МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

Учреждение образования «БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

Факультет Информационных технологий
Кафедра Информационные системы и технологии
Специальность 1-40 01 01 «Программное обеспечение информационных технологий»
Специализация Программирование интернет приложений

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА К КУРСОВОМУ ПРОЕКТУ НА ТЕМУ:

«База данных «Интернет-магазин алкогольных напитков» с реализацией технологии шифрования и маскирования БД»

Выполнил студент Шумский Евгений Сергеевич
(Ф.И.О.)

Руководитель проекта асс. Копыток Дарья Владимировна
(учен. степень, звание, должность, Ф.И.О., подпись)

Заведующий кафедрой к.т.н., доц. Смелов В.В .
(учен. степень, звание, должность, Ф.И.О., подпись)

Курсовой проект защищен с оценкой

Минск 2022

Оглавление

Введение

Целью данной работы была разработка реляционной базы данных на тему «Интернет-магазин алкогольных напитков». База данных должна быть составлена для работы онлайн магазина.

В связи с тем, что самым важным ресурсом современного мира становится информация, вполне закономерно развитие технологий в направлении хранения и управления данными.

Любое современное web-приложение невозможно представить без базы данных, работающей на основе одной из множества доступных СУБД.

Система управления базами данных (СУБД) — совокупность программных и лингвистических средств общего или специального назначения, обеспечивающих управление, созданием и использованием баз данных.

Основной функцией любой СУБД является поддержка независимости, целостности и непротиворечивости данных в условиях коллективного использования. Независимость данных понимается как способность СУБД создавать различные представления об одних и тех же хранимых данных, остающихся инвариантными к изменениям среды функционирования БД [25]. Требуемая степень независимости данных достигается в результате введения внешнего, концептуального и внутреннего уровней определения и манипулирования данными. С внешней точки зрения база данных - это совокупность различных информационных моделей ПО, предназначенных для информационных потребностей пользователей; с концептуальной - база данных есть общая модель ПО, обеспечивающая поддержку различных прикладных систем; с внутренней - база данных рассматривается как физическое представление данных в конкретной среде, используемой для хранения информации. Являясь информационной моделью ПО, база данных обеспечивает коллективное использование информации и необходимые условия для естественной эволюции существующих приложений ИС без их разрушения.

Благодаря концепции БД обеспечивается независимость описания предметной области и задач приложений от структур данных и методов их обработки, программ - от логической структуры базы данных, логической структуры данных - от методов их физической организации.

В информационных системах, использующих БД, можно:

- сделать программы ввода, модификации и поиска данных независимыми от программ содержательной обработки приложений;
- минимизировать объем хранимых данных путем сокращения их дублирования;
- избежать противоречий в хранимых данных;
- обеспечить сохранность и целостность информации:
- многократно использовать одни и те же данные различными прикладными программами;
- обеспечить гибкость и адаптивность структуры данных к изменяющимся информационным потребностям пользователей;

- поддерживать адекватность базы данных моделируемой ПО;
- обеспечить защиту данных от несанкционированного доступа.

Концепция БД позволяет создавать интегрированные информационные системы, поддерживающие сложные и разнообразные структуры объектов предметной области, содержащие большое число типов данных, значительные объемы фактографической или текстовой информации, а также сделать реальной задачу обеспечения высокой достоверности обработки и хранения больших объемов данных.

1. Постановка задачи и анализ прототипов

С развитием информационных технологий и интернета в частности начали появляться онлайн-магазины. Примечательной особенностью которых, в отличие от обычных магазинов, стало отсутствие необходимости физического присутствия покупателя в здании магазина. Таким образом они позволяют экономить огромное количество времени и сил покупателям.

Начнем рассмотрение аналогов с онлайн-магазина «Wine Style». Пример его интерфейса представлен на рисунке 1.1.

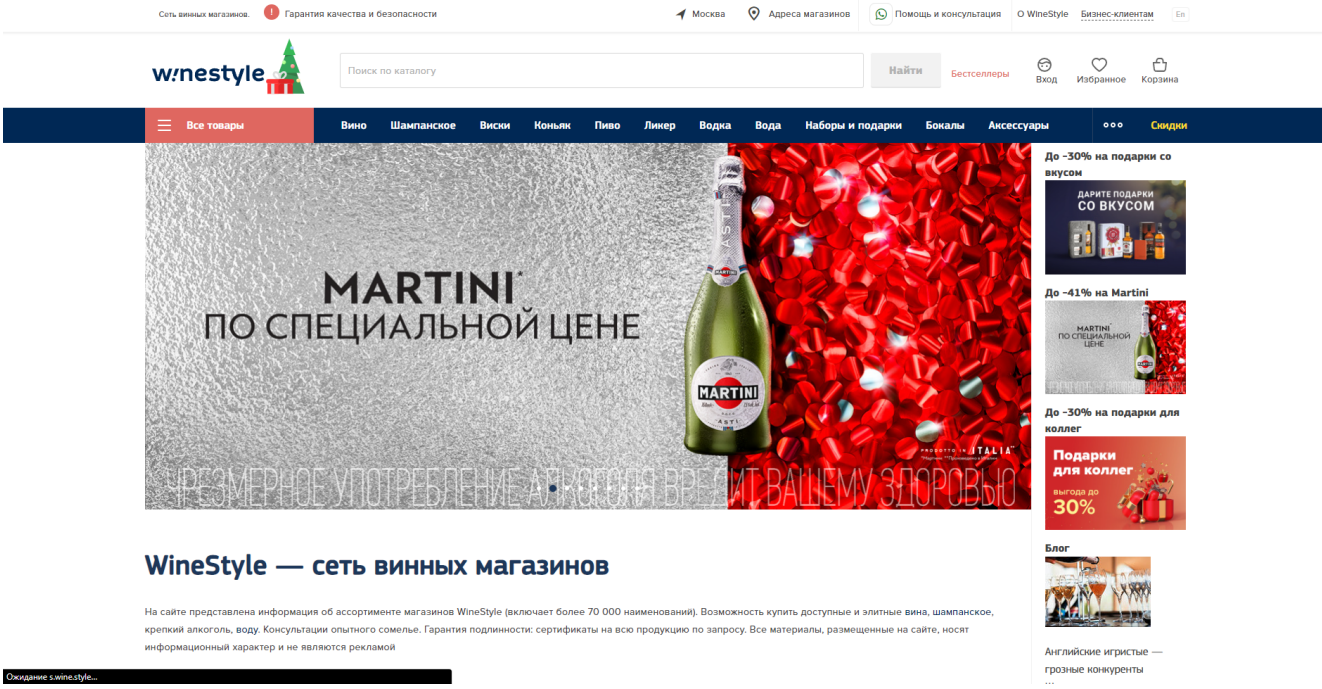


Рисунок 1.1 – Интерфейс сайта онлайн-магазина «WineStyle»

По итогам изучения аналогов были сделаны выводы, что разрабатываемая база данных должна обладать следующими возможностями:

- Регистрация и авторизация пользователя
- Добавление и удаление товара из корзины
- Оформление заказа
- Отслеживание состояния текущего заказа
- Просмотр истории заказов

2. Разработка базы данных

2.1 Проектирование базы данных

Процесс проектирования базы данных состоит из следующих этапов:

- сбор информации;
- определение сущностей;
- определение атрибутов для каждой сущности;
- определение связей между сущностями;
- нормализация;

На этапе сбора информации вам необходимо точно определить, как будет использоваться база данных, и какая информация будет в ней храниться.

