



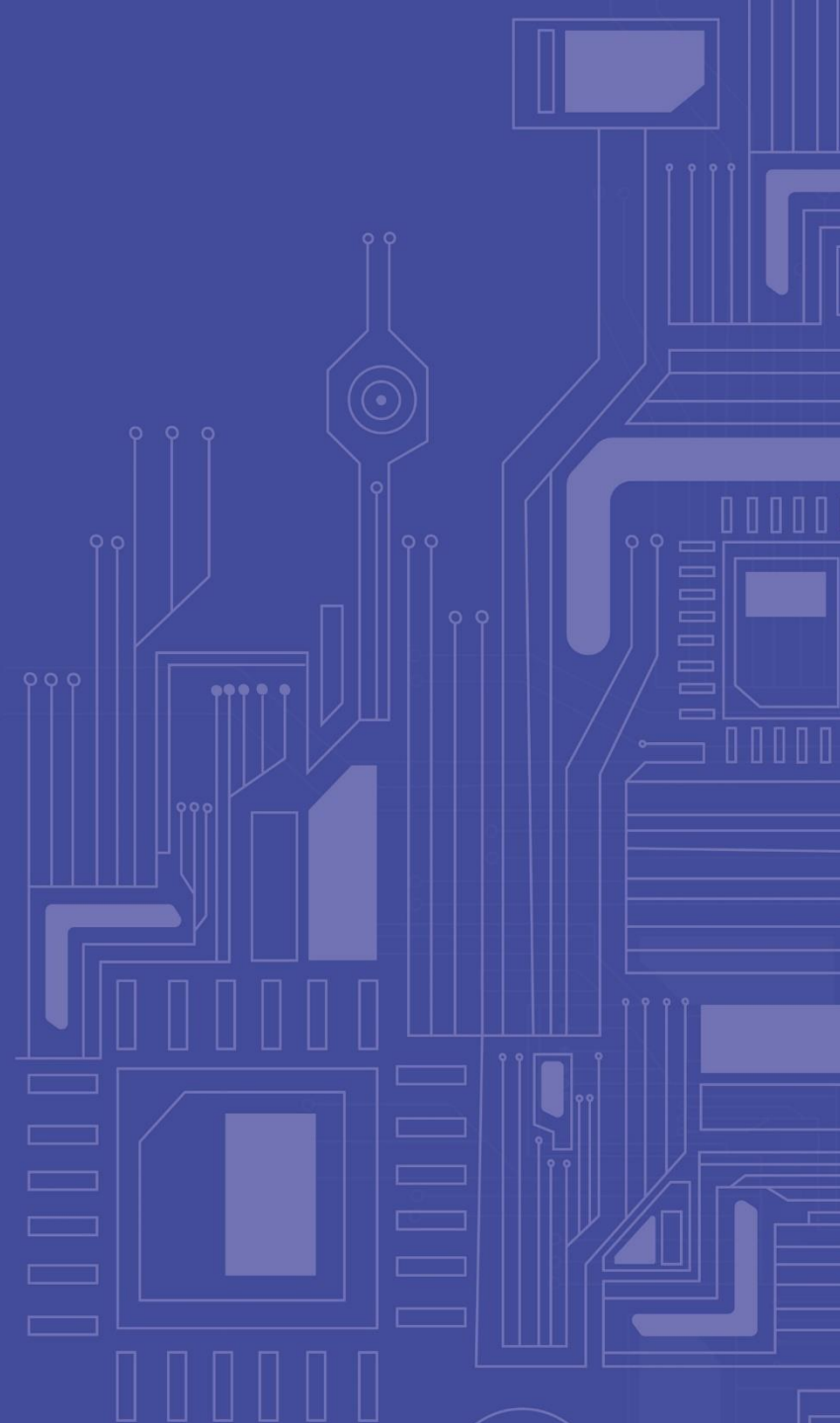
МИНОБРНАУКИ  
РОССИИ



Передовые  
инженерные  
школы

# СИСТЕМЫ УПРАВЛЕНИЯ БАЗАМИ ДАННЫХ

*Лекция 10*



- Изменения определений таблиц
  - Общие ограничения целостности
    - Скалярные выражения
  - Общая структура оператора выборки
  - Ссылки на таблицы раздела FROM

# ИЗМЕНЕНИЯ ОПРЕДЕЛЕНИЙ ТАБЛИЦ



## Изменение определения базовой таблицы

```
base_table_alteration ::= ALTER TABLE base_table_name  
    column_alteration_action  
    | base_table_constraint_alteration_action
```

Как видно из этого синтаксического правила, при выполнении одного оператора ALTER TABLE может быть выполнено либо действие по изменению определения столбца, либо действие по изменению определения табличного ограничения целостности.

## Добавление, изменение или удаление определения столбца

```
column_alteration_action ::=  
    ADD [ COLUMN ] column_definition  
    | ALTER [ COLUMN ] column_name  
        { SET default_definition | DROP DEFAULT }  
    | DROP [ COLUMN ] column_name  
        { RESTRICT | CASCADE }
```

с использованием оператора ALTER TABLE можно добавлять к определению таблицы определение нового столбца (действие ADD) и изменять или отменять определение существующего столбца (действия ALTER и DROP соответственно).

Смысл действия ADD COLUMN почти полностью совпадает со смыслом раздела определения столбца в операторе CREATE TABLE. Указывается имя нового столбца, его тип данных или домен. Могут определяться значение по умолчанию и ограничения целостности. Столбец, определяемый в действии ADD, обязательно должен иметь значение по умолчанию, поскольку он добавляется к существующей таблице.

**ALTER COLUMN** В действии ALTER COLUMN можно изменить (SET default\_definition) или отменить определение значения по умолчанию для существующего столбца. Изменение значения столбца по умолчанию не оказывает влияния на состояние существующих строк таблицы (даже если в некоторых из них хранится предыдущее значение столбца по умолчанию). Если столбец определен на домене, у которого существует значение по умолчанию, то после отмены определения значения столбца по умолчанию для этого столбца начинает действовать значение по умолчанию домена.

**DROP COLUMN** Действие DROP COLUMN отменяет определение существующего столбца (удаляет его из таблицы). Действие DROP COLUMN отвергается, если:

- указанный столбец является единственным столбцом таблицы;
- или в этом действии присутствует спецификация RESTRICT, и данный столбец используется в определении каких-либо представлений или ограничений целостности.

**CASCADE** Если в действии присутствует спецификация CASCADE, то его выполнение порождает неявное выполнение оператора DROP для всех представлений и ограничений целостности, в определении которых используется данный столбец.

## Изменение набора табличных ограничений

```
base_table_constraint_alteration_action ::=  
    ADD [ CONSTRAINT ] base_table_constraint_definition  
    | DROP CONSTRAINT constraint_name  
    { RESTRICT | CASCADE }
```

Действие ADD [ CONSTRAINT ] позволяет добавить к набору существующих ограничений таблицы новое ограничение целостности. Можно считать, что новое ограничение добавляется через AND к конъюнкции существующих ограничений, как если бы оно определялось в составе оператора CREATE TABLE.

### ADD [ CONSTRAINT ]

При добавлении нового табличного ограничения с использованием действия ADD [ CONSTRAINT ] мы имеем ситуацию, когда таблица, скорее всего, уже содержит некоторый набор строк, для которого условное выражение нового ограничения может принять значение false. В этом случае выполнение оператора ALTER TABLE, включающего действие ADD [ CONSTRAINT ], отвергается.

### DROP CONSTRAINT

Выполнение действия DROP CONSTRAINT приводит к отмене определения существующего табличного ограничения. Можно отменить определение только именованных табличных ограничений. Спецификации RESTRICT и CASCADE осмыслены только в том случае, если отменяемое ограничение является ограничением возможного ключа (UNIQUE или PRIMARY KEY). При указании RESTRICT действие отвергается, если на данный возможный ключ ссылается хотя бы один внешний ключ.

