

Представления. СТЕ.  
Безопасность типов. JOIN


# Представление

- VIEW – сохраненный запрос в виде объекта БД
- К VIEW можно делать обычный SELECT
- VIEWS можно соединять
- Позволяет сокращать сложные запросы
- Позволяет делать кеширование с помощью материализации
- Позволяет подменить реальную таблицу
- Позволяет создавать виртуальные таблицы соединяющие несколько таблиц
- Позволяет скрыть информацию (столбцы) от групп пользователей

# Представления

- При работе с базами данных зачастую приходится многократно выполнять одни и те же запросы, которые могут быть весьма сложными и требовать обращения к нескольким таблицам. Чтобы избежать необходимости многократного формирования таких запросов, можно использовать так называемые **представления (views)**.
- Если речь идет о выборке данных, то представления практически неотличимы от таблиц с точки зрения обращения к ним в командах SELECT.
- Упрощенный синтаксис команды CREATE VIEW, предназначенной для создания представлений, таков:

```
CREATE VIEW name [ ( column_name [, ...] ) ]  
AS query;
```



необязательные  
элементы команды

Если список имен столбцов не приведен, тогда их имена «вычисляются» (формируются) на основании текста запроса.

# Представление

- Временные
- Рекурсивные
- Обновляемые
- Материализуемые

```
CREATE OR REPLACE VIEW myview AS
SELECT name, temp_lo, temp_hi, prcp, date, location
FROM weather, cities
WHERE city = name;

SELECT * FROM myview;
```

## Изменение представлений:

- Можно добавлять новые столбцы в конец
- Нельзя: удалять существующие, менять имена столбцов, менять порядок следования столбцов

Задача: подсчитать количество мест в салонах для всех моделей самолетов с учетом класса обслуживания (бизнес-класс и экономический класс).

```
CREATE VIEW seats_by_fare_cond AS
  SELECT aircraft_code, fare_conditions, count( * )
  FROM seats
  GROUP BY aircraft_code, fare_conditions
  ORDER BY aircraft_code, fare_conditions;
```

Теперь мы можем вместо написания сложного первоначального запроса обращаться непосредственно к представлению, как будто это обычная таблица.

```
SELECT * FROM seats_by_fare_cond;
```

**ВАЖНО!** В отличие от таблиц, представления не содержат данных. Данные выбираются из таблиц, на основе которых представление создано, при каждом обращении к нему в команде SELECT.

- Предложение OR REPLACE – это *расширение* команды CREATE VIEW, которое предлагает PostgreSQL. Однако нужно помнить о том, что при создании новой версии представления (без явного удаления старой с помощью команды DROP VIEW) должны оставаться неизменными имена столбцов представления.
- Обратите внимание на добавление фразы OR REPLACE и ключевого слова AS после вызова функции count:

```
CREATE OR REPLACE VIEW seats_by_fare_cond AS
SELECT a.model, s.aircraft_code, s.fare_conditions,
       count( * ) AS num_seats
FROM seats
GROUP BY aircraft_code, fare_conditions
ORDER BY aircraft_code, fare_conditions;
```

ОШИБКА: изменить имя столбца "count" на "num\_seats" в представлении нельзя

- А дело в том, что при первоначальном создании этого представления третий столбец уже получил имя count (такое имя ему дала СУБД).
- Сначала следует удалить это представление, а затем создать его заново.

```
DROP VIEW seats_by_fare_cond;  
CREATE OR REPLACE VIEW seats_by_fare_cond AS  
    SELECT a.model, s.aircraft_code, s.fare_conditions,  
           count( * ) AS num_seats  
    ...
```

Второй способ задания имен столбцов в представлении — с помощью списка их имен, заключенного в скобки:

```
DROP VIEW seats_by_fare_cond;  
CREATE OR REPLACE VIEW seats_by_fare_cond  
    ( code, fare_cond, num_seats )  
AS  
    SELECT aircraft_code, fare_conditions, count( * )  
    FROM seats  
    GROUP BY aircraft_code, fare_conditions  
    ORDER BY aircraft_code, fare_conditions;
```

