МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ  
Учреждение образования «БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ   
ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

Факультет информационных технологий

Кафедра информационных систем и технологий

Специальность 1-40 05 01 «Информационные системы и технологии»

**ПОЯСНИТЕЛЬНАЯ ЗАПИСКА КУРСОВОГО ПРОЕКТА**

по дисциплине «Базы данных»

Тема: «База данных интернет-магазина велосипедов с применением Full Text Search-технологий СУБД»

**Исполнитель**

студент 2 курса 2 группы \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ В.О. Лемешевский

подпись, дата

**Руководитель**

\_\_\_\_\_\_\_\_\_\_\_\_ Н.И. Уласевич

(должность, уч. звание) (подпись, дата)

Допущен(а) к защите \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_

дата, подпись

Курсовой проект защищен с оценкой

Руководитель \_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_ Н.И. Уласевич

подпись дата инициалы и фамилия

Минск 2024

Содержание

[Введение 4](#_Toc166509083)

[1 Аналитический обзор литературы по теме проекта 5](#_Toc166509084)

[1.1 Постановка задачи 5](#_Toc166509085)

[1.1.1 Аналог iBike 5](#_Toc166509086)

[1.1.2 Аналог Source BMX 6](#_Toc166509087)

[1.2 Функциональные требования 7](#_Toc166509088)

[1.3. Определение вариантов использования 8](#_Toc166509089)

[1.4 Вывод 10](#_Toc166509090)

[2 Анализ и проектирование архитектуры проекта. 11](#_Toc166509091)

[2.1 UML схема базы данных 11](#_Toc166509092)

[2.2 Описание информационных объектов и ограничений целостности 12](#_Toc166509093)

[2.3 Вывод 13](#_Toc166509094)

[3. Проектирование и разработка базы данных 14](#_Toc166509095)

[3.1 Табличные пространства базы данных 14](#_Toc166509096)

[3.2 Создание основных ролей и пользователей базы данных 15](#_Toc166509097)

[3.3 Вывод 17](#_Toc166509098)

[4. Разработка необходимых объектов 18](#_Toc166509099)

[4.1 Создание таблиц 18](#_Toc166509100)

[4.2 Создание представлений 19](#_Toc166509101)

[4.3 Создание процедур 20](#_Toc166509102)

[4.3.1 Заполнение таблицы 100 000 строк 21](#_Toc166509103)

[4.4 Создание функций 22](#_Toc166509104)

[4.4.1 Выборка данных из таблиц 23](#_Toc166509105)

[4.4.2 Выборка данных по поисковому запросу 24](#_Toc166509106)

[4.5 Создание пакетов процедур и функций для базы данных 26](#_Toc166509107)

[4.6 Создание триггеров 27](#_Toc166509108)

[4.7 Создание индексов 29](#_Toc166509109)

[4.8 Используемая технология 29](#_Toc166509110)

[4.7 Создание директорий 31](#_Toc166509111)

[4.8 Вывод 32](#_Toc166509112)

[5. Описание процедур импорта и экспорта 33](#_Toc166509113)

[5.1 Экспорт данных в JSON 33](#_Toc166509114)

[5.2 Импорт данных из JSON 33](#_Toc166509115)

[6. Тестирование производительности 36](#_Toc166509116)

[6.1 Тестирование производительности базы данных 36](#_Toc166509117)

[Заключение 37](#_Toc166509118)

[Список используемых источников 38](#_Toc166509119)

[Приложение А Листинг создания таблиц 39](#_Toc166509120)

[Приложение Б Листинг создания процедур 40](#_Toc166509121)

[Приложение В Листинг создания функций 51](#_Toc166509122)

[Приложение Г Листинг процедуры для экспорта информации 62](#_Toc166509123)

[Приложение Д Создание триггеров 67](#_Toc166509124)

# Введение

Цель данной работы заключается в создании реляционной базы данных для интернет-магазина , которая обеспечивает пользователю доступ к имеющимся в нем товарам. В рамках работы также необходимо применить соответствующую технологию, Full Text Search, которая будет демонстрировать определенную функциональность базы данных, включая полнортекствый поиск товаров по индексу, предложению и т.д.

База данных - это совокупность взаимосвязанных данных, которые обычно хранятся в электронном виде в компьютерной системе. БД используются для хранения, организации и управления большим объемом структурированных и неструктурированных данных. Реляционная база данных является наиболее распространенной формой организации данных, в которой данные представлены в виде таблиц, состоящих из строк и столбцов, где каждый столбец представляет атрибут, а каждая строка представляет кортеж или запись. В данной работе для управления базой данных была выбрана СУБД «Oracle», поскольку эта система обладает высокой надежностью и производительностью, что позволяет обеспечить эффективное хранение, обработку и управление данными.

Для обеспечения функциональности приложения используются технология полнотекстового поиска, которая позволяет приложениям и пользователям выполнять полнотекстовые запросы к символьным данным в таблицах.

В пояснительной записке содержится информация о сопоставимых продуктах, структуре и реализации проекта, а также инструкции по использованию приложения.

1. Аналитический обзор литературы по теме проекта

## Постановка задачи

Интернет-магазины стали неотъемлемой частью нашей современной жизни. Они предоставляют нам доступ к бесчисленному количеству товаров и услуг прямо из домашних условий. Поэтому, для реализации базы данных интернет-магазина необходимо рассмотреть и объективно оценить возможный функционал базы данных, применимые технологии и т.п. По результатам анализа необходимо составить определенный список функциональных требований к базе данных и диаграмму вариантов использования. Задача проекта: разработать архитектуру приложения, взаимодействие с которой будет понятно любому пользователю. Построить базу данных и выполнить тестирование готового продукта.

Основные требования к реализации:

* Определение ролей (Покупатель, продавец, менеджер);
* Возможность добавления и удаления товара (Продавец, менеджер);
* Возможность оформление заказа товара (Покупатель);
* Поиск товара по категориям, критериям, названию (Покупатель);
* Возможность оценки товара (Покупатель);
* Возможность добавление товара в корзину/избранное (Покупатель);
* Взаимодействие с базой данных при помощи хранимых процедур и функций.

### 1.1.1 Аналог iBike

iBike [1] - это крупнейший экстрим-магазин в Беларуси. Они специализируются на продаже велосипедов BMX и MTB, запчастей, защиты и аксессуаров. Их магазин является официальным представителем многих брендов БМХ, таких как WTP, Eclat, Odyssey, Fuse, BSD, Cult и других. Пример интерфейса данного сервиса представлен на рисунке 1.1.

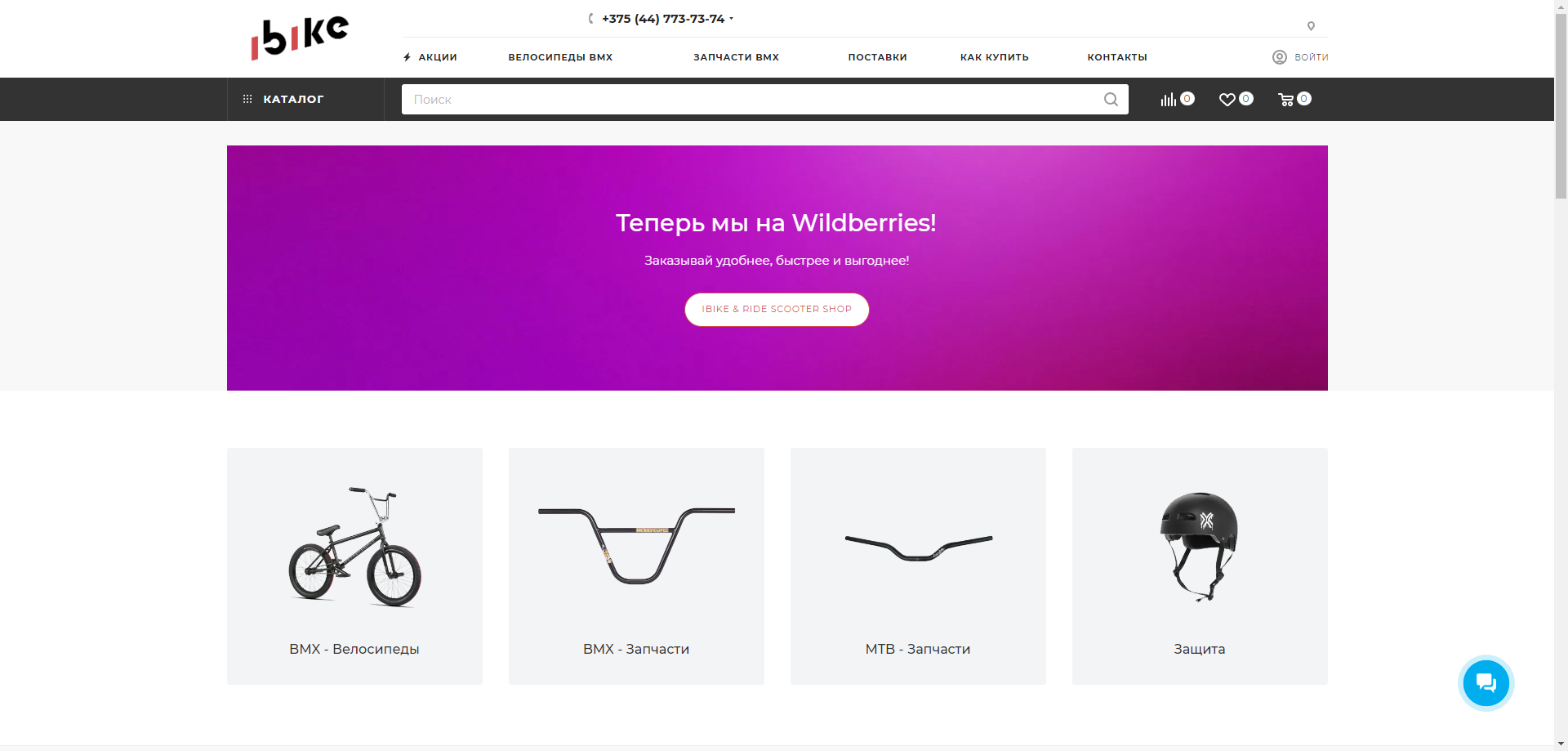


Рисунок 1.1 – Интерфейс сервиса iBike

Одной из основных функций iBike.by является продажа экстрим-велосипедов. Пользователи могут искать велосипеды по бренду, модели, типу или цене. Кроме того, iBike.by предлагает персонализированные рекомендации в соответствии с предпочтениями пользователя, а также список товаров, рекомендованных другими пользователями и кураторами платформы. Пользователи могут создавать свои собственные списки желаемых товаров и делиться ими с друзьями или сохранять товары для последующего просмотра. iBike.by также позволяет пользователям просматривать информацию о товарах, включая оригинальный контент, созданный самой платформой. Другая функция iBike.by - это возможность использовать платформу как социальную сеть, где пользователи могут подписываться на друг друга, просматривать их списки желаемых товаров и рекомендации, а также обмениваться сообщениями. Эти системы также используются для хранения информации о пользовательских аккаунтах и настройках. В общем, iBike.by использует масштабируемые и высокопроизводительные базы данных для обеспечения быстрого доступа к большому объему информации о товарах и обработки данных для предоставления персонализированных рекомендаций и других функций платформы.

### 1.1.2 Аналог Source BMX

Source BMX [2] - это популярная платформа для продажи BMX велосипедов, которая позволяет пользователям просматривать и покупать товары онлайн. Source BMX позволяет пользователям просматривать товары на различных устройствах, а также создавать собственные списки желаемых товаров. Source BMX предлагает пользователям доступ к широкому ассортименту BMX велосипедов, запчастей и аксессуаров. Сервис также предлагает персонализированные рекомендации, основанные на предпочтениях пользователя.

Интерфейс Source BMX схож с интерфейсом iBike и предлагает удобный поиск по брендам, моделям и типам товаров. Также сервис позволяет сохранять товары для последующего просмотра. Пример интерфейса данного сервиса представлен на рисунке 1.2

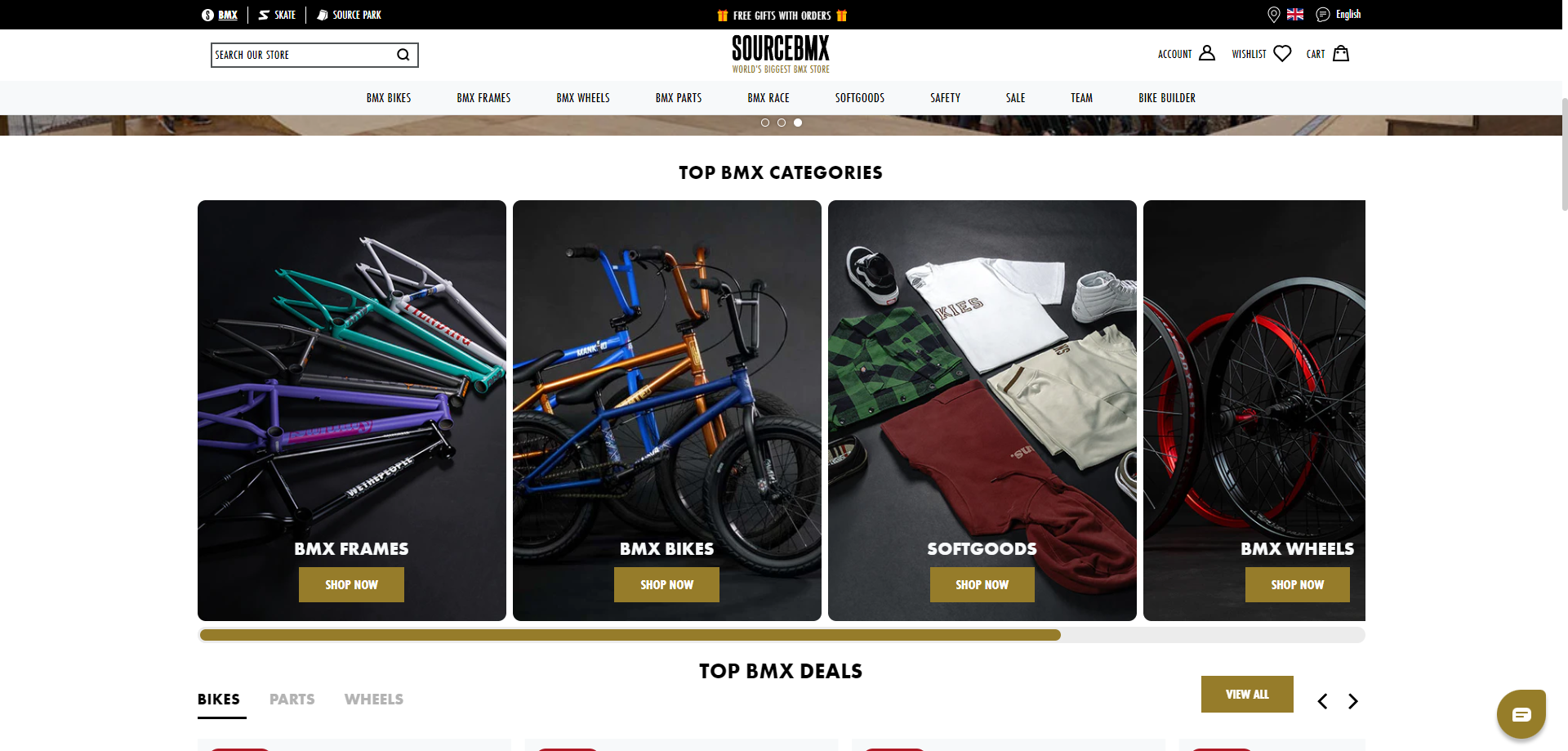


Рисунок 1.2 – Интерфейс сервиса Source BMX

Source BMX может использовать несколько баз данных в своей архитектуре. Возможно, они используют реляционную базу данных, такую как MySQL, для хранения информации о пользователях, такой как логины и пароли, а также информации о покупках и оплате. Для хранения метаданных товаров, включая информацию о брендах, моделях, типах и т.д., а также для предоставления быстрого доступа к этой информации пользователям, они могут использовать NoSQL базу данных, такую как Cassandra. Кроме того, Source BMX может использовать облачную инфраструктуру, например, Amazon Web Services или Google Cloud, для обеспечения высокой доступности и масштабируемости своей платформы.

В целом, Source BMX, вероятно, также использует масштабируемые и высокопроизводительные базы данных для обеспечения быстрого доступа к большому объему информации о товарах и управления информацией о пользователях и их покупках. Однако, конкретные детали об архитектуре и используемых технологиях могут быть доступны только у команды Source BMX

## 1.2 Функциональные требования

Функциональные требования базы данных определяют, как база данных должна обрабатывать данные и предоставлять пользователю определенной роли необходимую функциональность. Это может включать в себя описание того, как данные должны храниться и организовываться, как происходит поиск и выборка данных, каким образом обновляются данные и какие механизмы используются для защиты данных. Кроме того, функциональные требования могут определять интеграцию базы данных с другими системами и программами. Например, для интернет-магазина велосипедов функциональные требования могут включать в себя функции для хранения информации о описании товаров, изображений, поиска товаров по категориям и критериям, создание и удаление карточек товаров, добавление и удаление товаров из корзины, а также функции для оценки товаров.

По результатам рассмотренных аналогов основные задачи для базы данных интернет-магазина:

* разработать процедуры и функции для обработки информации и действий пользователя;
* разработать функционал добавления и удаления товаров из корзины;
* разработать структурную модель базы данных;
* разработать функционал добавления информации о товаре конкретным пользователем;
* разработать функционал оценки товара;
* разработать функционал для заказа товара;
* разработать функционал редактирования/удаления пользователя/товара.
* разработать функционал полнотекстового поиска товаров.
* оформить функционал сохранения определенной мультимедийной информации о товаре (фото).

## 1.3. Определение вариантов использования

Помимо функциональных требований, важно также определить роли пользователей и их варианты использования системы. Варианты использования описывают, как пользователи будут взаимодействовать с системой в зависимости от своих ролей. Варианты использования обычно представляются в виде UML диаграмм, которые позволяют наглядно отобразить взаимодействие между пользователями и системой.

В зависимости от роли пользователя, он может иметь доступ к различным функциям системы. В данном проекте роли пользователей будут следующими:

* Гость
* Покупатель
* Продавец
* Менеджер

На основе предоставленного списка ролей необходимо построить варианты использования. Варианты использования изображены на рисунке 1.3.

