

# 7. Таблицы. Триггеры. Процедуры.

к.т.н., доцент кафедры ИиСП  
Лучинин  
Захар Сергеевич

# Типы таблиц в SQL

1. Постоянные таблицы

2. Временные таблицы **TEMPORARY TABLE**

2+. Обобщенные табличные выражения

3. Виртуальные таблицы (представления) **VIEW**

# Временные таблицы

**Временные таблицы** отличаются от постоянных тем, что **автоматически** удаляются, когда необходимость в них отпадает.

## Применение:

- Сохранение промежуточных результатов
- Агрегирование данных из различных источников
- Уменьшение числа строк при соединениях

# Временные таблицы в MSSQL

- **Локальная временная таблица** имеет префикс в виде одной решетки '#local\_temp\_table' — видна в текущем соединении пользователя и удаляется, как только пользователь отсоединился от экземпляра (instance) MS SQL Server.
- **Глобальная временная таблица** имеет префикс в виде двух решеток '##global\_temp\_table' — видна для любого пользователя после ее создания и удаляется, когда все пользователи, ссылающиеся на таблицу, отсоединятся от экземпляра MS SQL Server.

# Временная таблица в MSSQL

-- создание временной таблицы

```
CREATE TABLE #author_books(  
    id_author int,  
    book_quantity int  
)  
GO
```

-- вставка данных во временную таблицу

```
INSERT #author_books  
SELECT id_author, count(*) FROM author_has_book  
GROUP BY id_author
```

-- чтение из временной таблицы

```
SELECT TOP 10 * FROM #author_books ORDER BY book_quantity DESC
```

-- удаление временной таблицы

```
DROP TABLE #author_books
```

# Эффективное использование временных таблиц

- Включайте только необходимые столбцы и строки вместо использования всех тех столбцов и данных.
- При создании временных таблиц, не используйте операторы **SELECT INTO** (из-за блокировок).
- Используйте индексы на временных таблицах.
- После использования временной таблицы удаляйте ее.
- Не создавайте временную таблицу в транзакции

# Обобщенные табличные выражения (СТЕ)

- Временно именованный результирующий набор
- Получается при выполнении простого запроса и определяется в области выполнения инструкции

	UserID	Post	ManagerID
1	1	Директор	NULL
2	2	Главный бухгалтер	1
3	3	Бухгалтер	2
4	4	Начальник отдела продаж	1
5	5	Старший менеджер по продажам	4
6	6	Менеджер по продажам	5
7	7	Начальник отдела информационных технологий	1
8	8	Старший программист	7
9	9	Программист	8
10	10	Системный администратор	7

```
WITH EmployeeCTE (UserID, Post, ManagerID)
AS
(
    SELECT UserID, Post, ManagerID FROM Employee
)
SELECT * FROM EmployeeCTE
```

# Рекурсивные обобщенные табличные выражения

-- Вывести уровень сотрудника в дереве подчинения

UserID	Post	ManagerID	LevelUser
1	Директор	NULL	0
2	Главный бухгалтер	1	1
4	Начальник отдела продаж	1	1
7	Начальник отдела информационных технологий	1	1
8	Старший программист	7	2
10	Системный администратор	7	2
5	Старший менеджер по продажам	4	2
3	Бухгалтер	2	2
6	Менеджер по продажам	5	3
9	Программист	8	3



# Рекурсивные обобщенные табличные выражения

```
WITH EmployeeCTE(UserID, Post, ManagerID, LevelUser)
AS
(
    -- Находим исходную точку рекурсии
    SELECT UserID, Post, ManagerID, 0 AS LevelUser
    FROM Employee WHERE ManagerID IS NULL
    UNION ALL
    -- Объединение с EmployeeCTE (хотя мы его еще не дописали)
    SELECT t1.UserID, t1.Post, t1.ManagerID, t2.LevelUser + 1
    FROM Employee t1
    JOIN EmployeeCTE t2 ON t1.ManagerID = t2.UserID
)
SELECT * FROM EmployeeCTE ORDER BY LevelUser
```

# Рекомендации по созданию и использованию CTE

- За CTE должны следовать одиночные инструкции SELECT, INSERT, UPDATE или DELETE, ссылающиеся на некоторые или на все столбцы
- Несколько CTE могут быть определены в рамках запроса
- Возможно рекурсивное использование CTE

# Представление (VIEW)

**Представление (VIEW)** — объект базы данных, являющийся результатом выполнения запроса к базе данных, определенного с помощью оператора SELECT, в момент обращения к представлению.

