МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ФАКУЛЬТЕТ РАДИОФИЗИКИ И КОМПЬЮТЕРНЫХ ТЕХНОЛОГИЙ

Кафедра телекоммуникаций и информационных технологий

МОРДАС
Кирилл Ричардович

МОДЕРНИЗАЦИЯ ЛАБОРАТОРНОЙ РАБОТЫ ПО КУРСУ «ПРОГРАММНО-АППАРАТНЫЕ СРЕДСТВА ОБЕСПЕЧЕНИЯ ИНФОРМАЦИОННОЙ БЕЗОПАСНОСТИ»

Аннотация к дипломной работе

Научный руководитель – старший преподаватель И.Н. Щербак

Минск, 2022

РЕФЕРАТ

Дипломная работа: 122 с., 25 рис., 1 табл., 10 источников, 3 прил.

Объект исследования – базы данных, существующие угрозы базам данных, методы защиты баз данных.

Цель работы – модернизация лабораторной работы по курсу «Программно-аппаратные средства обеспечения информационной безопасности».

Методы проведения работы – рассмотрение существующих методических указаний к лабораторным работам по защите баз данных и модернизация методических указаний на их основе.

РЭФЕРАТ

Дыпломная праца: 122 с., 25 мал., 1 табл., 10 крыніц, 3 дадатка

Аб'ект даследавання – базы дадзеных, якія існуюць пагрозы баз дадзеных, метады абароны баз дадзеных.

Мэта працы – мадэрнізацыя лабараторнай працы па курсе «Праграмна-апаратныя сродкі забеспячэння інфармацыйнай бяспекі».

Метады правядзення працы – разгляд існуючых метадычных указанняў да лабараторных работах па абароне баз дадзеных і мадэрнізацыі метадычных указанняў на іх аснове.

ABSTRACT

Thesis: 122 pages, 25 figures, 1 tables, 10 sources, 3 app.

Object of research – databases, existing threats to databases, methods of database protection.

Purpose of work – modernization of the laboratory works on the course «Software and hardware information security».

Methods of work – consideration of existing guidelines for laboratory work for database security and the modernization of guidelines based on them.

ВВЕДЕНИЕ

На четвёртом курсе специальности Компьютерная безопасность читается курс «Программно-аппаратные средства обеспечения информационной безопасности».

Цели и задачи учебной дисциплины**.**

Цель учебной дисциплины – формирование у студентов знаний и принципов построения, технического и программного обеспечения систем защиты информации.

Основные задачи дисциплины**:**

1. Научить студентов анализировать основные угрозы информационной безопасности.
2. Изучить программно-аппаратные средства обеспечения информационной безопасности.

Место учебной дисциплины в системе подготовки специалиста с высшим образованием.

Учебная дисциплина относится к циклу специальных дисциплин (государственный компонент).

Связи с другими учебными дисциплинами, включая учебные дисциплины компонента учреждения высшего образования, дисциплины специализации и др.

Для успешного усвоения дисциплины необходимы знания, полученные при изучении дисциплин «Микропроцессоры и аппаратные средства вычислительной техники», «Операционные системы», «Компьютерные сети», «Технологии программирования».

Структура учебной дисциплины

Дисциплина изучается в 7 и 8 семестрах. Всего на изучение учебной дисциплины «Программно-аппаратные средства обеспечения информационной безопасности» отведено:

– для очной формы получения высшего образования в 7 семестре – 164 часа часов, в том числе 86 аудиторных часов, из них: лекции – 32 часа, лабораторные занятия – 26 часа, дистанционные лабораторные занятия – 16 часов, управляемая самостоятельная работа – 12 часов. Трудоемкость учебной дисциплины в 7 семестре составляет – 5 з. ед.

Целью дипломной работы является модернизация лабораторного практикума по курсу «Программно-аппаратные средства обеспечения информационной безопасности», а именно, части, посвященной защите баз данных, о которой пойдёт речь в работе.

В рамках курса «Программно-аппаратные средства обеспечения информационной безопасности» (модуль защита баз данных), выполняется лабораторный практикум, на сегодняшний момент лабораторный практикум выполняется как единый большой проект, но такая форма выполнения имеет определённые недостатки, поэтому было принято решение разделить практикум

на несколько лабораторных работ. С учётом разделения необходимо увеличить объём теоретической части в каждой лабораторной работе, а также количество практических заданий.

Были поставлены следующие задачи:

1. Изучить имеющиеся методические указания для выполнения лабораторного практикума.
2. На основе имеющихся методических указаний необходимо модернизировать лабораторный практикум по курсу «Программно-аппаратные средства обеспечения информационной безопасности»
3. С учётом модернизации необходимо разделить лабораторный практикум на несколько лабораторных работ, увеличить объём теоретической части в каждой лабораторной работе, а также количество практических заданий.

ГЛАВА 1 АНАЛИЗ СФЕРЫ ПРИМЕНЕНИЯ БАЗ ДАННЫХ

1.1 Безопасность баз данных

Защита баз данных — это комплексный подход к обеспечению безопасности информации, хранящейся в них. Под этим понятием подразумеваются меры, направленные на предотвращение её потери, хищения или изменения.

База данных — это хранилище информации. В любой компании из любой сферы деятельности есть базы данных, которые различаются по типу содержимого и виду.

Примеры:

–CRM-системы имеют базы данных контрагентов;

–бухгалтерия имеет соответствующую базу данных;

–операторы связи хранят данные своих клиентов, журналы звонков и сообщений;

–медицинские учреждения заводят базы данных пациентов с историями болезней и персональной информацией.

Базы данных не только хранят ценную информацию для компаний и их клиентов, но и позволяют составлять аналитические отчёты.

1.2 Зачем необходима защита баз данных

Обеспечение безопасности баз данных позволяет защитить компанию от угроз:

–непреднамеренный или преднамеренный несанкционированный доступ с последующим хищением или уничтожением конфиденциальной информации, изменением метаданных, структуры, программ или безопасности со стороны хакеров, неавторизованных пользователей, администратора базы данных;

–ограничение пропускной способности, перегрузка, снижение производительности с последующим ограничением работы авторизованных пользователей;

–ввод неверных данных, команд, ошибки администрирования, саботаж с последующим повреждением данных или их потерей;

–хищение информации, получение запатентованных и личных данных, нанесение вреда программам, отказ в доступе к базам данных или его прерывание, сбой в работе вследствие инфекций вредоносных программ;

–несанкционированная эскалация привилегий, программные ошибки;

–преднамеренное или непреднамеренное причинение физического ущерба серверам баз данных [1].

Для защиты баз данных необходимо обеспечить им доступность, целостность и конфиденциальность.

1.3 Способы защиты

Защита базы данных требует комплексного подхода. Методов большое множество - рассмотрим только основные.

Штатный аудит и мониторинг

Это средство защиты часто используется коммерческими организациями и входит в состав систем управления базами данных (далее — СУБД). Механизм работы штатного аудита заключается в настройке и включении триггеров, а также создании специфических процедур, которые начинают срабатывать во время запроса доступа к чувствительной информации. При этом ведется журнал запросов и подключений к системе управления базами данных в виде таблицы, где указаны данные о том, в какое время, кем и какой запрос был сделан.

Штатный аудит отвечает основным отраслевым требованиям регуляторов, но бесполезен в случае необходимости проведения внутренних расследований инцидентов и решения задач информационной безопасности.

Резервное копирование

Настройка регулярного резервного копирования на жестком диске с дублированием на другом носителе позволяет восстановить информацию в случае сбоя в работе системы управления базами данных.

Шифрование

Это использование устойчивого криптоалгоритма для шифрования информации в базах данных. В случае применения такого метода защиты, злоумышленник увидит информацию в нечитаемом виде в отличие от пользователей, имеющих ключ доступа.

Проблема заключается в способе хранения ключей, так как нет гарантии того, что ключ не будет намеренно передан третьему лицу. Кроме того, шифрование не обеспечивает безопасность от администратора базы данных.

VPN и двухфакторная аутентификация

Организация доступа внутренних пользователей и администраторов к базам данных с применением VPN и двухфакторной аутентификацией (использование двух разных типов аутентификации) значительно повышает уровень защиты от несанкционированного проникновения.

Помимо стандартной пары логина и пароля для доступа к защищенному сегменту сети используется еще один фактор, участвующий в аутентификации, например,

- USB-ключи, смарт-карты, iButton;
- одноразовый код в виде СМС или email;
- генератор паролей (технология SecurID);
- биометрические данные и т. д.

Автоматизированные системы защиты

Это специализированные системы обеспечения безопасности баз данных, представляющие собой решения DAM (Database Activity Monitoring) и DBF (Database Firewall). Применяются, когда уже реализованы вышеописанные меры защиты баз данных.

DAM производит мониторинг пользователей в системах управления базами данных. При этом не требуется специально изменять настройки или конфигурации систем управления базами данных. Поэтому это решение называется независимым. DAM может работать пассивно с копией трафика, тем самым не оказывая влияния на бизнес-процессы [3].

ГЛАВА 2 ОБЕСПЕЧЕНИЕ БЕЗОПАСНОСТИ БАЗ ДАННЫХ

2.1 Контроль целостности

Главная особенность SQL-технологий наличие у сервера СУБД специальных средств контроля целостности данных, не зависящих от клиентских программ и привязанных непосредственно к таблицам. Т.е. принципиально не важно, каким образом осуществляется доступ к базе данных: через SQL-консоль, через ODBC-драйвера из приложения Windows, через WWW-connector из Internet-браузера или через DBI-интерфейс Perl. В любом из этих случаев, за контролем целостности данных следит сервер, и при нарушении правил целостности данных сервер известит клиента об ошибке.

К структурам контроля целостности данных относятся ограничители (constraint), которые привязаны к столбцам и триггеры (trigger), которые могут быть привязаны как к столбцам, так и к строкам в таблице.

Ограничители - это элементарные проверки или условия, которые выполняются для операций вставки и модификации значения столбца. Если данная проверка не проходит или условие не выполняется, то вставка или модификация отменяется, а в программу клиента передается ошибка.

SQL-серверы, как правило, поддерживают следующие ограничители.

NOT NULL - проверка на непустое значение. NULL - специальное понятие в СУБД, которое означает «пусто». «Пусто» и «0(ноль)» не равны друг другу!

UNIQUE - проверка на уникальность. Вставляемое значение должно быть уникально для данного столбца по всей таблице. Может содержать пустые значения.

PRIMARY KEY - первичный ключ. Значение в столбце считается первичным ключом, если оно непустое и уникально в пределах столбца данной таблицы. Первичный ключ может быть составным и представлять собой комбинацию столбцов. Тогда чтобы считаться первичным ключом, каждое из группы значений не должно быть пустыми и формируемые строки значений первичного ключа должны быть уникальны в пределах таблицы. Первичный ключ - основа для построения индексов по таблице.

SQL-технология позволяет на уровне столбца задавать домены значений, т.е. строго определенные наборы или диапазоны значений, для помещаемых в столбец данных. В частности, можно реализовывать ограничения ссылочной целостности (referential integrity constraint) и проверки фиксированного условия. Ограничение ссылочной целостности не позволяет значениям из столбца одной таблицы принимать значения кроме как из присутствующих в столбце другой таблицы. Это делается при помощи ограничителей FOREIGN KEY (внешний ключ) и REFERENCES (указатель ссылки). Таблица, содержащая FOREIGN KEY, считается родительской таблицей. Таблица, содержащая REFERENCES,

считается дочерней таблицей. Внешний ключ и указатель ссылки могут находиться в одной таблице, т.е. родительская таблица одновременно является дочерней.

FOREIGN KEY - внешний ключ. Назначает столбец или комбинацию столбцов в текущей (родительской) таблице в качестве внешнего ключа для ссылки из других таблиц.

REFERENCES - указатель ссылки (или родительский ключ). Указывает на столбец (комбинацию столбцов) в родительской таблице, ограничивающую значения в текущей (дочерней) таблице.

Для использования ограничений ссылочной целостности должны выполняться некоторые условия. В частности, родительская и дочерняя таблицы должны находиться в пределах одного аппаратного сервера базы данных, они не могут находиться на различных узлах распределенной базы данных. Столбцы, участвующие в отношении ограничения ссылочной целостности обязаны иметь один и тот же тип данных.

Ограничения ссылочной целостности используются при каскадном удалении, т.е. при удалении записи в родительской таблице удаляются все записи с указанным ключом из дочерних таблиц, и наоборот при запрете удаления/модификации, т.е. при наличии зависимых записей в дочерних таблицах, значение ключа записи в родительской таблице нельзя удалить или модифицировать.

CHECK - проверка фиксированного условия. В данном ограничителе явно указывается условие, которое должно выполняться для вставляемого или модифицируемого значения в столбце. Например, check (user in 'ALEX','JUSTAS') - в столбце user могут содержаться только значения 'ALEX' и 'JUSTAS', попытка вставки значения 'SHTIRLITZ' будет интерпретирована как ошибочная, check (user_salary between 1000 and 5000) - столбец user_salary может принимать целочисленные значения в диапазоне от 1000 до 5000 и т.д. При формировании условий с некоторыми ограничениями могут использоваться функции, например, check (user = upper(user)), в данном случае имя пользователя должно вводиться только в верхнем регистре. Есть и ограничения, например, CHECK не может содержать подзапросы (SELECT).

Обычно ограничители задаются при создании таблиц. Но в дальнейшем их можно изменять, удалять или временно запрещать при помощи соответствующих команд СУБД.

Обработка данных в многопользовательской СУБД.

Основное требование к многопользовательским СУБД - обеспечение непротиворечивости данных в системе, при сохранении максимальной производительности и конкуренции в доступе к данным для пользователей.

Конкуренция в доступе к данным означает, что каждый из пользователей независим от остальных пользователей в потребности обработки данных. Система, во избежание порчи данных, самостоятельно устанавливает очередность работы с данными для пользователей. В случае необходимости пользователи могут ожидать своей очереди для работы с данными. Одной из главных целей многопользовательской СУБД является максимальное уменьшение этого времени ожидания до такой степени, чтобы оно (в идеале) стало незаметным для пользователя.

Кроме того, сервер СУБД должен предотвращать взаимно разрушающие манипуляции с данными нескольких пользователей при их одновременной работе. Например, если система не предусматривает такую возможность, то менеджеры принимающие заказы от клиентов на поставку товара, и выполняющие их резервирование на складе, могут зарезервировать товара больше чем фактически имеется в наличии. В этом случае обеспечен неприятный разговор с клиентом, заказ которого будет впоследствии отменен.

Более неприятная ситуация возможна в банке: если одновременно исполняется несколько клиентских платежных поручений с одного счета, то при неконтролируемом списании с клиентского счета возможен отрицательный остаток, что недопустимо.

Контроль нужен также в системах резервирования билетов на транспорте, чтобы билет на одно и то же место не был продан разными кассирами разным пассажирам.

Несмотря на различия в реализации, серверы СУБД используют общие способы управления данными и доступом к ним.

Атомарность SQL-выражений при работе с данными.

Под атомарностью выражения понимается неизменность (фиксация во времени) набора данных, с которыми это выражение работает на всем протяжении своего исполнения. Т.е. если мы выполняем оператор UPDATE над определенной таблицей, то состояние таблицы на момент начала выполнения операции фиксируется во времени и не изменяется до конца выполнения оператора. Этот набор данных для текущего выполняемого выражения не может быть изменен другим пользователем или даже другой сессией этого же пользователя, которая пытается выполнить операцию модификации этих же данных в этой же таблице.

Распараллеливание операций.

Типовые операции с таблицей в базе данных состоят из многих однотипных операций, например, оператор UPDATE, который модифицирует 5000 строк в таблице, по своей сути состоит из 5000 операций, каждая из которых может быть выполнена независимо. В связи с этим такие операторы очень хорошо распараллеливаются при использовании многопроцессорных систем. Это

позволяет выровнять нагрузку в системе между разными процессорами, при том условии что СУБД умеет работать в многопроцессорной конфигурации, и уменьшить время ответа системы.

Обеспечение максимальной производительности.

С целью сокращения времени различных пользователей на манипуляции с данными используется ряд следующих методов. Их работа находится на уровне, скрытом даже от программиста СУБД, но о них стоит упомянуть т.к. они иллюстрируют серьезные различия с xBase-технологией.

Строго говоря, эта информация справедлива лишь в отношении Oracle, но другие СУБД используют подобные принципы.

Процессы, выполняющие чтение блоков данных, никогда не ожидают процессов, выполняющих запись тех же блоков данных.

Процессы, выполняющие запись блоков данных, при отсутствии явных блокировок со стороны пользователя, не ожидают процессов, выполняющих чтение тех же блоков данных.

Процессы, выполняющие запись блоков данных, ожидают другие процессы, выполняющие запись, только в случае если они пытаются выполнить запись данных в одни и те же блоки данных.

Данные приемы позволяют существенно уменьшить время ожидания ответа системы и увеличить ее производительность [4].

2.2 Представление

Представление — это виртуальная таблица, содержимое которой определяется запросом. Как и таблица, представление состоит из ряда именованных столбцов и строк данных. Пока представление не будет проиндексировано, оно не существует в базе данных как хранимая совокупность значений. Строки и столбцы данных извлекаются из таблиц, указанных в определяющем представление запросе и динамически создаваемых при обращениях к представлению.

Представление выполняет функцию фильтра базовых таблиц, на которые оно ссылается. Определяющий представление запрос может быть инициирован в одной или нескольких таблицах, или в других представлениях текущей или других баз данных. Кроме того, для определения представлений с данными из нескольких разнородных источников можно использовать распределенные запросы. Это полезно, например, если нужно объединить структурированные подобным образом данные, относящиеся к разным серверам, каждый из которых хранит данные конкретного отдела организации.

Представления обычно используются для направления, упрощения и настройки восприятия каждым пользователем информации базы данных. Представления могут использоваться как механизмы безопасности, давая возможность пользователям обращаться к данным через представления, но не

предоставляя им разрешений на непосредственный доступ к базовым таблицам, лежащим в основе представлений. Представления могут использоваться для обеспечения интерфейса обратной совместимости, моделирующего таблицу, которая существует, но схема которой изменилась. Представления могут также использоваться при прямом и обратном копировании данных в SQL Server для повышения производительности и секционирования данных [2].

2.2.1 Типы представлений

Кроме основных определяемых пользователем представлений, выполняющих стандартные роли, в SQL Server предусмотрены следующие типы представлений, которые соответствуют специальным назначениям в базе данных.

Индексированные представления
Индексированным называется материализованное представление. Это означает, что определение представления вычисляется, а результирующие данные хранятся точно так же, как и таблица. Индексировать представление можно, создав для него уникальный кластеризованный индекс. Индексированные представления могут существенно повысить производительность некоторых типов запросов. Индексированные представления эффективнее всего использовать в запросах, группирующих множество строк. Они не очень хорошо подходят для часто обновляющихся базовых наборов данных [2].

Секционированные представления.
Секционированным называется представление, соединяющее горизонтально секционированные данные набора таблиц-элементов, находящихся на одном или нескольких серверах. При этом данные выглядят так, как будто находятся в одной таблице. Представление, соединяющее таблицы-элементы одного экземпляра SQL Server , называется локальным секционированным представлением.

Системные представления.
Системные представления предоставляют доступ к метаданным каталога. Системные представления можно использовать для получения сведений об экземпляре SQL Server или объектах, определенных в экземпляре. Например, получить сведения об определяемых пользователем базах данных, доступных в экземпляре, можно через представление каталога sys.databases [2].

2.2.2 Создание представлений

Представления можно создать в SQL Server с помощью SQL Server Management Studio или Transact-SQL. Представление можно использовать в следующих целях.

Для направления, упрощения и настройки восприятия информации в базе данных каждым пользователем.

В качестве механизма безопасности, позволяющего пользователям обращаться к данным через представления, но не предоставляя им разрешений на непосредственный доступ к базовым таблицам.

Для предоставления интерфейса обратной совместимости, моделирующего таблицу, схема которой изменилась [2].

2.2.3 Ограничения

Представление может быть создано только в текущей базе данных.

Представление может включать не более 1 024 столбцов [2].

2.2.4 Безопасность

Для выполнения этой инструкции требуется разрешение CREATE VIEW в отношении базы данных и разрешение ALTER в отношении схемы, в которой создается представление [2].

2.2.5 Создание представления с использованием конструктора запросов и представлений

В обозревателе объектов разверните базу данных, в которой необходимо создать новое представление.

Щелкните правой кнопкой папку Представления и выберите «*Создать* представление»

В диалоговом окне «*Добавить таблицу»* выберите один или несколько элементов, которые необходимо включить в новое представление, на одной из следующих вкладок: «Таблицы», «Представления», «Функции» и «Синонимы».

Щелкните «*Добавить»*, а затем выберите «*Закрыть»*.

На Панели диаграмм выберите «*столбцы или другие элементы для включения в новое представление»*.

На Панели критериев выберите «*дополнительные условия сортировки или фильтрации для столбцов»*.

В меню Файл выберите пункт «*Сохранить view name»*.

В диалоговом окне Выбор имени введите имя нового представления и щелкните «*ОК»* [2].

2.2.6 Использование Transact-SQL

Создание представления

В обозревателе объектов подключитесь к экземпляру компонента Компонент Database Engine.

На стандартной панели выберите пункт Создать запрос.

Скопируйте следующий пример в окно запроса и нажмите кнопку Выполнить.

```
USE AdventureWorks2012 ;
GO
CREATE VIEW HumanResources.EmployeeHireDate
AS
```

```
SELECT p.FirstName, p.LastName, e.HireDate
FROM HumanResources.Employee AS e JOIN Person.Person AS  p
ON e.BusinessEntityID = p.BusinessEntityID ;
GO
-- Query the view
SELECT FirstName, LastName, HireDate
FROM HumanResources.EmployeeHireDate
ORDER ВУ LastName [2];
```

2.3 Триггеры

Триггер — SQL Server - это часть процедурного кода, подобная хранимой процедуре, которая выполняется только при наступлении определенного события. Существуют разные типы событий, которые могут вызвать срабатывание триггера. Например, вставка строк в таблицу, изменение структуры таблицы и вход пользователя в экземпляр SQL Server.

Триггеры отличаются от хранимых процедур по трем основным характеристикам:

– пользователь не может запускать триггеры вручную;

– триггеры не принимают параметры;

– вы не можете зафиксировать или откатить транзакцию внутри триггера;

Тот факт, что нельзя использовать параметры в триггерах, не является ограничением для получения информации из события срабатывания.

В SQL Server есть два класса триггеров:

– DDL (язык определения данных) триггеры. Этот класс триггеров срабатывает при событиях, которые изменяют структуру (например, создание, изменение или удаление таблицы), или при определенных событиях, связанных с сервером, таких как изменения безопасности или события обновления статистики.

– Триггеры DML (язык модификации данных). Это наиболее часто используемый класс триггеров. В этом случае событие срабатывания является заявлением об изменении данных; это может быть оператор вставки, обновления или удаления в таблице или в представлении.

Кроме того, триггеры DML бывают разных типов:

– FOR или AFTER [INSERT, UPDATE, DELETE]: эти типы триггеров выполняются после завершения инструкции запуска (вставка, обновление или удаление).

– INSTEAD OF [INSERT, UPDATE, DELETE]: в отличие от типа FOR (AFTER), триггеры INSTEAD OF выполняются вместо оператора запуска. Другими словами, этот тип триггера заменяет инструкцию firing. Это очень полезно в тех случаях, когда вам нужно иметь перекрестную ссылочную целостность базы данных.

Триггеры применяются для обеспечения целостности данных и реализации сложной бизнес-логики. Триггер запускается сервером автоматически при попытке изменения данных в таблице, с которой он связан. Все производимые им модификации данных рассматриваются как выполняемые в транзакции, в которой выполнено действие, вызвавшее срабатывание триггера. Соответственно, в случае обнаружения ошибки или нарушения целостности данных может произойти откат этой транзакции.

Когда происходит запуск триггера, говорят, что он активизируется (*fire*). Триггер создается по одной таблице базы данных, но он может осуществлять доступ и к другим таблицам и объектам других баз данных. Триггеры нельзя создать по временным таблицам или системным таблицам, а только по определенным пользователем таблицам или представлениям. Таблица, по которой определяется триггер, называется «*таблицей триггера»*.

В случае если триггер вызывается до события, он может внести изменения в модифицируемую событием запись (конечно, при условии, что событие - не удаление записи). Некоторые СУБД накладывают ограничения на операторы, которые могут быть использованы в триггере (например, может быть запрещено вносить изменения в структуру таблицы, на которой «висит» триггер, и т.п.) [6].

Одной из фундаментальных характеристик реляционных баз данных является согласованность данных. Это означает, что информация, хранящаяся в базе данных, должна быть согласованной в любое время для каждого сеанса и каждой транзакции. Механизмы реляционной базы данных SQL Server реализуют это путем наложения ограничений, таких как первичные и внешние ключи. Но иногда этого бывает недостаточно. В SQL Server нет возможности обеспечить ссылочную целостность между двумя таблицами с помощью внешних ключей, если эти таблицы находятся в разных базах данных или на разных серверах. В таком случае единственный способ реализовать это - использовать триггеры. Как узнать, какие строки были обновлены, вставлены или удалены с помощью триггера DML SQL Server? В случае триггеров DML во время выполнения триггера есть две виртуальные таблицы, которые содержат данные, на которые влияет выполнение триггера. Эти таблицы называются inserted и deleted, и они имеют ту же структуру, что и их базовая таблица. Следует иметь в виду, что эти таблицы не всегда доступны вместе (т.е. у вас может быть таблица inserted, но не таблица deleted, или наоборот) [7].

2.3.1 Создание триггеров

Триггер MS SQL Server создается с использованием оператора T-SQL CREATE TRIGGER, который имеет следующий синтаксис:

CREATE TRIGGER имя триггера ON {таблица |представление}[WITHENCRYPTION]

{FOR | AFTER | INSTEAD OF}

{[DELETE] [,] [INSERT] [,][UPDATE]}[WITH APPEND]
[NOT FOR REPLICATION]
AS
оператор_sql [...n]

Как видно из этого описания, триггер может быть создан для операторов INSERT, UPDATE, DELETE или для любой комбинации из этих операторов.

Тип AFTER указывает, что триггер срабатывает только после успешного выполнения всех операций в инструкции SQL, запускаемой триггером. Все каскадные действия и проверки ограничений, на которые имеется ссылка, должны быть успешно завершены, прежде чем триггер сработает. Если единственным заданным ключевым словом является FOR, аргумент AFTER используется по умолчанию.

При вызове триггера будут выполнены операторы SQL, указанные после ключевого слова AS. Здесь могут использоваться программные конструкции, такие как IF и WHILE.

Механизм триггеров в MS SQL Server использует логические (концептуальные) таблицы с именами deleted и inserted. Их называют таблицами, но они отличаются от реальных таблиц баз данных. Они хранятся в памяти, а не на диске. Эти две таблицы имеют одинаковую структуру с таблицей (одинаковые колонки и типы данных), по которой определяется данный триггер. Таблица deleted содержит копии строк, на которые повлиял оператор DELETE или UPDATE. В эту таблицу перемещаются строки, удаляемые из таблицы данного триггера. После этого к данным таблицы deleted можно осуществлять доступ из данного триггера. Таблица inserted содержит копии строк, добавленных к таблице данного триггера при выполнении оператора INSERT или UPDATE. Эти строки добавляются одновременно в таблицу триггера и в таблицу inserted. Поскольку оператор UPDATE обрабатывается как DELETE, после которого следует INSERT, то при использовании оператора UPDATE старые значения строк копируются в таблицу deleted, а новые значения строк – в таблицу триггера и в таблицу inserted.

Создадим INSERT/UPDATE-триггер, который будет реализовывать ограничение уровня строки для таблицы *Trip* нашей базы данных *TaxiService*. Этот триггер проверит, что дата окончания поездки не стоит раньше даты начала поездки.

Для создания триггера создадим SQL-запрос к базе данных, в котором выполним следующие инструкции [6]:

```
CREATE TRIGGER EndDate ON Trip
AFTER INSERT, UPDATE
AS BEGIN
SET NOCOUNT ON;
DECLARE @end AS datetime, @start AS datetime
SELECT @start = start_date FROM inserted
```

```
SELECT @end = end_date FROM inserted
IF datediff(second, @start, @end) < 0
RAISERROR ('Поездка не может закончится раньше ее начала', 16, 10)
END
```

Триггер EndDate создан. Он появился в папке триггеров таблицы Trip (что показано на рисунке 2.1)

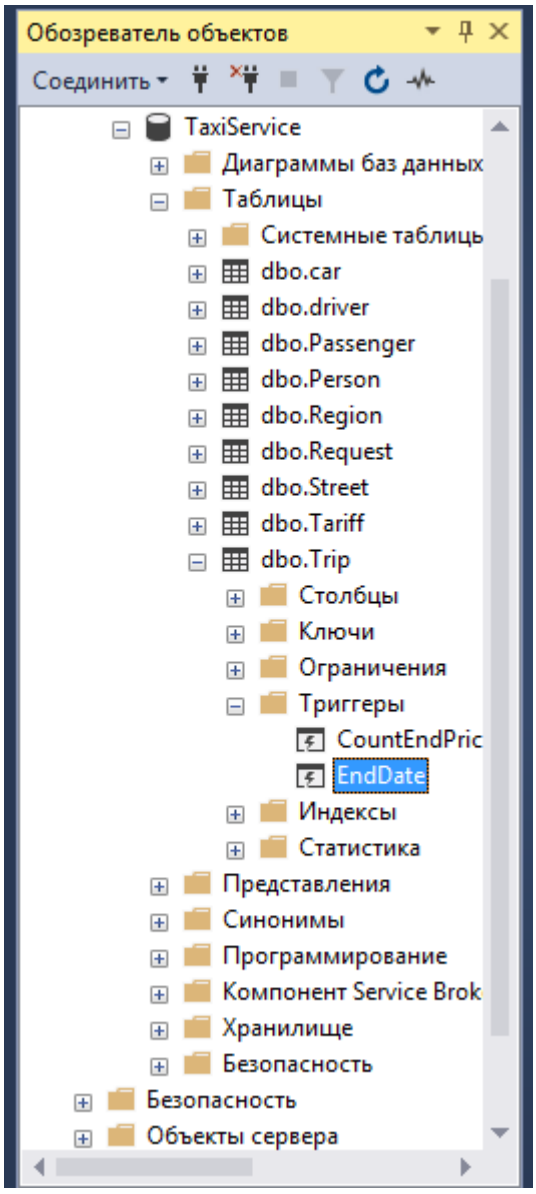

Рисунок 2.1 – Папка триггеров таблицы Trip

Чтобы проверить этот триггер, откроем таблицу Trip, выбрав в контекстном меню обозревателя объектов пункт «Изменить первые 200 строк», и попробуем ввести дату окончания меньше даты начала. СУБД отказывается сохранить некорректные данные и выдает запрограммированное нами сообщение об ошибке (что показано на рисунке 2.2) [6].

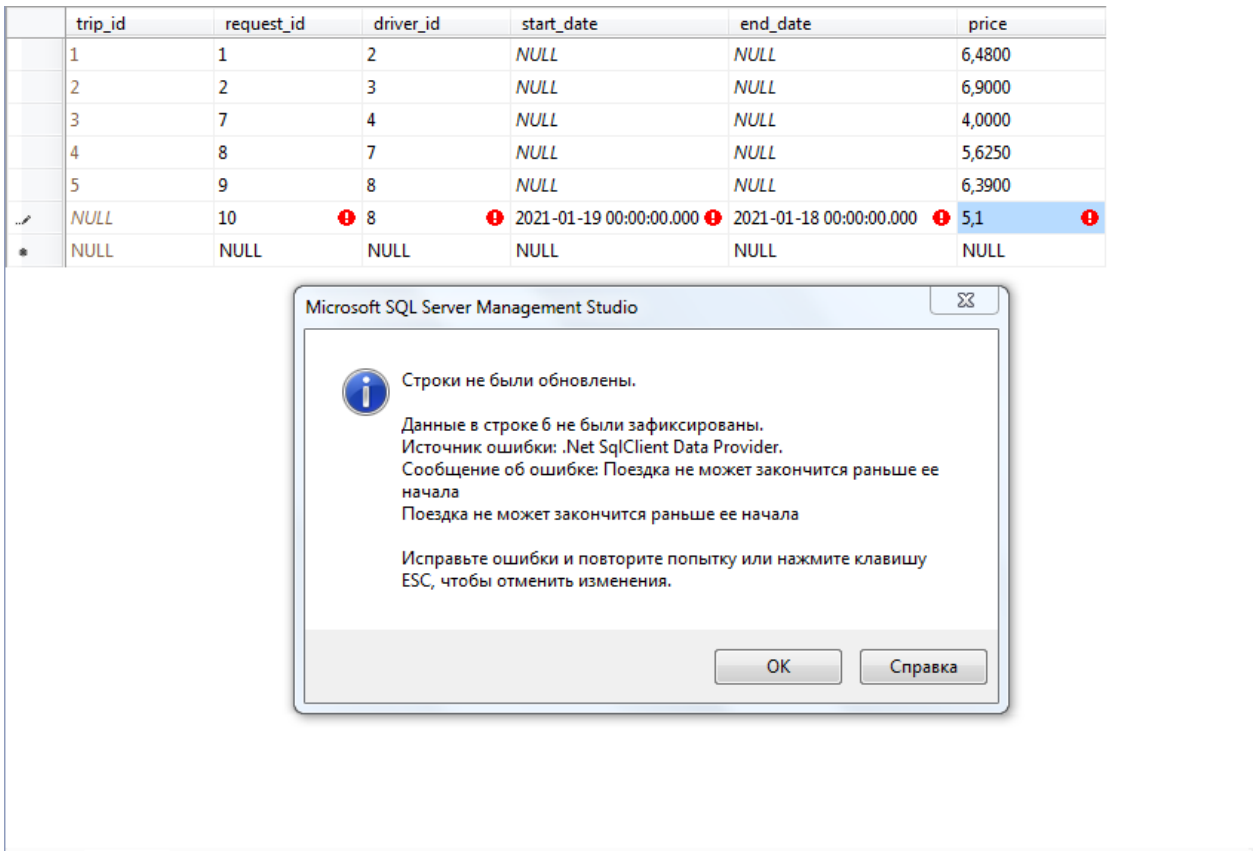

Рисунок 2.2 – Запрограммированное сообщение об ошибке

Далее будут приведены примеры ещё нескольких триггеров для БД TaxiService.

Проверка возраста для водителя (требование – 18+):

```
CREATE TRIGGER DriverAgeVerification ON driver
AFTER INSERT, UPDATE
AS BEGIN
SET NOCOUNT ON;
DECLARE @person_id AS int, @birthdate AS datetime
SELECT @person_id = person_id from inserted
SELECT @birthdate = birthdate
FROM Person
WHERE Person.person_id = @person_id
IF datediff(day,@birthdate,getdate()) / 365 < 18
RAISERROR ('Данный человек не может быть водителем, т.к. ему меньше 18 лет', 16, 10)
END
```

Триггер, проверяющий, не используется ли при создании пассажира Passenger человек Person в таблице водителей Driver:

```
CREATE TRIGGER PersonNotUsedInDriver ON Passenger
AFTER INSERT, UPDATE
```

```
AS BEGIN
SET NOCOUNT ON;
DECLARE @person_id AS int, @count_driver AS int
SELECT @person_id = person_id FROM inserted
SELECT @count_driver = COUNT(*) FROM Driver WHERE person_id = @person_id
IF @count_driver != 0
RAISERROR('Такой пользователь уже существует', 16, 10)
END
```

Триггер, проверяющий, может ли вводимый/модифицированный водитель иметь указанный опыт вождения *work_experience*:

```
CREATE TRIGGER WorkExperienceVerification ON driver
AFTER INSERT, UPDATE
AS BEGIN
SET NOCOUNT ON;
DECLARE @person_id AS int, @birthdate AS datetime, @exp AS int
SELECT @person_id = person_id from inserted
SELECT @exp = work_experience from inserted
SELECT @birthdate = birthdate
FROM Person
WHERE Person.person_id = @person_id
IF datediff (day, @birthdate, getdate ()) / 365 - 18 < @exp
RAISERROR ('Данный водитель не может иметь такой опыт работы', 16, 10)
END
```

[6].

Триггер, проверяющий, не производится ли поездка «на месте» (адрес посадки не совпадает с адресом назначения):

```
CREATE TRIGGER CheckAdress ON Request
AFTER INSERT, UPDATE
AS BEGIN
SET NOCOUNT ON;
DECLARE @request_street AS int, @request_house AS int, @destination_street AS int, @destination_house AS int
SELECT @request_street = request_street_id FROM inserted
SELECT @destination_street = destination_street_id FROM inserted
SELECT @request_house = request_house_number FROM inserted
SELECT @destination_house = destination_house_number FROM inserted
IF @request_street = @destination_street AND @request_house = @destination_house
RAISERROR ('Точка прибытия совпадает с точкой посадки', 16, 10)
```

END

2.4 Хранимые процедуры

Хранимая процедура - это специальный тип пакета инструкций Transact-SQL, созданный, используя язык SQL и процедурные расширения. Основное различие между пакетом и хранимой процедурой состоит в том, что последняя сохраняется в виде объекта базы данных. Иными словами, хранимые процедуры сохраняются на стороне сервера, чтобы улучшить производительность и постоянство выполнения повторяемых задач.

Компонент Database Engine поддерживает хранимые процедуры и системные процедуры. Хранимые процедуры создаются таким же образом, как и все другие объекты баз данных, т.е. при помощи языка DDL. Системные процедуры предоставляются компонентом Database Engine и могут применяться для доступа к информации в системном каталоге и ее модификации.

При создании хранимой процедуры можно определить необязательный список параметров. Таким образом, процедура будет принимать соответствующие аргументы при каждом ее вызове. Хранимые процедуры могут возвращать значение, содержащее определенную пользователем информацию или, в случае ошибки, соответствующее сообщение об ошибке.

Хранимая процедура предварительно компилируется перед тем, как она сохраняется в виде объекта в базе данных. Предварительно компилированная форма процедуры сохраняется в базе данных и используется при каждом ее вызове. Это свойство хранимых процедур предоставляет важную выгоду, заключающуюся в устранении (почти во всех случаях) повторных компиляций процедуры и получении соответствующего улучшения производительности. Это свойство хранимых процедур также оказывает положительный эффект на объем данных, участвующих в обмене между системой баз данных и приложениями. В частности, для вызова хранимой процедуры объемом в несколько тысяч байтов может потребоваться меньше, чем 50 байт. Когда множественные пользователи выполняют повторяющиеся задачи с применением хранимых процедур, накопительный эффект такой экономии может быть довольно значительным.

Хранимые процедуры можно также использовать для следующих целей:

управления авторизацией доступа;

для создания журнала логов о действиях с таблицами баз данных.

Использование хранимых процедур предоставляет возможность управления безопасностью на уровне, значительно превышающем уровень безопасности, предоставляемый использованием инструкций GRANT и REVOKE, с помощью которых пользователям предоставляются разные привилегии доступа. Это возможно вследствие того, что авторизация на выполнение хранимой процедуры не зависит от авторизации на модифицирование объектов, содержащихся в данной хранимой процедуре, как это описано в следующем разделе.

Хранимые процедуры, которые создают логи операций записи и/или чтения таблиц, предоставляют дополнительную возможность обеспечения безопасности базы данных. Используя такие процедуры, администратор базы данных может отслеживать модификации, вносимые в базу данных пользователями или прикладными программами.

