Power Query вызывает источник больше раз, чем вы ожидаете

В прошлый раз мы рассмотрели, как можно реализовать свёртывание запроса с помощью Table.View. Мы разобрались, почему нельзя однозначно ответить на вопрос «какие функции языка M сворачиваются?». Ответ зависит от версии Power Query, коннектора данных, аргументов функции и даже набора внешних данных, к которым осуществляется доступ.

Как вы думаете, Power Query взаимодействует с каждым внешним источником при обработке выражения только один раз? Гарантируется ли получение одинаковых результатов: а) при обработке выражения локально PQ, б) частичной свёртке в источник, в) полной свёртке?[1]

Предыдущая заметка Следующая заметка

Счастливо болтая без умолку

При получении данных из источника, как вы думаете, какое взаимодействие происходит между Power Query и источником? Может быть, что-то вроде: открой соединение, выполни аутентификацию. Отправь один запрос «базы данных запросов». В ответ каждая результирующая строка отправляется из источника в Power Query ровно один раз.

Звучит разумно и, возможно, именно это произойдет, если вы вручную запросите источник с помощью Azure Data Studio или SSMS. Power Query обычно работает не так. Он слишком болтлив и взаимодействует с источником больше, чем можно ожидать. И вам полезно знать об этом.

Вспомните, что мы в прошлый раз узнали об обработчиках Table.View. Представление дает информацию о типе таблицы через обработчик *GetType* отдельно от самих данных, запрашиваемых через обработчик *GetRows*. Когда Power Query извлекает данные из табличного представления, ему требуется как тип таблицы, так и ее данные, поэтому он вызывает оба обработчика.

Если свёртывание не выполняется, получение данных с помощью табличного представления выглядит следующим образом:

1) создай экземпляр представления, то есть, либо код, написанный вами, либо код внутри коннектора, написанный кем-то другим, вызывает функцию *Table.View*;
2) в этом представлении вызови обработчик *GetType*;
3) в этом представлении вызови обработчик *GetRows*.

В большинстве случаев эта последовательность приводит по крайней мере к двум взаимодействиям с источником данных. Первое позволяет вычислить тип. Второе – извлечь результирующий набор данных.

В зависимости от сложности табличного представления и возможностей источника данных то, что извлекается в рамках вычисления типа, может варьироваться от простых метаданных об ожидаемом результирующем наборе до целого результирующего набора, тип которого считывается, а его строки игнорируются. Потенциальная выборка всех строк может показаться странной, когда требуется только тип, но помните, что не каждый источник данных способен отдельно предоставить метаданные. Иногда единственный способ определить тип того, что будет возвращено, – это заставить источник вернуть данные полностью.

Общение все больше и больше

Как мы знаем, Power Query настраивает операции свёртывания, вызывая обработчики *On*. Оказывается, после вызова каждого обработчика *On* Power Query также запрашивать измененный тип. Результирующая цепочка вызовов обработчика часто выглядит так:

---

[1] Заметка написана на основе статьи Power Query M Primer (Part 24): Query Folding II. Если вы впервые сталкиваетесь с Power Query, рекомендую начать с Марк Мур. Power Query. – Здесь и далее примечания Багузина.

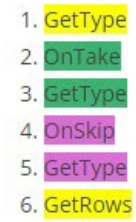

Рис. 1. Потенциальное число взаимодействий Power Query с источником данных

Каждый из этих обработчиков технически может вызывать источник. OnSkip и OnTake как правило программируют так, чтобы они не обращались к источнику. А вот GetType наоборот, связывается с источником, чтобы динамически вычислять тип, который он вернет по запросу (вместо того, чтобы обращаться к кэшированной информации).

В случае более сложной операции также нет определенности. Например, обработчик объединения OnJoin решая, сворачивать ли запрос, основывается на том, что ему уже известно (например, просматривая информацию, которая была кэширована). Или OnJoin может извлекать (мета)данные из источника.

