**Row-Level Security в РСУБД**

Антон Злыгостев

**Введение**

Разграничение прав доступа является необходимой функциональностью любой корпоративной СУБД. Практические все современные СУБД предоставляют набор базовых средств по управлению правами доступа. Как правило, поддерживаются такие концепции, как пользователи и группы, а также так называемая декларативная безопасность – возможность предоставить этим пользователям и группам права доступа к определенным объектам базы данных. Немаловажным вопросом является гранулярность этой безопасности, т.е. насколько детально можно назначить права.

Основными объектами безопасности в СУБД являются таблицы, представления (view), и хранимые процедуры. В зависимости от типа объекта можно управлять правами на конкретные действия с ним. Например, в случае таблиц можно независимо управлять правами на чтение, добавление, удаление и изменение записей. В тех СУБД, где поддерживаются какие-либо другие объекты (например, пользовательские функции), доступом к ним можно управлять аналогичным образом. В некоторых системах можно управлять доступом на уровне отдельного столбца представления или таблицы.

Это достаточно гибкая и развитая система, позволяющая администратору СУБД настраивать права доступа пользователей в соответствии с их служебными обязанностями.

Однако в некоторых случаях встроенной в СУБД функциональности недостаточно для реализации требований корпоративной политики доступа к данным. Дело в том, что очень немногие СУБД позволяют ограничить доступ пользователей к отдельным строкам таблиц, хотя подобное требование достаточно часто встречается в прикладных задачах. Например, менеджеры по продажам не имеют права «видеть» накладные, выписанные в другом офисе той же компании, а директору по продажам все эти данные доступны. Рядовой бухгалтер не должен изменять документы задним числом, но главному бухгалтеру это позволительно. В англоязычной традиции данная функциональность называется Row-Level Security. Устоявшейся русскоязычной терминологии нет, поэтому мы будем пользоваться английским термином, иногда сокращая его до RLS.

Там, где встроенных средств поддержки такой функциональности нет, разработчику придется придумать свой способ гарантировать выполнение требований заказчика. Как и в любой другой задаче, существует большой выбор таких способов. В этой статье предлагается несколько из них.

**Реализация RLS на клиенте**

Одним из наиболее популярных решений задачи RLS является встраивание правил безопасности в клиентское приложение. Поскольку в наши дни пользователи практически никогда не общаются с СУБД напрямую, это выглядит достаточно привлекательно. Все действия пользователя проходят через интерфейс приложения, и разработчик имеет возможность заложить сколь угодно сложные правила проверки допустимости действий. Основными преимуществами данного подхода являются:

большая гибкость – привычный процедурный язык программирования позволяет делать все, что угодно;

возможность определять, выполнимо ли некоторое действие, до попытки его выполнения (а все руководства по usability настоятельно рекомендуют заранее информировать пользователя об ограничениях на действия – например, выделяя серым соответствующий пункт меню);

Однако вместе с тем этот подход обладает несколькими существенными недостатками:

Небезопасность. Предполагается, что пользователи могут подключиться к СУБД только посредством конкретного клиентского приложения. Если злонамеренный пользователь достаточно квалифицирован, то он всегда сможет получить доступ к базе, минуя клиентское приложение, и игнорировать реализованные в нем правила безопасности. Еще более вероятным сценарием в корпоративных приложениях является одновременная работа нескольких версий клиентского приложения с отличающимися наборами правил безопасности. Таким образом, безопасность начинает зависеть уже не от разработчика, а от расторопности системных администраторов, осуществляющих эксплуатацию.

Производительность. Для запросов на изменение данных данный подход несколько повышает быстродействие, т.к. в случае запрета на действие вообще нет необходимости связываться с сервером. Однако запросы на чтение будут приводить к передаче по сети избыточных данных - тех строк, которые будут отброшены клиентским приложением в соответствии с правилами безопасности.

