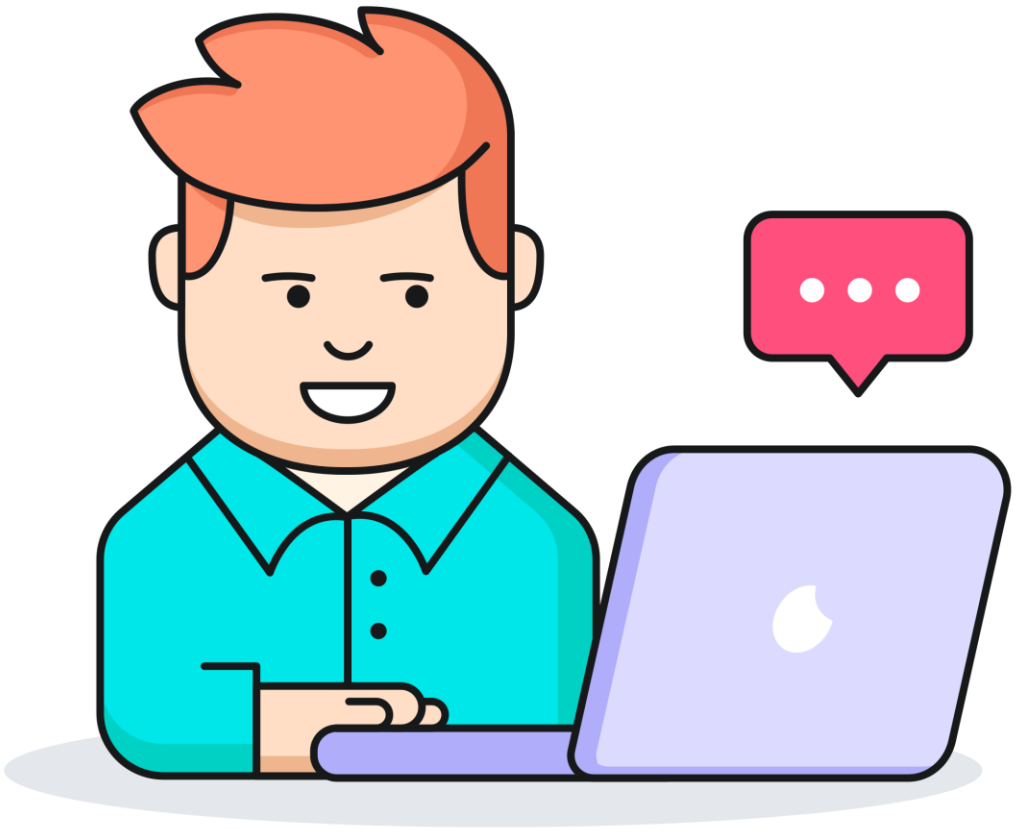
Севастопольский государственный университет
Кафедра «Информационные системы»

Управление данными

курс лекций

лектор:
ст. преподаватель кафедры ИС Абрамович А.Ю.



Лекция 2

ЯЗЫКИ ЗАПРОСОВ СОВРЕМЕННЫХ СУБД.

Представления. Генераторы и триггеры.

ПРЕДСТАВЛЕНИЯ

просмотры (VIEW), **представляют собой временные, производные таблицы и являются объектами базы данных, информация в которых не хранится постоянно, как в базовых таблицах, а формируется динамически при обращении к ним.**

Содержимое представлений выбирается из других таблиц с помощью выполнения запроса, причем при изменении значений в таблицах данные в представлении автоматически меняются.

У СУБД ЕСТЬ ДВЕ ВОЗМОЖНОСТИ РЕАЛИЗАЦИИ ПРЕДСТАВЛЕНИЙ.

Если его *определение простое*, то система формирует *каждую запись представления по мере необходимости*, постепенно считывая исходные данные из базовых таблиц.

В случае *сложного определения* СУБД приходится *сначала выполнить такую операцию, как материализация представления*, т.е. сохранить информацию, из которой состоит представление, во временной таблице. Затем *система приступает к выполнению пользовательской команды и формированию ее результатов*, после чего временная таблица удаляется.

