

**Федеральное государственное автономное образовательное учреждение высшего  
образования**

**«Национальный исследовательский университет «Высшая школа экономики»  
Факультет компьютерных наук**

**О Т Ч Е Т**  
**по дисциплине**  
**«Теория Баз Данных»**

**Выполнил студент гр. 184**

**Савосин Артем**

**Проверил: Незнанов Алексей Андреевич**

*18.12.2020*

**2020 год**

# Содержание

<b>Техническое задание.</b>	<b>3</b>
<b>Концептуальное проектирование:</b>	<b>3</b>
Краткое описание предметной области.	3
Описание процесса построения инфологической модели с обоснованием выделения сущностей и связей.	3
ER-диаграмма с комментариями.	6
<b>Проектирование реляционной модели:</b>	<b>7</b>
Описание процесса перехода к реляционной модели с акцентом на представлении связей «многие ко многим» и наследования.	7
Диаграмма схемы БД.	7
Дополнительные механизмы обеспечения целостности данных.	8
<b>Развёртывание БД в выбранной СУБД</b>	<b>8</b>
DDL-скрипт создания схемы БД.	8
Создание таблиц с ограничениями на внешние ключи.	8
Триггеры и хранимые процедуры	10
Примеры DML-операторов вставки тестовых данных.	13
<b>Разработка клиентского приложения</b>	<b>17</b>
Архитектура.	17
Сценарии использования.	17
Организация доступа к данным.	17
Интерфейс с пользователем.	17
Отчеты.	19
Результаты функционального тестирования.	19
<b>Заключение.</b>	<b>20</b>
<b>Источники</b>	<b>20</b>

## **1. Техническое задание.**

Разработка базы данных для локального магазина товаров для рукоделия. Основные задачи базы данных - хранение данных о пользователях, совершенных ими заказах, отслеживание активности менеджеров, хранение данных о товарах на складе.

## **2. Концептуальное проектирование:**

### **Краткое описание предметной области.**

Данный магазин продает две основные категории товаров - пряжа, производимая магазином и периферийные товары для рукоделия.

Основная цель проекта - на основе работающей системы магазина создать базу данных, которая помогает отслеживать наличие товара, хранить, анализировать и обрабатывать поступающие заказы, хранить данные покупателей для последующей работы с ними, проверять активность сотрудников.

На данный момент алгоритм работы магазина:

1. Пользователь собирает заказ из предметов, выставленных в магазине
2. Менеджер получает заказ и проверяет наличие всех пунктов на складе
3. Менеджер связывается с пользователем для подтверждения и оплаты заказа
4. Менеджер собирает и отправляет заказ
5. В случае, если товаров не достаточно, менеджер уведомляет покупателя.
6. Менеджер может добавить новые товары на склад или обновить количество товара на складе.

### **Описание процесса построения инфологической модели с обоснованием выделения сущностей и связей.**

Для начала нужно определиться с основными сущностями в проекте. Исходя из описания предметной области, можно заметить, что можно выделить следующие сущности и атрибуты у них:

#### **1. Товар**

- Id товара
- Артикул товара - то, как товар приходит по накладной
- Цена товара - цена за единицу данного товара.

- Всего доступно товара - суммарное количество единиц товара на всех складах - нужно для проверки того, не превосходит ли число товара в заказе этого максимума.

Из описания видно, что магазин продает два типа товаров, с разными характеристиками, но имеющими одинаковые признаки (которые находятся в товаре сейчас). Отсюда мы можем создать еще две сущности, которые будут наследовать признаки от Товара.

#### 2. Пряжа (наследуется от товара)

- Цвет
- Длина
- Материал, из которого изготовлена пряжа

#### 3. Прочие товары (наследуется от товара)

- Описание
- Наименование

Все товары хранятся на одном из складов магазина. Следовательно, нам хочется иметь в своей базе адреса и вместимость всех складов. Создаем для этого сущность.

#### 4. Склад

- Id
- Улица
- Город
- Вместимость - сколько единиц товара может принять склад
- Доступно места - Вместимость - занято место товарами.

Далее из описания видим, что покупатель будет выбирать товары и заказывать их, а работник магазина будет проверять и подтверждать заказ. Из этого следует, что нам нужно создать еще несколько сущностей.

#### 5. Работник

- Обработано заказов - будем считать продуктивность
- Имя
- Фамилия
- Номер телефона - храним, чтобы дать возможность связаться с работником.
- Id

#### 6. Покупатель

- Имя
- Фамилия
- Номер телефона
- Адрес эл.почты - храним для рассылки акций / предложений.

