Язык M Power Query. Свёртывание запросов

При свёртывании запроса Power Query принимает выражение M и преобразует его частично или полностью в собственный язык запросов или язык запросов источника данных (например, перевод M в T-SQL или в запрос OData), а затем просит источник выполнить этот запрос. Вы написали код M, но свёртывание запроса выгрузило некоторую часть выражения во внешний источник данных, переписав логику M на родной язык источника.[1]

Свёртывание запросов — это ключевая концепция в экосистеме Power Query. Она создает потенциал для значительного повышения производительности. Благодаря таким механизмам, как индексы и разбиение данных в памяти по страницам, источник данных часто находит нужные данные гораздо эффективнее, чем при потоковой передаче необработанных строк из источника в Power Query для локальной обработки.

[Предыдущая заметка](#)   Следующая заметка

Источник также может выполнять другую обработку, такую как агрегирование и объединение, опять же с гораздо большей производительностью, чем Power Query локально. В дополнение к этим преимуществам перенос выполнения в источник обычно уменьшает количество данных, которые необходимо отправить назад в Power Query. В связи с этим максимально возможное свёртывание запросов обычно более эффективно (и поэтому весьма желательно), вместо того, чтобы Power Query обрабатывал все данные на своей стороне.

Несмотря на всю важность, свёртывание запросов не является частью языка M. Вы можете написать собственный движок M, который на 100% совместим со спецификацией языка, даже не зная, что существует свёртывание запросов. Как случилось, что ключевая концепция Power Query не стала частью языка?

Этого не должно быть

Давайте еще раз посмотрим на свёртывание запросов. Мы сосредоточимся не на том, что это такое или почему это выгодно (мы изучили это ранее – см. Язык M Power Query. Парадигма и Табличное мышление), а рассмотрим, как это работает. Мы сделаем это на примере свёртывания с помощью Table.View.

Изучив тему, мы глубже поймём свёртывание, что повысит качество написания запросов и отладки свёртывания. Знания также пригодятся, если вы решите разработать пользовательский коннектор или переопределить / дополнить существующий.

Резюме свёртывания запроса

Для начала подведем итоги. Допустим, вы просите оценить следующий код M:

```
let
   Source = GetData(),
   Filtered = Table.SelectRows(Source, each [Code] = 50),
   First3 = Table.FirstN(Filtered, 3)
in
   First3
```

Предположим, что источник данных – Microsoft SQL Server. Свёртывание запросов перепишет вышеизложенное во что-то более эффективное, и передаст приложению на исполнение.

```
let
  Source = Sql.Database("SomeServer", "SomeDB"),
  NativeResults = Value.NativeQuery(
    Source,
    "SELECT TOP (3)
    FROM SomeTable
    WHERE Code = 50"
```

[1] Заметка написана на основе статьи Power Query M Primer (Part 23): Query Folding I. Если вы впервые сталкиваетесь с Power Query, рекомендую начать с Марк Мур. Power Query. – Здесь и далее примечания Багузина.

```
  )
in
 NativeResults
```

Логика, которую вы изначально выразили в виде пары вызовов функций Table.*, была преобразована в собственный запрос T-SQL. Логика M была свернута в T-SQL.

Переписать

Свёртывание запроса включает в себя интеллектуальное переписывание выражения M. Power Query – не является движком. Он лишь один из компонентов более крупной системы. Power Query принимает выражение, которое вы попросили обработать, и проверяет его. Если он решает, что более эффективным способом достижения того же результата было бы выполнение чего-то другого, в частности, чего-то, что свернуто в источник данных, он соответствующим образом адаптирует (или переписывает) ваше выражение.

Этот процесс оптимизации происходит после отправки выражения на оценку, но до фактического создания данных.

Язык M агностичен[2]

Независимо от того, происходит свёртывание запросов или нет, движок выполняет M-логику. Это верно независимо от того, является ли эта логика дословной из того, что вы написали, или адаптацией, созданной промежуточным процессом Power Query, который перехватил и отредактировал вашу логику. В любом случае, движок попросили оценить логику M, и он сделает все возможное.

