МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

Учреждение образования «БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

Факультет Информационных технологий
Кафедра Информационные системы и технологии
Специальность 6-05-0612-01 Программная инженерия (профилизация Программное обеспечение информационных технологий)

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА К КУРСОВОЙ РАБОТЕ НА ТЕМУ:

«Реализация базы данных кинотеатра с использованием мультимедийных типов данных»

Выполнил студент Мамонько Денис Александрович
(Ф.И.О.)

Руководитель работы ассистент Подрез А.А
(учен. степень, звание, должность, Ф.И.О., подпись)

Зав. кафедрой к.т.н. Блинова Е.А.
(учен. степень, звание, должность, Ф.И.О., подпись)

Курсовая работа защищена с оценкой

Минск 2025

Оглавление

Введение

В современном мире многие люди на регулярной основе посещают культурно-развлекательные мероприятия, в частности кинотеатры. Однако потенциальные зрители часто сталкиваются с проблемой заблаговременного выбора и покупки билетов, что может привести к ситуации, когда к моменту прибытия в кинотеатр все места на желаемый сеанс уже распроданы. Кроме того, многие ценители кино сталкиваются с проблемой просмотра на большом экране фильмов прошлых лет, так как в репертуаре современных кинотеатров такая возможность практически отсутствует. Эффективное решение этих проблем требует создания централизованной системы для управления всеми процессами кинотеатра.

Целью данной курсовой работы является разработка реляционной базы данных «Кинотеатр». Эта база данных предназначена для автоматизации процессов учёта и продажи билетов, а также для хранения и управления информацией о фильмах, расписании сеансов, клиентах и залах.

База данных — это организованная структура, предназначенная для хранения, изменения и обработки взаимосвязанной информации. Реляционная база данных — это база данных, основанная на реляционной модели, которая представляет данные в виде таблиц. В качестве системы управления базами данных (СУБД) был выбран Oracle. Этот выбор обусловлен его высокой производительностью, надежностью, а также широкими возможностями для работы с мультимедийными типами данных, что является ключевым аспектом данного проекта.

В основной части пояснительной записки будут рассмотрены все этапы проектирования базы данных: от концептуальной модели до физической реализации. Также будут обоснованы ключевые технические решения, принятые в ходе разработки. Особое внимание уделено использованию мультимедийных типов данных для хранения сопутствующих материалов, таких как постеры к фильмам (графические файлы) и трейлеры (видеофайлы), что обогащает информационное наполнение базы данных.

Для проектируемой базы данных определены следующие основные задачи, которые она должна решать:

– Хранение полной информации о фильмах (название, жанр, режиссер, возрастное ограничение, постер, трейлер, описание фильма).

– Ведение расписания киносеансов с привязкой к конкретному фильму, залу и времени.

– Учёт информации о кинозалах (название, вместимость, тип зала).

– Регистрация и управление данными о клиентах кинотеатра.

– Просмотр статуса заказа.

– Обработка операций по покупке билетов с фиксацией конкретного места за клиентом.

Главная задача данного курсового проекта — это разработка логической и физической структуры базы данных, которая обеспечивает эффективное хранение и управление всей необходимой информацией для функционирования современного кинотеатра и решает поставленные задачи.

1 Постановка задачи

Задачей курсового проектирования стала разработка базы данных для автоматизации деятельности современного кинотеатра. В современных условиях эффективное управление процессами продажи билетов, составления расписания и учета контента требует наличия централизованной и надежной информационной системы, ядром которой является база данных.

С теоретической точки зрения, постановка задачи включает изучение бизнес-процессов кинотеатра, их логическое представление в реляционной модели и выбор оптимальной структуры базы данных для последующей обработки запросов. При проектировании особое внимание уделено вопросам нормализации для исключения избыточности данных, обеспечения их целостности и согласованности с помощью ограничений и внешних ключей. Ключевым аспектом проекта является использование мультимедийных типов данных (BLOB) для хранения сопутствующих материалов, таких как постеры и трейлеры к фильмам. Кроме того, важной частью работы стала реализация ролевой модели доступа, позволяющей разграничить права пользователей (Пользователь, Менеджер, Администратор) и обеспечить безопасную работу с системой на уровне самой СУБД.

Исходя из поставленной цели, разработанная база данных должна обеспечивать выполнение следующих функций посредством хранимых процедур и пакетов:

– управление каталогом фильмов;
– администрирование пользователей;
– управление расписанием;
– обработка заказов;
– получение отчетности.

Решение данных задач позволило создать функциональное ядро для реляционной базы данных "Кинотеатр", а также послужило практическим примером применения теоретических положений по проектированию, администрированию и реализации бизнес-логики на стороне базы данных с использованием PL/SQL.

Диаграмма вариантов использования представлена в приложении А.

1.1 Анализ конкурентов

В данном разделе для определения функциональных требований к базе данных был проведен анализ аналогов. В качестве таких аналогов были рассмотрены веб-сайты и информационные системы действующих кинотеатров, которые реализуют схожие бизнес-задачи: продажу билетов, формирование расписания и предоставление каталога фильмов. Анализ их функционала позволил выявить основные сущности и операции, которые легли в основу проектирования структуры и программного интерфейса нашей базы данных.

1.1.1 Анализ сайта Кинопоиск.ru

Сайт «Кинопоиск» [1] является одним из самых популярных интернет-

ресурсов в СНГ, собравшим полноценную информацию о мировой киноиндустрии. Главными преимуществами данного ресурса является его наполненность информацией и постоянное её обновление. Пользователь данного ресурса имеет возможность детально и подробно ознакомиться с каждым фильмом. Ещё одним немаловажным достоинством данного программного средства является возможность пользователей ознакомиться с рецензиями как обычных пользователей, так и профессиональных критиков, а также оставить свою рецензию. Данное программное средство предоставляет пользователю возможность выбрать фильм по определенным критериям и составить полноценное мнение. Также, помимо теоретического ознакомления с информацией о фильмах, пользователь имеет возможность узнать, в каких кинотеатрах города в определенный день показывается интересующий его фильм. Интерфейс интернет-ресурса представлен на рисунке 1.1.

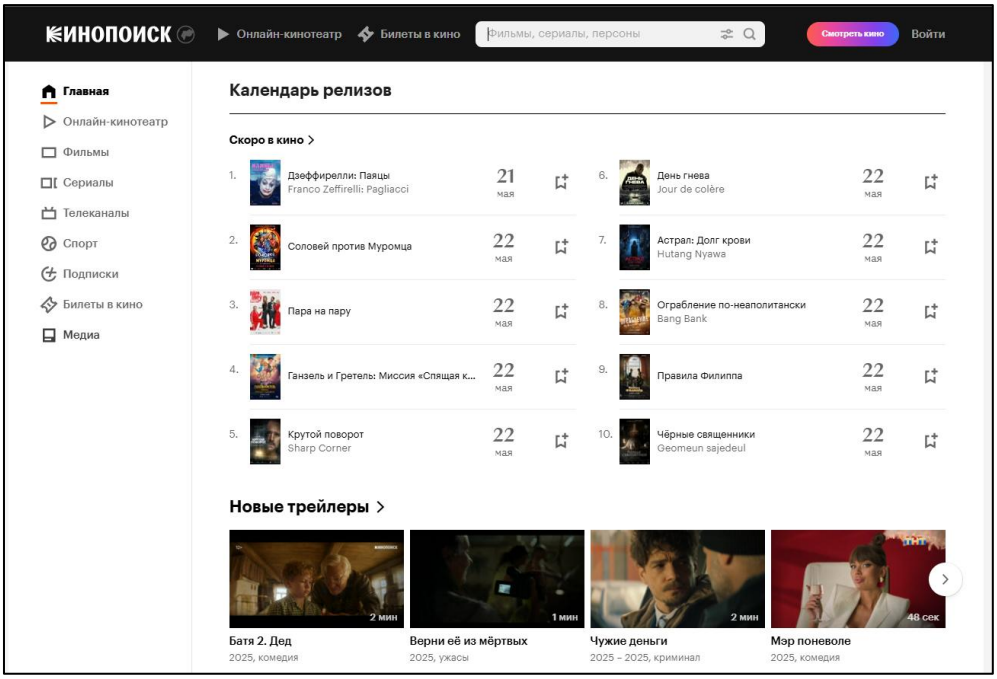


Рисунок 1.1 – Интерфейс Кинопоиск

Из выявленных недостатков данного ресурса необходимо выделить невозможность забронировать фиксированное количество билетов на реальный кинопоказ в кинотеатре.

1.1.2 Анализ сайта skyline.by

Skyline [2] – это современный кинотеатр, который сочетает в себе высокотехнологичные кинопоказы с продуманным комфортом и дополнительными сервисами, делая посещение кино особым событием.

Интерфейс данного сайта чистый, интуитивно понятный и с удобной навигацией. Есть возможность просматривать афишу на ближайшие дни, а также предварительный выбор будущих премьер. Доступна фильтрация по жанрам, форматам (2D, 3D) и специальным категориям. Интерфейс данного сайта

представлен на рисунке 1.2.

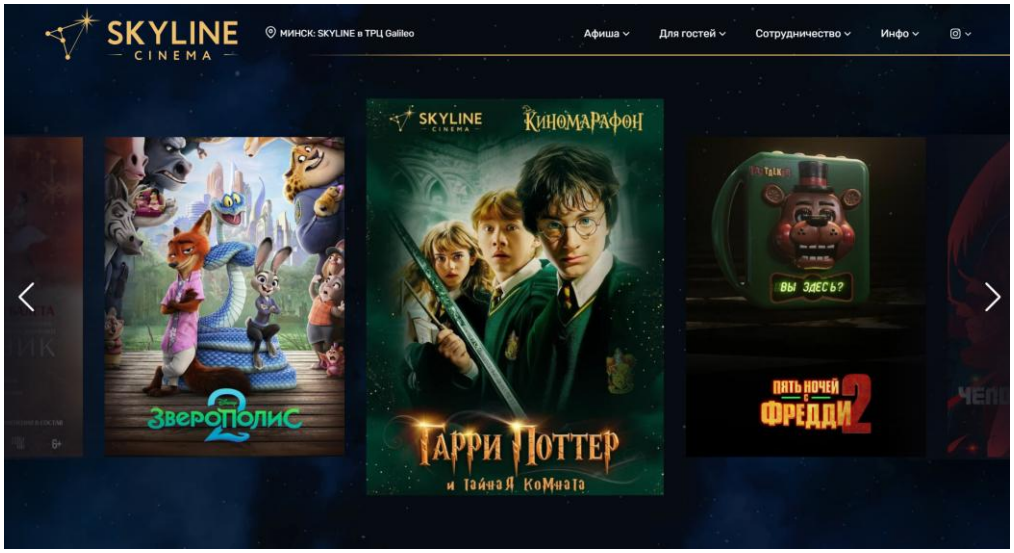


Рисунок 1.2 – Интерфейс skyline

Данный сайт позволяет также бронировать билеты на реальные сеансы с фиксированным количеством мест в отличие от предыдущего аналога. После выбора даты и зала, пользователю будет выгружена схема соответствующего зала, где он может выбрать место и завершить бронирование.

1.2 Обзор средства разработки

В качестве средства разработки базы данных в курсовом проекте была выбрана система управления базами данных Oracle Database. Данная СУБД относится к классу промышленных реляционных СУБД и широко применяется в корпоративных информационных системах благодаря высокой надежности, масштабируемости и богатому набору встроенных средств для администрирования и обеспечения безопасности.

Выбор Oracle Database для проекта "Кинотеатр" обусловлен ее ключевыми возможностями, которые наилучшим образом соответствуют поставленным задачам. Важнейшим фактором является расширенная поддержка мультимедийных типов данных, что напрямую соответствует теме курсовой работы. Oracle предоставляет специализированный тип данных BLOB и мощный встроенный пакет DBMS_LOB, которые позволят эффективно и безопасно управлять хранением постеров и трейлеров непосредственно в базе данных.

Кроме того, СУБД обладает развитыми процедурными возможностями благодаря языку PL/SQL. Это позволит инкапсулировать всю бизнес-логику на стороне сервера, реализовав функциональность в виде хранимых процедур и функций. Для повышения модульности и управляемости кода эти подпрограммы планируется сгруппировать в логические пакеты, соответствующие ролям (CLIENT_PKG, MANAGER_PKG, ADMIN_PKG).

Гибкая модель безопасности Oracle дает возможность реализовать требуемую трехуровневую ролевую модель. Путем создания отдельных пользователей СУБД и

выдачи им гранулированных привилегий EXECUTE на пакеты можно будет обеспечить строгий контроль доступа к данным. Архитектурно проект предполагается развернуть в изолированной Pluggable Database (PDB), что является современным стандартом для Oracle и упрощает администрирование.

Таким образом, использование Oracle в проекте позволит не только реализовать весь требуемый функционал, но и получить практические навыки работы с промышленной СУБД: от проектирования схемы до настройки ролевой модели и работы со специализированными типами данных. Это повышает практическую значимость работы и приближает условия разработки к требованиям реальных корпоративных систем.

1.3 Выводы

В рамках данного раздела была проведена постановка задачи курсового проектирования, определены основные требования к разрабатываемой базе данных и выполнен анализ существующих решений в области автоматизации кинотеатров.

Анализ предметной области показал, что, хотя существует множество систем онлайн-бронирования билетов, реализация комплексной внутренней базы данных требует особого подхода. Большинство публичных сервисов сосредоточены на пользовательском интерфейсе для покупки билетов, в то время как внутренняя логика управления контентом, расписаниями и ролями персонала остается скрытой. Ни одно из общедоступных решений не демонстрирует в полной мере реализацию хранения мультимедийных данных (постеров, трейлеров) и строгой ролевой модели доступа на уровне самой СУБД. Это подтверждает актуальность разработки собственной архитектуры базы данных, которая объединит функции учета, администрирования и обработки заказов в единой защищенной системе.

Выбор Oracle Database в качестве СУБД обоснован несколькими ключевыми факторами. Во-первых, Oracle предоставляет мощные и надежные инструменты для работы с мультимедийными типами данных (BLOB) через системные пакеты, что является центральным технологическим требованием данного проекта. Во-вторых, СУБД Oracle обладает развитой системой безопасности, позволяющей реализовать сложную ролевую модель доступа и разграничить права на выполнение операций на уровне самой базы данных. Наконец, использование промышленной СУБД позволяет получить практические навыки работы с корпоративными системами управления данными, что повышает образовательную ценность выполненной работы.

Таким образом, определены все необходимые теоретические и практические основы для последующего проектирования логической и физической структуры базы данных и реализации ее бизнес-логики средствами PL/SQL.

2 Проектирование базы данных

В этом разделе описаны концептуальная и логическая модели базы данных, разработанные для информационной системы "Кинотеатр". При построении концептуальной модели были подробно проанализированы ключевые бизнес-процессы кинотеатра, такие как продажа билетов, формирование репертуара и управление залами. На основе этого анализа были определены основные объекты предметной области и характер их взаимодействия.

