



# ЛЕКЦИЯ 5

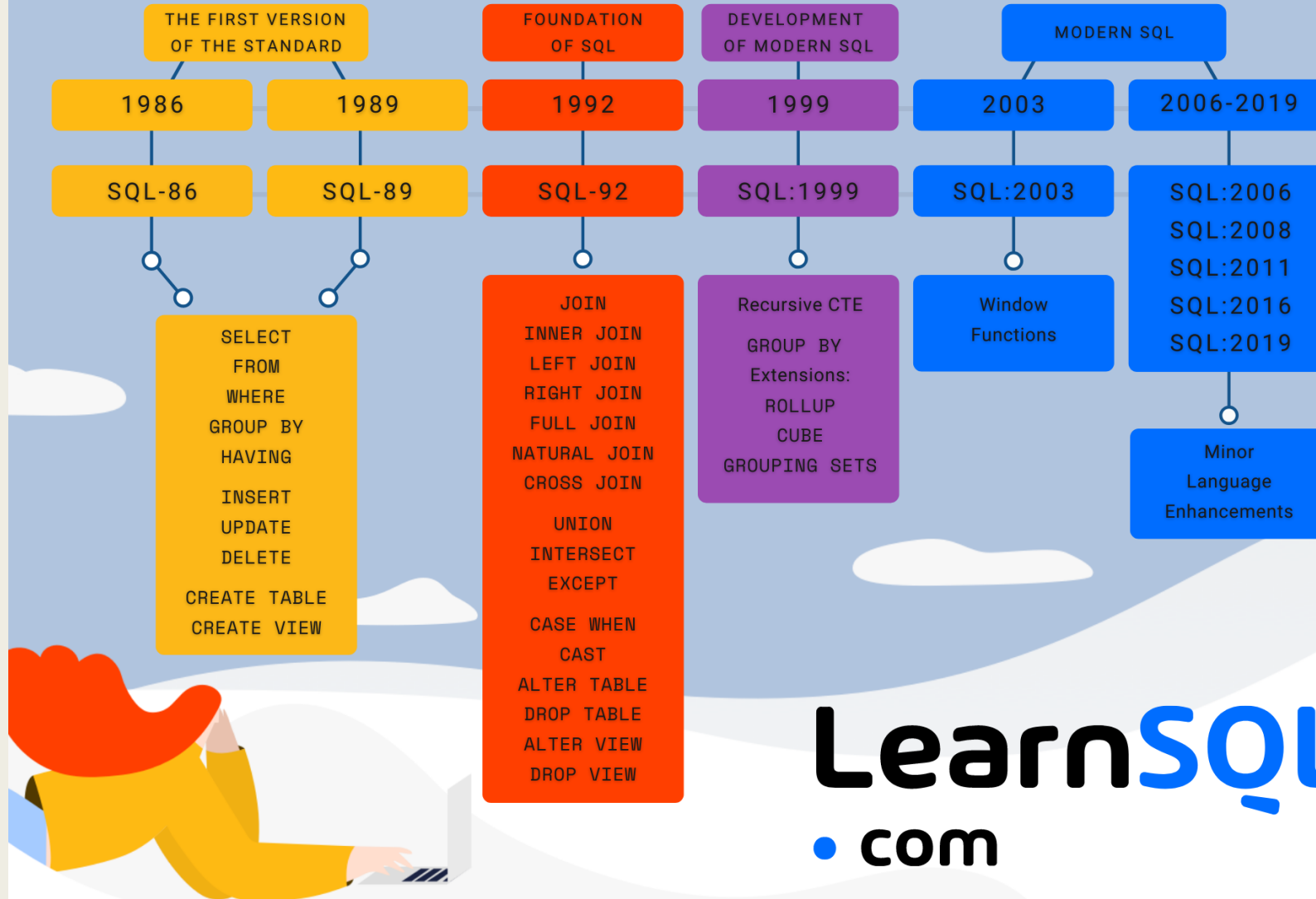
Стандарт SQL



# Цель лекции

- посмотреть, как развивался стандарт SQL
- изучить то, что до этого изучено не было

# The History of SQL Standards



# SQL-86 (SQL 1)

- синтаксис для выражений SELECT, INSERT, UPDATE, DELETE
- SELECT позволял фильтровать по WHERE, группировать по GROUP BY по нескольким колонкам (но не по выражению) и фильтровать группы с помощью HAVING
- как такового нет синтаксиса JOIN, но можно использовать FROM (несколько таблиц) и WHERE
- SELECT поддерживает подзапросы

