Язык M Power Query. Табличное мышление

Зачем вам понимание того, как Power Query думает о таблицах? В конце концов, вы пишете выражение, движок делают свою работу, в итоге получается таблица. Все довольны, и какое вам дело, что «за кулисами»... Верно? Да… до тех пор, пока вы не столкнетесь с проблемами производительности, не измените значение во время обработки или не возникнет ошибка брандмауэра… И что тогда? Понимание того, как M обрабатывает таблицы, является важным преимуществом при разработке эффективных запросов, позволяет избежать изменчивость результатов и обеспечить конфиденциальность данных. К табличному мышлению языка M относятся следующие темы: потоковая передача, свертывание запросов, буферизация, ключевые столбцы, встроенное кэширование, брандмауэр…[1]

Предыдущая заметка Следующая заметка

Как передаются данные

Начнем с потоковой передачи и свертывания запросов. Представьте, что вы – движок запросов. Как бы вы выполнили следующее?

```
let
  Source = SomeDataSourceReturningATable,
  Filtered = Table.SelectRows(Source, each [Office] = "Chicago"),
  Result = Table.FirstN(Filtered, 3)
in
  Result
```

Можно извлечь все строки из *SomeDataSourceReturningATable*, и сохранить их в переменной *Source*. Затем отфильтровать строки, удовлетворяющие условию [Office] = "Chicago", и сохранить их в переменной *Filtered*. Наконец, взять первые три строки этого датасета и сохранить в переменной *Result*, которую вернуть в виде таблицы из трех строк, как результат запроса.

Логично? Да. Эффективно? Нет. Почему? Во-первых, это использование ресурсов: в то время как нам нужны лишь три строки, мы для начала извлечем их все. А если это миллионы строк!? К счастью, Power Query вместо этого использует потоковую передачу и свертывание запросов. Вкратце мы обсудили эти вопросы ранее. В этой заметке мы погрузимся в детали и примеры.

Поток

Запрос выполняется с конца. *Result* обращается к функции *Table.FirstN*, которая запрашивает у шага *Filtered* первую строку данных из источника. *Table.SelectRows* проверяет, соответствует ли она условию [Office] = "Chicago". Если да, строка возвращается в *Result.* Если нет, строка не идет в работу, и запрашивается вторая строка из источника. Так до тех пор, пока не будет найдена строка, соответствующая условию фильтра. Как только *Table.FirstN* получит первую строку, он фиксирует ее, чтобы позже вернуть в качестве результата и запрашивает у *Filtered* вторую строку. Идет поиск второй строки, удовлетворяющей фильтру. Затем процесс повторится третий раз.

Каждый шаг создает строки по одной за раз, запрашивая столько строк, сколько требуется на предыдущем шаге для создания запрошенной строки. Благодаря этому Power Query способен обрабатывает наборы данных, которые целиком могут не помещаться в памяти движка, и не тратит ресурсы на хранение тех строк, которые в итоге не нужны.

Внутреннее хранение строк в памяти

За исключением поставщика данных (который по соображениям производительности может извлекать строки порциями из внешнего источника данных), ни одна из операций в коде не хранит строки в памяти. Когда функция обрабатывает строку, она либо передает ее дальше, либо отбрасывает. Однако это относится не ко всем операциям. Допустим, мы добавили сортировку:

```
let
  Source = SomeDataSourceReturningATable,
  Filtered = Table.SelectRows(Source, each [Office] = "Chicago"),
```

---

[1] Заметка написана на основе статей Power Query M Primer (Part 12): Tables—Table Think I и Power Query M Primer (Part 13): Tables—Table Think II. Если вы впервые сталкиваетесь с Power Query, рекомендую начать с Марк Мур. Power Query.

```
  Sorted = Table.Sort(Filtered,{{"TotalSales", Order.Descending}}),
  Result = Table.FirstN(Sorted , 3)
in
  Result
```

