ГЛАВА 12

Процессор запросов

Задачи обработки запросов и повышения производительности являются настолько важными для оптимальной эксплуатации программы SQL Server, что автор включает главу, посвященную каждой из этих тем, во все свои книги, посвященные программе SQL Server. В данной книге приведено обновленное изложение указанного материала из последней книги автора, *The Guru's Guide to SQL Server Stored Procedures, XML, and HTML*, и продолжено описание основных принципов оптимизации запросов, начатое в указанной книге. Безусловно, предоставление читателю целого ряда отдельных рекомендаций по повышению производительности позволило бы получить определенный кратковременный выигрыш, но, по мнению автора, понимание принципов, на которых основаны подобные рекомендации, намного важнее и в долговременном плане позволит читателю добиться гораздо более существенных успехов. Как было указано во "Введении", понимание проекта, лежащего в основе определенной технологии, гораздо важнее, чем простое изучение способов применения этой технологии. Способность успешно настраивать запросы на языке Transact-SQL является производной от знания и понимания самой программы SQL Server — как работает эта программа, какие действия выполняет при обработке запроса, какие ресурсы требует для формулировки эффективного плана и т.д. Поэтому в настоящей главе изложение практических сведений, необходимых для пользователя, сопровождается обсуждением некоторых малозаметных особенностей обработки запросов в программе SQL Server. В этой главе описаны различные этапы обработки запросов и указано, что происходит на каждом из них. Получив более полное представление о том, как действуют средства обработки запросов программы SQL Server, читатель сможет разрабатывать собственные приемы ускоренного выполнения кода T-SQL и применять при этом обоснованные и безопасные методы благодаря лучшему пониманию того, на каком проекте основано функционирование программы SQL Server.

Основные термины и определения

- **Предикат.** Выражение, которое принимает значение `true` (истина), `false` (ложь) или `unknown` (неизвестное значение).
- **Кардинальность.** Строго говоря, кардинальностью называется количество уникальных значений в таблице. Программа SQL Server допускает наличие в таблицах дублирующихся строк, а в индексах — дублирующихся значений ключа, поэтому при обсуждении программного продукта SQL Server кардинальность часто определяют в более общих терминах — как количество строк в таблице или количество строк, возвращенных оператором плана

запроса. Понятие кардинальности можно также определить как количество уникальных значений в индексе.

- **Плотность.** Плотность определяют как уникальность значений в наборе данных. Плотность индекса вычисляется путем деления количества строк, которые соответствуют данному конкретному значению ключа, на количество строк в таблице.
- **Избирательность.** Мера оценки количества строк, которые должны быть возвращены с помощью данного конкретного предиката. Этот показатель выражается в виде процентного соотношения, определяемого количеством строк в таблице.

Синтаксический анализ

Для того чтобы запрос на языке Transact-SQL можно было оптимизировать с помощью оптимизатора запросов SQL Server, запрос необходимо прежде всего представить в виде дерева реляционных операторов с помощью синтаксического анализа. Дерево реляционных операторов состоит из логических операторов, необходимых для выполнения работы, затребованной в запросе. Например, простой запрос, в котором осуществляется соединение таблиц `Customers` и `Orders` базы данных `Northwind`, может быть преобразован с помощью синтаксического анализа в дерево реляционных операторов, которое включает логический оператор `Inner Join` (внутреннее соединение), находящийся между двумя входными таблицами.

В программе SQL Server предпринимаются попытки избежать излишних действий по синтаксическому анализу и оптимизации запросов путем хэширования текста каждого вновь поступившего запроса и кэширования его в памяти, наряду с первоначальным текстом запроса и ссылкой на контекст выполнения и план выполнения, которые были получены после обработки первоначального запроса. Сервер сверяет хэшированный текст запроса каждого вновь поступившего запроса с тем текстом, который уже находится в памяти. Поскольку существует вероятность коллизий в хэше, сервер, обнаружив совпадение, сравнивает первоначальный текст нового запроса с первоначальным текстом запроса, находящегося в памяти, для определения того, имеется ли между ними точное совпадение. Если эти тексты действительно совпадают, то новый запрос не требует повторного синтаксического анализа и повторной оптимизации; для него можно использовать ранее созданный контекст выполнения и/или план выполнения.

Средства программы SQL Server, которые обеспечивают синтаксический анализ поступивших запросов с помощью языковых событий и вызовов RPC с последующим преобразованием в деревья реляционных операторов, называются средствами языковой обработки и выполнения (Language Processing and Execution — LPE). Средства LPE обеспечивают передачу в оптимизатор дерева реляционных операторов, сформированного в результате синтаксического анализа, а затем получение оптимизированного плана выполнения, выработанного оптимизатором на этапе преобразования, и выполнение запроса. Средства LPE не представлены в сервере в виде одного компонента или фрагмента кода, а фактически состоят из многочисленных модулей и привлекают к своей работе целый ряд других средств сервера.

Этапы оптимизации

Процесс оптимизации запроса представлен в общих чертах на рис. 12.1. После получения дерева реляционных операторов, в которое был преобразован первоначальный запрос в результате синтаксического анализа, оптимизатор приступает к его оптимизации — поиску способов выполнения требуемой работы с помощью меньшего объема издержек и модификации запроса в случае необходимости для достижения этой цели. Вообще говоря, в каждой последующей фазе или на каждом этапе осуществления процедуры оптимизации предусмотрена возможность рассматривать все более сложные (и часто более дорогостоящие) варианты преобразования исходного дерева реляционных операторов, сформированного синтаксическим анализатором, в оптимизированный план выполнения. Если в ходе выполнения этой процедуры удается найти подходящий эффективный план, этот план может быть возвращен без необходимости дальнейшей оптимизации или оценки потенциальных преобразований. Процедура оптимизации состоит из четырех отдельных этапов: тривиальная оптимизация плана, упрощение, полная оптимизация и преобразование. Ниже каждый из этих этапов рассматривается отдельно.

Тривиальная оптимизация плана

Применительно к некоторым запросам наиболее эффективный план выполнения можно выявить на основе самого запроса, поэтому какая-либо оценка издержек или проведение сравнений между планами не требуется. Например, если в запросе используется предикат равенства для выборки данных с помощью запроса из столбца с уникальным индексом, то нет необходимости оценивать стоимости различных планов и проводить сравнение этих стоимостей; очевидно, что наилучший вариант состоит в использовании поиска по уникальному индексу. Кроме того, в оптимизаторе предусмотрены тривиальные варианты оптимизации для некоторых типов запросов, отвечающих определенным критериям; соединений таблиц с условиями ограничения; операторов DML и некоторых других операций. (Но следует отметить, что тривиальный план не может применяться для обработки запроса, содержащего подзапрос.) Если оптимизатор обнаруживает, что наименее дорогостоящим планом, который можно надеяться получить, является тривиальный план, то оптимизатор передает этот план на этап преобразования для подготовки к выполнению. Для контроля над тем, был ли выбран тривиальный план, можно разрешить флажок трассировки 8759. Если этот флажок разрешен, то оптимизатор записывает первую часть запроса, для которого создан тривиальный план, в журнал регистрации ошибок.

Ниже показан пример запроса, который приводит к созданию тривиального плана.

```
DECLARE @ordno int
SET @ordno=10248
SELECT * FROM Orders WHERE OrderId=@ordno
```

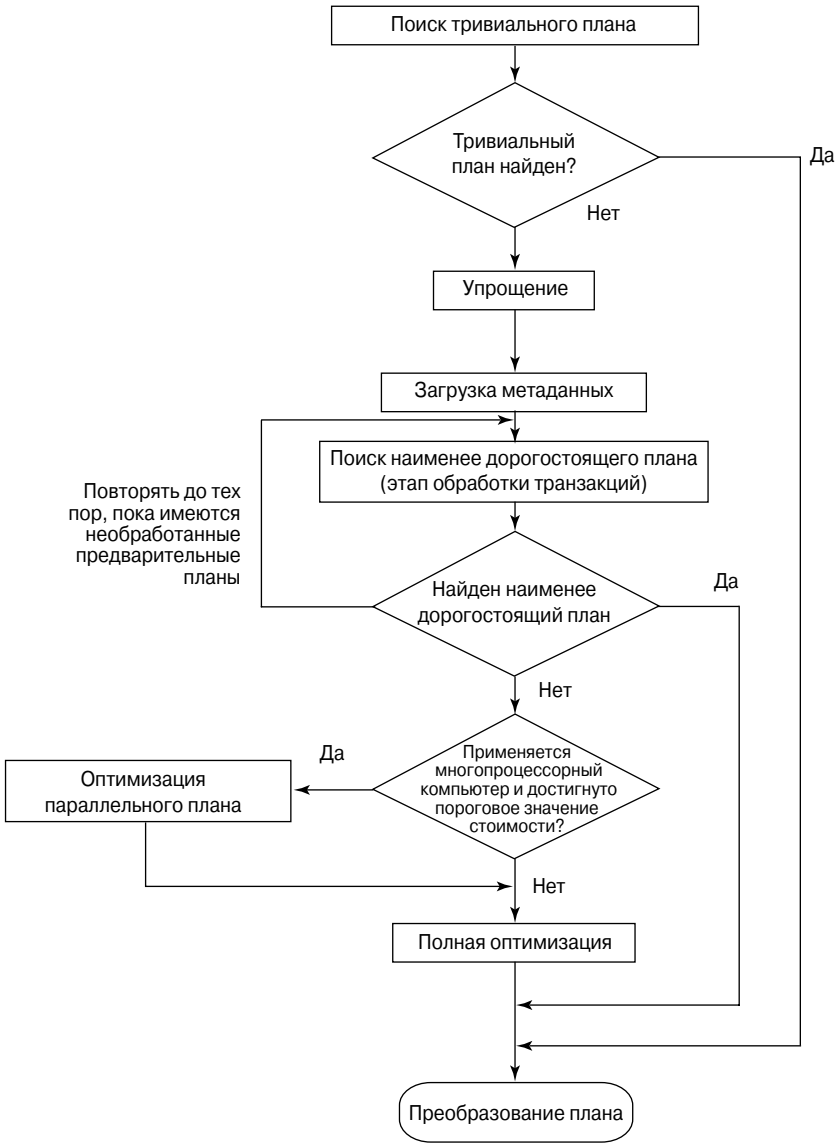

Рис. 12.1. Процедура оптимизации запроса

В этом запросе единственным используемым предикатом является предикат сравнения на равенство, а на столбце `Orders.OrderNo` имеется уникальный индекс, поэтому оптимизатор выбирает тривиальный план, в котором для обслуживания этого запроса предусмотрен поиск по индексу. Заслуживает внимания тот факт, что использование в данном запросе переменной в качестве условия запроса не исключает возможности для оптимизатора выбрать тривиальный план. Независимо от того, какое значение принимает эта переменная на этапе прогона, для сравнения

значения переменной с ключом столбца, на котором определен уникальный индекс, применяется простой оператор сравнения на равенство, поэтому оптимизатор может сразу же определить, что наилучшим вариантом является тривиальный план.

Этапы тривиальной оптимизации и полной оптимизации представляют собой единственные два этапа оптимизации, на которых может быть сформирован план, после чего процедура оптимизации может завершиться. Если какой-то запрос не удается оптимизировать на этапе тривиальной оптимизации, то этот запрос передается на этап упрощения, а затем на этап полной оптимизации.

Упрощение

На этапе упрощения применяются некоторые эвристические алгоритмы, которые позволяют использовать в дальнейшем более простые и эффективные методы оптимизации. Как указывает само название этого этапа, он предназначен для упрощения выражения запроса и обеспечения лучшего применения индекса и статистических данных во время полной оптимизации. При этом операторы могут перемещаться в дереве, заменяться другими операторами и в целом упрощаться, чтобы осуществление полной оптимизации проходило более просто, быстро и эффективно. Например, подзапросы, используемые в предикате, могут быть заменены полусоединениями и внутренними соединениями; критерии выборки, которые применяются только к одной таблице в многотабличном запросе, могут быть передвинуты в дереве таким образом, чтобы они применялись перед выполнением операций соединения; могут быть распознаны соотношения, которые имеют вид звездообразной схемы; некоторые операторы могут быть заменены более эффективными эквивалентами и т.д. Ниже приведен пример запроса, содержащего подзапрос, который преобразуется оптимизатором в полусоединение на этапе упрощения.

```
SELECT a.au_id
FROM authors a
WHERE au_id IN (SELECT au_id FROM titleauthor ta WHERE
    ta.au_id=a.au_id)
```

При просмотре графического изображения плана этого запроса в программе Query Analyzer можно обнаружить, что оптимизатор вырабатывает план, в котором используется оператор `Nested Loops/Left Semi Join` (соединение с помощью вложенных циклов и/или левое полусоединение). Этот запрос представлен в коде T-SQL с помощью подзапроса, но в нем фактически выражено требование на получение строк из таблицы `authors`, для которых имеются соответствия в таблице `titleauthor`. Таким образом, вместо выполнения подзапроса применительно к каждой строке в таблице `authors`, оптимизатор находит достаточно эффективное решение, преобразуя этот запрос в простое полусоединение двух таблиц.

Полная оптимизация

Если оптимизатор не способен найти приемлемый тривиальный план, то существующий план упрощается, а затем передается на этап полной оптимизации. На этапе полной оптимизации предпринимаются попытки оценить стоимость

различных альтернативных способов получения данных, затребованных в запросе, и определить тот способ, который является наименее дорогостоящим. Эти действия не продолжаются до бесконечности, поскольку время, в течение которого оптимизатор перебирает различные способы обработки запроса, прежде чем выбрать наиболее подходящий, строго регламентировано с помощью тайм-аута. Если оптимизатор не находит наиболее эффективный план выполнения ко времени истечения установки тайм-аута, то выбирает наименее дорогостоящий план из тех, которые были найдены к данному моменту, и передает этот план на этап преобразования для последующей передачи на выполнение.

Этап полной оптимизации состоит из четырех меньших этапов (которые можно даже назвать шагами); оптимизатор проходит эти шаги на этапе полной оптимизации запроса. Таковыми являются шаги обработки транзакции, быстрого составления плана, параллельной оптимизации и полной оптимизации. Необходимо учитывать, что оптимизатор может фактически не проходить все эти шаги. Если оптимизатор находит приемлемый план, характеризующийся низкими издержками (план, который является менее дорогостоящим по сравнению с заранее заданным пороговым значением издержек), то передает этот план на этап преобразования для его дальнейшего выполнения, не проходя все оставшиеся шаги. План выполнения может быть выработан и передан на этап преобразования в любой момент, кроме шага параллельной оптимизации. Кроме того, если компьютер, на котором работает программа SQL Server, не является многопроцессорным, то нет необходимости выполнять шаг параллельной оптимизации. Каждый из указанных шагов подробно описан ниже.

Обработка транзакции

В шаге обработки транзакции оптимизатор применяет подмножество потенциальных преобразований, имеющихся в его распоряжении во время полной оптимизации. Назначение этого шага состоит в поиске с максимально возможным быстродействием большинства планов запросов, ориентированных на обработку транзакций, которые не были найдены на этапе создания тривиального плана. По большей части в этом шаге разрешены только преобразования, в которых предусматривается использование соединения с поиском по индексу. Если же рассматриваемый оператор не поддерживает поиск по индексу, то могут быть разрешены преобразования, в которых применяется хэширование.