Далее следует этап определения сущностей и на нем определяются сущности, из которых будет состоять база данных.

Сущность — это объект в базе данных, в котором хранятся данные. Сущность может представлять собой нечто вещественное (дом, человек, предмет, место) или абстрактное (банковская операция, отдел компании, маршрут автобуса). В физической модели сущность называется таблицей.

Сущности состоят из атрибутов (столбцов таблицы) и записей (строк в таблице).

Обычно базы данных состоят из нескольких основных сущностей, связанных с большим количеством подчиненных сущностей. Основные сущности называются независимыми: они не зависят ни от какой-либо другой сущности. Подчиненные сущности называются зависимыми: для того, чтобы существовала одна из них, должна существовать связанная с ней основная таблица.

Следующий этап — это определение атрибутов созданных сущностей. Атрибут представляет свойство, описывающее сущность. Атрибуты часто бывают числом, датой или текстом. Все данные, хранящиеся в атрибуте, должны иметь одинаковый тип и обладать одинаковыми свойствами.

В физической модели атрибуты называют колонками.

После определения сущностей необходимо определить все атрибуты этих сущностей.

На диаграммах атрибуты обычно перечисляются внутри прямоугольника сущности.

На этапе определения связей работа происходит с ключами сущностей. Ключом (key) называется набор атрибутов, однозначно определяющий запись. Ключи делятся на два класса: простые и составные.

Простой ключ состоит только из одного атрибута. Например, в базе «Паспорта граждан страны» номер паспорта будет простым ключом: ведь не бывает двух паспортов с одинаковым номером.

Составной ключ состоит из нескольких атрибутов. В той же базе «Паспорта граждан страны» может быть составной ключ со следующими атрибутами: фамилия, имя, отчество, дата рождения. Это — как пример, т. к. этот составной ключ, теоретически, не обеспечивает гарантированной уникальности записи.

Первичным ключом называется совокупность атрибутов, однозначно идентифицирующих запись в таблице (сущности). Один из возможных ключей

становится первичным ключом. На диаграммах первичные ключи часто изображаются выше основного списка атрибутов или выделяются специальными символами.

Любой возможный ключ, не являющийся первичным, называется альтернативным ключом. Сущность может иметь несколько альтернативных ключей.

Внешним ключом называется совокупность атрибутов, ссылающихся на первичный или альтернативный ключ другой сущности. Если внешний ключ не связан с первичной сущностью, то он может содержать только неопределенные значения. Если при этом ключ является составным, то все атрибуты внешнего ключа должны быть неопределенными.

На диаграммах атрибуты, объединяемые во внешние ключи, обозначаются специальными символами.

Реляционные базы данных позволяют объединять информацию, принадлежащую разным сущностям.

Отношение — это ситуация, при которой одна сущность ссылается на первичный ключ второй сущности. Как, например, сущности Дом и Хозяин на предыдущем рисунке.

Отношения определяются в процессе проектирования базы. Для этого следует проанализировать сущности и выявить логические связи, существующие между ними.

Тип отношения определяет количество записей сущности, связанных с записью другой сущности. Отношения делятся на три основных типа:

- Один-к-одному. Каждой записи первой сущности соответствует только одна запись из второй сущности. А каждой записи второй сущности соответствует только одна запись из первой сущности. Например, есть две сущности: Люди и Свидетельства о рождении. И у одного человека может быть только одно свидетельство о рождении.
- Один-ко-многим. Каждой записи первой сущности могут соответствовать несколько записей из второй сущности. Однако каждой записи второй сущности соответствует только одна запись из первой сущности. Например, есть две сущности: Заказ и Позиция заказа. И в одном заказе может быть много товаров.
- Многие-ко-многим. Каждой записи первой сущности могут соответствовать несколько записей из второй сущности. Однако и каждой записи второй сущности может соответствовать несколько записей из первой сущности. Например, есть две сущности: Автор и Книга. Один автор может написать много книг. Но у книги может быть несколько авторов.

По критерию обязательности отношения делятся на обязательные и необязательные.

- Обязательное отношение означает, что для каждой записи из первой сущности непременно должны присутствовать связанные записи во второй сущности.
- Необязательное отношение означает, что для записи из первой сущности может и не существовать записи во второй сущности.

Далее следует этап нормализации. Нормализацией называется процесс удаления избыточных данных из базы данных. Каждый элемент данных должен храниться в базе в одном и только в одном экземпляре. Существует пять распространенных форм нормализации. Как правило, база данных приводится к третьей нормальной форме.

В процессе нормализации выполняются определенные действия по удалению избыточных данных. Нормализация повышает быстродействие, ускоряет сортировку и построение индекса, уменьшает количество индексов на сущность, ускоряет операции вставки и обновления.

Нормализованная база данных обычно отличается большей гибкостью. При модификации запросов или сохраняемых данных в нормализованную базу обычно приходится вносить меньше изменений, а внесение изменений имеет меньше последствий.

Чтобы преобразовать сущность в первую нормальную форму, следует исключить повторяющиеся группы значений и добиться того, чтобы каждый атрибут содержал только одно значение, списки значений не допускаются. Другими словами, каждый атрибут, в сущности, должен храниться только в одном экземпляре.

Для соответствия второй нормальной форме сущности должны быть в первой нормальной форме. Таблица во второй нормальной форме содержит только те данные, которые к ней относятся. Значения не ключевых атрибутов сущности зависят от первичного ключа. Если более точно, то атрибуты зависят от первичного ключа, от всего первичного ключа и только от первичного ключа.

В третьей нормальной форме исключаются атрибуты, не зависящие от всего ключа. Любая сущность, находящаяся в третьей нормальной форме, находится также и во второй. Это самая распространенная форма базы данных. В третьей нормальной форме каждый атрибут зависит от ключа, от всего ключа и ни от чего, кроме ключа.

Ограничения (*constrains*) — это правила, за соблюдением которых следит система управления базы данных. Ограничения определяют множество значений, которые можно вводить в столбец или столбцы.

Организовав данные в таблицы и определив связи между ними, можно считать, что была создана модель, правильным образом отражающая бизнес-среду. Теперь нужно обеспечить, чтобы данные, вводимые в базу, давали правильное представление о состоянии дела. Иными словами, нужно обеспечить выполнение деловых правил и поддержку целостности (integrity) базы данных.

Например, ваша компания занимается доставкой книг. Вы вряд ли примете заказ от неизвестного клиента, ведь тогда вы даже не сможете доставить заказ. Отсюда бизнес-правило: заказы принимаются только от клиентов, информация о которых есть в базе данных.

Корректность данных в реляционных базах обеспечивается набором правил. Правила целостности данных делятся на четыре категории.

Для реализации необходимого функционала была создана база данных, таблицы которой можно разделить на 3 логические группы: таблицы для хранения

информации о продуктах, таблицы для хранения информации о заказах пользователей и таблицы для хранения пользовательских данных.