Именно поэтому язык M не зависит от свёртывания запросов. Для того, чтобы Power Query мог сворачивать запросы, не требуется никаких специальных конструкций языка M, поэтому нет необходимости в том, чтобы спецификация языка содержала сведения, относящиеся к свёртыванию запросов. (Это хорошее разделение обязанностей. Почему язык должен заниматься чем-то, что с ним не связано?)

Что такое сворачиваемый (foldable)?[3]

Возможно, вы слышали как спрашивают: «Какие функции / методы свёртываются?». Скорее всего, вы и сами задумывались об этом. Почему нельзя дать простой ответ? Просто перечислить методы, которые сворачиваются?

Оказывается, нельзя! Но почему? Потому, что ответ на вопрос «какие методы можно свернуть?» зависит от множества факторов:

- версии Power Query – используемая версия определяет методы, которые могут сворачиваться, и методы, которые не могут сворачиваться; разные версии Power Query могут поддерживать сворачивание различных методов.
- коннектора – он может сворачивать только те операции, на которые был запрограммирован; внешний источник может не поддерживать все операции, которые Power Query готов свернуть.
- набора данных – иногда детали набора данных влияют на сворачиваемость.

Возьмем, например, Table.Group. Текущие версии Power Query могут сворачивать эту операцию, поэтому она проходит тест *версия Power Query*. Однако группировка может выполняться только некоторыми источниками данных, поэтому операция может удовлетворять или не удовлетворять критерию *коннектор.*

Например, Microsoft SQL Server имеет встроенные возможности для GROUP BY, поэтому его коннектор поддерживает свёртывание этой операции. А базовый веб-API может не предлагать

---

[2] Агностицизм в теории познания предполагает, что поскольку полученный в процессе познания опыт неизбежно искажается сознанием субъекта, то субъект принципиально не способен постичь точную и полную картину мира, то есть, хотя познание возможно, но оно всегда остаётся неточным.

[3] ChatGPT поясняет, что перевод термина *foldable* на русский язык в контексте языка M Power Query может быть проблематичным, так как нет однозначного эквивалента. Можно использовать следующие варианты: *Сворачиваемый* – подчеркивает способность функции сворачиваться и выполняться на стороне источника данных; *Оптимизируемый* – указывает на возможность оптимизации функции для выполнения на источнике данных; *Выполняемый на стороне источника данных* – прямой перевод.

способ запроса на выполнение группировки. В этом случае свёртывание Table.Group не имеет смысла, так как операция не может быть выгружена в источник.

При определении поддержки свёртывания коннектор может учитывать не только тип операции (например, *группировка по*), но и специфику операции (конкретные параметры) и даже конкретный набор данных, к которому осуществляется доступ.

Или, скажем, веб-API поддерживает два метода доступа к данным: разбиение по страницам всех записей или поиск одной записи по ее первичному ключу. В соответствии с этим коннектор может поддерживать свёртывание Table.SelectRows, но только в том случае, если фильтрация является тестом на равенство и тестируемый столбец является первичным ключом. Например, для *Orders* с первичным ключом *OrderID*, Table.SelectRows(source, each [OrderID] = 123) может свернуться, а Table.SelectRows(source, each [OrderID] <= 100) – нет. В этом случае вопрос о том, является ли функция Table.SelectRows сворачиваемой (foldable), не имеет статического фиксированного ответа. Вместо этого коннектор динамически определяет ответ на основе специфики типа фильтра строк и первичного ключа.

Из сказанного следует, что Power Query должен иметь какой-то механизм для динамического взаимодействия с коннектором, так как ему нужен способ спросить коннектор, может ли он сворачивать определенные операции.

Чтобы лучше понять это, давайте сделаем шаг назад.

