**Новые возможности T-SQL в MS SQL Server 2005**

Гайдар Магдануров

**Предисловие**

В MS SQL Server 2005 появилось множество новшеств, позволяющих еще более продуктивно использовать эту СУБД. Многие из них, такие как новые типы данных, интеграция с платформой .NET, поддержка XML, новые функции ранжирования, улучшения в системе безопасности и прочее, уже были ранее описаны в журнале RSDN Magazine [1, 2, 3, 4, 5]. В этой статье будут рассмотрены новые операторы и функции работы с данными. В связи с грядущим в ноябре выходом финальной версии, приведенная информация является предельно актуальной для всех разработчиков, использующих SQL Server.

Помимо описания новых возможностей в статье приведены примеры кода, реализующие сходную функциональность, но работающие в предыдущих версиях SQL Server, поэтому эта статья может быть полезна не только читателям, планирующим использование SQL Server 2005, но и пользователям более ранних версий этой СУБД.

**Демонстрационная база данных**

В статье не будут рассматривать стандартные демонстрационные базы данных, поставляемые с SQL Server (Northwind, AdventureWorks). Для наибольшей наглядности примеров, создадим небольшую базу данных некоторого воображаемого магазина, торгующего ноутбуками.

Пользователю этой базы данных необходимо: иметь информацию о моделях, имеющихся в наличии на складе, осуществлять выборку моделей на основании цены и марки производителя, также, пользователю необходимо иметь возможность оформлять заказы клиентов и составлять очередь заказов на выполнение.

Для реализации этих желаний будет использована база данных состоящая из четырех таблиц: Brands - справочник производителей, позволяющий отслеживать отношения компаний с точки зрения «родительская-дочерняя»; Products – таблица имеющихся на складе моделей, содержащая необходимую информацию о цене, количестве, названии и базовой конфигурации; Orders – таблица, содержащая информацию о заказах; QrdersQueue – таблица, реализующая функциональность очереди заказов.

|  |
| --- |
| CREATE TABLE [dbo].[Brands](  [BrandID] [int] IDENTITY(1,1) NOT NULL,  [Name] [nvarchar](32) NOT NULL,  [ParentID] [int] NULL DEFAULT ((0))  )  CREATE TABLE [dbo].[Orders](  [OrderID] [int] IDENTITY(1,1) NOT NULL,  [Date] [datetime] NOT NULL,  [ProductID] [int] NOT NULL,  [Quantity] [int] NOT NULL DEFAULT ((1))  )  CREATE TABLE [dbo].[Products](  [ProductID] [int] IDENTITY(1,1) NOT NULL,  [BrandID] [int] NOT NULL,  [Model] [nvarchar](32) NOT NULL,  [Configuration] [nvarchar](128) NOT NULL,  [Price] [money] NOT NULL,  [Quantity] [numeric](18, 0) NOT NULL  )  CREATE TABLE [dbo].[OrdersQueue](  [QueueID] [int] IDENTITY(1,1) NOT NULL,  [OrderID] [int] NOT NULL  ) |

Новые возможности T-SQL

В начале статьи будут рассмотрены новые функции и операторы T-SQL в SQL Server 2005 и примеры их использования, изменения, затронувшие имевшийся ранее оператор TOP, после чего рассказано о новой возможности обработки ошибок в T-SQL.

Общие табличные выражения

Общие табличные выражения (Common Table Expressions, CTE) позволяют определять временные именованный набор данных, функционально похожий на представление (View), доступный в пределах пакета (batch).

Для упрощения дальнейшего изложения, введем более простой термин виртуальное представление вместо дословного английского перевода общее табличное выражение.

Виртуальные представления позволяют использовать однажды объявленное в процедуре представление вместо вложенных запросов, как это приходилось делать в более ранних версиях SQL Server, что значительно улучшает читаемость T-SQL кода. Для объявления представление используется синтаксис

|  |
| --- |
| WITH ИмяПредставления(ИмяПоля, ИмяПоля, …) AS (Подзапрос) |

где число полей в объявлении представления должно соответствовать числу получаемых полей в подзапросе.

Допустим, необходимо осуществить выборку всех моделей, цена которых превосходит среднюю цену в модельном ряду своего производителя.