# SQL-86 (SQL 1)

- стандарт определения таблиц: создавать таблицы, создавать view, назначать привилегии
- ограничение на столбцы: UNIQUE и NOT NULL
- типы данных: строки и числа (NUMERIC, DECIMAL, INTEGER, SMALLINT, FLOAT, REAL, DOUBLE PRECISION)
- нет выражений: ALTER, DROP, REVOKE (отмена разрешения)
- SQL встраивается в несколько языков программирования (Cobol, Fortran, Pascal, PL/1)
- Что не рассмотрели ранее: VIEW

# VIEW (Представление)

- View — представление/виртуальная таблица
- View — специфический образ таблицы или набора таблиц, определенный оператором SELECT
- View всегда показывает самую свежую информацию из базы данных
- представление не существует физически как обычная таблица

# Команды SQL

- Создание представления:

```
CREATE VIEW <view_name> AS  
SELECT <col_name1>, <col_name2>, ...  
FROM <table_name>  
WHERE <condition>;
```

- Удаление представления:

```
DROP VIEW <view_name>;
```

# Зачем нужны представления?

- если есть длинный запрос, его можно один раз написать и использовать повторно в других случаях
- разделение логики хранения данных и ПО (если таблица изменилась, но приложению нужны данные в старом виде)
- ограничение доступа к данным (поле "владелец" совпадает с именем пользователя, который делает запрос)



# То же самое другими словами

- уменьшение сложности
- повышение безопасности
- повышение удобства
- переименование столбцов таблицы
- настройка данных для пользователей
- защита целостности данных

# Особенности View

- View можно создавать на основе нескольких таблиц

```
CREATE VIEW v AS
    SELECT a.id, b.id FROM a, b;
-- ERROR: column "id" specified more than once
-- SQL-состояние: 42701
```

```
CREATE VIEW v (a_id, b_id) AS
    SELECT a.id, b.id FROM a,b;
CREATE VIEW v AS
    SELECT a.id a_id, b.id b_id FROM a,b;
```

# Изменение View

- над простыми представлениями можно выполнять операции INSERT, UPDATE, DELETE
- если представление состоит из нескольких таблиц, в представлении есть агрегатные функции, то представление изменить нельзя

# Изменение View

```
CREATE TABLE films (  
    imdb      varchar(16) PRIMARY KEY,  
    title     varchar(40) NOT NULL,  
    kind      varchar(10)  
);
```

```
CREATE OR REPLACE VIEW comedies AS  
    SELECT *  
    FROM films  
    WHERE kind = 'Comedy'  
    WITH CASCADED CHECK OPTION;
```

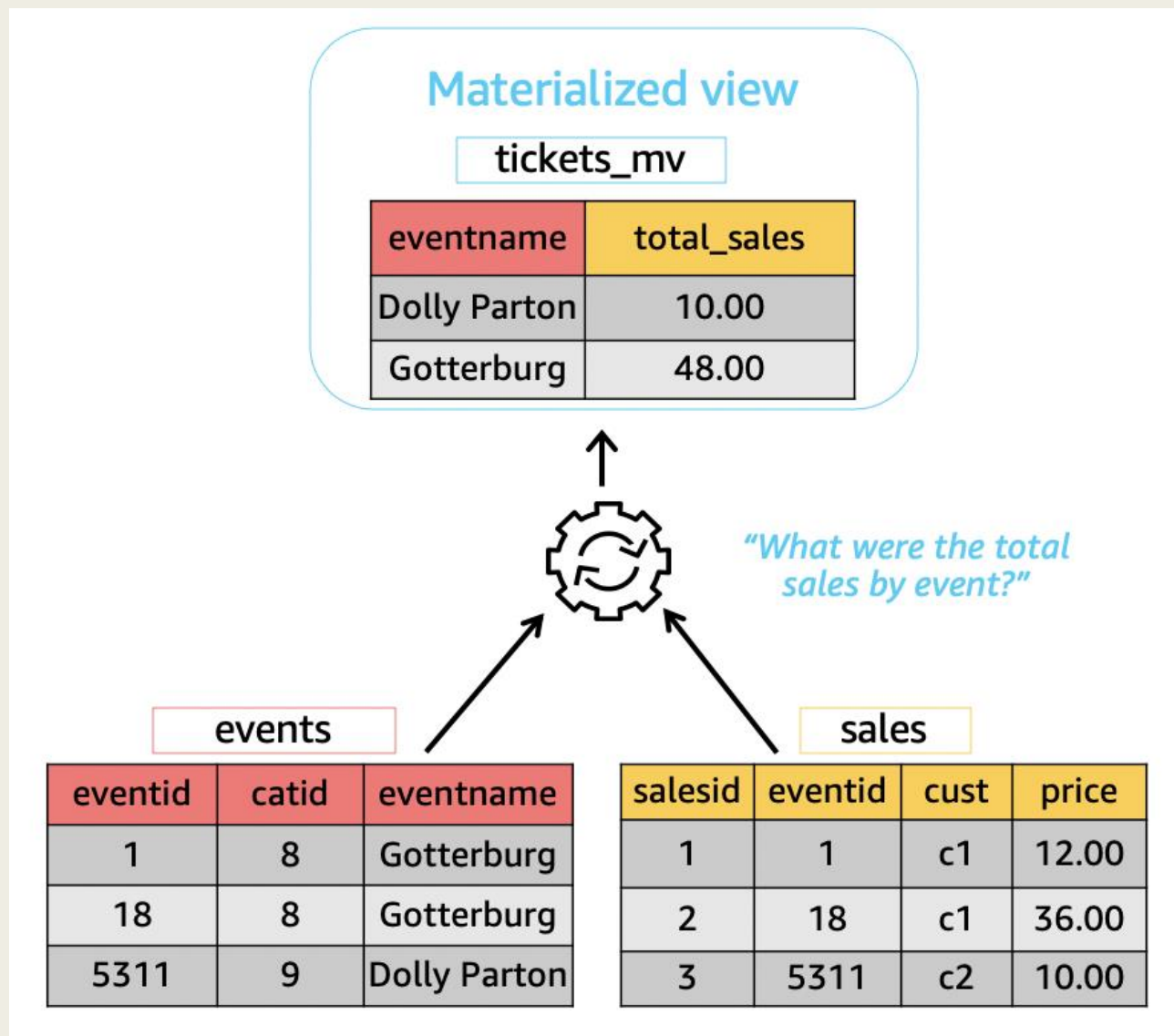
```
INSERT INTO comedies (imdb, title, kind)  
    VALUES ('tt0114709', 'Toy Story 1', 'Animation');  
-- ERROR:  new row violates check option for view "comedies"  
-- DETAIL:  Failing row contains (tt0114709, Toy Story 1, Animation).
```

```
INSERT INTO comedies (imdb, title, kind)  
    VALUES ('tt1156398', 'Zombieland', 'Comedy');
```