Создание и исполнение хранимых процедур

Хранимые процедуры создаются посредством инструкции CREATE PROCEDURE, которая имеет следующий синтаксис:

```
CREATE PROC[EDURE] [schema_name.]proc_name
   [({@param1} type1 [ VARYING] [= default1] [OUTPUT])] {, …}
   [WITH {RECOMPILE | ENCRYPTION | EXECUTE AS 'user_name'}]
   [FOR REPLICATION]
   AS batch | EXTERNAL NAME method_name
```

2.4.1 Соглашения по синтаксису

Параметр schema_name определяет имя схемы, которая назначается владельцем созданной хранимой процедуры. Параметр proc_name определяет имя хранимой процедуры. Параметр @param1 является параметром процедуры (формальным аргументом), чей тип данных определяется параметром type1. Параметры процедуры являются локальными в пределах процедуры, подобно тому, как локальные переменные являются локальными в пределах пакета. Параметры процедуры - это значения, которые передаются вызывающим объектом процедуре для использования в ней. Параметр default1 определяет значение по умолчанию для соответствующего параметра процедуры. (Значением по умолчанию также может быть NULL.)

Опция OUTPUT указывает, что параметр процедуры является возвращаемым, и с его помощью можно возвратить значение из хранимой процедуры вызывающей процедуре или системе.

Как уже упоминалось ранее, предварительно компилированная форма процедуры сохраняется в базе данных и используется при каждом ее вызове. Если же по каким-либо причинам хранимую процедуру требуется компилировать при каждом ее вызове, при объявлении процедуры используется опция WITH RECOMPILE. Использование опции WITH RECOMPILE сводит на нет одно из наиболее важных преимуществ хранимых процедур: улучшение производительности благодаря одной компиляции. Поэтому опцию WITH RECOMPILE следует использовать только при частых изменениях используемых хранимой процедурой объектов базы данных.

Предложение EXECUTE AS определяет контекст безопасности, в котором должна исполняться хранимая процедура после ее вызова. Задавая этот контекст, с помощью Database Engine можно управлять выбором учетных записей

пользователей для проверки полномочий доступа к объектам, на которые ссылается данная хранимая процедура.

По умолчанию использовать инструкцию CREATE PROCEDURE могут только члены предопределенной роли сервера sysadmin и предопределенной роли базы данных db_owner или db_ddladmin. Но члены этих ролей могут присваивать это право другим пользователям с помощью инструкции GRANT CREATE PROCEDURE.

В примере ниже показано создание простой хранимой процедуры для работы с таблицей Project:

```
USE TaxiService;
GO
   CREATE PROCEDURE IncreaseeTariff (@percent INT=5)
      AS UPDATE Tariff
      SET tariff_multiplier = tariff_multiplier + Bu tariff_multiplier * @percent/100;
```

Как говорилось ранее, для разделения двух пакетов используется инструкция GO. Инструкцию CREATE PROCEDURE нельзя объединять с другими инструкциями Transact-SQL в одном пакете. Хранимая процедура IncreaseBudget увеличивает бюджеты для всех проектов на определенное число процентов, определяемое посредством параметра @percent. В процедуре также определяется значение числа процентов по умолчанию, которое применяется, если во время выполнения процедуры этот аргумент отсутствует.

Хранимые процедуры могут обращаться к несуществующим таблицам. Это свойство позволяет выполнять отладку кода процедуры, не создавая сначала соответствующие таблицы и даже не подключаясь к конечному серверу.

В отличие от основных хранимых процедур, которые всегда сохраняются в текущей базе данных, возможно создание временных хранимых процедур, которые всегда помещаются во временную системную базу данных tempdb. Одним из поводов для создания временных хранимых процедур может быть желание избежать повторяющегося исполнения определенной группы инструкций при соединении с базой данных. Можно создавать локальные или глобальные временные процедуры. Для этого имя локальной процедуры задается с одинарным символом # (#proc_name), а имя глобальной процедуры - с двойным (##proc_name).

Локальную временную хранимую процедуру может выполнить только создавший ее пользователь и только в течение соединения с базой данных, в которой она была создана. Глобальную временную процедуру могут выполнять все пользователи, но только до тех пор, пока не завершится последнее

соединение, в котором она выполняется (обычно это соединение создателя процедуры).

Жизненный цикл хранимой процедуры состоит из двух этапов: ее создания и ее выполнения. Каждая процедура создается один раз, а выполняется многократно. Хранимая процедура выполняется посредством инструкции EXECUTE пользователем, который является владельцем процедуры или обладает правом EXECUTE для доступа к этой процедуре. Инструкция EXECUTE имеет следующий синтаксис:

[[EXEC[UTE]] [@return_status =] {proc_name | @proc_name_var}
{[[@parameter1 =] value
| [@parameter1=] @variable [OUTPUT]] | DEFAULT}.
[WITH RECOMPILE]

За исключением параметра return_status, все параметры инструкции EXECUTE имеют такое же логическое значение, как и одноименные параметры инструкции CREATE PROCEDURE. Параметр return_status определяет целочисленную переменную, в которой сохраняется состояние возврата процедуры. Значение параметру можно присвоить, используя или константу (value), или локальную переменную (@variable). Порядок значений именованных параметров не важен, но значения неименованных параметров должны предоставляться в том порядке, в каком они определены в инструкции CREATE PROCEDURE.

Предложение DEFAULT предоставляет значения по умолчанию для параметра процедуры, которое было указано в определении процедуры. Когда процедура ожидает значение для параметра, для которого не было определено значение по умолчанию и отсутствует параметр, либо указано ключевое слово DEFAULT, то происходит ошибка.

Когда инструкция EXECUTE является первой инструкцией пакета, ключевое слово EXECUTE можно опустить. Тем не менее будет надежнее включать это слово в каждый пакет. Использование инструкции EXECUTE показано в примере ниже:

```
USE SampleDb;
EXECUTE IncreaseBudget 10;
```

Инструкция EXECUTE в этом примере выполняет хранимую процедуру IncreaseBudget, которая увеличивает бюджет всех проектов на 10%.

В примере ниже показано создание хранимой процедуры для обработки данных в таблицах Passenger и Person:

```
USE SampleDb;
GO
CREATE PROCEDURE ModifyEmpId (@oldId INTEGER, @newId INTEGER)
```

```
AS UPDATE Passenger
   SET passenger_id = @newId
   WHERE passenger_id = @oldId;
UPDATE Person
   SET person_id = @newId
   WHERE person_id = @oldId;
```

Процедура ModifyEmpId в примере иллюстрирует использование хранимых процедур, как часть процесса обеспечения ссылочной целостности (в данном случае между таблицами Passenger и Person). Подобную хранимую процедуру можно использовать внутри определения триггера, который собственно и обеспечивает ссылочную целостность.

В примере ниже показано использование в хранимой процедуре предложения OUTPUT:

```
USE TaxiService;
GO
CREATE PROCEDURE DeleteEmployee @empId INT, @counter INT OUTPUT
   AS SELECT @counter = COUNT(*)
      FROM Driver
      WHERE driver_id = @empId
   DELETE FROM Person
      WHERE person_id = @empId
   DELETE FROM driver
      WHERE driver_id = @empId;
```

Данную хранимую процедуру можно запустить на выполнение посредством следующих инструкций:

```
DECLARE @quantityDeleteEmployee INT;
EXECUTE DeleteEmployee @empId=18316,
@counter=@quantityDeleteEmployee OUTPUT;
PRINT N'Удалено сотрудников: ' + convert(nvarchar(30),
@quantityDeleteEmployee);
```

Эта процедура подсчитывает количество проектов, над которыми занят сотрудник с табельным номером @empId, и присваивает полученное значение параметру ©counter. После удаления всех строк для данного табельного номера из таблиц Employee и Works_on вычисленное значение присваивается переменной @quantityDeleteEmployee.

Значение параметра возвращается вызывающей процедуре только в том случае, если указана опция OUTPUT. В примере выше процедура DeleteEmployee передает вызывающей процедуре параметр @counter, следовательно, хранимая процедура возвращает значение системе. Поэтому

параметр @counter необходимо указывать как в опции OUTPUT при объявлении процедуры, так и в инструкции EXECUTE при ее вызове.

2.5 SQL-инъекции

SQL-инъекция – это тип атаки с использованием инъекций, позволяющий выполнять вредоносные операторы SQL. Эти операторы управляют сервером базы данных за веб-приложением. Злоумышленники могут использовать уязвимости SQL-инъекциям, чтобы обойти меры безопасности приложений. Они могут обходить аутентификацию и авторизацию веб-страницы или веб-приложения и извлекать содержимое всей базы данных SQL. Они также могут использовать SQL-инъекцию для добавления, изменения и удаления записей в базе данных.

Уязвимость SQL-инъекциям может затронуть любой веб-сайт или веб-приложение, использующее базу данных SQL, например, MySQL, Oracle, SQL Server или другие. Преступники могут использовать его для получения несанкционированного доступа к вашим конфиденциальным данным: информации о клиентах, личным данным, коммерческой тайне, интеллектуальной собственности и т. д. Атаки с использованием SQL-инъекций - одна из самых старых, наиболее распространенных и самых опасных уязвимостей веб-приложений. Успешная атака с использованием SQL-инъекции может иметь очень серьезные последствия.

– Злоумышленники могут использовать SQL-инъекции для поиска учетных данных других пользователей в базе данных. Затем они могут выдавать себя за этих пользователей. Так злоумышленник может получить права администратора базы данных со всеми привилегиями.

– SQL позволяет выбирать и выводить данные из базы данных. Уязвимость SQL-инъекции может позволить злоумышленнику получить полный доступ ко всем данным на сервере базы данных.

– SQL также позволяет изменять и добавлять новые данные в БД. Например, в финансовом приложении злоумышленник может использовать SQL-инъекцию для изменения балансов, аннулирования транзакций или перевода денег на свой счет.

– SQL может использоваться для удаления записей из базы данных, даже для удаления таблиц. И даже если администратор создает резервные копии базы данных, удаление данных может повлиять на доступность приложения до тех пор, пока база данных не будет восстановлена. Кроме того, резервные копии могут не охватывать самые свежие данные.

– На некоторых серверах вы можете получить доступ к операционной системе с помощью сервера базы данных. Это может быть намеренно или случайно. В таком случае злоумышленник может использовать SQL-инъекцию

в качестве исходного вектора атаки, чтобы затем атаковать внутреннюю сеть за брандмауэром [6].

Существует несколько типов атак SQL-инъекций: внутриполосные (классические), логические (слепые) и внеполосные.

Внутриполосные инъекции**.** Злоумышленник использует один и тот же канал связи для запуска своих атак и сбора результатов. Простота и эффективность внутриполосных инъекций делают их одними из наиболее распространенных типов атак SQL-инъекций. Есть два под варианта этого метода:

– Инъекции на основе ошибок - злоумышленник выполняет действия, заставляющие базу данных создавать сообщения об ошибках. Злоумышленник потенциально может использовать данные, предоставленные этими сообщениями об ошибках, для сбора информации о структуре базы данных.

– Инъекции на основе объединения - этот метод использует преимущество оператора UNION SQL, который объединяет несколько операторов выбора, сгенерированных базой данных, для получения их в одном ответе. Этот ответ может содержать данные, которые могут быть использованы злоумышленником.

– Логические (слепые) инъекции**.**

Злоумышленник отправляет на сервер инъекции и наблюдает за реакцией и поведением сервера, чтобы узнать больше о его структуре. Этот метод называется слепой инъекцией, потому что данные из базы данных не передаются злоумышленнику, поэтому злоумышленник не может видеть информацию об атаке внутри канала.

Слепые SQL-инъекции зависят от реакции и поведенческих паттернов сервера, поэтому они обычно медленнее выполняются, но также опасны, как и другие виды инъекций. Слепые SQL-инъекции можно классифицировать следующим образом:

– Логические - этот злоумышленник отправляет SQL-запрос к базе данных, предлагая приложению вернуть результат. Результат будет зависеть от того, является ли запрос истинным или ложным. В зависимости от результата информация в ответе изменится или останется неизменной. Затем злоумышленник может определить, было ли сообщение верным или ложным.

– По времени - злоумышленник отправляет логический SQL-запрос к базе данных, который, допустим, заставляет базу данных ждать (в течение периода в секундах), прежде чем она сможет отреагировать, если логическое выражение истинно, и, наоборот, не ждать и вернуть ответ сразу, если выражение было ложно. Таким образом, злоумышленник может определить, вернуло ли используемое им сообщение истину или ложь, не полагаясь на данные из базы данных [6].

– Внеполосные инъекции**.**

Злоумышленник может осуществить эту форму атаки только в том случае, если на сервере базы данных, используемом приложением, включены определенные функции. Эта форма атаки в основном используется как альтернатива внутриполосным и логическим методам инъекций.

Внеполосные инъекции используются, когда злоумышленник не может использовать тот же канал для запуска атаки и сбора информации, или, когда сервер слишком медленный и нестабильный для выполнения этих действий. Эти методы рассчитаны на способность сервера создавать DNS или HTTP-запросы для передачи данных злоумышленнику [8].

Злоумышленник, желающий выполнить SQL-инъекцию, манипулирует стандартным SQL-запросом, чтобы использовать непроверенные входные уязвимости в базе данных. Есть много способов, которыми может быть выполнена атака, некоторые из которых будут показаны здесь, чтобы дать вам общее представление о том, как работают инъекции.

Допустим, есть запрос, который извлекает из таблицы Person имя определенного пользователя, который выбирается по переданному в элемент queryParamBox параметру person_id (что показано на рисунке 2.3а) [6].

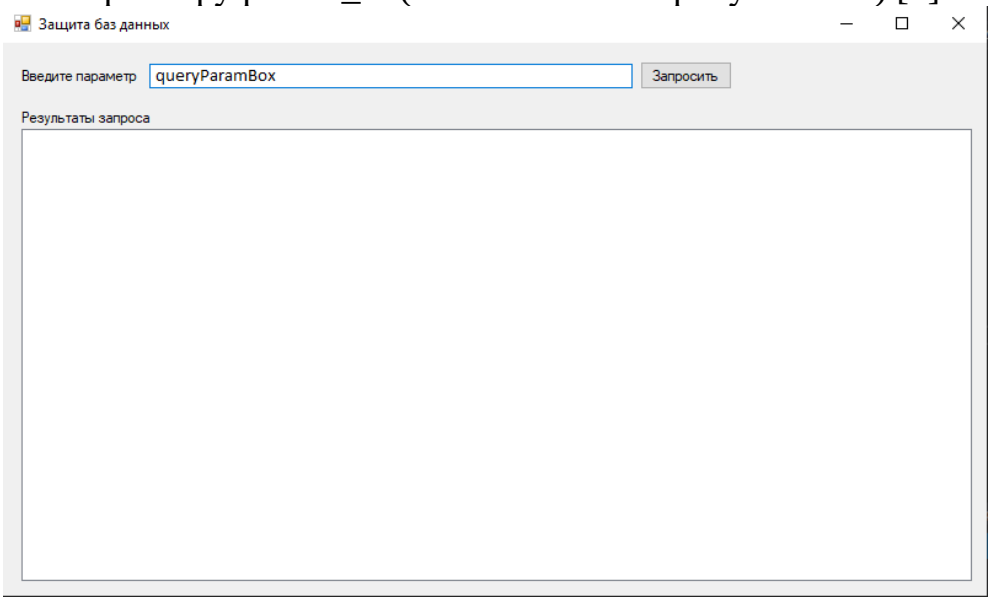

Рисунок 2.3а – Выбор параметра для получения результатов запроса

```
string query = $"SELECT first_name, last_name FROM Person WHERE person_id = {queryParamBox.Text}";
SqlCommand selectCommand = new SqlCommand(query, connection);
SqlDataReader reader = selectCommand.ExecuteReader();
```

Рисунок 2.3б – Запрос к базе данных

Мы можем передать в этот запрос строку «1 or 1=1». В результате запрос вернет все записи из таблицы, потому что выполнилось условие 1=1. Получившийся запрос указан на рисунке 2.3в.

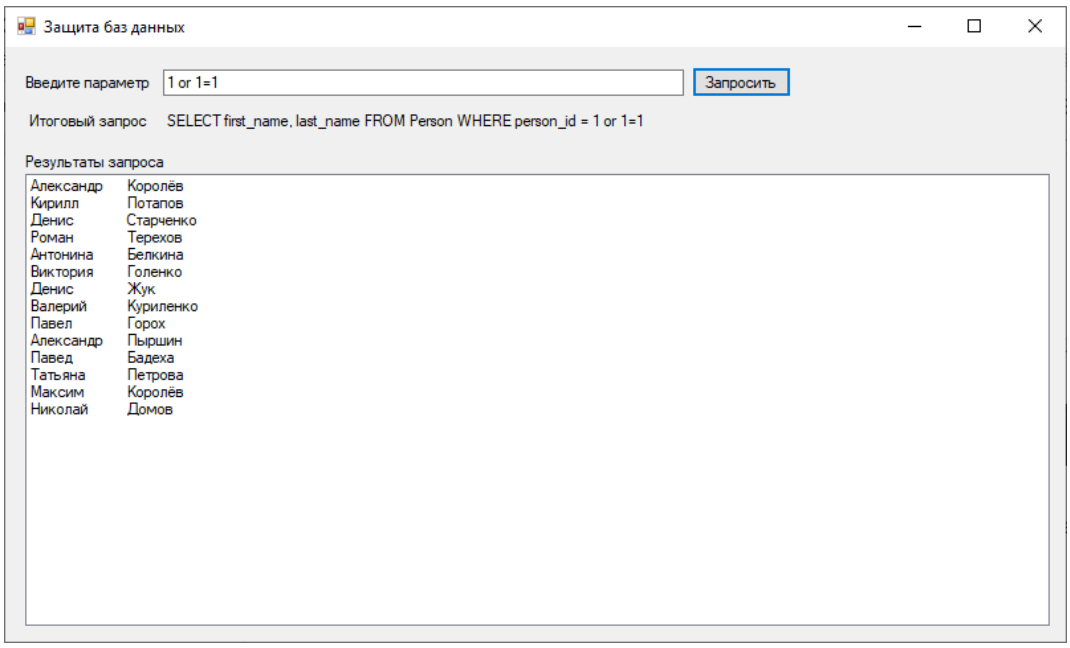

Рисунок 2.3в – Результат выполнения запроса

Другой способ обработки запросов SQL - это оператор UNION SELECT. Это объединяет два несвязанных запроса SELECT для извлечения данных из разных таблиц.

Например, в строку параметра введем «1 UNION SELECT car_brand, car_model FROM car». На выходе кроме имени пользователя с person_id = 1 также будут получены все названия машин из таблицы cars (что показано на рисунке 2.4) [6].

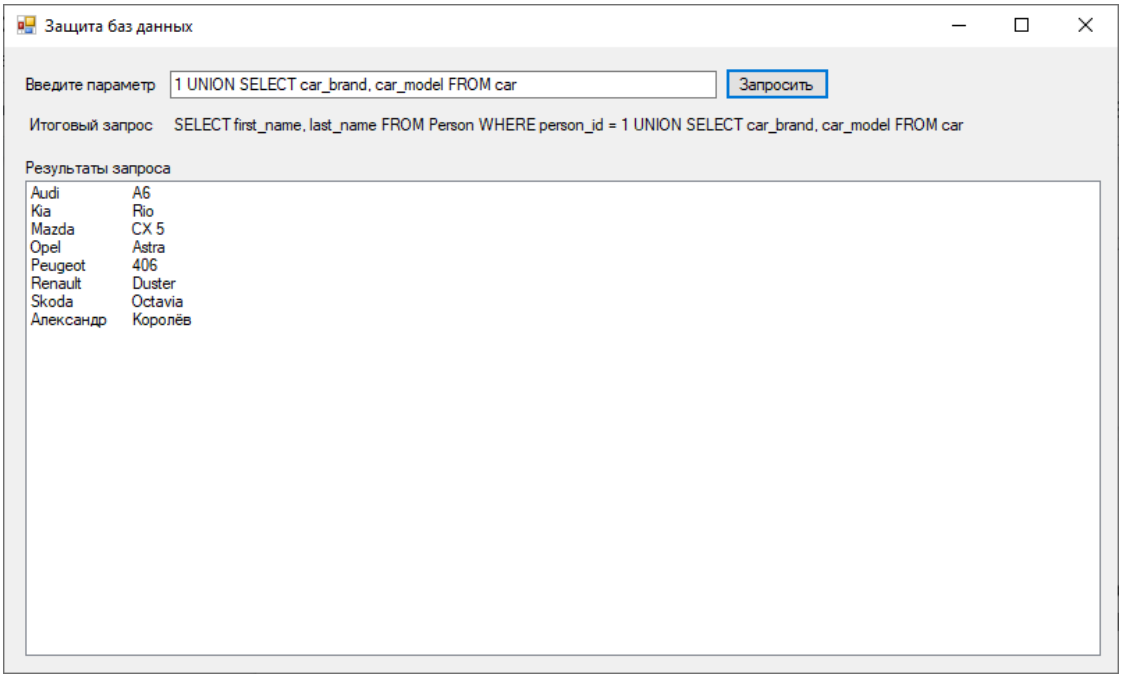

Рисунок 2.4 – Результат выполнения запроса

Злоумышленники также могут использовать неправильно отфильтрованные символы для изменения команд SQL, включая использование точки с запятой для разделения двух полей.

Например, если строкой параметра передать «1; DROP TABLE car», злоумышленник отправит запрос, который удалит таблицу cars из базы данных.

Удалённые таким способом данные могут быть восстановлены с помощью резервных копий, однако работа приложения может быть приостановлена до момента восстановления нарушенных данных. Однако всё равно некоторые из последних изменений тех данных могут быть утрачены [6].

2.5.1 Защита от SQL-инъекций

Недостатки внедрения SQL возникают, когда разработчики программного обеспечения создают динамические запросы к базе данных, которые включают вводимые пользователем данные. Чтобы защитить базу данных от атак с использованием SQL-инъекций, можно применить некоторые из этих основных методов [9]:

2.5.1.1 Использование подготовленных выражений (с параметризованными запросами)

Использование подготовленных выражений - один из лучших способов предотвратить внедрение SQL. Его также проще написать и легче понять, чем динамические запросы SQL.

В подготовленных выражениях команда SQL использует параметр вместо того, чтобы вставлять значения непосредственно в команду, тем самым предотвращая выполнение серверной частью вредоносных запросов, которые вредны для базы данных. Таким образом, если пользователь ввел «1 OR 1=1» в качестве входных данных, параметризованный запрос будет искать в таблице совпадение со всей строкой «1 OR 1=1» [9].

Рекомендации для конкретных языков:

Java EE - PreparedStatement() с переменными связывания

.NET - параметризованные запросы, такие как SqlCommand() или OleDbCommand() с переменными связывания

PHP - PDO со строго типизированными параметризованными запросами.

Перепишем используемый ранее запрос в базу данных (что показано на рисунке 2.3б) с использованием подготовленных выражений (что показано на рисунке 2.5).

```
string query = $"SELECT first_name, last_name FROM Person WHERE person_id = @person_id";
SqlCommand selectCommand = new SqlCommand(query, connection);

SqlParameter person_id = new SqlParameter("@person_id", System.Data.SqlDbType.Int);
person_id.Value = int.Parse(queryParamBox.Text);
selectCommand.Parameters.Add(person_id);

selectCommand.Prepare();
SqlDataReader reader = selectCommand.ExecuteReader();
```

Рисунок 2.5. – Запрос с использованием подготовленных выражений

В данном коде также появилось требование к типу вводимого параметра, и, поэтому, если вводимая строка не будет целочисленным значением, нам выдаст соответствующее сообщение (как показано на рисунке 2.6) [6].

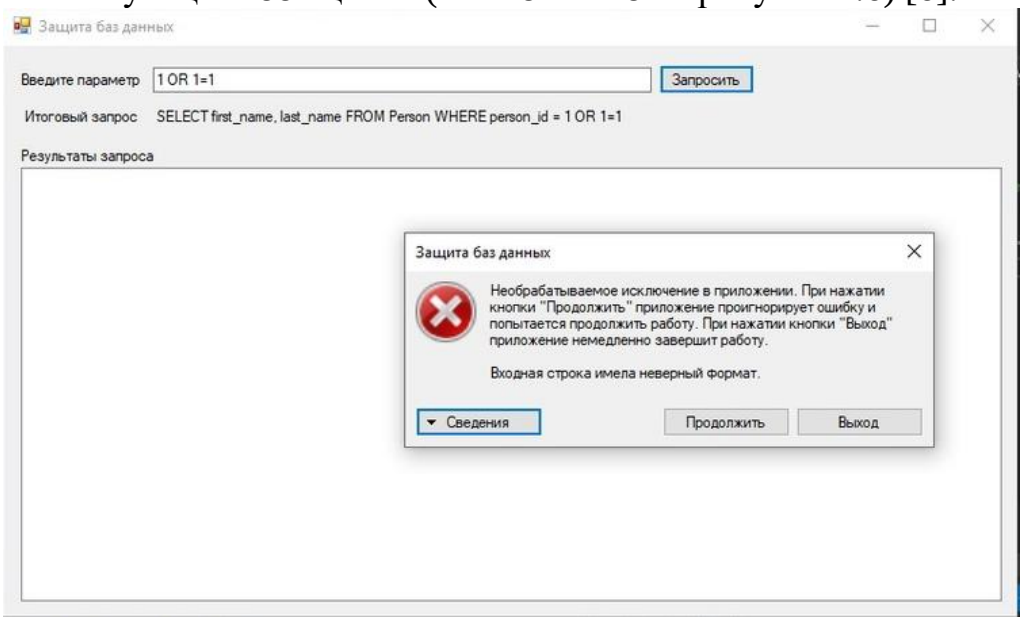

Рисунок 2.6. – Сообщение об ошибке

Разработчикам, как правило, нравится подход с подготовленными выражениями, потому что весь код SQL остается в приложении. Это делает приложение относительно независимым от базы данных.

2.5.1.2 Использование хранимых процедур

Хранимые процедуры добавляют дополнительный уровень безопасности в базу данных помимо использования подготовленных операторов. Он выполняет экранирование, необходимое для того, чтобы приложение рассматривало ввод как данные, над которыми нужно работать, а не как код SQL, который нужно выполнить.

Разница между подготовленными операторами и хранимыми процедурами заключается в том, что код SQL для хранимой процедуры записывается и хранится на сервере базы данных, а затем вызывается из веб-приложения.

Если доступ пользователей к базе данных разрешен только через хранимые процедуры, разрешение пользователям на прямой доступ к данным не нужно явно предоставлять для какой-либо таблицы базы данных. Таким образом, база данных останется в безопасности [6].

2.5.1.3 Проверка ввода данных пользователем

Даже когда используются подготовленные операторы, сначала нужно выполнить проверку ввода, чтобы убедиться, что значение имеет принятый тип, длину, формат и т. д. Только ввод, прошедший проверку, может быть обработан

в базе данных. Этот метод может остановить только самые тривиальные атаки, но он не устраняет основную уязвимость.

2.5.1.4 Ограничение привилегий

Не подключайтесь к базе данных с помощью учетной записи администратора, если это не требуется, потому что злоумышленники будут иметь возможность получить доступ ко всей системе. Поэтому лучше использовать учетную запись с ограниченными привилегиями, чтобы ограничить объем ущерба в случае SQL-инъекции.

2.5.1.5 Скрытие информации из сообщения об ошибке.

Сообщения об ошибках полезны для злоумышленников, чтобы узнать больше об архитектуре вашей базы данных, поэтому лучше показать общее сообщение об ошибке, говорящее о том, что что-то идет не так, и побудить пользователей обратиться в службу технической поддержки, если проблема не исчезнет.

2.5.1.6 Обновление вашей системы

Уязвимость SQL-инъекции является частой ошибкой программирования и обнаруживается регулярно, поэтому очень важно применять исправления и обновлять вашу систему до самой последней версии, насколько это возможно, особенно для вашего SQL Server.

2.5.1.7 Хранение учетных данных базы данных отдельно и в зашифрованном виде.

Если вы думаете, где хранить учетные данные вашей базы данных, также подумайте, насколько опасным может быть попадание в чужие руки. Поэтому всегда храните учетные данные своей базы данных в отдельном файле и надежно зашифруйте его, чтобы злоумышленники не смогли получить большую выгоду. Кроме того, не храните конфиденциальные данные, если они вам не нужны, и удаляйте информацию, когда она больше не используется [8].

2.6 Резервное копирование БД.

Метод 1: Использование плана обслуживания для настройки резервного копирования SQL Server.

Планы обслуживания SQL Server это самый распространенный способ настройки регулярного резервного копирования.

Рассмотрим настройку резервного базы данных на SQL Server копирования по плану:

– Полная резервная копия каждые 24 часа

– Копия журнала транзакций – каждые 30 минут

В SSMS (SQL Server Management Studio) перейдите в раздел Management -> Maintenance Planes и запустите -> мастер создания плана обслуживания (как показано на рисунке 2.7).

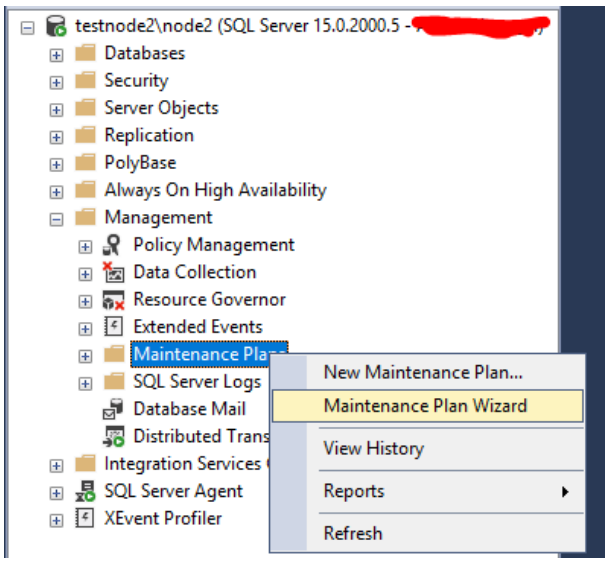

Рисунок 2.7 – Переход в мастер плана обслуживания

Укажите имя плана и выберите режим «Separate schedules for each task» (как показано на рисунке 2.8).

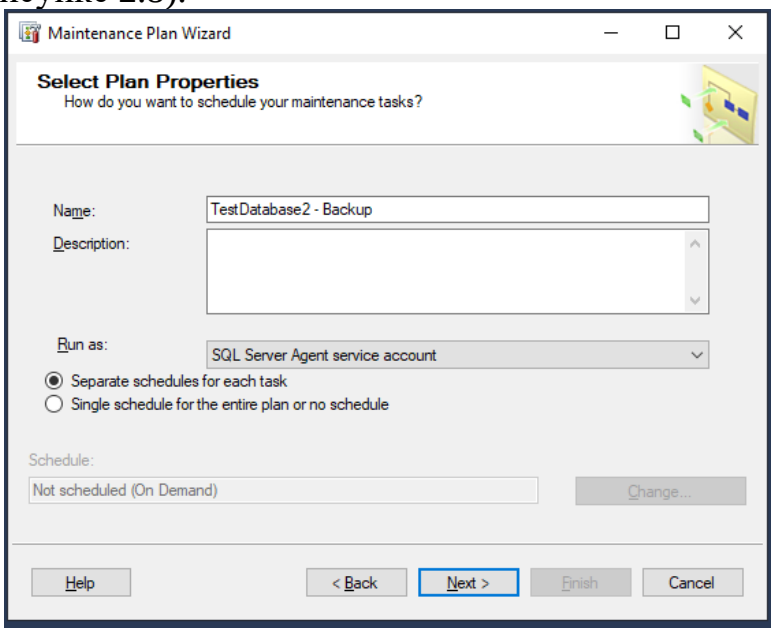

Рисунок 2.8 – Задание имени плана

Выберите операции, которые нужно сделать в этом плане обслуживания (как показано на рисунке 2.9):

– Back Up Database (Full);

– Back Up Database (Transaction Log).

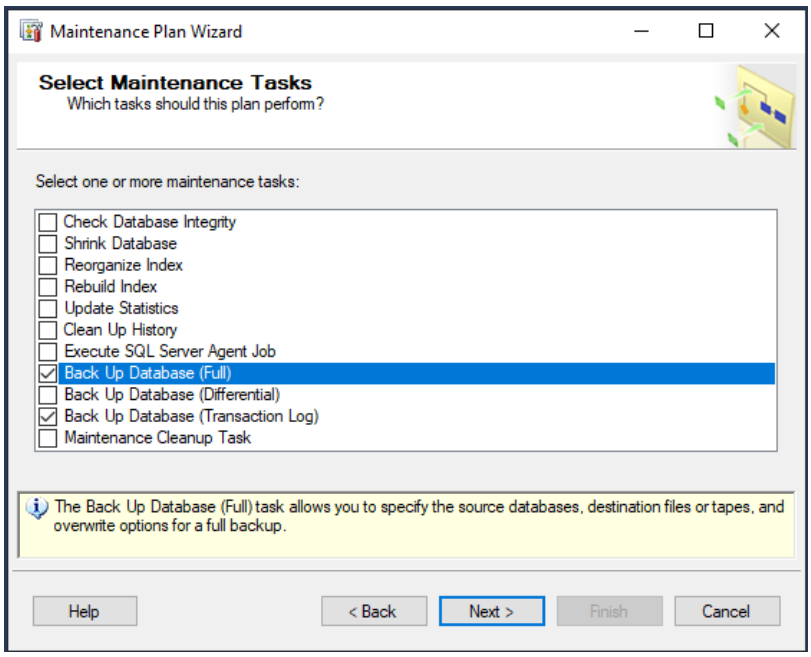

Рисунок.2.9 – Выбор опция для выполнения в рамках плана обслуживания

Используйте следующую последовательность операций (как показано на рисунке 2.10):

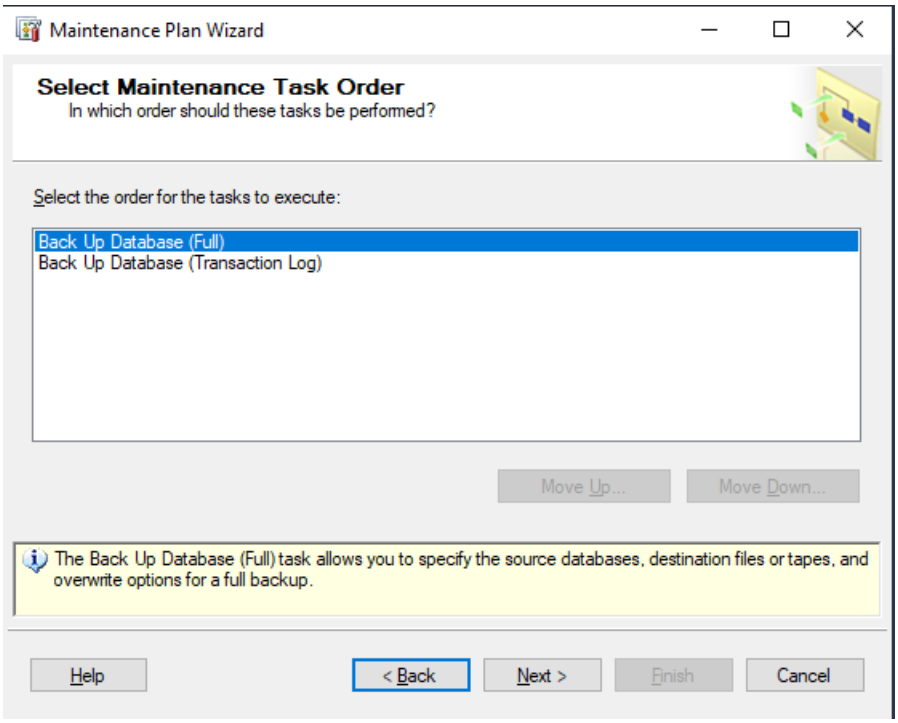

Рисунок.2.10 – Последовательность выполнения операций

Выберите базу данных SQL Server, которую нужно бэкапить (как показано на рисунке 2.11) и выберите расписание (как показано на рисунке 2.12).

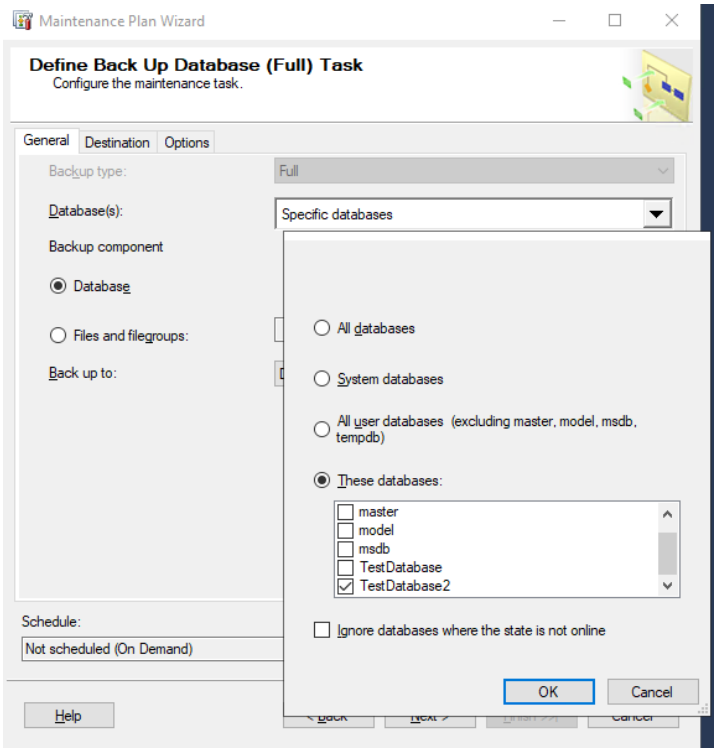

Рисунок 2.11 – Выбор нужной базы данных

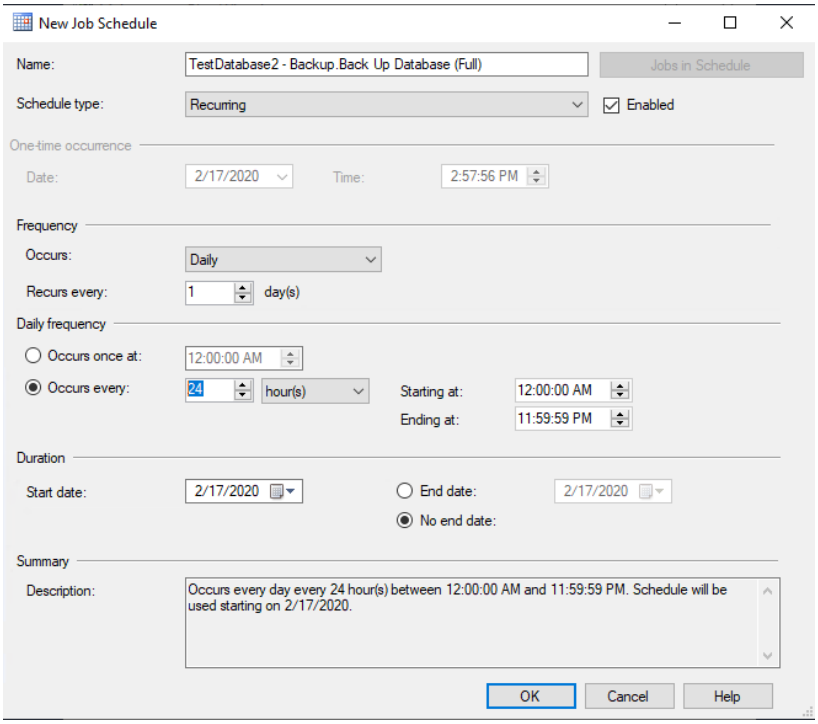

Рисунок 2.12 – Выбор расписания

Укажите путь к каталогу, в который нужно сохранять резервные копию ваше базы данных (как показано на рисунке 2.13).