Задачу можно решить, используя виртуальное представление

|  |
| --- |
| WITH AvgPrice(BrandID, Price) AS  (SELECT BrandID, AVG(Price) FROM Products GROUP BY BrandID)  SELECT P.Model FROM Products AS P INNER JOIN AvgPrice AS A ON  P.BrandID = A.BrandID AND P.Price > A.Price |

или используя вложенный подзапрос

|  |
| --- |
| SELECT P.Model FROM Products AS P  INNER JOIN (SELECT BrandID, AVG(Price) FROM Products GROUP BY BrandID) AS A ON  P.BrandID = A.BrandID AND P.Price > A.Price |

Как видно из примера, использование CTE и вложенных подзапросов эквивалентно, но, читатель легко может представить ситуацию, когда одно и то же виртуальное представление данных нужно использовать несколько раз в хранимой процедуре, в этом случае единожды объявленное представление с использованием CTE потребует значительно меньше SQL кода.

Рекурсия с использованием виртуальных представлений

Одним из основных преимуществ виртуальных представлений (CTE), является простое и наглядное построение рекурсивных выражений.

Достаточно часто встречаются таблицы с иерархической структурой данных («деревья»). В случае, когда необходимо получить уровень вложенности элемента, лучшим решением будет использование рекурсивного запроса с использованием виртуального представления. Допустим, в магазине, использующем демонстрационную базу данных, решили добавить возможность задавать «степень родства» фирм, для определения дочерних компаний известных брендов.

Для таблицы Brands нужно создать виртуальное представление, использующее рекурсию для получения уровня вложенности:

|  |
| --- |
| WITH C (BrandID, [Name], ParentID, NestingLevel) AS  (  SELECT B.BrandID, B.[Name], B.ParentID, 1 FROM Brands AS B WHERE ParentID = 0  UNION ALL  SELECT B.BrandID, B.[Name], B.ParentID, (NestingLevel + 1) FROM Brands AS B  INNER JOIN C ON C.BrandID = B.ParentID  )  SELECT \* FROM C |

Результатом выполнения запроса будет таблица, например такая:

|  |
| --- |
| BrandID Name ParentID NestingLevel  --------------------------------------------------  1 Parent1 0 1  2 Parent2 0 1  3 Child1 1 2  4 Child11 3 3  5 Child12 3 3 |

Общий принцип построения рекурсивного выражения

|  |
| --- |
| WITH ИмяCTE (Определение) AS  (  SELECT … -- Выборка с начальным условием,  UNION ALL -– Объединение результатов  SELECT … -- Выборка определяющаяя шаг рекурсии  INNER JOIN CTE.ДочернийID = ИмяТаблицы.РодительскийID –- Присоединение «по родителю»  ) |

Без использования виртуального представления, для достижения того же результата придется написать значительно более сложный запрос:

|  |
| --- |
| DECLARE @CurrentID int  DECLARE @Level int  SELECT TOP(1) @CurrentID = BrandID FROM Brands ORDER BY BrandID  DECLARE @StackTable TABLE (ID int, Level int)  DECLARE @OutputTable TABLE (ID int, [Name] nvarchar(32), ParentID int, Level int)  INSERT INTO @StackTable VALUES(@CurrentID, 1)  SET @Level = 1  WHILE @Level > 0  BEGIN  IF EXISTS (SELECT \* FROM @StackTable WHERE Level = @Level)  BEGIN  SELECT @CurrentID = ID FROM @StackTable WHERE Level = @Level  INSERT INTO @OutputTable  SELECT BrandID, [Name], ParentID, @Level AS Level FROM Brands  WHERE BrandID = @CurrentID  DELETE FROM @StackTable WHERE Level = @Level AND ID = @CurrentID    INSERT @StackTable  SELECT BrandID, @Level + 1 FROM Brands WHERE ParentID = @CurrentID  IF @@ROWCOUNT > 0  SET @Level = @Level + 1  END  ELSE  SET @Level = @Level - 1  END  SELECT \* FROM @OutputTable ORDER BY ID |

Как видит читатель, код без использования виртуальных представлений более громоздкий и сложный для понимания. Несмотря на то, что подобный код несложно написать, весьма просто допустить досадную ошибку, на поиски которой может уйти много времени.

