



МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение
высшего образования

«МИРЭА – Российский технологический университет»

РТУ МИРЭА

**Институт информационных технологий (ИИТ)
Кафедра цифровой трансформации (ЦТ)**

ОТЧЕТ ПО ПРАКТИЧЕСКОЙ РАБОТЕ
по дисциплине «Разработка баз данных»

Практическое занятие № 8

Студенты группы *ИКБО-66-23 Смирнов А.Ю.*

(подпись)

Ассистент *Копылова Я.А.*

(подпись)

Отчет представлен «__» _____ 2025 г.

Москва 2025 г.

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 car type

	AZ id_car_type	AZ type_name	I23 seats	I23 luggage_capacity	AZ fuel_type	AZ transmission
1	CT001	Эконом	5		2 Бензин	Механика
2	CT002	Комфорт	5		3 Бензин	Автомат
3	CT003	Бизнес	5		3 Дизель	Автомат
4	CT004	Премиум	5		4 Бензин	Автомат

Таблица 2. Таблица insurance

	AZ id_insurance	AZ insurance_name	I23 daily_cost	AZ coverage_description	is_active
1	INS003	Премиум	800	Покрытие без франшизы	[v]
2	INS001	Стандарт	300	Базовое покрытие	[v]
3	INS002	Расширенная	500	Полное покрытие	[v]

Таблица 3. Таблица client

	AZ ast_name	AZ middle_name	AZ driver_license	AZ phone	AZ email	AZ passport_number	registration_date	AZ client_type
1	Леонова	[NULL]	7788654321	+79180001122	larisa@gmail.com	4510987654	2025-09-25	VIP
2	Волков	[NULL]	77CC112233	+79182223344	volkov@yandex.ru	4510567890	2025-09-25	Corporate
3	Морозов	Z	77AA123456	+79181112233	oleg@mail.ru	4510123456	2025-09-25	Regular

Таблица 4. Booking status

	AZ description
1 BS001	Ожидание
2 BS002	Подтверждена
3 BS003	Активна
4 BS004	Завершена
5 BS005	Отменена

Таблица 5. Таблица car

	AZ id_location	AZ id_insurance	AZ brand	AZ model	I23 production_year	AZ license_plate	AZ color	I23 mileage	AZ status	last_maintenance_date
1	LOC001	INS001	Kia	Rio	2 023	A123BC77	Белый	15 000	Available	[N/A]
2	LOC001	INS002	Toyota	Camry	2 023	B456HE77	Черный	18 000	Available	[N/A]
3	LOC002	INS002	BMW	5 Series	2 024	C789TX77	Синий	5 000	Available	[N/A]
4	LOC003	INS003	Mercedes	E-Class	2 023	E111AA77	Черный	12 000	Available	[N/A]

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'), ('Менеджер аренды'), ('Агент поддержки');
```

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),
('rent_manager', 'RentCar2025', digest('RentCar2025', 'sha256'), 2),
('support_agent', 'Support2025!', digest('Support2025_bad', 'sha256'), 3);

-- 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	is_valid_check
1	admin_user	StrongPass123!	805bd951772627f3d1a607084df1727c6caad60447c5d73feb7be2d2fe17fd8	[v]
2	rent_manager	RentCar2025	f1d5ab23a685ec392e8370028f7814e1dd746a5eef202e9be8bcaa7ed8f07ea1	[v]
3	support_agent	Support2025!	f260ad2baf2aad4beadcf4245fd94c2b503a25aac5c4f42c7fdbd396e066d6f8	[]

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);
```