Знакомство с табличными представлениями

Там, где происходит свёртывание запросов, M-код взаимодействует с тем, что выглядит как таблица, но на самом деле таблицей не является. Имя этому подобию таблицы – *представление*. Оно часто ассоциируется с пользовательскими коннекторами, так как лежит в основе их реализации. Но использование не ограничивается этим контекстом, так как представления также могут быть созданы в обычном M-коде.

Изучим, как работают представления. Как можно использовать представления для реализации свёртывания.

Определение представления

Представление строится вокруг набора обработчиков. Power Query использует эти обработчики для двух целей: для выполнения запросов, запрашивая у представления настройку типа или формы данных, которые оно может позже вернуть, и для выдачи команд, сообщающих представлению о создании определенного фрагмента информации (например, строк таблицы).

Основное усилие, связанное с созданием представления, заключается в определении этих обработчиков. Однако, прежде чем перейти к обработчикам, давайте рассмотрим другой шаг, связанный с созданием представления: выбор режима представления.

Table.View – стандартная библиотечная функция для создания представлений, генерит представления в одном из двух режимов: «переопределить» или «с нуля» (это термины автора, а не официальные названия Microsoft).

В режиме «переопределения» представление, созданное Table.View, используется для переопределения или дополнения поведения существующей таблицы. Новое представление определяет обработчики для операций. Операции, не обработанные им, возвращаются к базовой таблице. Благодаря этому резервному варианту минимально необходимое число обработчиков в режиме «переопределения» может быть равно нулю.

В отличие от этого, представление «с нуля» не основано на верхней части базовой таблицы. Вместо этого оно определяет всё обязательное поведение таблицы, а также любое необязательное, которые он хочет реализовать.

Режим представления определяется первым аргументом Table.View:

- Если это таблица (включая таблицу, которая сама определена другим представлением), включается режим «переопределить».
- Если он равен null, это режим «с нуля».

Мы сосредоточимся на создании представления в режиме «с нуля».

Минимальное количество обработчиков

Представление «с нуля» должно, как минимум, определять обработчики для двух основных табличных операций: GetType и GetRows. Без них представление не будет действовать как таблица. В данном контексте они являются обязательными. Обработчики определяются с помощью записи, которая передается в качестве второго аргумента Table.View. Имена полей записи служат именами обработчиков.

Запрос 1[4]

```
let
  MyDataSource = Table.View(
    null,
    [
      GetType = () => type table [ID = number, Name = text],
      GetRows = () => #table({"ID", "Name"}, {{ 1, "Joe" }, { 2, "Paul" }})
    ]
  )
in
  MyDataSource
```

Обращение к этому представлению (например, при вычислении MyDataSource в предложении *in*) вернет таблицу с типом, заданным GetType(), и строками, созданными вызовом GetRows():

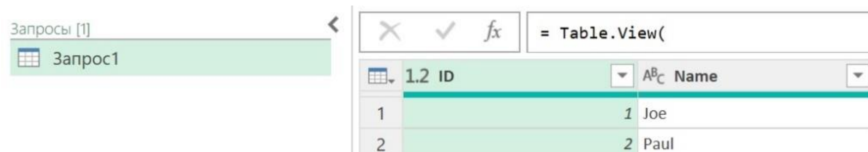

Рис. 1. Представление таблицы с нуля

Обратите внимание, что Power Query потребовалось два вызова функций, чтобы получить всё необходимое для представления таблицы. Строки таблицы извлекаются с помощью GetRows, а тип таблицы – GetType. Столбцы должны иметь одинаковые имена и задаваться в одном и том же порядке в двух обработчиках. Прелесть этого двухэтапного процесса заключается в том, что он позволяет Power Query получать информацию о типе без получения строк, что Power Query очень любит.

Обычно обработчик GetRows выполняет вызов внешнего источника данных. GetType также может вызывать источник данных для динамического считывания сведений о схеме, если они ранее не были кэшированы другими обработчиками в представлении.