# Представление

**CREATE OR REPLACE VIEW** Price\_View2 AS

SELECT price, name

FROM Book

INNER JOIN Price

ON Book.id = Price.id

WHERE price > 200;

SELECT \* FROM Price\_View2;

SELECT \* FROM Price\_View2 WHERE name LIKE '%ап%';



# Представление

**ALTER VIEW** old\_view\_name **RENAME TO** new\_view\_name

## **Модификация данных через представления**

- Только одна таблица в FROM
- Нет DISTINCT, GROUP BY, HAVING, UNION, EXCEPT, LIMIT
- Нет оконных функций, MIN, MAX, SUM, COUNT, AVG
- WHERE не под запретом

# Представление

**CREATE OR REPLACE VIEW** Price\_View2 AS

SELECT \*

FROM Book

WHERE price > 200;

INSERT INTO Price\_View2 VALUES (1, 'Война и Мир', 'Толстой Л.Н.', 500)

**CREATE OR REPLACE VIEW** Price\_View2 AS

SELECT \*

FROM Book

WHERE price > 200

WITH LOCAL [CASCADED] CHECK OPTION; - добавление данных только удовлетворяющих условию

INSERT INTO Price\_View2 VALUES (1, 'Война и Мир', 'Толстой Л.Н.', 100) **ОШИБКА**

DELETE FROM Price\_View2 WHERE price<300

PostgreSQL предлагает свое расширение — так называемое **материализованное представление**. Упрощенный синтаксис команды для создания материализованных представлений, таков:

```
CREATE MATERIALIZED VIEW [ IF NOT EXISTS ] table_name  
[ (column_name [, ...] ) ]  
AS query  
[ WITH [ NO ] DATA ] ;
```

В квадратных скобках  
необязательные элементы команды

- Материализованное представление заполняется данными *в момент выполнения команды для его создания*, если только в команде не было фразы WITH NO DATA.
- Если же она была включена в команду, тогда в момент своего создания представление данными не заполняется, а для заполнения его данными нужно использовать команду

**REFRESH MATERIALIZED VIEW**

Материализованное представление очень похоже на обычную таблицу. Однако оно отличается от таблицы тем, что не только сохраняет данные, но также *запоминает запрос*, с помощью которого эти данные были собраны.

# Представление

```
CREATE TABLE invoice (  
    invoice_no integer PRIMARY KEY,  
    seller_no integer, -- идентификатор продавца  
    invoice_date date, -- дата продажи  
    invoice_amt numeric(13,2) -- сумма продажи  
);
```

```
CREATE MATERIALIZED VIEW sales_summary AS  
SELECT  
    seller_no,  
    invoice_date,  
    sum(invoice_amt)::numeric(13,2) as sales_amt  
FROM invoice  
WHERE invoice_date < CURRENT_DATE  
GROUP BY  
    seller_no,  
    invoice_date;
```

Хотя обращение к данным в материализованном представлении часто выполняется гораздо быстрее, чем обращение к нижележащим таблицам напрямую или через представление, данные в нём не всегда актуальные.

# Что дают представления? (1)

1. Упрощение разграничения полномочий пользователей на доступ к хранимым данным.

Разным типам пользователей могут требоваться различные данные, хранящиеся в одних и тех же таблицах. Это касается как столбцов, так и строк таблиц. Создание различных представлений для разных пользователей избавляет от необходимости создавать дополнительные таблицы, дублируя данные, и упрощает организацию системы управления доступом к данным.

2. Упрощение запросов к базе данных.

Использование представлений позволяет скрыть сложные запросы от прикладного программиста и сделать запросы более простыми и наглядными.

### 3. Снижение зависимости прикладных программ от изменений структуры таблиц базы данных.

Столбцы представления, т. е. их имена, типы данных и порядок следования, — это, образно говоря, интерфейс к запросу, который реализуется данным представлением. Если этот интерфейс остается неизменным, то SQL-запросы, в которых используется данное представление, корректировать не потребуется. Нужно будет лишь в ответ на изменение структуры базовых таблиц, на основе которых представление сконструировано, соответствующим образом перестроить запрос, выполняемый данным представлением.