Целостность данных. Разграничение прав доступа на уровне СУБД выполняется в декларативном стиле, т.е. СУБД гарантированно проверяет одни и те же ограничения при любых операциях с данными. Но в клиентском приложении будет использован императивный стиль. Это означает, что потенциально больше возможностей сделать ошибку, или забыть применить единый набор правил для всех возможных способов обращения к данным. В вышеописанном примере с менеджерами продаж легко забыть внести соответствующие ограничения в какие-либо отчеты, основанные на данных из таблицы накладных, и дать пользователю возможность обойти правила безопасности.

Единственным способом борьбы с этими недостатками является реализация проверки всех правил безопасности на сервере.

Современные технологии разработки приложений предлагают два варианта реализации RLS на стороне сервера - средствами сервера баз данных и средствами сервера приложений.

Если для вашего приложения выбрана трехуровневая модель, то реализация RLS на уровне сервера приложений является вполне приемлемым, а возможно и наилучшим, решением.

Однако, если приложение разрабатывается в классической клиент-серверной модели, или предполагается наличие нескольких серверов приложений, работающих с одним сервером данных, то желательно реализовать RLS на уровне базы данных. Возможные подходы к решению этой задачи мы и рассмотрим в следующем разделе.

**Реализация RLS средствами сервера БД**

Итак, перед нами стоит задача - давать или не давать определенному пользователю право на выполнение тех или иных действий с различными строками таблицы. Теоретически, в самом общем случае, это означает, что перед выполнением любого действия проверяется значение некоторого предиката (булевой функции). Стандартные средства безопасности СУБД умеют проверять предикаты, параметрами которых являются идентификатор текущего пользователя, идентификатор(ы) таблиц, и, возможно, отдельных столбцов, к которым осуществляется доступ. Это позволяет вычислить значение один раз перед началом выполнения любого SQL запроса, и до окончания его обработки к вопросу безопасности не возвращаться. Таким образом, сводится к минимуму влияние проверки безопасности на производительность системы.

Нам придется расширить функциональность таким образом, чтобы предикат безопасности вычислялся независимо для каждой строки таблицы.

В терминах SQL это означает, например, что к каждому select-запросу нужно неявно добавить соответствующее условие:

|  |
| --- |
| select \* from Clients where CompanyName like 'Micro%' AND <Security\_Check\_Ok> |

В данном случае под <Security\_Check\_Ok> подразумевается некоторое булево выражение. Далее мы будем исследовать различные варианты построения таких выражений, но перед этим стоит убедиться в том, что мы можем гарантировать проверку этого условия.

Нам необходимо изолировать пользователя от прямого доступа к данным и гарантировать применение установленных правил безопасности. Если используемая СУБД не предоставляет встроенных механизмов обеспечения безопасности, то получить полноценное решение задачи не удастся.

Обычной техникой для изоляции пользователя от хранимых в СУБД данных является построение соответствующего представления (view), и запрет прямого доступа к нижележащей таблице. В MS SQL Server это можно сделать примерно таким образом:

|  |
| --- |
| create view SecureClients as select \* from Clients where <Security\_Check\_Ok>  deny public read on Clients  grant public read on SecureClients |

Тогда при выполнении запроса

|  |
| --- |
| select \* from SecureClients where CompanyName like 'Micro%' |

пользователи автоматически увидят только те строки, для которых <Security\_Check\_Ok> возвращает TRUE.

Теперь рассмотрим различные варианты предикатов, которые могут использоваться для проверки доступа.

**Пользователи и группы**

Теоретически, основой вычисления предиката безопасности является идентификатор текущего пользователя (он, как правило, доступен в любой СУБД, поддерживающей аутентификацию). Однако его прямое использование не рекомендуется, т.к. как корпоративная политика, в соответствии с которой должна строиться реализация системы, редко манипулирует персоналиями. В таком случае затруднительно сформулировать относительно стабильные правила, которые не придется пересматривать при каждом изменении списка сотрудников.

Обычно все правила построены на основе должностей. Их аналогом в программировании являются группы или роли. Поэтому в предикатах безопасности нам часто придется использовать выражения типа IsUserInRole(rolename). Если в используемую СУБД встроена подобная функциональность - прекрасно, лучше всего использовать именно ее. В таком случае субъекты безопасности будут образовывать единое пространство как для встроенной безопасности СУБД, так и для наших расширений.