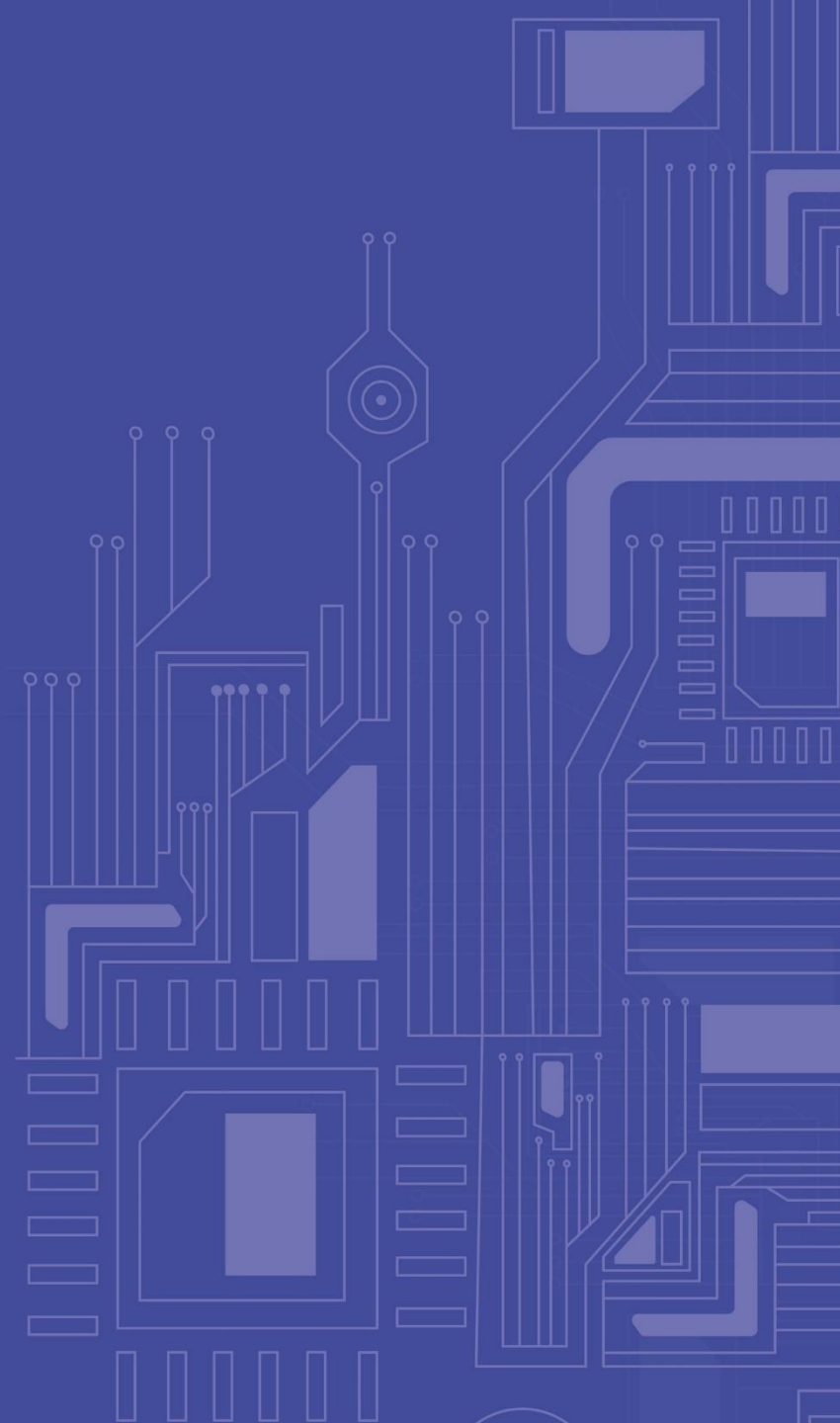
## Отмена определения (уничтожение) базовой таблицы

Для отмены определения (уничтожения) базовой таблицы служит оператор DROP TABLE, задаваемый в следующем синтаксисе:

```
DROP TABLE base_table_name { RESTRICT | CASCADE }
```

Успешное выполнение оператора приводит к тому, что указанная базовая таблица перестает существовать. Уничтожаются все ее строки, определения столбцов и табличные определения целостности. При наличии спецификации RESTRICT выполнение оператора DROP TABLE отвергается, если имя таблицы используется в каком-либо определении представления или ограничения целостности. При наличии спецификации CASCADE оператор выполняется в любом случае, и все определения представлений и ограничений целостности, содержащие ссылки на данную таблицу, также отменяются.

# ОБЩИЕ ОГРАНИЧЕНИЯ ЦЕЛОСТНОСТИ





# ВИДЫ ОГРАНИЧЕНИЙ ЦЕЛОСТНОСТИ



Ограничения целостности, входящие в определение домена, наследуются всеми столбцами, определенными на этих доменах, и являются ограничениями этих столбцов. Кроме того, в определение столбца могут входить определения дополнительных ограничений. Ограничения целостности, входящие в определение столбца (включая ограничения, унаследованные из определения домена), являются ограничениями таблицы, в состав определения которой входит определение данного столбца. Кроме того, в определение таблицы могут входить определения дополнительных ограничений.

Ограничения целостности, входящие в определение таблицы (включая явные и унаследованные от определения доменов ограничения столбцов), представляют собой ограничения базы данных, частью которой является данная таблица. Кроме того, могут определяться дополнительные ограничения базы данных. В стандарте SQL такие дополнительные ограничения базы данных называются ASSERTION, или их называют *общими ограничениями целостности*.

## Иерархия видов ограничений целостности



# ОПРЕДЕЛЕНИЕ ОБЩИХ ОГРАНИЧЕНИЙ ЦЕЛОСТНОСТИ (1/3)



Для определения общего ограничения целостности служит оператор CREATE ASSERTION, задаваемый в следующем синтаксисе:

```
CREATE ASSERTION constraint_name  
CHECK (conditional_expression)
```

При создании общего ограничения целостности его имя всегда должно указываться явно. Хотя синтаксис определения общего ограничения совпадает с синтаксисом определений ограничений столбца и таблицы, в данном случае допускаются только специальные виды условных выражений. Особые свойства условий связаны с тем, что при определении общих ограничений целостности контекстом, в котором вычисляется условное выражение, является весь набор таблиц базы данных, а не набор строк таблицы, как это было при определении табличных ограничений.

## ПРИМЕР

В определении таблицы EMP содержалось ограничение столбца EMP\_BDATE:

```
CHECK (EMP_BDATE >= '1917-10-24')
```

(к работе на предприятии допускаются только те лица, которые родились после Октябрьского переворота). Вот каким образом можно определить такое же ограничение на уровне общих ограничений целостности:

```
CREATE ASSERTION MIN_EMP_BDATE CHECK  
((SELECT MIN(EMP_BDATE)) FROM EMP) >= '1917-10-24')
```

В логическом условии этого общего ограничения выбирается минимальное значение столбца EMP\_BDATE (дата рождения самого старого служащего). Значением условного выражения будет false в том и только в том случае, если среди служащих имеется хотя бы один, родившийся до указанной даты.

# ОПРЕДЕЛЕНИЕ ОБЩИХ ОГРАНИЧЕНИЙ ЦЕЛОСТНОСТИ (2/3)



## Отмена определения общего ограничения целостности

```
DROP ASSERTION constraint_name
```

Для того чтобы отменить ранее определенное общее ограничение целостности, нужно воспользоваться оператором DROP ASSERTION, задаваемым в следующем синтаксисе:

## Немедленная и откладываемая проверка ограничений

```
CHECK (DEPT_EMP_NO =  
  (SELECT COUNT(*) FROM EMP  
   WHERE DEPT_NO = EMP.DEPT_NO))
```

**В контексте каждой выполняемой транзакции каждое ограничение целостности должно находиться в одном из двух режимов: режиме немедленной проверки (immediate) или режиме отложенной проверки (deferred).**

Предположим, например, что в отдел зачисляется новый служащий. Тогда нужно выполнить две операции: (a) вставить новую строку в таблицу EMP и (b) изменить соответствующую строку таблицы DEPT (прибавить единицу к значению столбца DEPT\_EMP\_NO). Очевидно, что в каком бы порядке ни выполнялись эти операции, сразу после выполнения первой из них ограничение целостности будет нарушено, соответствующее действие будет отвергнуто, и мы никогда не сможем принять на работу нового служащего.

В качестве заключительной синтаксической конструкции к любому определению ограничения целостности (любого вида) может быть добавлена спецификация INITIALLY в следующей синтаксической форме:

```
INITIALLY { DEFERRED | IMMEDIATE }  
[ [ NOT ] DEFERRABLE ]
```

Если действие операции нарушает немедленно проверяемое ограничение целостности, то это действие отвергается. Ограничения целостности, находящиеся в режиме отложенной проверки, проверяются при завершении транзакции (COMMIT). Если действия этой транзакции нарушают отложено проверяемое ограничение целостности, то транзакция откатывается (операция COMMIT трактуется как операция ROLLBACK).

# ОПРЕДЕЛЕНИЕ ОБЩИХ ОГРАНИЧЕНИЙ ЦЕЛОСТНОСТИ (3/3)



При выполнении транзакции можно изменить режим проверки некоторых или всех ограничений целостности для данной транзакции. Для этого используется оператор `SET CONSTRAINTS`, задаваемый в следующем синтаксисе:

```
SET CONSTRAINTS { constraint_name_comma_list | ALL }  
                { DEFERRED | IMMEDIATE }
```

Если в операторе указывается список имен ограничений целостности, то все они должны быть `DEFERRABLE`; если хотя бы для одного ограничения из списка это требование не выполняется, то операция `SET CONSTRAINTS` отвергается. При указании ключевого слова `ALL` режим устанавливается для всех ограничений, в определении которых явно или неявно было указано `DEFERRABLE`. Если в качестве желаемого режима проверки ограничений задано `DEFERRED`, то все указанные ограничения переводятся в режим отложенной проверки. Если в качестве желаемого режима проверки ограничений задано `IMMEDIATE`, то все указанные ограничения переводятся в режим немедленной проверки. При этом если хотя бы одно из этих ограничений не удовлетворяется, то операция `SET CONSTRAINTS` отвергается, и все указанные ограничения остаются в предыдущем режиме.