Рассмотрим таблицы базы данных на примере их диаграммы представленной на рисунке 2.1, а также в приложении А.

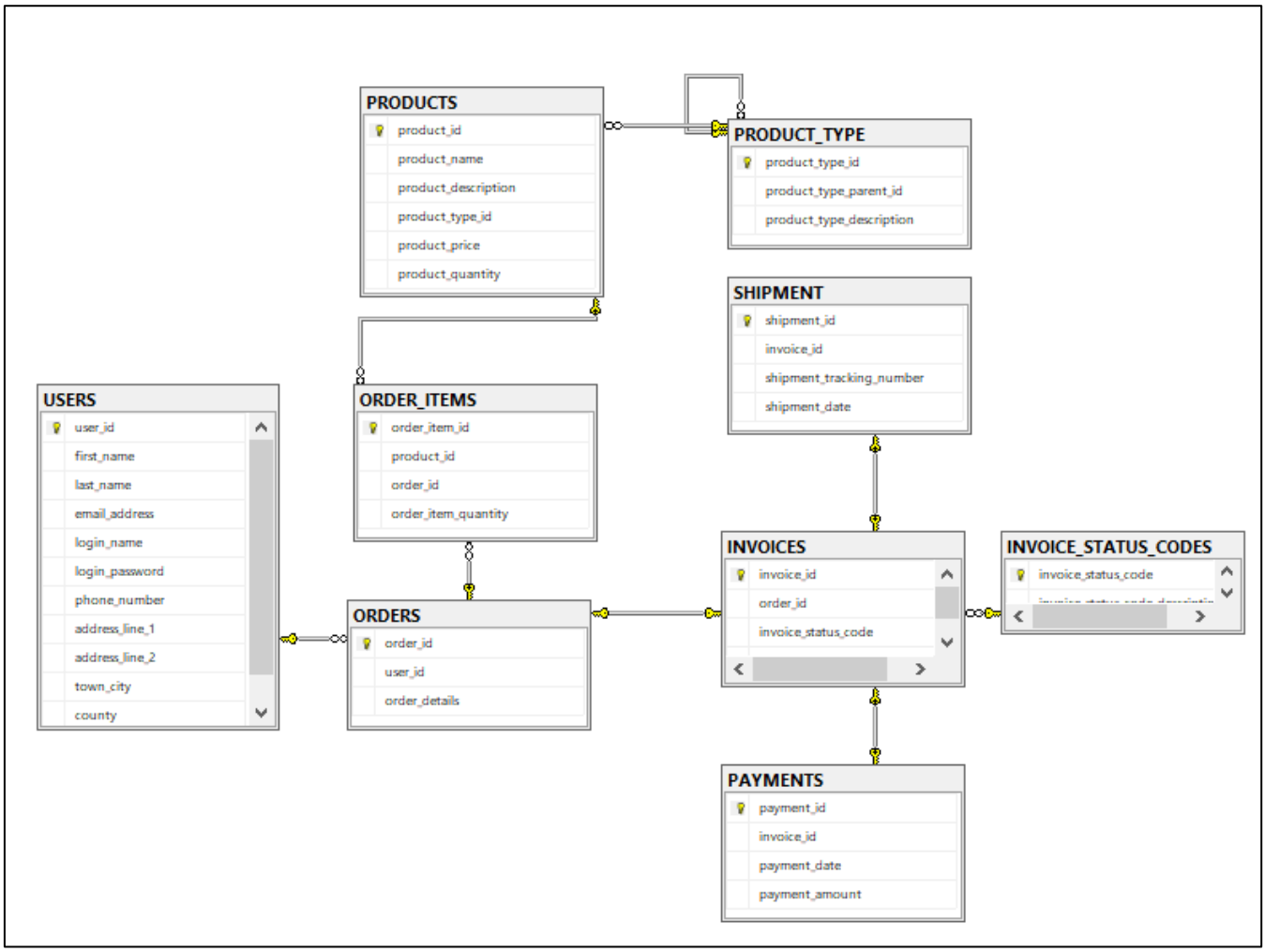


Рисунок 2.1 – Диаграмма таблиц базы данных

База данных состоит из 8 таблиц, связанных между собой внешними ключами.

Таблица «*USERS*» хранит в себе всю информацию о пользователях, включая пароль, логин, электронную почту и т.д.

Таблица «PRODUCTS» хранит информацию о продуктах, имеющихся в наличии, такую как: название продукта, описание продукта, тип продукта, цена продукта и его количество.

Таблица «*ORDERS*» хранит в себе информацию о заказах, включая номер заказа, идентификатор пользователя и описание заказа.

Таблица «ORDER_ITEMS» хранит в себе информацию о продуктах из каждого заказа, включая номер заказа, идентификатор продукта, его количество и т. д.

Таблица «*PRODUCT_TYPE*» содержит типы продукции и их описание. Типы могут наследовать другие типы таким образовывая свою иерархию.

Таблица «*INVOICES*» содержит информацию о оплаченных заказах, включая номер заказа, статус оплаты, дата оплаты и т. д.

Таблица «*INVOICE_STATUS_CODES*» содержит описание статусов оформления заказа.

Таблица «*PAYMENTS*» содержит информацию, связанную с оплатой заказа, включая дату оплаты, цену заказа и т. д.

Таблица «*SHIPMENT*» содержит информацию, связанную с доставкой заказа.

2.2 Процедуры для решения поставленных задач

Хранимые процедуры (*stored procedures*) — это предварительно откомпилированные процедуры, хранящиеся в базе данных. Хранимые процедуры можно использовать для определения деловых правил, с их помощью можно осуществлять более сложные вычисления, чем с помощью одних лишь ограничений.

Хранимые процедуры могут содержать логику хода выполнения программы, а также запросы к базе данных. Они могут принимать параметры и возвращать результаты в виде таблиц или одиночных значений.

Хранимые процедуры похожи на обычные процедуры или функции в любой программе.

При разработке курсового проекта было создано большое количество процедур для следующих целей:

1. Выборка данных из таблиц;
2. Выборка данных по поисковому запросу;
3. Заполнение таблиц 100 000 строк;
4. Вход в аккаунт;
5. Удаление данных из таблиц;
6. Добавление данных в таблицы;
7. Экспорт и импорт таблицы PRODUCTS в формате xml;

Наиболее используемые процедуры представлены в приложении Б.

2.3 Индексы

Системы баз данных обычно используют индексы для обеспечения быстрого доступа к реляционным данным. Индекс представляет собой отдельную физическую структуру данных, которая позволяет получать быстрый доступ к одной или нескольким строкам данных. Таким образом, правильная настройка индексов является ключевым аспектом улучшения производительности запросов.

Если для таблицы отсутствует подходящий индекс, для выборки строк система использует метод сканирования таблицы. Выражение сканирование таблицы означает, что система последовательно извлекает и исследует каждую строку таблицы (от первой до последней), и помещает строку в результирующий набор, если для нее удовлетворяется условие поиска в предложении WHERE. Таким образом, все строки извлекаются в соответствии с их физическим расположением в памяти. Этот метод менее эффективен, чем доступ с использованием индексов, как объясняется далее.