# Materialized view

- при каждом обращении БД выполняет запрос, по которому определено представление, и возвращает результат - процесс наполнения представления
- повторяется при каждом обращении
- если нужно часто обращаться к представлению, то можно использовать материализованное представление
- у materialized view есть физическое воплощение
- в postgresql появилось в версии 9.3 (2013г)

# Пример



# Команды SQL

- Создание материализованного представления (postgresql)

```
CREATE MATERIALIZED VIEW mymatview AS SELECT *  
FROM mytab;
```

- Актуализация

```
REFRESH MATERIALIZED VIEW mymatview;
```

- Материализованное представление нельзя изменить непосредственно

# Materialized View vs View

- View - логический объект, Materialized view - физический объект
- view - доступны все операции работы с данными (select, update, insert, delete), materialized view - только select
- если удалить таблицу, на которой основана view, то view станет невалидной. Materialized view останется доступной
- все операции над основной таблицей сразу доступны в view. Materialized view нужно обновить для получения актуального состояния



# Materialized View vs View

Выбирать материализованное представление, если всё перечисленное верно:

- результат запроса не часто меняется (вся таблица или только подмножество запроса представления)
- результат часто используется (гораздо чаще, чем меняется таблица)
- выполнение запроса потребляет много ресурсов (время, память и т.д.)

# Materialized View vs View

Использовать обычное представление, если верно хотя бы одно:

- результат часто меняется
- результат не так часто используется (сопоставимо с изменением)
- запрос не требует большого количества ресурсов

# SQL-89

- небольшие доработки стандарта SQL-86
- добавление ограничений на столбцы (было только UNIQUE и NOT NULL), добавились: первичные и внешние ключи, DEFAULT и CHECK
- добавлено расширение для языков C и Ada

# SQL-92 (SQL-2)

- появились соединения таблиц (внутренние и внешние) - JOIN, LEFT JOIN, RIGHT JOIN, FULL JOIN, NATURAL JOIN, CROSS JOIN
- операции над множествами: UNION, INTERSECT, EXCEPT
- скалярные операции: соединение строк, извлечение подстрок, манипуляции с датой-временем
- операция условия CASE WHEN
- приведение типов CAST

# SQL-92 (SQL-2)





- новые типы данных: date, time, timestamp, interval, bit string, varchar string, national character strings
- добавлена возможность изменять и удалять таблицы и view: ALTER и DROP
- информационная схема: способ получить метаданные таблицы - название таблиц, столбцов, типы данных столбцов, ограничения
- временные таблицы, уровень изоляции транзакций, dynamic SQL

# SQL-92 (SQL-2)

- обычно, когда говорят SQL, подразумевают именно SQL-92.
- этого достаточно для повседневной работы
- большинство СУБД поддерживают SQL-92 (ни одна СУБД не поддерживает стандарт на 100%)

