의 데이터를 주기적으로 지우는 데 사용됩니다.

```
SET ROWCOUNT 1000
DELETE FROM BigTable WHERE datetimecol < '20000101'
WHILE @@rowcount > 0
  DELETE FROM BigTable WHERE datetimecol < '20000101'
SET ROWCOUNT 0
```

이와 같은 방식으로 SET ROWCOUNT를 사용하면 지우기가 실행되는 동안 트랜잭션 로그를 백업하고 재활용할 수 있을 뿐 아니라 잠금 확대(lock escalation)를 방지할 수 있습니다. SET ROWCOUNT를 사용하는 대신 TOP를 이런 식으로 사용할 수 있습니다.

```
DELETE TOP(1000) FROM BigTable WHERE datetimecol < '20000101'
WHILE @@rowcount > 0
  DELETE TOP(1000) FROM BigTable WHERE datetimecol < '20000101'
```

TOP 옵션을 사용할 때 최적화 프로그램은 "row-goal"의 정의와 TOP 사용 여부를 알려 줄 수 있기 때문에 최적화 프로그램은 보다 효율적인 계획을 생성할 수 있습니다.

SELECT 쿼리에 항상 지정할 수 있기 때문에 INSERT 문에 TOP를 사용할 필요가 없다고 생각할 수도 있지만 EXEC 명령의 결과 또는 UNION 연산의 결과를 삽입할 때 유용하다는 것을 알 수 있습니다.

```
INSERT TOP ... INTO ...
  EXEC ...
INSERT TOP ... INTO ...
  SELECT ... FROM T1
  UNION ALL
  SELECT ... FROM T2
  ORDER BY ...
```

DML with Results

SQL Server 2005는 수정문(INSERT, UPDATE, DELETE)에서 데이터를 반환할 수 있도록 하는 새로운 OUTPUT 절을 도입했습니다. INTO 절과 함께 OUTPUT 절을 사용하여 수정 작업의 영향을 받은 데이터를 테이블이나 테이블 변수로 보낼 수 있습니다. 대상이 테이블인 경우 정의된 트리거 또는 외래 키를 가질 수 없습니다.

INTO 없이 OUTPUT 절을 지정하는 경우 데이터는 호출자에게 반환됩니다. 그러나 대부분의 경우 데이터를 테이블로 보내고자 할 것입니다. 트리거를 사용하는 것과 비슷한 측면에서 생각해 보십시오. 트리거 내 쿼리는 데이터를 호출자에게 반환할 수 있지만 혼란을 초래할 수 있으므로 차단되어야 합니다. 동일한 수정문에 INTO가 있는 OUTPUT과 INTO가 없는 OUTPUT을 모두 지정하여 데이터를 테이블/테이블 변수로 보낼 수 있을 뿐 아니라 호출자에게 반환할 수 있다는 점을 유념하십시오.

DML with results에 대한 유용한 시나리오에는 지우기 및 보관, 메시지 처리 응용 프로그램 등이 포함됩니다. 새로운 OUTPUT 절의 구문은 다음과 같습니다.

```
OUTPUT 〈dml_select_list〉 [INTO {table | @table_variable}]
```

트리거를 다루는 방식과 마찬가지로 삽입 및 삭제된 테이블을 참조하여 수정된 행의 기존/새 이미지를 액세스합니다. INSERT 문에서는 삽입된 테이블만 액세스할 수 있고, DELETE 문에서는 삭제된 테이블만 액세스할 수 있으며, UPDATE 문에서는 삽입 및 삭제된 테이블에 모두 액세스할 수 있습니다.

DML with results가 유용하게 사용될 수 있는 지우기 및 보관 시나리오의 예로서, 대용량 Orders 테이블을 보유하고 있으며 주기적으로 과거 데이터를 지워야 하는 상황을 가정해 봅시다. 또한 지운 데이터를 OrdersArchive라는 보관 테이블에 복제하고자 합니다. @DeletedOrders라는 테이블 변수를 선언하고(선택적으로 정기/임시 테이블 생성 가능) 앞서 다룬 'TOP 기능 향상' 섹션에서 설명한 지우기 방법을 사용해 청크의 과거 데이터(2003년 이전의 주문)를 삭제하는 루프를 입력합니다. 여기에 삭제된 모든 행의 모든 변수를 @DeletedOrders 테이블 변수로 복제한 다음 INSERT INTO 문을 사용해 테이블 변수의 모든 행을 OrdersArchive 테이블로 복제하는 OUTPUT 절이 추가됩니다.

```
DECLARE @DeletedOrders TABLE -- can also be a regular/temp table
(
  orderid   INT,
  orderdate DATETIME,
  empid     INT,
  AS VARCHAR(5)),
  qty       INT
)
WHILE 1=1
BEGIN
  BEGIN TRAN
    DELETE TOP(5000) FROM Orders
      OUTPUT deleted.* INTO @DeletedOrders
    WHERE orderdate < '20030101'
    INSERT INTO OrdersArchive
      SELECT * FROM @DeletedOrders
    IF @@rowcount < 5000
    BEGIN
      COMMIT TRAN
      BREAK
    END
  COMMIT TRAN
  DELETE FROM @DeletedOrders -- if regular/temp table use TRUNCATE
END
```