Кроме того, рекурсия с использованием виртуального представления значительно превосходит в производительности подход с использованием временных таблиц. Так, для тестового набора из ста тысяч записей в таблице Brands, время выполнения кода с использованием виртуального представления оказалось в четыре раза меньше, чем время выполнения кода без использования оного. Примерно то же отношение получено и для меньших наборов данных.

**Операторы EXCEPT и INTERSECT**

Операторы EXCEPT и INTERSECT позволяют осуществить выборку данных, общих или различных для нескольких наборов данных. Синтаксис новых операторов абсолютно аналогичен оператору UNION.

Допустим, необходимо получить BrandID производителей, модели которых не присутствуют в таблице Products. Тогда, применив оператор EXCEPT следующим образом:

|  |
| --- |
| SELECT B.BrandID FROM Brands B  EXCEPT  SELECT P.BrandID FROM Products P |

можно достичь того же результата, что и при использовании оператора EXITS в комбинации с оператором отрицания NOT:

|  |
| --- |
| SELECT B.BrandID FROM Brands B  WHERE NOT EXISTS (SELECT P.BrandID FROM Products P WHERE P.BrandID = B.BrandID) |

Аналогично, для того, чтобы получить BrandID производителей, чьи модели присутствуют в таблице Products. Можно использовать оператор INTERSECT:

|  |
| --- |
| SELECT B.BrandID FROM Brands B  INTERSECT  SELECT P.BrandID FROM Products P |

а можно и оператор EXIST без отрицания:

|  |
| --- |
| SELECT B.BrandID FROM Brands B  WHERE EXISTS (SELECT P.BrandID FROM Products P WHERE P.BrandID = B.BrandID) |

либо же совсем привычный синтаксис INNER JOIN:

|  |
| --- |
| SELECT DISTINCT B.BrandID FROM Brands B  INNER JOIN Products P ON B.BrandID = P.BrandID |

Очевидно, что новые операторы позволяют уменьшить количество кода, необходимого для реализации выборки, причем выигрыш в объеме кода растет пропорционально числу полей в выборке. Также, не менее важным преимуществом является лучшая читаемость кода.

Производительность запроса при использовании новых операторов, практически не отличается от производительности запросов с EXISTS и JOIN. Число чтений (Reads) и время выполнения (Duration) мало отличаются в обоих случаях.

Оператор APPLY

Иногда, при написании сложных процедур для работы с данными, удобно использовать функции, возвращающие не скалярное значение, а таблицу. В предыдущих версиях SQL Server не было возможности в конструкциях JOIN использовать в качестве аргументов вызываемой функции параметры из внешнего запроса. Например, создадим процедуру GetProductDetails:

|  |
| --- |
| CREATE FUNCTION GetProductDetails (@ProductID int) RETURNS TABLE  AS  RETURN SELECT P.Model, P.Configuration, P.Price FROM Products P  WHERE P.ProductID = @ProductID |

Следующий код, использующий GetProductDetails приведет к ошибке:

|  |
| --- |
| -- Внимание! Этот код не работает  SELECT O.[Date], P.Model, P.Configuration, P.Price FROM Orders O  OUTER JOIN GetProductDetails(O.ProductID) AS P ON P.ProductID = O.ProductID |

В SQL Server 2005 для подобного использования функций предназначен оператор APPLY. Используя его вместо JOIN можно достичь желаемого результата:

|  |
| --- |
| SELECT O.[Date], P.Model, P.Configuration, P.Price FROM Orders O  OUTER APPLY GetProductDetails(O.ProductID) AS P |

Оператор APPLY используется в комбинации с двумя ключевыми словами: CROSS и OUTER. Функциональность CROSS APPLY аналогична INNER JOIN – в случае, если процедура не возвращает результат, то строка не попадает в результирующий набор данных, OUTER APPLY работает аналогично OUTER JOIN – если процедура не возвращает результат, то строка все-таки попадает в результирующий набор, а в колонках, соответствующих получаемым из процедуры данным будут находиться значения NULL.