В результате были выделены ключевые сущности концептуальной модели, которые включают в себя:

– роли: определяет уровни доступа в системе (пользователь, менеджер, администратор).

– пользователи**:** содержит информацию о зарегистрированных пользователях системы и их ролях.

– фильмы: хранит всю атрибутивную и мультимедийную информацию о кинокартинах.

– залы: описывает кинозалы, их вместимость и характеристики.

– расписание: связывает конкретный фильм, зал и время показа.

– заказы: фиксирует факт покупки билетов на сеанс конкретным пользователем.

Логическая модель базы данных была построена на основании концептуальной модели и формализована в виде набора конкретных таблиц с четко определёнными связями между ними. В результате сформирована структура базы данных, включающая следующие шесть таблиц:

– таблица Roles является справочником ролей пользователей, определяющим их уровень привилегий в системе;

– таблица Users содержит сведения о зарегистрированных пользователях системы (имя, фамилия, email, пароль и ссылка на роль);

– таблица Movies хранит расширенную информацию о фильмах, включая название, режиссера, описание, жанр, длительность, рейтинг, возрастное ограничение, а также поля Poster и Trailer типа BLOB для хранения мультимедийных данных;

– таблица Halls содержит данные о залах кинотеатра, такие как название, вместимость и тип (например, 2D, 3D, IMAX);

– таблица Schedules предназначена для хранения расписания сеансов, связывая фильм (MovieId) с залом (HallId) и временем начала показа (ShowTime);

– таблица Orders фиксирует информацию о каждом заказе: купленные места (Seats), итоговую стоимость (TotalPrice), статус, а также содержит ссылки на пользователя, совершившего покупку (UserId), и на конкретный сеанс (ScheduleId).

Между таблицами установлены следующие связи:

– таблица Users связана с таблицей Roles через внешний ключ RoleId, реализуя связь «один ко многим» (одна роль может быть присвоена множеству пользователей);

– таблица Schedules имеет две связи «один ко многим»: с таблицей Movies через MovieId и с таблицей Halls через HallId;

– таблица Orders связана с Users через UserId и с Schedules через ScheduleId, также реализуя связи «один ко многим»;

Все внешние ключи содержат опцию ON DELETE CASCADE, что обеспечивает автоматическое удаление зависимых записей при удалении родительской. Например, при удалении фильма из таблицы Movies будут автоматически удалены все связанные с ним сеансы из Schedules, а затем и все заказы на эти сеансы из Orders, что поддерживает целостность и актуальность данных.

При проектировании структуры были применены принципы нормализации для минимизации избыточности данных. Использование автогенерируемых первичных ключей (GENERATED BY DEFAULT AS IDENTITY) обеспечивает уникальность каждой записи. Ограничения UNIQUE на ключевых полях, таких как Users.Email и Movies.Title, предотвращают создание дубликатов.

В результате спроектирована логическая модель базы данных, включающая шесть взаимосвязанных таблиц, которые обеспечивают полноценное хранение информации для автоматизации работы кинотеатра. Установленные связи и каскадные правила гарантируют поддержание целостности данных при операциях изменения и удаления. Разработанная структура является гибкой и масштабируемой, готовой для реализации бизнес-логики на уровне хранимых процедур и пакетов PL/SQL.

Схема базы данных, спроектированной в данной главе, представлена в приложении Б.

3 Разработка объектов базы данных

3.1 Таблицы базы данных

В структуру базы данных входят следующие шесть таблиц: Roles, Users, Movies, Halls, Schedules, Orders. Далее представлены таблицы с описанием полей и ограничений для каждой из них.

Таблица Roles используется как справочник для хранения возможных ролей пользователей в системе, таких как "Клиент", "Менеджер" и "Администратор". Структура таблицы Roles представлена в таблице 3.1.

Таблица 3.1 – Структура таблицы Roles

Столбец	**Тип данных**	**Ограничение целостности**	**Назначение**
Id	NUMBER	Primary key, Generated by default as identity	Уникальный идентификатор роли
Name	NVARCHAR2(50)	Unique, Not null	Название роли

Таблица Users хранит в себе все данные о зарегистрированных пользователях приложения, включая их персональную информацию, учетные данные для входа и присвоенную роль. Структура таблицы Users представлена в таблице 3.2.

Таблица 3.2 – Структура таблицы Users

Столбец	**Тип данных**	**Ограничение целостности**	**Назначение**
Id	NUMBER	Primary key, Generated by default as identity	Уникальный идентификатор пользователя
Name	NVARCHAR2(100)	Not null	Имя пользователя
Surname	NVARCHAR2(100)	Not null	Фамилия пользователя
Email	NVARCHAR2(255)	Unique, Not null	Email пользователя, используется как логин
Password	NVARCHAR2(100)	Not null	Пароль для аутенфикации
RoleId	NUMBER	Not null, Foreign key (Roles.id)	Идентификатор роли пользователя

Таблица Movies является центральным каталогом фильмов в системе. Она спроектирована для хранения всей ключевой атрибутивной информации о кинокартинах, необходимой как для отображения пользователям, так и для работы внутренней логики, например, для проверки возрастных ограничений или расчета длительности сеансов. Кроме того, таблица включает поля для мультимедийных

данных, что является основным технологическим аспектом данного проекта. Структура таблицы Movies представлена в таблице 3.3.

Таблица 3.3 – Структура таблицы Movies

Столбец	**Тип данных**	**Ограничение целостности**	**Назначение**
Id	NUMBER	Primary key, Generated by default as identity	Уникальный идентификатор фильма
Title	NVARCHAR2(255)	Unique, Not null	Название фильма
Director	NVARCHAR2(255)	-	Режиссер фильма
Description	NCLOB	-	Полное описание фильма
Genre	NVARCHAR2(100)	-	Жанр фильма
Duration	NUMBER	-	Длительность фильмав минутах
Rating	NUMBER(3,1)	-	Рейтинг фильма (например, 8.5)
AgeRestriction	NVARCHAR2(10)	-	Возрастное ограничение (например, 16+)
Poster	BLOB	-	Бинарные данные постера (изображение)
Trailer	BLOB	-	Бинарные данные трейлера (видео)

Таблица Halls хранит информацию о залах, доступных в кинотеатре. Структура таблицы Halls представлена в таблице 3.4.

Таблица 3.4 – Структура таблицы Halls

Столбец	**Тип данных**	**Ограничение целостности**	**Назначение**
Id	NUMBER	Primary key, Generated by default as identity	Уникальный идентификатор зала
Name	NVARCHAR2(100)	Unique, Not null	Название зала

Продолжение таблицы 3.4.

Столбец	**Тип данных**	**Ограничение целостности**	**Назначение**
Capacity	NUMBER	Not null	Вместимость зала
Type	NVARCHAR2(50)	-	Тип зала (2D, 3D)

Таблица Schedules используется для формирования расписания, связывая фильм, зал и время показа. Структура таблицы Schedules представлена в таблице 3.5.

Таблица 3.5 – Структура таблицы Schedules

Столбец	**Тип данных**	**Ограничение целостности**	**Назначение**
Id	NUMBER	Primary key, Generated by default as identity	Уникальный идентификатор сеанса
ShowTime	TIMESTAMP	Not null	Точная дата и время начала сеанса
MovieId	NUMBER	Not null, Foreign key (Movies.Id)	Идентификатор фильма
HallId	NUMBER	Not null, Foreign key (Halls.Id)	Идентификатор зала

Таблица Orders хранит историю всех заказов, сделанных пользователями. Структура таблицы Orders представлена в таблице 3.6.

Таблица 3.6 – Структура таблицы Orders

Столбец	**Тип данных**	**Ограничение целостности**	**Назначение**
Id	NUMBER	Primary key, Generated by default as identity	Уникальный идентификатор заказа
Seats	NVARCHAR2(255)	Not null	Список забронированных мест
TotalPrice	NUMBER(10, 2)	Not null	Итоговая стоимость заказа
OrderStatus	NVARCHAR2(50)	Not null, Default 'Забронирован'	Статус заказа
BookingTimestamp	TIMESTAMP	Not null, Default CURRENT_TIMESTAMP	Дата и время создания заказа
UserId	NUMBER	Foreign key (Users.Id)	Идентификатор пользователя, сделавшего заказ

Продолжение таблицы 3.6.

Столбец	Тип данных	Ограничение целостности	Назначение
ScheduleId	NUMBER	Not null, Foreign key (Schedules.Id)	Идентификатор сеанса, на который куплен билет

Скрипты создания таблиц представлены в приложении В.

3.2 Схемы пользователей и модель доступа

В разрабатываемой базе данных реализована централизованная модель управления доступом, основанная на разделении пользователей СУБД и логических ролей внутри приложения. Вместо создания отдельных схем для каждой роли, все объекты базы данных (таблицы, пакеты, процедуры) создаются и принадлежат единому пользователю-владельцу — C##CinemaAdmin. Этот пользователь выступает в роли административной схемы, ответственной за целостность и структуру данных.

Доступ конечных пользователей к функционалу базы данных осуществляется через трех специально созданных "общих" пользователей, каждый из которых представляет логическую роль в системе:

– C##CinemaClient – пользователь СУБД, представляющий роль обычного клиента кинотеатра.

– C##CinemaManager – пользователь СУБД, представляющий роль менеджера контента.

– C##CinemaAdmin – пользователь-владелец схемы, также представляющий роль главного администратора системы.

Такая архитектура, где данные и логика инкапсулированы в одной схеме, а доступ предоставляется другим пользователям через механизм привилегий, является современным стандартом и обладает рядом преимуществ:

– централизация: все объекты проекта находятся в одном месте, что упрощает их администрирование, резервное копирование и обновление.

– безопасность: конечные пользователи не имеют прямых прав на изменение таблиц. Весь доступ к данным осуществляется исключительно через вызов процедур и функций, сгруппированных в пакеты.

– гибкость: управление правами сводится к выдаче (GRANT) или отзыву (REVOKE) разрешений на выполнение конкретных пакетов, что является более гибким и контролируемым процессом.

Вся бизнес-логика сгруппирована в три пакета, соответствующих ролям: CLIENT_PKG, MANAGER_PKG и ADMIN_PKG. Разграничение доступа реализовано путем выдачи привилегий EXECUTE на эти пакеты соответствующим пользователям. Пример выдачи прав для пользователя представлен в листинге 3.1.

```
GRANT EXECUTE ON CLIENT_PKG TO C##CinemaClient;
```

Листинг 3.1 – Пример выдачи прав на пакет CLIENT_PKG

Пользователь C##CinemaManager, в свою очередь, получает права на выполнение как CLIENT_PKG, так и MANAGER_PKG, реализуя иерархическую модель доступа. Для упрощения работы с объектами из других схем были созданы публичные синонимы для всех таблиц и пакетов. Пример создания публичных синонимов представлен в листинге 3.2.

```
CREATE OR REPLACE PUBLIC SYNONYM Roles FOR C##CinemaAdmin.Roles;
CREATE OR REPLACE PUBLIC SYNONYM Users FOR C##CinemaAdmin.Users;
CREATE OR REPLACE PUBLIC SYNONYM Movies FOR C##CinemaAdmin.Movies;
CREATE OR REPLACE PUBLIC SYNONYM Halls FOR C##CinemaAdmin.Halls;
CREATE OR REPLACE PUBLIC SYNONYM Schedules FOR C##CinemaAdmin.Schedules;
CREATE OR REPLACE PUBLIC SYNONYM Orders FOR C##CinemaAdmin.Orders;
```

Листинг 3.2 – Пример создания публичных синонимов

Профили безопасности в данном проекте не создавались, так как управление ресурсами для данной задачи не являлось критичным и может быть реализовано на уровне настроек PDB по умолчанию.

Созданная структура пользователей и привилегий обеспечивает надежное и безопасное взаимодействие с базой данных, полностью инкапсулируя прямой доступ к таблицам и заставляя всех пользователей работать через строго определенный программный интерфейс, реализованный в виде пакетов PL/SQL.

Скрипты создания пользователей и выдачи привилегий представлены в приложении Г.

3.3 Хранимые процедуры и функции

Для реализации всей бизнес-логики и обеспечения контролируемого доступа к данным, в проекте используется подход с инкапсуляцией всех операций в хранимые объекты PL/SQL. Вся логика сгруппирована в три пакета, которые соответствуют ролевой модели: ADMIN_PKG, MANAGER_PKG и CLIENT_PKG.

Пакет ADMIN_PKG содержит подпрограммы, предназначенные исключительно для выполнения администратором системы. Данные подпрограммы описаны в таблице 3.7.

Таблица 3.7 – Процедуры пакета ADMIN_PKG

Название процедуры	**Описание**
DeleteUser	Выполняет физическое удаление пользователя и всех связанных с ним данных.
AssignManagerRole	Повышает роль обычного пользователя до "Менеджера".
AddHall	Добавляет новый кинозал в базу данных и возвращает его ID.

Продолжение таблицы 3.7.

Название процедуры	**Описание**
UpdateHallInfo	Изменяет атрибуты (название, вместимость, тип) существующего зала.
DeleteHall	Выполняет физическое удаление зала и каскадное удаление связанных сеансов.

Пакет MANAGER_PKG предоставляет интерфейс для управления контентом и операционной деятельностью кинотеатра. Процедуры, которые реализованы в рамках пакета MANAGER_PKG описаны в таблице 3.8.

Таблица 3.8 – Процедуры пакета MANAGER_PKG

Название процедуры	**Описание**
AddMovie	Добавляет новый фильм в каталог и возвращает его ID.
UpdateMovieInfo	Изменяет атрибуты существующего фильма.
DeleteMovie	Выполняет физическое удаление фильма и каскадное удаление связанных данных.
UpdateMoviePoster	Загружает файл изображения (постера) в BLOB-столбец фильма, используя пакет DBMS_LOB.
UpdateMovieTrailer	Загружает видеофайл (трейлера) в BLOB-столбец фильма.
AddSchedule	Создает новый сеанс, проверяя на конфликты времени в расписании, и возвращает ID сеанса.
UpdateScheduleTime	Изменяет время существующего сеанса с повторной проверкой на конфликты.
DeleteSchedule	Удаляет сеанс из расписания.
UpdateOrderStatus	Позволяет изменить статус любого заказа.

Функции, которые реализованы в рамках пакета MANAGER_PKG описаны в таблице 3.9.

Таблица 3.9 – Функции пакета MANAGER_PKG

Название функции	**Описание**
GetAllOrders	Возвращает SYS_REFCURSOR со списком всех заказов в системе для отчетности, с возможностью фильтрации.

Пакет CLIENT_PKG инкапсулирует основной пользовательский функционал. Процедуры, которые реализованы в рамках пакета CLIENT_PKG описаны в таблице 3.10.

Таблица 3.10 – Процедуры пакета CLIENT_PKG

Название процедуры	**Описание**
RegisterUser	Создает новую учетную запись с ролью "Пользователь" и возвращает ID нового пользователя.
UpdateUserEmail	Позволяет пользователю изменить свой email.
CreateOrder	Создает новый заказ (покупку билета), выполняя проверки на доступность мест и лимит билетов на пользователя. Возвращает ID заказа.
CancelOrder	Позволяет пользователю отменить свой заказ. Содержит бизнес-правило, запрещающее отмену незадолго до начала сеанса.

