```
        OUTER APPLY sys.dm_exec_query_statistics_xml(er.session_id) p
        LEFT JOIN sys.dm_exec_connections c WITH (NOLOCK) ON
            er.session_id = c.session_id
        LEFT JOIN sys.dm_exec_sessions es WITH (NOLOCK) ON
            er.session_id = es.session_id
WHERE
    er.status <> 'background'
    AND er.session_id > 50
ORDER BY
    er.cpu_time desc
OPTION (RECOMPILE, MAXDOP 1);
```

Характерные проблемы при настройке запросов

Оптимизация и настройка запросов может затрагивать несколько уровней в системе. Например, в случае сторонних приложений у вас может не быть доступа к их исходному коду, так что придется работать с предопределенным набором запросов. Пожалуй, это самый сложный случай — когда вся оптимизация сводится к созданию и изменению индексов.

Намного лучше, если можно модифицировать и запросы, и код базы данных. Эта модификация может занять много времени и требует тестирования, но результаты будут лучше, а производительность значительно повысится.

В некоторых случаях приходится выходить за рамки кода базы данных. Например, иногда стоит изменить схему базы данных, архитектуру приложения, а порой и технологию масштабирования системы. Хотя это чрезвычайно сложный процесс, он обеспечивает наилучшие результаты в долгосрочной перспективе.

Здесь мы не будем углубляться в эти подробности. Вместо этого я расскажу о нескольких распространенных недостатках, которые можно устранить с помощью индексации и изменения кода. Однако имейте в виду, что в реальной практике ваши возможности гораздо шире.

Неэффективный код

Если только вы не лишены доступа к коду, то настройку следует начать с просмотра запросов и, возможно, их рефакторинга. Существует несколько антишаблонов и проблем, которые стоит обнаружить.

Предикаты, не поддерживающие поиск и аргументы (SARG)

Предикаты, поддерживающие SARG, позволяют SQL Server использовать оператор *Index Seek* (просмотр индекса), ограничивая диапазон обрабатываемых значений ключа. Всюду, где возможно, удаляйте из запросов предикаты, не поддерживающие SARG.

«Не-SARG-абельные» предикаты часто возникают из-за функций. Чтобы вычислить предикат, SQL Server вызывает функцию для каждой обрабатываемой строки. Это относится как к системным, так и к скалярным пользовательским функциям (UDF).

SQL Server 2019 и более поздние версии умеют встраивать некоторые скалярные пользовательские функции в оператор запроса. Но даже в этих версиях встраиванию мешает множество факторов, поэтому по возможности избегайте скалярных UDF.

В табл. 5.3 показано несколько примеров того, как переписать предикаты, чтобы они поддерживали SARG. Не исключено, что в некоторых версиях SQL Server отдельные предикаты без поддержки SARG будут обрабатываться эффективнее, однако всегда безопаснее настраивать запросы вручную.

Таблица 5.3. Примеры рефакторинга не-SARGable-предикатов в SARGable-предикаты

Операция	Не поддерживает SARG	Поддерживает SARG
Математические вычисления	`Column – 1 = @Value`	`Column = @Value + 1`
	`ABS(Column) = 1`	`Column IN (-1, 1)`
Работа с датами	`CONVERT(DATETIME, CONVERT(VARCHAR(10),Column,121)) = @Date`	`Column >= @Date AND Column < DATEADD(DAY,1,@Date)`
	`DATEPART(YEAR,Column) = @Year`	`Column >= @Year AND Column < DATEADD(YEAR,1,@Year)`
	`DATEADD(DAY,7,Column) > GETDATE()`	`Column > DATEADD(DAY,-7,GETDATE())`
Поиск по префиксу	`LEFT(Column,3) = 'ABC'`	`Column LIKE 'ABC%'`
Поиск по подстроке	`Column LIKE '%ABC%'`	Используйте полнотекстовый поиск или другие технологии

Обращайте внимание на типы данных в предикатах. Операция неявного преобразования — это по сути вызов системной функции `CONVERT_IMPLICIT`, которая во многих случаях препятствует поиску по индексу. Не забудьте проанализировать предикаты `JOIN` и предложения `WHERE`. Частый источник проблем — несоответствие типов данных в столбцах, по которым происходит объединение.

Пользовательские функции

Как я уже говорил, системные и невстроенные скалярные пользовательские функции могут помешать SQL Server применять поиск по индексу. Более того, они значительно снижают производительность (особенно пользовательские функции). SQL Server вызывает их для каждой обрабатываемой строки, и это подобно вызовам хранимых процедур (в этом можно убедиться, если перехватить расширенное событие `rpc_starting` или событие трассировки `SP:Starting`).