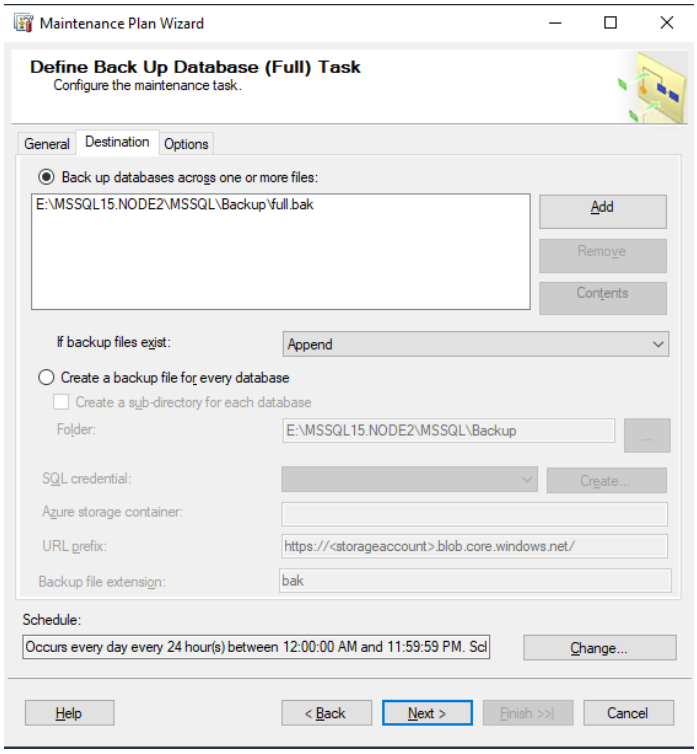

Рисунок 2.13 – Указание пути к каталогу для сохранения резервной копии

Укажите сколько будут храниться резервные копии (например, 14 дней).

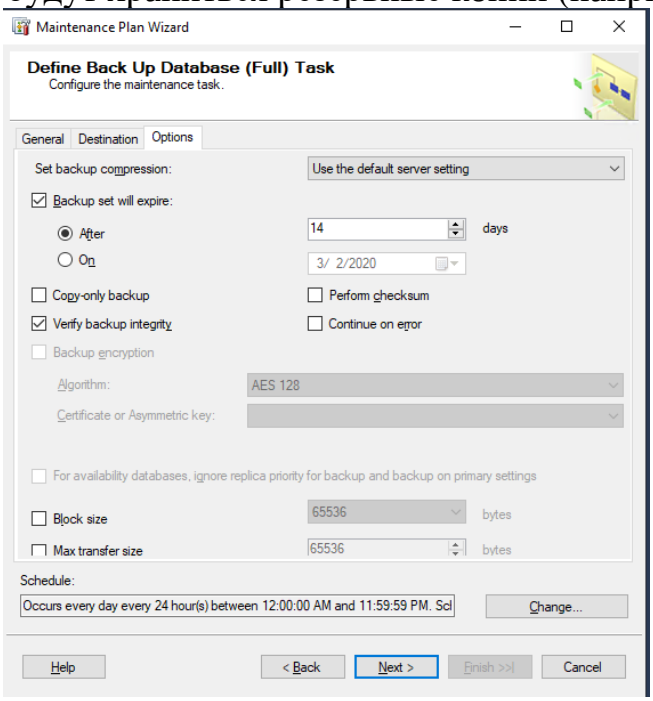

Рисунок 2.14 – Настройка срока хранения резервных копий

Нажмите Next и аналогично создайте расписание резервного копирования для журнала транзакций (как показано на рисунке 2.15).

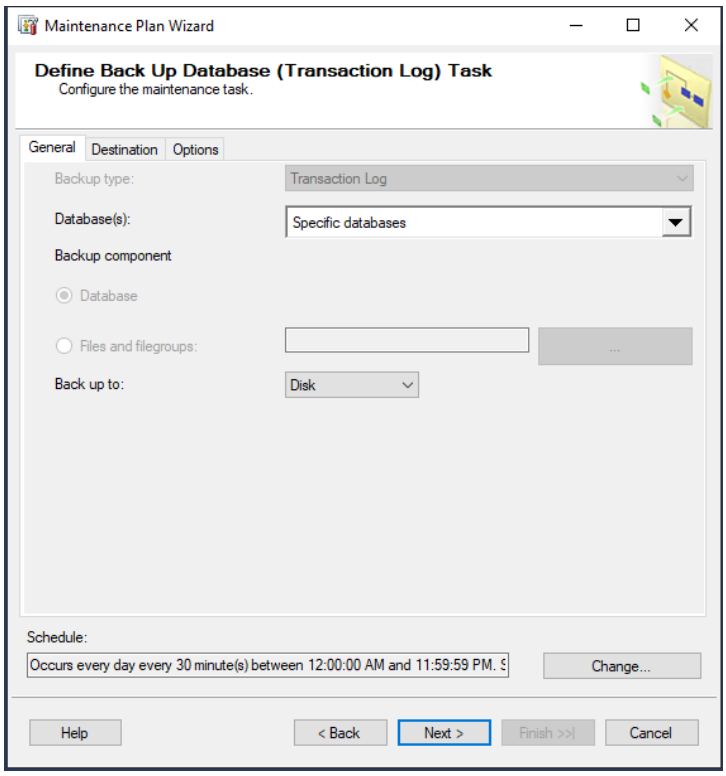

Рисунок 2.15 – Создание расписания резервного копирования для журнала транзакций

Опционально можно указать файл для ведения лога плана обслуживания (как показано на рисунке 2.16).

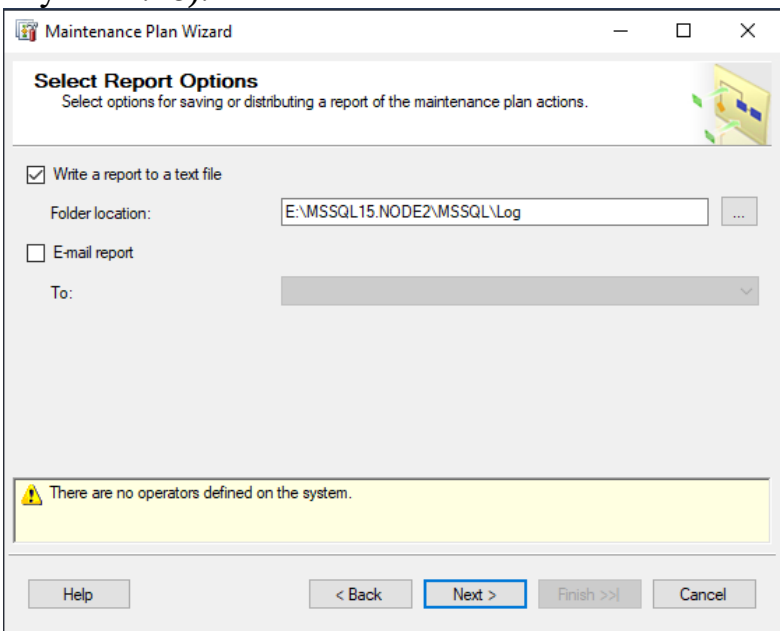

Рисунок 2.16 - Указание файла для ведения лога плана обслуживания

Завершение настройки плана обслуживания SQL Server (как показано на рисунке 2.17).

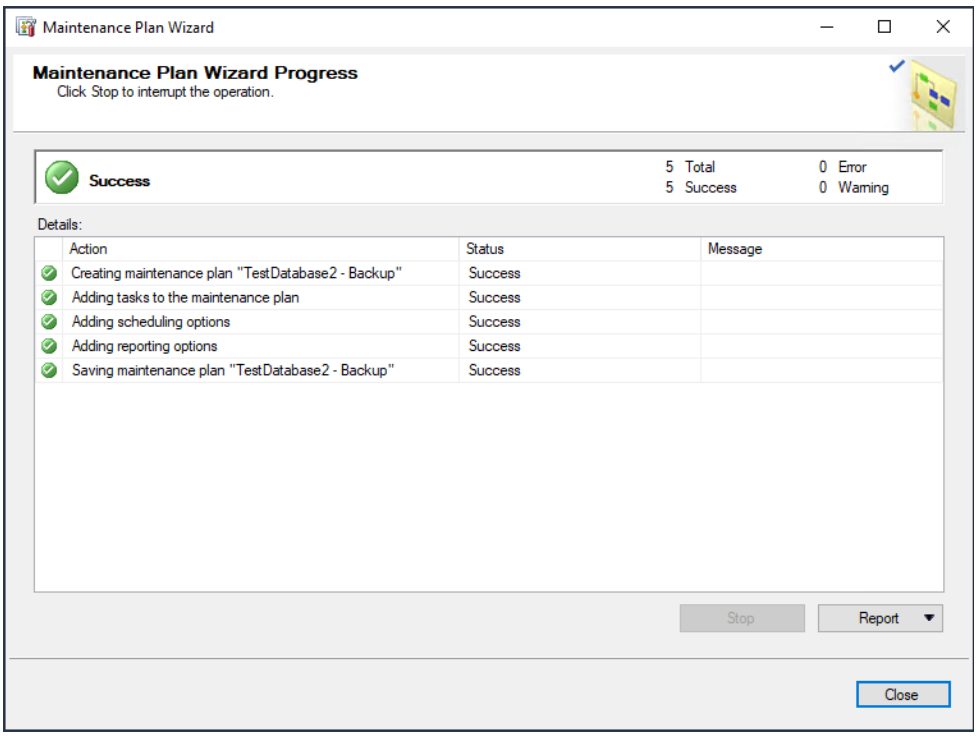

Рисунок 2.17 – Завершение настройки плана обслуживания SQL Server

Выполните план обслуживания вручную (как показано на рисунке 2.18) и проверьте журнал (как показано на рисунке 2.19).

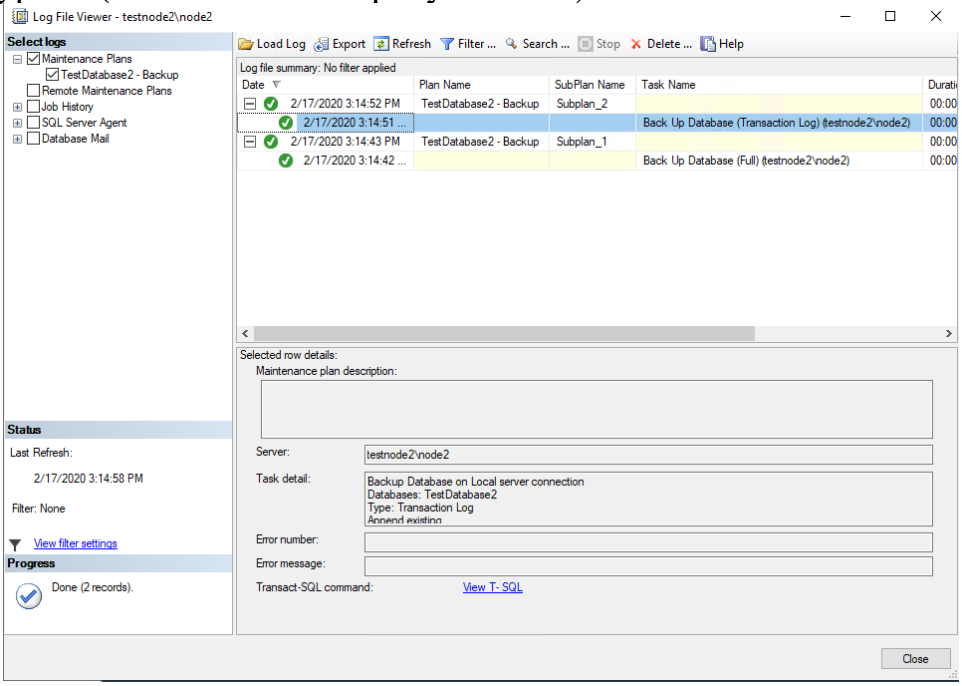

Рисунок 2.18 – Демонстрация журнала плана обслуживания

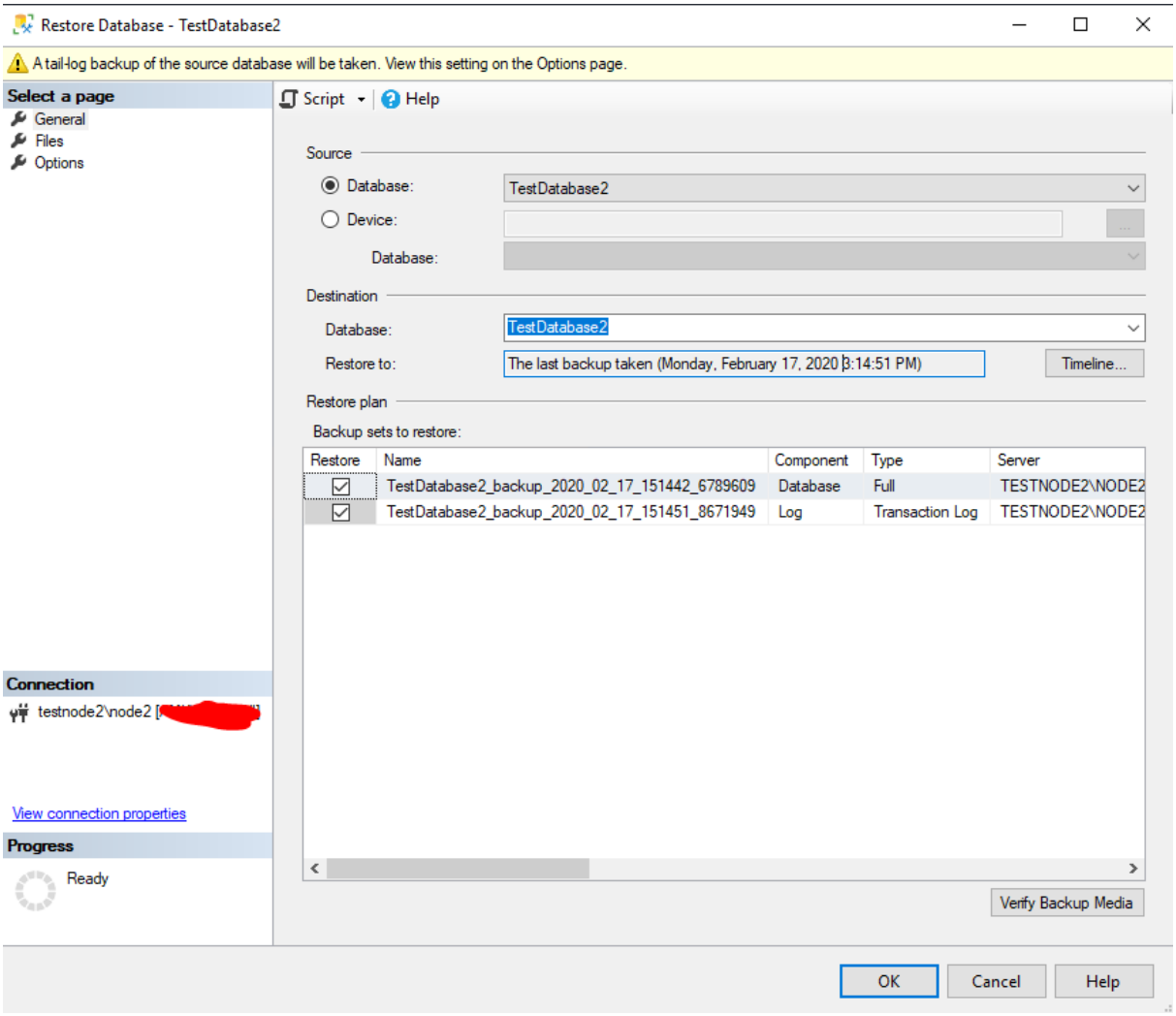

Рисунок 2.19 – Демонстрация созданных копий

Как вы видите была создана полная резервная копия базы данных SQL Server и следом копия журнала транзакций. На этом настройка резервного копирования закончена.

Метод 2: Использование T-SQL для создания резервной копии на SQL Server

T-SQL - проверенный и надежный метод резервного копирования баз данных. При использовании T-SQL доступно больше опций для создания бэкапов, чем при использовании графического интерфейса. Большинство этих опций являются более продвинутыми. Очень базовый пример команды backup, которая создает полную резервную копию, представлен ниже. Затем следуют примеры дифференциального бэкапа и бэкапа журнала.

BACKUP DATABASE [MyDB] TO DISK = 'C:\Program Files\Microsoft SQL Server\MSSQL14\MSSQL\Backup\MyDB_Full.bak'

BACKUP DATABASE [MyDB] TO DISK = 'C:\Program Files\Microsoft SQL Server\MSSQL14\MSSQL\Backup\MyDB_Differntial.bak' WITH DIFFERENTIAL

BACKUP LOG [MyDB] TO DISK = 'C:\Program Files\Microsoft SQL Server\MSSQL14\MSSQL\Backup\MyDB_log.trn' [10].

2.7 Транзакции

Транзакция – это последовательность операций, которые выполняется в логическом порядке пользователем, или программой, которая осуществляет работы с БД.

По сути, транзакция – это архив для запросов к базе данных, он защищает данные по принципу «всё или ничего».

В качестве примера можно привести следующее, допустим, вы решили отправить 10 файлов, какие есть варианты?

1. Отправить каждый файл по-отдельности
2. Отправить все файлы вместе

Казалось бы, что особой разницы как поступить – нет, но что, если соединение с интернетом во время отправки файлов прервётся? Если мы выбрали первый случай, то получатель получит 9 файлов, но не получит 1. Во втором случае получатель не получит ничего. На этом простом примере как раз и можно увидеть главное преимущество транзакций и то, почему они используются повсеместно – принцип «всё или ничего», получатель либо получит всю отправленную ему информацию, либо не получит ничего.

2.7.1 Использование транзакций

Транзакция задает последовательность инструкций языка Transact-SQL, применяемую программистами базы данных для объединения в один пакет операций чтения и записи для того, чтобы система базы данных могла обеспечить согласованность данных. Существует два типа транзакций:

Неявная транзакция – задает любую отдельную инструкцию INSERT, UPDATE или DELETE как единицу транзакции.

Явная транзакция - обычно это группа инструкций языка Transact-SQL, начало и конец которой обозначаются такими инструкциями, как BEGIN TRANSACTION, COMMIT и ROLLBACK.

2.7.2 Свойства транзакций

Транзакции обладают следующими свойствами, которые все вместе обозначаются сокращением ACID (Atomicity, Consistency, Isolation, Durability):

атомарность (Atomicity);

согласованность (Consistency);

изолированность (Isolation);

долговечность (Durability).

Свойство атомарности обеспечивает неделимость набора инструкций, который модифицирует данные в базе данных и является частью транзакции. Это означает, что или выполняются все изменения данных в транзакции, или в случае любой ошибки осуществляется откат всех выполненных изменений.

Свойство согласованности обеспечивает, что в результате выполнения транзакции база данных не будет содержать несогласованных данных. Иными словами, выполняемые транзакцией трансформации данных переводят базу данных из одного согласованного состояния в другое.

Свойство изолированности отделяет все параллельные транзакции друг от друга. Иными словами, активная транзакция не может видеть модификации данных в параллельной или незавершенной транзакции. Это означает, что для обеспечения изоляции для некоторых транзакций может потребоваться выполнить откат.

Свойство долговечности обеспечивает одно из наиболее важных требований баз данных: сохраняемость данных. Иными словами, эффект транзакции должен оставаться действенным даже в случае системной ошибки. По этой причине, если в процессе выполнения транзакции происходит системная ошибка, то осуществляется откат для всех выполненных инструкций этой транзакции.

Инструкции Transact-SQL и транзакции

Для работы с транзакциями язык Transact-SQL предоставляет некоторые инструкции. Инструкция BEGIN TRANSACTION запускает транзакцию. Синтаксис этой инструкции выглядит следующим образом:

```
BEGIN TRANSACTION [ {transaction_name | @trans_var }
   [WITH MARK ['description']]]
```

2.7.3 Соглашения по синтаксису

В параметре transaction_name указывается имя транзакции, которое можно использовать только в самой внешней паре вложенных инструкций BEGIN TRANSACTION/COMMIT или BEGIN TRANSACTION/ROLLBACK. В параметре @trans_var указывается имя определяемой пользователем переменной, содержащей действительное имя транзакции. Параметр WITH MARK указывает, что транзакция должна быть отмечена в журнале. Аргумент description - это строка, описывающая эту отметку. В случае использования параметра WITH MARK требуется указать имя транзакции.

Инструкция BEGIN DISTRIBUTED TRANSACTION запускает распределенную транзакцию, которая управляется Microsoft Distributed Transaction Coordinator (MS DTC - координатором распределенных транзакций Microsoft). Распределенная транзакция - это транзакция, которая используется на нескольких базах данных и на нескольких серверах. Поэтому для таких транзакций требуется координатор для согласования выполнения инструкций на всех вовлеченных серверах. Координатором распределенной транзакции является сервер, запустивший инструкцию BEGIN DISTRIBUTED TRANSACTION, и поэтому он и управляет выполнением распределенной транзакции.

Инструкция COMMIT WORK успешно завершает транзакцию, запущенную инструкцией BEGIN TRANSACTION. Это означает, что все выполненные транзакцией изменения фиксируются и сохраняются на диск. Инструкция COMMIT WORK является стандартной формой этой инструкции. Использовать предложение WORK не обязательно.

Язык Transact-SQL также поддерживает инструкцию COMMIT TRANSACTION, которая функционально равнозначна инструкции COMMIT WORK, с той разницей, что она принимает определяемое пользователем имя транзакции. Инструкция COMMIT TRANSACTION является расширением языка Transact-SQL, соответствующим стандарту SQL.

В противоположность инструкции COMMIT WORK, инструкция ROOLBACK WORK сообщает о неуспешном выполнении транзакции. Программисты используют эту инструкцию, когда они полагают, что база данных может оказаться в несогласованном состоянии. В таком случае выполняется откат всех произведенных инструкциями транзакции изменений. Инструкция ROOLBACK WORK является стандартной формой этой инструкции. Использовать предложение WORK не обязательно. Язык Transact-SQL также поддерживает инструкцию ROLLBACK TRANSACTION, которая функционально равнозначна инструкции ROOLBACK WORK, с той разницей, что она принимает определяемое пользователем имя транзакции.

Инструкция SAVE TRANSACTION устанавливает точку сохранения внутри транзакции. Точка сохранения (savepoint) определяет заданную точку в транзакции, так что все последующие изменения данных могут быть отменены без отмены всей транзакции. (Для отмены всей транзакции применяется инструкция ROLLBACK.) Инструкция SAVE TRANSACTION в действительности не фиксирует никаких выполненных изменений данных. Она только создает метку для последующей инструкции ROLLBACK, имеющей такую же метку, как и данная инструкция SAVE TRANSACTION.

Использование инструкции SAVE TRANSACTION показано в примере ниже:

```
USE TaxiService;
BEGIN TRANSACTION;
   INSERT INTO Region (region_id, region_name)
      VALUES (1, Minsk region');
   SAVE TRANSACTION a;
   INSERT INTO Region (region_id, region_name)
      VALUES (1, 'Gomel region');
   SAVE TRANSACTION b;
INSERT INTO Region (region_id, region_name)
      VALUES (1, Brest region');
```

```
    ROLLBACK TRANSACTION b;
    INSERT INTO Region (region_id, region_name)
      VALUES (1, Vitebsk region');
    ROLLBACK TRANSACTION a;
COMMIT TRANSACTION;
```

Единственной инструкцией, которая выполняется в этом примере, является первая инструкция INSERT. Для третьей инструкции INSERT выполняется откат с помощью инструкции ROLLBACK TRANSACTION b, а для двух других инструкций INSERT будет выполнен откат инструкцией ROLLBACK TRANSACTION a.

Инструкция SAVE TRANSACTION в сочетании с инструкцией IF или WHILE является полезной возможностью, позволяющей выполнять отдельные части всей транзакции. С другой стороны, использование этой инструкции противоречит принципу работы с базами данных, гласящему, что транзакция должна быть минимальной длины, поскольку длинные транзакции обычно уменьшают уровень доступности данных.

Каждая инструкция Transact-SQL всегда явно или неявно принадлежит к транзакции. Для удовлетворения требований стандарта SQL компонент Database Engine предоставляет поддержку неявных транзакций. Когда сеанс работает в режиме неявных транзакций, выполняемые инструкции неявно выдают инструкции BEGIN TRANSACTION. Это означает, что для того чтобы начать неявную транзакцию, пользователю или разработчику не требуется ничего делать. Но каждую неявную транзакцию нужно или явно зафиксировать, или явно отменить, используя инструкции COMMIT или ROLLBACK соответственно. Если транзакцию явно не зафиксировать, то все изменения, выполненные в ней, откатываются при отключении пользователя.

2.8 Проблемы одновременного конкурентного доступа

Когда транзакции не изолированы друг от друга, могут возникнуть следующие проблемы:

Грязное чтение**.** Ситуации, когда одна транзакция считывает незафиксированный набор данных, над которыми работает другая транзакция. Это может вызвать проблемы, если другая транзакция завершится неудачно или откатится.

Неповторяющееся чтение**.** Ситуация, когда фрагмент данных, который считывается дважды в рамках одной транзакции, не может гарантировать, что он содержит одну и ту же информацию. Такое может возникнуть, если другая транзакция изменит запрашиваемые данные между этими считываниями.

Фантомное чтение. Случай, когда транзакция А вставляет или удаляет строку из набора данных, который транзакция В в настоящее время читает.

Пропущенное или двойное чтение. Случай, когда одна транзакция может выполнять сканирование диапазона в таблице, а другая транзакция может переместить строку так, чтобы первая транзакция прочитала ее дважды или пропустила.

Потеря обновления. Это может произойти, когда два процесса читают одни и те же данные, а затем оба пытаются обновить их одновременно, но с разными значениями. Только один из них добьется успеха, а другой пропадет [6].

Эффект Хэллоуина. Относится к ситуации, когда данные перемещаются в результирующем наборе и, следовательно, могут обновляться несколько раз.

Уровни изоляции транзакций

Уровни изоляции SQL Server используются для определения степени, в которой одна транзакция должна быть изолирована от ресурсов или изменений данных, сделанных другими параллельными транзакциями.

На каждом уровне используются разные подходы к принятию решения о том, какие блокировки используются при чтении данных и как долго удерживаются блокировки. Более низкие уровни изоляции увеличивают возможность доступа нескольких пользователей к одним и тем же данным, но, следовательно, они уменьшают и надежность. Уровни изоляции сосредоточены на блокировках, используемых при чтении, и не мешают блокировкам, полученным для защиты изменения данных.

Уровни изоляции могут быть установлены на уровне сервера, базы данных или транзакции. Изменить уровень изоляции в запросе можно следующей строкой в начале запроса:

SET TRANSACTION ISOLATION LEVEL <уровень_изоляции>

Различные уровни изоляции обычно делятся на две группы: одни описываются как пессимистические, а другие - как оптимистичные. Основное отличие состоит в том, что оптимистичные уровни пытаются уменьшить количество необходимых блокировок, но, как следствие, страдают от других накладных расходов, таких как увеличение использования системной базы данных tempdb, в которой хранятся временные таблицы. Оптимистические уровни используют управление версиями строк вместо блокировок. Здесь пойдет речь только о 4 пессимистичных уровнях.

READ UNCOMMITTED: запрос в текущей транзакции может считывать данные, измененные в другой транзакции, но еще не зафиксированные. Ядро базы данных не создает разделяемые блокировки при использовании этого уровня, что делает его наименее ограничивающим из уровней изоляции. В результате возможно, что оператор будет читать строки, которые были вставлены, обновлены или удалены, но никогда не зафиксированы в базе данных (грязное чтение). Также возможно, что данные будут изменены другой

транзакцией между несколькими операторами чтения в рамках текущей транзакции, что приведет к неповторяющимся чтениям или фантомным чтениям.

Для демонстрации работы уровней изоляции создадим простую таблицу, содержащую ключ-идентификатор и два поля, и заполним её несколькими записями (как показано на рисунке 2.20). Конструкция IF в начале кода позволит после каждой демонстрации сбрасывать таблицу в изначальное состояние [6].

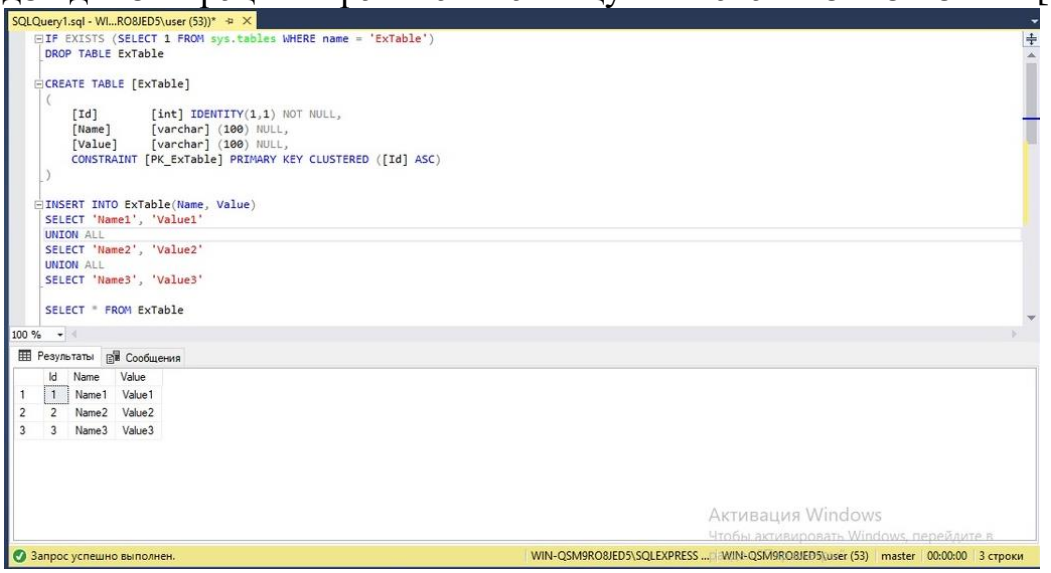

Рисунок 2.20 – Создание пустой таблице с ключом-идентификатором и двумя полями

В двух окнах SSMS подключимся к серверу, имитируя разные сеансы работы с базой данных, и будем запускать несколько сценариев транзакций:

1) Проверка на грязное чтение – попытка чтения ещё не закрепленных модификаций данных. В одном окне запустим транзакцию, которая будет изменять некоторые данные в таблице (как показано на рисунке 2.21а). В этой транзакции между операциями изменения данных будут задержки WAITFOR DELAY по 10 секунд, в которые транзакция, которую мы запустим во втором окне сразу после старта первой транзакции, будет пытаться прочитать данные из этой таблицы (как показано на рисунке.4 2.21б).

2) Проверка на неповторяющееся/фантомное чтение – изменение данных, которые транзакция считывает несколько раз. В этом сценарии первой запускается транзакция с рисунка 2.3б. В промежутках задержки во втором окне мы будем выполнять те же модификации данных, что и в первом сценарии, только каждый оператор будет выполняться как отдельная транзакция, то есть, например, в первую задержку – оператор DELETE, во вторую – операторы INSERT и UPDATE (как показано на рисунке 2.21в).

Для выполнения отдельных строк, а не всей процедуры сразу, необходимо выделить нужный для выполнения код и нажать кнопку *Выполнить*. Ориентироваться по задержкам можно с помощью времени выполнения запроса,

которое указывается в нижнем правом углу окна SSMS (как показано на рисунке 2.21г) [6].

```
BEGIN TRANSACTION
    DELETE FROM ExTable WHERE Id = 1
        WAITFOR DELAY '00:00:10'
    INSERT INTO ExTable(Name, Value) VALUES('Name4', 'Value4')
        WAITFOR DELAY '00:00:10'
    UPDATE ExTable SET Name = Name + Name WHERE Id = 2
COMMIT TRANSACTION
```

Рисунок 2.21а – Транзакция для изменения некоторых данных в таблице

```
BEGIN TRAN
    SELECT * FROM ExTable
        WAITFOR DELAY '00:00:10'
    SELECT * FROM ExTable
        WAITFOR DELAY '00:00:10'
    SELECT * FROM ExTable
ROLLBACK
```

Рисунок 2.21б – Транзакция с задержкой WAITFOR DELAУ между операциями изменения данных

```
BEGIN TRAN
    DELETE FROM ExTable WHERE Id = 1
COMMIT TRAN

BEGIN TRAN
    INSERT INTO ExTable(Name, Value) VALUES('Name4', 'Value4')
COMMIT TRAN

BEGIN TRAN
    UPDATE ExTable SET Name = Name + Name WHERE Id = 2
COMMIT TRAN
```

Рисунок 2.21в – Операторы INSERT и UPDATE

Рисунок 2.21г – Время выполнения запроса

Выполним эти сценарии для уровня READ UNCOMMITTED:

Сценарий 1**.** Транзакция с уровнем изоляции READ UNCOMMITTED смогла прочитать все ещё не зафиксированные изменения данных (как показано на рисунке 2.22а). Следовательно, этот уровень допускает грязное чтение.

Сценарий 2**.** Другие транзакции смогли изменять данные таблицы, которую считывала транзакция с данным уровнем изоляции, в следствии чего произошли неповторяющееся и фантомное чтения (как показано на рисунке. 2.22б) [6].

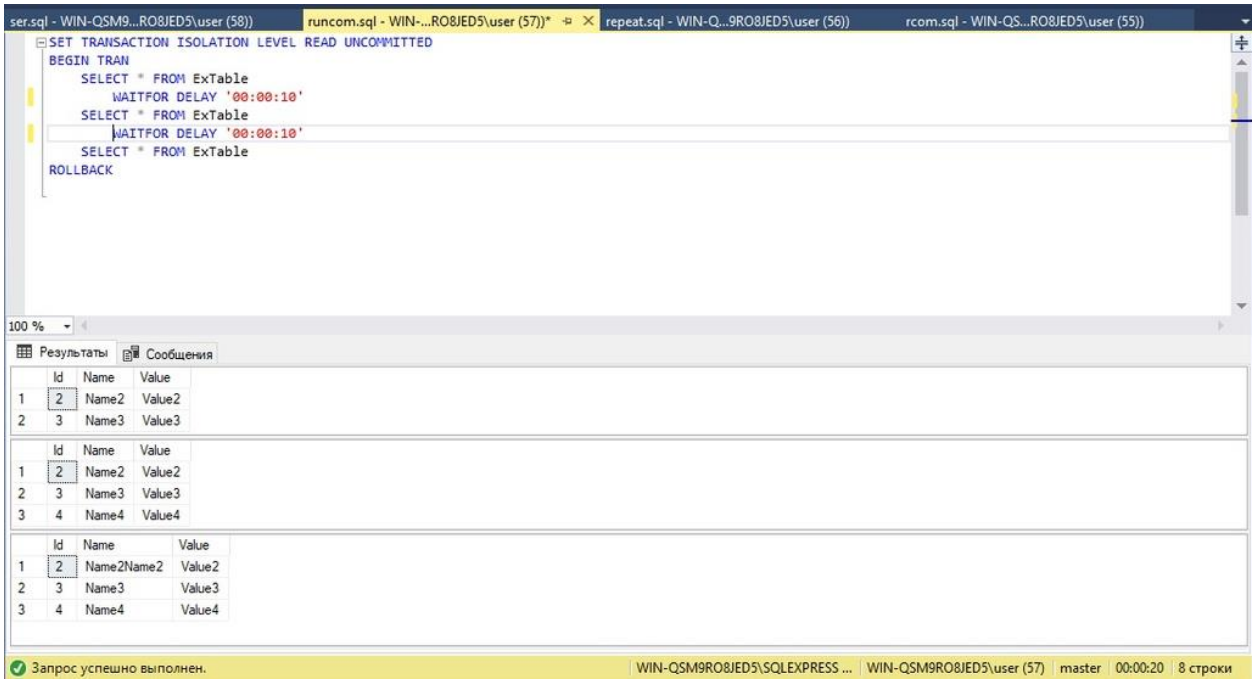

Рисунок 2.22а – Транзакция с уровнем изоляции READ UNCOMMITTED смогла прочитать все ещё не зафиксированные изменения данных

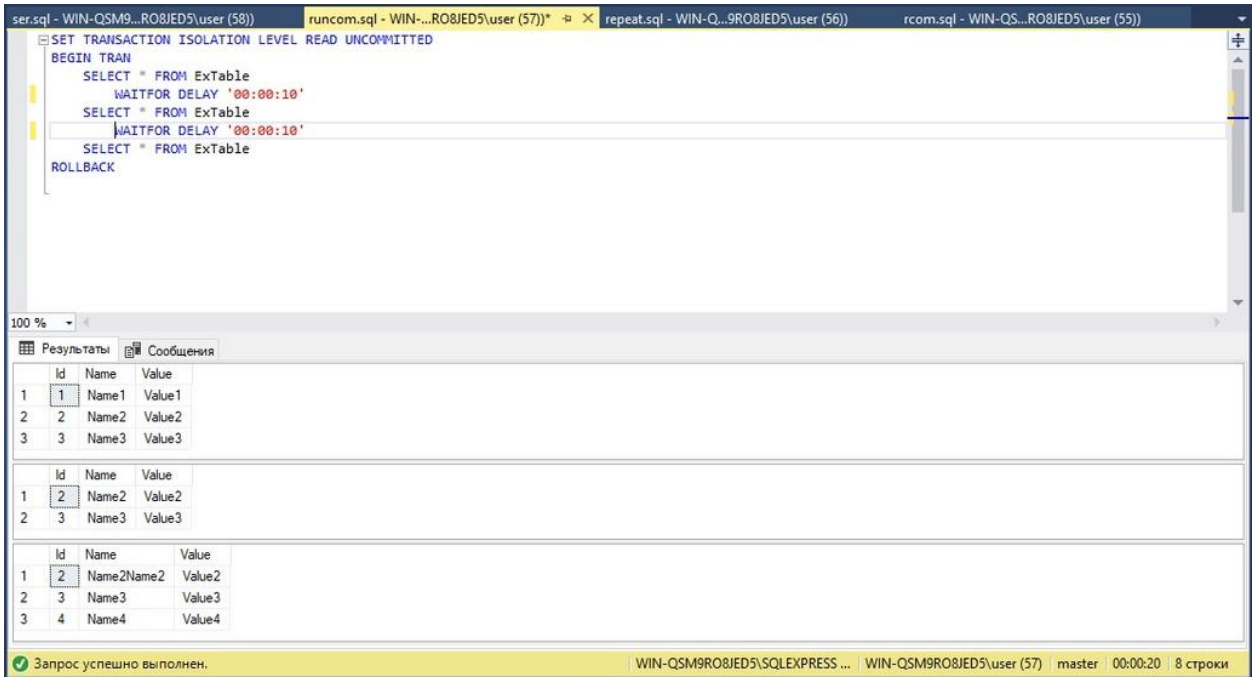

Рис 2.22б – Транзакции смогли изменять данные таблицы, которую считывала транзакция с данным уровнем изоляции

READ COMMITTED: запрос в текущей транзакции не может читать данные, измененные другой транзакцией, которая еще не зафиксирована, что предотвращает грязное чтение. Однако данные могут быть изменены другими транзакциями между выдачей операторов в рамках текущей транзакции, поэтому неповторяющиеся чтения и фантомные чтения все еще возможны. Уровень изоляции использует разделяемую блокировку для предотвращения грязного

чтения. READ COMMITTED – это уровень изоляции по умолчанию для всех баз данных SQL Server [6].