В шаге обработки транзакции не разрешается применять более сложные преобразования, поскольку такие преобразования откладываются до полной оптимизации. При обработке транзакции не разрешены и не оцениваются по стоимости преобразования, связанные с переупорядочением соединений, переформатированием (динамическим созданием индексов) и т.д. Как уже было сказано, назначение этого шага состоит в том, чтобы быстро провести проверку более широкого набора планов по сравнению с тем набором планов, который рассматривался на этапе получения тривиального плана. Если обнаруживается план, оцениваемая стоимость которого (в единицах измерения времени) меньше чем примерно две десятых секунды, то выбирается этот план и дальнейшая оптимизация не происходит.

Быстрое составление плана

В шаге быстрого составления плана разрешается использовать все правила преобразования, поддерживаемые оптимизатором. При этом допускается выполнение преобразований, предусматривающих переупорядочение соединений (в которых участвуют нижние четыре таблицы дерева реляционных операторов), и выбирается первый план с такой оценкой стоимости, которая не превышает примерно одной секунды.

Параллельная оптимизация

В шаге параллельной оптимизации оптимизатор оценивает операторы и преобразования, позволяющие использовать возможности многопроцессорной обработки применяемого компьютера, если наименее дорогостоящий план, найденный к этому моменту, превышает значение параметра `cost threshold for parallelism` (пороговое значение стоимости при параллельной обработке) процедуры `sp_configure`, а программа SQL Server эксплуатируется на многопроцессорном компьютере). Рассматриваемые на этом этапе операторы будут исследованы более подробно во время полной оптимизации.

Полная оптимизация

Если все другие попытки найти приемлемый план выполнения с низкой стоимостью оканчиваются неудачей, то оптимизатор переходит к выполнению полной оптимизации. Во время выполнения этого шага оптимизатор рекурсивно проходит по дереву реляционных операторов, имеющему такой вид, который это дерево приобрело после прохождения через все другие этапы и шаги оптимизации. Дерево реляционных операторов состоит из реляционных узлов `AND/OR` и реляционных операторов. Узлы `OR` представляют собой множество взаимоисключающих операторов (например, операторов обработки с помощью вложенных циклов, либо операторов хэшированного соединения, либо операторов соединения слиянием), относящихся к конкретному шагу плана. Оптимизатор сравнивает взаимоисключающие операторы друг с другом и выбирает наименее дорогостоящий. Узлы `AND` представляют операторы, в которых оптимизации требует каждый дочерний узел. Оптимизатор проходит по дочерним узлам и оценивает эти узлы отдельно. Стоимость узла `OR` равна стоимости наименее дорогостоящего дочернего узла, а стоимость узла `AND` равна сумме стоимостей дочерних узлов плюс стоимость некоторого оператора.

Рекурсивный алгоритм, выполняемый на этом этапе, по сути, остается аналогичным тому алгоритму, который выполнялся на предыдущих этапах; его отличительная особенность состоит лишь в том, что рассматривается более широкий набор потенциальных преобразований. Например, если предположить, что шаг полной оптимизации выполняется после шага параллельной оптимизации, то в текущий момент может осуществляться оценка стоимости операторов параллельной обработки. Может проводиться сравнение друг с другом различных типов соединений, а также осуществляться оценка стоимости и сопоставление стратегий переформатирования (динамического создания индексов) и других усовершенствованных стратегий оптимизации.

Преобразование

Независимо от того, на каком этапе был фактически выработан план выполнения, этот план должен вначале пройти преобразование для последующей передачи средствам LPE на выполнение. Такое преобразование осуществляется на этапе преобразования процедуры оптимизации. Каждый оптимизированный план запроса перед передачей на выполнение должен быть обработан на данном этапе.

Пределы оптимизации

В оптимизаторе используется несколько методов обеспечения того, чтобы процедура сравнения стоимостей операторов и планов не продолжалась бесконечно или даже невыносимо долго. В какой-то точке процедуры оптимизации оптимизатор может начать приближаться к такому состоянию, что достигнутое им последовательное уменьшение стоимости будет все меньше и меньше. В конечном итоге, если время, затрачиваемое в процессе оценки, намного превышает время выполнения даже самых дорогостоящих планов, то процедура оптимизации вообще не позволит сэкономить такое уж значительное время обработки. В крайнем случае оказывается, что оптимизация и последующее выполнение становится более дорогостоящим, чем выполнение без какой-либо оптимизации. Оптимизатор спроектирован так, чтобы подобные ситуации предотвращались в максимально возможной степени, и обеспечивалась гарантия того, что затраты на оптимизацию оправдывают достигнутое повышение производительности.

В настоящем разделе рассматривается несколько методов ограничения продолжительности работы оптимизатора. Первый из этих методов основан на принципе целевой стоимости. На каждом последующем этапе процедуры оптимизации определяется целевая стоимость для рассматриваемых планов. Планы, стоимость которых превышает заданное целевое значение, больше не рассматриваются, что позволяет оптимизатору быстро сузить выполняемый им поиск и остановиться на наилучших потенциальных планах. На этапе формирования тривиального плана целевая стоимость является бесконечной, а это означает, что не отвергается ни один существующий план. А для последующих этапов оптимизатор уменьшает это значение до 90% стоимости наименее дорогостоящего плана, который был сформирован до этого времени.

Кроме того, в оптимизаторе реализован принцип использования значения тайм-аута. Значение тайм-аута определяется с учетом заданного количества преобразований, которое, согласно прогнозу оптимизатора, может быть выполнено до того, как нужно будет завершить процедуру оптимизации по тайм-ауту. На протяжении выполнения всей процедуры оптимизации оптимизатор проверяет, не было ли превышено заданное значение количества преобразований, и в случае положительного ответа прекращает работу по тайм-ауту. Поскольку количество преобразований, используемое для вычисления тайм-аута, определяется на основе предположения о том, что оптимизатор может выполнить определенное количество преобразований в секунду, само значение тайм-аута лишь приблизительно коррелирует с фактически

израсходованным временем. Связь между значением тайм-аута и фактически израсходованным временем определяется в виде полулинейной функции, которая начинается примерно с 10%-й оценки начальной стоимости и возрастает более или менее линейно, пока наконец не выравнивается после достижения значения израсходованного времени, равного приблизительно 60 секундам. Следует отметить, что точность определения значения тайм-аута зависит от точности оценки значения времени преобразования, поэтому не применяется строго регламентированное понятие значения тайм-аута, определяемого с учетом времени преобразования. Дело в том, что количество преобразований, которые могут быть выполнены оптимизатором в секунду, может изменяться в широких пределах, и в связи с этим точное количество времени, израсходованного до того, как произойдет истечение тайм-аута, выделенного на обработку запроса, может зависеть от самого запроса.

Чтобы определить, когда произошло прекращение работы оптимизатора из-за истечения тайм-аута во время формирования плана запроса, можно использовать флажок трассировки 8675. Кроме того, этот флажок позволяет узнать, в какой момент оптимизатор достиг предела объема памяти, применяемого в программе SQL Server для ограничения объема памяти, используемого оптимизатором (примерно 80% от объема области BPool). К тому же, часто можно установить, когда был достигнут тайм-аут оптимизации, по значению продолжительности компиляции, возвращаемому командой `SET STATISTICS TIME ON`. Если это значение составляет приблизительно 60 секунд, то действительно может обнаруживаться тайм-аут оптимизации, особенно в тех ситуациях, когда безусловно ясно, что оптимизатор потерпел неудачу, пытаясь найти наилучший план для данного конкретного запроса.

Перехват параметров

Прежде чем откомпилировать план выполнения для хранимой процедуры, в программе SQL Server предпринимается попытка “перехватить” (т.е. распознать) значения параметров, передаваемых в эту программу, и использовать такие значения при компиляции плана. Если значения параметров применяются для сужения объема выборки в запросе (т.е. входят в состав предиката `WHERE` или `HAVING`), то перехват параметров позволяет оптимизатору вырабатывать более точный план выполнения, в котором учитываются значения, передаваемые в хранимую процедуру (с использованием статистических гистограмм для столбцов, указанных в критериях выборки), а не исходить исключительно из показателя средней плотности значений в соответствующих столбцах таблицы. Вообще говоря, перехват параметров — очень удобное средство, применение которого приводит к повышению производительности последних версий программы SQL Server по сравнению с более старыми версиями.

Тем не менее, если при первой компиляции плана для некоторой процедуры происходит передача нетипичного значения параметра, то может возникнуть нарушение в работе. При этом в памяти будет кэширован план, который окажется неоптимальным применительно к большинству значений, передаваемых в качестве значений этого параметра. После передачи более типичных значений для них

будет использоваться план, выработанный для нетипичного значения, который потребует более продолжительного времени для выполнения по сравнению с планом, подготовленным для тех значений, которые чаще встречаются на практике.

В такой ситуации можно применить один из нескольких вариантов. Прежде всего рассматриваемая процедура может быть отмечена как предназначенная для автоматической перекомпиляции с помощью опции `WITH RECOMPILE`. Применение этого варианта влечет за собой то, что план процедуры будет перестраиваться при каждом выполнении данной процедуры. Если время перекомпиляции пренебрежимо мало, то указанное решение позволит легко найти выход из той ситуации, в которой значения параметра изменяются в широких пределах, соответствующих распределению этих значений в столбце таблицы.

Кроме того, можно просто выполнить процедуру с использованием опции `WITH RECOMPILE`; такой способ вызова процедуры также приводит к перестройке плана. Аналогично можно воспользоваться процедурой `sp_recompile`, чтобы вынудить программу SQL Server перестроить план процедуры при следующем вызове процедуры на выполнение.

Еще один вариант состоит в том, чтобы "отменить" перехват параметров, применив в качестве критериев выборки в запросе локальные переменные, в которые скопированы значения параметров. Использование в качестве критериев выборки в запросе локальных переменных вместо параметров процедуры, вообще говоря, не рекомендуется, поскольку такой подход исключает возможность применить при вычислении избирательности статистические гистограммы, относящиеся к индексу, но из этого правила есть исключения. Если оптимизатор не может использовать статистические гистограммы для вычисления количества строк, которые могут быть возвращены запросом при применении какого-то конкретного критерия выборки, то оптимизатор использует "магические" числа — заранее заданные оценки процентного соотношения строк, которые будут возвращены в зависимости от применяемого оператора сравнения. В некоторых редких случаях указанные оценки могут оказаться более точными по сравнению с полученными с помощью самой гистограммы. Одним из таких случаев является применение нетипичного значения для просмотра гистограммы, что приводит к получению искаженной оценки количества строк, которые обычно соответствуют переданному параметру.

Иногда можно также реорганизовать запрос, выполнение которого нарушается из-за неправильного перехвата параметров, таким образом, чтобы этот запрос осуществлялся в другом контексте выполнения, связанном с собственным планом выполнения. Этот подход основан на том, что при динамическом выполнении в процедуре блока T-SQL или при вызове другой процедуры, каждый из подобных вызываемых модулей получает свой собственный план выполнения. В методах, основанных на использовании этого подхода для устранения недостатков перехвата параметров, предусматривается применение процедуры `sp_executesql` или функции `EXEC()`, а также разбиение одной процедуры на несколько процедур. Подход, основанный на использовании процедуры `sp_executesql` и разбиении процедуры на несколько процедур, все еще допускает повторное применение плана. А в подходе, основанном на использовании функции `EXEC()`, такая возможность, скорее всего, будет отсутствовать, поскольку рассматриваемый

план, вероятно, придется компилировать повторно при каждом выполнении процедуры. Такой подход может оказаться приемлемым или неприемлемым в зависимости от того, какие действия выполняются рассматриваемой процедурой и сколько времени затрачивается на ее компиляцию.

К тому же, можно явно очистить кэш процедуры с помощью команды DBCC FREEPROCCACHE. Выполнение этой команды вызывает удаление из памяти всех откомпилированных планов, а в связи с этим возникает необходимость осуществить повторную компиляцию планов при очередном вызове на выполнение каждой процедуры. Это — довольно радикальная мера, но может применяться в некоторых обстоятельствах. Например, очистку кэша можно производить до и после вызова на выполнение в ночную смену некоторого задания, связанного с применением большого количества процедур с нетипичными значениями параметров, если в дневную смену те же процедуры обычно вызываются с более типичными параметрами. Это позволяет добиться того, чтобы в ночную смену не использовались планы, выработанные в дневную смену, а планы, созданные для заданий, выполняемых в ночную смену, не применялись повторно на следующий день.

Еще одна ситуация, в которой при использовании перехвата параметров могут возникнуть нарушения в работе, состоит в том, что план выполнения, хранящийся в кэше, отражает переданные в процедуру типичные значения параметров, но возникает необходимость передать в процедуру нетипичное значение одного из параметров и выполнить эту процедуру как можно быстрее. В таком случае проблема состоит не в том, что в кэше находится неоптимальный план, поскольку для большинства запросов план действительно оптимален. Причиной проблемы является то, что для указанных двух запросов не следует совместно использовать один и тот же план выполнения. Например, допустим, что эксплуатируется хранимая процедура, в которую передается код страны для получения данных о сбыте в этой стране. Предположим также, что чаще всего передается код страны “US”, поскольку рассматриваемое предприятие находится в Соединенных Штатах и проводит основную часть торговых операций именно в этой стране. В связи с тем фактом, что записи с данными о сбыте в США составляют основную часть записей в таблице с данными о сбыте, оптимизатор вырабатывает план выполнения, в котором используется полный просмотр этой таблицы. В указанной ситуации полный просмотр таблицы эффективнее, чем поиск по индексу, поскольку в любой операции выборки происходит обработка основной части строк. Но иногда в процедуру передается код другой страны, допустим, такой страны, в которой происходит лишь небольшое количество сделок по сбыту. Пользователь рассчитывает на то, что соответствующий запрос вернет результаты довольно быстро, поскольку в этом запросе в конечном итоге требуется обработать лишь несколько строк, но ускорения работы не происходит. И в данном запросе осуществляется полный просмотр таблицы, поскольку повторно используется план, первоначально откомпилированный при передаче значения “US”. Одним из возможных решений является реорганизация указанной процедуры и создание на ее основе нескольких процедур, допустим, одной процедуры для получения данных о сбыте в США и еще одной — для получения данных о сбыте в других странах. При условии, что количество данных о сбыте в странах, отличных от Соединенных Штатов, относительно

невелико, а сами эти данные распределены достаточно равномерно, выполнение запросов для получения данных о сбыте в этих странах, скорее всего, приведет к использованию поиска по индексу (если существует соответствующий индекс). И наоборот, в запросах на получение данных о сбыте в Соединенных Штатах будет по-прежнему применяться полный просмотр таблицы, поскольку именно в этом состоит наиболее эффективный способ обслуживания таких запросов. Указанный подход, основанный на использовании нескольких процедур в указанной форме, может оказаться наиболее приемлемым решением, особенно если для компиляции плана требуется много времени (и поэтому вариант с автоматической перекомпиляцией при каждом выполнении подходит не очень хорошо).