ПРЕДСТАВЛЕНИЯ

```
CREATE VIEW <имя_представления>
[(<имя_столбца>,...)] [WITH ENCRYPTION]
AS <запрос> [WITH CHECK OPTION]
```

```
CREATE VIEW studs (fname, lname, number) AS
SELECT fname, lname, number from students, groups
WHERE students.group_id = groups.id
```

ОСОБЕННОСТИ ПРЕДСТАВЛЕНИЙ:

- основывается только *на одной* базовой таблице;
- содержит *первичный ключ* этой таблицы;
- не содержит DISTINCT в своем определении;
- не использует GROUP BY или HAVING в своем определении;
- по возможности *не применяет в своем определении подзапросы*;
- *не использует константы* или выражения значений среди выбранных полей вывода;
- в просмотр должен быть включен *каждый столбец таблицы*, имеющий атрибут *NOT NULL*;
- оператор SELECT просмотра *не использует агрегирующие (итоговые) функции, соединения таблиц, хранимые процедуры и функции*, определенные пользователем;
- основывается на одиночном запросе, поэтому объединение *UNION не разрешено*.

Пример: показать в представлении клиентов из Севастополя.

Создание представления:

```
CREATE  VIEW view1 AS
SELECT КодКлиента, Фамилия, ГородКлиента
FROM Клиент
WHERE ГородКлиента='Севастополь'
```

Выборка данных из представления:

```
SELECT * FROM view1
```

Обращение к представлению осуществляется с помощью оператора SELECT как к обычной таблице.

Выполним команду:

```
INSERT INTO view1 VALUES (12,'Петров', 'Симферополь')
```

```
ALTER VIEW view1 AS
SELECT КодКлиента, Фамилия,ГородКлиента
FROM Клиент
WHERE ГородКлиента='Москва' WITH CHECK OPTION
```

***Таким образом, представление** может изменяться командами модификации, **но фактически** модификация воздействует не на само представление, а на базовую таблицу.*

ГЕНЕРАТОРЫ (ПОСЛЕДОВАТЕЛЬНОСТЬ)

объект базы данных, предназначенный для получения уникального числового значения.

Генераторы используют для создания автоинкрементных полей. Для каждого такого поля придется создавать собственный генератор. ***Генератор гарантирует, что значение этого поля всегда будет уникальным.***

```
CREATE {SEQUENCE | GENERATOR} <name_GEN>;
```

Операторы *CREATE SEQUENCE* **и** CREATE GENERATOR **являются синонимами –** *оба создают новую последовательность.* **Можно использовать любое из них, но рекомендуется использовать CREATE SEQUENCE, если важно соответствие стандарту.**

В момент создания генератора ему устанавливается значение равное 0.

Значение генератора изменяется также при обращении к функции *GEN_ID*, **где в качестве параметра указывается имя последовательности и значение приращения.**

```
GEN_ID(name_GEN, step);
```

name_GEN – имя генератора; ***step*** – шаг, на который требуется увеличить значение.

ГЕНЕРАТОРЫ (ПОСЛЕДОВАТЕЛЬНОСТЬ)

Следует быть крайне аккуратным при таких манипуляциях в базе данных, они могут привести к потере целостности данных. Если step равен 0, функция не будет ничего делать со значением генератора и вернет его текущее значение.

Оператор **NEXT VALUE FOR** возвращает следующее значение в последовательности.

```
NEXT VALUE FOR <name_GEN>
```

NEXT VALUE FOR **не поддерживает значение приращения, отличное от 1.**
Если требуется другое значение шага, то используется старая функция GEN_ID.

Оператор **ALTER SEQUENCE** устанавливает значение последовательности или генератора в заданное значение.

```
ALTER SEQUENCE <name_GEN> RESTART WITH <new_val>
```

Оператор **SET GENERATOR** устанавливает значение последовательности или генератора в заданное значение.

```
SET GENERATOR <name_GEN> TO <new_val>
```

*оператор считается устаревшим и оставлен ради обратной совместимости

Для просмотра текущего значения генератора, необходимо выполнить команду:

```
SELECT GEN_ID(<name_GEN>,0) FROM RDB$DATABASE;
```

ГЕНЕРАТОРЫ (ПОСЛЕДОВАТЕЛЬНОСТЬ)

RDB$DATABASE – это системная таблица, которая хранит основные данные о базе данных, она всегда существует во всех базах данных Firebird и всегда содержит только одну строку.