Сортировка (как правило) требует извлечения всех строк с предыдущего шага. Когда выполняется этот код функция *Table.FirstN* с шага *Result* запрашивает у *Table.Sort* с шага *Sorted* первую строку отсортированной таблицы. Чтобы определить, какую строку возвращать, *Sorted* должен запросить все строки с шага *Filtered*. Далее строки будут отсортированы и сохранены в памяти. Затем шаг *Sorted* сможет отдать шагу *Result* первую строку из отсортированного набора. Каждый раз, когда запрашивается сортировка, сначала нужно сохранить в памяти отсортированный датасет. После того, как *Sorted* вернет строку, она ему больше не понадобится, поэтому он может удалить ее из памяти.

Это внутреннее хранение строк в памяти не является постоянным кэшем; скорее, его область действия ограничена одним вызовом функции в рамках выполнения запроса. Общий доступ к этим наборам строк, хранящимся в памяти, отсутствует. И если функция вызывается несколько раз, то каждый раз она будет запрашивать данные.

```
let
  Source = SomeDataSourceReturningATable,
  Filtered = Table.SelectRows(Source, each [Office] = "Chicago"),
  Sorted = Table.Sort(Filtered, {{"TotalSales", Order.Descending}}),
  Top3 = Table.FirstN(Sorted , 3)
in
  { List.Sum(Top3[TotalSales]), List.Average(Top3[TotalSales]) }
```

В этом коде *Table.Sort* сначала будет запрошена фрагментом List.Sum(Top3[TotalSales]), а затем фрагментом List.Average(Top3[TotalSales]).

Когда строки сохраняются в памяти? Во-первых, в операциях объединения (хотя и не во всех видах). При сортировке, группировке, отмене свертывания столбцов, сведении столбца и буферизации. К сожалению, MS предоставляет весьма скупую документацию на эту тему.

Удержание строк *в памяти* может означать разное. Память может быть выгружена на диск. В этом случае работа значительно замедлится по сравнению с удержанием строк в оперативной памяти. Запуск подкачки на диск зависит от среды. В некоторых средах подкачка начинается, когда общий объем памяти, используемый запросом, превышает 256 MB.

Эффективность

Порядок операций может оказать существенное влияние на то, сколько данных должно храниться в памяти. Сравним два варианта выражения. Они дают одинаковый результат, но существенно отличаются по используемым ресурсам.

```
let
  Source = SomeDataSourceReturningATable,
  Sorted = Table.Sort(Source, {{"TotalSales", Order.Descending}}),
  Filtered = Table.SelectRows(Sorted, each [Office] = "Chicago"),
  Result = Table.FirstN(Filtered, 3)
in
  Result
```

```
let
  Source = SomeDataSourceReturningATable,
  Filtered = Table.SelectRows(Source, each [Office] = "Chicago"),
  Sorted = Table.Sort(Filtered, {{"TotalSales", Order.Descending}}),
  Result = Table.FirstN(Sorted, 3)
in
  Result
```

В первом – сортируются все строки из источника, поэтому все они хранятся в памяти. Второе – сортирует после фильтрации. В памяти хранятся только отсортированные строки. Допустим, источник содержит два миллиарда строк, из которых 500 удовлетворяют условию [Office] = "Chicago". В первом выражении *Sorted* хранит два миллиарда строк, а во втором – 500!

> *Если ваш запрос содержит шаги, которые удерживают строки в памяти, старайтесь поместить фильтрацию перед такими шагами.*

Свёртывание запросов

Потоковая обработка может извлекать большого числа строк, которые позже будут отброшены. В примере, который мы используем, потенциально необходимо извлекать миллиарды строк из источника для выдачи всего трех строк. Если бы вы сразу передали источнику, что вам нужно в итоге, он бы вернул только это.