Автоматическая параметризация

В целях повышения вероятности повторного использования плана программа SQL Server предпринимает попытки автоматической параметризации произвольных запросов. Под "автоматической параметризацией" подразумевается выполняемая сервером замена значения константы в произвольном запросе маркером параметра для того, чтобы откомпилированный для такого запроса план можно было повторно использовать с другими значениями константы. Например, рассмотрим следующий запрос:

```
SELECT * FROM Orders WHERE OrderId=10248
```

В этом запросе значение `10248` является константой. Если бы сервер не был способен автоматически параметризировать этот запрос, то при следующем выполнении запроса со значением, отличным от `10248`, пришлось бы формировать отдельный план выполнения. Вместо этого в сервере применяются достаточно интеллектуальные алгоритмы, позволяющие заменить значение `10248` маркером параметра и передать `10248` в качестве значения параметра для данного конкретного контекста выполнения. В других случаях вызова того же запроса с иными значениями параметра каждый раз будет создаваться отдельный контекст выполнения, но использоваться один и тот же план выполнения.

Пользователь может не полагаться на способность сервера правильно подбирать параметры для конкретного запроса, а задать эти параметры непосредственно с помощью процедуры `sp_executesql` или с использованием маркеров параметров в тексте запроса. Если выбран такой подход, неоднозначность исключается, и пользователь получает больший контроль над тем, какие типы данных применяются для каждого параметра. Программа SQL Server определяет логическим путем тип данных каждого автоматически вводимого параметра с учетом переданного значения константы, но иногда выдвинутое программой предположение является ошибочным, что приводит к несовпадению типов данных между параметром и столбцом, с которым сравнивается этот параметр. Такая ситуация может препятствовать применению индекса, поэтому следует учитывать вероятность подобного развития событий. Например, в приведенном выше запросе программа SQL Server принимает предположение, что параметр с обозначением идентификатора заказа должен иметь тип данных `smallint`, поскольку

константа имеет значение 10248, для которого хорошо подходит двухбайтовый тип данных smallint программы SQL Server. Но если в действительности столбец OrderId таблицы Orders представлен в виде четырехбайтовых целочисленных значений, то в план выполнения придется включить оператор CONVERT, чтобы компенсировать различие между двумя указанными типами данных.

Узнать о том, был ли параметризован автоматически параметризуемый запрос, можно с помощью средств текстового или графического отображения плана выполнения. Если какой-либо произвольный запрос был автоматически параметризован, то в тексте запроса можно обнаружить вместо по меньшей мере некоторых значений констант такие метки-заполнители для параметров, как @1; кроме того, обнаруживается, что такие метки-заполнители используются в запросе в составе критериев выборки.

Для проверки результатов применения автоматической параметризации удобно также использовать таблицу syscacheobjects. Если в программе SQL Server был автоматически параметризован какой-либо произвольный запрос, то автоматически сформированные метки-заполнители для параметров и типы их данных можно найти в начале столбца sql в строках данного запроса таблицы syscacheobjects. Такие метки-заполнители используются также в тексте запроса, приведенном в столбце sql, в качестве критерия выборки данных, возвращаемых запросом, как показано ниже.

```
(@1 smallint)SELECT * FROM [Orders] WHERE [OrderId]=@1
```

Кроме того, для контроля над тем, происходит ли автоматическая параметризация того или иного запроса, нужно также разрешить использование флажка трассировки 8759. Если флажок трассировки 8759 разрешен, то первая часть автоматически параметризованного запроса записывается в журнал регистрации ошибок программы SQL Server следующим образом:

```
SAFE auto-paramd query: (@1 smallint) SELECT * FROM [Orders] WHERE
  [OrderId]=@1
```

Применение индексов

Трудно назвать подход, в большей степени способствующий повышению производительности запросов, чем создание удобных и эффективных индексов. Основные сложности, связанные с эксплуатацией крупных банков данных, обусловлены выполнением операции ввода-вывода, поэтому необходимо стремиться уменьшить объем ввода-вывода и затраты времени на ввод-вывод в максимально возможной степени. Для этого можно применять кэширование, наращивать обрабатывающие мощности и устанавливать быстродействующие жесткие диски. Но ничто не отражается на производительности запросов так существенно, как индексация. Если отсутствует применимый индекс, то для программы SQL Server нет иного способа, чем полностью просматривать одну или несколько таблиц в поиске нужных данных. Если же приходится применять соединение двух или большего количества таблиц, то программе SQL Server может потребоваться просматривать их несколько раз, чтобы найти все данные, необходимые для удовлетворения запроса. Индексы позволяют в значительной степени ускорить процесс поиска данных, а также процесс соединения таблиц друг с другом.

Местонахождение данных об индексах

Информация системного уровня об индексах, применяемых в программе SQL Server, хранится в системной таблице `sysindexes`. В таблице `sysindexes` имеется строка с данными о каждом индексе, а для идентификации самого индекса служит столбец `indid` этой таблицы. В столбце `indid` находятся целочисленные значения с отсчетом от 1, которые показывают, в каком порядке создавались индексы; кластеризованный индекс всегда имеет значение `indid`, равное 1. Если в таблице нет кластеризованного индекса, то в таблице `sysindexes` имеется строка со значением для самой таблицы, в которой указано значение `indid`, равное 0.

Схема распределения индекса

В программе SQL Server экстенты, принадлежащие таблице или индексу, отслеживаются с использованием страниц со схемами распределения индексов (Index Allocation Map — IAM). В таком объекте, как динамическая область памяти или индекс, имеется по меньшей мере по одной странице IAM для каждого файла, в котором для этого объекта распределены экстенты. Схема IAM представляет собой битовую структуру отображения, которая отображает экстенты на объекты; каждый бит этой структуры указывает, принадлежит ли соответствующий экстент тому объекту, в котором находится данная схема IAM. Каждая структура отображения IAM охватывает диапазон из 512 тысяч страниц. Первая страница IAM, относящаяся к индексу, хранится в столбце `First-IAM` таблицы `sysindexes`. Страницы IAM распределены в файле базы данных случайным образом и связаны друг с другом в виде одной цепочки. Но даже несмотря на то, что схемы IAM позволяют программе SQL Server, по существу, заранее выполнять выборку экстентов таблицы, в дальнейшем все равно приходится просматривать отдельные строки экстента, поскольку схема IAM служит просто в качестве метода доступа к самим страницам.

Типы индексов

В программе SQL Server поддерживаются индексы двух типов — кластеризованные и некластеризованные. Оба эти типа имеют много общих особенностей. В частности, оба эти типа состоят из страниц, хранящихся в B-деревьях (B — сокращение от balanced, сбалансированный). На уровнях узлов индексов каждого типа расположены указатели на страницы следующего уровня, а на листовом уровне находятся значения ключей.

B-деревья

Как уже было сказано, физическое хранение индексов в программе SQL Server осуществляется в виде B-деревьев. В частности, B-деревья обеспечивают поиск в данных с использованием определенной разновидности алгоритма двоичного поиска. Кроме того, в индексах на основе B-деревьев ключи хранятся так, что ненамного отличающиеся значения находятся близко друг к другу, а само дерево постоянно подвергается повторному уравновешиванию для обеспечения того, чтобы каждое конкретное значение могло быть достигнуто с помощью минимального количества переходов по страницам индекса. Поскольку B-деревья являются

уравновешенными, стоимость поиска любой строки остается относительно постоянной, независимо от того, где находится эта строка.

Первым узлом в индексе на основе B-дерева является корневой узел. Указатель на корневой узел каждого индекса хранится в столбце `root` таблицы `sysindexes`. Проводя поиск данных с помощью индекса, программа SQL Server начинает с корневого узла, затем проходит через все промежуточные уровни, которые могут находиться в индексе, и в конечном итоге либо находит, либо не находит требуемые данные в листовых узлах индекса, находящихся на самом нижнем уровне. Количество промежуточных уровней зависит от размера таблицы, размера ключа индекса и количества столбцов в ключе. Очевидно следующее — чем больше данных находится в таблице и чем больше места занимает каждый ключ, тем больше страниц должно находиться в индексе.

Страницы индекса, относящиеся к уровню выше листового, называются *страницами узлов*. Каждая строка на странице узлов содержит один или несколько ключей, а также указатель на страницу следующего уровня, которой соответствует строка первого ключа. На этом основана общая структура B-дерева. Программа SQL Server проходит по указанным связям до тех пор, пока не обнаруживает искомые данные или не достигает конца цепочки связей в узле листового уровня. Листовой уровень B-дерева содержит значения ключей, а при использовании некластеризованных индексов — закладки с информацией о соответствующем кластеризованном индексе или динамической области памяти. Значения ключей хранятся последовательно, причем в версии SQL Server 2000 и последующих версиях могут быть отсортированы в возрастающем или убывающем порядке.

В отличие от некластеризованных индексов, листовой узел кластеризованного индекса фактически содержит сами данные. Закладки в индексе отсутствуют, поскольку нет необходимости в их использовании. Если в базе данных имеется кластеризованный индекс, то на уровне листовых узлов этого индекса находятся сами данные.

Страницы данных в таблице хранятся в виде цепочки страниц, т.е. двусвязного списка страниц. Если в базе данных применяется кластеризованный индекс, то порядок строк на каждой странице и порядок страниц в цепочке определяются ключом индекса. Применение ключа кластеризованного индекса приводит к сортировке данных, поэтому важно выбрать этот ключ правильно. При выборе ключа необходимо учитывать некоторые важные условия, в том числе перечисленные ниже.

- Ключ должен иметь минимально возможный размер, поскольку он будет служить в качестве закладки для каждого некластеризованного индекса.
- Ключ необходимо выбирать так, чтобы он хорошо подходил для обычно применяемых запросов `ORDER BY` и `GROUP BY`.
- Ключ должен достаточно хорошо подходить для обычных запросов поиска в диапазоне (запросов, в которых происходит обращение к диапазону строк с учетом значений в столбце или столбцах).
- Ключ должен охватывать множество таких столбцов, данные в которых не обновляются слишком часто, поскольку обновление ключа кластеризованного индекса таблицы может потребовать перемещения некоторых строк и обновления закладок в каждом некластеризованном индексе таблицы.

Начиная с версии SQL Server 7.0, все кластеризованные индексы имеют уникальные ключи. Если создаваемый кластеризованный индекс не является уникальным (например, если индекс создается с помощью команды `CREATE INDEX` без ключевого слова `UNIQUE`), то программа SQL Server вынуждает этот индекс стать уникальным, добавляя по мере необходимости к значениям ключа четырехбайтовое значение, называемое *унификатором*, чтобы одинаковые значения ключа стали разными.

Страницы листового уровня в некластеризованном индексе содержат ключи индекса и закладки, указывающие на основополагающий объект — кластеризованный индекс или динамическую область памяти. Закладка может иметь одну из двух форм. Если в таблице существует кластеризованный индекс, то закладка представляет собой ключ кластеризованного индекса. Если в кластеризованном и некластеризованном индексах совместно используется общий столбец ключа, то в памяти хранится только единственный экземпляр каждой закладки. Если же кластеризованный индекс отсутствует, то закладка состоит из идентификатора строки (Row IDentifier — RID), в который входят номер файла, номер страницы и номер слота строки, на которую ссылается значение ключа некластеризованного индекса.

Даже тот факт, что применение динамической области памяти (так называется таблица без кластеризованного индекса) приводит к тому, что в некластеризованных индексах ссылка на такую таблицу содержит информацию о физическом местонахождении данных, служит достаточно весомым обоснованием рекомендации, согласно которой на каждой формируемой таблице необходимо создавать кластеризованный индекс. При отсутствии такого индекса внесение изменений в таблицу, которое вызывает разбиение страниц, может повлечь за собой цепную реакцию внесения изменений во все некластеризованные индексы таблицы, поскольку физическое местонахождение строк, на которые ссылаются эти индексы, будет изменяться довольно часто. В этом фактически состоял один из основных недостатков средств индексации SQL Server, которые применялись до версии 7.0, поскольку в некластеризованных индексах всегда хранилась информация о физическом местонахождении строк, а не значения ключей кластеризованного индекса. В результате этого средства индексации испытывали отрицательно влияние, связанное с частым изменением физического местонахождения строк в базовой таблице.

Некластеризованные индексы в наибольшей степени подходят для выполнения одноэлементных выборок — запросов, которые приводят к возврату единственной строки. Дело в том, что после перехода по узлам некластеризованного B-дерева для доступа к фактическим данным достаточно выполнить всего лишь одну операцию ввода-вывода страниц, т.е. прочитать страницу из основополагающей таблицы.

Покрывающие индексы

Некластеризованный индекс называется “покрывающим” некоторый запрос, если в нем содержатся все столбцы, требуемые в запросе. Применение такого индекса позволяет миновать шаг поиска закладки и просто возвратить данные, требуемые в запросе, из собственного B-дерева индекса. Если в таблице имеется кластеризованный индекс, то запрос может быть покрыт с использованием комбинации ключевых столбцов некластеризованного и кластеризованного индексов, поскольку

ключ кластеризованного индекса представляет собой закладку некластеризованного индекса. Иными словами, если некластеризованный индекс сформирован на столбцах `LastName` и `FirstName`, а ключом кластеризованного индекса является столбец `CustomerID`, то запрос, в котором необходимо получить данные из столбцов `CustomerID` и `LastName`, может быть покрыт с помощью некластеризованного индекса. Подход, в котором используется покрывающий некластеризованный индекс, не намного хуже по своим характеристикам того подхода, в котором используется несколько кластеризованных индексов, созданных на одной и той же таблице.

Проблемы производительности

Вообще говоря, рекомендуется использовать ключи индексов, имеющие как можно меньший формат представления. Применение ключей, имеющих большие размеры, приводит к увеличению объема ввода-вывода и к уменьшению количества строк с ключевыми значениями, которые могут поместиться на каждой странице B-дерева. В результате для индекса применяется большее количество страниц, чем было бы в ином случае, а индекс занимает больший объем дискового пространства. На практике принятая стратегия индексации в основном должна соответствовать конкретным деловым потребностям. Например, если применяется такой запрос, для выполнения которого требуется чрезвычайно продолжительное время, поскольку для него необходим индекс с такими ключевыми столбцами, которые отсутствуют в каком-либо из текущих индексов, то действительно может потребоваться расширить существующий индекс или создать новый.

Безусловно, при рассмотрении вопроса о введении дополнительных индексов или столбцов индексов необходимо учитывать определенные компромиссы, например, оценивать, как модификация индексов отразится на производительности операций DML. Операции сопровождения и обновления индексов таблицы выполняются в связи с любой операцией добавления или изменения данных, поэтому введение каждого нового индекса приводит к определенному повышению издержек, связанных с сопровождением индексов. Чем больше количество введенных индексов, тем медленнее выполняются операции обновления, вставки и удаления, применяемые к основополагающим таблицам, поэтому важно, чтобы индексы оставались настолько компактными и охватывающими небольшое количество столбцов, насколько это возможно, и вместе с тем соответствовали деловым потребностям, для удовлетворения которых предназначена разрабатываемая система.

