Практическая работа 5 Хранимые процедуры и определяемые пользователем функции

* Процедурные расширения
* Хранимые процедуры
* Определяемые пользователем функции

В этой работе мы познакомимся с пакетами и подпрограммами.

Пакет (batch) — это последовательность инструкций и процедурных расширений языка Transact-SQL.

Подпрограмма (routine) — это хранимая процедура или определяемая пользователем функция (ОПФ) (User Defined Function — UDF).

В начале мы рассмотрим все процедурные расширения, поддерживаемые компонентом Database Engine. Далее обсуждается использование процедурных расширений совместно с инструкциями Transact-SQL для реализации пакетов. Пакет можно сохранить в виде объекта базы данных как хранимую процедуру или ОПФ. Одни хранимые процедуры создаются пользователями, другие же предоставляются разработчиками Microsoft и называются системными хранимыми процедурами. В отличие от определяемых пользователем хранимых процедур, ОПФ возвращают значение вызывающему их объекту. Все подпрограммы можно реализовать или на языке Transact-SQL, или на другом языке программирования, таком как C# или Visual Basic.

**Процедурные расширения**

В предыдущих работах мы познакомились с инструкциями Transact-SQL языка описания данных DDL и языка манипуляции данными DML. Большинство этих инструкций можно сгруппировать в пакет. Как упоминалось ранее, пакет — это последовательность инструкций Transact-SQL и процедурных расширений, которые от правляются системе базы данных для совместного их выполнения. Количество инструкций в пакете ограничивается допустимым размером скомпилированного пакетного объекта. Преимущество пакета над группой отдельных инструкций состоит в том, что одновременное исполнение всех инструкций позволяет получить значительное улучшение производительности.

Существует несколько ограничений на включение разных инструкций языка Transact-SQL в пакет. Наиболее важным из них является то обстоятельство, что если пакет содержит инструкцию описания данных create view, create procedure или create trigger, то он не может содержать никаких других инструкций. Иными словами, такая инструкция должна быть единственной инструкцией пакета.

***ПРИМЕЧАНИЕ***

Инструкции языка описания данных разделяются с помощью инструкции go.

Каждое процедурное расширение языка Transact-SQL рассматривается по отдельности в следующих разделах.

**Блок инструкций**

Блок инструкций может состоять из одной или нескольких инструкций языка Transact-SQL. Каждый блок начинается с инструкции begin и заканчивается инструкцией END, как это показано далее:

BEGIN

statement1

statement 2

END

Блок можно разместить внутри инструкции if, чтобы в зависимости от определенного условия разрешить исполнение одной или нескольких инструкций (см. Пример 1).

**Инструкция IF**

Инструкция if языка Transact-SQL соответствует одноименной инструкции, поддерживаемой почти всеми языками программирования. Инструкция if выполняет одну или несколько составляющих блок инструкций, если логическое выражение, следующее после ключевого слова if, возвращает значение true (истина). Если же инструкция if содержит оператор else, то при условии, что логическое выражение возвращает значение false (ложь), выполняется вторая группа инструкций.

**Пример 1. Исполнение блока инструкций посредством инструкции if**

USE sample;

IF (SELECT COUNT(\*)

FROM works\_on

WHERE project\_no = 'p1'

GROUP BY project\_no ) > 3

PRINT 'The number of employees in the project p1 is 4 or more'

ELSE

BEGIN

PRINT 'The following employees work for the project p1'

SELECT employee.emp\_fname, employee.emp\_lname from employee, works\_on

WHERE employee.emp\_no = works\_on.emp\_no

AND project\_no = 'p1'

END

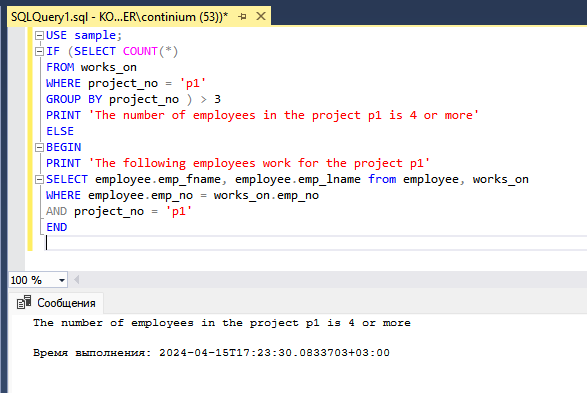


Рисунок 1. Исполнение блока инструкций посредством инструкции if

В примере 1 демонстрируется использование блока инструкций внутри инструкции if. Следующее далее логическое выражение инструкции if

(SELЕCT COUNT(\*)

FROM works\_on

WHERЕ projectno = ’p1’

GROP BY projectno) > 3

возвращает значение true (истина) для базы данных sample. Поэтому будет выполняться инструкция print, входящая в часть инструкции if. Обратите внимание на то обстоятельство, что в этом примере используется подзапрос, чтобы возвратить число строк (посредством агрегатной функции count), удовлетворяющих условию предложения where (project\_no=’p1’).

***ПРИМЕЧАНИЕ***

Оператор else инструкции if в примере 1 содержит две инструкции: print и select. Поэтому для выполнения этих инструкций их необходимо заключить в блок между ключевыми словами begin и end. (Инструкция print является процедурным расширением и возвращает определяемое пользователем сообщение.)

**Инструкция WHILE**

Инструкция while выполняет одну или несколько заключенных в блок инструкций, на протяжении времени, пока (while) логическое выражение возвращает значение true (истина). Иными словами, если выражение возвращает true, выполняется инструкция или блок инструкций, после чего снова осуществляется проверка выражения. Этот процесс повторяется до тех пор, пока выражение не возвратит значение FALSE (ложь).

Блок внутри инструкции while может содержать одну или две необязательных инструкций, применяемых для управления выполнением инструкций внутри блока: break или continue. Инструкция break останавливает выполнение инструкций внутри блока и начинает исполнение инструкций, следующих сразу же после этого блока. А инструкция continue останавливает выполнение только текущей инструкции в блоке и начинает выполнять его с самого начала.

В примере 2 показано использование инструкции while.