Реальный пример может выглядеть так:

```
let
  MyDataSource = Table.View(
    null,
    [
      GetType = () => ComputeTypeByReferencingExternalSource(),
      GetRows = () => ReadRowsFromExternalSource()
    ]
  )
in
  MyDataSource
```

Пока это не впечатляет. Мы могли бы отказаться от Table.View, просто взяв данные из ReadRowsFromExternalSource() и приписав им тип, возвращаемый ComputeTypeByReferencingExternalSource().

```
let
  Data = ReadRowsFromExternalSource(),
```

[4] Номер соответствует запросу в приложенном Excel-файле

```
  Type = ComputeTypeByReferencingExternalSource(),
  TypeAscribed = Value.ReplaceType(Data, Type)
in
  TypeAscribed
```

Мощь Table.View проявляется, когда мы добавляем другие обработчики.

Подсчет строк

Возвращаясь к нашему примеру, предположим, что нужно подсчитать число строк в таблице (например, выполнить Table.RowCount(MyDataSource)). В настоящее время нет специальной обработки для подсчета строк, поэтому операция не будет свёрнута.

Это не мешает работе операции. Она просто будет обрабатываться внутри M, а не выгружаться в источник данных. Чтобы вычислить счетчик, Power Query вызовет обработчик GetRows представления, подсчитает возвращаемые строки, а затем вернет общее количество. При этом все строки извлекаются из источника и обрабатываются локально в Power Query.

Было бы гораздо эффективнее, если бы источник предоставлял опцию, при которой он выполняет подсчет на стороне сервера, а затем просто возвращает общую сумму. Допустим источник позволяет это, если мы изменим вызов API с https://somewhere/data на https://somewhere/data?count. Мы хотим свернуть операцию Table.RowCount, и не обрабатывать её локально в Power Query. Как это можно использовать в Table.View?

Добавим еще один обработчик:

```
let
  MyDataSource = Table.View(
    null,
    [
      GetType = () => ComputeTypeByReferencingExternalSource(),
      GetRows = () => ReadRowsFromExternalSource(),
      GetRowCount = () => GetRowCountFromExternalSource() //вызывая https://somewhere/data?count
    ]
  )
in
  Table.RowCount(MyDataSource)
```

Теперь Table.RowCount не приводит к тому, что Power Query извлекает все строки из GetRows и подсчитывает их локально. Вместо этого вызывается обработчик GetRowCount. Число, которое он выводит, возвращается в качестве счетчика Table.RowCount. Благодаря этому обработчику часть логики M (операция Table.RowCount) была преобразована из M в запрос, выполняемый внешним источником данных.

Чтобы доказать, что свёртывание действительно произошло, вернемся к Запросу 1 и обновим его для обработки GetRowCount. Однако, мы заставим GetRowCount возвращать жестко запрограммированное, неправильное значение.

Запрос 2

```
let
  MyDataSource = Table.View(
    null,
    [
      GetType = () => type table [ID = number, Name = text],
      GetRows = () => #table({"ID", "Name"}, {{ 1, "Joe" }, { 2, "Paul" }}),
      GetRowCount = () => 1
    ]
  )
in
  Table.RowCount(MyDataSource) // возвращает 1
```

Запрос 2 возвращает значение 1, которое является неправильным, жестко заданным значением обработчика GetRowCount. Это доказывает, что использовался обработчик а не фактический подсчет строк таблицы.

Если вместо этого мы выполним подсчет строк самим Power Query (скажем, путем вызова Table.Buffer, который блокирует свёртывание последующих операций), возвращается число строк, равное 2. Это доказывает, что Power Query не использовал обработчик представления GetRowCount, а вместо этого сам подсчитывал строки:

Запрос 3

```
let
  MyDataSource = Table.View(
    null,
    [
      GetType = () => type table [ID = number, Name = text],
      GetRows = () => #table({"ID", "Name"}, {{ 1, "Joe" }, { 2, "Paul" }}),
      GetRowCount = () => 1
    ]
  )
in
  Table.RowCount(Table.Buffer(MyDataSource)) // возвращает 2
```

Настройка возвращаемого содержимого

Теперь у нас есть понимание потенциала Table.View. Мощность функции будет расти по мере того, как свёртывание расширится на всё большее число операций. Как это реализовать? Как вы уже догадались: внедрив больше обработчиков!