Пересечение индексов

До появления версии 7.0 оптимизатор запросов программы SQL Server должен был использовать только один индекс в расчете на каждую таблицу при поиске способа выполнения запроса. Версия SQL Server 7.0 и последующие версии позволяют применять несколько индексов в расчете на каждую таблицу и находить пересечение множеств закладок этих индексов еще до того, как начнется расходование реальных ресурсов на выборку данных из основополагающей таблицы. Реализация такого подхода привела к тому, что изменилось устройство индексов и стали применяться другие критерии выбора ключей, как вскоре будет описано.

Фрагментация индекса

Величиной фрагментации индекса можно управлять, задавая для индекса значение коэффициента заполнения и регулярно выполняя операции дефрагментации. Коэффициент заполнения индекса во многом влияет на производительность. С одной стороны, если при создании индекса был задан относительно низкий коэффициент заполнения, то количество случаев разделения страниц во время выполнения операций вставки уменьшается. Очевидно, что если страницы индекса заполнены лишь частично, то вероятность возникновения необходимости разделить одну из таких страниц для того, чтобы вставить новые строки, меньше по сравнению с вероятностью разделения полностью заполненных страниц. С другой стороны, высокое значение коэффициента заполнения позволяет формировать плотно заполненные страницы, что способствует уменьшению количества операций ввода-вывода, необходимых для обслуживания запроса. Последний метод часто применяется в хранилищах данных. Выборка страниц, которые заполнены лишь частично, связана с непроизводительным расходованием пропускной способности средств ввода-вывода.

Значение коэффициента заполнения индекса влияет только на страницы индекса листового уровня. Программа SQL Server обычно резервирует достаточный объем свободного пространства в страницах индекса промежуточных уровней для хранения по меньшей мере одной строки индекса максимального размера. Если же требуется, чтобы спецификация коэффициента заполнения распространялась не только на страницы листового уровня, но и на страницы промежуточных уровней, то необходимо ввести опцию `PAD_INDEX` оператора `CREATE INDEX`. Опция `PAD_INDEX` служит для программы SQL Server указанием на то, что заданное значение коэффициента заполнения должно распространяться на страницы индекса промежуточных уровней. Если заданное значение коэффициента заполнения настолько велико, что на страницах промежуточных уровней не хватает места даже для одной строки (например, коэффициент заполнения равен 100%), то программа SQL Server корректирует заданное в процентах значение для того, чтобы можно было поместить на страницу хотя бы одну строку. Если же заданное значение коэффициента заполнения является таким низким, что на страницах промежуточных уровней нельзя сохранить по меньшей мере две строки, то программа SQL Server корректирует значение коэффициента заполнения, выраженное в процентах, которое относится к страницам промежуточных уровней, для того, чтобы на каждой странице могли поместиться по меньшей мере две строки.

Следует учитывать, что значение коэффициента заполнения, заданное для индекса, в дальнейшем не поддерживается. Это значение применяется при первоначальном создании индекса, но после этого больше не учитывается. В качестве инструментального средства, позволяющего определить, насколько заполнены страницы в таблице и/или индексе, фактически служит команда `DBCC SHOWCONTIG`. Наиболее важными показателями, которые при этом следует анализировать, являются показатели `Logical Scan Fragmentation` (Фрагментация логического просмотра) и `Avg. Page Density` (Средняя плотность страницы). Команда `DBCC SHOWCONTIG` показывает три типа фрагментации — фрагментация просмотра экстента, фрагментация логического просмотра и плотность просмотра. Для устра-

нения фрагментации логического просмотра используется команда DBCC INDEXDEFRAG, а для полного устранения фрагментации таблицы и/или индекса применяется перестройка индексов.

В листинге 12.1 приведен пример вывода команды DBCC SHOWCONTIG, применяемой к таблице Customers базы данных Northwind.

Листинг 12.1. Пример вывода команды DBCC SHOWCONTIG

```
DBCC SHOWCONTIG (Customers)

(Результаты)

DBCC SHOWCONTIG scanning 'Customers' table...
Table: 'Customers' (2073058421); index ID: 1, database ID: 6
TABLE level scan performed.
- Pages Scanned................................: 5
- Extents Scanned..............................: 3
- Extent Switches..............................: 4
- Avg. Pages per Extent........................: 1.7
- Scan Density [Best Count:Actual Count].......: 20.00% [1:5]
- Logical Scan Fragmentation ..............:  40.00%
- Extent Scan Fragmentation ...................: 66.67%
- Avg. Bytes Free per Page.....................: 3095.2
- Avg. Page Density (full).................: 61.76%
```

Вполне очевидно, что таблица Customers немного фрагментирована. Показатель Logical Scan Fragmentation составляет 40%, а показатель Avg. Page Density – 61,76% (строки с этими показателями выделены полужирным шрифтом). Иными словами, страницы этой таблицы остаются незаполненными в среднем приблизительно на 40%. Выполним дефрагментацию кластеризованного индекса этой таблицы и определим, достигнуто ли улучшение. Данные, полученные в результате выполнения указанных действий, приведены в листинге 12.2.

Листинг 12.2. Результаты проверки таблицы Customers после дефрагментации

```
DBCC INDEXDEFRAG(Northwind,Customers,1)

(Результаты)

Pages Scanned   Pages Moved   Pages Removed
-------------   -----------   -------------
1               0             1

   DBCC SHOWCONTIG (Customers)

(Результаты)

DBCC SHOWCONTIG scanning 'Customers' table...
Table: 'Customers' (2073058421); index ID: 1, database ID: 6
TABLE level scan performed.
- Pages Scanned................................: 4
- Extents Scanned..............................: 3
- Extent Switches..............................: 2
- Avg. Pages per Extent........................: 1.3
```

```
- Scan Density [Best Count:Actual Count].......: 33.33% [1:3]
- Logical Scan Fragmentation .............:  25.00%
- Extent Scan Fragmentation ...................: 66.67%
- Avg. Bytes Free per Page.....................: 1845.0
- Avg. Page Density (full).................: 77.21%
```

Вполне очевидно, что результаты выполнения команды DBCC INDEXDEFRAG оказались весьма благоприятными. Показатель Logical Scan Fragmentation снизился до 25%, а показатель Avg. Page Density теперь лишь ненамного превышает 77%; это означает, что достигнуто улучшение примерно на 15%.

По умолчанию команда DBCC SHOWCONTIG позволяет получить информацию только об узлах листового уровня. Для просмотра других уровней таблицы/индекса необходимо задать опцию ALL_LEVELS (листинг 12.3).

Листинг 12.3. Использование команды DBCC SHOWCONTIG для просмотра данных о фрагментации на всех уровнях

```
DBCC SHOWCONTIG (Customers) WITH TABLERESULTS, ALL_LEVELS

(Результаты приведены в сокращенном виде)

ObjectName  IndexName     AveragePageDensity   ScanDensity    LogicalFragmenta
----------  ------------  -------------------  -------------  ----------------
Customers   PK_Customers   77.205337524414063   33.3333333329  25.0
Customers   PK_Customers   0.95132195949554443  0.0            0.0
```

В табл. 12.1 перечислены основные элементы данных, приведенные в отчете команды DBCC SHOWCONTIG, и показано, что они означают.

Таблица 12.1. Основные поля в выводе команды DBCC SHOWCONTIG

DBCC SHOWCONTIG	Описание
Avg. Bytes Free per Page	Среднее количество свободных байтов на каждой странице
Pages Scanned	Количество страниц, к которым был осуществлен доступ
Extents Scanned	Количество экстентов, к которым был осуществлен доступ
Out of order pages (этот показатель не отображается, но используется для вычисления показателя Logical Scan Fragmentation)	Количество таких случаев, когда в ходе просмотра текущая страница имела меньший номер, чем предыдущая
Extent Switches	Количество таких случаев, когда в ходе просмотра текущая страница относилась к другому экстенту, чем предыдущая

Для определения общей фрагментации таблицы и/или индекса автор использует поля Logical Scan Fragmentation и Avg. Page Density. На практике обычно обнаруживается, что значения этих полей по мере увеличения или уменьшения фрагментации во времени изменяются в одну и ту же сторону.

Дефрагментация

Как было показано выше, команда DBCC INDEXDEFRAG представляет собой удобный способ дефрагментации индекса. Операция, заданная этой командой, выполняется в оперативном режиме, поэтому в ходе выполнения команды индекс все еще остается применимым. Тем не менее реорганизация индекса с помощью данной команды осуществляется только на уровне листовых узлов; при этом происходит своего рода пузырьковая сортировка страниц листового уровня. Для полной дефрагментации индекса его необходимо перестроить. Для этой цели существует несколько способов. Прежде всего, можно просто удалить и воссоздать индекс с помощью операторов DROP INDEX /CREATE INDEX. Но такой способ имеет недостаток, связанный с тем, что во время повторного создания индекс находится в автономном режиме, кроме того, не разрешено уничтожать индексы, которые поддерживают ограничения. Еще один способ состоит в том, что можно использовать команду DBCC DBREINDEX или конструкцию DROP_EXISTING оператора CREATE INDEX, но и в этом случае индекс до окончания его перестройки остается недоступным. Единственным преимуществом данного способа является то, что при эксплуатации производственной версии Enterprise Edition программы SQL Server индекс может создаваться параллельно. Весь объем работы, выполняемый в операции параллельного создания индекса в программе SQL Server, распределяется по процессорам в многопроцессорной системе почти равномерно, поэтому в многопроцессорной системе продолжительность времени пребывания индекса в автономном режиме в ходе его воссоздания может оказаться намного меньше, чем в такой однопроцессорной системе, которая ничем не отличается от многопроцессорной, если не считать того, что в этих системах имеется разное количество процессоров.

Вообще говоря, команда DBCC INDEXDEFRAG является наилучшим инструментальным средством решения задачи дефрагментации индекса, за исключением тех ситуаций, когда обнаруживается обширная фрагментация на уровнях индекса, отличных от листового, и создается впечатление, что такая фрагментация приводит к недопустимо резкому снижению производительности выполнения запроса. Как уже было сказано выше, для проверки фрагментации на других уровнях индекса можно передать опцию ALL_LEVELS в команду DBCC SHOWCONTIG.

Команда DBCC INDEXDEFRAG не только обеспечивает дефрагментацию индекса на листовом уровне за счет сортировки страниц листового уровня, но и осуществляет этап уплотнения, на котором страницы индекса уплотняются с использованием первоначального коэффициента заполнения страниц в качестве целевого значения. В ходе выполнения этапа уплотнения предпринимается попытка оставить ко времени завершения этапа на каждой странице достаточно места по меньшей мере для одной строки. Если указанная команда не может получить блокировку на какую-то конкретную страницу во время уплотнения, то пропускает эту страницу. Кроме того, команда удаляет все страницы, которые в результате уплотнения становятся полностью пустыми.

Индексы на представлениях и вычисленных столбцах

Создание индекса на представлении или вычисленном столбце обеспечивает преобразование в постоянно хранимые таких данных, которые в противном случае существовали бы только как виртуальные. Обычно данные, возвращаемые из представления, существуют только в таблицах, к которым обращается это представление. При формирования запроса к представлению этот запрос объединяется с запросом, лежащим в основе представления, после чего осуществляется выборка данных из основополагающих объектов (таблиц). Такое же утверждение является справедливым по отношению к вычисленным столбцам. В обычных обстоятельствах данные, возвращаемые вычисленным столбцом, фактически не существуют, независимо от того, какие столбцы или выражения используются для получения этих данных. Каждый раз, когда вычисленные данные запрашиваются из таблицы, в которой находятся соответствующие этим данным исходные данные, повторно вычисляется выражение, предназначенное для получения расчетных данных, поэтому расчетные данные вырабатываются динамически.

Приступая к формированию индексов на представлении, следует начинать с создания уникального кластеризованного индекса. Именно на этом этапе происходит создание реальных постоянно хранимых данных. Как и в таблицах, кластеризованный индекс, созданный на представлении, фактически хранит в своих узлах листового уровня сами данные. А после формирования кластеризованного индекса можно также приступить к созданию на представлении некластеризованных индексов.

В этом состоит отличие представлений от вычисленных столбцов в таблицах. При использовании вычисленных столбцов не требуется вначале создавать кластеризованный индекс, чтобы можно было сформировать некластеризованные индексы. Дело в том, что вычисленные столбцы служат просто в качестве значений ключа индекса, поэтому для вычисленных столбцов вполне подходят некластеризованные индексы.

Условия создания индексов

Для того чтобы можно было создать индекс на представлении или на вычисленном столбце в программе SQL Server, требуется, чтобы были заданы определенные значения семи опций SET. Опции SET и требуемые для них значения перечислены в табл. 12.2. Как показывает эта таблица, все опции SET, кроме NUMERIC_ROUNDABORT, должны иметь значение ON.

Таблица 12.2. Значения опций SET, которые должны быть заданы перед созданием индекса на представлении или на вычисленном столбце

Опция SET	Требуемое значение
ARITHABORT	ON
CONCAT_NULL_YIELDS_NULL	ON
QUOTED_IDENTIFIER	ON
ANSI_NULLS	ON

Окончание табл. 12.2

Опция SET	Требуемое значение
ANSI_PADDING	ON
ANSI_WARNINGS	ON
NUMERIC_ROUNDABORT	OFF

Для работы с индексированными представлениями и с индексами на вычисленных столбцах могут использоваться только детерминированные выражения. Детерминированным выражением является такое выражение, которое всегда возвращает одни и те же выходные данные при передаче в него одних и тех же входных данных. Например, выражение `SUBSTRING('He who loves money more than truth will end up poor',23,7)` является детерминированным, а выражение `GETDATE` — нет.

Для проверки того, является ли представление или столбец индексируемым, можно воспользоваться функциями `OBJECTPROPERTY` и `COLUMNPROPERTY` языка Transact-SQL (листинг 12.4).

Листинг 12.4. Проверка того, является ли представление или столбец индексируемым

```
USE Northwind
SELECT OBJECTPROPERTY (OBJECT_ID('Invoices'), 'IsIndexable')
SELECT COLUMNPROPERTY (OBJECT_ID('syscomments'), 'text' ,
                       'IsIndexable')
SELECT COLUMNPROPERTY (OBJECT_ID('syscomments'), 'text' ,
                       'IsDeterministic')

(Результаты)

-----------
0

-----------
0

-----------
0
```

Еще одним, последним, условием применительно к представлениям является то, что представление индексируемо, только если оно было создано с помощью опции `SCHEMABINDING`. Если представление создано с опцией `SCHEMABINDING`, это вынуждает программу SQL Server исключать возможность уничтожения объектов, на которые ссылается это представление. Соответствующие объекты могут быть уничтожены только после того, как само представление будет уничтожено или модифицировано таким образом, что в нем будет удалена опция `SCHEMABINDING`. Кроме того, завершаются неудачей операторы `ALTER TABLE`, применяемые к таблицам, на которые ссылается представление, если эти операторы не соответствуют определению представления. В приведенном выше примере представление `Invoices` не является индексируемым, поскольку оно не было создано с опцией

SCHEMABINDING. А ниже приведена версия Invoices2 представления Invoices, которая является индексируемой, как показывают приведенные в том же листинге результаты проверки свойства IsIndexable (листинг 12.5).