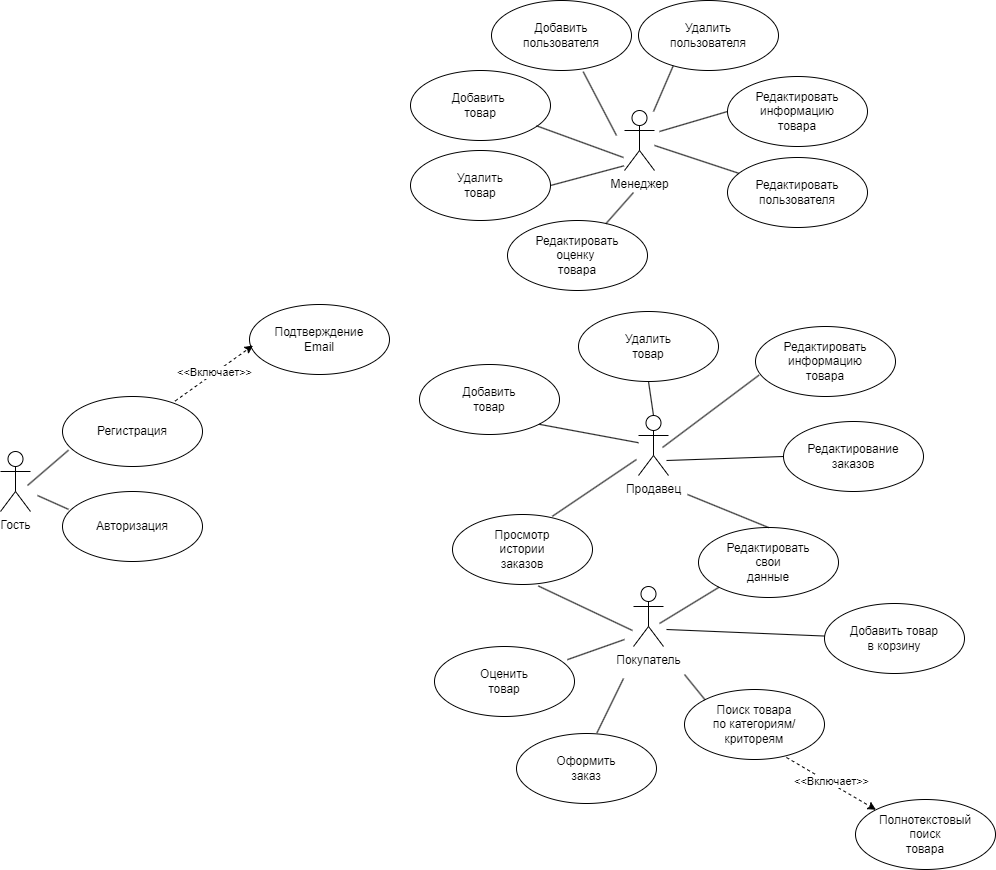


Рисунок 1.3 – UML диаграмма вариантов использования

В начале работы с приложением пользователь является гостем. Ему будет доступна только просмотр всей информации без возможности добавления товара в корзину. После регистрации пользователь становится зарегистрированным пользователем который может быть как покупать (Покупатель), так и продавать (Продавец) товар.

Роль Покупатель получает возможность добавлять понравившиеся товары в корзину, оценивать товары ставя им оценку и оставлять свой комментарий о товаре, осуществлять поиск товара по категориям или критериям, а также пользоваться технологией полнотекстового поиска.

Роль Продавец получает возможность добавлять товары на продажу, изменять параметры продаваемого товара, добавляя к нему описание или медиа контент, он так же может и снимать собственные товары с продажи, просматривать историю заказов, узнав информацию о тех кто заказал товар, Продавец так же может управлять текущими заказами закрывая их или редактируя. Так же помимо основных возможностей пользователю с ролью Продавец ничего не мешает использовать возможности Покупателя.

Оба пользователя с ролью Покупателя и Продавца могут изменять собственные данные: имя, адрес, номер телефона и т.д.

Роль Менеджер заключается в управлении площадкой и мониторинге действий пользователей. Менеджер в качестве администратора имеет доступ к просмотру информации о всех пользователях, в том числе их действиях на площадке. При необходимости Менеджер может редактировать информацию о товарах и пользователях, например, изменять категории и форматировать отзывы у товаров, а также изменять личную информацию о пользователях.

## 1.4 Вывод

Итого, был проведен аналитический обзор аналогов интернет-магазинов, которые уже существуют на рынке. Этот обзор позволил определить основные характеристики и функциональные возможности, которые необходимо предусмотреть в разрабатываемой базе данных. Также были определены функциональные требования базы данных, а также роли пользователей и варианты использования в зависимости от этих ролей. Была разработана UML-диаграмма, на которой отображены основные функции, которые доступны для каждой из ролей пользователей.

1. Анализ и проектирование архитектуры проекта.

## 2.1 UML схема базы данных

Схема базы данных представляет собой логическую конфигурацию либо целой реляционной базы данных, либо ее части. Схема может существовать как в виде наглядного представления базы данных, так и в виде набора формул (также именуемых «условиями целостности»), которые регулируют ее устройство. Эти формулы выражаются с помощью языка описания данных, например, SQL. Будучи частью словаря данных, схема показывает, как связаны между собой сущности, из которых состоит база данных (таблицы, представления, хранимые процедуры и так далее). Схема базы данных будет представлена на рисунке 2.1.

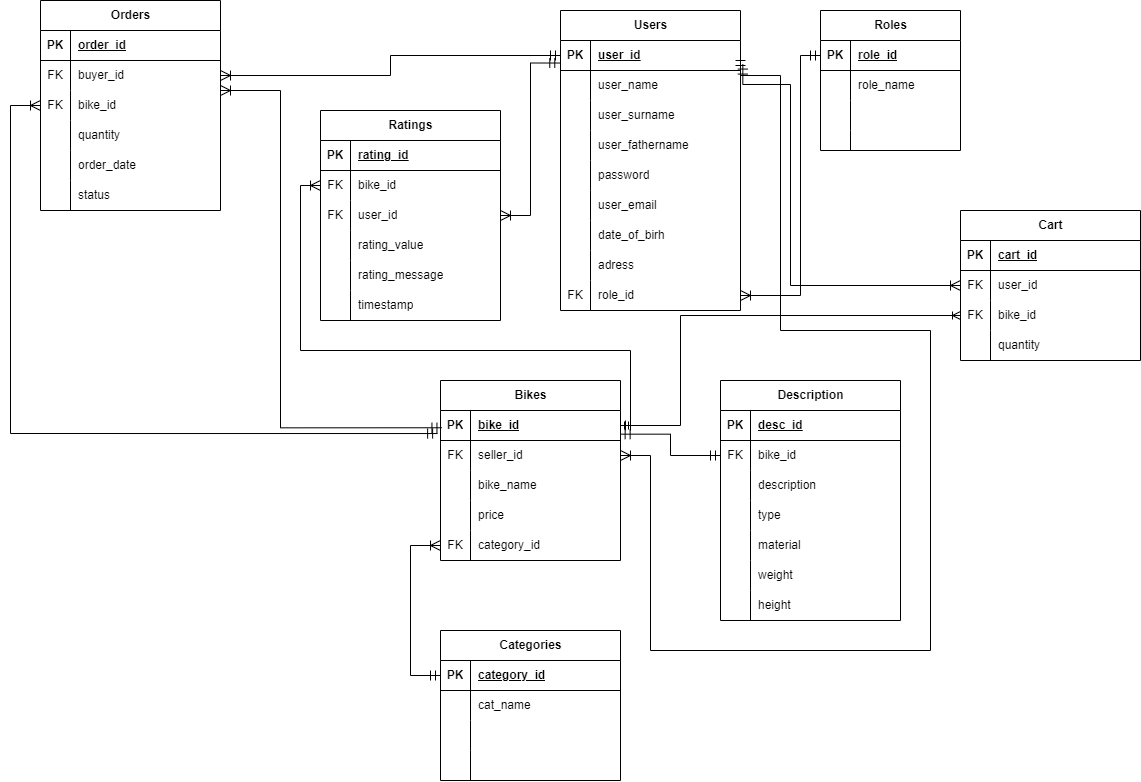


Рисунок 2.1 –Схема базы данных

Таким образом, схема показывает связи между таблицами и полями, а также типы отношений между ними, такие как связи "один-ко-многим" и "многие-ко-многим". Например, таблица Users связана с таблицами Bikes и Orders через внешние ключи seller\_id и buyer\_id. Также видно, что таблица Bikes связана с таблицей Categories и Descriptions через внешний ключ bike\_id. Кроме того, таблица Orders связана с таблицей Bikes по bike\_id, а таблица Ratings связана с товаром через внешний ключ bike\_id и с таблицей Users через внешний ключ user\_id.

## 2.2 Описание информационных объектов и ограничений целостности

Для реализации базы данных было разработано 8 таблиц. В структуру схемы базы данных для проекта входят следующие таблицы: Roles, Users, Ratings, Bikes, Cart, Categories, Description, Orders. Ниже будет описание про каждую из них более подробно. Описание представлено в таблицу 2.1.

Таблица 2.1 − Описание ограничений целостности

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Roles | role\_id | PK | Идентификатор роли | NUMBER |
| role\_name |  | Текстовое название роли | VARCHAR(255) |
| Users | user\_id | PK | Идентификатор пользователя | NUMBER |
| user\_name |  | Имя пользователя | VARCHAR(255) |
| user\_surname |  | Фамилия пользователя |
| user\_fathername |  | Отчество пользователя |
| user\_email |  | Почта пользователя |
| password |  | Пароль |
| date\_of\_birth |  | Дата рождения |
| address |  | Адрес проживания |
| role\_id | FK | Идентификатор роли | NUMBER |
| Bikes | bike\_id | PK | Идентификатор товара | NUMBER |
| seller\_id | FK | Идентификатор продавца |
| bike\_name |  | Название товара | VARCHAR(255) |
| price |  | Цена товара | DECIMAL(10,2) |
| category\_id | FK | Категория товара | NUMBER |
| Description | desc\_id | PK | Идентификатор описания | NUMBER |
| bike\_id | FK | Идентификатор товара |
| description |  | Словесное описание товара | VARCHAR(255) |
| type |  | Тип товара | VARCHAR(255) |
| material |  | Материл изготовления | CHAR(50) |
| weight |  | Масса товара | FLOAT(5) |
| height |  | Высота | FLOAT(5) |
| Categories | сategory\_id | PK | Идентификатор категории товара | NUMBER |
| сat\_name |  | Название категории товара | VARCHAR(255) |
| Orders | order\_id | PK | Идентификатор заказа | NUMBER |
| buyer\_id | FK | Идентификатор покупателя |
| bike\_id | FK | Идентификатор товара |
| quantity |  | Количество | FLOAT(5) |
| order\_date |  | Дата заказа | DATETIME |
| status |  | Статус заказа | VARCHAR(255) |
| Ratings | rating\_id | PK | Идентификатор оценки товара | NUMBER |
| bike\_id | FK | Идентификатор товара |
| user\_id | FK | Идентификатор пользователя |
| rating\_value |  | Значение оценки |
| rating\_message |  | Комментарий к оценке | VARCHAR(255) |
| timestamp |  | Дата оценки | DATETIME |
| Cart | cart\_id | PK | Идентификатор корзины | NUMBER |
| user\_id | FK | Идентификатор пользователя |
| bike\_id | FK | Идентификатор товара |
| quantity |  | Количество |

## 2.3 Вывод

Анализ и проектирование модели базы данных представляют собой важный этап разработки системы. Была разработана UML-схема базы данных, которая является логической конфигурацией реляционной базы данных. Схема показывает связи между таблицами и полями, а также типы отношений между ними. Для реализации базы данных было разработано восемь таблиц: Roles, Users, Ratings, Bikes, Cart, Categories, Description, Orders. Каждая из них имеет свою специфику и связана с другими таблицами через внешние ключи.

# 3. Проектирование и разработка базы данных

## 3.1 Табличные пространства базы данных

Для магазина велосипедов необходимо создать несколько табличных пространств, которые будут хранить таблицы о заказах, пользователях, корзинах, велосипедах и т.д. Будут созданы следующие табличные пространства:

Пространство TS\_ORDER (Заказы), содержащая информацию о заказах, такую как идентификатор заказа, дата заказа, статус заказа и т.д. Пример скрипта создания табличного пространства будет на листинге 3.1. Остальные табличные пространства будут аналогичны.

|  |
| --- |
| -- USERS  CREATE TABLESPACE TS\_USER  DATAFILE 'C:\Tablecpaces\CourseWork\TS\_USER.dbf'  size 50M  autoextend on next 5M  extent management local;  -- ORDERS  CREATE TABLESPACE TS\_ORDER  DATAFILE 'C:\Tablecpaces\CourseWork\TS\_ORDER.dbf'  size 50M  autoextend on next 5M  extent management local;  -- CART  CREATE TABLESPACE TS\_CART  DATAFILE 'C:\Tablecpaces\CourseWork\TS\_CART.dbf'  size 50M  autoextend on next 5M  extent management local;  CREATE TABLESPACE TS\_BIKE  DATAFILE 'C:\Tablecpaces\CourseWork\TS\_BIKE.dbf'  SIZE 50M  AUTOEXTEND ON NEXT 5M  EXTENT MANAGEMENT LOCAL; |

Листинг 3.1 – Табличные пространства

Пространство TS\_CART (Корзина), где хранятся данные о товарах, добавленных в корзину пользователем перед оформлением заказа.

Пространство TS\_BIKE (Велосипеды), содержащая информацию о доступных велосипедах в магазине, включая модель, цену, размер и т.д.

Пространство TS\_USER (Пользователи), содержащая информацию о пользователях базы данных, их ролях, оценках и т.п.

Для каждого табличного пространства были созданы новые файлы для хранения информации, а разработчику была выделена квота на добавление данных в табличные пространства пример на листинге 3.2.

|  |
| --- |
| ALTER USER DEVELOPER QUOTA UNLIMITED ON TS\_USER;  ALTER USER DEVELOPER QUOTA UNLIMITED ON TS\_BIKE;  ALTER USER DEVELOPER QUOTA UNLIMITED ON TS\_ORDER;  ALTER USER DEVELOPER QUOTA UNLIMITED ON TS\_CART; |

Листинг 3.2 – Выделение квоты на пространства

Квота необходима пользователю для внесения информации в таблицы лежащих в табличных пространствах.

## 3.2 Создание основных ролей и пользователей базы данных

Для проекта были выделены две основные роли, которые участвуют в реализации базы данных для магазина велосипедов с привилегиями для ограничения доступа к базе данных.

* DB\_ADMIN – для администратора базы данных.
* PROGRAMMER – для разработчика обьктов базы данных.

Для которых определены различные пользователи, такие как COURSE\_DB\_ADMIN и DEVELOPER, каждый из которых имеет свои дополнительные роли и привилегии.

Например, COURSE\_DB\_ADMIN получает полные права администратора базы данных, ALL PRIVILEGES , в то время как DEVELOPER получает роль PROGRAMMER и соответствующие привилегии для работы с данными. Пример администратора базы данных показана в листинге 3.3.

|  |
| --- |
| grant  ALL PRIVILEGES  to  DB\_ADMIN;  CREATE USER COURSE\_DB\_ADMIN IDENTIFIED BY 123;  GRANT DB\_ADMIN ,SYSDBA TO COURSE\_DB\_ADMIN; |

Листинг 3.3 – Пользователь СOURSE\_DB\_ADMIN

Кроме того, определены роли, такие как SESSION\_PRIV которая предоставляет определенный уровень доступа к базе данных.

Например, PROGRAMMER имеет привилегии для создания таблиц, представлений, процедур и индексов.

Роль DB\_ADMIN имеет привилегии для администрирования базы данных, такие как, ALL PRIVILEGIES. Это необходимо для того, чтобы в процессе создания базы данных не участвовал ни один встроенный в СУБД администратор напрямую. Пример создания роли вместе с пользователем и выданными привилегиями будет на листинге 3.4.

|  |
| --- |
| create role PROGRAMMER;  grant  CREATE TABLE,  CREATE VIEW,  CREATE PROCEDURE,  CREATE ANY INDEX,  CREATE USER,  DROP USER,  CREATE SEQUENCE,  CREATE TRIGGER,  CREATE ROLE,  CREATE TYPE,  CREATE ANY DIRECTORY  TO  PROGRAMMER  ;  CREATE ROLE SESSION\_PRIV;  GRANT  CREATE SESSION  TO SESSION\_PRIV;  CREATE USER DEVELOPER IDENTIFIED BY 12345;  GRANT PROGRAMMER TO DEVELOPER;  GRANT SESSION\_PRIV TO DEVELOPER WITH ADMIN OPTION;  GRANT CTXAPP TO DEVELOPER;  GRANT EXECUTE ON CTXSYS.DRUE TO DEVELOPER;  GRANT EXECUTE ON CTXSYS.TEXTINDEXMETHODS TO DEVELOPER;  GRANT EXECUTE ON CTXSYS.CTX\_CLS TO DEVELOPER;  GRANT EXECUTE ON CTXSYS.CTX\_DDL TO DEVELOPER;  GRANT EXECUTE ON CTXSYS.CTX\_DOC TO DEVELOPER;  GRANT EXECUTE ON CTXSYS.CTX\_OUTPUT TO DEVELOPER;  GRANT EXECUTE ON CTXSYS.CTX\_QUERY TO DEVELOPER;  GRANT EXECUTE ON CTXSYS.CTX\_REPORT TO DEVELOPER;  GRANT EXECUTE ON CTXSYS.CTX\_THES TO DEVELOPER;  GRANT EXECUTE ON CTXSYS.CTX\_ULEXER TO DEVELOPER; |

Листинг 3.4 – Пользователь DEVELOPER

Пользователем DEVELOPER были так же созданы пользователи USER и MANAGER для представления им доступа к соответствующему функционалу и дальнейшего наглядного примера сценария. Пользователи играют ключевую роль в выполнении запросов и вызова процедур. Пример создания пользователей и выдача привилегий будет на листинге 3.5.

|  |
| --- |
| CREATE USER USER\_1 IDENTIFIED BY 123;  CREATE USER MANAGER IDENTIFIED BY 123;  GRANT SESSION\_PRIV TO USER\_1;  GRANT SESSION\_PRIV TO USER\_1;  GRANT EXECUTE ON REGISTER TO USER\_1;  GRANT EXECUTE ON LOGIN TO USER\_1;  GRANT EXECUTE ON ADD\_BIKE TO USER\_1;  GRANT EXECUTE ON ADD\_DESC TO USER\_1;  GRANT EXECUTE ON ADD\_ORDER TO USER\_1;  GRANT EXECUTE ON UPDATE\_USER TO USER\_1;  GRANT EXECUTE ON ADD\_RATE TO USER\_1;  GRANT EXECUTE ON ADD\_TO\_CART TO USER\_1;  GRANT EXECUTE ON DEL\_FROM\_CART TO USER\_1;  GRANT EXECUTE ON UPDATE\_PASS TO USER\_1;  GRANT EXECUTE ON UPDATE\_BIKE TO USER\_1;  GRANT EXECUTE ON DEL\_BIKE TO USER\_1;  GRANT EXECUTE ON CLEAR\_CART TO USER\_1;  GRANT EXECUTE ON CLEAR\_ORDER TO USER\_1;  GRANT EXECUTE ON UPDATE\_ORDER TO USER\_1;  GRANT EXECUTE ON DEL\_ORDER TO USER\_1;  GRANT EXECUTE ON DEL\_DESC TO USER\_1;  GRANT EXECUTE ON UPD\_DESC TO USER\_1;  GRANT EXECUTE ON DEL\_RATE TO USER\_1;  GRANT EXECUTE ON UPD\_RATE TO USER\_1;  GRANT EXECUTE ON DEL\_FROM\_CART TO USER\_1;  GRANT EXECUTE ON AVG\_RATE TO USER\_1;  …  GRANT EXECUTE ON GETBIKEBYID TO MANAGER;  GRANT EXECUTE ON GETUSERBYID TO MANAGER;  GRANT EXECUTE ON GETORDERBYID TO MANAGER;  GRANT EXECUTE ON GETUSERS TO MANAGER;  GRANT EXECUTE ON GETBIKES TO MANAGER;  GRANT EXECUTE ON GETORDERS TO MANAGER;  … |

Листинг 3.5 – Создание пользователей разработчиком

В целом, была создана безопасная и эффективная база данных для магазина велосипедов, где различные пользователи имеют различные уровни доступа и привилегии в соответствии с их ролями и функциями.