Весь функционал, предназначенный для конечного пользователя системы (роль "Клиент"), инкапсулирован в пакете CLIENT_PKG. Этот пакет предоставляет программный интерфейс для выполнения всех базовых операций, таких как регистрация, авторизация и управление заказами. Функции, которые реализованы в рамках пакета CLIENT_PKG, подробно описаны в таблице 3.11..

Таблица 3.11 – Функции пакета CLIENT_PKG

Название функции	**Описание**
LoginUser	Выполняет аутентификацию пользователя по email и паролю. В случае успеха возвращает ID пользователя, иначе генерирует ошибку.
GetOrderHistory	Возвращает SYS_REFCURSOR с историей заказов. Клиент может просматривать только свои заказы.
FindMovies	Возвращает SYS_REFCURSOR со списком всех фильмов в прокате. Поддерживает фильтрацию по названию, жанру и сортировку.
GetUserIdByEmail	Возвращает идентификатор по заданному адресу электронной почты.
GetMovieIdByTitle	Получить идентификатор фильма по названию.

Продолжение таблицы 3.11.

Название функции	**Описание**
GetHallIdByName	Получить идентификатор по названию зала.
GetScheduleId	Получить идентификатор раписания.
GetLastOrderId	Получить идентификатор последнего заказа.

Листинги всех процедур и функций представлены в приложении Д.

3.4 Выводы

В данном разделе была реализована полная структура объектов базы данных для информационной системы "Кинотеатр". Разработано шесть взаимосвязанных таблиц с четко определенными типами данных, первичными и внешними ключами, а также ограничениями целостности, обеспечивающими корректность хранимой информации. Вместо множества схем была реализована современная архитектура с единой схемой-владельцем (C##CinemaAdmin) и тремя пользователями СУБД (C##CinemaAdmin, C##CinemaManager, C##CinemaClient), представляющими логические роли в системе.

Для инкапсуляции всей бизнес-логики и обеспечения безопасного доступа к данным был разработан набор подпрограмм на языке PL/SQL, сгруппированных в три пакета: ADMIN_PKG, MANAGER_PKG и CLIENT_PKG. Применение пакетов позволило четко разграничить функционал по ролям и управлять доступом на уровне целых модулей, а не отдельных процедур.

4 Описание процедур экспорта и импорта

Импорт и экспорт данных – это важные операции для обмена информацией между системами, создания резервных копий и миграции данных. В рамках данного проекта был реализован функционал для массовой выгрузки и загрузки данных о заказах, что позволяет, например, переносить историю транзакций между средами разработки и тестирования.

Вся логика инкапсулирована в пакете ADMIN_PKG, так как операции массового импорта и экспорта являются административными задачами. Для работы с файловой системой используется объект DIRECTORY и системный пакет DBMS_LOB, а для преобразования данных — встроенные JSON-функции Oracle.

Функция ExportOrdersToJSON предназначена для экспорта данных из таблицы Orders в формат JSON. Она формирует единый CLOB-объект, содержащий полную информацию о выгрузке, включая метаданные и массив всех экспортируемых заказов.

Процесс работы функции реализован методом "ручной" сборки JSON для обхода ограничений СУБД на размер агрегируемых данных, что критически важно при работе с большими объемами, такими как 100 000 строк.

Сначала функция инициализирует пустой CLOB и формирует "шапку" JSON-документа, вычисляя общее количество строк. Затем открывается курсор с SQL-запросом, который для каждой строки таблицы Orders формирует небольшой, отдельный JSON-объект с помощью функции JSON_OBJECT. Этот запрос также поддерживает необязательный параметр p_Limit для ограничения количества выгружаемых записей. Далее, в цикле, процедура последовательно считывает каждый маленький JSON-объект из курсора и дописывает (DBMS_LOB.WRITEAPPEND) его в основной CLOB-объект, расставляя запятые между элементами массива. В конце к CLOB-объекту дописываются закрывающие скобки, формируя валидный JSON-документ.

Такой подход, в отличие от использования одной агрегирующей функции JSON_ARRAYAGG на большом наборе данных, гарантированно работает с любым количеством строк, не вызывая ошибок переполнения памяти или строковых буферов.

Процедура ImportOrdersFromJSON_File выполняет обратную операцию: чтение данных из файла формата JSON и их вставку в таблицу Orders. Она принимает на вход имя директории и имя файла.

Сначала с помощью системного пакета DBMS_LOB процедура читает содержимое указанного файла в переменную типа CLOB. Для этого используется указатель на файл (BFILE), который создается с помощью функции BFILENAME, связывающей логический объект DIRECTORY и физическое имя файла. Процедура LOADCLOBFROMFILE выполняет побайтовое чтение, обеспечивая корректную работу с файлами большого размера. Далее идет разбор и вставка данных. Основой этого этапа является функция JSON_TABLE. Она "на лету" преобразует текстовый CLOB с JSON-массивом в реляционное представление. Оператор INSERT INTO FROM JSON_TABLE эффективно считывает данные из

этой виртуальной таблицы и вставляет их в таблицу Orders. Перед вставкой выполняется проверка WHERE EXISTS, чтобы гарантировать, что UserId и ScheduleId, указанные в JSON, действительно существуют в базе данных, поддерживая ее целостность.

По завершении операции процедура выводит информацию о количестве успешно импортированных записей. Использование COMMIT и ROLLBACK в блоке EXCEPTION обеспечивает атомарность операции: либо все записи из файла успешно добавляются, либо ни одной.

Реализованные механизмы экспорта и импорта позволяют эффективно управлять большими объемами данных заказов, создавая резервные копии или перенося их между различными экземплярами базы данных.

Функция экспорта в JSON представлена в приложении Е.

Процедура импорта из JSON представлена в приложении Ж.

5 Тестирование производительности

Для проверки способности базы данных эффективно обрабатывать большие объемы информации было проведено нагрузочное тестирование таблицы Orders. Этот этап позволяет оценить производительность запросов на неоптимизированной структуре и продемонстрировать эффект от применения индексов.

Для заполнения таблицы Orders была разработана хранимая процедура ADMIN_PKG.GenerateTestOrders. В отличие от подхода с использованием внешнего приложения, вся логика генерации инкапсулирована непосредственно в базе данных на языке PL/SQL. Это обеспечивает максимальную производительность вставки, так как исключает сетевые задержки и накладные расходы на вызов процедур извне.

Процедура GenerateTestOrders принимает на вход один параметр p_NumberOfOrders, указывающий количество записей для генерации.

Сначала процедура проверяет наличие и при необходимости создает технические записи, на которые будут ссылаться генерируемые заказы: временную роль 'Тестовый', пользователя 'TestGen', фильм 'Test Movie for Generation', зал 'Test Hall for Generation' и один сеанс. Использование блоков EXCEPTION THEN NULL позволяет запускать процедуру многократно без ошибок дублирования.

Процедура входит в основной цикл, который выполняется заданное количество раз (например, 100 000). На каждой итерации с помощью системного пакета DBMS_RANDOM генерируются случайные данные для заказа: номер ряда и места, а также итоговая стоимость.

Для оптимизации процесса и уменьшения нагрузки на журнал транзакций, операция COMMIT выполняется не после каждой вставки, а пакетами — после каждых 10 000 сгенерированных записей.

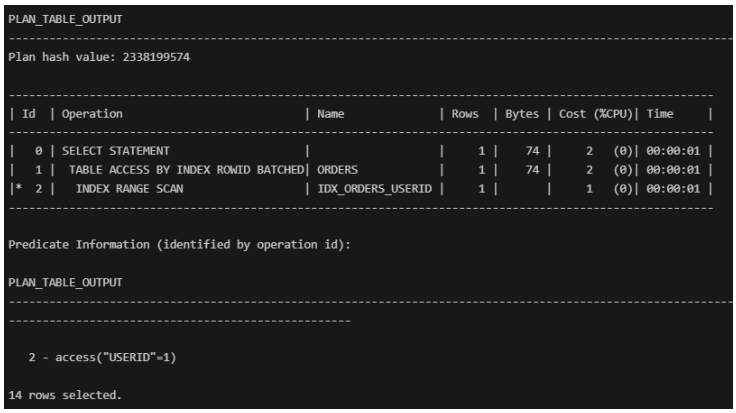
Этот подход позволил быстро и эффективно заполнить таблицу Orders необходимым для тестирования количеством строк.

Далее, было произведено несколько запросов к таблице с различными условиями. Для примера, возьмем запрос к полю UserId. План выполнения запроса до создания индекса idx_orders_userid с условием по полю UserId представлен на рисунке 5.1.

```
PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------------
Plan hash value: 1275100350

------------------------------------------------------------------------------
| Id  | Operation         | Name   | Rows  | Bytes | Cost (%CPU)| Time     |
------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |        |     1 |    74 |   339   (1)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| ORDERS |     1 |    74 |   339   (1)| 00:00:01 |
------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------------

   1 - filter("USERID"=1)

13 rows selected.
```

Рисунок 5.1 – Запрос без индекса

План выполнения запроса после создания индекса idx_orders_userid с условием по полю UserId представлен на рисунке 5.2.



```
PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------------------------------------
Plan hash value: 2338199574

-------------------------------------------------------------------------------------------------
| Id  | Operation                           | Name              | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                    |                   |     1 |    74 |     2   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID BATCHED| ORDERS            |     1 |    74 |     2   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN                  | IDX_ORDERS_USERID |     1 |       |     1   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):

PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------------------------------------
----------------------------------------------------

   2 - access("USERID"=1)

14 rows selected.
```

Рисунок 5.2 – Запрос с индексом

Как мы видим, использование индекса уменьшило стоимость плана запроса до 2, чем план запроса, который не использует индекса. В результате выполнения запроса без индекса стоимость была равна 339.

Результаты тестирования подтверждают правильность выбранной стратегии индексирования и демонстрируют критическую важность индексов при работе с большими объемами данных. Устранение операций полного сканирования таблиц и промежуточной сортировки существенно сокращает время отклика системы и снижает нагрузку на ресурсы сервера. Полученные данные свидетельствуют о том, что разработанная структура базы данных способна эффективно обрабатывать значительные объемы информации при сохранении высокой производительности запросов.

Код генерации 100000 строк представлен в приложении З.

6 Описание технологии и ее применение в базе данных

В качестве ключевой технологии, соответствующей теме курсовой работы "Реализация базы данных кинотеатра с использованием мультимедийных типов данных", была выбрана работа с большими двоичными объектами (LOB – Large Objects) в Oracle, в частности, с типом данных BLOB (Binary Large Object).

BLOB — это тип данных, предназначенный для хранения больших объемов неструктурированной бинарной информации непосредственно в базе данных. В отличие от текстовых типов (VARCHAR2, CLOB), BLOB хранит "сырые" байты, что делает его идеальным для содержания таких файлов, как изображения (постеры к фильмам) и видео (трейлеры).

Основное преимущество хранения мультимедийных данных в BLOB-столбцах заключается в транзакционной целостности. Файл становится частью записи в таблице. Это означает, что при создании резервной копии, восстановлении или миграции базы данных мультимедийные файлы переносятся вместе с остальными данными автоматически, что исключает риск рассинхронизации, возможный при хранении файлов в файловой системе отдельно от базы данных.

Для управления BLOB-объектами и их взаимодействия с файловой системой сервера Oracle предоставляет мощный системный пакет DBMS_LOB. Этот пакет содержит набор процедур и функций, позволяющих выполнять такие операции, как чтение длины LOB, побайтовое чтение и запись, а также загрузку содержимого целого файла в BLOB-столбец.

Для реализации технологии в проекте были выполнены два ключевых шага. Во-первых, на этапе проектирования структуры в таблицу Movies были добавлены два столбца типа BLOB (Poster для изображений и Trailer для видеофайлов) для непосредственного хранения мультимедийных данных.

Во-вторых, была выполнена настройка доступа к файловой системе. Поскольку база данных Oracle работает в изолированной среде Docker-контейнера, для связи с файлами на хост-машине была создана связка: с помощью механизма Docker Volumes папка с медиафайлами (D:\Уник\БД\Курсач\Media) была "проброшена" внутрь контейнера по пути /media. Затем внутри Pluggable Database CINEMAPDB был создан логический объект DIRECTORY, который выступает в роли безопасного "ярлыка" для этого пути, доступного из PL/SQL.

Создание объекта DIRECTORY и выдача прав пользователям представлено в листинге 6.1.

```
CREATE OR REPLACE DIRECTORY MEDIA_DIR AS '/media';
GRANT READ, WRITE ON DIRECTORY MEDIA_DIR TO C##CinemaAdmin;
GRANT READ, WRITE ON DIRECTORY MEDIA_DIR TO C##CinemaManager;
```

Листинг 6.1 – Создание объекта DIRECTORY и выдача прав пользователям

В-третьих, для инкапсуляции и безопасного управления процессом загрузки была реализована программная логика на языке PL/SQL. В пакете MANAGER_PKG были созданы две специализированные процедуры: UpdateMoviePoster и UpdateMovieTrailer. Каждая из этих процедур

отвечает за атомарную операцию по добавлению мультимедийного контента для конкретного фильма.

Процедуры используют системный пакет DBMS_LOB для взаимодействия с BLOB-объектами. Сначала с помощью функции BFILENAME создается указатель на внешний файл, после чего DBMS_LOB.LOADBLOBFROMFILE выполняет эффективную побайтовую загрузку его содержимого в соответствующий столбец. Такой подход не только скрывает сложность работы с LOB-объектами от конечного пользователя, но и обеспечивает целостность операции благодаря механизмам транзакций. Пример реализации процедуры для загрузки постера фильма представлен в листинге 6.2.

```
PROCEDURE UpdateMoviePoster (p_CurrentUserId IN Users.Id%TYPE, p_MovieId IN Movies.Id%TYPE, p_DirectoryName IN VARCHAR2, p_FileName IN VARCHAR2) IS
          v_CurrentUserRole Roles.Name%TYPE; dest_lob BLOB; src_bfile BFILE; dest_offset INTEGER := 1; src_offset INTEGER := 1;
    BEGIN
        SELECT r.Name INTO v_CurrentUserRole FROM Users u JOIN Roles r ON u.RoleId = r.Id WHERE u.Id = p_CurrentUserId;
        IF v_CurrentUserRole NOT IN ('Администратор', 'Менеджер') THEN RAISE_APPLICATION_ERROR(-20140, 'Ошибка доступа: Недостаточно прав.'); END IF;
         UPDATE Movies SET Poster = EMPTY_BLOB() WHERE Id = p_MovieId RETURNING Poster INTO dest_lob;
       IF dest_lob IS NULL THEN RAISE_APPLICATION_ERROR(-20114, 'Фильм не найден.'); END IF;
        src_bfile := BFILENAME(p_DirectoryName, p_FileName);
        DBMS_LOB.FILEOPEN(src_bfile, DBMS_LOB.FILE_READONLY);
                    DBMS_LOB.LOADBLOBFROMFILE(dest_lob,    src_bfile, DBMS_LOB.GETLENGTH(src_bfile), dest_offset, src_offset);
        DBMS_LOB.FILECLOSE(src_bfile);
        COMMIT;
          DBMS_OUTPUT.PUT_LINE('Постер для фильма ID '||p_MovieId||' загружен.');
    EXCEPTION
        WHEN NO_DATA_FOUND THEN RAISE_APPLICATION_ERROR(-20301, 'Вы не авторизованы.');
        WHEN OTHERS THEN ROLLBACK; RAISE;
END UpdateMoviePoster;
```

Листинг 6.2 – Процедура для загрузки постера фильма

Процедура UpdateMovieTrailer, предназначенная для загрузки видеофайла трейлера, реализована по аналогии с процедурой UpdateMoviePoster. Она использует тот же самый технологический стек: проверку прав доступа, создание указателя BFILE на файл и вызов процедуры для побайтовой записи в целевой BLOB-столбец. Единственным отличием является то, что операция выполняется над столбцом Trailer, а не Poster.