메시지 처리 시나리오의 예로 다음 Messages 테이블을 고려해 보십시오.

```
USE tempdb
CREATE TABLE Messages
(
  msgid   INT          NOT NULL IDENTITY ,
  msgdate DATETIME     NOT NULL DEFAULT(GETDATE(    )),
  msg     VARCHAR(MAX) NOT NULL,
  status  VARCHAR(20)  NOT NULL DEFAULT( 'new' ),
  CONSTRAINT PK_Messages
    PRIMARY KEY NONCLUSTERED(msgid),
  CONSTRAINT UNQ_Messages_status_msgid
    UNIQUE CLUSTERED(status, msgid),
  CONSTRAINT CHK_Messages_status
    CHECK (status IN( 'new' ,  'open' ,  'done' ))
)
```

각 메시지에 대해 메시지 ID, 입력 일자, 메시지 텍스트, 그리고 메시지가 아직 처리되지 않았는지("new"), 처리되는 중인지("open"), 아니면 이미 처리되었는지("done")를 나타내는 상태를 저장합니다.

다음 코드는 매초마다 임의의 텍스트로 메시지를 삽입하는 루프를 사용함으로써 메시지를 생성하는 세션을 시뮬레이션합니다. 새로 삽입된 메시지는 상태 열에 기본값인 'new' 가 지정되어 있기 때문에 메시지 상태가 "new" 입니다. 여러 세션에서 이 코드를 동시에 실행합니다.

```
USE tempdb
SET NOCOUNT ON
DECLARE @msg AS VARCHAR(MAX)
WHILE 1=1
BEGIN
  SET @msg = 'msg' + RIGHT( '000000000'
    + CAST(CAST(RAND(    )*2000000000 AS INT)+1 AS VARCHAR(10)), 10)
  INSERT INTO dbo.Messages(msg) VALUES(@msg)
  WAITFOR DELAY '00:00:01' ;
END
```

다음 코드는 아래의 단계에 따라 메시지를 처리하는 세션을 시뮬레이션합니다.

1. 끊임없이 메시지를 처리하는 무한 루프를 형성합니다.
2. 잠근 열을 건너뛰는 READPAST 힌트와 함께 UPDATE TOP(1) 문(또는 여러 개의 메시지에는 더 높은 값)을 사용하여 하나의 사용 가능한 새 메시지를 잠그고 메시지 상태를 "open" 으로 변경합니다.
3. OUTPUT 절을 사용하여 @Msgs 테이블 변수(선택적으로 임시 테이블 사용 가능)에 메시지 속성을 저장합니다.
4. 메시지를 처리합니다.
5. Messages 테이블과 @Msgs 테이블 변수를 조인하여 메시지 상태를 "done" 으로 설정합니다.
6. Messages 테이블에서 새로운 데이터가 발견되지 않으면 잠시 기다립니다.

여러 세션에서 다음 코드를 실행합니다.

```
USE tempdb
SET NOCOUNT ON
-- can also be a temp table
DECLARE @Msgs TABLE(msgid INT, msgdate DATETIME, msg VARCHAR(MAX))
WHILE 1 = 1
BEGIN
  -- Use a value higher than 1 to process multiple messages in one go
  UPDATE TOP(1) Messages WITH(READPAST) SET status = 'open'
    OUTPUT inserted.msgid, inserted.msgdate, inserted.msg
    INTO @Msgs
  WHERE status = 'new'
  IF @@rowcount > 0
  BEGIN
    PRINT 'Processing message...'
```

```
    -- process message here
    SELECT * FROM @msgs
    UPDATE M
      SET status =  'done'
    FROM Messages AS M
      JOIN @Msgs AS N
        ON M.msgid = N.msgid
    DELETE FROM @Msgs -- if temp table use TRUNCATE
  END
  ELSE
  BEGIN
    PRINT  'No messages to process.'
    WAITFOR DELAY  '00:00:01'
  END
END
```