## 3.3 Вывод

Проект базы данных для магазина велосипедов включает создание нескольких табличных пространств, предназначенных для хранения информации о заказах, пользователях, корзинах, велосипедах и прочем. Каждое пространство имеет свою функциональную область.

Каждое табличное пространство имеет свой собственный файл для хранения информации, и для разработчика была выделена квота на добавление данных в эти пространства.

В рамках создания базы данных также определены роли и пользователи. Три основные роли (COURSE\_DB\_ADMIN, DEVELOPER) были выделены с соответствующими привилегиями для управления базой данных и ее разработки. Каждый пользователь получил определенные роли и привилегии, обеспечивающие соответствующий уровень доступа и функциональность.

4. Разработка необходимых объектов

## 4.1 Создание таблиц

Для магазина велосипедов необходимо создать несколько таблиц, которые будут хранить данные о пользователях, велосипедах, ролях, заказах, корзинах, рейтингах и категориях.

Для магазина велосипедов были созданы следующие таблицы:

ROLES - таблица, содержащая роли пользователей. Здесь хранится информация о ролях, которые могут быть присвоены пользователям. Каждая роль имеет уникальный идентификатор (ROLE\_ID) и название (ROLENAME).

USERS - это таблица, где хранятся данные о пользователях. Здесь содержится информация, такая как идентификатор пользователя (USER\_ID), его имя (USER\_NAME), фамилия (USER\_SURNAME), отчество (USER\_FATHERNAME), электронная почта (USER\_EMAIL), пароль (PASSWORD), дата рождения (DATE\_OF\_BIRTH) и адрес (ADRESS). Также в таблице присутствует связь с таблицей ROLES через поле ROLE\_ID, определяющее роль пользователя.

CATEGORIES - таблица, предназначенная для хранения категорий велосипедов. Здесь содержатся различные категории (например, MTB, BMX, MOUNTAIN, KID, DEFAULT), каждая из которых имеет уникальный идентификатор (CATEGORY\_ID) и название категории (CAT\_NAME).

BIKES - таблица, содержащая информацию о велосипедах. Каждый велосипед имеет уникальный идентификатор (BIKE\_ID), название (BIKE\_NAME), цену (PRICE) и ссылку на соответствующую категорию (CATEGORY\_ID). Также есть связь с таблицей USERS через поле SELLER\_ID, определяющее продавца велосипеда.

DESCRIPTIONS - это таблица, где хранятся описания велосипедов. Здесь содержится информация о каждом велосипеде, такая как его описание (BIKE\_DESC), тип (BIKE\_TYPE), материал (MATERIAL), вес (WEIGHT) и высота (HEIGHT). Также каждое описание имеет уникальный идентификатор (DESC\_ID) и ссылку на соответствующий велосипед (BIKE\_ID).

RATINGS - таблица, в которой хранятся рейтинги велосипедов, оставленные пользователями. Каждый рейтинг содержит уникальный идентификатор (RATING\_ID), ссылку на велосипед (BIKE\_ID), пользователя (USER\_ID), значение рейтинга (RATING\_VALUE), сообщение о рейтинге (RATING\_MESSAGE) и метку времени (TIME\_STAMP).

CART - таблица, содержащая товары, добавленные в корзину пользователем. Здесь хранятся идентификатор корзины (CART\_ID), идентификатор пользователя (USER\_ID), идентификатор велосипеда (BIKE\_ID) и количество (QUANTITY).

ORDERS - таблица, в которой хранится информация о заказах. Здесь содержатся данные о каждом заказе, включая идентификатор заказа (ORDER\_ID), покупателя (BUYER\_ID), велосипеда (BIKE\_ID), количество (QUANTITY), дату заказа (ORDER\_DATE) и статус (STATUS).

Для эффективного использования базы данных в проекте, необходимо создать индексы на столбцах, используемых в запросах с поиском данных. Например, можно создать индексы на столбцах user\_email в таблице Users, чтобы быстро находить информацию о пользователе по его электронной почте. Пример создания таблицы будет на листинге 4.1.

|  |
| --- |
| CREATE TABLE USERS (  USER\_ID INTEGER GENERATED ALWAYS AS IDENTITY(START WITH 1 INCREMENT BY 1) PRIMARY KEY,  USER\_NAME VARCHAR2(255) NOT NULL,  USER\_SURNAME VARCHAR2(255) NOT NULL,  USER\_FATHERNAME VARCHAR2(255),  USER\_EMAIL VARCHAR2(255) NOT NULL,  PASSWORD NVARCHAR2(255) NOT NULL,  DATE\_OF\_BIRTH DATE,  ADRESS VARCHAR2(255) NOT NULL,  ROLE\_ID INTEGER NOT NULL,  CONSTRAINT FK\_USER\_ROLE FOREIGN KEY(ROLE\_ID) REFERENCES ROLES(ROLE\_ID)  ) TABLESPACE TS\_USER; |

Листинг 4.1 – Создание таблицы USERS

Таким же образом были созданы и остальные таблицы для ролей, заказов и т.п. Такие таблицы ориентированы на динамическое изменение и добавление данных в своих табличных пространствах.

## 4.2 Создание представлений

Представление (view) в базе данных представляет собой виртуальную таблицу, которая создается на основе запроса к одной или нескольким таблицам в базе данных. Представления позволяют обращаться к данным из нескольких таблиц одновременно, при этом не изменяя структуру этих таблиц.

В данном проекте были созданы четыре представления:

* USER\_V содержит информацию о всех пользователях
* BIKE\_V содержит полное описание велосипедов включая его характеристики, материал, тип и т.п.
* ORDER\_V содержит полную информацию о заказах пользователя включая того кто заказал, когда заказал, что заказал.
* CART\_V содержит информацию о товарах добавленных пользователем в корзину.

Представление USER\_V было создано для того, чтобы получить полную информацию о пользователях и их ролях, объединив данные из двух таблиц. Оно будет на листинге 4.2. Остальные представления будут аналогичны, только будут работать с другими таблицами.

|  |
| --- |
| CREATE OR REPLACE VIEW USER\_V AS  SELECT  u.USER\_ID,  u.USER\_NAME,  u.PASSWORD,  u.USER\_SURNAME,  u.USER\_FATHERNAME,  u.USER\_EMAIL,  u.DATE\_OF\_BIRTH,  COUNT(o.ORDER\_ID) AS ORDERS\_COUNT  FROM USERS u  LEFT JOIN ORDERS o ON o.BUYER\_ID = u.USER\_ID  GROUP BY  u.USER\_ID,  u.USER\_NAME,  u.PASSWORD,  u.USER\_SURNAME,  u.USER\_FATHERNAME,  u.USER\_EMAIL,  u.DATE\_OF\_BIRTH  ORDER BY u.USER\_ID; |

Листинг 4.2 – Представление USER\_V

Были созданы основные представления по доступу к данным таблиц, с последующей манипуляцией выборки.

## 4.3 Создание процедур

Процедура в PL/SQL — это модуль подпрограммы, состоящий из группы операторов PL/SQL, которые можно вызывать по имени. Каждая процедура в PL/SQL имеет собственное уникальное имя, по которому к ней можно обращаться и вызывать. Этот модуль подпрограммы в Oracle database хранится как объект базы данных. Процедуры, созданные для магазина, служат для эффективного управления базой данных онлайн магазина велосипедов. Они обеспечивают функциональность добавления (ADD\_BIKE\_TO\_CART) и удаления велосипедов (DEL\_FROM\_CART) из корзины покупателей, изменения паролей пользователей (UPDATE\_PASS), обновления информации о велосипедах (UPD\_BIKE) и заказах (UPD\_ORDER), а также удаления пользователей (DEL\_USER), заказов (DEL\_ORDER) и других объектов, сохраняя при этом целостность данных и обеспечивая безопасность операций с базой данных. Пример процедуры по добавлению товара в корзину пользователя представлен в листинге 4.3.

|  |
| --- |
| CREATE OR REPLACE PROCEDURE ADD\_TO\_CART  (  USER INTEGER ,  BIKE INTEGER ,  QNT NUMBER  )  IS  ID CART.CART\_ID%TYPE;  BEGIN  INSERT INTO CART(USER\_ID,BIKE\_ID,QUANTITY)VALUES(USER,BIKE,QNT)  RETURNING CART.CART\_ID INTO ID;  EXCEPTION  WHEN NO\_DATA\_FOUND THEN  raise\_application\_error(-20001,'An error was encountered - '||SQLCODE||' -ERROR- '||SQLERRM);  WHEN TOO\_MANY\_ROWS THEN  raise\_application\_error(-20001,'An error was encountered - '||SQLCODE||' -ERROR- '||SQLERRM);  WHEN OTHERS  THEN raise\_application\_error(-20001,'An error was encountered - '||SQLCODE||' -ERROR- '||SQLERRM);  END; |

Листинг 4.3 – Процедура ADD\_TO\_CART

В зависимости от того, какую задачу необходимо выполнить, следует использовать хранимую процедуру или функцию. Хранимые процедуры могут использоваться для выполнения сложных операций над данными, таких как массовые изменения в таблицах, а также для оптимизации производительности приложения. Функции же наиболее полезны в случаях, когда требуется выполнить вычисление на основе данных в базе данных, например, для подсчета статистики или фильтрации данных.

### 4.3.1 Заполнение таблицы 100 000 строк

Для заполнения таблицы USERS была разработана процедура INSERT\_100K\_LINES, которая вставляет 100000 строк в таблицу. Функция представлена на листинге 4.4.

|  |
| --- |
| CREATE OR REPLACE PROCEDURE INSERT\_100K\_LINES  IS  v\_start\_date DATE := TO\_DATE('1980-01-01', 'YYYY-MM-DD');  v\_end\_date DATE := TO\_DATE('2004-01-01', 'YYYY-MM-DD');  BEGIN  FOR i IN 1..100000 LOOP  INSERT INTO USERS (  USER\_NAME,  USER\_SURNAME,  USER\_FATHERNAME,  USER\_EMAIL,  PASSWORD,  DATE\_OF\_BIRTH,  ADRESS,  ROLE\_ID  ) VALUES (  'User'||i,  'Surname'||i,  'Fathername'||i,  'user'||i||'@example.com',  HASH\_PASS('password'||i),  v\_start\_date + DBMS\_RANDOM.VALUE \* (v\_end\_date - v\_start\_date),  'Address'||i,  ROUND(DBMS\_RANDOM.VALUE(1, 2))  );  END LOOP;  COMMIT;  END; |

Листинг 4.4 – Функция заполнения таблицы USERS

Эта процедура INSERT\_100K\_LINES создана для вставки 100 000 строк в таблицу USERS базы данных. Каждая вставляемая строка представляет собой уникального пользователя с именем, фамилией, отчеством, электронной почтой, паролем, датой рождения, адресом и ролью. Имя, фамилия, отчество, электронная почта и пароль генерируются с использованием инкрементируемого числа в цикле для обеспечения уникальности. Дата рождения и роль также генерируются случайным образом в заданном диапазоне. После вставки всех строк в таблицу выполняется коммит, чтобы сохранить изменения.

## 4.4 Создание функций

В Oracle, функции - это объекты базы данных, которые могут принимать аргументы, выполнять определенные операции и возвращать результат. Они предоставляют возможность упрощения сложных операций, повторного использования кода и улучшения производительности запросов. Функции позволяют разработчикам создавать более гибкие и мощные запросы к данным, а также повышают эффективность и легкость поддержки кода в базе данных Oracle. В рамках базы данных было разработано множество функций разного характера, такие как вычисление среднего рейтинга велосипеда (AVG\_RATE), хэширование пароля (HASH\_PASS), регистрация пользователя (REGISTER), вход пользователя (LOGIN), добавление велосипеда (ADD\_BIKE), добавление описания (ADD\_DESC), добавление категории (ADD\_CATEGORY), добавление роли (ADD\_ROLE), добавление заказа (ADD\_ORDER), обновление информации о пользователе (UPDATE\_USER), добавление рейтинга (ADD\_RATE) и другие. Пример функции представлен в листинге 4.5.

|  |
| --- |
| CREATE OR REPLACE FUNCTION REGISTER  (  NAME IN NVARCHAR2 ,  SURNAME IN NVARCHAR2 ,  FATHERNAME IN NVARCHAR2,  EMAIL IN NVARCHAR2,  PASSWORD IN NVARCHAR2,  BIRTH IN DATE,  USER\_ADRESS IN NVARCHAR2,  ROLE IN INTEGER  )  RETURN INTEGER  IS  ID INTEGER;  BEGIN  INSERT INTO USERS(USER\_NAME,USER\_SURNAME,USER\_FATHERNAME,USER\_EMAIL,PASSWORD,DATE\_OF\_BIRTH,ADRESS,ROLE\_ID)  VALUES (NAME , SURNAME ,FATHERNAME,EMAIL,HASH\_PASS(PASSWORD),BIRTH,USER\_ADRESS,ROLE)  RETURNING USERS.USER\_ID INTO ID;  RETURN ID;  EXCEPTION  WHEN NO\_DATA\_FOUND THEN  RETURN 0;  WHEN TOO\_MANY\_ROWS THEN  RETURN 0;  WHEN OTHERS  THEN RETURN 0;  END; |

Листинг 4.5– Функция REGISTER

Эти функции обеспечивают различные функциональные возможности, такие как аутентификация, добавление данных и обновление информации в базе данных.

### 4.4.1 Выборка данных из таблиц

Для вывода данных из таблиц были созданы следующие функции:

GETUSERBYID, GETBIKEBYID, GETORDERBYID – получение данных из основных таблиц по уникальному идентификатору.

GETUSERS, GETBIKES, GETORDERS – получение полного списка всех имеющихся данных из основных таблиц.

SORT\_USER\_BY\_NAME,SORT\_USER\_BY\_SURNAME, SORT\_USER\_BY\_BIRTH, SORT\_USER\_BY\_ID, SORT\_BIKE\_BY\_ID, SORT\_BIKE\_BY\_NAME, SORT\_BIKE\_BY\_SELLER, SORT\_BIKE\_BY\_PRICE, SORT\_BIKE\_BY\_CATEGORY, SORT\_BIKE\_BY\_RATE и т.д. – функции для сортировки данных и вывода их в виде выборки.

Пример одной из функций выборки представлен в листинге 4.6.

|  |
| --- |
| CREATE OR REPLACE FUNCTION GETUSERS  RETURN USER\_TABLE\_RESULT  IS  U\_RESULT USER\_TABLE\_RESULT := USER\_TABLE\_RESULT();  BEGIN  SELECT USER\_INFO(  U.USER\_ID,  U.USER\_NAME,  U.USER\_SURNAME,  U.USER\_FATHERNAME,  U.USER\_EMAIL,  U.PASSWORD,  U.DATE\_OF\_BIRTH,  U.ADRESS,  R.ROLENAME  )  BULK COLLECT INTO U\_RESULT  FROM USERS U  LEFT JOIN ROLES R ON R.ROLE\_ID = U.ROLE\_ID  ;  RETURN U\_RESULT;  EXCEPTION  WHEN NO\_DATA\_FOUND THEN  RETURN NULL;  WHEN TOO\_MANY\_ROWS THEN  RETURN NULL;  WHEN OTHERS  THEN RETURN NULL;  END; |

Листинг 4.6 – Функция GETUSERS

Для реализации создания функций производящих выборку данных их таблиц были созданы специальные типы (TYPE) для последующей записи в них данных и вывода в виде выборки.   
 USER\_INFO, USER\_TABLE\_RESULT – содержит все поля таблицы USERS.  
 BIKE\_INFO, BIKE\_TABLE\_RESULT – содержит все поля таблицы BIKES.  
 ORDER\_INFO, ORDER\_RESULT\_TABLE – содержит все поля таблицы ORDERS.  
 Пример скрипта по созданию типа вложенной таблицы есть на листинге 4.5.

|  |
| --- |
| CREATE OR REPLACE TYPE USER\_INFO AS OBJECT (  USER\_ID INTEGER ,  USER\_NAME NVARCHAR2(255),  USER\_SURNAME NVARCHAR2(255),  USER\_FATHERNAME NVARCHAR2(255),  USER\_EMAIL NVARCHAR2(255),  PASSWORD NVARCHAR2(255),  DATE\_OF\_BIRTH DATE,  ADRESS NVARCHAR2(255) ,  ROLE\_NAME NVARCHAR2(255)  );  CREATE OR REPLACE TYPE USER\_TABLE\_RESULT AS TABLE OF USER\_INFO; |

Листинг 4.5 – Тип USER\_INFO и USER\_TABLE\_RESULT

Тип записи — это составной тип данных, состоящий из одного или нескольких идентификаторов и соответствующих им типов данных.