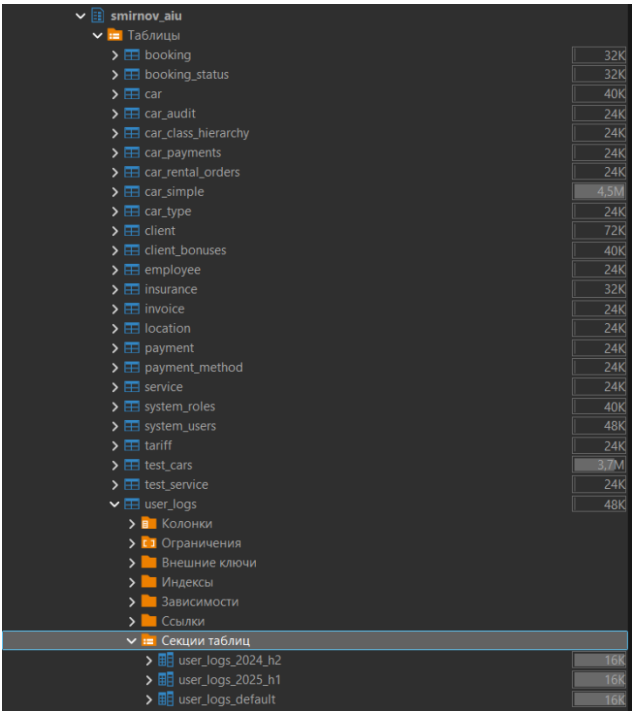


Рисунок 2 – Структура таблиц в навигаторе объектов (видна иерархия)

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 бронирование (booking), мы сохраняем ID автомобиля (car_id), количество дней аренды (days) и сумму (total_amount).

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 := 'booking';
    json_payload := jsonb_build_object(
        'event', event_type,
        'details', jsonb_build_object(
            'car_id', floor(random() * 4 + 1)::TEXT,
            'days', floor(random() * 10 + 1),
            'total_amount', (random() * 10000)::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;

```

	Az partition_name ▼	123 count ▼
1	2024_h2	10 167
2	2025_h1	9 833
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).

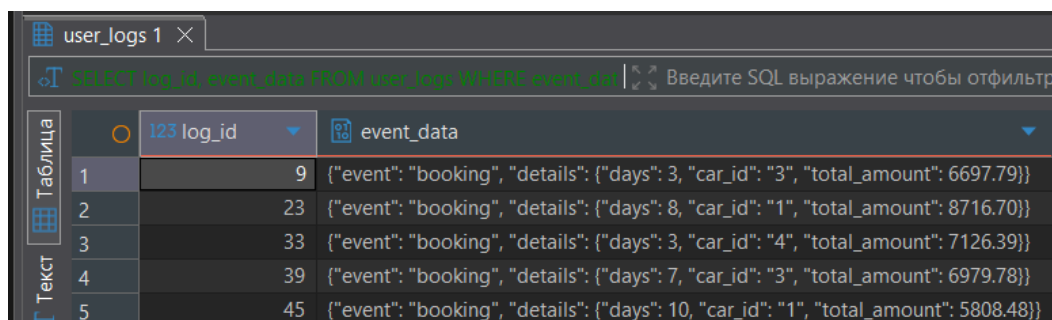
4.1 Поиск по ключу и фильтрация с условием

Найдем все бронирования, где сумма аренды превышает 5000 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' = 'booking'
AND (event_data -> 'details' ->> 'total_amount')::NUMERIC > 5000
LIMIT 5;
```



The screenshot shows a database interface with a query window titled 'user_logs 1'. The query is: `SELECT log_id, event_data FROM user_logs WHERE event_data ->> 'event' = 'booking' AND (event_data -> 'details' ->> 'total_amount')::NUMERIC > 5000 LIMIT 5;`. Below the query, there is a table view showing 5 rows of data. The first column is 'log_id' and the second is 'event_data'.

log_id	event_data
9	{"event": "booking", "details": {"days": 3, "car_id": "3", "total_amount": 6697.79}}
23	{"event": "booking", "details": {"days": 8, "car_id": "1", "total_amount": 8716.70}}
33	{"event": "booking", "details": {"days": 3, "car_id": "4", "total_amount": 7126.39}}
39	{"event": "booking", "details": {"days": 7, "car_id": "3", "total_amount": 6979.78}}
45	{"event": "booking", "details": {"days": 10, "car_id": "1", "total_amount": 5808.48}}

Рисунок 4 – Результат выборки дорогих покупок

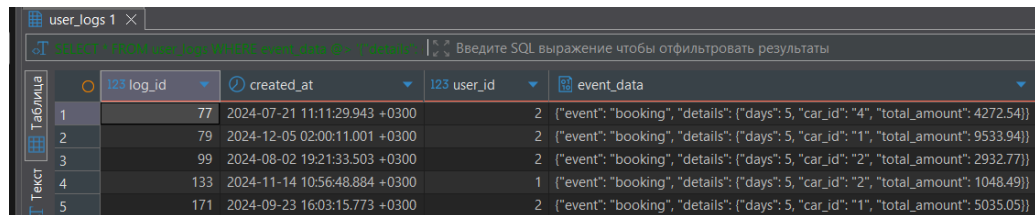
Обратите внимание: мы используем **->**, чтобы «провалиться» в объект details, а затем **->>**, чтобы получить текст числа, и приводим его к **::NUMERIC** для математического сравнения.