RDB$GENERATORS – это системная таблица, которая хранит сведения о генераторах (последовательностях).

Основное простое правило по переустановке значений генераторов в работающей базе данных –

не делать этого.

Удаление последовательности (генератора):

```
DROP {SEQUENCE | GENERATOR} <name_GEN>
```

Операторы ***DROP SEQUENCE и DROP GENERATOR эквивалентны***: оба удаляют существующую последовательность (генератор).

Удалить генератор может либо его владелец либо SYSDBA при условии что его нет в зависимостях других объектов.

ТРИГГЕРЫ

подпрограммы, которые всегда выполняются автоматически на стороне сервера, в ответ на изменение данных в таблицах БД. Это методы, с помощью которых разработчик может обеспечить целостность БД.

Триггер активизируется при попытке изменения данных в таблице, для которой определен. SQL выполняет эту процедуру при операциях добавления, обновления и удаления **(INSERT, UPDATE, DELETE)** в данной таблице.

Наиболее общее применение триггера – поддержка целостности в базах данных. Триггеры незначительно влияют на производительность сервера и часто используются для усиления предложений, выполняющих многокаскадные операции в таблицах и строках.

Триггер может выполняться в двух фазах изменения данных: до **(before)** какого-то события, или после **(after)** него.

ТРИГГЕРЫ

```
CREATE TRIGGER <trigname> FOR {<table_name> | <view_name>}
  [ACTIVE | INACTIVE]
  {BEFORE | AFTER} {DELETE | INSERT | UPDATE}
  [POSITION <number>]
AS
  [DECLARE [VARIABLE] <variable datatype>;]
BEGIN
   <compound_statement>[<compound_statement>]
END
<compound_statement> = {<block> | statement;}
```

ЗАГОЛОВОК ТРИГГЕРА

- имя триггера, уникальное по БД;
- имя таблицы, с которой ассоциируется триггер;
- момент, когда триггер должен вызываться.

ТЕЛО ТРИГГЕРА

состоит из опционального списка локальных переменных и операторов.

ACTIVE | INACTIVE – необязательный параметр определяет, будет триггер запускаться в ответ на событие, или не будет.

{BEFORE | AFTER} {DELETE | INSERT | UPDATE} – два обязательных параметра, комбинация которых может запрограммировать триггер на шесть различных вариантов реагирования на события

POSITION <number> – необязательный параметр, который определяет очередность запуска, если для той же таблицы и для того же события имеется другой триггер

AS – команда, начинающая тело триггера.

ТРИГГЕРЫ

Все, что следует за частью AS оператора CREATE TRIGGER составляет тело триггера. Тело триггера состоит из опционального списка локальных переменных, за которым идет блок операторов. Блок состоит из набора операторов на языке хранимых процедур и триггеров, заключенных в операторные скобки.

Хранимые процедуры представляют собой набор команд, состоящий из одного или нескольких операторов SQL или функций и сохраняемый в базе данных в откомпилированном виде.

Язык хранимых процедур и триггеров Interbase (ЯХПТ) является полным языком для написания хранимых процедур и триггеров. Язык включает в себя:

- операторы манипулирования данными SQL (INSERT, UPDATE, DELETE) а также оператор SELECT;
- операторы и выражения SQL, включая функции, определяемые пользователем (UDF);
- расширения SQL, включая, оператор присваивания, операторы управления последовательностью выполнения, локальные переменные, операторы отсылки сообщений, обработка исключительных ситуаций, операторы обработки ошибок.

ПЕРЕМЕННЫЕ NEW И OLD

Эти переменные объявлять не нужно, они уже присутствуют в каждом триггере**. Соответственно, переменные хранят старое и новое значения какого-либо поля.**

NEW.<имя_поля>

OLD.<имя_поля>

Значения NEW могут быть использованы в событиях INSERT и UPDATE, при удалении записи NEW имеет значение NULL.
Значения OLD доступны в событиях UPDATE и DELETE, а при вставке новой записи OLD имеет значение NULL.