При выполнении операции `COMMIT` неявно выполняется операция `SET CONSTRAINTS ALL IMMEDIATE`. Если эта операция отвергается, то `COMMIT` срабатывает как `ROLLBACK`.

# ASSERTION VS TRIGGER (1/3)



## 1. What are Assertions?

When a constraint involves 2 (or) more tables, the table constraint mechanism is sometimes hard and results may not come as expected. To cover such situation SQL supports the creation of assertions that are constraints not associated with only one table. And an assertion statement should ensure a certain condition will always exist in the database. DBMS always checks the assertion whenever modifications are done in the corresponding table.

Syntax –

```
CREATE ASSERTION [ assertion_name ]  
CHECK ( [ condition ] );
```

## 2. What are Triggers?

A trigger is a database object that is associated with the table, it will be activated when a defined action is executed for the table. The trigger can be executed when we run the following statements:

- 1.INSERT
- 2.UPDATE
- 3.DELETE

And it can be invoked before or after the event.

Syntax –

```
create trigger [trigger_name]  
[before | after]  
{insert | update | delete}  
on [table_name]  
[for each row]  
[trigger_body]
```

# ASSERTION VS TRIGGER (2/3)



## Difference between Assertions and Triggers :

S.No	Assertions	Triggers
1.	We can use Assertions when we know that the given particular condition is always true.	We can use Triggers even particular condition may or may not be true.
2.	When the SQL condition is not met then there are chances to an entire table or even Database to get locked up.	Triggers can catch errors if the condition of the query is not true.
3.	Assertions are not linked to specific table or event. It performs task specified or defined by the user.	It helps in maintaining the integrity constraints in the database tables, especially when the primary key and foreign key constraint are not defined.
4.	Assertions do not maintain any track of changes made in table.	Triggers maintain track of all changes occurred in table.
5.	Assertions have small syntax compared to Triggers.	They have large Syntax to indicate each and every specific of the created trigger.
6.	Modern databases do not use Assertions.	Triggers are very well used in modern databases.

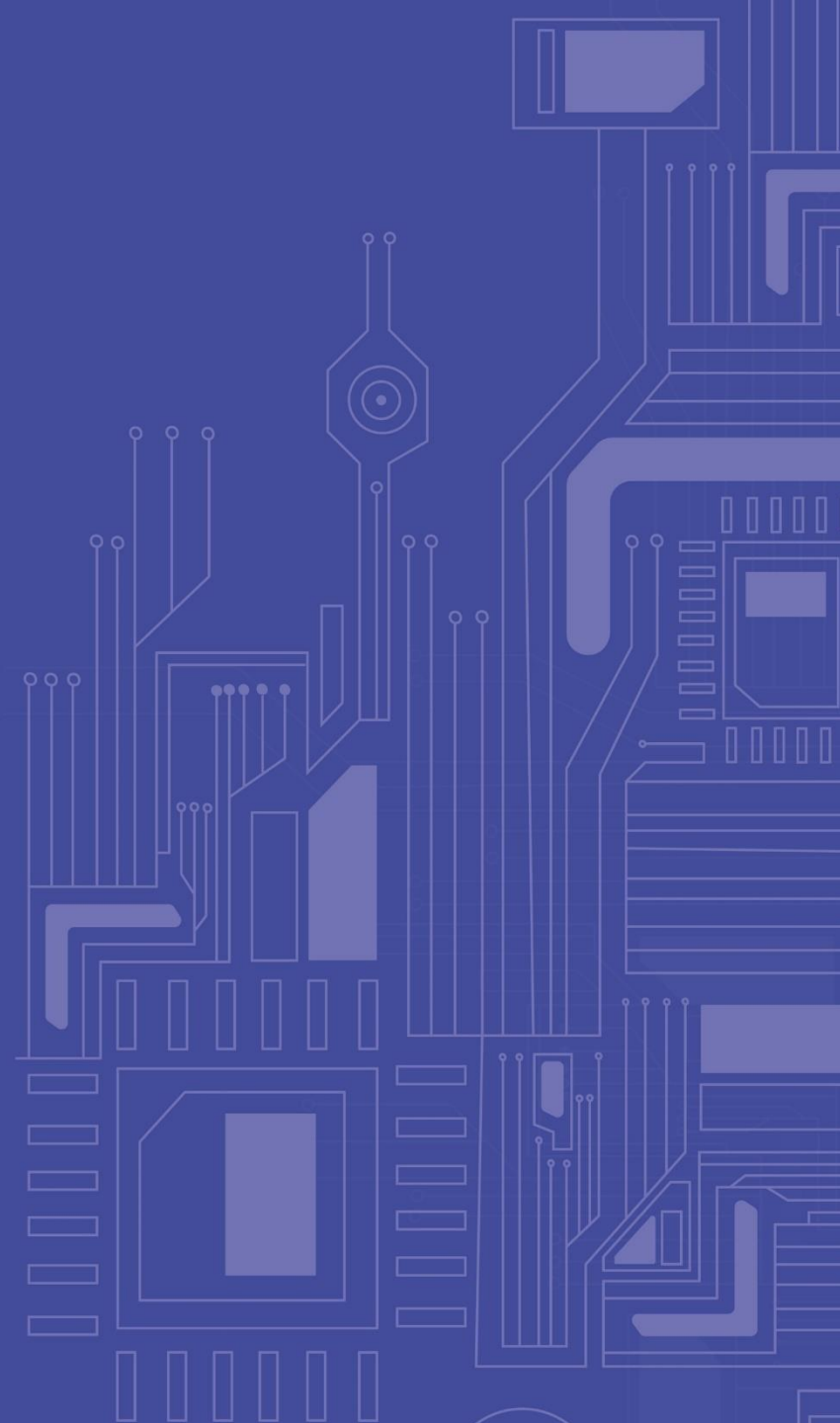


# ASSERTION VS TRIGGER (3/3)



S.No	Assertions	Triggers
7.	Purpose of assertions is to Enforces business rules and constraints.	Purpose of triggers is to Executes actions in response to data changes.
8.	Activation is checked after a transaction completes	Activation is activated by data changes during a transaction
9.	Granularity applies to the entire database	Granularity applies to a specific table or view
10.	Syntax Uses SQL statements	Syntax Uses procedural code (e.g. PL/SQL, T-SQL)
11.	Error handling Causes transaction to be rolled back.	Error handling can ignore errors or handle them explicitly
12.	Assertions may slow down performance of queries.	Triggers Can impact performance of data changes.
13.	Assertions are Easy to debug with SQL statements.	Triggers are more difficult to debug procedural code
14.	Examples- CHECK constraints, FOREIGN KEY constraints	Examples – AFTER INSERT triggers, INSTEAD OF triggers

# СКАЛЯРНЫЕ ВЫРАЖЕНИЯ (VALUE EXPRESSIONS)





*Скалярное выражение* – это выражение, вырабатывающее результат некоторого типа, специфицированного в стандарте. Скалярные выражения являются основой языка SQL, поскольку, хотя это реляционный язык, все условия, элементы списков выборки и т. д. базируются именно на скалярных выражениях.

**unsigned\_value\_specification:** литерал

соответствующего типа или  
вызов *ниладической* функции  
(например, CURRENT\_USER).

**column\_reference:** ссылка на значение данного столбца в данной строке.

**set\_function\_specification :** агрегатные функции  
(*функции над множествами*).

**scalar\_subquery:** скалярный запрос или подзапрос, результатом которого является таблица, состоящая из одной строки и одного столбца, и, если результат подзапроса пуст, то результат первичного выражения – неопределенное значение.

**case\_expression:** выражения с переключателем.

В SQL:2003 имеются девять разновидностей выражений в соответствии с девятью категориями типов данных, значения которых вырабатываются при вычислении выражения

```
value_expression ::=  
    numeric_value_expression  
    | string_value_expression  
    | datetime_value_expression  
    | interval_value_expression  
    | boolean_value_expression  
    | array_value_expression  
    | multiset_value_expression  
    | row_value_expression  
    | user_defined_value_expression  
    | reference_value_expression
```

В основе построения этих видов выражений лежит первичное выражение, определяемое следующим синтаксическим правилом:

```
value_expression_primary ::=  
    unsigned_value_specification  
    | column_reference  
    | set_function_specification  
    | scalar_subquery  
    | case_expression  
    | (value_expression)  
    | cast_specification
```

**Численное выражение** – это выражение, значение которого относится к числовому типу данных.

В численных выражениях SQL первичная составляющая (`numeric_primary`) является либо первичным выражением, либо вызовом функции с численным значением (`numeric_value_function`). Из этого, в частности, следует, что в численные выражения могут входить выражения с переключателем и операции преобразования типов.