Если источником это база данных SQL, вы бы написали что-то вроде:

```
SELECT TOP 3 *
FROM Customers
WHERE Office = 'Chicago'
ORDER BY TotalSales DESC;
```

Если же источник – OData REST API, ваш запрос мог выглядеть следующим образом:

```
GET /Customers?$filter=Office eq 'Chicago'&$top=3&$orderby=TotalSales
```

В обоих случаях вы получите не более 3 строк. Вместо перегонки миллиардов строк между источником и PQ, вся обработка выполняется во внешней системе (с использованием индексирования и кэширования), и в PQ возвращаются только результаты. Очевидно, что такой подход эффективнее.

К счастью, в языке M есть механизм свёртывания запросов. Движок M без участия пользователя определяет, как объединить несколько шагов и обратиться к источнику со своим собственным запросом. Например, если мы написали в PQ следующее выражение…

```
let
  Source = SomeDataSourceReturningATable,
  Filtered = Table.SelectRows(Source, each [Office] = "Chicago"),
  Sorted = Table.Sort(Filtered, {{"TotalSales", Order.Descending}}),
  Result = Table.FirstN(Sorted, 3)
in
  Result
```

… интерпретатор M заменит его на что-то типа:

```
let
  Result = Value.NativeQuery(
    SomeDataSourceReturningATable,
    "SELECT TOP 3 * FROM Customers
       WHERE Office = 'Chicago'
       ORDER BY TotalSales DESC;"
  )
in
  Result
```

Технически внутренний механизм может работать немного по-другому, но что касается результата, то он будет однозначным: из базы вернутся только три строки.

Собственный запрос может быть статическим, основанным на коде, или динамическим, учитывающим данные, появляющиеся в процессе выполнения запроса. Следующее выражение фильтрует данные (MainData), полученные из источника A, используя список значений (filterData), полученных из источника B.

```
let
  MainData = GetFromSourceA(),
```

```
  FilterData = GetFromSourceB(),
  Result = Table.SelectRows(MainData, each List.Contains(FilterData[ID], [ID]))
in
  Result
```

На первый взгляд, не похоже, что запрос можно свернуть, поскольку он объединяет данные из двух источников. Кажется, что необходимо извлечь все данные из обоих источников, а затем применить фильтр внутри *Table.SelectRows*.

Однако Power Query может извлечь данные из одного источника и записать эти данные в собственный запрос, который направит другому источнику. Скажем, *FilterData* (из источника B) содержит всего несколько строк. Power Query может сначала извлечь эти несколько строк, а затем сформировать собственный запрос, который отправит в источник A. Представьте, что *FilterData* содержит три строки со значениями столбцов идентификаторов этих строк 1, 2 и 3. Power Query сначала извлечет эти значения из источника B, а затем отправляет в источник A запрос:

```
SELECT *
FROM SomeTableInSourceA
WHERE ID IN (1, 2, 3); эти значения были извлечены
   из filterData (источник B), а затем записаны в собственный запрос
```

Такой запрос сообщает источнику A, какие именно строки необходимы, на основе фильтрации данных, полученных из источника B. В этом случае отбор строк произойдет в источнике A. Это эффективнее, чем извлечь все строки из источника A и применить фильтр локально, внутри PQ.

Такое извлечение данных из одного источника с последующей передачей их в другой может повысить производительность, но… вызвать проблемы с безопасностью. В среде Power Query есть механизм для управления последней, который мы вскоре рассмотрим.

Свёртывание + поток

Не все операции могут быть свёрнуты. Как только в цепочке выражений встречается первая несворачиваемая операция, все дальнейшие шаги не будут свёрнуты. Свёртывание запросов не исключает потоковой передачи. Несколько свёрнутых шагов передают поток последующим шагам, которые не были свёрнуты запросом.

Стремитесь помещать шаги, поддающиеся свёртыванию, до шагов, которые свёртывание блокируют. Это позволит передать в источник данных максимальный объем обработки.

Какие шаги свёртываются в запрос, а какие нет, может меняться по мере улучшения механизма запросов, изменения количества внешних источников, пересмотра библиотечных функций и изменения параметров безопасности.