- Улица - храним для доставки.
- Город - храним для доставки.
- Id

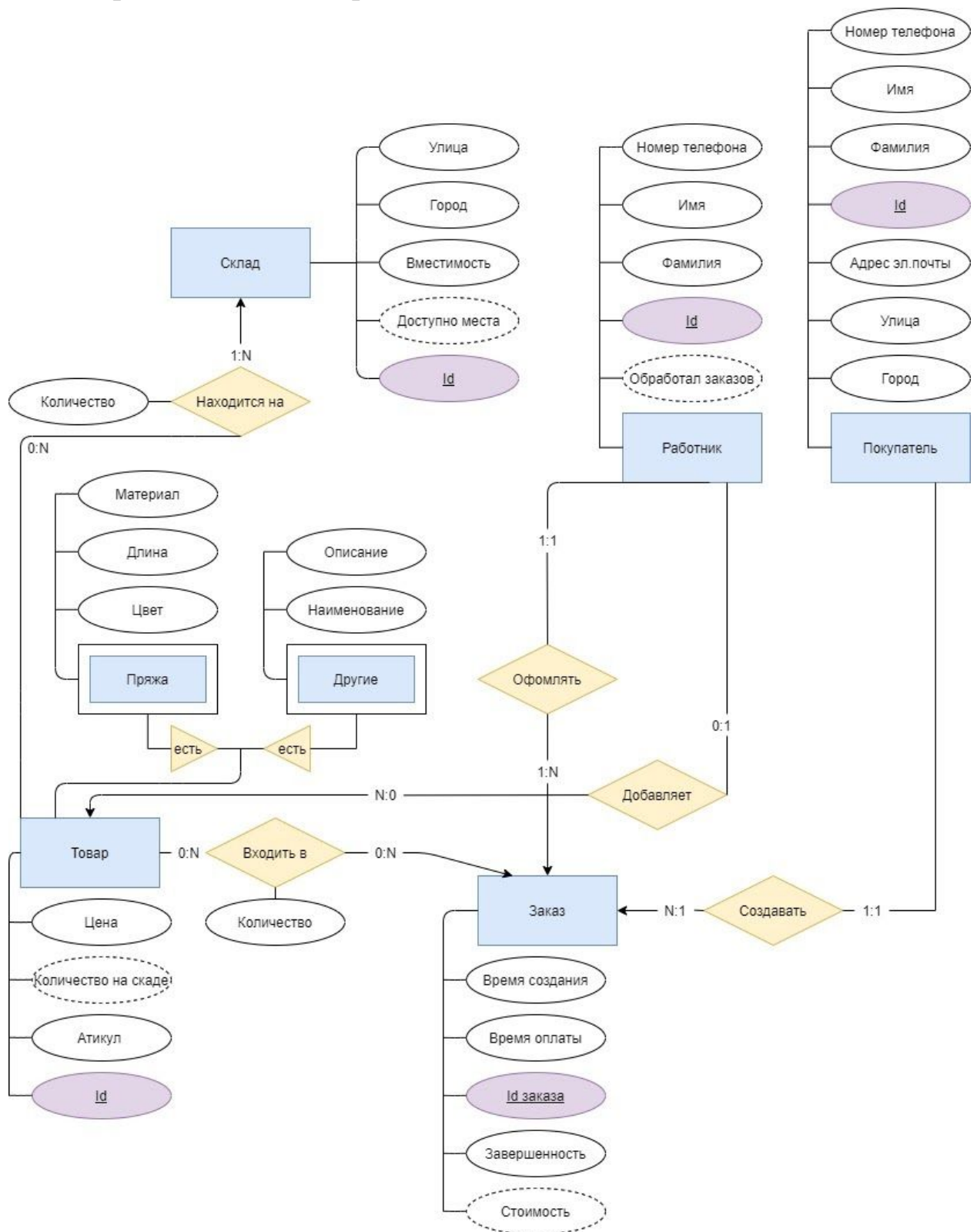
## 7. Заказ

- Id
- Время создания
- Время оплаты
- Стоимость - исчисляется как суммарная стоимость всех товаров в заказе.
- Завершенность заказа - выставляется True, когда заказ дошел до покупателя.

Также нам нужно выделить несколько связей.

- Наследование пряжи от товара.
- Наследование прочих товаров от товара.
- Связь работник оформляет заказ. Здесь ровно один работник обязательно должен оформить заказ. При этом каждый из работников может оформлять множество заказов.
- Аналогично с покупателем и связью покупатель создает заказ. Заказ создается ровно одним покупателем, покупатель может создать от 1 до бесконечности заказов.
- Связь товар входит в заказ. Товар не обязательно может войти в заказ, при этом может войти в любом количестве. В заказе может содержаться неограниченное число товаров.
- Товар находится на складе. Товар обязательно должен находиться на одном или нескольких складах. При этом на определенном складе может быть от 0 до бесконечности товара.
- Также хотим отслеживать, какой новый товар был добавлен каким из работников. Чтобы потом смотреть, какие товары лучше продаются, кто из работников выбрал его и добавил в магазин. Связь работник добавляет товар. Каждый работник может добавить товар, а может и не добавить. Товар может добавить только один работник. Работник может добавить от 0 до бесконечности товаров.

## ER-диаграмма с комментариями.



ER-диаграмма выполнена в нотации Чена. Синим цветом выделены сущности. Фиолетовым - их ключи. Желтым выделены связи. Слабые сущности обведены рамкой. Описание сущностей и связей выполнено в прошлом пункте.

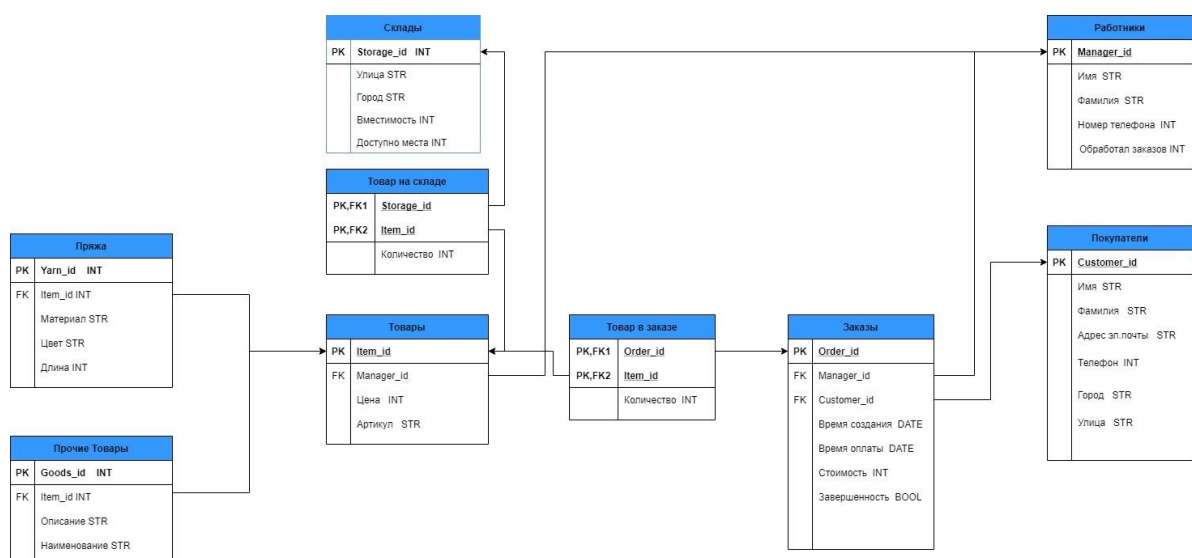
### 3. Проектирование реляционной модели:

Описание процесса перехода к реляционной модели с акцентом на представлении связей «многие ко многим» и наследования.

