Базы данных

Лекция 7.

DCL.

Продвинутые возможности SQL (CTE, рекурсия, представления, оконные функции).

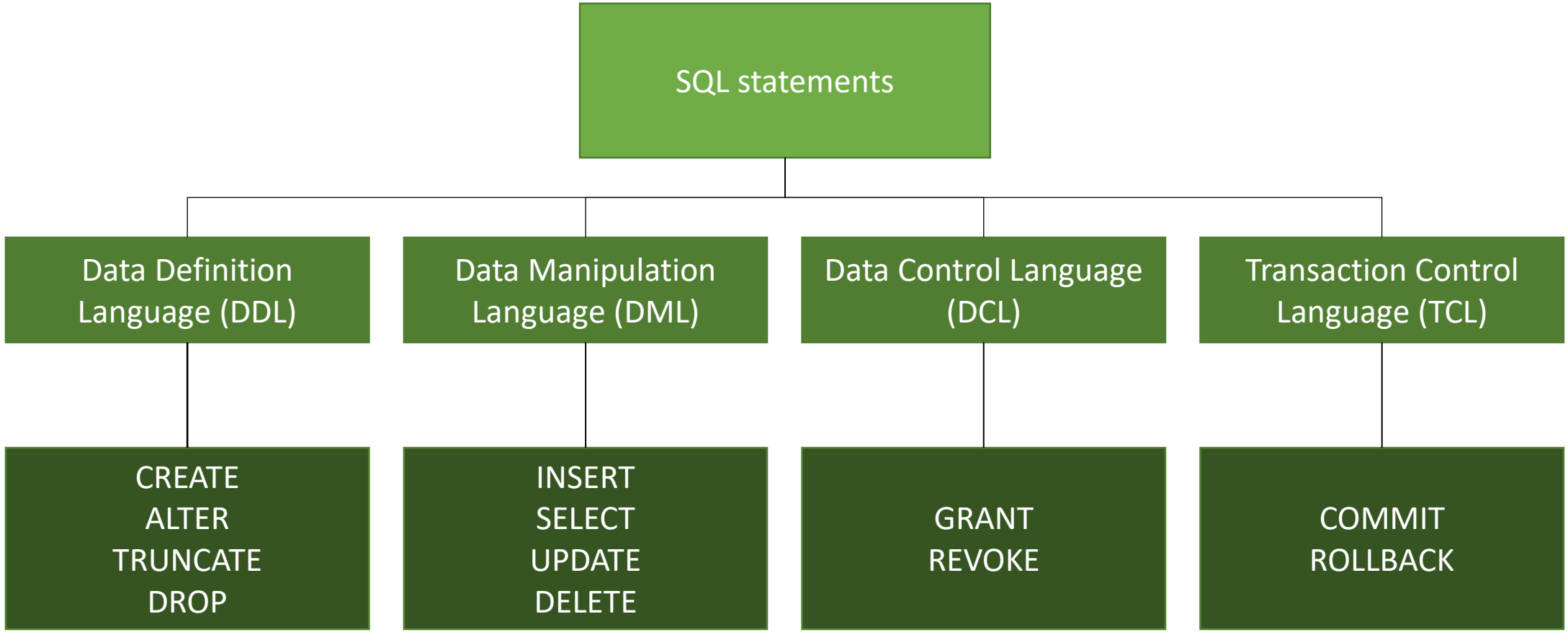
МФТИ, 2024

Игорь Шевченко

@igorshvch

I. DCL

Группы операторов SQL



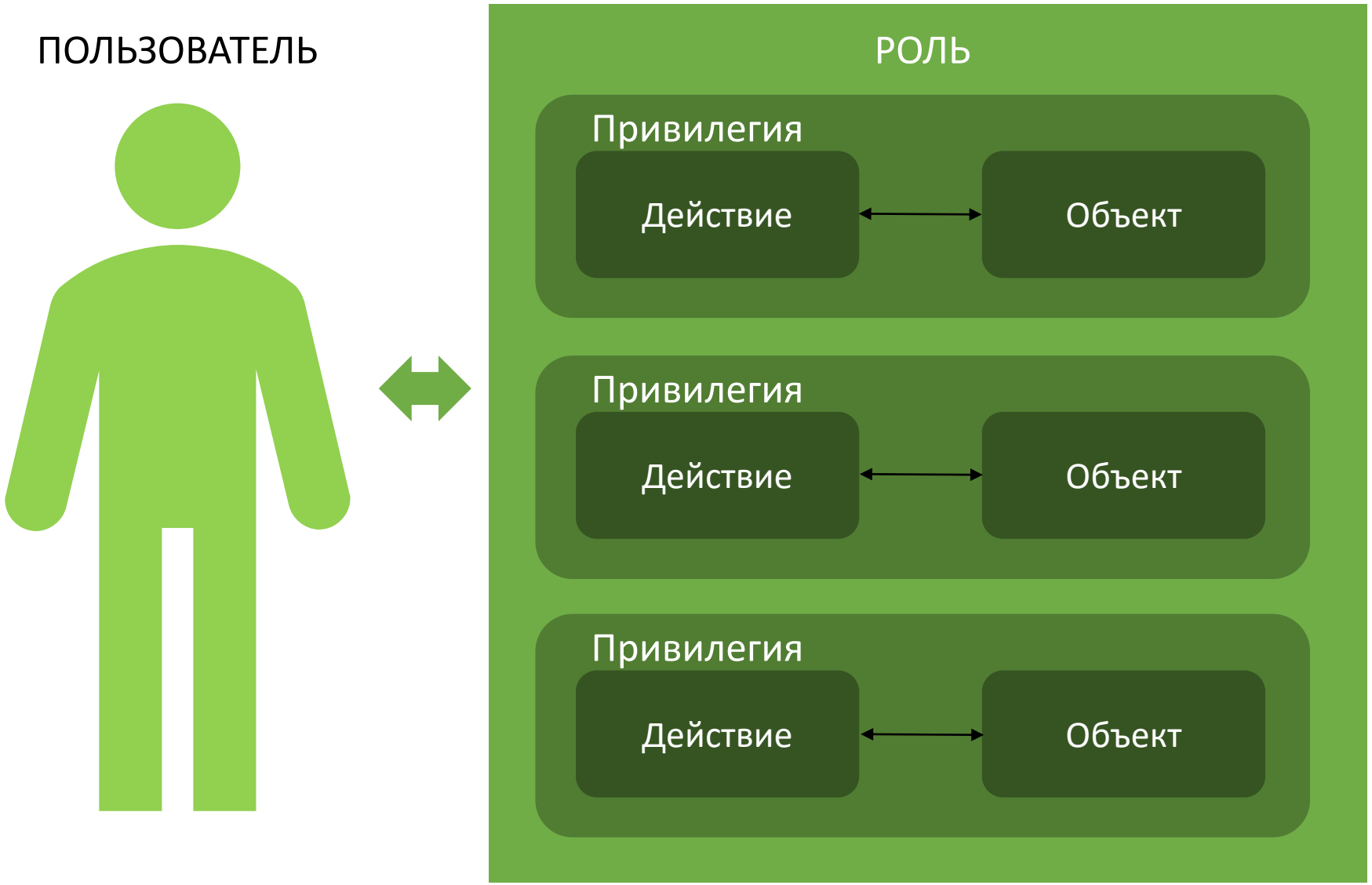
DCL – для чего нужен

- Это подмножество языка SQL, которое используется для управления правами доступа к данным и правами пользователей (ролями) в базах данных
- В свою очередь это необходимо для обеспечения безопасности данных
- В Postgres основной моделью безопасности данных является модель управления доступом (разграничения доступа) на основе ролей (role based access control, RBAC).
- Частично также реализована и поддерживается альтернативная модель – модель разграничения доступа на основе атрибутов объектов (attribute based access control, ABAC)

RBAC

- PostgreSQL использует концепцию ролей (roles) для управления разрешениями на доступ к базе данных.
- Роль можно рассматривать как **пользователя базы данных или как группу** пользователей, в зависимости от того, как роль настроена. Роли могут владеть объектами базы данных и выдавать другим ролям разрешения на доступ к этим объектам, управляя тем, кто имеет доступ и к каким объектам. Кроме того, можно предоставить одной роли членство в другой роли, таким образом одна роль может использовать права других ролей.