Функция `EXTRACT` извлечения поля из значений дата-время или интервал позволяет получить в виде точного числа с масштабом 0 значение любого поля (года, месяца, дня и т. д.). Какой конкретный тип точных чисел будет выбран – определяется в реализации. Функции `ABS` и `MOD` возвращают абсолютное значение числа и остаток от деления одного целого значения на другое соответственно.

## Формальный синтаксис

```
numeric_value_expression > ::= numeric_term
| numeric_value_expression + term
| numeric_value_expression - term
numeric_term ::= numeric_factor
| numeric_term * numeric_factor
| numeric_term / numeric_factor
numeric_factor ::= [ { + | - } ] numeric_primary
numeric_primary ::= value_expression_primary
| numeric_value_function
```

Вызовы функций с численным значением определяются следующими синтаксическими правилами:

```
numeric_value_function ::=
POSITION (character_value_expression
IN character_value_expression)
| {CHAR_LENGTH | CHARACTER_LENGTH }
(string_value_expression)
| OCTET_LENGTH (string_value_expression)
| BIT_LENGTH (string_value_expression)
| EXTRACT ({ datetime_field | time_zone field }
FROM { datetime_value_expression
| interval_value_expression })
| CARDINALITY (array_value_expression
| multiset_value_expression)
| ABS (numeric_value_expression)
| MOD (numeric_value_expression)
```

**Выражения символьных и битовых строк** – это выражения, значениями которых являются символьные или битовые строки.

Единственная применимая для построения выражений операция – это конкатенация, производящая «склейку» строк-операндов. Первичной составляющей выражения над строками может быть как первичное скалярное выражение, так и вызов функций, возвращающих строчные значения.

Соответствующие конструкции определяются следующим синтаксисом:

```
string_value_expression ::= character_value_expression
    | bit_value_expression
character_value_expression ::= concatenation
    | character_factor
concatenation ::= character_value_expression || character_factor
character_factor ::= character_primary [ collate_clause ]
character_primary ::= value_expression_primary
    | string_value_function
bit_value_expression ::= bit_concatenation
    | bit_factor
bit_concatenation ::= bit_value_expression || bit_primary
bit_primary ::= value_expression_primary
    | string value function
```

Синтаксис вызова функций, возвращающих строки, определяется следующими правилами:

```
string_value_function ::= character_value_function
    | bit_value_function
character_value_function ::= SUBSTRING
    (character_value_expression
    FROM start_position
    [ FOR string_length ])
    | SUBSTRING (character_value_expression
    SIMILAR character_value_expression
    ESCAPE character_value_expression)
    | { UPPER | LOWER }
    (character_value_expression)
    | CONVERT (character_value_expression
    USING conversion_name)
    | TRANSLATE (character_value_expression)
    USING translation_name)
    | TRIM ([ {LEADING | TRAILING | BOTH} ]
    [ character_value_expression ]
    [ character_value_expression ])
    | OVERLAY (character_value_expression
    PLACING character_value_expression
    FROM start_position
    [ FOR string_length ])
bit_value_function ::= SUBSTRING (bit_value_expression
    FROM start_position
    [ FOR string_length ])
start_position ::= numeric_value_expression
string_length ::= numeric_value_expression
```

# ВЫРАЖЕНИЯ ДАТЫ - ВРЕМЕНИ



К **выражениям даты-времени** мы относим выражения, вырабатывающие значения типа дата-время и интервал.

Выражения даты-времени определяются следующими синтаксическими правилами:

```
datetime_value_expression ::=
    datetime_term
    | interval_value_expression + datetime_term
    | datetime_value_expression + interval_term
    | datetime_value_expression - interval_term
datetime_term ::= datetime_primary
    [ AT { LOCAL | TIME_ZONE interval_value_expression } ]
datetime_primary ::= value_expression_primary
    | datetime_value_function
```

Синтаксис выражений со значениями типа интервал определяется следующими правилами:

Стоит только заметить, что квалификатор интервала указывается для того, чтобы явно специфицировать единицу измерения интервала. Поддерживается только одна функция ABS (абсолютное значение), аргументом которой является выражение со значением типа интервал.

Вызовы функций, возвращающих значение дата-время:

```
datetime_value_function ::= CURRENT_DATE
    | CURRENT_TIME [ (precision) ]
    | LOCALTIME [ (precision) ]
    | CURRENT_TIMESTAMP [ (precision) ]
    | LOCALTIMESTAMP [ (precision) ]
```

```
interval_value_expression ::=
    interval_term
    | interval_value_expression + interval_term
    | interval_value_expression - interval_term
    | (datetime_value_expression - datetime_term)
    interval_qualifier
interval_term ::= interval_factor
    | interval_term * numeric_factor
    | interval_term / numeric_factor
    | numeric_term * interval_factor
interval_factor ::= [ { + | - } ]
    interval_primary [ <interval qualifier> ]
interval_primary ::= value_expression_primary
    | interval_value_function
```

# БУЛЕВСКИЕ ВЫРАЖЕНИЯ

К **булевым выражениям** относятся выражения, вырабатывающие значения булевого типа (напомним, что булевский тип языка SQL содержит три логических значения – true, false и unknown).

Булевские выражения определяются следующими синтаксическими правилами:

```
boolean_value_expression ::= boolean_term
    | boolean_value_expression OR boolean_term
boolean_term ::= boolean_factor
    | boolean_term AND boolean_factor
boolean_factor ::= [ NOT ] boolean_test
boolean_test ::= boolean_primary [ IS [ NOT ] truth_value ]
truth_value ::= TRUE | FALSE | UNKNOWN
boolean_primary ::= predicate
    | (boolean_value_expression)
    | value_expression_primary
```

Выражения вычисляются слева направо с учетом приоритетов операций (наиболее высокий приоритет имеет унарная операция NOT, следующим уровнем приоритета обладает «мультипликативная» операция конъюнкции AND, и самый низкий приоритет у «аддитивной» операции дизъюнкции OR) и круглых скобок.

Операции IS и IS NOT определяются следующими таблицами истинности:

IS	TRUE	FALSE	UNKNOWN
TRUE	TRUE	FALSE	FALSE
FALSE	FALSE	TRUE	FALSE
UNKNOWN	FALSE	FALSE	TRUE

IS NOT	TRUE	FALSE	UNKNOWN
TRUE	FALSE	TRUE	TRUE
FALSE	TRUE	FALSE	TRUE
UNKNOWN	TRUE	TRUE	FALSE



Наиболее общим видом выражения с переключателем является выражение с поисковым переключателем (*searched\_case*). Правила вычисления выражений этого вида состоят в следующем. Вычисляется логическое выражение, указанное в первом разделе WHEN списка (*searched\_when\_clause\_list*). Если значение этого логического выражения равняется *true*, то значением всего выражения с поисковым переключателем является значение выражения, указанного в первом разделе WHEN после ключевого слова THEN. Иначе аналогичные действия производятся для второго раздела WHEN и т. д. Если ни для одного раздела WHEN при вычислении логического выражения не было получено значение *true*, то значением всего выражения с поисковым переключателем является значение выражения, указанного в разделе ELSE.

**Типы всех выражений, значения которых могут являться результатом выражения с поисковым переключателем, должны быть совместимыми**, и типом результата является «наименьший общий» тип набора типов выражений-кандидатов на выработку результата. Если в выражении отсутствует раздел ELSE, предполагается наличие раздела ELSE NULL.

В выражении с простым переключателем (*simple\_case*) тип данных операнда переключателя (выражения, непосредственно следующего за ключевым словом CASE, назовем его CO – Case Operand) должен быть совместим с типом данных операнда каждого варианта (выражения, непосредственно следующего за ключевым словом WHEN; назовем WO – When Operand).

```
case_expression ::= case_abbreviation
                  | case_specification

case_abbreviation ::= NULLIF (value_expression , value_expression)
                  | COALESCE (value_expression_comma_list)
case_specification ::= simple_case | searched_case

simple_case ::= CASE value_expression simple_when_clause_list
               [ ELSE value_expression ] END

searched_case ::= CASE searched_when_clause_list
                  [ ELSE value_expression ] END
simple_when_clause ::= WHEN value_expression
                     THEN value_expression
searched_when_clause ::= WHEN conditional_expression
                       THEN value_expression
```

# ВЫРАЖЕНИЯ С ПЕРЕКЛЮЧАТЕЛЕМ (2/2)



```
CASE CO WHEN W01 THEN result1
  WHEN W02 THEN result2
  . . . . .
  WHEN WOn THEN resultn
  ELSE result
END
```

эквивалентно выражению с поисковым переключателем

```
CASE WHEN CO = W01 THEN result1
  WHEN CO = W02 THEN result2
  . . . . .
  WHEN CO = WOn THEN resultn
  ELSE result
END
```

Выражение NULLIF (V1, V2) эквивалентно следующему выражению с переключателем:

```
CASE WHEN V1 = V2 THEN NULL ELSE V1 END.
```

Выражение COALESCE (V1, V2) эквивалентно следующему выражению с переключателем:

```
CASE WHEN V1 IS NOT NULL THEN V1 ELSE V2 END.
```

Выражение COALESCE (V1, V2, . . . Vn) для  $n \geq 3$  эквивалентно следующему выражению с переключателем:

```
CASE WHEN V1 IS NOT NULL THEN V1 ELSE COALESCE (V2, ... n) END.
```

# ОБЩАЯ СТРУКТУРА ОПЕРАТОРА ВЫБОРКИ





```
SELECT [ ALL | DISTINCT ] select_item_commalist  
FROM table_reference_commalist  
[ WHERE conditional_expression ]  
[ GROUP BY column_name_commalist ]  
[ HAVING conditional_expression ]  
[ ORDER BY order_item_commalist ]
```

На первом шаге выполняется раздел FROM. Если список ссылок на таблицы (table\_reference\_commalist) этого раздела соответствует таблицам A, B, ... C, то в результате выполнения раздела FROM образуется таблица (назовем ее T), являющаяся расширенным декартовым произведением таблиц A, B, ..., C. Если в разделе FROM указана только одна таблица, то она же и является результатом выполнения этого раздела.