Исходя из плана выполнения запроса, функция GetProductDetails выполняется для каждого параметра, получаемого из внешнего запроса, что приводит к драматическим последствиям с точки зрения производительности. Таким образом, использование APPLY для значительных объемов данных может являться далеко не самым лучшим решением. Поэтому, необходимо тщательно проверять производительность запросов, использующих APPLY на реальных объемах данных и, в соответствии с этим, делать вывод о разумности применения этого оператора при построении запроса.

Инструкция TABLESAMPLE

Иногда бывает необходимо получить некоторую выборку записей из базы данных, отражающую характер данных, содержащихся в базе. Для осуществления подобной выборки в SQL Server 2005 добавлена инструкция TABLESAMPLE, которая в качества параметра принимает количество строк или относительное количество строк в процентах от общего числа в таблице. Используется инструкция следующим образом:

|  |
| --- |
| SELECT СпискоПолей FROM ИмяТаблицы TABLESAMPLE(КоличествоПроцентов PERCENT) |

либо, если нужно выбрать определенное количество строк

|  |
| --- |
| SELECT СпискоПолей FROM ИмяТаблицы TABLESAMPLE(ЧислоСтрок ROWS) |

Однако стоит отметить, что будет возвращено не заданное количество строк (ROWS) или процентов (PERCENT), а лишь приблизительно соответствующее заданному количество.

Если необходимо получать выборку постоянного содержания в течение нескольких повторяющихся запросов, то нужно указать после инструкции TABLESAMLE дополнительно инструкцию REPEATTABLE:

|  |
| --- |
| SELECT СпискоПолей FROM ИмяТаблицы TABLESAMPLE(ЧислоСтрок ROWS) REPEATTABLE(ЧислоПовторений) |

Функция OUTPUT

Новая функция OUTPUT служит для повторного использования данных запроса. С помощью этой функции можно получить измененные в текущем запросе данные и использовать для вставки в другую таблицу, либо вернуть эти данные в вызывающий код.

Представим, что в магазине, использующем демонстрационную базу данных, произошло радостное событие, и вся партия поступивших в продажу ноутбуков была закуплена крупным заказчиком в момент поступления. В этом случае необходимо данные, вставляемые в таблицу Products, поместить также и в таблицу Orders. В SQL Server 2005 это можно сделать в одном запросе к базе данных:

|  |
| --- |
| INSERT INTO Products(BrandID, Model, Configuration, Price, Quantity)  OUTPUT GETDATE(), inserted.ProductID, inserted.Quantity  INTO Orders([Date], ProductID, Quantity)  VALUES (@BrandID, @Model, @Configuration, @Price, @Quantity) |

Читатель легко увидит из примера, что для доступа к изменяемым данным (вставляемым в таблицу в данном примере) используется идентификатор (имя псевдо таблицы), указывающий на характер операции, проводимой с данными. Допустимо использование следующих идентификаторов:

inserted, для команды INSERT

deleted, для команды DELETE

В случае использования функции OUTPUT в запросе UPDATE, измененные данные будут доступны в псевдо таблице inserted, а данные, которые подверглись изменению в псевдо таблице deleted.

Функция OUTPUT не может быть использована в запросе INSERT, в котором вставка проводится в представление данных (View), а также для вставки измененных данных в представление или табличную функцию.

Также стоит помнить о том, что OUTPUT не гарантирует, что элементы будут вставляться в таблицу в том же порядке, в котором происходит применение изменений. При этом, если в процессе выполнения запроса UPDATE изменяются какие-либо переменные или параметры, то OUTPUT возвращает не модифицированные значения параметров или переменных, то есть такие значения, которые переменные или параметры имели до выполнения запроса.

Применения функции OUTPUT

Выборка вставленных данных

Иногда бывает удобно получить в качестве результата выполнения процедуры, вставляющей данные в таблицу, результирующую строку, особенно когда эта строка содержит колонку с уникальным значением. Используя OUTPUT это можно следующим образом:

|  |
| --- |
| DECLARE @TempBrands TABLE (BrandID int, [Name] nvarchar(32))  INSERT INTO Brands([Name]) OUTPUT inserted.\* INTO @TempBrands  VALUES(@Name)  SELECT \* FROM @TempBrands |

Отметим, что этот пример показывает работу с одной строкой, поскольку при втавке большого количества строк, поряок следования может быть нарушен (как уже было написано выше, OUTPUT не гарантирует порядок строк) и использовать значения BrandID в вызывающем коде без дополнителных проверок будет проблематично.