Листинг 12.5. Индексируемое представление Invoices2

```
CREATE VIEW Invoices2
WITH SCHEMABINDING
AS
SELECT Orders.ShipName, Orders.ShipAddress, Orders.ShipCity,
  Orders.ShipRegion, Orders.ShipPostalCode, Orders.ShipCountry,
  Orders.CustomerID, Customers.CompanyName AS CustomerName,
  Customers.Address, Customers.City, Customers.Region,
  Customers.PostalCode, Customers.Country, Orders.OrderID,
  Orders.OrderDate, Orders.RequiredDate, Orders.ShippedDate,
  Shippers.CompanyName As ShipperName, "Order Details".ProductID,
  Products.ProductName, "Order Details".UnitPrice,
  "Order Details".Quantity, "Order Details".Discount,
  Orders.Freight
  FROM dbo.Shippers INNER JOIN
    (dbo.Products INNER JOIN
      (
        (dbo.Employees INNER JOIN
          (dbo.Customers INNER JOIN dbo.Orders
            ON Customers.CustomerID = Orders.CustomerID)
          ON Employees.EmployeeID = Orders.EmployeeID)
      INNER JOIN dbo.[Order Details]
          ON Orders.OrderID = "Order Details".OrderID)
    ON Products.ProductID = "Order Details".ProductID)
  ON Shippers.ShipperID = Orders.ShipVia
GO
SELECT OBJECTPROPERTY (OBJECT_ID('Invoices2'), 'IsIndexable')

(Результаты)

-----------
1
```

Обратите внимание на то, что во всех ссылках на объекты в этой версии используются имена, состоящие из двух частей (отметим, что имена не были таковыми в первоначальном представлении Invoices). Для создания представления с опцией SCHEMABINDING требуется, чтобы во всех ссылках на объекты применялись имена, состоящие из двух частей.

После создания индекса на представлении оптимизатор может использовать такой индекс при выполнении запросов к этому представлению. В действительности в версии Enterprise Edition программы SQL Server оптимизатор может также использовать индекс представления для обслуживания запроса к основополагающим таблицам представления, если будет обнаружено, что применение такого индекса приводит к снижению стоимости выполнения запроса, измеряемой продолжительностью времени выполнения. При обычных обстоятельствах индексированные представления вообще не используются оптимизатором, если только представление не применяется в версии Enterprise Edition. Например, рассмотрим следующий индекс и запрос (листинг 12.6).

Листинг 12.6. Запрос к индексированному представлению

```
CREATE UNIQUE CLUSTERED INDEX inv ON invoices2 (orderid, productid)
GO
SELECT * FROM invoices2 WHERE orderid=10844 AND productid=22

(Результаты приведены в сокращенном виде)

ShipName          ShipAddress  ShipCity ShipRegion ShipPostalCode
----------------- ------------ -------- ---------- --------------
Piccolo und mehr  Geislweg 14  Salzburg NULL       5020
```

Выдержка из плана выполнения для этого запроса приведена в листинге 12.7.

Листинг 12.7. План выполнения запроса, показанного в листинге 12.6

```
StmtText
------------------------------------------------------------------------------
SELECT * FROM [invoices2] WHERE [orderid]=@1 AND [productid]=@2
  |--Nested Loops(Inner Join)
       |--Nested Loops(Inner Join)
       |    |--Nested Loops(Inner Join, OUTER REFERENCES:([Orders].[ShipVia]))
       |    |    |--Nested Loops(Inner Join, OUTER REFERENCES:([Orders].[Employ
       |    |    |    |--Nested Loops(Inner Join, OUTER REFERENCES:([Orders].[C
       |    |    |    |     |--Clustered Index Seek(OBJECT:([Northwind].[dbo].[
       |    |    |    |     |--Clustered Index Seek(OBJECT:([Northwind].[dbo].[
       |    |    |    |--Clustered Index Seek(OBJECT:([Northwind].[dbo].[Employ
       |    |    |--Clustered Index Seek(OBJECT:([Northwind].[dbo].[Shippers].[
       |    |--Clustered Index Seek(OBJECT:([Northwind].[dbo].[Order Details].[
       |--Clustered Index Seek(OBJECT:([Northwind].[dbo].[Products].[PK_Product
```

В этом листинге правая часть текста плана обрезана, но рассматриваемый текст все равно позволяет определить, что индекс представления не используется, даже несмотря на то, что в этом индексе содержатся оба столбца, которые применяются в критериях выборки данного запроса. Такой результат является вполне предсказуемым в версиях программы SQL Server, отличных от версии Enterprise Edition. По умолчанию при формировании плана выполнения индексы представлений рассматриваются только в версии Enterprise Edition программы SQL Server. Тем не менее из этой ситуации есть выход. В версиях программы SQL Server, отличных от Enterprise Edition, можно использовать в запросе подсказку NOEXPAND, чтобы вынудить оптимизатор учитывать наличие индекса представления. Ниже еще раз приведен рассматриваемый запрос, но на этот раз с ключевым словом NOEXPAND, а также показан результирующий план выполнения запроса (листинг 12.8).

Листинг 12.8. Использование ключевого слова NOEXPAND, позволяющего вынудить оптимизатор применить индекс представления

```
SELECT * FROM invoices2 WITH (NOEXPAND) WHERE orderid=10844 AND productid=22

(Результаты)

StmtText
------------------------------------------------------------------------------
SELECT * FROM invoices2 (NOEXPAND) WHERE orderid=10844 AND productid=22
  |--Clustered Index Seek(OBJECT:([Northwind].[dbo].[Invoices2].[inv]), SEEK:([
```

Обратите внимание на то, что теперь индекс используется. Введение ключевого слова NOEXPAND вынуждает оптимизатор использовать индекс представления, даже если такое решение приводит к созданию неоптимального плана. Поэтому опцию NOEXPAND следует рассматривать под таким же критическим углом, как и все другие подсказки в запросах. Лучше всего позволить оптимизатору выполнять свою работу и вмешиваться самому лишь тогда, когда нет другого выхода.

Блокировка и индексы

Одним из характерных признаков того, что на таблице должен быть создан кластеризованный индекс, является возникновение таких ситуаций, при которых на таблице устанавливаются блокировки RID. В программе SQL Server никогда не устанавливаются блокировки RID на таблице с кластеризованным индексом; вместо этого всегда устанавливаются блокировки ключей.

Вообще говоря, следует предоставить возможность управлять блокировками всех типов, включая блокировки, связанные с индексами, самой программе SQL Server. В обычных условиях эта серверная программа сама принимает качественные решения и наилучшим образом управляет своими собственными ресурсами.

Для управления вручную тем, какие типы блокировок могут быть разрешены на индексированной таблице, применяется системная процедура sp_indexoption. С помощью этой процедуры можно запретить использование блокировок строк и/или страниц, но автор не рекомендует выполнять такое действие в обычных обстоятельствах. Как и для подсказок запросов, обычно лучше разрешить серверу самому решать, какого типа блокировка должна быть установлена на том или ином ресурсе.

Следует отметить, что процедура sp_indexoption применяется только к индексам, поэтому с ее помощью нельзя управлять блокировкой страниц в динамической области памяти. Тем не менее, если таблица имеет кластеризованный индекс, то значения, заданные с помощью процедуры sp_indexoption, влияют и на такую таблицу.

Статистические данные

Читатель, по-видимому, уже знает о том, что производительность выполнения запросов в программе SQL Server связана с таким понятием, как “статистические данные”. Статистические данные представляют собой метаданные, относящиеся к ключам индексов, и в качестве дополнительной опции — к значениям неиндексированных столбцов. Эти данные сопровождаются программой SQL Server. В программе SQL Server статистические данные используются для определения того, приведет ли применение индекса к ускоренному выполнению запроса. В сочетании с индексами статистические данные представляют собой один из наиболее важных источников информации, позволяющих оптимизатору разрабатывать оптимальные планы выполнения. Если статистические данные отсутствуют или

являются устаревшими, то способность оптимизатора формировать наилучшие планы выполнения запроса значительно уменьшается.

Прежде чем приступить к более подробному описанию статистических данных, рассмотрим несколько основных терминов, касающихся статистических данных.

Кардинальность

Кардинальностью данных называется количество уникальных значений в данных. Строго говоря, теория реляционных баз данных не допускает наличия дубликатов строк (кортежей) в отношении (таблице), поэтому теоретически кардинальность обозначает общее количество кортежей. Тем не менее в программе SQL Server допускается наличие дубликатов строк в таблице, поэтому будем считать, что термином "кардинальность" обозначается количество уникальных значений в наборе данных.

Плотность

Плотностью характеризует уникальность значений в наборе данных. Плотность индекса вычисляется путем деления количества строк, которые соответствуют данному конкретному значению ключа, на количество строк в таблице. Если индекс является уникальным, задача определения значения этого показателя сводится к выполнению операции деления числа 1 на общее количество строк в таблице. Значения плотности изменяются в пределах от 0 до 1; чем ниже показатель плотности, тем лучше.

Избирательность

Избирательность — это мера оценки количества строк, которые должны быть возвращены при использовании какого-то конкретного критерия запроса. Этот показатель выражает связь между критериями запроса и значениями ключей в индексе. Значение данного показателя вычисляется путем деления количества требуемых значений ключей на количество строк, к которым можно получить доступ с помощью этих ключей. Наиболее полезными для оптимизатора являются критерии запроса (обычно представленные в конструкции `WHERE`), которые характеризуются высокой избирательностью, поскольку такие критерии позволяют надежно предсказывать, какое количество операций ввода-вывода потребуется для выполнения запроса.

Проблемы производительности

Оптимизатор чаще всего игнорирует индексы, характеризующиеся высокими значениями плотности. Наиболее полезными для оптимизатора являются индексы, имеющие значения плотности от 0,10 и ниже. Рассмотрим в качестве примера таблицу `VoterRegistration` (список избирателей), которая включает 10 тыс. строк,

не имеет кластеризованного индекса, но имеет некластеризованный индекс на столбце `PartyAffiliation` (партия, сторонником которой является данный избиратель). Если в рассматриваемом избирательном округе зарегистрированы три политические партии и каждая из них имеет примерно одинаковое количество сторонников среди избирателей, то столбец `PartyAffiliation` будет, скорее всего, содержать только три уникальных значения. Таким образом, каждое конкретное значение ключа в индексе позволяет осуществить выборку из таблицы примерно 3333 строк, (возможно, и больше). Это означает, что индекс на столбце `PartyAffiliation` имеет плотность 0,33 (3333 ÷ 10 000), то есть фактически гарантирует, что оптимизатор не будет использовать индекс при формировании плана выполнения запросов, в которых требуются столбцы, не покрытые этим индексом.

Чтобы лучше разобраться в данной ситуации, сравним стоимости выполнения простого запроса с использованием и без использования такого индекса. Если бы нужно было получить список всех избирателей рассматриваемого округа, которые являются сторонниками демократической партии, то пришлось бы рассматривать ситуацию, в которой требуется осуществить выборку с помощью ключа примерно 3333 строк, т.е. третьей части таблицы. Если для доступа к этим строкам используется индекс `PartyAffiliation`, то выполняется примерно 3333 отдельных логических операций чтения страниц из основополагающей таблицы. Иными словами, после обнаружения каждого значения ключа в индексе необходимо найти закладку, которая указывает на основополагающую таблицу, чтобы получить значения столбцов, не содержащиеся в индексе, а при каждом выполнении такой операции приходится нести издержки, связанные с вводом-выводом логической (возможно, и физической) страницы. Это означает, что объем издержек при выполнении операций ввода-вывода страниц, связанных с поиском значений закладок, может достигать 26 Мбайт (3333 ключей * 8 Кбайт/страница). Теперь рассмотрим стоимость простого последовательного просмотра этой таблицы. Если на каждой странице данных помещается в среднем 50 строк, и необходимо прочитать всю таблицу, чтобы найти все строки, относящиеся к избирателям, поддерживающим демократическую партию, то придется в ходе выполнения ввода-вывода просмотреть всего лишь около 200 логических страниц (10 000 строк * 50 строк/страница = 200 страниц). Эта разница очень велика и является главной причиной того, что некластеризованные индексы с высокой плотностью игнорируются в пользу полного просмотра таблиц или просмотра с помощью кластеризованного индекса.

Теперь рассмотрим, при каких обстоятельствах некластеризованный индекс становится достаточно избирательным для того, чтобы оказаться полезным для оптимизатора. В данном примере критерием оценки применимости некластеризованного индекса является число 200. В частности, оптимизатор должен руководствоваться следующим прогнозом: с помощью индекса должна осуществляться такая выборка данных, при которой потребуется количество операций ввода-вывода страниц меньше 200. Только при этом условии применяемый индекс может рассматриваться как более эффективный путь доступа, чем простой просмотр всей таблицы. Первоначальную оценку в 3333 строк можно уменьшить, добавляя в индекс (а также в запрос) столбцы, позволяющие добиться большей избирательности. Однако попытка уменьшить таким образом количество выполняемых операций

имеет свою оборотную сторону. Введение дополнительных столбцов в индекс для того, чтобы этот индекс стал более избирательным, приводит к увеличению объема издержек, связанных с перемещением по B-дереву индекса. Если индекс занимает большой объем памяти, то велики и затраты, связанные с переходом по этому индексу. В какой-то момент обнаруживается, что менее дорогостоящим становится план, предусматривающий просто просмотр самих данных и не требующий издержек, обусловленных необходимостью продвижения в B-дереве.

Организация хранения статистических данных

В программе SQL Server статистические данные, относящиеся к ключу индекса или столбцу, хранятся в столбце `statblob` таблицы `sysindexes`. Столбец `statblob` имеет тип данных `image` и хранит гистограмму, содержащую результаты выборки значений в ключе индекса или столбце. В случае составных индексов выборка осуществляется только в первом столбце индекса, а для других столбцов сопровождаются значения плотности.

На этапе выбора индексов процедуры оптимизации запроса оптимизатор определяет, соответствует ли индекс столбцам, заданным в критериях выборки; устанавливает избирательность индекса применительно к указанным критериям и оценивает стоимость доступа к данным, требуемым в запросе.

Если индекс состоит только из одного столбца, то относящиеся к нему статистические данные включают одну гистограмму и одно значение плотности. Если же индекс состоит из нескольких столбцов, то для него сопровождается единственная гистограмма наряду со значениями плотности для каждой префиксной комбинации столбцов ключа (определяемой в порядке расположения столбцов слева направо). Оптимизатор использует указанное сочетание гистограммы индекса и его плотностей (т.е. статистические данные, относящиеся к индексу) при определении того, насколько полезным является индекс для выполнения данного конкретного запроса.