RBAC


ПОЛЬЗОВАТЕЛЬ
РОЛЬ
Привилегия
Действие
Объект
Привилегия
Действие
Объект
Привилегия
Действие
Объект

RBAC: основные абстракции

- Для того чтобы разграничить возможности разных пользователей, вводятся понятия **объекта** и **действия**. Действия могут быть как связаны с объектом (например, методы объекта) или с классом объектов, так и не связаны
- Любой пользователь, создавший объект, становится его владельцем. Владелец любого объекта имеет право выполнять любые действия, связанные с этим объектом, и может предоставлять (возможно, ограниченные) права доступа к своим объектам и действиям над ними другим пользователям. Права доступа к объектам и использования действий называются **привилегиями**.
- Для того, чтобы упростить управление передачей привилегий пользователям, вводится понятие **роли**. Каждой роли передаются привилегии, необходимые для выполнения всех операций, связанных с этой ролью.
- **Каждый пользователь получает право (привилегию) выполнять некоторую роль или несколько ролей**.

RBAC в Postgres

- Основные объекты Postgres:
  - таблицы
  - столбцы
  - представления
  - последовательности
  - базы данных
  - функции
  - процедуры
  - схемы
  - табличные пространства

DCL – синтаксис

- CREATE ROLE имя [[WITH] параметр [...]]
  - CREATE USER davide WITH PASSWORD 'jw8s0F4’;
- Параметр роли определяет её полномочия и взаимодействие с системой аутентификации клиентов
- Для изменения атрибутов роли применяется ALTER ROLE, а для удаления роли — DROP ROLE. Все атрибуты, заданные в CREATE ROLE, могут быть изменены позднее командами ALTER ROLE.
- Команда CREATE USER теперь является просто синонимом CREATE ROLE

DCL – синтаксис

GRANT { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES | TRIGGER }

[, ...] | ALL [PRIVILEGES] }

ON { [TABLE] имя_таблицы [, ...]

| ALL TABLES IN SCHEMA имя_схемы [, ...] }

TO указание_роли [, ...] [WITH GRANT OPTION]

[GRANTED BY указание_роли]

DCL – синтаксис

- Команда GRANT имеет две основные разновидности:
  - первая назначает права для доступа к объектам баз данных
  - вторая назначает одни роли членами других
- Если указано WITH GRANT OPTION, получатель права, в свою очередь, может давать его другим. Без этого указания распоряжаться своим правом он не сможет
- Примеры:
  - GRANT INSERT ON films TO PUBLIC;
  - GRANT ALL PRIVILEGES ON kinds TO manuel;
  - Включение в роль «admins» пользователя joe:
    - GRANT admins TO joe;

DCL – синтаксис

REVOKE [GRANT OPTION FOR]

{ { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES | TRIGGER }

[, ...] | ALL [PRIVILEGES] }

ON { [TABLE] имя_таблицы [, ...]

| ALL TABLES IN SCHEMA имя_схемы [, ...] }

FROM указание_роли [, ...]

[GRANTED BY указание_роли]

[CASCADE | RESTRICT]

DCL – синтаксис

- Команда REVOKE лишает одну или несколько ролей прав, назначенных ранее
- Если указано GRANT OPTION FOR, отзывается только право передачи права, но не само право. Без этого указания отзывается и право, и право распоряжаться им.
- Примеры:
  - REVOKE INSERT ON films FROM PUBLIC;
  - REVOKE ALL PRIVILEGES ON kinds FROM manuel;
  - REVOKE admins FROM joe;

II. CTE

CTE – общие табличные выражения

- Предложение WITH предоставляет способ записывать дополнительные операторы для применения в больших запросах. В частности, основное предназначение SELECT в предложении WITH заключается в разбиении *сложных запросов с подзапросами* **на простые части**
- Эти операторы, которые также называют общими табличными выражениями (Common Table Expressions, CTE), можно представить как определения временных таблиц, существующих только для одного запроса.
- Дополнительным оператором в предложении WITH может быть SELECT, INSERT, UPDATE или DELETE
- Само предложение WITH присоединяется к основному оператору, которым может быть SELECT, INSERT, UPDATE, DELETE или MERGE.