시뮬레이션 실행을 마치면 메시지를 삽입 및 처리하는 모든 세션을 중단시키고 Messages 테이블을 삭제합니다.

```
USE tempdb
DROP TABLE Messages
```

동적 열에 대한 MAX 지정자

SQL Server 2005는 〈datatype〉(MAX) 구문을 사용하는 MAX 지정자를 새롭게 제공하여 가변 길이 데이터 형식 VARCHAR, NVARCHAR 및 VARBINARY의 기능을 향상시킵니다. MAX 지정자를 포함한 가변 길이 데이터 형식은 향상된 기능으로 TEXT, NTEXT 및 IMAGE 데이터 형식을 대체합니다. TEXT, NTEXT 및 IMAGE와 같은 LOB(Large Object) 데이터 형식 대신 MAX 지정자를 포함한 가변 길이 데이터 형식을 사용하면 여러 가지 이점이 제공됩니다. SQL Server는 값을 직렬로 저장하는 시기와 포인터를 사용하는 시기를 내부적으로 파악하기 때문에 명시적인 포인터 조작이 필요 없습니다. 이제 데이터의 크기와 상관없이 통합된 프로그래밍 모델을 사용할 수 있습니다. MAX 지정자를 포함한 가변 길이 데이터 형식은 열, 변수, 매개 변수, 비교, 트리거, 모든 문자열 함수 등에서 지원됩니다.

MAX 지정자를 사용하는 예로, 다음 코드는 CustomerData라는 테이블을 생성합니다.

```
CREATE TABLE CustomerData
(
  custid INT  NOT NULL PRIMARY KEY,
  txt_data    VARCHAR(MAX)   NULL,
  ntxt_data   NVARCHAR(MAX)  NULL,

  binary_data VARBINARY(MAX) NULL
)
```

테이블에는 기본 키로 사용되는 custid 열과 대용량 데이터를 저장할 수 있는 VARCHAR(MAX), NVARCHAR(MAX) 및 VARBINARY(MAX) 데이터 형식으로 각각 정의되는 txt_data, ntxt_data 및 binary_data라는 NULL 값을 사용할 수 있는 열이 포함됩니다.

MAX 지정자를 이용해 동적 열에서 청크를 읽으려면 표준 동적 열을 사용하는 것과 동일한 방식으로 SUBSTRING 함수를 사용합니다. 청크를 업데이트하려면 WRITE 메서드를 제공하는 UPDATE 문의 향상된 구문을 사용합니다. 향상된 UPDATE 문의 구문은 다음과 같습니다.

```
UPDATE table_name
SET column_name.WRITE(@chunk, @offset, @len)
WHERE ...
```

WRITE 메서드는 @offset 위치에서 @len 문자를 제거하고 해당 위치에 @chunk를 삽입합니다. @offset은 0을 기준으로 하는데 이는 offset 0이 @chunk의 첫 번째 문자의 위치를 나타낸다는 것을 의미합니다. WRITE 메서드의 사용을 시연하기 위해 우선 CustomerData 테이블에 고객 ID가 102, txt_data 열 값이 'Customer 102 text data' 인 행을 삽입합니다.

```
INSERT INTO CustomerData(custid,txt_data)
  VALUES(102,  'Customer 102 text data' )
```

다음 UPDATE 문은 '102' 를 'one hundred and two' 로 바꿉니다.

```
UPDATE CustomerData
  SET txt_data.WRITE( 'one hundred and two' , 9, 3)
WHERE custid = 102
```

@chunk가 NULL이면 @len은 무시되고 값이 @offset 위치에서 잘립니다. 다음 명령문은 offset 28에서 끝까지의 모든 데이터를 제거합니다.

```
UPDATE CustomerData
  SET txt_data.WRITE(NULL, 28, 0)
WHERE custid = 102
```

@len이 NULL이면 @offset에서 끝까지의 모든 문자가 제거되고 @chunk를 덧붙입니다. 다음 명령문은 offset 9에서 끝까지의 모든 데이터를 제거하고 '102' 를 덧붙입니다.

```
UPDATE CustomerData
  SET txt_data.WRITE( '102' , 9, NULL)
WHERE custid = 102
```

@offset이 NULL이면 @len는 무시되고 @chunk가 끝에 덧붙습니다. 다음 명령문은 ‘is discontinued’ 라는 문자열을 끝에 덧붙입니다.

```
UPDATE CustomerData
  SET txt_data.WRITE( ‘ is discontinued’ , NULL, 0)
WHERE custid = 102
```

XML 및 XQuery

SQL Server 2005에는 XML 구조 데이터를 저장, 쿼리 및 업데이트할 수 있도록 하는 몇 가지 XML 관련 향상 기능이 도입되었습니다. XML과 관련 데이터를 모두 동일한 데이터베이스에 저장할 수 있으므로 저장 및 쿼리 처리를 위해 기존의 데이터베이스 엔진을 활용할 수 있습니다.