# Повторим JOIN

Виды соединений: (INNER) JOIN, LEFT (OUTER) JOIN, RIGHT JOIN, FULL JOIN, NATURAL JOIN, CROSS JOIN

Table	Join Type			Table	Statement	What we use	Visualization
A		Inner	Join	B	A Inner Join B	A Inner Join B	
	left	Outer			A Left Outer Join B	A Left Join B	
	Full				A Full Outer Join B	A Full Join B	
	right				A Righ Outer Join B	A Righ Join B	
		Cross			A Cross Outer Join B	A Cross Join B	Rarely being used
		Natural			A Natural Join B	A Natural Join B	

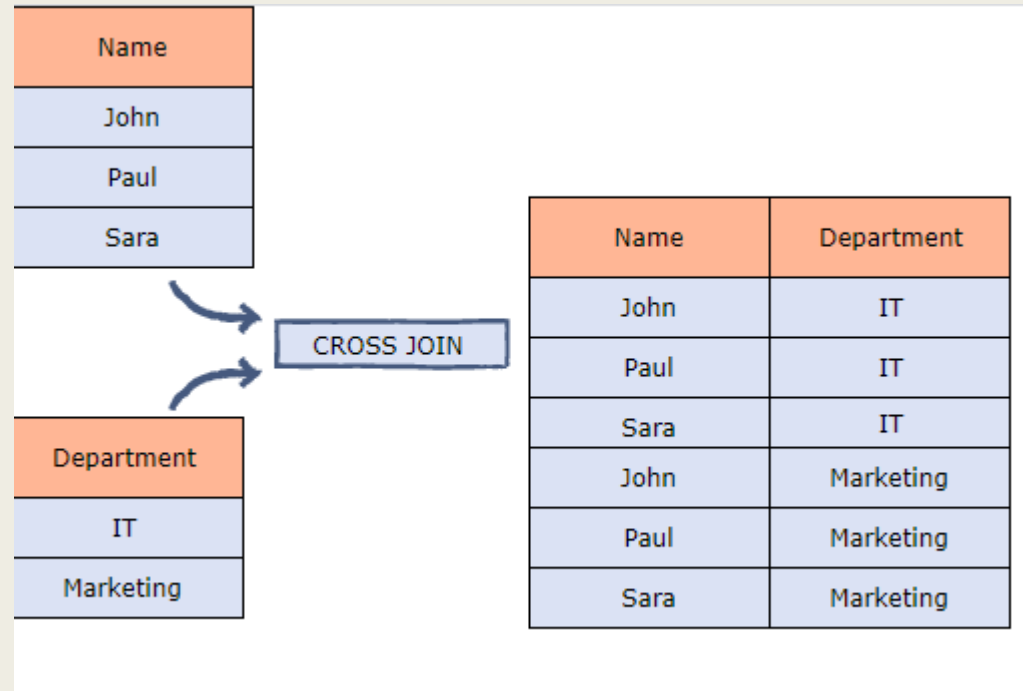
Outer is optional

Дополнительно: Статья "Понимание джойнов сломано. Это точно не пересечение кругов, честно"

<https://habr.com/ru/post/448072/>

Тут есть примеры табличек: <https://www.datacamp.com/community/tutorials/introduction-to-sql-joins>

# CROSS JOIN



```
SELECT    [column names]
FROM      [Table1]
CROSS JOIN [Table2]
```

```
SELECT [column names]
FROM   [Table1], [Table2]
```



# NATURAL JOIN

- JOIN происходит по всем столбцам с одинаковыми названиями
- не требует условия ON
- дублирующие название колонок удаляются из результата

```
SELECT id, aval1, cval1 FROM table1  
NATURAL JOIN table2;
```

```
SELECT table1.id, table1.aval1, table2.cval1  
FROM table1 INNER JOIN table2  
ON table1.id=table2.id;
```

SELECT \*  
FROM table\_a  
NATURAL JOIN table\_b;

common column

id	des1	des2
100	desc11	desc12
101	desc21	desc22
102	desc31	desc32

id	des3	des4
101	desc41	desc42
103	desc51	desc52
105	desc61	desc62

only matching row  
based on common column

id	des1	des2
100	desc11	desc12
101	desc21	desc22
102	desc31	desc32

id	des3	des4
101	desc41	desc42
103	desc51	desc52
105	desc61	desc62

id	des1	des2	id	des3	des4
101	desc21	desc22	101	desc41	desc42

# Строковые функции SQL

- CONCAT - соединение строк

```
SELECT CONCAT ('One', ' ', 'two');
```

Результат: One two

- REPLACE - замена подстроки новой строкой

```
SELECT REPLACE('From this string. This is great!', 'is',  
'was');
```

Результат: From thwas string. Thwas was great!

