



Понятие представлений, функций и процедур

Вот мы в лекции с Вами подытожили все рассмотренные темы выгрузки данных из баз данных понятием комплексных запросов (которые внутри себя содержат и вычисления, и условия отбора, и слияние таблиц, и группировку, и еще и использование подзапросов)

И пока что для «упрощения жизни» при написании SQL-скрипта мы использовали только форматирование кода, комментирование кода, а также опирались на различных помощников.

Но есть еще один способ борьбы со сложностью и упрощения себе жизни – это использовать **представления (Views), процедуры (Stored Procedures) и функции (Functions)**.

Тема считается не совсем для простых бизнес-пользователей - но по ней по отдельному ответвлению не пойдем – т.е., отдельного кина (мидквэла) как по оконным функциям не будет. Ограничимся просто базовым пониманием «что такое вообще есть» - кто захочет углубляться в SQL, то уже отдельно сам детали изучит.

Представления, процедуры и функции – это с одной стороны **просто разновидности хранения SQL-команд** (чтобы не повторяться, а раз написать – и потом их переиспользовать), но с другой стороны еще и имеют свои **особые функциональные предназначения**.

Мы как для бизнес-пользователя все же будем говорить о первой стороне медали. Итак, сложные SQL-запросы можно не просто сохранить как скрипт или добавить в сниппеты\шаблоны – их можно запихнуть в «оболочку» представления, процедуры или функции. И потом просто в коде ссылаться на них как на отдельную сущность.

Итак, **ПРЕДСТАВЛЕНИЕ (VIEW или в быту «вьюхи»)** – это виртуальная таблица. Она выглядит как обычная таблица: строки, столбцы, на пересечении значения. Но является виртуально сгенерированной из реальной таблицы (или нескольких таблиц) из БД.

И в отличие от реальных таблиц представление не содержит и не хранит в себе данные – только структуру таблицы. А данные в него подгружаются из

<https://www.udemy.com/user/nikita-sergeev-2/>

актуальных таблиц при его вызове (т.е., в момент обращения к самому представлению пользователем). Кроме случая, когда мы создаем так называемое «материализованное представление» (*MATERIALIZED VIEW*), которое содержит в себе также и данные – но не все СУБД поддерживают материализацию представлений. В любом случае для конечного пользователя представление выглядит и воспринимается как обычная таблица с данными.

Имя\название представления в классическом варианте берется в квадратные скобки и выглядит так: **[имя представления]**. Но на самом деле его можно писать и без всяких квадратных скобок аналогично обычным таблицам.

Создается оно командой **CREATE VIEW [имя представления] AS** перед обычным запросом **SELECT**: это значит, что **SELECT** становится представлением (виртуальной таблицей), на которую уже можно ссылаться в любых других запросах **SELECT**. Это считайте, как будто Вы написали запрос **SELECT** – и «облачив» его в представление используете далее вместо подзапроса.

К примеру, с подзапросом команда могла бы выглядеть так (серым в красной рамке выделен подзапрос, формирующий таблицу):

```
SELECT Sum(A3) FROM (SELECT A1, A2, A3 FROM T1)
GROUP BY A1
```

А если этот подзапрос «облачить» в представление:

```
CREATE VIEW [T1_a] AS
```

```
SELECT A1, A2, A3 FROM T
```

И после этого уже обращаться к этой таблице-представлению становится намного проще чем постоянно писать подзапрос – по сути Вы обращаетесь к представлению как к обычной таблице:

```
SELECT * FROM [T1_a]
```

Почитать и попрактиковаться с представлениями можно тут:
https://www.schoolsw3.com/sql/sql_view.php

Помимо такой «оптимизации» (когда запросам, в частности сложным\комплексным запросам, дается компактное простое имя как обычной таблице) представления подходят:

- для ограничения доступа к данным: когда реальная таблица (или несколько таблиц) формируют несколько представлений к каждому из которых получает доступ своя конкретная группа пользователей;
- по соображениям инфо-безопасности: для маскировки кода с обращением к реальным таблицам в БД.

Хранятся все представления в соответствующей папке **VIEWS** в конкретной БД.

<https://vk.com/id526939928>

Далее **ФУНКЦИИ (FUNCTIONS)** – это создаваемые Вами лично функции, которые потом можно вызывать в SQL-коде как обычные функции (математические, текстовые, дат и времени, логические). Т.е., когда Вам надо сделать навороченное вычисление, которое Вы еще и используете часто в разных скриптах - то самое оно преобразовать это вычисление в функцию и обращаться к ней при необходимости таких вычислений. Понятная для бизнес-пользователей аналогия – это пожалуй работает аналогично, как создаются формулы\функции с помощью **LAMBDA** в новых версиях Excel. Или, к примеру, когда Вы создаете свою **именованную функцию** в Excel.

