

ПРАКТИЧЕСКАЯ РАБОТА №8. СЛОЖНЫЕ ТИПЫ ДАННЫХ И МАСШТАБИРОВАНИЕ POSTGRESQL

Цель работы:

Целью данной практической работы является освоение методов работы со сложными и неструктурированными типами данных (JSONB, BYTEA) в PostgreSQL, а также получение практических навыков реализации горизонтального масштабирования базы данных с помощью декларативного секционирования.

По завершении работы студент должен уметь:

- Проектировать таблицы для хранения чувствительных данных (например, паролей) с использованием бинарного типа данных BYTEA и криптографических функций.
- Реализовывать хранение полуструктурированных данных (логов, событий) в формате JSONB.
- Выполнять выборку, фильтрацию и агрегацию данных из JSON-документов с использованием специализированных операторов (->>, @>, ?) и условий сравнения.
- Модифицировать структуру JSON-документов «на лету» без изменения схемы таблицы.
- Настраивать декларативное секционирование (Partitioning) таблиц по диапазонам для оптимизации производительности при работе с большими объемами данных.

Постановка задачи:

Для выполнения практической работы необходимо последовательно выполнить следующие шаги, адаптируя примеры из БД «Аптека» к вашей собственной базе данных.

Задание №1: хранение паролей (BYTEA)

1. Создать справочную таблицу ролей и таблицу системных пользователей (или модифицировать существующую).

2. В таблице пользователей предусмотреть поля для хранения «сырого» пароля (только для учебной демонстрации) и его хеша (тип BYTEA).
3. Сохранить хеши паролей пользователей, используя функцию хеширования digest.
4. Выполнить запрос, проверяющий соответствие введенного пароля сохраненному хешу. Одна проверка должна быть успешной, другая - нет (допустимо сделать это в одном запросе).

Задание №2: секционирование таблицы логов

1. Создать новую секционированную таблицу для хранения логов событий.
2. Использовать стратегию секционирования по диапазонам (RANGE) по полю даты.
3. Создать две секции для периодов:
 - 2-е полугодие 2024 (с 2024-07-01 по 2025-01-01).
 - 1-е полугодие 2025 (с 2025-01-01 по 2025-07-01).
4. Создать секцию по умолчанию, куда будут попадать все данные, выходящие за рамки указанных диапазонов (будущее время).

Задание №3: генерация данных

Написать скрипт на PL/pgSQL для генерации 20 000 записей в диапазоне дат, покрывающем созданные секции. Сгенерированные данные должны содержать JSON-структуру с различным набором полей для разных типов событий (например, чтобы логи «продажи» отличались по структуре от логов «входа»).

Вывести сгенерированные данные – показать запросом SELECT, что данные успешно созданы и физически распределились по разным таблицам-секциям.

Задание №4: анализ и поиск по JSONB

Написать три запроса:

1. **Фильтрация по значению** – найти записи, где числовое поле внутри JSON удовлетворяет математическому условию (например, > 800), используя операторы извлечения $->$ и $->>$.
2. **Поиск по вхождению** – найти записи, содержащие точный фрагмент JSON-структуры (например, `{"quantity": 5}`), используя оператор $@>$.
3. **Агрегация** – посчитать сумму или среднее значение числового поля из JSON-документа.

Задание №5: модификация данных JSONB

Продemonстрировать изменение данных внутри JSON-поля:

1. Массовое добавление нового поля во все записи определенного типа.
2. Точечное изменение значения по ключу в конкретной записи.

ХОД ВЫПОЛНЕНИЯ РАБОТЫ

Введение

В данной работе мы расширяем функционал базы данных, добавляя подсистему безопасности и аудита. Мы будем использовать таблицы из предыдущих работ для контекста, но создадим новые таблицы для пользователей системы и журнала событий.

Ниже приводятся таблицы, используемые для построения запросов.

Таблица 1. Таблица *manufacturers* (Производители)

	123 id	A-Z manufacturer_name	A-Z country
1	1	ООО "Фармстандарт"	Россия
2	2	Bayer AG	Германия

Таблица 2. Таблица *medicines* (Лекарства)

	123 id	A-Z name	123 quantity_in_stock	123 price	production_date	expiration_date	123 manufacturer_id
1	1	Парацетамол	200	50,5	2025-07-10	2028-07-10	1
2	2	Аспирин	150	120	2025-07-12	2027-07-12	2
3	3	Ибупрофен	100	85	2025-07-11	2028-07-11	1
4	4	Витамин С	300	250	2025-07-10	2027-07-10	2
5	5	Активированный уголь	500	25	2025-07-10	2030-07-10	1
6	8	Корвалол	140	45	2025-06-01	2029-06-01	1