Теперь выполним сценарии для этого уровня:

Сценарий 1. Транзакция получила только конечные зафиксированные данные, так что грязное чтение здесь недопустимо (как показано на рисунке 2.23а). Кроме того, если обратить внимание на время выполнения транзакции, то можно заметить, что она выполнялась приблизительно в два раза больше времени всех задержек. Всё потому, что эта транзакция была заблокирована, и ожидала, пока другая транзакция закончит изменения и снимет блокировку с таблицы. Если в транзакции с чтением убрать все WAITFOR и оставить только один оператор SELECT, то время выполнения этой транзакции будет примерно равно времени выполнения транзакции, блокировавшей таблицу (как показано на рисунке 2.23б). Забегая вперед, скажем, что остальные уровни тоже не допускают грязное чтение, поэтому в дальнейшем для сценариев проверки на грязное чтение будет использоваться транзакция с одним оператором SELECT.

Сценарий 2. В этом сценарии всё точно так же, как и с уровнем READ UNCOMMITTED - другие транзакции смогли изменять данные таблицы, которую считывала транзакция с данным уровнем изоляции. Так что этот уровень тоже допускает неповторяющееся и фантомное чтения. (как показано на рисунке 2.23в) [6].

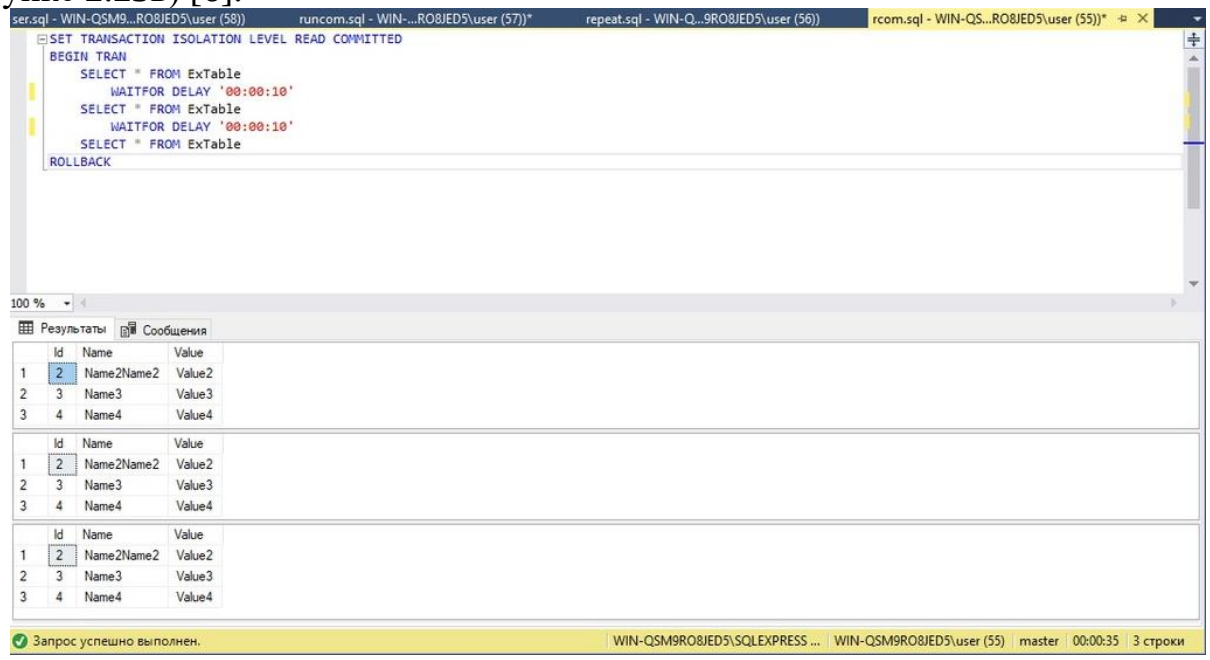

Рисунок. 2.23а – Получение конечных зафиксированных данных

Рисунок 2.23б – Время выполнения транзакции будет примерно равно времени выполнения транзакции, блокировавшей таблицу

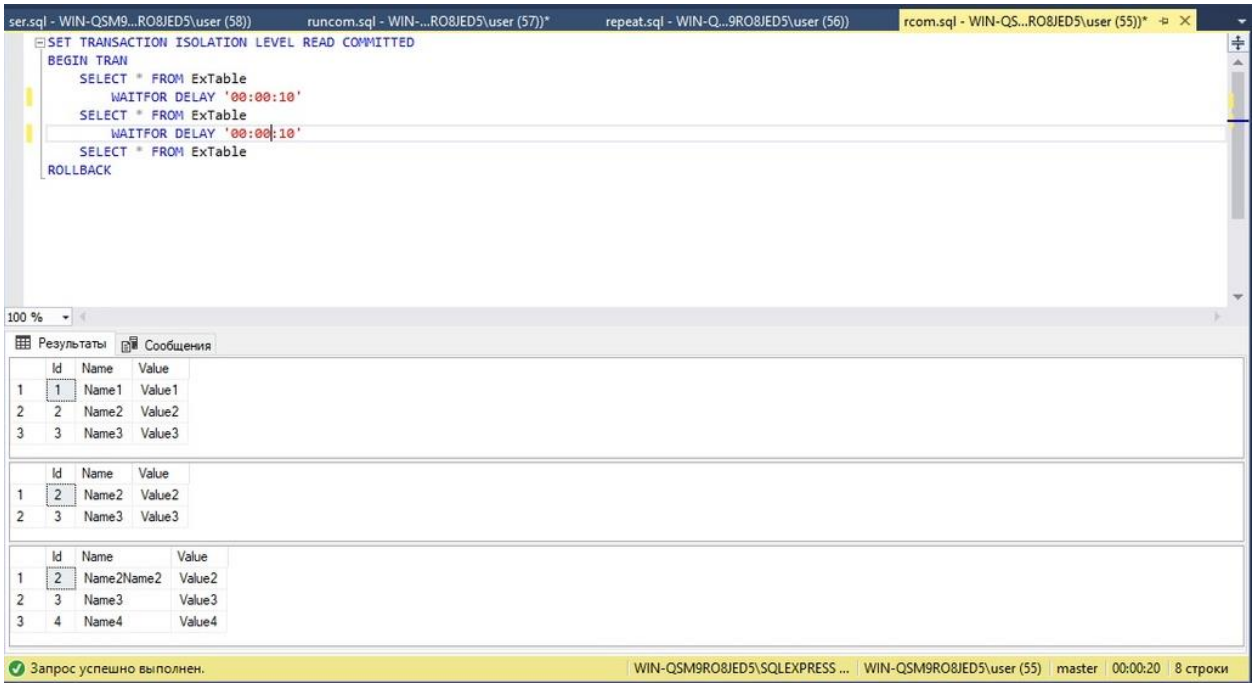

Рисунок 2.23в – Уровень допускает неповторяющееся и фантомное чтения

REPEATABLE READ: запрос в текущей транзакции не может читать данные, измененные другой транзакцией, которая еще не зафиксирована, что предотвращает грязное чтение. Кроме того, никакие другие транзакции не могут изменять данные, считываемые текущей транзакцией, до ее завершения, что исключает неповторяющиеся чтения. Однако, если другая транзакция вставляет новые строки, которые соответствуют условию поиска в текущей транзакции,

между текущей транзакцией, дважды обращающейся к одним и тем же данным, фантомные строки могут появиться во втором чтении.

Результаты сценариев для REPEATABLE READ:

Сценарий 1. Транзакция получила только конечные зафиксированные данные, так что грязное чтение здесь недопустимо (что показано на рисунке 2.24а).

Сценарий 2. В этом случае первым используем оператор INSERT, а следующим, допустим, DELETE. В итоге оператор INSERT выполниться, а следующий будет ожидать окончания транзакции REPEATABLE READ (что показано на рисунке 2.24б). Данный уровень позволяет добавить новые записи, но не изменять либо удалять их. Поэтому здесь пропадает возможность неповторяющихся чтений, но всё ещё остаются фантомные чтения [6].

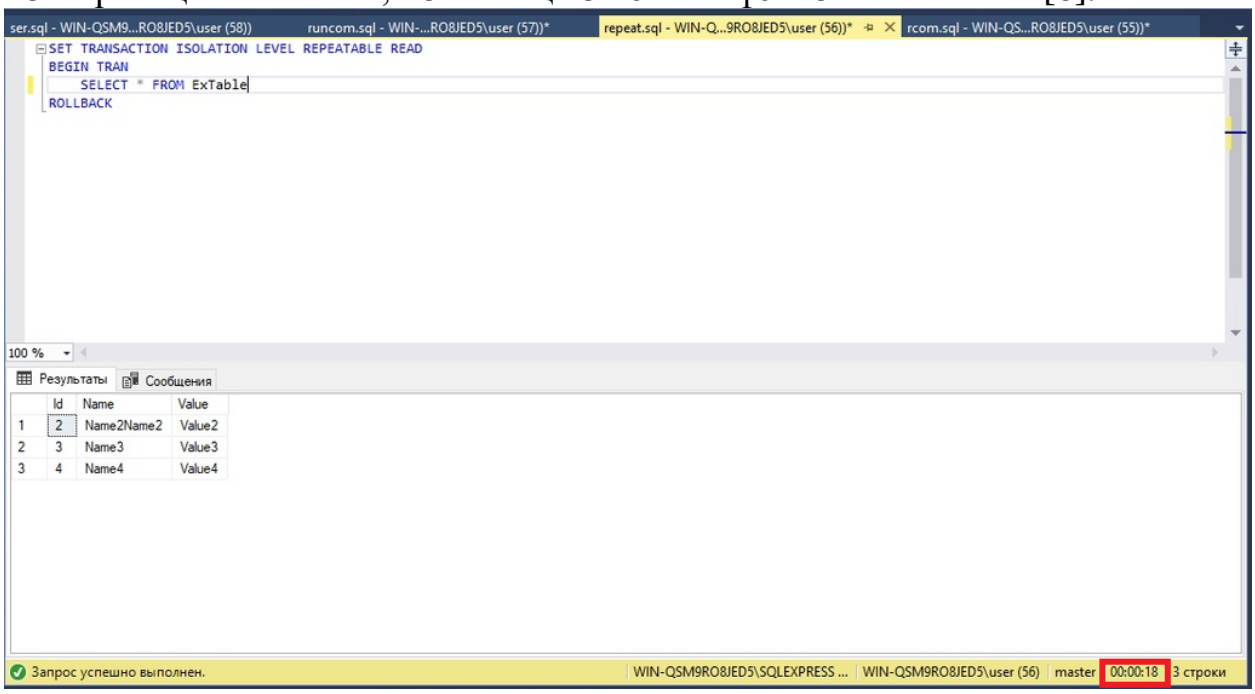

Рисунок 2.24а – Получение транзакцией только зафиксированных данных

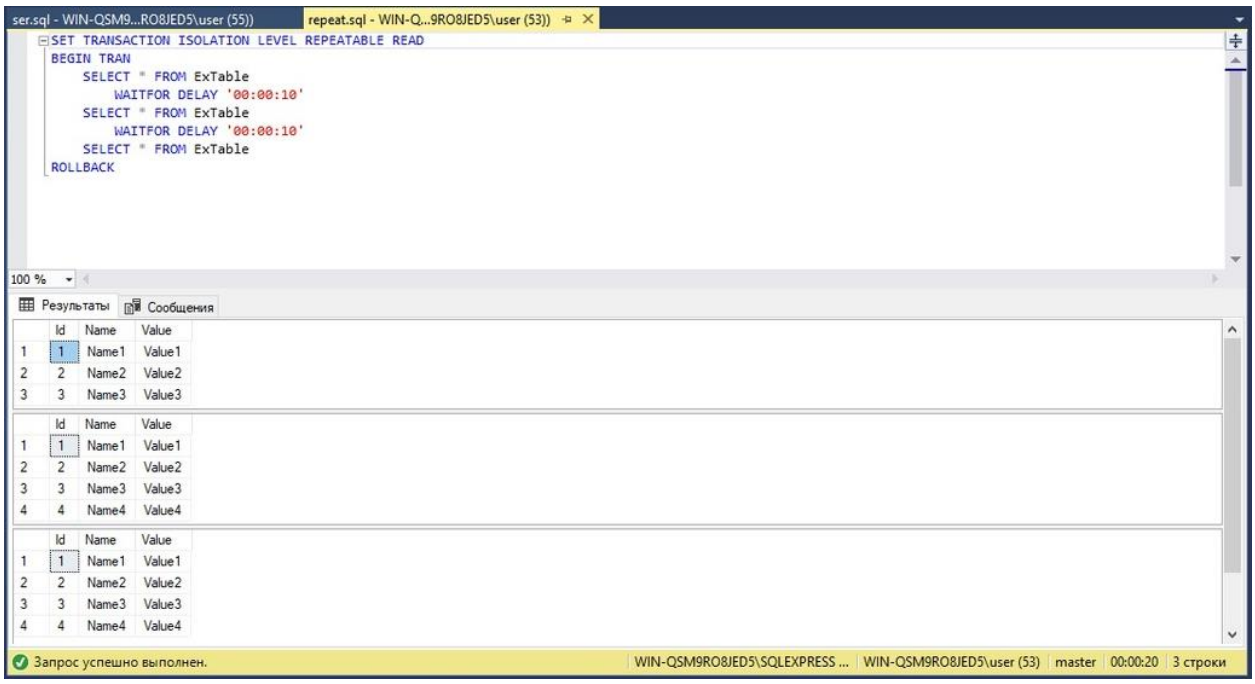

Рисунок 2.24б – Невозможность неповторяющихся чтений

SERIALIZABLE: запрос в текущей транзакции не может прочитать данные, измененные другой транзакцией, которая еще не зафиксирована. Никакая другая транзакция не может изменять данные, считываемые текущей транзакцией, до ее завершения, и никакая другая транзакция не может вставлять новые строки, которые соответствовали бы условию поиска в текущей транзакции, пока она не завершится. В результате уровень изоляции Serializable предотвращает грязное чтение, неповторяющееся чтение и фантомное чтение. Однако он может иметь наибольшее влияние на производительность по сравнению с другими уровнями изоляции [6].

Выполнение сценариев для этого уровня изоляции:

Сценарий 1**.** Транзакция получила только конечные зафиксированные данные, так что грязное чтение здесь недопустимо (что показано на рисунке 2.25а).

Сценарий 2**.** При использовании уровня SERIALIZABLE другим транзакциям запрещается изменять прочитанные первой данные, а также вставлять новые строчки в таблицу до конца транзакции. Поэтому любой из используемых в примере операторов будет ожидать окончания транзакции (что показано на рисунке 2.25б) [6].

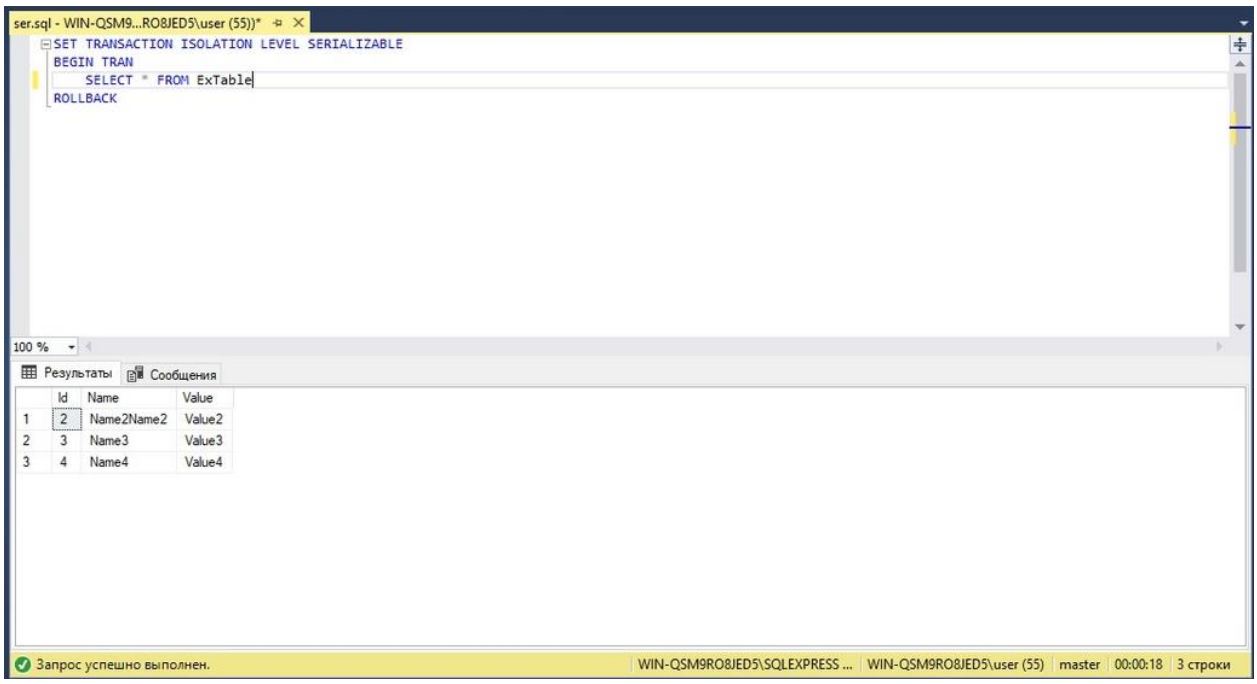

Рисунок 2.25а – Использование уровня SERIALIZABLE

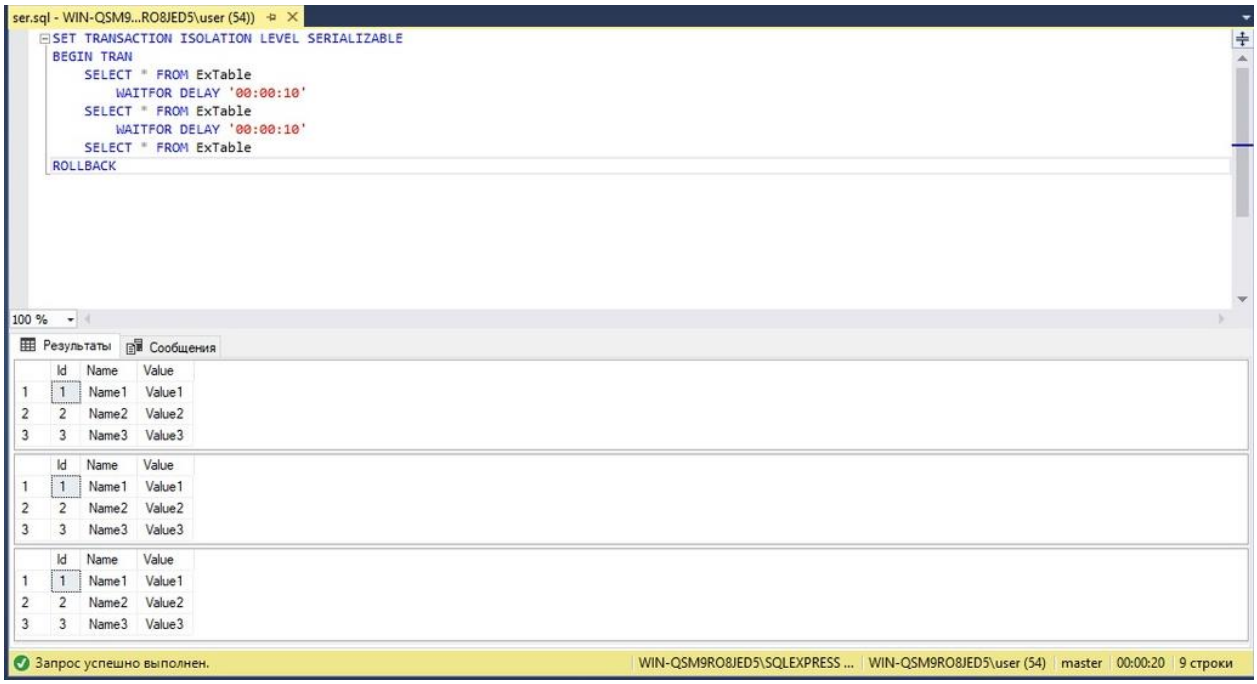

Рисунок 2.25б – использование уровня SERIALIZABLE

Кроме указанных выше уровней, есть ещё уровень оптимистической модели – SNAPSHOT. Основное отличие – этот уровень не использует блокировки. Вместо этого, в начале транзакции создается моментальный снимок данных, с которым транзакция работает до своего окончания [6].

В завершении раздела обобщим возможные проблемы одновременного доступа для каждого уровня в виде таблицы 2.1 (✓ - проблема допустима, ✗ - недопустима) [5]:

Таблица 2.1 – Допускаемые проблемы доступа уровней изоляции транзакций [6].

Уровень изоляции	Грязное чтение	Неповторяющеюся	Фантомное
READ UNCOMMITED	✓	✓	✓
READ COMMITED	✗	✓	✓
REPEATABLE READ	✗	✗	✓
SERIALIZABLE	✗	✗	✗
SNAPSHOT	✗	✗	✗

2.9 Блокировки

Механизм блокировки необходим для успешной обработки транзакций SQL Server. Он был разработан, чтобы позволить SQL Server беспрепятственно работать в многопользовательской среде. Блокировка - это способ, которым SQL Server управляет параллелизмом транзакций. По сути, блокировки - это структуры в памяти, у которых есть владельцы, типы и хэш ресурса, который он должен защищать. Размер блокировки как структуры в памяти составляет 96 байт.

Важно понимать, что блокировка предназначена для обеспечения целостности данных в БД, поскольку она заставляет каждую транзакцию проходить тест ACID.

Блокировка SQL Server является неотъемлемой частью требований к изоляции и служит для блокировки объектов, затронутых транзакцией. Пока объекты заблокированы, SQL Server не позволит другим транзакциям вносить какие-либо изменения в данные, хранящиеся в объектах, на которые наложена блокировка. После снятия блокировки путем фиксации изменений или отката изменений до исходного состояния другим транзакциям будет разрешено вносить необходимые изменения данных.

Работа блокировок SQL Server определяется с помощью режимов блокировки уровню в иерархии, к которым они применяются.

2.9.1 Режимы блокировки

Режимы блокировки определяют различные типы запретов, которые могут применяться к ресурсу, который необходимо заблокировать:

Монопольная блокировка

Общий доступ

Обновление

Намерение

Схемы

Массовое обновление

Монопольная блокировка (X - exclusive) - при использовании гарантирует, что страница или строка будут зарезервированы исключительно для транзакции, наложившей монопольную блокировку, пока транзакция удерживает блокировку.

Монопольная блокировка накладывается транзакцией, когда она хочет изменить данные страницы или строки, например, в случае применения операторов DML DELETE, INSERT или UPDATE. Монопольная блокировка может быть наложена на страницу или строку только в том случае, если на ресурс не наложена другая общая или монопольная блокировка. Из этого можно сделать вывод, что на страницу или строку может быть наложена только одна монопольная блокировка, а после наложения никакая другая блокировка не может быть наложена на заблокированные ресурсы.

Общая блокировка (S - shared) – данный тип блокировки при наложении резервирует страницу или строку, которые будут доступны только для чтения, что означает, что любая другая транзакция не сможет изменить заблокированную запись, пока блокировка активна. Однако общая блокировка может быть наложена несколькими транзакциями одновременно над одной и той же страницей или строкой, и таким образом несколько транзакций могут совместно использовать возможность чтения данных, поскольку сам процесс чтения никоим образом не повлияет на фактические данные страницы или строки. Кроме того, эта блокировка разрешает операции записи, но не допускаются изменения DDL (подробнее об операторах DDL будет рассказано в главе про триггеры) [6].

Обновление (U - update) – эта блокировка похожа на монопольную блокировку, но в некотором смысле более гибкая. Блокировка обновления может быть наложена на запись, которая уже имеет общую блокировку. В таком случае блокировка обновления накладывает другую общую блокировку на целевую строку. Как только транзакция, содержащая блокировку обновления, будет готова к изменению данных, блокировка обновления будет преобразована в монопольную блокировку. Важно понимать, что блокировка обновления асимметрична по отношению к общим блокировкам. В то время как блокировка обновления может быть наложена на запись с общей блокировкой, общая блокировка не может быть наложена на запись, которая уже имеет блокировку обновления.

Намерение (I - intent) – эта блокировка является средством, используемым транзакцией, чтобы сообщить другой транзакции о своем намерении получить блокировку. Цель такой блокировки - обеспечить правильное выполнение модификации данных, не допуская, чтобы другая транзакция установила блокировку следующего в иерархии объекта. На практике, когда транзакция хочет получить блокировку строки, она получает намеренную блокировку

таблицы, которая является объектом более высокой иерархии. Получив блокировку намерения, транзакция не позволит другим транзакциям получить монопольную блокировку для этой таблицы, т.к. в противном случае монопольная блокировка, наложенная какой-либо другой транзакцией, отменит блокировку строки.

Это важный тип блокировки с точки зрения производительности, поскольку ядро базы данных будет проверять блокировки намерения только на уровне таблицы, уточняя, возможно ли для транзакции получить блокировку безопасным способом в этой таблице, и, следовательно, данный тип блокировки устраняет необходимость проверять каждую блокировку строки или страницы в таблице, чтобы убедиться, что транзакция может получить блокировку для всей таблицы.

Существует три обычных блокировки намерения и три блокировки преобразования:

Обычные блокировки с намерением:

Монопольная блокировка с намерением (IX - intent exclusive) - когда получена монопольная блокировка с намерением (IX), это указывает SQL Server, что транзакция имеет намерение изменить некоторые из ресурсов более низкой иерархии, приобретая монопольные блокировки (X) индивидуально для этих ресурсов более низкой иерархии [6].

Блокировка с намерением совмещаемого доступа (IS - intent shared) - этот тип блокировки указывает SQL Server, что транзакция имеет намерение прочитать некоторые ресурсы более низкой иерархии, приобретая общие блокировки (S) индивидуально для этих ресурсов более низкого уровня иерархии.

Блокировка с намерением обновления (IU - intent update) - блокировка намеренного обновления может быть получена только на уровне страницы, и как только операция обновления выполняется, она преобразуется в монопольную блокировку с намерением(IX).

Блокировки преобразования:

Общий доступ с монопольной блокировкой намерения (SIX -shared intent exclusive) - при установке эта блокировка указывает, что транзакция намеревается прочитать все ресурсы в более низкой иерархии и, таким образом, получить общую блокировку для всех ресурсов, которые находятся ниже в иерархии, и, в свою очередь, изменить часть этих ресурсов, но не все. При этом он получит монопольную блокировку с намерением (IX) для тех ресурсов более низкой иерархии, которые должны быть изменены. На практике это означает, что как только транзакция получает данную блокировку таблицы, она приобретает монопольную блокировку с намерением (IX) на измененных страницах и монопольную блокировку (X) на измененных строках.

Только одна такая блокировка может быть получена для таблицы, и она будет блокировать другие транзакции от выполнения обновлений, но это не помешает другим транзакциям читать ресурсы более низкой иерархии.

Совмещаемая блокировка с намерением обновления (SIU - shared intent update) - это немного более конкретная блокировка, поскольку это комбинация блокировки общего (S) и обновления намерения (IU). Типичный пример этой блокировки - это когда транзакция использует запрос, выполняемый с ключевым словом PAGELOCK, а затем запрос на обновление. После того, как транзакция получит блокировку SIU для таблицы, запрос с подсказкой PAGELOCK получит общую (S) блокировку, в то время как запрос обновления получит блокировку намеренного обновления (IU).

Блокировка обновления с намерением монопольного доступа (UIX - update intent exclusive) - возникает, когда блокировка обновления (U) и блокировки намерения (IX) одновременно получены на ресурсах более низкой иерархии в таблице [6].

Блокировки схемы (Sch) - ядро базы данных SQL Server распознает два типа блокировок схемы: блокировка изменения схемы (Sch-M) и блокировка стабильности схемы (Sch-S).

Блокировка изменения схемы (Sch-M) будет получена при выполнении оператора DDL и предотвратит доступ к данным заблокированного объекта при изменении структуры объекта. SQL Server допускает единственную блокировку модификации схемы (Sch-M) для любого заблокированного объекта. Чтобы изменить таблицу, транзакция должна дождаться получения блокировки Sch-M на целевом объекте. После получения блокировки модификации схемы (Sch-M) транзакция может изменить объект, и после завершения модификации блокировка будет снята. Типичным примером блокировки Sch-M является перестроение индекса, а перестроение индекса - это процесс модификации таблицы. После выдачи идентификатора перестроения индекса для этой таблицы будет получена блокировка изменения схемы (Sch-M), которая будет снята только после завершения процесса перестроения индекса.

Блокировка стабильности схемы (Sch-S) будет получена во время компиляции и выполнения запроса, зависящего от схемы, и создания плана выполнения. Эта конкретная блокировка не будет блокировать другие транзакции для доступа к данным объекта и совместима со всеми режимами блокировки, кроме блокировки модификации схемы (Sch-M). По сути, блокировки стабильности схемы будут приобретаться любым DML запросом или запросом выбора, чтобы гарантировать целостность структуры таблицы (гарантировать, что таблица не изменяется во время выполнения запросов).

Блокировки массового обновления (BU - bulk update) - эта блокировка предназначена для использования в операциях массового импорта, когда они

запускаются с аргументом TABLOCK. Когда получена блокировка массового обновления, другие процессы не смогут получить доступ к таблице во время выполнения массовой загрузки. Однако блокировка массового обновления не препятствует параллельной обработке другой массовой загрузки.

Эскалация блокировки

Чтобы предотвратить ситуацию, когда при блокировке используется слишком много ресурсов, SQL Server имеет функцию эскалации (укрупнения) блокировки.

Эскалация блокировок позволяет исключить большую нагрузку на ресурсы памяти. Рассмотрим пример, в котором для выполнения операции удаления должна быть наложена блокировка на 30000 строк данных, каждая из которых имеет размер 500 байт. Без эскалации одна общая блокировка (S) будет наложена на базу данных, одна монопольная с намереннием (IX) на таблицу, 1875 монопольных с намерением (IX) на страницах (страница 8 КБ содержит 16 строк по 500 байтов, что составляет 1875 страниц, содержащих 30000 строк) и 30000 монопольных блокировок (X) на самих строках. Поскольку размер каждой блокировки составляет 96 байт, 31877 блокировок потребуют около 3 МБ памяти для одной операции удаления. Параллельное выполнение большого количества операций может потребовать значительных ресурсов только для того, чтобы менеджер блокировок мог выполнять операцию без задержек [6].

Чтобы предотвратить такую ситуацию, SQL Server использует эскалацию блокировок. Это означает, что в ситуации, когда на одном уровне установлено более 5000 блокировок, SQL Server превратит эти блокировки в одну блокировку на уровне таблицы. По умолчанию SQL Server всегда будет напрямую переходить на уровень таблицы, минуя переход на уровень страницы. Вместо получения блокировки множества строк и страниц, SQL Server перейдет к монопольной блокировке (X) на уровне таблицы.

Хотя это снизит потребность в ресурсах, монопольные блокировки (X) в таблице означают, что никакая другая транзакция не сможет получить доступ к заблокированной таблице, и все запросы, пытающиеся получить доступ к этой таблице, будут заблокированы. Следовательно, это снизит нагрузку на систему, но увеличит конкуренцию на доступ к данным.

Чтобы обеспечить контроль над эскалацией, начиная с SQL Server 2008 R2, параметр LOCK_ESCALATION вводится как часть оператора ALTER TABLE.

USE имя_базы_данных

GO

ALTER TABLE имя_таблицы

SET (LOCK_ESCALATION = <TABLE | AUTO | DISABLE> - один из этих параметров)

GO

Следующие параметры позволяют задать режим процесса эскалации блокировки:

TABLE - это параметр по умолчанию для любой созданной таблицы, так как по умолчанию SQL Server всегда выполняет эскалацию блокировки до уровня таблицы, который также включает секционированные таблицы.

AUTO - этот параметр позволяет эскалацию блокировки до уровня раздела, когда таблица разбита на разделы. Когда 5000 блокировок получены в одном разделе, при эскалации блокировки будет получена монопольная блокировка (X) на этом разделе, в то время как таблица получит монопольную блокировку с намерением (IX). В случае, если эта таблица не разделена на разделы, при эскалации блокировки будет установлена блокировка на уровне таблицы (как при параметре «TABLE»).

Хотя это выглядит очень полезным вариантом, его следует использовать очень осторожно, поскольку он может легко вызвать взаимоблокировки. В ситуации, когда у нас есть две транзакции в двух разделах, где получена монопольная блокировка (X), и транзакция пытается получить доступ к данным из раздела, используемого другой транзакцией, возникнет взаимная блокировка. Таким образом, очень важно тщательно контролировать шаблон доступа к данным, если этот параметр включен, и поэтому этот параметр не является настройкой по умолчанию в SQL Server.

DISABLE - этот параметр полностью отключает эскалацию блокировки для таблицы. Опять же, этот параметр следует использовать осторожно, чтобы избежать принудительного использования диспетчером блокировок SQL Server чрезмерного объема памяти.

Как видно, эскалация блокировок может стать проблемой для администраторов баз данных. Если работа приложения требует одновременного удаления или обновления более 5000 строк, решение, позволяющее избежать эскалации блокировок и связанных с этим эффектов, состоит в разделении одной транзакции на две или более транзакции, каждая из которых будет обрабатывать менее 5000 строк [6].

Получение информации об активных блокировках SQL Server

SQL Server имеет представление управления динамикой (DMV - Dynamics Management View) sys.dm_tran_locks, которое возвращает информацию о ресурсах диспетчера блокировок, которые используются в настоящее время, что означает, что он отображает все действующие блокировки, полученные транзакциями.

Наиболее важными столбцами, используемыми для идентификации блокировки, являются resource_type, request_mode и resource_description. При необходимости во время устранения неполадок можно добавить больше столбцов в качестве дополнительных ресурсов для информации.

Пример запроса

SELECT resource_type, request_mode, resource_description.

FROM sys.dm_tran_locks

WHERE resource_type <> «БАЗА ДАННЫХ»

Фильтрация WHERE в этом запросе используется, чтобы исключить из результатов те общие блокировки, которые наложены на базу данных, поскольку они всегда присутствуют на уровне базы данных.

Описание запрошенных столбцов:

resource_type - отображает ресурс базы данных, в котором устанавливаются блокировки. Столбец может отображать одно из следующих значений: ALLOCATION_UNIT, APPLICATION, DATABASE, EXTENT, FILE, HOBT, METADATA, OBJECT, PAGE, KEY, RID.

request_mode - отображает режим блокировки, установленный на ресурсе

resource_description - отображает краткое описание ресурса и заполняется не для всех режимов блокировки. Чаще всего столбец содержит идентификатор строки, страницы, объекта, файла и т. п [6].

ЗАКЛЮЧЕНИЕ

В ходе модернизации лабораторный практикум был разбит на две лабораторные работы, для каждой работы составлены контрольные вопросы, практические задания, увеличен объём теоретической части в каждой лабораторной работе, а также количество практических заданий. Был рассмотрен ряд вопросов: Контроль целостности и организация доступа через представления, сохранение процедур, функции и триггеры, SQL инъекции, резервное копирование данных, транзакции была описана теория баз данных, теория транзакций, принцип создания и работы транзакции, свойства транзакций, а также способы завершения, восстановление после сбоя и уровни изолированности пользователя. Были представлены примеры основных команд управления транзакциями, таких как COMMIT, ROLLBACK, SAVEPOINT, RELEASE SAVEPOINT, SET TRANSACTION, а также способы их блокировки (UPDLOCK, TABLOCK, ROWLOCK, PAGLOCK, NOLOCK, HOLDLOCK, XLOCK, READPAST) а также LOCK_TIMEOUT. Из проведенной работы можно сделать вывод, что во многих случаях, необходимо проведение группы операций по изменению данных таким образом, чтобы эта группа обладала свойством атомарности (либо вся целиком выполняется, либо вся целиком не выполняется). Пользователи в основном должны указывать только начало и конец транзакции, используя команды SQL или API (прикладного интерфейса программирования).

Хотелось бы отметить, что одной из наиболее оптимальных программ для реализации транзакций является SQL Server. Так как транзакции блокируют записи, рекомендуется делать их максимально быстрыми, и не рекомендуется использовать вложенные транзакций. Транзакции должны выполняться максимально быстро. Долгие транзакции увеличивают вероятность, что пользователи не получат доступ к заблокированным данным.

Выстраивая систему защиты информации, а также комбинируя различные способы защиты информации, можно добиться хорошего уровня защиты данных. Методы зашиты, рассмотренные в данной дипломной работе, а также в разработанном лабораторном практикуме позволят снизить риск

несанкционированного доступа к базе данных и позволят решить проблему безопасности данных.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Habr resource for IT professionals [Электронный ресурс] Режим доступа: https://habr.com/ru/post/446662/

2. Microsoft docs about SQL [Электронный ресурс] Режим доступа: https://docs.microsoft.com/ru-ru/sql/relational-databases/views/create-views?view=sql-server-ver15

3. Защита баз данных [Электронный ресурс] Режим доступа: https://www.azone-it.ru/zashchita-baz-dannyh

4. SQL – правила целостности данных [Электронный ресурс] Режим доступа: http://www.codenet.ru/db/sql/003.php

5. Redgate Software – Compliant Database DevOps Solutions and for the finance, healthcare and technology sectors [Электронный ресурс]. Режим доступа: https://www.red-gate.com/simple-talk/sql/t-sql-programming/questions-about-t-sql-transaction-isolation-levels-you-were-too-shy-to-ask/

6. Образовательный портал факультета радиофизики и компьютерных технологий БГУ [Электронный ресурс] Режим доступа: https://edurfe.bsu.by/mod/assign/view.php?id=5734.

7. SQL Server Tips, Articles and Training [Электронный ресурс]. Режим доступа: http://mssqltips.com/sqlservertip/5909/sql-server-trigger-example/

8. Cyber Security Leader | Imperva, Inc. [Электронный ресурс]. Режим доступа: https://www.imperva.com/learn/application-security/sql-injection-sqli/

9. TablePlus | Modern, Native Tool for Database Management [Электронный ресурс]. Режим доступа: https://tableplus.com/blog/2018/08/best-practices-to-prevent-sql-injection-attacks.html

10. SQL exercises and blogs [Электронный ресурс] Режим доступа: https://sql-ex.ru/blogs/?/5_sposobov_sdelat_rezervnye_kopii_v_SQL_Server.html

ПРИЛОЖЕНИЕ А

Методические указания для лабораторной работы №1

Лабораторная работа №1

«Контроль целостности и организация доступа через представления, хранимые процедуры, триггеры»

Цель: изучить контроль целостности в БД, организацию доступа через представления, изучение, а также использование на практике хранимых процедур, триггеров.

