

Лабораторная работа 4

КОНТРОЛЬ ВВОДИМЫХ ДАННЫХ И НАСТРОЙКА ИХ ОТОБРАЖЕНИЯ

Цель работы: познакомиться с механизмом триггеров, организовать контроль вводимых данных, реализовать фильтрацию данных.

ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

1. Ознакомиться с теоретическими сведениями.
2. Создать необходимые триггеры и проверить их работоспособность.
3. Открыть копию проекта лабораторной работы №3.
4. Организовать программную проверку вводимых значений при помощи событий **OnSetText**, **OnValidate**, **OnChange** и **BeforePost**.
5. Реализовать изменение отображения данных с использованием события **OnGetText**, свойства **DisplayFormat** и **EditMask**
6. Организовать возможность фильтрации данных на основе SQL-запросов.
7. Реализовать фильтрацию по дате с применением свойства **Filter**.
8. Ответить на контрольные вопросы.

ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Триггер – это хранимая откомпилированная **SQL**-процедура, которая не вызывается непосредственно, а выполняется при наступлении определенного события внутри базы данных: вставки, удаления, обновления записей. Достоинством триггеров является то, что они могут сократить сетевой трафик. Сложные повторяющиеся задачи обрабатываются на сервере баз данных без необходимости отсылки промежуточных результатов приложению.

Триггеры применяются для обеспечения целостности данных и реализации сложной бизнес-логики. Они запускаются сервером автоматически при попытке изменения данных в таблице, с которой они связаны. Все производимые ими модификации данных рассматриваются как выполняемые в транзакции, в которой произошло действие, вызвавшее срабатывание триггера. И в случае обнаружения ошибки или нарушения целостности данных может произойти откат этой транзакции.

4. РАЗРАБОТКА ПРОСТЫХ ТРИГГЕРОВ В MYSQL

Рассмотрим процесс создания триггера на примере триггеров для таблицы **goods_catalog**. Создание триггера можно инициировать двумя путями. Первый способ это через контекстное меню каталога **Триггеры** в проводнике, а второй – через контекстное меню, при редактировании таблицы находясь на вкладке **Триггеры** (рис. 4.1).

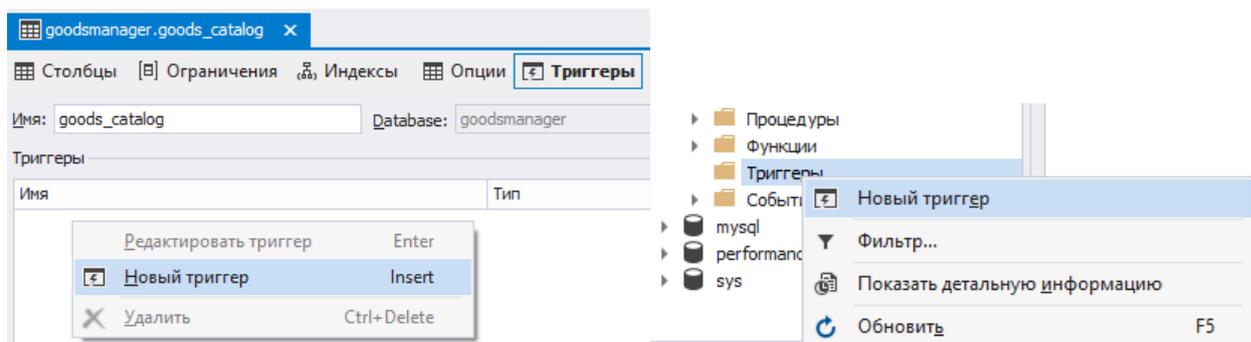


Рис. 4.1. Способы создания триггеров.

Триггеры выполняются автоматически при событиях **UPDATE**, **DELETE** и **INSERT** с возможностью выбрать момент срабатывания: до, после возникновения события. В триггерах к столбцам таблицы можно обращаться, используя псевдонимы **OLD** и **NEW**.

OLD.col_name ссылается на существующую строку столбца с именем **col_name** прежде, чем она будет модифицирована или удалена. **NEW.col_name** ссылается на новую строку столбца с именем **col_name**,

которая будет вставлена, или на уже существующую строку после того, как она будет модифицирована.

Создадим триггер который будет отвечать за постановку на учет товара на склад. Данный триггер должен срабатывать после вставки записи в таблицу **goods_catalog** (рис. 4.2):

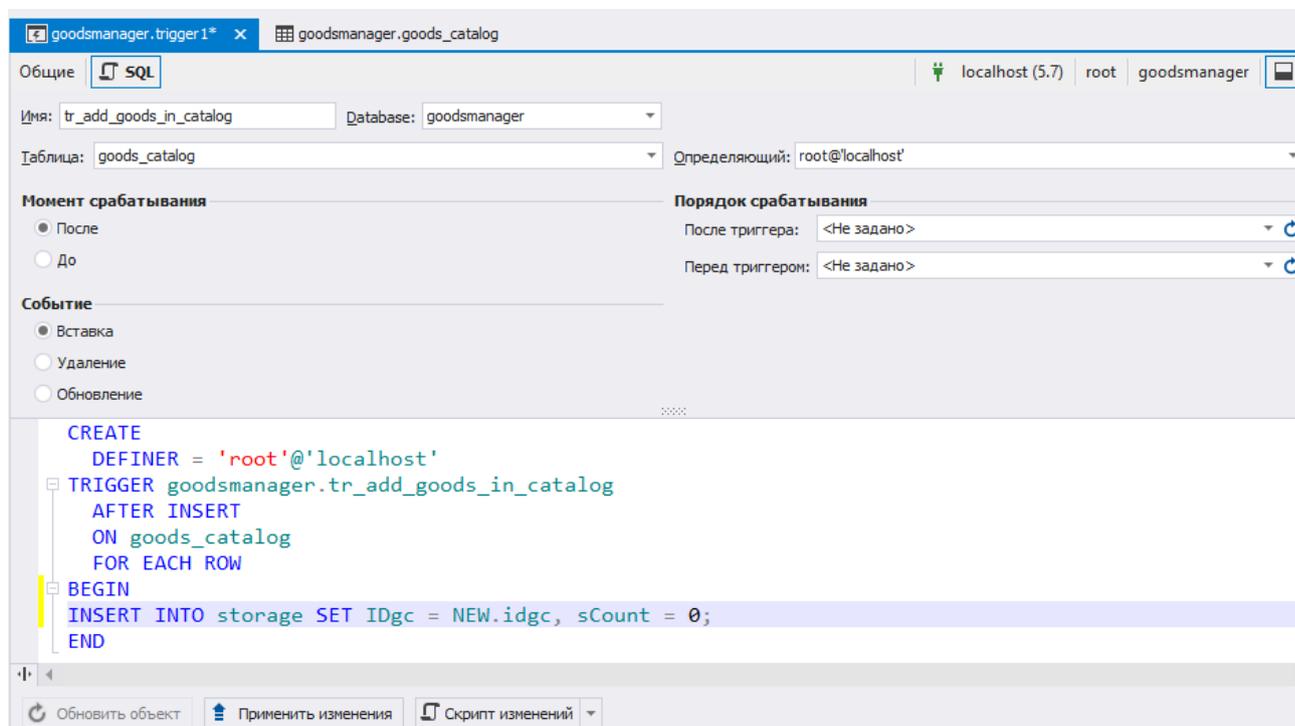


Рис. 4.2. Пример настройки триггера на добавление данных

Весь основной код, который пишет разработчик, располагается между строками **BEGIN** и **END**, остальная часть генерируется средой администрирования в зависимости от заданных настроек в интерфейсе. В нашем случае сформирована *SQL*-команда:

```
INSERT INTO storage SET IDgc = NEW.idgc, sCount = 0;
```

Она отвечает за добавление новой записи в таблицу **storage**, при этом полю **IDgc** устанавливается значение идентификатора нового товара, а полю **sCount** значение **0** (количество нового товара на складе равно нулю).