### 4.4.2 Выборка данных по поисковому запросу

Для поиска определенного велосипеда или пользователя, можно использовать функцию SEARCH\_USER\_BY\_NAME или SEARCH\_BIKE\_BY\_NAME. Данная функция принимает на вход поисковый запрос в виде текста и возвращает таблицу с данными о найденных композициях или пользователях. Функция представлена на листинге 4.7.

|  |
| --- |
| CREATE OR REPLACE FUNCTION SEARCH\_USER\_BY\_NAME  (  NAME IN USERS.USER\_NAME%TYPE  )  RETURN USER\_TABLE\_RESULT  IS  U\_SORTED USER\_TABLE\_RESULT := USER\_TABLE\_RESULT();  BEGIN  SELECT USER\_INFO(  U.USER\_ID,  U.USER\_NAME,  U.USER\_SURNAME,  U.USER\_FATHERNAME,  U.USER\_EMAIL,  U.PASSWORD,  U.DATE\_OF\_BIRTH,  U.ADRESS,  R.ROLENAME  )  BULK COLLECT INTO U\_SORTED  FROM USERS U  LEFT JOIN ROLES R ON R.ROLE\_ID = U.ROLE\_ID  WHERE U.USER\_NAME LIKE NAME  ORDER BY U.USER\_NAME;  RETURN U\_SORTED;  END; |

Листинг 4.7 – Функция SEARCH\_USER\_BY\_NAME

Для поиска используется оператор LIKE, который позволяет выполнить поиск без учета регистра символов. Поисковый запрос в виде текста передается в функцию в качестве аргумента query\_text. Затем возвращается таблица с данными о найденных композициях или пользователях. Кроме того, для поиска велосипедов или пользователей можно применить фильтрацию по ID, что может помочь узнать больше о конкретном жанре музыки или найти треки, которые подходят к определенному настроению. Для этого можно использовать функцию GETUSERBYID, которая принимает на вход идентификатор и возвращает таблицу с данными о найденных пользователях. Функция представлена на листинге 4.8.

|  |
| --- |
| CREATE OR REPLACE FUNCTION GETUSERBYID  (  USER IN INTEGER  )  RETURN USER\_TABLE\_RESULT  IS  U\_RESULT USER\_TABLE\_RESULT := USER\_TABLE\_RESULT();  BEGIN  SELECT USER\_INFO(  U.USER\_ID,  U.USER\_NAME,  U.USER\_SURNAME,  U.USER\_FATHERNAME,  U.USER\_EMAIL,  U.PASSWORD,  U.DATE\_OF\_BIRTH,  U.ADRESS,  R.ROLENAME  )  BULK COLLECT INTO U\_RESULT  FROM USERS U  INNER JOIN ROLES R ON R.ROLE\_ID = U.ROLE\_ID  WHERE U.USER\_ID = USER  ORDER BY U.USER\_ID  ;  RETURN U\_RESULT;  EXCEPTION  WHEN NO\_DATA\_FOUND THEN  RETURN NULL;  WHEN TOO\_MANY\_ROWS THEN  RETURN NULL;  WHEN OTHERS  THEN RETURN NULL;  END; |

Листинг 4.8 – Функция GETUSERBYID

Таким образом, в данном разделе были представлены примеры функций, которые позволяют искать треки.

## 4.5 Создание пакетов процедур и функций для базы данных

Для управления данными через приложение пользователи и администраторы используют хранимые процедуры и функции. Хранимая процедура представляет собой набор SQL-инструкций, который компилируется один раз и хранится на сервере. Функция также представляет собой набор SQL-инструкций, но возвращает значение, которое может быть использовано внутри другой инструкции SQL. Для упорядочивания функций и процедур были созданы пакеты USER\_PKG\_PROC и USER\_PGK\_FUNC, которые представляют из себя коллекции функций и процедур для быстрого доступа к ним для пользователей.

Написанные в ходе разработки курсового проекта процедуры и функции можно разбить на несколько категорий:

1. Выборка данных из таблиц.
2. Выборка данных по поисковому запросу.
3. Заполнение таблиц 100 000 строк.
4. Добавление данных в таблицы.
5. Удаление данных из таблиц.
6. Изменение данных в таблицах.
7. Дополнительные функции и процедуры.

Отличие функций от процедур состоит в том, что функции возвращают значение, которое может быть использовано в других SQL-запросах, а процедуры не возвращают значение. Кроме того, функции могут быть использованы в выражениях SQL, например, для вычисления значения поля в запросе SELECT. Пример создание спецификации пакета для процедур на листинге 4.9.

|  |
| --- |
| CREATE OR REPLACE PACKAGE USER\_PKG\_PROC AS  -- ADD TO CART  PROCEDURE ADD\_TO\_CART(  USER\_ID IN INTEGER,  BIKE\_ID IN INTEGER,  QUANTITY IN NUMBER);  -- DELETE FROM CART  PROCEDURE DEL\_FROM\_CART(  USER\_ID IN INTEGER,  BIKE\_ID IN INTEGER);  -- UPDATE PASSWORD  PROCEDURE UPDATE\_PASS(  USER\_ID IN INTEGER,  EMAIL IN NVARCHAR2,  PASSWORD IN NVARCHAR2);  . . .  END USER\_PKG\_PROC; |

Листинг 4.9 – Часть пакета процедур User\_PKG\_PROC

## 4.6 Создание триггеров

Триггер базы данных — это объект базы данных, который выполняет некоторое действие автоматически при определенных событиях в таблице или представлении базы данных. Триггер может быть запрограммирован на срабатывание при вставке, обновлении или удалении строк в таблице.

Триггеры используются для обеспечения целостности данных и контроля доступа к данным, а также для автоматической обработки данных при выполнении определенных операций в таблице.

Первый триггер USER\_BIRTH\_DATE, созданный в таблице USERS, проверяет, что дата рождения нового пользователя не является будущей датой. Если дата рождения нового пользователя больше или равна текущей дате, то триггер генерирует исключение с сообщением об ошибке " Дата рождения не может быть в будущем.". Триггер представлен в листинге 4.10.

|  |
| --- |
| CREATE OR REPLACE TRIGGER USER\_BIRTH\_DATE  BEFORE INSERT OR UPDATE ON USERS FOR EACH ROW  DECLARE  USER\_BIRTH EXCEPTION;  BEGIN  IF :NEW.DATE\_OF\_BIRTH IS NOT NULL AND :NEW.DATE\_OF\_BIRTH > SYSDATE  THEN RAISE USER\_BIRTH;  END IF;  EXCEPTION  WHEN USER\_BIRTH  THEN RAISE\_APPLICATION\_ERROR(-20001, 'Дата рождения не может быть в будущем.');  END; |

Листинг 4.10 – Триггер USER\_BIRTH\_DATE

Второй триггер CHECK\_EMAIL срабатывает перед вставкой или обновлением строки в таблице USERS для каждой строки. Его цель - проверить уникальность электронной почты (USER\_EMAIL) для новой или обновляемой записи. Внутри блока триггера, если новое значение электронной почты не является нулевым, и если новое значение отличается от старого (это условие нужно для обновлений, чтобы избежать срабатывания триггера, если электронная почта не была изменена), триггер выполняет запрос SELECT COUNT(\*) INTO EMAIL\_COUNT FROM USERS WHERE USER\_EMAIL = :NEW.USER\_EMAIL; для подсчета количества записей с таким же значением электронной почты в таблице. Если количество записей больше нуля, это означает, что электронная почта уже существует в базе данных, и триггер вызывает пользовательское исключение EMAIL\_EXISTS. В случае срабатывания этого исключения, триггер использует процедуру RAISE\_APPLICATION\_ERROR, чтобы сгенерировать ошибку с кодом -20001 и сообщением "Пользователь с такой же электронной почтой уже существует.". Триггер представлен в листинге 4.11.

|  |
| --- |
| /  CREATE OR REPLACE TRIGGER CHECK\_EMAIL  BEFORE INSERT OR UPDATE ON USERS  FOR EACH ROW  DECLARE  EMAIL\_EXISTS EXCEPTION;  EMAIL\_COUNT NUMBER;  BEGIN  IF :NEW.USER\_EMAIL IS NOT NULL THEN  IF :NEW.USER\_EMAIL <> :OLD.USER\_EMAIL THEN  SELECT COUNT(\*) INTO EMAIL\_COUNT FROM USERS WHERE USER\_EMAIL = :NEW.USER\_EMAIL;  IF EMAIL\_COUNT > 0 THEN  RAISE EMAIL\_EXISTS;  END IF;  END IF;  END IF;  EXCEPTION  WHEN EMAIL\_EXISTS THEN  RAISE\_APPLICATION\_ERROR(-20001, 'Пользователь с такой же электронной почтой уже существует.');  END;  / |

Листинг 4.11 – Триггер CHECK\_EMAIL

Таким образом, этот триггеры обеспечивает уникальность значений в таблице пользователей, предотвращая добавление или обновление записей. Были созданы так же и другие триггеры проверяющие данные на валидность. Они будут представлены в приложении Е.

## 4.7 Создание индексов

Индекс — объект базы данных, который используется для ускорения поиска данных. В случае большого количества строк в таблице, последовательный поиск данных может занимать много времени. Индекс формируется на основе значений одного или нескольких столбцов таблицы и указателей на соответствующие строки таблицы, что позволяет быстро искать строки, удовлетворяющие заданному критерию поиска. Использование индексов ускоряет работу с базой данных, потому что они имеют оптимизированную структуру для поиска, например, сбалансированное дерево. Для того, чтобы быстро находить данные по их названию или описанию, были созданы индексы:

* IDX\_BIKE\_NAME, IDX\_USER\_NAME, IDX\_USER\_SURNAME, IDX\_USER\_FATHERNAME, IDX\_USER\_EMAIL, IDX\_USER\_ADRESS – индексы созданные для полнотекстового поиска в таблицах BIKES и USERS.
* IDX\_DESC\_DESC, IDX\_DESC\_MATERIAL, IDX\_DESC\_TYPE, IDX\_RATE\_MSG - индексы созданные для полнотекстового поиска с помощью совпадений в тексте.
* IDX\_USER\_BIRTH, IDX\_ORDER\_DATE, IDX\_RATE\_DATE – простые индексы для быстрого поиска по дате.

## 4.8 Используемая технология

Технология Full Text Search (FTS) представляет собой метод поиска и анализа текстовых данных в базе данных с целью эффективного извлечения информации. В отличие от обычных запросов, которые работают с точным совпадением значений, FTS позволяет осуществлять поиск по частичным совпадениям, синонимам, а также учитывать формы слов и их вариации. Эта технология позволяет проводить поиск не только по конкретным словам, но и по их контексту, что особенно полезно при работе с большим объемом текстовых данных, таких как статьи, отзывы, описания товаров и т. д. Системы FTS обычно предоставляют возможности для создания индексов, которые значительно повышают скорость выполнения поисковых запросов. Такие индексы позволяют быстро находить соответствия между поисковым запросом и текстовыми данными, что делает технологию FTS неотъемлемой частью систем управления базами данных, где требуется эффективный и масштабируемый поиск текста.

После созданных индексов были созданы специальные функции по поиску данных в таблицах по совпадениям и содержанию параметров введенных пользователем.

Например функция CONTAINS\_BIKE, которая использует метод CONTAINS.

Он позволяет осуществлять поиск текстовых данных в структурированных или полуструктурированных документах, а также в текстовых полях таблиц базы данных. Метод CONTAINS проверяет, содержит ли указанное поле или документ указанный поисковый запрос или ключевое слово. Этот метод может учитывать различные языковые особенности, такие как стемминг, игнорирование стоп-слов и фразовый поиск, что делает его мощным инструментом для эффективного и гибкого поиска текстовых данных в базе данных Oracle. Пример функции представлен в листинге 4.12.

|  |
| --- |
| CREATE OR REPLACE FUNCTION CONTAINS\_BIKE  (  NAME IN VARCHAR2  )  RETURN BIKE\_TABLE\_RESULT  IS  B\_RESULT BIKE\_TABLE\_RESULT := BIKE\_TABLE\_RESULT();  BEGIN  SELECT BIKE\_INFO  (  B.BIKE\_ID,  B.SELLER\_ID,  B.BIKE\_NAME,  B.PRICE,  B.CATEGORY\_ID,  AVG\_RATE(B.BIKE\_ID),  D.BIKE\_DESC,  D.BIKE\_TYPE,  D.MATERIAL,  D.WEIGHT,  D.HEIGHT  )  BULK COLLECT INTO B\_RESULT  FROM BIKES B  LEFT JOIN RATINGS R ON R.BIKE\_ID = B.BIKE\_ID  LEFT JOIN DESCRIPTIONS D ON D.BIKE\_ID = B.BIKE\_ID  WHERE CONTAINS(BIKE\_NAME,NAME,1) > 0  ORDER BY B.BIKE\_ID;  RETURN B\_RESULT;  END; |

Листинг 4.12 – Функция CONTAINS\_BIKE

Также был использован метод MATCHES в функции MATCH\_DESC и других. В отличие от CONTAINS, который используется с полнотекстовыми индексами, метод MATCHES позволяет осуществлять поиск на основе регулярных выражений. Этот метод может быть более гибким, чем CONTAINS, так как он позволяет задавать более сложные шаблоны поиска с использованием регулярных выражений, которые могут включать в себя шаблоны символов, классы символов, квантификаторы и другие конструкции. Пример функции с этим методом представлен в листинге 4.13.

|  |
| --- |
| CREATE OR REPLACE FUNCTION MATCH\_DESC  (  TEXT VARCHAR2  )  RETURN INTEGER  IS  DID DESCRIPTIONS.DESC\_ID%TYPE;  BEGIN  SELECT DESC\_ID INTO DID FROM DESCRIPTIONS WHERE MATCHES(BIKE\_DESC,TEXT) > 0;  RETURN DID;  END; |

Листинг 4.13 – Функция MATCH\_DESC

Данная функция принимает на вход параметр который будет шаблоном поиска и возвращать идентификатор описания которые найдет. Еще больше примеров можно найти в приложении В.

## 4.7 Создание директорий

Всего была создано восемь директорий, показанные в листинге 4.14, для указания пути к файлам, содержащим данные импортированные из таблиц.

|  |
| --- |
| CREATE OR REPLACE DIRECTORY BIKES\_JSON AS 'C:\COURSE\_WORK\_DATA';  CREATE OR REPLACE DIRECTORY USERS\_JSON AS 'C:\COURSE\_WORK\_DATA';  CREATE OR REPLACE DIRECTORY ORDERS\_JSON AS 'C:\COURSE\_WORK\_DATA';  CREATE OR REPLACE DIRECTORY DESCRIPTIONS\_JSON AS 'C:\COURSE\_WORK\_DATA';  CREATE OR REPLACE DIRECTORY RATINGS\_JSON AS 'C:\COURSE\_WORK\_DATA';  CREATE OR REPLACE DIRECTORY CART\_JSON AS 'C:\COURSE\_WORK\_DATA';  CREATE OR REPLACE DIRECTORY ROLES\_JSON AS 'C:\COURSE\_WORK\_DATA';  CREATE OR REPLACE DIRECTORY CATEGORIES\_JSON AS 'C:\COURSE\_WORK\_DATA'; |

Листинг 4.14 – Директории указывающая на резервные файлы

Одним из преимуществ использования директории для хранения данных является то, что это позволяет значительно сократить объем кода, который необходим для импорта и экспорта данных в базе данных. Вместо того, чтобы указывать путь к файлу каждый раз при выполнении импорта или экспорта данных, вы можете просто указать директорию, которая содержит все необходимые файлы.

## 4.8 Вывод

В данном разделе была рассмотрена разработка объектов базы данных для магазина велосипедов. Tакже была описана использованная технология Full Text Search (FTS) которая, представляет собой метод поиска и анализа текстовых данных в базе данных с целью эффективного извлечения информации. Было разработано множество функций и процедур для заполнения, обновления, удаления, редактирования данных в таблицах. Были созданы триггеры для надежности создания обьектов без ошибок и для ее эффективной работы. Представления, которые дают общее представление данных в виде выборки. Индексы для быстрого поиска и анализа данных.

# 5. Описание процедур импорта и экспорта

## 5.1 Экспорт данных в JSON

В базе данных Oracle были разработаны несколько процедур, предназначенные для экспорта данных из таблиц в формате JSON. Например процедура EXPORT\_BIKES\_TO\_JSON эта процедура создает JSON-файл, содержащий информацию о велосипедах из базы данных. Она выбирает данные из таблицы BIKES, формирует JSON-объекты для каждого велосипеда и записывает их в файл BIKES.JSON.

|  |
| --- |
| CREATE OR REPLACE PROCEDURE EXPORT\_BIKES\_TO\_JSON  IS  CURSOR BIKES\_ARR IS SELECT '[' || LISTAGG(JSON\_OBJECT(  'ID' IS BIKE\_ID,  'SELLER' IS SELLER\_ID,  'NAME' IS BIKE\_NAME,  'PRICE' IS PRICE,  'CATEGORY' IS CATEGORY\_ID  ),',') || ']' AS JSON\_DATA  FROM BIKES;  JS\_FILE UTL\_FILE.FILE\_TYPE;  BEGIN  JS\_FILE := UTL\_FILE.FOPEN('BIKES\_JSON','BIKES.JSON','W');  FOR BIKES\_REC IN BIKES\_ARR LOOP  UTL\_FILE.PUT\_LINE(JS\_FILE, BIKES\_REC.JSON\_DATA);  END LOOP;  UTL\_FILE.FFLUSH(JS\_FILE);  UTL\_FILE.FCLOSE(JS\_FILE);  EXCEPTION  WHEN OTHERS THEN  UTL\_FILE.FCLOSE(JS\_FILE);  RAISE;  END; |

Листинг 5.1 – Процедура EXPORT\_BIKES\_TO\_JSON

Для каждой таблицы вызывается процедура с соответствующим запросом и именем файла, чтобы экспортировать данные этой таблицы в отдельный JSON-файл. Эти процедуры экспортирует данные из таблиц USERS, ORDERS, BIKES, RATINGS, CATEGORIES, DESCRIPTIONS, CART, ROLES.