SQL Server оптимизирует код в невстроенных пользовательских функциях из нескольких инструкций (скалярных и с табличным значением) отдельно от запроса вызывающего объекта. Обычно это приводит к менее эффективным планам выполнения. Но вот что еще важнее: когда SQL Server оценивает количество строк, которое возвратит функция с табличным значением из нескольких инструкций, то оценочное значение всегда равно либо 1, либо 100 строк — в зависимости от версии SQL Server, уровня совместимости базы данных и параметров конфигурации. Это может полностью свести на нет оценку количества элементов и породить крайне неэффективные планы.

В SQL Server 2017 это положение несколько выправилось. Одна из функций интеллектуальной обработки запросов — *выполнение с чередованием* — откладывает окончательную компиляцию запроса до времени выполнения, когда SQL Server может подсчитать фактическое количество строк, возвращаемых функцией, и завершить оптимизацию, используя эти данные. Сейчас это работает только с запросами `SELECT`, однако в будущем ситуация может измениться.

Тем не менее лучше избегать функций из нескольких инструкций и по возможности использовать встроенные функции с табличным значением. SQL Server встраивает и оптимизирует их вместе с запросами вызывающего объекта. К счастью, во многих случаях скалярные функции и функции из нескольких инструкций с табличным значением можно минимальными усилиями преобразовать во встроенные функции с табличным значением.

Временные таблицы и табличные переменные

Временные таблицы и табличные переменные оказывают неоценимую помощь при оптимизации запросов. Их можно использовать, чтобы сохранять промежуточные результаты запросов. Это позволяет упростить запросы, отчего улучшается оценка количества элементов, а планы выполнения получаются более эффективными.

С временными таблицами и табличными переменными связаны две распространенные ошибки.

Первая ошибка заключается в том, что некоторые специалисты предпочитают табличные переменные временным таблицам из-за распространенного заблуждения, будто табличные переменные — это объекты в памяти, которые не

используют базу данных tempdb и поэтому более эффективны, чем временные таблицы.

Это не так. Оба объекта обращаются к tempdb. Несмотря на то что табличные переменные немного эффективнее, чем временные таблицы, эта эффективность получается за счет ограничения: они не поддерживают статистику по первичным ключам и индексам.

С другой стороны, временные таблицы ведут себя как обычные таблицы. Они поддерживают статистику по индексам и разрешают SQL Server использовать ее во время оптимизации. Хотя в некоторых специальных случаях табличные переменные лучше, в большинстве ситуаций временные таблицы будут безопаснее. За свою карьеру я нередко добивался отличных результатов, заменяя табличные переменные на временные таблицы без каких-либо дополнительных изменений кода или индексации.

Вторая распространенная ошибка — не индексировать временные таблицы. Это плохо влияет на оценку количества элементов и может привести к неэффективным просмотрам таблиц. Рассматривайте временные таблицы как обычные и индексируйте их ради поддержки эффективных запросов, особенно когда в таблицах содержатся значительные объемы данных.

В грамотно проиндексированной временной таблице можно сохранять результаты функций из нескольких инструкций с табличным значением. Это улучшит оценку количества элементов, особенно в старых версиях SQL Server, где нет выполнения с чередованием.

Очевидно, временные таблицы и табличные переменные имеют свою цену. Их создание и заполнение связано с накладными расходами. Если применять их разумно, то преимущества могут преобладать над недостатками, но вряд ли стоит хранить в них миллионы строк. Мы поговорим об этом подробнее в главе 9.

Хранимые процедуры и ORM-фреймворки

Хотя эта тема напрямую не связана с настройкой запросов, нельзя не упомянуть фреймворки объектно-реляционного отображения (ORM, Object Relational Mapping). Сейчас они чрезвычайно распространены, и можно без преувеличения сказать, что все специалисты по базам данных их ненавидят. Эти фреймворки генерируют запросы, которые чрезвычайно сложны и плохо поддаются оптимизации.

К сожалению, приходится признать, что фреймворки упрощают разработку и сокращают ее время и стоимость. В большинстве случаев нереально и неразумно настаивать на том, чтобы разработчики приложений ими не пользовались. Что еще более важно, во многих случаях вполне можно мириться с неидеальными запросами, которые генерируют фреймворки.

Однако это не относится к запросам, критичным с точки зрения производительности. Их может быть не очень много, но все-таки эти немногие запросы потребуют скрупулезной настройки и оптимизации. В таких случаях автоматически сгенерированные и/или нерегламентированные запросы — неудачное решение. Лучше использовать хранимые процедуры, которые обеспечивают полную гибкость и поддерживают более широкий набор методов оптимизации.

Чтобы перейти на хранимые процедуры, может понадобиться модифицировать код приложения, но во многих случаях этот переход сокращает время и стоимость настройки.