Таблицы не являются неизменяемыми

Переменная, которая содержит таблицу (или список), на самом деле просто содержит идентификатор выражения, которое создает таблицу (или список). При обращении к этому идентификатору выполняется логика, которая выдает запрошенные данные. В то время как идентификатор остается неизменным на протяжении всего запроса, данные, возвращаемые при его вызове, таковыми не являются. Это связано с тем, что данные создаются по требованию каждый раз, когда задействован идентификатор. Изменяемость возвращаемых данных может привести к тому, что значения будут казаться изменяющимися во время выполнения запроса.

Следующее выражение возвращает две таблицы. В одной хранятся все клиенты, связанные с чикагским офисом; в другой – три клиента с наибольшим общим объемом продаж. Обе таблицы извлекаются из источника *Source*.

```
let
  Source = SomeDataSourceReturningATable,
  ChicagoOffice = Table.SelectRows(Source, each [Office] = "Chicago"),
  Top3Sales = Table.FirstN(Table.Sort(Source, {{ "TotalSales", Order.Descending }}), 3),
  Result = { ChicagoOffice, Top3Sales }
in
  Result
```

Представим, что после выполнения запроса вы просматриваете таблицу *ChicagoOffice*, и находите клиента ABC:

CustomerID = 123, Customer = ‘ABC’, Office = ‘Chicago’, TotalSales = **50 255**

Просматривая строки в таблице *Top3Sales*, вы также находите клиента ABC (он приписан к чикагскому офису и является одним из ваших лучших клиентов):

CustomerID = 123, Customer = ‘ABC’, Office = ‘Chicago’, TotalSales = **62 199**

Что за черт?! У одного и того же клиента значение *TotalSales* в разных таблицах разное. Как же так!?

Шаги *ChicagoOffice* и *Top3Sales* обращались к *Source* независимо. Между этими обращениями объем продаж для клиента ABC в источнике *SomeDataSourceReturningATable* мог измениться.

Напомним: выражение для переменной является неизменяемым, но данные, возвращаемые выражением при вызове переменной, могут изменяться. Таким образом, когда данные извлекаются из одного и того же источника несколько раз во время выполнения запроса, существует вероятность того, что данные могут изменяться между вызовами. Если это неприемлемо, можно переработать выражение, чтобы исключить многократные вызовы (что может и не получиться) или вручную кэшировать (буферизировать) входные данные.

Не думайте, что Power Query ошибается, допуская такую изменчивость. Эта изменчивость является побочным эффектом того, что M не всегда сохраняет все строки в памяти (и хорошо, что он этого не делает!). Вместо этого M дает вам контроль: если нужна неизменяемость, кэшируйте набор данных. Когда ресурсы и производительность важнее, оставляйте, как есть.

Буферизация

В M есть функции для буферизации списков и таблиц: Table.Buffer и List.Buffer. Table.Buffer считывает все значения из источника и сохраняет их в памяти. Эти кэшированные данные затем используются при любом к ним обращении. В том числе несколько шагов запроса могут ссылаться на один и тот же буфер. Адаптируем пример с чикагским офисом и ТОП продаж:

```
let
  Source = SomeDataSourceReturningATable,
  BufferedSource = Table.Buffer(Source),
  ChicagoOffice = Table.SelectRows(BufferedSource , each [Office] = "Chicago"),
  Top3Sales = Table.FirstN(Table.Sort(BufferedSource , {{ "TotalSales", Order.Descending }}), 3),
  Result = { ChicagoOffice, Top3Sales }
in
  Result
```

На втором шаге *BufferedSource* извлекает все строки из источника и сохраняет их в памяти. Этот кэш затем используется, как на шаге *ChicagoOffice*, так и на шаге *Top3Sales*. Когда выполнение запроса завершится, буфер обнулится. Если позже запрос будет выполнен снова, то создастся новый кэш.