При использовании внешних ключей нет возможности удалить запись при наличии зависимых записей. В связи с этим на удаление товара из каталога также создадим триггер. Он будет удалять из перечня товара на складе только те товары, количество которых равно **0**. Данный триггер будет срабатывать до удаления товара из каталога (рис. 4.3):

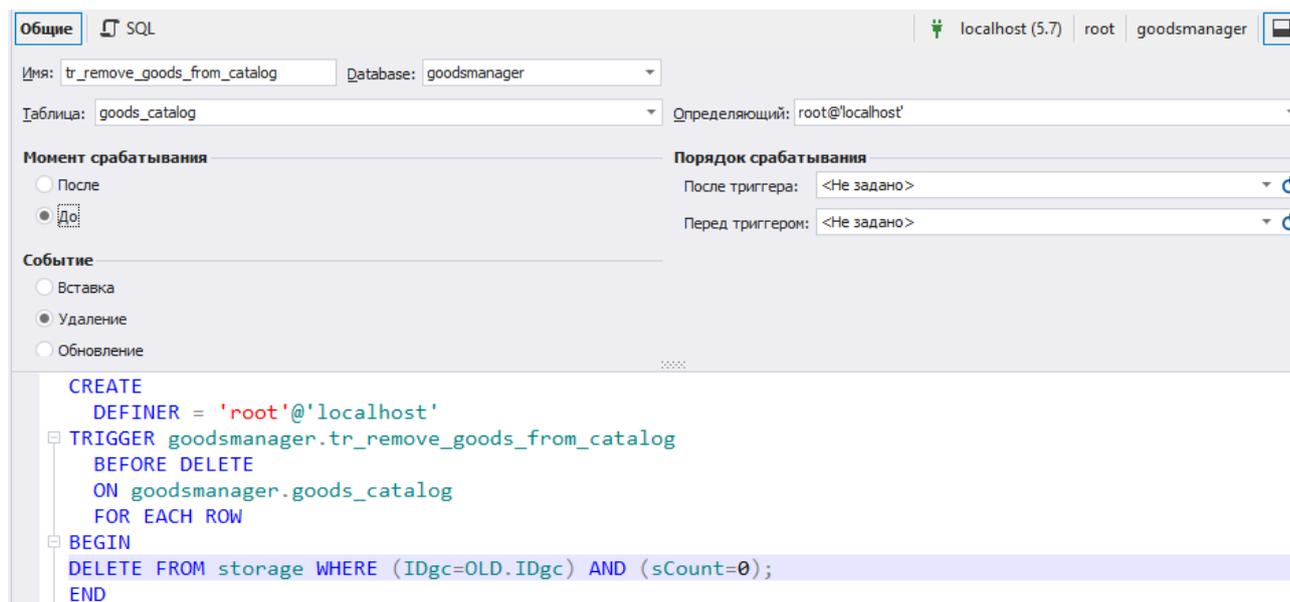


Рис. 4.3. Пример настройки триггера на удаление данных

В данном триггере строка *SQL*-кода:

```
DELETE FROM storage WHERE (IDgc=OLD.IDgc) AND (sCount=0);
```

Осуществляет удаление записи из таблицы **storage** при условии, что значение поля **IDgc** равно значению идентификатора удаляемого товара, а также **sCount** равно **0** (товар количество которого на складе равно **0**). При этом если товар участвовал в каких-либо операциях информацию о товаре нельзя удалить.

Обращаем ваше внимание. Если вы добавляли товары в **goods_catalog** и **не добавляли** их количество в **storage**, то для корректной работы ключевых элементов бизнес логики вам потребуется внести сведения в таблицу **storage**.

4.1 Реализация бизнес-логики с использованием триггеров

В рамках разрабатываемого в ходе лабораторных работ приложения бизнес-логика заключается в контроле складских запасов товара в компьютерном магазине.

Например, при оформлении поставки (поступления товара) необходимо проводить увеличение его количества на складе (изменение значения поля **sCount** таблицы **storage**) на величину, которая указана в поле **ocCount** соответствующей записи таблицы **op_goods_list**. Аналогичная ситуация и в случае оформления продажи – необходимо уменьшить значение количества товаров на складе. Для этого сначала создадим триггер **tr_op_goods_list_add**, который будет срабатывать после добавления записи в таблицу **op_goods_list** (рис. 4.4):