**Пример 2. Использование инструкции WHILE**

USE sample;

WHILE (select SUM(budget)

FROM project) < 500000

BEGIN

update project SET budget = budget\*1.1

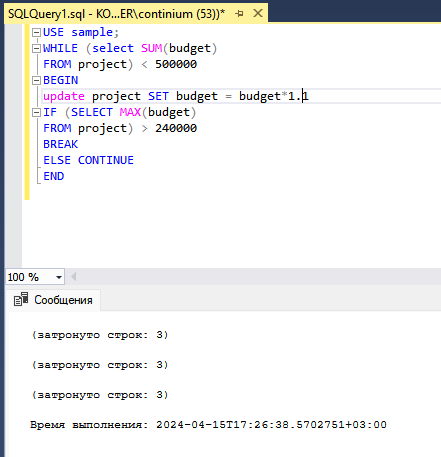
IF (SELECT MAX(budget)

FROM project) > 240000

BREAK

ELSE CONTINUE

END



***ПРИМЕЧАНИЕ***

Информационные сообщения о результате выполнения запроса наподобие сообщения, используемого в примере 2, можно подавить, применив инструкцию SET NOCOUNT ON.

**Локальные переменные**

Локальные переменные являются важным процедурным расширением языка Transact-SQL. Они применяются для хранения значений любого типа в пакетах и подпрограммах. Локальными они называются по той причине, что они могут быть использованы только в том пакете, в котором они были объявлены. (Компонент Database Engine также поддерживает глобальные переменные, которые уже были рассмотрены в работе 2.)

Все локальные переменные пакета объявляются, используя инструкцию declare. (Синтаксис этой инструкции приводится в примере 3.) Определение переменной состоит из имени переменной и ее типа данных. Имена локальных переменных в пакете всегда начинаются с префикса @. Присвоение значений локальной переменной осуществляется:

* используя специальную форму инструкции select;
* используя инструкцию set;
* непосредственно в инструкции declare посредством знака = (например, @extra\_ budget MONEY = 1500).

Использование первых двух способов присвоения значения локальным переменным показано в примере 3.

**Пример 3. Присвоение значения локальным переменным**

USE sample;

DECLARE @avg\_budget MONEY, @extra\_budget MONEY

SET @extra\_budget = 15000

SELECT @avg\_budget = AVG(budget)

FROM project

IF (SELECT budget

FROM project

WHERE project\_no='p1') < @avg\_budget

BEGIN

UPDATE project

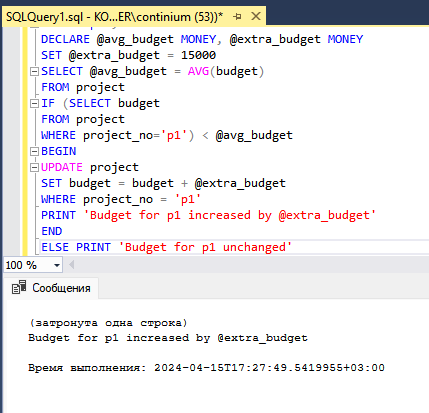
SET budget = budget + @extra\_budget

WHERE project\_no = 'p1'

PRINT 'Budget for p1 increased by @extra\_budget'

END

ELSE PRINT 'Budget for p1 unchanged'



**Прочие процедурные инструкции**

Процедурные расширения языка Transact-SQL также содержат следующие инструкции:

* RETURN;
* GOTO;
* RAISEERROR;
* WAITFOR.

Инструкция return выполняет ту же самую функцию внутри пакета, что и инструкция break внутри блока while. Иными словами, инструкция return останавливает выполнение пакета и начинает исполнение первой инструкции, следующей за пакетом.

Инструкция goto передает управление при выполнении пакета инструкции Transact-SQL внутри пакета, обозначенной маркером. Инструкция raiseerror выводит определенное пользователем сообщение об ошибке и устанавливает флаг системной ошибки. Номер ошибки в определяемом пользователем сообщении должен быть больше, чем 50 000, т. к. все номера ошибок меньшие или равные 50 000 определены системой и зарезервированы компонентом Database Engine. Значения номеров ошибок сохраняются в глобальной переменной @@error.

Инструкция waitfor определяет или период времени (с параметром delay) или определенное время (с параметром time), на протяжении или до которого, соответственно, система должна ожидать, прежде чем исполнять следующую инструкцию пакета. Синтаксис этой инструкции выглядит следующим образом:

WAITFOR {DELAY ’time’ | TIME ’time’ | TIMEOUT 'timeout’}

Параметр delay указывает СУБД ожидать, пока не истечет указанный период времени, а параметр time указывает точку во времени, в одном из допустимых форматов, до которой ожидать. Параметр timeout, за которым следует аргумент timeout, задает период времени в миллисекундах, в течение которого надо ожидать прибытия сообщения в очередь.

**Обработка исключений с помощью инструкций TRY, CATCH и THROW**

В версиях SQL Server более ранних, чем SQL Server 2005 требовалось наличие кода для обработки ошибок после каждой инструкции Transact-SQL, которая могла бы вызвать ошибку. (Для обработки ошибок можно использовать глобальную переменную @@error) Начиная с версии SQL Server 2005, исключения можно перехватывать для обработки с помощью инструкций try и catch. В этом разделе сначала объясняется значение понятия "исключение", после чего обсуждается работа этих двух инструкций.

Исключением (exception) называется проблема (обычно ошибка), которая не позволяет продолжать выполнение программы. Выполняющаяся программа не может продолжать исполнение по причине недостаточности информации, необходимой для обработки ошибки в данной части программы. Поэтому задача обработки ошибки передается другой части программы.

Роль инструкции try заключается в перехвате исключения. (Поскольку для реализации этого процесса обычно требуется несколько инструкций, то обычно применяется термин "блок try”, а не "инструкция try”.) Если же в блоке try возникает исключение, компонент системы, называющийся обработчиком исключений, доставляет это исключение для обработки другой части программы. Эта часть программы обозначается ключевым словом catch и поэтому называется блоком catch.

***ПРИМЕЧАНИЕ***

Обработка исключений с использованием инструкций try и catch является общим методом обработки ошибок, применяемым в современных языках программирования, таких как C# и Java.

Обработка исключений с помощью блоков try и catch предоставляет программисту множество преимуществ, включая следующие:

* исключения предоставляют аккуратный способ определения ошибок без загромождения кода дополнительными инструкциями;
* исключения предоставляют механизм прямой индикации ошибок, вместо использования каких-либо побочных признаков;
* программист может видеть исключения и проверить их в процессе компиляции.