### 4. Снижение времени выполнения сложных запросов за счет использования материализованных представлений.

Если, например, какой-нибудь сводный отчет формируется длительное время, а запросы к отчету будут неоднократными, то может оказаться целесообразным сформировать его заранее и сохранить в материализованном представлении

Недостаток материализованных представлений: необходимо своевременно обновлять с помощью команды REFRESH, чтобы они содержали актуальные данные.

PostgreSQL Common Table Expressions, или сокращенно CTE, предоставляет временную таблицу на уровне операторов, которая помогает создавать сложные и понятные SQL-выражения.

```
WITH [RECURSIVE] cte_name [(cte_column_list)] AS (  
    cte_definition  
)  
primary_statement;
```

```
CREATE TABLE category (  
    id SERIAL PRIMARY KEY,  
    name VARCHAR NOT NULL,  
    parent_id INT,  
    CONSTRAINT fk_category  
        FOREIGN KEY(parent_id) REFERENCES category(id)  
);
```

Синтаксис начинаются с ключевого слова **WITH**  
**RECURSIVE**

Ключевое слово указывает на то, что это общее табличное выражение является рекурсивным. Это необязательно.

**cte\_name**

Имя общего табличного выражения, эквивалентное имени временной таблицы.

**cte\_column\_list**

Это список имен столбцов для общего табличного выражения с несколькими именами столбцов, разделенными запятыми. Это необязательно.

**cte\_definition**

Оператор является оператором для общего табличного выражения, которое может быть инструкцией SELECT, INSERT, UPDATE или DELETE.

**primary\_statement**

Это основной оператор, использующий **cte\_name**. Это может быть SELECT, INSERT, UPDATE или DELETE.

Мы определяем обобщенное табличное выражение по формуле, а затем находим строки из обобщенного табличного выражения

Запросить категорию 2 и все ее подкатегории

```
WITH RECURSIVE cte_categories AS (  
  SELECT  
    id,  
    name,  
    parent_id  
  FROM category  
  WHERE id = 2  
  UNION  
  SELECT  
    c.id,  
    c.name,  
    c.parent_id  
  FROM category c, cte_categories cs  
  WHERE cs.id = c.parent_id  
)  
SELECT *  
FROM cte_categories;
```



# COALESCE и NULLIF

- COALESCE (arg1,arg2,...) – возвращает первый аргумент, который не null
- NULLIF (arg1,arg2) – сравнивает два аргумента, если они равны возвращает null, иначе arg1

select поле, поле, coalesce (поле, поле\_2, 'значение') as псевдоним  
from таблица;

Возвращает значение поля, если оно не равно NULL, либо поле\_2, если оно не равно NULL, иначе - 'значение'

select поле, поле, coalesce (NULLIF(поле, ' '), 'значение') as псевдоним  
from таблица;

NULLIF возвращает значение NULL, если поле равно пустой строке, иначе значение поля

# COALESCE и NULLIF

```
CREATE TABLE budgets
(
    dept serial,
    current_year decimal, // текущий год
    previous_year decimal // предыдущий год
);
```

```
INSERT INTO budgets (current_year, previous_year) VALUES(100000, 150000);
INSERT INTO budgets (current_year, previous_year) VALUES(NULL, 300000);
INSERT INTO budgets (current_year, previous_year) VALUES(0, 100000);
INSERT INTO budgets (current_year, previous_year) VALUES(NULL, 150000);
INSERT INTO budgets (current_year, previous_year) VALUES(300000, 250000);
INSERT INTO budgets (current_year, previous_year) VALUES(170000, 170000);
INSERT INTO budgets (current_year, previous_year) VALUES(150000, NULL);
```

```
SELECT dept, COALESCE(TO_CHAR(NULLIF(current_year, previous_year), 'FM99999999'), 'Одинаково') AS budget
FROM budgets
WHERE current_year IS NOT NULL
```