В данных примерах для работы с базой данных использовалась утилита **Microsoft SQL Management Studio**. Модель базы данных TaxiService представлена на рисунке ниже:

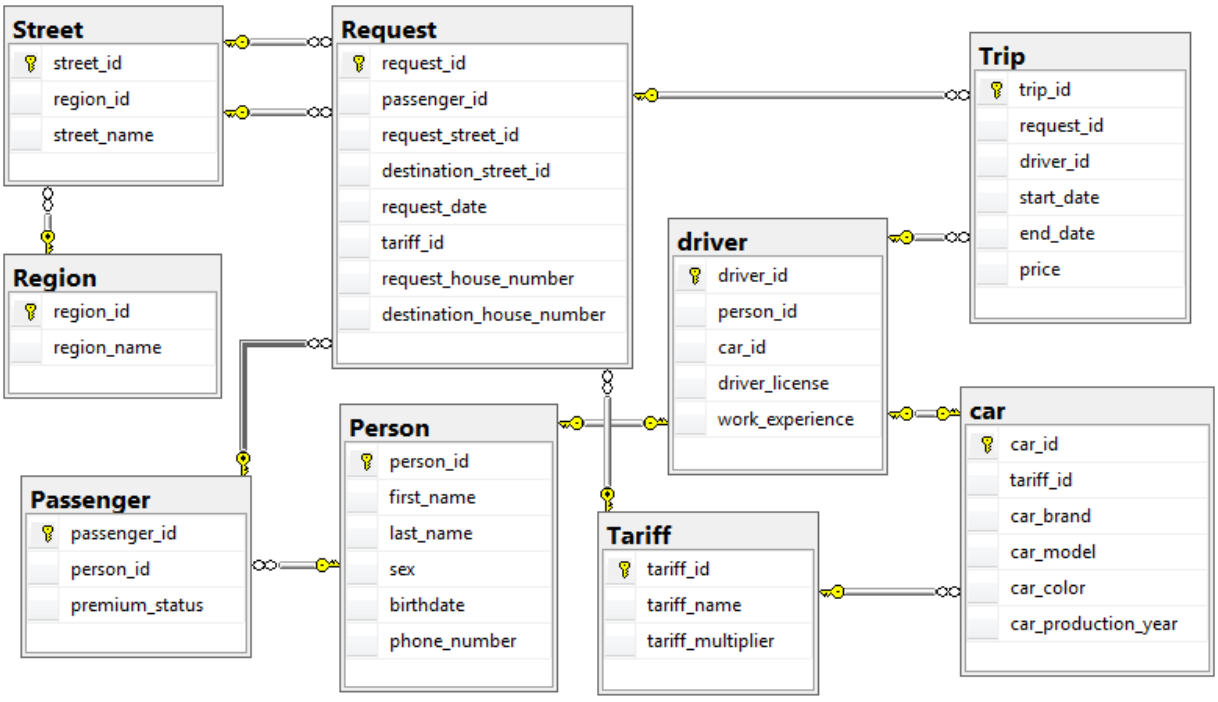

Рисунок А1 – модель данных базы данных

Скрипт, сформированный для данной базы данных:

```
CREATE TABLE [dbo].[car](
	[car_id] [int] IDENTITY(1,1) NOT NULL,
	[tariff_id] [int] NOT NULL,
	[car_brand] [nvarchar](20) NOT NULL,
	[car_model] [nvarchar](20) NOT NULL,
	[car_color] [nvarchar](20) NOT NULL,
	[car_production_year] [int] NULL,
 CONSTRAINT [PK_car] PRIMARY KEY CLUSTERED ([car_id] ASC) ON [PRIMARY]
GO

CREATE TABLE [dbo].[driver](
```

```
	[driver_id] [int] IDENTITY(1,1) NOT NULL,
	[person_id] [int] NOT NULL,
	[car_id] [int] NULL,
	[driver_license] [nvarchar](10) NOT NULL,
	[work_experience] [smallint] NULL,
 CONSTRAINT [PK_driver] PRIMARY KEY CLUSTERED ([driver_id] ASC) ON [PRIMARY],
 CONSTRAINT [IX_driver] UNIQUE NONCLUSTERED ([car_id] ASC) ON [PRIMARY],
 CONSTRAINT [IX_driver_1] UNIQUE NONCLUSTERED ([person_id] ASC) ON [PRIMARY]
GO

CREATE TABLE [dbo].[Passenger](
	[passenger_id] [int] IDENTITY(1,1) NOT NULL,
	[person_id] [int] NOT NULL,
	[premium_status] [bit] NOT NULL,
 CONSTRAINT [PK_Passenger] PRIMARY KEY CLUSTERED ([passenger_id] ASC) ON [PRIMARY]
GO

CREATE TABLE [dbo].[Person](
	[person_id] [int] IDENTITY(1,1) NOT NULL,
	[first_name] [nvarchar](20) NOT NULL,
	[last_name] [nvarchar](20) NOT NULL,
	[sex] [nvarchar](7) NULL,
	[birthdate] [datetime] NOT NULL,
	[phone_number] [nvarchar](13) NOT NULL,
 CONSTRAINT [PK_Person] PRIMARY KEY CLUSTERED ([person_id] ASC) ON [PRIMARY]
GO

CREATE TABLE [dbo].[Region](
	[region_id] [int] IDENTITY(1,1) NOT NULL,
	[region_name] [nvarchar](20) NOT NULL,
 CONSTRAINT [PK_Region] PRIMARY KEY CLUSTERED ([region_id] ASC) ON [PRIMARY]
GO

CREATE TABLE [dbo].[Request](
	[request_id] [int] IDENTITY(1,1) NOT NULL,
	[passenger_id] [int] NOT NULL,
	[request_street_id] [int] NULL,
	[destination_street_id] [int] NULL,
	[request_date] [datetime] NOT NULL,
	[tariff_id] [int] NOT NULL,
	[request_house_number] [smallint] NOT NULL,
	[destination_house_number] [smallint] NOT NULL,
 CONSTRAINT [PK_Request] PRIMARY KEY CLUSTERED ([request_id] ASC) ON [PRIMARY]
GO

CREATE TABLE [dbo].[Street](
	[street_id] [int] IDENTITY(1,1) NOT NULL,
	[region_id] [int] NOT NULL,
	[street_name] [nvarchar](20) NOT NULL,
 CONSTRAINT [PK_Street] PRIMARY KEY CLUSTERED ([street_id] ASC) ON [PRIMARY]
GO

CREATE TABLE [dbo].[Tariff](
	[tariff_id] [int] IDENTITY(1,1) NOT NULL,
	[tariff_name] [nvarchar](12) NOT NULL,
	[tariff_multiplier] [decimal](3, 2) NULL,
 CONSTRAINT [PK_Tariff] PRIMARY KEY CLUSTERED ([tariff_id] ASC) ON [PRIMARY]
GO
```

```
CREATE TABLE [dbo].[Trip](
	[request_id] [int] NOT NULL,
	[driver_id] [int] NULL,
	[start_date] [datetime] NULL,
	[end_date] [datetime] NULL,
	[price] [money] NOT NULL,
 CONSTRAINT [PK_Trip] PRIMARY KEY CLUSTERED ([request_id] ASC) ON [PRIMARY]
GO

ALTER TABLE [dbo].[car]  WITH CHECK ADD  CONSTRAINT [FK_car_Tariff] FOREIGN KEY([tariff_id])
REFERENCES [dbo].[Tariff] ([tariff_id])
ON UPDATE NO ACTION
ON DELETE NO ACTION
GO
ALTER TABLE [dbo].[car] CHECK CONSTRAINT [FK_car_Tariff]
GO
ALTER TABLE [dbo].[driver]  WITH CHECK ADD  CONSTRAINT [FK_driver_car] FOREIGN KEY([car_id])
REFERENCES [dbo].[car] ([car_id])
ON UPDATE NO ACTION
ON DELETE NO ACTION
GO
ALTER TABLE [dbo].[driver] CHECK CONSTRAINT [FK_driver_car]
GO
ALTER TABLE [dbo].[driver]  WITH CHECK ADD  CONSTRAINT [FK_driver_Person] FOREIGN KEY([person_id])
REFERENCES [dbo].[Person] ([person_id])
ON UPDATE NO ACTION
ON DELETE NO ACTION
GO
ALTER TABLE [dbo].[driver] CHECK CONSTRAINT [FK_driver_Person]
GO
ALTER TABLE [dbo].[Passenger]  WITH CHECK ADD  CONSTRAINT [FK_Passenger_Person] FOREIGN
KEY([person_id])
REFERENCES [dbo].[Person] ([person_id])
ON UPDATE NO ACTION
ON DELETE NO ACTION
GO
ALTER TABLE [dbo].[Passenger] CHECK CONSTRAINT [FK_Passenger_Person]
GO
ALTER TABLE [dbo].[Request]  WITH CHECK ADD  CONSTRAINT [FK_Request_Passenger] FOREIGN
KEY([passenger_id])
REFERENCES [dbo].[Passenger] ([passenger_id])
ON UPDATE NO ACTION
ON DELETE NO ACTION
GO
ALTER TABLE [dbo].[Request] CHECK CONSTRAINT [FK_Request_Passenger]
GO
ALTER TABLE [dbo].[Request]  WITH CHECK ADD  CONSTRAINT [FK_Request_Street] FOREIGN
KEY([request_street_id])
REFERENCES [dbo].[Street] ([street_id])
ON UPDATE NO ACTION
ON DELETE NO ACTION
GO
ALTER TABLE [dbo].[Request] CHECK CONSTRAINT [FK_Request_Street]
GO
ALTER TABLE [dbo].[Request]  WITH CHECK ADD  CONSTRAINT [FK_Request_Street1] FOREIGN
KEY([destination_street_id])
REFERENCES [dbo].[Street] ([street_id])
```

```
ON UPDATE NO ACTION
ON DELETE NO ACTION
GO
ALTER TABLE [dbo].[Request] CHECK CONSTRAINT [FK_Request_Street1]
GO
ALTER TABLE [dbo].[Request]  WITH CHECK ADD  CONSTRAINT [FK_Request_Tariff] FOREIGN KEY([tariff_id])
REFERENCES [dbo].[Tariff] ([tariff_id])
ON UPDATE NO ACTION
ON DELETE NO ACTION
GO
ALTER TABLE [dbo].[Request] CHECK CONSTRAINT [FK_Request_Tariff]
GO
ALTER TABLE [dbo].[Street]  WITH CHECK ADD  CONSTRAINT [FK_Street_Region] FOREIGN KEY([region_id])
REFERENCES [dbo].[Region] ([region_id])
ON UPDATE NO ACTION
ON DELETE NO ACTION
GO
ALTER TABLE [dbo].[Street] CHECK CONSTRAINT [FK_Street_Region]
GO
ALTER TABLE [dbo].[Trip]  WITH CHECK ADD  CONSTRAINT [FK_Trip_driver] FOREIGN KEY([driver_id])
REFERENCES [dbo].[driver] ([driver_id])
ON UPDATE NO ACTION
ON DELETE NO ACTION
GO
ALTER TABLE [dbo].[Trip] CHECK CONSTRAINT [FK_Trip_driver]
GO
ALTER TABLE [dbo].[Trip]  WITH CHECK ADD  CONSTRAINT [FK_Trip_Request] FOREIGN KEY([request_id])
REFERENCES [dbo].[Request] ([request_id])
ON UPDATE NO ACTION
ON DELETE NO ACTION
GO
ALTER TABLE [dbo].[Trip] CHECK CONSTRAINT [FK_Trip_Request]
GO
```

Контроль целостности

Главная особенность SQL-технологий наличие у сервера СУБД специальных средств контроля целостности данных, не зависящих от клиентских программ и привязанных непосредственно к таблицам. Т.е. принципиально не важно, каким образом осуществляется доступ к базе данных: через SQL-консоль, через ODBC-драйвера из приложения Windows, через WWW-connector из Internet-браузера или через DBI-интерфейс Perl. В любом из этих случаев, за контролем целостности данных следит сервер, и при нарушении правил целостности данных сервер известит клиента об ошибке.

К структурам контроля целостности данных относятся ограничители (constraint), которые привязаны к столбцам и триггеры (trigger), которые могут быть привязаны как к столбцам, так и к строкам в таблице.

Ограничители - это элементарные проверки или условия, которые выполняются для операций вставки и модификации значения столбца. Если данная проверка не проходит или условие не выполняется, то вставка или модификация отменяется, а в программу клиента передается ошибка.

SQL-серверы, как правило, поддерживают следующие ограничители.

NOT NULL - проверка на непустое значение. NULL - специальное понятие в СУБД, которое означает «пусто». «Пусто» и «0(ноль)» не равны друг другу!

UNIQUE - проверка на уникальность. Вставляемое значение должно быть уникально для данного столбца по всей таблице. Может содержать пустые значения.

PRIMARY KEY - первичный ключ. Значение в столбце считается первичным ключом, если оно непустое и уникально в пределах столбца данной таблицы. Первичный ключ может быть составным и представлять собой комбинацию столбцов. Тогда чтобы считаться первичным ключом, каждое из группы значений не должно быть пустыми и формируемые строки значений первичного ключа должны быть уникальны в пределах таблицы. Первичный ключ - основа для построения индексов по таблице.

SQL-технология позволяет на уровне столбца задавать домены значений, т.е. строго определенные наборы или диапазоны значений, для помещаемых в столбец данных. В частности, можно реализовывать ограничения ссылочной целостности (referential integrity constraint) и проверки фиксированного условия. Ограничение ссылочной целостности не позволяет значениям из столбца одной таблицы принимать значения кроме как из присутствующих в столбце другой таблицы. Это делается при помощи ограничителей FOREIGN KEY (внешний ключ) и REFERENCES (указатель ссылки). Таблица, содержащая FOREIGN KEY, считается родительской таблицей. Таблица, содержащая REFERENCES, считается дочерней таблицей. Внешний ключ и указатель ссылки могут находиться в одной таблице, т.е. родительская таблица одновременно является дочерней. FOREIGN KEY - внешний ключ. Назначает столбец или комбинацию столбцов в текущей (родительской) таблице в качестве внешнего ключа для ссылки из других таблиц.

REFERENCES - указатель ссылки (или родительский ключ). Указывает на столбец (комбинацию столбцов) в родительской таблице, ограничивающую значения в текущей (дочерней) таблице.

Для использования ограничений ссылочной целостности должны выполняться некоторые условия. В частности, родительская и дочерняя таблицы должны находиться в пределах одного аппаратного сервера базы данных, они не могут находиться на различных узлах распределенной базы данных. Столбцы, участвующие в отношении ограничения ссылочной целостности обязаны иметь один и тот же тип данных.

Ограничения ссылочной целостности используются при каскадном удалении, т.е. при удалении записи в родительской таблице удаляются все записи с указанным ключом из дочерних таблиц, и наоборот при запрете удаления/модификации, т.е. при наличии зависимых записей в дочерних

таблицах, значение ключа записи в родительской таблице нельзя удалить или модифицировать.

CHECK - проверка фиксированного условия. В данном ограничителе явно указывается условие, которое должно выполняться для вставляемого или модифицируемого значения в столбце. Например, check (user in 'ALEX','JUSTAS') - в столбце user могут содержаться только значения 'ALEX' и 'JUSTAS', попытка вставки значения 'SHTIRLITZ' будет интерпретирована как ошибочная, check (user_salary between 1000 and 5000) - столбец user_salary может принимать целочисленные значения в диапазоне от 1000 до 5000 и т.д. При формировании условий с некоторыми ограничениями могут использоваться функции, например, check (user = upper(user)), в данном случае имя пользователя должно вводиться только в верхнем регистре. Есть и ограничения, например, CHECK не может содержать подзапросы (SELECT).

Обычно ограничители задаются при создании таблиц. Но в дальнейшем их можно изменять, удалять или временно запрещать при помощи соответствующих команд СУБД.

Обработка данных в многопользовательской СУБД.

Основное требование к многопользовательским СУБД – обеспечение непротиворечивости данных в системе, при сохранении максимальной производительности и конкуренции в доступе к данным для пользователей.

Конкуренция в доступе к данным означает, что каждый из пользователей независим от остальных пользователей в потребности обработки данных. Система, во избежание порчи данных, самостоятельно устанавливает очередность работы с данными для пользователей. В случае необходимости пользователи могут ожидать своей очереди для работы с данными. Одной из главных целей многопользовательской СУБД является максимальное уменьшение этого времени ожидания до такой степени, чтобы оно (в идеале) стало незаметным для пользователя.

Кроме того, сервер СУБД должен предотвращать взаимно разрушающие манипуляции с данными нескольких пользователей при их одновременной работе. Например, если система не предусматривает такую возможность, то менеджеры принимающие заказы от клиентов на поставку товара, и выполняющие их резервирование на складе, могут зарезервировать товара больше чем фактически имеется в наличии. В этом случае обеспечен неприятный разговор с клиентом, заказ которого будет впоследствии отменен.

Более неприятная ситуация возможна в банке: если одновременно исполняется несколько клиентских платежных поручений с одного счета, то при неконтролируемом списании с клиентского счета возможен отрицательный остаток, что недопустимо.

Контроль нужен также в системах резервирования билетов на транспорте, чтобы билет на одно и то же место не был продан разными кассирами разным пассажирам.

Несмотря на различия в реализации, серверы СУБД используют общие способы управления данными и доступом к ним.

Атомарность SQL-выражений при работе с данными.

Под атомарностью выражения понимается неизменность (фиксация во времени) набора данных, с которыми это выражение работает на всем протяжении своего исполнения. Т.е. если мы выполняем оператор UPDATE над определенной таблицей, то состояние таблицы на момент начала выполнения операции фиксируется во времени и не изменяется до конца выполнения оператора. Этот набор данных для текущего выполняемого выражения не может быть изменен другим пользователем или даже другой сессией этого же пользователя, которая пытается выполнить операцию модификации этих же данных в этой же таблице.

Распараллеливание операций.

Типовые операции с таблицей в базе данных состоят из многих однотипных операций, например, оператор UPDATE, который модифицирует 5000 строк в таблице, по своей сути состоит из 5000 операций, каждая из которых может быть выполнена независимо. В связи с этим такие операторы очень хорошо распараллеливаются при использовании многопроцессорных систем. Это позволяет выровнять нагрузку в системе между разными процессорами, при том условии что СУБД умеет работать в многопроцессорной конфигурации, и уменьшить время ответа системы.

Обеспечение максимальной производительности.

С целью сокращения времени различных пользователей на манипуляции с данными используется ряд следующих методов. Их работа находится на уровне, скрытом даже от программиста СУБД, но о них стоит упомянуть т.к. они иллюстрируют серьезные различия с xBase-технологией.

Строго говоря, эта информация справедлива лишь в отношении Oracle, но другие СУБД используют подобные принципы.

Процессы, выполняющие чтение блоков данных, никогда не ожидают процессов, выполняющих запись тех же блоков данных.

Процессы, выполняющие запись блоков данных, при отсутствии явных блокировок со стороны пользователя, не ожидают процессов, выполняющих чтение тех же блоков данных.

Процессы, выполняющие запись блоков данных, ожидают другие процессы, выполняющие запись, только в случае если они пытаются выполнить запись данных в одни и те же блоки данных.

Данные приемы позволяют существенно уменьшить время ожидания ответа системы и увеличить ее производительность.

Представление – это виртуальная таблица, содержимое которой определяется запросом. Как и таблица, представление состоит из ряда именованных столбцов и строк данных. Пока представление не будет проиндексировано, оно не существует в базе данных как хранимая совокупность значений. Строки и столбцы данных извлекаются из таблиц, указанных в определяющем представление запросе и динамически создаваемых при обращениях к представлению.

Представление выполняет функцию фильтра базовых таблиц, на которые оно ссылается. Определяющий представление запрос может быть инициирован в одной или нескольких таблицах, или в других представлениях текущей или других баз данных. Кроме того, для определения представлений с данными из нескольких разнородных источников можно использовать распределенные запросы. Это полезно, например, если нужно объединить структурированные подобным образом данные, относящиеся к разным серверам, каждый из которых хранит данные конкретного отдела организации.

Представления обычно используются для направления, упрощения и настройки восприятия каждым пользователем информации базы данных. Представления могут использоваться как механизмы безопасности, давая возможность пользователям обращаться к данным через представления, но не предоставляя им разрешений на непосредственный доступ к базовым таблицам, лежащим в основе представлений. Представления могут использоваться для обеспечения интерфейса обратной совместимости, моделирующего таблицу, которая существует, но схема которой изменилась. Представления могут также использоваться при прямом и обратном копировании данных в SQL Server для повышения производительности и секционирования данных.

Типы представлений

Кроме основных определяемых пользователем представлений, выполняющих стандартные роли, в SQL Server предусмотрены следующие типы представлений, которые соответствуют специальным назначениям в базе данных.

Индексированные представления

Индексированным называется материализованное представление. Это означает, что определение представления вычисляется, а результирующие данные хранятся точно так же, как и таблица. Индексировать представление можно, создав для него уникальный кластеризованный индекс. Индексированные представления могут существенно повысить производительность некоторых типов запросов. Индексированные представления эффективнее всего

использовать в запросах, группирующих множество строк. Они не очень хорошо подходят для часто обновляющихся базовых наборов данных.

Секционированные представления

Секционированным называется представление, соединяющее горизонтально секционированные данные набора таблиц-элементов, находящихся на одном или нескольких серверах. При этом данные выглядят так, как будто находятся в одной таблице. Представление, соединяющее таблицы-элементы одного экземпляра SQL Server, называется локальным секционированным представлением.

Системные представления

Системные представления предоставляют доступ к метаданным каталога. Системные представления можно использовать для получения сведений об экземпляре SQL Server или объектах, определенных в экземпляре. Например, получить сведения об определяемых пользователем базах данных, доступных в экземпляре, можно через представление каталога sys.databases.

Создание представлений

Представления можно создать в SQL Server с помощью SQL Server Management Studio или Transact-SQL. Представление можно использовать в следующих целях.

Для направления, упрощения и настройки восприятия информации в базе данных каждым пользователем.

В качестве механизма безопасности, позволяющего пользователям обращаться к данным через представления, но не предоставляя им разрешений на непосредственный доступ к базовым таблицам.

Для предоставления интерфейса обратной совместимости, моделирующего таблицу, схема которой изменилась.

Ограничения

Представление может быть создано только в текущей базе данных.

Представление может включать не более 1 024 столбцов.

Безопасность

Для выполнения этой инструкции требуется разрешение CREATE VIEW в отношении базы данных и разрешение ALTER в отношении схемы, в которой создается представление.

Использование среды SQL Server Management Studio

Создание представления с использованием конструктора запросов и представлений

В обозревателе объектов разверните базу данных, в которой необходимо создать новое представление.

Щелкните правой кнопкой папку Представления и выберите Создать представление...

В диалоговом окне *Добавить таблицу* выберите один или несколько элементов, которые необходимо включить в новое представление, на одной из следующих вкладок: «Таблицы», «Представления», «Функции» и «Синонимы».

Щелкните *Добавить*, а затем выберите Закрыть.

На Панели диаграмм выберите столбцы или другие элементы для включения в новое представление.

На Панели критериев выберите дополнительные условия сортировки или фильтрации для столбцов.

В меню Файл выберите пункт Сохранитьview name.

В диалоговом окне Выбор имени введите имя нового представления и щелкните ОК.

Использование Transact-SQL

Создание представления

В обозревателе объектов подключитесь к экземпляру компонента Компонент Database Engine.

На стандартной панели выберите пункт Создать запрос.

Скопируйте следующий пример в окно запроса и нажмите кнопку Выполнить.

```
USE AdventureWorks2012 ;
GO
CREATE VIEW HumanResources.EmployeeHireDate
AS
SELECT p.FirstName, p.LastName, e.HireDate
FROM HumanResources.Employee AS e JOIN Person.Person AS  p
ON e.BusinessEntityID = p.BusinessEntityID ;
GO
-- Query the view
SELECT FirstName, LastName, HireDate
FROM HumanResources.EmployeeHireDate
ORDER ВУ LastName;
```

Триггеры

Триггер — SQL Server - это часть процедурного кода, подобная хранимой процедуре, которая выполняется только при наступлении определенного события. Существуют разные типы событий, которые могут вызвать срабатывание триггера. Например, вставка строк в таблицу, изменение структуры таблицы и вход пользователя в экземпляр SQL Server.

Триггеры отличаются от хранимых процедур по трем основным характеристикам:

- пользователь не может запускать триггеры вручную;
- триггеры не принимают параметры;

- вы не можете зафиксировать или откатить транзакцию внутри триггера;

Тот факт, что нельзя использовать параметры в триггерах, не является ограничением для получения информации из события срабатывания.

В SQL Server есть два класса триггеров:

- DDL (язык определения данных) триггеры. Этот класс триггеров срабатывает при событиях, которые изменяют структуру (например, создание, изменение или удаление таблицы), или при определенных событиях, связанных с сервером, таких как изменения безопасности или события обновления статистики.
- Триггеры DML (язык модификации данных). Это наиболее часто используемый класс триггеров. В этом случае событие срабатывания является заявлением об изменении данных; это может быть оператор вставки, обновления или удаления в таблице или в представлении.

Кроме того, триггеры DML бывают разных типов:

- FOR или AFTER [INSERT, UPDATE, DELETE]: эти типы триггеров выполняются после завершения инструкции запуска (вставка, обновление или удаление).
- INSTEAD OF [INSERT, UPDATE, DELETE]: в отличие от типа FOR (AFTER), триггеры INSTEAD OF выполняются вместо оператора запуска. Другими словами, этот тип триггера заменяет инструкцию firing. Это очень полезно в тех случаях, когда вам нужно иметь перекрестную ссылочную целостность базы данных.

Триггеры применяются для обеспечения целостности данных и реализации сложной бизнес-логики. Триггер запускается сервером автоматически при попытке изменения данных в таблице, с которой он связан. Все производимые им модификации данных рассматриваются как выполняемые в транзакции, в которой выполнено действие, вызвавшее срабатывание триггера. Соответственно, в случае обнаружения ошибки или нарушения целостности данных может произойти откат этой транзакции.

Когда происходит запуск триггера, говорят, что он активизируется (*fire*). Триггер создается по одной таблице базы данных, но он может осуществлять доступ и к другим таблицам и объектам других баз данных. Триггеры нельзя создать по временным таблицам или системным таблицам, а только по определенным пользователем таблицам или представлениям. Таблица, по которой определяется триггер, называется *таблицей триггера*.

В случае если триггер вызывается до события, он может внести изменения в модифицируемую событием запись (конечно, при условии, что событие — не удаление записи). Некоторые СУБД накладывают ограничения на операторы, которые могут быть использованы в триггере (например, может быть запрещено вносить изменения в структуру таблицы, на которой «висит» триггер, и т.п.).

Создание триггеров

Триггер MS SQL Server создается с использованием оператора T-SQL CREATE TRIGGER, который имеет следующий синтаксис:

```
CREATE TRIGGER имя_триггера ON {таблица |представление}[WITHENCRYPTION]
{FOR | AFTER | INSTEAD OF}
{[DELETE] [,] [INSERT] [,][UPDATE]}[WITH APPEND]
[NOT FOR REPLICATION]
AS
оператор_sql[...n]
```

Как видно из этого описания, триггер может быть создан для операторов INSERT, UPDATE, DELETE или для любой комбинации из этих операторов.

Тип AFTER указывает, что триггер срабатывает только после успешного выполнения всех операций в инструкции SQL, запускаемой триггером. Все каскадные действия и проверки ограничений, на которые имеется ссылка, должны быть успешно завершены, прежде чем триггер сработает. Если единственным заданным ключевым словом является FOR, аргумент AFTER используется по умолчанию.

При вызове триггера будут выполнены операторы SQL, указанные после ключевого слова AS. Здесь могут использоваться программные конструкции, такие как IF и WHILE.

Механизм триггеров в MS SQL Server использует логические (концептуальные) таблицы с именами deleted и inserted. Их называют таблицами, но они отличаются от реальных таблиц баз данных. Они хранятся в памяти, а не на диске. Эти две таблицы имеют одинаковую структуру с таблицей (одинаковые колонки и типы данных), по которой определяется данный триггер. Таблица deleted содержит копии строк, на которые повлиял оператор DELETE или UPDATE. В эту таблицу перемещаются строки, удаляемые из таблицы данного триггера. После этого к данным таблицы deleted можно осуществлять доступ из данного триггера. Таблица inserted содержит копии строк, добавленных к таблице данного триггера при выполнении оператора INSERT или UPDATE. Эти строки добавляются одновременно в таблицу триггера и в таблицу inserted. Поскольку оператор UPDATE обрабатывается как DELETE, после которого следует INSERT, то при использовании оператора UPDATE старые значения строк копируются в таблицу deleted, а новые значения строк – в таблицу триггера и в таблицу inserted.

Создадим INSERT/UPDATE-триггер, который будет реализовывать ограничение уровня строки для таблицы *Trip* нашей базы данных *TaxiService*. Этот триггер проверит, что дата окончания поездки не стоит раньше даты начала поездки.

Для создания триггера создадим SQL-запрос к базе данных, в котором выполним следующие инструкции:

```
CREATE TRIGGER EndDate ON Trip
AFTER INSERT, UPDATE
AS BEGIN
SET NOCOUNT ON;
DECLARE @end AS datetime, @start AS datetime
SELECT @start = start_date FROM inserted
SELECT @end = end_date FROM inserted
IF datediff(second, @start, @end) < 0
RAISERROR ('Поездка не может закончится раньше ее начала', 16, 10)
END
```

Триггер EndDate создан. Он появился в папке триггеров таблицы Trip (как показано на рисунке А2)

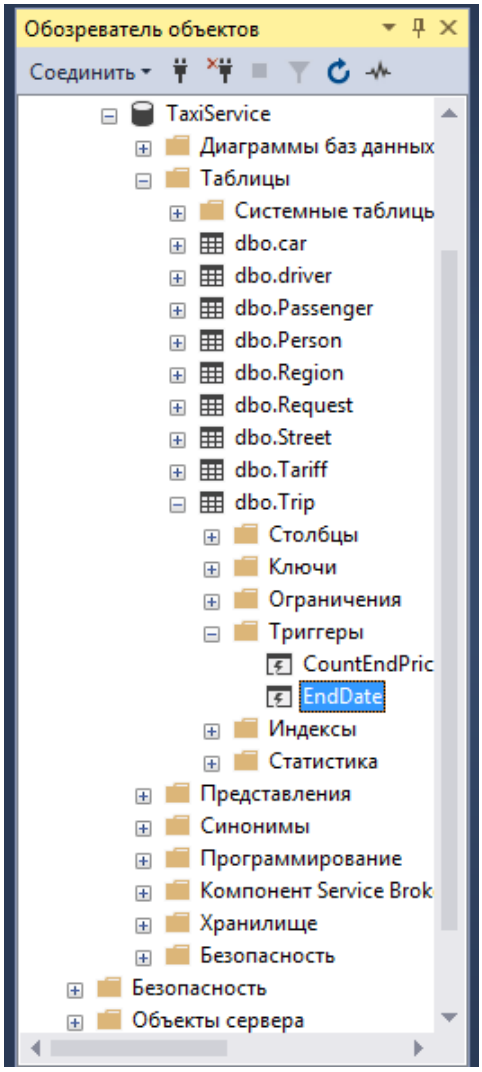

Рисунок А2 – Папка триггеров таблицы Trip

Чтобы проверить этот триггер, откроем таблицу Trip, выбрав в контекстном меню обозревателя объектов пункт «Изменить первые 200 строк»,

и попробуем ввести дату окончания меньше даты начала. СУБД отказывается сохранить некорректные данные и выдает запрограммированное нами сообщение об ошибке (как показано на рисунке А3).

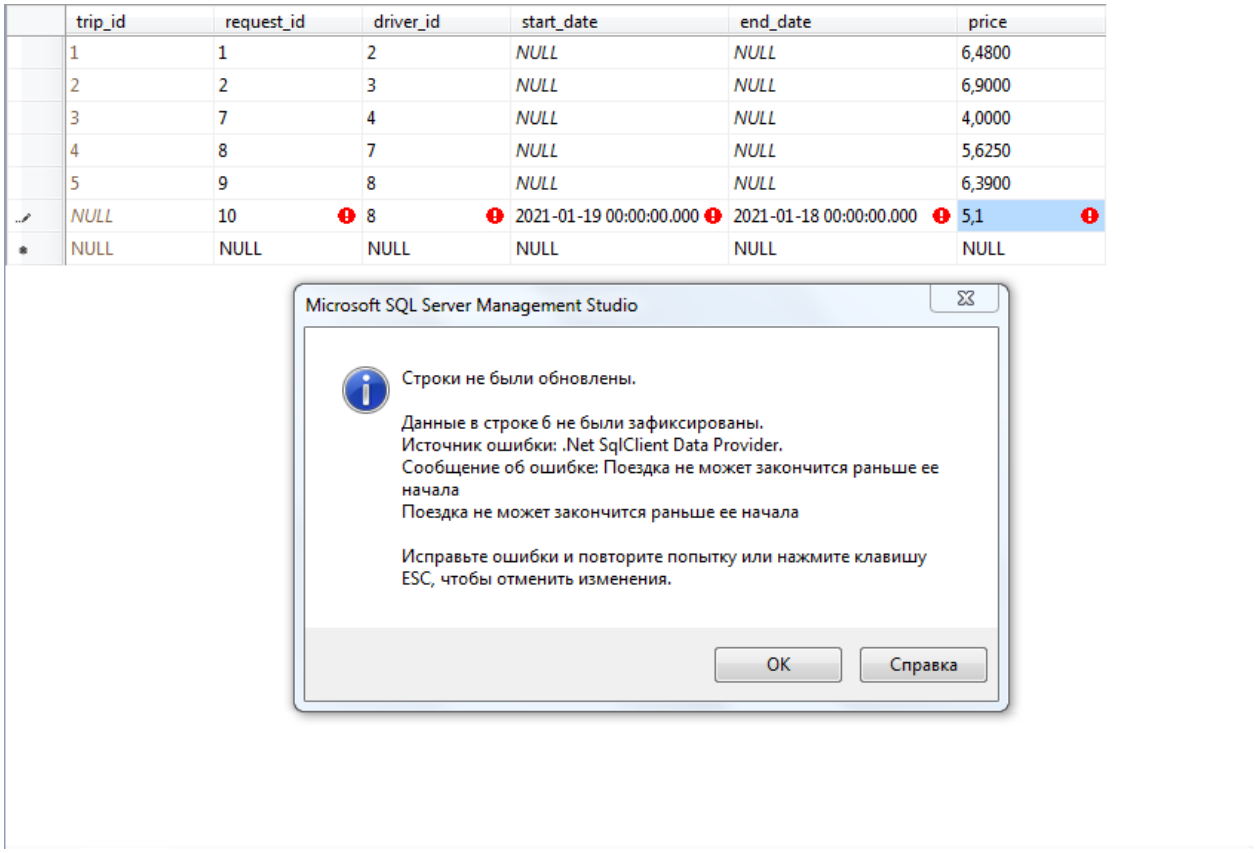

Рисунок А3 – Запрограммированное сообщение об ошибке

Далее будут приведены примеры ещё нескольких триггеров для БД TaxiService.

Проверка возраста для водителя (требование – 18+):

```
CREATE TRIGGER DriverAgeVerification ON driver
AFTER INSERT, UPDATE
AS BEGIN
SET NOCOUNT ON;
DECLARE @person_id AS int, @birthdate AS datetime
SELECT @person_id = person_id from inserted
SELECT @birthdate = birthdate
FROM Person
WHERE Person.person_id = @person_id
IF datediff(day,@birthdate,getdate()) / 365 < 18
RAISERROR ('Данный человек не может быть водителем, т.к. ему меньше 18 лет', 16, 10)
END
```

Триггер, проверяющий, не используется ли при создании пассажира Passenger человек Person в таблице водителей Driver:

```
CREATE TRIGGER PersonNotUsedInDriver ON Passenger
AFTER INSERT, UPDATE
AS BEGIN
SET NOCOUNT ON;
DECLARE @person_id AS int, @count_driver AS int
SELECT @person_id = person_id FROM inserted
SELECT @count_driver = COUNT(*) FROM Driver WHERE person_id = @person_id
IF @count_driver != 0
RAISERROR('Такой пользователь уже существует', 16, 10)
END
```

Триггер, проверяющий, может ли вводимый/модифицированный водитель иметь указанный опыт вождения *work_experience*:

```
CREATE TRIGGER WorkExperienceVerification ON driver
AFTER INSERT, UPDATE
AS BEGIN
SET NOCOUNT ON;
DECLARE @person_id AS int, @birthdate AS datetime, @exp AS int
SELECT @person_id = person_id from inserted
SELECT @exp = work_experience from inserted
SELECT @birthdate =  birthdate
FROM Person
WHERE Person.person_id = @person_id
IF datediff(day, @birthdate, getdate()) / 365 - 18 < @exp
RAISERROR ('Данный водитель не может иметь такой опыт работы', 16, 10)
END
```

Триггер, проверяющий, не производится ли поездка «на месте» (адрес посадки не совпадает с адресом назначения):

```
CREATE TRIGGER CheckAdress ON Request
AFTER INSERT, UPDATE
AS BEGIN
SET NOCOUNT ON;
DECLARE @request_street AS int, @request_house AS int, @destination_street AS int, @destination_house AS int
SELECT @request_street = request_street_id FROM inserted
SELECT @destination_street = destination_street_id FROM inserted
SELECT @request_house = request_house_number FROM inserted
SELECT @destination_house = destination_house_number FROM inserted
```

IF @request_street = @destination_street AND @request_house = @destination_house

RAISERROR('Точка прибытия совпадает с точкой посадки', 16, 10)

END

Хранимые процедуры

Хранимая процедура - это специальный тип пакета инструкций Transact-SQL, созданный, используя язык SQL и процедурные расширения. Основное различие между пакетом и хранимой процедурой состоит в том, что последняя сохраняется в виде объекта базы данных. Иными словами, хранимые процедуры сохраняются на стороне сервера, чтобы улучшить производительность и постоянство выполнения повторяемых задач.

Компонент Database Engine поддерживает хранимые процедуры и системные процедуры. Хранимые процедуры создаются таким же образом, как и все другие объекты баз данных, т.е. при помощи языка DDL. Системные процедуры предоставляются компонентом Database Engine и могут применяться для доступа к информации в системном каталоге и ее модификации.

При создании хранимой процедуры можно определить необязательный список параметров. Таким образом, процедура будет принимать соответствующие аргументы при каждом ее вызове. Хранимые процедуры могут возвращать значение, содержащее определенную пользователем информацию или, в случае ошибки, соответствующее сообщение об ошибке.

Хранимая процедура предварительно компилируется перед тем, как она сохраняется в виде объекта в базе данных. Предварительно компилированная форма процедуры сохраняется в базе данных и используется при каждом ее вызове. Это свойство хранимых процедур предоставляет важную выгоду, заключающуюся в устранении (почти во всех случаях) повторных компиляций процедуры и получении соответствующего улучшения производительности. Это свойство хранимых процедур также оказывает положительный эффект на объем данных, участвующих в обмене между системой баз данных и приложениями. В частности, для вызова хранимой процедуры объемом в несколько тысяч байтов может потребоваться меньше, чем 50 байт. Когда множественные пользователи выполняют повторяющиеся задачи с применением хранимых процедур, накопительный эффект такой экономии может быть довольно значительным.

Хранимые процедуры можно также использовать для следующих целей:

управления авторизацией доступа;

для создания журнала логов о действиях с таблицами баз данных.