Тот факт, что предусмотрено хранение гистограммы только для первого столбца составного индекса, является одной из причин, по которым следует ставить на первое место в многостолбцовом индексе наиболее избирательный столбец; в таком случае гистограмма становится для оптимизатора более полезной. Кроме того, в этом также состоит одна из причин, по которой иногда рекомендуется разбивать составные индексы на несколько одностолбцовых индексов. Сервер может применять операции пересечения и соединения к нескольким индексам одной и той же таблицы, поэтому использование нескольких индексов позволяет сохранить преимущество, обусловленное наличием индексов на столбцах, и вместе с тем получить дополнительное преимущество, связанное с наличием гистограммы для каждого столбца (в этом может также помочь применение статистических данных столбцов). Это — не абсолютная рекомендация (поэтому не следует, прочитав данные строки, сразу же приступать к уничтожению всех существующих составных индексов). Достаточно просто учитывать, что иногда одной из применимых опций настройки производительности является разбиение составных индексов на отдельные индексы.

Статистические данные о столбцах

Программа SQL Server способна не только накапливать статистические данные об индексах, но и формировать статистические данные о неиндексированных столбцах. (Такая операция осуществляется автоматически при выполнении запроса к неиндексированному столбцу, если для базы данных разрешена опция AUTO_CREATE_STATISTICS.) Имея возможность определить вероятность того, что в столбце может встретиться какое-то конкретное значение, оптимизатор получает ценную информацию от том, как лучше всего выполнить тот или иной запрос. Наличие статистических данных позволяет оптимизатору оценивать количество строк, которые будут получены из данной конкретной таблицы, участвующей в соединении, что дает возможность более точно выбирать порядок соединения. Кроме того, оптимизатор может использовать статистические данные о столбцах для предоставления информации в форме гистограммы для других столбцов в многостолбцовом индексе. По сути, действует такой принцип — чем больше объем информации об обрабатываемых данных, тем лучше.

Структура статистических данных

В программе SQL Server статистические данные используются для контроля над распределением значений ключей в таблице. Гистограмма, хранящаяся в составе статистических данных об индексе, содержит в первом столбце ключа индекса выборку значений, состоящую не больше чем из 200 значений. Кроме гистограммы, столбец statblob содержит также перечисленные ниже данные.

- Количество строк, которые использовались для вычисления гистограммы и плотностей.
- Средняя длина ключа индекса.
- Дата и время выполнения последней операции формирования статистических данных.
- Значения плотности для других префиксных комбинаций столбцов ключа.

Диапазон значений ключа между каждым из 200 значений выборки гистограммы называется шагом. Каждое значение выборки обозначает конец шага, а в каждом шаге хранятся три описанных ниже значения.

1. EQ_ROWS. Количество строк со значением ключа, соответствующим значению выборки.
2. RANGE_ROWS. Количество других значений в этом диапазоне.
3. RANGE_DENSITY. Результаты вычисления плотности в самом этом диапазоне.

Команда DBCC SHOW_STATISTICS выводит в непосредственном виде данные EQ_ROWS и RANGE_ROWS, а также использует значение RANGE_DENSITY для вычисления показателей DISTINCT_RANGE_ROWS и AVG_RANGE_ROWS, относящихся к данному шагу. Это команда вычисляет значение DISTINCT_RANGE_ROWS (общее количество различных строк в диапазоне, относящемся к данному шагу) путем де-

ления 1 на RANGE_DENSITY, а также вычисляет значение AVG_RANGE_ROWS (среднее количество строк в расчете на каждое различимое значение ключа) путем умножения RANGE_ROWS на RANGE_DENSITY.

Обновление статистических данных

Для обновления статистических данных может применяться целый ряд способов. Первый и наиболее очевидный из этих способов состоит в использовании опции базы данных AUTO_UPDATE_STATISTICS. (Чтобы разрешить применение этой опции, можно воспользоваться оператором ALTER DATABASE, процедурой sp_dboption или программой Enterprise Manager.) При автоматическом формировании статистических данных для таблицы любого размера в программе SQL Server для ускорения работы используется формирование выборок (а не просмотр всей таблицы). Такой подход себя оправдывает в большинстве случаев, но иногда ведет к формированию статистических данных, которые являются менее полезными, чем могли бы быть.

С операцией автоматического обновления статистических данных тесно связана операция автоматического создания статистических данных. Эта операция выполняется, если разрешена опция базы данных AUTO_CREATE_STATISTICS и на обработку поступил запрос, предусматривающий выборку по определенному критерию значений в неиндексированном столбце (или запрос к столбцам, не относящимся к префиксу, выровненному слева направо по отношению к критериям выборки). В таком случае программа SQL Server автоматически создает набор статистических данных о столбцах.

Еще один метод обновления статистических данных состоит в использовании команды UPDATE STATISTICS. В версиях, предшествующих SQL Server 7.0, команда UPDATE STATISTICS представляла собой единственный способ обновления статистических данных. При выполнении команды UPDATE STATISTICS может либо использоваться формирование выборок, как при автоматическом обновлении, либо осуществляться полный просмотр таблицы, что приводит к получению более качественных статистических данных, но может потребовать больше времени.

Команда CREATE STATISTICS выполняет функции, аналогичные команде UPDATE STATISTICS. Команда CREATE STATISTICS может использоваться для создания статистических данных о столбцах вручную. Эти статистические данные после их создания могут обновляться с помощью средств автоматического обновления или с помощью команды UPDATE STATISTICS (по такому же принципу, как обновляются обычные статистические данные об индексах).

В программе SQL Server предусмотрено применение нескольких хранимых процедур, позволяющих упростить создание и обновление статистических данных. Процедура sp_updatestats обеспечивает выполнение команды UPDATE STATISTICS применительно ко всем определяемым пользователем таблицам в текущей базе данных. Но в отличие от самой команды UPDATE STATISTICS, процедура sp_updatestats не позволяет использовать полный просмотр таблицы для формирования статистических данных, поскольку в ней всегда осуществляется выборка

данных. Если есть необходимость сформировать статистические данные с помощью полного просмотра, следует воспользоваться командой `UPDATE STATISTICS`.

Такой же удобной может оказаться процедура `sp_createstats`. Эта процедура позволяет автоматизировать создание статистических данных о столбцах для всех применимых столбцов во всех приемлемых таблицах базы данных. К применимым столбцам относятся невычисленные столбцы с данными типов, которые отличаются от типов `text`, `ntext` или `image` и еще не имеют статистических данных о столбцах или статистических данных о первых столбцах индексов. К приемлемым таблицам относятся все пользовательские таблицы (таблицы, отличные от системных). Автор не рекомендует бездумно выполнять процедуру `sp_createstats` во всех эксплуатируемых базах данных, поскольку маловероятно, что в процессе работы потребуются статистические данные, относящиеся к каждому столбцу в таблице. Но в том случае, если такая необходимость действительно существует, процедура `sp_createstats` реально экономит время.

Процедура `sp_autostats` позволяет управлять автоматическим обновлением статистических данных на уровнях таблицы и индекса. Вместо простого ввода в действие опции базы данных `AUTO_UPDATE_STATISTICS` с помощью указанной процедуры можно разрешать или запрещать автоматическое формирование статистических данных на более тонко детализированном уровне. Например, если предусмотрено обновление в ночную смену статистических данных большой таблицы с использованием просмотра, то желательно было бы отменить автоматическое обновление статистических данных для такой таблицы. С помощью процедуры `sp_autostats` можно запретить автоматические обновления статистических данных для одной подобной таблицы и вместе с тем разрешить выполнение указанной операции для остальной части базы данных (можно также использовать команду `UPDATE STATISTICS … WITH NORECOMPUTE`). Обновление статистических данных для больших таблиц, даже выполняемое с помощью выборки, может потребовать много времени, а также значительных ресурсов процессора и ресурсов ввода-вывода.

Следует учитывать, что отсутствие статистических данных или применение устаревших статистических данных оказывает отрицательное влияние на производительность, которое почти всегда перевешивает экономию ресурсов, достигнутую в связи с отказом от автоматического создания (обновления) статистических данных. Отменять автоматическое создание (обновление) статистических данных следует лишь в том случае, если всесторонняя проверка показала, что нет иного способа достичь требуемых показателей производительности или масштабируемости.

В листинге 12.9 приведена хранимая процедура, которая может использоваться для контроля над операциями обновления статистических данных. Эта процедура показывает тип статистических данных, время последнего обновления данных, а также много другой информации, которая может оказаться полезной при управлении статистическими данными об индексах и столбцах.

Листинг 12.9. Процедура sp_showstatdate, применяемая для контроля над операциями обновления статистических данных

```
CREATE PROC sp_showstatdate @tabmask sysname='%',
    @indmask sysname='%'
AS
SELECT
  LEFT(CAST(USER_NAME(uid)+'.'+o.name AS sysname),30) AS TableName,
  LEFT(i.name,30) AS IndexName,
  CASE WHEN INDEXPROPERTY(o.id,i.name,'IsAutoStatistics')=1
         THEN 'AutoStatistics'
       WHEN INDEXPROPERTY(o.id,i.name,'IsStatistics')=1
         THEN 'Statistics'
  ELSE 'Index'
  END AS Type,
  STATS_DATE(o.id, i.indid) AS StatsUpdated,
  rowcnt,
  rowmodctr,
  ISNULL(CAST(rowmodctr/CAST(NULLIF(rowcnt,0) AS decimal(20,2))*100
      AS int),0) AS PercentModifiedRows,
  CASE i.status & 0x1000000
  WHEN 0 THEN 'No'
  ELSE 'Yes'
  END AS [NoRecompute?],
  i.status
FROM dbo.sysobjects o JOIN dbo.sysindexes i ON (o.id = i.id)
WHERE o.name LIKE @tabmask
  AND i.name LIKE @indmask
  AND OBJECTPROPERTY(o.id,'IsUserTable')=1
  AND i.indid BETWEEN 1 AND 254
ORDER BY TableName, IndexName
GO
USE pubs
GO
EXEC sp_showstatdate

(Результаты приведены в сокращенном виде)

TableName         IndexName       Type       StatsUpdated
----------------- --------------- ---------- ----------------------
dbo.authors       au_fname         Statistics 2000-07-02 19:42:04.487
dbo.authors       aunmind         Index       2000-06-30 20:54:56.737
dbo.authors       UPKCL_auidind   Index       2000-06-30 20:54:56.737
dbo.dtproperties  pk_dtproperties Index      NULL
dbo.employee      employee_ind    Index       2000-06-30 20:54:45.280
dbo.employee      PK_emp_id       Index       2000-06-30 20:54:45.297
```

Оценка избирательности

Чтобы правильно определить относительную стоимость плана запроса, оптимизатор должен обладать способностью точно оценить количество строк, возвращаемых запросом. Как было указано выше, соответствующее значение называется избирательностью, причем крайне важно, чтобы оптимизатор был способен правильно определить это значение.

Для оценки избирательности производится сравнение критериев запроса со статистическими данными о ключе индекса или о столбце, на которые ссылаются эти критерии. Показатель избирательности позволяет определить, следует ли рассчитывать на получение одной строки или 10 тыс. строк после ввода данного конкретного значения ключа. Этот показатель позволяет получить представление о том, сколько строк соответствует искомому значению ключа или столбца. А информация о количестве строк, в свою очередь, позволяет установить, какой именно подход окажется наиболее эффективным для доступа к требуемым строкам. Очевидно, что для выборки одной строки следует применять иной метод, чем для выборки 10 тыс. строк.

Если оптимизатор обнаруживает, что в базе данных отсутствуют статистические данные об индексе или о столбце, относящиеся к одному из столбцов в критерии выборки запроса, то оптимизатор может автоматически создать статистические данные о столбце на уровне столбца, если разрешена опция базы данных `AUTO_CREATE_STATISTICS`. В первый момент в связи с этим придется понести затраты, связанные с формированием статистических данных, но благодаря наличию вновь полученных статистических данных последующие запросы должны выполняться быстрее.

Индексируемые выражения

В других книгах и материалах, посвященных программе SQL Server, часто можно встретить обсуждение темы об использовании индексов в планах запросов, в котором применяется термин SARG (Search ARGument in a query — параметр поиска в запросе). Кроме того, во многих информационных источниках говорится о том, что оптимизатор может преобразовать параметры SARG в операторы сравнения ключа индекса и некоторого значения. Тем не менее упоминание об указанном понятии в самом коде оптимизатора отсутствует. Термин SARG — это анахронизм, который остался от устаревшей кодовой базы Sybase и от выпущенных ранее учебных материалов Sybase. Оптимизатор во всех новейших версиях программы SQL Server является гораздо более развитым с точки зрения его способности использовать индексы для ускорения выполнения запросов, чем оптимизатор в старых версиях указанного программного продукта. Способность оптимизатора использовать индексы больше не ограничивается простыми выражениями, а само принимаемое оптимизатором решение о том, использовать или не использовать индекс, уже не обязательно зависит от того, какую форму может иметь выражение. Иными словами, может иметь или не иметь значение то, какое выражение фактически используется в предикате запроса, в зависимости от имеющихся индексов, способности оптимизатора выявить ссылки на столбцы в рассматриваемом выражении, а также от оценок стоимости, вырабатываемых оптимизатором. Термин SARG становится все менее и менее применимым. По мнению автора, его уже можно заменить более точным термином, который в большей степени соответствует определению выражений предиката запроса, позволяющих успешно использовать индексы. В качестве такого более правиль-

ного и точного термина (который, к тому же, используется в самом коде оптимизатора) можно назвать термин "индексируемое выражение" (так называется выражение, которое можно раскрыть с применением значений индекса для обеспечения более быстрого доступа к данным, возвращаемым запросом). Поэтому в настоящей книге используется термин "индексируемое выражение".

Определим индексируемое выражение как конструкцию в запросе, которая в принципе может использоваться оптимизатором в сочетании с индексом для регламентации объема результатов, возвращаемых запросом. Оптимизатор предпринимает попытки выявить индексируемые выражения в критериях запроса, чтобы иметь возможность определить наилучшие индексы, которые можно было бы применить для обслуживания запроса.

Вообще говоря, индексируемые выражения имеют следующую форму (крайние члены этого выражения могут быть указаны в обратном порядке):

```
Column op Constant/Variable
```

Здесь `Column` — столбец таблицы; `op` — один из следующих операторов: =, >=, <=, >, <, <>, !=, !>, !<, `BETWEEN` и `LIKE` (одни конструкции `LIKE` могут быть преобразованы в индексируемые выражения, а другие — нет); `Constant/Variable` — это значение константы, ссылка на переменную или одна из многочисленных функций.

Преобразование операторов

Некоторые из операторов, упомянутых выше, не могут непосредственно использоваться в поиске по индексу, но оптимизатор может преобразовать их в выражения, применимые для указанной цели. Например, рассмотрим следующий запрос:

```
SELECT * FROM authors
WHERE au_lname != 'Greene'
```

Является ли конструкция `WHERE` в этом запросе индексируемой? Ответ на этот вопрос положителен. Рассмотрим, какое при этом осуществляется преобразование, в следующей выдержке из плана запроса:

```
SEEK:([authors].[au_lname] < 'Greene' OR
[authors].[au_lname] > 'Greene')
```

Оптимизатор обладает достаточными возможностями, чтобы определить, что выражение `x != @parm` представляет собой то же, что и выражение `x < @parm OR x > @parm`, и преобразовывает первое выражение соответствующим образом. Два компонента конструкции `OR` можно выполнить параллельно и осуществить слияние полученных данных, поэтому применение указанного преобразования обеспечивает использование индекса для обслуживания приведенного выше критерия `WHERE`.