Для демонстрации работы реализованного механизма можно выполнить следующий тестовый сценарий. Сценарий сначала создает новый фильм, а затем

последовательно загружает для него постер и трейлер.

На рисунке 6.1 представлен результат выполнения процедур добавления фильма и мультимедийных типов данных.

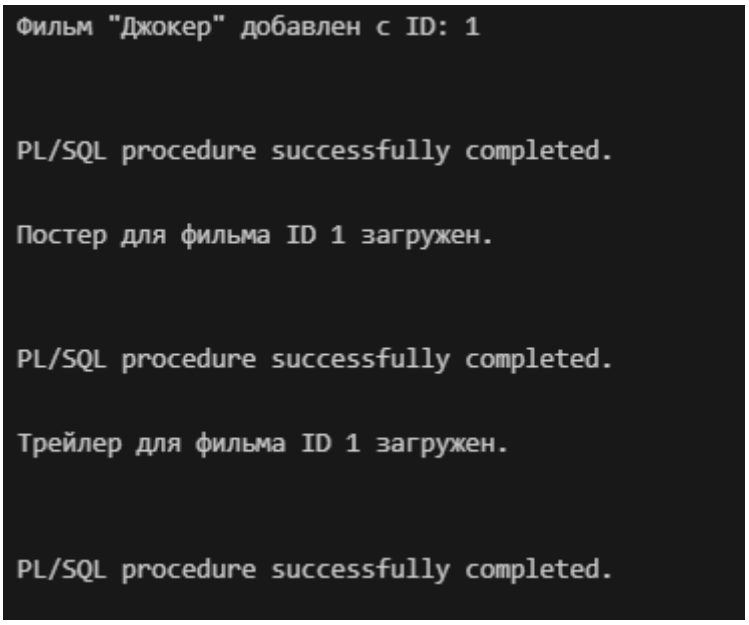


Рисунок 6.1 – Добавление фильма и мультимедийного контента

На рисунке 6.2 представлен вывод мультимедийного контента в двоичном формате для фильма, который был добавлен в результате выполнения процедур, описанных выше.

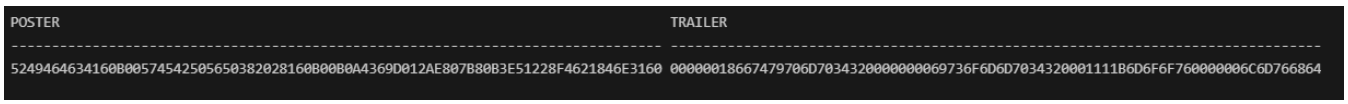


Рисунок 6.2 – Вывод мультимедийного контента в двоичном формате

Результаты выполнения данного скрипта подтверждают, что процедуры корректно считывают файлы из указанной директории и сохраняют их бинарное содержимое в соответствующие BLOB-столбцы таблицы Movies. Таким образом, технология работы с мультимедийными типами данных была успешно реализована и протестирована.

Заключение

Данная курсовая работа представляет собой разработку архитектуры и программного интерфейса реляционной базы данных для информационной системы "Кинотеатр". Проект демонстрирует успешное решение задачи создания полнофункционального серверного ядра с применением технологии хранения мультимедийных данных и реализацией строгой ролевой модели доступа в СУБД Oracle.

В ходе работы была спроектирована и реализована нормализованная структура базы данных, включающая шесть основных таблиц для хранения информации о пользователях, ролях, фильмах, залах, расписании и заказах. Архитектура базы данных построена с использованием механизма Pluggable Database (PDB), что позволило изолировать проект в отдельном контейнере. Целостность данных обеспечивается системой первичных и внешних ключей с использованием опции ON DELETE CASCADE для автоматического поддержания согласованности при удалении связанных записей.

Особое внимание уделено безопасности и разграничению доступа. Реализована трехуровневая ролевая модель (Клиент, Менеджер, Администратор), представленная отдельными пользователями СУБД. Вся бизнес-логика инкапсулирована в трех пакетах PL/SQL: CLIENT_PKG, MANAGER_PKG и ADMIN_PKG. Доступ к операциям строго регламентирован путем выдачи привилегий EXECUTE на пакеты соответствующим пользователям, что полностью исключает прямой доступ к таблицам для ролей с низким уровнем привилегий.

Ключевая технология проекта — работа с мультимедийными типами данных — была успешно реализована. Для хранения постеров и трейлеров используются столбцы типа BLOB. Разработаны специализированные процедуры, которые с помощью системного пакета DBMS_LOB и объектов DIRECTORY обеспечивают безопасную и эффективную загрузку бинарных файлов в базу данных.

Для выполнения требований по работе с большими объемами данных и обмену информацией были созданы процедуры для импорта и экспорта данных в формат JSON. Для тестирования производительности была разработана процедура, генерирующая 100 000 записей в таблицу заказов. Анализ плана выполнения запросов (EXPLAIN PLAN) до и после создания B-Tree индекса наглядно продемонстрировал его критическую важность, заменив полное сканирование таблицы на быстрый поиск по индексу.

Таким образом, в рамках курсового проекта была успешно создана масштабируемая и безопасная архитектура базы данных, полностью готовая для интеграции с внешним приложением. Все поставленные задачи выполнены: реализована структура данных, разработана бизнес-логика на PL/SQL, применена технология работы с мультимедийными данными и продемонстрированы методы оптимизации производительности.

Список используемых источников

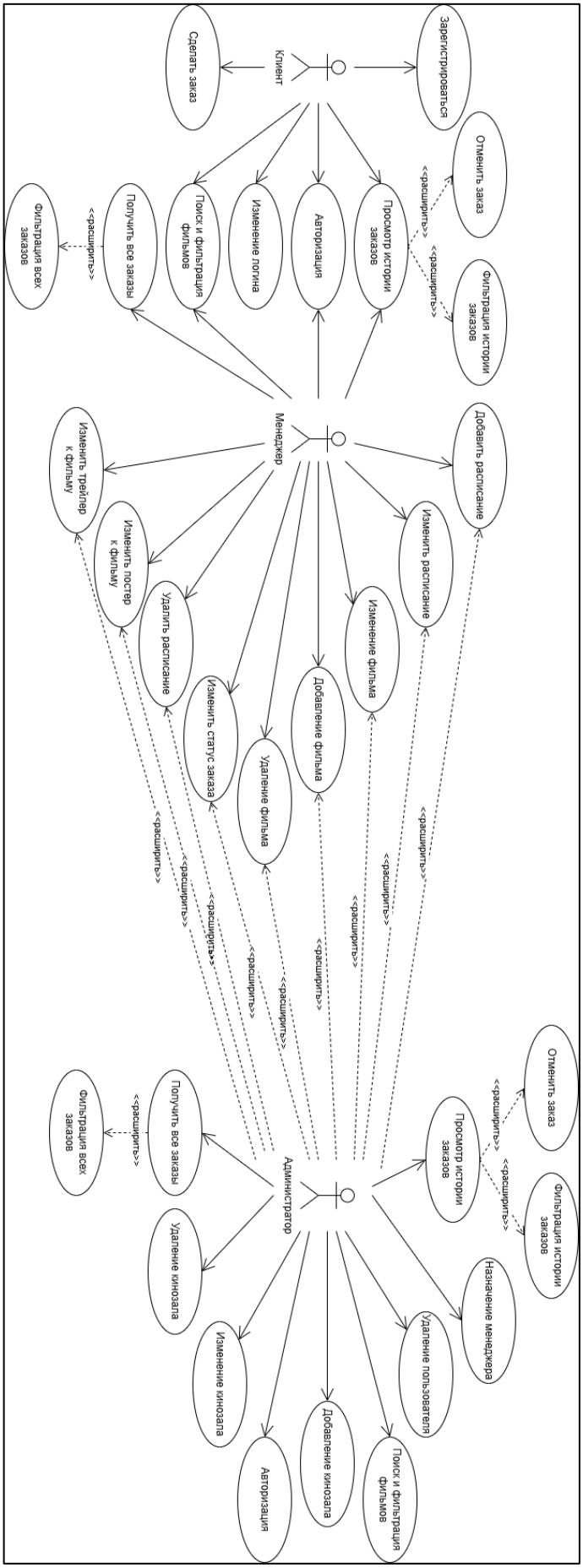
1 Кинопоиск [Электронный ресурс] / Режим доступа: https://www.kinopoisk.ru/ – Дата доступа: 15.09.2025.

2 Skyline [Электронный ресурс] / Режим доступа: https://skyline.by/ – Дата доступа: 16.09.2025.

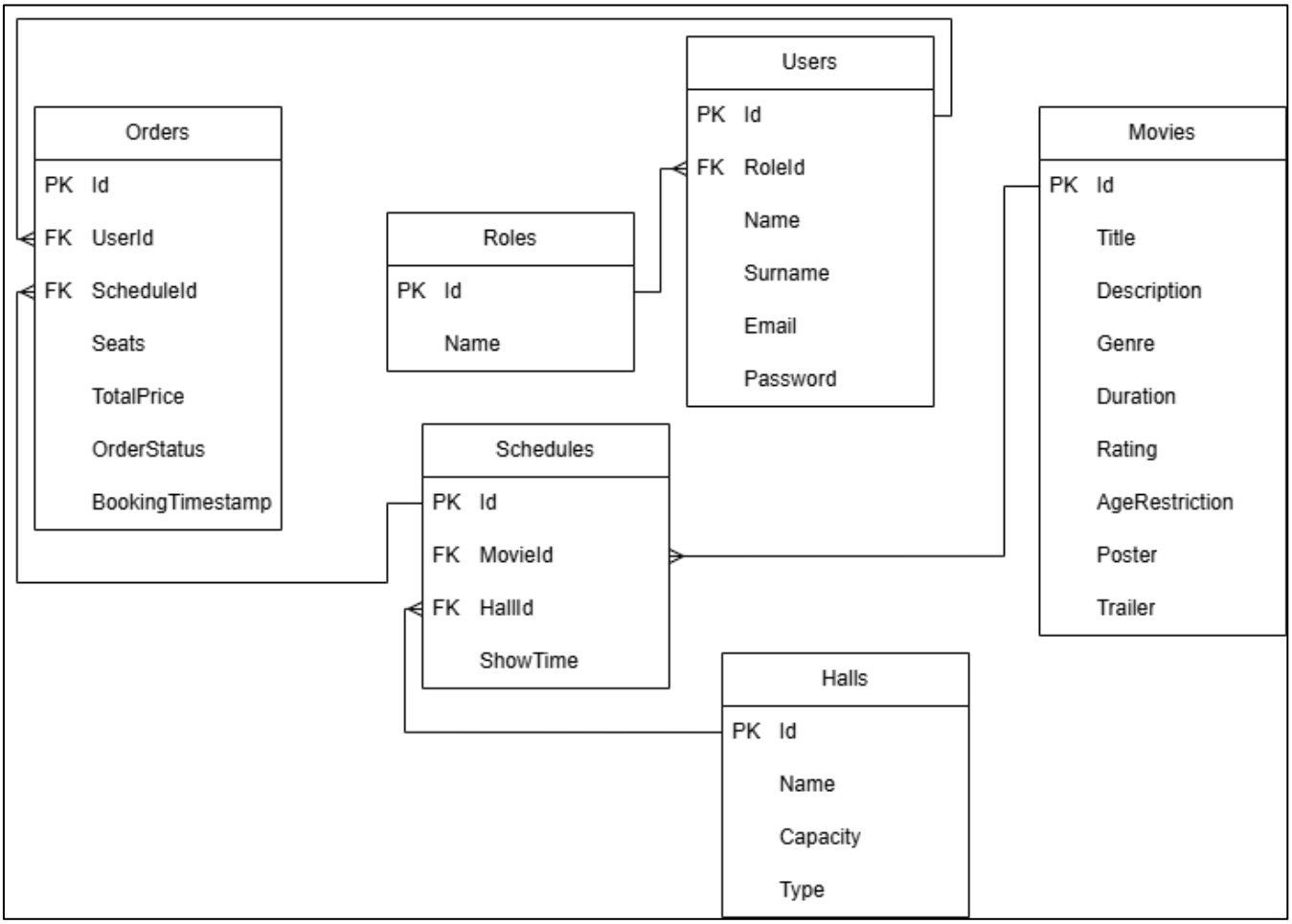
3 Oracle Corporation. Oracle Database 12c Documentation [Электронный ресурс]. / Режим доступа: https://docs.oracle.com/en/database/ – Дата доступа: 17.09.2025.

4 Oracle PL/SQL: хранимые процедуры и функции [Электронный ресурс]. / Режим доступа: https://oracle-base.com/articles/misc/plsql-introduction – Дата доступа: 21.09.2025.