Использование хранимых процедур предоставляет возможность управления безопасностью на уровне, значительно превышающем уровень безопасности, предоставляемый использованием инструкций GRANT и REVOKE, с помощью которых пользователям предоставляются разные привилегии доступа. Это

возможно вследствие того, что авторизация на выполнение хранимой процедуры не зависит от авторизации на модифицирование объектов, содержащихся в данной хранимой процедуре, как это описано в следующем разделе.

Хранимые процедуры, которые создают логи операций записи и/или чтения таблиц, предоставляют дополнительную возможность обеспечения безопасности базы данных. Используя такие процедуры, администратор базы данных может отслеживать модификации, вносимые в базу данных пользователями или прикладными программами.

Создание и исполнение хранимых процедур

Хранимые процедуры создаются посредством инструкции CREATE PROCEDURE, которая имеет следующий синтаксис:

```
CREATE PROC[EDURE] [schema_name.]proc_name
   [({@param1} type1 [ VARYING] [= default1] [OUTPUT])] {, …}
   [WITH {RECOMPILE | ENCRYPTION | EXECUTE AS 'user_name'}]
   [FOR REPLICATION]
   AS batch | EXTERNAL NAME method_name
```

Соглашения по синтаксису

Параметр schema_name определяет имя схемы, которая назначается владельцем созданной хранимой процедуры. Параметр proc_name определяет имя хранимой процедуры. Параметр @param1 является параметром процедуры (формальным аргументом), чей тип данных определяется параметром type1. Параметры процедуры являются локальными в пределах процедуры, подобно тому, как локальные переменные являются локальными в пределах пакета. Параметры процедуры - это значения, которые передаются вызывающим объектом процедуре для использования в ней. Параметр default1 определяет значение по умолчанию для соответствующего параметра процедуры. (Значением по умолчанию также может быть NULL.)

Опция OUTPUT указывает, что параметр процедуры является возвращаемым, и с его помощью можно возвратить значение из хранимой процедуры вызывающей процедуре или системе.

Как уже упоминалось ранее, предварительно компилированная форма процедуры сохраняется в базе данных и используется при каждом ее вызове. Если же по каким-либо причинам хранимую процедуру требуется компилировать при каждом ее вызове, при объявлении процедуры используется опция WITH RECOMPILE. Использование опции WITH RECOMPILE сводит на нет одно из наиболее важных преимуществ хранимых процедур: улучшение производительности благодаря одной компиляции. Поэтому опцию WITH RECOMPILE следует использовать только при частых изменениях используемых хранимой процедурой объектов базы данных.

Предложение EXECUTE AS определяет контекст безопасности, в котором должна исполняться хранимая процедура после ее вызова. Задавая этот контекст, с помощью Database Engine можно управлять выбором учетных записей пользователей для проверки полномочий доступа к объектам, на которые ссылается данная хранимая процедура.

По умолчанию использовать инструкцию CREATE PROCEDURE могут только члены предопределенной роли сервера sysadmin и предопределенной роли базы данных db_owner или db_ddladmin. Но члены этих ролей могут присваивать это право другим пользователям с помощью инструкции GRANT CREATE PROCEDURE.

В примере ниже показано создание простой хранимой процедуры для работы с таблицей Project:

```
USE TaxiService;
GO
   CREATE PROCEDURE IncreaseeTariff (@percent INT=5)
      AS UPDATE Tariff
      SET tariff_multiplier = tariff_multiplier + Bu tariff_multiplier * @percent/100;
```

Как говорилось ранее, для разделения двух пакетов используется инструкция GO. Инструкцию CREATE PROCEDURE нельзя объединять с другими инструкциями Transact-SQL в одном пакете. Хранимая процедура IncreaseBudget увеличивает бюджеты для всех проектов на определенное число процентов, определяемое посредством параметра @percent. В процедуре также определяется значение числа процентов по умолчанию, которое применяется, если во время выполнения процедуры этот аргумент отсутствует.

Хранимые процедуры могут обращаться к несуществующим таблицам. Это свойство позволяет выполнять отладку кода процедуры, не создавая сначала соответствующие таблицы и даже не подключаясь к конечному серверу.

В отличие от основных хранимых процедур, которые всегда сохраняются в текущей базе данных, возможно создание временных хранимых процедур, которые всегда помещаются во временную системную базу данных tempdb. Одним из поводов для создания временных хранимых процедур может быть желание избежать повторяющегося исполнения определенной группы инструкций при соединении с базой данных. Можно создавать локальные или глобальные временные процедуры. Для этого имя локальной процедуры задается с одинарным символом # (#proc_name), а имя глобальной процедуры - с двойным (##proc_name).

Локальную временную хранимую процедуру может выполнить только создавший ее пользователь и только в течение соединения с базой данных, в

которой она была создана. Глобальную временную процедуру могут выполнять все пользователи, но только до тех пор, пока не завершится последнее соединение, в котором она выполняется (обычно это соединение создателя процедуры).

Жизненный цикл хранимой процедуры состоит из двух этапов: ее создания и ее выполнения. Каждая процедура создается один раз, а выполняется многократно. Хранимая процедура выполняется посредством инструкции EXECUTE пользователем, который является владельцем процедуры или обладает правом EXECUTE для доступа к этой процедуре. Инструкция EXECUTE имеет следующий синтаксис:

```
[[EXEC[UTE]] [@return_status =] {proc_name | @proc_name_var}
   {[[@parameter1 =] value
                  | [@parameter1=] @variable [OUTPUT]] | DEFAULT}..
   [WITH RECOMPILE]
```

Соглашения по синтаксису

За исключением параметра return_status, все параметры инструкции EXECUTE имеют такое же логическое значение, как и одноименные параметры инструкции CREATE PROCEDURE. Параметр return_status определяет целочисленную переменную, в которой сохраняется состояние возврата процедуры. Значение параметру можно присвоить, используя или константу (value), или локальную переменную (@variable). Порядок значений именованных параметров не важен, но значения неименованных параметров должны предоставляться в том порядке, в каком они определены в инструкции CREATE PROCEDURE.

Предложение DEFAULT предоставляет значения по умолчанию для параметра процедуры, которое было указано в определении процедуры. Когда процедура ожидает значение для параметра, для которого не было определено значение по умолчанию и отсутствует параметр, либо указано ключевое слово DEFAULT, то происходит ошибка.

Когда инструкция EXECUTE является первой инструкцией пакета, ключевое слово EXECUTE можно опустить. Тем не менее будет надежнее включать это слово в каждый пакет. Использование инструкции EXECUTE показано в примере ниже:

```
USE SampleDb;
EXECUTE IncreaseBudget 10;
```

Инструкция EXECUTE в этом примере выполняет хранимую процедуру IncreaseBudget, которая увеличивает бюджет всех проектов на 10%.

В примере ниже показано создание хранимой процедуры для обработки данных в таблицах Passenger и Person:

```
USE SampleDb;
```

```
GO
CREATE PROCEDURE ModifyEmpId (@oldId INTEGER, @newId INTEGER)
    AS UPDATE Passenger
      SET passenger_id = @newId
      WHERE passenger_id = @oldId;
    UPDATE Person
      SET person_id = @newId
      WHERE person_id = @oldId;
```

Процедура ModifyEmpId в примере иллюстрирует использование хранимых процедур, как часть процесса обеспечения ссылочной целостности (в данном случае между таблицами Passenger и Person). Подобную хранимую процедуру можно использовать внутри определения триггера, который собственно и обеспечивает ссылочную целостность.

В примере ниже показано использование в хранимой процедуре предложения OUTPUT:

```
USE TaxiService;
GO
CREATE PROCEDURE DeleteEmployee @empId INT, @counter INT OUTPUT
    AS SELECT @counter = COUNT(*)
      FROM Driver
      WHERE driver_id = @empId
    DELETE FROM Person
      WHERE person_id = @empId
    DELETE FROM driver
      WHERE driver_id = @empId;
```

Данную хранимую процедуру можно запустить на выполнение посредством следующих инструкций:

```
DECLARE @quantityDeleteEmployee INT;
EXECUTE DeleteEmployee @empId=18316,
@counter=@quantityDeleteEmployee OUTPUT;

PRINT N'Удалено сотрудников: ' + convert(nvarchar(30),
@quantityDeleteEmployee);
```

Эта процедура подсчитывает количество проектов, над которыми занят сотрудник с табельным номером @empId, и присваивает полученное значение параметру ©counter. После удаления всех строк для данного табельного номера из таблиц Employee и Works_on вычисленное значение присваивается переменной @quantityDeleteEmployee.

Значение параметра возвращается вызывающей процедуре только в том случае, если указана опция OUTPUT. В примере выше процедура DeleteEmployee передает вызывающей процедуре параметр @counter, следовательно, хранимая процедура возвращает значение системе. Поэтому параметр @counter необходимо указывать как в опции OUTPUT при объявлении процедуры, так и в инструкции EXECUTE при ее вызове.

Практическая часть:

1. Создайте представление, содержащее только 2 поля.
2. Добавьте представление, содержащее поля из нескольких таблиц.
3. Напишите хранимую процедуру для добавления новых строк в таблицу.
4. Добавьте хранимую процедуру с оператором IF.
5. Создайте хранимую процедуру с оператором IF ELSE.
6. Напишите хранимую процедуру с оператором DECLARE.
7. Реализуйте INSERT/UPDATE-триггер, который будет реализовывать ограничение уровня строки для какой-либо таблицы модели вашей базы. Проверьте работу созданного триггера.
8. Добавьте AFTER INSERT – триггер.
9. Создайте триггер с условием WHEN.

Контрольные вопросы:

1. Какие структуры контроля целостности данных вы знаете?
2. Назовите виды ограничения целостности баз данных, которые вы знаете.
3. Что такое FOREIGN KEY, REFERENCES.
4. Что такое атомарность выражения?
5. Как вы понимаете, что означает распараллеливание операций?
6. Что такое представление? Какие типы представлений вы знаете?
7. Какие существуют ограничения на создание представлений?
8. Зачем нужны представления?
9. Какими способами можно создать представление?
10. Что такое триггеры?
11. Назовите несколько типов триггеров.
12. Какие классы триггеров вы знаете?
13. Для чего используются таблицы inserted и deleted?
14. Что такое хранимая процедура?
15. Каким оператором можно создать процедуру?
16. Зачем нужны хранимые процедуры?

17. Можем ли мы проверить блокировки в базе данных? Если так, как мы можем сделать эту проверку блокировки?

ПРИЛОЖЕНИЕ Б

Методические указания для лабораторной работы №2

Лабораторная работа №2

«SQL инъекции, резервное копирование данных, транзакции»

Цель: изучение, а также использование на практике методов резервного копирования данных, транзакций, SQL инъекций и способов защиты.

SQL-инъекции

SQL-инъекция – это тип атаки с использованием инъекций, позволяющий выполнять вредоносные операторы SQL. Эти операторы управляют сервером базы данных за веб-приложением. Злоумышленники могут использовать уязвимости SQL-инъекциям, чтобы обойти меры безопасности приложений. Они могут обходить аутентификацию и авторизацию веб-страницы или веб-приложения и извлекать содержимое всей базы данных SQL. Они также могут использовать SQL-инъекцию для добавления, изменения и удаления записей в базе данных.

Уязвимость SQL-инъекциям может затронуть любой веб-сайт или веб-приложение, использующее базу данных SQL, например, MySQL, Oracle, SQL Server или другие. Преступники могут использовать его для получения несанкционированного доступа к вашим конфиденциальным данным: информации о клиентах, личным данным, коммерческой тайне, интеллектуальной собственности и т. д. Атаки с использованием SQL-инъекций - одна из самых старых, наиболее распространенных и самых опасных уязвимостей веб-приложений. Успешная атака с использованием SQL-инъекции может иметь очень серьезные последствия.

– Злоумышленники могут использовать SQL-инъекции для поиска учетных данных других пользователей в базе данных. Затем они могут выдавать себя за этих пользователей. Так злоумышленник может получить права администратора базы данных со всеми привилегиями.

– SQL позволяет выбирать и выводить данные из базы данных. Уязвимость SQL-инъекции может позволить злоумышленнику получить полный доступ ко всем данным на сервере базы данных.

– SQL также позволяет изменять и добавлять новые данные в БД. Например, в финансовом приложении злоумышленник может использовать SQL-инъекцию для изменения балансов, аннулирования транзакций или перевода денег на свой счет.

– SQL может использоваться для удаления записей из базы данных, даже для удаления таблиц. И даже если администратор создает резервные копии базы данных, удаление данных может повлиять на доступность приложения до тех пор, пока база данных не будет восстановлена. Кроме того, резервные копии могут не охватывать самые свежие данные.

– На некоторых серверах вы можете получить доступ к операционной системе с помощью сервера базы данных. Это может быть намеренно или случайно. В таком случае злоумышленник может использовать SQL-инъекцию в качестве исходного вектора атаки, чтобы затем атаковать внутреннюю сеть за брандмауэром.

Существует несколько типов атак SQL-инъекций: внутриполосные (классические), логические (слепые) и внеполосные.

Внутриполосные инъекции**.** Злоумышленник использует один и тот же канал связи для запуска своих атак и сбора результатов. Простота и эффективность внутриполосных инъекций делают их одними из наиболее распространенных типов атак SQL-инъекций. Есть два под варианта этого метода:

– Инъекции на основе ошибок - злоумышленник выполняет действия, заставляющие базу данных создавать сообщения об ошибках. Злоумышленник потенциально может использовать данные, предоставленные этими сообщениями об ошибках, для сбора информации о структуре базы данных.

– Инъекции на основе объединения - этот метод использует преимущество оператора UNION SQL, который объединяет несколько операторов выбора, сгенерированных базой данных, для получения их в одном ответе. Этот ответ может содержать данные, которые могут быть использованы злоумышленником.

– Логические (слепые) инъекции**.**

Злоумышленник отправляет на сервер инъекции и наблюдает за реакцией и поведением сервера, чтобы узнать больше о его структуре. Этот метод называется слепой инъекцией, потому что данные из базы данных не передаются злоумышленнику, поэтому злоумышленник не может видеть информацию об атаке внутри канала.

Слепые SQL-инъекции зависят от реакции и поведенческих паттернов сервера, поэтому они обычно медленнее выполняются, но также опасны, как и другие виды инъекций. Слепые SQL-инъекции можно классифицировать следующим образом:

– Логические - этот злоумышленник отправляет SQL-запрос к базе данных, предлагая приложению вернуть результат. Результат будет зависеть от того, является ли запрос истинным или ложным. В зависимости от результата информация в ответе изменится или останется неизменной. Затем злоумышленник может определить, было ли сообщение верным или ложным.

– По времени - злоумышленник отправляет логический SQL-запрос к базе данных, который, допустим, заставляет базу данных ждать (в течение периода в секундах), прежде чем она сможет отреагировать, если логическое выражение истинно, и, наоборот, не ждать и вернуть ответ сразу, если выражение было ложно. Таким образом, злоумышленник может определить, вернуло ли используемое им сообщение истину или ложь, не полагаясь на данные из базы данных.

– Внеполосные инъекции**.**

Злоумышленник может осуществить эту форму атаки только в том случае, если на сервере базы данных, используемом приложением, включены определенные функции. Эта форма атаки в основном используется как альтернатива внутриполосным и логическим методам инъекций.

Внеполосные инъекции используются, когда злоумышленник не может использовать тот же канал для запуска атаки и сбора информации, или, когда сервер слишком медленный и нестабильный для выполнения этих действий. Эти методы рассчитаны на способность сервера создавать DNS или HTTP-запросы для передачи данных злоумышленнику.

Злоумышленник, желающий выполнить SQL-инъекцию, манипулирует стандартным SQL-запросом, чтобы использовать непроверенные входные уязвимости в базе данных. Есть много способов, которыми может быть выполнена атака, некоторые из которых будут показаны здесь, чтобы дать вам общее представление о том, как работают инъекции.

Допустим, есть запрос, который извлекает из таблицы Person имя определенного пользователя, который выбирается по переданному в элемент queryParamBox параметру person_id (что показано на рисунке Б1).

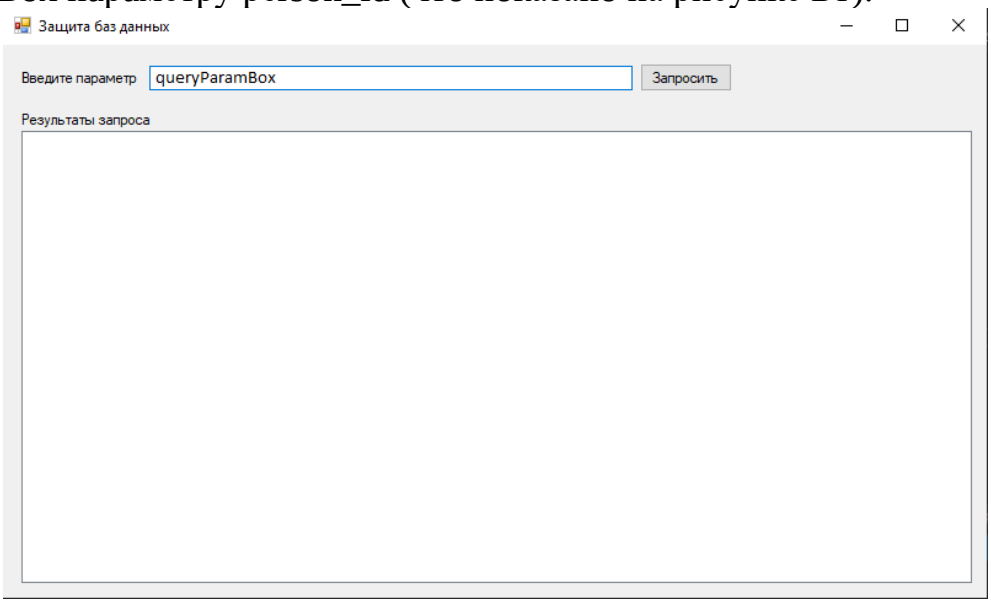

Рисунок Б1 – Выбор параметра для получения результатов запроса

```
string query = $"SELECT first_name, last_name FROM Person WHERE person_id = {queryParamBox.Text}";
SqlCommand selectCommand = new SqlCommand(query, connection);
SqlDataReader reader = selectCommand.ExecuteReader();
```

Рисунок Б2 – Запрос к базе данных

Мы можем передать в этот запрос строку «1 or 1=1». В результате запрос вернет все записи из таблицы, потому что выполнилось условие 1=1. Получившийся запрос указан на рисунке Б3.

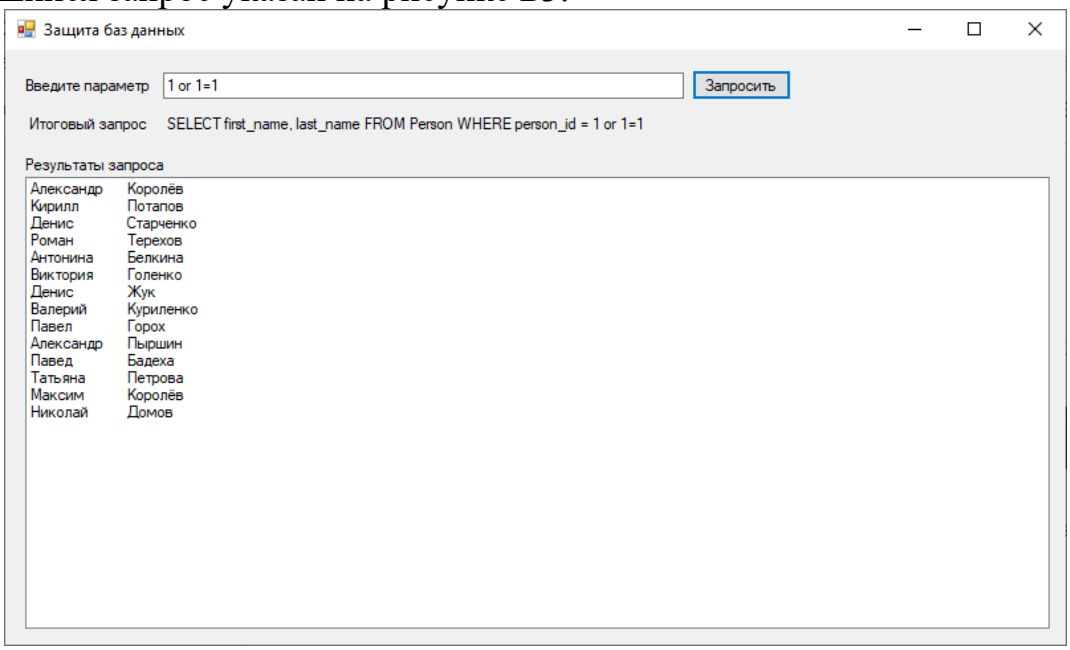

Рисунок Б3 – Результат выполнения запроса

Другой способ обработки запросов SQL - это оператор UNION SELECT. Это объединяет два несвязанных запроса SELECT для извлечения данных из разных таблиц.

Например, в строку параметра введем «1 UNION SELECT car_brand, car_model FROM car». На выходе кроме имени пользователя с person_id = 1 также будут получены все названия машин из таблицы cars (что показано на рисунке Б4).

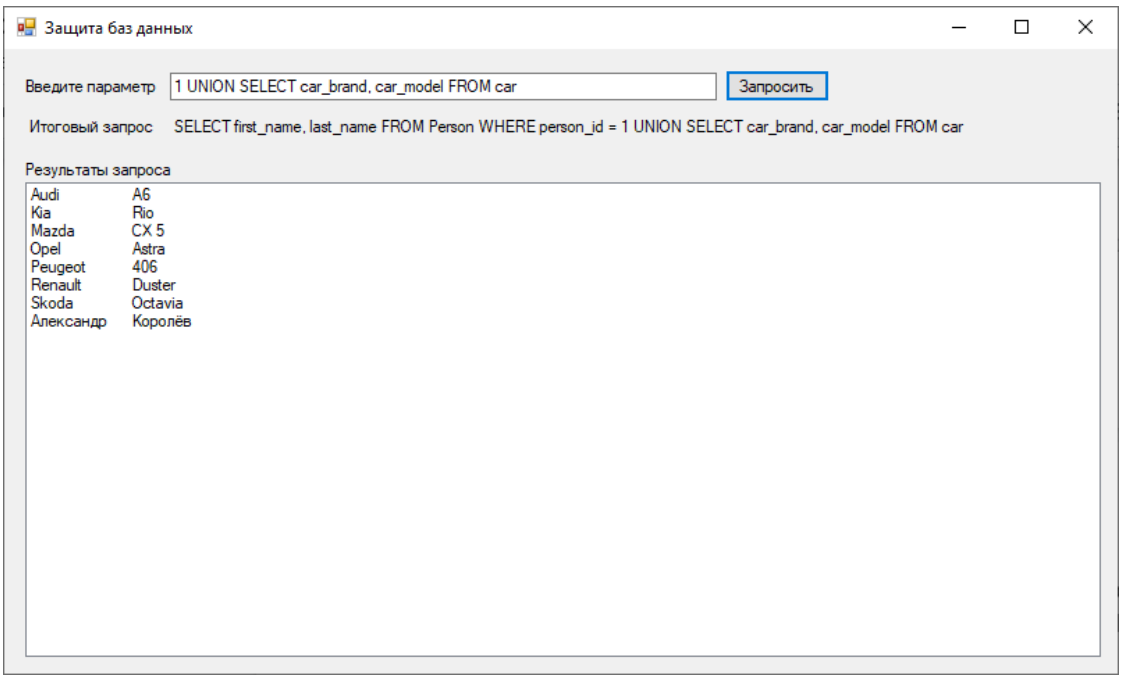

Рисунок Б4 – Результат выполнения запроса

Злоумышленники также могут использовать неправильно отфильтрованные символы для изменения команд SQL, включая использование точки с запятой для разделения двух полей.

Например, если строкой параметра передать «1; DROP TABLE car», злоумышленник отправит запрос, который удалит таблицу cars из базы данных.

Удалённые таким способом данные могут быть восстановлены с помощью резервных копий, однако работа приложения может быть приостановлена до момента восстановления нарушенных данных. Однако всё равно некоторые из последних изменений тех данных могут быть утрачены.

Защита от SQL-инъекций

Недостатки внедрения SQL возникают, когда разработчики программного обеспечения создают динамические запросы к базе данных, которые включают вводимые пользователем данные. Чтобы защитить базу данных от атак с использованием SQL-инъекций, можно применить некоторые из этих основных методов:

Использование подготовленных выражений (с параметризованными запросами)

Использование подготовленных выражений - один из лучших способов предотвратить внедрение SQL. Его также проще написать и легче понять, чем динамические запросы SQL.

В подготовленных выражениях команда SQL использует параметр вместо того, чтобы вставлять значения непосредственно в команду, тем самым предотвращая выполнение серверной частью вредоносных запросов, которые вредны для базы данных. Таким образом, если пользователь ввел «1 OR 1=1» в

качестве входных данных, параметризованный запрос будет искать в таблице совпадение со всей строкой «1 OR 1=1».

Рекомендации для конкретных языков:

Java EE - PreparedStatement() с переменными связывания

.NET - параметризованные запросы, такие как SqlCommand() или OleDbCommand() с переменными связывания

PHP - PDO со строго типизированными параметризованными запросами.

Перепишем используемый ранее запрос в базу данных (что показано на рисунке Б2) с использованием подготовленных выражений (что показано на рисунке Б5).

```
string query = $"SELECT first_name, last_name FROM Person WHERE person_id = @person_id";
SqlCommand selectCommand = new SqlCommand(query, connection);

SqlParameter person_id = new SqlParameter("@person_id", System.Data.SqlDbType.Int);
person_id.Value = int.Parse(queryParamBox.Text);
selectCommand.Parameters.Add(person_id);

selectCommand.Prepare();
SqlDataReader reader = selectCommand.ExecuteReader();
```

Рисунок Б5 – Запрос с использованием подготовленных выражений

В данном коде также появилось требование к типу вводимого параметра, и, поэтому, если вводимая строка не будет целочисленным значением, нам выдаст соответствующее сообщение (как показано на рисунке Б6).

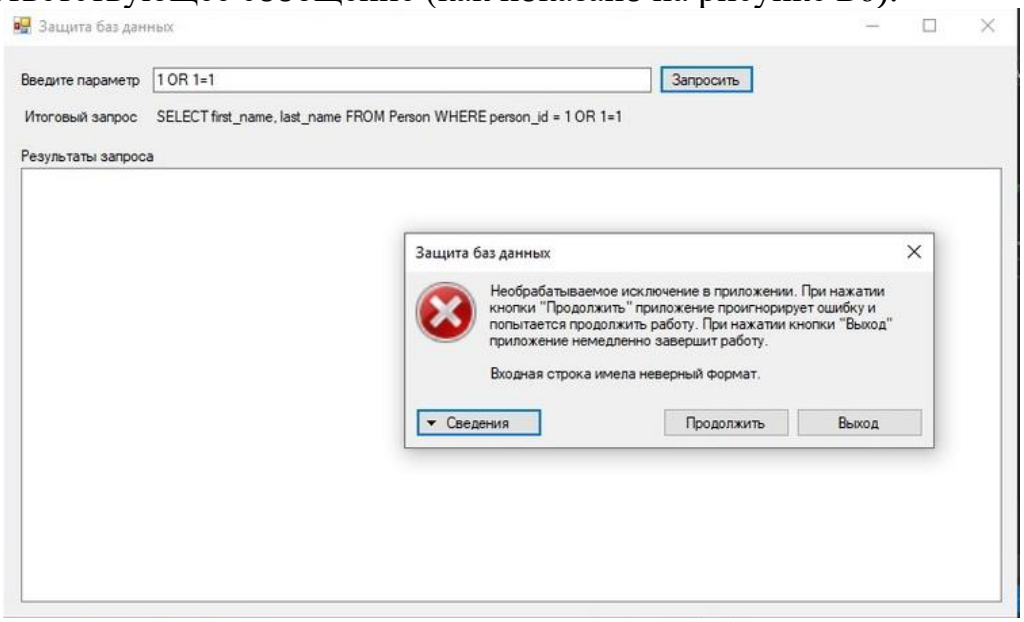

Рисунок Б6. – Сообщение об ошибке

Разработчикам, как правило, нравится подход с подготовленными выражениями, потому что весь код SQL остается в приложении. Это делает приложение относительно независимым от базы данных.

Использование хранимых процедур

Хранимые процедуры добавляют дополнительный уровень безопасности в базу данных помимо использования подготовленных операторов. Он выполняет экранирование, необходимое для того, чтобы приложение рассматривало ввод как данные, над которыми нужно работать, а не как код SQL, который нужно выполнить.

Разница между подготовленными операторами и хранимыми процедурами заключается в том, что код SQL для хранимой процедуры записывается и хранится на сервере базы данных, а затем вызывается из веб-приложения.

Если доступ пользователей к базе данных разрешен только через хранимые процедуры, разрешение пользователям на прямой доступ к данным не нужно явно предоставлять для какой-либо таблицы базы данных. Таким образом, база данных останется в безопасности.

Проверка ввода данных пользователем

Даже когда используются подготовленные операторы, сначала нужно выполнить проверку ввода, чтобы убедиться, что значение имеет принятый тип, длину, формат и т. д. Только ввод, прошедший проверку, может быть обработан в базе данных. Этот метод может остановить только самые тривиальные атаки, но он не устраняет основную уязвимость.

Ограничение привилегий

Не подключайтесь к базе данных с помощью учетной записи администратора, если это не требуется, потому что злоумышленники будут иметь возможность получить доступ ко всей системе. Поэтому лучше использовать учетную запись с ограниченными привилегиями, чтобы ограничить объем ущерба в случае SQL-инъекции.

Скрытие информации из сообщения об ошибке.

Сообщения об ошибках полезны для злоумышленников, чтобы узнать больше об архитектуре вашей базы данных, поэтому лучше показать общее сообщение об ошибке, говорящее о том, что что-то идет не так, и побудить пользователей обратиться в службу технической поддержки, если проблема не исчезнет.

Обновление вашей системы

Уязвимость SQL-инъекции является частой ошибкой программирования и обнаруживается регулярно, поэтому очень важно применять исправления и обновлять вашу систему до самой последней версии, насколько это возможно, особенно для вашего SQL Server.

Хранение учетных данных базы данных отдельно и в зашифрованном виде.

Если вы думаете, где хранить учетные данные вашей базы данных, также подумайте, насколько опасным может быть попадание в чужие руки. Поэтому

всегда храните учетные данные своей базы данных в отдельном файле и надежно зашифруйте его, чтобы злоумышленники не смогли получить большую выгоду. Кроме того, не храните конфиденциальные данные, если они вам не нужны, и удаляйте информацию, когда она больше не используется.

Резервное копирование БД.

Метод 1: Использование плана обслуживания для настройки резервного копирования SQL Server.

Планы обслуживания SQL Server это самый распространенный способ настройки регулярного резервного копирования.

Рассмотрим настройку резервного базы данных на SQL Server копирования по плану:

– Полная резервная копия каждые 24 часа

– Копия журнала транзакций – каждые 30 минут

В SSMS (SQL Server Management Studio) перейдите в раздел Management -> Maintenance Planes и запустите -> мастер создания плана обслуживания (как показано на рисунке Б7).

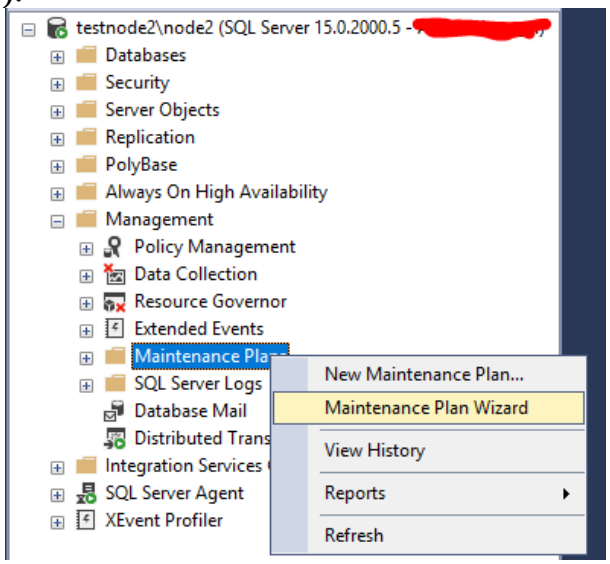

Рисунок Б7 – Переход в мастер плана обслуживания

Укажите имя плана и выберите режим «Separate schedules for each task» (как показано на рисунке Б8).

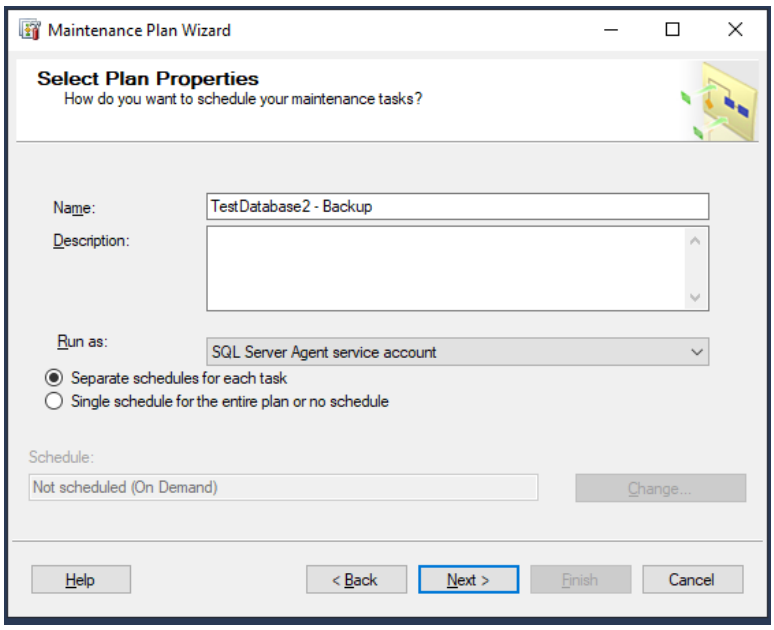

Рисунок Б8 – Задание имени плана

Выберите операции, которые нужно сделать в этом плане обслуживания (как показано на рисунке Б9):

– Back Up Database (Full);

– Back Up Database (Transaction Log).

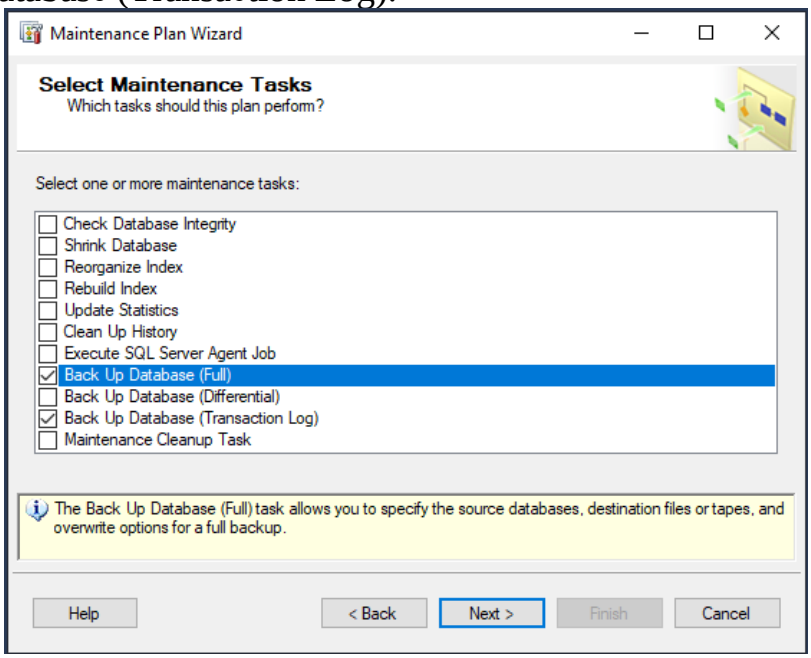

Рисунок.Б9 – Выбор опция для выполнения в рамках плана обслуживания

Используйте следующую последовательность операций (как показано на рисунке Б10):

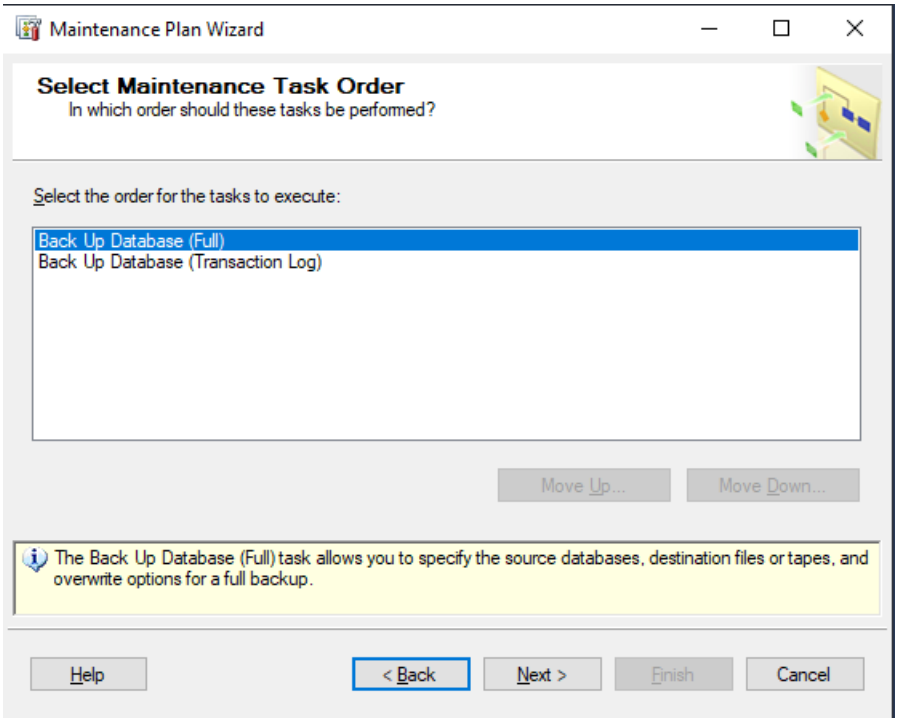

Рисунок.Б10 – Последовательность выполнения операций

Выберите базу данных SQL Server, которую нужно бэкапить (как показано на рисунке Б11) и выберите расписание (как показано на рисунке Б12).

