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 Row Level Security 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.

Содержание

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 становиться 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. Это позволяет реализовать политику безопасности на уровне сервера базы данных, причём приложение, работающее с данным, поставляемыми 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 курсовой 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

1. Безопасность на уровне строк

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 неравноценны 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 функция 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. Данных 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 дынных. Этот путь предполагает более детальную настройку ограничений прав доступа к записям. В отличии 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.). Также, RLS обычно имеет дело не со статичной политикой безопасности, в отличии от FLS, где изменение структуры таблицы происходит достаточно редко. Поэтому часто разработчики прибегают к созданию фреймворка 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 данную 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 доступ к только к тем данным клиента, которые относятся к их компании.

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 надежной 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.

Предикаты фильтров и блокировки, а также 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 наложить предикат на таблицу, которая уже имеет один определенный предикат для данной операции (независимо от того, включен он или выключен), приведет к ошибке.
- Что касается политик безопасности, привязанной к схеме, 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.

1.2 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, которых больше текущего. 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.

1.3 Способы применения

Ниже приведены примеры конструирования использования RLS.

- Больницы могут создать 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.
- Банк может создать политику для ограничения доступа 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.

- Мультитенантное 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 иметь 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.

1.4 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 тоже 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.

Если политика безопасности создается с использованием команды SCHEMABINDING = OFF, то для отправки запроса в целевую таблицу пользователям потребуется разрешение SELECT или EXECUTE в функции предиката и любых дополнительных таблицах, представлениях и функциях, используемых в функции предиката. Если политика безопасности создана с использованием SCHEMABINDING = ON (по умолчанию), при запросе целевой таблицы пользователями эти проверки разрешений не проводятся.

1.5 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 схемы для объектов RLS
- Необходимо выдавать минимальный набор прав для администратора политик безопасности, лучше всего чтобы у него не было возможно просмотреть 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/(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, в которых происходит 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

1.6 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 нефильтрованным данным, таким образом, он может вызвать утечку информации, которая в тоже 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 также, 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

2. Создания 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 операцией 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 для таблицу 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 получит и другим способам. 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 некоторое ограниченное множество 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 указанны 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>) [AND <role_n_restricted_restrictions >])

)

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 также, 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 большинтсве 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 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 = 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 = 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 быстродействие СУБД может оказаться недостаточным для вычисления предикатов безопасности на основе уже существующих атрибутов (особенно если таблицы большие и индексов мало).

3. Создания 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.

3.1 Подготовка базы данных для использования политик безопасности

Для использования механизма RLS в самом простом случае необходимо создать предикат, прикреплённый 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 строчке базы данных. К сожалёнию 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 возвращения 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 хранится связи - {текущий пользователь - текущая таблица, над которой выполняется запрос - список предикатов исполняемых над строками текущей таблицы}. Предикат задаётся с помощью простого выражения, рассмотренного позже.