## 5.2 Импорт данных из JSON

Для импорта JSON-данных в базу данных Oracle можно воспользоваться функцией JSON\_VALUE, которая позволяет преобразовать JSON-данные в строки и столбцы таблицы. Сначала необходимо создать таблицу, соответствующую структуре JSON-данных. Затем с помощью оператора INSERT INTO и функции JSON\_VALUE можно загрузить данные из JSON в таблицу. Была использована функция MERGE для предотвращения вставки идентичных данных. Так-же была написана процедура для универсального извлечения информации из файла и последующего использования. Процедура для импорта информации о пользователях представлена в листинге 5.2.

|  |
| --- |
| CREATE OR REPLACE PROCEDURE IMPORT\_USERS\_FROM\_JSON  IS  v\_file UTL\_FILE.FILE\_TYPE;  v\_line VARCHAR2(4000);  BEGIN  v\_file := UTL\_FILE.FOPEN('USERS\_JSON', 'USERS.JSON', 'R');  LOOP  BEGIN  UTL\_FILE.GET\_LINE(v\_file, v\_line);  EXIT WHEN v\_line IS NULL;    MERGE INTO USERS u  USING (  SELECT  JSON\_VALUE(v\_line, '$.NAME') AS USER\_NAME,  JSON\_VALUE(v\_line, '$.SURNAME') AS USER\_SURNAME,  JSON\_VALUE(v\_line, '$.FATHERNAME') AS USER\_FATHERNAME,  JSON\_VALUE(v\_line, '$.EMAIL') AS USER\_EMAIL,  JSON\_VALUE(v\_line, '$.PASSWORD') AS PASSWORD,  TO\_DATE(JSON\_VALUE(v\_line, '$.BIRTH'), 'YYYY-MM-DD"T"HH24:MI:SS') AS DATE\_OF\_BIRTH,  JSON\_VALUE(v\_line, '$.ADRESS') AS ADRESS,  JSON\_VALUE(v\_line, '$.ROLE') AS ROLE\_ID  FROM DUAL  ) json\_data  ON (u.USER\_EMAIL = json\_data.USER\_EMAIL)  WHEN NOT MATCHED THEN  INSERT (USER\_NAME, USER\_SURNAME, USER\_FATHERNAME, USER\_EMAIL, PASSWORD, DATE\_OF\_BIRTH, ADRESS, ROLE\_ID)  VALUES (json\_data.USER\_NAME, json\_data.USER\_SURNAME, json\_data.USER\_FATHERNAME, json\_data.USER\_EMAIL, json\_data.PASSWORD, json\_data.DATE\_OF\_BIRTH, json\_data.ADRESS, json\_data.ROLE\_ID);    EXCEPTION  WHEN NO\_DATA\_FOUND THEN  EXIT;  END;  END LOOP;  UTL\_FILE.FCLOSE(v\_file);  EXCEPTION  WHEN OTHERS THEN  IF UTL\_FILE.IS\_OPEN(v\_file) THEN  UTL\_FILE.FCLOSE(v\_file);  END IF;  RAISE;  END; |

Листинг 5.2 – Процедура импорта IMPORT\_USERS\_FROM\_JSON

Эта процедура позволяет эффективно импортировать данные пользователей из JSON-файла в таблицу базы данных

# 6. Тестирование производительности

## 6.1 Тестирование производительности базы данных

Для тестирования производительности базы данных, нам нужно заполнить таблицы достаточным количеством записей. Для данного случая мы заполнили таблицу USERS по 100.000 строк, скрипт заполнения таблицы представлен в пункте 4.3.1.

Далее, для улучшения производительности запросов, мы создали несколько индексов. Для таблицы USERS мы создали индексы на столбцы USER\_NAME, USER\_SURNAME, USER\_FATHERNAME, USER\_EMAIL, ADRESS., DATE\_OF\_BIRTH. Для таблицы RATINGS мы создали индекс на столбец USER\_ID, а для таблицы CART - на столбец USER\_ID. Пример для создания индексов показан в листинге 6.1.

Теперь давайте рассмотрим производительность базы данных в контексте запросов на выборку данных.

|  |
| --- |
| CREATE INDEX IDX\_USER\_BIRTH ON USERS(DATE\_OF\_BIRTH); |

Листинг 6.1 – Индекс для таблицы

При выполнении запроса показанного далее в листинге 5.1 и 5.2 на выборку данных из нескольких таблиц без использования индексов, время ожидания может значительно увеличиваться из-за необходимости выполнения большого количества операций сканирования таблиц. Так, при выполнении запроса на выборку данных из таблиц ORDERS, USERS без использования индексов, время выполнения, представленное на рисунке 5.1, составило около 113 миллисекунд.



Рисунок 5.1 – Скорость выполнения без индекса

Создание индекса на столбце USER\_EMAIL позволило сократить время выполнения запроса на выборку данных из таблиц ORDERS, USERS до 50 миллисекунд, как видно на рисунке 5.2.



Рисунок 5.2 – Скорость выполнения с индексом

Таким образом, использование индексов является важным приемом оптимизации производительности базы данных и может сократить время выполнения запросов на выборку данных из таблиц, в данном случае результат ускорился в 2 раза.

# Заключение

В проекте была успешно реализована база данных магазина велосипедов, которая включает управление информацией о пользователях, велосипедах, заказах, корзинах, описании и оценке товаров. Были применены разные возможности базы данных от простых операций вставки и удаления до написания сложных триггеров и процедур. Важным аспектом проекта было обеспечение безопасности данных с шифрования паролей, а так же применена технология полнотекстового поиска и анализа данных при помощи специальных методов для ускорения поиска данных. Была построена иерархия процедур управления данными. Это позволяет каждому пользователю получить доступ только к тем данным, которые ему необходимы. Было проведено тестирование производительности базы данных на большом объеме данных, что позволило оптимизировать ее структуру и проанализировать планы запросов к таблицам. Применение технологий базы данных в данной работе позволило значительно упростить и ускорить процессы управления данными магазина.

# Список используемых источников

1. Работа с файлами в Oracle [Электронный ресурс]. – Режим доступа: https://docs.oracle.com/cd/F49540\_01/DOC/server.815/a68001/utl\_file.htm. – Дата доступа: 01.05.2024.
2. Работа с JSON в Oracle [Электронный ресурс]. – Режим доступа: <https://docs.oracle.com/en/database/oracle/oracle-database/21/adjsn/query-json-data.html#GUID-119E5069-77F2-45DC-B6F0-A1B312945590>. – Дата доступа: 10.05.2024.
3. Функция хеширования в Oracle [Электронный ресурс]. – Режим доступа:

<https://docs.oracle.com/en/database/oracle/oracle/database/23/arpls/DBMS_CRYPTO.html> – Дата доступа: 01.05.2024.

1. Технология Oracle Text [Электронный ресурс]. – Режим доступа: <https://www.oracle.com/database/technologies/appdev/oracletext.html>. – Дата доступа: 13.05.2024.
2. Создание Oracle Text Indexes [Электронный ресурс]. – Режим доступа:

<https://www.oracle.com/webfolder/technetwork/tutorials/obe/db/12c/r1/appdev/text/CreateIndexes/CreatingTextIndexes.html#section3>. – Дата доступа: 13.05.2024.

# Приложение А Листинг создания таблиц

|  |
| --- |
| CREATE TABLE ROLES (  ROLE\_ID INTEGER GENERATED ALWAYS AS IDENTITY(START WITH 1 INCREMENT BY 1) PRIMARY KEY,  ROLENAME VARCHAR2(255)  ) TABLESPACE TS\_USER;  CREATE TABLE USERS (  USER\_ID INTEGER GENERATED ALWAYS AS IDENTITY(START WITH 1 INCREMENT BY 1) PRIMARY KEY,  USER\_NAME VARCHAR2(255) NOT NULL,  USER\_SURNAME VARCHAR2(255) NOT NULL,  USER\_FATHERNAME VARCHAR2(255),  USER\_EMAIL VARCHAR2(255) NOT NULL,  PASSWORD NVARCHAR2(255) NOT NULL,  DATE\_OF\_BIRTH TIMESTAMP(6),  ADRESS VARCHAR2(255) NOT NULL,  ROLE\_ID INTEGER NOT NULL,  CONSTRAINT FK\_USER\_ROLE FOREIGN KEY(ROLE\_ID) REFERENCES ROLES(ROLE\_ID)  ) TABLESPACE TS\_USER;  CREATE TABLE CATEGORIES(  CATEGORY\_ID INTEGER GENERATED ALWAYS AS IDENTITY(START WITH 1 INCREMENT BY 1) PRIMARY KEY,  CAT\_NAME VARCHAR2(255) NOT NULL,  CONSTRAINT CHECK\_CAT CHECK(CAT\_NAME = 'MTB' OR CAT\_NAME = 'BMX' OR CAT\_NAME = 'MOUNTAIN' OR CAT\_NAME='KID' OR CAT\_NAME = 'DEFAULT')  ) TABLESPACE TS\_BIKE;  CREATE TABLE BIKES(  BIKE\_ID INTEGER GENERATED ALWAYS AS IDENTITY(START WITH 1 INCREMENT BY 1) PRIMARY KEY,  SELLER\_ID INTEGER NOT NULL,  BIKE\_NAME VARCHAR2(255) NOT NULL,  PRICE DECIMAL(10,2) NOT NULL,  CATEGORY\_ID INTEGER NOT NULL,  CONSTRAINT FK\_BIKE\_SELLER FOREIGN KEY(SELLER\_ID) REFERENCES USERS(USER\_ID),  CONSTRAINT FK\_BIKE\_CATEGORY FOREIGN KEY(CATEGORY\_ID) REFERENCES CATEGORIES(CATEGORY\_ID)  ) TABLESPACE TS\_BIKE;  CREATE TABLE DESCRIPTIONS(  DESC\_ID INTEGER GENERATED ALWAYS AS IDENTITY(START WITH 1 INCREMENT BY 1) PRIMARY KEY,  BIKE\_ID INTEGER NOT NULL,  BIKE\_DESC VARCHAR2(2000) NOT NULL,  BIKE\_TYPE VARCHAR2(100),  MATERIAL VARCHAR2(50),  WEIGHT FLOAT(5),  HEIGHT FLOAT(5),  CONSTRAINT FK\_DESC\_BIKE FOREIGN KEY(BIKE\_ID) REFERENCES BIKES(BIKE\_ID)  ) TABLESPACE TS\_BIKE;  CREATE TABLE RATINGS(  RATING\_ID INTEGER GENERATED ALWAYS AS IDENTITY(START WITH 1 INCREMENT BY 1) PRIMARY KEY,  BIKE\_ID INTEGER NOT NULL,  USER\_ID INTEGER NOT NULL,  RATING\_VALUE NUMBER(2) NOT NULL ,  RATING\_MESSAGE VARCHAR2(100) NOT NULL,  TIME\_STAMP TIMESTAMP NOT NULL,  CONSTRAINT FK\_RATE\_BIKE FOREIGN KEY(BIKE\_ID) REFERENCES BIKES(BIKE\_ID),  CONSTRAINT FK\_RATE\_USER FOREIGN KEY(USER\_ID) REFERENCES USERS(USER\_ID)  ) TABLESPACE TS\_BIKE;  CREATE TABLE CART(  CART\_ID INTEGER GENERATED ALWAYS AS IDENTITY(START WITH 1 INCREMENT BY 1) PRIMARY KEY,  USER\_ID INTEGER NOT NULL,  BIKE\_ID INTEGER NOT NULL,  QUANTITY NUMBER(6) NOT NULL,  CONSTRAINT FK\_CART\_USER FOREIGN KEY(USER\_ID) REFERENCES USERS(USER\_ID),  CONSTRAINT FK\_CART\_BIKE FOREIGN KEY(BIKE\_ID) REFERENCES BIKES(BIKE\_ID)  ) TABLESPACE TS\_CART;  CREATE TABLE ORDERS (  ORDER\_ID INTEGER GENERATED ALWAYS AS IDENTITY(START WITH 1 INCREMENT BY 1) PRIMARY KEY,  BUYER\_ID INTEGER NOT NULL,  BIKE\_ID INTEGER NOT NULL,  QUANTITY NUMBER(6) NOT NULL,  ORDER\_DATE TIMESTAMP(6) NOT NULL,  STATUS VARCHAR2(40) NOT NULL,  CONSTRAINT FK\_ORDER\_BIKE FOREIGN KEY(BIKE\_ID) REFERENCES BIKES(BIKE\_ID),  CONSTRAINT FK\_ORDER\_BUYER FOREIGN KEY(BUYER\_ID) REFERENCES USERS(USER\_ID)  ) TABLESPACE TS\_ORDER; |