На втором шаге выполняется раздел WHERE. Условное выражение (conditional\_expression) этого раздела применяется к каждой строке таблицы T, и результатом является таблица T1, содержащая те и только те строки таблицы T, для которых результатом вычисления условного выражения является true. (Заголовки таблиц T и T1 совпадают.) Если раздел WHERE в операторе выборки отсутствует, то это трактуется как наличие раздела WHERE true, т. е. T1 содержит те и только те строки, которые содержатся в таблице T.

Если в операторе выборки присутствует раздел GROUP BY, то он выполняется на третьем шаге. Каждый элемент списка имен столбцов (column\_name\_commalist), указываемого в этом разделе, должен быть одним из имен столбцов таблицы T1. В результате выполнения раздела GROUP BY образуется *сгруппированная таблица* T2, в которой строки таблицы T1 расставлены в минимальное число групп, таких, что во всех строках одной группы значения столбцов, указанных в списке имен столбцов раздела GROUP BY (*столбцов группировки*), одинаковы. Заметим, что сгруппированные таблицы не могут являться окончательным результатом оператора выборки. Они существуют только на концептуальном уровне на стадии выполнения запроса, содержащего раздел GROUP BY.

```
SELECT [ ALL | DISTINCT ] select_item_commalist
FROM table_reference_commalist
[ WHERE conditional_expression ]
[ GROUP BY column_name_commalist ]
[ HAVING conditional_expression ]
[ ORDER BY order_item_commalist ]
```

Если в операторе выборки присутствует раздел HAVING, то он выполняется на следующем шаге. Условное выражение этого раздела применяется к каждой группе строк таблицы T2, и результатом является сгруппированная таблица T3, содержащая те и только те группы строк таблицы T2, для которых результатом вычисления условного выражения является true.

Условное выражение раздела HAVING строится по синтаксическим правилам, общим для всех условных выражений, но обладает той спецификой, что применяется к группам строк, а не к отдельным строкам. Поэтому предикаты, из которых строится это условное выражение, должны быть предикатами на группу в целом. В них могут использоваться имена столбцов группировки (*инварианты группы*) и так называемые агрегатные функции (COUNT, SUM, MIN, MAX, AVG) от других столбцов.

При наличии в запросе раздела HAVING, которому не предшествует раздел GROUP BY, таблица T1 рассматривается как сгруппированная таблица, состоящая из одной группы строк, без столбцов группирования. В этом случае логическое выражение раздела HAVING может состоять только из предикатов с агрегатными функциями, а результат вычисления этого раздела T3 либо совпадает с таблицей T1, либо является пустым.

Если в операторе выборки присутствует раздел GROUP BY, но отсутствует раздел HAVING, то это трактуется как наличие раздела HAVING true, т. е. T3 содержит те и только те группы строк, которые содержатся в таблице T2.

# ФОРМИРОВАНИЕ РЕЗУЛЬТАТА (1/2)



Только теперь выполняется **SELECT**.

В ЗАПРОСЕ ОТСУТСТВУЮТ РАЗДЕЛЫ GROUP BY И HAVING

На основе таблицы T1 строится таблица T4, содержащая столько строк, сколько строк или групп строк содержится в таблице T1.

ЯВНО ИЛИ НЕЯВНО ЗАДАН РАЗДЕЛ HAVING

На основе сгруппированной таблицы T3 строится таблица T4, содержащая столько строк, сколько строк или групп строк содержится в таблице T3 .

Число столбцов в таблице T4 зависит от числа элементов в списке элементов выборки (select\_item\_commalist) и от вида элементов.

Рассмотрим, каким образом формируются значения столбцов в таблице T4. Элемент списка выборки может задаваться одним из двух способов:

value\_expression [ [ AS ] column\_name ]

В этом случае каждый элемент списка элементов выборки соответствует столбцу таблицы T4. Столбцу может быть явным образом приписано имя. выражение, содержащееся в элементе выборки, может содержать литеральные константы и вызовы функций со значениями соответствующих типов (в том числе ниладические).

```
select_item ::= value_expression [ [ AS ] column_name ]  
              | [ correlation_name . ] *
```

# ФОРМИРОВАНИЕ РЕЗУЛЬТАТА (2/2)



```
select_item ::= value_expression [ [ AS ] column_name ]  
| [ correlation_name . ] *
```

В ЗАПРОСЕ ОТСУТСТВУЮТ РАЗДЕЛЫ GROUP  
BY И HAVING

В выражении могут использоваться имена столбцов таблицы T1. Выражение вычисляется для каждой строки таблицы T1, и именам столбцов соответствуют значения этих столбцов в данной строке таблицы T1.

```
value_expression [ [ AS ] column_name ]
```

ЕСТЬ GROUP BY, ЯВНО ИЛИ НЕЯВНО ЗАДАН  
РАЗДЕЛ HAVING

В выражение могут входить непосредственно имена только тех столбцов таблицы T3, которые входили в список столбцов группировки раздела GROUP BY оператора выборки.

(Если сгруппированная таблица T3 была образована за счет наличия раздела HAVING без присутствия раздела GROUP BY, то в выражении элемента выборки вообще нельзя непосредственно использовать имена столбцов таблицы T3). Имена других столбцов таблицы T3 могут использоваться только в конструкциях вызова агрегатных функций COUNT, SUM, MIN, MAX, AVG. Выражение вычисляется для каждой группы строк таблицы T3. Именам столбцов, входящих в выражение непосредственно, сопоставляются значения этих столбцов, которые соответствуют данной группе строк таблицы T3.

```
select_item ::= value_expression [ [ AS ] column_name ]  
| [ correlation_name . ] *
```

```
[ correlation_name . ] *
```

Во втором варианте спецификация элемента списка выборки вида  $[ Z. ]^*$  является сокращенной формой записи списка  $Z.a_1, Z.a_2, \dots, Z.a_n$ , где  $a_1, a_2, \dots, a_n$  представляет собой полный список имен столбцов таблицы, псевдоним которой  $Z$ .

- ➔ Для именованной таблицы, входящей в список раздела FROM только один раз, можно использовать имя таблицы вместо псевдонима.
- ➔ Можно опустить псевдоним только в том случае, если в разделе FROM указана только одна таблица.

ЕСТЬ GROUP BY, ЯВНО ИЛИ НЕЯВНО ЗАДАН  
РАЗДЕЛ HAVING

Этот второй вариант спецификации элемента выборки допустим только тогда, когда все столбцы таблицы с псевдонимом  $Z$  входят в список столбцов группировки раздела GROUP BY.

Итак, мы получили таблицу T4. Если в спецификации раздела SELECT отсутствует ключевое слово DISTINCT, или присутствует ключевое слово ALL, либо отсутствуют и ALL, и DISTINCT, то T4 является результатом выполнения раздела SELECT. В противном случае на завершающей стадии выполнения раздела SELECT в таблице T4 удаляются строки-дубликаты. Если в операторе выборки не содержится раздел ORDER BY, то таблица T4 является результирующей таблицей запроса.

