

МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное
учреждение высшего образования
«Ярославский государственный университет им. П.Г. Демидова»
Кафедра компьютерной безопасности и математических
методов обработки информации

Сдано на кафедру

« ____ » _____ 20 ____ г.

Заведующий кафедрой

д. ф.-м. н., профессор

_____ Дурнев В.Г.

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА
Реализация RLS (безопасность на уровне
строк) в реляционных базах данных
специальность 10.05.01 Компьютерная безопасность

Научный руководитель

(степень, звание)

(подпись)

(ФИО)

« ____ » _____ 20 ____ г.

Студент группы _____

(подпись)

(ФИО)

« ____ » _____ 20 ____ г.

Ярославль 2018 г.

Реферат

Объем 86 с., 4 гл., 16 рис., 6 источников, 5 прил.

RLS. FLS. Безопасность на уровне строк. Безопасность на уровне столбцов. Динамическая политика безопасности. CLR функции. Интерпретатор. Предикаты.

Объектом исследования являются гибкие политики безопасности на языке SQL в системах управления базами данных.

Цель работы - разработка приложения для просмотра и изменения данных на основе механизма гибкого разграничения прав на записи базы данных, на основе предикатов.

В процессе работы были изучены принципы создания политики безопасности в СУБД SQL SERVER 2016 как без использования встроенного механизма RLS, так и с ним.

В результате было написано Web приложение для работы сотрудников некоторого магазина. Основным достоинством данного приложения является возможность разграничивать доступ к строчкам базы данных на основе предикатов, написанных на специально разработанном для этой цели языке. Так как специально для данной задачи был написан язык, синтаксис которого является достаточно простым для конечных пользователей, то написание предикатов не представляет трудную задачу.

Содержание

| | |
|--|----|
| Введение..... | 4 |
| 1. Безопасность на уровне полей | 6 |
| 1.1 Описание | 6 |
| 1.2 Команды назначения прав на поля таблиц БД..... | 7 |
| 2. Безопасность на уровне строк..... | 9 |
| 2.1 Описание | 9 |
| 2.2 Особенности предикатов фильтров и блокировки | 11 |
| 2.3 Способы применения..... | 12 |
| 2.4 Разрешения | 13 |
| 2.5 Рекомендации по созданию безопасности на уровне строк | 13 |
| 2.6 Совместимость с разными компонентами..... | 15 |
| 3. Создание политики безопасности без встроенной поддержки RLS | 17 |
| 4. Создание политики безопасности с использованием RLS | 27 |
| 4.1 Подготовка базы данных для политик безопасности..... | 28 |
| 4.2 Описание CLR функции исполняющей предикаты | 33 |
| 4.3 Приложение для работы с политикой безопасности..... | 36 |
| 4.4 Пример создания политики безопасности..... | 40 |
| 4.5 Производительность | 44 |
| 4.6 Альтернативный вариант решения проблемы производительности.. | 52 |
| Заключение | 54 |
| Список литературы | 55 |
| Приложение А | 56 |
| Приложение Б..... | 58 |
| Приложение В..... | 63 |
| Приложение Г | 72 |
| Приложение Д..... | 79 |

Введение

В последнее время на рынке приложений для коммерческих организаций стало появляться всё больше и больше приложений для которых требуется умное разграничение прав - то есть необходимо иметь возможность создавать политики безопасности, которые бы имели механизм гибкой настройки прав и разрешений для конечных пользователей. Создание гибкой политики безопасности является довольно трудной задачей, так как обычно необходимо учесть очень много факторов. Зачастую это становится головной болью для разработчиков, так как требования к безопасности растут и механизм должен удовлетворять даже самым замысловатым требованиям.

В последнее время приложения необходимо устанавливать на многие платформы - мобильные устройства, планшеты, часы, настольные компьютеры и web приложения, поэтому возникает проблема выбора языка программирования на котором будет реализована политика безопасности. Сразу написать приложение под все платформы является ещё более трудной задачей на текущий момент из-за несовершенства программ исполняющих код на разных платформах.

Не так давно появился механизм для создания гибких политик безопасности на языке SQL во многих системах управления базами данных. Он позволяет реализовать политику безопасности на уровне сервера базы данных, причём приложение, работающее с данным, предоставляемыми сервером БД, не знает ни о какой политике безопасности. Таким способом можно изолировать код приложения от определения и реализации механизма разграничений прав на записи в таблицах.

Объект исследования - гибкие политики безопасности на языке SQL в системах управления базами данных.

Предмет исследования - возможность реализации механизма гибкого разграничения прав на записи базы данных, на основе предикатов.

Цель выпускной квалификационной работы - разработка приложения для просмотра и изменения данных на основе механизма гибкого разграничения прав на записи базы данных, на основе предикатов.

Для достижения цели выпускной квалификационной работы были поставлены следующие задачи:

- Изучить принципы создания политики безопасности в СУБД SQL SERVER 2016
- Изучить способы создания политики безопасности как без использования встроенного механизма RLS, так и с ним

- Спроектировать базу данных, на которую будет накладываться политики безопасности
- Написать приложение для просмотра и изменения данных подготовленной базы данных
- Реализовать механизм гибкого разграничения прав на записи базы данных, на основе предикатов
- Измерить производительность и практичность данного механизма

1. Безопасность на уровне полей

1.1 Описание

Разграничение доступа к данным обычно необходимо в том случае, когда пользователи какой-либо системы имеют разный уровень доступа к данным, то есть неравноправны по своему статусу, обязанностям возложенными на них или информация, хранимая в базе данных, является коммерческой тайной и корпоративная политика фирмы не предусматривает доступ к ней для любого сотрудника.

Рассматривая механизм контроля доступа к данным, можно выделить два основных подхода:

- FLS - Field Level Security - контроль безопасности на уровне полей (столбцы в базе данных)
- RLS - Row Level Security - контроль безопасности на уровне строк (отдельных записей)

Первый подход подразумевает использование встроенных функций базы данных, позволяющих контролировать для некоторых пользователей возможность выбрать определённые столбцы из базы данных. То есть можно наложить запрет на выполнение запросов, в которых явно или неявно участвуют поля сущности базы данных, на которые наложены ограничения. Данный подход достаточно прост в реализации и присутствует во всех коммерческих системах СУБД. Обычно разработчики обращаются непосредственно к нему, так как не составляет особо труда реализовать политику безопасности, основываясь на данном подходе.

Второй подход, он же самый сложный, предусматривает использование, как встроенных средств СУБД, так и сторонних процедур, и методов для обеспечения контроля на уровне строк базы данных. Этот путь предполагает более детальную настройку ограничений прав доступа к записям. В отличие от FLS, данный подход должен работать с часто изменяющимися сущностями БД, так как ограничение прав на выборку конкретной записи из некоторой таблицы зависит от многих факторов (кто исполняет запрос, в каком контексте и т.д.).

Подход с использованием безопасности на уровне полей чаще всего подходит для приложений, в которых политика безопасности в основном не изменяется на протяжении цикла работы приложения. Данный способ установки правил ограничения доступа пользователей к БД является самым простым и одновременно самым трудоёмким с точки зрения потраченного времени администратора базы данных. До того, как конечные пользователи (напрямую или через приложение) начнут работать с базой данных, необходимо создать:

- пользователей базы данных с логином и паролем для входа
- роли и присвоить их созданным пользователям
- каждой роли раздать разрешения на выборку, создание, удаления, исполнение и т.д.

Данный подход требует очень внимательной настройки разрешений, так как очень легко допустить типичную ошибку - запретить доступ к элементарным объектам, но выдать доступ к объекту, через которого можно получить доступ элементарным объектам, на которые был наложен явный запрет.

1.2 Команды назначения прав на поля таблиц БД

Для раздачи ограничений на поля таблицы в основном используются три команды - GRANT, DENY, REVOKE.

С помощью команды GRANT можно предоставить разрешение на таблицу, представление, табличную функцию и т.д. Упрощённый синтаксис команды для предоставления разрешений на колонки таблицы для одной или нескольких ролей.

GRANT <разрешение или список разрешений> ON

[Имя схемы].[Имя объекта] [(колонка или список колонок)]

TO <роль или несколько ролей>

Пример предоставления роли User разрешения на выборку полей Phone и Email таблицы Employees:

GRANT SELECT ON dbo.Employee (Email, Phone) TO User;

С помощью команды REVOKE можно отклонить разрешение на таблицу, представление, табличную функцию и т.д. Синтаксис команды для отклонения разрешений на колонки таблицы для одной или нескольких ролей.

REVOKE <разрешение или список разрешений> ON

[Имя схемы].[Имя объекта] [(колонка или список колонок)]

FROM <роль или несколько ролей>

Пример отклонения разрешения на выборку полей Phone и Email таблицы Employees роли User:

REVOKE SELECT ON dbo.Employee (Email, Phone) FROM User;

С помощью команды DENY можно запретить разрешение на таблицу, представление, табличную функцию и т.д. Синтаксис команды для запрещения разрешений на колонки таблицы для одной или нескольких ролей.

DENY <разрешение или список разрешений> ON

[Имя схемы].[Имя объекта] [(колонка или список колонок)]

TO <роль или несколько ролей>

Пример запрещения разрешения на выборку полей Phone и Email таблицы Employees роли User:

DENY SELECT ON dbo.Employee (Email, Phone) FROM User;

2. Безопасность на уровне строк

2.1 Описание

Для создания системы безопасности, ограничивающей пользователей в работе с данными базы данных разработчики часто прибегают к созданию различного вида фреймворков на уровне приложения для создания и поддержания политик безопасности, ограничивающих доступ пользователей к строкам, фильтруя записи как до, так и после послания запроса к БД. Это подразумевает создание большой кодовой базы и возможность использовать данный подход только на ограниченном количестве устройств из-за языка программирования, на котором и написан данный фреймворк. Но кроме ограничения выбора конкретного языка программирования и платформ, на которых это будет работать, есть также другие минусы – производительность (нет оптимизации запросов и возможно часть данных будет фильтроваться в оперативной памяти), незащищённость базы данных (возможно, что кто-то сможет обойти уровень приложения и послать запрос напрямую к БД, обойдя тем самым все ограничения политики безопасности), целостность данных (нельзя быть уверенным, что при обращении к базе данных и получении строк, будут задействованы те же правила ограничения прав доступа, хотя программист может администратор базы данных может ошибиться с прописыванием правил на уровне БД.

RLS - технология позволяющая пользователям управлять доступом к строкам в таблице базы данных в зависимости от характеристик пользователя, выполняющего запрос (например, членство или контекст выполнения). Также RLS упрощает проектирование и кодирование безопасности в приложении. RLS позволяет реализовать ограничения на доступ к строкам данных, не затрагивая код приложения. Например, обеспечивать сотрудникам доступ только к тем строкам данных, которые имеют отношение к их отделу, или ограничивать некоторым сотрудникам доступ к строчкам с конфиденциальной информацией.

Логика ограничения находится на уровне базы данных, а не на отдалении от данных на другом уровне приложения. Система базы данных применяет ограничения доступа каждый раз, когда выполняется попытка доступа к данным с любого уровня. Это делает систему безопасности более надёжной и устойчивой за счет уменьшения контактной зоны системы безопасности.

Основным элементом ограничения доступа в этом механизме является предикат, срабатывающий на получение или изменение данных. RLS поддерживает два типа предикатов безопасности:

- Предикаты фильтров автоматически фильтруют строки, доступные для операций чтения (SELECT, UPDATE и DELETE)
- Предикаты BLOCK явно блокируют операции записи (AFTER INSERT, AFTER UPDATE, BEFORE UPDATE, BEFORE DELETE), которые нарушают предикат

Предикаты фильтров применяются при считывании данных из базовой таблицы, и это влияет на все операции получения данных: SELECT, DELETE (т. е. пользователь не может удалять отфильтрованные строки) и UPDATE (т. е. пользователь не может обновить строки, которые фильтруются, хотя и существует возможность обновления строк таким образом, что они будут фильтроваться впоследствии). Предикаты блокировки влияют на все операции записи.

Предикаты AFTER INSERT и AFTER UPDATE могут блокировать обновление строк значениями, нарушающими предикат.

Предикаты BEFORE UPDATE могут блокировать обновление строк, нарушающих предикат на данный момент.

Предикаты BEFORE DELETE могут блокировать операции удаления.

Политики безопасности имеют следующие особенности:

- Можно определить функцию предиката, которая делает JOIN с другой таблицей или вызывает стороннюю функцию. Если политика безопасности создана с использованием команды SCHEMABINDING = ON, тогда команда JOIN или сторонняя функция доступны из запроса и работают должным образом без каких-либо дополнительных проверок разрешений. Если политика безопасности создана с использованием SCHEMABINDING = OFF, то для отправки запросов в целевую таблицу пользователям потребуются разрешения SELECT и EXECUTE в этих дополнительных таблицах и функциях.
- Также можно выполнить запрос к таблице, имеющей предикат безопасности, который определен, но отключен. В этом случае все строки, которые были бы отфильтрованы или заблокированы, не затрагиваются.
- Когда пользователь схемы dbo, член роли db_owner или владелец таблицы выполняет запрос к таблице, для которой определена или включена политика безопасности, строки фильтруются или блокируются в соответствии с такой политикой безопасности.

- Попытка изменить схему таблицы, на которую привязана политика безопасности, приведет к ошибке. Тем не менее, можно изменить столбцы, на которые не ссылается предикат.
- При попытке добавить предикат на таблицу, которая уже имеет один определенный предикат для данной операции в пределах политики безопасности (независимо от того, включен он или выключен), произойдет ошибка
- Попытка изменить функцию, которая используется в качестве предиката, наложенного на таблицу в пределах политики безопасности, приведет к ошибке.
- Определение нескольких активных политик безопасности, содержащих неперекрывающиеся предикаты, завершается успешно.

2.2 Особенности предикатов фильтров и блокировки

- При определении политики безопасности, которая фильтрует строки таблицы, приложение не знает, что какие-то строки были отфильтрованы для операций SELECT, UPDATE и DELETE, включая те ситуации, когда все строки будут исключены. Приложение может вызывать операцию INSERT, которая вставит любые строки независимо от того, будут ли они отфильтрованы во время любой другой операции. Приложение, в конечном счёте, ничего не знает о том, что произошло с данными на уровне сервера - то есть полная изолированность политики безопасности от действий самого приложения.
- Предикаты блокировки для операций UPDATE в свою очередь разбиваются на отдельные операции - BEFORE и AFTER. Следовательно, нельзя, например, запретить пользователям обновлять строки значением, больших текущего на момент обновления. В таком случае необходимо использовать триггеры с промежуточными таблицами DELETED и INSERTED, чтобы ссылаться как на новые, так и на старые значения.
- Оптимизатор не будет проверять предикат блокировки AFTER UPDATE, если не изменяется ни один из столбцов, используемых функцией предиката. Пример: Алисе запрещено изменять значение заработной платы, указывая сумму более 100 000, однако она должна иметь возможность изменить адрес сотрудника, зарплата которого уже больше 100 000 (и, таким образом, уже нарушает предикат).
- Политики безопасности также работают для пакетных API, в том числе для BULK INSERT. Таким образом, предикаты

блокировки AFTER INSERT будут применяться к операциям пакетной вставки так же, как к обычным операциям вставки.

2.3 Способы применения

Приведём примеры использования RLS.