# Приложение Б Листинг создания процедур

|  |
| --- |
| -- PROCEDURES:  -- UPDATE DB  -- ADD BIKE TO CART  CREATE OR REPLACE PROCEDURE ADD\_TO\_CART  (  USER INTEGER ,  BIKE INTEGER ,  QNT NUMBER  )  IS  ID CART.CART\_ID%TYPE;  BEGIN  INSERT INTO CART(USER\_ID,BIKE\_ID,QUANTITY)VALUES(USER,BIKE,QNT)  RETURNING CART.CART\_ID INTO ID;  EXCEPTION  WHEN NO\_DATA\_FOUND THEN  raise\_application\_error(-20001,'An error was encountered - '||SQLCODE||' -ERROR- '||SQLERRM);  WHEN TOO\_MANY\_ROWS THEN  raise\_application\_error(-20001,'An error was encountered - '||SQLCODE||' -ERROR- '||SQLERRM);  WHEN OTHERS  THEN raise\_application\_error(-20001,'An error was encountered - '||SQLCODE||' -ERROR- '||SQLERRM);  END;  -- DEL BIKE FROM CART  CREATE OR REPLACE PROCEDURE DEL\_FROM\_CART  (  USER IN INTEGER ,  BIKE IN INTEGER  )  IS  ID CART.CART\_ID%TYPE;  BEGIN  DELETE FROM CART WHERE USER\_ID = USER AND BIKE\_ID = BIKE  RETURNING CART.CART\_ID INTO ID;  EXCEPTION  WHEN OTHERS  THEN raise\_application\_error(-20001,'An error was encountered - '||SQLCODE||' -ERROR- '||SQLERRM);  END;    -- UPDATE PASSWORD  CREATE OR REPLACE PROCEDURE UPDATE\_PASS  (  USER IN INTEGER,  EMAIL IN NVARCHAR2,  PASS IN NVARCHAR2  )  IS  ID USERS.USER\_ID%TYPE;  BEGIN  UPDATE USERS SET PASSWORD = HASH\_PASS(PASS) WHERE USERS.USER\_ID = USER AND USER\_EMAIL = EMAIL  RETURNING USERS.USER\_ID INTO ID  ;  EXCEPTION  WHEN NO\_DATA\_FOUND THEN  raise\_application\_error(-20001,'An error was encountered - '||SQLCODE||' -ERROR- '||SQLERRM);  WHEN TOO\_MANY\_ROWS THEN  raise\_application\_error(-20001,'An error was encountered - '||SQLCODE||' -ERROR- '||SQLERRM);  WHEN OTHERS  THEN raise\_application\_error(-20001,'An error was encountered - '||SQLCODE||' -ERROR- '||SQLERRM);  END;  -- UPDATE BIKE  CREATE OR REPLACE PROCEDURE UPDATE\_BIKE  (  BIKE IN INTEGER,  NAME IN NVARCHAR2,  COST IN DECIMAL,  CAT IN INTEGER  )  IS  ID BIKES.BIKE\_ID%TYPE;  BEGIN  UPDATE BIKES SET BIKE\_NAME = NAME,PRICE=COST,CATEGORY\_ID = CAT WHERE BIKE\_ID = BIKE  RETURNING BIKES.BIKE\_ID INTO ID;  EXCEPTION  WHEN NO\_DATA\_FOUND THEN  raise\_application\_error(-20001,'An error was encountered - '||SQLCODE||' -ERROR- '||SQLERRM);  WHEN TOO\_MANY\_ROWS THEN  raise\_application\_error(-20001,'An error was encountered - '||SQLCODE||' -ERROR- '||SQLERRM);  WHEN OTHERS  THEN raise\_application\_error(-20001,'An error was encountered - '||SQLCODE||' -ERROR- '||SQLERRM);  END;  -- DELETE USER  CREATE OR REPLACE PROCEDURE DEL\_USER  (  USER\_ID IN INTEGER  )  IS  BEGIN  DELETE FROM ORDERS WHERE BUYER\_ID = USER\_ID;  DELETE FROM CART WHERE USER\_ID = USER\_ID;  DELETE FROM DESCRIPTIONS WHERE BIKE\_ID IN (SELECT BIKE\_ID FROM BIKES WHERE SELLER\_ID = USER\_ID);  DELETE FROM RATINGS WHERE USER\_ID = USER\_ID;  DELETE FROM BIKES WHERE SELLER\_ID = USER\_ID;  DELETE FROM USERS WHERE USER\_ID = USER\_ID;    COMMIT;  EXCEPTION  WHEN OTHERS THEN  raise\_application\_error(-20001,'An error was encountered - '||SQLCODE||' -ERROR- '||SQLERRM);  END;  BEGIN  DEL\_USER(5);  END;  SELECT \* FROM USERS;  -- DELETE BIKE  CREATE OR REPLACE PROCEDURE DEL\_BIKE  (  BIKE IN INTEGER  )  IS  BID BIKES.BIKE\_ID%TYPE;  BEGIN  DELETE FROM ORDERS WHERE BIKE\_ID = BIKE;  DELETE FROM CART WHERE BIKE\_ID = BIKE;  DELETE FROM RATINGS WHERE BIKE\_ID = BIKE;  DELETE FROM DESCRIPTIONS WHERE BIKE\_ID = BIKE;  DELETE FROM BIKES WHERE BIKE\_ID = BIKE  RETURNING BIKES.BIKE\_ID INTO BID;  EXCEPTION  WHEN NO\_DATA\_FOUND THEN  raise\_application\_error(-20001,'An error was encountered - '||SQLCODE||' -ERROR- '||SQLERRM);  WHEN TOO\_MANY\_ROWS THEN  raise\_application\_error(-20001,'An error was encountered - '||SQLCODE||' -ERROR- '||SQLERRM);  WHEN OTHERS  THEN raise\_application\_error(-20001,'An error was encountered - '||SQLCODE||' -ERROR- '||SQLERRM);  END;    -- CLEAR CART  CREATE OR REPLACE PROCEDURE CLEAR\_CART  (  USER INTEGER  )  IS  CURSOR CART\_CURS IS SELECT CART\_ID ,USER\_ID, BIKE\_ID , QUANTITY FROM CART;  CT CART%ROWTYPE;  BEGIN  FOR CT IN CART\_CURS  LOOP  DELETE FROM CART WHERE CART.CART\_ID = CT.CART\_ID AND CART.USER\_ID = USER AND CART.BIKE\_ID = CT.BIKE\_ID AND CART.QUANTITY = CT.QUANTITY;  END LOOP;  EXCEPTION  WHEN NO\_DATA\_FOUND THEN  raise\_application\_error(-20001,'An error was encountered - '||SQLCODE||' -ERROR- '||SQLERRM);  WHEN TOO\_MANY\_ROWS THEN  raise\_application\_error(-20001,'An error was encountered - '||SQLCODE||' -ERROR- '||SQLERRM);  WHEN OTHERS  THEN raise\_application\_error(-20001,'An error was encountered - '||SQLCODE||' -ERROR- '||SQLERRM);  END;  -- CLEAR ORDER  CREATE OR REPLACE PROCEDURE CLEAR\_ORDER  (  USER IN INTEGER  )  IS  BEGIN  IF USER IS NOT NULL THEN  DELETE FROM ORDERS WHERE BUYER\_ID = USER ;  ELSE  RAISE NO\_DATA\_FOUND;  END IF;  EXCEPTION  WHEN NO\_DATA\_FOUND THEN  raise\_application\_error(-20001,'An error was encountered - '||SQLCODE||' -ERROR- '||SQLERRM);  WHEN TOO\_MANY\_ROWS THEN  raise\_application\_error(-20001,'An error was encountered - '||SQLCODE||' -ERROR- '||SQLERRM);  WHEN OTHERS  THEN raise\_application\_error(-20001,'An error was encountered - '||SQLCODE||' -ERROR- '||SQLERRM);  END;  -- UPDATE ORDER  CREATE OR REPLACE PROCEDURE UPDATE\_ORDER  (  ORD IN INTEGER,  BUYER IN INTEGER ,  BIKE IN INTEGER ,  QNT IN NUMBER ,  ODATE IN DATE ,  STAT IN NVARCHAR2  )  IS  ID ORDERS.ORDER\_ID%TYPE;  BEGIN  UPDATE ORDERS SET  BUYER\_ID = BUYER,  BIKE\_ID = BIKE,  QUANTITY = QNT,  ORDER\_DATE = ODATE,  STATUS = STAT  WHERE ORDER\_ID = ORD  RETURNING ORDERS.ORDER\_ID INTO ID;  EXCEPTION  WHEN NO\_DATA\_FOUND THEN  raise\_application\_error(-20001,'An error was encountered - '||SQLCODE||' -ERROR- '||SQLERRM);  WHEN TOO\_MANY\_ROWS THEN  raise\_application\_error(-20001,'An error was encountered - '||SQLCODE||' -ERROR- '||SQLERRM);  WHEN OTHERS  THEN raise\_application\_error(-20001,'An error was encountered - '||SQLCODE||' -ERROR- '||SQLERRM);  END;  -- DEL ORDER  CREATE OR REPLACE PROCEDURE DEL\_ORDER  (  ORD IN INTEGER  )  IS  OID ORDERS.ORDER\_ID%TYPE;  BEGIN  DELETE FROM ORDERS WHERE ORDER\_ID = ORD  RETURNING ORDERS.ORDER\_ID INTO OID;  EXCEPTION  WHEN NO\_DATA\_FOUND THEN  raise\_application\_error(-20001,'An error was encountered - '||SQLCODE||' -ERROR- '||SQLERRM);  WHEN TOO\_MANY\_ROWS THEN  raise\_application\_error(-20001,'An error was encountered - '||SQLCODE||' -ERROR- '||SQLERRM);  WHEN OTHERS  THEN raise\_application\_error(-20001,'An error was encountered - '||SQLCODE||' -ERROR- '||SQLERRM);  END;  -- DEL ROLE  CREATE OR REPLACE PROCEDURE DEL\_ROLE  (  RID IN INTEGER  )  IS  ID ROLES.ROLE\_ID%TYPE;  BEGIN  IF RID IS NOT NULL THEN  DELETE FROM ROLES WHERE ROLE\_ID = RID  RETURNING ROLES.ROLE\_ID INTO ID  ;  ELSE  RAISE NO\_DATA\_FOUND;  END IF;  EXCEPTION  WHEN NO\_DATA\_FOUND THEN  raise\_application\_error(-20001,'An error was encountered - '||SQLCODE||' -ERROR- '||SQLERRM);  WHEN TOO\_MANY\_ROWS THEN  raise\_application\_error(-20001,'An error was encountered - '||SQLCODE||' -ERROR- '||SQLERRM);  WHEN OTHERS  THEN raise\_application\_error(-20001,'An error was encountered - '||SQLCODE||' -ERROR- '||SQLERRM);  END;  -- UPD ROLE  CREATE OR REPLACE PROCEDURE UPD\_ROLE  (  RID IN INTEGER,  RNAME IN NVARCHAR2  )  IS  ID ROLES.ROLE\_ID%TYPE;  BEGIN  IF RID IS NOT NULL THEN  UPDATE ROLES SET ROLENAME = RNAME WHERE ROLE\_ID = RID  RETURNING ROLES.ROLE\_ID INTO ID;  ELSE  RAISE NO\_DATA\_FOUND;  END IF;  EXCEPTION  WHEN NO\_DATA\_FOUND THEN  raise\_application\_error(-20001,'An error was encountered - '||SQLCODE||' -ERROR- '||SQLERRM);  WHEN TOO\_MANY\_ROWS THEN  raise\_application\_error(-20001,'An error was encountered - '||SQLCODE||' -ERROR- '||SQLERRM);  WHEN OTHERS  THEN raise\_application\_error(-20001,'An error was encountered - '||SQLCODE||' -ERROR- '||SQLERRM);  END;  -- DEL CATEGORY  CREATE OR REPLACE PROCEDURE DEL\_CATEGORY  (  CID IN INTEGER  )  IS  BID BIKES.BIKE\_ID%TYPE;  BEGIN  IF CID IS NOT NULL THEN  FOR order\_rec IN (SELECT BIKE\_ID FROM BIKES WHERE CATEGORY\_ID = CID) LOOP  DELETE FROM ORDERS WHERE BIKE\_ID = order\_rec.BIKE\_ID;  END LOOP;  FOR cart\_rec IN (SELECT BIKE\_ID FROM BIKES WHERE CATEGORY\_ID = CID) LOOP  DELETE FROM CART WHERE BIKE\_ID = cart\_rec.BIKE\_ID;  END LOOP;  FOR desc\_rec IN (SELECT BIKE\_ID FROM BIKES WHERE CATEGORY\_ID = CID) LOOP  DELETE FROM DESCRIPTIONS WHERE BIKE\_ID = desc\_rec.BIKE\_ID;  END LOOP;  FOR rating\_rec IN (SELECT BIKE\_ID FROM BIKES WHERE CATEGORY\_ID = CID) LOOP  DELETE FROM RATINGS WHERE BIKE\_ID = rating\_rec.BIKE\_ID;  END LOOP;  DELETE FROM BIKES WHERE CATEGORY\_ID = CID RETURNING BIKE\_ID INTO BID;  DELETE FROM CATEGORIES WHERE CATEGORY\_ID = CID;  ELSE  RAISE NO\_DATA\_FOUND;  END IF;  EXCEPTION  WHEN NO\_DATA\_FOUND THEN  raise\_application\_error(-20001,'An error was encountered - '||SQLCODE||' -ERROR- '||SQLERRM);  WHEN TOO\_MANY\_ROWS THEN  raise\_application\_error(-20001,'An error was encountered - '||SQLCODE||' -ERROR- '||SQLERRM);  WHEN OTHERS  THEN raise\_application\_error(-20001,'An error was encountered - '||SQLCODE||' -ERROR- '||SQLERRM);  END;  -- UPD CATEGORY  CREATE OR REPLACE PROCEDURE UPD\_CATEGORY  (  CID IN INTEGER,  CNAME IN NVARCHAR2  )  IS  BEGIN  IF CID IS NOT NULL AND CNAME IS NOT NULL THEN  UPDATE CATEGORIES SET CAT\_NAME = CNAME;  ELSE  RAISE NO\_DATA\_FOUND;  END IF;  EXCEPTION  WHEN NO\_DATA\_FOUND THEN  raise\_application\_error(-20001,'An error was encountered - '||SQLCODE||' -ERROR- '||SQLERRM);  WHEN TOO\_MANY\_ROWS THEN  raise\_application\_error(-20001,'An error was encountered - '||SQLCODE||' -ERROR- '||SQLERRM);  WHEN OTHERS  THEN raise\_application\_error(-20001,'An error was encountered - '||SQLCODE||' -ERROR- '||SQLERRM);  END;  -- DEL DESCRIPTION  CREATE OR REPLACE PROCEDURE DEL\_DESC  (  BID IN INTEGER  )  IS  BEGIN  IF BID IS NOT NULL THEN  DELETE FROM DESCRIPTIONS WHERE BIKE\_ID = BID;  ELSE  RAISE NO\_DATA\_FOUND;  END IF;  EXCEPTION  WHEN NO\_DATA\_FOUND THEN  raise\_application\_error(-20001,'An error was encountered - '||SQLCODE||' -ERROR- '||SQLERRM);  WHEN TOO\_MANY\_ROWS THEN  raise\_application\_error(-20001,'An error was encountered - '||SQLCODE||' -ERROR- '||SQLERRM);  WHEN OTHERS  THEN raise\_application\_error(-20001,'An error was encountered - '||SQLCODE||' -ERROR- '||SQLERRM);  END;  -- UPD DESCRIPTION  CREATE OR REPLACE PROCEDURE UPD\_DESC  (  DID IN INTEGER,  BIKE IN INTEGER,  DESCR IN NVARCHAR2,  TYPE IN NVARCHAR2,  MAT IN NVARCHAR2,  W IN FLOAT,  H IN FLOAT  )  IS  BEGIN  IF DID IS NOT NULL AND BIKE IS NOT NULL THEN  UPDATE DESCRIPTIONS SET BIKE\_DESC = DESCR , BIKE\_TYPE = TYPE , MATERIAL = MAT , WEIGHT = W , HEIGHT = H;  ELSE  RAISE NO\_DATA\_FOUND;  END IF;  EXCEPTION  WHEN NO\_DATA\_FOUND THEN  raise\_application\_error(-20001,'An error was encountered - '||SQLCODE||' -ERROR- '||SQLERRM);  WHEN TOO\_MANY\_ROWS THEN  raise\_application\_error(-20001,'An error was encountered - '||SQLCODE||' -ERROR- '||SQLERRM);  WHEN OTHERS  THEN raise\_application\_error(-20001,'An error was encountered - '||SQLCODE||' -ERROR- '||SQLERRM);  END;  -- DEL RATE  CREATE OR REPLACE PROCEDURE DEL\_RATE  (  BIKE IN INTEGER  )  IS  BEGIN  IF BIKE IS NOT NULL THEN  DELETE FROM RATINGS WHERE BIKE\_ID = BIKE;  ELSE  RAISE NO\_DATA\_FOUND;  END IF;  EXCEPTION  WHEN NO\_DATA\_FOUND THEN  raise\_application\_error(-20001,'An error was encountered - '||SQLCODE||' -ERROR- '||SQLERRM);  WHEN TOO\_MANY\_ROWS THEN  raise\_application\_error(-20001,'An error was encountered - '||SQLCODE||' -ERROR- '||SQLERRM);  WHEN OTHERS  THEN raise\_application\_error(-20001,'An error was encountered - '||SQLCODE||' -ERROR- '||SQLERRM);  END;  -- UPD RATE  CREATE OR REPLACE PROCEDURE UPD\_RATE  (  BIKE IN INTEGER,  USER IN INTEGER,  VALUE IN NUMBER,  MSG IN NVARCHAR2,  STAMP IN TIMESTAMP  )  IS  BEGIN  IF BIKE IS NOT NULL AND USER IS NOT NULL THEN  UPDATE RATINGS SET RATING\_VALUE = VALUE , RATING\_MESSAGE = MSG ,TIME\_STAMP = STAMP WHERE BIKE\_ID = BIKE AND USER\_ID = USER;  ELSE  RAISE NO\_DATA\_FOUND;  END IF;  EXCEPTION  WHEN NO\_DATA\_FOUND THEN  raise\_application\_error(-20001,'An error was encountered - '||SQLCODE||' -ERROR- '||SQLERRM);  WHEN TOO\_MANY\_ROWS THEN  raise\_application\_error(-20001,'An error was encountered - '||SQLCODE||' -ERROR- '||SQLERRM);  WHEN OTHERS  THEN raise\_application\_error(-20001,'An error was encountered - '||SQLCODE||' -ERROR- '||SQLERRM);  END; |