1. Каждая сущность формирует таблицу. Также будут сформированы таблицы для связей многие ко многим - товар в заказе и товар на складе.
2. В силу того, что сущности Пряжа и Прочие Товары - слабые и наследуются от товара, нам нужно добавить в каждую из них внешний ключ - id товара.
3. Для сущности Заказ создаем целых два внешних ключа - id работника и id покупателя, которые отражают связь “менеджер оформляет заказ” и “покупатель создает заказ”.
4. У сущности товар создаем внешний ключ id работника - отображает связь “работник добавил товар”.
5. Перейдем к связям многие ко многим.
  - 5.1. Товар на складе. Ключ отношения - пара (id склада, id товара).  
Дополнительным атрибутом будет количество данного товара на складе.
  - 5.2. Товар в заказе. Ключ отношения - пара (id заказа, id товара).  
Дополнительным атрибутом будет количество данного товара в заказе.

В качестве ограничения целостности всех связей будем использовать RESTRICT - запрет на удаление сущности, первичный ключ которой является для другой сущности внешним.

#### а. Диаграмма схемы БД.



## Дополнительные механизмы обеспечения целостности данных.

Для дополнительного обеспечения целостности данных будут использоваться триггеры, описанные в следующем пункте.

### 4. Развёртывание БД в выбранной СУБД

В качестве СУБД для выполнения проекта была выбрана PostgreSQL и диалект `plpgsql`. Давайте рассмотрим все этапы написания DDL скрипта и какие пункты были выполнены.

#### DDL-скрипт создания схемы БД.

##### Создание таблиц с ограничениями на внешние ключи.

Первым делом необходимо создать все таблицы, описанные в TR-диаграмме.

```
CREATE TABLE Customers (  
    id_customer SERIAL PRIMARY KEY,  
    FirstName TEXT,  
    Surname TEXT,  
    phone_number TEXT NOT NULL UNIQUE,  
    city TEXT,  
    street TEXT,  
    email TEXT UNIQUE  
);  
  
CREATE TABLE Managers (  
    id_Manager SERIAL PRIMARY KEY,  
    FirstName TEXT,  
    Surname TEXT,  
    phone_number TEXT NOT NULL UNIQUE,  
    orders_closed INTEGER  
);  
  
CREATE TABLE Storages (  
    id_storage SERIAL PRIMARY KEY,  
    city TEXT,  
    street TEXT,  
    capacity INTEGER,  
    available_place INTEGER  
);
```

Первыми создаем таблицы для сущностей, которые входят в большое количество связей, основной ключ которых будет являться внешним ключом для одной из последующих таблиц. Далее создаем еще одну важную таблицу - от нее будут наследоваться другие сущности, она входит в несколько связей.

```
CREATE TABLE Items (  
    id_item SERIAL PRIMARY KEY,  
    cost INTEGER,  
    articl TEXT,  
    id_Manager INTEGER,  
    FOREIGN KEY (id_Manager) REFERENCES Managers(id_manager) ON DELETE RESTRICT
```

Здесь видим первое ограничение - мы не сможем удалить работника, который добавил данный товар в магазин до тех пор, пока в базе есть товары, добавленные этим работником.

Далее добавил таблицы слабых сущностей, наследуемых от Items.



```
CREATE TABLE Yarn (
    id_yarn SERIAL PRIMARY KEY,
    material TEXT,
    color TEXT,
    length_m REAL,
    id_item INTEGER,
    FOREIGN KEY (id_item) REFERENCES Items(id_item) ON DELETE RESTRICT
);
```

```
CREATE TABLE Goods (
    id_good SERIAL PRIMARY KEY,
    goodName TEXT,
    textDescript TEXT,
    id_item INTEGER,
    FOREIGN KEY (id_item) REFERENCES Items(id_item) ON DELETE RESTRICT
);
```

Здесь накладывается аналогичное ограничение - не сможем удалить сущность item, пока в базе будет связанная с ней сущность yarn или good, что логично, ведь основные данные про слабую сущность лежат в items, и будет бессмысленно хранить данные по слабой сущности без ее сильной части.

Создадим самую большую и самую информативную таблицу для нашей базы данных - таблицу заказов.

```
CREATE TABLE Orders (
    id_order SERIAL PRIMARY KEY,
    created_time timestamp,
    payment_time timestamp,
    total_cost INTEGER,
    is_Finished INTEGER,
    id_customer INTEGER,
    id_manager INTEGER,
    FOREIGN KEY (id_customer) REFERENCES Customers(id_customer) ON DELETE RESTRICT,
    FOREIGN KEY (id_manager) REFERENCES Managers(id_manager) ON DELETE RESTRICT
);
```

Здесь присутствуют два аналогичных ограничения на удаление связанных сущностей. Помимо создания всех сущностей, так же необходимо создать таблицы для связей многие-ко-многим. У нас таких связей две и она практически идентичны.

```
CREATE TABLE Storage_item (
    amount INTEGER,
    id_storage INTEGER NOT NULL,
    id_item INTEGER NOT NULL,
    FOREIGN KEY (id_storage) REFERENCES Storages(id_storage),
    FOREIGN KEY (id_item) REFERENCES Items(id_item)
);
```

```
CREATE TABLE Order_item (
    amount INTEGER DEFAULT 1,
    id_order INTEGER,
    id_item INTEGER,
    FOREIGN KEY (id_order) REFERENCES Orders(id_order),
    FOREIGN KEY (id_item) REFERENCES Items(id_item)
);
```

В таблицы внесены ключи сущностей с ограничениями, связанных этой связью и поле amount - сколько сущностей item входит в эту связь.