Если же СУБД не предоставляет средств поддержки групп или ролей, то их тоже придется реализовывать вручную. Одним из простейших способов является создание специальной таблицы, содержащей список групп или ролей, и связь ее с таблицей пользователей. В зависимости от потребностей, можно выбрать различные схемы. Если пользователь может входить только в одну группу, то достаточно добавить ссылку на группу в таблицу пользователей. А если ему может быть назначено одновременно несколько ролей, то для связи надо будет создать отдельную таблицу.

В таких случаях предикат IsUserInRole(rolename) принимает вид:

|  |
| --- |
| exists(select \* from users where ID = CurrentUserID() and user\_group = rolename) |

или

|  |
| --- |
| exists(select \*  from UserRoles  where RoleName = rolename and UserID = CurrentUserID()) |

В дальнейшем мы будем подразумевать под выражением IsUserInRole(rolename) либо подходящую встроенную функцию СУБД, либо предикат, построенный вручную в соответствии с выбранной моделью.

**Ограничения на основе существующих атрибутов**

Если правила корпоративной политики выражаются в терминах предметной области, то можно сформировать соответствующий предикат безопасности в терминах данных, хранимых в СУБД.

В самом простом случае достаточно данных из той же таблицы. Предположим, доступ к еженедельным документам финансовой отчетности компании определяется следующими правилами:

младшие финансовые аналитики имеют право чтения отчетов старше шести месяцев;

старшие финансовые аналитики имеют право чтения отчетов старше одного месяца;

всем остальным сотрудникам доступ к документам запрещен.

В таком случае можно построить примерно такой предикат:

|  |
| --- |
| (IsUserInRole('senior\_analyst') and ReportDate < DateAdd(month, GetDate(), 1))  OR  (IsUserInRole('junior\_analyst') and ReportDate < DateAdd(month, GetDate(), 6)) |

В принципе, тот же самый результат можно получить и другими способами. Например, вот такое выражение SQL отражает те же самые правила:

|  |
| --- |
| case  when ReportDate > DateAdd(month, GetDate(), 1) then false  when ReportDate > DateAdd(month, GetDate(), 6) then  IsUserInRole('senior\_analyst')  else IsUserInRole('junior\_analyst')  end |

Однако здесь проследить логику уже труднее, и ситуация станет еще хуже, когда в рассмотрение войдут другие поля таблицы. Поэтому мы ограничим возможные предикаты вот такой структурой:

|  |
| --- |
| (IsUserInRole(<role1>) AND <role1\_restrictions>)  OR  (IsUserInRole(<role2>) AND <role2\_restrictions>)  OR ...  (IsUserInRole(<roleN>) AND <roleN\_restrictions>) |

Таким образом, предикат определяет для каждой группы пользователей требования, которым должна удовлетворять строка таблицы, чтобы доступ к ней был разрешен.

Лишение доступа

Форма предиката безопасности, рассмотренная выше, подразумевает «пессимистичный» режим, т.е. доступ к данным имеют только те категории пользователей, которые явно перечислены в предикате.

|  |
| --- |
| ПРИМЕЧАНИЕ  Строго говоря, если мы не опишем ни одного правила, то предикат будет пустым и доступ получат все пользователи. Однако выдача таким способом доступа хотя бы одной роли автоматически лишит доступа всех остальных пользователей. Решить эту проблему можно раз и навсегда - введя дополнительный член ...OR FALSE в предикат безопасности. Однако подобный вырожденный случай нетрудно обработать специальным образом, и далее в тексте везде предполагается наличие в списке доступа хотя бы одного явного разрешения. |

Иногда удобнее описывать правила безопасности в терминах «оптимистичного» режима, т.е. лишить доступа только некоторое множество пользователей. В этом случае предикат приобретет такой вид:

|  |
| --- |
| NOT  (  (IsUserInRole(<restricted\_role1>) [AND <restricted\_role1\_restrictions>])  OR ...  (IsUserInRole(<restricted\_roleN>) [AND <restricted\_roleN\_restrictions>])  ) |

В таком случае доступ предоставляется всем, кроме указанных ролей, да и они лишены доступа только к ограниченной части данных.