Таблица 3. Таблица *sale_items* (Проданные товары)

	123 sale_item_id	123 sale_id	123 medicine_id	123 quantity	123 unit_price
1	1	1	1	2	50,5
2	2	1	2	1	120
3	3	2	3	3	85
4	4	2	1	1	50,5
5	5	2	4	5	250

Таблица 4. Таблица *customer_audit* (Аудит)

	123 user_id	A-Z first_name	A-Z last_name	A-Z phone_number	change_date
1	1	Иван	Иванов	+79001234567	2025-10-20 01:26:02.183

Таблица 5. Таблица *customers* (Клиенты)

	123 customer_id	A-Z first_name	A-Z last_name	A-Z email	A-Z phone_number
1	2	Петр	Петров	petr@example.com	[NULL]
2	1	Иван	Иванов	ivan@example.com	+79998887766

1. РАБОТА С БИНАРНЫМИ ДАННЫМИ (BYTEA) И ХЕШИРОВАНИЕ

Тип данных BYTEA (byte array) предназначен для хранения бинарных строк. Это наиболее подходящий формат для хранения изображений, файлов, а также результатов криптографических операций.

Хеширование — это процесс преобразования входных данных (пароля) произвольной длины в битовую строку фиксированной длины (хеш) с помощью специального алгоритма (например, SHA-256 или MD5).

Ключевая особенность заключается в том, что это — **односторонняя** функция. Из хеша невозможно (или вычислительно крайне сложно) восстановить исходный пароль. При входе пользователя в систему, введенный им пароль снова хешируется, и результат сравнивается с тем, что хранится в базе.

1.1 Создание таблиц

Если в вашей базе данных уже есть таблица пользователей, добавьте в неё необходимые поля. Если нет — создайте новую структуру.

ВАЖНО: для учебных целей мы создадим поле **password_raw** для хранения пароля в открытом виде, чтобы наглядно сравнивать его с хешем.

В реальных системах **хранение паролей в открытом виде КАТЕГОРИЧЕСКИ ЗАПРЕЩЕНО**. Это создаёт критическую уязвимость безопасности. Храниться должны только хеши (желательно с «солью», которая представляет собой символы, добавляемые к паролю по одному и тому же алгоритму, перед выполнением хеширования, и предназначенные для защиты от взлома паролей через «радужные таблицы» - миллионы заранее посчитанных хешей для наиболее часто встречающихся паролей).

Листинг 1. Создание таблиц

```
CREATE TABLE system_roles (  
    role_id SERIAL PRIMARY KEY,  
    role_name VARCHAR(50) NOT NULL UNIQUE  
);  
  
CREATE TABLE system_users (  
    user_id SERIAL PRIMARY KEY,  
    username VARCHAR(100) NOT NULL UNIQUE,
```

```

password_raw TEXT,          -- ТОЛЬКО ДЛЯ УЧЕБНЫХ ЦЕЛЕЙ!!!
password_hash BYTEA NOT NULL,
role_id INT REFERENCES system_roles(role_id)
);

INSERT INTO system_roles (role_name)
VALUES ('Администратор'), ('Фармацевт'), ('Менеджер');

```

1.3 Хеширование и проверка паролей

Добавим пользователей. В поле `password_hash` мы запишем результат работы функции `digest` (алгоритм SHA-256).

Листинг 2. Вставка и проверка данных пользователей

```

-- Вставка пользователей
INSERT INTO system_users (username, password_raw, password_hash, role_id)
VALUES
('admin_user', 'StrongPass123!', digest('StrongPass123!', 'sha256'), 1),
('pharmacy_worker', 'Pharma2025', digest('Pharma2025_bad', 'sha256'), 2);

-- Проверка: сравнение хеша от "сырого" пароля с сохраненным хешем
SELECT
    username,
    password_raw,
    encode(password_hash, 'hex') AS password_hash,
    (digest(password_raw, 'sha256') = password_hash) as is_valid_check
FROM system_users;

```

В результате запроса поле `password_hash` будет отображаться в шестнадцатеричном формате.

Поле `is_valid_check` должно быть **true** для первого пользователя, и **false** для второго, что подтверждает корректность работы алгоритма.