Помимо создания таблиц с ограничениями на удаление, в базе данных нужно поддерживать еще несколько ограничений, а также поддерживать несколько вычисляемых полей в актуальном состоянии. Для этого создадим

## Триггеры и хранимые процедуры

```
CREATE OR REPLACE FUNCTION count_item_everywhere (INT)
    RETURNS INT
AS $$
BEGIN
    RETURN
        SUM(storage_item.amount)
    FROM
        storages JOIN storage_item ON storage_item.id_storage = storages.id_storage JOIN items ON storage_item.id_item = items.id_item
    WHERE
        items.id_item = $1;
END: $$
```

Первая функция, которая понадобится нам для реализации остальных - count\_item\_everywhere(id) - данная функция получает на вход айди товара, а далее вычисляет, сколько товара суммарно находится на каждом из складов. Данную функцию применим в триггере, который будет ограничивать количество товара, добавляемого в заказ. Если в заказе будет выставлено число большее, чем мы можем добыть товара со склада - мы выкинем ошибку. Давайте создадим такой триггер.

```
CREATE OR REPLACE FUNCTION check_available_amount_trg()
    RETURNS trigger AS
    $$
BEGIN
    IF (count_item_everywhere(NEW.id_item)) - NEW.amount < 0
    THEN
        RAISE EXCEPTION 'Not enough items in storages';
    END IF;
    RETURN NEW;
END;
$$
LANGUAGE 'plpgsql';
```

```
CREATE TRIGGER check_available_amount
    BEFORE INSERT
    ON "order_item"
    FOR EACH ROW
    EXECUTE PROCEDURE check available amount trg();
```

Аналогичный триггер создал и для UPDATE, он отличается лишь одной строчкой, поэтому не будем его вставлять. Работая с заказами, нам также нужно поддерживать актуальным одно из вычисляемых полей - total\_cost - которое отображает финальную цену заказа в данный момент времени. Создадим триггер, который будет пересчитывать стоимость заказа при каждой вставке / обновлении / удалении товара в заказе.

```
CREATE OR REPLACE FUNCTION order_cost_count_trg()
RETURNS trigger AS
$$
BEGIN
    UPDATE orders SET total_cost = (SELECT SUM(cost * amount) FROM Order_item JOIN Items ON Items.id_item = Order_item.id_item
    WHERE order_item.id_order = NEW.id_order) WHERE orders.id_order = new.id_order;
    RETURN NEW;
END;
$$
LANGUAGE 'plpgsql';

CREATE TRIGGER order_cost_count
AFTER INSERT
ON "order_item"
FOR EACH ROW
EXECUTE PROCEDURE order_cost_count_trg();
```

Аналогичные функции применил для вставки и удаления, они отличаются только в одном поле.

Теперь поговорим о складах - здесь так же нужно поддерживать несколько ограничений. Во-первых, мы хотим всегда получать актуальное значение поля available\_place, которое покажет нам сколько еще товаров можно добавить на склад, и мы не хотим добавлять на склад больше вещей, чем этот склад мог бы вместить. Давайте реализуем обе функции проверки и обновления.

```
CREATE OR REPLACE FUNCTION storage_place_count_trg()
RETURNS trigger AS
$$
BEGIN
    UPDATE Storages SET available_place = (SELECT capacity - SUM(amount) FROM storage_item JOIN Items ON Items.id_item = storage_item.id_item
    WHERE storage_item.id_storage = NEW.id_storage) WHERE Storages.id_storage = new.id_storage;
    RETURN NEW;
END;
$$
LANGUAGE 'plpgsql';

CREATE TRIGGER storage_place_count
AFTER INSERT
ON "storage_item"
FOR EACH ROW
EXECUTE PROCEDURE storage_place_count_trg();
```

Аналогичные реализованы для вставки и обновления.

Теперь реализуем функцию проверки.

```

CREATE OR REPLACE FUNCTION check_available_place_trg()
RETURNS trigger AS
$$
BEGIN
    IF (SELECT avialable_place FROM storages WHERE id_storage = NEW.id_storage) - NEW.amount < 0
    THEN
        RAISE EXCEPTION 'No place for such amount';
    END IF;
    RETURN NEW;
END;
$$
LANGUAGE 'plpgsql';

```

```

CREATE TRIGGER check_available_place
BEFORE INSERT
ON "storage_item"
FOR EACH ROW
EXECUTE PROCEDURE check_available_place_trg();

```

Отмечу, что данная функция реализуется BEFORE INSERT / UPDATE, что значит, что если она выдаст ошибку, то мы не добавим ничего лишнего, и нам не придется откатывать изменения и пересчитывать поле available\_place.

Бонусом к данным ограничениям, я добавил еще четыре хранимые процедуры - по айди заказа и склада они вернут таблицу, в которой будут указаны товары в этом заказе / на этом складе, а так же в каком они количестве. Еще одна процедура вернет для каждого товара сколько раз его продали, а последняя - сколько товара, добавленного каждым менеджером было продано. Пример с заказом:

```

CREATE OR REPLACE FUNCTION get_order_items (ind INT)
RETURNS TABLE (
    articul TEXT,
    amount INT,
    total_cost INT
)
AS $$
BEGIN
    RETURN QUERY SELECT
        items.articul,
        order_item.amount,
        order_item.amount * items.cost AS total_cost
    FROM
        orders JOIN order_item ON order_item.id_order = orders.id_order JOIN items ON order_item.id_item = items.id_item
    WHERE
        orders.id_order = ind;
END; $$

LANGUAGE 'plpgsql';

```

Код с подсчетом проданных товаров:

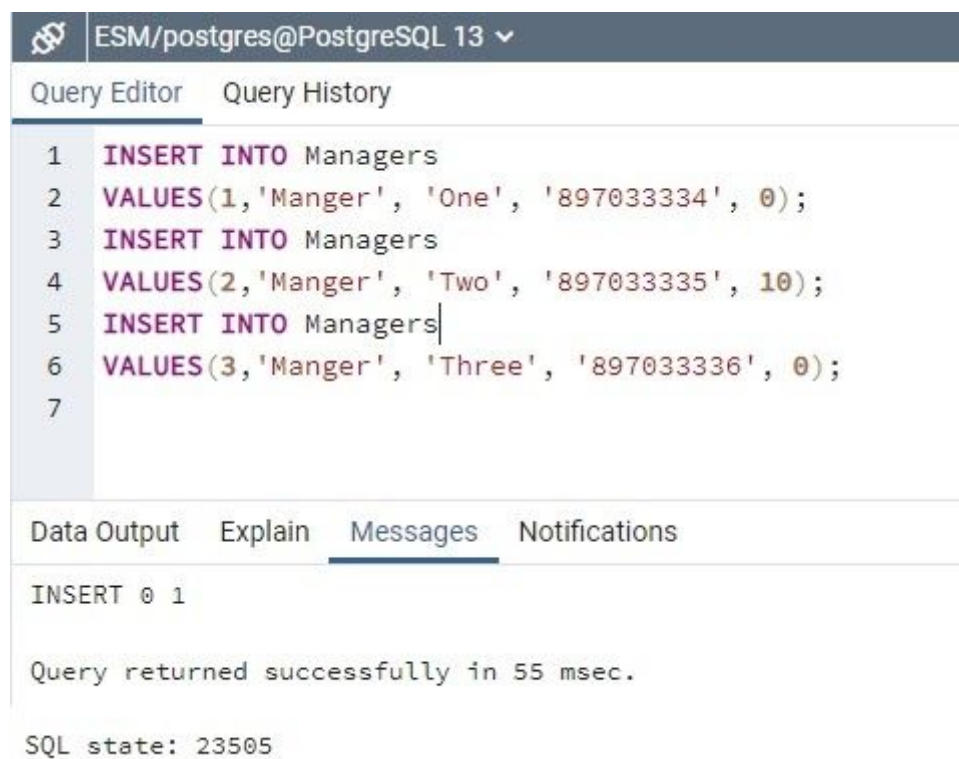
```
CREATE OR REPLACE FUNCTION items_sold_by_manager ()
    RETURNS TABLE (
        surname TEXT,
        amount bigint
    )
AS $$
BEGIN
    RETURN QUERY
        SELECT managers.surname as surname, SUM(order_item.amount) as amount FROM managers
        JOIN items ON managers.id_manager = items.id_manager
        JOIN order_item ON items.id_item = order_item.id_item
        GROUP BY managers.surname;
END; $$

LANGUAGE 'plpgsql';
```

### Примеры *DML*-операторов вставки тестовых данных.

Перед проверкой ограничений целостности, базу нужно заполнить значениями.

Давайте внесем некоторые значения, а потом проверим, как выполняются ограничения.



The screenshot shows a PostgreSQL query editor interface. At the top, the database connection is 'ESM/postgres@PostgreSQL 13'. Below the connection bar, there are two tabs: 'Query Editor' and 'Query History'. The 'Query Editor' tab is active, displaying the following SQL code:

```
1 INSERT INTO Managers
2 VALUES(1,'Manger', 'One', '897033334', 0);
3 INSERT INTO Managers
4 VALUES(2,'Manger', 'Two', '897033335', 10);
5 INSERT INTO Managers
6 VALUES(3,'Manger', 'Three', '897033336', 0);
7
```

Below the query editor, there are four tabs: 'Data Output', 'Explain', 'Messages', and 'Notifications'. The 'Messages' tab is active, showing the following output:

```
INSERT 0 1

Query returned successfully in 55 msec.

SQL state: 23505
```

Таким же образом заполнили несколько покупателей. Теперь добавим значения в другие таблицы.



```
ESM/postgres@PostgreSQL 13
Query Editor  Query History
1  INSERT INTO Storages
2  Values(0, 'Astana', 'Dykenyli 32', 1000, 1000);
3  Insert INTO Items
4  Values(0, 1800, 'Y-001', 1);
5  Insert INTO Orders
6  Values(4, null, null, 0, 0, 1, 1);

Data Output  Explain  Messages  Notifications
INSERT 0 1

Query returned successfully in 51 msec.
```

Здесь добавлен один склад, вместимостью 1000 предметов. Так же добавлен один товар, стоимостью 1800. Далее будем проверять ограничения целостности с помощью этих данных.

### Контроль работы ограничений целостности с примерами.

Проверим, что произойдет, если добавить двух пользователей с одинаковыми номерами телефона.

```
ESM/postgres@PostgreSQL 13
Query Editor  Query History
1  INSERT INTO CUSTOMERS
2  VALUES(9, 'Artyom', 'Savossin', '897033334', 'Astana', '175,4', 'new_post@example.com');
3  INSERT INTO CUSTOMERS
4  VALUES(10, 'Anonim', 'No', '897033334', 'Astana', '175,4', 'newer_post@example.com');

Data Output  Explain  Messages  Notifications
ERROR: ОШИБКА: повторяющееся значение ключа нарушает ограничение уникальности "customers_phone_number_key"
DETAIL: Ключ "(phone_number)=(897033334)" уже существует.

SQL state: 23505
```

Получаем ошибку, второй пользователь с одинаковым телефоном добавлен не будет.

Аналогичные ошибки получим при попытке добавить любую сущность с повтором в поле, на котором проставлено ограничение UNIQUE.