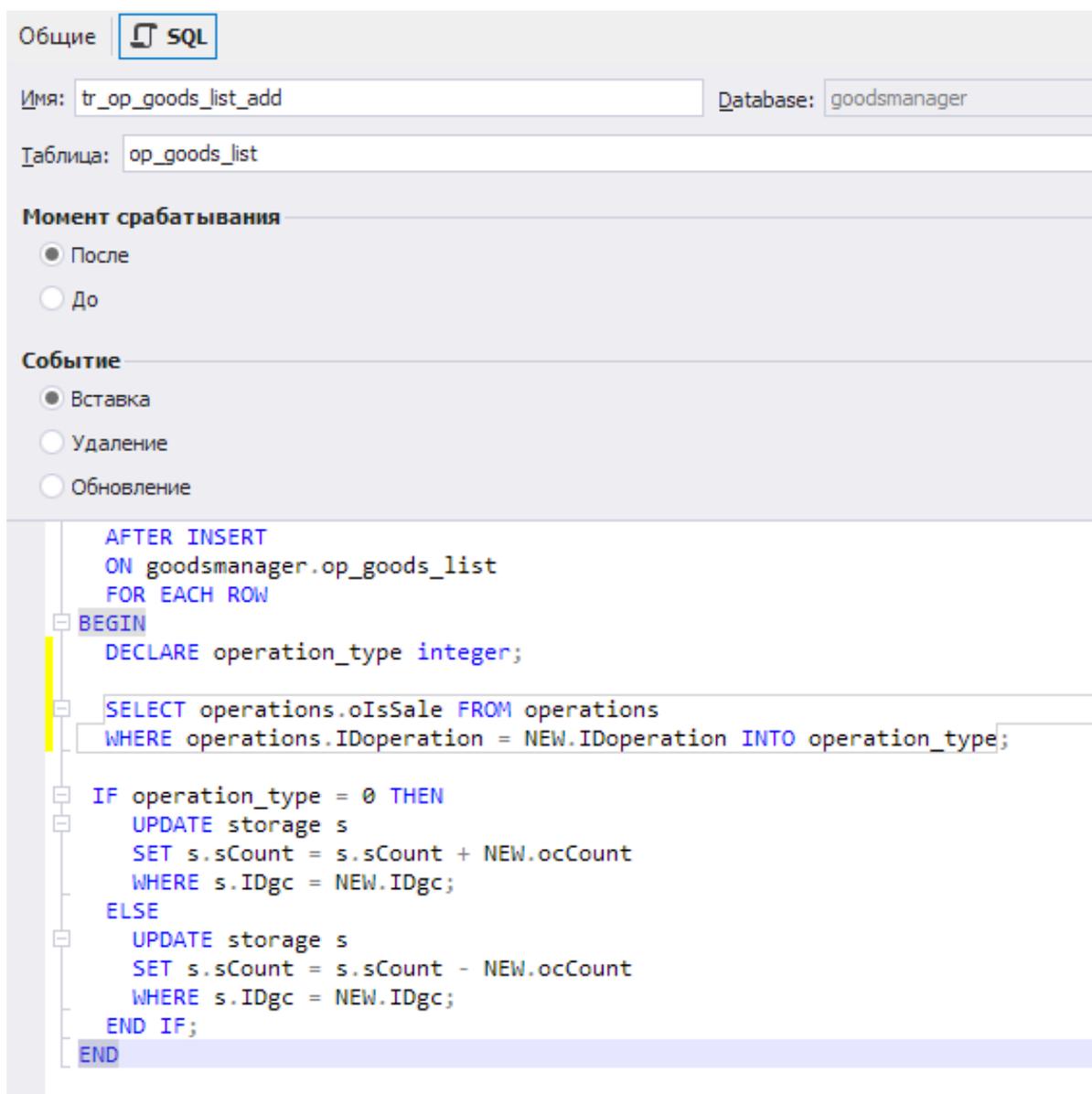


Рис. 4.4. Настройка триггера увеличения/уменьшения количества товаров на складе

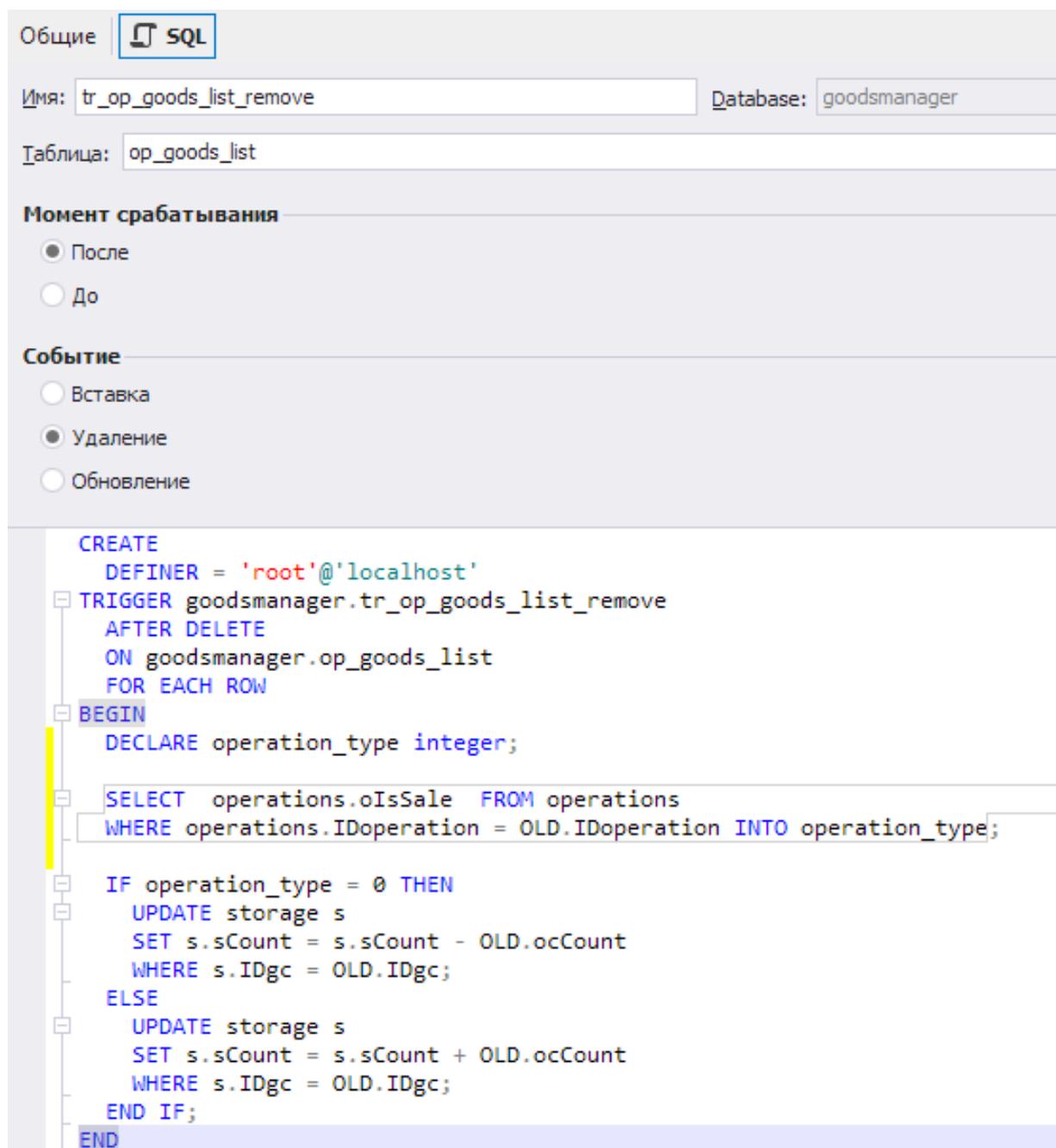
В данном коде конструкция **DECLARE** отвечает за объявление переменной. При помощи нее объявляется переменная **operation_type**, которая будет определять тип выполняемой операции **0** – поставка, **1** – продажа.

Блок кода **SELECT** отвечает за выборку значения типа операции из базы данных.

При помощи конструкции **IF** (условие) **THEN** выполняется изменение количества товаров на складе с применением команды **UPDATE** (имя таблицы) **SET** (поле = значение) с учетом условия для изменения в блоке **WHERE**.

В случае удаления записи из таблицы **op_goods_list** будет использован аналогичный код, но действия в зависимости от типа операции изме-

няться на противоположные. Настройка триггера **tr_op_goods_list_remove** срабатывающего после удаления записи приведена на рис. 4.5:



Общие **SQL**

Имя: Database:

Таблица:

Момент срабатывания

После
 До

Событие

Вставка
 Удаление
 Обновление

```
CREATE
  DEFINER = 'root'@'localhost'
  TRIGGER goodsmanager.tr_op_goods_list_remove
  AFTER DELETE
  ON goodsmanager.op_goods_list
  FOR EACH ROW
  BEGIN
    DECLARE operation_type integer;

    SELECT operations.oIsSale FROM operations
    WHERE operations.IDoperation = OLD.IDoperation INTO operation_type;

    IF operation_type = 0 THEN
      UPDATE storage s
      SET s.sCount = s.sCount - OLD.ocCount
      WHERE s.IDgc = OLD.IDgc;
    ELSE
      UPDATE storage s
      SET s.sCount = s.sCount + OLD.ocCount
      WHERE s.IDgc = OLD.IDgc;
    END IF;
  END
```

Рис. 4.5. Настройка триггера *tr_op_goods_list_remove*

Как можно заметить, для обращения к полям удаляемой записи используется псевдоним **OLD** вместо **NEW**.