Индексы сохраняются в дополнительных структурах базы данных, называющихся страницами индексов. Для каждой индексируемой строки имеется элемент индекса (*index entry*), который сохраняется на странице индексов. Каждый элемент индекса состоит из ключа индекса и указателя. Вот поэтому элемент индекса значительно короче, чем строка таблицы, на которую он указывает. По этой причине количество элементов индекса на каждой странице индексов намного больше, чем количество строк в странице данных.

Кластеризованный индекс определяет физический порядок данных в таблице. Компонент *Database Engine* позволяет создавать для таблицы лишь один кластеризованный индекс, т.к. строки таблицы нельзя упорядочить физически более чем одним способом. Код создания индексов в курсовом проекте представлен ниже на рисунке 2.2.

```
-- USERS
create unique nonclustered index indx_login_name on USERS(login_name);
create unique nonclustered index indx_email_address on USERS(email_address);

-- PRODUCTS
create unique nonclustered index indx_product_name on PRODUCTS(product_name);
create unique nonclustered index indx_product_name_desc on PRODUCTS(product_name desc);
create nonclustered index indx_product_type on PRODUCTS(product_type_id);
create nonclustered index indx_product_price on PRODUCTS(product_price);
create nonclustered index indx_product_price_desc on PRODUCTS(product_price desc);
create nonclustered index indx_product_quantity on PRODUCTS(product_quantity);
create nonclustered index indx_product_quantity_desc on PRODUCTS(product_quantity desc);

-- PRODUCT_TYPE
create nonclustered index indx_product_type_parent on PRODUCT_TYPE(product_type_parent_id);

-- ORDERS
create nonclustered index indx_user_id on ORDERS(user_id);

-- ORDER_ITEMS
create nonclustered index indx_order_items_order_id on ORDER_ITEMS(order_id);

-- INVOICES
create nonclustered index indx_invoices_order_id on INVOICES(order_id);
create nonclustered index indx_invoice_status_code on INVOICES(invoice_status_code);

-- SHIPMENT
create nonclustered index indx_shipment_invoice_id on SHIPMENT(invoice_id);
create nonclustered index indx_shipment_tracking_number on SHIPMENT(shipment_tracking_number)

-- PAYMENTS
create nonclustered index indx_payments_invoice_id on PAYMENTS(invoice_id);
```

Рисунок 2.2 – Код создания индексов в курсовом проекте

3. Резервное копирование и восстановление базы данных

Создание резервных копий баз данных *SQL Server,* выполнение проверочных процедур восстановления резервных копий и хранение резервных копий в безопасном месте вне рабочей площадки помогают предотвратить возможную необратимую потерю данных. Резервное копирование — единственный способ защитить данные.

Создание резервных копий баз данных *SQL Server,* выполнение проверочных процедур восстановления резервных копий и хранение резервных копий в безопасном месте вне рабочей площадки помогают предотвратить возможную необратимую потерю данных. Резервное копирование — единственный способ защитить данные.

При правильном создании резервных копий баз данных можно будет восстановить данные после многих видов сбоев, включая следующие:

- сбой носителя;
- ошибки пользователей (например, удаление таблицы по ошибке);
- сбои оборудования (например, поврежденный дисковый накопитель или безвозвратная потеря данных на сервере);
- стихийные бедствия. Используя *SQL Server* Резервное копирование для Хранилище BLOB-объектов *Azure*, вы можете создать резервную копию вне сайта в регионе, отличном от локального расположения, чтобы использовать ее в случае стихийного бедствия, затрагивающего ваше локальное расположение.

Кроме того, резервные копии баз данных полезны и при выполнении повседневных административных задач, например для копирования баз данных с одного сервера на другой, настройки Группы доступности *AlwaysOn* или зеркального отображения баз данных и архивирования.

Для копирования базы данных была разработана процедура *create_backup*. Для этого используется конструкция backup *database*, которая экспортирует базу данных в файл с расширением bak. Эта процедура представлена ниже на рисунке 3.1.

```
create or alter procedure create_backup
    @dir_path nvarchar(200),
    @file_name nvarchar(100)
as
begin
    declare @path nvarchar(200) = @dir_path + '/' +  isnull(@file_name, '')

    if @file_name is null
    begin
        set @path = concat(@dir_path, '/alconaft_')
        set @path = concat(@path, replace(convert(varchar, getdate(),101),'/','') + replace(convert(varchar, getdate(),108),':',''))
        set @path = concat(@path, '_.bak')
    end

    Backup database alconaft
        to disk = @path
end
```

Рисунок 3.1 – Пример создания процедуры create_backup

Для восстановления базы данных была разработана процедура *restore_DB*. Для этого используется конструкция restore database, которая восстанавливает базу данных из файла с расширением bak. Эта процедура представлена ниже на рисунке 3.2.

```
create or alter procedure restore_DB
    @path nvarchar(100)
as
begin
    alter database alconaft
    set single_user
    with rollback immediate;

    RESTORE DATABASE alconaft
        FROM disk = @path
            WITH replace;

    alter database alconaft
    set multi_user;
end
```

Рисунок 3.2 – Пример создания процедуры *restore_DB*

Для запуска и исполнения данной процедуры необходимо ее создавать системной базе данных master. Это необходимо для избегания коллизий при восстановлении. Так же восстановление невозможно, при наличии подключенных к базе пользователей, поэтому переводим базу данных в монопольный режим с помощью *set single_user* и отменяем все изменения через параметр *rollback immediate*. После восстановления возвращаем базу данных в прежнее состояние через *set multi_user*.

4. Технология шифрования и маскирования данных в БД

Защита информации от посторонних – это шифрование. Шифрование представляет собой способ скрытия данных с помощью ключа или пароля. Это делает данные бесполезными без соответствующего ключа или пароля для дешифрования.

Защита должна всегда начинаться с разграничения прав доступа, но, даже в случае обхода системы управления правами, шифрование поможет защитить информацию. Например, когда база данных попадет в чужие руки, украденная или изъятая информация будет бесполезна, если она была предварительно зашифрована.

Для защиты пользовательской базы данных можно принять следующие меры предосторожности:

- Разработка безопасной системы.
- Шифрование конфиденциальных ресурсов.
- Создание брандмауэра вокруг серверов баз данных.

Однако злоумышленник, который похищает физические носители, такие как диски или резервные ленты, может восстановить или подключить базу данных и просмотреть ее данные.

Одним из решений может стать шифрование конфиденциальных данных в базе данных и использование сертификата для защиты ключей шифрования данных. Таким образом, пользователи без ключей не смогут использовать эти данные. Но этот тип защиты необходимо запланировать заранее.

Функция прозрачного шифрования данных выполняет шифрование и дешифрование ввода-вывода в реальном времени для файлов данных и журналов. Шифрование использует ключ шифрования базы данных (DEK). Загрузочная запись базы данных хранит ключ для доступности во время восстановления. DEK является симметричным ключом. Он защищен сертификатом, который хранится в базе данных сервера master, или асимметричным ключом, который защищает модуль расширенного управления ключами.