Также при попытке удаления элемента, ключ которого является внешним для какого-либо элемента, удаление отменяется.

```
ESM/postgres@PostgreSQL 13
Query Editor  Query History
1  INSERT INTO yarn
2  VALUES (0, 'трикотаж', '#ffc0cb', 100, 0);
3
4  DELETE FROM items
5  WHERE id_item = 0

Data Output  Explain  Messages  Notifications
ERROR: ОШИБКА: UPDATE или DELETE в таблице "items" нарушает ограничение внешнего ключа "yarn_id_item_fkey" таблицы "yarn"
DETAIL:  На ключ (id_item)=(0) всё ещё есть ссылки в таблице "yarn".
```

Далее проверим ограничение по заполняемости склада - как помним, вместимость тестового склада была 1000 единиц товара.

```
ESM/postgres@PostgreSQL 13
Query Editor  Query History
1  INSERT INTO storage_item
2  VALUES (1001,0,0)

Data Output  Explain  Messages  Notifications
ERROR: ОШИБКА: No place for such amount
CONTEXT:  функция PL/pgSQL check_available_place_trg(), строка 5, оператор RAISE

SQL state: P0001
```

Получим ошибку. Теперь попробуем добавить число меньшее, чем доступно места на складе.

```
ESM/postgres@PostgreSQL 13
Query Editor Query History
1 INSERT INTO storage_item
2 VALUES(999,0,0)

Data Output Explain Messages Notifications
INSERT 0 1

Query returned successfully in 54 msec.
```

Сработало! Проверим ограничение добавления товара в заказ в объеме большем, чем есть на всех складах. В заказ с id = 1 добавляем товар с id = 0 (его 999 на первом складе).

```
ESM/postgres@PostgreSQL 13
Query Editor Query History
1 INSERT INTO order_item
2 VALUES(1001, 1, 0)

Data Output Explain Messages Notifications
ERROR: ОШИБКА: Not enough items in storages
CONTEXT: функция PL/pgSQL check_available_amount_trg(), строка 5, оператор RAISE
```

Получили ошибку, которую не получили бы при достаточном количестве товара на складе:

```
ESM/postgres@PostgreSQL 13
Query Editor Query History
1 INSERT INTO storage_item
2 VALUES(5, 1, 0);
3
4 INSERT INTO order_item
5 VALUES(1001, 1, 0);

Data Output Explain Messages Notifications
INSERT 0 1

Query returned successfully in 86 msec.
```

Добавил новый склад, и на него 5 единиц нашего товара, таким образом, суммарное количество товаров на всех складах стало 1004, что меньше 1001.



## 5. Разработка клиентского приложения

### Архитектура.

Проект выполнен в виде чат-бота в telegram. Для создания бота использовал библиотеку python TELEBOT, для общения бота и сервера с БД использовалась библиотека PSYCOPG2. Пользователь может взаимодействовать с базой данных путем простого общения с ботом, либо создания кастомных запросов на языке SQL и их отправки боту. Давайте подробнее рассмотрим структуру бота.

```
import telebot;
from telebot import types
bot = telebot.TeleBot('1437332614: [REDACTED]wKAbiqM');
```

Бот создается с помощью стандартной функции из библиотеки, в которой я указываю токен своего бота. Далее нужно организовать обработку сообщений. Давайте рассмотрим несколько примеров.

```
@bot.message_handler(content_types=['text'])
def start(message):
    print(message.text)
    if message.text == '/reg' or name == '' or password == '':
        bot.send_message(message.from_user.id, "Сейчас ты будешь тестировать функционал бота. Я спрошу у тебя логин")
        bot.register_next_step_handler(message, get_name); #следующий шаг - функция get_name
    elif message.text == '/close':
        conn.close()
        bot.send_message(message.from_user.id, 'Закружили соединение с базой.');
```

В данной функции организован основной функционал бота - регистрация пользователя, завершение работы и создание нового запроса к БД.

Подключение к БД осуществляется с помощью библиотеки PSYCOPG2, после получения пароля и логина пользователя:

```
bot.send_message(call.message.chat.id, 'Отлично! Сейчас я попробую авторизировать тебя...');
try:
    global conn
    global cur
    conn = psycopg2.connect(
        host='localhost',
        database='ESM',
        user=name,
        password=password
    )
    conn.autocommit = True
    cur = conn.cursor()
    bot.send_message(call.message.chat.id, 'Логин и пароль верный, работаем!');
```

Отмечу, что у бота есть три основных видов запросов - кастомный запрос на SQL, отчет путем запроса к хранимой процедуре, простые запросы к базе с помощью кнопок чат-бота. Для корректной работы запросов нужно реализовать еще несколько функций, подобных `start(message)`, которые будут получать новое сообщение и показывать пользователю варианты действий. Все функции довольно схожи с показанной выше, за исключением того, что вызывают в конце одну из функций, осуществляющих запрос к базе - функции создают из строк новый запрос и с помощью `PSYCOPG2` посылают его в БД, получают ответ и выводят его пользователю. Вот пример некоторых из них:

```
def get_data_by_id():
    global conn
    global cur
    try:
        if action == 'storage-items':
            cur.execute('SELECT * FROM ' + 'get_storage_items(' + id_item + ')'+ ';')
        elif action == 'order-items':
            cur.execute('SELECT * FROM ' + 'get_order_items(' + id_item + ')'+ ';')
        elif action == 'count-items':
            cur.execute('SELECT * FROM ' + 'count_item_everywhere(' + id_item + ')'+ ';')
        elif action == 'sold-by-manager':
            cur.execute('SELECT * FROM ' + 'items_sold_by_manager ()')
        elif action == 'sold-items':
            cur.execute('SELECT * FROM ' + 'items_sold ()')
        else:
            cur.execute('SELECT * FROM ' + table + ' WHERE id=' + id_item + ';')
        query_results = cur.fetchall()
        text = '\n\n'.join([' | '.join(map(str, x)) for x in query_results])
        if len(text) < 3:
            return 'По данному запросу ничего не нашлось'
        return text
    except Exception as e:
        print(e)
        return 'Произошла какая-то ошибка: ' + str(e)
        conn.close()
        conn = psycopg2.connect(
            host='localhost',
            database='ESM',
            user=name,
            password=password
        )
        conn.autocommit = True
        cur = conn.cursor()

def make_sql(text):
    global conn
    global cur
    try:
        cur.execute(text)
        query_results = cur.fetchall()
        text = '\n\n'.join([' | '.join(map(str, x)) for x in query_results])
        if len(text) < 3:
            return 'Выполнил запрос.'
        return text
    except Exception as e:
        print(e)
        return 'Произошла какая-то ошибка: ' + str(e)
        conn.close()
        conn = psycopg2.connect(
            host='localhost',
            database='ESM',
            user=name,
            password=password
        )
        conn.autocommit = True
        cur = conn.cursor()

def insert_data():
    global fields
    global conn
    global cur
    try:
        cur.execute('INSERT INTO ' + table + ' (' + fields[table] + ') VALUES (' + data + ');')
        return 'Успешно вставили данные'
    except Exception as e:
        print(e)
        return 'Произошла какая-то ошибка: ' + str(e)
        conn.close()
        conn = psycopg2.connect(
            host='localhost',
            database='ESM',
            user=name,
            password=password
        )
        conn.autocommit = True
        cur = conn.cursor()
```