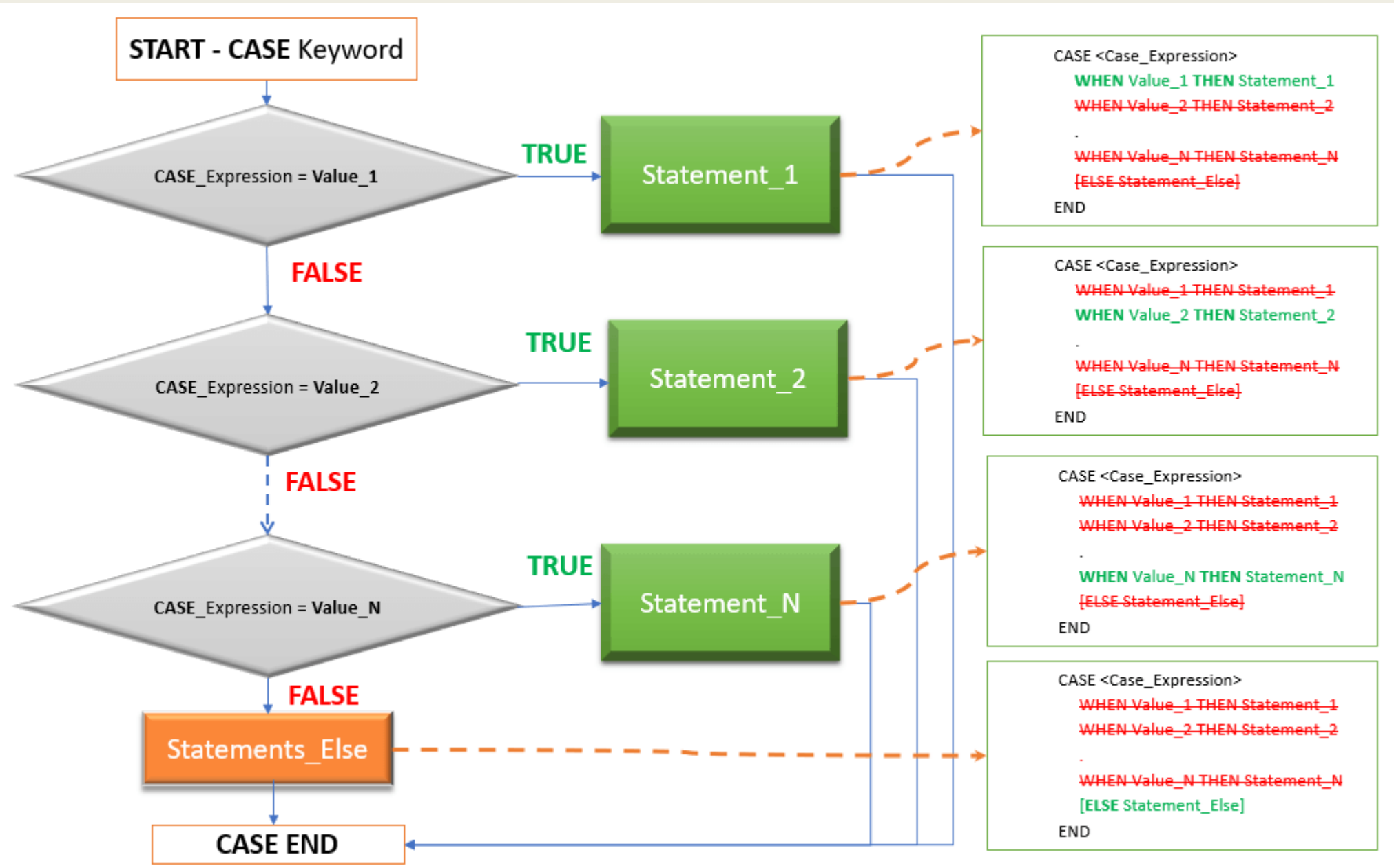
- получение подстроки SUBSTR

```
SUBSTR(char, position, length)
```

# CASE WHEN

- Разные значения для разных условий
- Синтаксис:

```
CASE <Case_Expression>  
    WHEN Value_1 THEN Statement_1  
    WHEN Value_2 THEN Statement_2  
    .  
    WHEN Value_N THEN Statement_N  
    [ELSE Statement_Else]  
END AS [ALIAS_NAME]
```



# Пример

```
SELECT Tutorial_ID, Tutorial_name,  
CASE Tutorial_name  
    WHEN 'SQL' THEN 'SQL is developed by IBM'  
    WHEN 'PL/SQL' THEN 'PL/SQL is developed by Oracle Corporation.'  
    WHEN 'MS-SQL' THEN 'MS-SQL is developed by Microsoft Corporation.'  
    ELSE 'This is NO SQL language.'  
END AS Description  
FROM my_table
```