Может быть задействовано еще больше коммуникаций. Представьте себе объединение, обрабатываемое табличным представлением, с таблицей из другого источника данных. Если другая таблица содержит умеренное количество строк, представление решит свернуть значения ключевых столбцов из этой другой таблицы в запрос, который отправляет источнику, а затем присоединить эти предварительно отфильтрованные результаты к данным другого источника.

Если же другой источник содержит больше определенного количества строк, представление может отказаться сворачивать объединение, оставив его для внутренней обработки Power Query. В этом случае представление выполнит выборку зонда к другому источнику, чтобы определить количество содержащихся в нем строк, а затем отклонит запрос на свёртывание. Позже, Power Query извлечет строки из другого источника в рамках обработки объединения. Поведение представления «с возможностью предварительного фильтра» привело к тому, что в коммуникации была добавлена дополнительная связь с источником (выборка зонда).

Дело не в особенностях того, как работает конкретное табличное представление, обработчик или коннектор. Скорее, это иллюстрирует, что табличные представления (и связанный с ними код) могут взаимодействовать с источниками больше, чем вы ожидаете.

Производные оценки

Мало того, что вычисление одного выражения может привести к многократному взаимодействию табличного представления с источником, так Power Query может вычислять выражение или его варианты несколько раз. Это приводит к еще большему числу сообщений между PQ и источником.

Например, брандмауэру защиты данных Power Query могут потребоваться сведения о вашем выражении, чтобы понять, как реализовать защиту уровня конфиденциальности. Чтобы получить необходимые сведения, он может инициировать предварительный запуск вашего выражения, но с обернутым в Table.FirstN(YourQuery, 1000). Затем брандмауэр использует ответ, чтобы пересмотреть ваш исходный запрос, подключив соответствующий код конфиденциальности.

Через несколько мгновений этот переписанный запрос выполняется для получения данных, которые Power Query возвращает в качестве ответа на исходный запрос. Вы ожидали, что ваш запрос будет выполнен один раз, а на самом деле он выполнялся несколько раз и не в точности, как был написан. Правда, это не означает, что никакое исходное выражение не выполняется в точности, как его написали, но часто это не так.

И это еще не все: оказывается, для некоторых источников данных (таких как OData, а иногда и ODBC) Power Query не всегда может заранее определить их точные возможности свёртывания. Чтобы компенсировать это, Power Query начинает со свертывания всех операций, которые, по его мнению, могут быть свёрнуты. Если такой запрос не работает, Power Query будет повторять попытки, каждый раз уменьшая количество свёртываний, до тех пор, пока не найдет работающую версию… Или вернется к исходному запросу без свёртывания. Такое динамичное поведение обеспечивает еще больше взаимодействий с источником.

Надеюсь, что я разрушил ваше ошибочное мнение, что после проверки подлинности Power Query выполняет один вызов источника.

Точную последовательность взаимодействий между Power Query и источником следует рассматривать как техническую подробность, на которую нельзя повлиять. У вас может возникнуть соблазн изучить конкретный сценарий, чтобы выяснить, как выглядит последовательность взаимодействий Power Query с источником. Хотя это может быть интересно, не думайте, что в следующий раз произойдет нечто аналогичное.

Изменения могут быть связаны с новой версией Power Query, обновлением коннектора, переносом выражения в другую среду Power Query, изменениями в метаданных в источнике и т.д.

Важные выводы

Power Query неоднозначен (poly-fetches). Задание на вычисление выражения может привести к множеству взаимодействий Power Query с внешними источниками, при этом точное количество подключений не предсказуемо.

Не используйте Power Query для изменения данных. Никаких *Вставить* или *Удалить*. Никаких *HTTP PUT* или *POST*, которые что-то меняют. Poly-fetches поведение Power Query означает, что запрос может быть выполнен несколько раз. Если запрос включает что-то типа «INSERT ...» или «UPDATE ...», эти изменения могут быть внесены несколько раз. Хотя вы планировали сделать это однократно.

Небольшой «секрет»: корпорация Майкрософт создала внутренний, недокументированный тип с соответствующими функциями, чтобы позволить Power Query выполнять изменения данных таким образом, чтобы избежать неоднозначного поведения. Если Microsoft посчитала необходимым добавить специальные функции для обработки внесения изменений в данные, значит они понимают, что стандарт Power Query для внесения изменений не годится. Следовательно и нам не нужно использовать Power Query для изменения данных.

