VIEW и оконные функции

Создание представлений и обзор возможностей оконных функций

Лазар В. И., Козлова Е. Р.

6 февраля 2025 г.

План занятия

1. Что такое VIEW и зачем он нужен
2. Оконные функции в SQLite
3. Задания

Зачем нужны VIEW (представления)?

- **VIEW** (представление) — виртуальная таблица, определяемая запросом `SELECT`.
- Хранит не данные сами по себе, а только определение запроса.
- Упрощает логику: можно «сохранить» сложный запрос как `VIEW` и затем работать с ним как с обычной таблицей.
- Повышает безопасность: даёт доступ к нужной части данных без прямого доступа к исходным таблицам.
- Удобно использовать для отображения объединённых или вычисляемых данных.

Синтаксис создания VIEW

Синтаксис (SQLite)

```
CREATE [OR REPLACE] VIEW view_name AS
SELECT ...
FROM ...
[WHERE ...];
```

- `CREATE VIEW view_name AS <SELECT>` — создаёт новое представление.
- `OR REPLACE` — перезаписывает существующее представление.
- В SQLite представления нельзя напрямую `UPDATE`, т. к. у них нет своих данных.

Пример: Создание VIEW

Допустим, у нас есть таблицы `employees` **и** `departments`.

```
CREATE VIEW v_emp_dept AS
SELECT e.emp_id,
       e.emp_name,
       e.salary,
       d.dept_name
FROM employees AS e
JOIN departments AS d
    ON e.dept_id = d.dept_id
WHERE e.salary > 50000;
```

Описание:

- `v_emp_dept` покажет сотрудников с окладом $> 50\,000$ и названия их отделов.
- Можно теперь делать `SELECT` из `v_emp_dept` как из таблицы.

Использование VIEW

Как работать с созданным VIEW?

```
SELECT *
FROM v_emp_dept
WHERE dept_name = 'Отдел разработки';
```

- Выполняется так, как будто `v_emp_dept` — это обычная таблица.
- По сути, СУБД «подставляет» исходный запрос при выполнении.

Удаление VIEW:

```
DROP VIEW v_emp_dept;
```

- Удаляет представление (не затрагивая исходные таблицы).

Что такое оконные функции?

- **Оконные функции** (*window functions*) — специальные функции SQL, позволяющие выполнять вычисления по «окну» строк (`OVER`).
- В отличие от обычных агрегатных функций, не свёртывают строки в одну, а «распределяют» вычисленное значение по каждой строке.
- Примеры: `ROW_NUMBER()`, `RANK()`, `LAG()`, `LEAD()`, агрегаты с `OVER`.
- В SQLite доступно, начиная с версии 3.25.0 (сентябрь 2018).

Общий синтаксис оконных функций

```
<function>() OVER (
    [PARTITION BY column_list]
    [ORDER BY column_list]
)
```

Пример:

```
SELECT
    employee_id,
    salary,
    AVG(salary) OVER (
        PARTITION BY dept_id
    ) AS avg_salary_in_dept
FROM employees;
```

- **PARTITION BY** — группирует строки в «окна» (например, по отделам).
- **ORDER BY** — порядок, важный для функций ранжирования

Пример 1: ROW_NUMBER()

```
SELECT
    emp_name,
    salary,
    ROW_NUMBER() OVER (ORDER BY salary DESC) AS row_num
FROM employees;
```

- ROW_NUMBER даёт порядковый номер каждой строки в рамках указанного окна.
- В данном случае окно — вся таблица, упорядоченная по убыванию зарплаты.

Пример 2: RANK() и DENSE_RANK()

```
SELECT
    emp_name,
    salary,
    RANK() OVER (ORDER BY salary DESC) AS salary_rank,
    DENSE_RANK() OVER (ORDER BY salary DESC) AS dense_rank
FROM employees;
```

Описание:

- RANK() — при совпадении зарплат пропускает следующий ранг (1,1,3,...).
- DENSE_RANK() — идёт подряд без пропусков (1,1,2,...).

Пример 3: Агрегат с PARTITION BY

Задача: для каждого сотрудника вывести его зарплату и среднюю зарплату в отделе.

```
SELECT
    e.emp_id,
    e.emp_name,
    e.dept_id,
    e.salary,
    AVG(e.salary) OVER (
        PARTITION BY e.dept_id
    ) AS avg_salary_in_dept
FROM employees e;
```

- Каждая строка сохраняется, но получает дополнительную инфу о средней зарплате по своему отделу.

Задания (1–4)

1. **Создание VIEW для объединения таблиц:** Создайте `VIEW` (например, `v_employee_info`) на основе `employees` и `departments`, выводя `emp_name`, `salary`, `dept_name`. Отберите только сотрудников, у которых зарплата выше среднего в их отделе (подумайте, можно ли использовать оконную функцию прямо в `CREATE VIEW`).
2. **Добавление вычисляемого поля в VIEW:** Создайте представление, где помимо основных полей будет столбец «годовая зарплата» = `salary * 12`. Покажите пример `SELECT` из этого представления.
3. **VIEW с условием:** Создайте представление `v_high_salary`, отображающее только тех сотрудников, у кого `salary` > 100000. Затем сделайте `SELECT` из `v_high_salary` и убедитесь, что вывелись только «высокие» зарплаты.
4. **Удаление VIEW:** Удалите одно из созданных вами представлений. Убедитесь, что исходные таблицы при этом не пострадали.

Задания (5–8)

5. **Применение оконной функции** `ROW_NUMBER()`: Напишите запрос, который присваивает каждой записи о сотруднике уникальный порядковый номер (по убыванию зарплаты).
6. `RANK()` **и одинаковая зарплата:** Покажите, как `RANK()` присваивает места сотрудникам, если у нескольких человек зарплата совпадает.
7. **Агрегация с** `PARTITION BY`: Для таблицы `employees` найдите, какому отделу принадлежит каждый сотрудник и какова средняя зарплата в этом отделе.
8. `DENSE_RANK()` **по отделам:** Сгруппируйте сотрудников по отделам и упорядочьте внутри отдела по зарплате по убыванию.

Задания (9–11)

9. **Создание VIEW с оконной функцией:** Создайте `VIEW`, в котором каждая строка содержит `emp_id`, `emp_name`, и позицию сотрудника (`ROW_NUMBER()`) в общем рейтинге по зарплате. Убедитесь, что при `SELECT * FROM <view_name>` таблица корректно отображает ранги.
10. **Сравнение зарплаты с предыдущей строкой (`LAG`):** Напишите запрос, который для каждого сотрудника показывает его `salary` и «предыдущую» зарплату — так вы сможете сравнить.
11. **Дополнительный вызов: обновление представления?** Попробуйте выполнить `UPDATE` или `INSERT` через созданное `VIEW` и проверьте, даёт ли это результат или ошибку в SQLite.

Задание 12

12. **Сложный запрос + VIEW:** Попробуйте объединить данные из нескольких таблиц (например, `employees`, `departments`, `projects` — если есть) в один запрос с оконной функцией, а затем сделайте из него `VIEW`, чтобы упростить повторный доступ к этим вычислениям.