Конечно, буферизация всех запросов может создавать проблему с ресурсами, если объем данных велик. Важно, что вы контролируете ситуацию: вы сами решаете, когда использовать буферизацию, а когда преимущества, которые она приносит, не стоят связанных с этим затрат.

Когда вы буферизуете, думайте, сколько данных вы кэшируете. Есть ли возможность применить фильтрацию перед буферизацией? Например, если вы анализируете продажи крупным клиентам чикагского офиса (с объемом продаж более $50K), измените BufferedSource на:

```
BufferedSource = Table.Buffer(
   Table.SelectRows(
      Source,
      each [Office] = "Chicago" and [TotalSales] >= 50000
   )
)
```

Такое изменение способно значительно уменьшить количество кэшированных данных.

Мысли о производительности

Следующие идеи не являются непреложными. Это, скорее рекомендуемая отправная точка для решения проблем с производительностью. Не следуйте им слепо, а оценивайте в сочетании с пониманием контекста и того, как Power Query обрабатывает таблицы.

1. Упорядочите шаги, начав с тех, что поддерживают свёртывание запросов. Это позволит максимально перенести обработку во внешний источник данных. Чтобы определить, какие шаги свёртываются, вам потребуется трассировщик или что-то иное, поскольку пользовательский интерфейс PQ не всегда корректен в отображении свёртывания.
2. Шаги, которые нельзя свернуть, попробуйте разместить в следующем порядке: фильтры, операции, не требующие удержания строк в памяти, иные операции. Это позволит минимизировать объем данных, удерживаемых в оперативной памяти.
3. Когда необходима буферизация, кэшируйте в конце цепочки выражений. Зачем расходовать ресурсы до того, как данные действительно нуждаются в буферизации?
4. Если указанные шаги не обеспечивают желаемой производительности создайте собственный запрос к источнику на поддерживаемом им языке, а результаты загрузите в Power Query. Возможно, вам удастся отразить логику запроса PQ, обойдясь без кода в PQ.

Ключи

Табличные функции Power Query работают с любой комбинацией допустимых значений, передаваемых в любом порядке. Однако, если предоставлена информация о ключах таблицы, функции работают быстрее.

Рассмотрим объединение (слияние) таблиц A и B по столбцу. Сначала в таблице A в столбце объединения операция находит строку со значением 1. Далее ищет строку с тем же значением в таблице B. Найдя её, операция возвращает объединенную пару. Затем возобновляет поиск в таблице B, ища другие строки с тем же значением 1. Если же операция знает, что столбец объединения таблицы B содержит уникальные значения, ей не нужно искать дополнительные строки со значением 1 в этой таблице, поскольку уникальность столбца гарантирует, что их больше нет.

Ключ – это столбец или набор столбцов, значения которых идентифицируют строки. Уникальным называется ключ, значения которого идентифицируют одну строку в текущей таблице. Из уникальных ключей таблицы один может быть идентифицирован как *первичный ключ*. Такой ключ является основным используемым идентификатором.

Представьте таблицу с информацией о компаниях. Среди её столбцов есть *CompanyID*. Если каждое значение *CompanyID* идентифицирует ровно одну строку в таблице, *CompanyID* является уникальным ключом и, исходя из его имени, вероятно, является первичным ключом таблицы (первичный уникальный идентификатор).

В языке M Power Query можно аннотирования таблицы ключевой информацией. Ключи можно просматривать, определять и заменять с помощью библиотечных функций: Table.Keys, Table.AddKey, Table.ReplaceKeys. В следующем примере сначала определяется ключ в таблице, а затем просматриваются сведения о ключах таблицы:[2]

```
let
  Source = #table(
    {"CompanyID", "Name", "Location"},
     {
       {1, "ABC Company", "Chicago"},
       {2, "ABC Company", "Charlotte"},
       {3, "Some Other Company", "Cincinnati"}
     }
    ),
  KeysTagged = Table.AddKey(Source, {"CompanyID"}, true)
in
```