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 количество 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 определится, 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.

```
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 добавления 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. В конце концов это совсем другая стратегия установки правил для базы данных. 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 строчки])))) = 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 применятся к каждой строчке указанной таблице при операции чтения. В основном нам также необходимо отказать в доступе на запись строчки, 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.

3.2 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 переводится 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 сравнивании 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 → литерал

| унарная операция

| бинарная операция

| группировка;

литерал → число | строка | "true" | "false" | "nil" ;

группировка → "("выражение")" ;

унарная операция → ("-" | "!") выражение;

бинарная операция → выражение оператор выражение;

оператор → "=" | "!=" | "<" | "<=" | ">" | ">="

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

число → 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.

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.

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.

3.3 Пример необходимости данного алгоритма

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/yyy, где xxx - это домен нашего приложения, а yyy - 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. Код REST сервиса приведён в приложении Г и Д.

Далее, как описано ранее, необходимо включить возможность подключения CRL 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.

3.4 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 1 и 2. Создание групп для двух городов показано на рисунках 3 и 4.

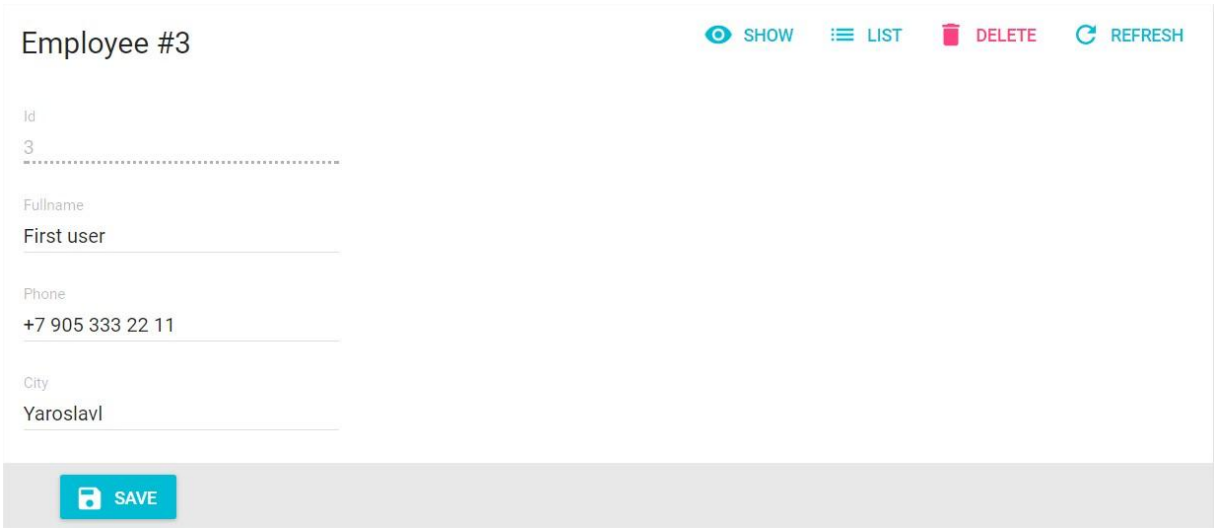

Рисунок 1

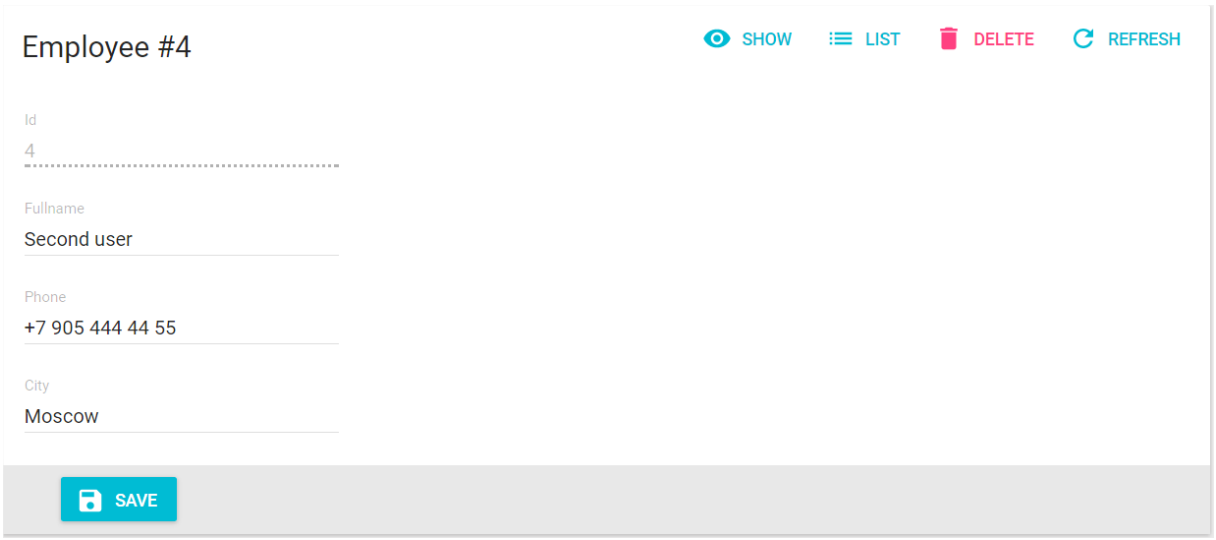

Рисунок 2

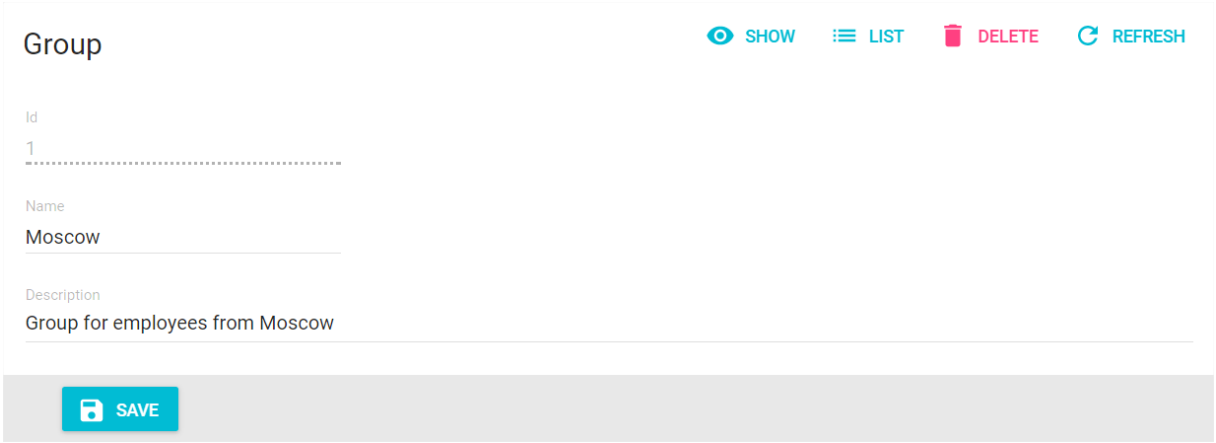

Рисунок 3

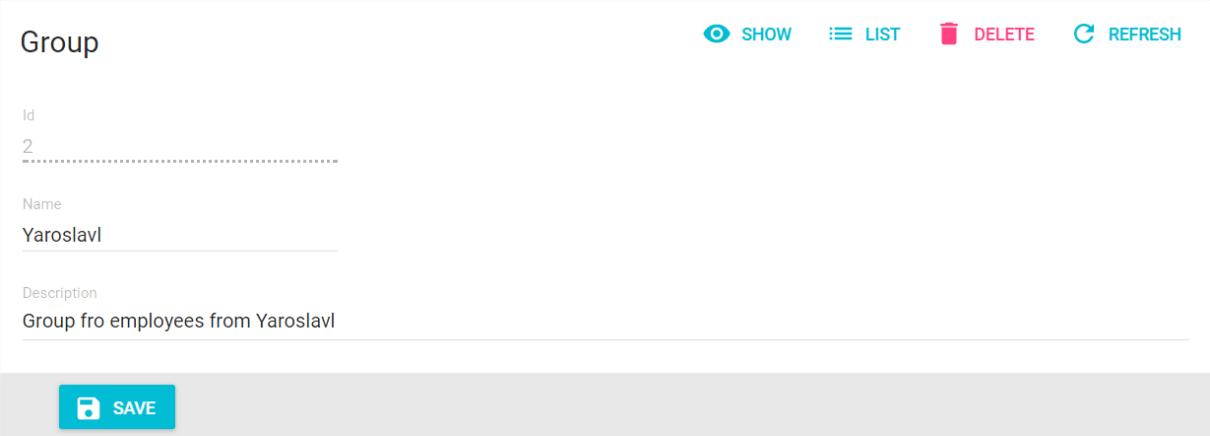

Рисунок 4

Оформим по заказу на каждого сотрудника. Создание заказов показано на рисунке 5 и 6.

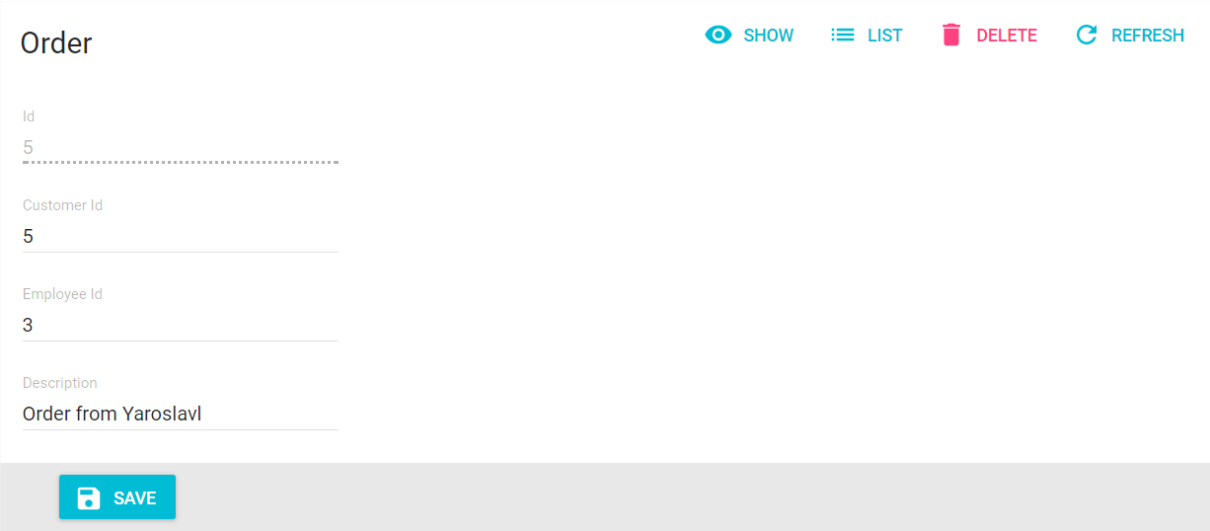

Рисунок 5

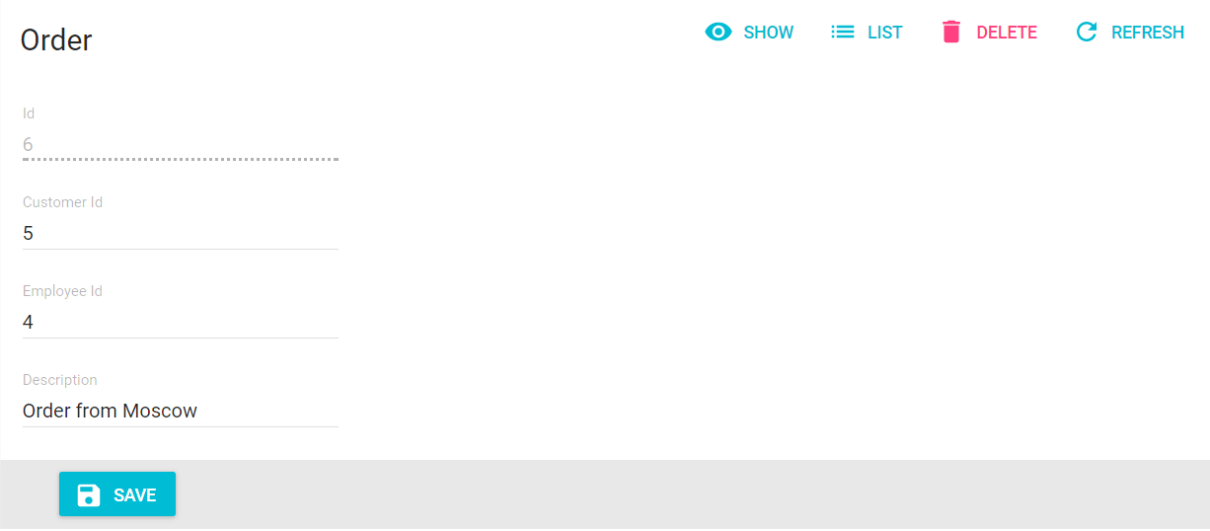

Рисунок 6

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 7.

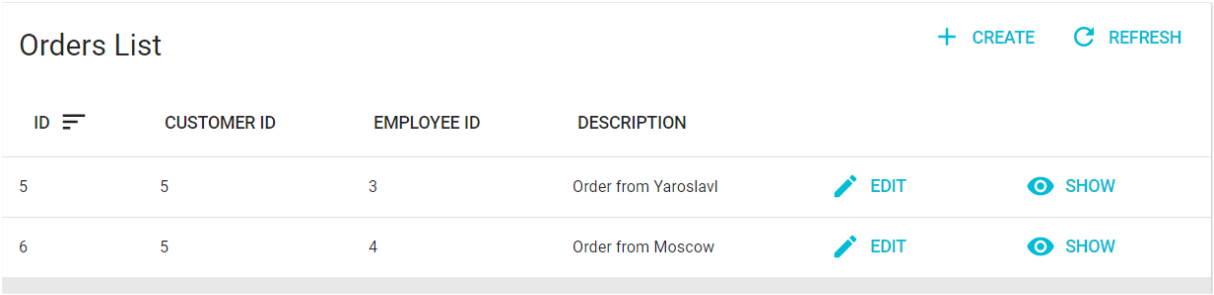

Рисунок 7

Создадим для каждой группы по примитивному предикату безопасности, указывающий на принадлежность к 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 8 и 9.