Неэффективный поиск по индексу

Как вы уже знаете, операция поиска по индексу обычно эффективнее, чем просмотр индекса. Однако это не означает, что всякий поиск по индексу эффективен. SQL Server использует поиск по индексу, когда предикаты запроса позволяют изолировать диапазон строк данных от индекса во время выполнения запроса. Если этот диапазон очень велик, эффективность операции может снизиться.

Рассмотрим простой пример: я создам таблицу и заполню ее данными. Затем я выполню две инструкции `SELECT` — с предложением `WHERE` и без него, как показано в листинге 5.6.

Листинг 5.6. Неэффективность поиска по индексу

```
CREATE TABLE dbo.T1
(
    IndexedCol INT NOT NULL,
    NonIndexedCol INT NOT NULL
);
CREATE UNIQUE CLUSTERED INDEX IDX_T1
ON dbo.T1(IndexedCol);

;WITH N1(C) AS (SELECT 0 UNION ALL SELECT 0) -- 2 строки
,N2(C) AS (SELECT 0 FROM N1 AS T1 CROSS JOIN N1 AS T2) -- 4 строки
,N3(C) AS (SELECT 0 FROM N2 AS T1 CROSS JOIN N2 AS T2) -- 16 строк
,N4(C) AS (SELECT 0 FROM N3 AS T1 CROSS JOIN N3 AS T2) -- 256 строк
,N5(C) AS (SELECT 0 FROM N4 AS T1 CROSS JOIN N4 AS T2) -- 65,536 строк
,N6(C) AS (SELECT 0 FROM N3 AS T1 CROSS JOIN N5 AS T2) -- 1,048,576 строк
,IDs(ID) AS (SELECT ROW_NUMBER() OVER (ORDER BY (SELECT NULL)) FROM N6)
INSERT INTO dbo.T1(IndexedCol, NonIndexedCol)
    SELECT ID, ID FROM IDs;

SET STATISTICS IO ON
SELECT COUNT(*) FROM dbo.T1;
SELECT COUNT(*) FROM dbo.T1 WHERE IndexedCol > 0;
```

На рис. 5.15 показаны планы выполнения и статистика ввода/вывода обоих запросов. У всех строк в таблице положительные значения `IndexedCol`, поэтому

оба запроса вынуждены просматривать весь индекс. В целом поиск по индексу оказался идентичен просмотру индекса.

Неэффективный поиск по индексу часто встречается в многопользовательских системах. Возьмем, к примеру, систему обработки заказов, где данные обычно распределены по относительно небольшому количеству складов. Как правило, идентификатор пользователя (или идентификатор склада в этом примере) становится крайним левым столбцом в ключах индекса.

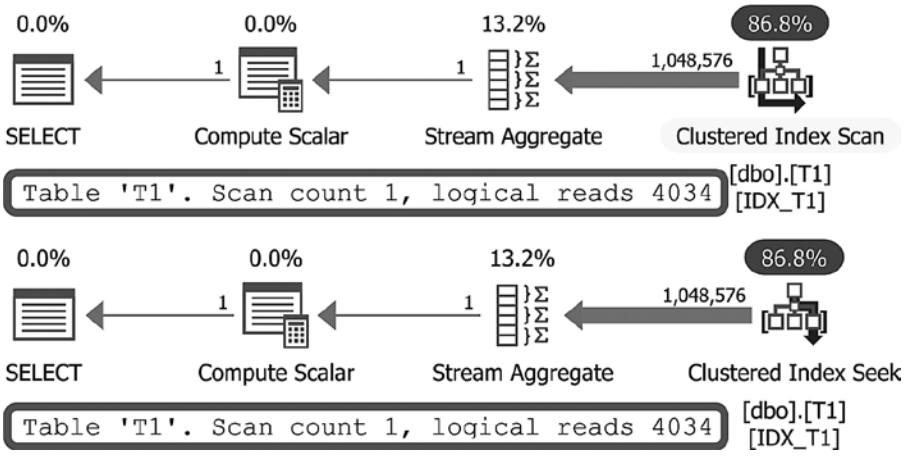

Рис. 5.15. Неэффективный поиск по индексу

Запросы в этих системах обычно имеют дело с данными от одного пользователя, который указывается в предикате в предложении `WHERE`, а это закономерно приводит к поиску по индексу в плане выполнения. Но если для каждого пользователя (то есть склада) хранится слишком много данных, вы можете получить идеальный на вид, но неэффективный план выполнения даже без просмотров. Вам понадобится применять другие предикаты, чтобы поиск по индексу стал выборочным и производительность повысилась.

Можно проанализировать эффективность поиска по индексу в плане выполнения, если изучить свойства оператора. Запустим запрос, показанный в листинге 5.7, на таблице из листинга 5.6.