[2] Всё не так просто. Функция Table.Keys возвращает список, в котором не видны ключевые столбцы. Нужно использовать детализацию, чтобы добраться до ключей.

```
Table.Keys(KeysTagged) /* возвращает список ключей,
в данном случае список представлен одной записью:
[Columns = {"CompanyID"}, Primary = true]*/
```

Некоторые коннекторы способны генерировать информацию о ключах автоматически. Например, коннектор базы данных использует ее индексы, чтобы определить, какие столбцы являются ключами, и пометить их в возвращаемой таблице. Некоторые функции способны добавлять информацию о ключах. Например, Table.Distinct:

Table.Distinct(table as table, optional equationCriteria as any) as table

Если второй аргумент опущен, то функция *Table.Distinct* считает дублями только строки, идентичные во всех столбцах. Удаляет такие дубли, и помечает первичным ключом в таблице все столбцы. Это не лишено смысла (хотя и странно), так как после применения *Table.Distinct* набор строк становится уникальным.

Хотя язык Power Query обеспечивает поддержку идентификации ключей таблиц, каждая операция решает, использовать ли эту информацию. Похоже, нет официальной документации, описывающей, когда и как используется информация о ключах, поэтому нам приходится прибегать к экспериментам, чтобы выяснить, когда выгодно определять ключи.

Чтобы повысить производительность путем предоставления информации о ключах, можно попробовать следующее:

- Определите столбец (столбцы), используемые функцией для идентификации строк. Если операция работает с несколькими таблицами, сделайте это для каждой исходной таблицы. Например, если операция представляет собой соединение, какой столбец из каждой входной таблицы используется для сопоставления строк между двумя таблицами?
- Затем для каждой исходной таблицы проверьте, являются ли значения в идентифицированных столбцах уникальными в этой таблице. Если да, проверьте, есть ли в таблице первичный ключ, определенный для этих столбцов. Если нет, пометьте эти столбцы в качестве первичного ключа. Если столбцы содержат не только уникальные значения, пометьте столбцы как вторичный ключ.
- После добавления ключей повторите запрос и посмотрите, улучшилась ли производительность.

Эта настройка полезна для операций, которые идентифицируют строки на основе значений столбцов. Ключ не повысит производительность при вызове функции Table.FirstN(Source, 5). Она и так знает, какие пять строк нужно вернуть!

То, что библиотечные функции делают с информацией о ключах таблицы, является внутренней особенностью реализации, которая может измениться при обновлении версии Power Query, поэтому спустя некоторое время может оказаться полезным повторно попробовать эту настройку производительности, даже если вы не изменили свой код.

Среда

Запросы выполняются в контексте среды – Microsoft Power BI или Microsoft Excel. Эта среда может включать функциональность, специально созданную для повышения производительности. Хотя эти факторы среды не являются частью языка Power Query или стандартной библиотеки, их важно понимать для эффективной работы с таблицами.

Среда *может* (но не всегда) сохранять результаты выполнения *собственного* запроса в постоянном кэше, который записывается на диск. Именно м*ожет* кэшировать. Ничто в спецификации языка M не требует, чтобы среда обеспечивала такое кэширование. Когда и как кэшировать зависит от среды. Пусть, в Excel вы кликнули *Обновить*. Во время загрузки два запроса запрашивают один и тот же коннектор для выполнения одного и того же собственного запроса к одному и тому же серверу (run SELECT * FROM Customers on Server123). Среда может кэшировать результаты первого запроса, и повторно передать их в ответ на второй запрос.

Кэширование применяется только к *собственным* запросам (например, запрос к базе данных, вызов веб-службы). Результаты, полученные с помощью выражений, шагов или запросов Power Query, не кэшируются.