# Пример 2

```
UPDATE my_table
SET Tutorial_Name =
(
CASE
WHEN Tutorial_Name = 'SQL' THEN 'Structured Query language.'
WHEN Tutorial_Name = 'PL/SQL' THEN 'Oracle PL/SQL'
WHEN Tutorial_Name = 'MSSQL' THEN 'Microsoft SQL.'
WHEN Tutorial_Name = 'Hadoop' THEN 'Apache Hadoop.'
END
)
```

# Временные таблицы

- Временные таблицы автоматически удаляются в конце сеанса или могут удаляться в конце текущей транзакции
- Могут быть удалены вручную до окончания сессии
- Нужны для хранения промежуточных результатов
- `CREATE TEMPORARY TABLE table_name (...)`



# SQL:1999 (SQL-3)

- появился тип BOOLEAN
- тип для больших объектов: BLOB, CLOB
- тип - массивы ARRAY
- создание пользовательских типов данных
- иерархические и рекурсивные запросы
- основы OLAP (online analytical processing, интерактивная аналитическая обработка)
- поддержка управления доступом на основе ролей
- триггеры

# СТЕ

- СТЕ = common table expression = общее табличное выражение
- Временный именованный набор результатов
- Определён в пределах области выполнения оператора SELECT, INSERT, UPDATE или DELETE
- Можно рассматривать как альтернативу подзапросам, представлениям и встроенным пользовательским функциям

# Задача

- Дана таблица Вакансий  
job\_offers(role, location, level, salary)
- Вторая таблица: сотрудники некой компании  
employee\_occupation (name, role, location)
- Нужно: получить средний размер зарплаты вакансий в зависимости от должности сотрудника
- Для этого нужно:
  1. посчитать средний размер ЗП каждой должности
  2. соединить полученную таблицу с таблицей employee\_occupation

# Решение 1: подзапрос

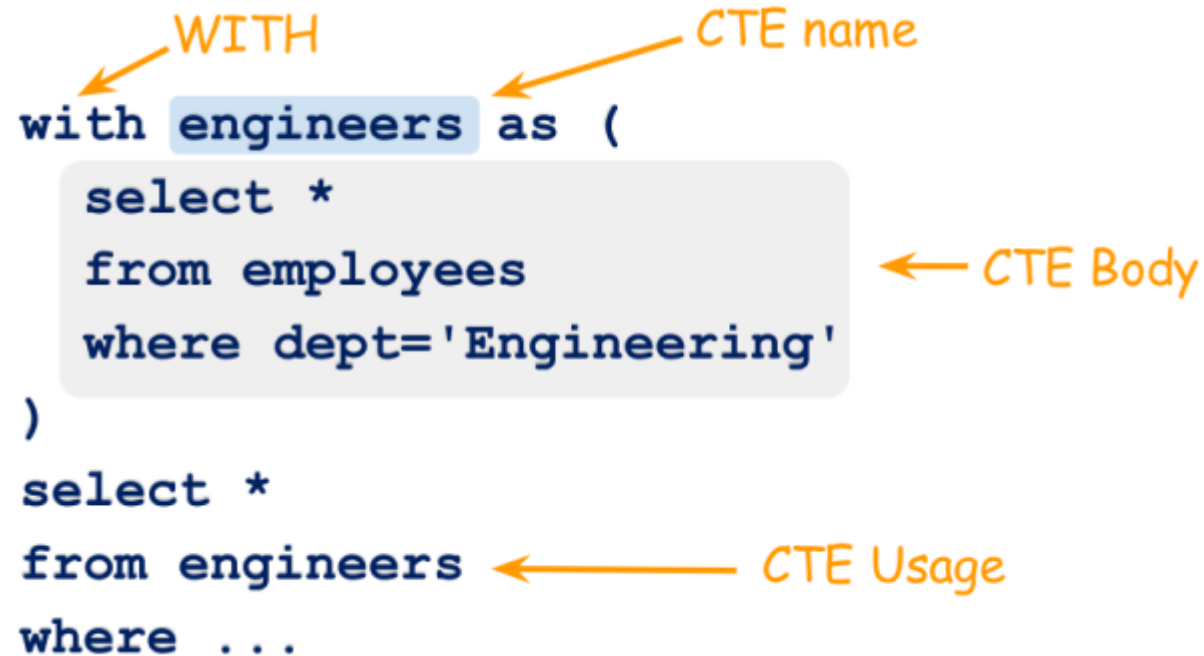
```
SELECT a.*, b.avg_salary  
FROM employee_occupation a  
LEFT JOIN  
    (SELECT role, avg(salary) AS avg_salary  
FROM job_offers  
GROUP BY role) b  
ON a.role = b.role;
```

- сначала выполняется подзапрос
- полученный результат соединяется с таблицей

## Решение 2: CTE

```
WITH average_salary AS (  
    SELECT role, avg(salary) AS avg_salary  
    FROM job_offers  
    GROUP BY role  
)  
  
SELECT a.*, b.avg_salary  
FROM employee_occupation a  
LEFT JOIN average_salary b  
ON a.role = b.role;
```