Листинг 5.7. Пример запроса

```
SELECT IndexedCol, NonIndexedCol
FROM dbo.T1
WHERE
    IndexedCol BETWEEN 100 AND 150 AND
    NonIndexedCol % 2 = 0;
```

На рис. 5.16 показано несколько ключевых свойств для анализа. Этот скриншот сделан в Plan Explorer, но в SSMS отображаются те же данные.

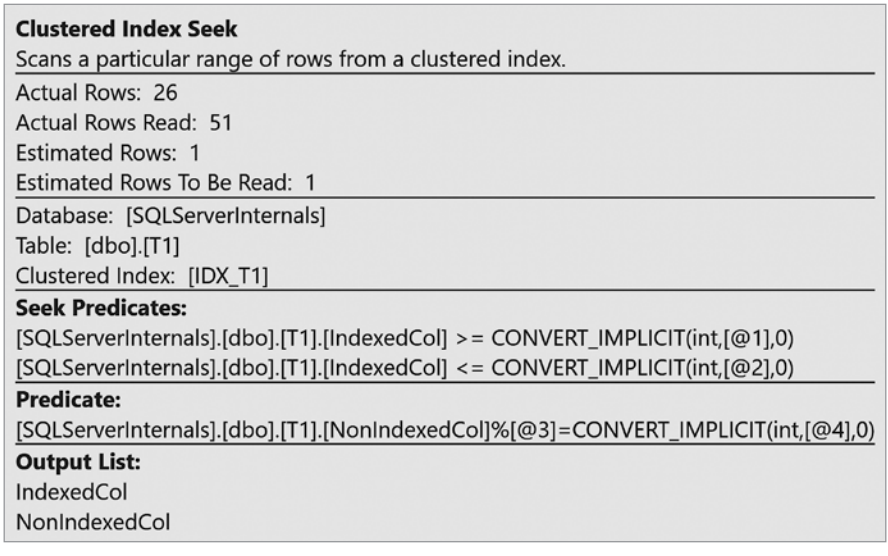

Рис. 5.16. Свойства поиска по индексу

Рассмотрим самые важные свойства оператора *Index Seek*.

Seek predicate (Предикат поиска)

Один или несколько предикатов, которые SQL Server использует, чтобы ограничить диапазон строк во время поиска по индексу. Чем избирательнее этот предикат, тем эффективнее поиск.

Predicate

Дополнительные критерии фильтрации, которые SQL Server применяет к каждой строке, прочитанной оператором *Index Seek*. Они не уменьшают размер данных, которые обрабатывает оператор, но могут сократить в плане выполнения количество строк, которые оператор возвращает. В данном случае оператор прочитал 51 строку из индекса и вернул 26 строк следующему оператору в плане выполнения.

Всегда целесообразнее уменьшать размер данных с помощью эффективного предиката поиска. Если предикаты поиска недостаточно избирательны, попробуйте реструктурировать индекс так, чтобы SQL Server мог использовать обычные предикаты в качестве предикатов поиска.

Actual rows и Actual rows read

В SSMS эти свойства называются Actual Number of Rows и Number of Rows Read. Они показывают, сколько строк вернул оператор и сколько строк было обработано во время выполнения соответственно. Большое значение Number of Rows Read говорит о том, что поиск по индексу обработал большой объем данных; это может потребовать дальнейшего изучения. Если эти два значения существенно расходятся, то, возможно, индекс неэффективен, потому что значительная часть данных отфильтровывается свойством Predicate, а не Seek Predicate.

Estimated rows и Estimated rows to be read

Эти свойства в SSMS называются Estimated Number of Rows и Estimated Number of Rows to be Read. Как уже отмечалось, можно сравнить оценочные и фактические показатели в плане выполнения, чтобы понять качество оценки количества элементов. Если количество элементов оценивается с большой ошибкой, это может указывать на неправильный выбор индекса и/или типа соединения (подробнее об этом позже).

Увидев большую ошибку оценки количества элементов, убедитесь, что статистика актуальна. Проверьте запрос и удалите конструкции, которые могут повлиять на оценку количества элементов (например, функции и табличные переменные). В некоторых случаях, особенно при сложных запросах, подумайте об их рефакторинге и/или разделении — возможно, используя временные таблицы для промежуточных данных.

Очевидно, анализировать план выполнения гораздо проще, если у вас есть фактические метрики выполнения. Хотя оценочные метрики полезны для первоначального анализа, ошибочная оценка количества элементов может дать очень неправильную или неполную картину. Имея дело с оценочными планами выполнения, проведите дополнительный анализ и посмотрите, как распределены данные в таблицах.