- В больнице требуется установить политику безопасности, которая позволяет медсестрам просматривать строки данных только их собственных пациентов. Заведующий отделением может просматривать и изменять данные только тех пациентов, которые принадлежат их отделению. Управляющий всей больницей может просматривать и изменять данные, как пациентов, так и сотрудников.
- Администратору базы данных банка необходимо установить ограничение на доступ к строкам финансовых данных на основе бизнес-подразделения сотрудника либо на основе роли сотрудника в компании. Возможность увидеть данные или изменить их может зависеть как от отдела, в котором работает сотрудник, так и от уровня доступа, присвоенному ранее этому сотруднику.
- Мультиотенантное приложение может создать политику для обеспечения логического разделения строк каждого клиента от строк любых других клиентов. Эффективность достигается путем хранения данных для многих клиентов в одной таблице. Конечно, каждый клиент можно видеть только свои строки данных. Каждый менеджер может изменять данные только своих сотрудников. Диспетчер политики безопасности может изменять политики безопасности, но не имеет возможности посмотреть или изменить данные клиентов.

Можно сказать, что предикаты фильтров RLS функционально эквивалентны добавлению предложения WHERE к результирующему запросу. Предикат может по сложности сравниваться с определением деловой практики или предложение может быть простым как WHERE City = "Yaroslavl" (предикат проверки на то, что для данного контекста и строки необходимо выполнение условия - город должен быть Ярославлем).

При использовании более формальных терминов можно сказать, что механизм RLS представляет управление доступом на основе предиката. Он поддерживает гибкую, централизованную оценку на основе предиката, которая может учитывать метаданные или другие критерии, определяемые администратором по своему усмотрению. Предикат используется как критерий для определения того, имеет ли текущий пользователь соответствующий доступ к данным на основе атрибутов пользователя.

2.4 Разрешения

Создание, изменение или удаление политик безопасности требует разрешения ALTER ANY SECURITY POLICY. Создание или удаление политики безопасности требует разрешения ALTER для схемы. Обычно, разрешение ALTER ANY SECURITY POLICY предназначено для пользователей с высокими привилегиями (например, для диспетчера политики безопасности). Но в то же самое время, диспетчеру политики безопасности вовсе не обязательно выдавать разрешение SELECT для таблиц, которые защищаются политиками безопасности.

Также, для каждого добавляемого предиката требуются следующие разрешения:

- Разрешения SELECT и REFERENCES для функции используемой в качестве предиката.
- Разрешение REFERENCES для целевой таблицы, которая привязывается к политике безопасности.
- Разрешение REFERENCES для каждого столбца из целевой таблицы, используемого в качестве аргументов функции предиката.

Особенностью политик безопасности является их применимость ко всем пользователям базы данных, включая пользователей схемы dbo в базе данных. Пользователи схемы dbo могут изменять или удалять политики безопасности, однако можно проводить аудит этих изменений в политиках безопасности. Если привилегированным пользователям (например, sysadmin или db_owner) нужно видеть все строки для устранения неполадок или проверки данных, необходимо написать политику безопасности, разрешающую эти действия.

2.5 Рекомендации по созданию безопасности на уровне строк

Для того чтобы избежать ошибок или снижения производительности при применении политики безопасности необходимо придерживаться следующих правил:

- Для лучшего конфигурирования настроек базы данных и распределения прав доступа на объекты необходимо создать отдельную схему для объектов RLS
- Необходимо выдавать минимальный набор прав для администратора политик безопасности, лучше всего чтобы у него не было возможности просматривать или изменять

данные - только минимальных назначение прав и просмотр колонок

- Необходимо избегать конвертации данных в другие типы в функции предиката. Чем больше преобразований, тем больше вероятность ошибки во время выполнения
- Необходимо следить за выражением, которое строит пользователь во время выборки из базы данных, так как он может создать условия для утечки информации. Например, пользователю разрешается вставлять вместо имени столбца какое-то математическое выражение - $1/(\text{SALARY}-100000)$. Таким образом, подставляя значения в знаменатель, можно узнать уровень дохода, так как при правильном числе появиться ошибка деления на 0.
- Необходимо добавить аудит операций, происходящих в базе данных, особенно - операций изменения предикатов, встроенных функций, участвующих в вычислении предиката и политик безопасности. Это необходимо, так как злонамеренный диспетчер политики безопасности может войти в сговор с конечным пользователем, и тогда утечки информации будет не избежать
- Избегать рекурсии в функции предиката, так как это может серьёзно снизить производительность. Оптимизатор запроса, возможно, сможет выявить рекурсии и трансформировать запрос, но лучше не полагаться на эту функциональность
- Избегать чрезмерного количества соединений таблиц в функции предиката, так как это тоже может снизить производительность. Если же избежать этого не получается, то необходимо создать индексы на таблицах, которые будут участвовать в соединении. Также можно наложить политику безопасности не на саму таблицу, а на VIEW, созданное на соединении нескольких таблиц - таким образом можно улучшить производительность запроса
- Избегать создания предикатов, которые могут зависеть от SET параметров sql сервера, так как это может привести к утечке информации и произвольным результатам. Как правило, функции предикатов должны подчиняться следующим правилам:
 - Функции предикатов не стоит создавать таким образом, чтобы в них происходила неявная конвертация в такие типы данных как - smalldatetime, date, datetime, datetime2 или datetimeoffset (и наоборот), так как на эти преобразования влияют SET параметры DATEFORMAT и SET LANGUAGE. Вместо этого лучше использовать функцию CONVERT и явно задать тип преобразования

- Функции предикатов не должны зависеть от параметра `SET DATEFIRST`, то есть значения первого дня недели
- Функции предикатов не должны зависеть от арифметических или агрегатных выражений, возвращающих значение `NULL` если произошла ошибка (например, когда происходит переполнении или делении на ноль), так как это поведение определяется параметрами `SET ANSI_WARNINGS`, `SET NUMERIC_ROUNDABORT` и `SET ARITHABORT`
- Функции предикатов не должны сравнивать сцепленные строки с параметром `NULL`, так как это поведение определяется параметром `SET CONCAT_NULL_YIELDS_NULL`

2.6 Совместимость с разными компонентами

Как правило, безопасность на уровне строк должна работать в разных компонентах и при любых условиях. Однако это не так, существуют несколько исключений:

- `DBCC SHOW_STATISTICS` предоставляет статистику по нефiltroванным данным, таким образом, он может вызвать утечку информации, которая в то же самое время защищена политикой безопасности. Таким образом, чтобы иметь возможность просматривать объект статистики для таблицы, к которой применяется безопасность на уровне строк, пользователь должен быть владельцем таблицы либо членом предопределенной роли сервера `sysadmin`, предопределенной роли базы данных `db_owner` или `db_ddladmin`
- Безопасности на уровне строк для оптимизируемых таблиц работает так же, как и для обычных таблиц, за исключением того, что встроенные функции с табличным значением, используемые в качестве предикатов безопасности, должны быть скомпилированы в собственном коде (созданы с помощью параметра `WITH NATIVE_COMPILATION`)
- Как правило, на основе представлений также можно создавать политики безопасности, а представления можно создавать на основе таблиц, связанных политиками безопасности. Тем не менее, нельзя создать индексированные представления на основе таблиц с политикой безопасности, так как операции поиска строк через индекс будут обходить политику
- Система отслеживания измененных данных может вызвать утечку целых строк, которые должны быть отфильтрованы,

предоставляя доступ членам `db_owner` или пользователям, являющимся членами "шлюзовой" роли, указанной при включении этой системы для таблицы. В результате члены такой шлюзовой роли могут просматривать все изменения данных в таблице даже при наличии политики безопасности для таблицы

- Также может вызвать утечку и функция отслеживания изменений, но утечку только первичного ключа строк, которые должны быть отфильтрованы, предоставляя доступ пользователям с разрешениями `SELECT` и `VIEW CHANGE TRACKING`. Доступ к фактическим значениям данных не предоставляется, становится известно только то, что столбец `A` был обновлен (вставлен или удален) для строки с первичным ключом `B`. Это создает проблему, если первичный ключ содержит конфиденциальные элементы, например, номер социального страхования. Тем не менее, на практике для получения последних данных инструкция `CHANGETABLE` почти всегда объединена с исходной таблицей
- Можно прогнозировать снижение производительности для запросов, использующих следующие функции полнотекстового и семантического поиска, из-за введения дополнительного соединения для применения безопасности на уровне строк и блокирования утечки первичных ключей строк, которые должны быть отфильтрованы: `CONTAINSTABLE`, `FREETEXTTABLE`, `semantickeyphrasetable`, `semanticsimilaritydetailstable`, `semanticsimilaritytable`
- Безопасность на уровне строк совместима как с кластеризованными, так и с некластеризованными индексами `columnstore`. Тем не менее, поскольку безопасность на уровне строк применяет функцию, оптимизатор может изменить план запроса таким образом, чтобы пакетный режим не использовался
- Предикаты блокировки нельзя определить в секционированных представлениях, и секционированные представления нельзя создавать на основе таблиц, использующих предикаты блокировки. Предикаты фильтров совместимы с секционированными представлениями
- Временные таблицы также совместимы с безопасностью на уровне строк. Тем не менее, предикаты безопасности в текущей таблице не реплицируются автоматически в прежнюю таблицу. Чтобы применить политику безопасности для текущей и прежней таблиц, необходимо по отдельности добавить предикат безопасности в каждую таблицу

3. Создание политики безопасности без встроенной поддержки RLS

В общем случае нам необходимо ограничить действия определенного пользователя (или группы пользователей), когда он осуществляет какую-либо операцию получения или изменения данных. Для этого необходимо вычислять значение некоторого предиката перед выполнением операций над каждой строкой таблицы. Пользуясь стандартными средствами, реализованными в большинстве СУБД, достаточно легко вычислять предикаты такого рода при определенных действиях над данными – выборке, удалении, изменении. Это реализуется это с помощью триггеров и представлений.

Нам необходимо создавать и где-то хранить предикаты. Также нам нужно вовремя их вызывать на определённые действия. Для этого можно применять триггеры – прописывать предикаты прямо в них и тогда сервер базы данных будет выполнять их при совершении определённых действий с базой и таблицами. Такой подход позволит нам легко запретить добавление/удаление/изменение данных какой-либо таблицы. Однако он не совсем гибок, так как запрос от пользователя к базе либо будет выполнен в исходном виде, если он удовлетворяет предикату, либо не будет выполнен вообще, если он не удовлетворяет предикату. Часто требуется скорректировать результат выполнения запроса перед выполнением, если он подпадает под действие предиката, а не просто выполнять или отклонять его полностью. В этом случае, исходя из синтаксиса языка запросов SQL логично было бы осуществлять проверку предикатов в выражении `where` (модифицируя или создавая его непосредственно перед запросом), а сами предикаты в этом случае представлять в виде хранимых процедур.

Рассмотрим, как механизм безопасности на уровне строк может быть реализован на паре примеров.

Например, необходимо выполнить запрос `SELECT` к таблице документов:

```
select * from documents where documentName like 'Report_%'
```

И чтобы проверить, что он действительно удовлетворяет требованиям политики безопасности, его необходимо модифицировать следующим образом:

```
select * from documents where documentName like 'Report_%' AND  
<Predicate>
```

В данном случае под `<Predicate>` понимается какой-то предикат, то есть булево выражение, применимое к каждой строке таблицы `documents` (хотя конечно там может стоять и более простое и примитивное

выражение, которое либо даст выполниться запросу целиком, либо вовсе не допустит его выполнение). Можно отметить, что если бы выражение where отсутствовало, то мы бы его добавили.

Но возникает проблема – нам необходимо изолировать пользователя от прямого доступа к данным и гарантировать применение установленных нами правил безопасности. И если СУБД не предоставляет встроенных механизмов обеспечения безопасности, то получить корректное и полное решение данной задачи нам не удастся (при реализации RLS в рамках описываемого метода естественно). В частности, необходимо уметь запрещать пользователю самому редактировать и создавать триггеры и встроенные процедуры, чтобы он не мог нарушить политику безопасности, внедренную администратором, а также давать ему выполнять запросы напрямую к таблицам базы данных, минуя тем самым создаваемые при помощи триггеров, процедур и содержащихся в них предикатов представления.

Основой вычисления предиката безопасности теоретически может являться идентификатор текущего пользователя (он, как правило, доступен в любой СУБД, поддерживающей аутентификацию). Однако его прямое использование не рекомендуется, поскольку корпоративная политика, в соответствии с которой обычно строится реализация системы, редко имеет дело и регламентирует действия конкретных людей. Часто бывает затруднительно сформулировать относительно стабильные правила, которые не придется пересматривать при каждом изменении списка сотрудников компании.

Обычно все правила доступа в компании создаются на основе должностей. В программировании их принято ассоциировать с группами или ролями. В связи с этим в предикатах безопасности часто придется использовать выражения типа `UserMatchRole(rolename)`. Если в используемую СУБД изначально встроена подобная функциональность, то лучше всего использовать именно ее. В таком случае субъекты безопасности будут образовывать единое пространство, как для встроенной системы безопасности СУБД, так и для наших расширений. Впрочем, при желании все это может быть реализовано и самостоятельно. Одним из наиболее очевидных способов является создание специальной таблицы, содержащей список групп или ролей, и связь ее с таблицей пользователей.

Схемы могут меняться в зависимости от конкретных потребностей. Если пользователь может входить только в одну группу, то достаточно просто добавить ссылку на нее в таблицу пользователей. Если же пользователь может выполнять одновременно несколько ролей, то придется организовать связь многие-ко-многим посредством создания отдельной таблицы. В этом случае предикат `UserMatchRole(rolename)`

может иметь, например следующий вид (мы предполагаем, что в таблице users базы securityinfodb содержатся идентификаторы пользователей, а функции CurrentUserID(), RoleName() возвращают идентификатор текущего пользователя и его роль):

```
exists(select * from securityinfodb.users  
where ID = CurrentUserID() and user_group = rolename)
```

или например такой:

```
exists(select * from securityinfodb.UserRoles  
where RoleName = RoleName() and UserID = CurrentUserID())
```

Рассмотрим случай, когда правила корпоративной политики безопасности компании выражаются в терминах предметной области, т.е. можно сформировать соответствующий предикат безопасности непосредственно в терминах данных, хранимых в СУБД без привлечения сторонней информации. В самом простом случае нам будет достаточно данных из той же таблицы, которая является объектом запроса пользователя (рассматриваем простой запрос, который затрагивает ровно одну таблицу).

Предположим, что у нас имеется таблица, содержащая документы по еженедельной финансовой отчетности компании, и есть 3 роли пользователей (младшие финансовые аналитики, старшие финансовые аналитики и «все остальные»). Пусть доступ к финансовым отчетам определяется следующими правилами:

- младшие финансовые сотрудники должны иметь право чтения отчетов старше 12 недель;
- старшие финансовые сотрудники должны иметь право чтения отчетов старше 4 недель;
- все остальные сотрудники доступ к отчетам иметь не должны в принципе.

В таком случае можно построить предикат следующего вида:

```
(UserMatchRole('senior_employee') and ReportDate < DateAdd(Day,  
GetDate(), 4*7))
```

OR

```
(UserMatchRole('junior_employee') and ReportDate < DateAdd(Day,  
GetDate(), 12*7))
```

Но теоретически, тот же самый результат можно получить и другим способом. Например, так:

```
case  
  
  when ReportDate < DateAdd(Day, 4*7, GetDate()) then false  
  
  when ReportDate > DateAdd(Day, 4*7, GetDate()) and ReportDate <  
    DateAdd(Day, 12*7, GetDate()) then  
  
    UserMatchRole('senior_employee')  
  
  when ReportDate > DateAdd(Day, 12*7, GetDate()) then  
  
    UserMatchRole('junior_employee')  
  
end
```

В этом случае логику проследить довольно сложно. Поэтому лучше, или во всяком случае нагляднее для диспетчера политики безопасности, строить предикаты в следующем виде:

(UserMatchRole(<role1>) AND <role_1_restrictions>)

OR

(UserMatchRole(<role2>) AND <role_2_restrictions>)

OR

...