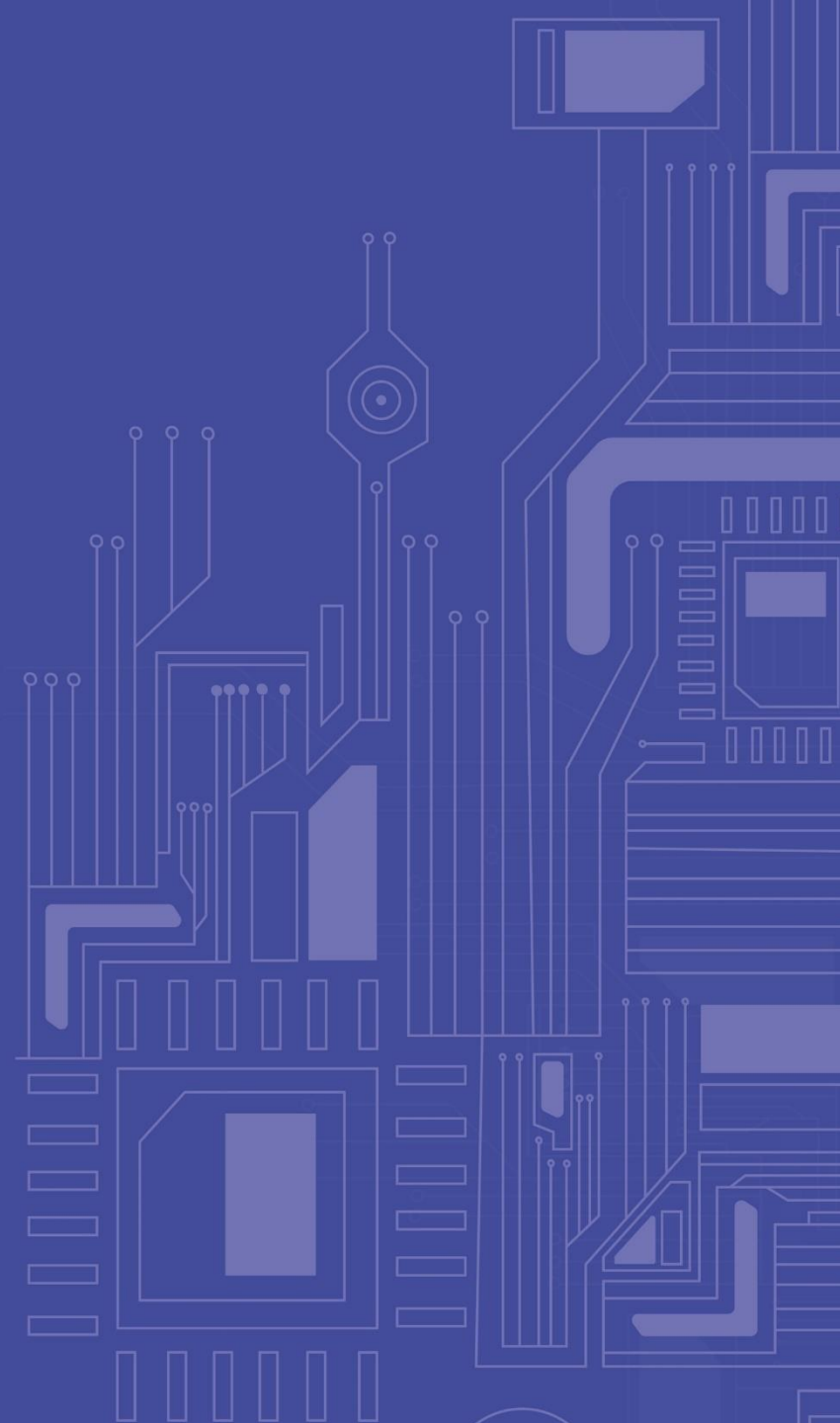
## ORDER BY

Как сравнивать  
строки

```
order_item ::= value_expression [ collate_clause ]  
[ { ASC | DESC } ]
```

Выбирается первый элемент списка сортировки, и строки таблицы T4 расставляются в порядке возрастания (если в элементе присутствует спецификация ASC; при отсутствии спецификации ASC/DESC предполагается наличие ASC) или в порядке убывания (при наличии спецификации DESC) в соответствии со значениями выражения, содержащегося в данном элементе, которые вычисляются для каждой строки таблицы T4. Далее выбирается второй элемент списка сортировки, и в соответствии со значениями заданного в нем выражения и порядка сортировки расставляются строки, которые после первого шага сортировки образовали группы с одинаковым значением выражения первого элемента списка сортировки. Операция продолжается до исчерпания списка элементов сортировки. Результирующий отсортированный список строк является окончательным результатом запроса.

# ССЫЛКИ НА ТАБЛИЦЫ РАЗДЕЛА FROM





Раздел FROM оператора выборки определяется синтаксическим правилом

```
FROM table_reference commalist
```

Полный набор синтаксических правил SQL:1999, определяющий table\_reference.

```
table_reference ::= table_primary | joined_table
table_primary ::= table_or_query_name [ [ AS ] correlation_name
    [ (derived_column_list) ] ]
    | derived_table [ [ AS ] correlation_name
    [ (derived_column_list) ] ]
    | lateral_derived_table [ [ AS ] correlation_name
    [ (derived_column_list) ] ]
    | collection_derived_table [ [ AS ] correlation_name
    [ (derived_column_list) ] ]
    | ONLY (table_or_query_name) [ [ AS ] correlation_name
    [ (derived_column_list) ] ]
    | (joined_table)
table_or_query_name ::= { table_name | query_name }
derived_table ::= (query_expression)
lateral_derived_table ::= LATERAL (query_expression)
collection_derived_table ::= UNNEST
    (collection_value_expression) [ WITH ORDINALITY ]
```



# ТАБЛИЧНОЕ ВЫРАЖЕНИЕ, СПЕЦИФИКАЦИЯ ЗАПРОСОВ И ВЫРАЖЕНИЕ ЗАПРОСОВ



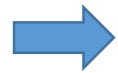
**Табличным выражением** (table\_expression)  
называется конструкция :

```
table_expression ::= FROM table_reference_comma_list  
[ WHERE conditional_expression ]  
[ GROUP BY column_name_comma_list ]  
[ HAVING conditional_expression ]
```



**Спецификацией запроса** (query\_specification)  
называется конструкция :

```
query_specification SELECT [ ALL | DISTINCT ]  
select_item_comma_list table_expression
```



**Выражением запросов** (query\_expression)  
называется конструкция :

```
query_expression ::= [ with_clause ] query_expression_body  
query_expression_body ::= { non_join_query_expression  
| joined_table }  
non_join_query_expression ::= non_join_query_term  
| query_expression_body  
{ UNION | EXCEPT } [ ALL | DISTINCT ]  
[ corresponding_spec ] query_term  
query_term ::= non_join_query_term | joined_table  
non_join_query_term ::= non_join_query_primary  
| query_term INTERSECT [ ALL | DISTINCT ]  
[ corresponding_spec ] query_primary  
query_primary ::= non_join_query_primary | joined_table  
non_join_query_primary ::= simple_table  
| (non_join_query_expression)  
simple_table ::= query_specification  
| table_value_constructor  
| TABLE table_name  
corresponding_spec ::= CORRESPONDING  
[ BY column_name_comma_list ]
```

В основном выражение запросов строится из выражений, значениями которых являются таблицы, с использованием «теоретико-множественных» операций UNION (объединение), EXCEPT (вычитание) и INTERSECT (пересечение). Операция пересечения является «мультипликативной» и обладает более высоким приоритетом, чем «аддитивные» операции объединения и вычитания. Вычисление выражения производится слева направо с учетом приоритетов операций и круглых скобок.

- Если выражение запросов не включает ни одной теоретико-множественной операции, то результатом вычисления выражения запросов является результат вычисления простой или соединенной таблицы.
- Если в терме (`non_join_query_term`) или выражении запросов (`non_join_query_expression`) без соединения присутствует теоретико-множественная операция, то пусть `T1`, `T2` и `TR` обозначают соответственно первый операнд, второй операнд и результат терма или выражения соответственно, а `OP` – используемую теоретико-множественную операцию.
- Если в операции присутствует спецификация `CORRESPONDING`, то:
  1. если присутствует конструкция `BY column_name_comma_list`, то все имена в этом списке должны быть различны, и каждое имя должно являться одновременно именем некоторого столбца таблицы `T1` и именем некоторого столбца таблицы `T2`, причем типы этих столбцов должны быть совместимыми; обозначим данный список имен через `SL`;
  2. если список соответствия столбцов не задан, пусть `SL` обозначает список имен столбцов, являющихся именами столбцов и в `T1`, и в `T2`, в том порядке, в котором эти имена фигурируют в `T1`;
  3. вычисляемые терм или выражение запросов без соединения эквивалентны выражению `(SELECT SL FROM T1) OP (SELECT SL FROM T2)`, не включающему спецификацию `CORRESPONDING`.
- При отсутствии в операции спецификации `CORRESPONDING` операция выполняется таким образом, как если бы эта спецификация присутствовала и включала конструкцию `BY column_name_comma_list`, в которой были бы перечислены все столбцы таблицы `T1`.