Неправильный тип соединения

Во время выполнения запроса SQL Server применяет множество операторов физического соединения. Они принадлежат к одному из трех типов логического соединения: *цикл*, *хеш* и *слияние*. Каждый из них оптимизирован для определенных условий, и неправильно выбранный тип может серьезно повлиять на производительность запроса. К сожалению, многие не обращают внимания на тип соединения, упуская возможности для оптимизации.

Давайте рассмотрим все три типа подробнее.

Соединение в цикле

Соединение в цикле (или соединение с вложенным циклом) — это простейший алгоритм соединения. Как и любой тип соединения, он принимает два входных операнда, которые называются *внешней* и *внутренней* таблицами. Алгоритм соединения очень прост (см. листинг 5.8). SQL Server обходит внешнюю таблицу и для каждой ее строки ищет во внутренней таблице строки для объединения.

Листинг 5.8. Алгоритм соединения в цикле (псевдокод)

```
/* Inner join */
for each row R1 in outer table
    find row(s) R2 in inner table
        if R1 joins with R2
            return join (R1, R2)

/* Outer join */
for each row R1 in outer table
    find row(s) R2 in inner table
        if R1 joins with R2
            return join (R1, R2)
    else
            return join (R1, NULL)
```

Ресурсоемкость соединения зависит от двух факторов, первый из которых — размер внешней таблицы. SQL Server обходит каждую строку внешней таблицы, находя во внутренней таблице соответствующие строки для объединения. Чем больше данных нужно обработать, тем затратнее процедура.

Второй фактор — эффективность поиска по внутренней таблице. Когда столбец (или столбцы) соединения во внутренней таблице правильно проиндексирован, SQL Server может использовать эффективную операцию поиска по индексу. В этом случае издержки поиска по внутренней таблице на каждой итерации будут относительно низкими. Без индекса SQL Server, возможно, будет вынужден просматривать внутреннюю таблицу несколько раз: по одному разу для каждой строки из внешней таблицы. Как нетрудно догадаться, это крайне неэффективно.

Соединение в цикле оптимизировано для ситуаций, когда одна из таблиц небольшая, а у другой есть индекс, поддерживающий операцию поиска по индексу для соединения. Нельзя точно определить момент, после которого соединение станет неэффективным. Оно может хорошо работать с тысячами, а иногда и с десятками тысяч строк внешней таблицы, но с миллионами строк могут возникнуть проблемы. Тем не менее в подходящих условиях этот тип соединения чрезвычайно эффективен. Он требует мало ресурсов для запуска, не использует `tempdb` и не потребляет больших объемов памяти.

Наконец, соединение в цикле — это единственный тип соединения, которому не требуется предикат равенства. SQL Server может вычислить предикат соедине-

ния между любыми двумя строками из обеих таблиц, но этот предикат может и вообще отсутствовать. Например, инструкция `CROSS JOIN` приведет к физическому соединению с вложенным циклом, в результате которого каждая строка внешней таблицы будет соединена с каждой строкой внутренней. Очевидно, SQL Server не сможет использовать поиск по индексу, если предикат соединения не поддерживает SARG, а это чревато крайне неэффективной обработкой больших входных данных.

Соединение слиянием

Соединение слиянием работает с двумя отсортированными входными таблицами. Оно сравнивает две строки и возвращает их соединение, если они равны. Если нет, оно отбрасывает меньшее значение и переходит к следующей строке входной таблицы. Алгоритм соединения показан в листинге 5.9.

Листинг 5.9. Алгоритм соединения слиянием (псевдокод)

```
/* Inputs I1 and I2 are sorted */
get first row R1 from input I1
get first row R2 from input I2
while not end of either input
begin
    if R1 joins with R2
    begin
        return join (R1, R2)
        get next row R2 from I2
    end
    else if R1 < R2
        get next row R1 from I1
    else /* R1 > R2 */
        get next row R2 from I2
end
```

Соединение слиянием оптимизировано для средних и больших данных, когда обе входные таблицы отсортированы. Это значит, что входные данные должны быть проиндексированы по столбцам предиката соединения. Однако на практике SQL Server может заняться сортировкой входных данных уже во время выполнения запроса, и тогда не исключено, что сортировка потребует гораздо больше ресурсов, чем само слияние. Проверьте, так ли это, и учтите ресурсоемкость оператора *Sort* в ходе анализа.

Есть еще одно предостережение. Слияние менее эффективно в сценариях соединения «многие ко многим», когда у обеих входных таблиц есть дубликаты в значениях предиката соединения. В таких случаях SQL Server сохраняет повторяющиеся значения в рабочей таблице в `tempdb`, что может ухудшить производительность, когда дубликатов много. Чтобы определить, выполняется ли соединение слиянием в этом режиме, можно просмотреть свойство `Many to Many` оператора соединения в плане выполнения.