-- Синтаксис для SQL Server

```
CREATE [ OR ALTER ] VIEW [ schema_name . ] view_name [
(column [ ,...n ] ) ]
[ WITH <view_attribute> [ ,...n ] ]
AS select_statement
[ WITH CHECK OPTION ]
[ ; ]
```

# Применение VIEW

- Для скрывтия подробностей сложных запросов.
- В качестве механизма безопасности, позволяющего пользователям обращаться к данным через представления, но не предоставляя им разрешений на непосредственный доступ к базовым таблицам.
- Для предоставления интерфейса обратной совместимости, моделирующего таблицу, схема которой изменилась.

# Алгоритмы VIEW (MERGE)

-- Создаем VIEW

```
CREATE VIEW readers_over_23000 AS  
SELECT * FROM reader  
WHERE reader_num > 23000
```

-- Делаем запрос к VIEW

```
SELECT * FROM readers_over_23000 WHERE first_name  
like 'A%'
```

-- Выполняемый запрос

```
SELECT * FROM reader  
WHERE reader_num > 23000 AND first_name like 'A%'
```

# Алгоритмы VIEW (TEMPTABLE)

-- Создаем VIEW

```
CREATE VIEW reader_rating AS
  SELECT i.id_reader, COUNT(*) AS quantity
  FROM issuance AS i
  GROUP BY i.id_reader
```

GO

-- Извлекаем максимальный рейтинг

```
SELECT MAX (quantity) FROM reader_rating
```

-- Алгоритм VIEW (MERGE)

-- Получаем ошибку

-- Cannot perform an aggregate function on an expression containing an aggregate or a subquery.

```
SELECT MAX(COUNT(*)) AS quantity FROM issuance GROUP BY id_reader
```

-- Алгоритм VIEW (TEMPTABLE)

-- Создание временной таблицы и выполнение над ней запросов

# Обновляемые представления

- Любые изменения, в том числе инструкции UPDATE, INSERT и DELETE, должны ссылаться на столбцы только одной таблицы.
- Нельзя обновлять сформированные столбцы:
  - с помощью агрегатной функции: AVG, COUNT, SUM...
  - на основе вычисления. Столбец нельзя вычислить по выражению, включающему другие столбцы.
- CHECK OPTION. Обеспечивает соответствие всех выполняемых для представления инструкций, изменяющих данные, критериям, заданным в выражении *select\_statement*.

# Индексированные представления

**Индексированное представление** - это представление, когда результирующий набор запроса материализуется.

## Условия создания:

- Создайте представление с помощью параметра `WITH SCHEMABINDING`.
- Создайте уникальный кластеризованный индекс для представления.

## Ограничения:

- Не работают подзапросы
- Доступен только `INNER JOIN`



# Хранимая процедура

**Хранимая процедура** – объект базы данных, представляющий собой набор SQL-инструкций, в которых можно объявлять переменные, управлять потоками данных, а также применять другие техники программирования.

# Создание хранимых процедур MSSQL

```
-- создание процедуры
CREATE PROCEDURE reader_procedure
AS
    SELECT *
    FROM reader
GO

-- вызов процедуры
EXECUTE reader_procedure

-- удаление процедуры
IF OBJECT_ID('reader_procedure','P') IS
NOT NULL
    DROP PROC reader_procedure
GO
```

id_reader	first_name	last_name	reader_num
1	Clyde	Adams	11111
2	Helen	Adams	23657
3	Dianne	Adkins	23637
4	Theresa	Adkins	23544
5	Silvia	Armstrong	23636
6	Ellis	Arnold	23528
7	Jeremy	Bailey	23676
8	Marta	Baldwin	23668
9	Bonnie	Ballard	23475

# Хранимые процедуры 3А

- Повторное использование кода. Обеспечивает связность доступа к данным и управления ими между различными приложениями.
- Изоляция пользователей от таблиц базы данных. Позволяет давать доступ к хранимым процедурам, но не к самим данным таблиц.
- Сокращения сетевого трафика. С помощью хранимых процедур множество запросов могут быть объединены.
- Повышенная производительность. При первом запуске хранимой процедуры она компилируется и в дальнейшем её обработка осуществляется быстрее.

# Хранимые процедуры ПРОТИВ

- Повышение нагрузки на сервер баз данных в связи с тем, что большая часть работы выполняется на серверной части, а меньшая - на клиентской.
- Распределение логики приложения в нескольких местах: программный код и код хранимых процедур, тем самым усложняя процесс понимания манипулирования данными.
- Миграция с одной СУБД на другую может привести к проблемам.