В SQL Server 2012 добавлена еще одна инструкция throw, имеющая отношение к обработке ошибок. Эта инструкция позволяет вызвать исключение, которое улавливается в блоке обработки исключений. Иными словами, инструкция throw — это другой механизм возврата, который работает подобно рассмотренной ранее инструкции RAISEERROR.

Использование инструкций try, catch и throw для обработки исключений показано в примере 4. В данном примере показано, как для обработки исключений оформлять инструкции в пакет и выполнять откат результатов исполнения всей группы инструкций при возникновении ошибки. Для правильного функционирования данного примера требуется наличие ссылочной целостности между таблицами department и employee базы данных sample. По этой причине обе эти таблицы нужно создать, используя ограничения первичного (primary key) и внешнего (foreign key) ключей, как это сделано в примере 11 лабораторной работы 3.

**Пример 5 Применение инструкций TRY, CATCH и THROW для обработки исключений**

USE sample;

BEGIN TRY

BEGIN TRANSACTION

insert into employee values (11111, 'Ann', 'Smith','d2');

insert into employee values(22222, 'Matthew', 'Jones','d4');

insert into employee values(33333, 'John', 'Barrimore', 'd2');

COMMIT TRANSACTION

PRINT 'Transaction committed'

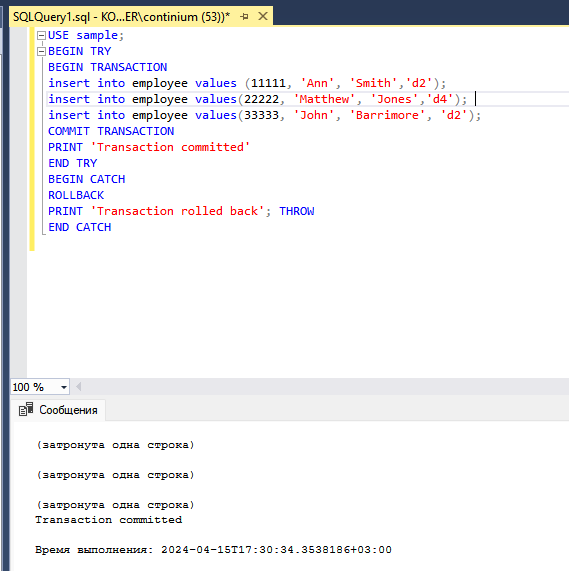
END TRY

BEGIN CATCH

ROLLBACK

PRINT 'Transaction rolled back'; THROW

END CATCH



***ПРИМЕЧАНИЕ***

Инструкции Transact-SQL begin transaction, commit transaction и rollback начинают, фиксируют и выполняют откат транзакций соответственно.

**Хранимые процедуры**

Хранимая процедура — это специальный тип пакета инструкций Transact-SQL, созданный, используя язык SQL и процедурные расширения. Основное различие между пакетом и хранимой процедурой состоит в том, что последняя сохраняется в виде объекта базы данных. Иными словами, хранимые процедуры сохраняются на стороне сервера, чтобы улучшить производительность и постоянство выполнения повторяемых задач.

Компонент Database Engine поддерживает хранимые процедуры и системные процедуры. Хранимые процедуры создаются таким же образом, как и все другие объекты баз данных, т. е. при помощи языка DDL. Системные процедуры предоставляются компонентом Database Engine и могут применяться для доступа к информации в системном каталоге и ее модификации. В этом разделе рассматриваются хранимые процедуры, которые определяются пользователями, а системные процедуры рассматриваются в следующей работе.

При создании хранимой процедуры можно определить необязательный список параметров. Таким образом, процедура будет принимать соответствующие аргументы при каждом ее вызове. Хранимые процедуры могут возвращать значение, содержащее определенную пользователем информацию или, в случае ошибки, соответствующее сообщение об ошибке.

Хранимая процедура предварительно компилируется перед тем, как она сохраняется в виде объекта в базе данных. Предварительно компилированная форма процедуры сохраняется в базе данных и используется при каждом ее вызове. Это свойство хранимых процедур предоставляет важную выгоду, заключающуюся в устранении (почти во всех случаях) повторных компиляций процедуры и получении соответствующего улучшения производительности. Это свойство хранимых процедур также оказывает положительный эффект на объем данных, участвующих в обмене между системой баз данных и приложениями. В частности, для вызова хранимой процедуры объемом в несколько тысяч байтов может потребоваться меньше, чем 50 байтов. Когда множественные пользователи выполняют повторяющиеся задачи с применением хранимых процедур, накопительный эффект такой экономии может быть довольно значительным.

Хранимые процедуры можно также использовать для следующих целей:

* управления авторизацией доступа;
* создания аудиторского следа действий с таблицами баз данных.

Использование хранимых процедур предоставляет возможность управления безопасностью на уровне, значительно превышающем уровень безопасности, предоставляемый использованием инструкций grant и revoke, с помощью которых пользователям предоставляются разные привилегии доступа. Это возможно вследствие того, что авторизация на выполнение хранимой процедуры не зависит от авторизации на модифицирование объектов, содержащихся в данной хранимой процедуре, как это описано в следующем разделе.

Хранимые процедуры, которые выполняют аудит операций записи и/или чтения таблиц, предоставляют дополнительную возможность обеспечения безопасности базы данных. Используя такие процедуры, администратор базы данных может отслеживать модификации, вносимые в базу данных пользователями или прикладными программами.

**Создание и исполнение хранимых процедур**

Хранимые процедуры создаются посредством инструкции create procedure, которая имеет следующий синтаксис:

CREATE PROC[EDURE] [schema\_name.]proc\_name

[({@param1} type1 [VARYING] [= default1] [OUTPUT])] {, ...}

[WITH {RECOMPILE | ENCRYPTION | EXECUTE AS ’user\_name’}]

[FOR REPLICATION]

AS batch | EXTERNAL NAME method\_name

Параметр schema\_name определяет имя схемы, которая назначается владельцем созданной хранимой процедуры. Параметр proc\_name определяет имя хранимой процедуры. Параметр @param1 является параметром процедуры (формальным аргументом), чей тип данных определяется параметром type1. Параметры процедуры являются локальными в пределах процедуры, подобно тому, как локальные переменные являются локальными в пределах пакета. Параметры процедуры — это значения, которые передаются вызывающим объектом процедуре для использования в ней. Параметр default1 определяет факультативное значение по умолчанию для соответствующего параметра процедуры. (Значением по умолчанию также может быть NULL.)