# Приложение В Листинг создания функций

|  |
| --- |
| -- FULL TEXT SEARCH FUNCS:  CREATE OR REPLACE FUNCTION CONTAINS\_BIKE  (  NAME IN VARCHAR2  )  RETURN BIKE\_TABLE\_RESULT  IS  B\_RESULT BIKE\_TABLE\_RESULT := BIKE\_TABLE\_RESULT();  BEGIN  SELECT BIKE\_INFO  (  B.BIKE\_ID,  B.SELLER\_ID,  B.BIKE\_NAME,  B.PRICE,  B.CATEGORY\_ID,  AVG\_RATE(B.BIKE\_ID),  D.BIKE\_DESC,  D.BIKE\_TYPE,  D.MATERIAL,  D.WEIGHT,  D.HEIGHT  )  BULK COLLECT INTO B\_RESULT  FROM BIKES B  LEFT JOIN RATINGS R ON R.BIKE\_ID = B.BIKE\_ID  LEFT JOIN DESCRIPTIONS D ON D.BIKE\_ID = B.BIKE\_ID  WHERE CONTAINS(BIKE\_NAME,NAME,1) > 0  ORDER BY B.BIKE\_ID;  RETURN B\_RESULT;  END;  SELECT \* FROM TABLE(CONTAINS\_BIKE('BIKE 1'));  ------  CREATE OR REPLACE FUNCTION CONTAINS\_USER\_BY\_NAME  (  NAME IN VARCHAR2  )  RETURN USER\_TABLE\_RESULT  IS  U\_RESULT USER\_TABLE\_RESULT := USER\_TABLE\_RESULT();  BEGIN  SELECT USER\_INFO(  U.USER\_ID,  U.USER\_NAME,  U.USER\_SURNAME,  U.USER\_FATHERNAME,  U.USER\_EMAIL,  U.PASSWORD,  U.DATE\_OF\_BIRTH,  U.ADRESS,  R.ROLENAME  )  BULK COLLECT INTO U\_RESULT  FROM USERS U  INNER JOIN ROLES R ON R.ROLE\_ID = U.ROLE\_ID  WHERE CONTAINS(USER\_NAME,NAME,1)>0  ORDER BY U.USER\_ID  ;  RETURN U\_RESULT;  END;  ------  CREATE OR REPLACE FUNCTION CONTAINS\_USER\_BY\_SURNAME  (  SURNAME IN VARCHAR2  )  RETURN USER\_TABLE\_RESULT  IS  U\_RESULT USER\_TABLE\_RESULT := USER\_TABLE\_RESULT();  BEGIN  SELECT USER\_INFO(  U.USER\_ID,  U.USER\_NAME,  U.USER\_SURNAME,  U.USER\_FATHERNAME,  U.USER\_EMAIL,  U.PASSWORD,  U.DATE\_OF\_BIRTH,  U.ADRESS,  R.ROLENAME  )  BULK COLLECT INTO U\_RESULT  FROM USERS U  INNER JOIN ROLES R ON R.ROLE\_ID = U.ROLE\_ID  WHERE CONTAINS(USER\_SURNAME,SURNAME,1)>0  ORDER BY U.USER\_ID  ;  RETURN U\_RESULT;  END;  --------  CREATE OR REPLACE FUNCTION CONTAINS\_USER\_BY\_FATHERNAME  (  FATHERNAME IN VARCHAR2  )  RETURN USER\_TABLE\_RESULT  IS  U\_RESULT USER\_TABLE\_RESULT := USER\_TABLE\_RESULT();  BEGIN  SELECT USER\_INFO(  U.USER\_ID,  U.USER\_NAME,  U.USER\_SURNAME,  U.USER\_FATHERNAME,  U.USER\_EMAIL,  U.PASSWORD,  U.DATE\_OF\_BIRTH,  U.ADRESS,  R.ROLENAME  )  BULK COLLECT INTO U\_RESULT  FROM USERS U  INNER JOIN ROLES R ON R.ROLE\_ID = U.ROLE\_ID  WHERE CONTAINS(USER\_FATHERNAME,FATHERNAME,1)>0  ORDER BY U.USER\_ID  ;  RETURN U\_RESULT;  END;  ------------------  CREATE OR REPLACE FUNCTION CONTAINS\_USER\_BY\_EMAIL  (  EMAIL IN VARCHAR2  )  RETURN USER\_TABLE\_RESULT  IS  U\_RESULT USER\_TABLE\_RESULT := USER\_TABLE\_RESULT();  BEGIN  SELECT USER\_INFO(  U.USER\_ID,  U.USER\_NAME,  U.USER\_SURNAME,  U.USER\_FATHERNAME,  U.USER\_EMAIL,  U.PASSWORD,  U.DATE\_OF\_BIRTH,  U.ADRESS,  R.ROLENAME  )  BULK COLLECT INTO U\_RESULT  FROM USERS U  INNER JOIN ROLES R ON R.ROLE\_ID = U.ROLE\_ID  WHERE CONTAINS(USER\_EMAIL,EMAIL,1)>0  ORDER BY U.USER\_ID  ;  RETURN U\_RESULT;  END;  -------------------  CREATE OR REPLACE FUNCTION CONTAINS\_USER\_BY\_ADRESS  (  ADR IN VARCHAR2  )  RETURN USER\_TABLE\_RESULT  IS  U\_RESULT USER\_TABLE\_RESULT := USER\_TABLE\_RESULT();  BEGIN  SELECT USER\_INFO(  U.USER\_ID,  U.USER\_NAME,  U.USER\_SURNAME,  U.USER\_FATHERNAME,  U.USER\_EMAIL,  U.PASSWORD,  U.DATE\_OF\_BIRTH,  U.ADRESS,  R.ROLENAME  )  BULK COLLECT INTO U\_RESULT  FROM USERS U  INNER JOIN ROLES R ON R.ROLE\_ID = U.ROLE\_ID  WHERE CONTAINS(ADRESS,ADR,1)>0  ORDER BY U.USER\_ID  ;  RETURN U\_RESULT;  END;  --------------  CREATE OR REPLACE FUNCTION CONTAINS\_RATING\_MESSAGE  (  QUR IN VARCHAR2  )  RETURN INTEGER  IS  RID RATINGS.RATING\_ID%TYPE;  BEGIN  SELECT RATING\_ID INTO RID FROM RATINGS WHERE CONTAINS(RATING\_MESSAGE, QUR, 1) > 0;  RETURN RID;  EXCEPTION  WHEN NO\_DATA\_FOUND THEN  RETURN NULL;  END;  --------- MATCHES:  CREATE OR REPLACE FUNCTION MATCH\_DESC  (  TEXT VARCHAR2  )  RETURN INTEGER  IS  DID DESCRIPTIONS.DESC\_ID%TYPE;  BEGIN  SELECT DESC\_ID INTO DID FROM DESCRIPTIONS WHERE MATCHES(BIKE\_DESC,TEXT) > 0;  RETURN DID;  END;  CREATE OR REPLACE FUNCTION MATCHES\_TYPE  (  TEXT VARCHAR2  )  RETURN INTEGER  IS  DID DESCRIPTIONS.DESC\_ID%TYPE;  BEGIN  SELECT DESC\_ID INTO DID FROM DESCRIPTIONS WHERE MATCHES(BIKE\_TYPE,TEXT) > 0;  RETURN DID;  END;  CREATE OR REPLACE FUNCTION MATCHES\_RATEMSG  (  TEXT VARCHAR2  )  RETURN INTEGER  IS  RID RATINGS.RATING\_ID%TYPE;  BEGIN  SELECT RATING\_ID INTO RID FROM RATINGS WHERE MATCHES(RATING\_MESSAGE,TEXT) > 0;  RETURN RID;  END;  -- DEFAULT FUNCTIONS:  -- AVERAGE RATING  CREATE OR REPLACE FUNCTION AVG\_RATE  (  BIKE IN INTEGER  )  RETURN DECIMAL  IS  CURSOR RATE\_CURS IS SELECT RATING\_VALUE FROM RATINGS WHERE BIKE\_ID = BIKE;  RATE RATINGS%ROWTYPE;  CNT NUMBER := 0;  SUM\_RATE NUMBER := 0;  AVG\_RATE DECIMAL;  BEGIN  FOR RATE IN RATE\_CURS  LOOP  SUM\_RATE := SUM\_RATE + RATE.RATING\_VALUE;  CNT := CNT + 1;  END LOOP;  IF CNT > 0 THEN  AVG\_RATE := SUM\_RATE / CNT;  ELSE  AVG\_RATE := 0;  END IF;  RETURN AVG\_RATE;  EXCEPTION  WHEN NO\_DATA\_FOUND THEN  RETURN 0;  WHEN TOO\_MANY\_ROWS THEN  RETURN 0;  WHEN OTHERS  THEN RETURN 0;  END;  -- ENCRYPT PASS  CREATE OR REPLACE FUNCTION HASH\_PASS  (PASSWORD IN NVARCHAR2)  RETURN NVARCHAR2  IS  BEGIN  IF PASSWORD IS NULL OR LENGTH(PASSWORD) > 20 THEN  RETURN 'FALSE';  ELSE  RETURN DBMS\_CRYPTO.HASH(UTL\_I18N.STRING\_TO\_RAW(PASSWORD,'AL32UTF8'),DBMS\_CRYPTO.HASH\_SH256);  END IF;  EXCEPTION  WHEN NO\_DATA\_FOUND THEN  RETURN 'FALSE';  WHEN TOO\_MANY\_ROWS THEN  RETURN 'FALSE';  WHEN OTHERS  THEN RETURN 'FALSE';  END;  -- REGISTER    CREATE OR REPLACE FUNCTION REGISTER  (  NAME IN NVARCHAR2 ,  SURNAME IN NVARCHAR2 ,  FATHERNAME IN NVARCHAR2,  EMAIL IN NVARCHAR2,  PASSWORD IN NVARCHAR2,  BIRTH IN DATE,  USER\_ADRESS IN NVARCHAR2,  ROLE IN INTEGER  )  RETURN INTEGER  IS  ID INTEGER;  BEGIN  INSERT INTO USERS(USER\_NAME,USER\_SURNAME,USER\_FATHERNAME,USER\_EMAIL,PASSWORD,DATE\_OF\_BIRTH,ADRESS,ROLE\_ID)  VALUES (NAME , SURNAME ,FATHERNAME,EMAIL,HASH\_PASS(PASSWORD),BIRTH,USER\_ADRESS,ROLE)  RETURNING USERS.USER\_ID INTO ID;  RETURN ID;  EXCEPTION  WHEN NO\_DATA\_FOUND THEN  RETURN 0;  WHEN TOO\_MANY\_ROWS THEN  RETURN 0;  WHEN OTHERS  THEN RETURN 0;  END;    -- LOGIN  CREATE OR REPLACE FUNCTION LOGIN  (  EMAIL IN NVARCHAR2,  PASS IN NVARCHAR2  )  RETURN INTEGER  IS  USER USERS%ROWTYPE;  BEGIN  SELECT \* INTO USER FROM USERS WHERE PASSWORD = HASH\_PASS(PASS) AND USER\_EMAIL = EMAIL;  IF PASS IS NULL  THEN RETURN 0;  ELSIF USER.USER\_ID IS NOT NULL  THEN  RETURN USER.USER\_ID;  ELSE  RETURN 0;  END IF;  EXCEPTION  WHEN NO\_DATA\_FOUND THEN  RETURN 0;  WHEN TOO\_MANY\_ROWS THEN  RETURN 0;  WHEN OTHERS  THEN RETURN 0;  END;  -- ADD BIKE  CREATE OR REPLACE FUNCTION ADD\_BIKE  (  SELLER IN INTEGER,  NAME IN NVARCHAR2,  PRICE IN DECIMAL,  CATEGORY IN INTEGER  )  RETURN INTEGER  IS  ID BIKES.BIKE\_ID%TYPE;  BEGIN  INSERT INTO BIKES(SELLER\_ID,BIKE\_NAME,PRICE,CATEGORY\_ID) VALUES(SELLER,NAME , PRICE , CATEGORY )  RETURNING BIKES.BIKE\_ID INTO ID;  RETURN ID;  END;  drop function CATSERACH\_CATEGORY;  -- ADD DESCRIPTION  CREATE OR REPLACE FUNCTION ADD\_DESC  (  BIKE IN INTEGER ,  DESCR IN NVARCHAR2,  TYPE IN NVARCHAR2,  MAT IN NVARCHAR2,  W IN FLOAT,  H IN FLOAT  )  RETURN INTEGER  IS  ID DESCRIPTIONS.DESC\_ID%TYPE;  BEGIN  INSERT INTO DESCRIPTIONS(BIKE\_ID,BIKE\_DESC,BIKE\_TYPE,MATERIAL,WEIGHT,HEIGHT) VALUES(BIKE,DESCR,TYPE,MAT,W,H)  RETURNING DESCRIPTIONS.DESC\_ID INTO ID;  RETURN ID;  EXCEPTION  WHEN NO\_DATA\_FOUND THEN  RETURN 0;  WHEN TOO\_MANY\_ROWS THEN  RETURN 0;  WHEN OTHERS  THEN RETURN 0;  END;  -- ADD CATEGORY    CREATE OR REPLACE FUNCTION ADD\_CATEGORY  (  NAME IN VARCHAR2  )  RETURN NUMBER  IS  ID CATEGORIES.CATEGORY\_ID%TYPE;  BEGIN  IF NAME IS NOT NULL  THEN  INSERT INTO CATEGORIES(CAT\_NAME)VALUES(NAME)  RETURNING CATEGORIES.CATEGORY\_ID INTO ID;  RETURN ID;  ELSE  RETURN 0;  END IF;  EXCEPTION  WHEN NO\_DATA\_FOUND THEN  RETURN 0;  WHEN TOO\_MANY\_ROWS THEN  RETURN 0;  WHEN OTHERS  THEN RETURN 0;  END;  -- ADD ROLE    CREATE OR REPLACE FUNCTION ADD\_ROLE  (  NAME IN NVARCHAR2  )  RETURN INTEGER  IS  ID ROLES.ROLE\_ID%TYPE;  BEGIN  IF NAME IS NOT NULL  THEN  INSERT INTO ROLES(ROLENAME) VALUES (NAME)  RETURNING ROLES.ROLE\_ID INTO ID;  RETURN ID;  ELSE  RETURN 0;  END IF;  EXCEPTION  WHEN NO\_DATA\_FOUND THEN  RETURN 0;  WHEN TOO\_MANY\_ROWS THEN  RETURN 0;  WHEN OTHERS  THEN RETURN 0;  END;    -- ADD ORDER  CREATE OR REPLACE FUNCTION ADD\_ORDER  (  BUYER IN INTEGER ,  ODATE IN DATE ,  STAT IN NVARCHAR2  )  RETURN INTEGER  IS  CURSOR CART\_CURS IS SELECT BIKE\_ID , QUANTITY FROM CART WHERE USER\_ID = BUYER;  ID ORDERS.ORDER\_ID%TYPE;  CART\_REC CART%ROWTYPE;  BEGIN  FOR CART\_REC IN CART\_CURS LOOP  INSERT INTO ORDERS(BUYER\_ID,BIKE\_ID,QUANTITY,ORDER\_DATE,STATUS) VALUES(BUYER,CART\_REC.BIKE\_ID,CART\_REC.QUANTITY,ODATE,STAT)  RETURNING ORDERS.ORDER\_ID INTO ID;  END LOOP;  CLEAR\_CART(BUYER);  RETURN ID;  EXCEPTION  WHEN NO\_DATA\_FOUND THEN  RETURN 0;  WHEN TOO\_MANY\_ROWS THEN  RETURN 0;  WHEN OTHERS  THEN RETURN 0;  END;  -- UPDATE USER  CREATE OR REPLACE FUNCTION UPDATE\_USER  (  UID IN INTEGER ,  NAME IN NVARCHAR2,  SURNAME IN NVARCHAR2,  FATHERNAME IN NVARCHAR2,  EMAIL IN NVARCHAR2,  PASS IN NVARCHAR2,  BIRTH IN DATE,  ADR IN NVARCHAR2,  ROLE IN INTEGER  )  RETURN INTEGER  IS  ID USERS.USER\_ID%TYPE;  BEGIN  UPDATE USERS  SET USER\_NAME = NAME ,  USER\_SURNAME = SURNAME,  USER\_FATHERNAME = FATHERNAME ,  USER\_EMAIL = EMAIL ,  PASSWORD = HASH\_PASS(PASS) ,  DATE\_OF\_BIRTH = BIRTH,  ADRESS = ADR ,  ROLE\_ID = ROLE WHERE USER\_ID = UID  RETURNING USERS.USER\_ID INTO ID;  RETURN ID;  END;    -- ADD RATE  CREATE OR REPLACE FUNCTION ADD\_RATE  (  BIKE IN INTEGER ,  USER IN INTEGER ,  VALUE IN NUMBER ,  MESSAGE IN NVARCHAR2 ,  TIMES IN TIMESTAMP  )  RETURN INTEGER  IS  ID RATINGS.RATING\_ID%TYPE;  BEGIN  IF VALUE > 10 OR VALUE < 0  THEN RETURN 0;  ELSE  INSERT INTO RATINGS(BIKE\_ID,USER\_ID,RATING\_VALUE,RATING\_MESSAGE,TIME\_STAMP) VALUES(BIKE,USER,VALUE,MESSAGE,TIMES)  RETURNING RATINGS.RATING\_ID INTO ID;  RETURN ID;  END IF;  EXCEPTION  WHEN NO\_DATA\_FOUND THEN  RETURN 0;  WHEN TOO\_MANY\_ROWS THEN  RETURN 0;  WHEN OTHERS  THEN RETURN 0;  END; |

# Приложение Г Листинг процедуры для экспорта информации

|  |
| --- |
| -- PROCEDURES THAT EXPORT\IMPORT DATA FROM TABLE TO JSON  /\*  SELECT '[' || LISTAGG(JSON\_OBJECT(  'ID' IS BIKE\_ID,  'SELLER' IS SELLER\_ID,  'NAME' IS BIKE\_NAME,  'PRICE' IS PRICE,  'CATEGORY' IS CATEGORY\_ID  ),',') || ']'  FROM BIKES  ;  \*/  CREATE OR REPLACE DIRECTORY BIKES\_JSON AS 'C:\COURSE\_WORK\_DATA';  CREATE OR REPLACE DIRECTORY USERS\_JSON AS 'C:\COURSE\_WORK\_DATA';  CREATE OR REPLACE DIRECTORY ORDERS\_JSON AS 'C:\COURSE\_WORK\_DATA';  CREATE OR REPLACE DIRECTORY DESCRIPTIONS\_JSON AS 'C:\COURSE\_WORK\_DATA';  CREATE OR REPLACE DIRECTORY RATINGS\_JSON AS 'C:\COURSE\_WORK\_DATA';  CREATE OR REPLACE DIRECTORY CART\_JSON AS 'C:\COURSE\_WORK\_DATA';  CREATE OR REPLACE DIRECTORY ROLES\_JSON AS 'C:\COURSE\_WORK\_DATA';  CREATE OR REPLACE DIRECTORY CATEGORIES\_JSON AS 'C:\COURSE\_WORK\_DATA';  -- BIKES  CREATE OR REPLACE PROCEDURE EXPORT\_BIKES\_TO\_JSON  IS  CURSOR BIKES\_ARR IS SELECT '[' || LISTAGG(JSON\_OBJECT(  'ID' IS BIKE\_ID,  'SELLER' IS SELLER\_ID,  'NAME' IS BIKE\_NAME,  'PRICE' IS PRICE,  'CATEGORY' IS CATEGORY\_ID  ),',') || ']' AS JSON\_DATA  FROM BIKES;  JS\_FILE UTL\_FILE.FILE\_TYPE;  BEGIN  JS\_FILE := UTL\_FILE.FOPEN('BIKES\_JSON','BIKES.JSON','W');  FOR BIKES\_REC IN BIKES\_ARR LOOP  UTL\_FILE.PUT\_LINE(JS\_FILE, BIKES\_REC.JSON\_DATA);  END LOOP;  UTL\_FILE.FFLUSH(JS\_FILE);  UTL\_FILE.FCLOSE(JS\_FILE);  EXCEPTION  WHEN OTHERS THEN  UTL\_FILE.FCLOSE(JS\_FILE);  RAISE;  END;  BEGIN  EXPORT\_BIKES\_TO\_JSON();  END;  -- USERS  CREATE OR REPLACE PROCEDURE EXPORT\_USERS\_TO\_JSON  IS  CURSOR BIKES\_ARR IS SELECT '[' || LISTAGG(JSON\_OBJECT(  'ID' IS USER\_ID,  'NAME' IS USER\_NAME,  'SURNAME' IS USER\_SURNAME,  'FATHERNAME' IS USER\_FATHERNAME,  'EMAIL' IS USER\_EMAIL,  'PASSWORD' IS PASSWORD,  'BIRTH' IS DATE\_OF\_BIRTH,  'ADRESS' IS ADRESS,  'ROLE' IS ROLE\_ID  ),',') || ']' AS JSON\_DATA  FROM USERS;  JS\_FILE UTL\_FILE.FILE\_TYPE;  BEGIN  JS\_FILE := UTL\_FILE.FOPEN('USERS\_JSON','USERS.JSON','W');  FOR BIKES\_REC IN BIKES\_ARR LOOP  UTL\_FILE.PUT\_LINE(JS\_FILE, BIKES\_REC.JSON\_DATA);  END LOOP;  UTL\_FILE.FFLUSH(JS\_FILE);  UTL\_FILE.FCLOSE(JS\_FILE);  EXCEPTION  WHEN OTHERS THEN  UTL\_FILE.FCLOSE(JS\_FILE);  RAISE;  END;  BEGIN  EXPORT\_USERS\_TO\_JSON();  END;  -- ORDERS  CREATE OR REPLACE PROCEDURE EXPORT\_ORDERS\_TO\_JSON  IS  CURSOR BIKES\_ARR IS SELECT '[' || LISTAGG(JSON\_OBJECT(  'ID' IS ORDER\_ID,  'BUYER' IS BUYER\_ID,  'BIKE' IS BIKE\_ID,  'QNT' IS QUANTITY,  'DATE' IS ORDER\_DATE,  'STATUS' IS STATUS  ),',') || ']' AS JSON\_DATA  FROM ORDERS;  JS\_FILE UTL\_FILE.FILE\_TYPE;  BEGIN  JS\_FILE := UTL\_FILE.FOPEN('ORDERS\_JSON','ORDERS.JSON','W');  FOR BIKES\_REC IN BIKES\_ARR LOOP  UTL\_FILE.PUT\_LINE(JS\_FILE, BIKES\_REC.JSON\_DATA);  END LOOP;  UTL\_FILE.FFLUSH(JS\_FILE);  UTL\_FILE.FCLOSE(JS\_FILE);  EXCEPTION  WHEN OTHERS THEN  UTL\_FILE.FCLOSE(JS\_FILE);  RAISE;  END;  BEGIN  EXPORT\_ORDERS\_TO\_JSON();  END;  -- DESCRIPTION  SELECT \* FROM DESCRIPTIONS;  CREATE OR REPLACE PROCEDURE EXPORT\_DESC\_TO\_JSON  IS  CURSOR BIKES\_ARR IS SELECT '[' || LISTAGG(JSON\_OBJECT(  'ID' IS DESC\_ID,  'BIKE' IS BIKE\_ID,  'TYPE' IS BIKE\_TYPE,  'MATERIAL' IS MATERIAL,  'WEIGHT' IS WEIGHT,  'HEIGHT' IS HEIGHT  ),',') || ']' AS JSON\_DATA  FROM DESCRIPTIONS;  JS\_FILE UTL\_FILE.FILE\_TYPE;  BEGIN  JS\_FILE := UTL\_FILE.FOPEN('DESCRIPTIONS\_JSON','DESCRIPTIONS.JSON','W');  FOR BIKES\_REC IN BIKES\_ARR LOOP  UTL\_FILE.PUT\_LINE(JS\_FILE, BIKES\_REC.JSON\_DATA);  END LOOP;  UTL\_FILE.FFLUSH(JS\_FILE);  UTL\_FILE.FCLOSE(JS\_FILE);  EXCEPTION  WHEN OTHERS THEN  UTL\_FILE.FCLOSE(JS\_FILE);  RAISE;  END;  BEGIN  EXPORT\_DESC\_TO\_JSON();  END;  -- CART  CREATE OR REPLACE PROCEDURE EXPORT\_CART\_TO\_JSON  IS  CURSOR BIKES\_ARR IS SELECT '[' || LISTAGG(JSON\_OBJECT(  'ID' IS CART\_ID,  'USER' IS USER\_ID,  'BIKE' IS BIKE\_ID,  'QNT' IS QUANTITY  ),',') || ']' AS JSON\_DATA  FROM CART;  JS\_FILE UTL\_FILE.FILE\_TYPE;  BEGIN  JS\_FILE := UTL\_FILE.FOPEN('CART\_JSON','CART.JSON','W');  FOR BIKES\_REC IN BIKES\_ARR LOOP  UTL\_FILE.PUT\_LINE(JS\_FILE, BIKES\_REC.JSON\_DATA);  END LOOP;  UTL\_FILE.FFLUSH(JS\_FILE);  UTL\_FILE.FCLOSE(JS\_FILE);  EXCEPTION  WHEN OTHERS THEN  UTL\_FILE.FCLOSE(JS\_FILE);  RAISE;  END;  BEGIN  EXPORT\_CART\_TO\_JSON();  END;  -- RATINGS  CREATE OR REPLACE PROCEDURE EXPORT\_RATE\_TO\_JSON  IS  CURSOR BIKES\_ARR IS SELECT '[' || LISTAGG(JSON\_OBJECT(  'ID' IS RATING\_ID,  'BIKE' IS BIKE\_ID,  'USER' IS USER\_ID,  'VALUE' IS RATING\_VALUE,  'MESSAGE' IS RATING\_MESSAGE,  'TIMESTAMP' IS TIME\_STAMP  ),',') || ']' AS JSON\_DATA  FROM RATINGS;  JS\_FILE UTL\_FILE.FILE\_TYPE;  BEGIN  JS\_FILE := UTL\_FILE.FOPEN('RATINGS\_JSON','RATINGS.JSON','W');  FOR BIKES\_REC IN BIKES\_ARR LOOP  UTL\_FILE.PUT\_LINE(JS\_FILE, BIKES\_REC.JSON\_DATA);  END LOOP;  UTL\_FILE.FFLUSH(JS\_FILE);  UTL\_FILE.FCLOSE(JS\_FILE);  EXCEPTION  WHEN OTHERS THEN  UTL\_FILE.FCLOSE(JS\_FILE);  RAISE;  END;  BEGIN  EXPORT\_RATE\_TO\_JSON();  END;  -- ROLES  CREATE OR REPLACE PROCEDURE EXPORT\_ROLE\_TO\_JSON  IS  CURSOR ROLES\_ARR IS SELECT '[' || LISTAGG(JSON\_OBJECT(  'ID' IS ROLE\_ID,  'NAME' IS ROLENAME  ),',') || ']' AS JSON\_DATA  FROM ROLES;  JS\_FILE UTL\_FILE.FILE\_TYPE;  BEGIN  JS\_FILE := UTL\_FILE.FOPEN('ROLES\_JSON','ROLES.JSON','W');  FOR ROLES\_REC IN ROLES\_ARR LOOP  UTL\_FILE.PUT\_LINE(JS\_FILE, ROLES\_REC.JSON\_DATA);  END LOOP;  UTL\_FILE.FFLUSH(JS\_FILE);  UTL\_FILE.FCLOSE(JS\_FILE);  EXCEPTION  WHEN OTHERS THEN  UTL\_FILE.FCLOSE(JS\_FILE);  RAISE;  END;  BEGIN  EXPORT\_ROLE\_TO\_JSON();  END;  -- CATEGORIES  CREATE OR REPLACE PROCEDURE EXPORT\_CATEGORY\_TO\_JSON  IS  CURSOR BIKES\_ARR IS SELECT '[' || LISTAGG(JSON\_OBJECT(  'ID' IS CATEGORY\_ID,  'NAME' IS CAT\_NAME  ),',') || ']' AS JSON\_DATA  FROM CATEGORIES;  JS\_FILE UTL\_FILE.FILE\_TYPE;  BEGIN  JS\_FILE:= UTL\_FILE.FOPEN('CATEGORIES\_JSON','CATEGORIES.JSON','W');  FOR BIKES\_REC IN BIKES\_ARR LOOP  UTL\_FILE.PUT\_LINE(JS\_FILE, BIKES\_REC.JSON\_DATA);  END LOOP;  UTL\_FILE.FFLUSH(JS\_FILE);  UTL\_FILE.FCLOSE(JS\_FILE);  EXCEPTION  WHEN OTHERS THEN  UTL\_FILE.FCLOSE(JS\_FILE);  RAISE;  END;  BEGIN  EXPORT\_CATEGORY\_TO\_JSON();  END; |