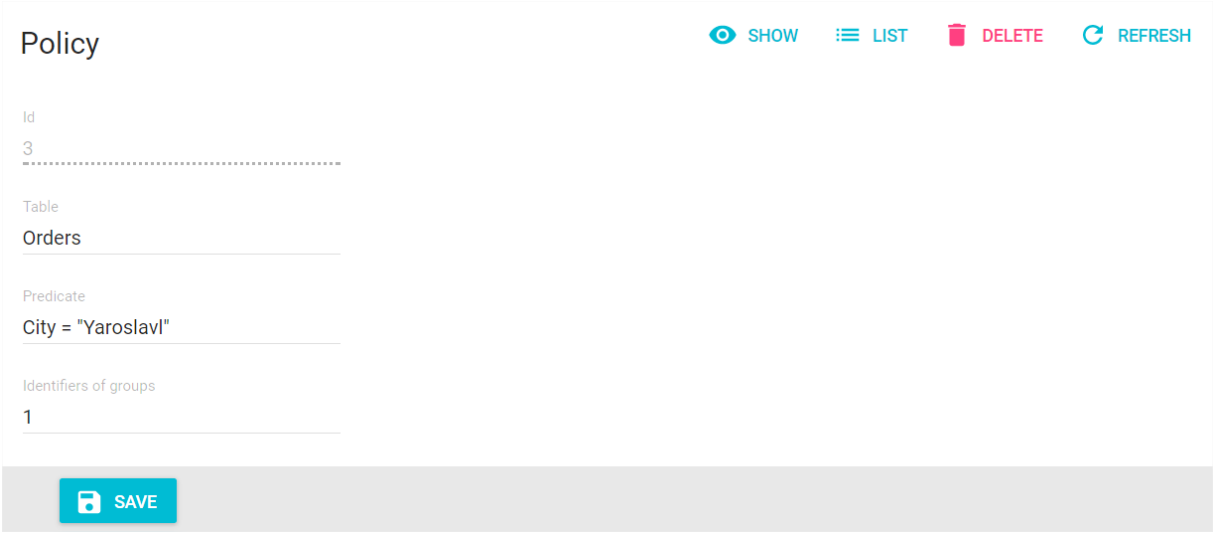

Рисунок 8

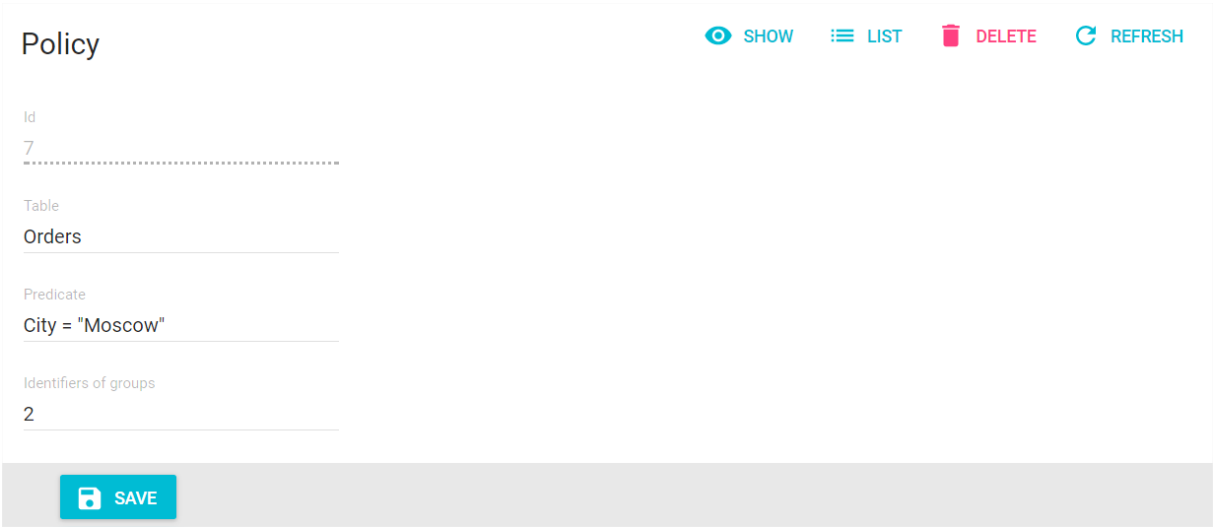

Рисунок 9

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 9 и 10.

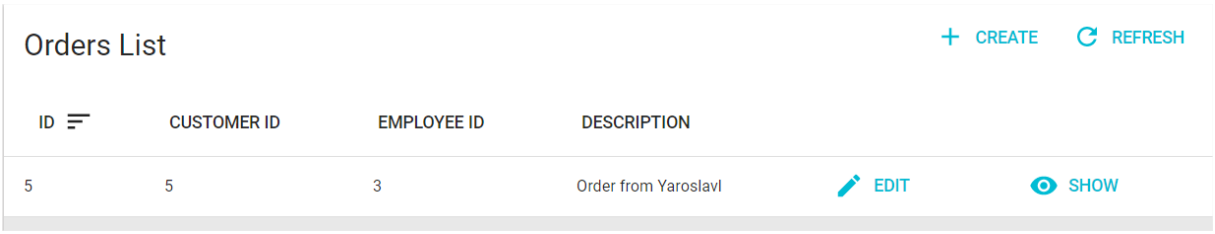

Рисунок 9

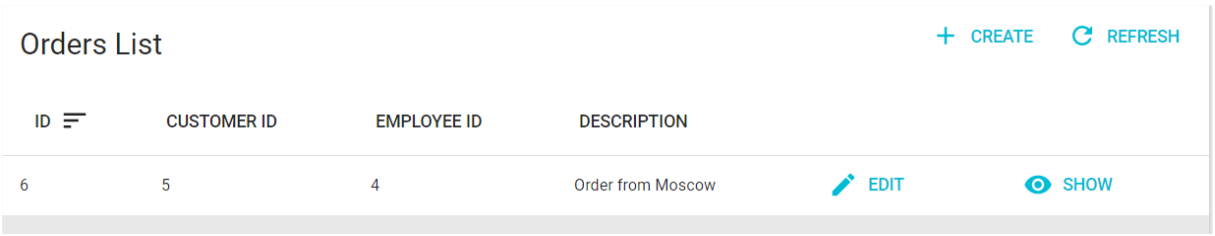

Рисунок 10

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.

Можно отметить, что создание политики безопасности с помощью встроенных функций гораздо проще и производительнее, так как оптимизатор запросов может повлиять на выполнение запроса, тем самым сократив количество ненужных операций.

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

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://rsdn.org/ - URL: http://rsdn.org/article/db/RowLevelSecurity.xml
6. https://msdn.microsoft.com - URL: https://msdn.microsoft.com/ru-ru/library/dn765131.aspx

Приложение А