Параметр output указывает, что параметр процедуры является возвращаемым, и с его помощью можно возвратить значение из хранимой процедуры вызывающей процедуре или системе (см. Пример 9 далее в этом разделе).

Как уже упоминалось ранее, предварительно компилированная форма процедуры сохраняется в базе данных и используется при каждом ее вызове. Если же по каким-либо причинам хранимую процедуру требуется компилировать при каждом ее вызове, при объявлении процедуры используется опция with recompile.

*ПРИМЕЧАНИЕ*

Использование опции with recompile сводит на нет одно из наиболее важных преимуществ хранимых процедур: улучшение производительности благодаря одной компиляции. Поэтому опцию with recompile следует использовать только при частых изменениях используемых хранимой процедурой объектов базы данных.

Предложение execute as определяет контекст безопасности, в котором должна исполняться хранимая процедура после ее вызова. Задавая этот контекст, с помощью Database Engine можно управлять выбором учетных записей пользователей для проверки полномочий доступа к объектам, на которые ссылается данная хранимая процедура.

По умолчанию использовать инструкцию create proceure могут только члены предопределенной роли сервера sysadmin и предопределенной роли базы данных db\_owner или db\_ddladmin. Но члены этих ролей могут присваивать это право другим пользователям с помощью инструкции grant create procedure. 1

В примере 6 показано создание простой хранимой процедуры для работы с таблицей project.

**Пример 6. Создание хранимой процедуры для изменения данных таблицы**

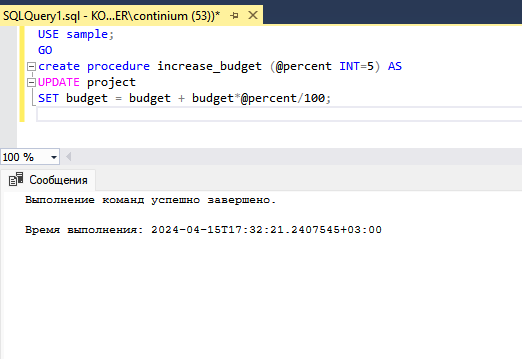
USE sample;

GO

create procedure increase\_budget (@percent INT=5) AS

UPDATE project

SET budget = budget + budget\*@percent/100;



***ПРИМЕЧАНИЕ***

Для разделения двух пакетов используется инструкция go. Инструкцию create procedure нельзя объединять с другими инструкциями Transact-SQL в одном пакете.

Хранимая процедура increase\_budget увеличивает бюджеты для всех проектов на определенное число процентов, определяемое посредством параметра @percent. В процедуре также определяется значение числа процентов по умолчанию (5), которое применяется, если во время выполнения процедуры этот аргумент отсутствует.

***ПРИМЕЧАНИЕ***

Хранимые процедуры могут обращаться к несуществующим таблицам. Это свойство позволяет выполнять отладку кода процедуры, не создавая сначала соответствующие таблицы и даже не подключаясь к конечному серверу.

В отличие от основных хранимых процедур, которые всегда сохраняются в текущей базе данных, возможно создание временных хранимых процедур, которые всегда помещаются во временную системную базу данных tempdb. Одним из поводов для создания временных хранимых процедур может быть желание избежать повторяющегося исполнения определенной группы инструкций при соединении с базой данных. Можно создавать локальные или глобальные временные процедуры. Для этого имя локальной процедуры задается с одинарным символом # (#proc\_name), а имя глобальной процедуры — с двойным (##proc\_name). Локальную временную хранимую процедуру может выполнить только создавший ее пользователь и только в течение соединения с базой данных, в которой она была создана. Глобальную временную процедуру могут выполнять все пользователи, но только до тех пор, пока не завершится последнее соединение, в котором она выполняется (обычно это соединение создателя процедуры).

Жизненный цикл хранимой процедуры состоит из двух этапов: ее создания и ее выполнения. Каждая процедура создается один раз, а выполняется многократно. Хранимая процедура выполняется посредством инструкции execute пользователем, который является владельцем процедуры или обладает правом execute для доступа к этой процедуре (см. работу 12). Инструкция execute имеет следующий синтаксис:

[[EXEC[UTE]] [@return\_status =] {proc\_name | @proc\_name\_var}

{[[Sparameter1 =] value|[@parameter1=] @variable [OUTPUT]]|DEFAULT}..

[WITH RECOMPILE]

За исключением параметра return\_status, все параметры инструкции execute имеют такое же логическое значение, как и одноименные параметры инструкции create procedure. Параметр return\_status определяет факультативную целочисленную переменную, в которой сохраняется состояние возврата процедуры. Значение параметру можно присвоить, используя или константу (value), или локальную переменную (@variabie). Порядок значений именованных параметров не важен, но значения неименованных параметров должны предоставляться в том порядке, в каком они определены в инструкции create procedure.

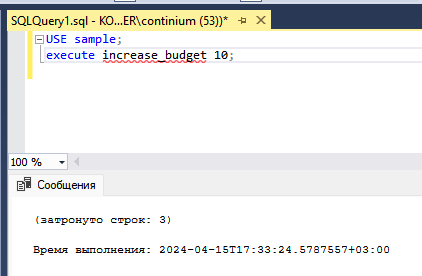
Предложение default предоставляет значения по умолчанию для параметра процедуры, которое было указано в определении процедуры. Когда процедура ожидает значение для параметра, для которого не было определено значение по умолчанию и отсутствует параметр, либо указано ключевое слово default, то происходит ошибка.

***ПРИМЕЧАНИЕ***

Когда инструкция execute является первой инструкцией пакета, ключевое слово execute можно опустить. Тем не менее будет надежнее включать это слово в каждый пакет.

Использование инструкции execute показано в примере 7.

Пример 7. Применение инструкции EXECUTE



USE sample;

EXЕCUTE increase\_budget 10;

Инструкция execute в примере 7 выполняет хранимую процедуру increase\_budget (см. Пример 6), которая увеличивает бюджете всех проектов на 10%.

В примере 8 показано создание хранимой процедуры для обработки данных в таблицах employee и works\_on.