# Создание хранимых процедур MySQL

```
DELIMITER //
```

```
CREATE PROCEDURE `hello_world` ()  
  LANGUAGE SQL  
  DETERMINISTIC  
  SQL SECURITY DEFINER  
  COMMENT 'A procedure'  
BEGIN  
  SELECT 'Hello World !';  
END//
```

```
DELIMITER ;
```

- **Language:** в целях обеспечения переносимости, по умолчанию указан SQL.
- **Deterministic:** если процедура все время возвращает один и тот же результат, и принимает одни и те же входящие параметры. Значение по умолчанию - NOT DETERMINISTIC.
- **SQL Security:** проверка прав пользователя. INVOKER – это пользователь, вызывающий хранимую процедуру. DEFINER - это “создатель” процедуры. Значение по умолчанию - DEFINER.

# Пример хранимой процедуры MySQL

```
mysql> DELIMITER //
```

```
mysql> CREATE PROCEDURE count_records (OUT counter INT)
-> BEGIN
-> SELECT COUNT(*) INTO counter FROM t;
-> END;
-> //
```

Query OK, 0 rows affected (0.00 sec)

```
mysql> DELIMITER ;
```

```
mysql> CALL count_records(@a);
```

Query OK, 0 rows affected (0.00 sec)

```
mysql> SELECT @a;
```

```
+-----+
| @a   |
+-----+
| 3    |
+-----+
```

1 row in set (0.00 sec)

# Возврат данных из хранимой процедуры

- **Результирующий набор**
  - Возврат значений через инструкцию SELECT
- **Выходной параметр**
  - Возврат значений через OUTPUT параметр
- **Код возврата**
  - Процедура может возвращать целочисленное значение, называемое кодом возврата, чтобы указать состояние выполнения процедуры

# Результирующий набор

```
-- создание процедуры
CREATE PROCEDURE reader_procedure
AS
    SELECT *
    FROM reader
GO

-- вызов процедуры
EXECUTE reader_procedure
```

id_reader	first_name	last_name	reader_num
1	Clyde	Adams	11111
2	Helen	Adams	23657
3	Dianne	Adkins	23637
4	Theresa	Adkins	23544
5	Silvia	Armstrong	23636
6	Ellis	Arnold	23528
7	Jeremy	Bailey	23676
8	Marta	Baldwin	23668
9	Bonnie	Ballard	23475



# Выходной параметр

```
-- создание процедуры
CREATE PROCEDURE count_readers
-- входной параметр
@cnt INT OUTPUT
AS
    SELECT @cnt = COUNT(*)
    FROM reader
GO

-- объявляем переменную
DECLARE @cnt INT;

-- вызов процедуры
EXECUTE count_readers @cnt = @cnt OUTPUT

-- выводим количество записей
SELECT @cnt AS total_readers
```

Results		Messages	
		total_readers	
1		244	

# Параметры хранимой процедуры MySQL

- Пустой список параметров

```
CREATE PROCEDURE proc1 ()
```

- Один входящий параметр

```
CREATE PROCEDURE proc1 (IN varname DATA-TYPE)
```

- Один возвращаемый параметр.

```
CREATE PROCEDURE proc1 (OUT varname DATA-TYPE)
```

- Один параметр, одновременно входящий и возвращаемый.

```
CREATE PROCEDURE proc1 (INOUT varname DATA-TYPE)
```

# Код возврата

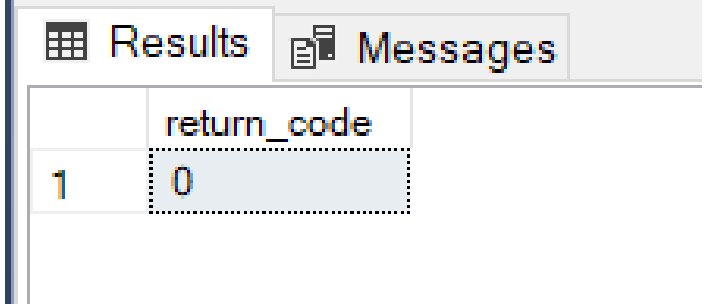
Процедура может возвращать целочисленное значение, называемое кодом возврата, чтобы указать состояние выполнения процедуры.

```
-- создание процедуры
CREATE PROCEDURE return_code
AS
    RETURN (0)
GO

-- объявляем переменную
DECLARE @cnt INT;

-- вызов процедуры
EXECUTE @cnt = return_code

-- выводим количество записей
SELECT @cnt AS return_code
```



The screenshot shows a SQL Server interface with two tabs: 'Results' and 'Messages'. The 'Results' tab is active, displaying a table with one column named 'return\_code' and one row containing the value '0'. The table is highlighted with a blue border.

	return_code
1	0

# Переменные

Локальная переменная представляет собой объект, содержащий одно значение определенного типа. Переменные обычно используются в пакетах и скриптах:

- в качестве счетчика цикла;
- для хранения значения, которое необходимо проверить инструкцией управления потоком;
- для хранения значения, возвращенного функцией или хранимой процедурой.