Функция прозрачного шифрования данных защищает неактивные данные, то есть файлы данных и журналов. Благодаря ей обеспечивается соответствие требованиям различных законов, постановлений и рекомендаций, действующих в разных отраслях. Это позволяет разработчикам программного обеспечения шифровать данные с помощью алгоритмов шифрования AES и 3DES, не меняя существующие приложения.

Говоря о алгоритмах шифрования нужно сказать, что не существует одного алгоритма, идеально подходящего для всех случаев. Однако можно руководствоваться следующими общими принципами:

- Надежные алгоритмы шифрования обычно потребляют больше ресурсов ЦП, чем менее надежные средства шифрования.
- Использование длинных ключей, как правило, дает более надежные результаты, чем шифрование с помощью коротких ключей.
- Асимметричное шифрование медленнее симметричного.

- Длинные сложные пароли надежнее, чем короткие пароли.
- Симметричное шифрование обычно рекомендуется использовать, только если ключ хранится локально; асимметричное — если ключи должны передаваться по каналу связи.
- При необходимости шифровать большие объемы данных, шифруйте данные с помощью симметричного ключа, а симметричный ключ — с помощью асимметричного ключа.
- Зашифрованные данные не поддаются сжатию, но сжатые данные могут быть зашифрованы. При использовании сжатия данных, выполняйте эту операцию до их шифрования.

В *Microsoft SQL Server* 2012 функции шифрования были улучшены и расширены. Для увеличения надежности криптозащиты и уменьшения нагрузки на систему применяется специальная иерархия ключей.

Пример включения прозрачного шифрования представлен на рисунке 3.1.

```
if not exists (SELECT * FROM sys.symmetric_keys WHERE name LIKE '%MS_DatabaseMasterKey%')
begin
    CREATE MASTER KEY ENCRYPTION BY PASSWORD = 'strongpassword';
end
go

BACKUP MASTER KEY TO FILE = 'C:\DB_course\master_key_backup\masterkey1.mk'
    ENCRYPTION BY PASSWORD = 'strongpassword';
go

CREATE CERTIFICATE alconaft_cert WITH SUBJECT = 'alconaft_certificate';
go

BACKUP CERTIFICATE alconaft_cert TO FILE = 'C:\DB_course\certificate\alconaft1.cer'
   WITH PRIVATE KEY (
        FILE = 'C:\DB_course\certificate\alconaft1.pvk',
        ENCRYPTION BY PASSWORD = 'strongpassword');
go

use alconaft

create database encryption key
   with algorithm = AES_256
   encryption by server certificate alconaft_cert;
go

alter database alconaft set encryption on
```

Рисунок 3.1 – Пример включения прозрачного шифрования данных

Динамическое маскирование данных (DDM) ограничивает возможность раскрытия конфиденциальных данных за счет маскирования этих данных для непривилегированных пользователей. Оно позволяет значительно упростить проектирование и написание кода для системы безопасности в приложении.

Динамическое маскирование данных помогает предотвратить несанкционированный доступ к конфиденциальным данным, позволяя клиентам задать объем раскрываемых конфиденциальных данных с минимальным влиянием на уровень приложения. DDM можно настроить для отдельных полей базы данных, чтобы скрыть конфиденциальные данные в результирующих наборах запросов. При использовании DDM данные в базе данных не изменяются. DDM легко использовать с существующими приложениями, так как правила маскирования применяются к результатам запроса. Многие приложения могут маскировать конфиденциальные данные без изменения существующих запросов.

- Центральная политика маскирования данных применяется непосредственно к конфиденциальным полям в базе данных.
- Вы можете назначать привилегированных пользователей или роли, которые имеют доступ к конфиденциальным данным.
- DDM включает функции полного и частичного маскирования, а также возможность использования случайной маски для числовых данных.
- Назначение и использование масок осуществляется простыми командами Transact-SQL.

Назначение динамического маскирования данных — ограничение раскрытия конфиденциальных данных, при котором пользователи, у которых нет доступа к данным, не смогут их просматривать. Динамическое маскирование данных не сможет помешать пользователям подключиться к базе данных напрямую и выполнить запросы для получения фрагментов конфиденциальных данных. Динамическое маскирование данных дополняет другие функции безопасности SQL Server (аудит, шифрование, безопасность на уровне строк и т. д.). Настоятельно рекомендуется использовать эту возможность с этими функциями для лучшей защиты конфиденциальных данных в базе данных.

Код создания таблицы USERS с включением маскирования данных представлен на рисунке 3.2.

```
create table USERS
(
    user_id int identity(1,1) constraint CUSTOMER_PK primary key,
    first_name nvarchar(50) not null,
    last_name nvarchar(50) not null,
    email_address nvarchar(255) masked with ( function = 'Email()' ) not null unique,
    login_name nvarchar(50) masked with ( function = 'DEFAULT()' ) not null unique,
    login_password nvarchar(50) not null,
    phone_number nvarchar(13) masked with ( function = 'Partial(1, "XXX", 3)' ),
    address_line_1 nvarchar(50) masked with ( function = 'DEFAULT()' ),
    address_line_2 nvarchar(50) masked with ( function = 'DEFAULT()' ),
    town_city nvarchar(50) masked with ( function = 'DEFAULT()' ),
    county nvarchar(50) masked with ( function = 'DEFAULT()' ),
)
```

Рисунок 3.2 – Пример включения прозрачного шифрования данных

5. Импорт и экспорт данных

Для импорта и экспорта был выбран формат XML, так как XML разрабатывался как язык с простым формальным синтаксисом, удобный для создания и обработки документов как программами, так и человеком, с акцентом на использование в Интернете. Язык называется расширяемым, поскольку он не фиксирует разметку, используемую в документах: разработчик волен создать разметку в соответствии с потребностями к конкретной области, будучи ограниченным лишь синтаксическими правилами языка. Расширение XML — это конкретная грамматика, созданная на базе XML и представленная словарём тегов и их атрибутов, а также набором правил, определяющих, какие атрибуты и элементы могут входить в состав других элементов. Сочетание простого формального синтаксиса, удобства для человека, расширяемости, а также базирование на кодировках Юникод для представления содержания документов привело к широкому использованию как, собственно, XML, так и множества производных специализированных языков на базе XML в самых разнообразных программных средствах.

Спецификация XML описывает язык и ряд вопросов, касающихся кодировки и обработки документов. Материал этой секции представляет собой сокращённое изложение описания языка в Спецификации XML, адаптированное для настоящей статьи.

Нормативным считается английский вариант документа, поэтому основные термины приводятся с их английскими оригиналами.

Перевод основных терминов в основном следует доступному в интернете переводу Спецификации на русский язык, исключение составляют термины *tag* и *declaration*. Для термина *tag* здесь используется перевод тег. Для термина *declaration* отдано предпочтение распространённому переводу объявление (против также распространённой кальки декларация).

С физической точки зрения документ состоит из сущностей (англ. *entities*), из которых каждая может ссылаться на другую сущность. Единственный корневой элемент — документная сущность. Содержание сущностей — символы.