Конечно, нет никакой проблемы в том, чтобы получить в результате запроса BrandID, не используя OUTPUT, поскольку обычно все данные уже имеются в вызывающем процедуру коде (они же и передаются в качестве аргументов самой процедуре), за исключением элемента с уникальным значением.

|  |
| --- |
| INSERT INTO Brands([Name]) VALUES(@Name)  SELECT @@IDENTITY |

Реализация функциональности очереди

Функция OUTPUT также позволяет удобно реализовать функциональность очереди, «извлекая» из таблицы запись, при этом удалять ее. Например, если потребуется функциональность очередей на выполнение заказа, то необходимо будет создать таблицу-очередь, например

|  |
| --- |
| CREATE TABLE [Queue](  [QueueID] [int] IDENTITY(1,1) NOT NULL,  [OrderID] [int] NOT NULL  ) |

и с помощью нее реализовать необходимую функциональность, используя функцию OUTPUT:

|  |
| --- |
| DECLARE @Queue TABLE (QueueID int, OrderID int)  DELETE TOP 1 FROM [Queue] ORDER BY QueueID  OUTPUT deleted.QueueID, deleted.OrderID INTO @Queue  SELECT \* FROM @Queue |

Без использования функции OUTPUT, код получается несколько более громоздким:

|  |
| --- |
| DECLARE @Queue TABLE (QueueID int, OrderID int)  INSERT INTO @Queue(QueueID, OrderID) SELECT TOP 1 [Queue].QueueID, [Queue].OrderID FROM [Queue]  DELETE [Queue] FROM [Queue] AS Q1  INNER JOIN @Queue AS Q2 ON Q1.QueueID = Q2.QueueID  SELECT \* FROM @Queue |

Функции PIVOT и UNPIVOT

Магазин ноутбуков с успехом использует демонстрационную базу данных в течение многих лет, и накопил огромную статистику по продажам ноутбуков. Естественно желание знать, для сравнения, объемы продаж за разные годы и общую сумму прибыли. Для того, чтобы из таблиц Orders и Products получить интересующую владельцев магазина информацию лучшим способом является использование ключевого функции PIVOT, позволяющей как бы «развернуть» данные в таблице.

|  |
| --- |
| SELECT Model, [2005], [2004] FROM (  SELECT P.Model, DATEPART(year, O.[Date]) AS [Year], O.Quantity FROM Orders O  INNER JOIN Products P ON P.ProductID = O.ProductID  ) AS C  PIVOT (SUM(Quantity) FOR [Year] IN ([2005], [2004])) AS PVT |

С использованием виртуального представления код можно написать несколько иначе:

|  |
| --- |
| WITH C(Model, [Year], Quantity) AS (  SELECT P.Model, DATEPART(year, O.[Date]) AS [Year], O.Quantity FROM Orders O  INNER JOIN Products P ON P.ProductID = O.ProductID  )  SELECT Model, [2005], [2004] FROM C  PIVOT (SUM(Quantity) FOR [Year] IN ([2005], [2004])) AS PVT |

Результатом выполнения данного кода в демонстрационной базе данных будет таблицу с тремя колонками: Model, 2005 и 2004. Например:

|  |
| --- |
| Model 2005 2004  -----------------------------------  A75-S206 10 24  M40-110 17 38  S215SR 2 10  T2XRP 35 12  V6800V 12 4 |

В предыдущих версиях SQL Server, где не была реализована функция PIVOT и CTE, чтобы достичь требуемого результата, пришлось бы писать код вроде приведенного ниже.

|  |
| --- |
| SELECT C.Model,  SUM(CASE C.[Year] WHEN 2005 THEN C.Quantity ELSE 0 END) AS [2005],  SUM(CASE C.[Year] WHEN 2004 THEN C.Quantity ELSE 0 END) AS [2004]  FROM (  SELECT P.Model, DATEPART(year, O.[Date]) AS [Year], O.Quantity FROM Orders O  INNER JOIN Products P ON P.ProductID = O.ProductID  ) AS C  GROUP BY C.Model |

Читатель, даже неопытный в программировании на T-SQL, наверняка отметит сложность работы с кодом, написанный без использования функции PIVOT, в случае более сложных запросов. В то же время, PIVOT является лишь синтаксической оболочкой для приведенной выше конструкции. Посмотрев планы исполнения примера, приведенного выше и примера, с использованием функции PIVOT, можно убедиться в их идентичности.