Как это работает в SQL на примитивном пример. Допустим, Вы часто умножаете столбец на столбец. Можно создать функцию, в которой Вы указываете столбцы, а она их умножает друг на друга.

Например, код:

```
SELECT Price*Count AS Revenue
```

```
FROM T1
```

Если создать свою собственную функцию умножения 2-х столбцов (причем функция обязательно сводится к абстрактному виду как *название функции, аргументы и операции с ними*:

```
DELIMITER //  
  
CREATE FUNCTION Multiply (X, Y)  
  
RETURNS INT deterministic  
  
Begin  
  
  Declare R INT;  
  
  SET R=X*Y;  
  
  RETURN R;  
  
End //  
  
DELIMITER ;
```

Имя и Аргументы функции

Тип данных и вид возвращаемого результата

Имя для вычисляемого функцией результата и его тип данных

Устанавливаем формулу вычисления результата: по сути указываем как именно используются аргументы функции при его вычислении

Выводимый функцией результат

Обращу внимание, что в начале и конце стоит команда **DELIMITER** – это просто смена разделителя инструкций\ключевых команд вместо классической точки с запятой на **//** (потому что «;» используется внутри синтаксиса функции – и система не будет понимать, а когда же Вы закончили эту функций писать). Т.е., сначала надо временно сменить разделитель точку с запятой «;» (который ставится в конце строки) на какой-то другой (обычно меняют на два слеши **//**), а потом вернуть назад после создания функции «;» обратно.

Тогда можно будет использовать в SELECT'ах эту функцию и будет это выглядеть тогда уже так:

SELECT Multiply(Price,Count) **AS** Revenue

FROM T1

Я показал примитивный пример для того, чтобы понял полный новичок. Но думаю ясно, что в реальности такие простые матоперации «загонять» в функцию не имеет никакого смысла – оно только усложнит дело.

Но когда Вы используете **(не)реально сложные «многоэтажные» вычисления или операции**, то намного проще загнать их в функцию и вызывать эту функцию, чтобы просто подавать в нее аргументы и параметры, чем писать все время ее код. Да и Ваш код с именем функции будет намного понятнее, нежели скопировать и вставить в него сложное «трехэтажное» вычисление в нескольких местах.

Ну и **ПРОЦЕДУРЫ (STORED PROCEDURES)** – это часто используемый нами SQL-скрипт, который мы сохраняем в виде процедуры, чтобы повторно его использовать снова и снова когда нам будет необходимо.

Ценность процедуры в отличие от вьюхи (представления) в том, что процедуру можно ПАРАМЕТРИЗИРОВАТЬ (*т.е., указать какие в ней могут быть подставляемые\изменяемые **параметры** при ее использовании*).

Например, отдел организационного проектирования постоянно отбирает в выгрузки количество тех или иных должностей в компании. И делается это обычно простым скриптом:

SELECT count(должность) **FROM** штатка

WHERE должность='менеджер'

И в скрипте все время меняется просто условие в инструкции **WHERE** - а именно имя должности «директор», «менеджер», «инженер», «специалист»...

Ну, начинаем с того же, с чего начинали писать функцию – со смены разделителя в SQL-скрипте. И ровно по той же причине: потому что внутри кода создания процедуры используется обязательный разделитель «;».

DELIMITER //

А после смены разделителя уже можем создавать **процедуру**:



Запуск этого кода создаст процедуру с названием CountPosition и сохранит ее в БД в соответствующей папке **Stored Procedures**.

После завершения процедуры возвращаем стандартный разделитель:

DELIMITER ;

И далее можем **вызывать** эту процедуру в SQL-скрипте командным словом **CALL** (вызов) и указанием необходимых нам для конкретной выгрузки параметров:

CALL CountPosition ('инженер', @count);

где мы

- входным (in) параметром вводим «инженер», которое подставится внутри процедуры в условие WHERE

- результатом функции (out) объявляем фразу @count

И после этого можем писать **SELECT @count** – и на выходе получаем заданный в процедуре параметр **количество**, в который мы передали с помощью **into** вычисленное с помощью функции **count(*)** количество строк в таблице содержащей только строки с должностью «инженер».

На этом все. Тема это конечно не для бизнес-пользователей, да и вряд ли Вы будете это использовать в работе. Но если кого вдруг тема заинтересовала: регистрируйтесь на более продвинутые курсы по SQL или разбирайтесь самостоятельно по справочникам - и углубляйтесь