Приложение А



Приложение Б



Приложение В

```
CREATE TABLE Roles (
    Id NUMBER GENERATED BY DEFAULT AS IDENTITY,
    Name NVARCHAR2(50) NOT NULL UNIQUE,
    CONSTRAINT pk_roles PRIMARY KEY (Id)
);

CREATE TABLE Users (
    Id NUMBER GENERATED BY DEFAULT AS IDENTITY,
    Name NVARCHAR2(100) NOT NULL,
    Surname NVARCHAR2(100) NOT NULL,
    Email NVARCHAR2(255) NOT NULL UNIQUE,
    Password NVARCHAR2(100) NOT NULL,
    RoleId NUMBER NOT NULL,
    CONSTRAINT pk_users PRIMARY KEY (Id),
    CONSTRAINT fk_users_roles FOREIGN KEY (RoleId) REFERENCES Roles(Id)
);

CREATE TABLE Movies (
    Id NUMBER GENERATED BY DEFAULT AS IDENTITY,
    Title NVARCHAR2(255) NOT NULL UNIQUE,
    Director NVARCHAR2(255),
    Description NCLOB,
    Genre NVARCHAR2(100),
    Duration NUMBER,
    Rating NUMBER(3, 1),
    AgeRestriction NVARCHAR2(10),
    Poster BLOB,
    Trailer BLOB,
    CONSTRAINT pk_movies PRIMARY KEY (Id)
);

CREATE TABLE Halls (
    Id NUMBER GENERATED BY DEFAULT AS IDENTITY,
    Name NVARCHAR2(100) NOT NULL UNIQUE,
    Capacity NUMBER NOT NULL,
    Type NVARCHAR2(50),
    CONSTRAINT pk_halls PRIMARY KEY (Id)
);

CREATE TABLE Schedules (
    Id NUMBER GENERATED BY DEFAULT AS IDENTITY,
    ShowTime TIMESTAMP NOT NULL,
    MovieId NUMBER NOT NULL,
    HallId NUMBER NOT NULL,
    CONSTRAINT pk_schedules PRIMARY KEY (Id),
     CONSTRAINT fk_schedules_movies FOREIGN KEY (MovieId) REFERENCES
Movies(Id) ON DELETE CASCADE,
      CONSTRAINT fk_schedules_halls FOREIGN KEY (HallId) REFERENCES
Halls(Id) ON DELETE CASCADE
);
```

```
CREATE TABLE Orders (
    Id NUMBER GENERATED BY DEFAULT AS IDENTITY,
    Seats NVARCHAR2(255) NOT NULL,
    TotalPrice NUMBER(10, 2) NOT NULL,
    OrderStatus NVARCHAR2(50) DEFAULT 'Забронирован' NOT NULL,
    BookingTimestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP NOT NULL,
    UserId NUMBER,
    ScheduleId NUMBER NOT NULL,
    CONSTRAINT pk_orders PRIMARY KEY (Id),
   CONSTRAINT fk_orders_users FOREIGN KEY (UserId) REFERENCES Users(Id)
ON DELETE CASCADE,
    CONSTRAINT fk_orders_schedules FOREIGN KEY (ScheduleId) REFERENCES
Schedules(Id) ON DELETE CASCADE
);
```

Приложение Г

```
CREATE USER C##CinemaAdmin IDENTIFIED BY CinemaAdmin123;
GRANT CONNECT, RESOURCE, DBA TO C##CinemaAdmin CONTAINER=ALL;

CREATE USER C##CinemaManager IDENTIFIED BY CinemaManager123;
GRANT CONNECT TO C##CinemaManager CONTAINER=ALL;

CREATE USER C##CinemaClient IDENTIFIED BY CinemaClient123;
GRANT CONNECT TO C##CinemaClient CONTAINER=ALL;

CREATE OR REPLACE PUBLIC SYNONYM Roles FOR C##CinemaAdmin.Roles;
CREATE OR REPLACE PUBLIC SYNONYM Users FOR C##CinemaAdmin.Users;
CREATE OR REPLACE PUBLIC SYNONYM Movies FOR C##CinemaAdmin.Movies;
CREATE OR REPLACE PUBLIC SYNONYM Halls FOR C##CinemaAdmin.Halls;
CREATE OR REPLACE PUBLIC SYNONYM Schedules FOR C##CinemaAdmin.Schedules;
CREATE OR REPLACE PUBLIC SYNONYM Orders FOR C##CinemaAdmin.Orders;

CREATE OR REPLACE PUBLIC SYNONYM ADMIN_PKG FOR C##CinemaAdmin.ADMIN_PKG;
CREATE OR REPLACE PUBLIC SYNONYM MANAGER_PKG FOR
C##CinemaAdmin.MANAGER_PKG;
CREATE OR REPLACE PUBLIC SYNONYM CLIENT_PKG FOR
C##CinemaAdmin.CLIENT_PKG;

GRANT EXECUTE ON CLIENT_PKG TO C##CinemaClient;

GRANT EXECUTE ON CLIENT_PKG TO C##CinemaManager;
GRANT EXECUTE ON MANAGER_PKG TO C##CinemaManager;

GRANT EXECUTE ON CLIENT_PKG TO C##CinemaAdmin;
GRANT EXECUTE ON MANAGER_PKG TO C##CinemaAdmin;
GRANT EXECUTE ON ADMIN_PKG TO C##CinemaAdmin;

GRANT SELECT ON Roles TO C##CinemaManager;
GRANT SELECT ON Users TO C##CinemaManager;
GRANT SELECT ON Movies TO C##CinemaManager;
GRANT SELECT ON Halls TO C##CinemaManager;
GRANT SELECT ON Schedules TO C##CinemaManager;
GRANT SELECT ON Orders TO C##CinemaManager;
```