К сожалению, с этим мало что можно поделать. Поскольку столбцы предикатов объединения слиянием обычно индексируются, убедитесь, что индексы определены как уникальные — если данные, на которых построены эти индексы, должны быть уникальными.

Хеш-соединение

Хеш-соединение предназначено для обработки больших объемов несортированных входных данных. Его алгоритм состоит из двух этапов.

На первом этапе, или на этапе *сборки*, хеш-соединение просматривает одну из входных таблиц (обычно меньшую), вычисляет хеш-значения ключа соединения и помещает их в хеш-таблицу. На втором этапе (*зондирование*) алгоритм сканирует вторую входную таблицу и проверяет, есть ли в хеш-таблице хеш-значение ключа соединения из второй таблицы. Если есть, то SQL Server вычисляет предикат соединения для строки из второй таблицы и всех строк из первой таблицы, которые принадлежат к одному и тому же контейнеру кэша. Алгоритм показан в листинге 5.10.

Листинг 5.10. Алгоритм внутреннего хеш-соединения (псевдокод)

```
/* Build Phase */
for each row R1 in input I1
begin
    calculate hash value on R1 join key
    insert hash value to appropriate bucket in hash table
end

/* Probe Phase */
for each row R2 in input I2
begin
    calculate hash value on R2 join key
    for each row R1 in hash table bucket
    if R1 joins with R2
        return join (R1, R2)
end
```

Хеш-соединению требуется память, чтобы хранить хеш-таблицу. При нехватке памяти соединение сохраняет часть контейнеров хеш-таблицы в `tempdb`. Этот эффект называется *переносом* (*spill*) и может сильно повлиять на производительность соединения, потому что доступ к базе данных `tempdb` работает значительно медленнее.

Переносы часто происходят из-за неправильной оценки необходимого объема памяти, что, в свою очередь, может быть вызвано ошибочной оценкой количества элементов. В этом случае убедитесь, что статистика актуальна; если это не помогает, то попробуйте упростить или переработать запрос.

В интеллектуальной обработке запросов (IPQ) в SQL Server 2017 Enterprise Edition появилась *обратная связь по выделению памяти (memory grant feedback)*, которая увеличивает или уменьшает объем памяти, выделяемой для запроса, в зависимости от того, как память использовалась в предыдущих выполнениях. В SQL Server 2017 эта функция доступна только в пакетном режиме, а начиная с SQL Server 2019, она также включена в построчном режиме.

Ознакомьтесь с подробностями в документации Microsoft[1] и подумайте о том, чтобы перейти на уровень совместимости базы данных, который поддерживает обратную связь по выделению памяти. Это может снизить количество переносов в `tempdb`. Я расскажу об этом подробнее в главах 7 и 9.

Сравнение типов соединений

Таблица 5.4 содержит сводную информацию о различных типах соединений и вариантах использования, для которых они оптимизированы.

Таблица 5.4. Сравнение типов соединений

	Соединение в цикле	Соединение слиянием	Хеш-соединение
Лучший случай использования	Одна из таблиц — маленькая, у другой — индексы на столбцах соединения	Средние или крупные таблицы, отсортированные по ключу индекса	Средние или крупные таблицы
Требуется сортировка входных данных	Нет	Да	Нет
Требуется предикат равенства	Нет	Да	Да
Блокирующий оператор	Нет	Нет	Да (на этапе сборки)
Использует память	Нет	Нет	Да
Использует `tempdb`	Нет	Нет (при сортировке возможен перенос в tempdb)	Да, в случае переноса
Сохраняет порядок	Да (внешняя таблица)	Да	Нет

В средствах интеллектуальной обработки запросов в SQL Server 2017 появилась концепция *адаптивного соединения (adaptive join)*. При таком соединении SQL Server во время выполнения выбирает, использовать цикл или хеш-соединение,

[1] https://oreil.ly/qyyNx

в зависимости от размера входных данных. К сожалению, в SQL Server 2017 и 2019 это работает только в пакетном режиме выполнения, который в большинстве случаев активируется индексами columnstore. Чтобы в SSMS адаптивное соединение отображалось в плане выполнения, нужно включить динамическую статистику запросов.

Я только что упоминал, что каждый тип соединения оптимизирован для конкретных случаев и может плохо работать в других случаях. Давайте сравним производительность разных типов соединения на простом примере. В листинге 5.11 мы создадим еще одну таблицу (аналогичную таблице из листинга 5.6) и заполним ее теми же данными. В обеих таблицах по два столбца, для одного из которых определен кластеризованный индекс.