# Безопасность типов

- SQL – строго типизированный язык
- Разрешена перегрузка функций
- Информация о существующих преобразованиях типов, для каких типов они определены и как их выполнять, хранится в системных каталогах
- Когда применяется оператор к двум операндам, интерпретатор ищет оператор в таблице `pg_operator` и проверяет может ли он работать с типами операндов
- Если типы между собой совместимы – интерпретатор старается произвести неявное преобразование
- Результат неявных преобразования всегда должен быть предсказуемым и понятным

# Безопасность типов

- Для явного преобразования используются:

`CAST (expression AS target_type)` – совместимо стандартом

`expression::target_type` – несовместимо со стандартом

```
SELECT CAST('1000' as INTERVAL);
```

```
interval  
00:16:40
```

```
SELECT 'abc' || 1 //abc1
```

```
SELECT '10' + 10 //20
```

```
SELECT CAST(24000 AS money);
```

```
SELECT 24000::money;
```

```
money  
$24,000.00
```

```
SELECT CAST('October 18, 2023' as DATE);
```

```
date  
2023-10-18
```

```
SELECT 'Сегодня' || CAST(CURRENT_DATE AS TEXT);
```

```
?column?  
Сегодня 2023-11-08
```

```
SELECT CAST(TRIM(LEADING 'DATE' FROM 'DATE October 18, 2023') AS DATE);
```

```
date  
2023-10-18
```

# Правила для INSERT, UPDATE и DELETE

Правила, определяемые для команд INSERT, UPDATE и DELETE, значительно отличаются от правил представлений. Во-первых, команда **CREATE RULE** позволяет создавать правила со следующими особенностями:

- Они могут не определять действия.
- Они могут определять несколько действий.
- Они могут действовать в режиме INSTEAD или ALSO (по умолчанию).
- Становятся полезными псевдоотношения NEW и OLD.
- Они могут иметь условия применения.

# Правила для INSERT, UPDATE и DELETE

```
CREATE [ OR REPLACE ] RULE имя AS ON событие  
    TO таблица [ WHERE условие ]  
    DO команда ;
```

```
CREATE TABLE log (  
    name text,          -- товары, количество которых изменилось  
    avail integer,      -- новое количество  
    log_who text,       -- кто изменил  
    log_when timestamp  -- когда  
);
```

# Правила для INSERT, UPDATE и DELETE

```
CREATE [ OR REPLACE ] RULE имя AS ON событие  
  TO таблица [ WHERE условие ]  
  DO команда ;
```

```
CREATE RULE log AS ON UPDATE TO tovar_data  
  WHERE NEW.avail <> OLD.avail  
  DO INSERT INTO log VALUES (  
    NEW.name,  
    NEW.avail,  
    current_user,  
    current_timestamp  
  );
```

```
UPDATE tovar_data SET avail = 6 WHERE name = 'sl7';
```

```
SELECT * FROM log;
```

name	avail	log_who	log_when
sl7	6	Al	Tue Oct 20 16:14:45 2022 MET DST

(1 row)

## Типы JOIN и примеры

cross join (CROSS JOIN)

inner join (INNER JOIN)

natural join (NATURAL JOIN)

left outer join (LEFT [OUTER] JOIN)

right outer join (RIGHT [OUTER] JOIN)

full outer join (FULL [OUTER] JOIN)

```
CREATE TABLE student (  
    student_id INTEGER NOT NULL,  
    name varchar(45) NOT NULL,  
    PRIMARY KEY (student_id)  
);
```

```
CREATE TABLE student_score (  
    student_id INTEGER NOT NULL,  
    subject varchar(45) NOT NULL,  
    score INTEGER NOT NULL  
);
```

```
INSERT INTO  
    student (student_id, name)  
VALUES  
    (1,'Иванов Иван'),(2,'Петров Петр'),(3,'Егоров Егор');
```

```
INSERT INTO  
    student_score (student_id, subject, score)  
VALUES  
    (1,'Информатика',90),  
    (1,'Матан',80),  
    (2,'Информатика',85),  
    (5,'Информатика',92);
```








# Типы JOIN и примеры

## Cross Join

**Перекрестное соединение** возвращает декартово произведение двух множеств. То есть все возможные комбинации всех строк в обеих таблицах. Это эквивалентно внутреннему соединению без условия соединения или условию соединения, которое всегда истинно.