# Синтаксис



The diagram illustrates the syntax of a Common Table Expression (CTE) in SQL. It shows a code snippet with four orange arrows pointing to specific parts: 'WITH' points to the keyword, 'engineers' points to the CTE name, the subquery 'select \* from employees where dept='Engineering'' is enclosed in a light gray box and labeled 'CTE Body', and 'from engineers' in the final query is labeled 'CTE Usage'.

```
with engineers as (  
    select *  
    from employees  
    where dept='Engineering'  
)  
select *  
from engineers  
where ...
```

- можно использовать CTE в последующем выражении SQL, обращаясь к нему как будто бы к другой таблице
- CTE доступно только в выражении, до которого оно определяется, т.е. после выполнения запроса оно исчезает и не может быть использовано где-то ещё

# CTE vs Подзапросы

- CTE более читаемы
- CTE проще использовать несколько раз в одном и том же запросе
- CTE могут быть рекурсивными (об этом дальше)
- CTE нельзя использовать в WHERE, т.е. фильтрация возможна только по подзапросу

# Рекурсивный запрос

```
1 WITH RECURSIVE print_numbers AS (  
2   SELECT 1 AS n  
3   UNION ALL  
4   SELECT n + 1 FROM print_numbers WHERE n < 5  
5 )  
6 SELECT n FROM print_numbers;
```

carbon  
carbon.now.sh

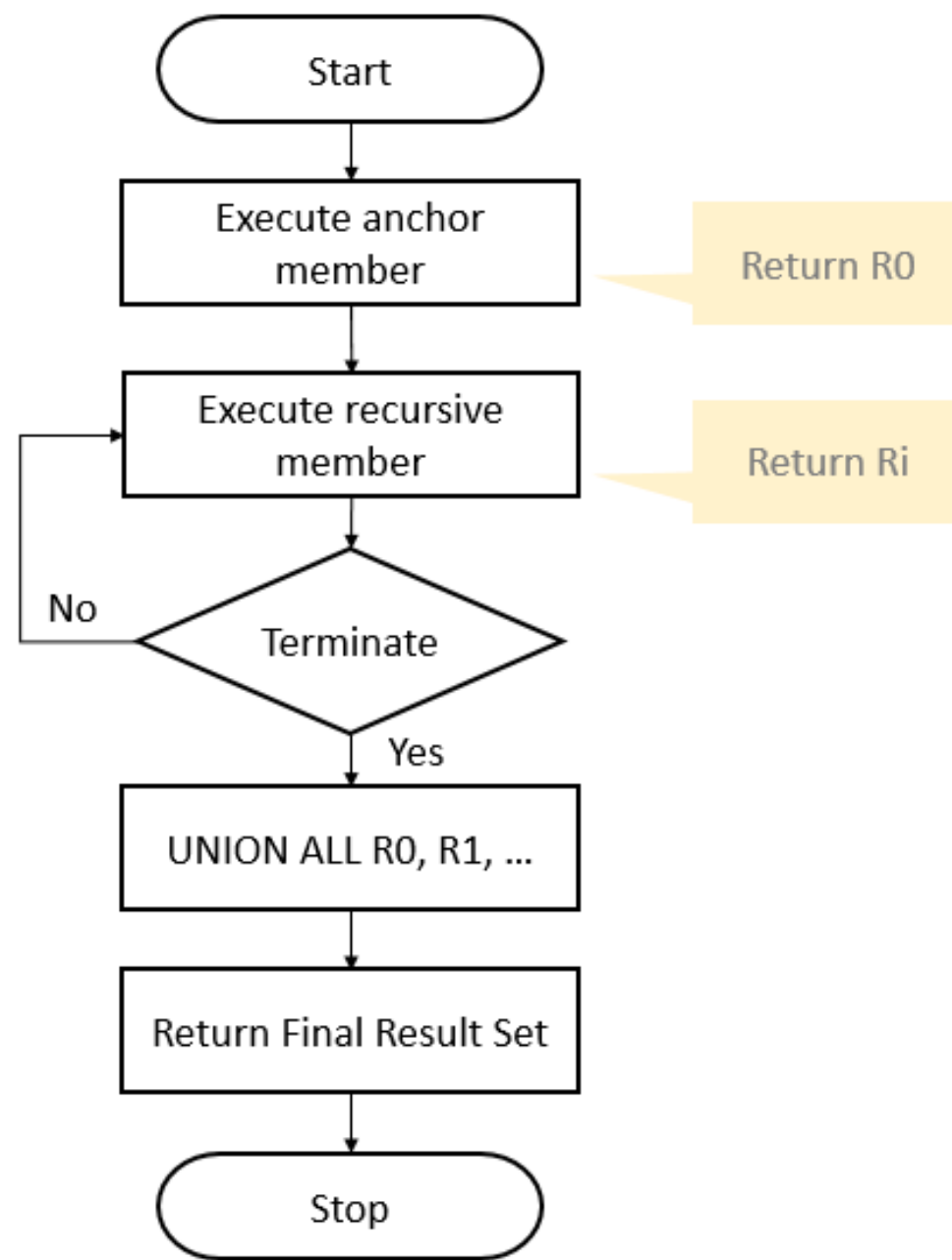
```
n  
---  
1  
2  
3  
4  
5  
(5 rows)
```