# Объявление переменных

Инструкция DECLARE инициализирует переменную следующим образом:

- Назначение имени. Первым символом имени должен быть одиночный символ @.
- Назначение длины и типа данных, определяемого системой или пользователем.
- Присваивает созданной переменной значение NULL.

```
DECLARE @MyCounter int;
```

```
DECLARE @LastName nvarchar(30), @FirstName nvarchar(20);
```

# Присвоение значения

При объявлении переменной присваивается значение NULL. Чтобы изменить значение переменной, применяется инструкция SET.

```
-- объявляем переменную  
DECLARE @cnt INT;
```

```
-- присвоение значения переменной  
SET @cnt = 3;
```

```
-- присвоение значения переменной через SELECT  
SELECT @cnt = COUNT(*) FROM reader
```

# Переменные в SQL

```
DECLARE a, b INT DEFAULT 5;
```

```
DECLARE str VARCHAR(50);
```

```
DECLARE today TIMESTAMP DEFAULT CURRENT_DATE;
```

```
DECLARE v1, v2, v3 TINYINT;
```

# Переменные в хранимых процедурах

**DECLARE** var\_name[,...] **type** [**DEFAULT** value]

**SET** var\_name = expr [, var\_name = expr] ...

**SELECT** col\_name[,...] **INTO** var\_name[,...]  
table\_expr



# Управляющие конструкции

- BEGIN...END
- BREAK
- CONTINUE
- ELSE (IF...ELSE)
- END  
(BEGIN...END)
- GOTO
- IF...ELSE
- RETURN
- THROW
- TRY...CATCH
- WAITFOR
- WHILE
- CASE

# Конструкция IF...ELSE

Инструкция, следующая за ключевым словом IF и его условием, выполняется только в том случае, если логическое выражение возвращает TRUE. Необязательное ключевое слово ELSE представляет другую инструкцию, которая выполняется, если условие IF не удовлетворяется и логическое выражение возвращает FALSE.

```
IF DATENAME(weekday, GETDATE()) IN (N'Saturday', N'Sunday')  
    SELECT 'Weekend';  
ELSE  
    SELECT 'Weekday';
```

# Конструкция WHILE

Ставит условие повторного выполнения SQL-инструкции или блока инструкций. Эти инструкции вызываются в цикле, пока указанное условие истинно. Вызовами инструкций в цикле WHILE можно контролировать из цикла с помощью ключевых слов BREAK и CONTINUE.

```
WHILE (SELECT AVG(ListPrice) FROM Production.Product) < $300
BEGIN
    UPDATE Production.Product
        SET ListPrice = ListPrice * 2
    SELECT MAX(ListPrice) FROM Production.Product
    IF (SELECT MAX(ListPrice) FROM Production.Product) > $500
        BREAK
    ELSE
        CONTINUE
END
PRINT 'Too much for the market to bear';
```

# Конструкция TRY...CATCH

Группа инструкций может быть заключена в блок TRY. Если ошибка возникает в блоке TRY, управление передается следующей группе инструкций, заключенных в блок CATCH.

```
BEGIN TRY
    -- Generate divide-by-zero error.
    SELECT 1/0;
END TRY
BEGIN CATCH
    -- Execute error retrieval routine.
    -- Resolve Error
END CATCH;
```

# SQL Курсор

**Курсор** - объект базы данных, который позволяет приложениям работать с записями «по одной», а не сразу с множеством, как это делается в обычных SQL командах.

- В схеме со статическим курсором информация читается из базы данных один раз и хранится в виде моментального снимка. Изменения, внесенные в базу данных другим пользователем, не видны.
- На время открытия курсора сервер устанавливает блокировку на все строки, включенные в его полный результирующий набор.
- Статический курсор не изменяется после создания и всегда отображает тот набор данных, который существовал на момент его открытия.