В идеале коннекторы данных и табличные представления должны получать сведения о типах без получения фактических данных. Если это не так вы можете *жестко указать тип данных* в обработчике *GetType*.

Предположим, вы выполняете Value.NativeQuery(SomeSource, SomeQuery) в условиях, когда коннектор не может получить метаданные для вычисления типов, но вы знаете, какими они должен быть. Укажите типы в Table.View:

```
Table.View(
  null,
  [
    GetType = () => (укажите тип здесь)
    GetRows = () => Value.NativeQuery(SomeSource, SomeQuery)
  ]
)
```

Конечно, важен контекст. Если ваш исходный запрос выполняется мгновенно, зачем беспокоиться о повышении производительности.

Поскольку Power Query в попытке свернуть запрос выполняет несколько итераций, желательно иметь такое *исходное выражение, которое может быть полностью свёрнуто*. Например, если какой-либо компонент Power Query решит выполнить Table.FirstN(YourQuery, 1000) и YourQuery полностью свернётся, то велика вероятность, что выражение с оболочкой Table.FirstN также будет полностью свёрнуто.

Кэш постоянных запросов

Как мы узнали, Power Query при обработке одного выражения взаимодействует с внешними источниками много раз. Некоторые из этих обращений могут быть точными копиями. Отправка одного и того же запроса источнику несколько раз неэффективна. К счастью, Power Query имеет

постоянный кэш запросов, который помогает дедуплицировать[2] (deduplicate) эти сообщения, ускоряя время выполнения и экономя усилия по обработке исходного кода.

Ранее мы обсуждали кэш довольно подробно. Напомню. Power Query может отслеживать запросы и кэшировать ответы. Когда он видит запрос идентичный прежнему (то же выражение, тот же источник и т.д.), он может перехватить запрос. Вместо того, чтобы отправлять запрос во внешний источник, Power Query вернет кэшированные данные.

Возникновение кэша запросов зависит от множества факторов, и его сложно предсказать. К этим факторам относятся размер набора данных и время завершения идентичных запросов. Включение/выключение кэша может быть одной из причин изменения производительности загрузки данных. Если возвращаемый набор станет столь большим, что превысит предельный размер кэша, то данные перестанут кэшироваться. В результате запросы, на которые ранее отвечал кэш, станут выполняться дольше, поскольку теперь каждый из них отправляется в источник.

Кэш постоянных запросов применяется только на уровне запросов на языке источника данных (native query/request level[3]). Результаты, полученные выражениями M, не кэшируются.

Как правило, время существования кэша ограничено одной операцией обновления (например, одной командой *Обновить все* в Microsoft Power BI Desktop. Однако средства разработки запросов могут использовать кэш с более длительным сроком жизни. Вот почему в редакторе Power Query вы иногда видите предупреждение «Этот предварительный просмотр может быть старше X дней». Редактор запросов информирует вас о длительном сроке действия кэша и предлагает вам запустить обновление.

Power Query и свёрнутый запрос к источнику не всегда одно и то же

Свертывание запросов должно быть прозрачным. Результаты не должны отличаться, если выражение M частично или полностью свёрнуто в источник или обрабатывается внутри Power Query. Производительность обработки может измениться, но результаты – нет.

---

[2] Дедупликация (от лат. deduplicatio — устранение дубликатов) — специализированный метод сжатия массива данных, использующий в качестве алгоритма сжатия исключение дублирующих копий повторяющихся данных. Обычно применяется для оптимизации использования дискового пространства систем хранения данных, однако может применяться и при сетевом обмене данных для сокращения объема передаваемой информации.

[3] ChatGPT поясняет, что, хотя термины *query* и *request* иногда используются как синонимы, в контексте Power Query им придается разный смысл. Query (запрос) относится к последовательности действий, выполняемых над данными в Power Query. Запрос включает шаги загрузки данных, преобразования, фильтрации, объединения и другие операции. Query – это набор инструкций, описывающих, как обрабатывать данные и получать требуемый результат. Request – запрос данных из источника данных (базы данных, файла или веб-сервиса). Request (запрос) определяет, какие данные нужно получить, какие параметры использовать и какие операции выполнить на стороне источника данных. Request – это инструкция, отправляемая источнику данных для извлечения конкретных данных или выполнения операций на основе указанных критериев.