## Сценарии использования.

Внесение актуальной информации по новым заказам, обновление информации по товарам на складе, добавление новых товаров, анализ продуктивности сотрудников, выполнение SQL-запросов.

## Организация доступа к данным.

Каждый менеджер получит свой ник и пароль для работы с чат-ботом, каждый с выставленными привилегиями.

## Интерфейс с пользователем.

Клиентское приложение разработано в виде чат-бота в telegram. Все взаимодействия происходят либо через выбор одного из действий, либо через сообщение боту.



При первом использовании бот попросит ввести логин и пароль для подключения к БД. Далее предложит выбрать одно действие из функционала - составить SQL запрос, получить отчет или выполнить простое взаимодействие с БД. К числу простых взаимодействий относятся: получение всех данных или одной записи по id из любой БД, вставка новых записей, обновление текущей записи в таблице по id.

ESM\_project\_bot bot

Запросы к базе

Отчеты

Кастомные запросы к базе на SQL

Какой запрос? 12:56:57 AM

Выбрать

Обновить значения

Добавить запись

Из какой таблицы берем данные? 12:57:01 AM

Работники

Покупатели

Товары

Склады

Заказы

Все данные или по id? 12:57:06 AM

Id

All

0 | Astana | Dykenyli 32 | 1000 | 1 12:57:09 AM

1 | Almaty | Bogenaby 12/2 | 100 | 85

2 | Astana | 187, 4 | 5000 | 5000

3 | Nur-Sultan | Bogenbay 32 | 5000 | 1000

Write a message...

📎 😊

EB

ESM\_project\_bot bot

Обновить значения

Добавить запись

В какой таблице обновим значения? 12:57:58 AM

Работники

Покупатели

Товары

Склады

Заказы

Введите через запятую название полей, которые хотите изменить, доступные поля: Город, улица 12:57:59 AM

Артем 12:58:05 AM

город, улица

ESM\_project\_bot 12:58:06 AM

Введите значения, которые хотите вставить в эти поля через запятую

Артем 12:58:22 AM

Nur-sultan, 187 4

ESM\_project\_bot 12:58:23 AM

Введите id объекта, у которого вы хотите обновить значения

Артем 12:58:28 AM

2

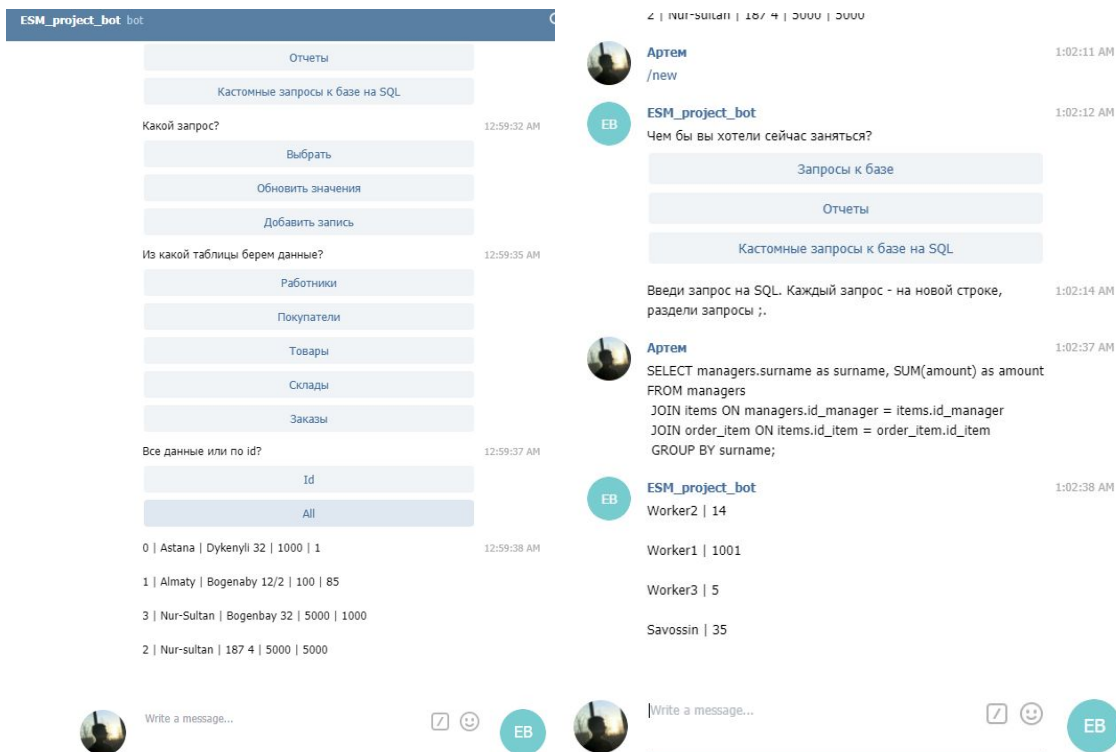
ESM\_project\_bot 12:58:28 AM

Обновление прошло успешно

Write a message...