Пример: создадим триггер, который срабатывает перед вставкой новой записи и проверяет входящее целое число. Если оно отрицательно, триггер изменяет его на ноль.

```
SET TERM ^;
CREATE TRIGGER NotOtric FOR Table_Cel
ACTIVE BEFORE INSERT
AS
BEGIN
  IF (NEW.Dlinnoe < 0) THEN NEW.Dlinnoe = 0;
END^
SET TERM ;^
```

Если затем выполнить следующие команды:

```
INSERT INTO Table_cel(Dlinnoe) VALUES(5);
INSERT INTO Table_cel(Dlinnoe) VALUES(-10);
SELECT * FROM Table_cel;
```

то в таблице появились две новые строки: в первом случае значение 5 сохранилось без изменения, а во второй записи триггер изменил значение –10 на 0.

РЕАЛИЗАЦИЯ АВТОИНКРЕМЕНТНЫХ КЛЮЧЕВЫХ ПОЛЕЙ

Для создания поля, значение которого автоматически увеличивается на единицу, нужно сделать несколько действий:

1. **создать генератор для ключевого поля.** Ключевое поле должно иметь тип INTEGER, быть NOT NULL и объявлено как PRIMARY KEY. Собственно, генератор можно использовать для любого автоинкрементного поля, не обязательно ключевого, но чаще всего генераторы используют именно для ключевых полей;

2. **присвоить генератору значение 0** (или иное, если таблица перенесена из другой БД, и уже содержит записи);

3. **создать триггер BEFORE INSERT**, увеличивающий это значение на 1.

РЕАЛИЗАЦИЯ АВТОИНКРЕМЕНТНЫХ КЛЮЧЕВЫХ ПОЛЕЙ

```
/*Создаем генератор*/
CREATE GENERATOR Gen_Tovar;
/*Присваиваем генератору начальное значение*/
SET GENERATOR Gen_Tovar TO 0;
/*Создаем триггер*/
SET TERM ^;
CREATE TRIGGER Tr_Tovar FOR Tovar
ACTIVE BEFORE INSERT
AS
BEGIN
  IF (NEW.ID IS NULL) THEN
    NEW.ID = GEN_ID(Gen_Tovar, 1);
END^
SET TERM ;^
/* Завершаем транзакцию: */
COMMIT;
```

Операторы из данного примера создают автоматическое увеличение значения поля на 1.

Таким образом, вставка первой же записи установит значение 1. Следующая запись будет 2 и так далее.

Все это реализуется в пределах транзакции, то есть даже если множество пользователей вносит изменения в таблицу, значения генератора всегда будут уникальны

РЕАЛИЗАЦИЯ АВТОИНКРЕМЕНТНЫХ КЛЮЧЕВЫХ ПОЛЕЙ

Теперь можно проверить работу автоинкремента. После выполнения следующего запроса:

```
INSERT INTO Tovar(Nazvanie, Stoimost) VALUES('Сахар', 10.50);
INSERT INTO Tovar(Nazvanie, Stoimost) VALUES('Крупа', 8.20);
```

в таблице появятся две записи, а поле ID будет автоматически увеличиваться на 1. Причем значения вносились только в поля Nazvanie и Stoimost. Значения для поля ID генерировались триггером автоматически.

ПРИМЕРЫ РЕАЛИЗАЦИИ ТРИГГЕРОВ

Необходимо создать триггер, который не позволяет добавлять более пяти поставщиков из одного города.

```
CREATE EXCEPTION s_c 'Sellers amount > 5';

SET TERM !! ;
CREATE TRIGGER TEMP FOR S
 ACTIVE BEFORE INSERT
 AS
  DECLARE VARIABLE s_amount INTEGER;
  BEGIN
    s_amount = (SELECT COUNT(*) FROM S WHERE city = NEW.city);
     IF (s_amount = 5) THEN EXCEPTION s_c;
  END!!

SET TERM ; !!
```

ПРИМЕРЫ РЕАЛИЗАЦИИ ТРИГГЕРОВ

Сделать для таблицы S поле, значение которого автоматически увеличивается на единицу.

```
CREATE GENERATOR Gen_S_ID;
ALTER SEQUENCE Gen_S_ID RESTERT WITH 0;

SET TERM !! ;
CREATE TRIGGER auto_id_s FOR S
ACTIVE BEFORE INSERT
AS
BEGIN
  IF (NEW.ID IS NULL) THEN
    NEW.ID = GEN_ID(Gen_S_ID, 1);
END!!
SET TERM ; !!

COMMIT;
```