Теперь остается реализовать триггер **tr_op_goods_list_edit**, который будет срабатывать после изменения какой-либо записи в таблице **op_goods_list** (рис. 4.6):

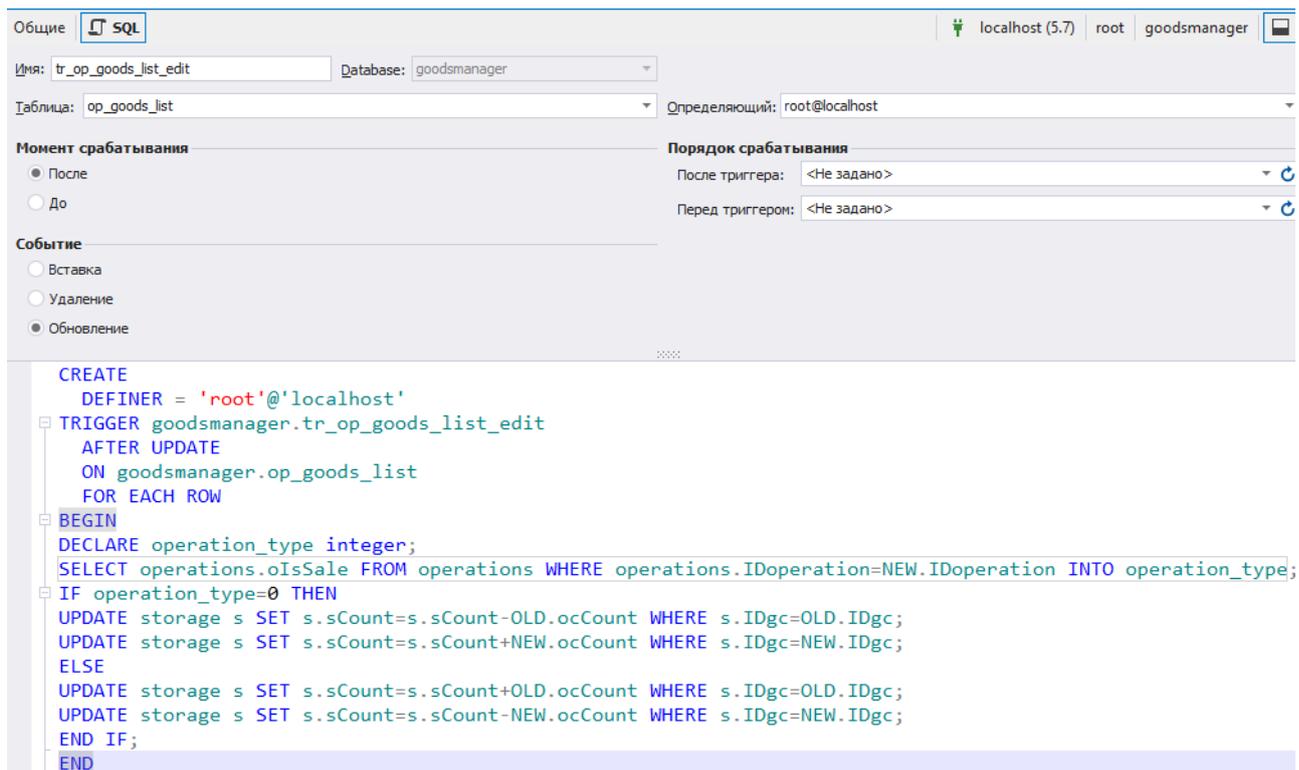


Рис. 4.6. Триггера на редактирование данных в *op_goods_list*

Для операции изменения нам доступны старые и новые значения поля **ocCount**. При модификации данных учитываются изменения нового значения относительно старого.

4.2 Проверка введенного в поле значения

Рассмотрим средства проверки вводимых значений в разрабатываемом приложении, сделайте и откройте копию проекта **лабораторной работы №3**.

Для контроля вводимых данных будем использовать обработчики событий **OnSetText**, **OnValidate**, **OnChange** и **BeforePost**.

Проверить введенное в поле значение на его соответствие некоторым ограничениям или условиям можно в обработчике события **OnValidate**. Оно наступает при изменении значения поля вручную или программно до выполнения метода **Post**, сохраняющего изменения в БД.

Поэтому если полю присвоено неверное значение, то выполнение метода **Post** необходимо предотвратить с помощью метода **Abort** или вызовом исключительной ситуации (**raise Exception.Create**).

Выполним проверку поля **scEmail** (**T_suppliers_catalogscEmail**) таблицы **suppliers_catalog** на наличие символа **@**.

Настройка событий для полей доступна в инспекторе объектов после выбора поля в редакторе полей компонента (рис 4.7):

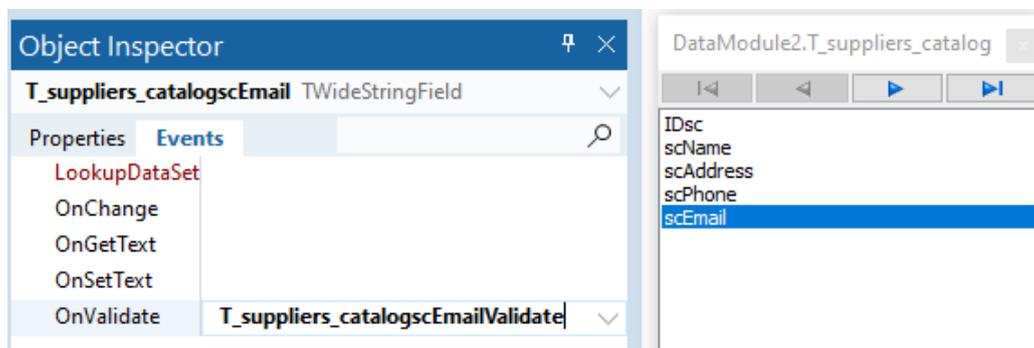


Рис. 4.7. Выбор события *OnValidate* для поля *scEmail*

Пропишите следующий код события:

```
procedure TDataModule2.T_suppliers_catalogscEmailValidate(Sender: TField);
begin
  if not(pos('@', T_suppliers_catalogscEmail.AsString) > 0) then
  begin
    ShowMessage('В поле отсутствует символ @!');
    Abort;
  end;
end;
```

Чтобы работала процедура **ShowMessage** необходимо добавить в блок **uses** раздела **interface** исходного кода заголовочный файл **Dialogs** (в версиях начиная с ХЕ – **Vcl.Dialogs**).

Если символ **@** не содержится в значении, присвоенном полю, то метод **Abort** не позволят выполнить метод **Post** и запись с неверными данными не будет физически сохранена в БД, а НД останется в том состоянии, в котором он находился – в режиме редактирования **dsEdit** или добавления новой записи **dsInsert**.

Для проверки введенного значения может быть использовано и другое событие – **OnSetText**. Подобно событию **OnValidate**, оно возникает при изменении значения поля.

Особенностью события **OnSetText** является то, что в обработчик передается константа-параметр **Text**, содержащая в текстовом виде новое значение, назначенное полю, в то время как действительное значение поля остается без изменения.

Выполним проверку введенного значения в поле **gcCost** таблицы **goods_catalog**, которое должно быть выше некоторой минимальной цены, допустим **5** руб.:

```
procedure TDataModule2.T_goods_cataloggcCostSetText(Sender: TField;
  const Text: string);
var
  Tmp: double;
begin
  Tmp := StrToFloat(Text);

  if Tmp < 5 then
    ShowMessage('Стоимость товара должна быть больше 5 руб.')
  else
    T_goods_cataloggcCost.Value := Tmp;
end;
```

Функция **StrToFloat** конвертирует строку в значение с плавающей точкой.

Событие **OnChange** может быть использовано для тех же целей, что и событие **OnValidate**. Выполните проверку вводимого количества товара в таблице список операций (**op_goods_list**), значение должно быть не менее **1**:

```
procedure TDataModule2.T_op_goods_listocCountChange(Sender: TField);
begin
  if T_op_goods_listocCount.Value < 1 then
    raise Exception.Create
      ('Необходимо указать количество не менее одной единицы товара');
end;
```

При изменении значения записи, события вызываются в следующем порядке:

- **OnSetText**;
- **OnValidate**;
- **OnChange**.

В качестве дополнительного средства проверки данных можно использовать обработчик события **BeforePost**. Это событие вызывается не для конкретного поля таблицы, а в целом для компонента **ADOTable** перед пересылкой любых изменений, выполненных в текущей записи.