새로 도입된 xml 데이터 형식은 테이블 열에 사용할 수 있을 뿐 아니라 인덱싱도 가능합니다. 또한 xml 데이터 형식은 변수, 뷰, 함수 및 저장 프로시저에 사용할 수도 있습니다. xml 데이터 형식은 관계형 FOR XML 쿼리에 의해 생성되거나 OPENXML을 사용하여 관계형 행 집합으로서 액세스할 수 있습니다. 스키마를 데이터베이스로 가져오거나 데이터베이스에서 내보낼 수 있습니다. 스키마를 사용해 XML 데이터를 검증하고 제한할 수 있으며, XQuery를 사용해 XML 형식 데이터를 쿼리 및 수정할 수 있습니다. 트리거, 복제, 대량 복사, DBCC, 전체 텍스트 검색에서 xml 데이터 형식이 지원됩니다. 그러나 xml은 비교가 불가능합니다. 즉, xml 열에 PRIMARY KEY, UNIQUE 또는 FOREIGN KEY 제약 조건을 정의할 수 없습니다.

다음 예제는 xml 데이터 형식을 사용합니다. 다음 코드는 @x라는 XML 변수를 정의하고 고객 주문 데이터를 그 변수로 로드합니다.

```
USE AdventureWorks
DECLARE @x AS XML
SET @x = (SELECT C.CustomerID, O.SalesOrderID
        FROM Sales.Customer C
          JOIN Sales.SalesOrderHeader O
            ON C.CustomerID=O.CustomerID
        ORDER BY C.CustomerID
        FOR XML AUTO, TYPE)
SELECT @x
```

다음 코드는 xml 열이 있는 테이블을 만들고 OPENROWSET 함수를 사용해 XML 파일을 대량 로드합니다.

```
CREATE TABLE T1
(
  keycol INT NOT NULL PRIMARY KEY,
  xmldoc XML NULL
)

INSERT INTO T1(keycol, xmldoc)
  SELECT 1 AS keycol, xmldoc
  FROM OPENROWSET(BULK  'C:\documents\mydoc.xml' , SINGLE_NCLOB)
      AS X(xmldoc)
```

또한 SQL Server 2005에서는 W3C 표준 XML 쿼리 언어인 Xquery를 새롭게 지원합니다. Microsoft는 SQL Server의 표준 기능을 확장하여 삽입, 업데이트 및 삭제를 위해 XQuery를 사용할 수 있도록 했습니다. XQuery는 UDT(User-defined Type) 스타일 메서드를 통해 Transact-SQL과 함께 내장됩니다.

XQuery는 다음과 같은 쿼리 메서드를 제공합니다.

```
Manipulating XML data: @x.query (xquery string) returning XML
Checking existence: @x.exist (xquery string) returning bit
Returning a scalar value @x.value (xquery string, sql_type string) returning sql_type
```

XQuery는 @x.modify (xDML string)라는 수정 메서드를 제공합니다.

예를 들면 Jobs 테이블은 jobinfo 열에 XML 포맷의 작업 정보를 포함합니다. 다음 쿼리는 일부 기준에 부합하는 모든 열에 몇 가지 조작을 수행한 다음 ID와 XML 데이터를 반환합니다. 원하는 행을 필터링하기 위해 WHERE 절에 jobinfo.exist( ) 메서드를 호출합니다. 이 메서드는 jobinfo 열에 @date라는 Transact-SQL 변수보다 큰 date 속성을 가진 편집된 요소가 포함된 행에 대해서만 1을 반환합니다. 각각의 반환된 행에 대해, XML 결과는 jobinfo.query( ) 메서드를 호출함으로써 생성됩니다. jobinfo에서 제공되는 모든 작업 요소에 대해, query( ) 메서드는 해당 작업의 id 속성에 기반한 id 속성을 가진 jobschedule 요소를 생성하고, jobinfo의 start 및 end 속성에 기반한 데이터를 포함한 begin 및 end 하위 요소를 생성합니다.

```
SELECT id, jobinfo.query(
   'for $j in //job
    return
      〈jobschedule id=" {$j/@id}" 〉
        〈begin〉{data($j/@start)}〈/begin〉
        〈end〉{data($j/@end)}〈/end〉
      〈/jobschedule〉' )
FROM Jobs
WHERE 1 = jobinfo.exist(
   '//edited[@date 〉 sql:variable("@date")]' )
```