Теперь рассмотрим аналогичный запрос, в котором применяется выражение `LIKE`:

```
SELECT * FROM authors
WHERE au_lname LIKE 'Gr%'
```

Является ли это выражение индексируемым? И на этот вопрос можно ответить утвердительно. Оптимизатор и в данном случае преобразует критерии конструкции `WHERE` в такой план запроса, который считается более приемлемым, как показывает приведенная ниже выдержка из плана запроса.

```
SEEK:([authors].[au_lname] >= 'GQ' AND
[authors].[au_lname] < 'GS')
```

Рассмотрим еще один запрос:

```
SELECT * FROM authors
WHERE au_lname !> 'Greene'
```

Может ли оптимизатор использовать индекс для выполнения данного запроса? Да, такая возможность существует. Ниже приведена выдержка из плана запроса.

```
SEEK:([authors].[au_lname] <= 'Greene')
```

Рассмотрим еще один запрос:

```
SELECT * FROM authors
WHERE au_lname !< 'Greene'
```

А ниже приведен соответствующий фрагмент плана запроса.

```
SEEK:([authors].[au_lname] >= 'Greene')
```

Внимательный читатель должен заметить то, что связывает между собой эти примеры. Оптимизатор в каждом случае предпринимает попытки преобразовать выражения, которые внешне кажутся неиндексируемыми, в такие выражения, которые можно более легко обслуживать с помощью индексов.

Индексируемые выражения можно соединять друг с другом с помощью оператора AND для формирования составных конструкций. Эмпирическое правило, позволяющее определить, является ли выражение индексируемым, состоит в следующем: некоторая конструкция может обслуживаться с помощью индекса, если оптимизатор способен определить, что эта конструкция представляет собой сравнение значения ключа индекса с константой или переменной. Распространенная ошибка, которую допускают неопытные разработчики, состоит в том, что в выражение, применяемое для сравнения с константой или переменной, включают столбец. Чаще всего такой подход исключает возможность обслуживать полученную конструкцию с помощью индекса, поскольку оптимизатор не может определить, какое значение фактически приобретет данное выражение в результате вычисления (ведь это невозможно определить до этапа прогона). (Из этого правила есть исключение; ниже приведено описание подхода, позволяющего предоставить оптимизатору возможность использовать результаты свертывания выражений.) Поэтому разработчик должен представлять столбцы в выражениях подобных типов отдельно, чтобы эти столбцы не были никоим образом модифицированы или инкапсулированы. Разработчику достаточно применить простейшие алгебраические преобразования, чтобы переместить модификаторы в подготавливаемой конструкции в ту часть выражения, которая соответствует значению/константе, и оставить сам столбец немодифицированным.

Если оптимизатор не имеет возможности идентифицировать выражение как индексируемое, то не может также использовать статистические данные для оценки количества строк, возвращаемых соответствующим оператором. В таком случае оптимизатор должен руководствоваться определенными предположениями. При этом в качестве оценок применяются жестко заданные, так называемые "магические" значения, которые определены заранее для различных типов операторов сравнения. Общие сведения о "магических" числах, используемых оптимизатором для оценки некоторых операторов, приведены в табл. 12.3.

Таблица 12.3. Оценки количества строк для неиндексируемых выражений

Оператор сравнения	Оцениваемое процентное значение количества строк
=	10
>	30
<	30
BETWEEN	10

Свертывание

В некоторых редких обстоятельствах оптимизатор обладает способностью должным образом выявить индексируемое выражение, даже если столбец в этом выражении заключен в функцию или каким-то иным образом вошел в состав подвыражения. При этом применяется подход, известный под названием *свертывания* выражений, представляющий собой одно из усовершенствований по сравнению с более старыми версиями программы SQL Server, в которых оптимизатор не обладал способностью использовать индекс для обслуживания конструкции запроса, если столбец таблицы участвовал в выражении или скрывался в какой-то функции. Свертывание выражений позволяет оптимизатору выявлять некоторые типы индексируемых выражений, даже если столбец в таком выражении участвует в подвыражении или скрывается в функции. К числу свертываемых выражений относятся некоторые выражения, созданные с использованием функций `DATEADD`, `ISNULL` и `ROUND`, некоторые формы предиката `LIKE`, а также несколько других выражений. Например, рассмотрим следующий запрос:

```
SELECT *
FROM Orders
WHERE ISNULL(OrderDate,GETDATE())>'2003-04-06 19:55:00.000'
```

Оптимизатор, не способный свертывать выражение, в котором участвует столбец `OrderDate` и функция `ISNULL`, не имел бы возможности использовать индекс `OrderDate` таблицы `Orders` (ключом которой является столбец `OrderDate`) для обслуживания этого запроса. Тем не менее, ознакомившись с графическим изображением плана для данного запроса в программе Query Analyzer, можно убедиться в том, что индекс фактически используется.

В подобных случаях следует руководствоваться таким эмпирическим правилом: стараться при любой возможности избегать включения столбцов в подвыражения применяемых выражений. Тем не менее следует знать, что в некоторых ситуациях оптимизатор все равно обладает способностью выявлять индексируемые выражения.

Порядок соединений и выбор типа

Оптимизатор не только выбирает индексы и определяет индексируемые выражения, но и устанавливает порядок соединений, а также находит стратегию соединений для операторов, в которых требуются соединения. Задачи выбора

индексов и поиска стратегии соединения тесно связаны друг с другом — от выбора индексов зависит определение приемлемых типов стратегий соединения, а от выбранной стратегии соединения зависят типы индексов, которые потребуются оптимизатору для выработки эффективного плана.

В программе SQL Server поддерживаются три описанных ниже типа соединений.

1. Соединение с помощью вложенных циклов действует успешно, если внешняя таблица меньше внутренней, а для внутренней таблицы задан индекс.
2. Соединение слиянием действует успешно, если обе входные таблицы отсортированы по соединяющему столбцу (оптимизатор может в случае необходимости предусмотреть сортировку одной из входных таблиц).
3. Хэшированные соединения выполняются успешно в таких ситуациях, при которых отсутствуют применимые индексы. Но обычно достижение более высокой производительности обеспечивается с помощью создания индекса (что позволяет использовать другую стратегию соединения).

Оптимизатор определяет стратегию соединения, которая должна применяться для обслуживания запроса. При этом оптимизатор оценивает стоимость каждой стратегии и выбирает ту из них, которая согласно прогнозам должна быть наименее дорогостоящей. Оптимизатор оставляет за собой право переупорядочивать таблицы в конструкции `FROM`, если это позволяет повысить производительность запроса. Пользователь всегда может определить, применяется ли переупорядочение, ознакомившись с планом выполнения. Порядок таблиц в плане выполнения — это именно тот порядок, который согласно прогнозу оптимизатора должен обеспечить максимальную производительность.

Оптимизатор можно вынудить отказаться от использования его способности определять порядок соединения, применяя в запросе конструкцию `OPTION (FORCE ORDER)`, опцию сеанса `SET FORCEPLAN ON` и подсказки, касающиеся порядка соединения (например, `INNER LOOP JOIN`). В случае использования каждого из этих вариантов оптимизатор вынужден соединять таблицы в том порядке, который задан в конструкции `FROM`.

Следует отметить, что подход, в котором предусмотрено принудительное задание порядка соединения, может повлечь за собой побочный эффект, связанный с тем, что будет также принудительно задана какая-то конкретная стратегия соединения. Например, рассмотрим запрос и соответствующий ему план выполнения, которые показаны в листинге 12.10.

Листинг 12.10. Запрос, в котором применяется правое внешнее соединение, и план выполнения этого запроса

```
(Запрос)

SELECT o.OrderId, p.ProductId
FROM [Order Details] o RIGHT JOIN Products p
ON (o.ProductId=p.ProductId)

(План выполнения)

StmtText
```

```
------------------------------------------------------------------------
SELECT o.OrderId, p.ProductId
FROM [Order Details] o RIGHT JOIN Products p
ON (o.ProductId=p.ProductId)
  |--Nested Loops(Left Outer Join, OUTER REFERENCES:(p.ProductID))
       |--Index Scan(OBJECT:(Northwind.dbo.Products.SuppliersProducts AS p))
       |--Index Seek(OBJECT:(Northwind.dbo.[Order Details].ProductID AS o),
          SEEK:(o.ProductID=p.ProductID) ORDERED FORWARD)
```

Здесь заслуживает внимания то, что оптимизатор использует соединение с помощью вложенных циклов и переупорядочивает таблицы (на первом месте в плане указана таблица `Products`, даже несмотря на то, что в конструкции `FROM` первой указана таблица `Order Details`). Теперь зададим принудительно порядок соединения с применением подсказки запроса `FORCE ORDER` и посмотрим, как изменится план запроса (листинг 12.11).

Листинг 12.11. Принудительное задание порядка соединения с помощью подсказки `FORCE ORDER`

```
(Запрос)

SELECT o.OrderId, p.ProductId
FROM [Order Details] o RIGHT JOIN Products p ON
    (o.ProductId=p.ProductId)
OPTION(FORCE ORDER)

(План запроса)
StmtText
------------------------------------------------------------------------
SELECT o.OrderId, p.ProductId
FROM [Order Details] o RIGHT JOIN Products p ON (o.ProductId=p.ProductId)
OPTION(FORCE ORDER)
  |--Merge Join(Right Outer Join, MANY-TO-MANY MERGE:(o.ProductID)=
    (p.ProductID), RESIDUAL:(o.ProductID=p.ProductID))
       |--Index Scan(OBJECT:(Northwind.dbo.[Order Details].
          ProductsOrder_Details AS o), ORDERED FORWARD)
       |--Clustered Index Scan(OBJECT:(Northwind.dbo.Products.
          PK_Products AS p), ORDERED FORWARD)
```

Оптимизатор в данном случае не имеет возможности переупорядочить таблицы, поэтому переключается на стратегию соединения слиянием. Такой подход является менее эффективным по сравнению с тем подходом, в котором оптимизатору разрешено упорядочивать таблицы так, как он считает нужным, и соединять таблицы с помощью вложенных циклов.

Соединения с помощью вложенных циклов

Соединения с помощью вложенных циклов состоят из цикла, вложенного в другой цикл. В соединении с помощью вложенных циклов одна таблица в соединении обозначается как таблица внешнего цикла (или внешняя таблица), а другая — как таблица внутреннего цикла (или внутренняя таблица). В каждой итерации внешнего цикла просматривается вся таблица внутреннего цикла. Такой способ соединения вполне подходит для таблиц с размерами от небольших до

средних, но по мере увеличения таблиц, обрабатываемых в циклах, данная стратегия становится все менее эффективной. Общая процедура выполнения соединения описана ниже.

1. Найти строку в первой таблице.
2. Использовать значение из этой строки для поиска строки во второй таблице.
3. Повторять указанные действия до тех пор, пока в первой таблице не останется больше строк, которые соответствовали бы критериям поиска.

Оптимизатор оценивает по меньшей мере четыре комбинации соединений, даже если эти комбинации не заданы в предикате соединения. При этом оптимизатор учитывает, что стоимость оценки дополнительных комбинаций не должна превосходить общую стоимость выработки плана запроса.

Соединения с помощью вложенных циклов обладают гораздо более высокой производительностью по сравнению с соединениями слиянием и хэшированными соединениями, при условии, что объемы данных, применяемых в соединении, находятся в пределах от небольших до средних. Оптимизатор запросов использует соединения с помощью вложенных циклов, если внешняя входная таблица весьма мала, а внутренняя входная таблица индексирована и весьма велика. Оптимизатор переупорядочивает таблицы так, чтобы меньшая входная таблица стала внешней таблицей, а также требует введения применимого индекса на внутренней таблице. Кроме того, оптимизатор всегда использует стратегию вложенных циклов при выполнении тета-соединений (соединений по критерию, отличному от критерия равенства).

Соединения слиянием

Применительно к большим наборам данных соединения слиянием являются гораздо более эффективными по сравнению с соединениями с помощью вложенных циклов. Для успешного использования такого соединения обе таблицы должны быть отсортированы по столбцу, применяемому для слияния. Оптимизатор обычно выбирает соединение слиянием при работе с большими наборами данных, которые уже отсортированы по столбцам соединения. Оптимизатор может использовать деревья индексов для подготовки отсортированных входных таблиц, а также может предусмотреть применение операций сортировки с помощью конструкций `GROUP BY`, `CUBE` и `ORDER BY`; при этом достаточно выполнить сортировку только один раз. Если одна из входных таблиц еще не отсортирована, то оптимизатор может принять решение о ее предварительной сортировке, чтобы иметь возможность выполнить соединение слиянием (если по прогнозу оптимизатора соединение слиянием будет более эффективным, чем соединение с помощью вложенных циклов). Такая ситуация возникает очень редко и характеризуется наличием оператора `Sort` в плане запроса.

Соединение слиянием сводится к выполнению пяти описанных ниже этапов.

1. Получить первые входные значения из каждой таблицы.
2. Сравнить их.

3. Если эти значения равны, вернуть соответствующие им строки.
4. Если эти значения не равны, отбросить меньшее значение и использовать для очередного сравнения следующее входное значение из этой таблицы.
5. Повторять указанные действия до тех пор, пока не будут обработаны все строки одной из таблиц.

Оптимизатор выполняет в каждой таблице только по одному проходу. Операция соединения прекращается после обработки всех входных значений одной из таблиц. Все значения в другой таблице, оставшиеся необработанными, не обрабатываются.

Оптимизатор может выполнить операцию соединения слиянием для реляционного оператора соединения любого типа, за исключением `CROSS JOIN` и `FULL JOIN`. Для объединения таблиц друг с другом с помощью оператора `UNION` могут также использоваться операции слияния (поскольку таблицы должны быть отсортированы для устранения дублирующихся строк).

Хэшированные соединения

Хэшированные соединения также являются более эффективными по сравнению с соединениями с помощью вложенных циклов, если обрабатываемые наборы данных велики. Кроме того, хэшированные соединения успешно применяются для обработки таблиц, которые не отсортированы по столбцу (столбцам) соединения. Оптимизатор обычно выбирает хэшированное соединение, если сталкивается с большими входными таблицами и отсутствует индекс, с помощью которого можно было бы выполнить соединение этих таблиц, или индекс существует, но является неприменимым.

В программе SQL Server хэшированные соединения осуществляются путем хэширования строк меньшей из двух таблиц (*формирующей*) и вставки полученных хэшированных значений в хэш-таблицу с последующей обработкой более крупной таблицы (*проверочной*). Обработка осуществляется последовательно, по одной строке, при этом выполняется поиск совпадений в хэш-таблице. Источником значений в хэш-таблице является меньшая из двух таблиц, поэтому хэш-таблица имеет минимально возможные размеры. Кроме того, используются хэшированные, а не реальные значения, поэтому операции сравнения значений двух таблиц выполняются очень быстро.