# Работа с курсорами

-- объявление курсора

```
DECLARE reader_cursor CURSOR FOR  
SELECT first_name, last_name FROM reader  
WHERE first_name LIKE 'A%'
```

-- открываем курсор

```
OPEN reader_cursor;
```

-- объявляем переменные для хранения прочитанных данных

```
DECLARE @LastName varchar(50), @FirstName varchar(50);
```

## DECLARE CURSOR

Определяет такие атрибуты серверного курсора, как свойства просмотра и запрос, используемый для построения результирующего набора, на котором работает курсор.

## OPEN

Открывает серверный курсор и заполняет его с помощью инструкции, определенной в инструкции DECLARE CURSOR или SET cursor\_variable.

# Работа с курсорами

```
-- читаем данные
FETCH NEXT FROM reader_cursor
    INTO @FirstName, @LastName;
WHILE @@FETCH_STATUS = 0
BEGIN
    -- выводим значение переменных
    PRINT 'Contact Name: ' + @FirstName + ' ' + @LastName
    FETCH NEXT FROM reader_cursor
END;
```

**FETCH** Получает определенную строку из серверного курсора

- **NEXT** Возвращает строку результата сразу же за текущей строкой и перемещает указатель текущей строки на возвращенную строку.
- **FIRST** Возвращает первую строку в курсоре и делает ее текущей.
- **LAST** Возвращает последнюю строку в курсоре, и делает ее текущей.

# Работа с курсорами

-- закрываем курсор

```
CLOSE reader_cursor;
```

-- удаляем ссылку курсора

```
DEALLOCATE reader_cursor;
```

## CLOSE

Закрывает открытый курсор, высвобождая текущий результирующий набор и снимая блокировки курсоров для строк, на которых установлен курсор. Инструкция CLOSE оставляет структуры данных доступными для повторного открытия

## DEALLOCATE

Удаляет ссылку курсора. Когда удаляется последняя ссылка курсора, Microsoft SQL Server освобождает структуры данных, составляющие курсор.



# User Defined Functions

**Функции** — это конструкции, содержащие исполняемый код. Функция выполняет какие-либо действия над данными и возвращает некоторое значение/набор данных. К функциям можно обращаться из триггеров, хранимых процедура и из других программных компонентов.

# Виды табличных функций

- **Табличная функция** — функция, в которой задается структура результирующей таблицы. В теле функции выполняется запрос, который наполняет эту результирующую таблицу данными. После чего, возвращается эта таблица.

```
SELECT * FROM sales.sale_by_store (602);
```

- **Скалярная функция** — Функция, которая возвращает 1 значение. Значение может иметь различный формат данных.

```
SELECT id, name, get_inventory_stock(id) AS current_supply  
FROM product  
WHERE id BETWEEN 75 and 80;
```

# Триггеры

Триггер — это хранимая процедура особого типа, которую пользователь не вызывает непосредственно, а исполнение которой обусловлено определенной модификацией данных в заданной таблице или столбце реляционной базы данных.

- Запускаются сервером при наступлении определенного события:
  - Вставка записи в таблицу
  - Обновление записи в таблице
  - Удаление записи из таблицы
- Формальное определение триггера:

```
CREATE TRIGGER имя_триггера
ON {имя_таблицы | имя_представления}
{AFTER | INSTEAD OF} [INSERT |
UPDATE | DELETE ]
AS выражения_sql
```

# Применение триггера

- Триггеры применяются для проверки правил предметной области и для обеспечения целостности данных
- Триггеры применяются при денормализации с целью поддержания актуальности данных;
- Накопление аудиторской информации посредством фиксации сведений о внесенных изменениях и тех лицах, которые их выполнили

# Особенности применения триггеров

- Не активируются при каскадных изменениях по внешним ключам.
- Триггер запускается сервером автоматически при попытке изменения данных в таблице, с которой он связан.
- Все производимые им модификации данных рассматриваются как одна транзакция. В случае обнаружения ошибки или нарушения целостности данных происходит откат этой транзакции.

# Создание триггера

```
CREATE TRIGGER issuance_check
ON issuance
AFTER INSERT, UPDATE
AS
    IF (ROWCOUNT_BIG() = 0)
    RETURN;
    -- проверяем существование невалидных записей
    IF EXISTS (SELECT * FROM issuance WHERE issue_date >
deadline_date)
    BEGIN
        -- кидаем ошибку и откатываем изменения
        RAISERROR ('issue_date could not be greater then
deadline_date', 16, 1);
        ROLLBACK TRANSACTION;
        RETURN
    END;
GO
```