(UserMatchRole(<role_n>) AND <role_n_restrictions>)

где:

- role_(1,2,3, ...n) - это роль, хранящаяся в базе данных
- role_(1,2,3, ...n)_restrictions - булевы условия или предикаты для конкретной роли

В данном случае для каждой группы пользователей (или роли) необходимо добавить список предикатов, соединённых логическими операторами. При осуществлении выборки СУБД будет идти по одному из условий, в зависимости от роли, в которой находится текущий пользователь, и проверять, что оно выполнено в текущем контексте и для текущей записи. Можно сказать, что это пессимистичный режим предиката - когда мы запрещаем доступ сразу всем пользователям и разрешаем только ограниченной части.

Но часто бывает удобно описывать предикаты в "оптимистичном" режиме, т.е. ограничить доступ к данным только некоторым ограниченное множеством пользователей. Тогда предикат может выглядеть примерно так:

```

NOT
(
  (UserMatchRole(<role_1_restricted>)                                [AND
<role_1_restricted_restrictions>])
  OR ...
  (UserMatchRole(<role_n_restricted                                >)      [AND
<role_n_restricted_restrictions>])
)

```

где

- role_(1,2,3, ...n)_restricted - роль, для которой наложены ограничения
- role_(1,2,3, ...n)_ restricted_restrictions - булевы условия или предикаты для "ограниченной" роли

В этом случае доступ предоставляется сразу всем пользователям, за исключением тех, которым назначены роли и указаны ограничения в предикатах.

Соединив 2 этих подхода (оптимистичный и пессимистичный) с преобладание пессимистического, мы получаем более гибкий механизм ограничения доступа к данным:

```

(-- секция доступа
(UserMatchRole(<role_1>) AND <role_1_restrictions>)
OR ...
(UserMatchRole(<role_n>) AND <role_n_restrictions>)
)
AND NOT
(-- секция запрета
(UserMatchRole(<role_1_restricted>)                                [AND
<role_1_restricted_restrictions>])
OR ...

```

```

(UserMatchRole(<role_n_restricted>)
<role_n_restricted_restrictions >])
[AND
)

```

Приведём несколько примеров того как можно использовать наш механизм ограничений доступа к данным на примере демонстрационной базы MS SQL server - northwind.

В самом простом случае предикаты для всех ролей зависят только от значений полей защищаемой записи. Рассмотрим таблицу Orders (несущественные для рассматриваемой задачи ограничения мы опустили):

```

TABLE Orders {
  OrderID int IDENTITY(1, 1) NOT NULL ,
  CustomerID nchar(5) NULL ,
  EmployeeID int NULL ,
  OrderDate datetime NULL ,
  RequiredDate datetime NULL ,
  ShippedDate datetime NULL ,
  ShipVia int NULL ,
  Freight money NULL CONSTRAINT DF_Orders_Freight DEFAULT(0),
  ShipName nvarchar(40) NULL ,
  ShipAddress nvarchar(60) NULL ,
  ShipCity nvarchar(15) NULL ,
  ShipRegion nvarchar(15) NULL ,
  ShipPostalCode nvarchar(10) NULL ,
  ShipCountry nvarchar(15) NULL ,
  CONSTRAINT FK_Orders_Employees FOREIGN KEY (EmployeeID)
  REFERENCES Employees (EmployeeID)
}

```

Предположим, что правила корпоративной политики безопасности по отношению к данным заказов, хранящихся в таблице Orders, определены следующим образом:

- Менеджеры по продажам (введем для них роль ‘Sales Representative’) имеют право просматривать только свои заказы (которые они ввели) и не могут видеть заказы, созданные другими менеджерами по продажам.
- Директор по продажам (его роль назовем ‘Vice President, Sales’) имеет право просматривать любые заказы.
- Все остальные сотрудники доступа к заказам не имеют никакого.

Создадим представление, которое соответствует этим правилам:

```
CREATE VIEW [Secure Orders] AS
```

```
SELECT * FROM Orders where
```

```
(UserMatchRole('Sales Representative') AND EmployeeID =
CurrentEmployeeID())
```

```
OR
```

```
(UserMatchRole('Vice President, Sales') AND TRUE)
```

Здесь мы подразумеваем, что функция CurrentEmployeeID() каким-либо образом возвращает нам идентификатор сотрудника, соответствующий пользователю, от имени которого было произведено подключение к базе данных. Реализация этой функции так же, как и функции UserMatchRole(), зависит от используемой СУБД. Следует обратить внимание на вторую часть предиката, участвующего в определении представления: для директора по продажам никаких дополнительных ограничений не предусмотрено, но для представления этого факта было использовано выражение AND TRUE. При ручном создании предиката фрагмент AND TRUE можно опустить, хотя оптимизаторы, используемые в большинстве современных СУБД, достаточно интеллектуальны, чтобы выбросить избыточные выражения из плана запроса уже на этапе выполнения.

Теперь предположим, что руководство компании решило, что доступ к заказам, отгруженным более восьми календарных месяцев назад, можно предоставить всем сотрудникам рассматриваемой компании. В этом случае предикат легко может быть переписан в таком виде:

```
(UserMatchRole('Sales Representative') AND EmployeeID =
CurrentEmployeeID())
```

```
OR
```

```
(UserMatchRole('Vice President, Sales') AND TRUE)
```

```
OR
```

```
(UserMatchRole('Everyone')      AND      Orders.ShippedDate      <
DateAdd(month, -8, GetDate()))
```

В приведенном выше предикате в дополнительном условии мы проверяем принадлежность текущего пользователя к группе Everyone, чтобы предикат полностью соответствовал введенному выше общему шаблону. Эта специальная группа по определению включает всех сотрудников, и выражение UserMatchRole('Everyone') должно являться тождественно истинным (следует это учитывать при написании кода указанной функции). Однако в целях оптимизации эту проверку можно отключить. Также отметим, что для ускорения запроса не применяется никакой функции для сравнения даты отгрузки заказа с текущей датой. Это сделано из-за того, что оптимизатор СУБД в противном случае, вероятно не смог бы использовать индекс по полю ShippedDate, если конечно этот индекс присутствует.

В корпоративную политику безопасности компании могут входить и более сложные правила, связывающие различные сущности предметной области между собой. Предположим, к примеру, что компания Northwind расширилась, и в ней появилось несколько филиалов (пусть идентификаторы этих филиалов (DivisionID) содержатся в отдельной таблице). Структура таблицы сотрудников в этом случае претерпит соответствующие изменения:

```
ALTER TABLE [Employee]
ADD [DivisionID] int CONSTRAINT [DivisionID_FK]
REFERENCES [Division]([DivisionID])
```

В таком случае новый вариант одного из указанных выше правил доступа может выглядеть следующим образом:

Менеджеры по продажам (используем старую роль - 'Sales Representative') имеют право просматривать только заказы, введенные менеджерами из того же филиала (а не только ими лично).

Если мы примем это изменение, то соответствующая часть предиката безопасности будет выглядеть, например, так:

```
(UserMatchRole('Sales Representative')
AND
(select DivisionID from Employees where EmployeeID =
CurrentEmployeeID()))
= (select DivisionID from Employees where EmployeeID = Orders.
EmployeeID)
```


Рассмотрим еще один пример правил безопасности, который требует обращения к другим таблицам. Он связан с защитой подчиненных таблиц. Пусть вместе с записями в таблице заказов необходимо защитить также и записи в таблице деталей заказов (Order Details). Применим правила из предыдущего примера (тот их вариант, где мы еще не ввели филиалы) к таблице Order Details и в итоге получим выражение, похожее на нижеследующее:

```
(UserMatchRole('Sales Representative')
AND
(select EmployeeID from Orders where Orders.OrderID = OrderID) =
CurrentEmployeeID())
OR
(UserMatchRole('Vice President, Sales') AND TRUE)
OR
(UserMatchRole('Everyone')
AND
(select ShippedDate from Orders where Orders.OrderID = [Order
Details].OrderID) <
DateAdd(month, -6, GetDate()))
```

В принципе можно поступить и иначе. В частности мы можем переписать правила, путем создания нового представления:

```
create view [Secure Order Details] as select od.* from [Order Details] od
join [Secure Orders] so on od.OrderID = so.OrderID
```

В таком виде сущность используемого ограничения безопасности прозрачна. Кроме того, изменение правил безопасности для заказов, которое повлияет на определение представления Secure Orders, автоматически отразится и на деталях заказов (Secure Order Details).

Отметим, что в данном случае мы не накладываем никаких дополнительных ограничений на детали заказа. Однако при необходимости мы можем точно так же добавить локальный предикат безопасности в условие where...

Рассмотренные приемы позволяют обеспечить разделение прав доступа в терминах значений защищаемых данных. Во многих случаях этот способ является наиболее удобным, и его главное преимущество – малые усилия по административной поддержке. Модификации потребуются только при изменении корпоративной политики, что, как правило, достаточно редкое явление для большинства компаний. При этом, нам также не требуется динамического управления доступом на уровне отдельных объектов – например, заказы, отгруженные сегодня,

автоматически станут доступными всем сотрудникам для просмотра через 8 месяцев и таким образом не требуется заботиться об открытии доступа к ним по прошествии определенного времени.

Однако в некоторых случаях требуется предоставлять или отказывать в доступе к конкретным записям в административном порядке, независимо от хранящихся в них значений. Такие требования могут быть связаны со стремительно меняющимися правилами безопасности, для которых недопустимы задержки в реализации, неизбежные при модификации схемы базы данных. Кроме того, иногда быстрое действие СУБД может оказаться недостаточным для вычисления предикатов безопасности на основе уже существующих атрибутов (особенно если таблицы большие и индексов мало).

4. Создание политики безопасности с использованием RLS

Ключевые слова:

- Web-приложение SPA - веб-приложение, использующее единственный HTML документ, с подгружаемыми JavaScript файлами и CSS. Обычно при работе с данным типом приложения не происходит обновление страницы, так как все операции получения и отправки данных происходят посредством AJAX запросов
- REST (Representational State Transfer) API (application programming interface) сервис - сервер, написанный в специальном архитектурном стиле взаимодействия компонентов для построения распределенных масштабируемых веб-сервисов
- Entity Framework - объектно-ориентированная технология доступа к данным. Предоставляет возможность работать с сущностями базы данных через объекты языка C#. Улучшает производительность и позволяет упростить работу с базой данных на уровне приложения
- ASP.NET - технология создания веб-сервисов и веб-приложений от компании Microsoft, разработанная как часть платформы Microsoft .NET
- CLR (Common Language Runtime) - исполняющая среда для байт-кода CIL, в который компилируются программы, написанные на .NET-совместимых языках программирования (C#, Managed C++, Visual Basic .NET, F# и прочие). CLR является одним из основных компонентов пакета Microsoft .NET Framework
- Web-API 2 платформа - платформа основанная на технологии ASP.NET, позволяя с лёгкостью создавать HTTP службы для широкого диапазона клиентских приложений. В том числе имеет удобный механизм для создания и детальной настройки контроллеров в архитектуре REST

В последней версии SQL SERVER появилась встроенная поддержка безопасности на уровне строк. Теперь SQL SERVER 2016 имеет ряд встроенных механизмов для создания политик безопасности и привязки к ним предикатов. Настройка не предоставляет особого труда, так как она основывается на тех же самых предикатах, которые будут участвовать во время формирования запроса. Нет необходимости создавать представления и триггеры для изменения запроса, чтобы добавить к нему определённый предикат. Механизм RLS СУБД SQL SERVER автоматически добавляет условие, определённое как предикат безопасности.

Реализация самого предиката практически не отличается от предиката, используемого в случае отсутствия поддержки RLS в СУБД. Единственная разница - необходимо чтобы функция предиката возвращала табличное значение с булевым флагом доступа. Создание политики безопасности также не предоставляет особо труда - главное привязать определённый предикат к таблице, указав режим его срабатывания.

4.1 Подготовка базы данных для политик безопасности

Для использования механизма RLS в самом простом случае необходимо создать предикат, прикрепленный к таблице и политику безопасности, указывающую на предикат. Использование одного статического предиката с одной активной политикой безопасности часто представляется невозможным, если необходимо часто менять требования и условия, определяющие доступ к конкретной строке базы данных. К сожалению нельзя создать больше одного активного предиката для одной таблицы.

Под "активным предикатом" понимается предикат, использующийся при текущей конфигурации политик безопасности базы данных. Соответственно нельзя создать несколько активных политик безопасности для одной таблицы. Единственным решением для динамического добавления политик безопасности является создание предикатов, делегирующих свою работу некоторой сторонней функции, которая не является табличной функцией. В табличную функцию невозможно добавить условия, что является сильным ограничением для данной задачи.

Данная не табличная функция должна возвращать true или false (есть доступ или нет), в зависимости от данных текущей строки, для которой выполняется проверка. Мы могли бы передавать данные всех столбцов из таблицы к которой привязана эта функция чтобы потом выполнить некие действия, которые привели бы нас к булеву результату - давать разрешения на данную строку или нет. Но если передавать все столбцы в эту не табличную функцию, то тогда мы бы жёстко связали бы каждую функцию, ответственную за возвращение булева значения с соответствующей таблицей. Более того, нам необходимо как то менять условия доступа к таблицам, без изменения схемы базы данных.

Мы можем это делать с помощью создания некоторого набора таблиц, в которых будут храниться связи - {текущий пользователь - текущая таблица, над которой выполняется запрос - список предикатов исполняемых над строками текущей таблицы}. Где позже предикат задаётся с помощью простого выражения.

Создание функции для каждой таблицы, которая бы использовалась в предикате для получения данных из текущей строки неприемлемо, так как при изменении схемы базы данных, например ,добавления нового столбца, приходилось бы изменять код предиката чтобы добавить ещё один столбец. Также это невозможно в силу неспособности SQL Server создавать функции и процедуры с динамическим количеством параметров, что необходимо нам для дальнейшей работы. Поэтому для отвязки предикатов и таблиц создадим небольшую функцию, которая принимает на вход два строковых значения. Первое - это имя текущей таблицы. Второе - это строка идентификаторов, содержащих в себе тройки - имя столбца, значение в строке, тип. Это необходимо чтобы в этой универсальной функции можно было обращаться к строчке, над которой сейчас происходит проверка. Если ключ составной, необходимо указать столько троек, сколько столбцов учувствуют в составном ключе текущей таблицы.

Сама функция должна возвращать два значения - true или false(1или 0). В SQL Server самый подходящий тип для данного результата это bit. Функция состоит из 2 частей. В первой необходимо проверить, есть ли предикаты для текущего пользователя; если таких предикатов нет, то нужно сразу возвращать 1 или 0, в зависимости какой тип алгоритма доступа выбран - не давать доступ, если пользователь не имеет никаких предикатов, либо наоборот - всегда давать доступ пользователям, для которых не были добавлены предикаты. Во второй части необходимо исполнить все предикаты текущего пользователя для текущей таблицы. Результат исполнения предикатов является результатом данной функции.

Исполнять предикаты в SQL Server алгоритмом написанном на языке SQL очень сложно и скорее всего будет потеря производительности, так как некоторые операции не оптимизированы в данном языке. Поэтому проще воспользоваться CLR функцией написанной на языке C#, в которой гораздо легче написать логику проверки доступа к текущей строке, а за оптимизацию позаботится среда .NET. Для того чтобы использовать CLR функцию необходимо включить поддержку CLR в SQL server

```
EXEC sp_configure 'clr enabled', 1  
RECONFIGURE;
```

Для использования функции написанной на языке C#, необходимо определиться, что передавать в эту функцию, так как она должна быть универсальной для любой таблицы. Необходимо передавать имя текущей таблицы, идентификаторы текущей строки и конкатенированную строку предикатов. Конкатенация предикатов будет происходить вместе со строкой " and " являющуюся логическим И между предикатами.