С точки зрения безопасности постоянное кэширование может привести к тому, что данные останутся даже после удаления отчета, в который эти данные были загружены. Постоянный кэш не хранится внутри файла отчета, поэтому удаление отчета не приводит к удалению данных из кэша. Чтобы избежать компрометации данных, вам нужно вручную очистить кэш вашей среды после удаления отчетов.

Чтобы оптимизировать вероятность постоянного кэширования, вы можете отключить параллельную загрузку таблиц (если ваша хост-среда позволяет это). Это снижает вероятность одновременного выполнения одного и того же собственного запроса и, таким образом, увеличивает вероятность того, что повторные вызовы одного и того же собственного запроса будут обслуживаться из кэша. Хотя я бы не рекомендовал делать это по умолчанию. Отключение параллельной загрузки – это вариант, который следует рассмотреть, если повторяющиеся собственные запросы приводят к потери производительности.

Брандмауэр

При первом запуске запроса, который извлекает данные из нескольких источников, вас попросят установить уровень конфиденциальности источника. Этот уровень контролирует изолированность между источниками данных. Но разве источники данных не являются изолированными? Когда вы используете M для извлечения данных из нескольких источников, M выполняет объединение, поэтому каждый источник изолирован от всех остальных, верно? Не совсем. Как насчет свёртывания запросов? При свёртывании данные, возвращаемые одним источником, могут быть записаны в собственный запрос, отправленный другому источнику.

В следующем коде данные для условий фильтрации извлекаются из одного источника, а затем помещаются в собственный запрос, отправляемый в другой источник.

```
let
  TableFromSourceA = GetFromSourceA(),
  TableFromSourceB = GetFromSourceB(),
  Result = Table.SelectRows(TableFromSourceA, each List.Contains(TableFromSourceB[ID], [ID]))
in
  Result
```

Если происходит свёртывание запроса, для источника A может быть сгенерирован собственный запрос, аналогичный следующему (при условии, что источник A – это база данных SQL).:

```
SELECT * FROM SomeTable
WHERE ID IN (1, 2, 3); эти три значения были извлечены из источника B
```

Свёртывание запроса привело к тому, что данные извлекались из одного источника, а затем передавались в другой. Иногда такое раскрытие данных является приемлемым, а выигрыш в производительности ощутим. Если же речь идет о конфиденциальных медицинских записях или коммерческой тайне, незаметная утечка данных может стать серьезной проблемой и поэтому не должна допускаться, независимо от влияния на производительность.

Уровни конфиденциальности

Уровни конфиденциальности – это механизм контроля обмена данными, который разрешен между источниками данных во время свёртывания запросов. Уровни конфиденциальности не мешают вам комбинировать источники или писать код, который извлекает данные из одного источника, а затем передает их другому. Они существуют лишь для управления свёртыванием запроса данных из одного источника в собственные запросы, направляемые в другой источник.

Общедоступный уровень конфиденциальности указывает на то, что данные из источника могут свободно передаваться другим источникам во время свёртывания запроса. Источники организационного уровня могут предоставлять доступ к своим данным другим источникам организационного уровня только во время свёртывания. Данные из частных источников не могут быть объединены с каким-либо другим источником, даже с другими частными источниками (см. также спецификацию Power Query).

Уровни конфиденциальности влияют на производительность. Разрешение совместного использования данных между источниками повышает производительность за счет свёртывания

запросов. Блокирование совместного использования может привести к затратам на выборку и локальную буферизацию большего, чем строго необходимого, набора данных перед объединением.

Уровни конфиденциальности также оказывают влияние на код. Когда включены уровни конфиденциальности, интерпретатор Power Query разделяет ваш код на блоки, называемые разделами. Затем он переписывает любые ссылки на код, которые обращаются к данным из других разделов, чтобы передать эти данные через брандмауэр Power Query. Это позволяет брандмауэру контролировать поток данных между разделами. Когда необходимо заблокировать свёртывание запросов из разных источников данных, тот факт, что данные из разных разделов проходят через брандмауэр, позволяет брандмауэру буферизировать эти данные на соответствующей границе раздела. Поскольку буферизация блокирует свёртывание запроса в точке, где оно происходит, это действие предотвращает свёртывание запроса через границу раздела и, таким образом, предотвращает утечку данных между источниками.