Приложение Д

```
PROCEDURE DeleteUser (p_CurrentUserId IN Users.Id%TYPE, p_TargetUserId
IN Users.Id%TYPE) IS
        v_CurrentUserRole Roles.Name%TYPE;
    BEGIN
        SELECT r.Name INTO v_CurrentUserRole FROM Users u JOIN Roles r
ON u.RoleId = r.Id WHERE u.Id = p_CurrentUserId;
                  IF   v_CurrentUserRole   !=   'Администратор'   THEN
RAISE_APPLICATION_ERROR(-20140, 'Ошибка доступа: Недостаточно прав.');
END IF;
                    IF   p_CurrentUserId   =   p_TargetUserId   THEN
RAISE_APPLICATION_ERROR(-20220, 'Нельзя удалить управляющие роли.');
END IF;
        DELETE FROM Users WHERE Id = p_TargetUserId;
                      IF    SQL%ROWCOUNT    >    0    THEN    COMMIT;
DBMS_OUTPUT.PUT_LINE('Пользователь  '||p_TargetUserId||'  и  все  его
данные удалены.'); END IF;
    EXCEPTION
        WHEN NO_DATA_FOUND THEN RAISE_APPLICATION_ERROR(-20301, 'Вы не
авторизованы.');
        WHEN OTHERS THEN ROLLBACK; RAISE;
    END DeleteUser;

     PROCEDURE  AssignManagerRole  (p_CurrentUserId  IN  Users.Id%TYPE,
p_TargetUserId IN Users.Id%TYPE) IS
                v_CurrentUserRole   Roles.Name%TYPE;   v_ManagerRoleId
Roles.Id%TYPE; v_TargetUserRole Roles.Name%TYPE;
    BEGIN
        SELECT r.Name INTO v_CurrentUserRole FROM Users u JOIN Roles r
ON u.RoleId = r.Id WHERE u.Id = p_CurrentUserId;
                  IF   v_CurrentUserRole   !=   'Администратор'   THEN
RAISE_APPLICATION_ERROR(-20200, 'Только Администратор может назначать
менеджеров.'); END IF;
         SELECT r.Name INTO v_TargetUserRole FROM Users u JOIN Roles r
ON u.RoleId = r.Id WHERE u.Id = p_TargetUserId;
        IF v_TargetUserRole = 'Менеджер' THEN RAISE_APPLICATION_ERROR(-
20210, 'Пользователь уже является менеджером.'); END IF;
           SELECT  Id  INTO  v_ManagerRoleId  FROM  Roles  WHERE  Name  =
'Менеджер';
            UPDATE  Users  SET  RoleId  =  v_ManagerRoleId  WHERE  Id  =
p_TargetUserId;
                      IF    SQL%ROWCOUNT    >    0    THEN    COMMIT;
DBMS_OUTPUT.PUT_LINE('Пользователю '||p_TargetUserId||' присвоена роль
Менеджера.'); END IF;
    EXCEPTION
             WHEN  NO_DATA_FOUND  THEN  RAISE_APPLICATION_ERROR(-20222,
'Пользователь с таким ID не найден.');
        WHEN OTHERS THEN ROLLBACK; RAISE;
    END AssignManagerRole;
```

```
     PROCEDURE AddHall (p_CurrentUserId IN Users.Id%TYPE, p_Name IN
Halls.Name%TYPE,  p_Capacity  IN  Halls.Capacity%TYPE,  p_Type  IN
Halls.Type%TYPE, p_NewHallId OUT Halls.Id%TYPE) IS
        v_CurrentUserRole Roles.Name%TYPE;
    BEGIN
        SELECT r.Name INTO v_CurrentUserRole FROM Users u JOIN Roles r
ON u.RoleId = r.Id WHERE u.Id = p_CurrentUserId;
                 IF  v_CurrentUserRole  !=  'Администратор'  THEN
RAISE_APPLICATION_ERROR(-20140, 'Ошибка доступа: Недостаточно прав.');
END IF;
          INSERT INTO Halls (Name, Capacity, Type) VALUES (p_Name,
p_Capacity, p_Type) RETURNING Id INTO p_NewHallId;
        COMMIT;
           DBMS_OUTPUT.PUT_LINE('Зал "'||p_Name||'" добавлен с ID:
'||p_NewHallId);
    EXCEPTION
        WHEN NO_DATA_FOUND THEN RAISE_APPLICATION_ERROR(-20301, 'Вы не
авторизованы.');
         WHEN DUP_VAL_ON_INDEX THEN RAISE_APPLICATION_ERROR(-20102,
'Такой зал уже существует.');
        WHEN OTHERS THEN ROLLBACK; RAISE;
    END AddHall;

   PROCEDURE UpdateHallInfo (p_CurrentUserId IN Users.Id%TYPE, p_HallId
IN  Halls.Id%TYPE,  p_NewName  IN  Halls.Name%TYPE  DEFAULT  NULL,
p_NewCapacity  IN  Halls.Capacity%TYPE  DEFAULT  NULL,  p_NewType  IN
Halls.Type%TYPE DEFAULT NULL) IS
        v_CurrentUserRole Roles.Name%TYPE;
    BEGIN
        SELECT r.Name INTO v_CurrentUserRole FROM Users u JOIN Roles r
ON u.RoleId = r.Id WHERE u.Id = p_CurrentUserId;
                 IF  v_CurrentUserRole  !=  'Администратор'  THEN
RAISE_APPLICATION_ERROR(-20140, 'Ошибка доступа: Недостаточно прав.');
END IF;
          UPDATE Halls SET Name = NVL(p_NewName, Name), Capacity =
NVL(p_NewCapacity, Capacity), Type = NVL(p_NewType, Type) WHERE Id =
p_HallId;
         IF SQL%ROWCOUNT = 0 THEN RAISE_APPLICATION_ERROR(-20114, 'Зал
не найден.'); END IF;
        COMMIT;
        DBMS_OUTPUT.PUT_LINE('Зал ID '||p_HallId||' обновлен.');
    EXCEPTION
        WHEN NO_DATA_FOUND THEN RAISE_APPLICATION_ERROR(-20301, 'Вы не
авторизованы.');
        WHEN OTHERS THEN ROLLBACK; RAISE;
    END UpdateHallInfo;

    PROCEDURE DeleteHall (p_CurrentUserId IN Users.Id%TYPE, p_HallId IN
Halls.Id%TYPE) IS
        v_CurrentUserRole Roles.Name%TYPE;
    BEGIN
        SELECT r.Name INTO v_CurrentUserRole FROM Users u JOIN Roles r
ON u.RoleId = r.Id WHERE u.Id = p_CurrentUserId;
```

```
                IF   v_CurrentUserRole   !=   'Администратор'   THEN
RAISE_APPLICATION_ERROR(-20140, 'Ошибка доступа: Недостаточно прав.');
END IF;
        DELETE FROM Halls WHERE Id = p_HallId;
         IF SQL%ROWCOUNT = 0 THEN RAISE_APPLICATION_ERROR(-20114, 'Зал
не найден.'); END IF;
        COMMIT;
        DBMS_OUTPUT.PUT_LINE('Зал '||p_HallId||' и все связанные данные
удалены.');
    EXCEPTION
        WHEN NO_DATA_FOUND THEN RAISE_APPLICATION_ERROR(-20301, 'Вы не
авторизованы.');
        WHEN OTHERS THEN ROLLBACK; RAISE;
    END DeleteHall;

    PROCEDURE GenerateTestOrders (p_NumberOfOrders IN NUMBER) IS
             v_TestUserId          Users.Id%TYPE;   v_TestScheduleId
Schedules.Id%TYPE;
    BEGIN
         BEGIN INSERT INTO Roles (Name) VALUES ('Тестовый'); COMMIT;
EXCEPTION WHEN DUP_VAL_ON_INDEX THEN NULL; END;
       BEGIN INSERT INTO Users (Name, Surname, Email, Password, RoleId)
VALUES ('TestGen', 'User', 'testgen@user.com', '123', (SELECT Id FROM
Roles WHERE Name = 'Тестовый')); COMMIT; EXCEPTION WHEN DUP_VAL_ON_INDEX
THEN NULL; END;
          BEGIN INSERT INTO Movies (Title) VALUES ('Test Movie for
Generation'); INSERT INTO Halls (Name, Capacity) VALUES ('Test Hall for
Generation', 200000); INSERT INTO Schedules (MovieId, HallId, ShowTime)
VALUES ((SELECT MAX(Id) FROM Movies WHERE Title = 'Test Movie for
Generation'), (SELECT MAX(Id) FROM Halls WHERE Name = 'Test Hall for
Generation'), SYSTIMESTAMP); COMMIT; EXCEPTION WHEN OTHERS THEN NULL;
END;
           SELECT  Id  INTO  v_TestUserId  FROM  Users  WHERE  Email  =
'testgen@user.com';
        SELECT MAX(Id) INTO v_TestScheduleId FROM Schedules;
         DBMS_OUTPUT.PUT_LINE('Начинаем генерацию ' || p_NumberOfOrders
|| ' заказов...');
        FOR i IN 1..p_NumberOfOrders LOOP
            INSERT INTO Orders (Seats, TotalPrice, UserId, ScheduleId,
OrderStatus) VALUES ('Ряд ' || TRUNC(DBMS_RANDOM.VALUE(1, 100)) || ',
Место        '        ||        TRUNC(DBMS_RANDOM.VALUE(1,       100)),
ROUND(DBMS_RANDOM.VALUE(10,  50),  2),  v_TestUserId,  v_TestScheduleId,
CASE WHEN MOD(i, 20) = 0 THEN 'Отменен' ELSE 'Оплачен' END);
           IF MOD(i, 10000) = 0 THEN COMMIT; DBMS_OUTPUT.PUT_LINE('...
добавлено ' || i || ' заказов'); END IF;
        END LOOP;
          COMMIT;  DBMS_OUTPUT.PUT_LINE('Генерация завершена. Успешно
добавлено ' || p_NumberOfOrders || ' заказов.');
    EXCEPTION
        WHEN OTHERS THEN DBMS_OUTPUT.PUT_LINE('Критическая ошибка: ' ||
SQLERRM); ROLLBACK;
    END GenerateTestOrders;
```

```
    FUNCTION ExportOrdersToJSON (p_Limit IN NUMBER DEFAULT NULL) RETURN
CLOB IS
    v_JsonData CLOB;
    v_Cursor SYS_REFCURSOR;
    v_RowCount NUMBER;
    v_LF CHAR(1) := CHR(10);
    BEGIN
    SELECT COUNT(*) INTO v_RowCount FROM Orders;

    DBMS_LOB.CREATETEMPORARY(v_JsonData, TRUE);

    DBMS_LOB.WRITEAPPEND(v_JsonData,
                         LENGTH('{"export_details":{"timestamp":"'     ||
TO_CHAR(SYSTIMESTAMP, 'YYYY-MM-DD"T"HH24:MI:SS.FF') || '","row_count":'
|| v_RowCount || '},"orders":[' || v_LF),
                                '{"export_details":{"timestamp":"'    ||
TO_CHAR(SYSTIMESTAMP, 'YYYY-MM-DD"T"HH24:MI:SS.FF') || '","row_count":'
|| v_RowCount || '},"orders":[' || v_LF);

     OPEN v_Cursor FOR 'SELECT JSON_OBJECT (''seats'' VALUE o.Seats,
''total_price'' VALUE o.TotalPrice, ''status'' VALUE o.OrderStatus,
''user_id'' VALUE o.UserId, ''schedule_id'' VALUE o.ScheduleId) FROM
(SELECT * FROM Orders ORDER BY BookingTimestamp DESC FETCH FIRST
NVL(:lim, 999999999) ROWS ONLY) o' USING p_Limit;

    DECLARE
        v_JsonObject VARCHAR2(4000);
        v_IsFirst BOOLEAN := TRUE;
    BEGIN
        LOOP
                      FETCH   v_Cursor   INTO   v_JsonObject;   EXIT   WHEN
v_Cursor%NOTFOUND;

            IF NOT v_IsFirst THEN
                DBMS_LOB.WRITEAPPEND(v_JsonData, 2, ',' || v_LF);
            END IF;

            DBMS_LOB.WRITEAPPEND(v_JsonData, 2, '  ');

                DBMS_LOB.WRITEAPPEND(v_JsonData,  LENGTH(v_JsonObject),
v_JsonObject);

            v_IsFirst := FALSE;
        END LOOP;
    END;
    CLOSE v_Cursor;

    DBMS_LOB.WRITEAPPEND(v_JsonData, 3, v_LF || ']}');

    RETURN v_JsonData;
    END ExportOrdersToJSON;
```

```
        PROCEDURE   ImportOrdersFromJSON_File   (p_CurrentUserId   IN
Users.Id%TYPE, p_DirectoryName IN VARCHAR2, p_FileName IN VARCHAR2) IS
         v_UserRole  Roles.Name%TYPE;  v_JsonData  CLOB;  v_RowsInserted
NUMBER;
    BEGIN
         SELECT  r.Name  INTO  v_UserRole  FROM  Users  u  JOIN  Roles  r  ON
u.RoleId = r.Id WHERE u.Id = p_CurrentUserId;
              IF   v_UserRole   NOT   IN   ('Администратор')   THEN
RAISE_APPLICATION_ERROR(-20140, 'Доступ запрещен.'); END IF;
        DECLARE
            lob_loc BFILE; dest_offset INTEGER := 1; src_offset INTEGER
:= 1; lang_ctx INTEGER := DBMS_LOB.DEFAULT_LANG_CTX; warning INTEGER;
        BEGIN
            DBMS_LOB.CREATETEMPORARY(v_JsonData, TRUE);
            lob_loc := BFILENAME(p_DirectoryName, p_FileName);
            DBMS_LOB.FILEOPEN(lob_loc, DBMS_LOB.FILE_READONLY);
                    DBMS_LOB.LOADCLOBFROMFILE(v_JsonData,   lob_loc,
DBMS_LOB.GETLENGTH(lob_loc),          dest_offset,          src_offset,
DBMS_LOB.DEFAULT_CSID, lang_ctx, warning);
            DBMS_LOB.FILECLOSE(lob_loc);
        END;
         INSERT  INTO  Orders  (Seats,  TotalPrice,  OrderStatus,  UserId,
ScheduleId)
            SELECT  j.Seats,  j.TotalPrice,  'Импортирован',  j.UserId,
j.ScheduleId
        FROM JSON_TABLE(v_JsonData, '$.orders[*]'
           COLUMNS  (Seats  NVARCHAR2(255)  PATH  '$.seats',  TotalPrice
NUMBER(10,  2)  PATH  '$.total_price',  UserId  NUMBER  PATH  '$.user_id',
ScheduleId NUMBER PATH '$.schedule_id')) j WHERE EXISTS (SELECT 1 FROM
Users WHERE Id = j.UserId)
        AND EXISTS (SELECT 1 FROM Schedules WHERE Id = j.ScheduleId);
        v_RowsInserted := SQL%ROWCOUNT;
        COMMIT;
                DBMS_OUTPUT.PUT_LINE(v_RowsInserted   ||   '   заказов
импортировано.');
        DBMS_LOB.FREETEMPORARY(v_JsonData);
    EXCEPTION
        WHEN NO_DATA_FOUND THEN RAISE_APPLICATION_ERROR(-20301, 'Вы не
авторизованы.');
        WHEN OTHERS THEN ROLLBACK; RAISE;
    END ImportOrdersFromJSON_File;

PROCEDURE AddMovie (
                        p_CurrentUserId           IN     Users.Id%TYPE,
p_Title            IN  Movies.Title%TYPE,
                p_Director                  IN    Movies.Director%TYPE,
p_Description      IN  Movies.Description%TYPE,
                p_Genre                         IN    Movies.Genre%TYPE,
p_Duration         IN  Movies.Duration%TYPE,
                p_Rating                        IN    Movies.Rating%TYPE,
p_AgeRestriction   IN  Movies.AgeRestriction%TYPE,
        p_NewMovieId       OUT Movies.Id%TYPE
    ) IS
```

```
        v_CurrentUserRole Roles.Name%TYPE;
    BEGIN
        SELECT r.Name INTO v_CurrentUserRole FROM Users u JOIN Roles r
ON u.RoleId = r.Id WHERE u.Id = p_CurrentUserId;
        IF v_CurrentUserRole NOT IN ('Администратор', 'Менеджер') THEN
RAISE_APPLICATION_ERROR(-20140, 'Ошибка доступа: Недостаточно прав.');
END IF;
          INSERT INTO Movies (Title, Director, Description, Genre,
Duration, Rating, AgeRestriction) VALUES (p_Title, p_Director,
p_Description, p_Genre, p_Duration, p_Rating, p_AgeRestriction)
RETURNING Id INTO p_NewMovieId;
        COMMIT;
          DBMS_OUTPUT.PUT_LINE('Фильм "'||p_Title||'" добавлен с ID:
'||p_NewMovieId);
    EXCEPTION
          WHEN DUP_VAL_ON_INDEX THEN RAISE_APPLICATION_ERROR(-20103,
'Такое название фильма уже существует.');
        WHEN NO_DATA_FOUND THEN RAISE_APPLICATION_ERROR(-20301, 'Вы не
авторизованы.');
        WHEN OTHERS THEN ROLLBACK; RAISE;
    END AddMovie;

    PROCEDURE UpdateMovieInfo (
        p_CurrentUserId     IN Users.Id%TYPE,
        p_MovieId           IN Movies.Id%TYPE,
        p_NewTitle          IN Movies.Title%TYPE DEFAULT NULL,
        p_NewDirector       IN Movies.Director%TYPE DEFAULT NULL,
        p_NewDescription    IN Movies.Description%TYPE DEFAULT NULL,
        p_NewGenre          IN Movies.Genre%TYPE DEFAULT NULL,
        p_NewDuration       IN Movies.Duration%TYPE DEFAULT NULL,
        p_NewRating         IN Movies.Rating%TYPE DEFAULT NULL,
        p_NewAgeRestriction IN Movies.AgeRestriction%TYPE DEFAULT NULL
    ) IS
        v_CurrentUserRole Roles.Name%TYPE;
    BEGIN
        SELECT r.Name INTO v_CurrentUserRole FROM Users u JOIN Roles r
ON u.RoleId = r.Id WHERE u.Id = p_CurrentUserId;
        IF v_CurrentUserRole NOT IN ('Администратор', 'Менеджер') THEN
RAISE_APPLICATION_ERROR(-20140, 'Ошибка доступа: Недостаточно прав.');
END IF;
         UPDATE Movies SET Title = NVL(p_NewTitle, Title), Director =
NVL(p_NewDirector, Director), Description = NVL(p_NewDescription,
Description), Genre = NVL(p_NewGenre, Genre), Duration =
NVL(p_NewDuration, Duration), Rating = NVL(p_NewRating, Rating),
AgeRestriction = NVL(p_NewAgeRestriction, AgeRestriction) WHERE Id =
p_MovieId;
        IF SQL%ROWCOUNT = 0 THEN RAISE_APPLICATION_ERROR(-20114, 'Фильм
не найден.'); END IF;
        COMMIT;
        DBMS_OUTPUT.PUT_LINE('Фильм ID '||p_MovieId||' обновлен.');
    EXCEPTION
        WHEN NO_DATA_FOUND THEN RAISE_APPLICATION_ERROR(-20301, 'Вы не
авторизованы.');
```

```
        WHEN OTHERS THEN ROLLBACK; RAISE;
    END UpdateMovieInfo;

    PROCEDURE DeleteMovie (p_CurrentUserId IN Users.Id%TYPE, p_MovieId
IN Movies.Id%TYPE) IS
        v_CurrentUserRole Roles.Name%TYPE;
    BEGIN
        SELECT r.Name INTO v_CurrentUserRole FROM Users u JOIN Roles r
ON u.RoleId = r.Id WHERE u.Id = p_CurrentUserId;
        IF v_CurrentUserRole NOT IN ('Администратор', 'Менеджер') THEN
RAISE_APPLICATION_ERROR(-20140, 'Ошибка доступа: Недостаточно прав.');
END IF;
        DELETE FROM Movies WHERE Id = p_MovieId;
        IF SQL%ROWCOUNT = 0 THEN RAISE_APPLICATION_ERROR(-20114, 'Фильм
не найден.'); END IF;
        COMMIT;
        DBMS_OUTPUT.PUT_LINE('Фильм ID '||p_MovieId||' и все связанные
данные удалены.');
    EXCEPTION
        WHEN NO_DATA_FOUND THEN RAISE_APPLICATION_ERROR(-20301, 'Вы не
авторизованы.');
        WHEN OTHERS THEN ROLLBACK; RAISE;
    END DeleteMovie;

     PROCEDURE UpdateMoviePoster (p_CurrentUserId IN Users.Id%TYPE,
p_MovieId IN Movies.Id%TYPE, p_DirectoryName IN VARCHAR2, p_FileName IN
VARCHAR2) IS
         v_CurrentUserRole Roles.Name%TYPE; dest_lob BLOB; src_bfile
BFILE; dest_offset INTEGER := 1; src_offset INTEGER := 1;
    BEGIN
        SELECT r.Name INTO v_CurrentUserRole FROM Users u JOIN Roles r
ON u.RoleId = r.Id WHERE u.Id = p_CurrentUserId;
        IF v_CurrentUserRole NOT IN ('Администратор', 'Менеджер') THEN
RAISE_APPLICATION_ERROR(-20140, 'Ошибка доступа: Недостаточно прав.');
END IF;
        UPDATE Movies SET Poster = EMPTY_BLOB() WHERE Id = p_MovieId
RETURNING Poster INTO dest_lob;
        IF dest_lob IS NULL THEN RAISE_APPLICATION_ERROR(-20114, 'Фильм
не найден.'); END IF;
        src_bfile := BFILENAME(p_DirectoryName, p_FileName);
        DBMS_LOB.FILEOPEN(src_bfile, DBMS_LOB.FILE_READONLY);
                    DBMS_LOB.LOADBLOBFROMFILE(dest_lob,    src_bfile,
DBMS_LOB.GETLENGTH(src_bfile), dest_offset, src_offset);
        DBMS_LOB.FILECLOSE(src_bfile);
        COMMIT;
          DBMS_OUTPUT.PUT_LINE('Постер для фильма ID '||p_MovieId||'
загружен.');
    EXCEPTION
        WHEN NO_DATA_FOUND THEN RAISE_APPLICATION_ERROR(-20301, 'Вы не
авторизованы.');
        WHEN OTHERS THEN ROLLBACK; RAISE;
    END UpdateMoviePoster;
```

```
    PROCEDURE UpdateMovieTrailer (p_CurrentUserId IN Users.Id%TYPE,
p_MovieId IN Movies.Id%TYPE, p_DirectoryName IN VARCHAR2, p_FileName IN
VARCHAR2) IS
         v_CurrentUserRole Roles.Name%TYPE; dest_lob BLOB; src_bfile
BFILE; dest_offset INTEGER := 1; src_offset INTEGER := 1;
    BEGIN
        SELECT r.Name INTO v_CurrentUserRole FROM Users u JOIN Roles r
ON u.RoleId = r.Id WHERE u.Id = p_CurrentUserId;
        IF v_CurrentUserRole NOT IN ('Администратор', 'Менеджер') THEN
RAISE_APPLICATION_ERROR(-20140, 'Ошибка доступа: Недостаточно прав.');
END IF;
        UPDATE Movies SET Trailer = EMPTY_BLOB() WHERE Id = p_MovieId
RETURNING Trailer INTO dest_lob;
       IF dest_lob IS NULL THEN RAISE_APPLICATION_ERROR(-20114, 'Фильм
не найден.'); END IF;
        src_bfile := BFILENAME(p_DirectoryName, p_FileName);
        DBMS_LOB.FILEOPEN(src_bfile, DBMS_LOB.FILE_READONLY);
                    DBMS_LOB.LOADBLOBFROMFILE(dest_lob,    src_bfile,
DBMS_LOB.GETLENGTH(src_bfile), dest_offset, src_offset);
        DBMS_LOB.FILECLOSE(src_bfile);
        COMMIT;
         DBMS_OUTPUT.PUT_LINE('Трейлер для фильма ID '||p_MovieId||'
загружен.');
    EXCEPTION
        WHEN NO_DATA_FOUND THEN RAISE_APPLICATION_ERROR(-20301, 'Вы не
авторизованы.');
        WHEN OTHERS THEN ROLLBACK; RAISE;
    END UpdateMovieTrailer;

    PROCEDURE AddSchedule (p_CurrentUserId IN Users.Id%TYPE, p_MovieId
IN   Schedules.MovieId%TYPE,   p_HallId   IN   Schedules.HallId%TYPE,
p_ShowTime    IN    Schedules.ShowTime%TYPE,    p_NewScheduleId    OUT
Schedules.Id%TYPE) IS
                v_CurrentUserRole   Roles.Name%TYPE;   v_MovieDuration
Movies.Duration%TYPE;   v_NewSessionEnd   TIMESTAMP;   v_ConflictCount
NUMBER;
    BEGIN
        SELECT r.Name INTO v_CurrentUserRole FROM Users u JOIN Roles r
ON u.RoleId = r.Id WHERE u.Id = p_CurrentUserId;
        IF v_CurrentUserRole NOT IN ('Администратор', 'Менеджер') THEN
RAISE_APPLICATION_ERROR(-20140, 'Ошибка доступа: Недостаточно прав.');
END IF;
         SELECT Duration INTO v_MovieDuration FROM Movies WHERE Id =
p_MovieId;
       v_NewSessionEnd := p_ShowTime + NUMTODSINTERVAL(v_MovieDuration,
'MINUTE');
       SELECT COUNT(*) INTO v_ConflictCount FROM Schedules s JOIN Movies
m ON s.MovieId = m.Id WHERE s.HallId = p_HallId AND (p_ShowTime <
(s.ShowTime    +    NUMTODSINTERVAL(m.Duration,    'MINUTE')))    AND
(v_NewSessionEnd > s.ShowTime);
       IF v_ConflictCount > 0 THEN RAISE_APPLICATION_ERROR(-20132, 'Зал
уже занят в это время.'); END IF;
```

```
          INSERT INTO Schedules (MovieId, HallId, ShowTime) VALUES
(p_MovieId, p_HallId, p_ShowTime) RETURNING Id INTO p_NewScheduleId;
        COMMIT;
        DBMS_OUTPUT.PUT_LINE('Сеанс создан с ID: '||p_NewScheduleId);
    EXCEPTION
        WHEN NO_DATA_FOUND THEN RAISE_APPLICATION_ERROR(-20114, 'Фильм
или зал не найдены.');
        WHEN OTHERS THEN ROLLBACK; RAISE;
    END AddSchedule;

     PROCEDURE UpdateScheduleTime (p_CurrentUserId IN Users.Id%TYPE,
p_ScheduleId IN Schedules.Id%TYPE, p_NewShowTime IN
Schedules.ShowTime%TYPE) IS
                    v_CurrentUserRole Roles.Name%TYPE; v_MovieId
Schedules.MovieId%TYPE; v_HallId Schedules.HallId%TYPE; v_MovieDuration
Movies.Duration%TYPE; v_NewSessionEnd TIMESTAMP; v_ConflictCount
NUMBER;
    BEGIN
        SELECT r.Name INTO v_CurrentUserRole FROM Users u JOIN Roles r
ON u.RoleId = r.Id WHERE u.Id = p_CurrentUserId;
        IF v_CurrentUserRole NOT IN ('Администратор', 'Менеджер') THEN
RAISE_APPLICATION_ERROR(-20140, 'Ошибка доступа: Недостаточно прав.');
END IF;
        SELECT MovieId, HallId INTO v_MovieId, v_HallId FROM Schedules
WHERE Id = p_ScheduleId;
         SELECT Duration INTO v_MovieDuration FROM Movies WHERE Id =
v_MovieId;
                        v_NewSessionEnd := p_NewShowTime +
NUMTODSINTERVAL(v_MovieDuration, 'MINUTE');
       SELECT COUNT(*) INTO v_ConflictCount FROM Schedules s JOIN Movies
m ON s.MovieId = m.Id WHERE s.HallId = v_HallId AND s.Id != p_ScheduleId
AND (p_NewShowTime < (s.ShowTime + NUMTODSINTERVAL(m.Duration,
'MINUTE'))) AND (v_NewSessionEnd > s.ShowTime);
       IF v_ConflictCount > 0 THEN RAISE_APPLICATION_ERROR(-20132, 'Зал
уже занят в новое время.'); END IF;
          UPDATE Schedules SET ShowTime = p_NewShowTime WHERE Id =
p_ScheduleId;
        COMMIT;
           DBMS_OUTPUT.PUT_LINE('Время сеанса ID '||p_ScheduleId||'
обновлено.');
    EXCEPTION
        WHEN NO_DATA_FOUND THEN RAISE_APPLICATION_ERROR(-20115, 'Сеанс
или связанный фильм не найдены.');
        WHEN OTHERS THEN ROLLBACK; RAISE;
    END UpdateScheduleTime;

      PROCEDURE DeleteSchedule (p_CurrentUserId IN Users.Id%TYPE,
p_ScheduleId IN Schedules.Id%TYPE) IS
        v_CurrentUserRole Roles.Name%TYPE;
    BEGIN
        SELECT r.Name INTO v_CurrentUserRole FROM Users u JOIN Roles r
ON u.RoleId = r.Id WHERE u.Id = p_CurrentUserId;
```

```
        IF v_CurrentUserRole NOT IN ('Администратор', 'Менеджер') THEN
RAISE_APPLICATION_ERROR(-20140, 'Ошибка доступа: Недостаточно прав.');
END IF;
        DELETE FROM Schedules WHERE Id = p_ScheduleId;
        IF SQL%ROWCOUNT = 0 THEN RAISE_APPLICATION_ERROR(-20115, 'Сеанс
не найден.'); END IF;
        COMMIT;
         DBMS_OUTPUT.PUT_LINE('Сеанс ID '||p_ScheduleId||' и связанные
заказы удалены.');
    EXCEPTION
        WHEN NO_DATA_FOUND THEN RAISE_APPLICATION_ERROR(-20301, 'Вы не
авторизованы.');
        WHEN OTHERS THEN ROLLBACK; RAISE;
    END DeleteSchedule;

      PROCEDURE UpdateOrderStatus (p_CurrentUserId IN Users.Id%TYPE,
p_OrderId IN Orders.Id%TYPE, p_NewStatus IN Orders.OrderStatus%TYPE) IS
        v_CurrentUserRole Roles.Name%TYPE;
    BEGIN
        SELECT r.Name INTO v_CurrentUserRole FROM Users u JOIN Roles r
ON u.RoleId = r.Id WHERE u.Id = p_CurrentUserId;
        IF v_CurrentUserRole NOT IN ('Администратор', 'Менеджер') THEN
RAISE_APPLICATION_ERROR(-20140, 'Ошибка доступа: Недостаточно прав.');
END IF;
          IF p_NewStatus NOT IN ('Оплачен', 'Отменен', 'Возвращен',
'Забронирован') THEN RAISE_APPLICATION_ERROR(-20150, 'Недопустимый
статус.'); END IF;
       UPDATE Orders SET OrderStatus = p_NewStatus WHERE Id = p_OrderId;
        IF SQL%ROWCOUNT = 0 THEN RAISE_APPLICATION_ERROR(-20151, 'Заказ
не найден.'); END IF;
        COMMIT;
             DBMS_OUTPUT.PUT_LINE('Статус заказа ID '||p_OrderId||'
изменен.');
    EXCEPTION
        WHEN NO_DATA_FOUND THEN RAISE_APPLICATION_ERROR(-20301, 'Вы не
авторизованы.');
        WHEN OTHERS THEN ROLLBACK; RAISE;
    END UpdateOrderStatus;

    FUNCTION GetAllOrders (p_CurrentUserId IN Users.Id%TYPE, p_StartDate
IN DATE DEFAULT NULL, p_EndDate IN DATE DEFAULT NULL, p_Status IN
Orders.OrderStatus%TYPE DEFAULT NULL) RETURN SYS_REFCURSOR IS
       v_CurrentUserRole Roles.Name%TYPE; v_ResultCursor SYS_REFCURSOR;
    BEGIN
        SELECT r.Name INTO v_CurrentUserRole FROM Users u JOIN Roles r
ON u.RoleId = r.Id WHERE u.Id = p_CurrentUserId;
        IF v_CurrentUserRole NOT IN ('Администратор', 'Менеджер') THEN
RAISE_APPLICATION_ERROR(-20140, 'Ошибка доступа: Недостаточно прав.');
END IF;
       OPEN v_ResultCursor FOR SELECT o.Id, o.BookingTimestamp, u.Email,
m.Title, o.Seats, o.TotalPrice, o.OrderStatus FROM Orders o JOIN Users
u ON o.UserId = u.Id JOIN Schedules s ON o.ScheduleId = s.Id JOIN Movies
m ON s.MovieId = m.Id WHERE (TRUNC(o.BookingTimestamp) >= p_StartDate
```

```
OR p_StartDate IS NULL) AND (TRUNC(o.BookingTimestamp) <= p_EndDate OR
p_EndDate IS NULL) AND (o.OrderStatus = p_Status OR p_Status IS NULL)
ORDER BY o.BookingTimestamp DESC;
        RETURN v_ResultCursor;
    EXCEPTION
         WHEN NO_DATA_FOUND THEN RAISE_APPLICATION_ERROR(-20301, 'Вы не
авторизованы.');
        WHEN OTHERS THEN RAISE;
    END GetAllOrders;

PROCEDURE RegisterUser (p_Name IN Users.Name%TYPE, p_Surname IN
Users.Surname%TYPE, p_Email IN Users.Email%TYPE, p_Password IN
Users.Password%TYPE, p_NewUserId OUT Users.Id%TYPE) IS
        v_DefaultRoleId Roles.Id%TYPE;
    BEGIN
             IF p_Email IS NULL OR p_Password IS NULL THEN
RAISE_APPLICATION_ERROR(-20004, 'Логин и пароль не могут быть
пустыми.'); END IF;
          SELECT Id INTO v_DefaultRoleId FROM Roles WHERE Name =
'Пользователь';
       INSERT INTO Users (Name, Surname, Email, Password, RoleId) VALUES
(p_Name, p_Surname, p_Email, p_Password, v_DefaultRoleId) RETURNING Id
INTO p_NewUserId;
        COMMIT;
       DBMS_OUTPUT.PUT_LINE('Пользователь '||p_Email||' зарегистрирован
с ID: '||p_NewUserId);
    EXCEPTION
            WHEN NO_DATA_FOUND THEN RAISE_APPLICATION_ERROR(-20004,
'Критическая ошибка: Роль "Пользователь" не найдена.');
          WHEN DUP_VAL_ON_INDEX THEN RAISE_APPLICATION_ERROR(-20001,
'Пользователь с таким логином уже существует.');
        WHEN OTHERS THEN ROLLBACK; RAISE;
    END RegisterUser;

     FUNCTION LoginUser (p_Email IN Users.Email%TYPE, p_Password IN
Users.Password%TYPE) RETURN NUMBER IS
        v_UserId Users.Id%TYPE := NULL;
    BEGIN
         SELECT Id INTO v_UserId FROM Users WHERE Email = p_Email AND
Password = p_Password;
        RETURN v_UserId;
    EXCEPTION
            WHEN NO_DATA_FOUND THEN RAISE_APPLICATION_ERROR(-20002,
'Неверный логин или пароль.');
        WHEN OTHERS THEN RAISE_APPLICATION_ERROR(-20003, 'Пользователь
не найден (непредвиденная ошибка).');
    END LoginUser;

    PROCEDURE UpdateUserEmail (
            p_CurrentUserId    IN Users.Id%TYPE,
            p_NewEmail         IN Users.Email%TYPE
        ) IS
        BEGIN
```

```
            UPDATE Users
            SET Email = p_NewEmail
            WHERE Id = p_CurrentUserId;

            IF SQL%ROWCOUNT = 0 THEN
                  RAISE_APPLICATION_ERROR(-20301, 'Вы не авторизованы
(пользователь не найден).');
            END IF;

            COMMIT;
             DBMS_OUTPUT.PUT_LINE('Ваш email успешно обновлен на ' ||
p_NewEmail);
        EXCEPTION
            WHEN DUP_VAL_ON_INDEX THEN
                 RAISE_APPLICATION_ERROR(-20001, 'Пользователь с таким
логином (email) уже существует.');
            WHEN OTHERS THEN
                ROLLBACK;
                RAISE;
    END UpdateUserEmail;

    FUNCTION GetOrderHistory (
        p_CurrentUserId   IN Users.Id%TYPE,
        p_TargetUserId    IN Users.Id%TYPE,
        p_FilterStartDate IN DATE DEFAULT NULL,
        p_FilterEndDate   IN DATE DEFAULT NULL,
        p_FilterMinSum    IN Orders.TotalPrice%TYPE DEFAULT NULL,
        p_FilterMaxSum    IN Orders.TotalPrice%TYPE DEFAULT NULL,
        p_FilterStatus    IN Orders.OrderStatus%TYPE DEFAULT NULL
    ) RETURN SYS_REFCURSOR IS
        v_CurrentUserRole Roles.Name%TYPE;
        v_ResultCursor  SYS_REFCURSOR;
    BEGIN
        SELECT r.Name INTO v_CurrentUserRole FROM Users u JOIN Roles r
ON u.RoleId = r.Id WHERE u.Id = p_CurrentUserId;
         IF v_CurrentUserRole NOT IN ('Администратор', 'Менеджер') AND
p_CurrentUserId != p_TargetUserId THEN
            RAISE_APPLICATION_ERROR(-20310, 'Вы не можете просматривать
чужой заказ.');
        END IF;

        OPEN v_ResultCursor FOR
               SELECT  o.Id,  o.BookingTimestamp,  m.Title,  s.ShowTime,
o.Seats, o.TotalPrice, o.OrderStatus
            FROM Orders o
            JOIN Schedules s ON o.ScheduleId = s.Id
            JOIN Movies m ON s.MovieId = m.Id
            WHERE
                o.UserId = p_TargetUserId
                 AND (TRUNC(o.BookingTimestamp) >= p_FilterStartDate OR
p_FilterStartDate IS NULL)
                  AND (TRUNC(o.BookingTimestamp) <= p_FilterEndDate OR
p_FilterEndDate IS NULL)
```

```
                AND (o.TotalPrice >= p_FilterMinSum OR p_FilterMinSum
IS NULL)
                AND (o.TotalPrice <= p_FilterMaxSum OR p_FilterMaxSum
IS NULL)
                AND (o.OrderStatus = p_FilterStatus OR p_FilterStatus
IS NULL)
            ORDER BY o.BookingTimestamp DESC;
        RETURN v_ResultCursor;
    EXCEPTION
        WHEN NO_DATA_FOUND THEN RAISE_APPLICATION_ERROR(-20301, 'Вы не
авторизованы.');
        WHEN OTHERS THEN RAISE;
    END GetOrderHistory;

       PROCEDURE CreateOrder (p_CurrentUserId IN Users.Id%TYPE,
p_ScheduleId IN Orders.ScheduleId%TYPE, p_Seats IN Orders.Seats%TYPE,
p_TotalPrice IN Orders.TotalPrice%TYPE, p_NewOrderId OUT
Orders.Id%TYPE) IS
        v_HallCapacity Halls.Capacity%TYPE;
        v_TicketsSold NUMBER;
        v_UserTicketsCount NUMBER;
        v_SeatOccupied NUMBER;
        v_TicketLimitPerUser CONSTANT NUMBER := 5;
    BEGIN
       SELECT COUNT(*) INTO v_UserTicketsCount FROM Orders WHERE UserId
= p_CurrentUserId AND ScheduleId = p_ScheduleId AND OrderStatus !=
'Отменен';
             IF v_UserTicketsCount >= v_TicketLimitPerUser THEN
RAISE_APPLICATION_ERROR(-20330, 'Количество билетов превышает лимит
('||v_TicketLimitPerUser||').'); END IF;

         SELECT h.Capacity INTO v_HallCapacity FROM Schedules s JOIN
Halls h ON s.HallId = h.Id WHERE s.Id = p_ScheduleId;
        SELECT COUNT(*) INTO v_TicketsSold FROM Orders WHERE ScheduleId
= p_ScheduleId AND OrderStatus != 'Отменен';
                   IF v_TicketsSold >= v_HallCapacity THEN
RAISE_APPLICATION_ERROR(-20331, 'Все билеты на сеанс проданы.'); END IF;

        SELECT COUNT(*) INTO v_SeatOccupied
        FROM Orders
        WHERE ScheduleId = p_ScheduleId
            AND Seats = p_Seats
            AND OrderStatus != 'Отменен';

        IF v_SeatOccupied > 0 THEN
                RAISE_APPLICATION_ERROR(-20332, 'Выбранное место
('||p_Seats||') уже занято.');
        END IF;

          INSERT INTO Orders (Seats, TotalPrice, UserId, ScheduleId,
OrderStatus) VALUES (p_Seats, p_TotalPrice, p_CurrentUserId,
p_ScheduleId, 'Оплачен') RETURNING Id INTO p_NewOrderId;
        COMMIT;
```

```
            DBMS_OUTPUT.PUT_LINE('Заказ  '||p_NewOrderId||'  создан  для
пользователя ID='||p_CurrentUserId);
        EXCEPTION
         WHEN NO_DATA_FOUND THEN RAISE_APPLICATION_ERROR(-20322, 'Сеанс
или зал не найдены.');
             WHEN  OTHERS  THEN  ROLLBACK;  IF  SQLCODE  =  -2291  THEN
RAISE_APPLICATION_ERROR(-20300,    'Для    оформления    заказа    нужно
авторизоваться.'); ELSE RAISE; END IF;
    END CreateOrder;

    PROCEDURE CancelOrder (p_CurrentUserId IN Users.Id%TYPE, p_OrderId
IN Orders.Id%TYPE) IS
       v_CurrentUserRole Roles.Name%TYPE; v_OrderOwnerId Users.Id%TYPE;
v_ShowTime            Schedules.ShowTime%TYPE;            v_OrderStatus
Orders.OrderStatus%TYPE; v_HoursBeforeShow NUMBER;
    BEGIN
         SELECT r.Name INTO v_CurrentUserRole FROM Users u JOIN Roles r
ON u.RoleId = r.Id WHERE u.Id = p_CurrentUserId;
        SELECT o.UserId, o.OrderStatus, s.ShowTime INTO v_OrderOwnerId,
v_OrderStatus, v_ShowTime FROM Orders o JOIN Schedules s ON o.ScheduleId
= s.Id WHERE o.Id = p_OrderId;
         IF v_CurrentUserRole NOT IN ('Администратор', 'Менеджер') AND
p_CurrentUserId != v_OrderOwnerId THEN RAISE_APPLICATION_ERROR(-20320,
'Нет прав на отмену этого заказа.'); END IF;
          IF  v_OrderStatus  =  'Отменен'  THEN  RAISE_APPLICATION_ERROR(-
20321, 'Заказ уже отменен.'); END IF;
              v_HoursBeforeShow   :=   (CAST(v_ShowTime   AS   DATE)   -
CAST(SYSTIMESTAMP AS DATE)) * 24;
         IF v_CurrentUserRole NOT IN ('Администратор', 'Менеджер') AND
v_HoursBeforeShow  <=  2  THEN  RAISE_APPLICATION_ERROR(-20321,  'Отмена
невозможна: до сеанса менее 2 часов.'); END IF;
        UPDATE Orders SET OrderStatus = 'Отменен' WHERE Id = p_OrderId;
        COMMIT;
        DBMS_OUTPUT.PUT_LINE('Заказ '||p_OrderId||' успешно отменен.');
    EXCEPTION
             WHEN  NO_DATA_FOUND  THEN  RAISE_APPLICATION_ERROR(-20322,
'Пользователь или заказ/сеанс не найдены.');
        WHEN OTHERS THEN ROLLBACK; RAISE;
    END CancelOrder;

    FUNCTION FindMovies (
        p_TitleQuery IN Movies.Title%TYPE DEFAULT NULL,
        p_GenreQuery IN Movies.Genre%TYPE DEFAULT NULL,
        p_SortBy     IN VARCHAR2 DEFAULT 'rating'
    ) RETURN SYS_REFCURSOR IS
        v_ResultCursor SYS_REFCURSOR;
        BEGIN
        OPEN v_ResultCursor FOR
             SELECT Id, Title, Director, Description, Genre, Duration,
Rating, AgeRestriction
            FROM Movies
            WHERE
```

```
                (LOWER(Title) LIKE '%' || LOWER(p_TitleQuery) || '%' OR
p_TitleQuery IS NULL)
               AND (LOWER(Genre) LIKE '%' || LOWER(p_GenreQuery) || '%'
OR p_GenreQuery IS NULL)
            ORDER BY
                CASE WHEN p_SortBy = 'title' THEN Title END ASC,
                CASE WHEN p_SortBy = 'rating' THEN Rating END DESC,
                Title ASC;
        RETURN v_ResultCursor;
    END FindMovies;

     FUNCTION GetUserIdByEmail (p_Email IN Users.Email%TYPE) RETURN
NUMBER IS
        v_UserId Users.Id%TYPE;
    BEGIN
        SELECT Id INTO v_UserId FROM Users WHERE Email = p_Email;
        RETURN v_UserId;
    EXCEPTION
        WHEN NO_DATA_FOUND THEN RETURN NULL;
    END GetUserIdByEmail;

     FUNCTION GetMovieIdByTitle (p_Title IN Movies.Title%TYPE) RETURN
NUMBER IS
        v_MovieId Movies.Id%TYPE;
    BEGIN
        SELECT Id INTO v_MovieId FROM Movies WHERE Title = p_Title;
        RETURN v_MovieId;
    EXCEPTION
        WHEN NO_DATA_FOUND THEN RETURN NULL;
    END GetMovieIdByTitle;

    FUNCTION GetHallIdByName (p_Name IN Halls.Name%TYPE) RETURN NUMBER
IS
        v_HallId Halls.Id%TYPE;
    BEGIN
        SELECT Id INTO v_HallId FROM Halls WHERE Name = p_Name;
        RETURN v_HallId;
    EXCEPTION
        WHEN NO_DATA_FOUND THEN RETURN NULL;
    END GetHallIdByName;

     FUNCTION GetScheduleId (p_MovieId IN NUMBER, p_HallId IN NUMBER,
p_ShowTime IN TIMESTAMP) RETURN NUMBER IS
        v_ScheduleId Schedules.Id%TYPE;
    BEGIN
          SELECT Id INTO v_ScheduleId FROM Schedules WHERE MovieId =
p_MovieId AND HallId = p_HallId AND ShowTime = p_ShowTime;
        RETURN v_ScheduleId;
    EXCEPTION
        WHEN NO_DATA_FOUND THEN RETURN NULL;
    END GetScheduleId;
```

```
    FUNCTION GetLastOrderId (p_CurrentUserId IN NUMBER) RETURN NUMBER
IS
        v_OrderId Orders.Id%TYPE;
    BEGIN
        SELECT Id
        INTO v_OrderId
        FROM Orders
        WHERE UserId = p_CurrentUserId
        ORDER BY BookingTimestamp DESC
        FETCH FIRST 1 ROW ONLY;

        RETURN v_OrderId;
    EXCEPTION
        WHEN NO_DATA_FOUND THEN
            RETURN NULL;
    END GetLastOrderId;
```