**Пример 8. Создание хранимой процедуры для обработки таблиц EMPLOYEE и WORKS ON**

USE sample;

GO

CREАTE PROCEDURE modify\_empno (@old\_no INTEGER, @new\_no INTEGER) AS

UPDATE employee

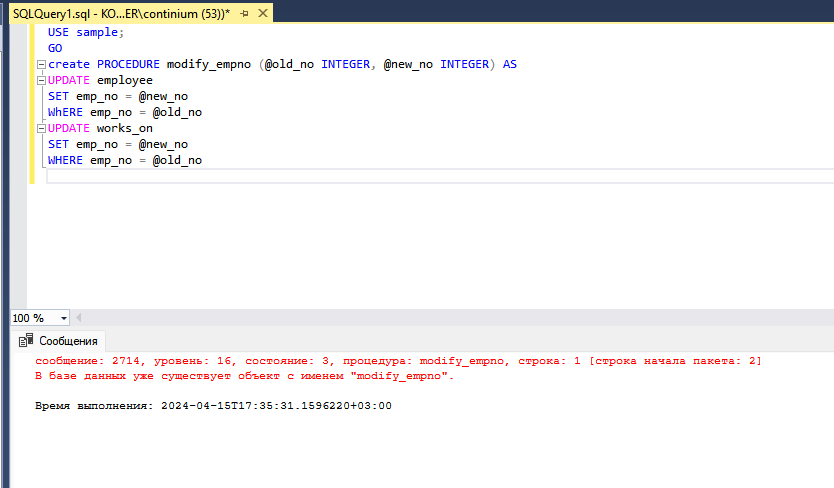
SET emp\_no = @new\_no

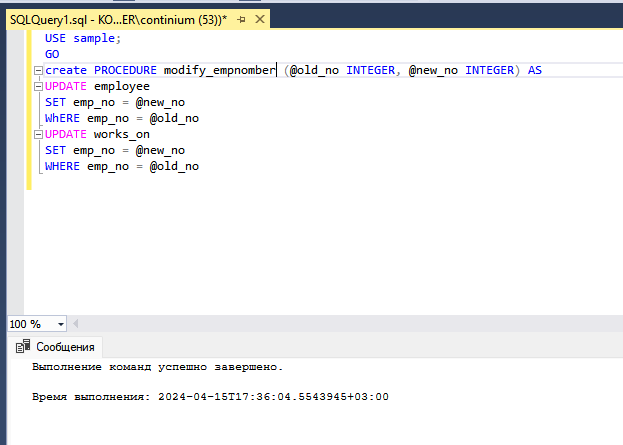
WERE emp\_no = @old\_no

UPDATE works\_on

SET emp\_no = @new\_no

WHERE emp\_no = @old\_no





**Пример 9. Использование в хранимой процедуре предложения OUTPUT**

USE sample;

GO

create procedure delete\_emp @employee\_no INT, @counter INT OUTPUT AS

SELECT @counter = COUNT(\*)

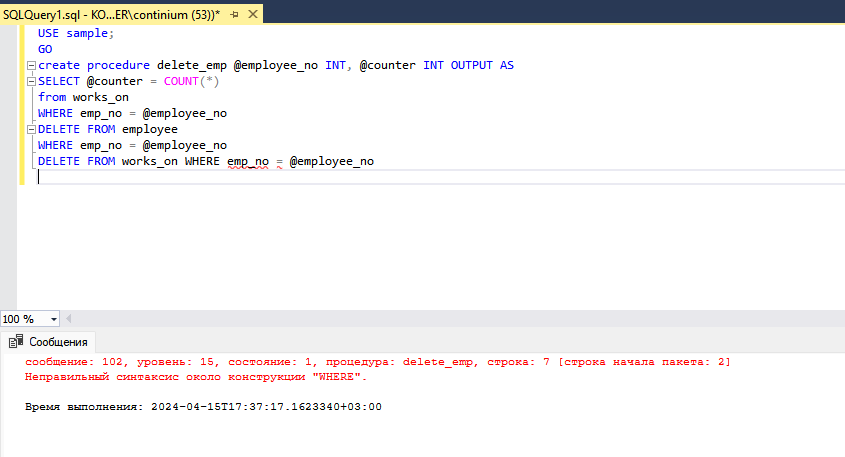
from works\_on

WHERE emp\_no = @employee\_no

DELETE FROM employee

WHERE emp\_no = @employee\_no

DELETE FROM works\_on WНERE emp\_no = @employee\_no



***ПРИМЕЧАНИЕ***

Значение параметра возвращается вызывающей процедуре только в том случае, если указана опция output. В примере 9 процедура delete\_emp передает вызывающей процедуре параметр @counter, следовательно, хранимая процедура возвращает значение системе. Поэтому параметр @counter необходимо указывать как в опции output при объявлении процедуры, так и в инструкции execute при ее вызове.

**Предложение** WITH RESULTS SETS **инструкции** EXECUTE

В SQL Server 2012 для инструкции execute вводится предложение with result sets, посредством которого при выполнении определенных условий можно изменять форму результирующего набора хранимой процедуры.

Следующие два примера помогут объяснить это предложение. Пример 10 является вводным примером, который показывает, как может выглядеть результат, когда опущено предложение with result sets.

**Пример 10. Определение хранимой процедуры**

Выполнение этой процедуры выводит таблицу с двумя столбцами, заработия которых совпадают с наименованиями соответствующих столбцов таблицы базы данных, т. е. emp\_no и emp\_lname. Чтобы изменить заработия столбцов результата (а также их тип данных), в SQL Server 2012 применяется новое предложение with result sets. Применение этого предложения показано в примере 11.

**Пример 11. Применение инструкции EXECUTE с**  **предложением WITH RESULT SETS**

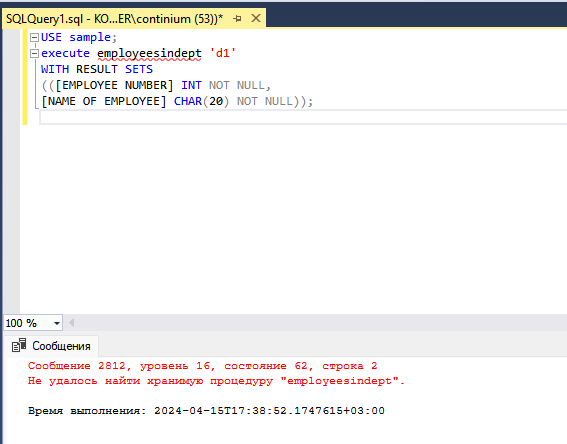
USE sample;

execute employeesindept 'd1'

WITH RESULT SETS

(([EMPLOYEE NUMBER] INT NOT NULL,

[NAME OF EMPLOYEE] CHAR(20) NOT NULL));



Проанализируйте результат выполнения хранимой процедуры, вызванной таким способом.