Однако, в отличие от *GetType*, *GetRows* и *GetRowCount*, которые выводят либо данные, либо сведения о них, большинство других обработчиков имеют другую цель – настройку того, что возвращать, а что – нет.

Рассмотрим операцию Table.FirstN(data, 10) и Table.Skip(data, 5). Их соответствующими обработчиками являются *OnTake* и *OnSkip* соответственно.

```
let
  MyDataSource = Table.View(
    null,
    [
      GetType = () => ComputeTypeByReferencingExternalSource(),
      GetRows = () => ReadRowsFromExternalSource(),
      GetRowCount = () => GetRowCountFromExternalSource(),
      OnTake = (count as number) => …,
      OnSkip = (count as number) => …
    ]
  )
in
  Table.FirstN(Table.Skip(MyDataSource, 5), 10)
```

Заметили разницу в именовании? Новые имена обработчиков начинаются с *On*, в отличие от обработчиков *Get*, с которыми мы встречались ранее. Эти два префикса разграничивают основные категории обработчиков:

- обработчики *Get* возвращают что-либо (например, данные или сведения о данных),
- обработчики *On* настраивают поведение представления (за одним исключением, к которому мы вернемся позже).

В то время как обработчики *Get* возвращают данные или связанные с ними сведения, обработчики *On* ведут себя иначе. Чтобы настроить обработку группы сворачиваемых операций, Power Query сначала вызовет несколько обработчиков *On*, а затем перейдет к получению окончательного набора данных. В примере выше Power Query сначала вызовет *OnTake* и *OnSkip*, и лишь затем перейдет к *GetRows* для получения строк из представления. Возвращение каждым

обработчиком *On* фактических данных было бы преждевременным, что привело бы к выполнению ненужной работы. Что же тогда должны вернуть эти обработчики?

Новый Table.View «запоминает», что нужно свернуть.

Обработчик *On* первым захватывает сведения о запросе на свёртывание, сохраняя инструкции таким образом, чтобы возвращаемый им новый Table.View получал доступ к релевантным данным. Обработчики *Get* позже будут вызваны в этом новом представлении, и извлекут данные из него. Если вместо этого в новом представлении снова вызывается обработчик *On*, он дополнит или изменит представление, возвращая более новое представление Table.View.

В языке с изменяемыми переменными обработчики *On*, вероятно, не возвращали бы новые представления, а просто обновили переменную *instructions*, принадлежащую текущему представлению. Неизменность M этого не позволяет. Вместо этого, чтобы отразить новые инструкции, необходимо вернуть новое представление, имеющее доступ к соответствующим измененным инструкциям, вычисленным текущим обработчиком. Как нам это сделать?

Мы можем сохранить инструкции по свёртыванию в записи состояния. Далее определить функцию, которая принимает запись состояния в качестве входных данных и возвращает Table.View. Код внутри этой функции, создающей представление, включая все обработчики, которые она определяет для создаваемого представления, будет иметь доступ к аргументам, передаваемым функции. Благодаря замыканию обработчики, определенные внутри функции, создающей представление, могут получить доступ к записи о переданном состоянии при их последующем вызове, даже если функция, создающая представление, давно завершила свою работу.

```
let
  GetView = (state as record) => Table.View(
    null,
    [
      GetType = () => ComputeTypeByReferencingExternalSource(state),
      GetRows = () => ReadRowsFromExternalSource(state),
      GetRowCount = () => GetRowCountFromExternalSource(state),
      OnTake = (count as number) => @GetView(state & [Take = count]),
      OnSkip = (count as number) => @GetView(state & [Skip = count])
    ]
  ),
  MyDataSource = GetView([])
in
  Table.FirstN(Table.Skip(MyDataSource, 5), 10)
```

Для создания начального представления вызывается функция, создающая представление (в данном случае GetView), передающая пустую запись в качестве состояния.