Соединив эти два подхода, мы получим возможность как «карать», так и «миловать» пользователей:

|  |
| --- |
| (-- секция разрешений  (IsUserInRole(<role1>) AND <role1\_restrictions>)  OR ...  (IsUserInRole(<roleN>) AND <roleN\_restrictions>)  )  AND NOT  (-- секция запретов  (IsUserInRole(<restricted\_role1>) [AND <restricted\_role1\_restrictions>])  OR ...  (IsUserInRole(<restricted\_roleN>) [AND <restricted\_roleN\_restrictions>])  ) |

Такой принцип защиты очень похож на используемый в Windows NT.

Рассмотрим варианты применения данной техники на примере базы данных Northwind, которая входит в поставку MS SQL Server.

Локальные атрибуты

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

|  |
| --- |
| CREATE 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 безопасности по отношению к данным заказов определены следующим образом:

Менеджеры по продажам (роль ‘Sales Representative’) имеют право просматривать только «свои» заказы и не могут видеть заказы, введенные другими менеджерами по продажам.

Директор по продажам (роль ‘Vice President, Sales’) имеет право просматривать любые заказы.

Все остальные сотрудники доступа к заказам не имеют.

Этим правилам соответствует вот такое представление:

|  |
| --- |
| CREATE VIEW [Secure Orders] AS  SELECT \* FROM Orders where  (IsUserInRole('Sales Representative') AND EmployeeID = CurrentEmployeeID())  OR  (IsUserInRole('Vice President, Sales') AND TRUE) |

Здесь подразумевается, что функция CurrentEmployeeID() неким «магическим» образом возвращает идентификатор сотрудника, соответствующий пользователю, от имени которого произведено подключение. Реализация этой функции, как и функции IsUserInRole(), зависит от используемой СУБД. Обратите внимание на вторую часть предиката: для директора по продажам никаких дополнительных ограничений не предусмотрено, но для общности мы использовали выражение TRUE для представления этого факта. При подготовке предиката вручную фрагмент AND TRUE можно опустить, хотя оптимизаторы, используемые в современных СУБД, достаточно интеллектуальны, чтобы выбросить избыточные выражения из плана запроса.

|  |
| --- |
| СОВЕТ  Иногда предикат безопасности может оказаться настолько сложным, что оптимизатор СУБД не сможет самостоятельно построить оптимальный план выполнения запроса. В таких случаях может потребоваться ручное преобразование выражения предиката в более адекватную форму. Пока что мы будем считать, что оптимизатор идеален, и все выражения будут представлены в наиболее удобном для чтения виде. |

Если в будущем руководство компании решит, что доступ к заказам, отгруженным более шести календарных месяцев назад, можно предоставить всем сотрудникам компании, то предикат безопасности примет такой вид:

|  |
| --- |
| (IsUserInRole('Sales Representative') AND EmployeeID = CurrentEmployeeID())  OR  (IsUserInRole('Vice President, Sales') AND TRUE)  OR  (IsUserInRole('Everyone') AND ShippedDate < DateAdd(month, -6, GetDate()) |

Стоит обратить внимание на две особенности предиката:

Во-первых, (опять же из соображений общности) в дополнительном условии проверяется принадлежность текущего пользователя к группе Everyone. Эта специальная группа по определению включает всех сотрудников, и выражение IsUserInRole('Everyone') является тождественно истинным. Это позволяет исключить его из предиката в целях оптимизации.

Во-вторых, условие сравнения даты отгрузки заказа с текущей датой сформулировано так, чтобы к полю не применялось никакой функции. Альтернативные представления того же выражения:

|  |
| --- |
| DateAdd(month, 6, ShippedDate) < GetDate() |

и

|  |
| --- |
| DateDiff(month, ShippedDate, GetDate()) >= 6 |

хотя и являются математически эквивалентными, скорее всего, помешают оптимизатору СУБД использовать индекс по полю ShippedDate (если он есть).