Также в основном в предикатах нам необходимо иметь возможность написать условие не только для значений текущей строки, но для каких то данных пользователя, которые производит запрос к текущей таблице. Данные пользователя также хранятся в некоторой таблице или таблицах и могут быть получены напрямую из таблиц или с помощью представления SQL. Самым простым решением является передача идентификаторов строки с данными текущего пользователя в функцию исполняющую предикаты.

Добавим библиотеку с CLR функцией

```
create ASSEMBLY Parser FROM '[путь к библиотеке]/[имя библиотеки].dll'  
WITH PERMISSION_SET = UNSAFE;
```

Создадим SQL функцию, привязанную к CLR функции библиотеки

```
create FUNCTION dbo.getUserAccessClr(  
    @currentTableName nvarchar(500),  
    @userTableName nvarchar(500),  
    @predicates nvarchar(max),  
    @currentRowIdentifiers nvarchar(500),  
    @userRowIdentifiers nvarchar(500)  
    ) RETURNS bit  
AS EXTERNAL NAME  
Parser.[SqlParcer.ContextParcer].ExecutePredicate;
```

Во время запроса к базе данных к таблице к которой привязаны предикаты текущего пользователя необходимо каким-либо образом указать идентификатор текущего пользователя. То есть такие данные, по которым можно однозначно определить строчку таблицы или представления, в которой содержатся данные этого пользователя. Самое простое решение - поместить идентификатор пользователя в контекст сессии текущего запроса. Его необходимо помещать такими же тройками, какие были использованы ранее, то есть - имя колонки, значение, тип. Это необходимо чтобы потом в функции выдающей доступ можно было разобрать эти тройки и однозначно определить строку с данными текущего пользователя.

Напишем функцию, вставляющую в контекст сессии эти тройки значений.

```
create PROCEDURE setInitialContext(  
    @UserId int)  
AS
```

```

SET NOCOUNT ON;
EXEC sp_set_session_context 'UserId', @UserId;
GO

```

Напишем функцию дающую право доступа к текущей строке таблице, к которой происходит запрос. Полный код данной функции находится в приложении А.

```

CREATE FUNCTION dbo.getUserAccess(@CurrentTableName
nvarchar(200), @RowIdentifiers nvarchar(max))
RETURNS bit
WITH SCHEMABINDING
AS
BEGIN
    DECLARE @predicates nvarchar(max);
    Declare @result bit;
    select @predicates = COALESCE(@predicates + ' and ', '') +
    dbo.Predicates.Value from
        dbo.Predicates join dbo.Policies
    on dbo.Predicates.id = dbo.Policies.PredicateId
    and dbo.Predicates.TableName = @CurrentTableName
    join dbo.EmployeeGroups
    on dbo.EmployeeGroups.GroupId = dbo.Policies.GroupId
    and dbo.EmployeeGroups.EmployeeId =
    CAST(SESSION_CONTEXT(N'UserId') AS int);

    if @predicates is Null
    begin
        set @result = 1
    end
    else
    begin
        select @result = dbo.getUserAccessClr(
            @CurrentTableName,
            'Имя таблицы или представления с данными о пользователе',
            @predicates,
            @RowIdentifiers,
            CONCAT ('[id]', cast(SESSION_CONTEXT(N'UserId') as
nvarchar(50)), '[int]')
        )
    end
    return @result
END;

```

В первой части этой функции находится конструкция, конкатенирующая предикаты текущего пользователя, чей идентификатор занесён в контекст сессии. Для установки предикатов для конкретных пользователей мы используем набор таблиц - Predicates(Предикаты), Groups(группы пользователей), Employees(таблица пользователей), EmployeeGroups(таблица для связи типа "многие ко многим" для таблиц Groups и Employees). После соединения трёх таблиц, с помощью join, мы получаем список предикатов, точнее их строковых значений для выбранного пользователя и составляем одну строку, путём конкатенирования строк с добавлением ключевого слова 'and', являющегося в языке выражений для предикатов логическим "И".

Далее необходимо создать по предикату и политике безопасности на каждую таблицу, к которым далее можно будет добавлять предикаты для пользователей. Можно также создать одну политику безопасности и все предикаты для всех таблиц прикрепить к ней, но тогда невозможно будет отключать проверки для конкретных таблиц. Или же можно создать политики безопасности для одних и тех же таблиц и при надобности активировать одну из политик.

```
CREATE FUNCTION [Имя функции-предиката](@id int)
    RETURNS TABLE
    WITH SCHEMABINDING
AS
    RETURN SELECT 1 as Result where ((select dbo.getUserAccess([Имя
таблицы к которой привязывается предикат], concat([конкатенированные
тройки идентификаторов текущей строки]))) = 1)
```

```
CREATE SECURITY POLICY [Имя политики безопасности]
ADD FILTER PREDICATE [Имя функции-предиката] ([список
колонок как параметры функции составляющие вместе составной ключ])
    ON [Имя таблицы]
WITH (STATE = ON);
```

Данной конструкцией мы добавили функцию, которая будет применяться к каждой строке указанной таблице при операции чтения. В основном нам также необходимо отказать в доступе на запись строки, не удовлетворяющей тому же предикату, который был использован для фильтрации выборки. Поэтому мы можем усложнить немного политику безопасности, добавив блокирующий предикат для операции вставки

```
CREATE SECURITY POLICY [Имя политики безопасности]
ADD FILTER PREDICATE [Имя функции-предиката] ([список
колонок как параметры функции составляющие вместе составной ключ])
```



```
ON [Имя таблицы]
ADD BLOCK PREDICATE [Имя функции-предиката] ([список
колонок как параметры функции составляющие вместе составной ключ])
ON [Имя таблицы] AFTER INSERT
WITH (STATE = ON);
```

Также, мы можем не создавать политику безопасности на каждую таблицу. SQL Server позволяем нам добавлять неограниченное количество предикатов в одной политике безопасности, но только для разных таблиц.

После всех этих шагов политика безопасности должна работать как при выборке, так и при вставке значений в таблицу, на которую наложены предикаты, будет срабатывать CLR функция `getUserAccess`, которая должна возвращать булево значение. Полный код данной функции приведён в приложении Б.

4.2 Описание CLR функции исполняющей предикаты

На вход данной функции подаётся 5 строк - имя текущей таблицы, для которой происходит выборка или вставка, имя таблицы или представления с данными пользователя, конкатенированная строка предикатов, соединенных логическим "И", строка идентификаторов строки, над которой сейчас происходит вызов функции, строка идентификаторов строки с данными о пользователе.

Так как мы имеем идентификаторы строки пользователя и текущей строки, мы можем запросить сами данные, построив SQL запрос с выражением "where" которое будет содержать условие равенства переданных идентификаторов, в рамках текущего соединения. То есть SQL Server не будет создавать новое соединение с базой данных, а просто использует то, через которое происходит вызов данной функции.

Так как нам надо запросить только те колонки, которые используются в предикатах, необходимо разобрать переданную нам строку предикатов и выяснить - используются ли данные из таблицы пользователей или только данные из текущей таблицы, или же используется данные из обеих таблиц. Это позволит избежать лишнего запроса к базе данных и получения ненужных данных.

Для того чтобы разобрать и исполнить предикат или предикаты, необходимо понять в каком виде можно задавать эти булевы условия. Самым удобным способом описать способ задания логических уравнений можно с помощью языка с контекстно-свободной грамматикой. Приведем некоторые правила -

1. Последовательность символов в конечном итоге представляющую предикат должна логически переводиться в функцию, возвращающую булево значение.
Пример: "2 < 4 and 3 = 3"
2. В предикате можно записывать условия, используя переменные вида - R.[название столбца из текущей таблицы] и C.[название столбца из таблицы пользователя].
Пример: "R.[Город поставщика] = C.[Город пользователя]"
3. Можно использовать операторы - >, >=, <=, =, !=, <, -, +, /, *, and, or, as, like.
Пример: "3 < 2 + 2 and 3 * 7 = 21 and "машина" like "%шин%" "
4. Можно использовать скобки для группировки операндов.
Пример: "(3 + 2) * 4 - (1 - 3) / 2 = 12"
5. Можно использовать числа типов - Int16, Int32, Int64, Double среды C#.
Пример: "12.2 + 13 = 25.2"
6. Можно использовать двойные кавычки для указания строки.
Пример: "R.[Город поставщика] = "Ярославль" "
7. При сравнении значений они могут быть как одного типа, так и разных. Если при сравнении двух значений типы разные, то они оба приводятся к одному, более вместительному по памяти
8. Можно писать выражения, использующие практически все основные типа языка C# - byte, string, byte[], bool, Int16, Int32, Int64, Double, Float, DateTime, TimeSpan, DateTimeOffset, Guid.
Пример: "R.[Дата поставки] + R.[Возможная задержка] < "12.12.2017" as datetime "
9. Можно совершать явное приведение типов с помощью оператора as. Приведение к некоторому значению заданного типа возможно, только если это можно сделать в языке C#. Также с помощью этого оператора можно конвертировать строку в указанный тип, при условии, что значение данного типа можно конвертировать в строку.
Пример: "R.[Дата поставки] as string = "12.12.2017" "

Для более формального представления данного языка и для возможности написать алгоритм разбора выражений необходимо составить грамматику.

выражение \rightarrow литерал

- | унарная операция
- | бинарная операция
- | группировка;

литерал \rightarrow число | строка | "true" | "false" | "nil" ;

группировка \rightarrow "("выражение")" ;

унарная операция \rightarrow ("-" | "!") выражение;

бинарная операция \rightarrow выражение оператор выражение;

оператор \rightarrow "=" | "!=" | "<" | "<=" | ">" | ">="

| "+" | "-" | "*" | "/" | "as" | "like" | ;

число \rightarrow Int16 | Int32 | Int64 | Double

С помощью данной грамматики происходит разбор выражений в предикате, построением списка узлов, являющихся идентифицированными объектами. Далее по ним можно определить какой это узел - переменная, служебный символ, оператор и т.д. Выражения распознаются рекурсивно, полностью соответствуя представленной контекстно-свободной грамматике. Код сканирующей функции находится в приложении В.

После того как выражение разобрано в список типизированных элементов, можно определить из каких таблиц какие колонки используются, чтобы далее выполнить запросы за данными этих столбцов к базе данных. После того как мы поменяем все переменные на реальные значения, необходимо построить бинарное дерево, где каждый узел является либо оператором, либо значением. Так как у нас все операторы являются бинарными либо унарными, мы легко можем построить дерево выражений. Самым простым решением для этой задачи является использование обратной польской записи. Код функции построения узлов дерева находится в приложении Г.

После того как дерево построено, необходимо обойти его в глубину подменяя узлы операторов на вычисленные значения. Тем самым мы можем вычислять выражения любой сложности, лишь бы они соответствовали грамматике данного языка. Код функции вычисления узлов дерева находится в приложении Д.

Пример дерева для выражения "R.City = C.City and R.Revenue - 53.5 > 0" приведён на рисунке 1. Обход дерева производится в глубину слева направо. Действия при обходе данного дерева:

1. проверить на равенство переменную City из строки и переменную City из контекста
2. вычесть из значения типа double переменной строки Revenue число 53.5 также имеющее тип double
3. привести число 0 к числу double
4. проверить, что значение типа double вычисленное на шаге 2 больше, чем значение того же типа вычисленное на шаге 3

5. вычислить логическое "И" значений вычисленных на шагах 1 и 4

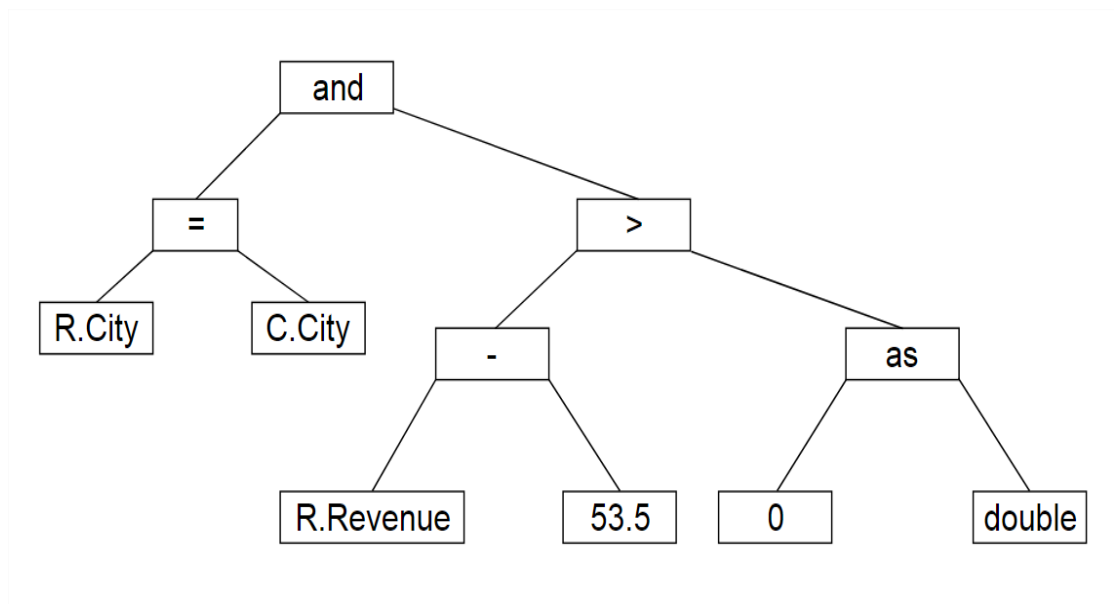


Рисунок 1

4.3 Приложение для работы с политикой безопасности

Рассмотрим пример, где необходимо динамически менять условия ограничения конечных пользователей к данным в базе данных. Работать с базой данных сотруднику компании напрямую не удобно, а часто просто не безопасно. Поэтому создадим небольшое приложение, в котором он сможет просматривать, создавать и изменять какие-либо сущности из базы данных в той компании, в котором он работает. Предположим, что у нас имеется похожая структура базы данных, как в примере реализации RLS без встроенной поддержки этого механизма.

Кратко опишем сущности базы данных:

Таблица заказов с идентификаторами клиента, сделавшего заказ и идентификатора сотрудника, оформившего этот заказ

```
table Orders {  
    id int IDENTITY(1,1) NOT NULL,  
    CustomerID int NOT NULL,  
    EmployeeID int NOT NULL,  
    OrderDate datetime NULL
```

```
...  
}
```

Таблица продуктов с ценой, количеством товара, идентификатором категории и названием продукта

```
table Products {  
    id int IDENTITY(1,1) NOT NULL,  
    Name nvarchar(40) NOT NULL,  
    CategoryId int NULL,  
    Price money NULL,  
    Number smallint NULL  
    ...  
}
```

Таблица соединения заказов и продуктов, так как в один заказ может входить несколько товаров

```
table OrderDetails {  
    OrderId int NOT NULL,  
    ProductID int NOT NULL,  
    Number smallint NOT NULL  
}
```

Таблица сотрудников, где указаны его полное имя, дата устройства, телефон и т.д.

```
table Employees {  
    id int IDENTITY(1,1) NOT NULL,  
    FullName nvarchar(150) NOT NULL,  
    BirthDate datetime NULL,  
    HireDate datetime NULL,  
    City nvarchar(20) NULL,  
    Phone nvarchar(24) NULL
```

...

