Лекція №9. Особливості застосування операторів мови SQL (тригери, збережені процедури)

**Тригери об'єктів баз даних**

Забезпечення цілісності даних має вирішальне значення для підтримки точності та узгодженості бази даних. Тригери являють собою особливий вид збережених процедур, прив'язаних до таблиць і поданнями і автоматично які спрацьовують при додавання, видалення або змінення даних. Перш ніж реалізовувати тригер, слід з ’ ясувати, чи не можна отримати аналогічні результати з використанням обмежень. Тригери застосовуються в наступних випадках:

1. якщо потрібно реалізувати складні обмеження на дані;
2. якщо необхідно каскадне зміна через пов'язані таблиці в базі даних;
3. якщо база даних денормализована і потрібно спосіб автоматизованого оновлення надлишкових (дублюючих) даних в декількох таблицях;
4. якщо потрібно висновок користувацьких повідомлень, журналювання проведених користувачем операцій з даними або складна обробка помилок.

Для створення тригера використовується конструкція наступного виду:

CREATE TRIGGER {ім'я тригера}  
ON {ім'я таблиці}  
FOR {тип події для обробки}  
AS  
{текст тригера}

При створенні тригера слід задати тип події, що викликає його спрацьовування. Типів події три: INSERT, UPDATE, DELETE. Один і той же тригер може спрацьовувати на один, два або три події. Конструкція AS і наступні за нею команди мови SQL визначають завдання, яке буде виконувати тригер.

При спрацьовуванні тригера на події INSERT, UPDATE або DELETE створюється одна або дві псевдотаблицы (логічні таблиці). Існує два типи логічних таблиць: INSERTED і DELETED. INSERTED створюється в результаті події додавання або оновлення даних. У ній знаходиться набір доданих або змінених записів. DELETED створюється в результаті видалення або відновлення даних.

У разі видалення даних в ній знаходяться видалені записи, а у разі оновлення даних в ній знаходяться оновлені запису в тому стані, в якому вони були до операції оновлення. Для визначення того, значення якого поля записів було змінено при оновленні даних, використовується конструкція IF UPDATE({ім'я поля}). UPDATE({ім'я поля}) повертає TRUE, якщо поле було змінено, і FALSE інакше. Необхідно звернути увагу на той факт, що при виконанні тригера таблиця, для якої був створений тригер, вже містить додані і змінені записи і не містить видалені записи.

Для скасування додавання, видалення або зміни даних в триггері необхідно викликати оператор відкату транзакції ROLLBACK TRAN.

*Тригер -* це процедура, що зберігається спеціального типу, автоматично виконується при настанні заданого події. Тригери не містять явних вхідних або вихідних параметрів і не можуть бути запущені явно (по команді EXEC). SQL Server підтримує три класи тригерів: тригери мови обробки даних (тригери DML), тригери мови опису даних (тригери DDL) і тригери входу в систему.

Тригери DML запускаються при спробі змінити дані в таблиці або поданні за допомогою операторів INSERT, UPDATE або DELETE. Формат команди, що створює тригер, наступний:

CREATE TRIGGER [schema\_name.] Trigger\_name

ON {table | view]

[WITH <option> [... n]]

{FOR | AFTER | INSTEAD OF]

{[INSERT] [,] [UPDATE] [,] [DELETE]}

[NOT FOR REPLICATION]

AS {sql\_statement [; ] [, ... n]}

<Option> :: = {[ENCRYPTION]! [EXECUTE AS Clause]}

Тип тригера AFTER означає, що він спрацьовує тільки після успішного виконання оператора, що запускає тригер. При цьому спочатку будуть виконані всі перевірки обмежень, і якщо вони пройдуть успішно, спрацює тригер. Наприклад, якщо додавання запису в таблицю неможливо через повторюваного значення первинного ключа, то пов'язаний з цією операцією тригер типу AFTER не запуститься.

Якщо вказано тільки ключове слово FOR, це вказує на створення тригера AFTER, тобто FOR і AFTER в даному випадку - синоніми. При цьому треба враховувати, що оператори INSERT, UPDATE, DELETE виконуються в рамках неявній транзакції. Тригер запускається в рамках цієї ж транзакції, і з нього транзакцію можна цілком відкотити. Тригери AFTER не можуть бути визначені для представлень/уявлень.