В большинстве случаев результат перекрестного соединения не имеет смысла, и необходимо использовать предложение WHERE для фильтрации нужных строк.

```
SELECT
  student.*,
  student_score.*
FROM
  student CROSS JOIN student_score;
```

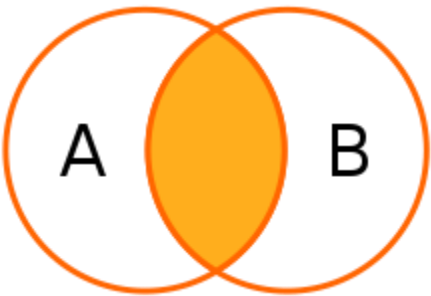
	 student_id integer	 name character varying (45)	 student_id integer	 subject character varying (45)	 score integer
1	1	Иванов Иван	1	Информатика	90
2	1	Иванов Иван	1	Матан	80
3	1	Иванов Иван	2	Информатика	85
4	1	Иванов Иван	5	Информатика	92
5	2	Петров Петр	1	Информатика	90
6	2	Петров Петр	1	Матан	80
7	2	Петров Петр	2	Информатика	85
8	2	Петров Петр	5	Информатика	92
9	3	Егоров Егор	1	Информатика	90
10	3	Егоров Егор	1	Матан	80
11	3	Егоров Егор	2	Информатика	85
12	3	Егоров Егор	5	Информатика	92

# Типы JOIN и примеры

## Inner Join

**Внутренние соединения** объединяют строки из двух таблиц на основе условий соединения. Внутренние соединения эквивалентны перекрестным соединениям с добавленными условиями фильтра.

Внутреннее соединение сравнивает каждую строку первой таблицы с каждой строкой второй таблицы и, если выполняется заданное условие соединения, объединяет строки двух таблиц в результирующий набор.



```
SELECT
  student.*,
  student_score.*
FROM
  student
  INNER JOIN student_score
  ON student.student_id = student_score.student_id;
```

```
SELECT
  student.*,
  student_score.*
FROM
  student
  INNER JOIN student_score USING(student_id);
```

	<div>student_id</div> <div>integer</div>	<div>name</div> <div>character varying (45)</div>	<div>student_id</div> <div>integer</div>	<div>subject</div> <div>character varying (45)</div>	<div>score</div> <div>integer</div>
1	1	Иванов Иван	1	Информатика	90
2	1	Иванов Иван	1	Матан	80
3	2	Петров Петр	2	Информатика	85

# Типы JOIN и примеры

## Natural Join

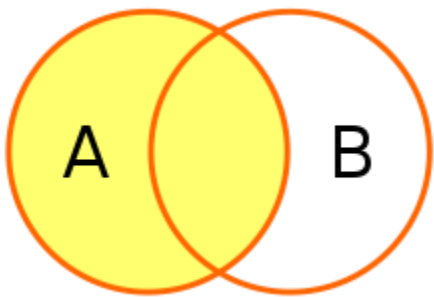
**Естественные соединения** также являются соединениями на основе условий, которые являются особым видом внутренних соединений. При естественном соединении двух таблиц все столбцы с одинаковыми именами в двух таблицах будут сравниваться на равенство. Эти условия соединения создаются неявно.

```
SELECT
*
FROM
student NATURAL JOIN student_score;
```

	<div>student_id</div> <div>integer</div>	<div>name</div> <div>character varying (45)</div>	<div>subject</div> <div>character varying (45)</div>	<div>score</div> <div>integer</div>
1	1	Иванов Иван	Информатика	90
2	1	Иванов Иван	Матан	80
3	2	Петров Петр	Информатика	85

# Типы JOIN и примеры

## Left Join



**Левое соединение** — это сокращение от левого внешнего соединения, левое соединение требует условий соединения.

Когда две таблицы соединены слева, первая таблица называется левой таблицей, а вторая таблица называется правой таблицей.