📎 😊

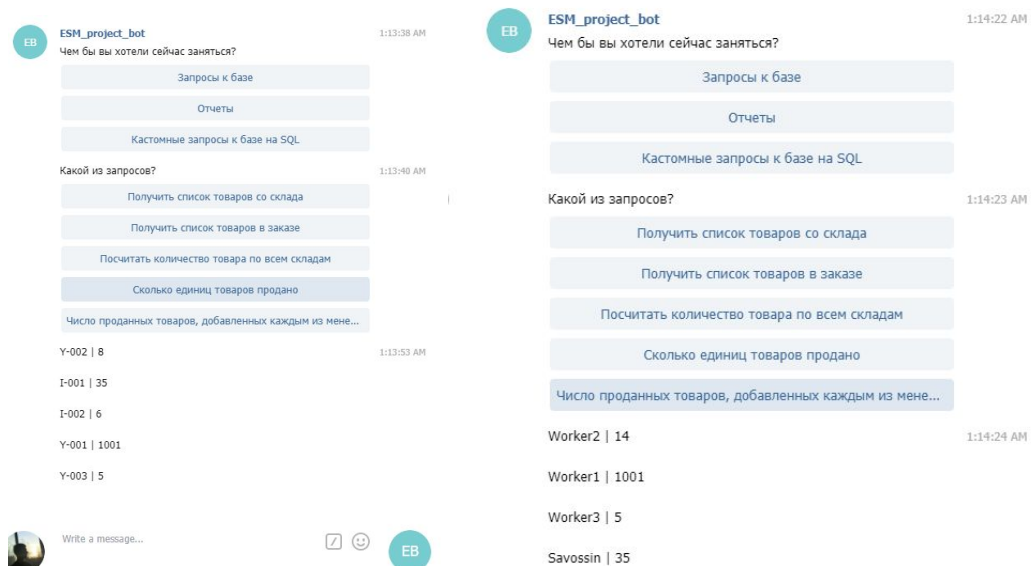
EB

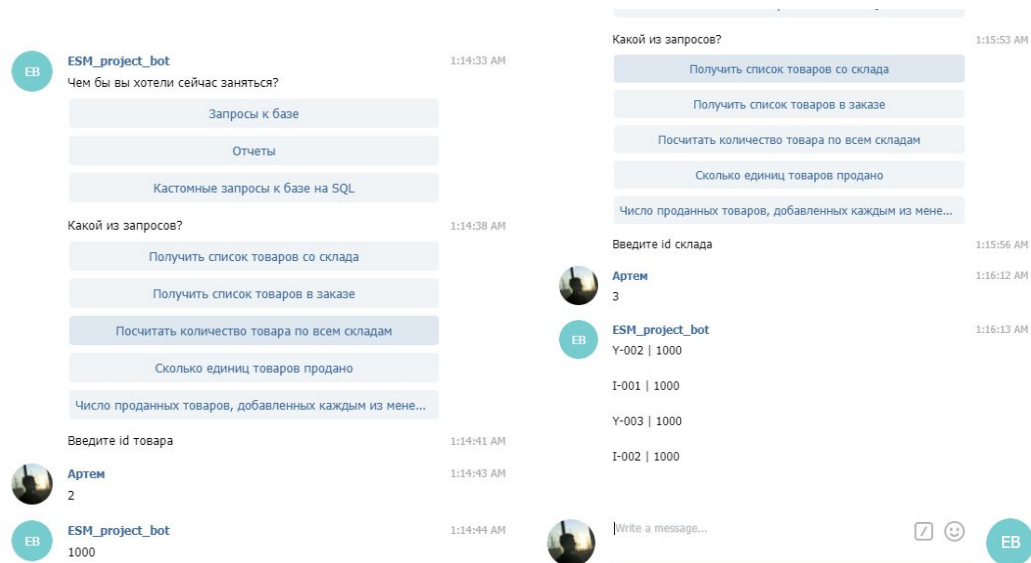


Первые три скриншота описывают работу упрощенного функционала для пользователя - через взаимодействия с ботом на простом языке можно обновить данные, вставить новые, а также получить данные из БД. К сожалению, список упрощенного взаимодействия ограничен, более комплексные запросы можно отправить на SQL через выделенную для этого ветку в чате.

## Отчеты.

Отчеты по БД реализованы таким же образом - в чате выбираем “получить отчет”, уточняем, какой нас интересует, получаем ответ.





Каждый из отчетов выполнен с помощью обращения к одной из хранимых процедур. `SELECT * FROM stored_procedure(args)`, о которых я писал выше.

## Результаты функционального тестирования.

Бот успешно выполняет все функции, описанные выше.

## 6. Заключение.

Реализуя данный проект коснулся всех основных аспектов изучаемой дисциплины. С помощью СУБД PostgreSQL развернул собственный сервер, создал свою базу данных, а также наладил к ней доступ пользователей с помощью чат-бота, реализованного на python.

## 7. Источники

- Microsoft. SQL Server technical documentation [Электронный ресурс] URL: <https://docs.microsoft.com/en-us/sql/sql-server/> (дата обращения: 10.10.2020).
- Date C.J. Database Design and Relational Theory. Normal Forms and All That Jazz. O'Reilly Media, 2012. 260 p.
- <https://github.com/eternnoir/pyTelegramBotAPI>
- <https://www.postgresqltutorial.com/>
- Документация к PostgreSQL 13.1