다음과 같이 value( ) 메서드를 호출하면 jobinfo의 첫 번째 작업의 start 속성이 Transact-SQL datetime 포맷으로 반환됩니다.

```
SELECT id, jobinfo.value( '(//job)[1]/@start' , 'DATETIME' ) AS startdt
FROM Jobs
WHERE id = 1
```

XQuery는 데이터를 수정하는 데 사용할 수도 있습니다. 예를 들어, 다음 코드를 사용해 직원 1에 대한 Employees 테이블의 empinfo XML 열을 업데이트할 수 있습니다. resume 요소의 편집된 하위 요소의 date 속성을 새로운 값으로 업데이트합니다.

```
UPDATE Employees SET empinfo.modify(
    'update /resume/edited/@date
    to xs:date("2000-6-20")' )
WHERE empid = 1
```

DDL 트리거

SQL Server의 이전 버전에서는 테이블에 대해 실행된 DML 문(INSERT, UPDATE, DELETE)에 대해서만 AFTER 트리거를 정의할 수 있습니다. 그러나 SQL Server 2005를 사용하면 전체 서버 또는 데이터베이스의 범위로 DDL 이벤트에 대한 트리거를 정의할 수 있습니다. CREATE_TABLE과 같은 개별 DDL 문이나 DDL_DATABASE_LEVEL_EVENTS와 같은 명령문 그룹에 대한 DDL 트리거를 정의할 수 있습니다. 트리거 내에서 eventdata( ) 함수를 액세스함으로써 트리거를 실행시킨 이벤트에 관한 데이터를 가져올 수 있습니다. 이 함수는 이벤트에 대한 XML 데이터를 반환합니다. 각 이벤트에 대한 스키마는 Server Events 기반 스키마를 상속합니다.

이벤트 정보는 다음과 같습니다.

이벤트가 발생한 시점
이벤트가 실행된 SPID
이벤트 유형
영향을 받는 개체
SET 옵션
이벤트를 발생시킨 Transact-SQL 문

SQL Server의 이전 버전에서 제공된 트리거와 유사한 DDL 트리거는 트리거를 야기한 트랜잭션의 컨텍스트에서 실행됩니다. 트리거를 발생시킨 이벤트를 실행 취소하기로 결정한 경우 ROLLBACK 명령어를 실행할 수 있습니다. 예를 들어 다음 트리거는 현재 데이터베이스에 새 테이블이 만들어지는 것을 차단합니다.

```
CREATE TRIGGER trg_capture_create_table ON DATABASE FOR CREATE_TABLE
AS
-- PRINT event information For DEBUG
PRINT  'CREATE TABLE Issued'
PRINT EventData(    )
-- Can investigate data returned by EventData(    ) and react accordingly.
RAISERROR( 'New tables cannot be created in this database.' , 16, 1)
ROLLBACK
GO
```

트리거가 생성된 데이터베이스에서 CREATE TABLE 문을 실행하면 다음과 같은 출력이 나타납니다.

```
CREATE TABLE T1(col1 INT)
CREATE TABLE Issued
〈EVENT_INSTANCE〉
  〈PostTime〉2003-04-17T13:55:47.093〈/PostTime〉
  〈SPID〉53〈/SPID〉
  〈EventType〉CREATE_TABLE〈/EventType〉
  〈Database〉testdb〈/Database〉
  〈Schema〉dbo〈/Schema〉
  〈Object〉T1〈/Object〉
  〈ObjectType〉TABLE〈/ObjectType〉
  〈TSQLCommand〉
    〈SetOptions ANSI_NULLS="ON"  ANSI_NULL_DEFAULT="ON"  ANSI_PADDING="ON"
              QUOTED_IDENTIFIER="ON"  ENCRYPTED="FALSE"  /〉
    〈CommandText〉CREATE TABLE T1(col1 INT)〈/CommandText〉
  〈/TSQLCommand〉
〈/EVENT_INSTANCE〉
```

.Net SqlClient Data Provider: Msg 50000, Level 16, State 1, Procedure trg_capture_create_table, Line 10
New tables cannot be created in this database.
.Net SqlClient Data Provider: Msg 3609, Level 16, State 1, Line 1
Transaction ended in trigger. Batch has been aborted.