	AZ username	AZ password_raw	AZ password_hash	<input checked="" type="checkbox"/> is_valid_check
1	admin_user	StrongPass123!	805bd951772627f3d1a607084df1727c6caad60447c5d73febf7be2d2fe17fd8	[v]
2	pharmacy_worker	Pharma2025	7f1ee9720d471254a598971cec76580e6549eabf586f784f5826a53984c75f3a	[]

Рисунок 1 – Результат вставки и проверки хешей

2. МАСШТАБИРОВАНИЕ БАЗЫ ДАННЫХ (СЕКЦИОНИРОВАНИЕ)

Секционирование (Partitioning) — это разделение большой таблицы на более мелкие части (секции) по определенному критерию (например, по дате).

Секционирование vs Модифицируемые представления

Часто возникает вопрос: чем секционирование отличается от использования представлений (VIEW) с правилами?

Модифицируемые представления — это логическая **абстракция**. Данные физически могут лежать где угодно, а представление лишь фильтрует или перенаправляет запросы. Это удобно для разграничения прав доступа или упрощения запросов, но не всегда дает выигрыш в производительности.

Секционирование — это **физическое разделение** данных. Планировщик запросов точно знает, в какой таблице-секции лежат данные за определенный месяц, и вообще не читает остальные таблицы (механизм Partition Pruning). Это даёт колоссальный прирост скорости на больших объемах (миллионы строк).

2.1 Создание таблицы логов

Создадим таблицу **user_logs** для хранения истории действий. Используем стратегию разделения данных **RANGE** (по диапазону) для поля **created_at**.

Листинг 3. Создание секционированной таблицы

```
CREATE TABLE user_logs (  
  log_id BIGSERIAL,  
  created_at TIMESTAMPTZ NOT NULL,  
  user_id INT,  
  event_data JSONB  
) PARTITION BY RANGE (created_at);
```

2.2 Создание секций (партиций)

Создадим секции согласно заданию: 2-е полугодие 2024 и 1-е полугодие 2025. Также добавим секцию **DEFAULT**, которая будет автоматически

принимать все данные, не попадающие в явные диапазоны (например, текущие данные конца 2025 года).

Листинг 4. Создание секций

```
-- Секция 1: Июль 2024 - Декабрь 2024
CREATE TABLE user_logs_2024_h2 PARTITION OF user_logs
    FOR VALUES FROM ('2024-07-01') TO ('2025-01-01');

-- Секция 2: Январь 2025 - Июнь 2025
CREATE TABLE user_logs_2025_h1 PARTITION OF user_logs
    FOR VALUES FROM ('2025-01-01') TO ('2025-07-01');

-- Секция по умолчанию (для остальных дат)
CREATE TABLE user_logs_default PARTITION OF user_logs DEFAULT;
```

2.3 Создание индекса

Индексы создаются на родительской таблице и наследуются всеми секциями.

Листинг 5. Создание индекса

```
CREATE INDEX idx_user_logs_date ON user_logs(created_at);
```

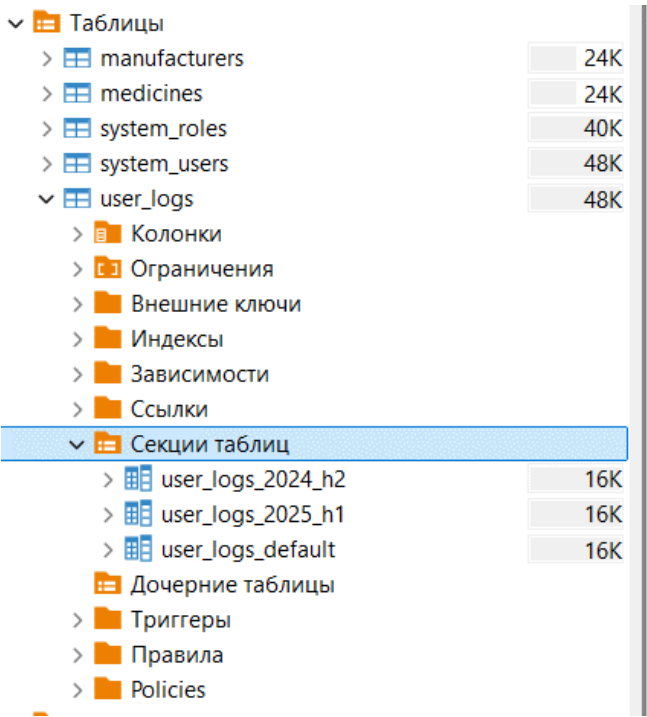


Рисунок 2 – Структура таблиц в навигаторе объектов (видна иерархия)

3. ГЕНЕРАЦИЯ ДАННЫХ И ТИП JSONB

JSONB — это эффективный формат хранения JSON в PostgreSQL. Он позволяет хранить данные, структура которых может меняться от записи к записи.