С логической точки зрения документ состоит из комментариев (англ. *comments*), объявлений (англ. *declarations*), элементов (англ. *elements*), ссылок на сущности (англ. character *references*) и инструкций обработки (англ. *processing instructions*). Всё это в документе структуризуется разметкой (англ. *markup*).

Все составляющие части документа обобщаются в пролог и корневой элемент. Корневой элемент — обязательная часть документа, составляющая всю его суть (пролог, вообще говоря, может отсутствовать). Корневой элемент может включать (а может не включать) вложенные в него элементы, символьные данные и комментарии. Вложенные в корневой элемент элементы, в свою очередь, могут включать вложенные в них элементы, символьные данные и комментарии, и так далее. Пролог может включать объявления, инструкции обработки, комментарии. Его следует начинать с объявления XML, хотя в определённой ситуации допускается отсутствие этого объявления.

Элементы документа должны быть правильно вложены: любой элемент, начинающийся внутри другого элемента (то есть любой элемент документа, кроме корневого), должен заканчиваться внутри элемента, в котором он начался. Символьные данные могут встречаться внутри элементов как непосредственно, так и в специальных секциях «CDATA». Объявления, инструкции обработки и элементы могут иметь связанные с ними атрибуты. Атрибуты используются для связывания с логической единицей текста пар имя-значение.

Для экспорта таблицы PRODUCTS в формате XML была разработана процедура export_products. Для формирования XML в select запросе используется конструкция FOR XML. За неимением встроенных возможностей экспорта данных в формат XML, для вывода в файл была использована расширенная хранимая процедура xp_cmdshell. Процедура export_products представлена ниже на рисунке 4.1.

```
create or alter procedure export_products
as
Begin
    exec master.dbo.sp_configure 'show advanced options', 1
        RECONFIGURE WITH OVERRIDE
    exec master.dbo.sp_configure 'xp_cmdshell', 1
        RECONFIGURE WITH OVERRIDE;

    declare @fileName varchar(100)
    declare @sqlStr varchar(1000)
    declare @sqlCmd varchar(1000)
    set @fileName = 'c:\DB_course\alconaft_products.xml'
    set @sqlStr = 'use alconaft; declare @text varchar(max) = (select * from products for xml path(''PRODUCT''), ' +
                  'ROOT(''PRODUCTS'')); select replace(@text, ''</PRODUCT>'', ''</PRODUCT>'' + char(13))'
    set @sqlCmd = 'bcp "' + @sqlStr + '" queryout ' + @fileName + ' -w -T -S DESKTOP-FT44TPL\SQLEXPRESS'
    exec xp_cmdshell @sqlCmd;
end
```

Рисунок 4.1 – Пример создания процедуры export_products

Пример XML документа представлен ниже на рисунке 4.2

```
<PRODUCTS>
    <PRODUCT>
        <product_id>1</product_id>
        <product_name>KFKbAxMGhCoBzGJiebQT</product_name>
        <product_description>Tk4NKTA1sSWkQHGcqHNANhnZsuhRCoyZxHbfvfk10K
            :eUc52QMU8ZRJAXD1ahbF33ctxtwzivhTPKqgCBKw:uefULCjw3L5r
            -V5ZH0F5CPWrNaFieciyTUjG8TV2Ypi7KY1ELFqEgrI9Ka8BkS6MDO3ZYChO-p4KN
            R5TX5KPfYcTb2g2dQYlZmKMJSkiRngbYdMF5ar7sSJqgAsjS5Zivks867yMjoRdf7nS3
            nF-0Awc-umlMV7G Y q</product_description>
        <product_type_id>15</product_type_id>
        <product_price>286.1400</product_price>
        <product_quantity>473</product_quantity>
    </PRODUCT>
```

Рисунок 4.2 – Пример XML документа

Для импорта данных в таблицу PRODUCTS, из файла формата XML (пример содержимого входного файла представлен выше на рисунке 4.2), была разработана

процедура import_products. Для получения xml файла и последующего разбора со вставкой используется конструкция FROM OPENROWSET совместно с параметром BULK. Процедура import_products представлена ниже на рисунке 4.2.

```
create or alter procedure import_products
    @path nvarchar(100)
as
begin
    set identity_insert PRODUCTS on;
    insert into PRODUCTS (product_id, product_name, product_description, product_type_id, product_price, product_quantity)
    select
       MY_XML.PRODUCT.query('product_id').value('.', 'int'),
       MY_XML.PRODUCT.query('product_name').value('.', 'nvarchar(50)'),
       MY_XML.PRODUCT.query('product_description').value('.', 'nvarchar(max)'),
       MY_XML.PRODUCT.query('product_type_id').value('.', 'int'),
       MY_XML.PRODUCT.query('product_price').value('.', 'money'),
       MY_XML.PRODUCT.query('product_quantity').value('.', 'int')

    from (select CAST(MY_XML as xml)
          from OPENROWSET(bulk @path, single_blob ) as T(MY_XML)) as T(MY_XML)
          cross apply MY_XML.nodes('PRODUCTS/PRODUCT') as MY_XML (PRODUCT);
    set identity_insert PRODUCTS off;
end
```

Рисунок 4.3 – Пример создания процедуры export_products

6. Тестирование

Тестирование отыгрывает важную роль при разработке любого программного продукта. Чем качественнее тестирование, тем лучше в итоге должен выйти конечный продукт.

Часто можно столкнуться с ситуацией, когда тестирование программного кода проходит очень кропотливо, а на тестирование базы данных времени уже не остается либо оно делается по остаточному принципу.

В итоге работа с БД может стать узким местом в производительности нашего приложения. Собственно, не будем допускать такую ошибку и протестируем созданную базу данных.

Для тестирования базы данных были проведены запросы для определения разницы времени в обработке запроса. Наиболее часто используемой является процедура login_user, позволяющая авторизоваться пользователю.

Время выполнения процедуры и план выполнения запроса показаны на рисунках 5.1 и 5.2 соответственно.

```
alconaft> exec login_user '6onW5MGfzp', null, 'Wt8XuW12FS'
[2021-12-20 02:56:29] 1 row retrieved starting from 1 in 75 ms (execution: 22 ms, fetching: 53 ms)
```