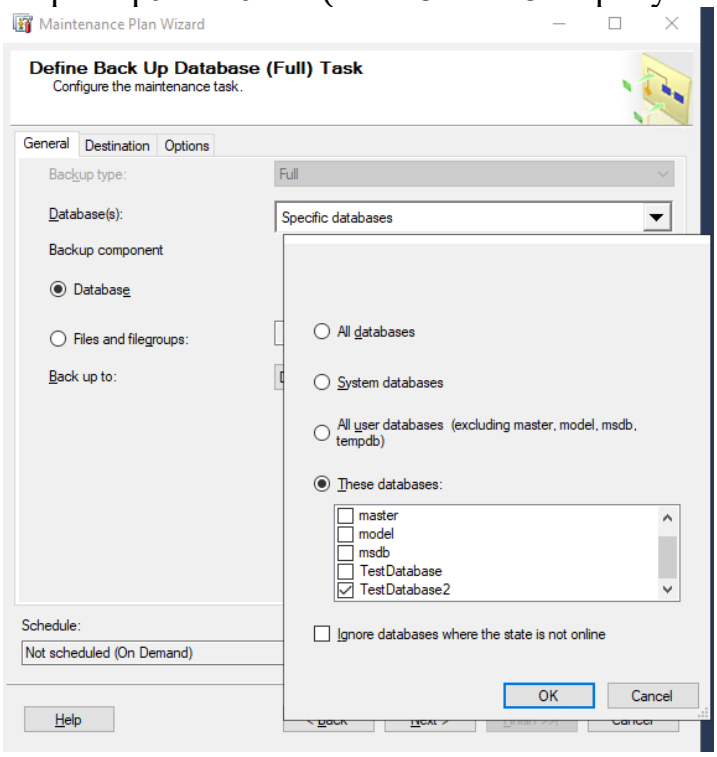

Рисунок Б11 – Выбор нужной базы данных

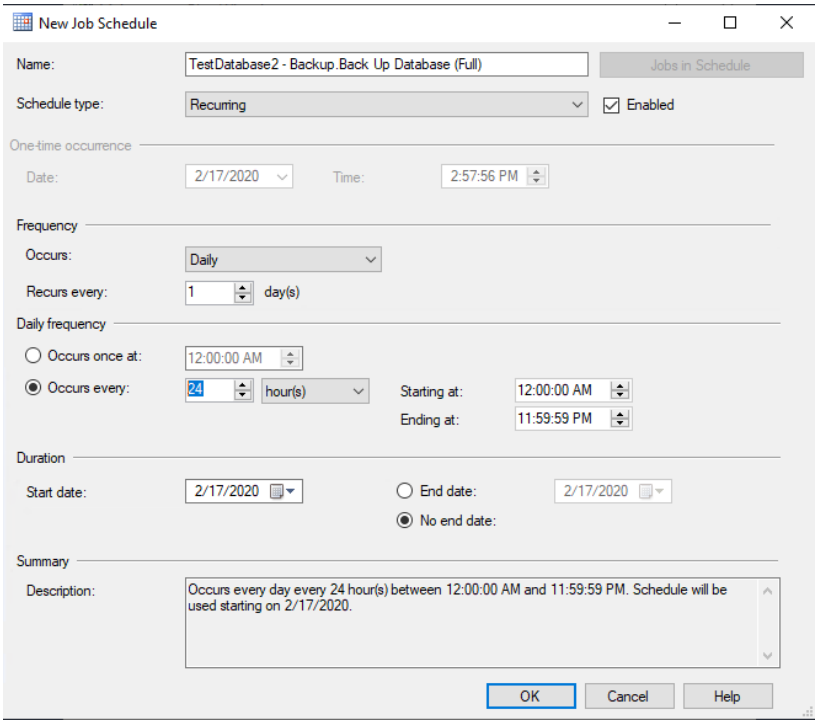

Рисунок Б12 – Выбор расписания

Укажите путь к каталогу, в который нужно сохранять резервные копию ваше базы данных (как показано на рисунке Б13).

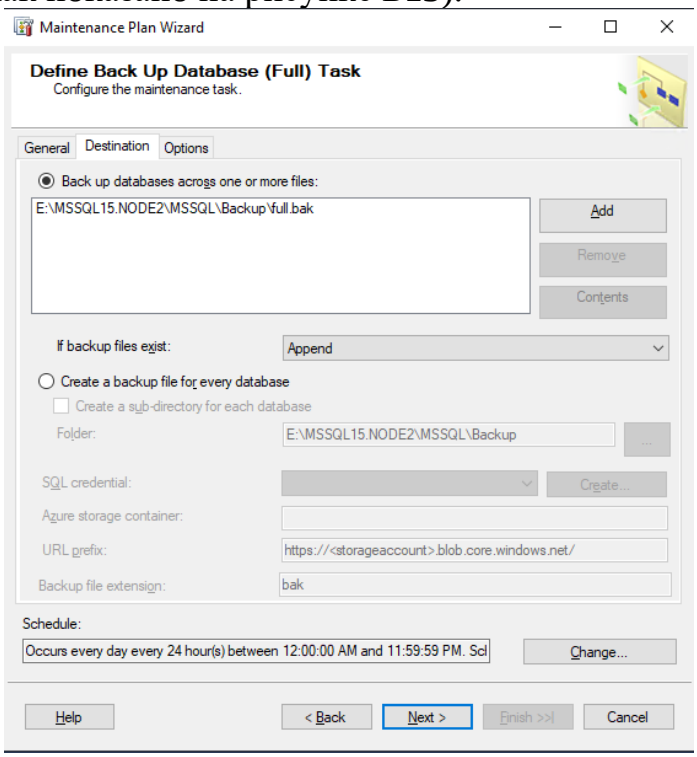

Рисунок Б13 – Указание пути к каталогу для сохранения резервной копии

Укажите сколько будут храниться резервные копии, как показано на рисунке Б14 (например, 14 дней).

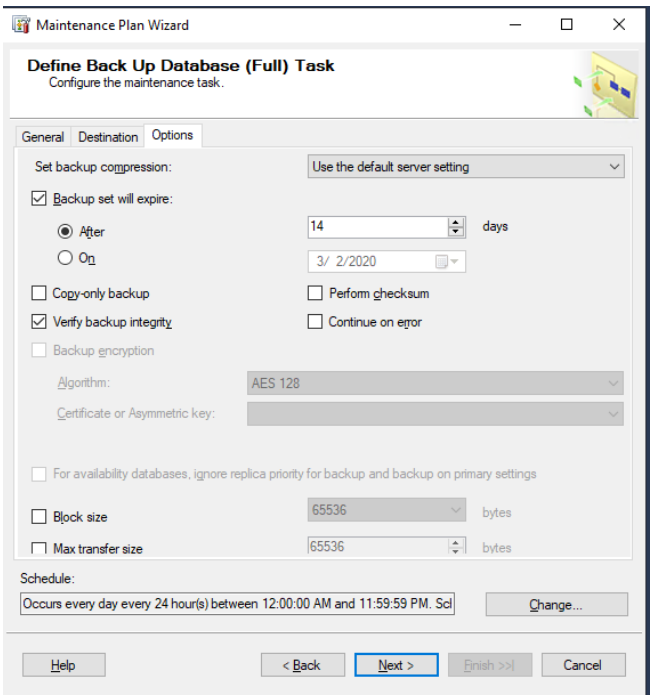

Рисунок Б14 – Настройка срока хранения резервных копий

Нажмите Next и аналогично создайте расписание резервного копирования для журнала транзакций (как показано на рисунке Б15).

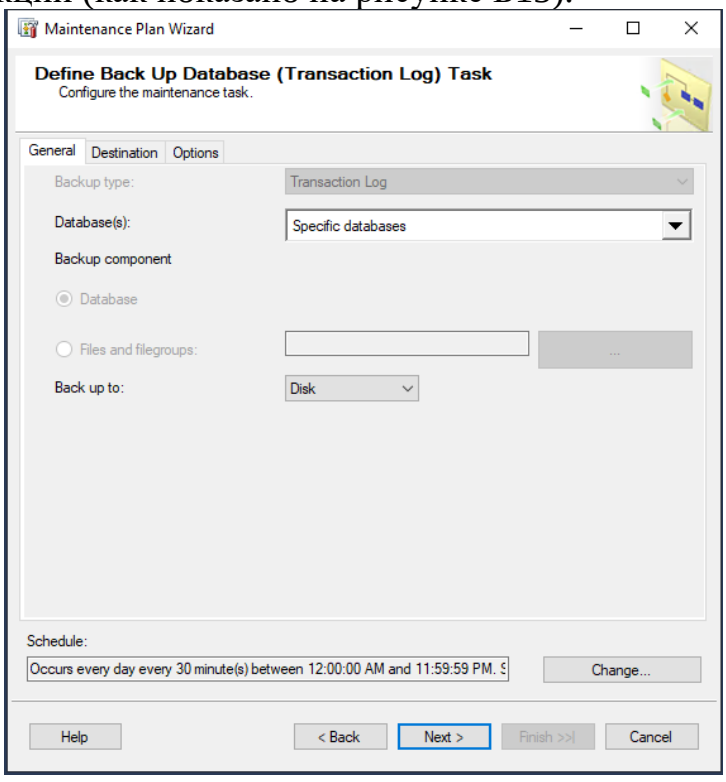

Рисунок Б15 – Создание расписания резервного копирования для журнала транзакций

Опционально можно указать файл для ведения лога плана обслуживания (как показано на рисунке Б16).

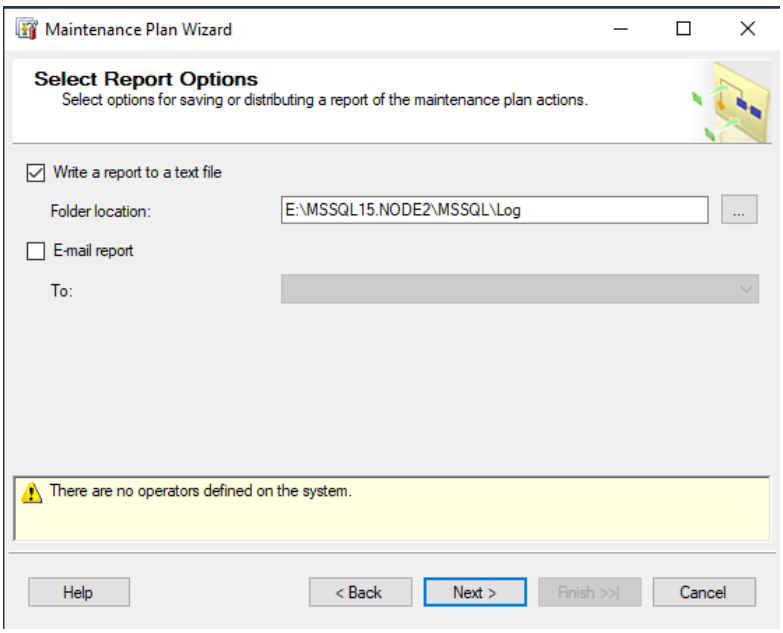

Рисунок Б16 - Указание файла для ведения лога плана обслуживания

Завершение настройки плана обслуживания SQL Server (как показано на рисунке Б17).

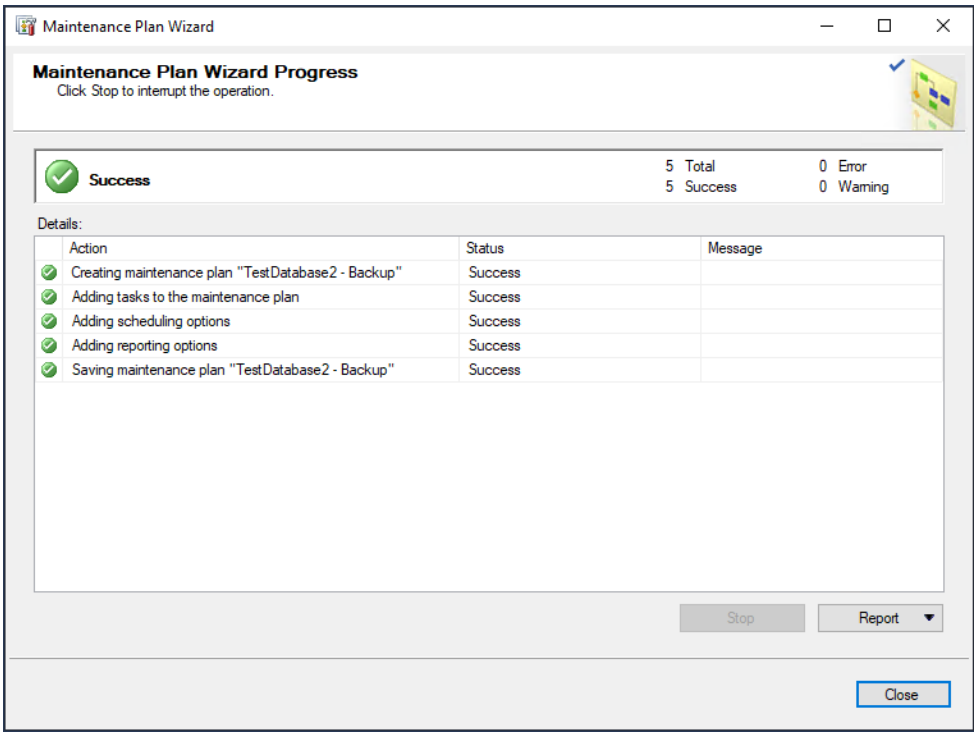

Рисунок Б17 – Завершение настройки плана обслуживания SQL Server

Выполните план обслуживания вручную (как показано на рисунке Б18) и проверьте журнал (как показано на рисунке Б19).

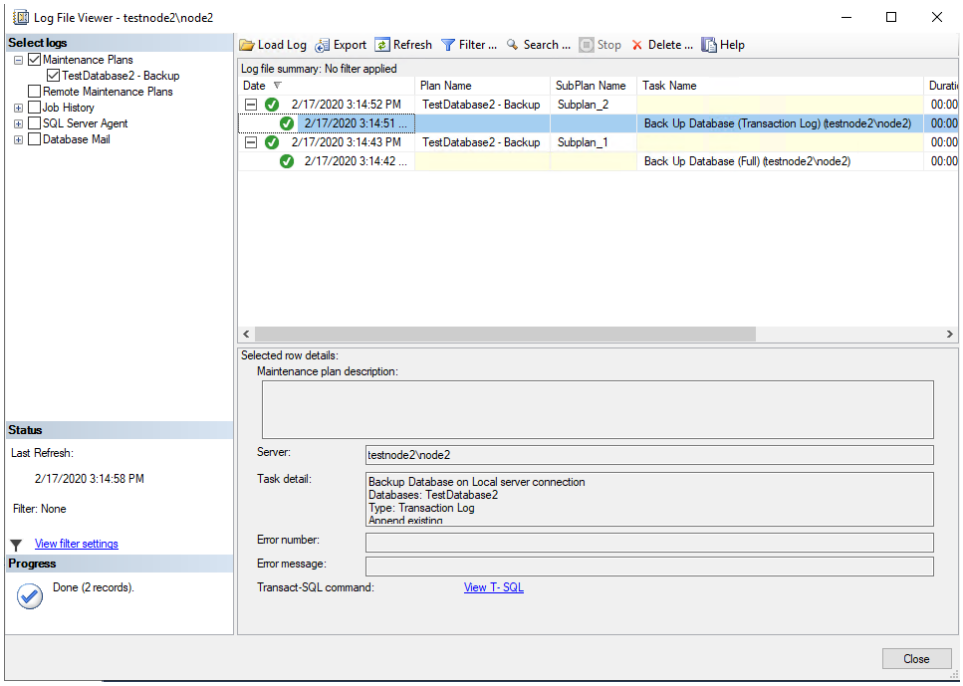

Рисунок Б18 – Демонстрация журнала плана обслуживания

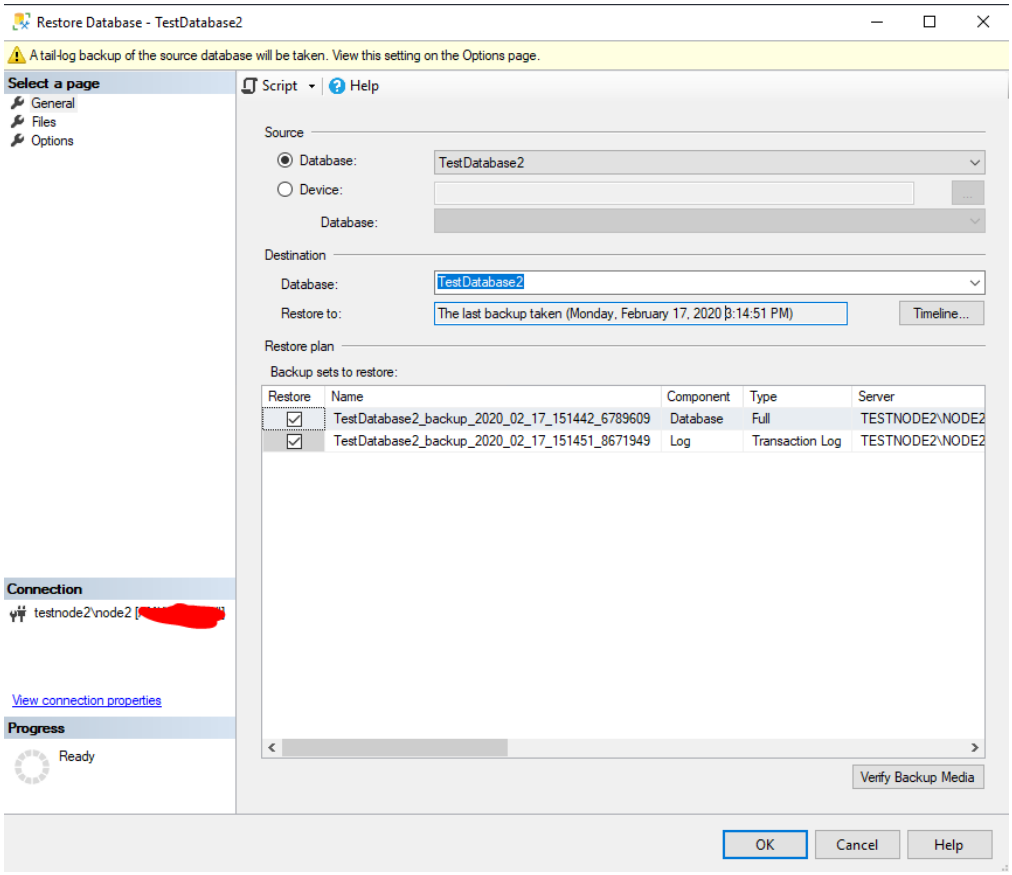

Рисунок Б19 – Демонстрация созданных копий

Как вы видите была создана полная резервная копия базы данных SQL Server и следом копия журнала транзакций. На этом настройка резервного копирования закончена.

Метод 2: Использование T-SQL для создания резервной копии на SQL Server

T-SQL - проверенный и надежный метод резервного копирования баз данных. При использовании T-SQL доступно больше опций для создания бэкапов, чем при использовании графического интерфейса. Большинство этих опций являются более продвинутыми. Очень базовый пример команды backup, которая создает полную резервную копию, представлен ниже. Затем следуют примеры дифференциального бэкапа и бэкапа журнала.

```
BACKUP DATABASE [MyDB] TO DISK = 'C:\Program Files\Microsoft SQL Server\MSSQL14\MSSQL\Backup\MyDB_Full.bak'
BACKUP DATABASE [MyDB] TO DISK = 'C:\Program Files\Microsoft SQL Server\MSSQL14\MSSQL\Backup\MyDB_Differntial.bak' WITH DIFFERENTIAL
BACKUP LOG [MyDB] TO DISK = 'C:\Program Files\Microsoft SQL Server\MSSQL14\MSSQL\Backup\MyDB_log.trn'
```

Транзакции

Транзакция – это последовательность операций, которые выполняется в логическом порядке пользователем, или программой, которая осуществляет работы с БД.

По сути, транзакция – это архив для запросов к базе данных, он защищает данные по принципу «всё или ничего».

В качестве примера можно привести следующее, допустим, вы решили отправить 10 файлов, какие есть варианты?

1. Отправить каждый файл по-отдельности
2. Отправить все файлы вместе

Казалось бы, что особой разницы как поступить – нет, но что, если соединение с интернетом во время отправки файлов прервётся? Если мы выбрали первый случай, то получатель получит 9 файлов, но не получит 1. Во втором случае получатель не получит ничего. На этом простом примере как раз и можно увидеть главное преимущество транзакций и то, почему они используются повсеместно – принцип «всё или ничего», получатель либо получит всю отправленную ему информацию, либо не получит ничего.

Использование транзакций

Транзакция задает последовательность инструкций языка Transact-SQL, применяемую программистами базы данных для объединения в один пакет операций чтения и записи для того, чтобы система базы данных могла обеспечить согласованность данных. Существует два типа транзакций:

Неявная транзакция – задает любую отдельную инструкцию INSERT, UPDATE или DELETE как единицу транзакции.

Явная транзакция - обычно это группа инструкций языка Transact-SQL, начало и конец которой обозначаются такими инструкциями, как BEGIN TRANSACTION, COMMIT и ROLLBACK.

Свойства транзакций

Транзакции обладают следующими свойствами, которые все вместе обозначаются сокращением ACID (Atomicity, Consistency, Isolation, Durability):

атомарность (Atomicity);

согласованность (Consistency);

изолированность (Isolation);

долговечность (Durability).

Свойство атомарности обеспечивает неделимость набора инструкций, который модифицирует данные в базе данных и является частью транзакции. Это означает, что или выполняются все изменения данных в транзакции, или в случае любой ошибки осуществляется откат всех выполненных изменений.

Свойство согласованности обеспечивает, что в результате выполнения транзакции база данных не будет содержать несогласованных данных. Иными словами, выполняемые транзакцией трансформации данных переводят базу данных из одного согласованного состояния в другое.

Свойство изолированности отделяет все параллельные транзакции друг от друга. Иными словами, активная транзакция не может видеть модификации данных в параллельной или незавершенной транзакции. Это означает, что для обеспечения изоляции для некоторых транзакций может потребоваться выполнить откат.

Свойство долговечности обеспечивает одно из наиболее важных требований баз данных: сохраняемость данных. Иными словами, эффект транзакции должен оставаться действенным даже в случае системной ошибки. По этой причине, если в процессе выполнения транзакции происходит системная ошибка, то осуществляется откат для всех выполненных инструкций этой транзакции.

Инструкции Transact-SQL и транзакции

Для работы с транзакциями язык Transact-SQL предоставляет некоторые инструкции. Инструкция BEGIN TRANSACTION запускает транзакцию. Синтаксис этой инструкции выглядит следующим образом:

```
BEGIN TRANSACTION [ {transaction_name | @trans_var }
   [WITH MARK ['description']]]
```

Соглашения по синтаксису

В параметре transaction_name указывается имя транзакции, которое можно использовать только в самой внешней паре вложенных инструкций BEGIN TRANSACTION/COMMIT или BEGIN TRANSACTION/ROLLBACK. В параметре @trans_var указывается имя определяемой пользователем

переменной, содержащей действительное имя транзакции. Параметр WITH MARK указывает, что транзакция должна быть отмечена в журнале. Аргумент description - это строка, описывающая эту отметку. В случае использования параметра WITH MARK требуется указать имя транзакции.

Инструкция BEGIN DISTRIBUTED TRANSACTION запускает распределенную транзакцию, которая управляется Microsoft Distributed Transaction Coordinator (MS DTC - координатором распределенных транзакций Microsoft). Распределенная транзакция - это транзакция, которая используется на нескольких базах данных и на нескольких серверах. Поэтому для таких транзакций требуется координатор для согласования выполнения инструкций на всех вовлеченных серверах. Координатором распределенной транзакции является сервер, запустивший инструкцию BEGIN DISTRIBUTED TRANSACTION, и поэтому он и управляет выполнением распределенной транзакции.

Инструкция COMMIT WORK успешно завершает транзакцию, запущенную инструкцией BEGIN TRANSACTION. Это означает, что все выполненные транзакцией изменения фиксируются и сохраняются на диск. Инструкция COMMIT WORK является стандартной формой этой инструкции. Использовать предложение WORK не обязательно.

Язык Transact-SQL также поддерживает инструкцию COMMIT TRANSACTION, которая функционально равнозначна инструкции COMMIT WORK, с той разницей, что она принимает определяемое пользователем имя транзакции. Инструкция COMMIT TRANSACTION является расширением языка Transact-SQL, соответствующим стандарту SQL.

В противоположность инструкции COMMIT WORK, инструкция ROOLBACK WORK сообщает о неуспешном выполнении транзакции. Программисты используют эту инструкцию, когда они полагают, что база данных может оказаться в несогласованном состоянии. В таком случае выполняется откат всех произведенных инструкциями транзакции изменений. Инструкция ROOLBACK WORK является стандартной формой этой инструкции. Использовать предложение WORK не обязательно. Язык Transact-SQL также поддерживает инструкцию ROLLBACK TRANSACTION, которая функционально равнозначна инструкции ROOLBACK WORK, с той разницей, что она принимает определяемое пользователем имя транзакции.

Инструкция SAVE TRANSACTION устанавливает точку сохранения внутри транзакции. Точка сохранения (savepoint) определяет заданную точку в транзакции, так что все последующие изменения данных могут быть отменены без отмены всей транзакции. (Для отмены всей транзакции применяется инструкция ROLLBACK.) Инструкция SAVE TRANSACTION в действительности не фиксирует никаких выполненных изменений данных. Она

только создает метку для последующей инструкции ROLLBACK, имеющей такую же метку, как и данная инструкция SAVE TRANSACTION.

Использование инструкции SAVE TRANSACTION показано в примере ниже:

```
USE TaxiService;
BEGIN TRANSACTION;
   INSERT INTO Region (region_id, region_name)
      VALUES (1, Minsk region');
   SAVE TRANSACTION a;
   INSERT INTO Region (region_id, region_name)
      VALUES (1, 'Gomel region');
   SAVE TRANSACTION b;
INSERT INTO Region (region_id, region_name)
      VALUES (1, Brest region');
   ROLLBACK TRANSACTION b;
   INSERT INTO Region (region_id, region_name)
      VALUES (1, Vitebsk region');
   ROLLBACK TRANSACTION a;
COMMIT TRANSACTION;
```

Единственной инструкцией, которая выполняется в этом примере, является первая инструкция INSERT. Для третьей инструкции INSERT выполняется откат с помощью инструкции ROLLBACK TRANSACTION b, а для двух других инструкций INSERT будет выполнен откат инструкцией ROLLBACK TRANSACTION a.

Инструкция SAVE TRANSACTION в сочетании с инструкцией IF или WHILE является полезной возможностью, позволяющей выполнять отдельные части всей транзакции. С другой стороны, использование этой инструкции противоречит принципу работы с базами данных, гласящему, что транзакция должна быть минимальной длины, поскольку длинные транзакции обычно уменьшают уровень доступности данных.

Каждая инструкция Transact-SQL всегда явно или неявно принадлежит к транзакции. Для удовлетворения требований стандарта SQL компонент Database Engine предоставляет поддержку неявных транзакций. Когда сеанс работает в режиме неявных транзакций, выполняемые инструкции неявно выдают инструкции BEGIN TRANSACTION. Это означает, что для того чтобы начать неявную транзакцию, пользователю или разработчику не требуется ничего делать. Но каждую неявную транзакцию нужно или явно зафиксировать, или явно отменить, используя инструкции COMMIT или ROLLBACK соответственно. Если транзакцию явно

не зафиксировать, то все изменения, выполненные в ней, откатываются при отключении пользователя.

Проблемы одновременного конкурентного доступа

Когда транзакции не изолированы друг от друга, могут возникнуть следующие проблемы:

Грязное чтение**.** Ситуации, когда одна транзакция считывает незафиксированный набор данных, над которыми работает другая транзакция. Это может вызвать проблемы, если другая транзакция завершится неудачно или откатится.

Неповторяющееся чтение**.** Ситуация, когда фрагмент данных, который считывается дважды в рамках одной транзакции, не может гарантировать, что он содержит одну и ту же информацию. Такое может возникнуть, если другая транзакция изменит запрашиваемые данные между этими считываниями.

Фантомное чтение. Случай, когда транзакция А вставляет или удаляет строку из набора данных, который транзакция В в настоящее время читает.

Пропущенное или двойное чтение. Случай, когда одна транзакция может выполнять сканирование диапазона в таблице, а другая транзакция может переместить строку так, чтобы первая транзакция прочитала ее дважды или пропустила.

Потеря обновления. Это может произойти, когда два процесса читают одни и те же данные, а затем оба пытаются обновить их одновременно, но с разными значениями. Только один из них добьется успеха, а другой пропадет [6].

Эффект Хэллоуина. Относится к ситуации, когда данные перемещаются в результирующем наборе и, следовательно, могут обновляться несколько раз.

Уровни изоляции транзакций

Уровни изоляции SQL Server используются для определения степени, в которой одна транзакция должна быть изолирована от ресурсов или изменений данных, сделанных другими параллельными транзакциями.

На каждом уровне используются разные подходы к принятию решения о том, какие блокировки используются при чтении данных и как долго удерживаются блокировки. Более низкие уровни изоляции увеличивают возможность доступа нескольких пользователей к одним и тем же данным, но, следовательно, они уменьшают и надежность. Уровни изоляции сосредоточены на блокировках, используемых при чтении, и не мешают блокировкам, полученным для защиты изменения данных.

Уровни изоляции могут быть установлены на уровне сервера, базы данных или транзакции. Изменить уровень изоляции в запросе можно следующей строкой в начале запроса:

SET TRANSACTION ISOLATION LEVEL <уровень_изоляции>

Различные уровни изоляции обычно делятся на две группы: одни описываются как пессимистические, а другие - как оптимистичные. Основное отличие состоит в том, что оптимистичные уровни пытаются уменьшить количество необходимых блокировок, но, как следствие, страдают от других накладных расходов, таких как увеличение использования системной базы данных tempdb, в которой хранятся временные таблицы. Оптимистические уровни используют управление версиями строк вместо блокировок. Здесь пойдет речь только о 4 пессимистичных уровнях.

READ UNCOMMITTED: запрос в текущей транзакции может считывать данные, измененные в другой транзакции, но еще не зафиксированные. Ядро базы данных не создает разделяемые блокировки при использовании этого уровня, что делает его наименее ограничивающим из уровней изоляции. В результате возможно, что оператор будет читать строки, которые были вставлены, обновлены или удалены, но никогда не зафиксированы в базе данных (грязное чтение). Также возможно, что данные будут изменены другой транзакцией между несколькими операторами чтения в рамках текущей транзакции, что приведет к неповторяющимся чтениям или фантомным чтениям.

Для демонстрации работы уровней изоляции создадим простую таблицу, содержащую ключ-идентификатор и два поля, и заполним её несколькими записями (как показано на рисунке Б20). Конструкция IF в начале кода позволит после каждой демонстрации сбрасывать таблицу в изначальное состояние [6].

Рисунок Б20 – Создание пустой таблице с ключом-идентификатором и двумя полями

В двух окнах SSMS подключимся к серверу, имитируя разные сеансы работы с базой данных, и будем запускать несколько сценариев транзакций:

1) Проверка на грязное чтение – попытка чтения ещё не закрепленных модификаций данных. В одном окне запустим транзакцию, которая будет

изменять некоторые данные в таблице (как показано на рисунке 2.21а). В этой транзакции между операциями изменения данных будут задержки WAITFOR DELAY по 10 секунд, в которые транзакция, которую мы запустим во втором окне сразу после старта первой транзакции, будет пытаться прочитать данные из этой таблицы (как показано на рисунке Б21).

2) Проверка на неповторяющееся/фантомное чтение – изменение данных, которые транзакция считывает несколько раз. В этом сценарии первой запускается транзакция с рисунка Б22. В промежутках задержки во втором окне мы будем выполнять те же модификации данных, что и в первом сценарии, только каждый оператор будет выполняться как отдельная транзакция, то есть, например, в первую задержку – оператор DELETE, во вторую – операторы INSERT и UPDATE (как показано на рисунке Б23).

Для выполнения отдельных строк, а не всей процедуры сразу, необходимо выделить нужный для выполнения код и нажать кнопку *Выполнить*. Ориентироваться по задержкам можно с помощью времени выполнения запроса, которое указывается в нижнем правом углу окна SSMS (как показано на рисунке Б24) [6].

```
BEGIN TRANSACTION
    DELETE FROM ExTable WHERE Id = 1
        WAITFOR DELAY '00:00:10'
    INSERT INTO ExTable(Name, Value) VALUES('Name4', 'Value4')
        WAITFOR DELAY '00:00:10'
    UPDATE ExTable SET Name = Name + Name WHERE Id = 2
COMMIT TRANSACTION
```

Рисунок Б21 – Транзакция для изменения некоторых данных в таблице

```
BEGIN TRAN
    SELECT * FROM ExTable
        WAITFOR DELAY '00:00:10'
    SELECT * FROM ExTable
        WAITFOR DELAY '00:00:10'
    SELECT * FROM ExTable
ROLLBACK
```

Рисунок Б22 – Транзакция с задержкой WAITFOR DELAY между операциями изменения данных

```
BEGIN TRAN
    DELETE FROM ExTable WHERE Id = 1
COMMIT TRAN

BEGIN TRAN
    INSERT INTO ExTable(Name, Value) VALUES('Name4', 'Value4')
COMMIT TRAN

BEGIN TRAN
    UPDATE ExTable SET Name = Name + Name WHERE Id = 2
COMMIT TRAN
```

Рисунок Б23 – Операторы INSERT и UPDATE

Рисунок Б24 – Время выполнения запроса

Выполним эти сценарии для уровня READ UNCOMMITTED:

Сценарий 1**.** Транзакция с уровнем изоляции READ UNCOMMITTED смогла прочитать все ещё не зафиксированные изменения данных (как показано на рисунке Б25). Следовательно, этот уровень допускает грязное чтение.

Сценарий 2**.** Другие транзакции смогли изменять данные таблицы, которую считывала транзакция с данным уровнем изоляции, в следствии чего произошли неповторяющееся и фантомное чтения (как показано на рисунке. Б26) [6].

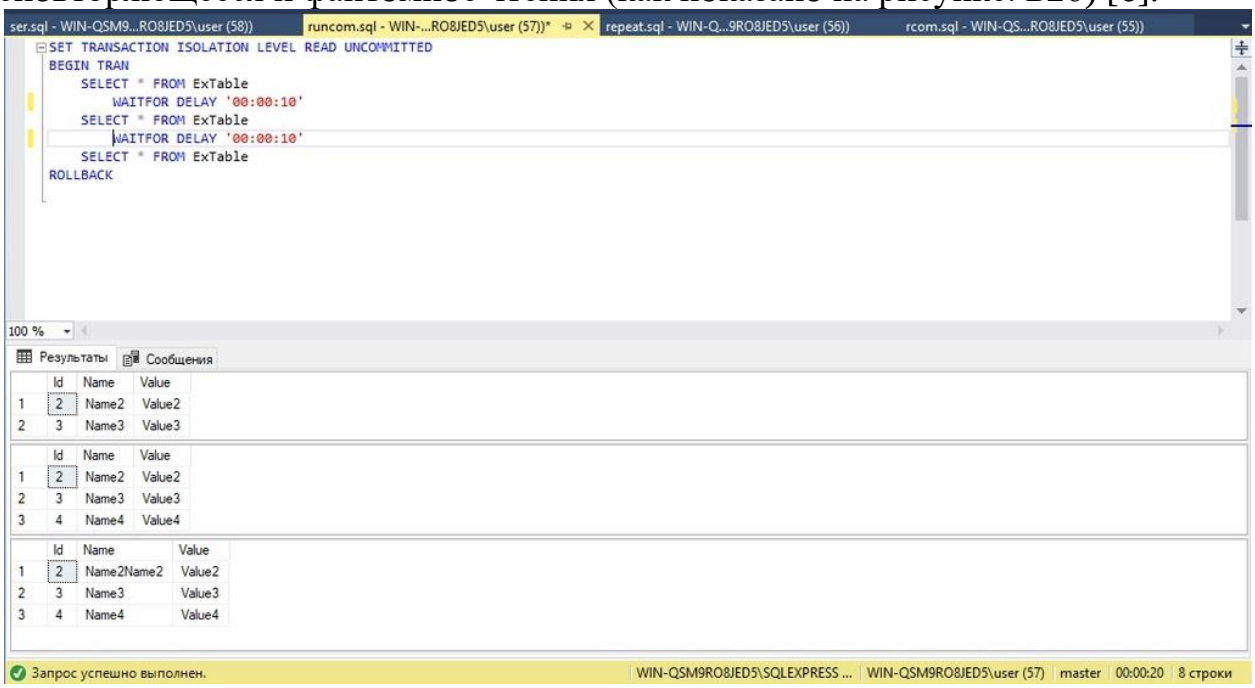

Рисунок Б25 – Транзакция с уровнем изоляции READ UNCOMMITTED смогла прочитать все ещё не зафиксированные изменения данных

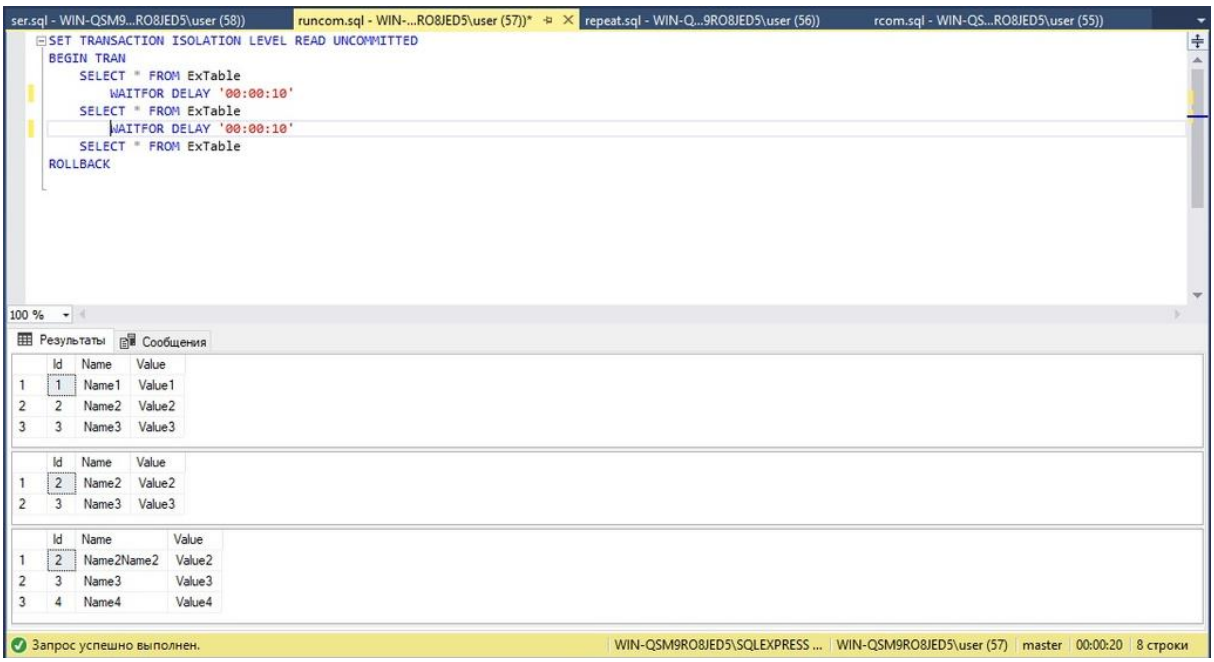

Рис Б26 – Транзакции смогли изменять данные таблицы, которую считывала транзакция с данным уровнем изоляции

READ COMMITTED: запрос в текущей транзакции не может читать данные, измененные другой транзакцией, которая еще не зафиксирована, что предотвращает грязное чтение. Однако данные могут быть изменены другими транзакциями между выдачей операторов в рамках текущей транзакции, поэтому неповторяющиеся чтения и фантомные чтения все еще возможны. Уровень изоляции использует разделяемую блокировку для предотвращения грязного чтения. READ COMMITTED – это уровень изоляции по умолчанию для всех баз данных SQL Server.

Теперь выполним сценарии для этого уровня:

Сценарий 1. Транзакция получила только конечные зафиксированные данные, так что грязное чтение здесь недопустимо (как показано на рисунке Б27). Кроме того, если обратить внимание на время выполнения транзакции, то можно заметить, что она выполнялась приблизительно в два раза больше времени всех задержек. Всё потому, что эта транзакция была заблокирована, и ожидала, пока другая транзакция закончит изменения и снимет блокировку с таблицы. Если в транзакции с чтением убрать все WAITFOR и оставить только один оператор SELECT, то время выполнения этой транзакции будет примерно равно времени выполнения транзакции, блокировавшей таблицу (как показано на рисунке Б28). Забегая вперед, скажем, что остальные уровни тоже не допускают грязное чтение, поэтому в дальнейшем для сценариев проверки на грязное чтение будет использоваться транзакция с одним оператором SELECT.