본 백서에서는 가독성을 높이기 위해 XML 출력이 수작업으로 포맷되었음을 유념하십시오. 이 코드를 실행하면 서식 없는 XML 출력을 가져오게 됩니다.
트리거를 삭제하려면 다음 명령문을 실행합니다.

```
DROP TRIGGER trg_capture_create_table ON DATABASE
```

DDL 트리거는 특히 DDL 변경에 대한 무결성 검사, 감사 시나리오 등에 유용하게 사용됩니다. DDL 무결성 적용의 예제로서, 다음과 같은 데이터베이스 수준의 트리거는 기본 키 없이 테이블을 만드려는 시도를 차단합니다.

```
CREATE TRIGGER trg_create_table_with_pk ON DATABASE FOR CREATE_TABLE
AS
DECLARE @eventdata AS XML, @objectname AS NVARCHAR(257),
  @msg AS NVARCHAR(500)

SET @eventdata = eventdata(    )
SET @objectname =
  N' [ ' + CAST(@eventdata.query( 'data(//SchemaName)' ) AS SYSNAME)
  + N' ].[ ' +
  CAST(@eventdata.query( 'data(//ObjectName)' ) AS SYSNAME) + N' ]'
IF OBJECTPROPERTY(OBJECT_ID(@objectname),  'TableHasPrimaryKey' ) = 0
BEGIN
  SET @msg = N' Table  ' + @objectname +  ' does not contain a primary key.'
    + CHAR(10) + N' Table creation rolled back.'
  RAISERROR(@msg, 16, 1)
  ROLLBACK
  RETURN
END
```

CREATE TABLE 문이 실행될 때 트리거가 야기됩니다. 트리거는 XQuery를 사용하여 스키마와 개체 이름을 추출하고, OBJECTPROPERTY 함수를 사용하여 테이블에 기본 키가 포함되어 있는지 여부를 확인합니다. 기본 키가 없을 경우 트리거는 오류를 발생시키고 트랜잭션을 롤백합니다. 트리거를 생성한 후, 기본 키 없이 테이블을 만들려는 다음과 같은 시도는 실패합니다.

```
CREATE TABLE T1(col1 INT NOT NULL)
Msg 50000, Level 16, State 1, Procedure trg_create_table_with_pk, Line 19
Table [dbo].[T1] does not contain a primary key.
Table creation rolled back.
Msg 3609, Level 16, State 2, Line 1
Transaction ended in trigger. Batch has been aborted.
```

다음 명령문은 성공적으로 실행됩니다.

```
CREATE TABLE T1(col1 INT NOT NULL PRIMARY KEY)
```

트리거와 테이블 T1을 삭제하려면 다음 코드를 실행합니다.

```
DROP TRIGGER trg_create_table_with_pk ON DATABASE
DROP TABLE T1
```

감사 트리거의 예로서 다음 데이터베이스 수준의 트리거는 AuditDDLEvents 테이블에 대한 모든 DDL 문을 감사합니다.

```
CREATE TABLE AuditDDLEvents
(
  LSN              INT      NOT NULL IDENTITY,
  posttime         DATETIME NOT NULL,
  eventtype        SYSNAME  NOT NULL,

  loginname        SYSNAME  NOT NULL,
  schemaname       SYSNAME  NOT NULL,
  objectname       SYSNAME  NOT NULL,
  targetobjectname SYSNAME  NOT NULL,
  eventdata        XML      NOT NULL,
  CONSTRAINT PK_AuditDDLEvents PRIMARY KEY(LSN)
)
GO
CREATE TRIGGER trg_audit_ddl_events ON DATABASE FOR DDL_DATABASE_LEVEL_EVENTS
AS
DECLARE @eventdata AS XML
SET @eventdata = eventdata(    )
INSERT INTO dbo.AuditDDLEvents(
  posttime, eventtype, loginname, schemaname,
  objectname, targetobjectname, eventdata)
  VALUES(
    CAST(@eventdata.query( 'data(//PostTime)' )         AS VARCHAR(23)),
    CAST(@eventdata.query( 'data(//EventType)' )        AS SYSNAME),
    CAST(@eventdata.query( 'data(//LoginName)' )        AS SYSNAME),
    CAST(@eventdata.query( 'data(//SchemaName)' )       AS SYSNAME),
    CAST(@eventdata.query( 'data(//ObjectName)' )       AS SYSNAME),
    CAST(@eventdata.query( 'data(//TargetObjectName)' ) AS SYSNAME),
    @eventdata)
GO
```

트리거는 단순히 XQuery를 사용하여 관심 있는 모든 이벤트 속성들을 eventdata( ) 함수로부터 추출하고 이를 AuditDDLEvents 테이블에 삽입합니다. 트리거를 테스트하려면 몇 개의 DDL 문을 전송하고 감사 테이블을 쿼리합니다.

```
CREATE TABLE T1(col1 INT NOT NULL PRIMARY KEY)
ALTER TABLE T1 ADD col2 INT NULL
ALTER TABLE T1 ALTER COLUMN col2 INT NOT NULL
CREATE NONCLUSTERED INDEX idx1 ON T1(col2)
SELECT * FROM AuditDDLEvents
```