Приложение Е

```
FUNCTION ExportOrdersToJSON (p_Limit IN NUMBER DEFAULT NULL) RETURN
CLOB IS
    v_JsonData CLOB;
    v_Cursor SYS_REFCURSOR;
    v_RowCount NUMBER;
    v_LF CHAR(1) := CHR(10);
    BEGIN
    SELECT COUNT(*) INTO v_RowCount FROM Orders;

    DBMS_LOB.CREATETEMPORARY(v_JsonData, TRUE);

    DBMS_LOB.WRITEAPPEND(v_JsonData,
                        LENGTH('{"export_details":{"timestamp":"'     ||
TO_CHAR(SYSTIMESTAMP,          'YYYY-MM-DD"T"HH24:MI:SS.FF')          ||
'","row_count":' || v_RowCount || '},"orders":[' || v_LF),
                                '{"export_details":{"timestamp":"'    ||
TO_CHAR(SYSTIMESTAMP,          'YYYY-MM-DD"T"HH24:MI:SS.FF')          ||
'","row_count":' || v_RowCount || '},"orders":[' || v_LF);

     OPEN v_Cursor FOR 'SELECT JSON_OBJECT (''seats'' VALUE o.Seats,
''total_price'' VALUE o.TotalPrice, ''status'' VALUE o.OrderStatus,
''user_id'' VALUE o.UserId, ''schedule_id'' VALUE o.ScheduleId) FROM
(SELECT * FROM Orders ORDER BY BookingTimestamp DESC FETCH FIRST
NVL(:lim, 999999999) ROWS ONLY) o' USING p_Limit;

    DECLARE
        v_JsonObject VARCHAR2(4000);
        v_IsFirst BOOLEAN := TRUE;
    BEGIN
        LOOP
                    FETCH   v_Cursor   INTO   v_JsonObject;   EXIT   WHEN
v_Cursor%NOTFOUND;

            IF NOT v_IsFirst THEN
                DBMS_LOB.WRITEAPPEND(v_JsonData, 2, ',' || v_LF);
            END IF;

            DBMS_LOB.WRITEAPPEND(v_JsonData, 2, '  ');

               DBMS_LOB.WRITEAPPEND(v_JsonData,  LENGTH(v_JsonObject),
v_JsonObject);

            v_IsFirst := FALSE;
        END LOOP;
    END;
    CLOSE v_Cursor;

    DBMS_LOB.WRITEAPPEND(v_JsonData, 3, v_LF || ']}');

    RETURN v_JsonData;
    END ExportOrdersToJSON;
```