4.2 Фильтрация по вхождению (JSON Containment)

Найдем все записи, содержащие конкретную структуру: бронирование на 5 дней аренды любого автомобиля.

Листинг 9. Поиск по вхождению фрагмента

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



	log_id	created_at	user_id	event_data
1	77	2024-07-21 11:11:29.943 +0300	2	{"event": "booking", "details": {"days": 5, "car_id": "4", "total_amount": 4272.54}}
2	79	2024-12-05 02:00:11.001 +0300	2	{"event": "booking", "details": {"days": 5, "car_id": "1", "total_amount": 9533.94}}
3	99	2024-08-02 19:21:33.503 +0300	2	{"event": "booking", "details": {"days": 5, "car_id": "2", "total_amount": 2932.77}}
4	133	2024-11-14 10:56:48.884 +0300	1	{"event": "booking", "details": {"days": 5, "car_id": "2", "total_amount": 1048.49}}
5	171	2024-09-23 16:03:15.773 +0300	2	{"event": "booking", "details": {"days": 5, "car_id": "1", "total_amount": 5035.05}}

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.*).

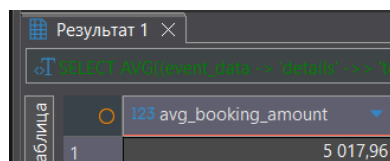
PostgreSQL проверяет только наличие указанных ключей и совпадение их значений. Остальное игнорируется.

4.3 Агрегация данных

Посчитаем среднюю сумму бронирования по данным из JSON

Листинг 10. Агрегация данных

```
SELECT
AVG((event_data -> 'details' ->> 'total_amount')::NUMERIC)::NUMERIC(10,2) AS
avg_booking_amount
FROM user_logs
WHERE event_data ->> 'event' = 'booking';
```



	avg_booking_amount
1	5 017,96

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

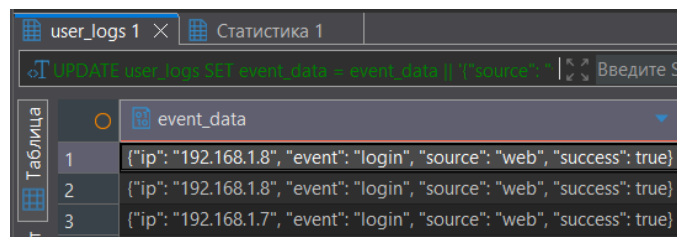
Добавим поле `"source": "web"` во все логи.

Оператор `||` объединяет два JSON-объекта. Если ключ уже существовал, он перезапишется. 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": "192.168.1.8", "event": "login", "source": "web", "success": true}
2	{"ip": "192.168.1.8", "event": "login", "source": "web", "success": true}
3	{"ip": "192.168.1.7", "event": "login", "source": "web", "success": true}

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;
```

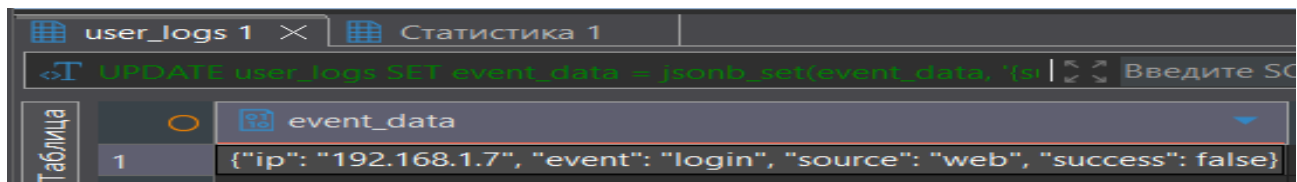


Рисунок 8 – Результат изменения статуса