Функция UNPIVOT выполняет процедуру обратную PIVOT, «разворачивая» обратно таблицу данных, подвергшихся «обработке» функцией PIVOT , в исходное состояние. Положим, что для ведения статистики имеется следующая таблица

|  |
| --- |
| CREATE TABLE [Statistics](  [Model] [nvarchar](32) NOT NULL,  [2005] [int] NOT NULL,  [2004] [int] NOT NULL  ) |

содержащая данные, полученные в ходе выполнения предыдущего запроса – примера использования функции PIVOT.

|  |
| --- |
| SELECT \* FROM [Statistics]  UNPIVOT(TotalQuantity FOR [Year] IN ([2005], [2004])) AS PVT |

Результатом выполнения будет таблица трех из столбцов: Model, TotalQuantity, Year.

|  |
| --- |
| Model TotalQuantity Year  ------------------------------------------  A75-S206 10 2004  A75-S206 24 2005  M40-110 17 2004  M40-110 38 2005 |

Обновленный оператор TOP

Оператор TOP широко используется для ограничения числа строк, возвращаемых командой SELECT. В предыдущих версиях SQL Server, оператор TOP принимал в качестве параметра только константу. В SQL Server 2005 параметром этого оператора может быть переменная, выражение или вложенный вопрос.

Например, следующим образом можно осуществить выборку такого числа моделей некоторого производителя, которое соответствует среднему количеству моделей каждого производителя в таблице Products:

|  |
| --- |
| SELECT TOP (SELECT AVG(AvgNum) \* FROM  (SELECT COUNT(\*) AS AvgNum FROM Products GROUP BY BrandID) AS NumTable) P.Model FROM Products P |

Для использования в качестве параметра оператора TOP переменной, выражения или вложенного запроса необходимо заключать ее в круглые скобки:

|  |
| --- |
| SELECT TOP (@Num) \* FROM ИмяТаблицы |

При использовании константы, скобки не обязательны в команде SELECT, но обязательны при использовании с командами, изменяющими данные, например:

|  |
| --- |
| DELETE TOP(10) FROM Orders ORDER BY Date DESC |

В предыдущих версиях SQL Server для использования переменной в качестве параметра оператора TOP приходилось прибегать к сложным конструкциям или динамическому созданию запроса.

|  |
| --- |
| DECLARE @DynamicQuery varchar(100)  SET @DynamicQuery = 'SELECT TOP ' + CAST (@N AS varchar) + ' \* FROM Products'  EXECUTE(@DynamicQuery) |

Разбиение данных на страницы с использованием оператора TOP

Наиболее простым способом использования оператора TOP для разбиения на страницы является использование смещения по колонке ID. Процедура принимает параметр – количество записей на странице и «последний» ID, полученный в предыдущем запросе. При этом, благодаря новым возможностям оператора TOP, можно варьировать количество записей на странице, используя такую простую процедуру.

|  |
| --- |
| SELECT TOP (@N) \* FROM ИмяТаблицы WHERE КолонкаID > @ID |

Более общий подход, не требующий сохранения «последнего» ID предыдущего запроса и формально не зависящий от значения ID выглядит так:

|  |
| --- |
| SELECT \* FROM (SELECT TOP (@N \* (@PageNum + 1)) \* FROM ИмяТаблицы  ORDER BY КолонкаID) AS PTable  WHERE КолонкаID NOT IN (SELECT TOP (@N \* @PageNum) КолонкаID FROM ИмяТаблицы  ORDER BY КолонкаID) |

Следует обратить внимание читателя, что запрос будет выполнять свою задачу лишь при условии того, что колонка КолонкаID содержит уникальные значения. Таким образом, для обеспечения разбиения данных с помощью такой процедуры нужно иметь в таблице колонку IDENTITY.