3.1 Генерация тестовых данных

Сгенерируем 20 000 записей.

Пояснение к генерации данных:

Поскольку JSON — это гибкая структура, мы можем хранить в нем ссылки на любые сущности нашей базы данных. В примере ниже мы симулируем реальную ситуацию:

- если событие — это **вход в систему** (login), мы сохраняем **IP-адрес**.
- если событие — это **продажа** (sale), мы сохраняем **ID проданного товара** (*medicine_id*), количество и сумму.

ВАЖНО: При адаптации под свою базу данных, используйте функцию `floor(random() * N + 1)`, чтобы генерировать случайные ID, которые реально существуют в ваших таблицах-справочниках (например, ID товаров, ID услуг, ID сотрудников).

Листинг 6. Скрипт генерации данных

```
DO $$
DECLARE
    i INT;
    random_date TIMESTAMPTZ;
    random_user INT;
    event_type TEXT;
    json_payload JSONB;
BEGIN
    FOR i IN 1..20000 LOOP
        -- Генерация даты в диапазоне с середины 2024 по середину 2025
        random_date := '2024-07-01'::TIMESTAMPTZ + random() * ('2025-07-01'::TIMESTAMPTZ - '2024-07-01'::TIMESTAMPTZ);

        random_user := floor(random() * 2 + 1)::INT;
```

```

IF (i % 2 = 0) THEN
    event_type := 'login';
    -- JSON для входа: сохраняем IP и статус
    json_payload := jsonb_build_object(
        'event', event_type,
        'ip', '192.168.1.' || floor(random() * 255)::TEXT,
        'success', (random() > 0.1)
    );
ELSE
    event_type := 'sale';
    json_payload := jsonb_build_object(
        'event', event_type,
        'details', jsonb_build_object(
            'medicine_id', floor(random() * 5 + 1),
            'quantity', floor(random() * 10 + 1),
            'total_sum', (random() * 1000)::NUMERIC(10,2)
        )
    );
END IF;

INSERT INTO user_logs (created_at, user_id, event_data)
VALUES (random_date, random_user, json_payload);
END LOOP;
END $$;

```

3.2 Проверка распределения данных

Убедимся, что данные «разлетелись» по нужным таблицам.

Листинг 7. Проверка количества записей в секциях

```

SELECT '2024_h2' as partition_name, count(*) FROM user_logs_2024_h2
UNION ALL
SELECT '2025_h1', count(*) FROM user_logs_2025_h1
UNION ALL
SELECT 'default', count(*) FROM user_logs_default;

```

	A-Z partition_name ▼	123 count ▼
1	2024_h2	10 064
2	2025_h1	9 936
3	default	0

Рисунок 3 – Результат распределения данных по секциям

4. АНАЛИЗ ДАННЫХ JSONB

Для работы с JSONB используются специфические операторы.

Теоретическая справка: тип возвращаемых данных (-> vs ->>)

Это – самая частая ошибка новичков.

- Оператор **->** возвращает данные в **формате JSON**. Например, если по ключу лежит строка "login", он вернет её в кавычках: "login". Этот оператор нужен, если вы хотите спускаться дальше по вложенности (объект внутри объекта).
- Оператор **->>** возвращает данные в **формате TEXT**. Он вернет чистую строку: login. Этот оператор нужен для сравнения значений (=, LIKE) или для приведения к числам (::numeric).

4.1 Поиск по ключу и фильтрация с условием

Найдем все продажи, где сумма чека (total_sum) превышает 800 рублей. Для этого нужно извлечь значение из JSON, привести его к числу и сравнить.

Листинг 8. Фильтрация со сложным условием

```
SELECT log_id, event_data
FROM user_logs
WHERE event_data ->> 'event' = 'sale'
      AND (event_data -> 'details' ->> 'total_sum')::NUMERIC > 800
LIMIT 5;
```

	123 log_id	{ } event_data
1	11	{"event": "sale", "details": {"quantity": 3, "total_sum": 807.57, "medicine_id": 1}}
2	25	{"event": "sale", "details": {"quantity": 7, "total_sum": 833.20, "medicine_id": 2}}
3	31	{"event": "sale", "details": {"quantity": 10, "total_sum": 932.44, "medicine_id": 2}}
4	37	{"event": "sale", "details": {"quantity": 7, "total_sum": 977.20, "medicine_id": 4}}
5	59	{"event": "sale", "details": {"quantity": 5, "total_sum": 860.79, "medicine_id": 2}}

Рисунок 4 – Результат выборки дорогих покупок

Обратите внимание: мы используем **->**, чтобы «провалиться» в объект details, а затем **->>**, чтобы получить текст числа, и приводим его к **::NUMERIC** для математического сравнения.