}

Таблица групп сотрудников компании

table Groups {

id int IDENTITY(1,1) NOT NULL,

Name nvarchar(50) NOT NULL,

Description nvarchar(1000) NULL

...

}

Таблица соединения сотрудников и групп

table EmployeeGroups {

EmployeeId int not null

GroupId int not null

}

Таблица предикатов с указанием таблицы, на которую будет применяться условие предиката(Value)

table Predicates {

id int IDENTITY(1,1) NOT NULL,

Value nvarchar(1000) NOT NULL,

TableName nvarchar(100) not null

}

Таблица политик безопасности - соединение групп и предикатов, для которых они должны быть выполнены

table Policies {

GroupId int NOT NULL,

PredicateId int NOT NULL

}

В данной схеме мы создали 4 вспомогательные таблицы - группы, таблица соединения групп и сотрудников, таблица предикатов с их

значениями, написанными в простой текстовой форме, таблица политик для соединения идентификатора группы и предиката.

Для работы с базой данных необходимо приложение, чтобы создавать, редактировать и просматривать информацию о сотрудниках, заказах, продуктах и т.д. Приложение в связи с последними трендами будет типа web SPA - то есть все операции будут происходить в браузере без перезагрузки страницы.

Также нам необходимо обеспечить поддержку запросов с клиента на серверной части - то есть REST API сервис для create/read/update/delete операций. Таким образом, для каждой сущности, то есть сотрудники, заказы, предикаты и т.д. необходимо обеспечить обработку операций с базой данных. Приложение будет отправлять запросы к серверной части на адрес - xxxx/api/ууу, где xxx - это домен нашего приложения, а ууу - это название сущности к которому происходит доступ. Домен мы выберем локальный, то есть localhost для простоты разработки. Серверная часть будет написана на языке C# с применением объектно-ориентированной технологией Entity Framework на платформе WEB API 2 фреймворка ASP.NET MVC.

Далее, как описано ранее, необходимо включить возможность подключения CLR библиотек в SQL SERVER и подключить библиотеку для выполнения предикатов, создав процедуру, которая будет шлюзом между API SQL SERVER и CLR функции:

```
EXEC sp_configure 'clr enabled', 1
RECONFIGURE;
GO
create ASSEMBLY Parser FROM '**\SqlParser.dll';
GO
create FUNCTION getUserAccessClr(@Predicate nvarchar(max))
RETURNS bit
AS EXTERNAL NAME Parser.ContextParser.ExecutePredicate;
```

Далее необходимо создать предикаты для каждой таблицы, которые будут возвращать таблицу с одним столбцом и одной записью, где значение 1 - означает допуск, 0 - отказ. Код предикатов для всех таблиц будет практически один и тот же, меняться будут только имена таблиц и идентификаторы.

```
create FUNCTION dbo.securityPredicateOrders(@id int)
RETURNS TABLE
WITH SCHEMABINDING
AS
```

```
RETURN SELECT 1 as Resu where ((select
dbo.getUserAccess('dbo.Orders', concat('[id][', @id, '][int]')) = 1)
```

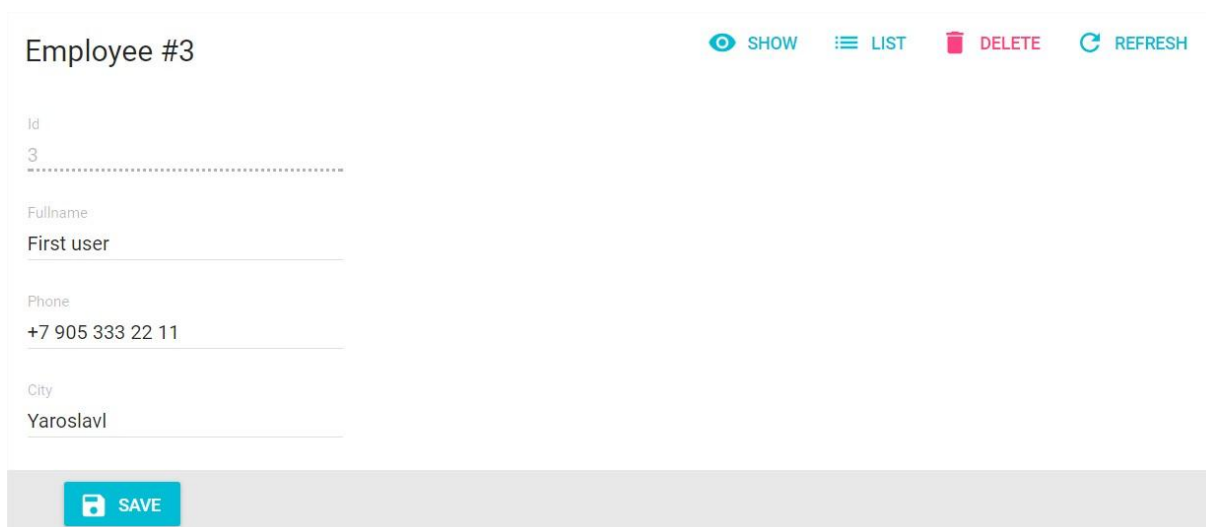
Осталось только создать политику и привязать к ней предикат

```
create SECURITY POLICY dbo.[OrdersPolicy]
ADD FILTER PREDICATE dbo.securityPredicateOrders(id)
ON [dbo].[Orders]
ADD BLOCK PREDICATE dbo.securityPredicateOrders(id)
ON [dbo].[Orders] AFTER INSERT
WITH (STATE = ON);
```

Таким образом, мы создали приложение не только для CRUD операций над сущностями базы данных, но и для управления доступом к записям с помощью системы предикатов. Код

4.4 Пример создания политики безопасности

Предположим, что нам необходимо ограничить для определённых групп сотрудников показ и редактирование заказов, оформленных в нашем приложении. У нас имеется две группы пользователей - из города Ярославль и из города Москва. Нам необходимо для сотрудников из Москвы ограничить доступ для всех 4 операций (create, read, update, delete) с сущностью базы данных - Заказ. Создание сотрудников показано на рисунках 2 и 3. Создание групп для двух городов показано на рисунках 4 и 5.



Employee #3

SHOW LIST DELETE REFRESH

Id
3

Fullname
First user

Phone
+7 905 333 22 11

City
Yaroslavl

SAVE

Рисунок 2

Employee #4

[SHOW](#) [LIST](#) [DELETE](#) [REFRESH](#)

Id

4

Fullname

Second user

Phone

+7 905 444 44 55

City

Moscow

SAVE

Рисунок 3

Group

[SHOW](#) [LIST](#) [DELETE](#) [REFRESH](#)

Id

1

Name

Moscow

Description

Group for employees from Moscow

SAVE

Рисунок 4

Group

[SHOW](#) [LIST](#) [DELETE](#) [REFRESH](#)

Id

2

Name

Yaroslavl

Description

Group fro employees from Yaroslavl

SAVE

Рисунок 5

Оформим по заказу на каждого сотрудника. Создание заказов показано на рисунке 6 и 7.

Order
SHOW LIST DELETE REFRESH

Id
5

Customer Id
5

Employee Id
3

Description
Order from Yaroslavl

SAVE

Рисунок 6

Order
SHOW LIST DELETE REFRESH

Id
6

Customer Id
5

Employee Id
4

Description
Order from Moscow

SAVE

Рисунок 7

После добавления сотрудника First User в группу Moscow, а сотрудника Second User в группу Yaroslavl, каждый из сотрудников пока может видеть заказы друг друга. Список заказов для обоих сотрудников показан на рисунке 8.

| Orders List | | | | + CREATE REFRESH | |
|-------------|-------------|-------------|----------------------|------------------|------|
| ID | CUSTOMER ID | EMPLOYEE ID | DESCRIPTION | | |
| 5 | 5 | 3 | Order from Yaroslavl | EDIT | SHOW |
| 6 | 5 | 4 | Order from Moscow | EDIT | SHOW |

Рисунок 8

Создадим для каждой группы по одному примитивному предикату безопасности. Предикат будет содержать выражение проверки на

принадлежность сотрудника соответствующему городу. Для пользователя из Москвы укажем предикат - City = "Moscow", а для пользователя из Ярославля - City = "Yaroslavl". Также укажем таблицу - Orders, к которой будет привязан предикат. Создание предикатов показано на рисунках 9 и 10.

The screenshot shows a web interface for configuring a policy. At the top, there's a title "Policy" and four action buttons: "SHOW" (eye icon), "LIST" (list icon), "DELETE" (trash icon), and "REFRESH" (refresh icon). Below the title, there are several input fields: "Id" with the value "3", "Table" with the value "Orders", "Predicate" with the value "City = 'Yaroslavl'", and "Identifiers of groups" with the value "1". At the bottom, there is a blue "SAVE" button with a floppy disk icon.

Рисунок 9

The screenshot shows a web interface for configuring a policy. At the top, there's a title "Policy" and four action buttons: "SHOW" (eye icon), "LIST" (list icon), "DELETE" (trash icon), and "REFRESH" (refresh icon). Below the title, there are several input fields: "Id" with the value "7", "Table" with the value "Orders", "Predicate" with the value "City = 'Moscow'", and "Identifiers of groups" with the value "2". At the bottom, there is a blue "SAVE" button with a floppy disk icon.

Рисунок 10

Политика безопасности для таблицы Orders включается сразу же, как только мы привязали к ней предикат. Поэтому зайдя под каждым пользователем, мы можем увидеть, что они видят отфильтрованные результаты - каждый видит заказы только из своего города. Списки заказов для каждого сотрудника отображены на рисунках 11 и 12.

| Orders List | | | | + CREATE ↻ REFRESH | |
|-------------|-------------|-------------|----------------------|-----------------------|------|
| ID | CUSTOMER ID | EMPLOYEE ID | DESCRIPTION | | |
| 5 | 5 | 3 | Order from Yaroslavl | EDIT | SHOW |

Рисунок 11

| Orders List | | | | + CREATE ↻ REFRESH | |
|-------------|-------------|-------------|-------------------|-----------------------|------|
| ID | CUSTOMER ID | EMPLOYEE ID | DESCRIPTION | | |
| 6 | 5 | 4 | Order from Moscow | EDIT | SHOW |

Рисунок 12

4.5 Производительность

Рассмотрим данную систему динамически изменяемых предикатов безопасности с точки зрения быстродействия. С научной точки зрения можно рассмотреть и реализовать огромное количество задач, алгоритмов, схем и т.д. Но с практической точки зрения необходимо быть уверенным, что данная система или алгоритм имеет право на жизнь в реальных условиях. Чтобы это проверить, необходимо воссоздать примерные условия, в данном контексте - использовать выборку из больших данных.

Для тестирования быстродействия предикатов безопасности необходимо следующее:

1. создать базовый набор таблиц, требуемых для политики безопасности
2. создать по 2 предиката на каждый тип данных, доступный в представленной грамматике языка. Первый предикат будет использовать данные из строки и использовать сравнение с одним типом данных. Второй предикат будет представлять из себя сравнение данных из строки и контекста того же типа
3. создать ещё 4 предиката, в которых 7, 14, 21, 28 конкатенированных условий являющихся сравнениями данных из текущей строки и строки с данными пользователя
4. создать по пользователю на каждый вид предиката, то есть всего $7 + 7 + 4 = 18$ пользователей
5. создать по группе на каждого пользователя
6. создать 4 базы данных с 10 000, 100 000, 500 000, 1 000 000 строчек текущей таблицы соответственно

Для проверки ограничимся основными типами данных, обычно используемых в условиях where. В нашем тесте этими типами будут - bit, nvarchar, int, datetime, time, datetimeoffset, uniqueidentifier. Таблица с данными о пользователе будет иметь такую же структуру и данные, как и таблица, к которой будут привязаны предикаты. Данные в каждой строке текущей таблицы будут одни и те же, так как нет смысла заполнять их случайными данными в силу, того что CLR функция не кэширует и не сохраняет результат предыдущего сравнения.

Замеряем время выборки из базы данных, когда для каждой строчки используется предикат с одним условием на простое сравнение со значением одного из типов данных. Запрос для всех измерений будет использоваться - select count(1) from TestTable, где TestTable - это текущая таблица. Результаты замеров скорости для выборок размером 10 000, 100 000, 500 000, 1 000 000 записей показан на рисунке 13. Результаты измерений для предикатов с условием, где происходит сравнение данных из текущей строчки с данными из таблицы пользователей показан на рисунке 14.

На горизонтальной оси находятся типы данных в условии предиката. На вертикальной оси время выполнения запроса в секундах.

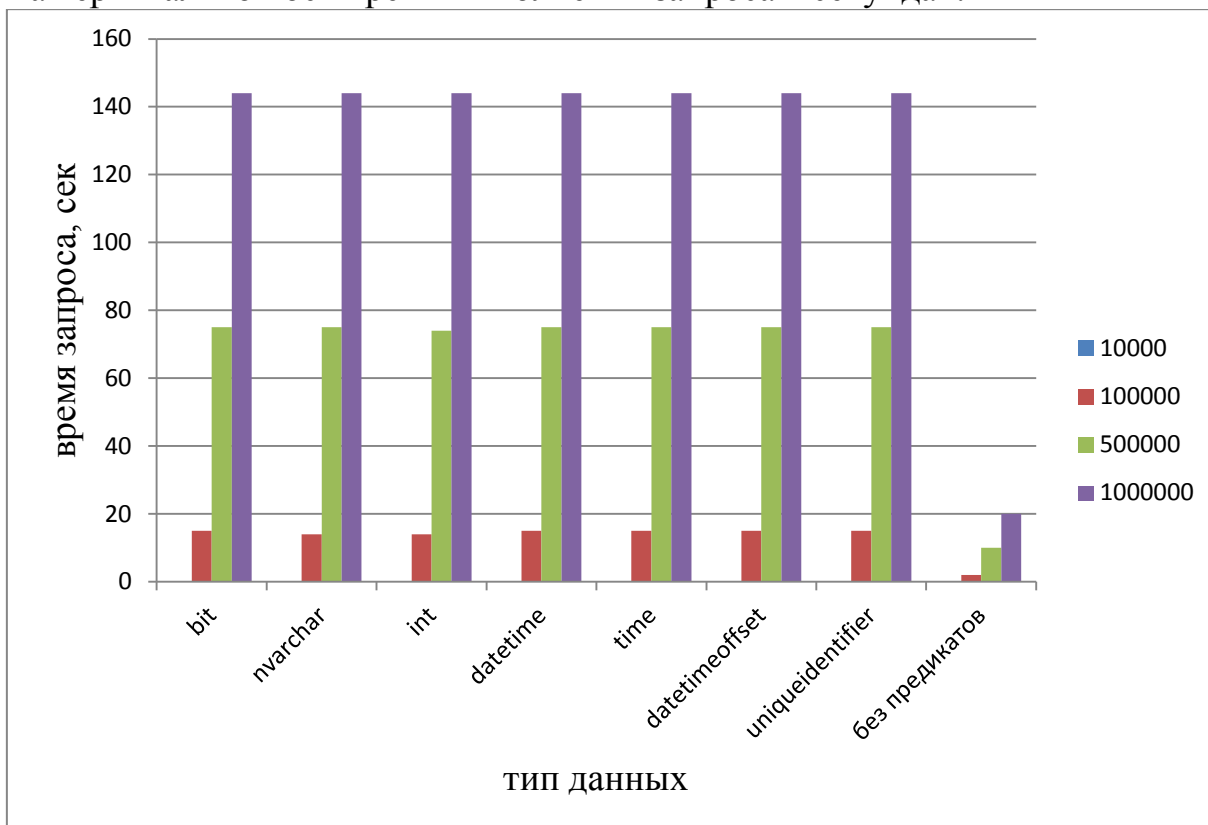


Рисунок 13. Результат измерений предикатов с одним условием

Для базы данных с 10 000 строк время запроса составляет меньше секунды, поэтому оно не отражено на графике. Для базы данных с 100 000

записей - примерно 14 секунд. Для базы данных с 500 000 записей - примерно 75 секунд. Для базы данных 1 000 000 - примерно 144 секунды.