- Стартовая часть - SELECT 1 AS n
- Рекурсивная часть - SELECT n+1 FROM print\_number WHERE n<5
- Части разделены словом UNION



# Алгоритм работы

1. Берем стартовые данные
2. Подставляем в «рекурсивную» часть запроса.
3. Смотрим, что получилось:
  - а. если результат рекурсивной части не пустой, то добавляем его в результирующую выборку, а также используем этот результат как данные для следующего вызова рекурсивной части, т.е. goto 2
  - б. если пусто, то завершаем обработку



# Использование

- Чаще всего рекурсивные запросы нужны для иерархических данных
- Пример: вывести всех подчинённых с id начальников
- Сначала выводятся сотрудники, у которых нет начальства, и далее по нисходящей

```
WITH cte_org AS (  
    SELECT staff_id, first_name, manager_id  
    FROM sales.staffs  
    WHERE manager_id IS NULL  
    UNION ALL  
    SELECT e.staff_id, e.first_name, e.manager_id  
    FROM sales.staffs e  
    INNER JOIN cte_org o  
    ON o.staff_id = e.manager_id  
)  
SELECT * FROM cte_org;
```

staff_id	first_name	manager_id
1	Fabiola	NULL
2	Mireya	1
5	Jannette	1
8	Kali	1
6	Marcelene	5
7	Venita	5
9	Layla	7
10	Bernardine	7
3	Genna	2
4	Virgie	2

# Массивы

- (будем рассматривать реализацию в PostgreSQL, т.к. общую документацию на все СУБД найти непросто)
- Объявление столбца как массив:

```
CREATE TABLE students (  
    name text,  
    scores int[],  
    contacts varchar ARRAY -- or varchar[]  
);
```

# Работа с массивом

- Нумерация массива начинается с 1
- Например, есть таблица с сотрудниками и номерами телефонов

```
Demo=# SELECT * FROM Employees;
id | name      | contact
---+-----+-----
 1 | Alice John | <(408)-743-9045,<(408)-567-7834>
 2 | Kate Joel  | <(408)-783-5731>
 3 | James Bush | <(408)-745-8965,<(408)-567-78234>
(3 rows)
Demo=#
```

- Тогда для вывода сотрудника и первого номера телефона нужно написать:  
SELECT name, contact[1] FROM Employees;

```
Demo=# SELECT name, contact[1]
Demo=# FROM Employees;
name      | contact
---+-----+-----
Alice John | <(408)-743-9045
Kate Joel  | <(408)-783-5731
James Bush | <(408)-745-8965
(3 rows)
Demo=#
```

# Изменение массива

- Заменить значения массива полностью:

```
UPDATE sal_emp SET pay_by_quarter = '{25000,25000,27000,27000}'
```

```
WHERE name = 'Carol';
```

```
UPDATE sal_emp SET pay_by_quarter = ARRAY[25000,25000,27000,27000]
```

```
WHERE name = 'Carol';
```

- Изменить один элемент или срез

```
UPDATE sal_emp SET pay_by_quarter[4] = 15000
```

```
WHERE name = 'Bill';
```

```
UPDATE sal_emp SET pay_by_quarter[1:2] = '{27000,27000}'
```

```
WHERE name = 'Carol';
```

# Изменение массива

- Значение можно сконструировать с помощью оператора конкатенации

```
SELECT ARRAY[1,2] || ARRAY[3,4];
```

- Оператор конкатенации позволяет вставить один элемент в начало или в конец одномерного массива
- Можно использовать функцию из postgresql

```
UPDATE table1
```

```
SET integer_array = array_append(integer_array, 5);
```

# Поиск в массиве

- Можно искать по конкретному элементу массива

```
SELECT * FROM sal_emp  
WHERE pay_by_quarter[1] = 10000 OR  
       pay_by_quarter[2] = 10000;
```

- Можно искать по любому из (хотя бы одно):

```
SELECT * FROM sal_emp WHERE 10000 = ANY (pay_by_quarter);
```

- Поиск по всем элементам

```
SELECT * FROM sal_emp WHERE 10000 = ALL (pay_by_quarter);
```



На сегодня всё  
Остальное в следующий раз :)

Вопросы?