Для того, чтобы подобная процедура работала в более ранних версиях SQL Server, необходимо использовать динамическое создание запроса:

|  |
| --- |
| DECLARE @Query nvarchar(200)  SET @Query = 'SELECT \* FROM (SELECT TOP ' + CAST(@N \* (@PageNum + 1) AS nvarchar) +  ' \* FROM ИмяТаблицы ORDER BY КолонкаID) AS P  WHERE КолонкаID NOT IN (SELECT TOP ' + CAST(@N \* @PageNum AS nvarchar) +  'КолонкаID FROM ИмяТаблицы ORDER BY КолонкаID)'  EXECUTE(@Query) |

Обработка ошибок в SQL

В связи с интеграцией SQL Server с платформой .NET, языки которой поддерживают гибкий механизм обработки исключений, разработчики SQL Server включили в T-SQL давно желанную SQL-программистами возможность обработки исключений. Текущая реализация в SQL Server 2005 позволяет обрабатывать некритические ошибки с помощью похожего на ставший уже стандартным синтаксис TRY … CATCH.

|  |
| --- |
| BEGIN TRY  -- «Опасный» запрос  END TRY  BEGIN CATCH  -- Обработка ошибки  END CATCH |

Механизм обработки ошибок в T-SQL, конечно, не такой гибкий, как в .NET языках, но, тем не менее, позволяет сделать достаточно много, анализируя код ошибки, возвращаемый функцией @@ERROR. Например, если таблица Products задана так, что не позволяет хранить отрицательное значение в колонке Quantity:

|  |
| --- |
| CREATE TABLE [Products](  [ProductID] [int] IDENTITY(1,1) NOT NULL,  [BrandID] [int] NOT NULL,  [Model] [nvarchar](32) NOT NULL,  [Configuration] [nvarchar](128) NOT NULL,  [Price] [money] NOT NULL,  [Quantity] [int] NOT NULL, CHECK ([Quantity] >= 0)  ) |

то можно использовать следующий код для изменения количества имеющихся на складе ноутбуков и создания заказа:

|  |
| --- |
| BEGIN TRY  BEGIN TRAN -– Создание транзакции  INSERT INTO Orders([Date], ProductID, Quantity, [Year] ) VALUES(GETDATE(), @ProductID, @Quantity, @Year)  SET @OrderID = @@IDENTITY  UPDATE Products2 SET Quantity = Quantity - @Quantity WHERE ProductID = @ProductID  UPDATE Orders SET Quantity = @Quantity WHERE OrderID = @OrderID  COMMIT –- Если нет ошибок, то подтверждаем транзакцию  END TRY  BEGIN CATCH  DECLARE @Err int  SET @Err = @@ERROR  ROLLBACK –- Откат транзакции  IF @ERR = 547  BEGIN  SELECT 'Недостаточно ноутов на складе' AS Error  END  ELSE  BEGIN  SELECT 'Неизвестная ошибка' AS Error, @Err AS ErrorNumber  END  END CATCH |

Заключение

Пусть новые возможности Transact-SQL в SQL Server 2005 не вносят ничего принципиально нового в устройство SQL Server, но позволяют значительно быстрее создавать более понятный и легко читаемый код, упрощая, таким образом, жизнь разработчика.

В заключение, автор настоятельно рекомендует читателю ознакомиться со статьями из списка литературы, поскольку в них описаны кардинальные изменения основной концепции SQL Server.

**Список литературы**

Иван Бодягин, Новые возможности MS SQL Server 2004 "Yukon", RSDN Magazine #6-2003

Антон Злагостев, MS SQL Server 9 “Yukon”. Интеграция с .NET, RSDN Magazine #6-2003

Иван Бодягин, Версионность в “Yukon”, RSDN Magazine #6-2003

Иван Бодягин, MS SQL 2005: оконные функции, RSDN Magazine #6-2004

Алексей Ширшов, Использование XML совместно с SQL, RSDN Magazine #2-2004