Тригер типу instead of означає, що дії оператора, який викликав тригер, не виконуються, а замість цього виповнюється код тригера. У таблиці або поданні для кожного з операторів INSERT, UPDATE або DELETE може бути визначено не більше одного тригера типу INSTEAD OF. Однак для таблиці можна створити кілька подань, у кожного з яких може бути власний тригер INSTEAD OF. Використання тригерів INSTEAD OF не допускається в підтримують оновлення уявленнях, які використовують параметр WITH CHECK OPTION.

Перерахування "[INSERT] [,] [UPDATE] [,] [DELETE]" вказує оператор або оператори, виконання яких призведе до запуску створюваного тригера.

Опції тригера ENCRYPTION і EXECUTE AS мають таке ж значення, що і для збережених процедур (див. Параграф 10.3). Параметр NOT FOR REPLICATION керує поведінкою тригера при внесенні змін механізмом реплікації. За замовчуванням дії по модифікації даних (UPDATE, DELETE, INSERT), що виконуються механізмом реплікації, викликають запуск відповідного тригера. Однак якщо тригер створений із зазначенням цього параметра, в описаних випадках тригер не запускається буде.

Тригери DML в ході роботи мають доступ до спеціальних таблицях inserted і deleted, що зберігають результати змін, вироблених оператором, що викликав тригер. У табл. 1 наведені дані про те, що саме в цих таблицях зберігається, в залежності від оператора, для якого визначено тригер.

*Таблиця 1* **Вміст спеціальних таблиць, доступних триггерам DML**

|  |  |  |
| --- | --- | --- |
| оператор | Таблиця | вміст таблиці |
| INSERT | inserted | Вставлені в таблицю нові рядки |
| DELETE | deleted | віддалені рядки |
| UPDATE | inserted | Новий стан змінених рядків |
| deleted | Старе стан змінених рядків |

За допомогою тригерів можна реалізувати перевірку складних логічних умов, виконання яких забезпечує несуперечність даних. Наприклад, потрібно при внесенні якийсь записи в таблицю перевірити наявність пов'язаних даних в іншій таблиці, що знаходиться в іншій БД. Обмеження зовнішнього ключа тут не допоможе, тому що БД різні, а за допомогою тригера таку перевірку виконати можна.

Розглянемо наступний приклад. В БД MyTest є таблиця Bookl, вже використовувалася в прикладах. В БД MyTest2 є таблиця BlackList, створена за допомогою наведеного нижче скрипта:

USE [MyTest2]

GO

CREATE TABLE [dbo]. [BlackList] (

[Id] [int] primary key,

[Author] [varchar] (50) NOT NULL,

[Title] [varchar] (50) NOT NULL)

Нехай таблиця Bookl містить перелік використовуваних навчальних видань, а таблиця BlackList - підручники, до якості яких є претензії. Створимо тригер, перевіряючий, що додається в Bookl книжка не знаходиться в BlackList. Тип тригера - AFTER, викликається він операторами INSERT І UPDATE:

use MyTest

go

CREATE TRIGGER CheckBlackList

ON dbo.Bookl

AFTER INSERT, UPDATE

AS BEGIN

IF EXISTS

(SELECT \* FROM [MyTest2]. [Dbo]. [BlackList] b

INNER JOIN inserted i ON

* ((B. [Author] = i. [Author]) AND
* (B. [Title] = i. [Title])))

BEGIN

RAISERROR ( 'Одна з книг в таблиці BlackList, додавання неможливо', 10,1);

ROLLBACK TRANSACTION END END

Якщо хоча б одна з додаються книг виявилася в таблиці BlackList, тригером ініціюється помилка і проводиться відкат транзакції. Помилка генерується системної функцією RAISERROR, перший аргумент якої - повідомлення про помилку, другий - рівень серйозності помилки *(англ.* Severity), третій - стан *(англ.* State). Чи не розбираючи детально можливості функції, відзначимо, що задавати помилки з рівнем серйозності від 0 до 18 може будь-який користувач, від 19 до 25 - тільки адміністратор.

Нехай таблиця BlackList містить такі записи (табл. 2), а в таблиці Bookl вже є записи зі значенням поля Id (первинний ключ), що дорівнює 1, 2, 3 і 4.