Типичным применением процедуры обработчика событий **BeforePost** является реализация правил проверки для всей записи. Если запись не удовлетворяет проверке, то генерируется исключение, и некорректная запись не принимается.

Так в рамках рассматриваемой базы данных будем вести контроль товаров, участвующих в операции. В рамках контроля будем проверять выбран ли товар и в случае операции продажи корректно ли указано количество товара, которое не должно превышать запас на складе. Для оценки модификации будем использовать старые и новые значения полей **OldValue** и **NewValue** соответственно.

```

procedure TDataModule2.T_op_goods_listBeforePost(DataSet: TDataSet);
var
  ErrMsgCnt: string; // для сообщения о недостаточном количестве товара
  mod_val: integer; // для расчета допустимого количества
begin
  ErrMsgCnt := 'Количество продаваемого товара должно быть не меньше имеющего
ся на складе. Доступно для оформления продажи: ';
  if T_op_goods_listIDgc.IsNull or T_op_goods_listocCount.IsNull then
    raise Exception.Create('Необходимо выбрать наименования товара и ' +
      ' указать количество ');
  else if (T_operationsoIsSale.Value = 1) then
    begin
      if T_op_goods_list.State = dsInsert then
        if (T_op_goods_listocCount.Value > Q_v_selector_goodssCount.Value)
          then raise Exception.Create(ErrMsgCnt +
            Q_v_selector_goodssCount.AsString);
      if T_op_goods_list.State = dsEdit then
        begin
          mod_val := Q_v_selector_goodssCount.Value +
            T_op_goods_listocCount.OldValue;
          if (T_op_goods_listIDgc.OldValue <> T_op_goods_listIDgc.NewValue) and
            (T_op_goods_listocCount.Value > Q_v_selector_goodssCount.Value) then
            raise Exception.Create(ErrMsgCnt + Q_v_selector_goodssCount.AsString)
          else
            if (T_op_goods_listIDgc.OldValue = T_op_goods_listIDgc.NewValue) and
              (T_op_goods_listocCount.Value > mod_val) then
                raise Exception.Create(ErrMsgCnt + inttostr(mod_val));
            end;
          end;
        end;
      end;
    end;
end;

```

Помимо этого, в событии **BeforePost**, целесообразна проверка текстовых полей. Так, например, в нашем случае в каталоге товаров (**goods_catalog**) названия товаров должны быть не меньше **двух** символов, обязательно указан тип товара и установлена цена. В случае реализации этих проверок, в обработчик события **BeforePost**, набора данных **T_goods_catalog** запишите следующий код:

```

procedure TDataModule2.T_goods_catalogBeforePost(DataSet: TDataSet);
begin
  if T_goods_catalogIDtg.IsNull or T_goods_cataloggcCost.IsNull or

```

```

(Length(T_goods_cataloggcName.AsString) < 2) then
raise Exception.Create('Наименования товара, тип товара и ' +
'цена не должны иметь пустые значения, а также Наименования ' +
'товара иметь короткую запись (менее трех символов)');
end;

```

4.3 Форматирование отображения значений полей

При отображении данных часто возникает задача модификации отображаемого значения для улучшения восприятия этих данных. Так, например, может потребоваться изменение структуры вывода значения (для даты – указать определенное расположение элементов, для числа – поставить разделители или ограничения на количество знаков после запятой) или определенное форматирование для значения (добавить дополнительную подпись к значениям, взять в кавычки, изменить регистр символов и т. п.). Для выполнения такого форматирования можно воспользоваться событием **OnGetText** со свойствами полей.

Рассмотрим работу с событием **OnGetText**. Пусть поле таблицы базы данных хранится не в том виде, в котором оно должно быть представлено. Тогда его можно отформатировать перед тем, как оно будет показано в визуальных компонентах, работающих с данными, определив алгоритм форматирования в обработчике события **OnGetText** компонента **TField**. В процедуре обработчика присутствуют следующие параметры:

- **Text** – выдает отформатированное значение, показываемое в визуальных компонентах, связанных с таблицами базы данных;
- **DisplayText** – определяет, когда произошло событие **OnGetText**, при показе поля (значение **True**) или при модификации поля (значение **False**).

Например, пусть поле **scName** из набора данных **T_SuppliersCatalog** отображается в компоненте **DBGrid1**. При показе содержимое данного поля необходимо заключить в кавычки (хотя хранится оно без кавычек). Для этого создадим обработчик (рис. 4.8):

```

procedure TDataModule2.T_suppliers_catalogscNameGetText(Sender: TField;
var Text: string; DisplayText: Boolean);
begin
if DisplayText then
Text := ' ' + T_suppliers_catalogscName.AsString + ' '
end;

```

IDsc	scName	scAddress	scPhone	scEmail
1	"ТехноМаркет "	Москва, пр. Ленинский 14	4953222517	info@tehnomark.ru
2	"КрасТрейд "	Красноярск, пр. Мира 31	3912223517	sales@krastrade.com
3	Техноград	Новосибирск, Новосельская 14		

Рис. 4.8. Изменение отображения при помощи события *OnGetText*

Альтернативой для изменения формата отображения является свойство **DisplayFormat**. Однако если для поля был определен обработчик события **OnGetText**, то режимы форматирования, определенные в свойствах **DisplayFormat** и **EditMask** данного поля, игнорируются.

Свойству **DisplayFormat** присваивается текстовая строка, которая задает формат отображения (табл. 4.1).

Таблица 4.1

Спецификаторы форматов для числовых полей

Спецификатор	Описание
0	Число. Если незначащий разряд равен 0, то показывает его
#	Число. Если незначащий разряд равен 0, то не показывает его
.	Десятичная точка. Разделяет целую и дробную часть числа. Принимается во внимание только первая точка, остальные игнорируются
,	Разделитель тысяч. Каждая группа чисел из трех разрядов в целой части отделяется от иных разрядов запятой
;	Разделитель положительного, отрицательного и нулевого значения
E+	Научный формат действительных чисел
"xx" или 'xx'	Символы внутри двойных или одинарных парных кавычек не форматированы и выводятся как есть. Например, число 123.45 с форматом ## "рублей" выведется как '123.5 рублей'

К примеру, пусть значение поля равно **3456.777**, тогда если свойство **DisplayFormat** имеет значение **'#.##'**, то будет показано **3456.78**.

Теперь для **Q_v_goods_catalog**, отображающего информацию о товаре, добавим возможность изменения формата отображения поля **gcCost**. Для этого на соответствующей вкладке разместим компоненты **Edit** с именем **E_dFormat** и **Button** с именем **B_setFormat** (**Применить формат**) (рис 4.9):

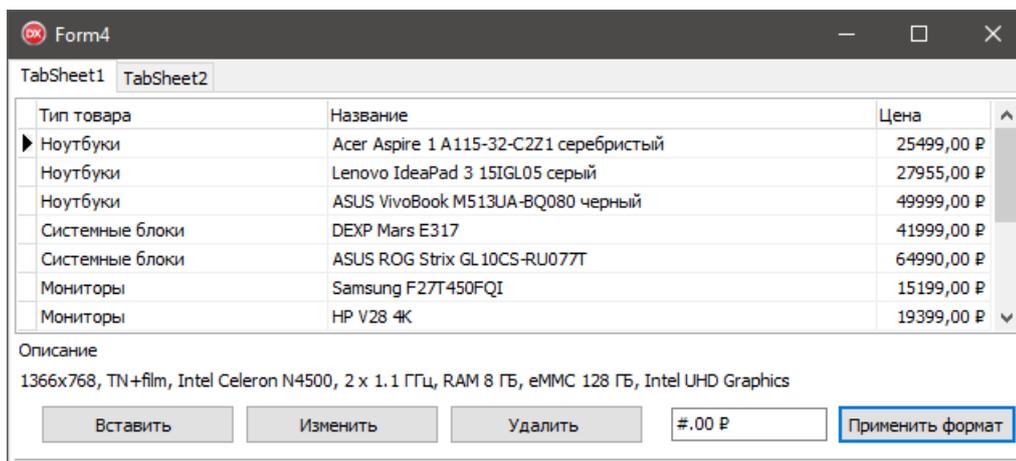


Рис. 4.9. Применение маски «#.00 P» к полю *gcCost*

Для кнопки **B_setFormat** напишем обработчик события **OnClick**:

```
procedure TForm4.B_setFormatClick(Sender: TObject);
begin
    DataModule2.Q_v_goods_cataloggcCost.DisplayFormat := E_dFormat.Text;
end;
```

Для компонента **E_dFormat** зададим свойству **Text** значение «#.00 P» (без кавычек).