CTE – общие табличные выражения

- **Важно**! Порядок выполнения запроса с WITH: сначала выполняются все дополнительные запросы «внутри» WITH, только потом начинает выполняться предложение FROM в основном операторе. При выполнении WITH, по сути, формируются временные таблицы, к которым затем можно обращаться в FROM
- В соответствии со стандартом SQL, запросы, содержащие CTE, должны выполняться, как если бы каждое CTE было вычислено один раз. В системе PostgreSQL это было реализовано буквально: все CTE выполняются как отдельные запросы, результат материализуется (записывается) во временную память и затем используется при выполнении всего запроса, содержащего CTE

CTE – синтаксис

- [WITH [RECURSIVE] запрос_WITH [, ...]]

  SELECT…

- Где **запрос_WITH**:
  - имя_запроса_WITH [(имя_столбца [, ...])] AS [[NOT] MATERIALIZED] (выборка | values | insert | update | delete)

CTE – Пример

```
WITH regional_sales AS (
   SELECT region, SUM(amount) AS total_sales
   FROM orders
   GROUP BY region
), top_regions AS (
   SELECT region
   FROM regional_sales
   WHERE total_sales > (SELECT SUM(total_sales)/10 FROM regional_sales)
)
SELECT region,
      product,
      SUM(quantity) AS product_units,
      SUM(amount) AS product_sales
FROM orders
WHERE region IN (SELECT region FROM top_regions)
GROUP BY region, product;
```

CTE – Пример

```
WITH regional_sales AS (
    SELECT region, SUM(amount) AS total_sales
    FROM orders
    GROUP BY region
), top_regions AS (
    SELECT region
    FROM regional_sales
    WHERE total_sales > (SELECT SUM(total_sales)/10 FROM regional_sales)
)
SELECT region,
       product,
       SUM(quantity) AS product_units,
       SUM(amount) AS product_sales
FROM orders
WHERE region IN (SELECT region FROM top_regions)
GROUP BY region, product;
```

- Запрос выводит итоги по продажам только для передовых регионов
- Предложение WITH определяет два дополнительных оператора regional_sales и top_regions так, что результат regional_sales используется в top_regions, а результат top_regions используется в основном запросе SELECT
- Этот пример можно было бы переписать без WITH, но тогда нам понадобятся два уровня вложенных подзапросов SELECT

III. Рекурсия

Рекурсия

- Особым вариантом CTE являются «рекурсивные» запросы
- На самом деле никакой рекурсии нет, происходит итерация. (Исследования в 80-х, 90-х годах дали следующий результат: рекурсивные запросы не могут быть выражены в рамках реляционных языков)
- Синтаксис:

  [WITH [***RECURSIVE***] запрос_WITH [, ...]]
  SELECT…
- Используя RECURSIVE, запрос WITH может обращаться к собственному результату
- Для чего нужны: например, для поиска данных, имеющих иерархическую организацию, но хранящихся в виде таблицы

Рекурсия – пример

```
WITH RECURSIVE t(n) AS (
    VALUES (1)
    UNION ALL
    SELECT n+1 FROM t WHERE n < 100
)
SELECT sum(n) FROM t;
```

Рекурсия – пример

«Инициализация переменной» - объявление столбцов, к которым идет обращение в рекурсивной части

```
WITH RECURSIVE t(n) AS (
  VALUES (1)
  UNION ALL
  SELECT n+1 FROM t WHERE n < 100
)
SELECT sum(n) FROM t;
```

Нерекурсиивная часть

Рекурсиивная часть

Рекурсия – объяснение

```
WITH RECURSIVE t(n) AS (
    VALUES (1)
  UNION ALL
    SELECT n+1 FROM t WHERE n < 100
)
SELECT sum(n) FROM t;
```