Внутри этой функции каждый обработчик *On* определен для вызова одной и той же функции, создающей представление, передавая ей текущее (т.е. существующее) состояние, объединенное с соответствующими сведениями о новом состоянии, связанным с вызовом обработчика. Например, при вызове OnTake(6) будет создано и возвращено новое представление, состояние которого равно предыдущему состоянию с добавленным в него [Take = 6].

Когда, в конце концов, вызывается один или несколько обработчиков *Get*, они получают доступ к последнему состоянию благодаря замыканию. Таким образом, когда вызывается GetRows и ReadRowsFromExternalSource, этому последнему методу присваивается запись состояния [Take = 6]. Он учитывает это состояние при получении данных из источника, ограничивая количество возвращаемых строк шестью.

Итак, в представлении таблицы Power Query взаимодействует с обработчиками *On* для настройки свёртывания. Каждый успешный вызов обработчика *On* постепенно создает внутренний набор инструкций, возвращая всё новое и новое представление. Затем один или несколько обработчиков *Get* вызываются в последнем возвращенном представлении.

Условная обработка

Свёртываемость операции может зависеть не только от типа операции, но и от параметров операции и даже от внешнего набора данных. То, что такое возможно, подразумевает, что должен существовать какой-то механизм для Table.View, чтобы условно рассматривать свёртывание операции.

Если мы не хотим, чтобы представление поддерживало свёртывание операции в целом, мы просто не предоставляем для него обработчик. Здесь ситуация иная: операция иногда должна обрабатываться, поэтому обработчик должен существовать. Вместо этого при вызове обработчик должен решить, будет ли он на самом деле обрабатывать запрос на свёртывание. Если он решит отклонить запрос, он должен сообщить об этом обратно в Power Query. Как это реализуется?

Заставьте обработчик вызвать ошибку, если он решит не обрабатывать запрос. Ошибка — это способ сигнализировать Power Query, что запрос на свёртывание был отклонен.

Допустим, мы хотим поддерживать свёртывание запросов сортировки, например Table.Sort. *OnSort* является соответствующим обработчиком. При вызове ему передается список записей сортировки, по одной записи на столбец:

[Name = "имя столбца для сортировки", Order = целое число]

где Order = Order.Ascending или 0, для сортировки по возрастанию и Order.Descending или 1 – для сортировки по убыванию.

Однако есть одна загвоздка. API удаленной системы, с которой мы взаимодействуем, поддерживает сортировку только в порядке возрастания. Чтобы соответствовать этому, наше представление должно уметь условно обрабатывать *OnSort*. Запрос на свёртывание, если запись сведений о сортировке указывает несовместимый порядок сортировки, должен отклоняться.

```
Table.View(
  null,
  [
    OnSort = (sortDetails as list) =>
      if List.MatchesAny(sortDetails, each [Order] <> Order.Ascending)
      then error "Only supports Order.Ascending sorting"
      else @GetView(state & [AscendingSortColumns = List.Transform(sortDetails, each [Column])]),
   …
  ]
)
```

Здесь *OnSort* сначала проверяет, содержат ли сведения о сортировке какие-либо записи, в которых порядок сортировки не *по возрастанию*. Если это так, возвращается ошибка. В противном случае возвращается новое представление с записью состояния, включающей список столбцов для сортировки.

Подводя итог, можно сказать, что вызов обработчика в Power Query – это запрос, запрашивающий представление для свёртывания операции. Представление может принять запрос, и вернуть новое представление, как для обработчиков *On*. Обработчик может отклонить запрос, вызвав ошибку. Это означает, что конкретная операция не будет свёрнута.