*Таблиця 2* **Вміст таблиці Blacklist**

|  |  |  |
| --- | --- | --- |
| Id | Author | Title |
| 1 | Автор1 | поганий учебнік1 |
| 2 | Автор2 | поганий учебнік2 |

Розглянемо три команди поновлення даних і одержуваний результат:

INSERT INTO dbo.Bookl (Id, Author, Title, Publisher, [Year])

VALUES ( ',' ABTopl ',' Поганий учебнік1 ',' Іздательство1 ',

2012);

Запис додана не буде, але не через тригера, а через збіг первинних ключів. Обмеження первинного ключа перевіряється до запуску тригера AFTER. Якщо воно не виконується, то і тригер запущений не буде.

INSERT INTO dbo.Bookl (Id, Author, Title, Publisher, [Year])

VALUES ( ',' ABTopl ',' Поганий учебнік1 ',' Іздательство1 ', 2012);

Спроба додавання запису буде неуспішною: тригер після перевірки 5'словія ініціює помилку і виконає відкат транзакції.

INSERT INTO dbo.Bookl (Id, Author, Title, Publisher, [Year])

VALUES ( ',' ABTopl ',' Поганий учебнік1 ',' Іздательство1 ', 2012), (6,' Автор1 ',' Хороший підручник ',' Іздательство1 ', 2013);

Тут одним оператором INSERT в таблицю Bookl додаються два записи, причому одна книга знаходиться в "чорному списку", а друга - ні. Але тригер виконує відкат всій транзакції, тому жодна з книг додана не буде. У зв'язку з останнім прикладом хочеться відзначити, що рекомендується писати тригери таким чином, щоб вони коректно працювали в будь-якій ситуації: і якщо була змінена тільки один рядок, і якщо змінені були кілька рядків.

Перейдемо тепер до триггерам DDL. Вони створюються для окремої БД або цілком примірника SQL Server і запускаються по подіям, пов'язаним зі зміною об'єктів БД, наприклад при спробі видалення таблиці: CREATE TRIGGER trigger\_name

ON (ALL SERVER | DATABASE)

[WITH <option> [... n]]

(FOR I AFTER] (event\_type | event\_group) [... n]

AS (sql\_statement [;] [, ... n] |)

Ключове слово DATABASE вказує на те, що тригер створюється для поточної БД, ALL SERVER цілком для сервера. Задаються в секції WITH опції аналогічні тим, що були для тригерів DML. Типи подій повністю описані в документації SQL Server, деякі часто використовувані наведені в табл. 3. Замість типу події може бути вказана група подій, і тригер буде викликатися по кожній події з цієї групи. Наприклад, група DDL GDR DATABASE EVENTS включає події, пов'язані з управлінням дозволами на рівні БД:

* • DENY DATABASE (заборона дії);
* • GRANT DATABASE (надання права на виконання дії);
* • REVOKE DATABASE (відгук права на виконання дії).

*Таблиця 3* **Приклади подій для тригерів** DDL

|  |  |  |
| --- | --- | --- |
| оператор DDL | Тип події | опис |
| CREATE DATABASE | CREATE\_DATABASE | створення БД |
| DROP LOGIN | DROP\_LOGIN | Щоб видалити обліковий запис |
| UPDATE STATISTICS | UPDATE\_STATISTICS | оновлення статистики |
| DROP TRIGGER | DROP\_TRIGGER | видалення тригера |
| ALTER TABLE | ALTER\_TABLE | зміна таблиці |

Розглянемо тепер приклад тригера DDL. Він запобігає випадкове видалення таблиці - при спробі виконати подібну операцію виводиться повідомлення, і транзакція відкочується:

Use MyTest2 go

CREATE TRIGGER NoDropTab

ON DATABASE

FOR DROP\_TABLE

AS

Print N'Ви намагаєтеся видалити потрібну таблицю! ';

Print N'Еслі ви впевнені в своїх діях, спочатку відключіть цей тригер. ';

ROLLBACK;

Тепер видалити таблицю просто так не вдасться - при спробі виконати наведений нижче код, спрацює тригер і скасує дію;