Листинг 5.11. Эффективность соединения: создание таблицы

```
CREATE TABLE dbo.T2
(
    IndexedCol INT NOT NULL,
    NonIndexedCol INT NOT NULL
);
CREATE UNIQUE CLUSTERED INDEX IDX_T2
ON dbo.T2(IndexedCol);
INSERT INTO dbo.T2(IndexedCol, NonIndexedCol)
    SELECT IndexedCol, NonIndexedCol FROM dbo.T1;
```

Затем сравним производительность разных типов соединения с помощью кода из листинга 5.12. Здесь я принудительно применяю конкретные типы соединений с помощью указаний соединения (подробнее об этом позже). Время выполнения инструкций в моей тестовой среде указано в комментариях к коду.

Листинг. 5.12. Эффективность производительности: тестовые примеры

```
-- Соединение в цикле с поиском по индексу во внутренней таблице
-- Время выполнения: 137 мс
SELECT COUNT(*)
FROM dbo.T1 INNER LOOP JOIN dbo.T2 ON
    T1.IndexedCol = T2.IndexedCol
WHERE
    T1.NonIndexedCol <= 100;

-- Соединение в цикле с неэффективным просмотром индекса во внутренней таблице
-- Время выполнения: 16 732 мс
SELECT COUNT(*)
FROM dbo.T1 INNER LOOP JOIN dbo.T2 ON
    T1.IndexedCol = T2.NonIndexedCol
WHERE
    T1.NonIndexedCol <= 100;

-- Хеш-соединение. Медленнее, чем соединение в цикле, на небольших входных данных
-- Время выполнения: 411 мс
```

```
SELECT COUNT(*)
FROM dbo.T1 INNER HASH JOIN dbo.T2 ON
    T1.IndexedCol = T2.IndexedCol
WHERE
    T1.NonIndexedCol <= 100;

-- Соединение в цикле с поиском по индексу
-- во внутренней таблице с большим объемом входных данных
-- Время выполнения: 1514 мс
SELECT COUNT(*)
FROM dbo.T1 INNER LOOP JOIN dbo.T2 ON
    T1.IndexedCol = T2.IndexedCol;

-- Хеш-соединение по индексированным столбцам
-- Быстрее, чем соединение в цикле, при больших объемах входных данных
-- Время выполнения: 1215 мс
SELECT COUNT(*)
FROM dbo.T1 INNER HASH JOIN dbo.T2 ON
    T1.IndexedCol = T2.IndexedCol;

-- Хеш-соединение по неиндексированным столбцам
-- Производительность не зависит от того, индексированы ли столбцы
-- Время выполнения: 1235 мс
SELECT COUNT(*)
FROM dbo.T1 INNER HASH JOIN dbo.T2 ON
    T1.IndexedCol = T2.NonIndexedCol;

-- Соединение слиянием с предварительно отсортированными входными данными
-- Время выполнения: 440 мс
SELECT COUNT(*)
FROM dbo.T1 INNER MERGE JOIN dbo.T2 ON
    T1.IndexedCol = T2.IndexedCol;

-- Соединение слиянием без предварительной сортировки входных данных
-- Время выполнения: 774 мс
SELECT COUNT(*)
FROM dbo.T1 INNER MERGE JOIN dbo.T2 ON
    T1.IndexedCol = T2.NonIndexedCol;
```

На небольших входных данных соединение в цикле работает быстрее, чем хеш, однако хеш-соединение становится эффективнее по мере роста объема входных данных. Вместе с тем соединение слиянием отлично подходит для ситуаций, когда входные данные отсортированы. В противном случае оно добавляет в план выполнения оператор *Sort*. Это годится для небольших входных данных, но на очень объемных данных такая сортировка работает плохо.

Эти примеры показывают, что если неправильно выбрать тип соединения, то производительность запросов может резко снизиться. Виной тому в большинстве случаев неправильная оценка количества элементов, особенно когда SQL Server существенно недооценивает размер входных данных.

Из-за этой недооценки хеш-соединение и соединение слиянием могут вызвать перенос в `tempdb`, что снизит производительность соединения. Однако эта ситуация наиболее опасна при соединении в цикле, особенно когда обработка внутренней таблицы требует больших ресурсов. (Вспомните, что SQL Server обрабатывает внутреннюю входную таблицу для каждой строки из внешней таблицы, поэтому издержки быстро увеличиваются с каждой итерацией.)

Эту неприятность можно обнаружить, если сравнить фактическое и оценочное количество строк во внешней таблице или фактическое и оценочное количество выполнений первого оператора внутренней входной таблицы (рис. 5.17). Большое расхождение будет признаком неправильной оценки количества элементов. Когда эта неправильная оценка приводит к большому числу выполнений во внутренней таблице, соединение в цикле не годится.