```
public class CustomInterceptor : IDbConnectionInterceptor
   {
      public void Opened(DbConnection connection,
DbConnectionInterceptionContext interceptionContext)
      {
         // Set SESSION_CONTEXT to current UserId whenever EF opens a
connection
         try
         {
            var userId = HttpContext.Current.User.Identity.GetUserId();
            if (userId != null)
            {
               DbCommand cmd = connection.CreateCommand();
               DbParameter param = cmd.CreateParameter();
               param.Value = userId;
               param.ParameterName = "@UserId";
               cmd.Parameters.Add(param);
               cmd.CommandText = "EXEC sp_set_session_context
@key=N'UserId', @value=@UserId";
               cmd.ExecuteNonQuery();
            }
         }
         catch (System.NullReferenceException)
         {
            // If no user is logged in, leave SESSION_CONTEXT null (all rows
will be filtered)
         }
      }
```

```
        public void Opening(DbConnection connection,
DbConnectionInterceptionContext interceptionContext)       {  }

        public void BeganTransaction(DbConnection connection,
BeginTransactionInterceptionContext interceptionContext)    {   }

        ... другие методы интерфейса IDbConnectionInterceptor

    }

public class CustomConfiguration : DbConfiguration

    {

        public SessionContextConfiguration()

        {

            AddInterceptor(new CustomInterceptor());

        }

    }
```

Приложение Б

```
[SqlFunction(DataAccess = DataAccessKind.Read)]
   public static bool ExecutePredicate(string predicate, string rowValue){
      var predicates = predicate.Split(new[] { ',' },
StringSplitOptions.RemoveEmptyEntries);
      var predicateTuples = new List<Tuple>();
      foreach (var pred in predicates){
         var values = pred.Split(new[] { '=', ' ' },
StringSplitOptions.RemoveEmptyEntries);
         predicateTuples.Add(new Tuple(values[0], values[1])); }
      var rowValues = rowValue.Split(new[] { ',' },
StringSplitOptions.RemoveEmptyEntries);
      var rowTuples = new List<Tuple>();
      foreach (var value in rowValues){
         var values = value.Split(new[] { '=', ' ' },
StringSplitOptions.RemoveEmptyEntries);
         rowTuples.Add(new Tuple(values[0], values[1].Trim('')));}
      var contextDict = predicateTuples.Where(x => x.Value.IndexOf('') == -
1).ToDictionary(t => t.Value, y => '');
      var rowDict = rowTuples.ToDictionary(x => x.Variable, y => y.Value);
      var contextQuery = "select " + string.Join(", ", contextDict.Keys.Select(x
=> string.Format("SESSION_CONTEXT(N'{0}') as {0}", x)));
      if (contextDict.Keys.Count != 0)    {
         using (SqlConnection connection = new SqlConnection("context
connection=true")){
            SqlCommand cmd = new SqlCommand();
            SqlDataReader reader;
            cmd.CommandText = contextQuery;
            cmd.CommandType = CommandType.Text;
            cmd.Connection = connection;
```