최근 24시간 내에 테이블 T1의 스키마를 누가 변경했고 스키마가 어떻게 변경되었는지 확인하려면 다음 쿼리를 실행합니다.

```
SELECT posttime, eventtype, loginname,
  CAST(eventdata.query( 'data(//TSQLCommand)' ) AS NVARCHAR(2000))
  AS tsqlcommand
FROM dbo.AuditDDLEvents
WHERE schemaname = N' dbo'  AND N' T1'  IN(objectname, targetobjectname)
ORDER BY posttime
```

이미 생성한 트리거와 테이블을 삭제하려면 다음 코드를 실행합니다.

```
DROP TRIGGER trg_audit_ddl_events ON DATABASE
DROP TABLE dbo.T1
DROP TABLE dbo.AuditDDLEvents
```

서버 수준 감사 트리거의 예로, 다음 트리거는 AuditDDLLogins라는 감사 테이블에 대한 모든 DDL 로그인 관련 이벤트를 감사합니다.

```
USE master
CREATE TABLE dbo.AuditDDLLogins
(
  LSN             INT      NOT NULL IDENTITY,
  posttime        DATETIME NOT NULL,
  eventtype       SYSNAME  NOT NULL,
  loginname       SYSNAME  NOT NULL,
  objectname      SYSNAME  NOT NULL,
  logintype       SYSNAME  NOT NULL,
  eventdata       XML      NOT NULL,
  CONSTRAINT PK_AuditDDLLogins PRIMARY KEY(LSN)
)
CREATE TRIGGER audit_ddl_logins ON ALL SERVER
  FOR DDL_LOGIN_EVENTS
AS
DECLARE @eventdata AS XML
SET @eventdata = eventdata(    )
INSERT INTO master.dbo.AuditDDLLogins(
  posttime, eventtype, loginname,
  objectname, logintype, eventdata)

  VALUES(
    CAST(@eventdata.query( 'data(//PostTime)' )        AS VARCHAR(23)),
    CAST(@eventdata.query( 'data(//EventType)' )       AS SYSNAME),
    CAST(@eventdata.query( 'data(//LoginName)' )       AS SYSNAME),
    CAST(@eventdata.query( 'data(//ObjectName)' )      AS SYSNAME),
    CAST(@eventdata.query( 'data(//LoginType)' )       AS SYSNAME),
    @eventdata)
GO
```

트리거를 테스트하려면 로그인을 생성, 변경 및 삭제하는 다음과 같은 DDL 로그인 문을 실행한 다음 감사 테이블을 쿼리합니다.

```
CREATE LOGIN login1 WITH PASSWORD = '123'
ALTER LOGIN login1 WITH PASSWORD = 'xyz'
DROP LOGIN login1
SELECT * FROM AuditDDLLogins
```

트리거와 감사 테이블을 삭제하려면 다음 코드를 실행합니다.

```
DROP TRIGGER audit_ddl_logins ON ALL SERVER
DROP TABLE dbo.AuditDDLLogins
DROP DATABASE testdb
```

DDL 및 시스템 이벤트 알림

SQL Server 2005에서는 DDL과 시스템 이벤트를 포착하여 Service Broker 구축으로 이벤트 알림을 보낼 수 있습니다. 동시에 처리되는 트리거와는 달리, 이벤트 알림은 비동기 방식으로 실행되는 이벤트 메커니즘입니다. 이벤트 알림은 지정된 Service Broker 서비스로 XML 데이터를 전송하고 이벤트 사용자는 비동기적으로 이 데이터를 사용합니다. 이벤트 사용자는 WAITFOR 구문에 대한 확장을 사용하여 도착할 새 데이터를 기다릴 수 있습니다.

이벤트 알림을 정의하는 기준은 다음과 같습니다.

범위(SERVER, DATABASE, ASSEMBLY, 개별 개체)
이벤트 또는 이벤트 그룹의 목록(예: CREATE_TABLE, DDL_EVENTS 등)
SQL Server Events 메시지 유형 및 계약을 실행하는 Broker 서비스
Broker 서비스를 수행하는 Broker 인스턴스

이벤트 데이터는 SQL Server Events 스키마를 사용하여 XML 포맷으로 전송됩니다. 이벤트 알림을 생성하기 위한 일반적인 구문은 다음과 같습니다.

```
CREATE EVENT NOTIFICATION 〈name〉
  ON 〈scope〉
  FOR 〈list_of_event_or_event_groups〉
  TO SERVICE 'broker_service' , { 'broker_instance_specifier' | 'current database' }
```