1. Вычисляется **нерекурсивная часть**. Для UNION (но не UNION ALL) отбрасываются дублирующиеся строки. Все оставшиеся строки включаются в результат рекурсивного запроса **и также помещаются во временную рабочую таблицу**.
   (*Количество столбцов в предложении WITH RECURSIVE t(…), VALUES (в данном случае) и в рекурсивной части должно совпадать!*)
1. **Пока рабочая таблица не пуста**, повторяются следующие действия:
   a) Вычисляется **рекурсивная часть** так, что рекурсивная ссылка на сам запрос обращается к текущему содержимому **рабочей таблицы**. Для UNION (но не UNION ALL) отбрасываются дублирующиеся строки и строки, дублирующие ранее полученные. Все оставшиеся строки включаются в результат рекурсивного запроса и также помещаются во временную **промежуточную таблицу**.
   b) Содержимое **рабочей таблицы** заменяется содержимым **промежуточной таблицы**, а затем **промежуточная таблица** очищается.

Рекурсия – объяснение

```
WITH RECURSIVE t(n) AS (
   VALUES (1)
   UNION ALL
   SELECT n+1 FROM t WHERE n < 100
)
SELECT…
```

```
CREATE TABLE t (n INTEGER);
INSERT INTO t VALUES (1);
```

```
SELECT n FROM t
      UNION ALL
SELECT n+1 from t WHERE n < 100;
```

Для полного тождеств с рекурсивным запросом на этом этапе мы должны были бы «складывать» результаты в еще одну таблицу, а из нее мы бы потом запрашивали данные для очередных SELECT… UNION ALL SELECT…

Рекурсия – пример

```
WITH RECURSIVE t(n) AS (
    VALUES (1)
    UNION ALL
    SELECT n+1 FROM t WHERE n < 100
)
SELECT sum(n) FROM t;
```

Результат: 5050

Рекурсия – пример

```
WITH RECURSIVE t(n) AS (
    VALUES (1)
    UNION ALL
    SELECT n+1 FROM t WHERE n < 100
)
SELECT * FROM t;
```

n
1
2
…
99
100

Рекурсия – более сложный пример

У нас есть таблица parts, и мы хотим посчитать, сколько деталей нам нужно для нашего продукта our_product

sub_part	part	quantity
bolt	our_product	4
nut	our_product	4
washer	our_product	8
screw	bolt	2
metal	screw	1
rubber	washer	1

Ожидаемый результат

sub_part	total_quantity
bolt	4
metal	8
nut	4
rubber	8
screw	8
washer	8

Рекурсия – более сложный пример

```
WITH RECURSIVE included_parts(sub_part, part, quantity) AS (
    SELECT sub_part, part, quantity FROM parts WHERE part = 'our_product’
    UNION ALL
    SELECT p.sub_part, p.part, p.quantity * pr.quantity
    FROM included_parts pr, parts p
    WHERE p.part = pr.sub_part
)
SELECT sub_part, SUM(quantity) as total_quantity
FROM included_parts
GROUP BY sub_part
```

V. Представления (views)

Представления - описание

- Представления (views) — это виртуальные таблицы, которые представляют собой результат выполнения SQL-запросов. Представление не содержит реальных данных, а только определение запроса.
- Представление сохраняет SQL-запрос, предоставляя возможность обращаться к нему как к таблице. Представления могут включать в себя данные, полученные из одной или нескольких таблиц, и даже других представлений
- Цели использования представлений:
  - **Абстракция**: представления позволяют скрыть сложность SQL-запросов от конечных пользователей
  - **Безопасность**: с помощью представлений можно ограничить доступ пользователей к определенным данным
  - **Логическое разделение данных**: представления могут помочь в организации данных так, чтобы они были представлены пользователю наиболее логичным образом, не меняя при этом физической структуры базы данных
- Postgres поддерживает материализованные представления (Materialized Views). В отличие от обычных, они сохраняют результаты запроса в физической таблице, которая обновляется периодически или вручную. Это полезно для ускорения сложных запросов