4.2 Фильтрация по вхождению (JSON Containment)

Найдем все записи, содержащие конкретную структуру: продажа 5 единиц любого товара.

Листинг 9. Поиск по вхождению фрагмента

```
SELECT * FROM user_logs
WHERE event_data @> '{"details": {"quantity": 5}}'
LIMIT 5;
```

	123 log_id	created_at	123 user_id	{ } event_data
1	45	2024-10-07 22:31:42.116 +0300	1	{"event": "sale", "details": {"quantity": 5, "total_sum": 160.60, "medicine_id": 5}}
2	59	2024-12-31 17:27:16.040 +0300	2	{"event": "sale", "details": {"quantity": 5, "total_sum": 860.79, "medicine_id": 2}}
3	91	2024-07-02 05:19:08.881 +0300	2	{"event": "sale", "details": {"quantity": 5, "total_sum": 670.09, "medicine_id": 1}}
4	129	2024-07-22 05:42:20.342 +0300	1	{"event": "sale", "details": {"quantity": 5, "total_sum": 465.74, "medicine_id": 4}}
5	135	2024-07-06 13:08:08.704 +0300	2	{"event": "sale", "details": {"quantity": 5, "total_sum": 628.65, "medicine_id": 2}}

Рисунок 5 – Результат поиска продаж с количеством 5

Пояснение: Оператор **@>** проверяет, является ли JSON справа подмножеством JSON-а слева.

Запрос работает, даже если внутри details есть еще 10 других полей (*цена, ID товара и т.д.*).

PostgreSQL проверяет только наличие указанных ключей и совпадение их значений. Остальное игнорируется.

4.3 Агрегация данных

Посчитаем среднюю сумму чека по данным из JSON.

Листинг 10. Агрегация данных

```
SELECT
    AVG((event_data -> 'details' ->> 'total_sum')::NUMERIC)::NUMERIC(10,2) AS
    avg_check
FROM user_logs
WHERE event_data ->> 'event' = 'sale';
```

	123 avg_check
1	504,78

Рисунок 6 – Результат расчета среднего чека

5. МОДИФИКАЦИЯ ДАННЫХ JSONB

В отличие от обычной записи данных, изменение JSON требует понимания того, как PostgreSQL объединяет структуры.

5.1 Массовое обновление

Добавим поле `"source": "web"` во все логи.

Оператор `||` объединяет два JSON-объекта. Если ключ уже существовал, он перезаписывается. Если нет — добавится.

Листинг 11. Массовое обновление

```
UPDATE user_logs
SET event_data = event_data || '{"source": "web"}'::jsonb;

SELECT event_data FROM user_logs LIMIT 3;
```

	event_data
1	{"ip": "192.168.1.1", "event": "login", "source": "web", "success": true}
2	{"ip": "192.168.1.25", "event": "login", "source": "web", "success": true}
3	{"event": "sale", "source": "web", "details": {"quantity": 9, "total_sum": 359.36, "medicine_id": 2}}

Рисунок 7 – Результат добавления поля source

5.2 Точечное изменение (функция jsonb_set)

Изменим статус успеха для одной конкретной записи входа.

Функция `jsonb_set(target, path, new_value)` создает **новую версию** JSON-документа, в которой значение по указанному пути заменено на новое.

Путь указывается как массив текстовых ключей: `{details, quantity}`.

Новое значение должно быть типа **JSONB**.

Листинг 12. Использование jsonb_set

```
UPDATE user_logs
SET event_data = jsonb_set(event_data, '{success}', 'false')
WHERE log_id = (SELECT min(log_id) FROM user_logs WHERE event_data ->>
'event' = 'login');

SELECT event_data FROM user_logs WHERE event_data ->> 'success' = 'false'
LIMIT 1;
```


	 event_data
1	{"ip": "192.168.1.198", "event": "login", "source": "web", "success": false}

Рисунок 8 – Результат изменения статуса

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Почему для хранения результатов хеш-функций (SHA-256) тип BYTEA лучше, чем TEXT?
2. В чем главное отличие JSONB от JSON в PostgreSQL?
3. Что произойдет при попытке вставить в секционированную таблицу данные, дата которых не попадает ни в одну из созданных секций (при отсутствии секции DEFAULT)?
4. Чем отличается оператор `->` от `->>`?
5. Как работает оператор `@>` при фильтрации данных?

КРАТКИЙ СПРАВОЧНЫЙ МАТЕРИАЛ