Изменение структуры хранимых процедур

Компонент Database Engine также поддерживает инструкцию alter procedure для модификации структуры хранимых процедур. Инструкция alter procedure обычно применяется для изменения инструкций Transact-SQL внутри процедуры. Все параметры инструкции alter procedure имеют такое же значение, как и одноименные параметры инструкции create procedure. Основной целью использования этой инструкции является избежание переопределения существующих прав хранимой процедуры.

***ПРИМЕЧАНИЕ***

Компонент Database Engine поддерживает тип данных cursor. Этот тип данных используется для объявления курсоров в хранимых процедурах. Курсор — это конструкция программирования, применяемая для хранения результатов запроса (обычно набора строк) и для предоставления пользователям возможности отображать этот результат построчно.

Для удаления одной или группы хранимых процедур используется инструкция drop procedure. Удалить хранимую процедуру может только ее владелец или члены предопределенных ролей db\_owner и sysadmin.

**Определяемые пользователем функции**

В языках программирования обычно имеется два типа подпрограмм:

* хранимые процедуры;
* определяемые пользователем функции (ОПФ).

Как уже было рассмотрено ранее в этой работе, хранимые процедуры состоят из нескольких инструкций и имеют от нуля до нескольких входных параметров, но обычно не возвращают никаких параметров. В отличие от хранимых процедур, функции всегда возвращают одно значение. В этом разделе мы рассмотрим создание и использование определяемых пользователем функций (ОПФ) (User Defined Functions — UDF).

**Создание и выполнение определяемых пользователем функций**

Определяемые пользователем функции создаются посредством инструкции create function, которая имеет следующий синтаксис:

CREATE FUNCTION [schema\_name.] function\_name [({@param} type [= default]) {,...}

RETURNS {scalar\_type | [©variable] TABLE}

[WITH {ENCRYPTION | SCHEMABINDING}

[AS] {block | RETURN (select\_statement)}

Параметр schema\_name определяет имя схемы, которая назначается владельцем создаваемой ОПФ, а параметр function\_name определяет имя этой функции. Параметр @param является входным параметром функции (формальным аргументом), чей тип данных определяется параметром type. Параметры процедуры — это значения, которые передаются вызывающим объектом определяемой пользователем функции для использования в ней. Параметр default определяет факультативное значение по умолчанию для соответствующего параметра функции. (Значением по умолчанию также может быть null.)

Предложение returns определяет тип данных значения, возвращаемого ОПФ. Это может быть почти любой стандартный тип данных, поддерживаемый системой баз данных, включая тип данных table. Единственным типом данных, который нельзя указывать, является тип данных timestamp.

Определяемые пользователем функции могут быть либо скалярными, либо табличными. Скалярные функции возвращают атомарное (скалярное) значение. Это означает, что в предложении returns скалярной функции указывается один из стандартных типов данных. Функция является табличной, если предложение returns возвращает набор строк (см. следующий подраздел для дополнительной информации).

Параметр with encryption в системном каталоге кодирует информацию, содержащую текст инструкции create function. Таким образом, предотвращается несанкционированный просмотр текста, который был использован для создания функции. Данная опция позволяет повысить безопасность системы баз данных.

Альтернативное предложение with schemabinding привязывает ОПФ к объектам базы данных, к которым эта функция обращается. После этого любая попытка модифицировать объект базы данных, к которому обращается функция, претерпевает неудачу. (Привязка функции к объектам базы данных, к которым она обращается, удаляется только при изменении функции, после чего параметр chemabinding больше не задан.)

Для того чтобы во время создания функции использовать предложение schemabinding, объекты базы данных, к которым обращается функция, должны удовлетворять следующим условиям,

* все представления и другие ОПФ, к которым обращается определяемая функция, должны быть привязаны к схеме;
* все объекты базы данных (таблицы, представления и ОПФ) должны быть в той же самой базе данных, что и определяемая функция.

Параметр block определяет блок begin/end, содержащий реализацию функции. Последней инструкцией блока должна быть инструкция return с аргументом. (Значением аргумента является возвращаемое функцией значение.) Внутри блока begin/end разрешаются только следующие инструкции:

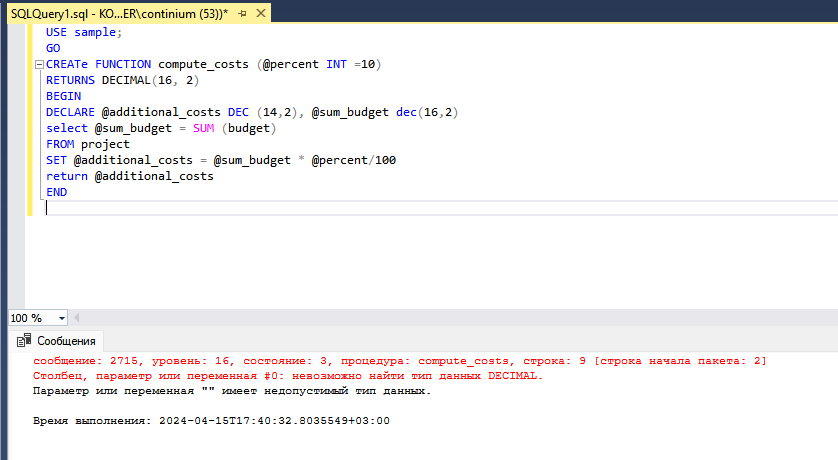
* инструкции присвоения, такие как set;
* инструкции для управления ходом выполнения, такие как while и if;
* инструкции declare, объявляющие локальные переменные;
* инструкции select, содержащие списки столбцов выборки с выражениями, значения которых присваиваются переменным, являющимися локальными для данной функции;
* инструкции insert, update и delete, которые изменяют переменные с типом данных table, являющиеся локальными для данной функции.

По умолчанию инструкцию create function могут использовать только члены предопределенной роли сервера sysadmin и предопределенной роли базы данных db\_owner или db\_ddladmin. Но члены этих ролей могут присвоить это право другим пользователям с помощью инструкции grant create function .

В примере 12 показано создание функции compute\_costs.