Атрибуты связанных таблиц

В корпоративную политику безопасности могут входить и более сложные правила, которые связывают различные сущности предметной области между собой. Например, предположим, что компания Northwind выросла, и в ней есть несколько филиалов. Структура таблицы сотрудников претерпевает соответствующие изменения:

|  |
| --- |
| ALTER TABLE [Employee]  ADD [DivisionID] int CONSTRAINT [DivisionID\_FK]  REFERENCES [Division]([DivisionID]) |

Новый вариант правила №1 из предыдущего раздела формулируется так:

Менеджеры по продажам (роль ‘Sales Representative’) имеют право просматривать только заказы, введенные менеджерами из того же филиала.

Соответствующая часть предиката безопасности теперь примет такой вид:

|  |
| --- |
| (IsUserInRole('Sales Representative')  AND  (select DivisionID from Employees where EmployeeID = CurrentEmployeeID())  = (select DivisionID from Employees where EmployeeID = EmployeeID) |

Еще один пример правил безопасности, требующий обращения к другим таблицам, связан с защитой подчиненных таблиц. Вместе с записями в таблице заказов необходимо защитить и записи в таблице деталей заказов. Применим правила из предыдущего примера (где еще не было филиалов) к таблице Order Details:

|  |
| --- |
| (IsUserInRole('Sales Representative')  AND  select(EmployeeID from Orders o where o.OrderID = OrderID)  = CurrentEmployeeID())  OR  (IsUserInRole('Vice President, Sales') AND TRUE)  OR  (IsUserInRole('Everyone')  AND  select(ShippedDate from Orders o where o.OrderID = OrderID)  < DateAdd(month, -6, GetDate()) |

К сожалению, такой вид предиката не слишком нагляден. Он не отражает взаимосвязи между правилами безопасности для деталей заказов и для самих заказов. Поэтому лучше применить немного другой способ построения представления, чем был рассмотрен ранее:

|  |
| --- |
| 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 очевидна. Кроме того, изменение правил безопасности для заказов (которое повлияет на определение представления Secure Orders) автоматически отразится и на их деталях.

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.

**Некоторые итоги**

Итак, рассмотренная техника позволяет 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, а это достаточно редкое явление. Не требуется динамического управления доступом на уровне отдельных объектов – например, заказы, отгруженные сегодня, автоматически станут доступными всем сотрудникам через полгода.

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 с быстроменяющимися правилами безопасности, для которых недопустимы задержки в реализации, неизбежные при модификации схемы базы данных. Кроме того, иногда быстродействия СУБД будет недостаточно для вычисления предикатов безопасности на основе существующих атрибутов.

Естественным решением данной задачи является внесение в базу дополнительной информации, не связанной напрямую с моделируемой предметной областью. Модификация этих данных и позволит управлять доступом к конкретным записям без изменения схемы БД. Способы реализации такого рода решений рассматриваются в следующем разделе.

**Ограничения на основе дополнительных атрибутов**

Все соображения, рассмотренные ранее для существующих атрибутов, остаются справедливыми и для дополнительных атрибутов. Мы по-прежнему будем рассматривать предикаты безопасности общего вида. Однако теперь задача ставится несколько шире: если раньше основной проблемой было преобразовать правила корпоративной безопасности в соответствующие выражения с использованием существующей модели, то теперь нам предстоит дополнить модель данных. Теперь нужно отдельно хранить информацию о том, каким ролям предоставлен доступ к каким записям.

**Вспомогательная таблица**

С точки зрения реляционной модели, записи защищаемой таблицы и роли пользователей находятся в отношении многие-ко-многим. Традиционным способом реализации таких отношений является создание вспомогательной таблицы:

|  |
| --- |
| CREATE TABLE [Orders Security] (  OrderID int CONSTRAINT Order\_FK REFERENCES Orders(OrderID),  RoleID int CONSTRAINT Role\_FK REFERENCES UserRoles(RoleID),  CONSTRAINT [Orders\_Security\_PK] PRIMARY KEY (OrderID, RoleID)  ) |

Теперь предикат безопасности будет выглядеть так:

|  |
| --- |
| exists (  select \*  from [Orders Security] os  join UserRoles ur on ur.RoleID = os.RoleID  where ur.UserID = GetCurrentUserID() and os.OrderID = OrderID  ) |

Мы проверяем, что хотя бы одной роли из тех, в которые входит пользователь, предоставлен доступ к соответствующей записи в таблице заказов.