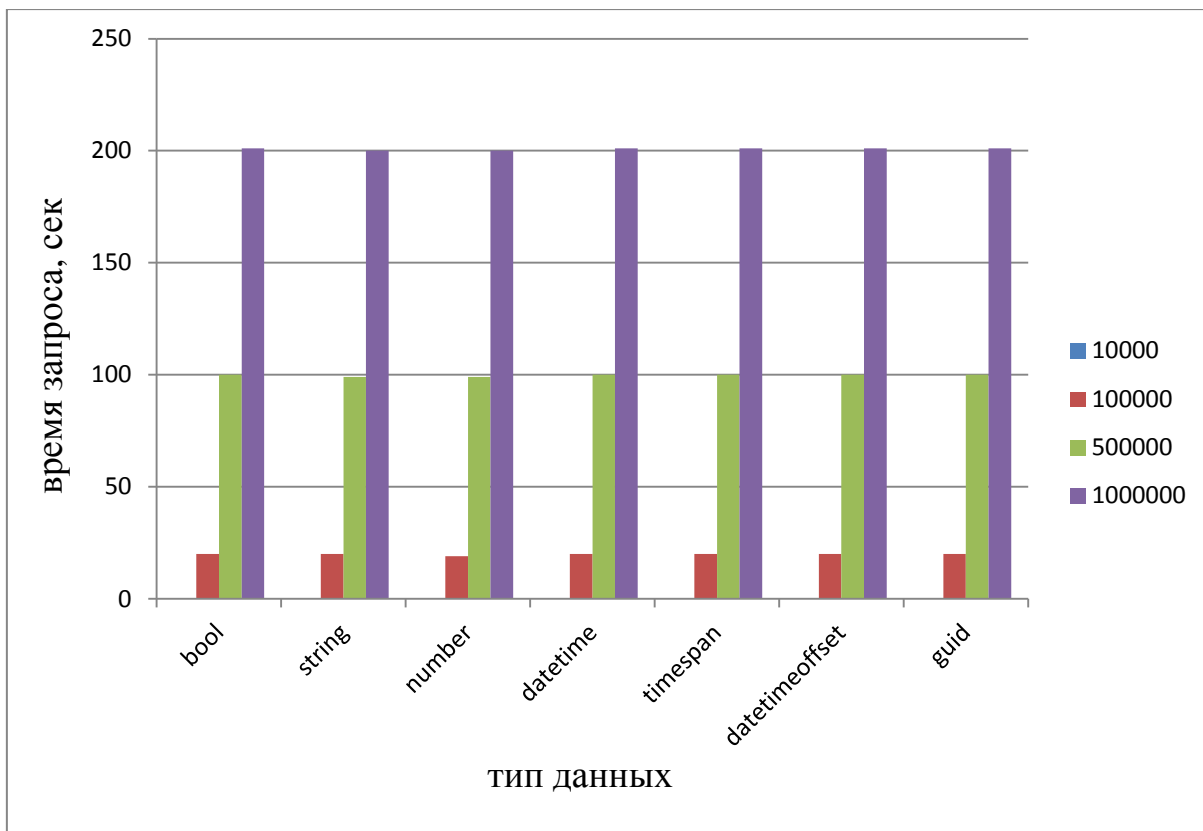


Рисунок 14. Результат измерений предикатов с сложным условием

Для базы данных с 10 000 строк время запроса составляет меньше секунды, поэтому оно не отражено на графике. Для базы данных с 100 000 записей - примерно 20 секунд. Для базы данных с 500 000 записей - примерно 100 секунд. Для базы данных с 1 000 000 записей - примерно 200 секунд.

Также необходимо проверить скорость выполнения запроса, когда на необходимо исполнить несколько предикатов. Для это замеряем время для того же размера выборки с количеством предикатов 7, 14, 21, 28. Результаты измерений приведены на рисунке 15.

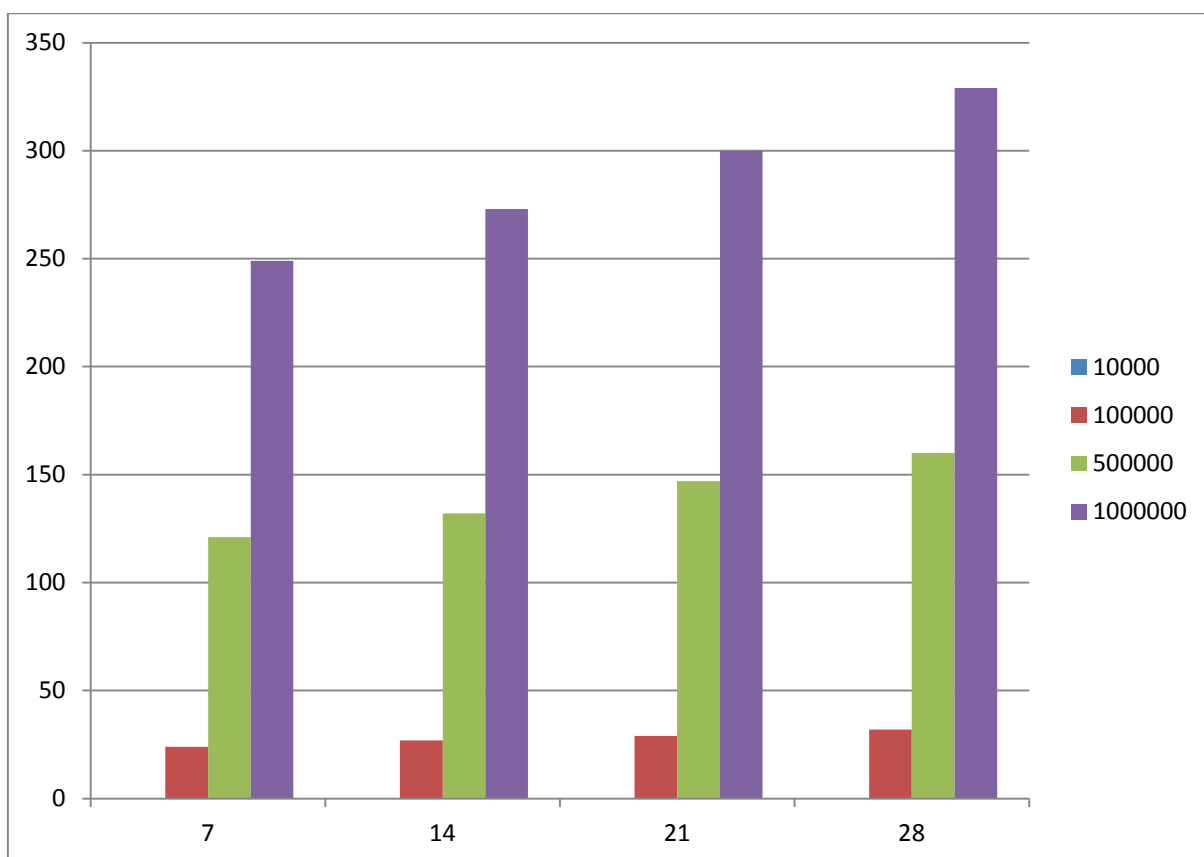


Рисунок 15. Результат измерений предикатов с несколькими условиями

Для базы данных с 10 000 строк время запроса составляет меньше секунды, поэтому оно не отражено на графике. Для базы данных с 100 000 записей время составляет от 24 секунд до 32 секунд в зависимости от количества предикатов. С каждым шагом добавления 7 предикатов к условию, добавляется примерно 2-3 секунды к времени запроса. Для базы данных с 500 000 записей время составляет от 121 секунд до 160 секунд в зависимости от количества предикатов. С каждым шагом добавления 7 предикатов к условию, добавляется примерно 12-13 секунд к времени запроса. Для базы данных с 1 000 000 записей время составляет от 249 секунд до 329 секунд в зависимости от количества предикатов. С каждым шагом добавления 7 предикатов к условию, добавляется примерно 23 секунды к времени запроса.

Как видно из результатов использование данного подхода с динамическим добавлением предикатов непрактично в реальном мире, так как запрос получения записей происходит слишком медленно. Для количества записей меньше 10 тысяч система работает достаточно быстро, но скорее всего найдётся очень мало реальных приложений с количеством записей меньше 10 000 для которых требуется такая гибкая система разграничения прав доступа, основанная на предикатах для строчек баз данных.

Можно ускорить данный алгоритм путём убирания строк идентификаторов из аргументов функции предиката и добавлением для каждой таблицы CLR функции, которая будет принимать на вход все

значения строки текущей таблицы и все значения строки с данными текущего пользователя. В таком случае мы теряем гибкость - когда необходимо добавить столбец к таблице, нам нужно будет перекомпилировать CLR функцию и все связанные с ней предикаты и политики безопасности. Предположим, что схема базы данных меняется не часто и мы можем допустить способ решения проблемы.

Для проверки данного решения необходимо заменить часть CLR функции, где происходил разбор предикатов и выполнение запросов за значениями столбцов из текущей таблицы и таблицы пользователя, на вызов функции, сразу исполняющей предикат, так как надобность в получении дополнительных данных на этом моменте уже отпала. Также нам необходимо поменять аргументы функции SQL `getUserAccessClr` которая отражает аргументы CLR функции `ExecutePredicate`. В код функции получения результата от CLR функции необходимо добавить выборку из таблицы пользователей, так как нам все равно необходимо получить данные о текущем пользователе в момент исполнения запроса. В итоге для возможности явной передачи всех значений в CLR функцию, необходимо для каждой таблицы, на которую необходимо наложить условия доступа, создать по политике безопасности, функции предиката, CLR функции, функции вызывающей CLR функцию.

Протестируем данный подход на тех же данных, только на базе данных с 1 000 000 записей, так как время исходя из измерений растёт прямо пропорционально количеству записей. Результаты измерений приведены на рисунке 16. Горизонтальная ось отвечает за количество предикатов. Вертикальная ось за время выполнения запроса.

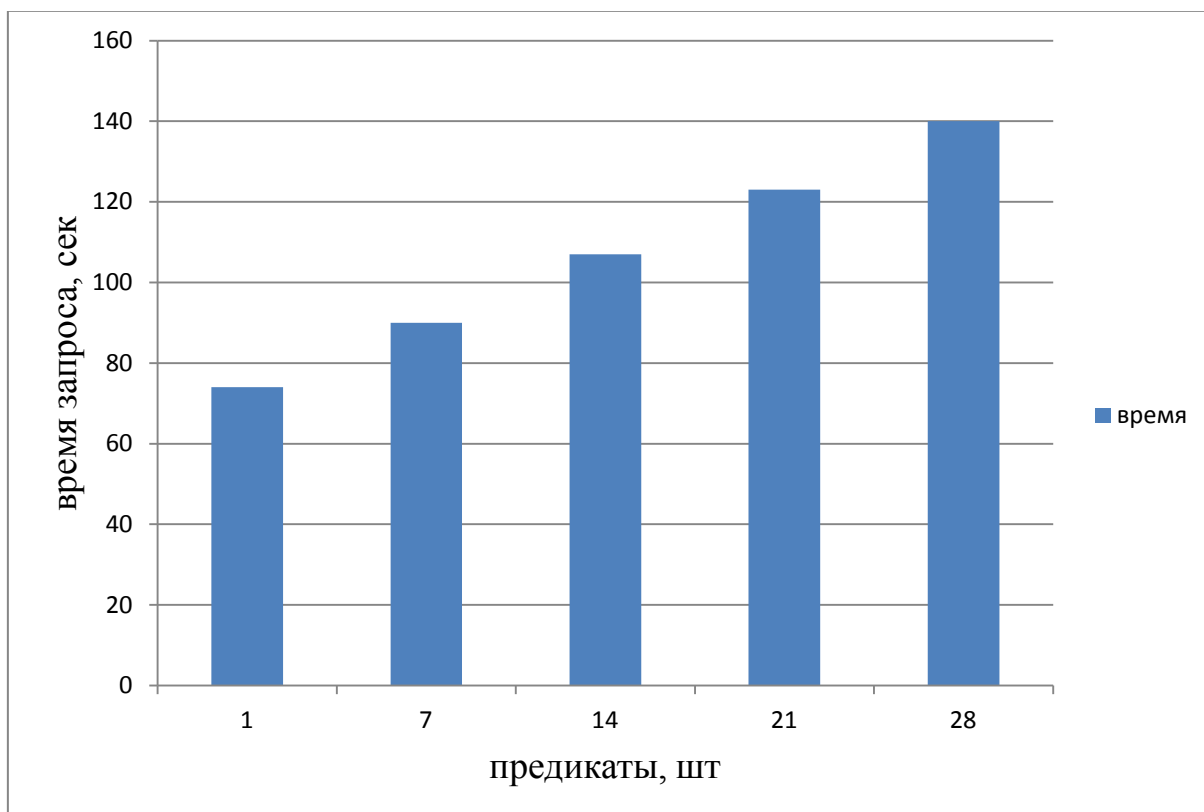


Рисунок 16. Результат измерений предикатов с возрастающим количеством условий

Из результатов можно увидеть, что время выполнения запросов уменьшилось примерно в 3 раза, но всё равно составляет достаточно приличную величину. Данный подход гораздо быстрее предыдущего за счёт потери гибкости, но всё равно время выполнения запроса слишком большое, чтобы его можно было применить в реальном мире.

Можно выяснить, что больше всех влияет на скорость запроса к базе данных. При запросе к таблице, на каждую строчку вызывается функция-предикат и происходит ряд вещей:

1. вызов функции `getUserAccess`
2. получения контекста сессии и запрос к 3 таблицам оператором `join` - пользователям, предикатам, связывающей таблице предикатов и групп
3. запрос к таблице пользователей за данными о текущем пользователе, чей идентификатор находится в контексте сессии
4. вызов CLR функции с передачей ей всех данных из строки текущей таблицы и строки текущего пользователя
5. разбор и исполнение предикатов в CLR функции

Будем изменять код, таким образом чтобы можно было протестировать запрос к таблице, добавляя код из данных шагов по одному.

Уберём вызов функции `getUserAccess`, тем самым оставив в функции предикате только проверку на проверку значения одного из столбцов на случайное значение.

```
CREATE FUNCTION dbo.securityPredicateTestTable(@id int,
@BoolType bit, @IntType int, @StringType nvarchar(500), @DateTimeType
datetime, @DateTimeOffsetType datetimeoffset,
@TimeType time, @GuidType uniqueidentifier)
RETURNS TABLE
WITH SCHEMABINDING
AS
RETURN SELECT 1 as Resu where @BoolType = 1
```

Результат является ожидаемым так как SQL server скорее всего оптимизировал данный запрос и поэтому запрос выполняется мгновенно. Значит, добавление предиката и использование политики безопасности не влияет существенным образом на время выполнения запроса.

Оставим вызов функции `getUserAccess` и перенесём сравнение одной из колонок в её код, убрав всё остальное.

```

...
BEGIN
  Declare @result bit;
  if @BoolType1 = 1
  begin
    set @result = 1
  end
  else
  begin
    set @result = 0
  end
  return @result
END;

```

Время выполнения теперь составляет 1 секунду. То есть время, затраченное на вызов сторонней функции в табличной функции предиката с простым логическим условием, занимает немного времени, но существенно, если усложнять запрос.