**Пример 12. Создание определяемой пользователем функции**

* Эта функция вычисляет возникающие дополнительные обшие затраты,
* при увеличении бюджетов проектов
* USE sample;
* GO
* CREATe FUNCTION compute\_costs (@percent INT =10)
* RETURNS DEСIMAL(16, 2)
* BEGIN
* DECLARE @additional\_costs DEC (14,2), @sum\_budget dec(16,2)
* select @sum\_budget = SUM (budget)
* FROM project
* SET @additional\_costs = @sum\_budget \* @percent/100
* return @additional\_costs
* END



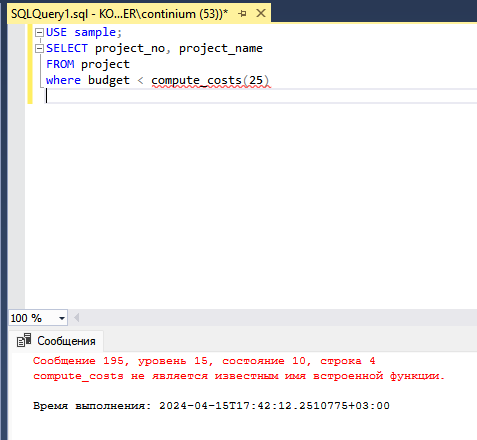
Функция compute\_costs вычисляет дополнительные расходы, возникающие при увеличении бюджетов проектов. Единственный входной параметр, @percent, определяет процентное значение увеличения бюджетов. В блоке begin/end сначала объявляются две локальные переменные: @additional\_costs и @sum\_budget, а затем с помощью инструкции select переменной @sum\_budget присваивается общая сумма всех бюджетов. После этого функция вычисляет общие дополнительные расходы и посредством инструкции return возвращает это значение.

Вызов определяемой пользователем функции

Определенную пользователем функцию можно вызывать с помощью инструкций Transact-SQL, таких как select, insert, update или delete. Вызов функции осуществляется, указывая ее имя с парой круглых скобок в конце, в которых можно задать один или несколько аргументов. Аргументы — это значения или выражения, которые передаются входным параметрам, определяемым сразу же после имени функции. При вызове функции, когда для ее параметров не определены значения по умолчанию, для всех этих параметров необходимо предоставить аргументы в том же самом порядке, в каком эти параметры определены в инструкции CREATE FUNCTION.

В примере 12 показан вызов функции compute\_costs (см. Пример 11) в инструкции SELECT.

**Пример 12. Вызов ОПФ в инструкции selec**



***ПРИМЕЧАНИЕ***

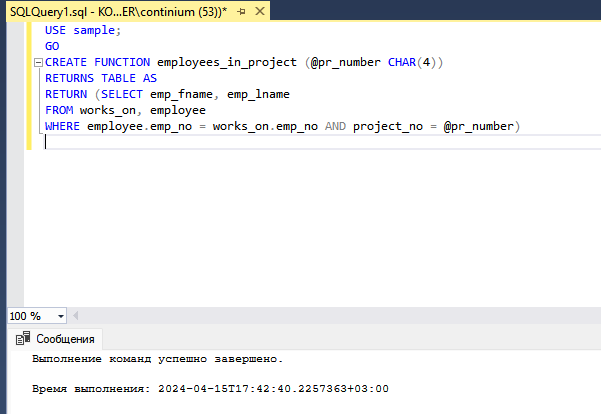
В инструкциях Transact-SQL имена функций необходимо задавать, используя имена, состоящие из двух частей: schema\_name. function\_name.

Возвращающие табличное значение функции

Как уже упоминалось ранее, функция является возвращающей табличное значение, если ее предложение returns возвращает набор строк. В зависимости от того, каким образом определено тело функции, возвращающие табличное значение функции классифицируются как встраиваемые (inline) и многоинструкционные (multistatement). Если в предложении returns ключевое слово table указывается без сопровождающего списка столбцов, такая функция является встроенной. Инструкция select встраиваемой функции возвращает результирующий набор в виде переменной с типом данных table (см. Пример 20). Многоинструкционная возвращающая табличное значение функция содержит имя, определяющее внутреннюю переменную с типом данных table. Этот тип данных указывается ключевым словом table, которое следует за именем переменной. В эту переменную вставляются выбранные строки, и она служит возвращаемым значением функции.

Создание возвращающей табличное значение функции показано в примере 13.

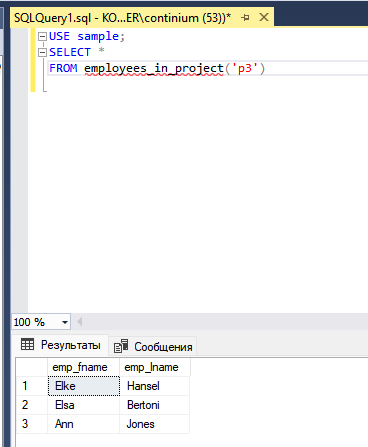
**Пример 13. Создание возвращающей табличное значение функции**



Тогда как функция в общем случае возвращает набор строк, предложение returns в определение данной функции содержит ключевое слово table, указывающее, что функция возвращает табличное значение. (Обратите внимание на то, что в примере 13 блок begin/end необходимо опустить, а предложение return содержит инструкцию SELECT.)

Использование функции employees\_in\_project приведено в примере 14.

**Пример 14. Использование возвращающей табличное значение функции**



**Возвращающие табличное значение функции и инструкция APPLY**

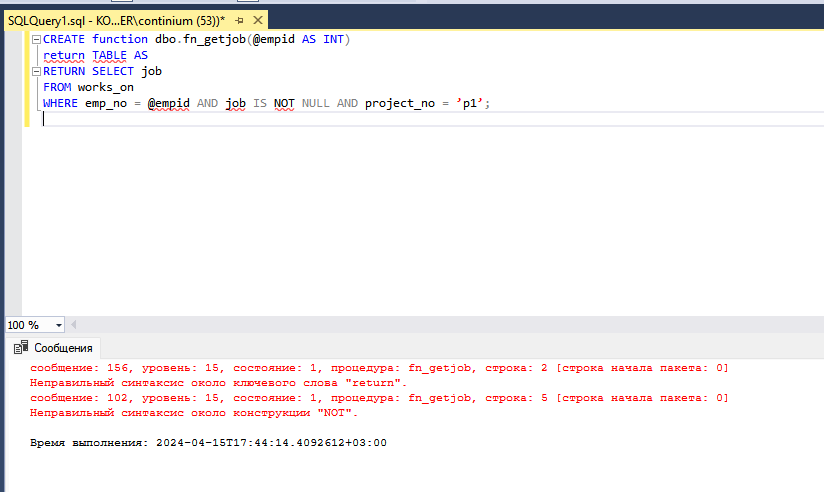
Реляционная инструкция apply позволяет вызывать возвращающую табличное значение функцию для каждой строки табличного выражения. Эта инструкция задается в предложении from соответствующей инструкции select таким же образом, как и инструкция join. Инструкция apply может быть объединена с табличной функцией для получения результата, похожего на результирующий набор операции соединения двух таблиц. Существует две формы инструкции apply:

* CROSS APPLY
* OUTER APPLY

Инструкция cross apply возвращает те строки из внутреннего (левого) табличного выражения, которые совпадают с внешним (правым) табличным выражением. Таким образом, логически, инструкция cross apply функционирует так же, как и инструкция INNER JOIN.