Необходимо соблюдать следующее правило, чтобы гарантировать, что логика брандмауэра будет вставлена в соответствующие места:

- либо раздел может содержать источники данных с совместимыми уровнями конфиденциальности; такими, что позволяют объединять источники,
- либо раздел может ссылаться на другие разделы,
- но и то и другое одновременно недопустимо.

Если первая часть этого правила нарушена, то есть, в разделе более одного источника данных, и не все они имеют совместимые уровни конфиденциальности, будет возвращена ошибка, типа:

Formula.Firewall: Запрос ImportantData (шаг Source) обращается к источникам данных, имеющим уровни конфиденциальности, которые нельзя использовать вместе. Пожалуйста, перестройте эту комбинацию данных.

Если нарушена третья часть правила, то есть, если раздел содержит источник данных и ссылается на другой раздел, будет возвращена ошибка:

Formula.Firewall: Запрос ImportantData (шаг Source) ссылается на другие запросы или шаги, поэтому он не может напрямую обращаться к источнику данных. Пожалуйста, перестройте эту комбинацию данных.

В любом случае, не волнуйтесь! Вы всё еще можете объединить два источника. Просто уровень защиты данных требует переработки кода, чтобы он позволял брандмауэру выполнять свою работу. Код, который объединяет источники, не может находиться в том же разделе, что и источники, а несовместимые источники не могут находиться вместе в одном разделе. Ключом к корректировке кода в соответствии с этими требованиями является понимание того, где проводятся границы разделов.

К сожалению, правила, определяющие порядок разделения кода, сложны. Быстрое решение заключается в следующем. Если вы столкнулись с одной из указанных ошибок, поместите код, специфичный для каждого источника данных, в отдельный запрос (по одному запросу на источник данных), а затем ссылайтесь на эти запросы из другого запроса, который объединяет их. Это избавит вас от необходимости разбираться в сложных особенностях разбивки на разделы. Подробнее см. Behind the Scenes of The Data Privacy Firewall.

Отключение уровня защиты данных

Также есть возможность полностью забыть об уровнях конфиденциальности и разделах, отключив защиту данных. Тогда три условия, описанные выше, можно нарушать, и выполнению кода ничто не будет мешать. Является ли это хорошей идеей? Только вы можете решить это. Свёртывание запросов может привести к незаметной утечке данных. Я бы советовал не делать отключение опцией по умолчанию. Вместо этого можно отключать проверку конфиденциальности только тогда, когда вы сталкиваетесь с проблемой.

Дополнительные собственные вызовы запросов

Если вы проследите за собственными запросами, отправленными из Power Query во внешние источники, вы удивитесь. Там, где вы ожидали, что собственный запрос обратится к источнику

один раз, вы обнаружите несколько обращений. Это связано с реализацией хост-среды или используемых функций. Иногда запрос обращается к источнику дважды – первый раз для получения имен заголовков, второй – для потоковой передачи результирующего набора. Возможно, брандмауэр запросил фрагмент строк для анализа, чтобы решить, как разделить код.

Иногда коннектор оптимизирует эти процессы. Если коннектор этого не делает, производительность может падать, поскольку запрос выполняется несколько раз в полном объеме. Появление дополнительных собственных запросов является одной из причин, по которой рекомендуется не использовать Power Query для выполнения собственных запросов, изменяющих данные. Для повышения производительности вы можете попробовать реализовать свою собственную логику оптимизации с помощью библиотечной функции Table.View.

В следующей заметке

… вы узнаете о структуре управления Power Query (нет, это не опечатка – существует только одна структура управления!). Мы также рассмотрим, как справляться с ситуациями, когда вы чувствуете необходимость в цикле или повторении.