Use MyTest2

go

DROP TABLE dbo.BlackList; - видалення не виконати

Тимчасово відключити тригер можна командою DISABLE TRIGGER, де вказуються ім'я тригера і об'єкт, для якого він визначений. Знову включити тригер можна по команді ENABLE TRIGGER. Наприклад, якщо таблицю в базі MyTest2 все ж треба видалити, це можна зробити наступним чином:

Use MyTest2

go

DISABLE TRIGGER NoDropTab ON DATABASE;

DROP TABLE dbo.BlackList;

ENABLE TRIGGER NoDropTab ON DATABASE;

Перейдемо до розгляду третьої категорії тригерів - тригерів входу ( *англ.* Logon trigger). Такий тригер запускається після успішної автентифікації, але до того, як встановлюється сеанс роботи користувача. Формат оператора створення тригера наведено нижче, опції в секції WITH - такі ж, як в попередньому випадку:

CREATE TRIGGER trigger\_name

ON ALL SERVER

[WITH <option> [... n]]

{FORI AFTER} LOGON

AS {sql\_statement [; ] [, ... n]}

Розглянемо наступний приклад, який ілюструє застосування тригера входу. Спочатку створюється обліковий запис login\_test і їй надаються права на перегляд інформації про стан сервера:

USE master;

GO

CREATE LOGIN login\_test WITH PASSWORD = '3KHJ6dhx (OxVYsdf' MUST\_CHANGE,

CHECK\_EXPIRATION = ON;

GO

GRANT VIEW SERVER STATE TO login\_test;

Далі на сервері створюється тригер входу connection\_limit\_ trigger, що виконується в контексті безпеки записи login\_test. Тригер забороняє спробу підключення до SQL Server з обліковим записом login\_test, якщо від її імені вже встановлено три сеанси:

CREATE TRIGGER connection\_limit\_trigger

ON ALL SERVER WITH EXECUTE AS 'login\_test'

FOR LOGON

AS

BEGIN

IF ORIGINAL\_LOGIN () = 'login\_test' AND

(SELECT COUNT (\*) FROM sys.dm\_exec\_sessions

WHERE is\_user\_process = 1 AND

original\_login\_name = 'login\_test')> 3

ROLLBACK-

END;

Що використовується в коді тригера системне уявлення sys. dm\_exec\_ sessions повертає по одному рядку на кожен минулий аутентифікацію сеанс, підключений до SQL Server. Його стовпець original\_ login\_name містить ім'я облікового запису, а стовпець is\_user\_process містить 0, якщо сеанс є системним, і 1 - в іншому випадку. Функція ORIGINAL\_LOGIN () дозволяє отримати ім'я облікового запису, яка встановлює сеанс, що запустив тригер.

Видалення тригера виконується оператором DROP TRIGGER. Його формат буде дещо відрізнятися в залежності від типу тригера:

- для тригера DML

DROP TRIGGER [schema\_name.] Trigger\_name [... n] [;

]

- для тригера DDL

DROP TRIGGER trigger\_name [... n]

ON {DATABASE I ALL SERVER} [; ]

- для тригера входу

DROP TRIGGER trigger\_name [... n]

ON ALL SERVER [; ]

Як приклад виконаємо видалення створеного раніше в таблиці Bookl БД MyTest тригера CheckBlackList:

use MyTest

go

DROP TRIGGER dbo.CheckBlackList

Змінити існуючий тригер можна за допомогою оператора ALTER TRIGGER. Крім ключового слова ALTER, формат оператора такий же, як при створенні тригера.

**ЗБЕРЕЖЕНІ ПРОЦЕДУРИ**

*Процедура (англ.* Stored procedure) - це іменований програмний об'єкт БД. У SQL Server є процедури, що зберігаються декількох типів.

*Системні збережені процедури (англ.* System stored procedure) поставляються розробниками СУБД і використовуються для виконання дій з системним каталогом або отримання системної інформації. Їх назви зазвичай починаються з префікса "sp\_". Запускаються збережені процедури всіх типів за допомогою команди EXECUTE, яку можна скоротити до ЕХЕС. Наприклад, збережена процедура sp\_helplogins, запущена без параметрів, формує два звіти про імена облікових записів *(англ.* Logins) і відповідних їм в кожній БД користувачів *(англ.* Users).