Рисунок 5.1 – Время выполнения запроса при 100000 строках

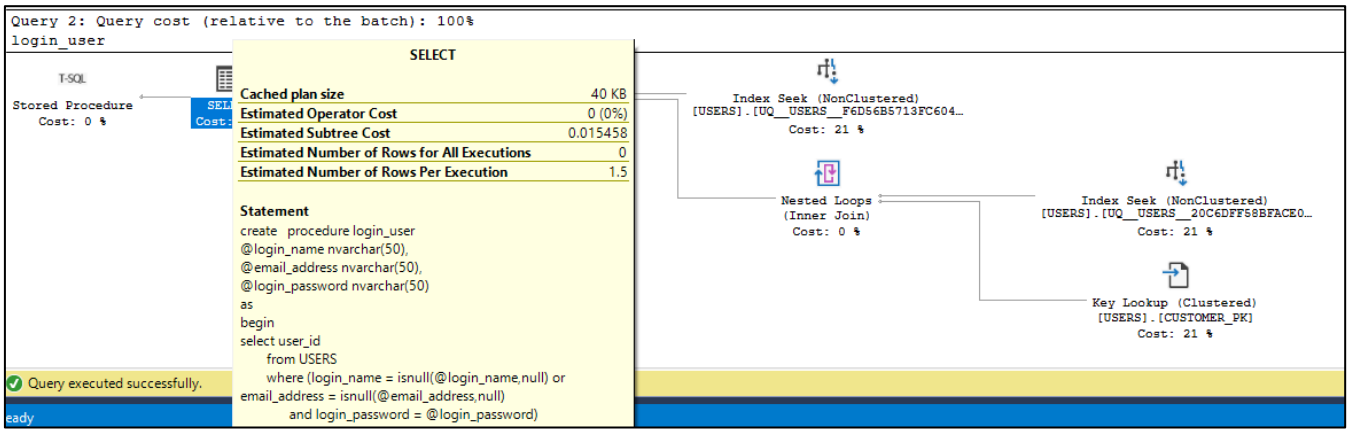


Рисунок 5.2 – План выполнения запроса при 100000 строках

Для 100000 строк результаты по времени выполнения были 75 мс., а по предполагаемой стоимости запроса – 0,015.

Подведя итоги тестирования производительности базы данных для различного объема данных в таблице можно сделать вывод, что приложение будет корректно работать с объемом выборки до 100000 значений.

Заключение

В рамках работы над проектом был проведен обзор аналогичных решений и программных продуктов, спроектирована архитектура и структура базы данных, проведено тестирование производительности запросов к таблицам базы данных при различном объеме данных.

Разработанная база банных содержит в себе 8 связанных таблиц. Разработка базы данных проводилась в рамках *Microsoft SQL Server* – система управления реляционными базами данных. Данная СУБД используется для работы с базами данных размером от персональных до крупных баз данных масштаба предприятия.

Одной из главных задач данной курсовой работы - освоение технологии шифрование и маскирование данных. Данная технология и ее возможности были успешно изучены и реализованы в процессе разработки базы данных.

База данных является законченной, хотя и возможна доработка и расширение функционала. Данная база данных соответствует представленной задаче и отвечает всем необходимым требованиям.

Список использованных источников

1. BACKUP (Transact-SQL) [Электронный ресурс] / Microsoft. – Режим доступа: https://docs.microsoft.com/en-us/sql/t-sql/statements/backup-transact-sql?view=sql-server-ver15. – Дата доступа: 15.11.2021.

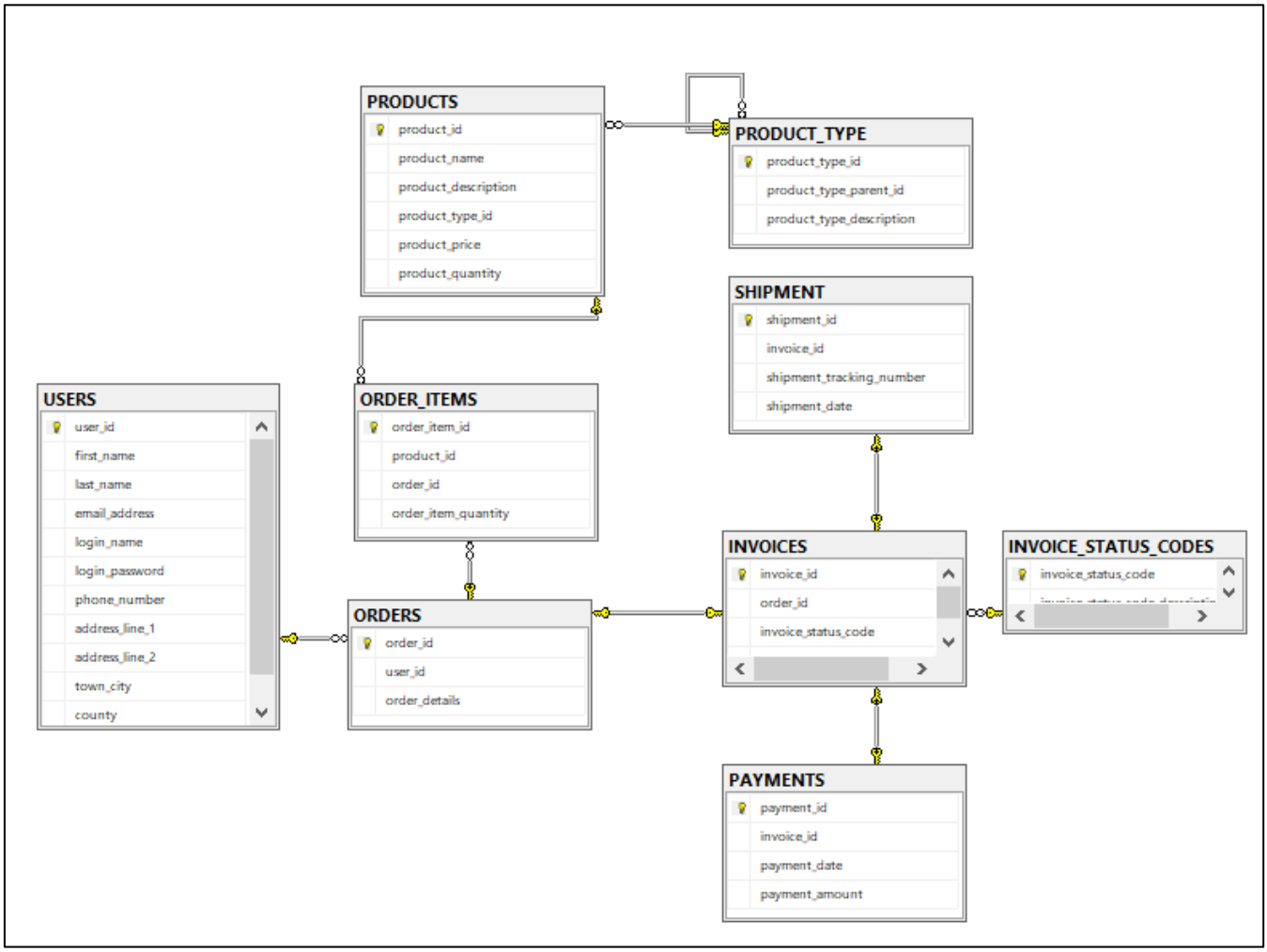
2. Restore a Database Backup Using SSMS [Электронный ресурс] / Microsoft. – Режим доступа: https://docs.microsoft.com/en-us/sql/relational-databases/backup-restore/restore-a-database-backup-using-ssms?view=sql-server-ver15. – Дата доступа: 15.11.2021.