4.4 Форматирование полей во время их редактирования

Для контроля корректности вводимых значений применяется свойство **EditMask**.

Ограничения на вводимую информацию накладываются при помощи формата. При этом если введенный символ не удовлетворяет маске, то он не воспринимается. Маска представляет собой строку, состоящую из символов, которые приведены в табл. 4.2:

Таблица 4.2

Символы маски

Символ	Описание
!	Подавляет ведущие пробелы. Если этот символ в данных отсутствует, то хвостовые пробелы подавляются
>	Все следующие символы будут на верхнем регистре, пока не встретится символ <
<	Все следующие символы будут на нижнем регистре, пока не встретится символ >
<>	Регистр не проверяется
\	Следующий за этим знаком символ является литералом, т. е. включается в формируемое значение

L	В этой позиции должен появиться только символ алфавита
l	Аналогично L , но символ в данной позиции может и отсутствовать
A	В этой позиции должен появиться только символ алфавита или цифра
a	Аналогично A , но символ в данной позиции может и отсутствовать
C	В позиции обязателен любой символ
c	Аналогично C , но символ в данной позиции может и отсутствовать
0	В данной позиции обязателен цифровой символ
9	В данной позиции может находиться цифра или она будет пустой;
#	В этой позиции должен появляться только цифровой символ, плюс или минус или не появляться никакой
:	Разделитель часов, минут и секунд для значения типа времени. Если в данной национальной кодировке для указанных целей используется иной символ, то он используется вместо символа :
/	Разделитель месяца, дня и года в датах. Если в данной национальной кодировке для указанных целей используется иной символ, то он используется вместо символа /
;	Разделитель частей маски
_	Замениватель пробела в маске

К примеру имеется маска '!|(999\|) 000\|-00\|-00;0;_ ' и введено значение '0952223344', то во время ввода это значение представлялось бы на экране как '(095) 222-33-44', а сохранилось бы в БД как '0952223344'.

Настроим маску для поля **scPhone** таблицы **suppliers_catalog**. Выберем нужное поле в редакторе полей, после чего нажмем на кнопку с многоточием у свойства **EditMask** и зададим маску для редактирования телефона поставщика в федеральном формате (рис. 4.10):

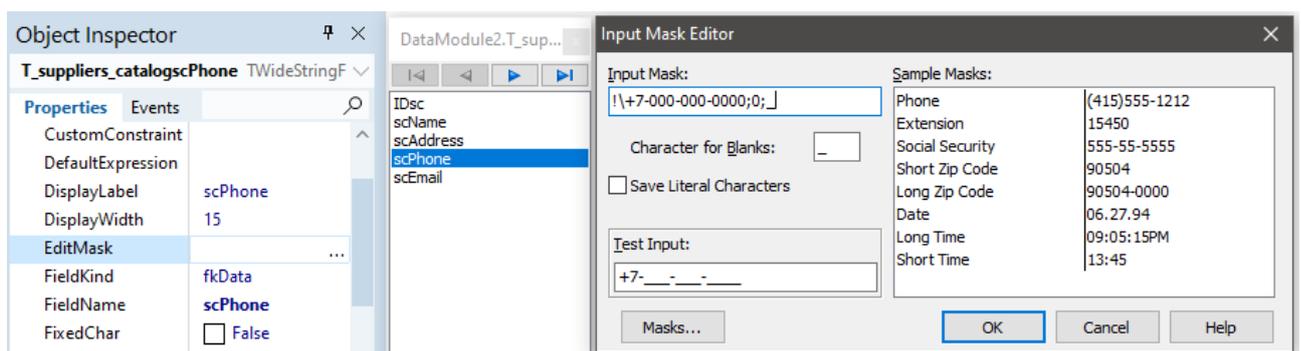


Рис. 4.10. Настройка маски для поля *scPhone*

IDsc	scName	scAddress	scPhone	scEmail
1	"ТехноМаркет "	Москва, пр. Ленинский 14	+7-495-322-2517	info@tehnomark.ru
2	"КрасТрейд "	Красноярск, пр. Мира 31	+7-391-222-3517	sales@krastrade.com
3	"Техноград "	Новосибирск, Новосельская 14	+7-382- -	

Рис. 4.11. Пример работы маски поля *scPhone*

4.5 Фильтрация данных на основе SQL запроса

Важным механизмом для работы с данными является фильтрация/поиск информации.

Реализуем механизм фильтрации на вкладке **Состояние склада** (*Form1*). Сначала разместим панель где будут расположены необходимые элементы (рис 4.12). У панели зададим свойство **Align** в значение **alTop** (чтобы панель была всегда вверху). На панели разместим один компонент **Label** для поясняющего текста и один компонент **Edit**.

Компоненту **Edit** дадим имя **E_Filter** и очистим его свойство **Text**, а свойству **TextHint** зададим значение: **Введите фрагмент названия товара или его типа**. Свойство **TextHint** отвечает за вывод подсказки пользователю.

Тип товара	Название	Количество
Мониторы	HP V28 4K	7
Мониторы	Samsung F27T450FQI	6
Ноутбуки	Acer Aspire 1 A115-32-C2Z1 серебристый	4
Ноутбуки	ASUS VivoBook M513UA-BQ080 черн	6
Ноутбуки	Lenovo IdeaPad 3 15IGL05 серый	7
Системные блоки	ASUS ROG Strix GL10CS-RU077T	5
Системные блоки	DEXP Mars E317	4

Рис. 4.12. Компоненты для фильтрации данных.

Для того чтобы фильтр применялся автоматически при вводе данных, напишите обработчик события **OnChange** для **E_Filter**:

```
procedure TForm1.E_FilterChange(Sender: TObject);
var
  baseSQL, filter_s: string;
begin
```

```

DataModule2.Q_v_storage.DisableControls;
DataModule2.Q_v_storage.Active := false;

baseSQL := 'SELECT * FROM view_storage ';

filter_s := '';
if length(E_Filter.Text) > 0 then
  filter_s := 'WHERE tgName LIKE ' + QuotedStr('%' + E_Filter.Text + '%') +
    ' OR gcName LIKE ' + QuotedStr('%' + E_Filter.Text + '%');

DataModule2.Q_v_storage.SQL.Text := baseSQL + filter_s;

DataModule2.Q_v_storage.Active := true;
DataModule2.Q_v_storage.EnableControls;
end;

```

Переменная **baseSQL** содержит основной запрос на выбор данных из представления **view_storage**, а переменная **filter_s** формирует строку с условием для отбора необходимых записей.