Обычно отклонение представлением запроса на свёртывание не означает остановку всех свёртываний. Скорее, представление просто отклоняет текущий запрос на свёртывание. Ответственность за выполнение операции, которую пытались свернуть, ложится на Power Query. Операция будет обработана локально. Благодаря неизменности M ранее возвращенное представление по-прежнему находится в той же форме, что и при создании. Представление по-прежнему может обрабатывать вызовы обработчика *Get*, а также вызовы других обработчиков *On*.

Если в последнем примера кто-то пытается выполнить сортировку по убыванию…

```
Table.Sort(MyDataSource, {"Amount", Order.Descending})
```

… обработчик *OnSort* представления выдает ошибку (так как он поддерживает только сортировку по возрастанию), но это не останавливает выполнение сортировки или использование

представления. Вместо этого Power Query извлекает строки из представления, а затем сортирует их локально.

Прямой запрос (Direct Query)

Прямой запрос является особым случаем: возврат к локальной обработке в случае ошибки не предусмотрен. При использовании Direct Query как необработанные операции (когда представление не реализует связанный обработчик), так и отклоненные запросы на свёртывание (когда связанный обработчик выдает ошибку) приводят к ошибке всего запроса Direct Query. Таким образом, в Direct Query представление отвечает за обработку всех операций свёртывания. Все, что не обрабатывается представлением, будет полностью непригодно для использования в Direct Query.

К счастью, вам не нужно беспокоиться об этом, если вы не создаете пользовательский коннектор с поддержкой Direct Query, потому что (насколько мне известно) вы не можете определить Table.View для Direct Query вне исходного кода коннектора.

Остальная часть истории

Наша цель изучения Table.View состояла в том, чтобы получить общее представление, как обработчики представлений используются для реализации свёртывания запросов, а не исследовать каждую их деталь. Для полноты картины стоит отметить, что существует (как и следовало ожидать) ряд других обработчиков *On*. За исключением *OnInvoke*, все они возвращают новые представления.

OnInvoke — это обработчик, используемый для включения поддержки свёртывания для методов, которые не сопоставляются с конкретными обработчиками *On*. Таким образом, *OnInvoke* может возвращать новое представление (как и другие обработчики *On*) или фактические значения данных (в отличие от других обработчиков *On*).

Подробнее см. [Custom Folding Arbitrary Functions: OnInvoke & Table.ViewFunction]().

Мораль

Надеюсь, теперь вы понимаете сложность ответа на вопрос «какие методы свёртываются?» Может показаться, что это простой вопрос, но эта простота испаряется после изучения динамических взаимодействий между Power Query и коннекторами данных.

Не существует фиксированного списка методов, которые всегда свёртываются. Список потенциально подходящих методов может изменяться в зависимости от используемой версии Power Query. Потенциально подходящий для свёртывания метод, может быть заблокирован предыдущим несвёрнутым методом или неподходящим коннектором. При принятии решения о свёртывании коннектор учитывает аргументы метода, а также внешний набор данных.[5]

В следующей заметке

Это не единственная мораль, которую можно извлечь из свёртывания запросов, но наше исследование достаточно длинное для этой части. В следующий раз мы вернемся к обсуждению, получив больше знаний о том, как работает свёртывание запросов, и почерпнем больше практических сведений, которые помогут нам эффективнее работать с Power Query. Свёртывание запросов может привести к тому, что представление будет взаимодействовать с источником данных больше, чем вы интуитивно ожидаете. Почему? Читайте в следующей заметке.

См. также

[Custom Folding Arbitrary Functions: OnInvoke & Table.ViewFunction]()

[Custom Folding Joins: Tunneling State Between Table.Views]()

[Custom Connectors: Introducing Table.ViewError & Terminal Errors]()

---

[5] Любопытный комментарий к этой заметке на сайте автора: «Только что использовал в Table.View жесткое кодирование GetType, и это значительно ускорило редактирование запросов в потоке данных Power BI».