Добавим в функцию getUserAccess запрос к трём соединённым таблицам, чтобы получить конкатенированный предикат. Условие результата будет вычисляться так - если есть предикаты для текущего пользователя, то возвращаем 1, в противном случае 0.

```

...
DECLARE @predicates nvarchar(max);
Declare @userId int = CAST(SESSION_CONTEXT(N'UserId') AS int)
select @predicates = COALESCE(@predicates + ' and ', '') +
dbo.Predicates.Value from
  dbo.Predicates join dbo.Policies
on dbo.Predicates.id = dbo.Policies.PredicateId
and dbo.Predicates.TableName = @CurrentTableName
join dbo.EmployeeGroups
on dbo.EmployeeGroups.GroupId = dbo.Policies.GroupId
and dbo.EmployeeGroups.EmployeeId = @userId;
if @predicates is Null
...

```

Время выполнения составляет 44 секунды. Видимо основную часть времени уходит на запрос к 3 таблицам, примерно 40 секунд.

Добавим в данную функцию запрос за данными о пользователе.

```

...
DECLARE @id2 int;
DECLARE @BoolType2 bit;

```

```

DECLARE @IntType2 int;
DECLARE @StringType2 nvarchar(400);
DECLARE @DateTimeType2 datetime;
DECLARE @DateTimeOffsetType2 datetimeoffset;
DECLARE @TimeType2 time;
DECLARE @GuidType2 uniqueidentifier;

```

```

SELECT @id2 = id, @BoolType2 = BoolType, @IntType2 =IntType,
@StringType2 = StringType, @DateTimeType2 = DateTimeType,
@DateTimeOffsetType2 = DateTimeOffsetType, @DateTimeOffsetType2 =
DateTimeOffsetType, @TimeType2 =TimeType, @GuidType2 = GuidType
from dbo.Employees where id = @userId

```

```

if @predicates is Null

```

```

...

```

Время запроса составило 59 секунд. Значит, среднее время дополнительного запроса за данными текущего пользователя на миллион записей составляет примерно 15 секунд.

Добавим в код этой функции вызов CLR функции, сама функция всегда будет возвращать true.

```

...

```

```

else

```

```

begin

```

```

select @result = dbo.getUserAccessClr(
    @predicates, @id1, @BoolType1, @IntType1, @StringType1,
@DateTimeType1, @DateTimeOffsetType1, @TimeType1, @GuidType1,
    @id2, @BoolType2, @IntType2, @StringType2,
@DateTimeType2, @DateTimeOffsetType2, @TimeType2, @GuidType2 )

```

```

end

```

```

return @result

```

```

END;

```

```

...

```

Время выполнения запроса составляет 71 секунда. То есть время вызова функции, которая просто возвращает true, является примерно 12 секунд. Так как время запроса со всеми шагами является примерно 74 секунды, значит среднее время выполнения простого предиката на миллионе записей примерно 3 секунды.

Из вышеуказанных измерений ясно, что основное время занимают подзапросы за данными пользователя и предикатами и вызов CLR функции. Следовательно, использование встроенных механизмов политики безопасности SQL Server вместе с системой, позволяющей

динамически добавлять предикаты, не может использоваться в реальных приложениях, где база данных содержит большое количество данных.

4.6 Альтернативный вариант решения проблемы производительности

Главная проблема RLS подхода, где используется встроенный механизм поддержки предикатов - лишние подзапросы на каждую строчку таблицы. Мы уже выяснили, что если необходимо задавать предикаты динамически без перекомпиляции скриптов базы данных, то нам необходимо иметь вышеописанную систему таблиц и функций, отвечающих за разбор и исполнение предикатов. В SQL Server нет возможности поставить триггер на оператор SELECT, только на вставку, удаление или обновление. Добавить какое-то условие к выражению SELECT для таблицы можно с помощью представления, но тогда нет гибкости в добавлении предикатов на лету, так как придётся как-то перекомпилировать представления после добавления нового или изменения старого предиката. Также это практически невозможно, так как предикаты могут быть одни и те же для нескольких групп пользователей, что сильно усложнило бы статическую систему политик безопасности, созданную на основе представлений и триггеров.

Самым простым решением проблемы производительности системы политик безопасности является вынесением логики выдачи доступа на определённые строки базы данных на сторону приложения. Данный подход самый популярный, так как является самым быстрым с точки зрения выдачи результатов пользователю. В данном подходе реализация гибкой системы безопасности может быть такая:

- имеется та же система таблиц с предикатами, которая описана выше
- предикаты получаются во время запроса к базе данных. Так как мы знаем на момент запроса какие данные пользователь хочет получить, мы можем сделать запрос за предикатами конкретной таблицы или таблиц, если используется соединение
- конкатенированный предикат разбирается тем же алгоритмом, описанным выше, на идентифицированные элементы
- разобранный список элементов из строки предикатов не исполняется, а переводится в язык запросов SQL Server, с помощью построения синтаксического дерева
- сконвертированный в SQL предикат, написанный изначально на языке с описанной выше грамматикой, добавляется к условию where перед отправкой всего запроса на сервер базы данных

- так как запрос за предикатами текущего пользователя происходит один раз, и так как предикат добавляется явно как фильтр для данных, то время запроса значительно сокращается до вполне приемлемого - меньше секунды для простого предиката на миллион записей

Заключение

В данной работе были проведены исследования в создании политик безопасности, как без поддержки встроенных функций и методов сервера базы данных, так и при помощи уже готового фреймворка, предоставляемого СУБД SQL SERVER 2016.

Были исследованы методы создания механизмов для формирования политик безопасности, основанных на предикатах. Также были проведены эксперименты по формированию статических предикатов и динамических, созданных через разбор строки и данных контекста выполнения запросов.

Был реализован гибкий механизм разграничение доступа к данным посредством создания динамически изменяемой системы политик безопасности для созданного в рамках выпускной квалификационной работы приложения управления сущностями базы данных. Была разработана библиотека для анализа и разбора булевого выражения, используемого в качестве функции доступа к конкретной строке.

Можно отметить, что создание политики безопасности с помощью встроенных функций гораздо проще, но не производительнее. Замеры времени запросов к базе данных с включённой политикой безопасности показали, что использование реализованной системы динамических предикатов не является практичной для использования её в реальном приложении.

Список литературы

1. Адам Ф. "ASP.NET MVC 5 с примерами на C# 5.0 для профессионалов". - М.: ДМК Вильямс, 2015. – 736с.: ил.
2. Аруп Н., Стивен Ф. "Oracle PL/SQL для администраторов баз данных" - М.: ДМК Символ-Плюс, 2008. – 496с.: ил.
3. Джеймс Р., Пол Н., Эндрю О. "SQL. Полное руководство" - М.: ДМК Вильямс, 2014. – 960 с.: ил.
4. Ицик Бен-Ган. "Node Microsoft SQL Server 2012. Основы T-SQL" - М.: ДМК Эксмо, 2015. – 400 с.: ил.
5. <http://rdsn.org/> - URL: <http://rdsn.org/article/db/RowLevelSecurity.xml>
6. <https://msdn.microsoft.com> - URL: <https://msdn.microsoft.com/ru-ru/library/dn765131.aspx>

Приложение А

```
create PROCEDURE setInitialContext(@UserId int)
AS
    SET NOCOUNT ON;
    EXEC sp_set_session_context 'UserId', @UserId;
GO
EXEC sp_configure 'clr enabled', 1
RECONFIGURE;
go
create FUNCTION dbo.getUserAccessClr(
    @currentTableName nvarchar(500),
    @userTableName nvarchar(500),
    @predicates nvarchar(max),
    @currentRowIdentifiers nvarchar(500),
    @userRowIdentifiers nvarchar(500)
) RETURNS bit
AS EXTERNAL NAME Parser.[SqlParser.ContextParser].ExecutePredicate;
GO
CREATE FUNCTION dbo.getUserAccess(@CurrentTableName nvarchar(200),
@RowIdentifiers nvarchar(max))
RETURNS bit
WITH SCHEMABINDING
AS
BEGIN
    DECLARE @predicates nvarchar(max);
    Declare @result bit;
    select @predicates = COALESCE(@predicates + ' and ', '') +
dbo.Predicates.Value from
```



```

    dbo.Predicates join dbo.Policies
on  dbo.Predicates.id = dbo.Policies.PredicateId
and  dbo.Predicates.TableName = @CurrentTableName
join  dbo.EmployeeGroups
on  dbo.EmployeeGroups.GroupId = dbo.Policies.GroupId
and  dbo.EmployeeGroups.EmployeeId =
CAST(SESSION_CONTEXT(N'UserId') AS int);
if @predicates is Null
begin
    set @result = 1
end
else
begin
    select @result = dbo.getUserAccessClr(
        @CurrentTableName,
        'dbo.Employees',
        @predicates,
        @RowIdentifiers,
        CONCAT ('[id][', cast(SESSION_CONTEXT(N'UserId') as
nvarchar(50)), '][int]')
    )
end
return @result
END;

```

Приложение Б

```
public static class ContextParser
{
    public static readonly string ConnectionString = "context connection=true";
    public static readonly Random random = new Random();
    private static List<Identifier> GetIdentifiers(string identifierRow)
    {
        var keyValues = identifierRow.Split(new[] { '[', ']' },
StringSplitOptions.RemoveEmptyEntries);
        if (keyValues.Length % 3 != 0)
        {
            throw new Exception("should be 3 values in a row");
        }
        var result = new List<Identifier>();

        for (var i = 0; i < keyValues.Length; i += 3)
        {
            result.Add(new Identifier(keyValues[i], keyValues[i + 1], keyValues[i
+ 2]));
        }
        return result;
    }
    private static List<Point> GetPredicates(List<Token> tokens)
    {
        return tokens.Where(x => x._type == TokenType.Identifier)
            .Select(x => Point.Parse(x._lexeme))
            .Where(x => x.Type == VariableType.Context || x.Type ==
VariableType.Row).ToList();
    }
}
```

```

    }

    private static List<string> GetColumns(List<Point> predicates,
VariableType variableType)
    {
        return predicates
            .Where(x => x.Type == variableType)
            .Select(x => x.Value)
            .Distinct()
            .ToList();
    }

    private static Dictionary<string, object> GetRowValues(string
sqlEntityName, List<Identifier> identifiers, List<string> columns)
    {
        var whereStatement = string.Join(" and ", identifiers.Select(x =>
x.Column + " = @" + x.Column));

        var columnString = string.Join(", ", columns);

        var sqlstringRequest = "select " + columnString + " from " +
sqlEntityName + " where " + whereStatement;

        var result = new Dictionary<string, object>();

        using (var connection = new SqlConnection(ConnectionString))
        {
            SqlCommand cmd = new SqlCommand(sqlstringRequest, connection)
{ CommandType = CommandType.Text };

            identifiers.ForEach(x =>
            {
                cmd.Parameters.AddWithValue(@" + x.Column,
TypeDescriptor.GetConverter(Operations.types[x.Type]).ConvertFromString(x.
Value));
            });

            cmd.Connection.Open();

```

```

        SqlDataReader reader = cmd.ExecuteReader();
        try
        {
            while (reader.Read())
            {
                for (var i = 0; i < reader.FieldCount; i++)
                {
                    result.Add(reader.GetName(i), reader.GetValue(i));
                }
            }
        }
        finally
        {
            cmd.Connection.Close();
            reader.Close();
        }
    }

    return result;
}

[SqlFunction(DataAccess = DataAccessKind.Read)]
public static bool ExecuteStaticPredicate
    (string expressions,

     int id1, bool boolType1, int intType1, string stringType1, DateTime
dateTimeType1, DateTimeOffset datetimeOffsetType1, TimeSpan timeType1,
Guid guidType1,

     int id2, bool boolType2, int intType2, string stringType2, DateTime
dateTimeType2, DateTimeOffset datetimeOffsetType2, TimeSpan timeType2,
Guid guidType2)
    {

```

```

var tokens = ReversePolishNotation.GetTokens(expressions);
var dict = new Dictionary<string, object>()
{
    {"R.BoolType",boolType1 }, {"R.IntType",id1 },
    {"R.StringType",stringType1 }, {"C.StringType",stringType2 },
    {"R.DateTimeType",dateTimeType1 },
    {"R.DateTimeOffsetType",datetimeOffsetType1 },
    {"R.TimeType",timeType1 }, {"R.GuidType",guidType1 },
    {"C.BoolType",boolType2 }, {"C.IntType",id2 },
    {"C.DateTimeType",dateTimeType2 },
    {"C.DateTimeOffsetType",datetimeOffsetType2 },
    {"C.TimeType",timeType2 }, {"C.GuidType",guidType2 },
};
return ReversePolishNotation.Evaluate(tokens, dict);
}

[SqlFunction(DataAccess = DataAccessKind.Read)]
public static bool ExecutePredicate(string currentTableName, string
contextTableName, string expressions,
    string rowIdentifierKeys, string contextIdentifierKeys)
{
    var tokens = ReversePolishNotation.GetTokens(expressions);
    var predicates = GetPredicates(tokens);
    var rowIdentifiers = GetIdentifiers(rowIdentifierKeys);
    var rowColumns = GetColumns(predicates, VariableType.Row);
    var contextIdentifiers = GetIdentifiers(contextIdentifierKeys);
    var contextColumns = GetColumns(predicates, VariableType.Context);
    var resultValues = new Dictionary<string, object>();
    if(rowColumns.Count > 0)

```

```

    {
        resultValues = GetRowValues(currentTableName, rowIdentifiers,
rowColumns).ToDictionary(key => "R." + key.Key, value => value.Value);
    }
    if(contextColumns.Count > 0)
    {
        var contextValues = GetRowValues(contextTableName,
contextIdentifiers, contextColumns).ToDictionary(key => "C." + key.Key, value
=> value.Value);

        foreach (var sqlResult in contextValues)
        {
            resultValues.Add(sqlResult.Key, sqlResult.Value);
        }
    }
    return ReversePolishNotation.Evaluate(tokens, resultValues);
}
}

```

Приложение В

```
public class Scanner
{
    public static readonly IDictionary<string, TokenType> Keywords = new
Dictionary<string, TokenType>()
    {
        {"and", TokenType.And},
        {"false", TokenType.False},
        {"nil", TokenType.Nil},
        {"or", TokenType.Or},
        {"true", TokenType.True},
        {"as", TokenType.As},
        {"like", TokenType.Like }
    };

    public static readonly List<TokenType> Operations = new
List<TokenType>()
    {
        TokenType.Minus, TokenType.Plus, TokenType.Star, TokenType.Slash,
        TokenType.Less, TokenType.LessEqual, TokenType.BangEqual,
        TokenType.Equal, TokenType.Greater, TokenType.GreaterEqual,
        TokenType.As, TokenType.And, TokenType.Or, TokenType.Like
    };

    private readonly string _source;
    private readonly List<Token> _tokens = new List<Token>();
    private int _start = 0;
    private int _current = 0;

    public Scanner(string source)
```

```

{
    _source = source;
}

public List<Token> ScanTokens()
{
    while (!IsAtEnd())
    {
        _start = _current;
        ScanToken();
    }
    return _tokens;
}

private void ScanToken()
{
    var c = Advance();
    switch (c)
    {
        case '(':
            AddToken(TokenType.LeftParen);
            break;
        case ')':
            AddToken(TokenType.RightParen);
            break;
        case '-':
            AddToken(TokenType.Minus);
            break;
        case '+':
            AddToken(TokenType.Plus);