Запустив приложение и введя фрагмент строки, будут показаны только те товары у которых в названии встречается данный текст (рис 4.13):

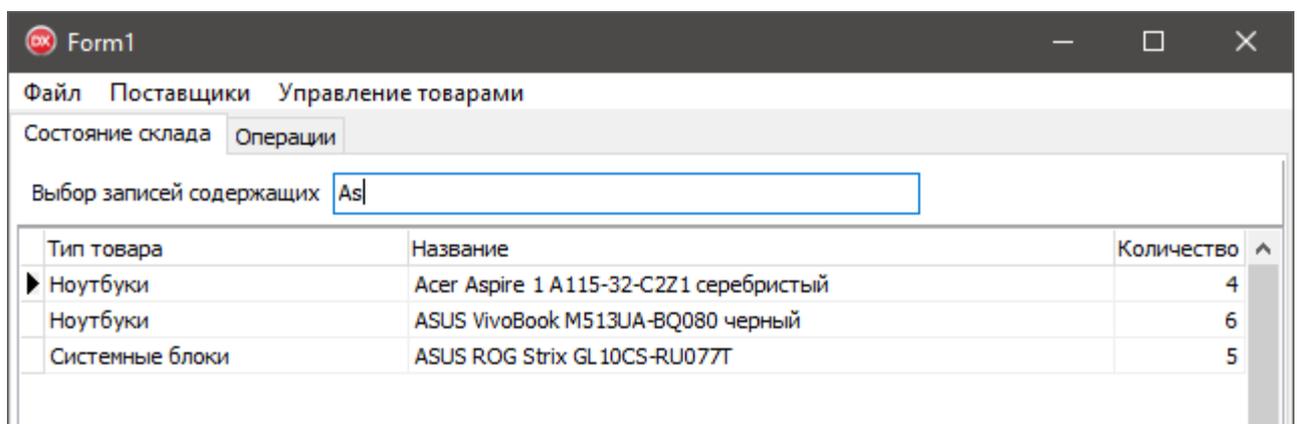


Рис. 4.13. Пример работы фильтра

Рассмотрим фильтрацию на основе SQL запроса применительно к отбору товаров по цене.

Для этого на форме **Form4** увеличим высоту панели **InsertEditDeletePanel** и разместим на неё **GroupBox** (имя **GroupBox1**) для организации области фильтра (рис. 4.14 – 1). В области **GroupBox** разместим компоненты **ComboBox** (зададим имя **ComboCostCondition**) (рис. 4.14 – 2), **SpinEdit** (зададим имя **SpinCostValue**) (рис. 4.14 – 3), **CheckBox** (зададим имя **ChkCostFilter**) (рис. 4.14 – 4) и два компонента **Label** для вывода поясняющего текста.

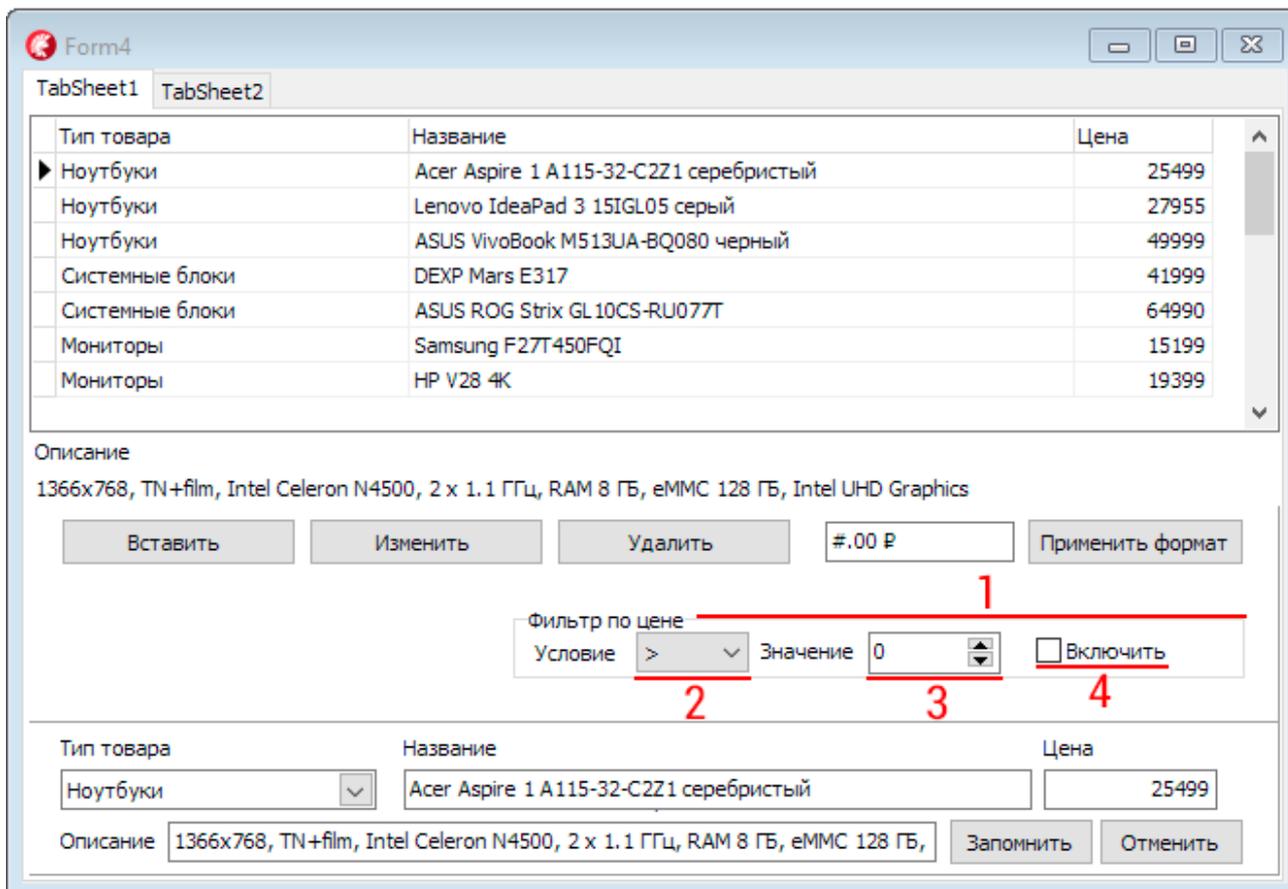


Рис. 4.14. Компоненты для фильтрации данных по цене

Для начала настроим компонент **ComboCostCondition**, в свойстве **Style** установим значение **csDropDownList**. Через редактор (рис 4.15) в свойстве **Items** зададим операторы сравнения **>**, **<**, **=**, **<>** (каждый на новой строке). После определения списка в свойстве **ItemIndex** установим значение **0**.