Теперь можно динамически выдавать или отбирать разрешения на записи таблицы заказов каждой из ролей. Очевидно, что сразу после введения предиката в действие никто ничего не увидит – таблица Orders Security пуста. Давайте выдадим вице-президенту разрешение на доступ ко всем заказам:

|  |
| --- |
| insert into [Orders Security] (OrderID, RoleID)  select OrderID, @VicePresidentRoleID from Orders |

Мы предполагаем, что переменная @VicePresidentRoleID уже проинициализирована соответствующим значением. Обратите внимание на то, что выполнение такой команды требует привилегий администратора БД – нужен доступ на чтение из таблицы Orders и запись в таблицу Orders Security. Отметим также то, что, в отличие от ограничений на основе существующих атрибутов, которые автоматически применяются к новым записям, динамические ограничения требуют модификации вспомогательной таблицы при каждой вставке в основную.

**Хранение данных в той же таблице**

Рассмотренный выше способ реализации динамических ограничений может оказаться не самым эффективным. Если в случае естественных ограничений, выражаемых при помощи локального предиката, накладные расходы были связаны только с вычислением дополнительных выражений, то теперь в любом случае требуется обращение к вспомогательной таблице. В большинстве случаев список ролей, которым доступна конкретная запись, достаточно короток. Правило №3 нашей воображаемой корпоративной политики безопасности позволяет связать заказы старше полугода с единственной ролью Everyone, так как все остальные роли уже присутствуют в ней. Это означает, что рано или поздно большинство записей таблицы заказов будут связаны с этой ролью.

В связи с этим возникает искушение сохранить список ролей прямо в защищаемой записи. Некоторые СУБД предоставляют возможность хранить списки значений в поле записи, однако это является экзотикой. Поэтому мы выберем другой способ нарушить требования первой нормальной формы – будем хранить список ролей в виде строки. Язык SQL предоставляет нам возможность проверить строку на наличие заданной подстроки при помощи оператора like.

В таблицу Orders придется внести следующие изменения:

|  |
| --- |
| alter table orders add ACL varchar(1024) default ',' |

В поле ACL (Access Control List, список контроля доступа) мы будем хранить список идентификаторов ролей, разделенный запятыми. Поэтому предикат безопасности примет такой вид:

|  |
| --- |
| ACL like '%,' + cast(GetCurrentUserRoleID() as varchar(12)) + ',%' |

Чтобы упростить задачу серверу, нам придется добавить «лишние» запятые в начале и в конце строки.

Стоит отметить, что подобная техника оправдывает себя только в некоторых случаях, так что не стоит применять ее без экспериментальной проверки.

**Битовые маски**

Еще один вариант представления ACL в защищаемой таблице состоит в использовании битовых операций с целыми числами. Предположим, что полное количество ролей в системе ограничено каким-то разумным числом, например 32. Тогда можно хранить ACL в виде целого числа (или нескольких целых чисел, если ролей больше 32х), сопоставив каждой роли фиксированную позицию в этом числе. Предикат безопасности примет вот такой вид:

|  |
| --- |
| (ACL & GetCurrentUserRoleMask()) > 0 |

Как и в предыдущем случае, производительность подобного способа может быть далека от оптимальной, и перед его внедрением необходимо провести экспериментальную проверку.

**Ограничения на модификацию**

До сих пор мы рассматривали исключительно запросы на выборку данных. Модель безопасности, не защищающая данные от несанкционированной модификации, не имеет смысла. Давайте рассмотрим способы распространить наши достижения на остальные типы запросов SQL.

Точно так же, как и для запросов Select, мы должны вычислить предикат безопасности для каждой из строк, которые пользователь попытается модифицировать. Нам нужно сообщить пользователю об ошибке, а также позаботиться о неизбежной отмене результатов некорректного действия. Этого можно добиться, применяя триггеры.