Представления – изменяемые и неизменяемые

- Представления бывают **изменяемые** (обновляемые – updatable views) и **неизменяемые** (необновляемые).
- Изменяемые представления позволяют не только выполнять операции чтения (SELECT), но и изменять данные (INSERT, UPDATE, DELETE) через представление, как если бы эти операции выполнялись непосредственно над базовыми таблицами.
- По умолчанию «простые» представления являются автоматически изменяемыми (automatically updatable views). Для «сложных» представлений такого эффекта можно добиться, например, используя триггеры INSTEAD OF

Представления – какие является «простыми»

- Представление считается «простым», если оно соответствует следующим условиям:
  - Список FROM в запросе, определяющем представлении, должен содержать ровно один элемент, и это должна быть таблица или другое изменяемое представление.
  - Определение представления не должно содержать предложения WITH, DISTINCT, GROUP BY, HAVING, LIMIT и OFFSET (на верхнем уровне запроса).
  - Определение представления не должно содержать операции с множествами (UNION, INTERSECT и EXCEPT) (на верхнем уровне запроса).
  - Список выборки (…SELECT ***список_выборки*** FROM…) в запросе не должен содержать агрегатные и оконные функции, а также функции, возвращающие множества.

Представления - синтаксис

CREATE [OR REPLACE] [TEMP | TEMPORARY] [RECURSIVE] VIEW имя [(имя_столбца [, ...])]

[WITH (имя_параметра_представления [=значение_параметра_представления] [, ...])]

AS запрос

[WITH [CASCADED | LOCAL] CHECK OPTION]

Представления - примеры

```
CREATE VIEW comedies AS
    SELECT *
    FROM films
    WHERE kind = 'Comedy’;
```

- команда создаст представление со столбцами, которые содержались в таблице film в момент выполнения команды. Хотя при создании представления было указано *, столбцы, добавляемые в таблицу позже, частью представления не буду

Представления - примеры

```
CREATE VIEW universal_comedies AS
    SELECT *
    FROM comedies
    WHERE classification = 'U'
    WITH LOCAL CHECK OPTION;
```

- эта команда создаст представление на базе представления comedies, выдающее только комедии (kind = 'Comedy') универсальной возрастной категории classification = 'U'. Любая попытка выполнить в представлении INSERT или UPDATE со строкой, не удовлетворяющей условию classification = 'U', будет отвергнута, но ограничение по полю kind (тип фильма) проверяться не будет

Представления - примеры

```
CREATE VIEW pg_comedies AS
    SELECT *
    FROM comedies
    WHERE classification = 'PG'
    WITH CASCADED CHECK OPTION;
```

- это представление будет проверять, удовлетворяют ли новые строки обоим условиям: по столбцу kind и по столбцу classification.

Представления - примеры

```
CREATE VIEW comedies AS
   SELECT f.*,
          country_code_to_name(f.country_code) AS country,
          (SELECT avg(r.rating)
           FROM user_ratings r
           WHERE r.film_id = f.id) AS avg_rating
   FROM films f
   WHERE f.kind = 'Comedy’;
```

- Это представление будет поддерживать операции INSERT, UPDATE и DELETE. Изменяемыми будут все столбцы из таблицы films, тогда как вычисляемые столбцы country и avg_rating будут доступны только для чтения.

IV. Оконные функции

Оконные функции - описание

- Оконная функция выполняет вычисления для набора строк, некоторым образом связанных с текущей строкой. Её действие можно сравнить с вычислением, производимым агрегатной функцией
- С оконными функциями **строки не группируются в одну выходную строку**
- Вызов оконной функции всегда содержит предложение OVER, следующее за названием и аргументами оконной функции. Это синтаксически отличает её от обычной, не оконной агрегатной функции. Предложение OVER определяет, как именно нужно разделить строки запроса для обработки оконной функцией

Оконные функции - синтаксис

имя_функции ([выражение]) OVER (определение_окна)

определение_окна может состоять из следующих компонентов:

- [имя_существующего_окна]
- [PARTITION BY выражение [, ...]]
- [ORDER BY выражение [ASC | DESC | USING оператор] [NULLS { FIRST | LAST }] [, ...]]
- [определение_рамки]