이벤트 알림이 생성되면 시스템이 제공하는 초기화 서비스(initiating service)와 사용자가 지정한 대상 서비스 사이에 Service Broker 대화가 설정됩니다. broker_service는 SQL Server가 이벤트에 대한 데이터를 전달하기 위해 대화를 시작할 때 사용하는 대상 서비스를 지정합니다. broker_instance_specifier는 대상 서비스가 포함되어 있는 데이터베이스를 나타내는 GUID입니다. sys.databases 카탈로그 뷰의 service_broker_guid 열을 쿼리함으로써 이러한 GUID를 볼 수 있습니다. 현재 데이터베이스의 Broker를 짧게 줄여 'current database' 라는 문자열을 사용할 수 있습니다(master 데이터베이스에는 Broker가 없음에 유의). 대상 서비스는 SQL Server Events 메시지 유형 및 계약을 실행해야 합니다(이름은 다음 예제에 표시). 이벤트 알림이 존재하는 이벤트가 발생하면 XML 메시지가 해당 이벤트 데이터로부터 구성되고 이벤트 알림의 대화를 통해 대상 서비스에 전송됩니다. 예를 들어, 다음 코드는 T1이라는 테이블을 생성하고, T1 테이블의 스키마가 변경될 때마다 특정 배치에 알림을 전송하는 이벤트 알림을 정의합니다.

```
CREATE TABLE dbo.T1(col1 INT);
GO
-- Create a queue.
CREATE QUEUE SchemaChangeQueue;
GO
--Create a service on the queue that references
--the event notifications contract.
CREATE SERVICE SchemaChangeService
  ON QUEUE SchemaChangeQueue
(
-- This is the predefined contract name for event notification
-- conversations
[http://schemas.microsoft.com/SQL/Notifications/PostEventNotification]
);
GO
--Create a route on the service to define the address
--to which Service Broker sends messages for the service.
CREATE ROUTE SchemaChangeRoute
  WITH SERVICE_NAME = 'SchemaChangeService',
  ADDRESS = 'LOCAL';
GO
--Create the event notification.
CREATE EVENT NOTIFICATION NotifySchemaChangeT1
ON DATABASE
  FOR ALTER_TABLE TO SERVICE 'SchemaChangeService', 'current database';
GO
```

그러면 다음 ALTER에 의해 XML 메시지가 SchemaChangeQueue를 기반으로 구축된 SchemaChangeService에 전송됩니다.

```
ALTER TABLE dbo.T1 ADD col2 INT;
```

그 다음 아래의 명령문을 사용하면 대기열에서 XML 메시지를 검색할 수 있습니다.

```
RECEIVE TOP (1) CAST(message_body AS nvarchar(MAX))
FROM SchemaChangeQueue
```

결과 출력은 다음과 같습니다(서식 제외).

```
〈EVENT_INSTANCE〉
〈EventType〉ALTER_TABLE〈/EventType〉
〈PostTime〉2004-06-15T11:16:32.963〈/PostTime〉
〈SPID〉55〈/SPID〉
〈ServerName〉MATRIX\S1〈/ServerName〉
〈LoginName〉MATRIX\Gandalf〈/LoginName〉
〈UserName〉MATRIX\Gandalf〈/UserName〉
〈DatabaseName〉testdb〈/DatabaseName〉
〈SchemaName〉dbo〈/SchemaName〉
〈ObjectName〉T1〈/ObjectName〉
〈ObjectType〉TABLE〈/ObjectType〉
〈TSQLCommand〉
〈SetOptions ANSI_NULLS="ON" ANSI_NULL_DEFAULT="ON" ANSI_PADDING="ON" QUOTED_IDENTIFIER="ON"
ENCRYPTED="FALSE" /〉
〈CommandText〉ALTER TABLE dbo.T1 ADD col2 INT;〈/CommandText〉
〈/TSQLCommand〉
〈/EVENT_INSTANCE〉
```

WAITFOR 문은 다음과 같이 차단 모드에서 알림을 수신하는 데 사용할 수 있습니다.

```
WAITFOR (RECEIVE * FROM SchemaChangeQueue)
```

결론

SQL Server 2005의 Transact-SQL 기능 개선을 통해 쿼리 작성의 표현 능력이 향상되어 코드의 성능을 향상시키고 오류 관리 기능을 확장할 수 있습니다. Transact-SQL을 향상시키기 위한 지속적인 노력을 통해 SQL Server 내에서 Transact-SQL이 매우 중대한 역할을 담당한다는 확고한 신념을 확인할 수 있습니다.