3. Transparent Data Encryption (TDE) SSMS [Электронный ресурс] / Microsoft. – Режим доступа: https://docs.microsoft.com/en-us/sql/relational-databases/security/encryption/transparent-data-encryption?view=sql-server-ver15

4. Dynamic Data Masking (DDM) [Электронный ресурс] / Microsoft. – Режим доступа: https://docs.microsoft.com/en-us/sql/relational-databases/security/dynamic-data-masking?view=sql-server-ver15

Приложение А

Диаграмма базы данных



Приложение Б

Хранимые процедуры для вывода данных

```
create or alter procedure get_user
    @user_id int
as
begin

    execute as user = 'reader';
    select
        user_id,
        login_name,
        first_name,
        last_name,
        email_address,
        phone_number
    from
        USERS
    where
        user_id = 1
    revert
end
go

create or alter procedure get_product_types
as
    select
        *
    from
        PRODUCT_TYPE
go

create or alter procedure get_product_by_id
    @product_id int
as
    select
        *
    from
        PRODUCTS
    where product_id = @product_id
go
```

```
create or alter procedure get_product_by_name
    @product_name nvarchar(50)
as
    select
        *
    from
        PRODUCTS
    where
        product_name like concat('%', @product_name, '%')
go

create or alter procedure get_order
    @order_id int
as
    select
        *
    from
        ORDERS
    where order_id = @order_id
go

create or alter procedure get_user_orders
    @user_id int
as
    select
        *
    from
        ORDERS
    where
        user_id = @user_id
    order by
        order_id
go
```

```
create or alter procedure get_order_items
    @order_id int
as
    select
        *
    from
        ORDER_ITEMS
    where
        order_id = @order_id
    order by
        order_item_id
go

create or alter procedure get_invoice_by_order
    @order_id int
as
    select
        *
    from
        INVOICES
    where
        order_id = @order_id
go

create or alter procedure get_invoice
    @invoice_id int
as
    select
        *
    from
        INVOICES
    where
        invoice_id = @invoice_id
go
```

```
create or alter procedure get_products
as
    select
        *
    from
        PRODUCTS
go

create or alter procedure get_products_by_price
    @min_price money,
    @max_price money
as
    select
        *
    from
        PRODUCTS
    where
        product_price between @min_price and @max_price
go
```

```
create or alter procedure get_product_types_recursive
    @product_type_id int
as
begin
    with tree (product_type_id, product_type_parent_id, product_type_description)
    as
    (
        select
            product_type_id,
            product_type_parent_id,
            product_type_description
        from
            PRODUCT_TYPE
        where
            product_type_parent_id = @product_type_id
        union all
        select
            PRODUCT_TYPE.product_type_id,
            PRODUCT_TYPE.product_type_parent_id,
            PRODUCT_TYPE.product_type_description
        from
            PRODUCT_TYPE inner join tree
        on tree.product_type_id = PRODUCT_TYPE.product_type_parent_id
    )
    select
        product_type_id,
        product_type_parent_id,
        product_type_description
    from
        tree
    order by product_type_parent_id
end
go
```

```
create or alter procedure get_done_orders
    @user_id int
as
    select
        O.order_id,
        order_details,
        I.invoice_id,
        payment_date,
        payment_amount,
        shipment_id,
        shipment_date
    from
        ORDERS O join INVOICES I on O.order_id = I.order_id
                 join PAYMENTS P on I.invoice_id = P.invoice_id
                 left join SHIPMENT S on I.invoice_id = S.invoice_id
    where O.user_id = @user_id
go

create or alter procedure get_open_order
    @user_id int
as
    select
        O.order_id,
        O.order_details,
        OI.product_id,
        OI.order_item_quantity
    from
        ORDERS O join ORDER_ITEMS OI
        on O.order_id = OI.order_id
        left join INVOICES I
        on O.order_id = I.order_id
    where
        O.user_id = @user_id and
        I.invoice_id is null
go
```

```
create or alter procedure get_open_shipments
    @user_id int
as
    select
        *
    from
        ORDERS O join INVOICES I on O.order_id = I.order_id
        join SHIPMENT S on I.invoice_id = S.invoice_id
    where
        O.user_id = @user_id and
        shipment_date is null
go
```

Хранимые процедуры авторизации и регистрации пользователя

```
create or alter procedure login_user
    @login_name nvarchar(50),
    @email_address nvarchar(50),
    @login_password nvarchar(50),
    @ret int output
as
begin
    select @ret=1
       from USERS
       where (login_name = isnull(@login_name,null) or email_address = isnull(@email_address,null)
             and login_password = @login_password)
end
```

```
create or alter procedure register_user
    @first_name nvarchar(50),
    @last_name nvarchar(50),
    @login_name nvarchar(50),
    @login_password nvarchar(50),
    @email_address nvarchar(50),
    @login_name nvarchar(50),
    @ret int output
as
begin
    select
        @ret=-1
    from
        USERS
    where
        login_name = @login_name
        or
        email_address = @email_address

    if @ret != -1
    begin
        exec insert_user
            @first_name: @first_name,
            @last_name: @last_name,
            @email_address: @email_address,
            @login_name: @login_name,
            @login_password: @login_password,
            @phone_number: null,
            @address_line_1: null,
            @address_line_2: null,
            @town_city: null,
            @county: null
    end
end
```

Хранимые процедуры для работы с заказами

```
create or alter procedure add_order_product
    @user_id int,
    @product_id int,
    @product_quantity int
as
begin
    declare @order_id int;
    exec get_open_order @user_id, @order_id output

    if @order_id is null
    begin
        insert into ORDERS (user_id) values (@user_id, default)
        exec get_open_order @user_id, @order_id output
    end

    if exists (select order_item_id from ORDER_ITEMS where order_id = @order_id and product_id = @product_id)
    begin
        select
            @product_quantity = @product_quantity + order_item_quantity
        from
            ORDER_ITEMS
        where
            order_id = @order_id and product_id = @product_id

        update
            PRODUCTS
        set
            product_quantity = product_quantity - @product_quantity
        where
              product_id = @product_id

        update
            ORDER_ITEMS
        set
            order_item_quantity = @product_quantity
        where
            order_id = @order_id and product_id = @product_id
    end
    else
        exec insert_order_item
            @order_id,
            @product_id,
            @product_quantity
end
go
```

```
create or alter procedure buy_order
    @order_id int
as
begin
    if exists (select * from INVOICES where order_id = @order_id)
    begin
        select -1
    end

    insert into INVOICES (order_id, invoice_status_code, invoice_date) values (@order_id, 0, getdate())

    declare @invoice_id int = (select invoice_id from INVOICES where order_id = @order_id)

    declare @payment_amount money = (
        select
            order_id,
            sum(p.product_price * oi.order_item_quantity) as "order_price"
        from
            ORDER_ITEMS oi join PRODUCTS p
            on oi.product_id = p.product_id
        where
            oi.order_id = @order_id
        group by
            order_id
    )

    insert into PAYMENTS (invoice_id, payment_date, payment_amount) values (@invoice_id, getdate(), @payment_amount)

    select 1
end
```

```
create or alter procedure open_shipment
    @invoice_id int
as
begin
    declare @shipment_tracking_number nvarchar(20)
    exec RandomString @length: 20, @char_pool: '1234567890qwertyuiopasdfghjklzxcvbnm-', @random_string: @shipment_tracking_number output

    insert into SHIPMENT (invoice_id, shipment_tracking_number, shipment_date) values (@invoice_id, @shipment_tracking_number, null)
end
go

create or alter procedure close_shipment
    @shipment_id int
as
begin
    update SHIPMENT set shipment_date = getdate() where shipment_id = @shipment_id
end
go
```