Левое соединение основано на строках левой таблицы и соответствует каждой строке правой таблицы в соответствии с условием соединения. Если совпадение прошло успешно, строки левой и правой таблиц объединяются в новую строку и возвращаются; если совпадение не удалось, строки левой таблицы и значения NULL правой таблицы объединяются в новую строку возвращаемых данных.

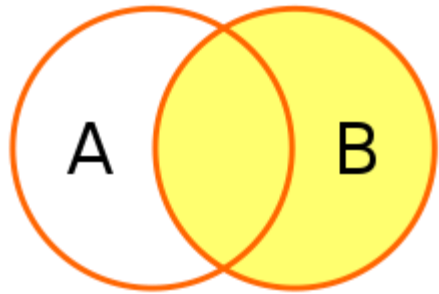
```
SELECT
  student.name AS ФИО,
  student_score.subject AS Дисциплина,
  COALESCE (TO_CHAR(student_score.score, 'FM99999999'), 'Отсутствует') AS Балл
FROM
  student
LEFT JOIN student_score
ON student.student_id = student_score.student_id;
```

	ФИО character varying (45)	Дисциплина character varying (45)	Балл text
1	Иванов Иван	Информатика	90
2	Иванов Иван	Матан	80
3	Петров Петр	Информатика	85
4	Егоров Егор	[null]	Отсутствует

```
FROM
  student
LEFT JOIN student_score USING(student_id);
```

## Типы JOIN и примеры

### Right Join



**Правое соединение** является сокращением от правого внешнего соединения, правое соединение требует условия соединения.

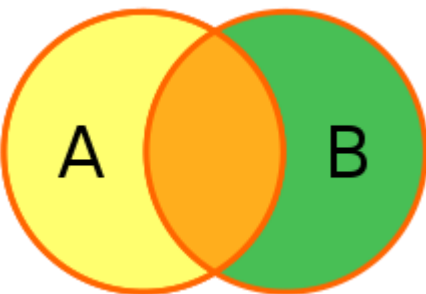
Логика обработки правого соединения противоположна левому. Правое соединение основано на строках правой таблицы и соответствует данным в левой таблице в соответствии с условиями. Если данные в левой таблице не могут быть сопоставлены, столбец в левой таблице является значением .NULL

```
SELECT
  COALESCE (student.name, 'Фамилия отсутствует') AS ФИО,
  student_score.subject AS Дисциплина,
  student_score.score AS Балл
FROM
  student
  RIGHT JOIN student_score
    ON student.student_id = student_score.student_id;
```

	ФИО character varying	Дисциплина character varying (45)	Балл integer
1	Иванов Иван	Информатика	90
2	Иванов Иван	Матан	80
3	Петров Петр	Информатика	85
4	Фамилия отсутствует	Информатика	92

# Типы JOIN и примеры

## Full Join



**Полное соединение** является сокращением от полного внешнего соединения, которое представляет собой объединение левого и правого соединения. Для полного соединения требуется условие соединения.

```
SELECT
  student.*,
  student_score.*
FROM
  student
FULL JOIN student_score
ON student.student_id = student_score.student_id;
```

```
FROM
  student
FULL JOIN student_score USING(student_id);
```

	<div>student_id</div> <div>integer</div>	<div>name</div> <div>character varying (45)</div>	<div>student_id</div> <div>integer</div>	<div>subject</div> <div>character varying (45)</div>	<div>score</div> <div>integer</div>
1	1	Иванов Иван	1	Информатика	90
2	1	Иванов Иван	1	Матан	80
3	2	Петров Петр	2	Информатика	85
4	[null]	[null]	5	Информатика	92
5	3	Егоров Егор	[null]	[null]	[null]