В слеующих примерах выражение <Security\_Check\_Ok> означает предикат, аналогичный рассмотренным выше. Как и для select, предикат может быть представлять либо естественное, либо динамическое ограничение.

Для естественных ограничений принципиально важны два предиката: условие, которое проверяет состояние данных до изменения, и отдельное условие, которое проверяет состояние данных после изменения. Первое условие необходимо проверять в триггерах на удаление и изменение записей, а второе – в триггерах на изменение и вставку записей. Может появиться искушение использовать более сложные правила для изменения данных, однако делать этого обычно не стоит. Слишком легко получить нецелостную схему безопасности, которая позволяет пользователю получить различные результаты в зависимости от порядка действий. Например, запрет на увеличение суммы заказа более чем на 500 рублей (в один прием) легко обходится путем последовательного неоднократного увеличения суммы. А запрет на удаление заказов VIP-клиента, не сопровожденный запретом на изменение таких заказов, может привести к искушению просто «переписать» заказ на другого клиента.

Для динамических ограничений придется хранить чуть больше информации, чтобы учесть возможность раздельного предоставления прав на различные типы модификаций.

Примерно так будут выглядеть наши триггеры на T-SQL:

Delete

|  |
| --- |
| create trigger CheckSecurity on <table\_name> for delete  as  if exists (select \* from deleted where not <Security\_Check\_Ok>)  begin  RAISERROR ('Access denied', 16, 1)  ROLLBACK TRANSACTION  end |

Insert

|  |
| --- |
| create trigger CheckSecurity on <table\_name> for insert  as  if exists (select \* from inserted where not <Security\_Check\_Ok>)  begin  RAISERROR ('Access denied', 16, 1)  ROLLBACK TRANSACTION  end |

Update

Этот пример чуть-чуть сложнее, т.к. нам надо проверить не только старые, но и новые значения в таблице:

|  |
| --- |
| create trigger CheckSecurity on <table\_name> for insert  as  if exists (select \* from inserted where not <Insert\_Check\_Ok>)  or  exists (select \* from deleted where not <Delete\_Check\_Ok>)  begin  RAISERROR ('Access denied', 16, 1)  ROLLBACK TRANSACTION  end |

**Заключение**

Эти рассуждения носили в значительной мере теоретический характер. С одной стороны, это вселяет уверенность в том, что никакие важные особенности поставленной задачи не остались без внимания и излишние подробности не отвлекали от решения.

С другой стороны, тем читателям, которым захочется применить RLS на практике, придется довольно много потрудиться. Для них предназначен следующий раздел.

**Соображения по реализации**

Первый и основной совет: экспериментируйте, прежде чем запускать модификацию в эксплуатацию. Убедитесь, что предикаты составлены верно.

Второе: производительность. Самые невинные запросы теперь будут скрывать в себе дополнительные вычисления. Поэтому не забывайте про индексы. Несомненно, помимо создания и модификации таблиц потребуется создать необходимые индексы. Возможно, некоторые из уже существующих индексов придется изменить, чтобы планы запросов не слишком пострадали. Проверяйте не только фактическое время выполнения запросов, но и планы, которые строит СУБД.

Ну и последнее: не забудьте аккуратно назначить привилегии на объекты СУБД таким образом, чтобы пользователи не могли получить прямой доступ к данным, минуя проверки. Кроме того, позаботьтесь о защите самих определений триггеров и представлений от модификации.

**Альтернативы**

Ведущие производители СУБД не оставили рассмотренный в статье вопрос без своего внимания. Oracle 8i и выше поддерживает так называемый fine grained access control. Фактически, это применение точно такой же математики, но несколько другими техническими средствами. Вместо механизма View используются специальные возможности дополнительных пакетов Oracle по установлению политики безопасности.

MS SQL Server 2005 (codename “Yukon”) также будет поддерживать возможности RLS. Подробная документация на этот механизм еще не опубликована. Точно известно лишь то, что в T-SQL будут введены специальные конструкции. Судя по имеющимся документам, предикаты безопасности (RULES) будут создаваться как отдельные сущности, а специальные версии команд grant и revoke будут назначать эти предикаты пользователям и группам.