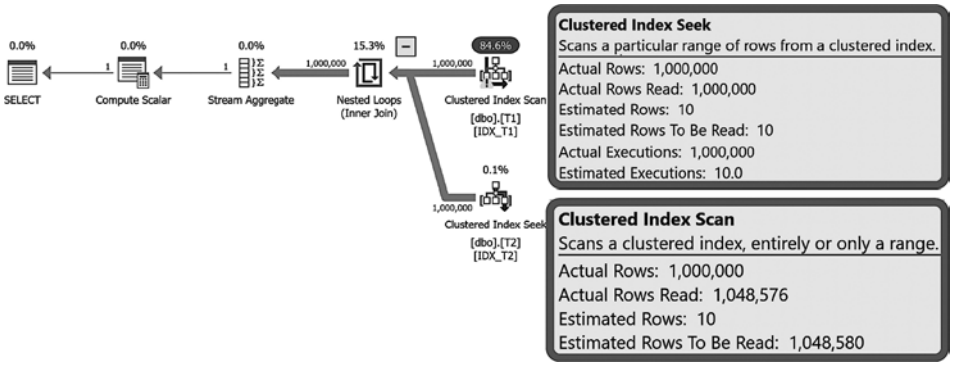

Рис. 5.17. Ошибка оценки количества элементов при соединении в цикле

Решить эту проблему можно по-разному. Для начала посмотрите на запрос и поищите возможности для рефакторинга. Избавьтесь от конструкций, которые могут повлиять на оценку количества элементов, например функции с несколькими инструкциями и табличные переменные. Посмотрите, можно ли переработать индексацию, чтобы улучшить план выполнения.

Стоит проверить, влияет ли на работу сканирование параметров в планах, чувствительных к параметрам. SQL Server иногда кэширует и повторно использует планы выполнения, скомпилированные с нетипичными значениями параметров, особенно если данные распределены в таблице очень неравномерно. Представьте многопользовательские системы, в которых у одних пользователей может быть очень мало данных, а у других — много. Планы выполнения, созданные для одной категории пользователей, будут неэффективны для другой.

В этом случае попробуйте перекомпиляцию на уровне инструкций с параметром `OPTION (RECOMPILE)` или отключите сканирование параметров в базе данных. Эти и другие варианты подробнее рассматриваются в следующей главе.

Еще можно попробовать обновить статистику в таблицах из запроса. К сожалению, это не всегда помогает. Когда оптимизатор запросов в SQL Server оценивает количество элементов в сложных запросах с несколькими соединениями, он не всегда делает верные предположения.

Я уже упоминал, что здесь может помочь упрощение запросов. Попробуйте разделить запрос и сохранить промежуточные результаты в грамотно проиндексированных временных таблицах. SQL Server увидит, как распределены данные, и точно оценит количество элементов в запросах. Очевидно, следует помнить о накладных расходах, которые связаны с временными таблицами, и не использовать их для кэширования очень больших наборов данных.

В крайнем случае можно принудительно задавать типы соединений с помощью указаний запроса. Это опасный метод, и его следует применять очень осторожно. Указания заставят оптимизатор запросов выполнять оптимизацию конкретным методом, а он может оказаться неэффективным в будущем, если изменится распределение данных. Когда вы используете указания запроса, не забывайте про них и периодически проверяйте, остаются ли они актуальными.

Указания соединения можно применять двумя способами. Первый — указать список разрешенных типов соединений в параметрах запроса. Первая инструкция в листинге 5.13 показывает, как с помощью такого списка заставить оптимизатор не использовать соединения в цикле ни в одной точке запроса. Второй вариант — указать конкретный тип соединения таблиц. Вторая инструкция в этом же примере заставляет SQL Server использовать хеш-соединение для таблиц `A` и `B`.

Листинг 5.13. Принудительные типы соединений

```
SELECT A.Col1, B.Col2
FROM
    A JOIN B ON A.ID = B.ID
    JOIN C ON B.CID = C.ID
OPTION (MERGE JOIN, HASH JOIN);

SELECT A.Col1, B.Col2
FROM
    A INNER HASH JOIN B ON A.ID = B.ID
    JOIN C ON B.CID = C.ID;
```

К сожалению, второй подход также задает порядок соединения для *всех* соединений в запросе. SQL Server будет соединять таблицы именно в том порядке, в котором они перечислены в запросе, не пытаясь поменять их местами. Например, в листинге 5.13 таблица `A` всегда будет сначала соединяться с таблицей `B` с помощью хеш-соединения, а результат их соединения будет соединяться с таблицей `C`. Таким образом, SQL Server лишится возможности оптимизировать порядок для запросов с несколькими соединениями. Будьте осторожны с этим подходом!