В Power Query есть возможность выполнять нативные запросы, используя язык запросов, понятный источнику данных. Такой подход позволяет использовать специфичные функции, синтаксис и возможности источника данных, которые могут быть недоступны или ограничены в Power Query. Нативный запрос в Power Query реализуется с помощью функции Value.NativeQuery, которая принимает строку с нативным запросом и возвращает результат выполнения запроса.

```
let
   source = Sql.Database("сервер", "база данных"),
   query = "SELECT * FROM таблица",
   result = Value.NativeQuery(source, query)
in
   result
```

В этом примере мы устанавливаем подключение к базе данных SQL Server с помощью функции "Sql.Database", задаем нативный запрос и используем функцию "Value.NativeQuery" для его выполнения. Результат выполнения запроса сохраняется в переменной "result".

Такова теория. Гарантировано ли это на практике?

Если для реализации свертывания запросов используется табличное представление, оно преобразует операции, которые согласилось свернуть, в запрос на языке источника данных, а источник выполняет этот запрос, возвращая результат. Чтобы свертывание запроса было прозрачным, совокупные эффекты этих двух шагов должны давать результат аналогичный обработке внутри Power Query без свёртывания. Это происходит не всегда!

Представьте себе таблицу базы данных Microsoft SQL, содержащую строки для *Joe* и *joe*. Отфильтруйте эту таблицу с помощью выражения:

Table.SelectRows(Source, each [FirstName] = "Joe")

Если выражение обрабатывается локально Power Query (без свёртывания), будет возвращена строка для *Joe*, так как оператор равенства Power Query (=) выполняет сравнение строк с учетом регистра.

Рис. 2. Результат при локальной обработке в Power Query

При свёртывании запроса коннектор преобразует логику M во фрагмент SQL, который включит в SQL-запрос, отправляемый на сервер базы данных:

WHERE FirstName = 'Joe'

Пока всё в порядке. Этот фрагмент SQL соответствует логике исходного выражения M. Но какой результат вернется после выполнения инструкции SQL сервером базы данных?

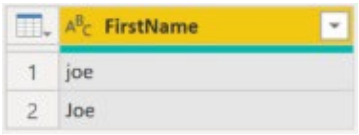

Рис. 3. Результат фильтрации свёрнутого запроса к базе SQL

Когда фильтр Table.SelectRows был свёрнут на сервер базы данных, его логика переключилась с обработки с использованием определения равенства в Power Query на определение равенства сервера, и эти два определения не идентичны. Как это принято в мире SQL Server, столбец таблицы базы данных был настроен на параметры сортировки без учета регистра, поэтому FirstName = 'Joe' соответствовал как *Joe*, так и *joe*.

К сожалению, из-за различий в парадигме между Power Query и внешним источником данных свертывание запросов не всегда прозрачно.

Иногда возникновение непрозрачности очевидно и быстро вскрывается. В других случаях отличие может оставаться незамеченным в течение длительного времени. Однако, не позволяйте непрозрачности отпугнуть вас от свёртывания запросов. Обработка, основанная на Power Query, продолжает процветать, несмотря на то, что иногда свёртывание запросов непрозрачно.

Понимание того, что поведение Power Query не всегда прозрачно (и почему оно не прозрачно), может сэкономить массу времени. В то же время, к счастью, эта проблема, похоже, не кусается так часто или сильно, как может показаться на первый взгляд.

Больше информации см. Equals Is Not Always Equivalent: When Query Folding Does Not Produce Identical Results и Equals Is Not Always Equivalent: Power Query Joins vs. SQL Joins.

В следующий раз

Довольно много функций (стандартная библиотека и коннекторы) писали не вы. Как они попадают в глобальную среду? Благодаря расширениям. Отличная тема для изучения!