EXEC sp\_helplogins;

Щоб дати уявлення про дії, які виконуються за допомогою системних збережених процедур, в табл. 4 наведені деякі приклади. Всього ж системних збережених процедур в SQL Server більше тисячі.

*Таблиця 4* **Приклади системних збережених процедур SQL Server**

|  |  |
| --- | --- |
| Назва | Дія |
| spchangedbowner | Дозволяє змінити власника БД |
| spdatabases | Повертає список БД і їх розмір |
| spholp | Приймає в якості вхідного параметра ім'я об'єкта і повертає про нього відомості. Наприклад: exec sp\_help @ objname = 'dbo.Bookl' |
| spstoredprocedures | Повертає список збережених процедур в поточній БД |
| sp\_tables | Повертає список таблиць і уявлень в поточній БД. які можна запитувати |

Користувачеві доступно створення збережених процедур в призначених для користувача БД і в БД для тимчасових об'єктів. В останньому випадку процедура, що зберігається буде *тимчасовою.* Так само як у випадку з тимчасовими таблицями, назва тимчасової процедури, що має починатися з префікса "#", якщо це локальна тимчасова процедура, що зберігається, або з "##" - якщо глобальна. Локальна тимчасова процедура може використовуватися тільки в рамках з'єднання, в якому її створили, глобальна - і в рамках інших з'єднань.

Програмовані об'єкти SQL Server можуть створюватися як з використанням засобів Transact-SQL, так і за допомогою збірок *(англ.* Assembly) в середовищі CRL (Common Language Runtime) платформи Microsoft.Net Framework. В даному підручнику буде розглядатися тільки перший спосіб.

Для створення збережених процедур використовується оператор CREATE PROCEDURE (можна скоротити до PROC), формат якого наведено нижче:

CREATE {PROC I PROCEDURE) [schema\_name.] Proc\_name [; number]

[{Gparameter [type\_schema\_name.] Data\_type}

[VARYING] [ "default] [OUT | OUTPUT] | [READONLY]

] [, ... n]

[WITH <procedure\_option> [... n]]

[FOR REPLICATION]

AS {[BEGIN] sql\_statement [;] [... n] [END]}

[;]

де

<Procedure\_option> :: =

[ENCRYPTION] [RECOMPILE] [EXECUTE AS Clause]

Якщо процедура (або тригер, функція, уявлення) створюється з опцією ENCRYPTION, її код перетворюється таким чином, що текст стає нечитабельним. У той же час, використовується алгоритм перенесений з ранніх версій SQL Server і не може розглядатися в якості надійного алгоритму захисту - існують утиліти, що дозволяють швидко виконати зворотне перетворення.

Опція RECOMPILE вказує на те, що при кожному виклику процедури система буде перекомпілювати текст. У звичайному випадку скомпільована при першому запуску процедура зберігається в кеші, що дозволяє збільшити швидкодію.

EXECUTE AS визначає контекст безпеки, в якому повинна бути виконана процедура. Далі вказується одне зі значень f CALLER | SELF | OWNER | 'User\_name'). CALLER є значенням за замовчуванням і означає, що код буде виконуватися в контексті безпеки користувача, що викликає цей модуль. Відповідно, користувач повинен мати дозволу не тільки на сам програмований об'єкт, але і на інші які поставлені їм об'єкти БД. EXECUTE AS SELF означає використання контексту користувача, що створює або змінює програмований об'єкт. OWNER вказує, що код буде виконуватися в контексті поточного власника процедури. Якщо для неї не визначений власник, то мається на увазі власник схеми, до якої вона відноситься. EXECUTE AS 'user\_name' дозволяє явно вказати ім'я користувача (в одинарних лапках).

Для процедури можуть зазначатися параметри. Це локальні змінні, використовувані для передачі значень в процедуру. Якщо параметр оголошений з ключовим словом OUTPUT (або скорочено OUT), він є вихідним: заданий йому в процедурі значення після її закінчення може бути використано викликала процедуру програмою. Ключове слово READONLY означає, що значення параметра не може бути змінено всередині збереженої процедури.

Параметрами можуть бути призначені значення але замовчуванням, які будуть використані, якщо при виклику процедури значення варіанту не буде вказано в явному вигляді. Розглянемо приклад:

CREATE PROC surma (@ а int, @b int = 0,

© result int OUTPUT) AS

SET @ result = 0a + 0b

Ми створили процедуру з трьома параметрами, причому у параметра @b значення за замовчуванням = 0, а параметр @result - вихідний: через нього повертається значення в викликала програму. Що Їх дії досить прості - вихідний параметр отримує значення суми двох вхідних.