Сценарий 2. В этом сценарии всё точно так же, как и с уровнем READ UNCOMMITTED - другие транзакции смогли изменять данные таблицы,

которую считывала транзакция с данным уровнем изоляции. Так что этот уровень тоже допускает неповторяющееся и фантомное чтения. (как показано на рисунке Б29).

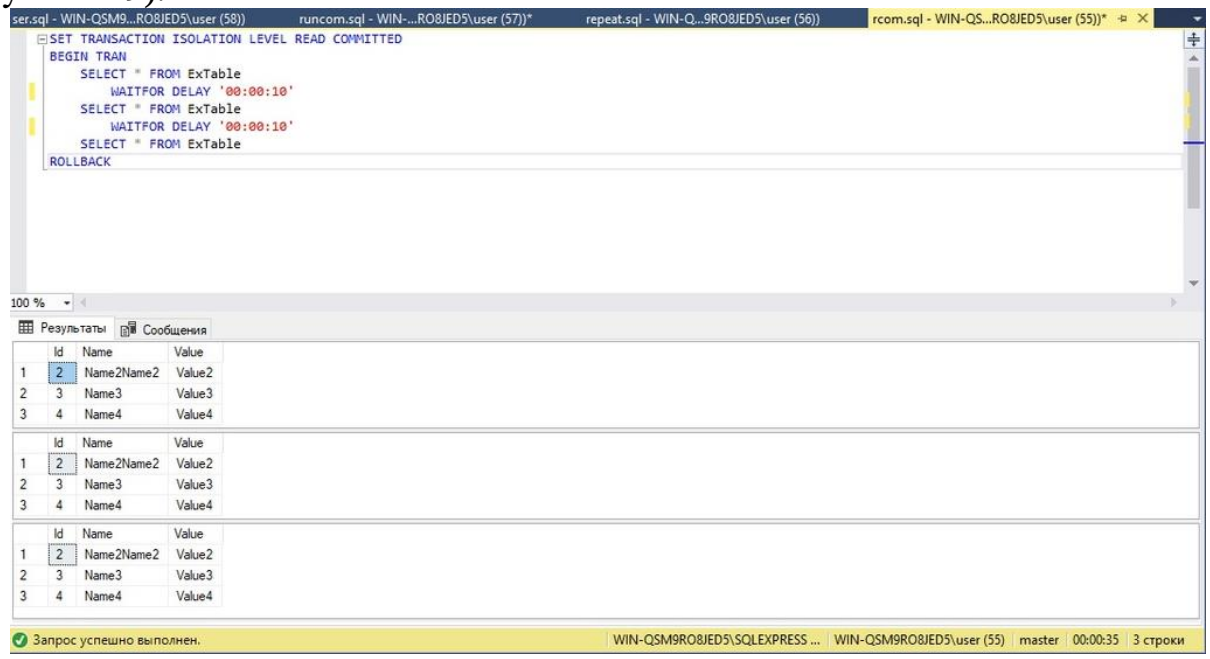

Рисунок. Б27 – Получение конечных зафиксированных данных

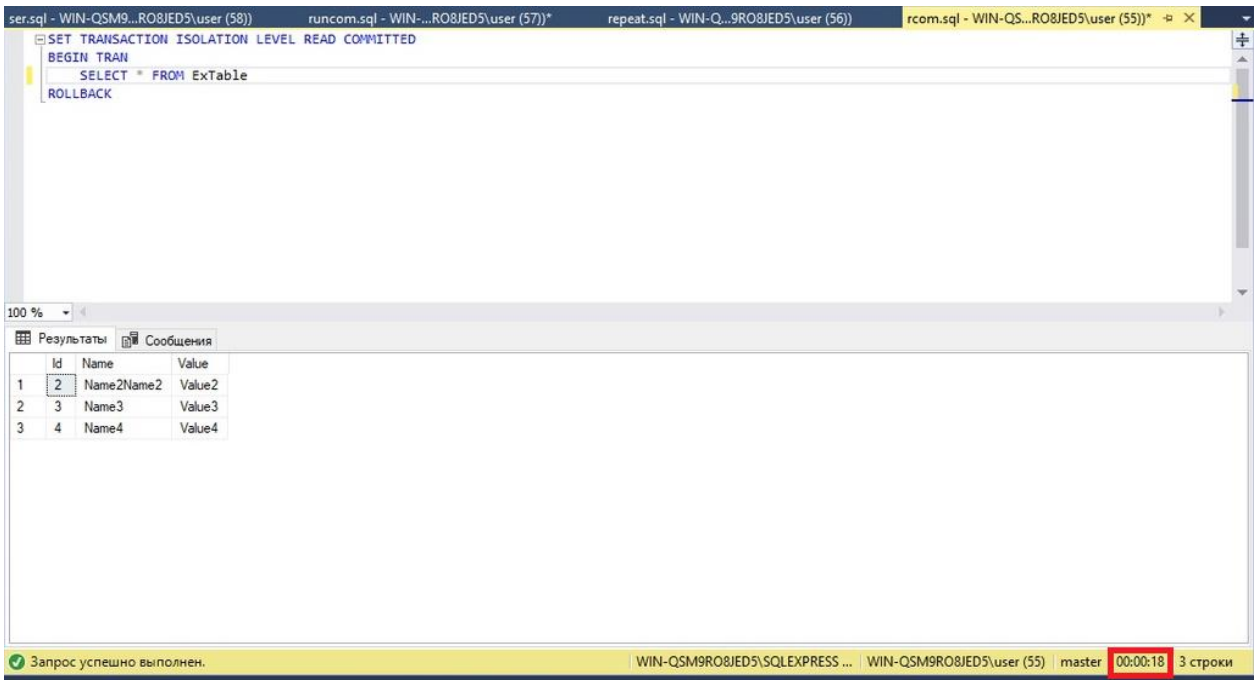

Рисунок Б28 – Время выполнения транзакции будет примерно равно времени выполнения транзакции, блокировавшей таблицу

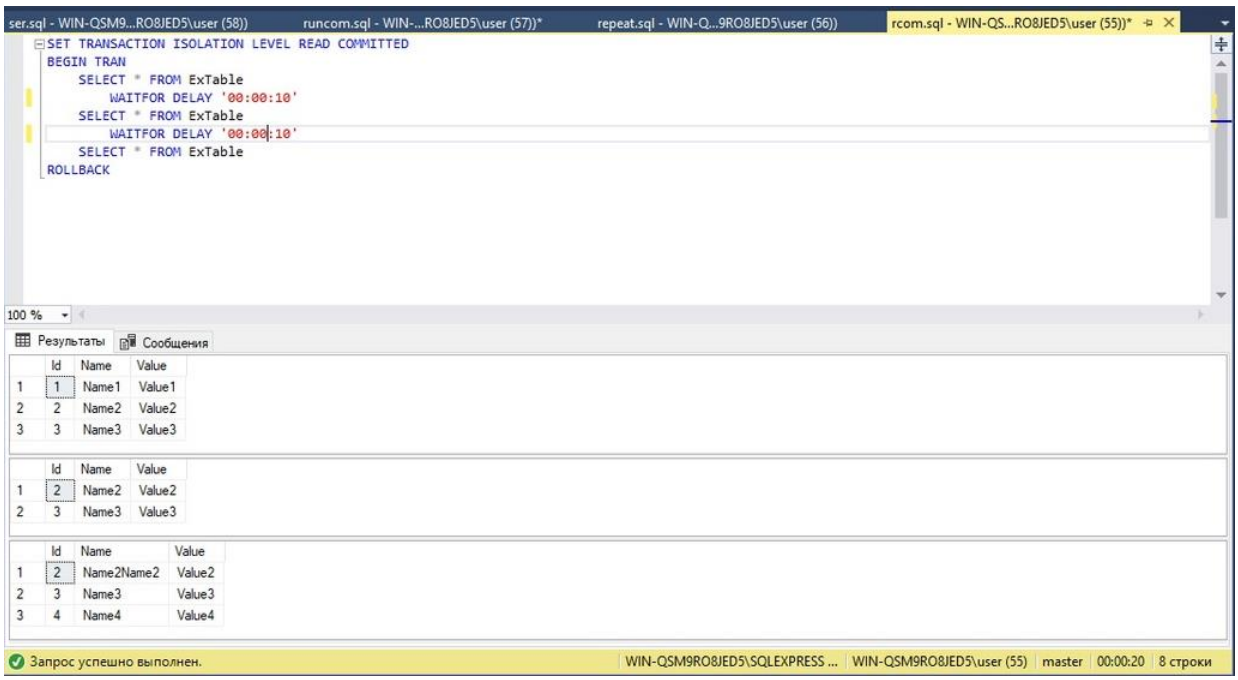

Рисунок Б29 – Уровень допускает неповторяющееся и фантомное чтения

REPEATABLE READ: запрос в текущей транзакции не может читать данные, измененные другой транзакцией, которая еще не зафиксирована, что предотвращает грязное чтение. Кроме того, никакие другие транзакции не могут изменять данные, считываемые текущей транзакцией, до ее завершения, что исключает неповторяющиеся чтения. Однако, если другая транзакция вставляет новые строки, которые соответствуют условию поиска в текущей транзакции, между текущей транзакцией, дважды обращающейся к одним и тем же данным, фантомные строки могут появиться во втором чтении.

Результаты сценариев для REPEATABLE READ:

Сценарий 1. Транзакция получила только конечные зафиксированные данные, так что грязное чтение здесь недопустимо (что показано на рисунке Б30).

Сценарий 2. В этом случае первым используем оператор INSERT, а следующим, допустим, DELETE. В итоге оператор INSERT выполниться, а следующий будет ожидать окончания транзакции REPEATABLE READ (что показано на рисунке Б31). Данный уровень позволяет добавить новые записи, но не изменять либо удалять их. Поэтому здесь пропадает возможность неповторяющихся чтений, но всё ещё остаются фантомные чтения.

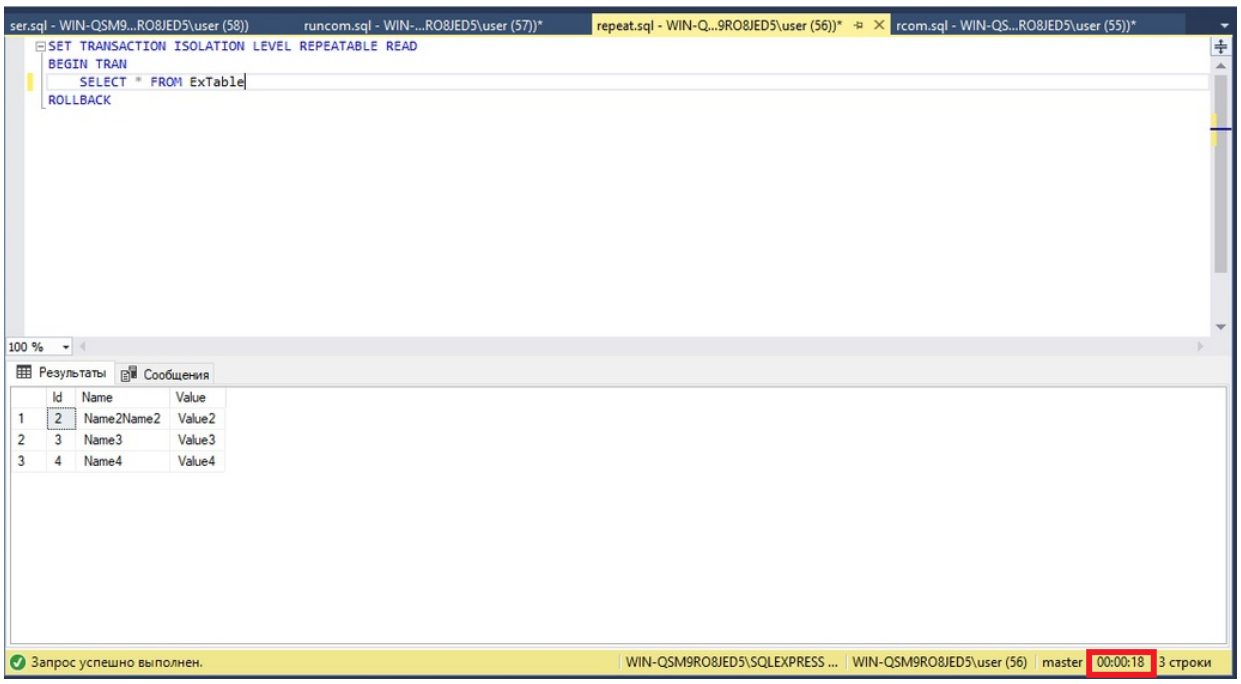

Рисунок Б30 – Получение транзакцией только зафиксированных данных

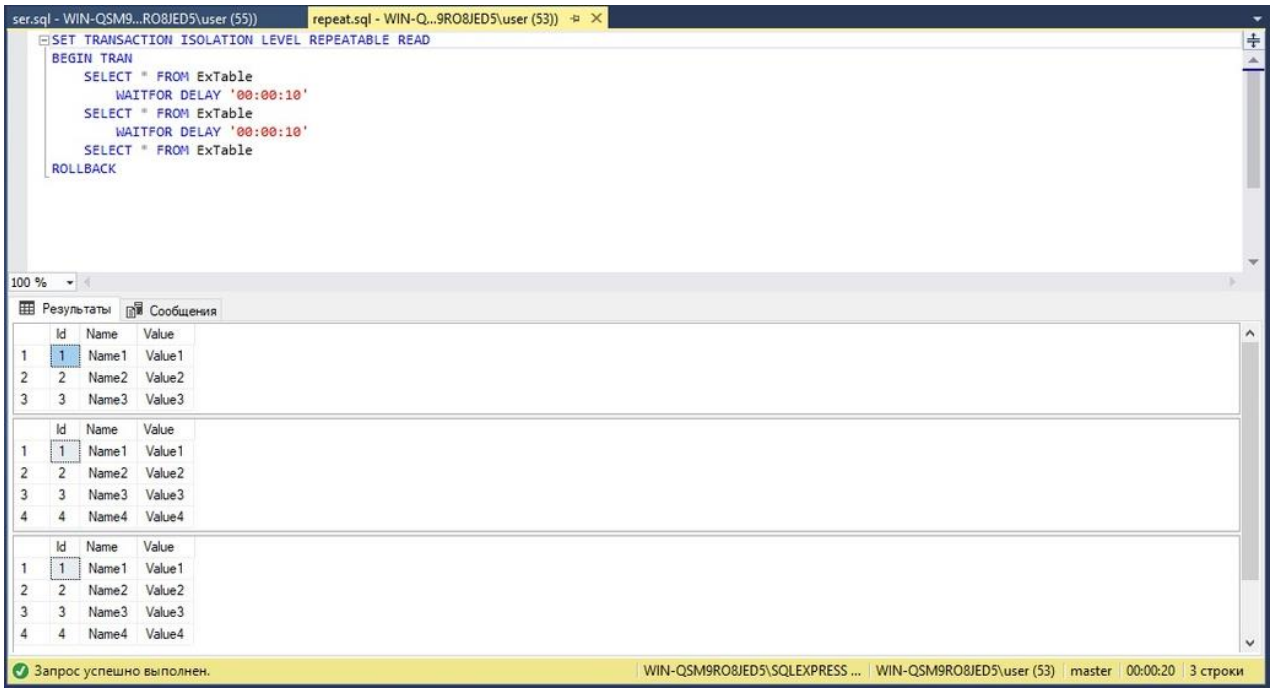

Рисунок Б31 – Невозможность неповторяющихся чтений

SERIALIZABLE: запрос в текущей транзакции не может прочитать данные, измененные другой транзакцией, которая еще не зафиксирована. Никакая другая транзакция не может изменять данные, считываемые текущей транзакцией, до ее завершения, и никакая другая транзакция не может вставлять новые строки, которые соответствовали бы условию поиска в текущей транзакции, пока она не завершится. В результате уровень изоляции Serializable предотвращает грязное чтение, неповторяющееся чтение и фантомное чтение. Однако он может иметь

наибольшее влияние на производительность по сравнению с другими уровнями изоляции.

Выполнение сценариев для этого уровня изоляции:

Сценарий 1. Транзакция получила только конечные зафиксированные данные, так что грязное чтение здесь недопустимо (что показано на рисунке Б32).

Сценарий 2. При использовании уровня SERIALIZABLE другим транзакциям запрещается изменять прочитанные первой данные, а также вставлять новые строчки в таблицу до конца транзакции. Поэтому любой из используемых в примере операторов будет ожидать окончания транзакции (что показано на рисунке Б33).

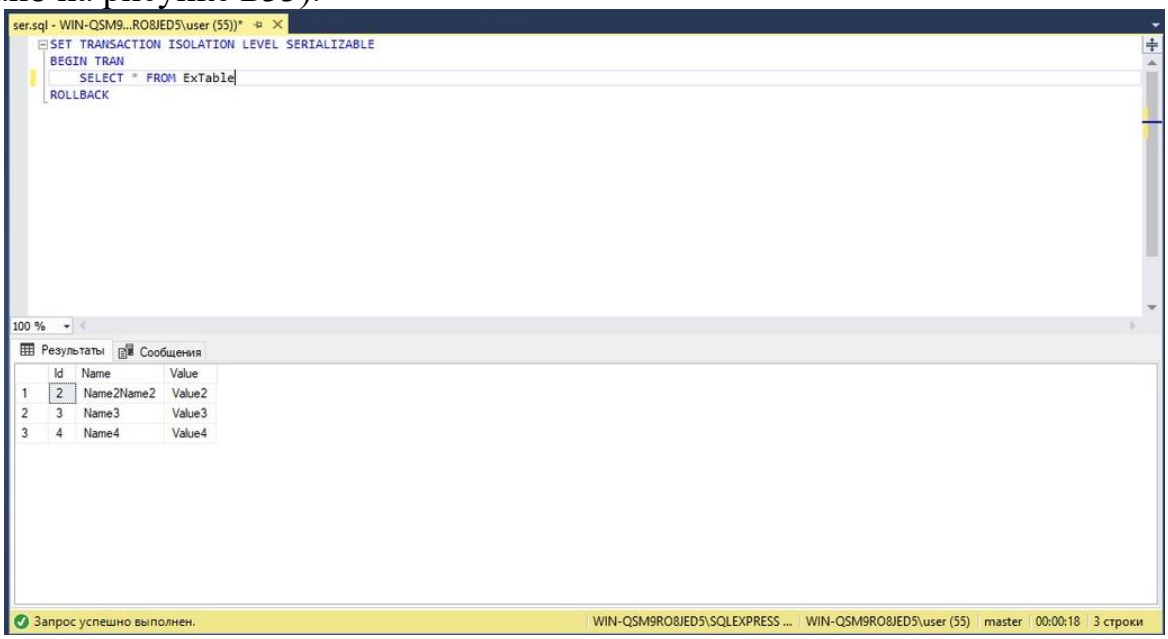

Рисунок Б32 – Использование уровня SERIALIZABLE

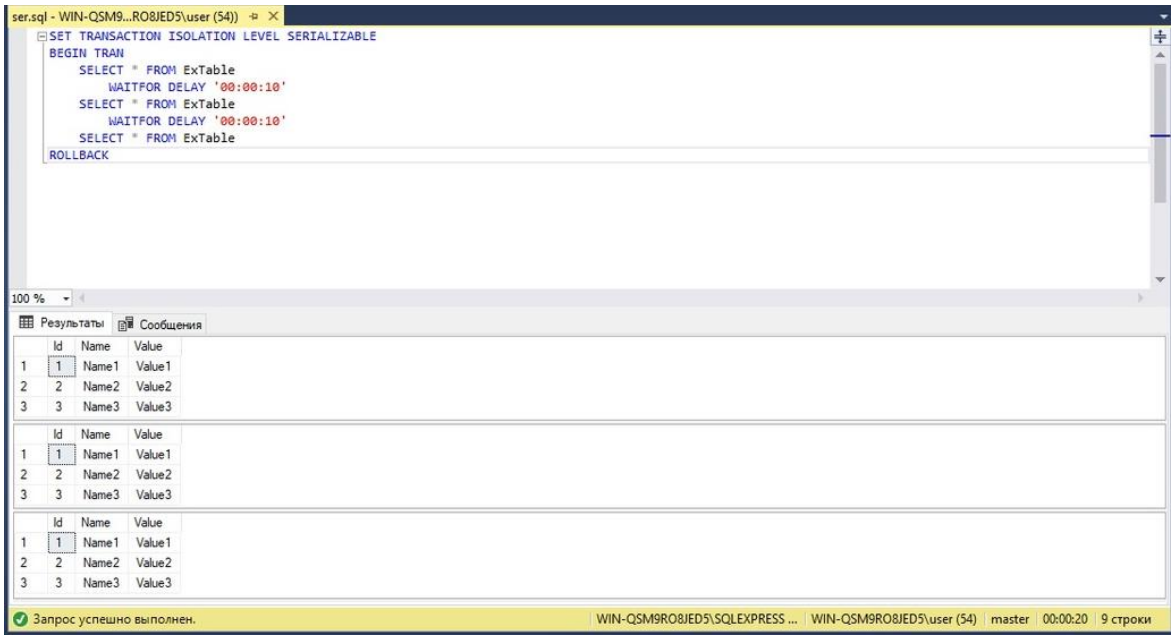

Рисунок Б33 – использование уровня SERIALIZABLE

Кроме указанных выше уровней, есть ещё уровень оптимистической модели – SNAPSHOT. Основное отличие – этот уровень не использует блокировки. Вместо этого, в начале транзакции создается моментальный снимок данных, с которым транзакция работает до своего окончания.

В завершении раздела обобщим возможные проблемы одновременного доступа для каждого уровня в виде таблицы Б1 (✓ - проблема допустима, ✗ - недопустима):

Таблица Б1 – Допускаемые проблемы доступа уровней изоляции транзакций.

Уровень изоляции	Грязное чтение	Неповторяющеюся	Фантомное
READ UNCOMMITED	✓	✓	✓
READ COMMITED	✗	✓	✓
REPEATABLE READ	✗	✗	✓
SERIALIZABLE	✗	✗	✗
SNAPSHOT	✗	✗	✗

Блокировки

Механизм блокировки необходим для успешной обработки транзакций SQL Server. Он был разработан, чтобы позволить SQL Server беспрепятственно работать в многопользовательской среде. Блокировка - это способ, которым SQL Server управляет параллелизмом транзакций. По сути, блокировки - это структуры в памяти, у которых есть владельцы, типы и хэш ресурса, который он должен защищать. Размер блокировки как структуры в памяти составляет 96 байт.

Важно понимать, что блокировка предназначена для обеспечения целостности данных в БД, поскольку она заставляет каждую транзакцию проходить тест ACID.

Блокировка SQL Server является неотъемлемой частью требований к изоляции и служит для блокировки объектов, затронутых транзакцией. Пока объекты заблокированы, SQL Server не позволит другим транзакциям вносить какие-либо изменения в данные, хранящиеся в объектах, на которые наложена блокировка. После снятия блокировки путем фиксации изменений или отката изменений до исходного состояния другим транзакциям будет разрешено вносить необходимые изменения данных.

Работа блокировок SQL Server определяется с помощью режимов блокировки уровню в иерархии, к которым они применяются.

Режимы блокировки

Режимы блокировки определяют различные типы запретов, которые могут применяться к ресурсу, который необходимо заблокировать:

Монопольная блокировка
Общий доступ
Обновление
Намерение
Схемы
Массовое обновление

Монопольная блокировка (X - exclusive) - при использовании гарантирует, что страница или строка будут зарезервированы исключительно для транзакции, наложившей монопольную блокировку, пока транзакция удерживает блокировку.

Монопольная блокировка накладывается транзакцией, когда она хочет изменить данные страницы или строки, например, в случае применения операторов DML DELETE, INSERT или UPDATE. Монопольная блокировка может быть наложена на страницу или строку только в том случае, если на ресурс не наложена другая общая или монопольная блокировка. Из этого можно сделать вывод, что на страницу или строку может быть наложена только одна монопольная блокировка, а после наложения никакая другая блокировка не может быть наложена на заблокированные ресурсы.

Общая блокировка (S - shared) – данный тип блокировки при наложении резервирует страницу или строку, которые будут доступны только для чтения, что означает, что любая другая транзакция не сможет изменить заблокированную запись, пока блокировка активна. Однако общая блокировка может быть наложена несколькими транзакциями одновременно над одной и той же страницей или строкой, и таким образом несколько транзакций могут совместно использовать возможность чтения данных, поскольку сам процесс чтения никоим образом не повлияет на фактические данные страницы или строки. Кроме того, эта блокировка разрешает операции записи, но не допускаются изменения DDL (подробнее об операторах DDL будет рассказано в главе про триггеры).

Обновление (U - update) – эта блокировка похожа на монопольную блокировку, но в некотором смысле более гибкая. Блокировка обновления может быть наложена на запись, которая уже имеет общую блокировку. В таком случае блокировка обновления накладывает другую общую блокировку на целевую строку. Как только транзакция, содержащая блокировку обновления, будет готова к изменению данных, блокировка обновления будет преобразована в монопольную блокировку. Важно понимать, что блокировка обновления асимметрична по отношению к общим блокировкам. В то время как блокировка обновления может быть наложена на запись с общей блокировкой, общая блокировка не может быть наложена на запись, которая уже имеет блокировку обновления.

Намерение (I - intent) – эта блокировка является средством, используемым транзакцией, чтобы сообщить другой транзакции о своем намерении получить блокировку. Цель такой блокировки - обеспечить правильное выполнение модификации данных, не допуская, чтобы другая транзакция установила блокировку следующего в иерархии объекта. На практике, когда транзакция хочет получить блокировку строки, она получает намеренную блокировку таблицы, которая является объектом более высокой иерархии. Получив блокировку намерения, транзакция не позволит другим транзакциям получить монопольную блокировку для этой таблицы, т.к. в противном случае монопольная блокировка, наложенная какой-либо другой транзакцией, отменит блокировку строки.

Это важный тип блокировки с точки зрения производительности, поскольку ядро базы данных будет проверять блокировки намерения только на уровне таблицы, уточняя, возможно ли для транзакции получить блокировку безопасным способом в этой таблице, и, следовательно, данный тип блокировки устраняет необходимость проверять каждую блокировку строки или страницы в таблице, чтобы убедиться, что транзакция может получить блокировку для всей таблицы.

Существует три обычных блокировки намерения и три блокировки преобразования:

Обычные блокировки с намерением:

Монопольная блокировка с намерением (IX - intent exclusive) - когда получена монопольная блокировка с намерением (IX), это указывает SQL Server, что транзакция имеет намерение изменить некоторые из ресурсов более низкой иерархии, приобретая монопольные блокировки (X) индивидуально для этих ресурсов более низкой иерархии [6].

Блокировка с намерением совмещаемого доступа (IS - intent shared) - этот тип блокировки указывает SQL Server, что транзакция имеет намерение прочитать некоторые ресурсы более низкой иерархии, приобретая общие блокировки (S) индивидуально для этих ресурсов более низкого уровня иерархии.

Блокировка с намерением обновления (IU - intent update) - блокировка намеренного обновления может быть получена только на уровне страницы, и как только операция обновления выполняется, она преобразуется в монопольную блокировку с намерением(IX).

Блокировки преобразования:

Общий доступ с монопольной блокировкой намерения (SIX -shared intent exclusive) - при установке эта блокировка указывает, что транзакция намеревается прочитать все ресурсы в более низкой иерархии и, таким образом, получить общую блокировку для всех ресурсов, которые находятся ниже в

иерархии, и, в свою очередь, изменить часть этих ресурсов, но не все. При этом он получит монопольную блокировку с намерением (IX) для тех ресурсов более низкой иерархии, которые должны быть изменены. На практике это означает, что как только транзакция получает данную блокировку таблицы, она приобретает монопольную блокировку с намерением (IX) на измененных страницах и монопольную блокировку (X) на измененных строках.

Только одна такая блокировка может быть получена для таблицы, и она будет блокировать другие транзакции от выполнения обновлений, но это не помешает другим транзакциям читать ресурсы более низкой иерархии.

Совмещаемая блокировка с намерением обновления (SIU - shared intent update) - это немного более конкретная блокировка, поскольку это комбинация блокировки общего (S) и обновления намерения (IU). Типичный пример этой блокировки - это когда транзакция использует запрос, выполняемый с ключевым словом PAGELOCK, а затем запрос на обновление. После того, как транзакция получит блокировку SIU для таблицы, запрос с подсказкой PAGELOCK получит общую (S) блокировку, в то время как запрос обновления получит блокировку намеренного обновления (IU).

Блокировка обновления с намерением монопольного доступа (UIX - update intent exclusive) - возникает, когда блокировка обновления (U) и блокировки намерения (IX) одновременно получены на ресурсах более низкой иерархии в таблице.

Блокировки схемы (Sch) - ядро базы данных SQL Server распознает два типа блокировок схемы: блокировка изменения схемы (Sch-M) и блокировка стабильности схемы (Sch-S).

Блокировка изменения схемы (Sch-M) будет получена при выполнении оператора DDL и предотвратит доступ к данным заблокированного объекта при изменении структуры объекта. SQL Server допускает единственную блокировку модификации схемы (Sch-M) для любого заблокированного объекта. Чтобы изменить таблицу, транзакция должна дождаться получения блокировки Sch-M на целевом объекте. После получения блокировки модификации схемы (Sch-M) транзакция может изменить объект, и после завершения модификации блокировка будет снята. Типичным примером блокировки Sch-M является перестроение индекса, а перестроение индекса - это процесс модификации таблицы. После выдачи идентификатора перестроения индекса для этой таблицы будет получена блокировка изменения схемы (Sch-M), которая будет снята только после завершения процесса перестроения индекса.

Блокировка стабильности схемы (Sch-S) будет получена во время компиляции и выполнения запроса, зависящего от схемы, и создания плана выполнения. Эта конкретная блокировка не будет блокировать другие транзакции для доступа к данным объекта и совместима со всеми режимами

блокировки, кроме блокировки модификации схемы (Sch-M). По сути, блокировки стабильности схемы будут приобретаться любым DML запросом или запросом выбора, чтобы гарантировать целостность структуры таблицы (гарантировать, что таблица не изменяется во время выполнения запросов).

Блокировки массового обновления (BU - bulk update) - эта блокировка предназначена для использования в операциях массового импорта, когда они запускаются с аргументом TABLOCK. Когда получена блокировка массового обновления, другие процессы не смогут получить доступ к таблице во время выполнения массовой загрузки. Однако блокировка массового обновления не препятствует параллельной обработке другой массовой загрузки.

Эскалация блокировки

Чтобы предотвратить ситуацию, когда при блокировке используется слишком много ресурсов, SQL Server имеет функцию эскалации (укрупнения) блокировки.

Эскалация блокировок позволяет исключить большую нагрузку на ресурсы памяти. Рассмотрим пример, в котором для выполнения операции удаления должна быть наложена блокировка на 30000 строк данных, каждая из которых имеет размер 500 байт. Без эскалации одна общая блокировка (S) будет наложена на базу данных, одна монопольная с намереннием (IX) на таблицу, 1875 монопольных с намерением (IX) на страницах (страница 8 КБ содержит 16 строк по 500 байтов, что составляет 1875 страниц, содержащих 30000 строк) и 30000 монопольных блокировок (X) на самих строках. Поскольку размер каждой блокировки составляет 96 байт, 31877 блокировок потребуют около 3 МБ памяти для одной операции удаления. Параллельное выполнение большого количества операций может потребовать значительных ресурсов только для того, чтобы менеджер блокировок мог выполнять операцию без задержек.

Чтобы предотвратить такую ситуацию, SQL Server использует эскалацию блокировок. Это означает, что в ситуации, когда на одном уровне установлено более 5000 блокировок, SQL Server превратит эти блокировки в одну блокировку на уровне таблицы. По умолчанию SQL Server всегда будет напрямую переходить на уровень таблицы, минуя переход на уровень страницы. Вместо получения блокировки множества строк и страниц, SQL Server перейдет к монопольной блокировке (X) на уровне таблицы.

Хотя это снизит потребность в ресурсах, монопольные блокировки (X) в таблице означают, что никакая другая транзакция не сможет получить доступ к заблокированной таблице, и все запросы, пытающиеся получить доступ к этой таблице, будут заблокированы. Следовательно, это снизит нагрузку на систему, но увеличит конкуренцию на доступ к данным.

Чтобы обеспечить контроль над эскалацией, начиная с SQL Server 2008 R2, параметр LOCK_ESCALATION вводится как часть оператора ALTER TABLE.

USE имя_базы_данных

GO

ALTER TABLE имя_таблицы

SET (LOCK_ESCALATION = <TABLE | AUTO | DISABLE> - один из этих параметров)

GO

Следующие параметры позволяют задать режим процесса эскалации блокировки:

TABLE - это параметр по умолчанию для любой созданной таблицы, так как по умолчанию SQL Server всегда выполняет эскалацию блокировки до уровня таблицы, который также включает секционированные таблицы.

AUTO - этот параметр позволяет эскалацию блокировки до уровня раздела, когда таблица разбита на разделы. Когда 5000 блокировок получены в одном разделе, при эскалации блокировки будет получена монопольная блокировка (X) на этом разделе, в то время как таблица получит монопольную блокировку с намерением (IX). В случае, если эта таблица не разделена на разделы, при эскалации блокировки будет установлена блокировка на уровне таблицы (как при параметре «TABLE»).

Хотя это выглядит очень полезным вариантом, его следует использовать очень осторожно, поскольку он может легко вызвать взаимоблокировки. В ситуации, когда у нас есть две транзакции в двух разделах, где получена монопольная блокировка (X), и транзакция пытается получить доступ к данным из раздела, используемого другой транзакцией, возникнет взаимная блокировка. Таким образом, очень важно тщательно контролировать шаблон доступа к данным, если этот параметр включен, и поэтому этот параметр не является настройкой по умолчанию в SQL Server.

DISABLE - этот параметр полностью отключает эскалацию блокировки для таблицы. Опять же, этот параметр следует использовать осторожно, чтобы избежать принудительного использования диспетчером блокировок SQL Server чрезмерного объема памяти.

Как видно, эскалация блокировок может стать проблемой для администраторов баз данных. Если работа приложения требует одновременного удаления или обновления более 5000 строк, решение, позволяющее избежать эскалации блокировок и связанных с этим эффектов, состоит в разделении одной транзакции на две или более транзакции, каждая из которых будет обрабатывать менее 5000 строк [6].

Получение информации об активных блокировках SQL Server

SQL Server имеет представление управления динамикой (DMV - Dynamics Management View) sys.dm_tran_locks, которое возвращает информацию о ресурсах диспетчера блокировок, которые используются в настоящее время, что

означает, что он отображает все действующие блокировки, полученные транзакциями.

Наиболее важными столбцами, используемыми для идентификации блокировки, являются resource_type, request_mode и resource_description. При необходимости во время устранения неполадок можно добавить больше столбцов в качестве дополнительных ресурсов для информации.

Пример запроса

SELECT resource_type, request_mode, resource_description.

FROM sys.dm_tran_locks

WHERE resource_type <> «БАЗА ДАННЫХ»

Фильтрация WHERE в этом запросе используется, чтобы исключить из результатов те общие блокировки, которые наложены на базу данных, поскольку они всегда присутствуют на уровне базы данных.

Описание запрошенных столбцов:

resource_type - отображает ресурс базы данных, в котором устанавливаются блокировки. Столбец может отображать одно из следующих значений: ALLOCATION_UNIT, APPLICATION, DATABASE, EXTENT, FILE, HOBT, METADATA, OBJECT, PAGE, KEY, RID.

request_mode - отображает режим блокировки, установленный на ресурсе

resource_description - отображает краткое описание ресурса и заполняется не для всех режимов блокировки. Чаще всего столбец содержит идентификатор строки, страницы, объекта, файла и т. п.

Практическая часть:

1. Осуществите резервное копирование с помощью плана обслуживания SQL Server.
2. Выполните полное резервное копирование с помощью T-SQL.
3. Сделайте дифференциальное резервное копирование с помощью T-SQL.
4. Выполните резервное копирование логов с помощью T-SQL.
5. Напишите транзакцию нулевого уровня на добавление информации в таблицу.
6. Запустите транзакцию из предыдущего задания, посмотрите на изменения, а затем завершите выполнение транзакции ее откатом.
7. Запустите транзакцию еще раз, а потом подтвердите её выполнение. Изучите измененные данные.
8. Напишите транзакции для каждого уровня изоляции.
9. Создайте транзакцию, которая будет создавать изменения в двух таблицах.

Контрольные вопросы:

1. Как можно совершить резервное копирование в SQL Server?
2. Какие виды резервного копирования вы знаете?
3. Назовите свойства транзакций, их операторы управления.
4. С чем связаны проблемы блокировки и взаимоблокировки транзакций?
5. Назовите режимы блокировок для данных.
6. За что отвечает параметр LOCK_TIMEOUT?
7. Какие проблемы одновременного конкурентного доступа вы знаете?
8. Какие уровни изоляции транзакции существуют для SQL SERVER, а также проблемы, которые каждый из них решает?
9. Какими свойствами характеризуются транзакции?
10. Перечислите операторы управления транзакциями.
11. Назовите режимы транзакций.
12. Что такое SQL-инъекция?
13. Какие виды SQL-инъекций вы знаете?
14. Назовите методы предотвращения SQL-инъекций, которые вы знаете.
15. Что хранит в себе таблица dm.sys_tran_locks?

ПРИЛОЖЕНИЕ В

Код программы для демонстрации SQL инъекций

```
using System;
using System.Data.SqlClient;
using System.Windows.Forms;

namespace SQL_injection
{
    public partial class QueryWindow : Form
    {
        // Server -  название экземпляра SQL Servera
        // Database - имя базы данных
        // Trusted_Connection - true - для аутентификации будет использоваться текущая учетная запись Windows
        private static readonly string _connectionString = @"Server = .\SQLEXPRESS;
                                                             Database = TaxiService;
                                                             Trusted_Connection = true";

        public QueryWindow()
        {
            InitializeComponent();
        }

        private void selectButton_Click(object sender, EventArgs e)
        {
            queryResults.Items.Clear();
            queryResults.CustomTabOffsets.Add(50);
            queryResults.UseCustomTabOffsets = true;

            using (SqlConnection connection = new SqlConnection(_connectionString))
            {
                try
                {
                    connection.Open();
                    //Запрос в базу данных, queryParamBox.Text - параметр, введенный в окне приложения
                    string query = $"SELECT first_name, last_name FROM Person WHERE person_id = {queryParamBox.Text}";
                    SqlCommand selectCommand = new SqlCommand(query, connection);
                    SqlDataReader reader = selectCommand.ExecuteReader();

                    string item;

                        while (reader.Read())
                        {
                            //reader;
                            item = String.Empty;
                            for (int i = 0; i < reader.FieldCount; i++)
                            {
                                item += reader.GetValue(i) + "\t";
                            }
                            this.queryResults.Items.Add(item);
                        }
                    }
                    catch(SqlException ex)
                    {
                        MessageBox.Show(ex.Message);
                    }
                }
            }
        }
    }
}
```