- Вызов оконной функции всегда содержит предложение OVER, следующее за названием и аргументами оконной функции. Это синтаксически отличает её от обычной, не оконной агрегатной функции. Предложение OVER определяет, как именно нужно разделить строки запроса для обработки оконной функцией

Оконные функции - синтаксис

имя_функции ([выражение]) OVER (определение_окна)

определение_окна может состоять из следующих компонентов:

- [имя_существующего_окна]
- [PARTITION BY выражение [, ...]]
- [ORDER BY выражение [ASC | DESC | USING оператор] [NULLS { FIRST | LAST }] [, ...]]
- [определение_рамки]

- Вызов оконной функции всегда содержит предложение OVER, следующее за названием и аргументами оконной функции. Это синтаксически отличает её от обычной, не оконной агрегатной функции. Предложение OVER определяет, как именно нужно разделить строки запроса для обработки оконной функцией
- Предложение PARTITION BY, дополняющее OVER, разделяет строки по группам, или разделам, объединяя одинаковые значения выражений PARTITION BY. Оконная функция вычисляется по строкам, попадающим в один раздел с текущей строкой
- Предложение ORDER BY позволяет определять порядок, в котором строки будут обрабатываться оконными функциями
- Параметр рамки окна позволяет определить набор строк в ее разделе (партиции), которые будут обрабатываться оконной функцией

Оконные функции - примеры

SELECT salary, sum(salary) OVER () FROM empsalary;

Salary	Sum
5200	47100
5000	47100
3500	47100
4800	47100
3900	47100
4200	47100
4500	47100
4800	47100
6000	47100
5200	47100

Оконные функции - примеры

```
SELECT depname, empno, salary, avg(salary) OVER (PARTITION BY depname)
    FROM empsalary;
```

Department	Employee Number	Salary	Average Salary
develop	11	5200	5020.0
develop	7	4200	5020.0
develop	9	4500	5020.0
develop	8	6000	5020.0
develop	10	5200	5020.0
personnel	5	3500	3700.0
personnel	2	3900	3700.0
sales	3	4800	4866.7
sales	1	5000	4866.7
sales	4	4800	4866.7

Оконные функции - примеры

```
SELECT depname, empno, salary, avg(salary) OVER (PARTITION BY
depname)
        FROM empsalary;
```

- Первые три столбца извлекаются непосредственно из таблицы empsalary, при этом для каждой строки таблицы есть строка результата. В четвёртом столбце оказалось среднее значение, вычисленное по всем строкам, имеющим то же значение depname, что и текущая строка

Оконные функции - примеры

```
SELECT depname, empno, salary,
    rank() OVER (PARTITION BY depname ORDER BY salary DESC)
    FROM empsalary;
```

Department	Employee Number	Salary	Rank
develop	8	6000	1
develop	10	5200	2
develop	11	5200	2
develop	9	4500	4
develop	7	4200	5
personnel	2	3900	1
personnel	5	3500	2
sales	1	5000	1
sales	4	4800	2
sales	3	4800	2

Оконные функции - примеры

```
SELECT depname, empno, salary,
        rank() OVER (PARTITION BY depname ORDER BY salary DESC)
        FROM empsalary;
```

- Здесь, функция rank выдаёт порядковый номер для каждого уникального значения в разделе текущей строки, по которому выполняет сортировку предложение ORDER BY. У функции rank нет параметров, так как её поведение полностью определяется предложением OVER.

Оконные функции - примеры

SELECT salary, sum(salary) OVER (ORDER BY salary) FROM empsalary;

Salary	Sum
3500	3500
3900	7400
4200	11600
4500	16100
4800	25700
4800	25700
5000	30700
5200	41100
5200	41100
6000	47100

Оконные функции - примеры

SELECT salary, sum(salary) OVER (ORDER BY salary) FROM empsalary

- Здесь в сумме накапливаются зарплаты от первой (самой низкой) до текущей, включая повторяющиеся текущие значения (обратите внимание на результат в строках с одинаковой зарплатой)
- Дело в том, что по умолчанию с указанием ORDER BY рамка состоит из всех строк от начала раздела до текущей строки и строк, равных текущей по значению выражения ORDER BY