При роботі в SQL Server Management Studio створену збережену процедуру можна знайти в розділі програмованих об'єктів БД *(англ.* Programmability) в підрозділі для збережених процедур.

При виклику процедури в якості вхідних параметрів можна використовувати як змінні, так і константи. Розглянемо два приклади. У першому вхідні параметри процедури явно задані константами, для вихідного параметра в виклику вказано ключове слово OUTPUT. У другому варіанті в якості першого вхідного параметра використовується значення змінної, а для другого параметра за допомогою ключового слова DEFAULT зазначено, що повинно бути використано значення за замовчуванням:

вар. 1

DECLARE @ з int;

EXEC summa 10,5, @ c OUTPUT;

PRINT 0c; - буде виведено 15

вар. 2

DECLARE Gi int = 5;

- при виклику використовуємо значення за замовчуванням

EXEC summa Gi, DEFAULT, 0С OUTPUT;

PRINT 0c; - буде виведено 5

Розглянемо тепер приклад з аналізом коду повернення, з яким закінчується процедура. Нехай треба підрахувати, скільки в таблиці Bookl книг, виданих в заданому діапазоні років. При цьому якщо початковий рік виявився більше кінцевого, процедура повертає "1" і підрахунок не проводить, інакше - вважаємо кількість книг і повертаємо 0:

CREATE PROC dbo.rownum (0FirsYear int, GLastYear int, 0result int OUTPUT) AS

IF 0FirsYear> 0LastYear RETURN 1

ELSE BEGIN

SET @ result = (SELECT COUNT (\*) FROM dbo.Bookl

WHERE [Year] BETWEEN 0FirsYear AND 0LastYear);

RETURN 0;

END

Розглянемо варіант виклику даної процедури, в якому код повернення зберігається в цілочисельний змінної 0ret, після чого аналізується його значення (в даному випадку це буде 1). Використовувана в операторі PRINT функція CAST служить для перетворення значення цілочисельний змінної Gres до строковому типу:

DECLARE 0ret int, Gres int

EXEC Gret = rownum 2004, 2002 Gres OUT;

IF 0ret = l PRINT 'Початковий рік більше кінцевого'

ELSE

PRINT 'Число книг' + CAST (Gres as varchar (20))

Збережені процедури можуть не тільки зчитувати дані з таблиці, але і змінювати дані і навіть створювати таблиці і ряд інших об'єктів БД.

Однак створювати схеми, функції, тригери, процедури та подання з збереженої процедури не можна.

Наступний приклад ілюструє як ці можливості, так і питання, пов'язані з областю видимості тимчасових об'єктів. Наведена нижче збережена процедура перевіряє наявність тимчасової таблиці # ТаЬ2; якщо цієї таблиці немає, то створює її. Після цього в таблицю # ТаЬ2 заносяться значення двох стовпців, і вміст таблиці виводиться оператором SELECT:

CREATE PROC My\_Procl (@id int, @name varchar (30))

AS

BEGIN

IF OBJECT\_ID ( 'tempdb.dbo. # Tab21) IS NULL

CREATE TABLE dbo. # Tab2 (id int, name varchar (30));

INSERT INTO dbo. # Tab2 (id, name) VALUES (0id, 0name)

SELECT \* FROM dbo. # Tab2 -№1

END

Перед першим викликом процедури, що створимо використовувану в ній тимчасову таблицю # ТаЬ2. Зверніть увагу на оператор ЕХЕС. У попередніх прикладах параметри передавалися в процедуру "по позиції", а в даному випадку використовується інший формат передачі параметрів - "по імені", явно вказується ім'я параметра і його значення:

CREATE TABLE dbo. # Tab2 (id int, name varchar (30));

EXEC My\_Procl 0name = 'lvan', 0id = 2;

SELECT \* FROM dbo. # Tab2; -№2

У наведеному прикладі оператор SELECT відпрацює двічі: перший раз - всередині процедури, другий раз - з викликає фрагмента коду (відзначений коментарем "№ 2").