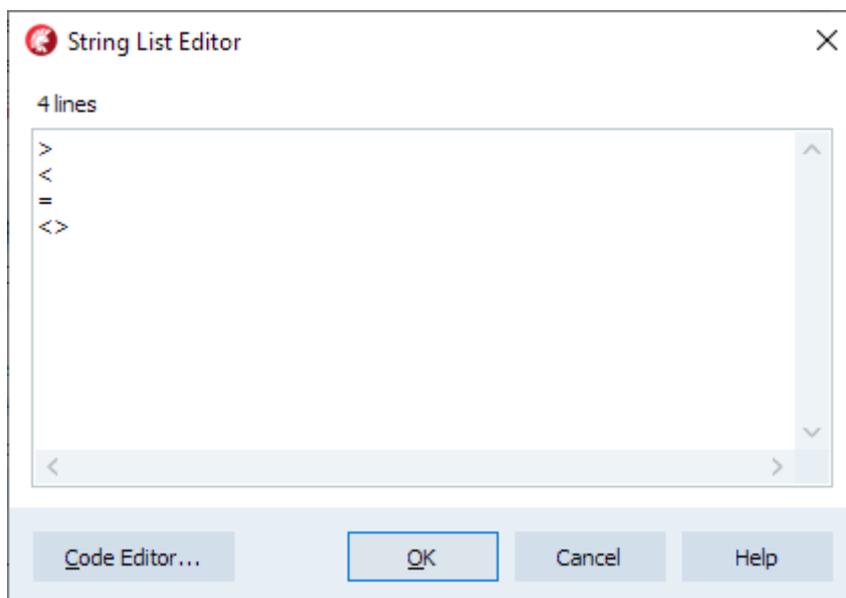


Рис. 4.15. Настройка списка условий фильтрации

Теперь напишем код фильтрации. Фильтр будет формироваться и применяться для компонента **ChkCostFilter** в обработчике события **OnClick**:

```
procedure TForm4.ChkCostFilterClick(Sender: TObject);
var
  baseSQL, filter_s: string;
begin
  DataModule2.Q_v_goods_catalog.Active := False;
  baseSQL := 'SELECT * FROM view_goods_catalog';
  DataModule2.Q_v_goods_catalog.SQL.Text := baseSQL;
  filter_s := '';

  if ChkCostFilter.Checked then
  begin
    // Формирования условия выборки
    filter_s := ' WHERE gcCost ' + ComboCostCondition.Text + ' ' +
      IntToStr(SpinCostValue.Value);

    DataModule2.Q_v_goods_catalog.SQL.Text := baseSQL + filter_s;
  end;

  DataModule2.Q_v_goods_catalog.Active := True;
end;
```

Для того чтобы фильтр автоматически обновлялся при изменении значения цены в компоненте **SpinCostValue**, а также выборе условия в **ComboCostCondition** необходимо имитировать наступление события **OnClick** компонента **ChkCostFilter**. Для этого пропишем код обработчиков события **OnChange** у компонентов **SpinCostValue** и **ComboCostCondition**:

```
procedure TForm4.ComboCostConditionChange(Sender: TObject);
begin
  ChkCostFilterClick(nil);
end;

procedure TForm4.SpinCostValueChange(Sender: TObject);
begin
  ChkCostFilterClick(nil);
end;
```

4.6 Свойство Filter ADO-компонентов

Теперь реализуем механизм фильтрации данных операций по диапазону дат.

Для этого на панель вкладки **Операции** главной формы поместим **GroupBox** на который разместим один компонент **CheckBox**, два компонента **DateTimePicker** у которых будут имена **DateTimePicker1** и **DateTimePicker2** и компоненты **Label** (необходимы для поясняющего текста). Компонент **DateTimePicker1** будет отвечать за начальное время интервала, а **DateTimePicker2** – за конечное (рис 4.16):

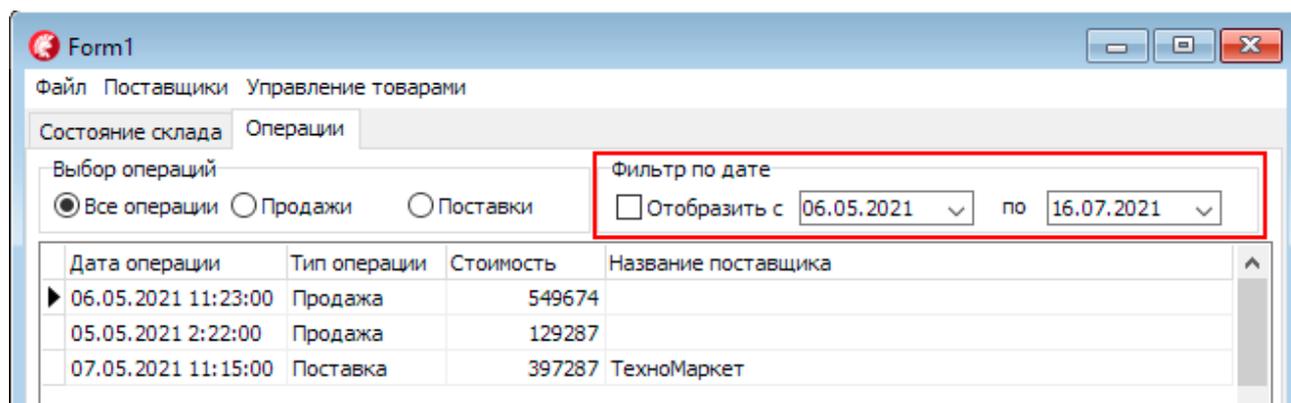


Рис. 4.16. Фрагмент формы с компонентами для организации фильтра по дате

Для компонентов **DateTimePicker1** и **DateTimePicker2** в свойстве **Time** можно выставить значения **0:00:00** и **23:59:59** соответственно. Это необходимо, когда фильтрация будет осуществляться по полю **DateTime** (дата и время) с задействованием времени. В нашем примере будет происходить преобразование формата, которое отсекает время, поэтому данное действие не обязательно.

Активация фильтра будет осуществляться с применением события **OnClick** компонента **CheckBox1**. Обработчик события будет иметь следующим код:

```
procedure TForm1.CheckBox1Click(Sender: TObject);
begin
    // отключение фильтра
    DataModule2.Q_v_goods_operations.Filtered := false;
    // Формирование строки фильтрации
    DataModule2.Q_v_goods_operations.Filter := 'oDateTime >= ' +
        QuotedStr(FormatDateTime('YYYY-MM-DD', DateTimePicker1.Date)) +
        ' and oDateTime <= ' + QuotedStr(FormatDateTime('YYYY-MM-DD',
        DateTimePicker2.Date));
    // задание параметра активности фильтра как у состояния CheckBox1
    DataModule2.Q_v_goods_operations.Filtered := CheckBox1.Checked;
end;
```

В коде применяется приведение формата даты при помощи функции **FormatDateTime** к конструкции **'YYYY-MM-DD'**, которая является стандартной для представления даты во внутреннем формате *MySQL*.

Поскольку в режиме активной фильтрации пользователь может изменить дату, то необходимо прописать код обрабатывающий данную ситуацию в компонентах **DateTimePicker** используя событие **OnChange**:

```
procedure TForm1.DateTimePicker1Change(Sender: TObject);
begin
    CheckBox1Click(nil);
end;

procedure TForm1.DateTimePicker2Change(Sender: TObject);
begin
    CheckBox1Click(nil);
end;
```

В данных обработчиках происходит имитация активации события **OnClick** для компонента **CheckBox1** и тем самым фильтр перестраивается с учетом измененных данных в компонентах **DateTimePicker**.

КОНТРОЛЬНЫЕ ВОПРОСЫ

Дайте характеристику триггерам, каково их назначение?

Что из себя представляют псевдонимы **OLD** и **NEW**?

В какой момент срабатывают триггеры?

Опишите назначение конструкции **DECLARE**?

Для чего применяется функция **StrToFloat**, какие подобные функции существуют в **Delphi**?

Чем событие **BeforePost** отличается от **OnSetText**, **OnValidate**, **OnChange**?

Охарактеризуйте свойство **DisplayFormat**, для чего оно применяется?

Влияет ли формат вводимых данных в **EditMask** на сохраняемые значения в БД?

За что отвечает свойство **TextHint**?

Опишите механизм работы *SQL*-конструкции **LIKE**, каков формат применяемых масок?

За что отвечает *SQL*-конструкция **WHERE**?

Чем компонент **SpinEdit** отличается от **Edit**?

Для чего применяются функции **FormatDateTime** и **QuotedStr**?

Какой формат хранения даты в *MySQL*?