Инструкция outer apply возвращает все строки из внутреннего (левого) табличного выражения. (Для тех строк, для которых нет совпадений во внешнем табличном выражении, он содержит значения null в столбцах внешнего табличного выражения.) Логически, инструкция outer apply эквивалентна инструкции LEFT OUTER JOIN.

Применение инструкции APPLY показано в примерах 15—16.

**Пример 15. Создание возвращающей табличное значение функции** ****

В примере 16 этот результирующий набор "соединяется” предложением apply с содержимым таблицы employee.

**Пример 16. "Соединение" двух таблиц посредством предложения APPLY**

— используется CROSS APPLY

SELECT E.emp\_no, emp\_fname, emp\_lname, job

FROM employee as E CROSS APPLY dbo.fn\_getjob(E.emp\_no) AS A

— используется OUTER APPLY

SELECT E.emp\_no, emp\_fname, emp\_lname, job

FROM employee as E OUTER APPLY dbo.fn\_getjob(E.emp\_no) AS A

Результатом выполнения этих двух функций будут следующие две таблицы.

Вывести результат. Объяснить результат.

**Возвращающие табличное значение параметры**

Во всех версиях сервера, предшествующих SQL Server 2008, задача передачи подпрограмме множественных параметров была сопряжена со значительными сложностями. Для этого сначала нужно было создать временную таблицу, вставить в нее передаваемые значения, и только затем можно было вызывать подпрограмму. Начиная с версии SQL Server 2008, эта задача упрощена, благодаря возможности использования возвращающих табличное значение параметров, посредством которых результирующий набор может быть передан соответствующей подпрограмме.

Использование возвращающего табличное значение параметра показано в примере 17.

**Пример 24. Применение возвращающего табличное значение параметра**

USE sample;

GО

CREATETYPE departmentType AS TABLE

(dept\_no CHAR(4),

dеpt\_name CHAR(25),

location CHAR(30));

GO

CREATE TABLE #dallasTable

(dept\_no CHAR(4),

dept\_nаme CHAR(25),

location CHAR(30));

GO

CREATE PROCEDURE insertРroc @Dallas departmentType READONLY AS

SET NOCOUNT ON

INSERТ INTO #dallasTable (dept\_no, dept\_name, location)

SELECT \* FROM @Dallas

GO

DECLARE @Dallas AS departmentType;

INSERT INTO @Dallas( dept\_no, dept\_name, location)

SELECT \* FROM department WHERE location = 'Dallas'

EХEC insertProc @Dallas;

Что выполняет данная инструкция. Объяснить

Найдите временную таблицу. Где она создается?

Исправьте пакет так, чтобы он выводил содержимое временной таблицы

Использование возвращающих табличное значение параметров предоставляет следующие преимущества:

* упрощается модель программирования подпрограмм;
* уменьшается количество обращений к серверу и получений соответствующих ответов;
* таблица результата может иметь произвольное количество строк.

**Изменение структуры определяемых пользователями инструкций**

Язык Transact-SQL также поддерживает инструкцию alter function, которая модифицирует структуру определяемых пользователями инструкций (ОПФ). Эта инструкция обычно используется для удаления привязки функции к схеме. Все параметры инструкции alter function имеют такое же значение, как и одноименные параметры инструкции create function.

Для удаления ОПФ применяется инструкция drop function. Удалить функцию может только ее владелец или член предопределенной роли db\_owner или sysadmin.

**Резюме**

Хранимая процедура представляет собой специальный пакет, созданный либо на языке Transact-SQL, либо используя общеязыковую среду выполнения CLR. Хранимые процедуры используются для следующих целей:

* управления авторизацией доступа;
* создания контрольной записи действий с таблицами баз данных;
* принудительного обеспечения целостности данных и бизнес-правил применительно к модификации данных;
* улучшения производительности повторяемых задач.

Определяемые пользователем функции во многом схожи с хранимыми процедурами. Основное различие между ними заключается в том, что ОПФ не поддерживают параметров, но возвращают одно значение данных, которое может быть таблицей.

Для создания серверных объектов разработчики корпорации Microsoft рекомендуют использовать по умолчанию язык Transact-SQL.

**Упражнения**

**Упражнение 1**

Создайте пакет для вставки 3000 строк в таблицу employee. Значения столбца emp\_no должны быть однозначными в диапазоне от 1 до 3000. Всем ячейкам столбцов emp\_lname, emp\_fname и dept\_no присваиваются значения "Jane", "Smith" и "d1" соответственно.

CREATE OR ALTER PROCEDURE insert\_employees

AS

BEGIN

DECLARE @i INT = 1;

WHILE @i <= 3000

BEGIN

INSERT INTO employee (emp\_no, emp\_lname, emp\_fname, dept\_no)

VALUES (@i, 'Jane', 'Smith', 'd1');

SET @i = @i + 1;

END

COMMIT;

END;

**Упражнение 2**

Измените пакет из упражнения 1 таким образом, чтобы генерировать случайные значения для столбца emp\_no, используя функцию rand.

CREATE OR ALTER PROCEDURE insert\_employees

AS

BEGIN

DECLARE @i INT = 1;

WHILE @i <= 3000

BEGIN

DECLARE @emp\_no INT;

SET @emp\_no = ROUND(RAND(CHECKSUM(NEWID())) \* 1000000, 0);

INSERT INTO employee (emp\_no, emp\_lname, emp\_fname, dept\_no)

VALUES (@emp\_no, 'Jane', 'Smith', 'd1');

SET @i = @i + 1;

END

COMMIT;

END;