Перед другим викликом процедури видалимо тимчасову таблицю # ТаЬ2. Тоді однойменна тимчасова таблиця буде створена з збереженої процедури:

DROP TABLE dbo. # Tab2;

EXEC My\_Procl 0name = 'Ivan', 0id = 2;

SELECT \* FROM dbo. # Tab2; -№2

У цьому випадку дані виведе тільки оператор SELECT, що знаходиться всередині процедури (з коментарем "Ха 1"). Виконання SELECT "№ 2" призведе до помилки, так як створена в збереженій процедурі тимчасова таблиця на момент повернення з процедури буде вже вилучена з бази tempdb.

Видалити збережену процедуру можна за допомогою оператора DROP PROCEDURE. Його формат представлений нижче. Одним оператором можна видалити кілька збережених процедур, перерахувавши їх через кому:

DROP (PROC I PROCEDURE) {[schema\_name.] Procedure} [

, ... n]

Наприклад, видалимо раніше створену процедуру summa:

DROP PROC summa;

Внести зміни в існуючу процедуру (а фактично - перевизначити її) можна за допомогою оператора ALTER PROCEDURE (допу

Стиму скорочення PROC). За винятком ключового слова ALTER, формат оператора практично збігається з форматом CREATE PROCEDURE. Наприклад, змінимо процедуру dbo. rownum, встановивши їй опцію виконання в контексті безпеки власника:

ALTER PROC dbo.rownum (SFirsYear int,

SLastYear int, Sresult int OUTPUT)

WITH EXECUTE AS Owner - встановлюється опція

AS

IF 0FirsYear> 0LastYear RETURN 1 ELSE BEGIN

SET 0result = (SELECT COUNT (\*) FROM dbo.Bookl

WHERE [Year] BETWEEN SFirsYear AND SLastYear);

RETURN 0;

END

У деяких випадках може виникнути необхідність у динамічному формуванні команди і виконанні її на сервері БД. Це завдання також може вирішуватися за допомогою оператора ЕХЕС. У наведеному нижче прикладі виконується вибірка записів з таблиці Bookl за умовою рівності атрибута Year значенням, що задається за допомогою змінної:

DECLARE 0у int = 2000;

EXEC ( 'SELECT \* FROM dbo.Bookl WHERE [Year] =' + @ y);

Виконання динамічно сформованих інструкцій створює передумови для реалізації комп'ютерних атак типу "SQL-ін'єкція" *(англ.* SQL injection). Суть атаки полягає в тому, що порушник впроваджує в динамічно формується запит власний код на SQL. Зазвичай це відбувається, коли підставляються параметри беруть з результатів введення даних користувачем.

Трохи змінимо попередній приклад:

DECLARE 0у varchar (100);

SET 0у = '2ООО'; - це ми отримали від користувача

EXEC ( 'SELECT \* FROM dbo.Book2 WHERE [Year] =' + 0y);

Якщо припустити, що привласнюється в операторі SET строкове значення ми отримали від користувача (неважливо яким чином, наприклад, через веб-додаток), то приклад ілюструє "штатний" поведінку нашого коду.

У наступному прикладі порушник додав до введеного параметру код на SQL. В результаті ми не тільки вивели потрібні рядки з таблиці, але і видалили всі її вміст:

DECLARE 0у varchar (100);

SET 0у = '2000; DELETE FROM dbo.Book2 '; - ін'єкція

EXEC ( 'SELECT \* FROM dbo.Book2 WHERE [Year] =' + 0y);

Рекомендується по можливості використовувати в подібних випадках системну збережену процедуру sp\_executcsql, яка дозволяє контролювати тип параметрів, що є одним з бар'єрів на шляху SQL- ін'єкцій. Не розглядаючи в подробицях її формат, розберемо приклад, аналогічний представленому раніше:

EXECUTE sp\_executesql

N'SELECT \* FROM dbo.Bookl WHERE [Year] = 0y ',

N '@ y int;

6 у = 2000;

Тут явно вказується тип використовуваного в запиті параметра, і SQL Server при виконанні буде його контролювати. Буква "N" перед лапками вказує, що це літерні константа в форматі Unicode, як того вимагає процедура. Параметру можна привласнити не тільки постійне значення, а й значення іншої змінної.