Хэшированные соединения основаны на использовании понятия хэшированных индексов, которые широко применяются во многих развитых продуктах СУБД в течение длительного времени. В хэшированных индексах предусмотрено постоянное хранение хэш-таблицы, поскольку индексом является именно эта таблица. Хэшированные данные распределяются по слотам, которые имеют одинаковое хэшированное значение. Если индекс имеет уникальный непрерывный ключ, то существует функция хэширования, известная под названием *минимальной идеальной* хэш-функции; при использовании такой функции каждое хэшированное значение попадает в свой собственный слот, а пустые промежутки между слотами в индексе отсутствуют. Если индекс является уникальным, но не непрерывным,

может существовать функция хэширования, находящаяся на втором месте после наилучшей функции хэширования — так называемая идеальная хэш-функция; при использовании этой функции каждое хэшированное значение попадает в свой собственный слот, но между слотами могут возникать пустые промежутки.

Если входные таблицы выбраны в качестве формирующей и проверяемой таблиц неправильно (например, из-за того, что оценки плотности оказались неточными), оптимизатор меняет эти таблицы местами динамически с использованием процедуры, называемой *обменом ролями*.

Операции хэшированного соединения позволяют обслуживать реляционные операции соединения любого типа (включая операции `UNION` и `DIFFERENCE`), за исключением операций `CROSS JOIN`. Хэширование может также использоваться для группирования данных и удаления дублирующихся строк (например, в векторных операциях агрегирования, таких как `SUM(Quantity) GROUP BY ProductId`). Если в оптимизаторе хэширование осуществляется в такой форме, то в качестве формирующей и проверяемой таблиц используется одна и та же таблица.

Если входные таблицы в операции соединения велики и имеют примерно одинаковые размеры, то производительность хэшированного соединения сравнима с производительностью соединения слиянием. Если же входные таблицы в операции соединения велики, но значительно отличаются по размерам, то хэшированные соединения обычно заметно превосходят по своей производительности соединения слиянием.

Подзапросы и альтернативные варианты соединений

Подзапросом называется запрос, вложенный в более крупный запрос. Обычно подзапрос служит для получения нескольких значений в операторах предикатов (таких как `IN`, `ANY` и `EXISTS`) или одного значения, которое используется для присваивания столбцу или переменной. Подзапросы могут применяться во многих местах, включая конструкции `WHERE` и `HAVING` запроса.

Следует учитывать, что операции соединения не являются во всех обстоятельствах более удобными, чем подзапросы. Оптимизатор часто выбирает соединение для выполнения подзапроса, но это не означает, что применение подзапроса в коде — это неэффективное решение.

Пытаясь усовершенствовать запрос, чтобы исключить издержки выполнения соединений, влияющие на производительность, следует помнить, что всегда можно воспользоваться переменными таблицы и временными таблицами для сохранения рабочих данных, подлежащих дальнейшей обработке. При создании чрезвычайно сложных запросов такой вариант может оказаться наилучшим, поскольку он позволяет пользователю получить больший контроль над кодом оптимизации. Запрос можно разбить на несколько шагов и взять на себя контроль над тем, что выполняется на каждом шаге.

Для запросов, принадлежащих к категории от простых до умеренно сложных, производные таблицы позволяют получить аналогичные преимущества, поскольк-

ку с их помощью можно в определенной степени улучшить порядок обработки запросов. Производные таблицы могут служить в ходе вычисления выражения в качестве круглых скобок (ведь выражения с определениями производных таблиц действительно обозначаются с помощью круглых скобок). Дело в том, что производные таблицы позволяют установить определенный порядок вычислений. Если часть сложной конструкции SELECT передается в производную таблицу, к которой затем применяется оставшаяся часть конструкции SELECT, это равносильно такому указанию, передаваемому оптимизатору: "Вначале необходимо выполнить эти действия, затем передать полученные результаты во внешнюю конструкцию SELECT". При этом оптимизатор не обязан соблюдать указанный порядок действий; он может переупорядочивать таблицы в выражениях производной таблицы и даже отказываться от вложенных конструкций в пользу более эффективного плана, но по крайней мере подзапросы предоставляют синтаксический механизм, позволяющий разработчику указывать предпочитаемый им порядок таблиц. Автор рекомендует опробовать этот метод в характерных ситуациях для определения того, позволяет ли он добиться требуемой производительности.

Логические и физические операторы

Логические операторы описывают реляционные операции, используемые для обработки запроса. В целях последующего выполнения логические операторы должны быть преобразованы в физические. Физические операторы описывают действия, которые должна осуществить программа SQL Server, чтобы выполнить работу, затребованную в запросе (например, связанную с выборкой данных). Один логический оператор часто преобразуется в несколько физических операторов. Планы выполнения состоят из физических операторов, поскольку компонент LPE сервера задает именно эти операторы, когда возникает необходимость выполнить работу, обусловленную в запросе. Компонент LPE преобразует такие физические операторы в вызовы OLE DB, передаваемые из реляционной машины в машину хранения для выполнения работы, затребованной в запросе.

Каждый шаг в плане выполнения соответствует одному физическому оператору. Как уже было сказано выше, планы выполнения состоят из последовательностей физических операторов. В графическом изображении плана выполнения программы Query Analyzer эти операторы отображаются в области заголовка желтых всплывающих окон с подсказками. Если на каком-то этапе, кроме физического, показан логический оператор, последний также отображается в области заголовка окна справа от физического оператора и отделяется от него косой чертой. В текстовом отображении плана выполнения физический оператор находится в столбце PhysicalOp, а логический оператор, соответствующий данному шагу, хранится в столбце LogicalOp.

Описанное выше проще всего продемонстрировать на примере. Рассмотрим следующий запрос, для которого требуется реляционное внутреннее соединение:

```
SELECT *
FROM Orders o JOIN [Order Details] d ON (o.OrderID = d.OrderID)
```

Оптимизатор выбирает для выполнения этого запроса соединение слиянием, несмотря на то, что в самом запросе, безусловно, не указано, что должна быть выполнена такая операция соединения. Но с точки зрения реляционной теории данный запрос должен быть выполнен с помощью внутреннего соединения двух таблиц. Ниже приведена выдержка из текстового представления плана выполнения этого запроса.

```
PhysicalOp            LogicalOp             Argument
--------------------  --------------------  ------------------------------

Merge Join            Inner Join            MERGE:([o].[OrderID])=([d].[Or
Clustered Index Scan  Clustered Index Scan  OBJECT:([Northwind].[dbo].[Ord
Clustered Index Scan  Clustered Index Scan  OBJECT:([Northwind].[dbo].[Ord
```

Обратите внимание на то, что в столбце `PhysicalOp` в качестве оператора указан оператор соединения слиянием `Merge Join`. Именно эта операция применяется незаметно для пользователя при обслуживании операции, показанной в столбце `LogicalOp`, т.е. операции внутреннего соединения `Inner Join`. Дело в том, что в запросе содержалось требование выполнить внутреннее соединение, а сервер выбрал в качестве физического оператора для выполнения этого требования оператор `Merge Join`.

Оптимизатор не только определяет, какой индекс должен использоваться и какая стратегия соединения должны быть принята, но и вырабатывает дополнительные решения, касающиеся операций других типов. Некоторые из таких операции описаны ниже.

Конструкция DISTINCT

Обнаружив в запросе ключевое слово `DISTINCT` или `UNION`, оптимизатор должен удалить дублирующиеся строки, полученные из входных таблиц, прежде чем возвратить результирующий набор. Для этого может применяться целый ряд вариантов: оптимизатор может отсортировать данные для удаления дублирующихся строк или осуществить их хэширование. Кроме того, оптимизатор может обеспечить обслуживание логической операции `DISTINCT` или `UNION` с использованием физического оператора хэширования или сортировки. (В данном случае также часто применяется оператор `Stream Aggregate` — аналогичный физический оператор, широко используемый при обработке запросов `GROUP BY`.)

Конструкция GROUP BY

Оптимизатор может обслуживать запросы, включающие конструкцию `GROUP BY`, с помощью простых операций сортировки или хэширования. И в этом случае в качестве физического оператора может применяться `Hash` или `Sort`, а в качестве логического оператора по-прежнему используется `Aggregate` или какой-то аналогичный оператор. Кроме того, как уже было сказано, для выполнения операций `GROUP BY` часто применяется оператор `Stream Aggregate`.

Оптимизатор может выбрать для группирования данных вариант, в котором осуществляется операция хэширования, поэтому полученный результирующий

набор может не находиться в отсортированном порядке. Это означает, что при использовании конструкции GROUP BY не следует рассчитывать на получение автоматически отсортированных данных. Если требуется, чтобы результирующий набор был отсортирован, необходимо включить в запрос конструкцию ORDER BY.

Конструкция ORDER BY

Оптимизатору приходится вырабатывать определенные решения даже применительно к конструкции ORDER BY. Допустим, существует кластеризованный индекс, в котором уже представлены отсортированные данные; в таком случае оптимизатор должен определить приемлемый способ представления результирующего набора в требуемом порядке. При этом оптимизатор может применить сортировку данных, на что вполне может рассчитывать пользователь, или же предусмотреть просмотр листового уровня некластеризованного индекса, созданного с помощью соответствующих ключей. Решение по выбору того или иного варианта, принятое оптимизатором, зависит от множества факторов. Наиболее важным из них является избирательность — количество строк, возвращаемых запросом. Кроме того, важен показатель покрытия индекса, при оценке которого оптимизатор определяет, может ли некластеризованный индекс покрыть данный запрос. Если количество строк относительно невелико, то вариант, предусматривающий использование некластеризованного индекса, может оказаться менее дорогостоящим по сравнению с сортировкой всей таблицы. Аналогичным образом, если индекс может покрыть данный запрос, то соответствующий вариант находится на втором месте по своему качеству, в сравнении с вариантом, предусматривающим применение еще одного кластеризованного индекса, поэтому оптимизатор, по всей вероятности, будет использовать покрывающий индекс.

Накопление промежуточных результатов

Наличие оператора Spooling в плане запроса свидетельствует о том, что оптимизатор предусмотрел сохранение результатов промежуточного запроса во вспомогательной таблице для дальнейшей обработки. При использовании метода отложенного накопления промежуточных результатов рабочая таблица заполняется по мере необходимости, а при использовании метода опережающего накопления промежуточных результатов таблица заполняется в одном шаге. Оптимизатор предпочитает метод отложенного, а не опережающего накопления промежуточных результатов, поскольку существует вероятность того, что удастся избежать необходимости полного заполнения рабочей таблицы благодаря тому, что логические операторы, глубже заложенные в плане запроса, позволят завершить работу с использованием не полностью реализованных планов. Тем не менее в некоторых случаях необходимо применять метод опережающего накопления промежуточных результатов (например, для защиты от проблемы порочного круга, см. главу 14), но обычно оптимизатор предпочитает метод отложенного накопления промежуточных результатов, поскольку последний характеризуется меньшими издержками.

Операции накопления промежуточных результатов могут применяться не только к таблицам, но и к индексам. Если в оптимизаторе требуется определить, существует ли какая-то конкретная строка, для этой цели используются промежуточные результаты выполнения команды `rowcount`.

Резюме

Процессор запросов программы SQL Server использует индексы, статистические данные и поступившие запросы в качестве входных данных и вырабатывает оптимизированные планы выполнения, которые после этого осуществляются сервером для доступа и возврата затребованных данных. Сервер может применять кластеризованные и некластеризованные индексы, а также статистические данные, формируемые автоматически и вручную. Выполняя каждый конкретный план запроса, сервер может использовать несколько индексов в расчете на каждую таблицу и применять к индексам операции пересечения и соединения.

Процессор запросов выбирает план с наименьшей стоимостью. Обычно в качестве критерия определения стоимости применяется оценка объема ввода-вывода, относящаяся к рассматриваемому плану, но учитываются также другие факторы, определяющие стоимость. Оценка объема ввода-вывода в расчете на каждый шаг плана обычно вычисляется на основе учета прогнозируемого количества строк, возвращаемых на данном этапе в каждом цикле выполнения, и исходит из прогнозируемого количества итераций цикла выполнения. Расхождения между прогнозируемым и фактическим количеством строк и количеством итераций выполнения могут указывать на неточности в прогнозах, которые обычно обусловлены применением устаревших статистических данных или статистических данных, основанных на выборках с интервалом выборки, слишком низким для того, чтобы можно было точно представить его базовые данные.

Ключом к эффективной обработке запросов является предоставление оптимизатору достаточного объема информации для принятия обоснованных решений. Важнейшими факторами достижения высокой производительности запросов являются индексы, статистические данные, индексируемые выражения, ограничения и повторно применяемые планы запросов.

В программе SQL Server предусмотрено несколько инструментальных средств, позволяющих упростить настройку запросов. Великолепным инструментальным средством, позволяющим определить, являются ли оптимальными применяемые индексы и статистические данные, является программа Index Tuning Wizard. Для обработки в эту программу можно направить поток результатов трассировки или отдельный запрос, после чего формируются рекомендации по внесению изменений в организацию индексов и статистических данных, которые позволили бы добиться повышения быстродействия выполняемого кода. Инструментальное средство Profiler программы SQL Server (как и программа Perfmon) позволяет ознакомиться с информацией о количестве повторных компиляций процедур и об использовании кэша. Графическое отображение плана выполнения в программе Query Analyzer дает возможность ознакомиться с конкретной информацией об оценках, принятых в плане, и о входных таблицах, применяемых на каждом этапе.

Вопросы для самопроверки

1. Подтвердите или опровергните следующее утверждение. Если на таблице создается кластеризованный индекс с групповым ключом, то программа SQL Server автоматически "делает его уникальным".
2. Если на столбце `col` существует индекс, может ли его использовать оптимизатор для обслуживания предиката конструкции `WHERE col <> 1`?
3. Объясните, в чем состоит различие между избирательностью и кардинальностью.
4. Для чего предназначен оператор поиска закладки?
5. Подтвердите или опровергните следующее утверждение. При создании индекса определяемое для него значение коэффициента заполнения влияет на количество строк, хранящихся на каждой странице в B-дереве индекса, кроме корневой страницы.
6. Чему равно максимальное количество шагов в гистограмме статистических данных индекса?
7. В каком столбце таблицы `sysindexes` хранятся статистические данные индекса?
8. Какая команда `DBCC` может использоваться для вывода на внешнее устройство данных о фрагментации, относящихся к некоторому индексу?
9. Какая хранимая процедура может служить для указания типов блокировок, которые должны использоваться программой SQL Server для конкретного индекса?
10. Какая команда `DBCC` может использоваться для вывода на внешнее устройство данных гистограммы статистических данных, относящейся к некоторому индексу?
11. Что означает термин "покрывающий индекс"?
12. Какой параметр процедуры `sp_configure` определяет, будет ли оптимизатор запроса предпринимать попытки создать параллельный план выполнения на серверном многопроцессорном компьютере?
13. Подтвердите или опровергните следующее утверждение. Вычисленный столбец, который является недетерминированным, не может быть индексирован.
14. Объясните, что означает термин "плотность", применяемый в контексте оптимизации запросов.
15. Какое заключение, касающееся индексов рассматриваемой таблицы, можно сразу же сделать, обнаружив упоминание блокировки RID в выводе команды `sp_lock`?
16. Если на столбце `col` существует индекс, может ли этот индекс использоваться оптимизатором для обслуживания предиката `ISNULL(col,'')='foo'` конструкции `WHERE`?