```



```

        break;
    case '*':
        AddToken(TokenType.Star);
        break;
    case '!':
        AddToken(Match('=') ? TokenType.BangEqual : TokenType.Bang);
        break;
    case '=':
        AddToken(TokenType.Equal);
        break;
    case '<':
        AddToken(Match('=') ? TokenType.LessEqual : TokenType.Less);
        break;
    case '>':
        AddToken(Match('=') ? TokenType.GreaterEqual :
TokenType.Greater);
        break;
    case '/':
        if (Match('/'))
        {
            while (Peek() != '\n' && !IsAtEnd())
            {
                Advance();
            }
        }
        else
        {
            AddToken(TokenType.Slash);

```

```

        }
        break;
    case ' ':
    case '\r':
    case '\t':
        // Ignore whitespace.
        break;
    case '\n':
        return;
    case '"':
        ParseString();
        break;
    default:
        if (IsDigit(c))
        {
            Number();
        }
        else if (IsAlpha(c))
        {
            Identifier();
        }
        else
        {
            Console.WriteLine("Unexpected character.");
        }
        break;
    }
}

```

```

private void Identifier()
{
    while (IsAlphaNumeric(Peek())) Advance();
    var text = Substring(_source, _start, _current);
    TokenType type;
    type = Keywords.ContainsKey(text) ? Keywords[text] :
TokenType.Identifier;
    AddToken(type, text);
}
private void Number()
{
    while (IsDigit(Peek())) Advance();
    if (Peek() == '.' && IsDigit(PeekNext()))
    {
        Advance();
        while (IsDigit(Peek())) Advance();
        AddToken(TokenType.Double,
            double.Parse(Substring(_source, _start, _current)));
        return;
    }
    short shortNum;
    int intNum;
    long longNum;
    var str = Substring(_source, _start, _current);
    if (short.TryParse(str, out shortNum))
    {
        AddToken(TokenType.ShortInt, shortNum);
        return;
    }

```

```

    }
    if (int.TryParse(str, out intNum))
    {
        AddToken(TokenType.Int, intNum);
        return;
    }
    if (long.TryParse(str, out longNum))
    {
        AddToken(TokenType.BigInt, longNum);
    }
    AddToken(TokenType.String, str);
}

private string Substring(string line, int start, int end)
{
    return line.Substring(start, end - start);
}

private void ParseString()
{
    while (Peek() != "" && !IsAtEnd())
    {
        if (Peek() == '\n') return;
        Advance();
    }
    if (IsAtEnd())
    {
        Console.WriteLine("Unterminated string.");
        return;
    }
}

```

```

    Advance();

    var value = Substring(_source, _start + 1, _current - 1);
    AddToken(TokenType.String, value);
}

private bool Match(char expected)
{
    if (IsAtEnd())
    {
        return false;
    }

    if (_source[_current] != expected)
    {
        return false;
    }

    _current++;

    return true;
}

private char Peek()
{
    if (IsAtEnd())
    {
        return '\\';
    }

    return _source[_current];
}

private char PeekNext()
{
    if (_current + 1 >= _source.Length)

```

```

    {
        return '\\';
    }
    return _source[_current + 1];
}

private bool IsAlpha(char c)
{
    return c >= 'a'
        && c <= 'z' || c >= 'A'
        && c <= 'Z' || c == '.';
}

private bool IsAlphaNumeric(char c)
{
    return IsAlpha(c) || IsDigit(c);
}

private bool IsDigit(char c)
{
    return c >= '0' && c <= '9';
}

private bool IsAtEnd()
{
    return _current >= _source.Length;
}

private char Advance()
{
    _current++;
    return _source[_current - 1];
}

```

```
}

private void AddToken(TokenType type)
{
    AddToken(type, null);
}

private void AddToken(TokenType type, object literal)
{
    var text = Substring(_source, _start, _current);
    _tokens.Add(new Token(type, text, literal));
}
}
```

Приложение Г

```
public class Point
{
    public VariableType Type { get; set; }
    public string Value { get; set; }
    public Point(VariableType type, string value)
    {
        Type = type;
        Value = value;
    }
    public static Point Parse(string row)
    {
        if (row.StartsWith("C."))
        {
            return new Point(VariableType.Context, row.Substring(2));
        }
        else if (row.StartsWith("R."))
        {
            return new Point(VariableType.Row, row.Substring(2));
        }
        else
        {
            return new Point(VariableType.Constant, row);
        }
    }
}

public class ReversePolishNotation
```



```

{
    private static readonly TokenType[][] _tokenTypes = new[]
    {
        new[] { TokenType.LeftParen, TokenType.RightParen },
        new[] { TokenType.And, TokenType.Or },
        new[]
        {
            TokenType.Equal, TokenType.BangEqual, TokenType.Greater,
            TokenType.GreaterEqual, TokenType.Less,
            TokenType.LessEqual, TokenType.Like
        },
        new[] { TokenType.Minus, TokenType.Plus },
        new[] { TokenType.Star, TokenType.Slash },
        new[] { TokenType.As }
    };
    public static int GetPriority(TokenType token)
    {
        for (var i = 0; i < _tokenTypes.Length; i++)
        {
            if (_tokenTypes[i].Contains(token))
            {
                return i;
            }
        }
        return -1;
    }
    public static List<Token> GetTokens(string expression)
    {

```

```

var tokens = new Scanner(expression).ScanTokens();
var resultTokens = new List<Token>();
var leftTokens = new List<TokenType>
{
    TokenType.And,
    TokenType.Or,
    TokenType.LeftParen,
    TokenType.BangEqual,
    TokenType.Equal,
    TokenType.Greater,
    TokenType.GreaterEqual,
    TokenType.LessEqual,
    TokenType.Less,
};
for (var i = 0; i < tokens.Count; i++)
{
    if (tokens[i]._type == TokenType.Minus)
    {
        if (i == 0 || leftTokens.Contains(tokens[i - 1]._type))
        {
            resultTokens.Add(new Token(TokenType.ShortInt, "0",
(short)0));
        }
    }
    resultTokens.Add(tokens[i]);
}
return resultTokens;
}

```

```

public static List<Token> ConvertToPrefixVersion(List<Token> tokens)
{
    var stack = new Stack<Token>();
    var result = new List<Token>();
    foreach (var node in tokens)
    {
        if (node.IsOperator())
        {
            while (stack.Count != 0 && GetPriority(node._type) <=
GetPriority(stack.Peek()._type))
                result.Add(stack.Pop());
            stack.Push(node);
        }
        else if (node._type == TokenType.RightParen)
        {
            while (stack.Count != 0 && stack.Peek()._type !=
TokenType.LeftParen)
                result.Add(stack.Pop());
            if (stack.Count != 0)
                stack.Pop();
            else
                throw new Exception("there was not ");
        }
        else if (node._type == TokenType.LeftParen)
        {
            stack.Push(node);
        }
        else
    }
}

```

```

        {
            result.Add(node);
        }
    }
    while (stack.Count != 0)
        result.Add(stack.Pop());
    return result;
}

public static object GetValue(object first, object second, Token oper)
{
    switch (oper._type)
    {
        case TokenType.Plus:
            return Operations.Add(first, second);
        case TokenType.As:
            return Operations.As(first, second);
        case TokenType.Minus:
            return Operations.Subtract(first, second);
        case TokenType.Star:
            return Operations.Multiply(first, second);
        case TokenType.Slash:
            return Operations.Divide(first, second);
        case TokenType.Greater:
            return Operations.More(first, second);
        case TokenType.Less:
            return Operations.Less(first, second);
        case TokenType.GreaterEqual:

```

```

        return Operations.Equal(first, second) || Operations.More(first,
second);
    case TokenType.LessEqual:
        return Operations.Equal(first, second) || Operations.Less(first,
second);
    case TokenType.Equal:
        return Operations.Equal(first, second);
    case TokenType.BangEqual:
        return !Operations.Equal(first, second);
    case TokenType.And:
        return Operations.And(first, second);
    case TokenType.Or:
        return Operations.Or(first, second);
    }
    return null;
}

public static bool Evaluate(List<Token> tokens, Dictionary<string, object>
dict = null)
{
    dict = dict ?? new Dictionary<string, object>();
    var nodes = ConvertToPrefixVersion(tokens);
    var stack = new Stack<object>();
    foreach (var node in nodes)
        if (node.IsOperator())
        {
            var second = stack.Pop();
            var first = stack.Pop();
            var result = GetValue(first, second, node);
            stack.Push(result);
        }
}

```

```

    }

    else if (node._type == TokenType.Identifier &&
dict.ContainsKey(node._lexeme))
    {
        var identifierValue = dict[node._lexeme];
        stack.Push(identifierValue);
    }
    else
    {
        stack.Push(node._literal);
    }
    var res = stack.Pop();
    if (res.GetType() != typeof(bool))
        throw new Exception("expression should return boolean");
    return (bool)res;
}
}

```

Приложение Д

```
public static class Operations
{
    public static readonly Dictionary<string, Type> types = new
Dictionary<string, Type>
    {
        {"byte", typeof(byte)},
        {"short", typeof(short)},
        {"int", typeof(int)},
        {"long", typeof(long)},
        {"double", typeof(double)},
        {"float", typeof(float)},
        {"byte[]", typeof(byte[])},
        {"bool", typeof(bool)},
        {"string", typeof(string)},
        {"datetime", typeof(DateTime)},
        {"timespan", typeof(TimeSpan)},
        {"datetimeoffset", typeof(DateTimeOffset)},
        {"guid", typeof(Guid) }
    };
    public static readonly List<Type> TypeConvert = new List<Type>()
    {
        typeof(byte), typeof(short), typeof(int), typeof(long), typeof(double)
    };
};
```

```

public static Tuple<object, object> ConvertToOneType(object first, object
second)
{
    if (first.GetType() == second.GetType())
        return new Tuple<object, object>(first, second);

    if (!TypeConvert.Contains(first.GetType()) ||
!TypeConvert.Contains(second.GetType()))
    {
        throw new Exception("Impossible to make implicit conversion");
    }

    object firstResult;
    object secondResult;

    if (TypeConvert.IndexOf(first.GetType()) >
TypeConvert.IndexOf(second.GetType()))
    {
        secondResult = Convert.ChangeType(second, first.GetType());
        firstResult = first;
    }
    else
    {
        secondResult = second;
        firstResult = Convert.ChangeType(first, second.GetType());
    }

    return new Tuple<object, object>(firstResult, secondResult);
}

```



```

public static object As(object first, object second)
{
    if (second.GetType() != typeof(string))
        throw new Exception("wrong type name");
    if (first is string)
    {
        if (!types.ContainsKey((string)second))
            throw new Exception("wrong type");
        var typeToConvert = types[(string)second];
        if (typeToConvert == null)
            throw new Exception("there was not found a such type");
        var result =
TypeDescriptor.GetConverter(typeToConvert).ConvertFromString((string)first);
        return result;
    }
    var type = types[(string)second];
    var newValue = Convert.ChangeType(first, type);

    return newValue;
}

public static object Add(object first, object second)
{
    var tuple = (first is DateTime && second is TimeSpan)
        ? new Tuple<object, object>(first, second)
        : ConvertToOneType(first, second);

```

```

first = tuple.Item1;
second = tuple.Item2;
if (first is string)
    return (string)first + (string)second;
if (first is short)
    return (short)first + (short)second;
if (first is int)
    return (int)first + (int)second;
if (first is long)
    return Convert.ToInt64(first) + Convert.ToInt64(second);
if (first is double)
    return (double)first + (double)second;
if (first is float)
    return (float)first + (float)second;
if (first is DateTime && second is TimeSpan)
    return (DateTime)first + (TimeSpan)second;
if (first is TimeSpan && second is TimeSpan)
    return (TimeSpan)first + (TimeSpan)second;
throw new Exception("It is impossible to add these values");
}

public static object Subtract(object first, object second)
{
    var tuple = (first is DateTime && second is TimeSpan)
        ? new Tuple<object, object>(first, second)
        : ConvertToOneType(first, second);

```

```

    first = tuple.Item1;
    second = tuple.Item2;
    if (first is short)
        return (short)first - (short)second;
    if (first is int)
        return (int)first - (int)second;
    if (first is long)
        return Convert.ToInt64(first) - Convert.ToInt64(second);
    if (first is double)
        return (double)first - (double)second;
    if (first is float)
        return (float)first - (float)second;
    if (first is TimeSpan)
        return (TimeSpan)first - (TimeSpan)second;
    if (first is DateTime && second is TimeSpan)
        return (DateTime)first - (TimeSpan)second;
    throw new Exception("It is impossible to SUBTRACT these values");
}

public static object Multiply(object first, object second)
{
    var tuple = ConvertToOneType(first, second);
    first = tuple.Item1;
    second = tuple.Item2;
    if (first is short)
        return (short)first * (short)second;

```

```

    if (first is int)
        return (int)first * (int)second;
    if (first is long)
        return Convert.ToInt64(first) * Convert.ToInt64(second);
    if (first is double)
        return (double)first * (double)second;
    if (first is float)
        return (float)first * (float)second;

    throw new Exception("It is impossible to MULTIPLY these values");
}

public static object Divide(object first, object second)
{
    var tuple = ConvertToOneType(first, second);
    first = tuple.Item1;
    second = tuple.Item2;
    if (first is short)
        return (short)first / (short)second;
    if (first is int)
        return (int)first / (int)second;
    if (first is long)
        return Convert.ToInt64(first) / Convert.ToInt64(second);
    if (first is double)
        return (double)first / (double)second;
    if (first is float)

```

```

        return (float)first / (float)second;

        throw new Exception("It is impossible to DIVIDE these values");
    }

    public static bool More(object first, object second)
    {
        var tuple = ConvertToOneType(first, second);
        first = tuple.Item1;
        second = tuple.Item2;

        if (first is short)
            return (short)first > (short)second;
        if (first is int)
            return (int)first > (int)second;
        if (first is long)
            return Convert.ToInt64(first) > Convert.ToInt64(second);
        if (first is double)
            return (double)first > (double)second;
        if (first is float)
            return (float)first > (float)second;
        if (first is DateTime)
            return (DateTime)first > (DateTime)second;
        if (first is TimeSpan)
            return (TimeSpan)first > (TimeSpan)second;

        throw new Exception("It is impossible to make MORE these values");
    }

```

```

public static bool Less(object first, object second)
{
    var tuple = ConvertToOneType(first, second);
    first = tuple.Item1;
    second = tuple.Item2;
    if (first is short)
        return (short)first < (short)second;
    if (first is int)
        return (int)first < (int)second;
    if (first is long)
        return Convert.ToInt64(first) < Convert.ToInt64(second);
    if (first is double)
        return (double)first < (double)second;
    if (first is float)
        return (float)first < (float)second;
    if (first is DateTime)
        return (DateTime)first < (DateTime)second;
    if (first is TimeSpan)
        return (TimeSpan)first < (TimeSpan)second;
    throw new Exception("!!1");
}

public static bool And(object first, object second)
{
    if (first is bool && second is bool)
        return (bool)first && (bool)second;

```

```

        throw new Exception("mismatched types");
    }

    public static bool Or(object first, object second)
    {
        if (first is bool && second is bool)
            return (bool)first || (bool)second;

        throw new Exception("mismatched types");
    }

    public static bool Equal(object first, object second)
    {
        var tuple = ConvertToOneType(first, second);
        first = tuple.Item1;
        second = tuple.Item2;
        if (first is byte[])
            return ((byte[])first).SequenceEqual((byte[])second);

        var result = Equals(first, second);

        return result;
    }
}

```