- При выполнении операции  $OP$  две строки  $s_1$  с именами столбцов  $c_1, c_2, \dots, c_n$  и  $s_2$  с именами столбцов  $d_1, d_2, \dots, d_n$  считаются строками-дубликатами, если для каждого  $i$  ( $i = 1, 2, \dots, n$ ) либо  $c_i$  и  $d_i$  не содержат  $NULL$ , и  $(c_i = d_i) = true$ , либо и  $c_i$ , и  $d_i$  содержат  $NULL$ .
- Если в операции  $OP$  не задана спецификация  $ALL$ , то в  $TR$  строки-дубликаты удаляются.
- Если спецификация  $ALL$  задана, то пусть  $s$  – строка, являющаяся дубликатом некоторой строки  $t_1$ , или некоторой строки  $t_2$ , или обеих; пусть  $m$  – число дубликатов  $s$  в  $t_1$ , а  $n$  – число дубликатов  $s$  в  $t_2$ . Тогда:
  - если указана операция  $UNION$ , то число дубликатов  $s$  в  $TR$  равно  $m + n$ ;
  - если указана операция  $EXCEPT$ , то число дубликатов  $s$  в  $TR$  равно  $\max((m-n), 0)$ ;
  - если указана операция  $INTERSECT$ , то число дубликатов  $s$  в  $TR$  равно  $\min(m, n)$ .

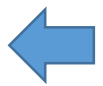
Как видно из синтаксиса выражения запросов, в этом выражении может присутствовать раздел WITH. Он задается в следующем синтаксисе:

```
with_clause ::= WITH [ RECURSIVE ] with_element_comma_list
with_element ::= query_name [ (column_name_list) ]
               AS (query_expression) [ search_or_cycle_clause ]
```

Общую форму раздела WITH мы обсудим, когда будем рассматривать средства формулировки рекурсивных запросов. Пока ограничимся случаем, когда в разделе WITH отсутствуют спецификация RECURSIVE и search\_or\_cycle\_clause. Тогда конструкция

```
WITH query_name (c1, c2, ... cn) AS (query_exp_1) query_exp_2
```

означает, что в любом месте выражения запросов query\_exp\_2, где допускается появление ссылки на таблицу, можно использовать имя query\_name. Можно считать, что перед выполнением query\_exp\_2 происходит выполнение query\_exp\_1, и результирующая таблица с именами столбцов c1, c2, ... cn сохраняется под именем query\_name. Как мы увидим позже, в этом случае раздел WITH фактически служит для локального определения представляемой таблицы (VIEW).



```
query_expression ::= [ with_clause ] query_expression_body
query_expression_body ::= { non_join_query_expression
                          | joined_table }
non_join_query_expression ::= non_join_query_term
                             | query_expression_body
                             { UNION | EXCEPT }[ ALL | DISTINCT ]
                             [ corresponding_spec ] query_term
query_term ::= non_join_query_term | joined_table
non_join_query_term ::= non_join_query_primary
                     | query_term INTERSECT [ ALL | DISTINCT ]
                     [ corresponding_spec ] query_primary
query_primary ::= non_join_query_primary | joined_table
non_join_query_primary ::= simple_table
                       | (non_join_query_expression)
simple_table ::= query_specification
             | table_value_constructor
             | TABLE table_name
corresponding_spec ::= CORRESPONDING
                   [ BY column_name_comma_list ]
```



# КОНСТРУКТОРЫ ЗНАЧЕНИЯ СТРОКИ И ТАБЛИЦЫ



В определении конструктора значения-таблицы используется конструктор значения-строки, который строит упорядоченный набор скалярных значений, представляющий строку (возможно и использование подзапроса):

```
row_value_constructor ::= row_value_constructor_element  
| [ ROW ] (row_value_constructor_element_comma_list)  
| row_subquery  
row_value_constructor_element ::= value_expression | NULL | DEFAULT
```



```
table_value_constructor ::= VALUES  
row_value_constructor_comma_list
```

Заметим, что значение элемента по умолчанию можно использовать только в том случае, когда конструктор значения-строки применяется в операторе INSERT (тогда этим значением будет значение по умолчанию соответствующего столбца).

Конструктор значения-таблицы производит таблицу на основе заданного набора конструкторов значений-строк.

Конечно, для того чтобы корректно построить таблицу, требуется, чтобы строки, производимые всеми конструкторами строк, были одной и той же степени и чтобы типы (или домены) соответствующих столбцов являлись приводимыми.

```
query_expression ::= [ with_clause ] query_expression_body  
query_expression_body ::= { non_join_query_expression  
| joined_table }  
non_join_query_expression ::= non_join_query_term  
| query_expression_body  
{ UNION | EXCEPT } [ ALL | DISTINCT ]  
[ corresponding_spec ] query_term  
query_term ::= non_join_query_term | joined_table  
non_join_query_term ::= non_join_query_primary  
| query_term INTERSECT [ ALL | DISTINCT ]  
[ corresponding_spec ] query_primary  
query_primary ::= non_join_query_primary | joined_table  
non_join_query_primary ::= simple_table  
| (non_join_query_expression)  
simple_table ::= query_specification  
| table_value_constructor  
| TABLE table_name  
corresponding_spec ::= CORRESPONDING  
[ BY column_name_comma_list ]
```

Конструкция `TABLE table_name` является сокращенной формой записи выражения `SELECT * FROM table_name.`

## Ссылки на базовые, представляемые и порождаемые таблицы

```
table_reference ::= table_primary
table_primary ::= table_or_query_name [ [ AS ] correlation_name
               [ (derived_column_list) ] ]
               | derived_table [ AS ] correlation_name
               [ (derived_column_list) ]
table_or_query_name ::= { table_name | query_name }
derived_table ::= (query_expression)
```

в самом простом случае в качестве ссылки на таблицу используется *имя таблицы* (базовой или представляемой) или имя запроса, присоединенного к данному запросу с помощью раздела WITH. В другом случае (derived\_table) *порождаемая таблица* задается выражением запроса, заключенным в круглые скобки. Явное указание имен столбцов результата запроса из раздела WITH или порождаемой таблицы требуется в том случае, когда эти имена не выводятся явно из соответствующего выражения запроса. Обратите внимание, что в таких случаях в соответствующем элементе списка раздела FROM должен указываться псевдоним (correlation\_name), потому что иначе таблица была бы вообще лишена имени. Можно считать, что выражение запроса вычисляется и сохраняется во временной таблице при обработке раздела FROM.

Оператор создания представления в общем случае определяется следующими синтаксическими правилами:

```
create_view ::= CREATE [ RECURSIVE ] VIEW table_name  
              [ column_name_comma_list ]  
              AS query_expression  
              [ WITH [ CASCADED | LOCAL ] CHECK OPTION ]
```

Рекурсивные представления (такие, в определении которых присутствует ключевое слово RECURSIVE) и необязательный раздел WITH CHECK OPTION мы обсудим позже.

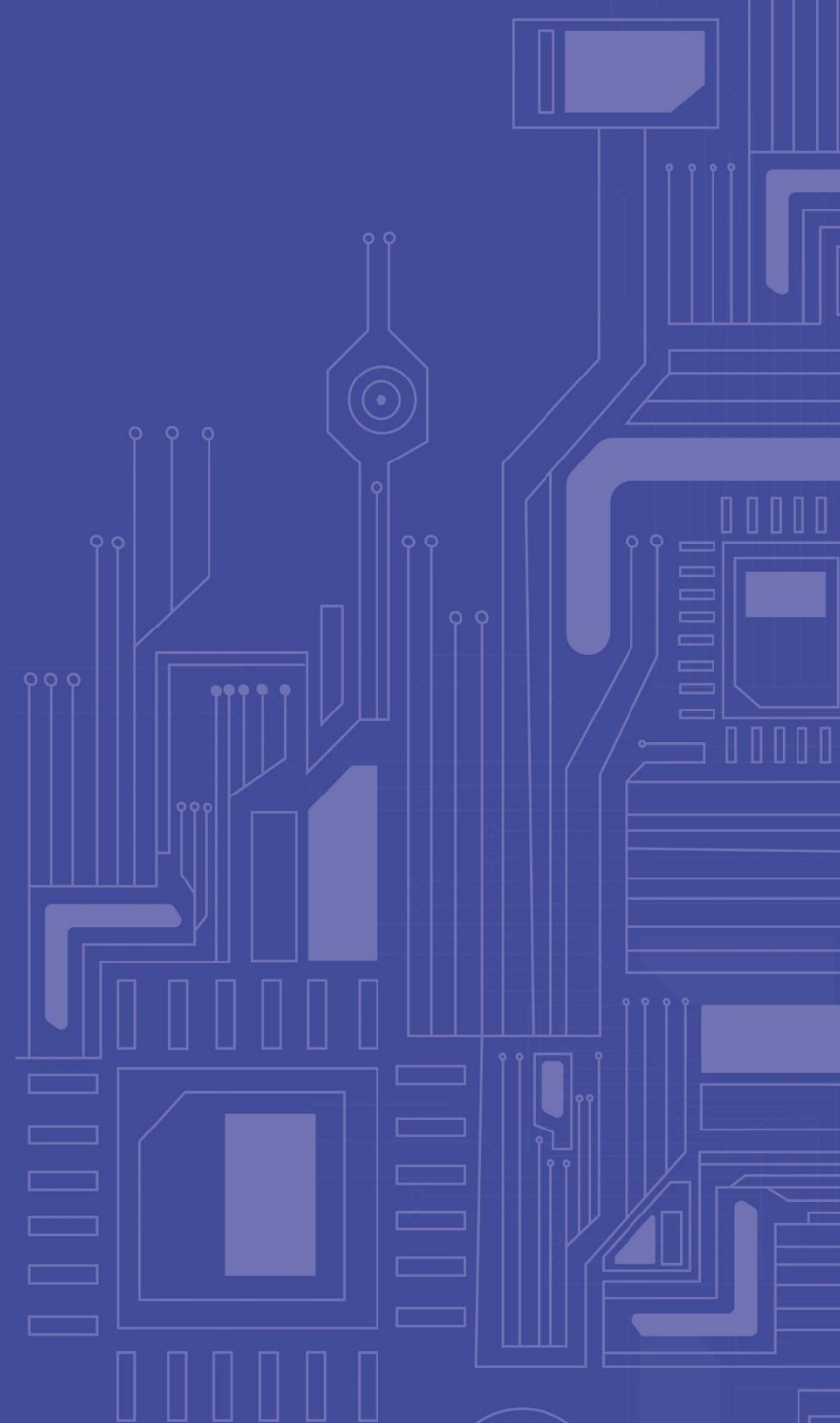
Рассмотрим только простую форму представлений, определяемых по следующим правилам:

```
create_view ::= CREATE VIEW table_name  
              [ column_name_comma_list ]  
              AS query_expression
```