```
            cmd.Connection.Open();
            reader = cmd.ExecuteReader();
            try{
               while (reader.Read()){
                  contextDict = contextDict.Keys.ToDictionary(x => x, y =>
reader[y].ToString());}}
            finally{
               cmd.Connection.Close();
               reader.Close();}}}
      foreach (var keyValue in predicateTuples) {
         var key = "";
         if (keyValue.Variable.IndexOf("") == -1) {
            key = rowDict[keyValue.Variable];
         }
         else{
            key = keyValue.Variable.Trim("");}
         var value = "";
         if (keyValue.Value.IndexOf("") == -1){
            value = contextDict[keyValue.Value];}
         else{
            value = keyValue.Value.Trim("");}

         if (key != value){
            return false;
         }
      }
      return true;
   }
```

Приложение В

```
create PROCEDURE setInitialContext(
   @UserId int)
AS
        SET NOCOUNT ON;
        declare @FullName nvarchar(150);
        declare @City nvarchar(20);
        declare @Phone nvarchar(24);
   select @FullName = FullName, @City = City, @Phone = Phone from
Employees where id = @UserId
        EXEC sp_set_session_context 'FullName', @FullName;
        EXEC sp_set_session_context 'UserId', @UserId;
        EXEC sp_set_session_context 'City', @City;
        EXEC sp_set_session_context 'Phone', @Phone;
GO


drop function if exists [Security].getUserAccessClr
drop ASSEMBLY if exists Parser
create ASSEMBLY Parser FROM 'C:\Users\Сергей\Documents\Visual Studio
2015\Projects\ClassLibrary1\ClassLibrary1\Parser.dll';
GO
create FUNCTION [Security].getUserAccessClr(@Predicate nvarchar(50))
RETURNS bit
 AS EXTERNAL NAME Parser.ContextParser.ExecutePredicate;
GO


CREATE FUNCTION [Security].getUserAccess(@TableName nvarchar(50))
RETURNS bit
```

```
AS



drop function if exists getUserAccess
go
CREATE FUNCTION getUserAccess(@TableName nvarchar(50))
RETURNS bit
 WITH SCHEMABINDING
AS
-- Returns the stock level for the product.
BEGIN
 DECLARE @predicates nvarchar(4000);
 Declare @result bit;
 select @predicates = COALESCE(@predicates + ',', '') + dbo.Predicates.Value
from
          dbo.Predicates join  dbo.Policies
         on  dbo.Predicates.id =  dbo.Policies.PredicateId
         and  dbo.Policies.TableName = @TableName
         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(@predicates)
        end
        return @result
END;
drop function if exists getUserAccessClr
go
drop ASSEMBLY if exists Parser
go
create ASSEMBLY Parser FROM 'C:\Users\Сергей\Documents\Visual Studio 2015\Projects\ClassLibrary1\ClassLibrary1\Parser.dll';
GO
create FUNCTION getUserAccessClr(@Predicate nvarchar(4000)) RETURNS bit
 AS EXTERNAL NAME Parser.ContextParser.ExecutePredicate;
GO
drop function if exists [securityPredicate]
go
create FUNCTION securityPredicateOrders(@emId int)
   RETURNS TABLE
        WITH SCHEMABINDING
AS
   RETURN SELECT 1 as Resu where ((select dbo.getUserAccess('Orders')) = 1)
go
create SECURITY POLICY dbo.[OrdersPolicy]
ADD FILTER PREDICATE dbo.securityPredicateOrders(EmployeeId)
   ON [dbo].[Orders]
WITH (STATE = ON);
```

Приложение Г

```
import config from '../../configuration'
import { queryParameters, fetchJson } from '../util/fetch';
import {
  GET_LIST,
  GET_ONE,
  GET_MANY,
  GET_MANY_REFERENCE,
  CREATE,
  UPDATE,
  DELETE,
} from './types';

/**
 * Maps admin-on-rest queries to a json-server powered REST API
 *
 * @see https://github.com/typicode/json-server
 * @example
 * GET_LIST     => GET http://my.api.url/posts?_sort=title&_order=ASC&_start=0&_end=24
 * GET_ONE      => GET http://my.api.url/posts/123
 * GET_MANY     => GET http://my.api.url/posts/123, GET http://my.api.url/posts/456, GET http://my.api.url/posts/789
 * UPDATE       => PUT http://my.api.url/posts/123
 * CREATE       => POST http://my.api.url/posts/123
 * DELETE       => DELETE http://my.api.url/posts/123
 */
export default (apiUrl, httpClient = fetchJson) => {
```