Приложение Ж

```
PROCEDURE ImportOrdersFromJSON_File (p_CurrentUserId IN Users.Id%TYPE, p_DirectoryName IN VARCHAR2, p_FileName IN VARCHAR2) IS
         v_UserRole Roles.Name%TYPE; v_JsonData CLOB; v_RowsInserted NUMBER;
    BEGIN
         SELECT r.Name INTO v_UserRole FROM Users u JOIN Roles r ON u.RoleId = r.Id WHERE u.Id = p_CurrentUserId;
                IF v_UserRole NOT IN ('Администратор') THEN RAISE_APPLICATION_ERROR(-20140, 'Доступ запрещен.'); END IF;
        DECLARE
           lob_loc BFILE; dest_offset INTEGER := 1; src_offset INTEGER := 1; lang_ctx INTEGER := DBMS_LOB.DEFAULT_LANG_CTX; warning INTEGER;
        BEGIN
            DBMS_LOB.CREATETEMPORARY(v_JsonData, TRUE);
            lob_loc := BFILENAME(p_DirectoryName, p_FileName);
            DBMS_LOB.FILEOPEN(lob_loc, DBMS_LOB.FILE_READONLY);
                    DBMS_LOB.LOADCLOBFROMFILE(v_JsonData, lob_loc, DBMS_LOB.GETLENGTH(lob_loc), dest_offset, src_offset, DBMS_LOB.DEFAULT_CSID, lang_ctx, warning);
            DBMS_LOB.FILECLOSE(lob_loc);
        END;
         INSERT INTO Orders (Seats, TotalPrice, OrderStatus, UserId, ScheduleId)
           SELECT j.Seats, j.TotalPrice, 'Импортирован', j.UserId, j.ScheduleId
        FROM JSON_TABLE(v_JsonData, '$.orders[*]'
          COLUMNS (Seats NVARCHAR2(255) PATH '$.seats', TotalPrice NUMBER(10, 2) PATH '$.total_price', UserId NUMBER PATH '$.user_id', ScheduleId NUMBER PATH '$.schedule_id')) j WHERE EXISTS (SELECT 1 FROM Users WHERE Id = j.UserId)
        AND EXISTS (SELECT 1 FROM Schedules WHERE Id = j.ScheduleId);
        v_RowsInserted := SQL%ROWCOUNT;
        COMMIT;
                DBMS_OUTPUT.PUT_LINE(v_RowsInserted || ' заказов импортировано.');
        DBMS_LOB.FREETEMPORARY(v_JsonData);
    EXCEPTION
        WHEN NO_DATA_FOUND THEN RAISE_APPLICATION_ERROR(-20301, 'Вы не авторизованы.');
        WHEN OTHERS THEN ROLLBACK; RAISE;
    END ImportOrdersFromJSON_File;
```

Приложение 3

```
    PROCEDURE GenerateTestOrders (p_NumberOfOrders IN NUMBER) IS
              v_TestUserId          Users.Id%TYPE;   v_TestScheduleId
Schedules.Id%TYPE;
    BEGIN
         BEGIN INSERT INTO Roles (Name) VALUES ('Тестовый'); COMMIT;
EXCEPTION WHEN DUP_VAL_ON_INDEX THEN NULL; END;
       BEGIN INSERT INTO Users (Name, Surname, Email, Password, RoleId)
VALUES ('TestGen', 'User', 'testgen@user.com', '123', (SELECT Id FROM
Roles   WHERE   Name   =   'Тестовый'));   COMMIT;   EXCEPTION   WHEN
DUP_VAL_ON_INDEX THEN NULL; END;
          BEGIN INSERT INTO Movies (Title) VALUES ('Test Movie for
Generation'); INSERT INTO Halls (Name, Capacity) VALUES ('Test Hall for
Generation', 200000); INSERT INTO Schedules (MovieId, HallId, ShowTime)
VALUES ((SELECT MAX(Id) FROM Movies WHERE Title = 'Test Movie for
Generation'), (SELECT MAX(Id) FROM Halls WHERE Name = 'Test Hall for
Generation'), SYSTIMESTAMP); COMMIT; EXCEPTION WHEN OTHERS THEN NULL;
END;
           SELECT  Id  INTO  v_TestUserId  FROM  Users  WHERE  Email  =
'testgen@user.com';
        SELECT MAX(Id) INTO v_TestScheduleId FROM Schedules;
        DBMS_OUTPUT.PUT_LINE('Начинаем генерацию ' || p_NumberOfOrders
|| ' заказов...');
        FOR i IN 1..p_NumberOfOrders LOOP
            INSERT INTO Orders (Seats, TotalPrice, UserId, ScheduleId,
OrderStatus) VALUES ('Ряд ' || TRUNC(DBMS_RANDOM.VALUE(1, 100)) || ',
Место       '       ||       TRUNC(DBMS_RANDOM.VALUE(1,       100)),
ROUND(DBMS_RANDOM.VALUE(10,  50),  2),  v_TestUserId,  v_TestScheduleId,
CASE WHEN MOD(i, 20) = 0 THEN 'Отменен' ELSE 'Оплачен' END);
           IF MOD(i, 10000) = 0 THEN COMMIT; DBMS_OUTPUT.PUT_LINE('...
добавлено ' || i || ' заказов'); END IF;
        END LOOP;
          COMMIT;  DBMS_OUTPUT.PUT_LINE('Генерация завершена. Успешно
добавлено ' || p_NumberOfOrders || ' заказов.');
    EXCEPTION
         WHEN OTHERS THEN DBMS_OUTPUT.PUT_LINE('Критическая ошибка: '
|| SQLERRM); ROLLBACK;
    END GenerateTestOrders;
```