# Приложение Д Создание триггеров

|  |
| --- |
| -- TRIGGERS:  -- USER BIRTH  CREATE OR REPLACE TRIGGER USER\_BIRTH\_DATE  BEFORE INSERT OR UPDATE ON USERS FOR EACH ROW  DECLARE  USER\_BIRTH EXCEPTION;  BEGIN  IF :NEW.DATE\_OF\_BIRTH IS NOT NULL AND :NEW.DATE\_OF\_BIRTH > SYSDATE  THEN RAISE USER\_BIRTH;  END IF;  EXCEPTION  WHEN USER\_BIRTH  THEN RAISE\_APPLICATION\_ERROR(-20001, 'Дата рождения не может быть в будущем.');  END;  -- USERS EMAIL  CREATE OR REPLACE TRIGGER CHECK\_EMAIL  BEFORE INSERT OR UPDATE ON USERS  FOR EACH ROW  DECLARE  EMAIL\_EXISTS EXCEPTION;  EMAIL\_COUNT NUMBER;  BEGIN  IF :NEW.USER\_EMAIL IS NOT NULL THEN  IF :NEW.USER\_EMAIL <> :OLD.USER\_EMAIL THEN  SELECT COUNT(\*) INTO EMAIL\_COUNT FROM USERS WHERE USER\_EMAIL = :NEW.USER\_EMAIL;  IF EMAIL\_COUNT > 0 THEN  RAISE EMAIL\_EXISTS;  END IF;  END IF;  END IF;  EXCEPTION  WHEN EMAIL\_EXISTS THEN  RAISE\_APPLICATION\_ERROR(-20001, 'Пользователь с такой же электронной почтой уже существует.');  END;  -- DESC BIKES  CREATE OR REPLACE TRIGGER CHECK\_BIKE\_IN\_DESC  BEFORE INSERT ON DESCRIPTIONS  FOR EACH ROW  DECLARE  BIKE\_EXCIST EXCEPTION;  BIKE\_COUNT NUMBER;  BEGIN  IF :NEW.BIKE\_ID IS NOT NULL THEN  SELECT COUNT(\*) INTO BIKE\_COUNT FROM DESCRIPTIONS WHERE BIKE\_ID = :NEW.BIKE\_ID;  IF BIKE\_COUNT > 0 THEN  RAISE BIKE\_EXCIST;  END IF;  END IF;  EXCEPTION  WHEN BIKE\_EXCIST  THEN RAISE\_APPLICATION\_ERROR(-20001, 'Велосипед уже имеет описание.');  END;  -- USER RATE  CREATE OR REPLACE TRIGGER CHECK\_BIKE\_USER\_RATE  BEFORE INSERT ON RATINGS  FOR EACH ROW  DECLARE  USER\_EXIST EXCEPTION;  RATE\_COUNT NUMBER;  BEGIN  IF :NEW.USER\_ID IS NOT NULL AND :NEW.BIKE\_ID IS NOT NULL THEN  SELECT COUNT(\*) INTO RATE\_COUNT FROM RATINGS WHERE USER\_ID = :NEW.USER\_ID AND BIKE\_ID = :NEW.BIKE\_ID;  IF RATE\_COUNT > 0 THEN  RAISE USER\_EXIST;  END IF;  END IF;  EXCEPTION  WHEN USER\_EXIST  THEN RAISE\_APPLICATION\_ERROR(-20001, 'Велосипед уже имеет описание.');  END;  -- ORDER DATE  CREATE OR REPLACE TRIGGER ORDER\_DATE\_CHECK  BEFORE INSERT OR UPDATE ON ORDERS  FOR EACH ROW  DECLARE  DATE\_EX EXCEPTION;  BEGIN  IF :NEW.ORDER\_DATE IS NOT NULL AND :NEW.ORDER\_DATE > SYSDATE THEN  RAISE DATE\_EX;  END IF;  EXCEPTION  WHEN DATE\_EX  THEN RAISE\_APPLICATION\_ERROR(-20001, 'Дата заказа не может быть в будущем.');  END;  -- CHECK RATE TIMESTAMP  CREATE OR REPLACE TRIGGER CHECK\_RATE\_DATE  BEFORE INSERT OR UPDATE ON RATINGS  FOR EACH ROW  DECLARE  RATE\_EX EXCEPTION;  BEGIN  IF :NEW.TIME\_STAMP IS NOT NULL AND :NEW.TIME\_STAMP > SYSTIMESTAMP THEN  RAISE RATE\_EX;  END IF;  EXCEPTION  WHEN RATE\_EX THEN  RAISE\_APPLICATION\_ERROR(-20001, 'Время рейтинга не может быть в будущем.');  END;  -- CHECK RATING RANGE  CREATE OR REPLACE TRIGGER CHECK\_RATE\_VALUE  BEFORE INSERT OR UPDATE ON RATINGS  FOR EACH ROW  DECLARE  RATE\_EX EXCEPTION;  BEGIN  IF :NEW.RATING\_VALUE IS NOT NULL THEN  IF :NEW.RATING\_VALUE > 10 OR :NEW.RATING\_VALUE < 0 THEN  RAISE RATE\_EX;  END IF;  END IF;  EXCEPTION  WHEN RATE\_EX THEN  RAISE\_APPLICATION\_ERROR(-20001, 'Рейтинг не может быть больше 10 и меньше 0.');  END;  -- CHECK DESCRIPTION MATERIAL  CREATE OR REPLACE TRIGGER CHECK\_DESC\_MATERIAL  BEFORE INSERT OR UPDATE ON DESCRIPTIONS  FOR EACH ROW  DECLARE  DESC\_EX EXCEPTION;  BEGIN  IF :NEW.MATERIAL IS NOT NULL THEN  IF :NEW.MATERIAL != 'STEEL' AND :NEW.MATERIAL != 'TIN' AND :NEW.MATERIAL != 'LEAD' AND :NEW.MATERIAL != 'СТАЛЬ' AND :NEW.MATERIAL != 'ОЛОВО' AND :NEW.MATERIAL != 'СВИНЕЦ' THEN  RAISE DESC\_EX;  END IF;  END IF;  EXCEPTION  WHEN DESC\_EX THEN  RAISE\_APPLICATION\_ERROR(-20001, 'Материала не существует.');  END;  -- CHECK DESCRIPTION TYPE  CREATE OR REPLACE TRIGGER CHECK\_DESC\_TYPE  BEFORE INSERT OR UPDATE ON DESCRIPTIONS  FOR EACH ROW  DECLARE  DESC\_EX EXCEPTION;  BEGIN  IF :NEW.BIKE\_TYPE IS NOT NULL THEN  IF :NEW.BIKE\_TYPE != 'STREET' AND :NEW.BIKE\_TYPE != 'RACING' AND :NEW.BIKE\_TYPE != 'DIRT' AND :NEW.BIKE\_TYPE != 'УЛИЧНЫЙ' AND :NEW.BIKE\_TYPE != 'ГОНОЧНЫЙ' AND :NEW.BIKE\_TYPE != 'ВНЕДОРОЖНЫЙ' THEN  RAISE DESC\_EX;  END IF;  END IF;  EXCEPTION  WHEN DESC\_EX THEN  RAISE\_APPLICATION\_ERROR(-20001, 'Типа не существует.');  END;  -- CHECK DESCRIPTION HEIGHT WEIGHT  CREATE OR REPLACE TRIGGER CHECK\_DESC\_HEIGTH\_WEIGHT  BEFORE INSERT OR UPDATE ON DESCRIPTIONS  FOR EACH ROW  DECLARE  DESC\_EX EXCEPTION;  BEGIN  IF :NEW.HEIGHT IS NOT NULL AND :NEW.WEIGHT IS NOT NULL THEN  IF :NEW.HEIGHT > 21 OR :NEW.HEIGHT < 19 OR :NEW.WEIGHT > 14 OR :NEW.WEIGHT < 7 THEN  RAISE DESC\_EX;  END IF;  END IF;  EXCEPTION  WHEN DESC\_EX THEN  RAISE\_APPLICATION\_ERROR(-20001, 'Масса/высота не корректна.');  END;  -- CART QNT  CREATE OR REPLACE TRIGGER CHECK\_CART\_QNT  BEFORE INSERT OR UPDATE ON CART  FOR EACH ROW  DECLARE  CART\_EX EXCEPTION;  BEGIN  IF :NEW.QUANTITY IS NOT NULL THEN  IF :NEW.QUANTITY > 100 OR :NEW.QUANTITY < 0 THEN  RAISE CART\_EX;  END IF;  END IF;  EXCEPTION  WHEN CART\_EX THEN  RAISE\_APPLICATION\_ERROR(-20001, 'Нарушен ллимит колличества.');  END;  -- CATEGORY CHECK  CREATE OR REPLACE TRIGGER CHECK\_CATEGORY  BEFORE INSERT OR UPDATE ON CATEGORIES  FOR EACH ROW  DECLARE  CAT\_EX EXCEPTION;  BEGIN  IF :NEW.CAT\_NAME IS NOT NULL THEN  IF :NEW.CAT\_NAME != 'BMX' AND :NEW.CAT\_NAME != 'MTB' AND :NEW.CAT\_NAME != 'KID' AND :NEW.CAT\_NAME != 'PROFESSIONAL' AND :NEW.CAT\_NAME != 'DEFAULT' THEN  RAISE CAT\_EX;  END IF;  END IF;  EXCEPTION  WHEN CAT\_EX THEN  RAISE\_APPLICATION\_ERROR(-20001, 'Такой категории не существует.');  END;  -- ORDER SATTUS CHECK  CREATE OR REPLACE TRIGGER CHECK\_ORDER\_STATUS  BEFORE INSERT OR UPDATE ON ORDERS  FOR EACH ROW  DECLARE  ORD\_EX EXCEPTION;  BEGIN  IF :NEW.STATUS IS NOT NULL THEN  IF :NEW.STATUS != 'IN WAY' AND :NEW.STATUS != 'ARRIVED' AND :NEW.STATUS != 'COMPLETED' AND :NEW.STATUS != 'AND ASSEMBLY' AND :NEW.STATUS != 'В ПУТИ'  AND :NEW.STATUS != 'ПРИБЫЛ' AND :NEW.STATUS != 'ЗАВЕРШЁН' AND :NEW.STATUS != 'НА СБОРКЕ' THEN  RAISE ORD\_EX;  END IF;  END IF;  EXCEPTION  WHEN ORD\_EX THEN  RAISE\_APPLICATION\_ERROR(-20001, 'Такого статуса не существует.');  END;  -- CHECK ROLE NAME  CREATE OR REPLACE TRIGGER CHECK\_ROLE\_NAME  BEFORE INSERT OR UPDATE ON ROLES  FOR EACH ROW  DECLARE  ROLE\_EX EXCEPTION;  BEGIN  IF :NEW.ROLENAME IS NOT NULL THEN  IF :NEW.ROLENAME != 'USER' AND :NEW.ROLENAME != 'MANAGER' THEN  RAISE ROLE\_EX;  END IF;  END IF;  EXCEPTION  WHEN ROLE\_EX THEN  RAISE\_APPLICATION\_ERROR(-20001, 'Такой роли не существует.');  END;  -- CHECK CATEGORIES AGAIN  CREATE OR REPLACE TRIGGER CHECK\_CAT\_INSERT  BEFORE INSERT OR UPDATE ON CATEGORIES  FOR EACH ROW  DECLARE  CAT\_COUNT NUMBER;  CAT\_EX EXCEPTION;  BEGIN  IF :NEW.CAT\_NAME IS NOT NULL THEN  SELECT COUNT(\*) INTO CAT\_COUNT FROM CATEGORIES WHERE CAT\_NAME = :NEW.CAT\_NAME;  IF CAT\_COUNT > 0 THEN  RAISE CAT\_EX;  END IF;  END IF;  EXCEPTION  WHEN CAT\_EX THEN  RAISE\_APPLICATION\_ERROR(-20001, 'Такая категория существует.');  END;  -- CHECK ROLE AGAIN    CREATE OR REPLACE TRIGGER CHECK\_ROLE\_INSERT  BEFORE INSERT OR UPDATE ON ROLES  FOR EACH ROW  DECLARE  ROLE\_COUNT NUMBER;  ROLE\_EX EXCEPTION;  BEGIN  IF :NEW.ROLENAME IS NOT NULL THEN  SELECT COUNT(\*) INTO ROLE\_COUNT FROM ROLES WHERE ROLENAME = :NEW.ROLENAME;  IF ROLE\_COUNT > 0 THEN  RAISE ROLE\_EX;  END IF;  END IF;  EXCEPTION  WHEN ROLE\_EX THEN  RAISE\_APPLICATION\_ERROR(-20001, 'Такая роль существует.');  END;  -- BIKE PRICE  CREATE OR REPLACE TRIGGER CHECK\_BIKE\_PRICE  BEFORE INSERT OR UPDATE ON BIKES  FOR EACH ROW  DECLARE  BIKE\_EX EXCEPTION;  BEGIN  IF :NEW.PRICE IS NOT NULL THEN  IF :NEW.PRICE < 0 AND :NEW.PRICE > 10000000 THEN  RAISE BIKE\_EX;  END IF;  END IF;  EXCEPTION  WHEN BIKE\_EX THEN  RAISE\_APPLICATION\_ERROR(-20001, 'Цена неправильного формата.');  END; |