```
/**
 * @param {String} type One of the constants appearing at the top if this file, e.g. 'UPDATE'
 * @param {String} resource Name of the resource to fetch, e.g. 'posts'
 * @param {Object} params The REST request params, depending on the type
 * @returns {Object} { url, options } The HTTP request parameters
 */
const convertRESTRequestToHTTP = (type, resource, params) => {
  let url = '';
  const options = {};
  switch (type) {
    case GET_LIST:
    {
      const { page, perPage } = params.pagination;
      const { field, order } = params.sort;
      const query = {
        ...params.filter,
        _sort: field,
        _order: order,
        _start: (page - 1) * perPage,
        _end: page * perPage,
      };
      url = `${apiUrl}/${resource}?${queryParameters(query)}`;
      break;
    }
    case GET_ONE:
      url = `${apiUrl}/${resource}/${params.id}`;
      break;
```

```
case GET_MANY_REFERENCE:
{
  const { page, perPage } = params.pagination;
  const { field, order } = params.sort;
  const query = {
    ...params.filter,
    [params.target]: params.id,
    _sort: field,
    _order: order,
    _start: (page - 1) * perPage,
    _end: page * perPage,
  };
  url = `${apiUrl}/${resource}?${queryParameters(query)}`;
  break;
}
case UPDATE:
  url = `${apiUrl}/${resource}/${params.id}`;
  options.method = 'PUT';
  options.body = JSON.stringify(params.data);
  break;
case CREATE:
  url = `${apiUrl}/${resource}`;
  options.method = 'POST';
  options.body = JSON.stringify(params.data);
  break;
case DELETE:
  url = `${apiUrl}/${resource}/${params.id}`;
  options.method = 'DELETE';
```

```
      break;
    default:
      throw new Error(`Unsupported fetch action type ${type}`);
  }
  const isAuthenticated = !localStorage.getItem('not_authenticated')
  const userName = localStorage.getItem('username')
  options.user = {
    authenticated: isAuthenticated,
    token: userName
  }
  return {url, options};
};


/**
 * @param {Object} response HTTP response from fetch()
 * @param {String} type One of the constants appearing at the top if this file, e.g. 'UPDATE'
 * @param {String} resource Name of the resource to fetch, e.g. 'posts'
 * @param {Object} params The REST request params, depending on the type
 * @returns {Object} REST response
 */
const convertHTTPResponseToREST = (response, type, resource, params) => {
  const { headers, json } = response;
  switch (type) {
    case GET_LIST:
    case GET_MANY_REFERENCE:
      // if (!headers.has('x-total-count')) {
```

```
        //     throw new Error('The X-Total-Count header is missing in the HTTP Response. The jsonServer REST client expects responses for lists of resources to contain this header with the total number of results to build the pagination. If you are using CORS, did you declare X-Total-Count in the Access-Control-Expose-Headers header?');
        // }
        return {
          data: json,
          total: 12,
        };
      case CREATE:
        return {data: {...params.data, id: json.id}};
      default:
        return {data: json};
    }
  };

  /**
   * @param {string} type Request type, e.g GET_LIST
   * @param {string} resource Resource name, e.g. "posts"
   * @param {Object} payload Request parameters. Depends on the request type
   * @returns {Promise} the Promise for a REST response
   */
  return (type, resource, params) => {
    // json-server doesn't handle WHERE IN requests, so we fallback to calling GET_ONE n times instead
    if (type === GET_MANY) {
      return Promise.all(params.ids.map(id => httpClient(`${apiUrl}/${resource}/${id}`)))
        .then(responses => ({data: responses.map(response => response.json)}));
```

```
    }

    const { url, options } = convertRESTRequestToHTTP(type, resource,
params);

    return httpClient(config.host + url, options)

      .then(response => convertHTTPResponseToREST(response, type, resource,
params));

  };

};
```

Приложение Д

```
import HttpError from './HttpError';

export const fetchJson = (url, options = {}) => {
    const requestHeaders = options.headers || new Headers({
        Accept: 'application/json'
    });
    if (!(options && options.body && options.body instanceof FormData)) {
        requestHeaders.set('Content-Type', 'application/json');
    }
    if (options.user && options.user.authenticated && options.user.token) {
        requestHeaders.set('Authorization', options.user.token);
    }
    return fetch(url, { ...options, headers: requestHeaders })
        .then(response => response.text().then(text => ({
            status: response.status,
            statusText: response.statusText,
            headers: response.headers,
            body: text
        })))
        .then(({ status, statusText, headers, body }) => {
            let json;
            try {
                json = JSON.parse(body);
            } catch (e) {
                // not json, no big deal
            }
```

```
        if (status < 200 || status >= 300) {
            return Promise.reject(new HttpError((json && json.message) || statusText, status));
        }
        return { status, headers, body, json };
    });
};

export const queryParameters = data => Object.keys(data)
  .map(key => [key, data[key]].map(encodeURIComponent).join('='))
  .join('&');
```