Имя таблицы, задаваемое в определении представления, существует в том же пространстве имен, что и имена базовых таблиц, и, следовательно, должно отличаться от всех имен таблиц, созданных тем же пользователем.

Если имя представления встречается в разделе FROM какого-либо оператора выборки, то вычисляется выражение запроса, указанное в разделе AS, и оператор выборки работает с результирующей таблицей этого выражения запроса.

Как и для всех других вариантов оператора CREATE, для CREATE VIEW имеется обратный оператор DROP VIEW table\_name, выполнение которого приводит к отмене определения представления (реально это выражается в удалении данных о представлении из таблиц-каталогов базы данных).

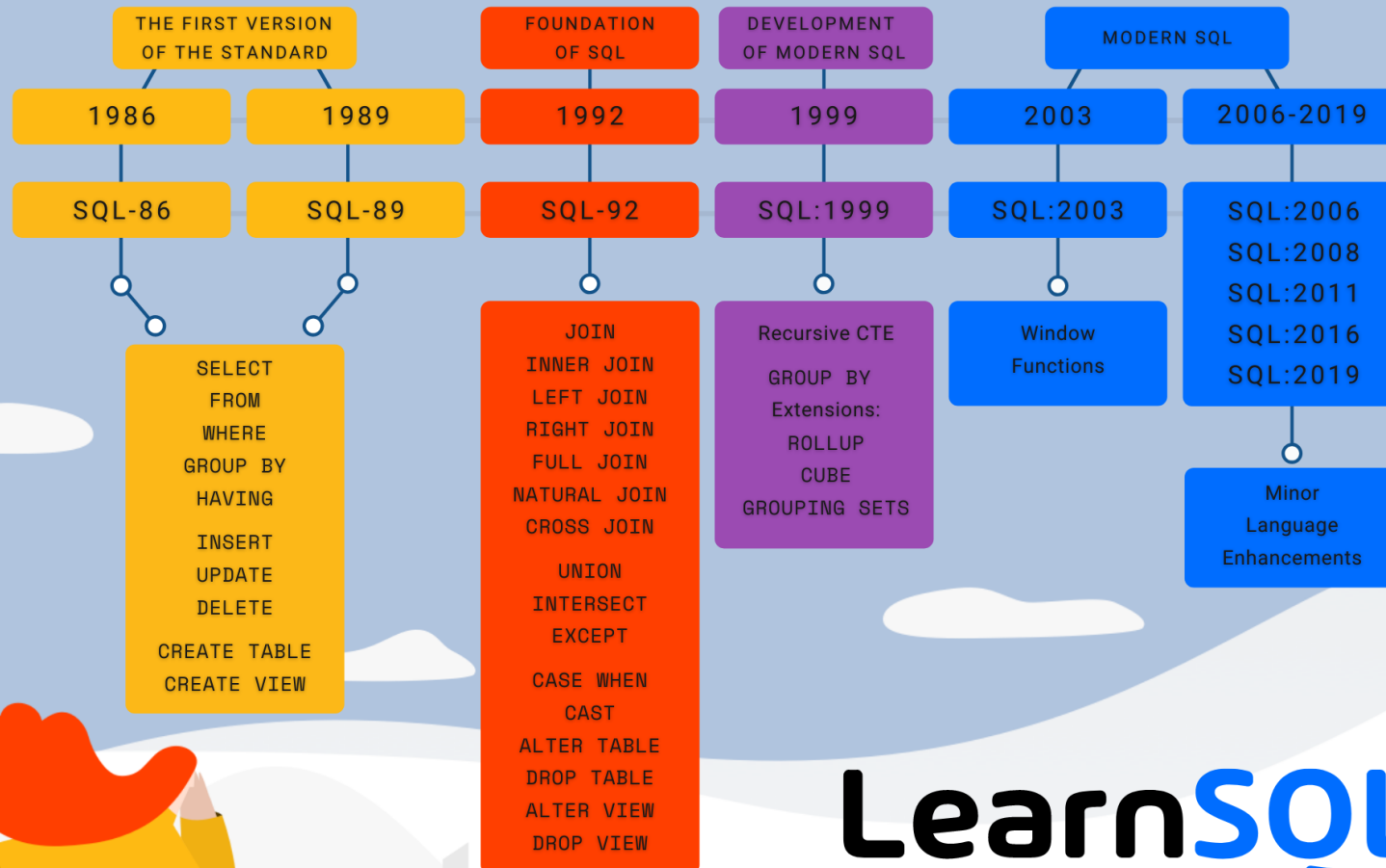




СЕМИНАР



## The History of SQL Standards



Предикаты SQL – большое разнообразие, даже избыточность

В документации полезно смотреть, что нового появилось

В последнее время - расширения/улучшения для XML/JSON

# ИЗМЕНЕНИЕ ОПРЕДЕЛЕНИЯ СТОЛБЦА



```
ALTER TABLE EMP  
ADD EMP_BONUS SALARY DEFAULT NULL  
CONSTRAINT BONSAL CHECK (VALUE < EMP_SAL);
```

Обратите внимание, что мы присвоили проверочному ограничению столбца явное имя, чтобы в случае, если ограничения на размер премии изменятся (что вполне возможно), можно было бы легко отменить это ограничение, воспринимая его как табличное.

При определении столбца EMP\_SAL таблицы EMP для этого столбца явно не определялось значение по умолчанию (оно наследовалось из определения домена). Если в какой-то момент это стало неправильным (например, повысился размер минимальной зарплаты), можно установить новое значение по умолчанию:

```
ALTER TABLE EMP ALTER EMP_SAL SET DEFAULT 15000.00.
```

При определении столбца DEPT\_TOTAL\_SAL таблицы DEPT для него было установлено значение по умолчанию 1000000. Главный бухгалтер предприятия может быть недоволен тем, что такие важные данные, как объем фонда зарплаты отделов, могут устанавливаться по умолчанию. Тогда можно отменить это значение по умолчанию:

```
ALTER TABLE DEPT ALTER DEPT_TOTAL_SAL DROP DEFAULT.
```

Обратите внимание, что после выполнения этого оператора при вставке новой строки в таблицу DEPT всегда потребуется явно указывать значение столбца DEPT\_TOTAL\_SAL. Хотя формально у столбца будет существовать значение по умолчанию, наследуемое от домена SALARY (10000.00), оно не может быть занесено в таблицу DEPT, поскольку противоречит ограничению столбца DEPT\_TOTAL\_SAL CHECK (VALUE >= 100000.00).

# ПРЕДСТАВЛЕНИЯ (VIEW) (2/2)



Практическое использование в бизнес-архитектурах



Краткое упоминание:

При работе с БД на клиенте, внутри БД, занимаясь исключительно запросами, работаем с файловым дескриптором – работаем внутри файловой системы

При работе приложения, клиент-серверная архитектура – работаем с дескриптором соединения, курсором, запросы передаются, результаты возвращаются через соединение, а не крутятся внутри файловой системы на клиенте.

Для автоматизации соединения создаем connection, передаем cursor, который соответствует одной транзакции.

# ПРИМЕР БД С ТЕСТАМИ ПРИБОРОВ



```
ivasonik@DESKTOP-802FCMA:~$ nano schema.sql
```

schema.sql

```
CREATE table devices (  
    id INTEGER PRIMARY KEY AUTOINCREMENT,  
    sn TEXT NOT NULL UNIQUE  
);  
  
CREATE TABLE tests (  
    id INTEGER PRIMARY KEY AUTOINCREMENT,  
    ts INTEGER NOT NULL,  
    device_id INTEGER NOT NULL REFERENCES devices(id),  
    result INTEGER --0 - fail, 1 - passed  
);
```

```
ivasonik@DESKTOP-802FCMA:~$ sqlite3 dev-tests.s3db <schema.sql
```